SCHOLAR Study Guide

Advanced Higher Computing Science Software design and development

Authored by:

Charlie Love (CompEdNet)

Andy McSwan (Knox Academy)

Heriot-Watt University

Edinburgh EH14 4AS, United Kingdom.

First published 2020 by Heriot-Watt University.

This edition published in 2020 by Heriot-Watt University SCHOLAR.

Distributed by the SCHOLAR Forum.

SCHOLAR Study Guide Advanced Higher Computing Science: Software design and development

Advanced Higher Computing Science Course Code: C816 77

Print Production and Fulfilment in UK by Print Trail www.printtrail.com

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.

Contents

Topic 1

Analysis

Contents

Prerequisites

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

- Identify the end-user requirements of a program, database or website as relates to the design and implementation at Higher level.
- Identify the functional requirements of a program, database or website as relates to the design and implementation at Higher level.

Learning objective

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

- Identify the purpose of a software solution that relates to the design and implementation at this level
- Describe, exemplify, and implement research for:
  - Feasibility studies:
    - Economic
    - Time
    - Legal
    - Technical
  - User surveys
- Describe, exemplify, and implement planning in terms of:
  - Scheduling
  - Resources
  - Gantt charts
- Produce requirement specifications for end-users and develop:
  - End-user requirements
  - Scope, boundaries and constraints
  - Functional requirements in terms of
    - inputs
    - processes
    - processes
- Describe, exemplify, and implement Unified Modelling Language (UML):
  - Use case diagrams:
    - Actors
    - Use cases
    - Relationships

You may have already covered the material in this topic as it applies to Software, Database and Web design and development.

1.1 Introduction

Analysis is the process of understanding what it is that the client, users and stakeholders want from the system that is being developed. There needs to be a detailed list of the requirements for the product. This will form the basis of a contract between the developer and the client and will be legally binding.

We will create the following at the analysis stage:

- A Requirements Specification
- End User Requirements
- The Scope and Boundaries of the project
- A list of Constraints on the project
- The Functional Requirements for the Project

1.1.1 Requirement Specifications

The requirements specification details the scope, boundaries and constraints of the intended software, details the basis of payment for the work to be completed and sets out how the software will be designed, tested, documented and evaluated before hand over to the customer. It may also detail any longer term maintenance agreement between the customer and the developer.

1.1.2 End user requirements

In the Higher Computing Science course we looked at how the end user requirements would be generated by using **Personas**, **User Stories**, **User Scenarios**, **Use Case**, User Devices and Software.

At Advanced Higher level can use these techniques to help generate a list of what the users would expect from the software being created.

1.1.3 Scope and boundaries

The Scope of the project is held in the **product backlog** - it is the definition of the features that the product must have - also known as the requirements of the software. If a feature is no longer required then it is dropped from the product backlog because it is out of scope.

Constraints are limitations that affect the development of the product. A project may be constrained by time (so only a certain number of development sprints can be completed), or may be limited by budget or may be limited by legal position regarding the data that it processes.

It is very important at the outset to establish clearly the scope, boundaries and constraints of the project. Scope and boundaries are opposite sides of the same "coin". Between them, they give a precise description of the extent of the project.

Here is a typical statement about scope and boundaries. You will find similar statements by

searching the web. "The project scope states what will and will not be included as part of the project. Scope provides a common understanding of the project for all stakeholders by defining the project's overall boundaries."

One way of thinking about it is:

- the scope clarifies what the project must cover;
- the boundaries clarify what the project will not cover.

For example, suppose your project was to develop an expert system giving students guidance on job opportunities which they should consider after graduating from University.

The scope of the project would be to create an expert system. Then it would be necessary to describe the range of jobs and degrees that would be included in the system, the level of information that would be output by the system (does it suggest contact addresses as well as simply job types), the types of questions that the user will be asked. Does it cover all degrees, or is it only for students with Computing Science degrees, and so on ... All these things will define the boundaries of the system.

Sometimes it is also helpful to spell out exactly what will NOT be covered. So, for example, a clear statement could be made which states that the system will NOT cover advice on jobs for those with medical and veterinary degrees, or jobs overseas.

The scope and boundaries could also refer to technical issues. For example, they might state that the resultant system will run on any computer capable of running any version of Windows after Windows 7, but not on any other operating system.

Defining scopes and boundaries

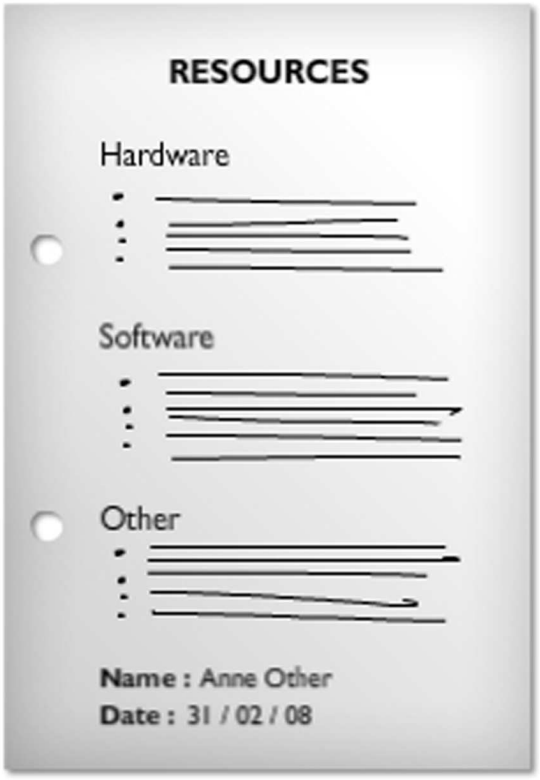
Now, starting from your project proposal, create clear statement of scope and boundaries for your project. Hint: write this as a bulleted list, rather than as a paragraph. This has two benefits - it helps you to clarify your ideas, and it gives you a clear list to use at the end of your project when you are evaluating what you have produced.

Discuss this with your tutor.

Add the scope and boundaries list to your Record of Work.

Don't forget to put your name/initials and the date on the page.

Your Record of Work should now include:



Scope

Examples of scopes for a modular program and a relational database

Scope of a programming task

This development involves creating a modular program.
The deliverables include:

- a detailed design of the program structure
- a test plan with a completed test data table
- a working program
- the results of testing
- an evaluation report

Scope of a relational database task

This development involves creating a relational database.
The deliverables include:

- a detailed design of the database structure
- a test plan with a completed test data table

- a working database
- the results of testing
- an evaluation report

Scope of a website task

This development involves creating a multi-level website.
The deliverables include:

- detailed wireframe designs of each page of the website
- a test plan with a completed test data table
- a working website
- a working website
- an evaluation report

1.1.4 Constraints

Constraints are limitations that affect the development of a product.
A project may be constrained by:

- Time (so only a certain number of development sprints can be completed)
- The scope of the project i.e. what the end result of the project will be and what should the developer deliver (project deliverables) to the client?
- Limited by budget (Cost)
- Legal position regarding the data that it processes i.e. What country will the stored data be held in and does it comply with GDPR

At this stage we should detail what operating systems the project will work on (Windows, iOS, Android, Linux, etc. . .), what environment will be used for development

1.1.5 Functional Requirements

Functional requirements are defined as what the product should do and are split into three sections — Input, Process and Output.

Consider this example of a forum website which allows users to create a user account.

Functional requirements

The new user page should present the user with a form enabling them to create a new account by entering username and email address. On submitting the form, the user receives a welcome email containing a link to confirm the account.

This link should take them to their account details where they can perform the following tasks:

- Inputs
  - enter /change personal information;
  - upload an avatar image;
  - change forum preferences;
  - change password;
  - delete account;
  - confirm their account details by pressing a submit button.
- Process
  - The welcome email will warn the user that their account will expire if they do not go to the link and confirm their details within a certain time period.
  - The site should reject usernames or email addresses which belong to existing users.
  - The site should reject obscene or racist usernames.
  - The site should be secure.
  - If the user details are not submitted within 48 hours then user is deleted from the user database.
  - The page contains a form with text entry boxes with maximum and minimum size restrictions.
  - Client side validation is used for email address and date of birth.
  - The username text box entry is a required one, with a minimum size of 8 characters and a maximum of 16.
  - The Email text box is also required and uses client side validation to check for a valid email address.
  - The page allows an avatar upload restricted to 1Mb. The default avatar is allocated if no upload occurs.
  - Forum options are check boxes.
  - On submit, the form data is passed to a PHP script which sanitizes the data then the username and email address are checked against existing entries in the user database. Usernames are also checked against the database table containing unacceptable values and an appropriate rejection advice response is sent if a match is found in either case. If the username and email are unique, a new database entry is created for that user and a welcome email is sent including a link to the user configuration page using a further PHP script.
  - On submit, the form data is passed to a PHP script where it is sanitized and the user database is updated.
- Outputs
  - Confirmation of successful
    - account creation
    - change of password
    - deletion of account
  - Database reports will be available to administrators.

1.2 Research: feasibility study, user surveys

Learning objective

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

- explain the different types of feasibility;
- explain the use of users surveys to gather information about a project.

Research is a vital part of the initial analysis of what the project sponsor is requesting.

1.2.1 Feasibility study

A key part of the project initialisation is undertaking research to ensure that the project is feasible. Software development projects, depending on their size, can be in development for many years and can have budgets worth millions of pounds. It would be foolish to proceed with any project without assessing if completing it is possible.

The basic purpose of a feasibility study is to work out if the proposed expenditure of time and money is likely to be worthwhile and whether the objectives of the project can be achieved: what some projects set out to do might be totally unrealistic.

The results of the feasibility study will determine which, if any, of a number of possible solutions can be further developed at the design phase.

One result of undertaking a feasibility study may indicate that only a simple solution is required to solve the problem:

- a software fault may be identified that is easily fixed;
- more staff training might be required if particular software is to be used.

The feasibility study is most often carried out by the **project leader** - an experienced member of staff who is able to understand the needs of the project and the various aspects of feasibility that apply to it.

1.2.2 Types of feasibility

A feasibility study will look at four main areas of feasibility: Technical (is it technically possible to create a solution with the available technology), Economic (is it possible to complete the project with the budget available or is the cost of creating the project justified by the financial reward of doing so), Legal (can the solution be created and adhere to existing laws) and Schedule (is there enough time to complete the project, are the right people and resources available when required to deliver the project on time).

1.2.2.1 Technical feasibility

The feasibility study must ascertain what technologies are necessary for the proposed system to work as it should. It may be the case that suitably advanced technology does not yet exist. Unless it is the object of the project to design a system to use such advanced technology, this would rule the project out as being a non-starter. It would be a foolish move for a feasibility study to evaluate

technologies which are either under development or undergoing testing.

Given that suitable technology does exist, the study must establish if the organisation already has the necessary resources. If not, the study must make clear what new resources the organisation would have to acquire. This will also involve determining whether the hardware and software recommended will operate effectively under the proposed workload and in the proposed environmental conditions. The development of a new system involves risks of one kind or another. Every understanding that might be reached could carry the risk of some misunderstanding:

- software companies and their clients often have different vocabularies and consequently they appear to be in perfect agreement until the finished product is supplied;
- management may have unrealistic expectations of computer systems. The feasibility study is where idealism meets reality.

Further issues might include the training of personnel to use the new system, consideration of service contracts, warranty conditions and the establishing of help desk facilities for inexperienced users.

1.2.2.2 Economic feasibility

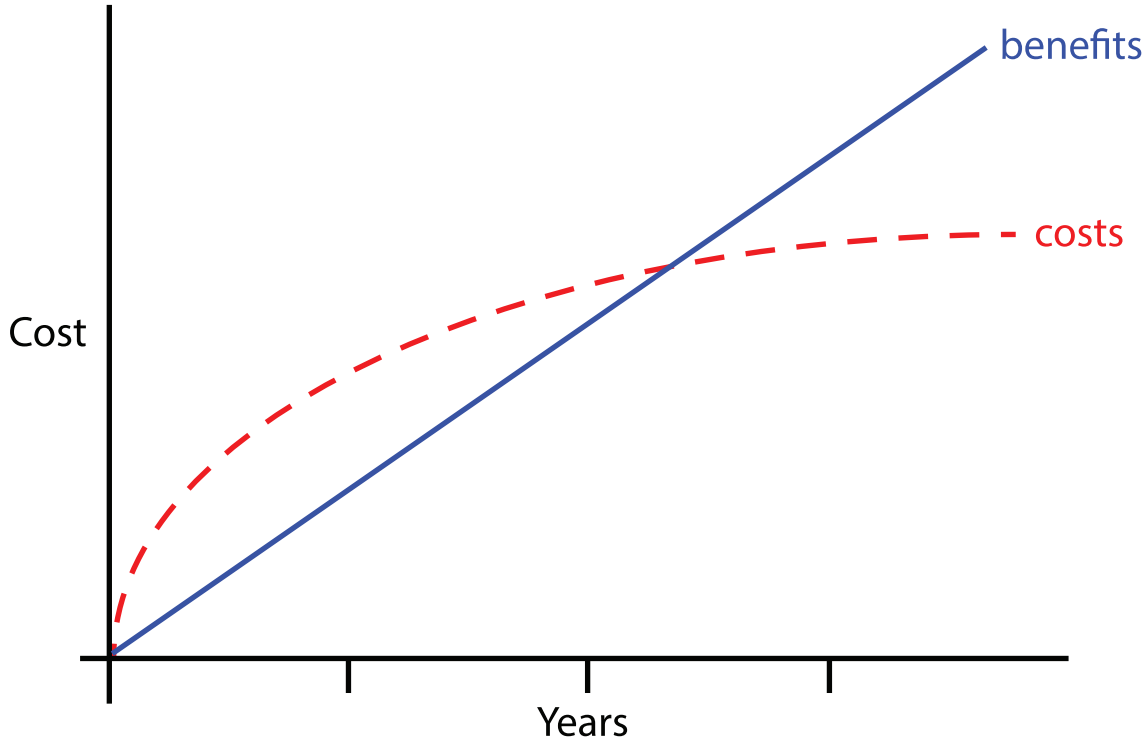
This deals with the cost implications involved. Management will want to know how much each option will cost, what is affordable within the company's budget and what they get for their money. A cost-benefit-analysis is part of the budgetary feasibility study. If the project is not cost-effective then there is no point proceeding.

Setting up a new computer system is an investment and involves capital outlay. The costs of a new system include the costs of acquiring it in the first place (consultancy fees, program development, etc.); the costs of installing it (disruption of current operations, cost of new equipment, alteration of workplace, etc.); and the costs of maintaining it which also includes training.

In the long term management will also want to know the 'break-even point' when the new system stops costing money and starts to make money. This is extremely difficult to quantify. However an accurate estimate of a system's operational life span is a valid option and will rely solely on the knowledge and experience of the systems analyst involved.

Figure 1.1 depicts such a case where the break-even point is at the intersection of the graphs:

Figure 1.1: Break-even point



Tangible benefits that management would certainly be looking for in the new system would be:

- reduced running costs;
- increased operational speed;
- increased throughput of work;
- better reporting facilities.

Note that not all the costs and benefits lend themselves to direct measurement. For example, new systems generally affect the morale of the staff involved, for good or ill. This can only be resolved by competent personnel management practices.

1.2.2.3 Legal feasibility

This has to do with any conflicts that might arise between the proposed system and legal requirements: how would the new system affect contracts and liability, are health and safety issues in place and would the system be legal under such local laws as the UK Data Protection Act? What are the software licensing implications for the new system?

Software licensing can be quite a thorny problem. Licences can be purchased as: client licence (per seat), server licence, network licence or site licence and the period of operation may be annual or perpetual. Software vendors vary in their licensing regulations so this has to be fully investigated.

1.2.2.4 Schedule feasibility

Schedule feasibility may be assessed as part of technical feasibility. Most organisations have an annual schedule of events such as the AGM, end of financial year, main holiday period and so on. Obviously time is a main factor in the development of a new system.

Questions to be asked at this stage might include:

- how long will the proposed system take to develop?
- will it be ready within the specified time-frame?
- when is the best time to install?

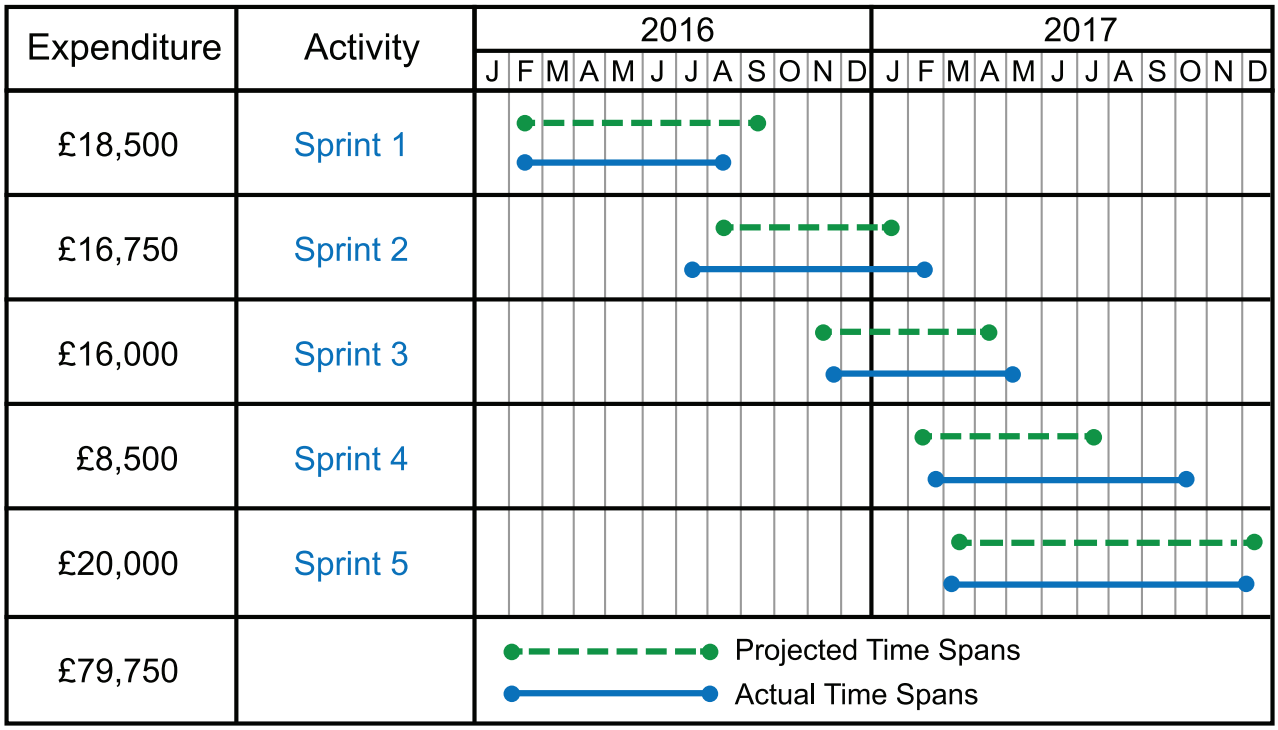
For example, a project might have to start within six months; assuming it would take three months to purchase and install the required hardware and software and a further six months to train the end users. Such a project is not technically feasible because of shortage of time so it would not go ahead unless some of the time constraints were reviewed and changed.

In many cases of project management the scheduling component can be aided by means of a Gantt chart.

A Gantt chart is like a horizontal bar graph used to plan and schedule projects involving several concurrent tasks. The horizontal axis represents the time scale and start and finish times of component parts are graphically represented.

The advantage of a Gantt chart is that it shows, at a glance, the progress of a project as shown in Figure 1.2

Figure 1.2: Example of a Gantt chart



You will look more closely at the creation of a Gantt chart and the concepts of Project management in the Project management Topic of this course.

Activity: Types of feasibility Go online 

Q1:

Match the parts together to generate true statements:

1. A project that cannot be delivered in the time available because of the complexity of the task
2. A project which has insufficient finance to hire the necessary staff to complete the project when required
3. A project that processes banking transactions to indicate spending habits in contravention of data protection laws
4. A project which cannot proceed because the software to link two data sources into the project is not available

a) has an issue with technical feasibility.
b) has an issue with legal feasibility.
c) has an economic feasibility issue.
d) has an issue with schedule feasibility.

1.2.3 User surveys

Often with the development of software, the experience of existing users is vital in shaping what the new software should do better than the old software. The users of a particular program are the best people to identify the existing problems with the software and how it could be improved.

Equally, if the software being developed is entirely new, the target user group should, generally, understand the process that the programme will carry out. For example, a new library system, which is to replace a manual card based system, would be developed following some time spent capturing the experience of existing librarians and how they operate.

When the opinion of a number of users is required, it is often easier to create a survey or questionnaire which will capture the information that the project team require.

1.3 Planning

Project management is the process of planning and controlling the activities of a project to ensure that it is delivered on time, with the required features and within the budget available to the required quality.

The Association Of Project Management define project management as:

"Project management is the application of processes, methods, knowledge, skills and experience to achieve the project objectives."

Projects are different from the day-to-day business of an organisation - a project is a specific piece of work which is defined by what it attempts to achieve. A project is usually judged to be successful if it achieves the **objectives** (normally according to some **acceptance criteria**) within an agreed

timescale and **budget**.

A project plan is more than just a list of tasks to be completed. It is a plan that details individual responsibilities, resources available to complete work, the order of work and the key **milestones**.

1.3.1 Relationship between time, cost, scope and quality

Learning objective

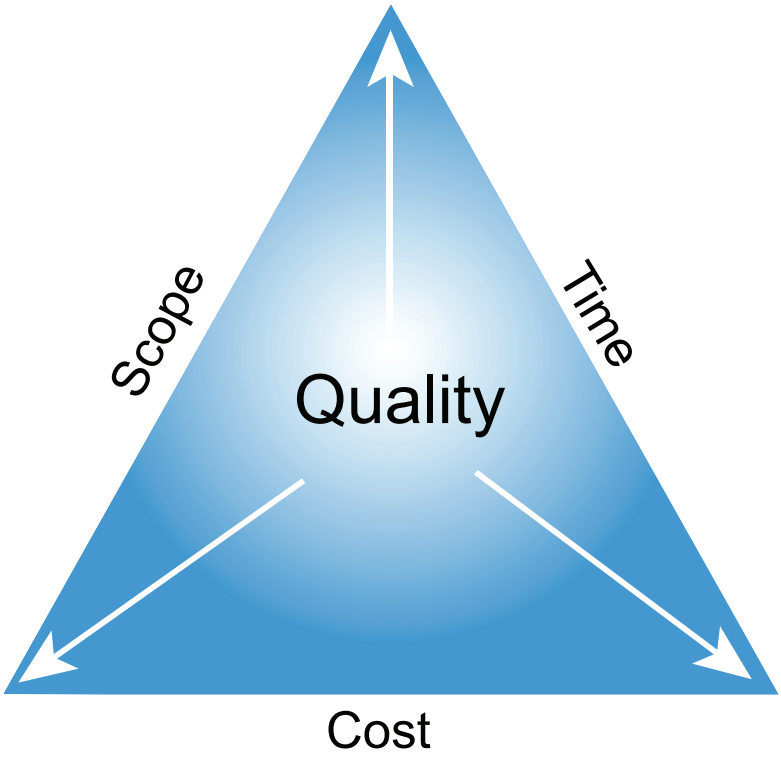
By the end of this section you will be able to:

- describe the relationship between time, cost, scope and quality in the project "triangle".

Before looking more closely at the process of creating a simple project plan, it is important to understand the relationship between **time**, **cost**, **scope** and **quality** when planning a project.

These are the **constraints** of the project. Changes to any one of these constraints has an immediate impact on the others.

Figure 1.3: The Project management triangle



The **Scope** of a project details what needs to be produced and delivered. In the case of a software development project, the system requirements will give us this information. With the **waterfall model** of software development, the requirements of the project are completely known after the analysis phase. Any subsequent changes to the scope will add **time** and/or **cost** to the project because more time and/or more resources (people who have to be paid or over time hours for existing staff) will be required to deliver the project to the same required level of **quality**.

Time is the amount of time it takes to complete the project, to the required **scope** with the available **budget** (cost) to the required quality.

Cost is the budget for the project. If the budget changes then less or more can done. An increase in budget may increase the quality of the work to be done, allow more to be achieved (increased scope) or for the work to be completed in less time.

In a project that uses an **agile Scrum** approach, the time and cost constraints are generally fixed, because the agreement between the developer and the client is for a specific fee for work completed. The reality of large and medium scale projects is that it is almost impossible to understand all of the requirements for the software at the start of the project - using an agile approach reduces the risk to the client as they can change the scope of the project by removing requirements which are no longer needed, adding new ones and by changing the priority of the existing requirements as the project proceeds. This evolving list of requirements is held in the product backlog and is updated prior to each **sprint** in the agile process.

Typically, **Scrum** projects do not have a formal **Project manager** as the team is self-organising and supported by the ScrumMaster, however, in large scale Scrum projects, where there are multiple teams working to produce elements of the project, the role of project manager to act as a buffer between outside organisations and the teams can be useful. This project manager may construct a high-level project management plan based on the **product backlog** and maintain this to estimate completion of the project.

1.4 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).

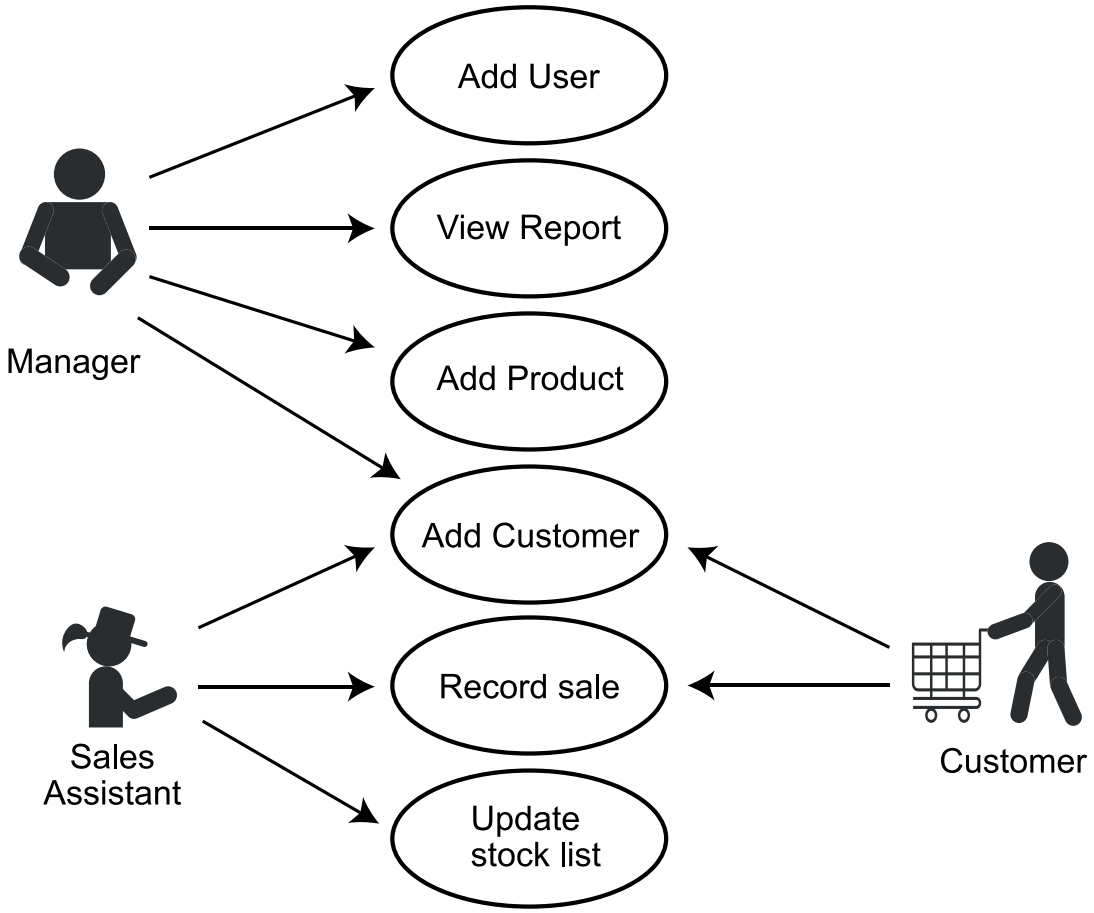
UML is a design notation created specifically to help programmers design systems using the object oriented programming paradigm. As its name suggests it is a formal system for creating models for software design. UML provides a specification for a number of different types of diagram which can be used 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.

There are a number of different types of diagrams specified by UML however for the analysis topic we will only look at **Use Case diagrams**. UML Class Diagrams will be looked at in the design stage of Software Development.

1.4.1 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.

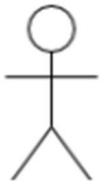
The Customer is involved when a sale is made. If they are a new customer then their details are added to the customer table. When they make a purchase the sale is added to their details.

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

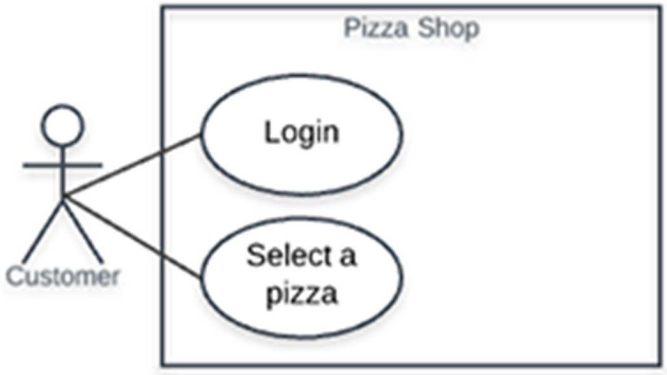
The manager needs to be able to update the staff table if a new sales assistant joins the company, add new products to the stock table and generate reports of sales.

Use case diagrams contain Actors (the people or entities who interact with the system), Use Cases which are the procedures/actions which the user will interact with and the relationships that exist between actors and use cases.

Actors:



Actors should be general categories of users for a system, i.e. customer, staff or another system that it may need to interact with (i.e. a database that the website needs to connect to). We depict an actor in a Use Case Diagram by using a stick figure and putting the type of actor underneath. Each actor must have at least one use case associated with it.



There are two types of actors in use case diagrams, primary and secondary. The primary actors are the ones who start the system. These actors are placed on the left side of the system boundary.

The secondary actors are those users who react to something happening in the system and are displayed on the right side of the system boundary.

Use Cases:

The Use Cases in the diagram are identified by an ellipse with text in the middle explain a task the system is to do. Each individual action that the user can perform in the system will get it's own use case.

Relationships:

- Association
- Include
- Extend
- Generalisation of an Actor
- Generalisation of a Use Case

Association— Solid line shows a basic communication between the actors and the system

Include:

- This is a use case that is not directly accessed by the actors, it shows a dependency between

a **Base Use Case** and a **Dependent Use Case**.

- It is indicated by a dashed line with an arrow that points to the **Dependent Use Case** with the word include.
- The relationship should also be identified in the diagram using double angle brackets and the word <<**include**>>.

Extend:

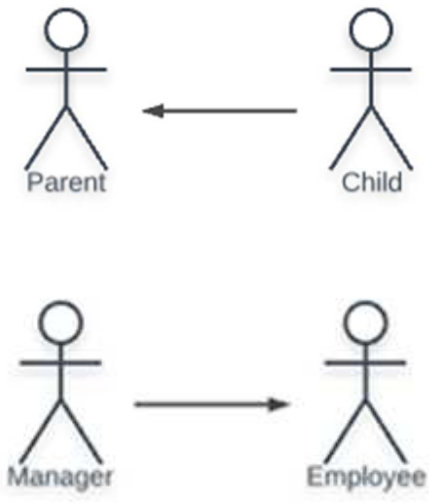
- Similar to an include the **Extend Use Case** will not be directly accessed by an actor and will have a **Base Use Case** but with this relationship the **Extend Use Case** may not always be actioned, certain criteria needs to be met before it can be ran.
- This is also indicated by a dashed line and an arrowhead but this time it will point to the **Base Use Case**.
- The relationship should also be identified in the diagram using double angle brackets and the word <<**extend**>>.

Generalisation (inheritance)

There are two types of **Generalisation** we need to look at, generalisation of actors and generalisation of use cases.

Generalisation of Actors

For actors generalisation uses the idea of inheritance to allow new actor (child) who need to make use of the same basic use cases as the parent actor and can then be given access to any additional use cases they need.

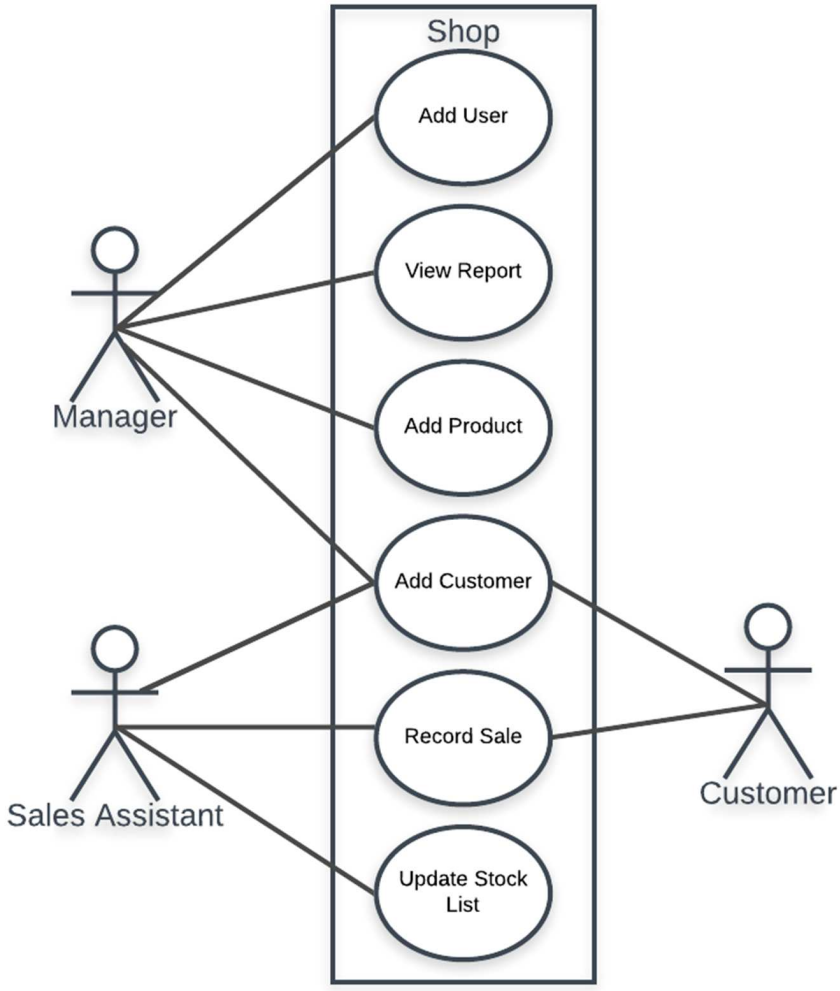


Generalisation of Use Cases

This is Indicated by a solid line and an arrowhead pointing at the **General Use Case**.

Diagrams (Basic example)

The Actor's interactions (associations) with them are identified by solid lines. This is a type of **relationship**.



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 into categories, 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.

Review of Use Case Diagrams Go online

Q2:

Fill in the gaps with the word from the word list.

Word list: Generalisation, Actors, system, ellipse, actioned, Actor, Extend, Use Case, boundary, Include

_____ diagrams are made using _____ that will interact with the system. The system is indicated by a _____ _____ where all the use cases will be placed. A Use Case in the diagram will be represented by an _____.

There are 5 relationships we need to use in a Use Case Diagram:

Association

_____ — this is a use case that is always _____ but not directly by the Actor.

_____ — This Use Case is not directly accessed by the Actor and doesn't always run.

Generalisation of an _____

_____ of a Use Case

1.5 End of topic test

End of topic test: Analysis Go online

Try the end of topic test: Analysis.

Q3: 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 relationship between a person and a system.
d) A method.

. . . . . .

Q4: A use case diagram illustrates:

a) interactions between objects.
b) how an object is derived from a class.
c) how actors interact with the system.
d) relationships between classes.

. . . . . .

Q5: There are four types of feasibility: legal, schedule,. . . .

a) sequential and iterative
b) social and economic
c) economic and technical
d) economic and logistical

. . . . . .

Q6: Which four constraints are applied to any project?

- Management
- Sequence
- Milestones
- Rentals
- Product backlog
- Time
- Developer resource
- Scope
- Legal
- Compiler time
- Cost

. . . . . .

Q7: A requirements specification will consist of:

- End User Requirements
- System Specifications
- Use Case Diagram
- Scope
- Boundaries
- Constraints

Topic 2

Design

Contents

Prerequisites

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

- read and understand designs that make use of structure diagrams and pseudocode;
- use pseudocode to implement efficient design solutions to a program that makes use of top level design, data flow and refinements;
- describe and create user interface designs in terms of input and output, using a wireframe.

Learning objective

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

- describe and make use of object oriented programming to develop solutions;
- read structure diagrams, pseudocode and UML to understand the design of programming solutions;
- use pseudocode to create a solution to a problem making use of top level design, data flow and refinements;
- use UML to create class diagrams demonstrating key aspects of object oriented programming;
- describe and use wireframes for user interface design to show:
  - visual layout;
  - inputs;
  - validation;
  - underlying processes;
  - outputs.

2.1 Revision

Quiz: Revision Go online

Q1: 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

.....

Q2: `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

.....

Q3: Which one of the following statements is true?

a) Structure charts are not hierarchical.
b) The modules in a structure chart will become modules in the finished program.
c) A structure chart cannot show the data flow between modules.
d) It is not necessary to bother about the module names as these will change in the code.

.....

Q4: Which one of the following is **not** a graphical design notation?

a) Wireframe
b) Data flow diagram
c) Structure diagram
d) Pseudocode

.....

Q5: Which design notation is the easiest to create source code from?

a) Wireframe
b) Pseudocode
c) Data flow diagram
d) Structure diagram

.....

Q6: 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.

. . . . . .

Q7: 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.
c) writing source code.
d) creating a wireframe interface design.

2.2 Object oriented programming

Learning objective

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

- describe and make use of object oriented programming to develop solutions.

Before creating **class** diagrams, we'll cover the basics of the object oriented programming approach that you will make use of in Advanced Higher.

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.

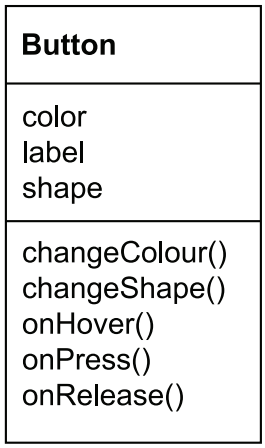
2.2.1 Classes and objects in object oriented programming

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.

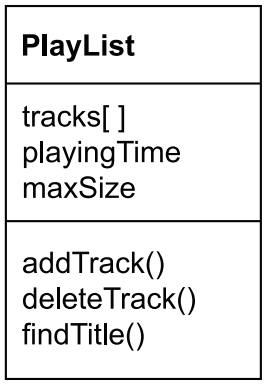
Figure 2.1: An example class - Button



Example : Music player: PlayList class

A music player application might contain a **PlayList** class which 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 and it would have methods: addTrack, deleteTrack, findTitle etc.

Figure 2.2: An example class - PlayList



A procedural language program is built from a number of procedures called from a main procedure. An object oriented program will have a number of methods linked to the class within which they are defined, in line with the idea of keeping classes and objects self-contained.

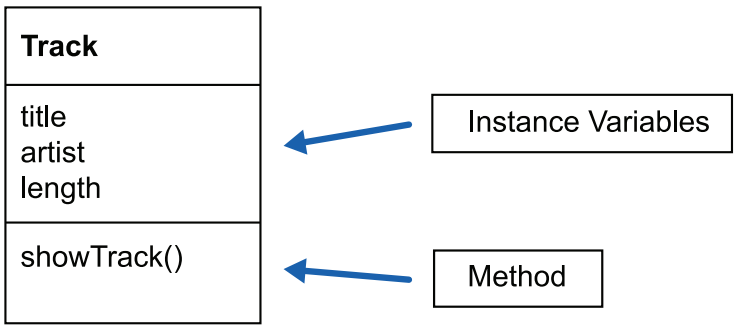
2.2.2 Classes and objects example: The music player

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

Example : Music player: Track class

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

Figure 2.3: An example class - Track




```
1 CLASS Track IS {STRING title, STRING artist, REAL length}
2 METHODS
3
4 PROCEDURE showTrack()
5   SEND THIS.title & " " & THIS.artist & "  " & THIS.length TO
        DISPLAY
6 END PROCEDURE
7
8 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:

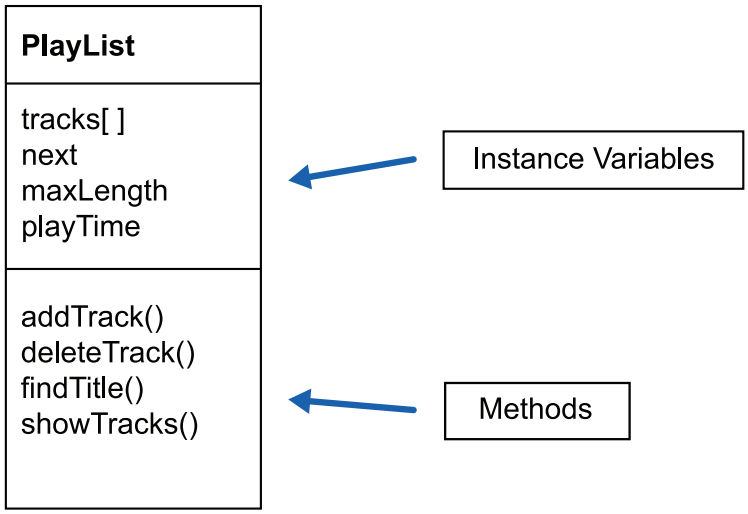
```
favouriteTrack.showTrack()
```

The result of this command would be to send: **Telstar Tornados 3.4** to the screen.

Example : Music player: PlayList class

The PlayList class has four instance variables: **tracks** which is an array of **Track** (this will be an array of objects), **next** which is used to store the next available place in the list, **maxLength** which is used to store the maximum number of tracks, and **playTime** to store the total playing time.

Figure 2.4: PlayList




```
1  CLASS PlayList IS {ARRAY OF Track tracks, INTEGER next, INTEGER
       maxLength, REAL playTime}
2
3  # Creating instance of a class is like creating an instance of a
       record.
4
5  METHODS
6
7  CONSTRUCTOR PlayList(INTEGER maxlength)
8
9  # the CONSTRUCTOR is used to give initial values to the instance
       variables
10
11 DECLARE THIS.maxlength INITIALLY  maxlength
12 DECLARE THIS.tracks INITIALLY THIS.Track[maxLength]
13 DECLARE THIS.next INITIALLY 0
14 DECLARE THIS.playTime INITIALLY  0.0
15
16 # note the use of THIS to refer to the instance variables
17
18 END CONSTRUCTOR
19
20 PROCEDURE addTrack(Track newtrack)
21
22 # adds a new track to the playList
23 # if next is less than the maximum number allowed
24
25   IF THIS.next = THIS.maxLength THEN
26     <error - too many tracks>
27   ELSE
28     SET THIS.tracks[THIS.next] TO newTrack
29     SET THIS.next TO THIS.next + 1
30     SET THIS.playTime TO THIS.playTime + THIS.newTrack.length
31   END IF
```

```
32
33 END PROCEDURE
34
35 FUNCTION findTitle(STRING trackTitle) RETURNS INTEGER
36
37 # finds a track in the playList
38
39    DECLARE result INITIALLY -1
40
41 # will return -1 if title not found
42
43   FOR index FROM 0 TO maxLength - 1 DO
44     IF THIS.tracks[index].title = trackTitle THEN
45       SET result TO index
46     END IF
47     END FOR
48   RETURN result
49
50 END FUNCTION
51
52 PROCEDURE deleteTrack(INTEGER index)
53
54 # deletes a track from the playList
55
56   IF index >= THIS.maxLength OR index <0 THEN
57     <error - invalid index>
58   ELSE
59      SET THIS.playTime TO THIS.playtime - THIS.tracks[index].length
60     FOR counter FROM index TO next-1 DO
61       SET THIS.tracks[counter] TO THIS.tracks[counter + 1]
62     END FOR SET
63     THIS.next TO THIS.next-1
64   END IF
65
66 END PROCEDURE
67
68 PROCEDURE showTracks()
69
70 # loops through the playList displaying each track
71
72   FOR counter FROM 0 TO next - 1 DO
73     THIS.tracks[counter].showTrack()
74   END FOR
75
76 END PROCEDURE
77
78 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 **maxlength** 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 which sets up the myTracks playList with 10 tracks like this:

```
DECLARE myTracks INITIALLY PlayList(10)
```

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

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

Then using the **favouriteTrack** object we created from the Track class we can add it to the playList as well:

```
myTracks.addTrack(favouriteTrack)
```

2.2.3 Encapsulation

The bundling of methods and instance variables into one unit is referred to as **encapsulation**. This also allows the items defined in a class: the methods and the instance variables, to be defined as private or public.

Private means that an item can only be accessed from ***inside*** the class. **Public** means that the item can be accessed directly by any method within any other object in the program.

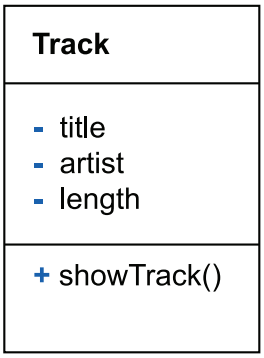
Key point

When developing object oriented programming based solutions, it is good practice to minimise the number of public instance variables and methods. Otherwise, it becomes easy for mistakes to be made and variables to be updated unintentionally.

Example : Music player: Track class showing private and public items

In the music player example, the Track class could be shown with the private and public items defined.

Figure 2.5: Track class with private/public



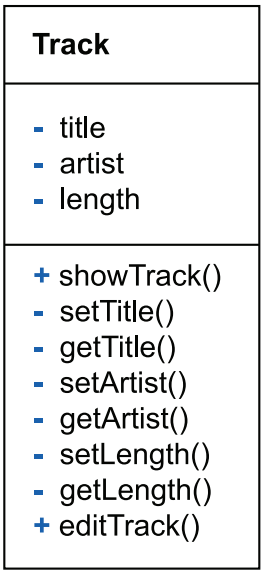
This diagram shows that the instance variables (title, artist and length) are all private (shown by a "-") and the showTrack() method is public (shown by a "+").

When designing object oriented programs, it is common practice to make instance variables private in this way and to allow them to be updated or retrieved by **setters** and **getters**.

A setter is a method that sets a value for an instance variable. A getter is a method that retrieves the current value of an instance variable. In the Track class, each instance variable would have setters and getters.

Example : Music player: Track class showing setters and getters

Figure 2.6: Track class with getters and setters



In this example, the setters and getters are shown as private. This is because they are used by other methods within the class. The showTrack() method would make use of the getter methods (getTitle, getArtist, getLength) and the editTrack() method would make use of the setter methods (setTitle, setArtist, setLength).

Key point

In the SQA Reference Language, the assumption made is that all instance variables are private and that all methods are public. However, you may be asked to explain, using diagrams like the one shown in the previous example, the application of private and public settings for both instance variables and methods.

2.2.4 Constructor

A **constructor** is a default method for a class which is used when an object is created. The constructor may give values to none, some or all of the instance variables for the object when it is created from the class.

The SQA reference language allows objects to be constructed without a constructor function by just using the name of the class.

Example : Music player: declare an object from the Track class

To declare an object from the Track class:

```
1 DECLARE rockTrack INITIALLY Track {"Ace of Spades", "Motorhead",
     2.46}
```

The reference language does, however, also include a way to create a constructor as well.

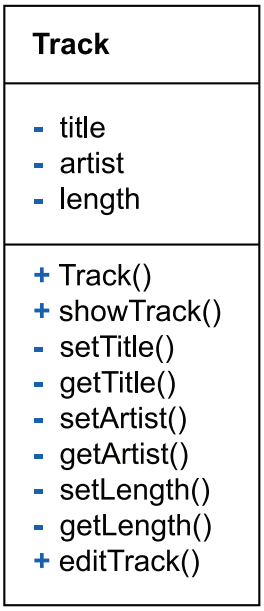
Example : Music player: constructor for the Track class

A suitable constructor for the Track class is as follows:

```
 1 CLASS Track IS {STRING title, STRING artist, REAL length}
 2
 3 METHODS
 4
 5   CONSTRUCTOR (STRING title, STRING artist, REAL length)
 6
 7     DECLARE THIS.title INITIALLY title
 8     DECLARE THIS.artist INITIALLY artist
 9     DECLARE THIS.length INITIALLY length
10
11   END CONSTRUCTOR
12
13 END CLASS
```

In a Unified Modelling Language (UML) class diagram (we will look at UML in more detail later on in this topic), this could be shown as:

Figure 2.7: Track class with constructor



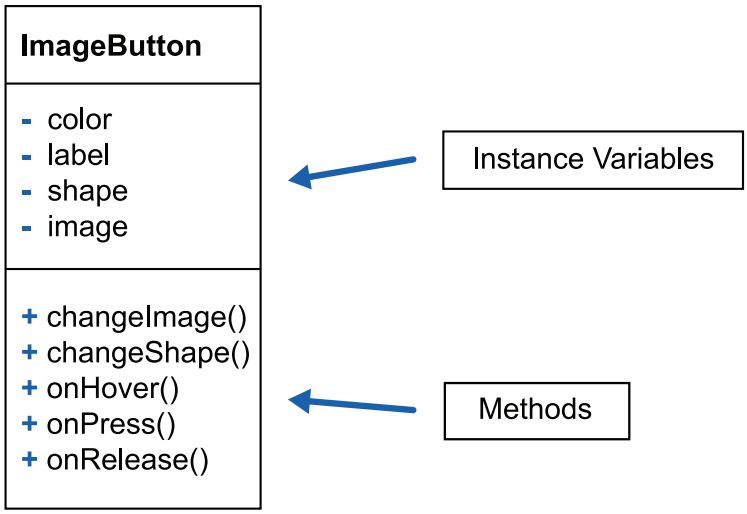
The constructor has the same name as the class and is shown in the diagram as Track().

2.2.5 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 subclasses which inherit the methods of the class above them (the class above is referred to as a **superclass**). If you need one of the subclasses to have a different method then that method can override these methods for its specific circumstances.

A subclass of the Button class from Figure 2.1 might be ImageButton which changes its image when the mouse is moved over it. This ability to create a subclass from a predefined class is called **inheritance**. The benefit of using this technique is that once a class has been defined and tested, subclasses can then be created which share the main characteristics of the parent class, thus saving testing and development time.

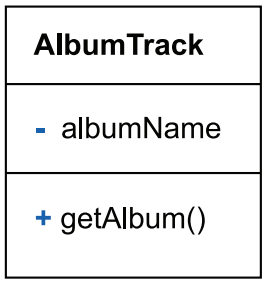
Figure 2.8: Inherited class - ImageButton



Example : Music player inheritance: AlbumTrack class

We can define a new class which inherits all the instance variables and methods from the Track superclass in Figure 2.7 and is extended with additional instance variables and methods.

Figure 2.9: Class that inherits from Track - AlbumTrack




```
1 CLASS AlbumTrack INHERITS Track WITH {STRING AlbumName}
2
3 METHODS
4
5 FUNCTION GetAlbum() RETURNS STRING
6   RETURN THIS.AlbumName
7 END FUNCTION
8
9 END CLASS
```

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.

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

2.3 Design techniques

Learning objective

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

- read and understand the design of programming solutions using the following design techniques:
  - structure diagrams;
  - pseudocode;
  - Unified Modelling Language (UML).

2.3.1 Structure diagrams

A **structure diagram** is created as part of the **top down analysis** of a software specification. Creating a structure diagram allows a developer to break down a complex problem description into a series of smaller sub problem descriptions. These sub problems can be regarded as modules within the system and they themselves may be further divided into smaller (and hopefully simpler) sub problems.

You will know from your Higher Computing Science studies that the shapes used for structure diagrams are:

Shape	Meaning
	Process
	Loop
	Selection
	Predefined function or procedure

Top level design

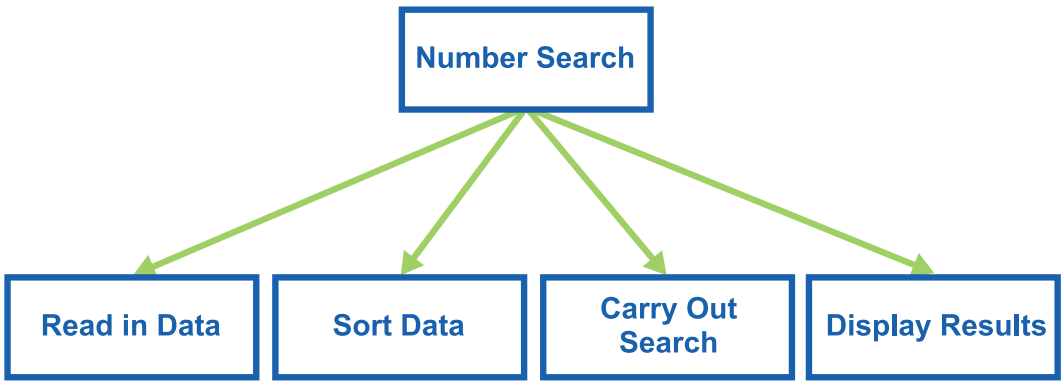
As you learned in Higher Computing Science, it is usual to approach the design of complex solutions that are solved at this level by developing a ***top level*** algorithm and then developing the refinements for each module.

Example : Structure diagram

A program is required to load 100 numbers from a file, sort the numbers into ascending order and then run a search to discover if a particular number is found in the data.

The top level structure diagram for this algorithm would be:

Figure 2.10: Top level structure diagram



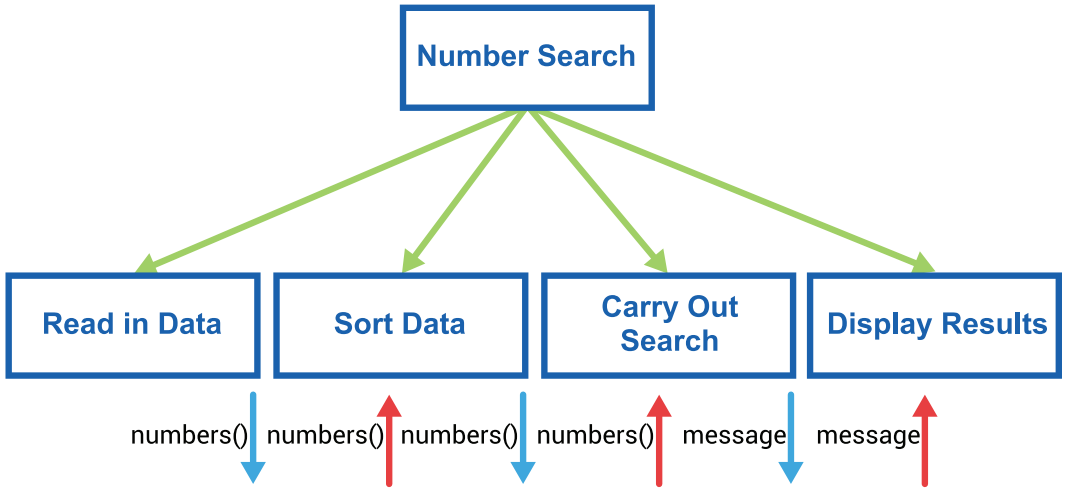
Data flow

Data flow in structure diagrams is shown using input/output arrows with variable names to show the input and the output.

Example : Structure diagram showing data flow

The data flow for the 100 numbers example in Figure 2.10 would be shown on the structure diagram as follows:

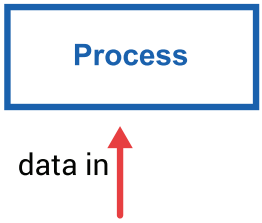
Figure 2.11: Top level structure diagram with data flow



The diagram shows an array of numbers being the output of **Read in Data**, then passed in and out of **Sort Data** and into **Carry Out Search**. The output of **Carry Out Search** is a message to be displayed depending on the required number being found or not found in the numbers array. This message is then received as the input to the **Display Results** process.

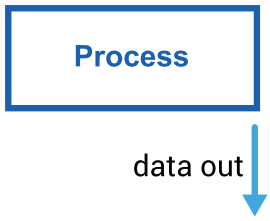
Data going into each program module is shown by an arrow pointing into it as follows:

Figure 2.12: Data flow in



Data going out of each program module is shown by an arrow pointing away from it as follows:

Figure 2.13: Data flow out



Data flow can be shown at any level in a structure diagram. As the solution to a problem is refined, data flows within processes may also be added.

Refinements

Refinements are the development of further details for a solution and they define more precisely what will happen within an element of the structure diagram.

Example : Process refinements

Each of the processes in the 100 numbers example in Figure 2.10 can be further refined as follows:

Figure 2.14: Refinement of 'Read in Data' process

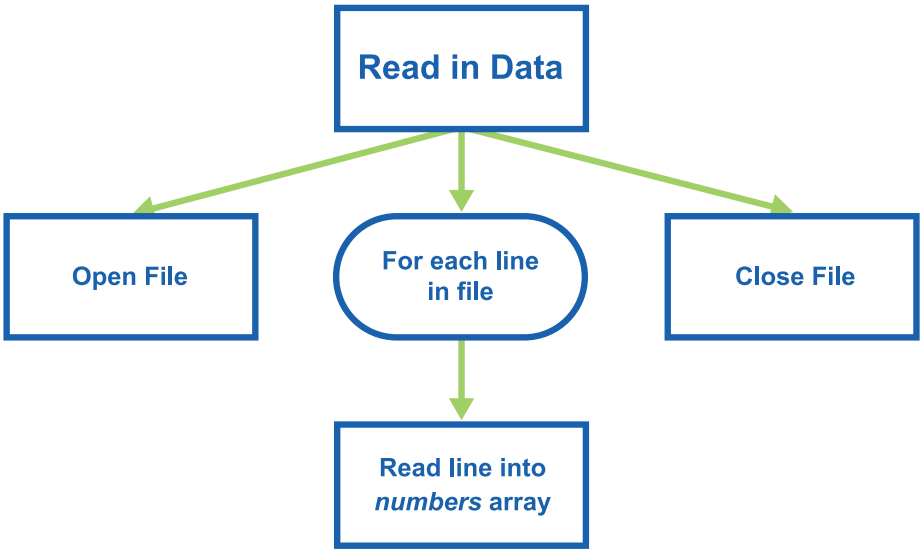


Figure 2.15: Refinement of 'Sort Data' process

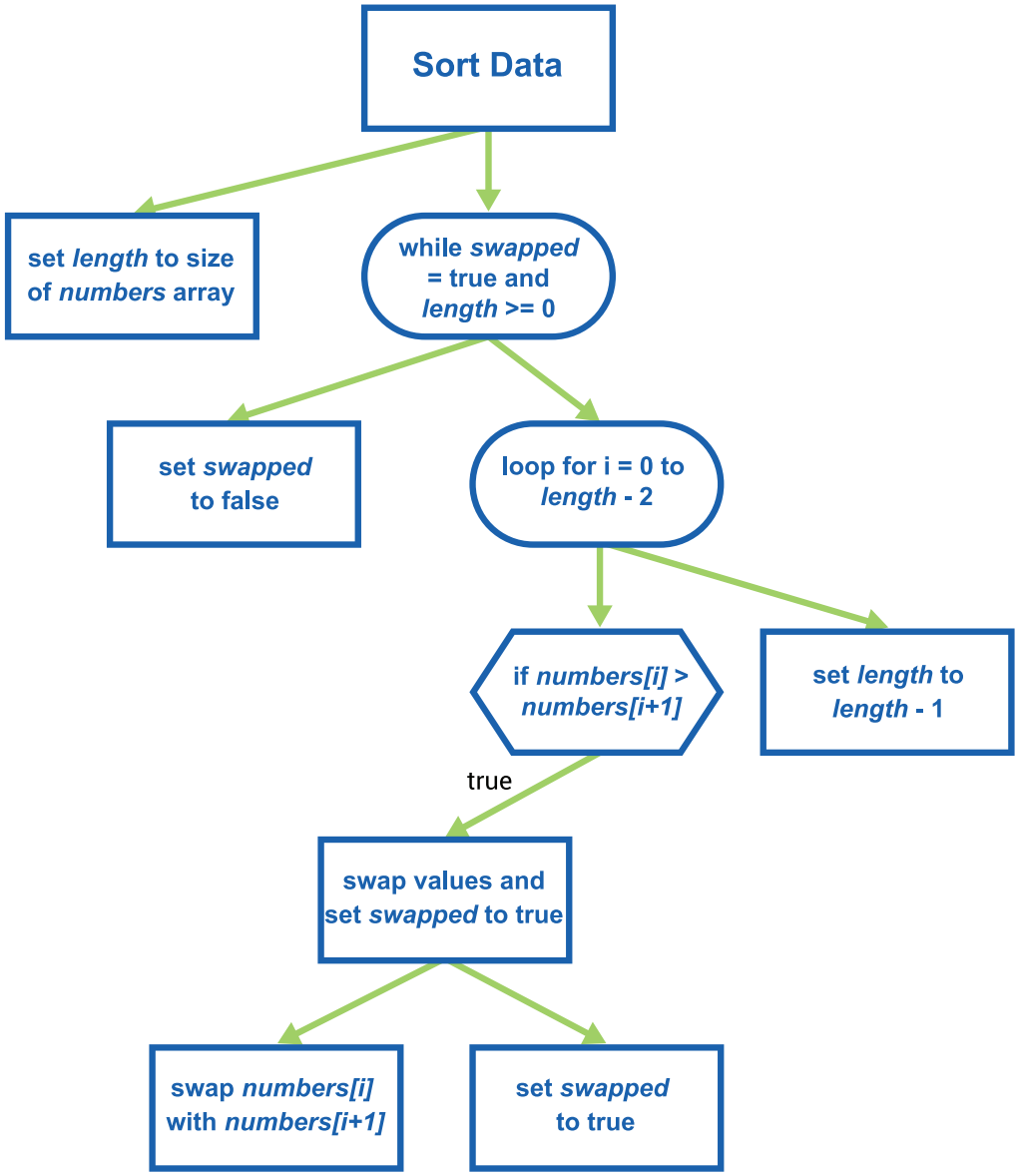
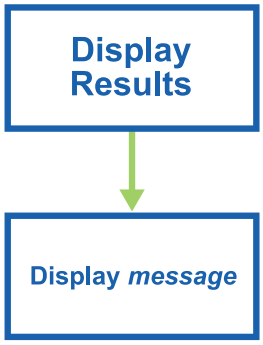


Figure 2.16: Refinement of 'Carry Out Search' process

Figure 2.17: Refinement of 'Display Results' process



As you develop solutions to problems at this level, it will be appropriate to use a variety of approaches to design. Often, structure diagrams are used to define the high level processes within a solution and then these high level processes are broken down and refined using pseudocode.

2.3.2 Pseudocode

Pseudocode is a method of describing an algorithm in a way that 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 in line with the SQA reference language specification because that is the style used in SQA assessment materials.

You will have already seen many examples of pseudocode in the Higher course.

Example : Pseudocode: linear search algorithm

This example describes a linear search algorithm to find an item in an array of numbers and give its index position.

```
1  PROCEDURE LinearSearch(ARRAY OF INTEGER numbers, INTEGER
       highestIndex)
2
3    DECLARE itemToFind INITIALLY getvalidItem(1,100)
4    DECLARE found AS BOOLEAN INITIALLY false
5    DECLARE arraySize INITIALLY highestIndex
6    DECLARE counter INITIALLY 0
7
8    REPEAT
9      SET found TO numbers[counter] = itemToFind
10     SET counter TO counter + 1
11   UNTIL found OR counter > arraySize
12
13   IF found THEN
14     SEND itemToFind & " found at position" & (counter-1) TO DISPLAY
15   ELSE SEND "Item not found" TO DISPLAY
16   END IF
17
18 END PROCEDURE
```

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

- the syntax for input from the keyboard and output to the 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 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 the following example, 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.

Example : Pseudocode: shopping algorithm

Figure 2.18: Pseudocode for a shopping algorithm

```
1  PROCEDURE Main()
2    Setup()
3    <Ask customer for choice>
4    Calculate()
5  END PROCEDURE
6
7
8  RECORD Product IS {STRING productName, INTEGER stockNumber, REAL
       price, BOOLEAN purchased}
9
10 PROCEDURE Setup()
11
12   DECLARE shoppingBasket AS ARRAY OF Product INITIALLY []
13
14   SET shoppingBasket[0] TO {productName = "USB cable", stockNumber
       = 624, price = 1.74, purchased = false}
15   SET shoppingBasket[1] TO {productName = "HDMI adaptor",
       stockNumber = 523, price = 5.00, purchased = false}
16   SET shoppingBasket[2] TO {productName = "DVD-RW pack",
       stockNumber = 124, price = 10.99, purchased = false}
17
18 END PROCEDURE
19
20 PROCEDURE Calculate()
21
22   DECLARE total INITIALLY 0
23
24   FOR counter FROM 0 TO 2 DO
25     IF shoppingBasket[counter].purchased = true THEN
26       SET total TO total + shoppingBasket[counter].price
27     END IF
28   END FOR
29   SEND "Total cost of items = £" & total TO DISPLAY
30
31 END PROCEDURE
```

Pseudocode: shopping algorithm (10 min) Go online

Q8: Expand the <ask customer for Choice> pseudocode code in the previous shopping algorithm example in Figure 2.18.

2.3.3 Unified Modelling Language (UML)

Unified Modelling Language (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 meaning that 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 one of the many diagrams specified by UML: the class diagram.

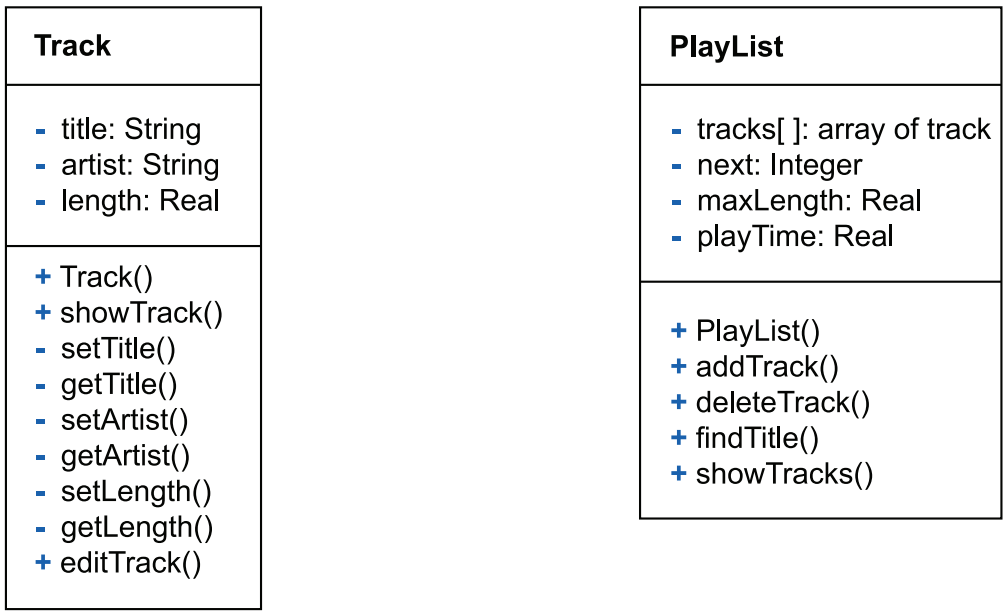
2.3.3.1 Class diagrams

Class diagrams show the classes of objects required for the system and how they are related. The class notation for a UML class diagram consists of three parts:

- **Class name:** the name of the class appears in the first section.
- **Class instance variables:** this middle section contains details of the instance variables, their access modifiers (private or public) and their data types.
- **Class methods:** the bottom section contains details of all the methods associated with the class and their access modifiers.

Let's look at the Track class and the PlayList class that you are familiar with from the earlier music player playList example.

Figure 2.19: Track class and PlayList class



Private instance variables and methods are shown with a "-" (minus) identifier and public instance variables and methods are shown with a "+" (plus) identifier.

As we can see in the diagram in Figure 2.19 we declare the data types for variables in beside their name.

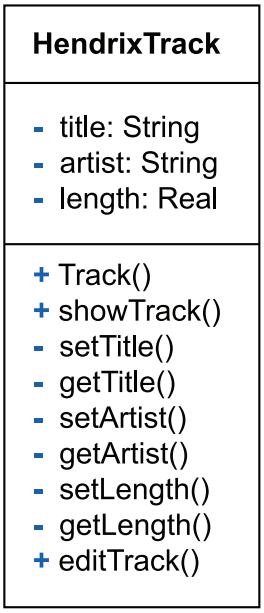
Inheritance

Suppose we wanted to define a new type of track. The **HendrixTrack** will be a class that takes on all the instance variables and methods of Track but changes the class so that it only creates tracks where the artist is "Jimi Hendrix".

The pseudocode for the HendrixTrack class would be:

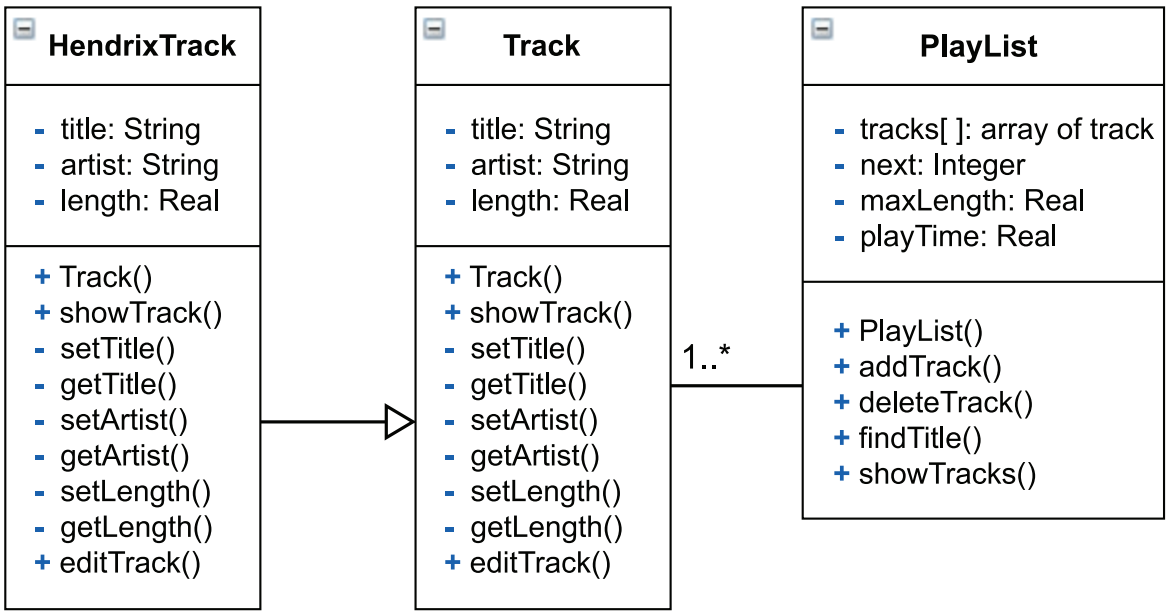
```
1  CLASS HendrixTrack INHERITS Track
2  METHODS
3
4  CONSTRUCTOR Track (STRING title, REAL length)
5    DECLARE THIS.trackTitle INITIALLY title
6    DECLARE THIS.artist INITIALLY "Jimi Hendrix"
7    DECLARE THIS.trackLength INITIALLY  0.0
8  END CONSTRUCTOR
9
10 END CLASS
```

The UML diagram for the HendrixTrack class would be:



We can show the relationship between these classes by linking them.

Figure 2.20: HendrixTrack class showing inheritance



The class HendrixTrack inherits its attributes and methods from the class Track. So, inheritance is shown as an arrow from HendrixTrack to Track.

Key point

Notice that the head of the arrow is not filled in. **This is very important**.

To show inheritance in a UML diagram, a solid line from the child class to the parent class is drawn using an unfilled arrowhead.

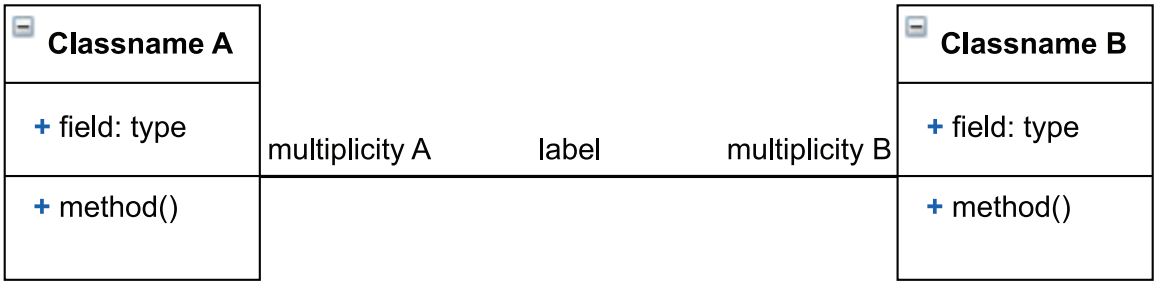
Association

In Figure 2.20 we can see that the **Track** class is associated with the **PlayList** class.

The **PlayList** has an association with **Track**. One PlayList will have at least one or more instances of Track. PlayList has an instance variable of an array of Track.

Associations show how classes may depend on each other.

Figure 2.21: Association



Associations are similar to relationships in database systems.

In UML we refer to the active logical associations as ***multiplicity***. Multiplicity is a definition of cardinality/relationships that we learned about when studying databases at Higher level.

Some examples of multiplicity are:

Multiplicity	Option	Cardinality
0..0	0	Collection must be empty
0..1		No instances or one instance
1..1	1	Exactly one instance
0..*	*	Zero or more instances
1..*		At least one instance
5..5	5	Exactly 5 instances
m..n		At least m but no more than n instances

Class: bank statement (20 min) Go online

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).

Q9: Create class diagrams for **BankStatement** and **Entry** with appropriate methods.

. . . . . .

Q10: Write the pseudocode to define these two classes with methods to add an entry to a statement and display a statement.

. . . . . .

Q11: Draw the class diagram for the bank statement activity.

2.4 Wireframes

Learning objective

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

- describe and use wireframes for user interface design to show:
  - visual layout;
  - inputs;
  - validation;
  - underlying processes;
  - outputs.

A **wireframe** is a visual layout of the user interface of an application. A wireframe typically includes details of how the interface will be laid out:

- the location of interactive elements such as buttons, dropdowns, data entry fields;
- the labels and visual directions for the user to follow to make use of the application;
- the placement of other elements of the interface such as images and text headings.

In addition to the visual elements, a wireframe will detail:

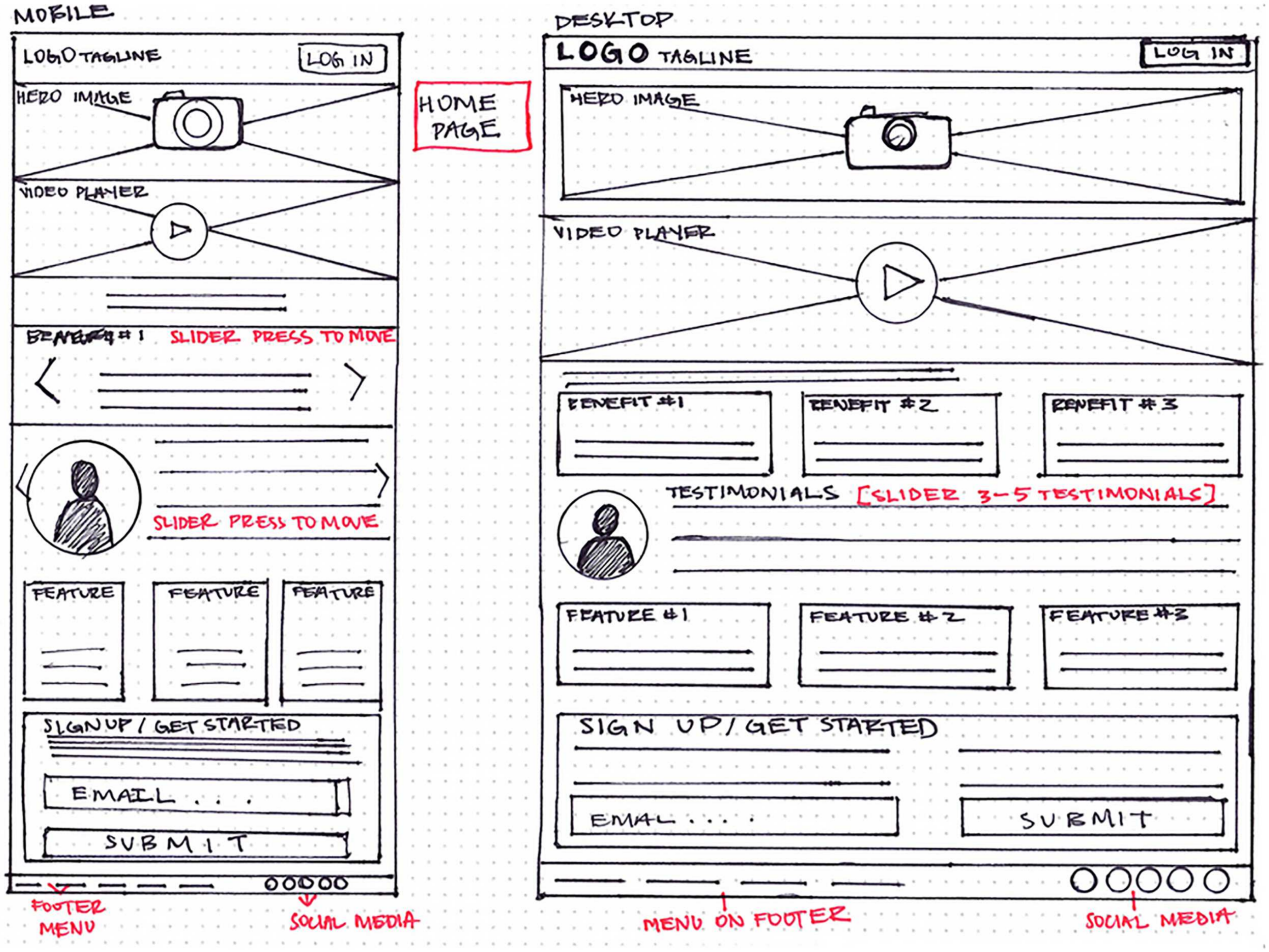
- any validation to be applied to interface elements;
- the underlying processes applied;
- the intended output to be generated at a high level.

Wireframing Go online

Watch a short video about wireframing created by UX Mastery: https://www.youtube.com/watch?v=8-vTd7GRk-w

There are different types of wireframes depending on the task required and the complexity of the job. The simplest wireframes are just a paper sketch and this is the best way to start. Don't worry if you are not a great artist - you don't need to be. Wireframes tend to be just boxes to show the layout with annotations (notes) to show what happens or what will happen when the interface is used.

Figure 2.22: A simple sketch wireframe



2.4.1 Visual layout

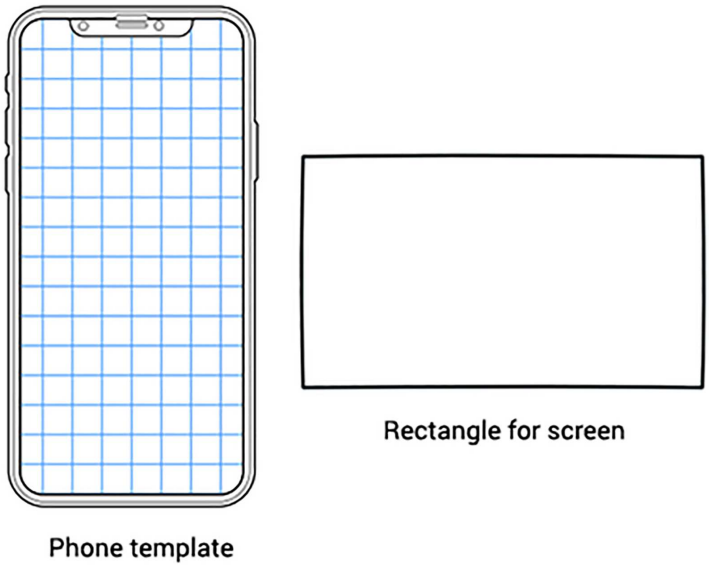
To start a wireframe, an area is defined to represent the user interface. This typically matches the display area for the application being designed.

There are many online applications which can be used for wireframe design including:

- https://www.draw.io/
- https://www.invisionapp.com/
- https://www.lucidchart.com/

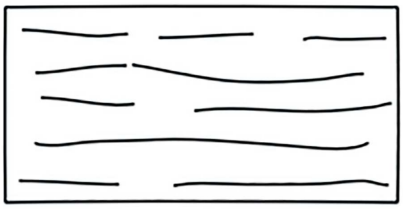
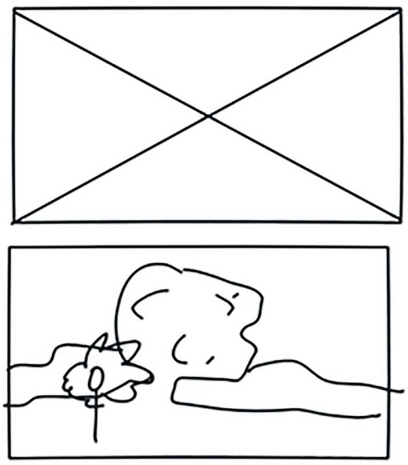
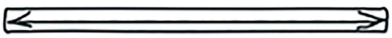
Many wireframing tools have quick start templates and predesigned interface objects that will help you to make a quick start to your design. Equally, you can grab a pencil and some paper and get started.

Figure 2.23: Example templates



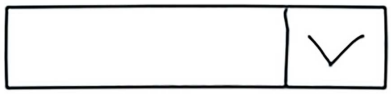
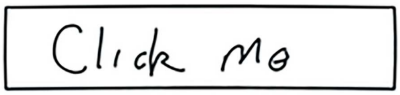
The visual elements of the design can be represented using boxes. Often text is shown as a rectangle with lines within it, images are shown as a rectangle with diagonally crossed lines or an image doodled within the rectangle.

Some examples of how visual elements can be represented are:

Visual element	Meaning
	Text
	Image
	Scroll bar

2.4.2 Inputs

The data inputs can also be represented visually. Some examples of these are:

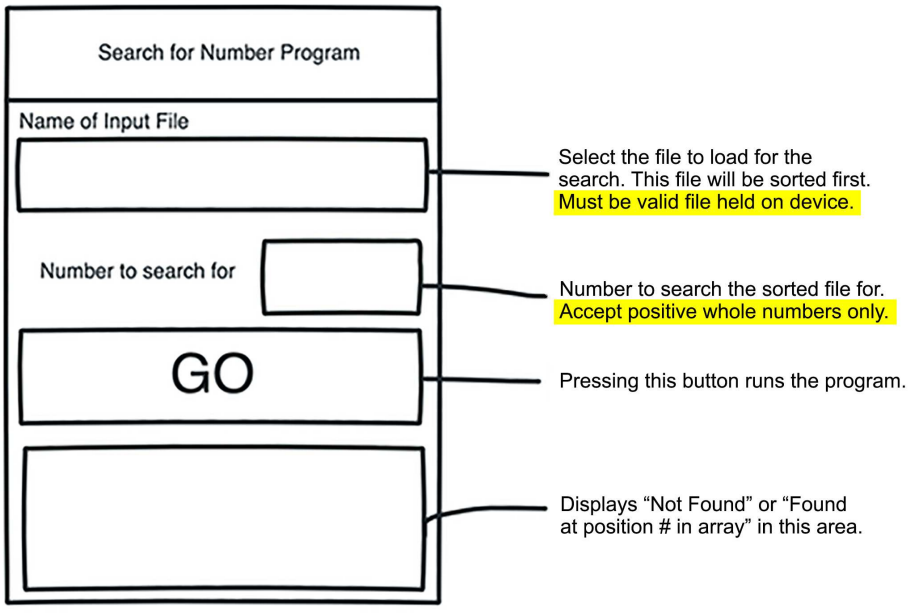
Data input	Meaning
	Data entry area
	Dropdown data selection menu
	Button
	Radio button
	Checkbox

These elements will be annotated to indicate how they will be used by the program.

2.4.3 Validation

Validation is shown through the annotations that are added to a wireframe. An example of this is:

Figure 2.24: Example of annotations showing validation



2.4.4 Underlying processes

The underlying processes can be shown as annotations or as a sequence of wireframes showing any steps in a process as they are presented to the user.

Figure 2.25: Sequence of steps interaction (underlying process) shown in wireframes

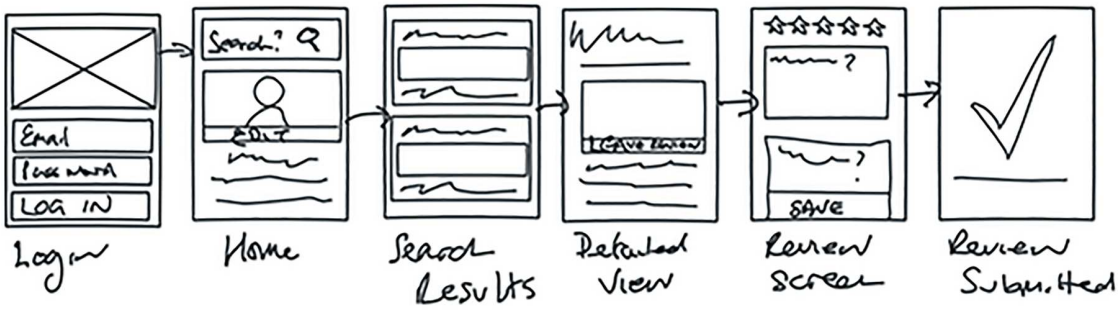
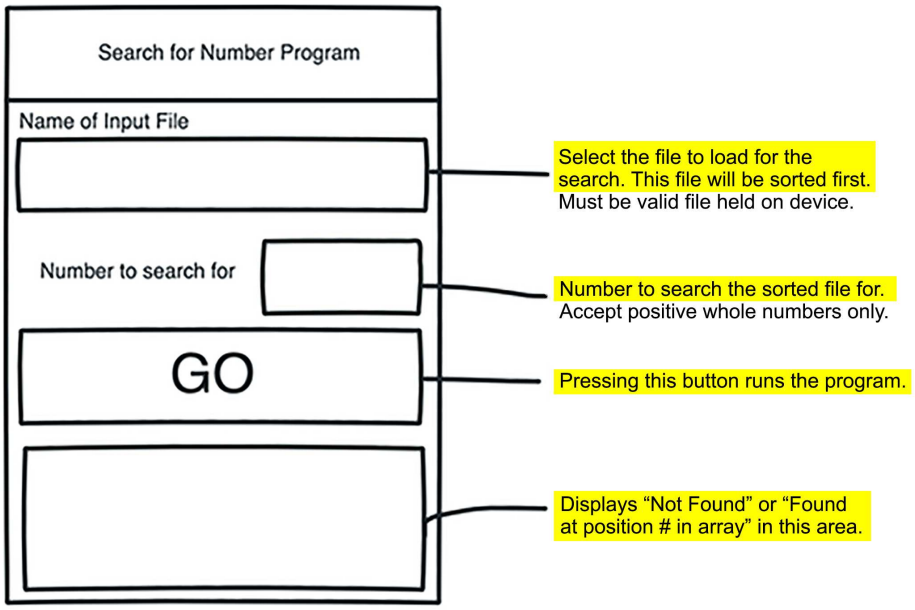


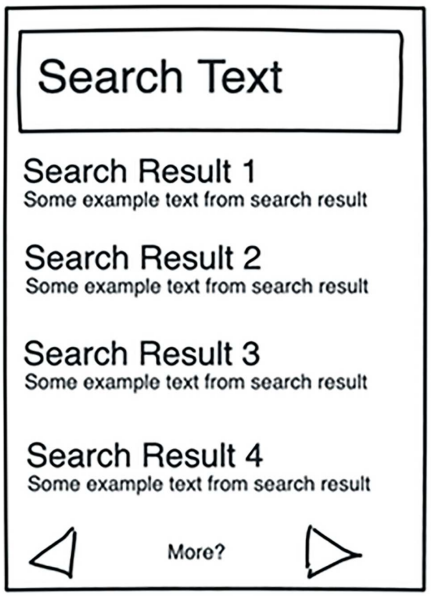
Figure 2.26: Underlying process included in annotations



2.4.5 Outputs

Outputs can also be shown in wireframes to illustrate how the output will be displayed to the user.

Figure 2.27: Showing output for search results in a program



2.4.6 Creating a wireframe

Through combining all these elements, a designer can create a wireframe representation of the application being developed. This can then be used to implement the user interface as part of the creation of the final program.

Create wireframes for the PlayList system (20 min) Go online

Let's look again at the PlayList system which we used as an example earlier. Create a wireframe for each of the activities outlined in the following questions.

You can use the UML class diagrams from the previous examples at Figure 2.19 to help you work out which instance variables each wireframe will need to include.

Q12: Create a wireframe for an application menu to allow users to select to manage playlists and tracks and navigate through the current playlist.

. . . . . .

Q13: Create a wireframe to allow users to enter and save a track.

. . . . . .

Q14: Create a wireframe to allow users to manage playlists (edit, delete and create new playlists).

. . . . . .

Q15: Create a wireframe to allow users to create a new empty playlist.

. . . . . .

Q16: Create a wireframe to allow users to add a track to a playlist.

2.5 Learning points

Summary

You should now be able to:

- describe and make use of object oriented programming to develop solutions;
- read structure diagrams, pseudocode and UML to understand the design of programming solutions;
- use pseudocode to create a solution to a problem making use of top level design, data flow and refinements;
- use UML to create class diagrams demonstrating key aspects of object oriented programming;
- describe and use wireframes for user interface design to show:
  - visual layout;
  - inputs;
  - validation;
  - underlying processes;
  - outputs.

2.6 End of topic test

End of topic test: Design Go online

Q17: 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

. . . . . .

Q18: 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

. . . . . .

Q19: In object oriented programming _____ is the process of creating new classes, called subclasses, from an existing class.

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

. . . . . .

Q20: In a structure diagram data flow is shown by:

a) a process box containing instance variables.
b) labels on the lines connecting the elements of the structure diagram.
c) arrows below the process showing the data flowing in and out with appropriate labels.
d) a linked data flow diagram with suitable labels.

. . . . . .

Q21: Which of these statements are true of a class diagram (choose all that are true):

a) It shows the instance variables and methods for a class and the nature of the encapsulation (private/public) for each.
b) It shows the association between classes.
c) It shows how one class may inherit the details of another class.
d) It shows the code definition for a class in pseudocode.

Topic 3

Implementation

Contents

Prerequisites

From your studies at Higher you should already know:

- the difference between a simple and a structured data type;
- the different data types available in your chosen programming language;
- how your programming language handles records;
- how to use standard algorithms such as counting occurrences and linear search.

Learning objective

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

- the features of object oriented languages:
  - classes;
  - properties;
  - methods;
  - constructors;
  - instantiation;
  - object;
  - inheritance;
  - subclasses;
  - polymorphism.
- records and arrays of records;
- parallel 1D arrays;
- 2D arrays;
- arrays of objects;
- single and double linked lists;
- database connectivity using PHP and SQL;
- algorithm specifications for bubble sort, insertion sort and binary search.

3.1 Revision

Quiz: Revision Go online

Q1:

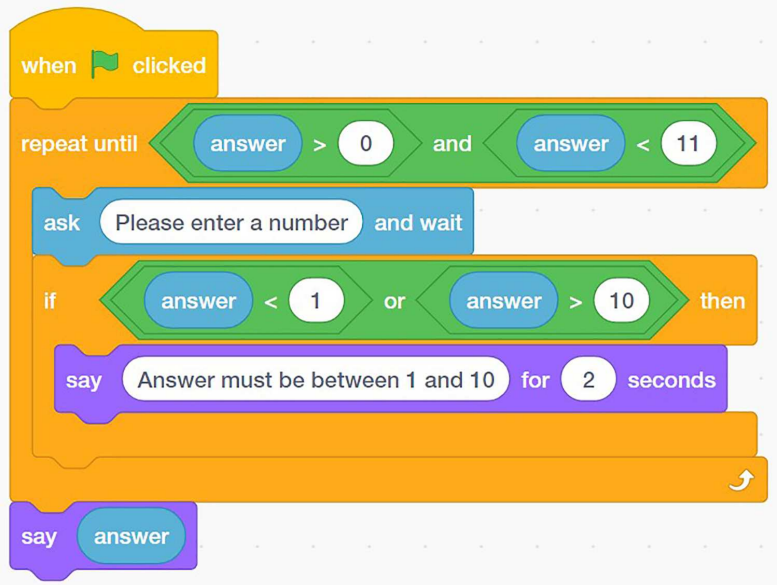


Image courtesy of Lifelong Kindergarten Group at the MIT Media Lab (CC BY-SA 2.0)

This Scratch code is an example of:

a) counting occurrences.
b) input validation.
c) linear search.
d) finding the maximum.

. . . . . .

Q2:

```
1  FUNCTION getstuff(List,Value)
2    DECLARE total AS INTEGER INITIALLY 0
3    FOR counter = 0 TO length(List) - 1
4      IF List(counter) = Value THEN
5        SET total = total + 1
6      END IF
7    END FOR
8    RETURN total
9  END FUNCTION
```

This code is an example of:

a) counting occurrences.
b) input validation.
c) linear search.
d) finding the maximum.

.....

Q3: The linear search algorithm may make use of:

a) a fixed loop and a boolean variable.
b) a conditional loop and a total 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:

```
1 FUNCTION CheckForValue (List, Target) RETURNS STRING
2   FOR i = 0 TO length(List)-1
3     IF List(i) = Target THEN
4       RETURN "Found at position " & i
5     END IF
6   END FOR
7   RETURN "Not Found"
8 END FUNCTION
```

This code is an example of:

a) counting occurrences.
b) input validation.
c) linear search.
d) finding the maximum.

3.2 Computation constructs

Learning objective

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

- understand the features of object oriented languages:
  - classes;
  - properties;
  - methods;
  - constructors;
  - instantiation;
  - object;
  - inheritance;
  - subclasses;
  - polymorphism.

Object oriented programming makes use of a number of programming constructs. Some of these were introduced in the previous **Design** topic. In this section, we will focus more on how to implement these computational constructs.

3.2.1 Class

A **class** is a template for what will be stored about a thing and what processes will be applied to that thing. Think of a class as a way of bundling everything together, creating a template that can be used to make ***real*** versions (instances) of the thing.

Example : Dog class

A class of **dog** doesn't represent a real dog, it just represents what will be ***stored*** about the dog:

- colour;
- height;
- length;
- weight;
- breed etc.

and the ***processes*** that a dog might have:

- sit;
- run;
- jump etc.

The SQA reference language defines a class as follows:

```
CLASS classname IS {DATATYPE property, ...}

METHODS
  ...
END CLASS
```

3.2.2 Property

A **property** is something which is held in the class and is recorded about the class.

For example, the properties for the **dog** class would be:

- colour;
- height;
- length;
- weight;
- breed etc.

A property is part of the ***template***, the class, that is used to create or instantiate actual objects. For example Dog is the class which would be used to create an object, Rex.

The properties or data items in a class definition are also known by other names such as **attributes** and **instance variables**.

Properties will have constraints such as data type, size etc. which will be defined in the class declaration.

The SQA reference language defines the properties of a class within the class declaration as follows:

```
CLASS classname IS {DATATYPE property, ...}
```

The properties of the class are established within the curly brackets { }.

Example : Dog class: properties

If we think about a class for dog this would be:

```
CLASS dog IS {STRING name * 30, STRING colour * 12, REAL height,
    REAL length, REAL weight, STRING breed}
```

This defines the properties of dog as:

- a text string, 30 characters long, called name;
- a text string, 12 character long, called colour;
- a real value called height;
- a real value called length;
- a real value weight;
- a text string, breed, of indeterminate length.

3.2.3 Method

A **method** defines the processing that will be applied to real things known as ***objects*** that are created or instantiated from a class.

All properties in a class will have a **getter** and a **setter** method at the very least. These are methods which assign a value to the property and which read the current value of the property.

Methods can be procedures or functions and you will be familiar with these from your studies at Higher level.

In the SQA reference language, methods are declared within a class as:

```
1  CLASS classname IS {DATATYPE property, ...}
2
3  METHODS
4
5    PROCEDURE methodname (DATATYPE parameter, ...)
6    ...
7    END PROCEDURE
8    ...
9    FUNCTION methodname (DATATYPE parameter, ...) RETURNS DATATYPE
10   ...
11   END FUNCTION
12
13 END CLASS
```

Constructor

There is also a **constructor** method which is automatically called when an object is instantiated based on the class. Think of this as a procedure that runs when the object is created.

This constructor method is used to setup the initial values for properties and to carry out any processing required when the object is created.

Key point

The constructor only runs once, when an object based on the class is created (instantiated).

A constructor method is usually the first method defined in a class and it has a special declaration.

```
CLASS classname IS {DATATYPE property, ...}

METHODS

  CONSTRUCTOR (DATATYPE parameter, ...)
  ...
  END CONSTRUCTOR

END CLASS
```

Referencing properties

Referring to **properties** within a method is done using the **THIS.** prefix. This will reference the properties defined within the curly brackets { } in the class definition.

Example : Dog class: referencing properties

For the dog class, we would reference properties using THIS. as follows:

Figure 3.1: Dog class

```
CLASS dog IS {STRING name * 30, STRING colour * 12, REAL height,
    REAL length, REAL weight, STRING breed}

METHOD

  CONSTRUCTOR (STRING dogName * 30, STRING dogColour * 12, REAL
      dheight, REAL dlength, REAL dweight, STRING dbreed)

  DECLARE THIS.name INITIALLY dogName
  DECLARE THIS.colour INITIALLY dogColour
  DECLARE THIS.height INITIALLY dheight
  DECLARE THIS.length INITIALLY dlength
  DECLARE THIS.weight INITIALLY dweight
  DECLARE THIS.breed INITIALLY dbreed

  END CONSTRUCTOR

  FUNCTION getName() RETURNS STRING
    RETURN THIS.name
  END FUNCTION

  PROCEDURE setName(STRING dogName)
    SET THIS.name TO dogName
  END PROCEDURE
  ...
END CLASS
```

3.2.4 Instantiation

The process of creating an object from a class is called **instantiation**. An object is a ***real*** item rather than an abstract definition (a class).

When the object is instantiated from the class, it is given a unique name, declared with a set of properties (instance variables) and defined methods from the class. The constructor method is also executed.

In the SQA reference language an object is instantiated by declaring it using the class name and any initial values required by the constructor.

```
DECLARE object INITIALLY class (dataValue, ...)
```

3.2.5 Object

An **object** is created when a class instantiated. This process sets up the properties (instance variables) for the object, against the name used to created it, and runs the constructor method.

Example : Dog class: object

Rex is an object of the dog class. Rex is declared using the syntax:

Figure 3.2: Rex object

```
DECLARE Rex INITIALLY dog ("Rex", "Black", 58.1, 62.2, 34.4,
   "Labrador")
```

This creates an object, called Rex, with properties as shown in the example code, and the methods as defined in the dog class.

The methods of this object can then be accessed just as if they were set up as a record field as follows:

```
Rex.setWeight(35.1)
```

Because of the rules of object oriented programming, it is not possible to directly access the property values (instance variables) of the object.

3.2.6 Encapsulation

Encapsulation is the concept of holding the properties and methods of a class (and any objects instantiated from it) in one ***container***.

Encapsulation protects the contents of this container, the class, from the outside world. This means that the properties of the class will be **private** (as opposed to **public**) as they are only accessible by methods that are defined as part of the class they belong to.

This is why ***getter*** and ***setter*** methods are so important. Without these methods, it would not be possible to view the property values or set the property values within an object.

In the SQA reference language, all properties (instance variables) are private and inaccessible/invisible outside the methods that are defined within the class.

Create a dog class Go online

Q6: Using your preferred programming language create a dog class as described in Figure 3.1. Include the constructor, getter and setter methods for all properties.

Declare an object based on this class using the information for Rex in Figure 3.2. Use the getter methods of this class to display this information for each property.

3.2.7 Inheritance

Inheritance is a key concept in object oriented programming which means that a class can be defined based on another class. The class being defined takes on all the characteristics of the class it is based on and therefore inherits all the properties and all the methods of the original class.

When a class is declared and inherits from another class, it is possible to add additional methods and properties which are specific to the new class.

In the SQA reference language a class is declared, making use of inheritance, as follows:

```
1 CLASS newClass INHERITS baseClass WITH {DATATYPE additionalProperty,
    ...}
```

Example : Police dog class: inherits from dog class

If we want to create a new class of **police dog**, this would inherit everything from the class **dog** and add some new properties and methods:

```
1 CLASS policeDog INHERITS dog WITH {STRING policeID, STRING handler}
```

This speeds up the development of software as programmers can reuse classes which are similar rather than create new ones.

3.2.8 Subclass

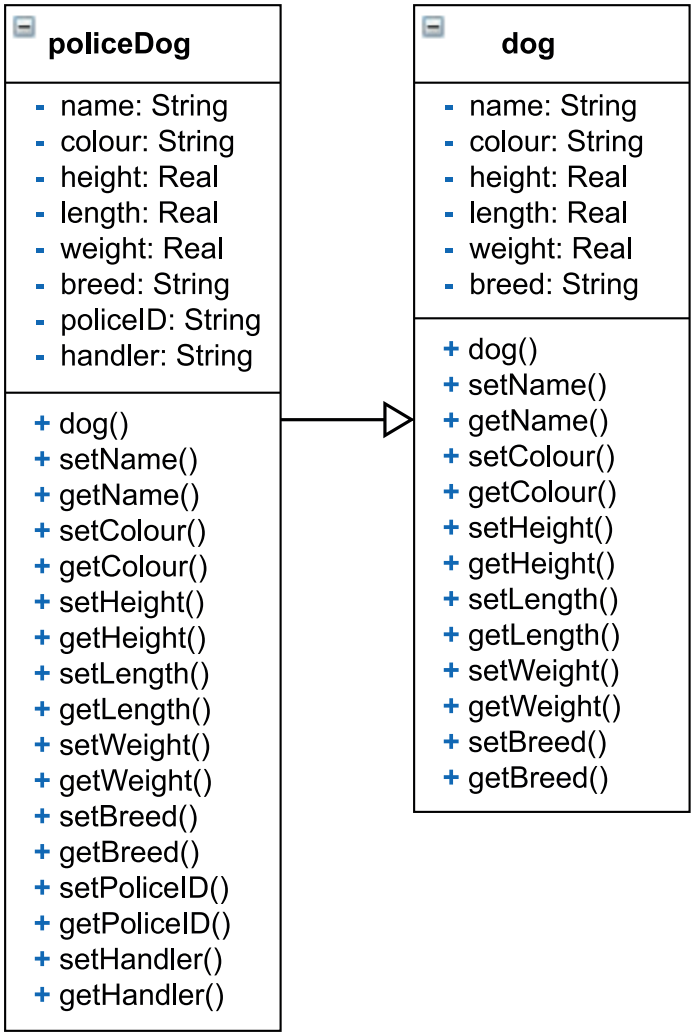
When a class inherits from another class, we call it a **subclass**. The class that a subclass inherits from is referred to as a **base class** or **super class**.

Example : PoliceDog subclass

Using object oriented programming language, this would be explained as: the subclass policeDog is derived from the super class dog. This means that the subclass policeDog inherits all the properties and methods of the super class dog.

We can see this in the following **Unified Modelling Language (UML)** diagram:

Figure 3.3: UML diagram showing subclass and inheritance



Implement the policeDog subclass Go online

Q7: Using your preferred programming language implement the policeDog subclass from Figure 3.3 including the two additional properties and their associated setters and getters.

3.2.9 Polymorphism

As well as being able to add new properties and methods, a subclass can also change the properties and methods that it inherits. This ability of a subclass to inherit and change properties and methods from the base class it inherits from is called **polymorphism**.

Example : SheepDog subclass: polymorphism

This defines the properties of the class dog:

```
CLASS dog IS {STRING name * 30, STRING colour * 12, REAL height,
    REAL length, REAL weight, STRING breed}
```

The following code replaces the string breed with a similarly named property of type boolean:

```
CLASS sheepDog INHERITS dog WITH (OVERRIDE BOOLEAN breed)
```

The new class will also need to change the methods associated with this property:

Figure 3.4: SheepDog subclass

```
CLASS sheepDog INHERITS dog WITH (OVERRIDE BOOLEAN breed)

METHODS
  ...
  OVERRIDE FUNCTION getBreed()
    IF THIS.breed = TRUE THEN
      RETURN "Border Collie"
    ELSE
      RETURN "Unknown"
    END IF
  END FUNCTION
  ...
END CLASS
```

Implement the sheepDog subclass using polymorphism Go online

Q8: Using your preferred programming language implement the sheepDog subclass from Figure 3.4.

3.3 Data types and structures

Learning objective

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

- records and arrays of records;
- parallel 1D arrays;
- 2D arrays;
- arrays of objects.

Programming at this level uses a number of data types and structures which allow solutions to be developed to more complex problems. We will look at each of the following data types and structures in turn:

1. records;
2. arrays of records
3. parallel 1D arrays;
4. 2D arrays;
5. arrays of objects.

3.3.1 Records

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 consist of fields of different types.

Example : Storing product information for a website

Storing the product information for items for sale on a website might need the following information to be stored:

- productName
- stockNumber
- price

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

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

A record can be defined in this way and the record can be used to define an array of records of this structure. We will look at arrays of records in the next section.

Dot notation

To refer to a field within a record, **dot notation** is used. Using dot notation, an object can be specified and then a property of that object can be referenced.

For example, the object **Rex** has a property of `name`. The value of `name` can be referenced using the dot notation:

```
1 Rex.name
```

To refer to the value of `ProductName` in the record `product` we would use the dot notation:

```
1 product.ProductName
```

Implement a record structure Go online

Q9: Using a programming language of your choice, write a programme to implement the following record structure, assign the values shown to it and display these values on screen with appropriate labelling.

- Define a record structure called tweet with the fields: handle (string), message (string), views (integer).
- Allocate the values:
  - handle: "compednet"
  - message: "This is a great example of a record"
  - views: 178

3.3.2 Arrays of records

A record structure can be treated as a data type and an array can be created of that record structure.

Example : Creating an array of records called products

The following declaration will create an array called `products` based on the record structure `product` which is initially empty and has 10 elements:

```
1 DECLARE products AS ARRAY OF product INITIALLY [] * 10
```

To reference the field `price` in the fourth element of the array the following code would be used:

```
products[3].price
```

Example : Creating an array of records for a product inventory

We have the following information for three items for sale on a website:

productName	stockNumber	Price £
USB Cable	624	1.74
HDMI Adaptor	523	5.00
DVD-RW pack	124	10.99

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 as follows:

Figure 3.5: Product inventory 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}
SET productInventory[2] TO {productName = "DVD-RW pack",
    stockNumber = 124, price = 10.99}
```

The following code would display the contents of the product inventory and the total value of inventory 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 inventory held = £" & totalValue TO DISPLAY
```

Arrays of records (30 min) Go online

Q10: Implement the code for the product inventory example in Figure 3.5 in your chosen programming language, but with an additional boolean field in the record structure indicating whether the inventory is to be reordered or not.

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

3.3.3 Parallel 1D arrays

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

The following statement would set up an array of 10 integers indexed from 0 to 9.

```
DECLARE myArray AS ARRAY OF INTEGER INITIALLY [] * 10
```

Once values have been assigned to the array it could look like this:

myArray:

Index	Value
0	34
1	6
2	77
3	0
4	45
5	23
6	6
7	27
8	92
9	5

Parallel 1D arrays are a method for storing related data values using multiple arrays.

Example : Parallel 1D arrays

To store the details of a song, the artist and number of plays it has, three 1D arrays could be used.

Figure 3.6: Parallel 1D arrays: song, artist, number of plays

song:

Index	Value
0	Xanadu
1	I'm Not Okay
2	Kiss
3	Light My Fire
4	Barcelona

artist:

Index	Value
0	Rush
1	My Chemical Romance
2	Prince
3	The Doors
4	Twin Atlantic

plays:

Index	Value
0	3092
1	2980
2	3010
3	1893
4	429

Each array is a separate data structure, however the elements in the same positions are linked.

So we can see, on the highlighted row, that the song "Xanadu", in position 0, is by the artist "Rush" and has been played 3092 times.

Song, artist, number of plays program (30 min) Go online

Q11: Write code in your preferred programming language to implement the song/artist/number of plays program from Figure 3.6.

When run, the program will set up three arrays and assign the data to them. It will then display song, artist and plays for each song.

3.3.4 2D arrays

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 2D grid is the obvious solution.

A **2D array** is an array with two indexes.

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

This will declare the following array:

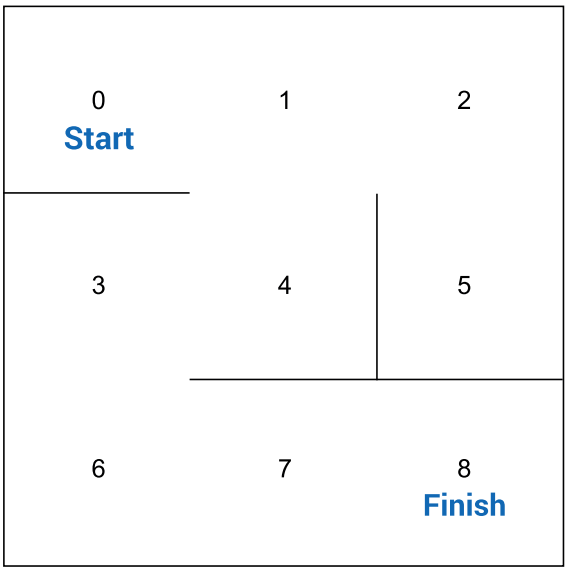
	0	1	2	3
0	23	6	78	0
1	16	4	14	27
2	18	8	102	34
3	49	4	8	45
4	5	7	93	2

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

This command will display 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!

Example : 3 x 3 maze using a 2D array

Let's look at how we could represent a very simple 3 x 3 maze using a 2D 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 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.

	North (0)	East (1)	South (2)	West (3)
Room 0		1		
Room 1		2	4	0
Room 2			5	1
Room 3		4	6	
Room 4	1			3
Room 5	2			
Room 6	3	7		
Room 7		8		6
Room 8				7

A move in a direction which hits a wall can be represented by an integer outwith 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:

	0	1	2	3
0	-1	1	-1	-1
1	-1	2	4	0
2	-1	-1	5	1
3	-1	4	6	-1
4	1	-1	-1	3
5	2	-1	-1	-1
6	3	7	-1	-1
7	-1	8	-1	6
8	-1	-1	-1	7

This grid can be represented as a 2D array:

```
1 DECLARE maze [9,4] AS INTEGER
```

The easiest way to fill this maze would be to read in the contents as a sequential text file. To start with however we are going to fill it with a set of commands:

```
1 FOR row FROM 0 TO 8 DO
2   FOR column FROM 0 TO 3 DO
3     SET maze[row, column] TO -1
4   END FOR
5 END FOR
```

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

```
1 SET maze[0,1] TO 1
2 SET maze[1,1] TO 2
3 SET maze[1,2] TO 4
4 SET maze[2,2] TO 5
5 SET maze[2,3] TO 1
6 ...
```

. . . and so on.

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

```
1 PROCEDURE ChangeRoom (REF room, direction)
2   IF maze[room, direction]= -1  THEN
3     SEND "you have hit a wall" TO DISPLAY
4   ELSE
5     newRoom = maze[room, direction]
6     SEND "You are now in room "& newRoom
7   END IF
8   room = newRoom
9 END PROCEDURE
```

3 x 3 maze using a 2D array (60 min) Go online

Q12: Implement the maze from the example 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.

3.3.5 Arrays of objects

An array of objects behaves just like an array of records would. The following example is similar to the Playlist example we looked at in the **Design** topic, but uses a simpler track class.

Example : Music player: array of objects

First we can declare a `Track` class:

```
1 CLASS Track IS {STRING title, STRING artist}
2
3 METHODS
4
5   PROCEDURE showTrack()
6     SEND THIS.title & " " & THIS.artist TO DISPLAY
7   END PROCEDURE
8
9 END CLASS
```

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

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

METHODS

  CONSTRUCTOR Collection (INTEGER size)
    DECLARE tracks INITIALLY []
    DECLARE size INITIALLY 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:

```
DECLARE sixtiesTracks INITIALLY Collection(10)

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:

```
sixtiesTracks.showTracks
```

Music player: array of objects

Q13: Implement the music player algorithm example in your chosen programming language, using your own choice of music.

3.4 Linked lists

Learning objective

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

- single and double linked lists.

A **linked list** is a data structure that allows sequences of values to be chained together. This chain can be followed to find values in the list.

Each link in the chain is called a **node**. The first node in the list is often referred to as the **head** and the remainder of the list, the **tail**. Sometimes, tail is also used to refer to the last node in the list.

Normally a linked list is set up as a data structure with a special variable called **head**, for the practical purpose of knowing where the head node is! This head variable refers to the location of the first node in the linked list.

We will look at two types of linked lists:

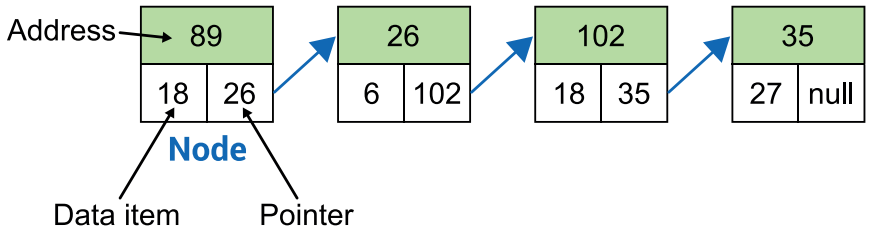
1. Single;
2. Double.

3.4.1 Single linked lists

A single linked list has two items for each node in the chain. These are:

- data item;
- pointer (to the next node).

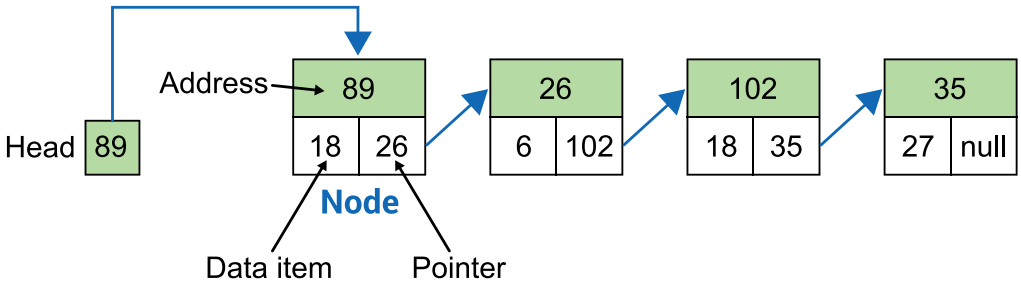
The data item is the data stored in the node and the pointer is a link to the next node in the sequence.



In this example, **head** would be set to 89 (which is the address of the first node in the linked list).

Traversing a single linked list

A single linked list can only be easily traversed in one direction through the nodes.



In this example, the head variable has a value of 89. This points to the first node (in location 89 with a data value of 18) which has a next pointer of 26. 26 points to the next node. From 26 we go to 102 and from 102 we go to 35 which is the last node. The null is an empty value (null is used where no value is set) and is used to terminate the linked list.

Attempting to navigate in the reverse direction in a single linked list is extremely difficult. If you start at the last node, there is no pointer to the previous node, making any reverse traversal of the list extremely difficult (you would have to complete a traversal in the normal direction and record the previous addresses to complete this!).

Searching a single linked list

A search of a linked list is a traversal with the addition of checking the data value of each node with the value or values being searched for.

Insertion in a single linked list

Linked lists are used in programs because it is relatively easy to insert and delete items from a list by amending the pointers.

Insertion at the beginning of a linked list

Inserting a node at the beginning of a linked list is relatively straightforward.

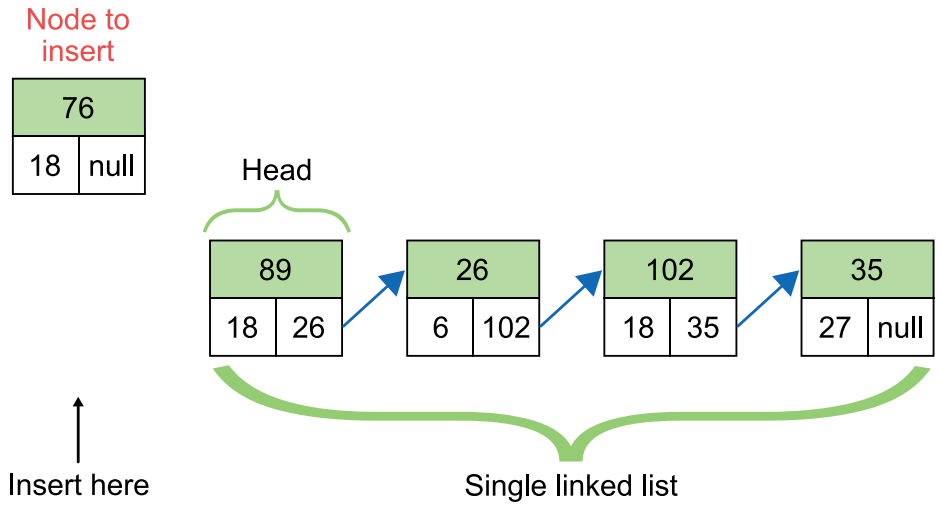
The node to be inserted is set as the new head of the list and the pointer for the inserted node is set to link to the previous head node. The variable for head is updated to point to the inserted node.

Example : Insertion at the beginning of a linked list

In this example, the node at address 76 (with the value of 18) is to be inserted at the start of the linked list before the node with address 89.

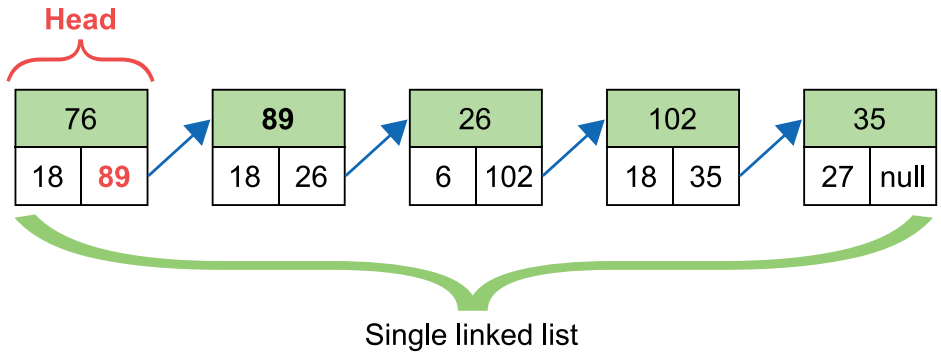
So this:

Original



Becomes this:

Updated



In this example, the head variable would be set to 76, and the pointer for node 76 set to 89.

Insertion at the end of a linked list

Inserting a node at the end of a linked list is again, relatively straightforward.

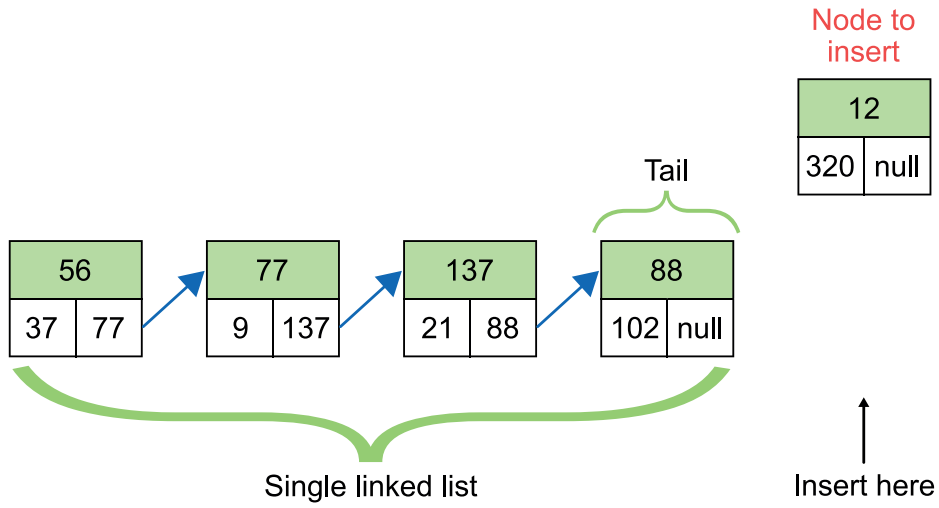
To insert at the end of a list, the last node has to be found. This requires that the list is traversed until we arrive at the last node. Then the pointer for the last node is updated (from null) to point to the node being added. The pointer for the node being added remains as a null value.

Example : Insertion at the end of a linked list

In this example, the node at address 12 (with the value of 320) is to be inserted at the end of the linked list after the node with address 88.

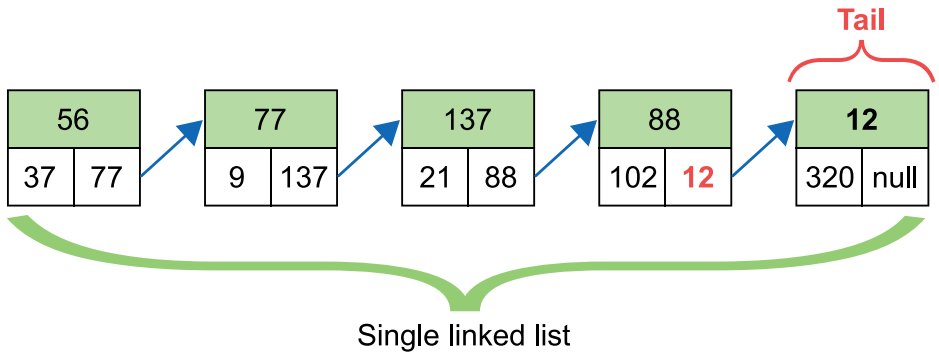
So this:

Original



Becomes this:

Updated



Insertion after a specified node

To insert a new node after a specified node we firstly need to identify the insertion point. Once we have done this we only need to handle the node before the insertion point and the node being inserted.

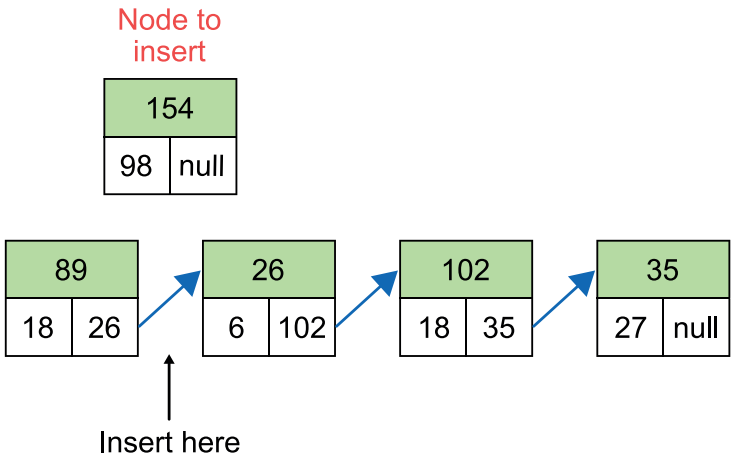
To insert the node:

1. Write the value of `specified.pointer` to a temporary variable.
2. Write the address of the node being inserted to `specified.pointer`.
3. Write the temp value to the pointer of the node being inserted.

Example : Insertion after a specified node

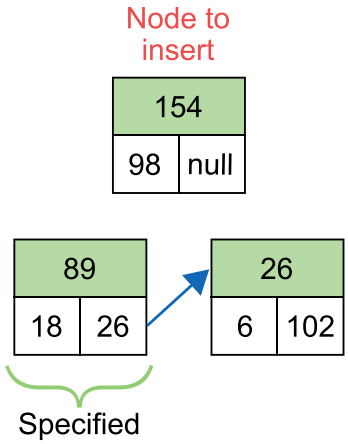
In this example, the node at address 154 (with the value of 98) is to be inserted after the node with address 89.

Original



Once we have identified the insertion point, we only need to handle the node before the insertion point and the node being inserted (the node following is included in the diagram here for reference).

We refer to the node before the insertion point as the **specified** node:

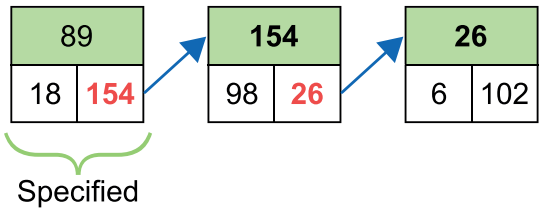


To insert the node:

1. Write the value of `specified.pointer` to a temporary variable (`tmp = 26`).
2. Write the address of the node being inserted to `specified.pointer` (`specified.pointer = 154`).
3. Write the temp value to the pointer of the node being inserted (`newnode.pointer = 26`).

So, the nodes then become:

Updated



Deletion in a single linked list

Deleting a node from a linked list just requires removing the pointer value that points to the node.

Deleting from the beginning of a linked list

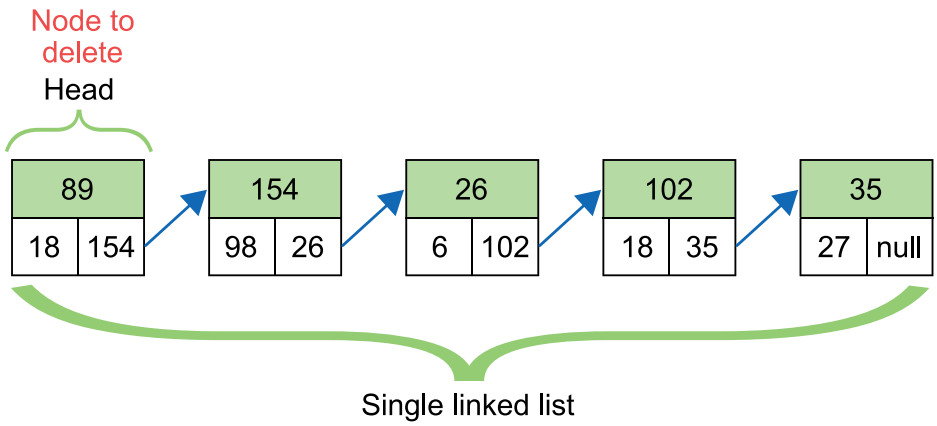
This simply requires the head of the list to be changed to the next node.

Example : Deleting from the beginning of a linked list

In this example, the node at address 89 (with the value of 18) is to be deleted at the start of the linked list.

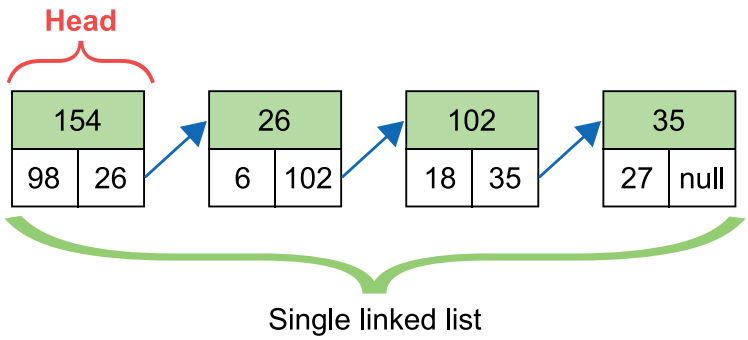
So this:

Original



Becomes this:

Updated



In this example, once the node at address 89 has been deleted we update the head variable for the linked list to 154.

Deleting from the end of a linked list

Deleting a node from the end of a linked list requires the null value that marks the end of the list to be moved to the preceding node.

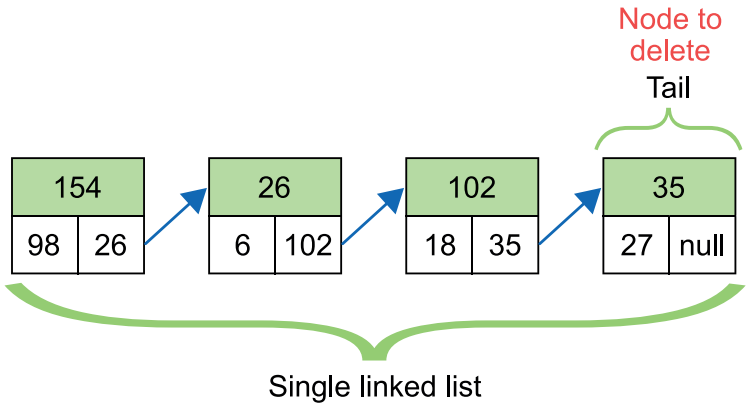
This means that the pointer for the second last node will be set to null.

Example : Deleting from the end of a linked list

In this example, the node at address 35 (with the value of 27) is to be deleted at the end of the linked list.

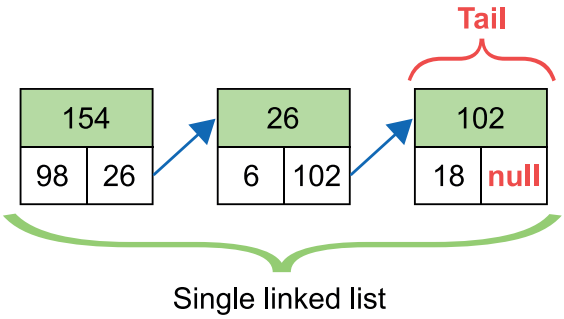
So this:

Original



Becomes this:

Updated



Deleting after a specified node

To be able to delete a node from after a specified node, we firstly need to locate the node that we want to delete (by traversing the list) and then to delete the required node we do the following:

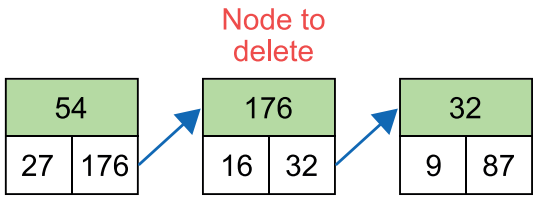
1. Write the pointer of the node being deleted to a temporary variable.
2. Set the pointer of the specified node to the temporary variable.

Example : Deleting after a specified node

In this example the node to be deleted has an address of 176 (and data value of 16).

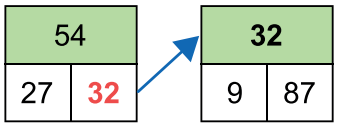
So this:

Original



Becomes this:

Updated



Key point

In Advanced Higher Computing Science, you need to be able to describe and explain the operation of single linked lists but you are not required to write program code for their operation.

3.4.2 Double linked lists

A double linked list resolves the difficulty of traversing the list in the opposite direction by having two pointers rather than just one.

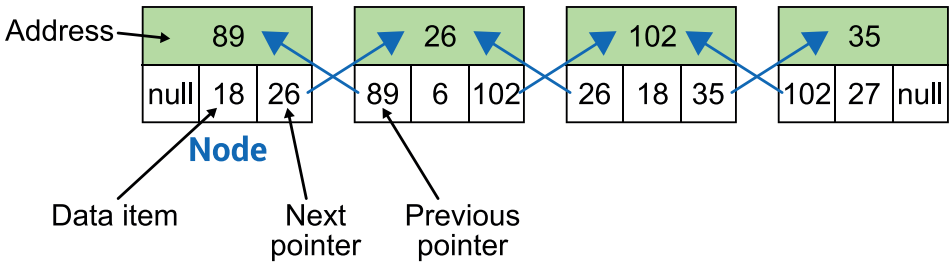
This data structure has a **next pointer**, which points to the next node *and* a **previous pointer**, which points to the previous node. This makes traversing the data structure relatively easy.

Top tip

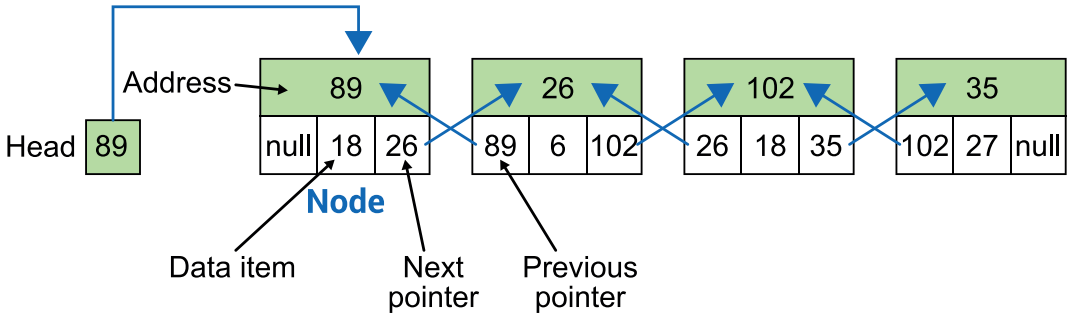
When a null value is encountered in the next pointer, the node will be the **last** in the list.

When a null value is encountered in the previous pointer, the node will be the **first** in the list (the head).

This data structure uses additional storage to overcome the difficulties of single linked lists.



The **head** variable is still used and it points to the address of the first node.



Traversing a double linked list

A double linked list can be traversed in either direction by following either the next pointers or the previous pointers. The traversal of the list ends when a null value is found in either pointer.

When the `node.next` value is null, then we are at the end of the list (the tail). When the `node.previous` value is null, then we are at the start of the list (the head).

Searching a double linked list

A search of a double linked list is a traversal with the addition of checking the data value of each node with the value or values being searched for.

A search of the list may be used to find a specific node as a point for inserting or deleting a node.

Insertion in a double linked list

Linked lists are used in programs because it is relatively easy to insert and delete items from the list by amending the pointers. The process for inserting in a double linked list is similar to a single linked

list but includes handling the previous pointers. Double linked lists still have a special variable, the head, which points to the address of the start of the list.

Insertion at the beginning of a double linked list

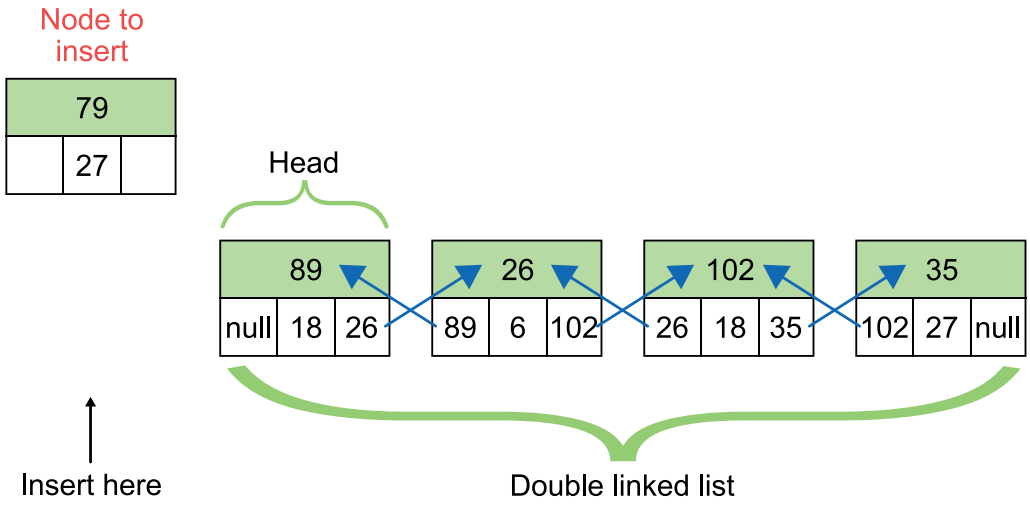
Inserting a new node at the start of a double linked list requires four operations:

1. Set the previous pointer of the new node to null.
2. Set the next pointer of new node to the value for head.
3. Set the previous pointer for the current head node to the address of the new node.
4. Set the head variable to the address of the node being inserted.

Example : Insertion at the beginning of a double linked list

In this example, the node at address 79 (with the value of 27) is to be inserted at the start of the linked list before the node with address 89.

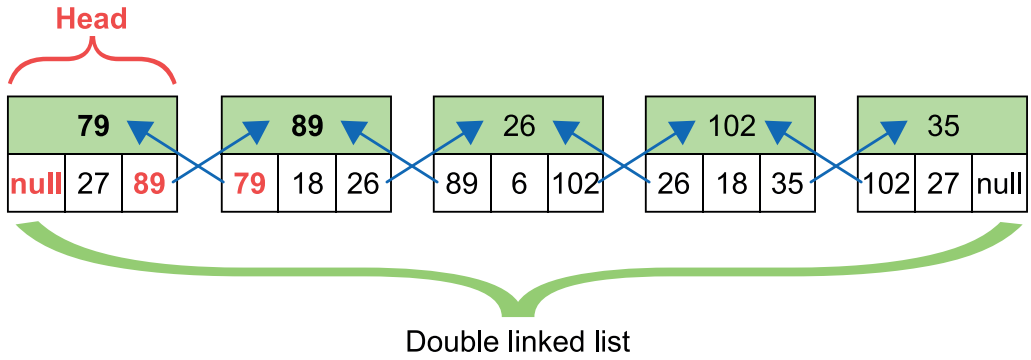
Original



Then the node can be inserted as follows:

1. Set the previous pointer for the new node 79 to null.
2. Set the next pointer for the new node 79 to 89.
3. Set the previous pointer for the current head node 89 to the address of the new node 79.
4. Set the head variable to 79 which is the address of the node being inserted.

Updated



Insertion at the end of a double linked list

Inserting a node at the end of a double linked list again requires that we traverse to the last node, the tail.

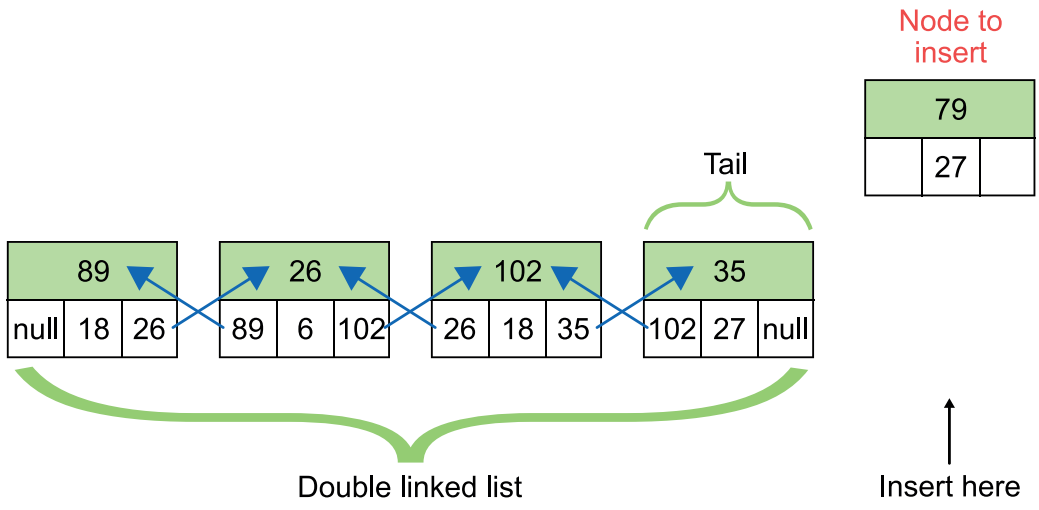
The new node is then inserted at the end of the list as follows:

1. Set the next pointer of the new node to null.
2. Set the previous pointer of the new node to the address of the tail.
3. Set the next pointer of the tail to the address of the new node.

Example : Insertion at the end of a double linked list

In this example, the node at address 79 (with the value of 27) is to be inserted at the end of the linked list after the node with address 35.

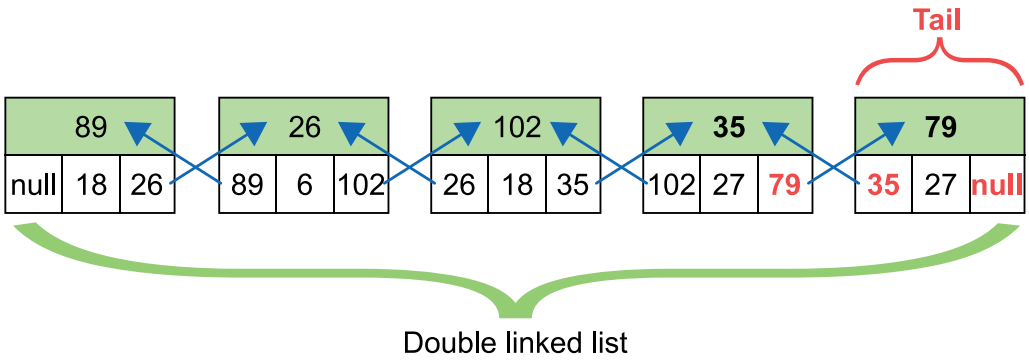
Original



Then the node can be inserted as follows:

1. Set the next pointer of the new node 79 to null.
2. Set the previous pointer of the new node 79 to the address of the current tail 35.
3. Set the next pointer of the current tail node 35 to the address of the new node 79.

Updated



Insertion after a specified node in a double linked list

To insert a new node after a specified node in a double linked list we firstly need to traverse the list until the specified node is found.

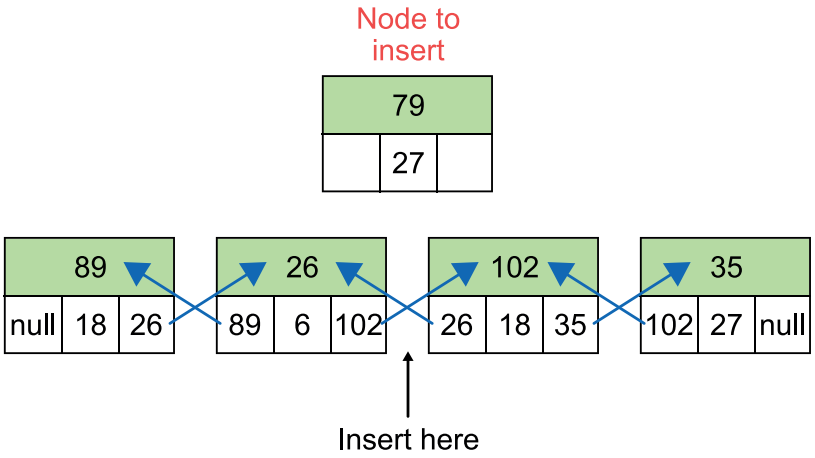
Then the node can be inserted as follows:

1. Set the next pointer of the new node to the value of `specified.next`.
2. Set the previous pointer of the new node to the value of `following.previous`.
3. Set the next pointer of the specified node to the address of the new node.
4. Set the previous pointer of the following node to the address of the new node.

Example : Insertion after a specified node in a double linked list

In this example, the node at address 79 (with the value of 27) is to be inserted after the node with address 26.

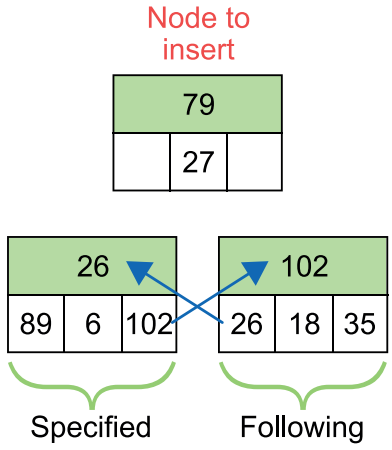
Original



Then the node can be inserted as follows:

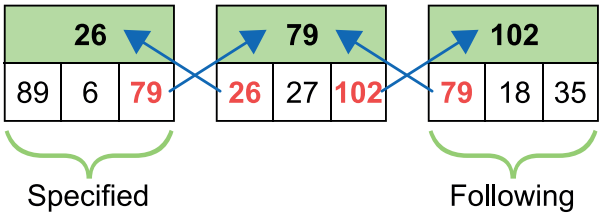
1. Set the next pointer of the new node to the value of `specified.next` (102).
2. Set the previous pointer of the new node to the value of `following.previous` (26).
3. Set the next pointer of the specified node to the address of the new node (79).
4. Set the previous pointer of the following node to the address of the new node (79).

So this:



Becomes this:

Updated



Deletion in a double linked list

Deletion from a double linked list is similar to that of a single linked list but requires additional pointers to be updated when a node is deleted.

Deleting from the beginning of a double linked list

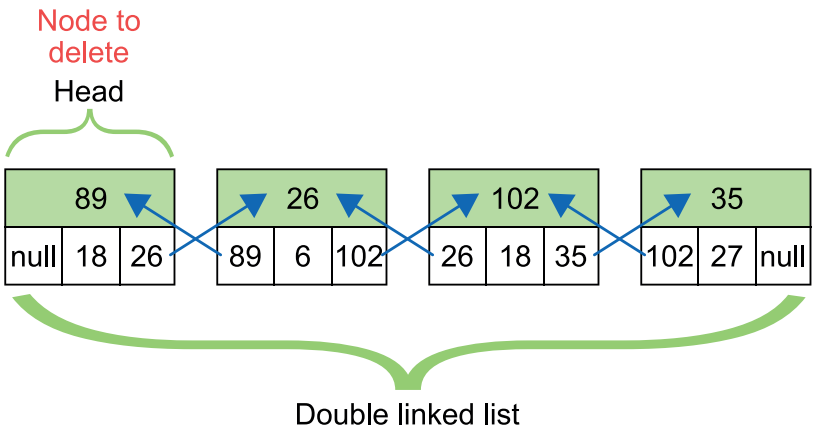
Deleting from the head of the list is done by updating the head variable for the list and setting the new head previous pointer to null.

Example : Deleting from the beginning of a double linked list

In this example, the node at address 89 (with the value of 18) is to be deleted at the start of the linked list.

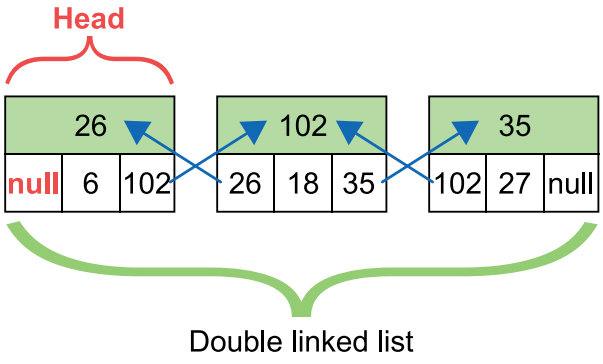
So this:

Original



Becomes this:

Updated



Deleting from the end of a double linked list

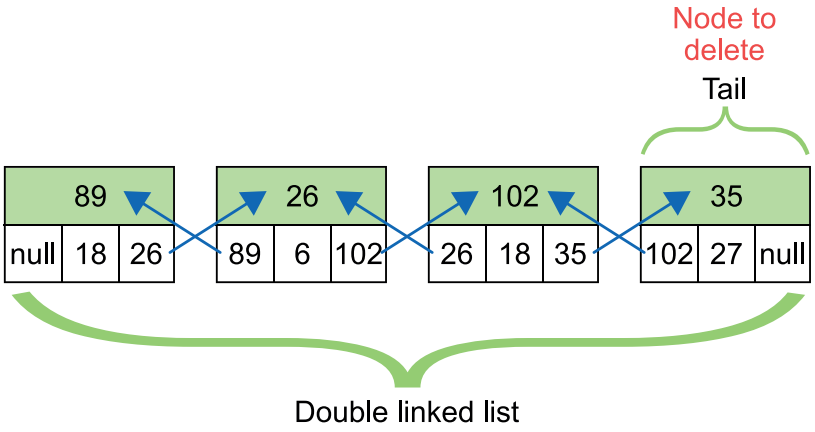
Deleting a node from the end of a double linked list requires setting the next pointer of the second last node to null.

Example : Deleting from the end of a double linked list

In this example, the node at address 35 (with the value of 27) is to be deleted at the end of the linked list.

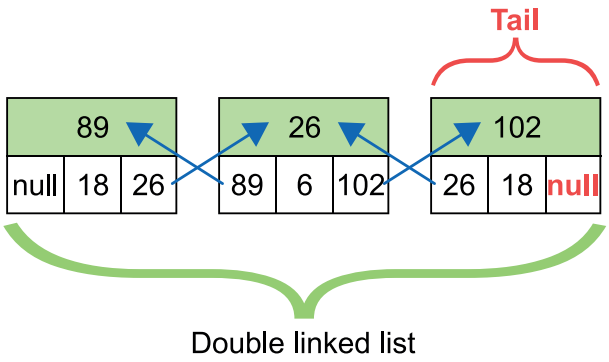
So this:

Original



Becomes this:

Updated



Deleting after a specified node in a double linked list

To delete a node from after a specified node in a double linked list, we need to update the pointers for both the specified node and the node following the node that we want to delete.

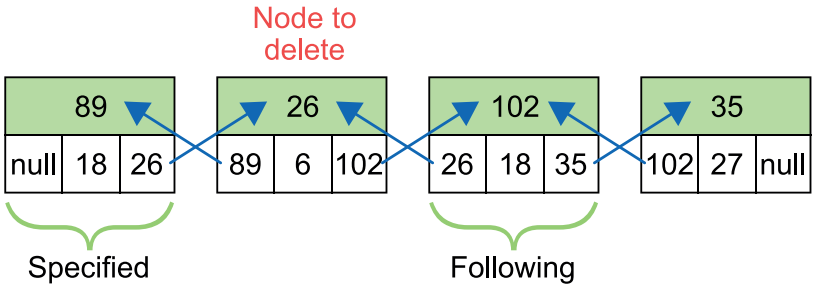
The sequence to delete the node after the one specified is:

1. Set the specified next pointer to the value of the deleted node next pointer.
2. Set the following previous pointer to the value of the deleted previous pointer.

Example : Deleting after a specified node in a double linked list

In this example the node to be deleted has an address of 26 (and data value of 6).

Original

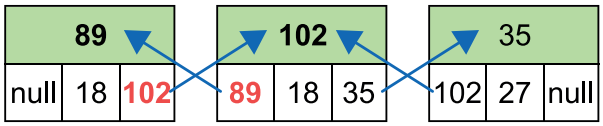


To delete the node after the one specified:

1. Set the specified next pointer to the value of the deleted node next pointer (102).
2. Set the following previous pointer to the value of the deleted previous pointer (89).

So, the nodes then become:

Updated



Linked lists Go online

Q14: A linked list is a dynamic data structure and an array is a static data structure. This is because:

a) the data within a linked list can be updated however the data within an array is fixed after it has been declared.
b) the name of a linked list can be changed but the name of an array is fixed.
c) a linked list can increase and decrease in size as nodes are added or removed but an array is declared with a fixed size.
d) a linked list can hold more than one value in each node but an array can only hold one value in each element.

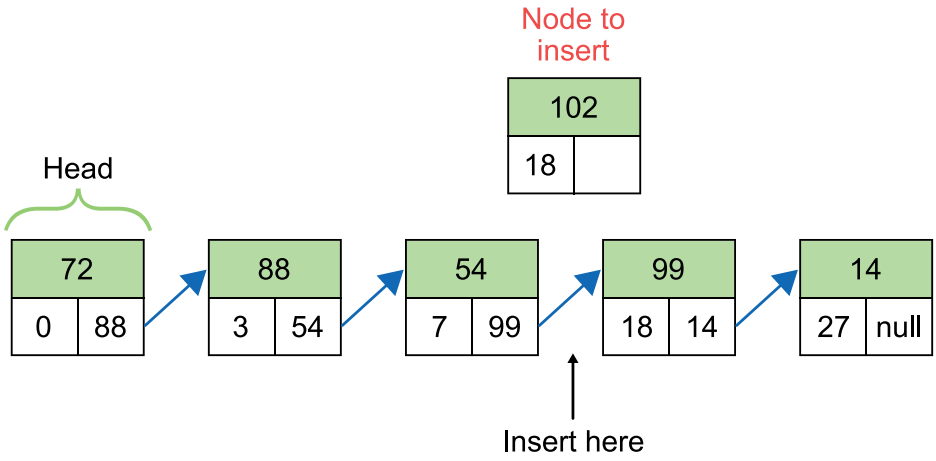
. . . . . .

Q15: What does the ***head*** variable do in a linked list?

a) Points to the start of the list (the first node).
b) Points to the end of the list (the last node).
c) Records the amount of locations used to stored nodes.
d) Holds the total of all the nodes.

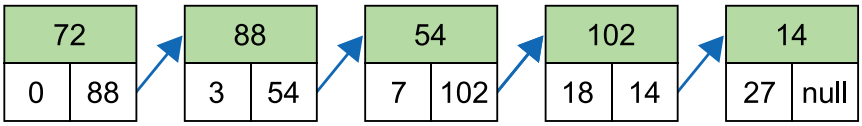
. . . . . .

Q16: A single linked list is shown below.

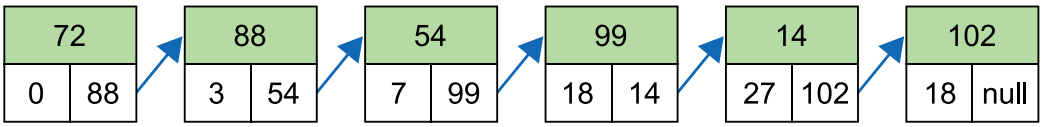


Which diagram shows the correct state of the list after the node shown is inserted at the point shown?

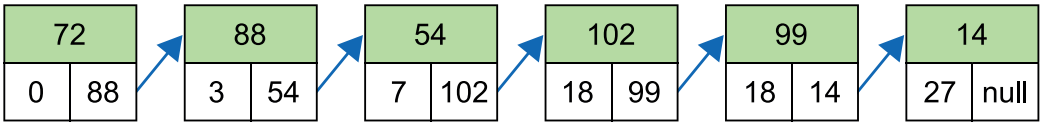
a)



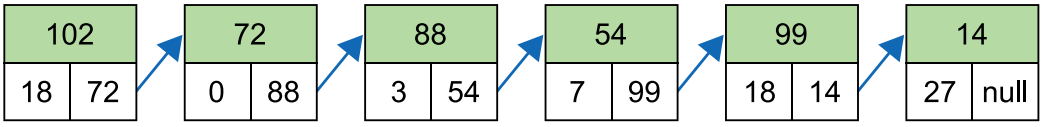
b)



c)



d)



3.5 Database connectivity

Learning objective

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

- understand and implement database connectivity using PHP and SQL;
- open and close connection to a database server;
- execute an SQL query;
- format query results.

In Advanced Higher Computing Science, the means of connecting to a database server to read, write and delete data through the use of structured query language is set out using **PHP** and **MySQL**.

The activities in this topic require a web and database server with PHP and MySQL. This is required for the Web design and development content and the following assumes that you are familiar with using PHP and SQL.

3.5.1 Open and close connection to a database server

To connect to a database server four pieces of information are required:

- server location (a domain name or an IP address);
- username required to authenticate with the server;
- password required to authenticate with the server;
- name of the database to be used.

To set up a connection to a database server, these four values are usually assigned in a PHP script as follows:

```
1 <?php
2 $servername = "localhost";
3 $username = "root";
4 $password = "serverpasssword";
5 $dbname = "webusers";
6 ...
7 ?>
```

`$servername` is the domain name or IP address at which the database server can be found. Here, the address is "localhost" because the database server is running on the same computer as the web server which is interpreting the script.

The username and password and the target database to use are defined using `$username`, `$password` and `$dbname`.

The following PHP function call establishes the connection and selects the database required (or returns an error):

```
$mysqli = new mysqli($servername, $username, $password, $dbname);
```

PHP functions usually return (output) a value when they are called and this can be stored for use later.

For example, the `mysqli` function shown above returns an object that includes details of the connection that has been established. We need to store this object so that we can make use of it later.

Since the MySQL server is a completely separate piece of software, we must consider the possibility that the server is unavailable, or inaccessible due to a network failure, or because the username/password combination you provided is not accepted by the server. In such cases, the `mysqli` function returns an object containing details of the error.

```
if ($mysqli->connect_errno) {
  //if the connect_errno value is set then show the error.
  echo "Failed to connect to MySQL: (" .
    $mysqli->connect_errno . ") ";
  die;
} else {
  echo "Connected to database";  //is okay, we connected
}
```

Important points to note about this code are:

- `$mysqli` is an object created from the constructor method of the class `mysqli`.
- `connect_errno` is an instance variable set in this object by the class constructor method when a connection fails. The `->` is used in PHP to indicate we are referring to a value or function within the object.
- The `die` function is our first example of a function that does not use parameters. All this function does is cause PHP to stop reading the source file at this point. This is a good response to a failed database connection, because in most cases the page will be unable to display any useful information without the database connection.

Note: The username and password for the MySQL server will depend on how the software was installed. You may need to create a new user instead of the default root user.

Once the processing of the data in the database is completed the connection can be closed with:

```
$mysqli -> close();
```

Alternative Method

There is a procedural method which can be used as an alternative to the class based `mysqli` means of connecting to the database and later, executing queries.

A script can initiate a connection to the database using the `mysqli_connect()` command as follows:

```
1 $conn = mysqli_connect($servername, $username, $password, $dbname);
```

The connection to the database can be tested to check that it is working with the PHP code:

```
1 if (mysqli_connect_errno()) {
2   //if the connect_errno value is set then show the error.
3   echo "Failed to connect to MySQL: " . mysqli_connect_error();
4   die;
5 } else {
6   echo "Connected to database";  //is okay, we connected
7 }
```

When the connection to the server and database is no longer required it should be closed. This is done with the PHP command:

```
1 mysqli_close($conn);
```

`$conn` is the variable used to store the database connection.

Open a connection to a database server (30 min) Go online

You are going to create a *server side include*. This is a piece of code, a module or small library of code, that you can import into your main script. In this case the *include* will establish the database connection for us.

Before you begin download the esports.zip file here: /vle/asset/Downloads/AHCCMP/Course%20Downloads/29705BC3-A560-E581-BA08-9A2C242402F2/esports.zip

If you don't have online access you can create the file called ***esports.sql*** in a text editor from the following text:

```
1  CREATE DATABASE 'esports';
2  USE 'esports';
3
4  CREATE TABLE IF NOT EXISTS 'game' (
5    'gameid' int(11) NOT NULL AUTO_INCREMENT,
6    'game' varchar(11) NOT NULL,
7    'platform' varchar(9) DEFAULT NULL,
8    PRIMARY KEY ('gameid')
9  ) ENGINE=InnoDB AUTO_INCREMENT=4 DEFAULT CHARSET=utf8;
10
```

```
11 INSERT INTO 'game' ('gameid', 'game', 'platform') VALUES
12 (1, 'Massive RPG', 'X-station'),
13 (2, 'SuperJoe', 'S-box'),
14 (3, 'Terra 1999', 'PC');
15
16 CREATE TABLE IF NOT EXISTS 'gamescore' (
17   'scoreid' int(11) NOT NULL AUTO_INCREMENT,
18   'playername' varchar(20) DEFAULT NULL,
19   'gameid' int(11) DEFAULT NULL,
20   'score' int(10) DEFAULT NULL,
21   PRIMARY KEY ('scoreid')
22 ) AUTO_INCREMENT=632;
23
24 INSERT INTO 'gamescore' ('scoreid', 'playername', 'gameid',
       'score') VALUES
25 (625, 'shocker', 1, 726122),
26 (626, 'peach', 2, 102928),
27 (627, 'peach', 1, 625100),
28 (628, 'shocker', 3, 821200),
29 (629, 'destroyer', 3, 120001),
30 (630, 'peach', 1, 283102),
31 (631, 'destroyer', 2, 299000);
32
33 CREATE TABLE IF NOT EXISTS 'player' (
34   'player' varchar(9) NOT NULL,
35   'realname' varchar(15) DEFAULT NULL,
36   'password' varchar(9) DEFAULT NULL,
37   'email' varchar(16) DEFAULT NULL,
38   'tagphase' varchar(28) DEFAULT NULL,
39   'status' varchar(2) DEFAULT NULL,
40   PRIMARY KEY ('player')
41 )
42
43 INSERT INTO 'player' ('player', 'realname', 'password', 'email',
       'tagphase', 'status') VALUES
44 ('destroyer', 'Chloe Davidson', 'shadow99', 'chloe@coders.org', 'I
       love games', 'on'),
45 ('peach', 'Sally MacDonald', 'trustme1', 'sally@scott.com', 'I'm
       part of an eSports team', 'on'),
46 ('shocker', 'Paul White', 'pink10red', 'paul@gamers.org', 'I want
       to play competitively', 'on');
```

Use the ***esports.sql*** file in phpMyAdmin (or similar) to create and populate a database called ***esports*** on your database server.

Start a new document in a text editor. Enter the following text (Note: **only** this text should appear in the file).

```
<?php
//connect to the database using mysqli API
$mysqli = new mysqli($host, $username, $pword, $database);
if ($mysqli->connect_errno) {
  echo "Failed to connect to MySQL: (" .
  $mysqli->connect_errno . ") ";
  die;
}
?>
```

Save this file in the root of your web server as ***dbconnection.php***.

Start a new PHP Script and enter the following code:

```
<?php
  //set up the connection variables for the dbconnection.php
     include
  $host="localhost"; //use the domain/IP for your database server
  $username="root"; //use the username for your database server
  $pword="root"; //use the password for your database server
  $database="esports";

  require("dbconnection.php"); //require the include code
?>
<html>
  <head>
    <title>Database Connect</title>
  </head>
  <body>
  <?php
  //Just to check, let's get the name of the current database
  //return name of current default database

  if ($result = $mysqli->query("SELECT DATABASE()")) {
        //run the query method to get the database result object
        $row = $result->fetch_row();
        //read this from the query results object as an array of
           data
        $default_database = $row[0];
        //set the value to the first element [0] in the array
        $result->close();   //free the $result set (clear it)
  }
  ?>
  You are connected to the MySQL server and the
     <i><?=$default_database?></i> database.
  </body>
</html>
```

Save this file as ***database1.php***. View the PHP file using your browser.

The text "You are connected to the MySQL server and the esports database." should be displayed.

3.5.2 Execute an SQL query

In PHP, database queries are typically written to a variable and then this variable is used with the `mysqli->query` command to execute the query.

All SQL queries are executed using the `mysqli->query()` method of the `mysqli` class. This executes the query and returns the result set as an object for SELECT queries. For other queries it returns a true or false result depending on the success of the query.

Examples

1. Return the number of rows that match a query

The following code will display the number of rows from the game table.

```
1 /* Select queries return a resultset */
2 $querystring = "SELECT * FROM game";
3 if ($result = $mysqli->query($querystring))
4     printf("Select returned %d rows.\n", $result->num_rows);
5
6     /* free result set */
7     $result->close();
8 }
```

.....

2. Create a table and receive some feedback

The following code will create a table called `tmplogs`. If this is successful it will display the message "Table tmplogs successfully created."

```
1 /* Create table doesn't return a resultset */
2 if ($mysqli->query("CREATE TABLE tmplogs (
3         id INT PRIMARY KEY,
4         username varchar(30),
5         time timestamp ) ") === TRUE)
6 {
7     printf("Table tmplogs successfully created.\n");
8 }
```

MySQL also keeps track of the number of rows affected by `INSERT`, `DELETE` and `UPDATE` queries and the number of rows in the results of a `SELECT` query. This number can be accessed using the `mysqli->affected_rows` method.

So a script to run a query and trap any errors will be:

```
1 <?php
2
3 If ( $mysqli->query($query_string) ) {
4   echo ("Query successful with " . $mysqli->affected_rows . " rows
      changed/accessed.");
5 } else {
6   echo ("Error performing query: " . $mysqli->error() );
7 }
8
9 ?>
```

Alternative method

The procedural method will execute a query using the connection to the database and the command `mysqli_query` as follows:

Examples

1. Alternative method: return the number of rows that match a query

The following code will display the number of rows from the game table.

```
1 /* Select queries return a resultset */
2 $querystring = "SELECT * FROM game";
3 if ($result = mysqli_query($conn, $querystring)) {
4     printf("Select returned %d rows.\n", mysqli_num_rows($result));
5
6     /* free result set */
7     mysqli_close($conn);
8 }
```

.....

2. Alternative method: create a table and receive some feedback

The following code will create a table called `tmplogs`. If this is successful it will display the message "Table tmplogs successfully created."

```
1  /* Create table doesn't return a resultset */
2  $querystring = "CREATE TABLE tmplogs (
3          id INT PRIMARY KEY,
4          username varchar(30),
5          time timestamp ) ";
6
7  if ($result=mysqli_query($conn,$querystring) )
8  {
9      printf("Table tmplogs successfully created.\n");
10 }
```

MySQL also keeps track of the number of rows affected by `INSERT`, `DELETE` and `UPDATE` queries and the number of rows in the results of a `SELECT` query. This number can be accessed using `mysqli_affected_rows` with the connection variable (in this case `$conn`).

So using the alternative procedural method a script to run a query and trap any errors will be:

```
<?php

If ( $mysqli->query($query_string) ) {
  echo ("Query successful with " . mysqli_affected_rows($conn) . " rows
     changed/accessed.");
} else {
  echo ("Error performing query: " . $mysqli->error() );
}

?>
```

3.5.3 Format query results

The results from a query will be held in the result set for the query. This can be read and the data formatted using CSS and presented using HTML.

```
SELECT game, platform, playername, score from game, gamescore
WHERE game.gameid = gamescore.gameid
```

This query when executed, produces the following answer table:

game	**platform**	**playername**	**score**
Massive RPG	X-station	shocker	726122
SuperJoe	S-box	peach	102928
Massive RPG	X-station	peach	625100
Terra 1999	PC	shocker	821200
Terra 1999	PC	destroyer	120001
Massive RPG	X-station	peach	283102
SuperJoe	S-box	destroyer	299000

This code would execute the query and hold the results:

```
$query_string = "SELECT game, platform, playername, score from game,
   gamescore WHERE game.gameid = gamescore.gameid";

//run the query
if ($result = $mysqli->query($query_string)) {
```

The query is executed within the `if` condition, so if the query fails we can recover from it without crashing. If the query is successful, the contents of the query answer table will be held in the object called `$result`.

To read the rows and display the data from each row, we need to read a row at a time and display its values.

The code:

```
1 while ($row = $result->fetch_object()) {
```

retrieves each row from the `$result` (an object that holds the result set) and repeats the process until there are no rows left in the result set object.

In order to read the instance variables from each row object (these correspond to the column names) we use the code:

```
1 echo  $row->game . " ". $row->platform . " ". $row->playername . " ".
    $row->score);
```

The code `$row->game` returns the game column value for the current row and the `$row->platform` code returns the platform column value and so on.

So, by printing these values to the screen, the resulting webpage output would be:

Massive RPG X-station shocker 726122
SuperJoe S-box peach 102928
Massive RPG X-station peach 625100
Terra 1999 PC shocker 821200
Terra 1999 PC destroyer 120001
Massive RPG X-station peach 283102
SuperJoe S-box destroyer 299000

These results would be better presented in a table. To do this we can use the HTML table tags (table, th, tr, td).

The code to output the results would be:

```
<?php

$query_string = "SELECT game, platform, playername, score from game,
   gamescore WHERE game.gameid = gamescore.gameid";

//run the query
if ($result = $mysqli->query($query_string)) {

//start the table
?>
<table>
<tr>
<th>Game</th><th>Platform</th><th>Player Name</th><th>Score</th>
</tr>
<?php
while ($row = $result->fetch_object()) {

  echo "<tr><td>" . $row->game . "</td>";
  echo "<td>" . $row->platform . "</td>";
  echo "<td>" . $row->playername . "</td>";
  echo "<td>" . $row->score . "</td></tr>";
}
?>
</table>
<?php
$mysqli->close();
?>
```

This will retrieve each row from the result set and display each one within a row in an HTML table.

Alternative Method

The code to execute the query and display the results in an HTML table using the procedural code for `mysqli` is:

```
<?php

$query_string = "SELECT game, platform, playername, score from game,
    gamescore WHERE game.gameid = gamescore.gameid";

//run the query
if ($result = $mysqli_query($conn, $query_string)) {

//start the table
?>
<table>
<tr>
<th>Game</th><th>Platform</th><th>Player Name</th><th>Score</th>
</tr>
<?php
while ($row = $result_fetch_assoc($result) {

  echo "<tr><td>" . $row['game'] . "</td>";
  echo "<td>" . $row['platform'] . "</td>";
  echo "<td>" . $row['playername'] . "</td>";
  echo "<td>" . $row['score'] . "</td></tr>";
}
?>
</table>
<?php
$mysqli_close($conn);
?>
```

Execute and display results (30 min) Go online

Q17: Go back to the *server side include* you created to connect to the database in the ***Open a connection to a database*** activity.

Using what you have learned from the **Execute an SQL query** and **Format query results** sections, add code to execute and display the results. Use either the object oriented style or the procedural style.

If you need a reminder of the starting point for this activity you can download the PHP file as a zip file here: /vle/asset/Downloads/AHCCMP/Course%20Downloads/39B255FB-25F7-7DDC-7882-3300D751BE21/executeanddisplayresults_start.zip

3.6 Algorithm specification

Learning objective

By the end of this section you should be able to describe, explain and implement three standard algorithms:

- bubble sort;
- insertion sort;
- binary search.

For Advanced Higher Computing Science you are required to be able to describe, explain and implement three standard algorithms. These are:

1. bubble sort;
2. insertion sort;
3. binary search.

Each of these algorithms is commonly used in programs and understanding these is essential for the examination.

3.6.1 Bubble sort

A bubble sort takes the items in the list/array and swaps them until the they are in the correct position. Each pass through the list places one number in the correct position and the length of the list to be sorted is reduced by one.

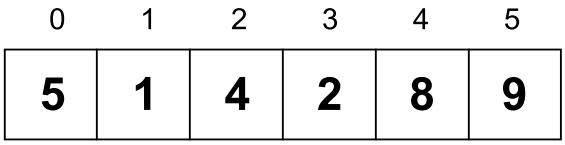
The bubble sort algorithm is:

```
1  PROCEDURE bubble_sort(list)
2    DECLARE n INITIALLY length(list)
3    DECLARE swapped INITIALLY TRUE
4    WHILE swapped AND n >= 0
5      SET swapped TO False
6      FOR i = 0 to n-2 DO
7        IF list[i] > list[i+1] THEN
8          SET temp TO list[i]
9          SET list[i] TO list[i+1]
10         SET list[i+1] TO temp
11         SET swapped TO TRUE
12       END IF
13     END FOR
14     SET n TO n - 1
15   END WHILE
16 END PROCEDURE
```

Example : Bubble sort

Let's look at a bubble sort example where the list of values to be sorted is:

Figure 3.7: Bubble sort: list of values to be sorted



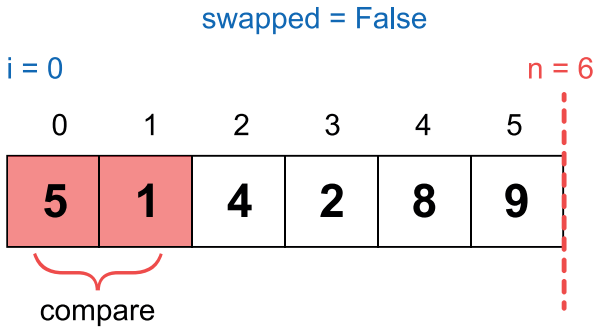
The initial values of the variables are set as follows:

- `n` is set to `length(list)` which is 6.
- `swapped` is set to TRUE.
- `i` is set to 0.

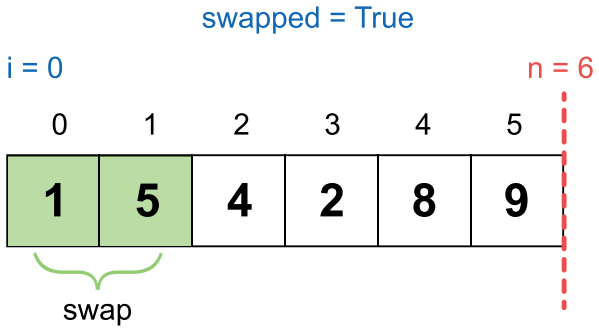
First pass:

The first pass, step by step of the sort would be:

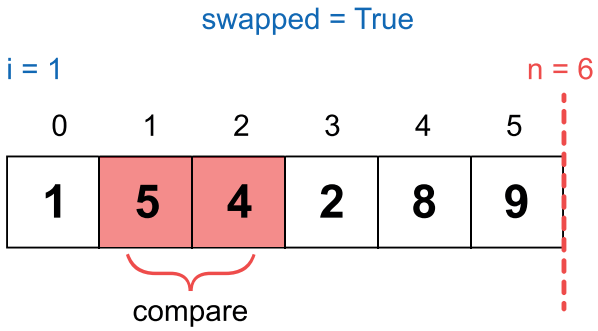
1. `swapped` is set to FALSE. Compare the items at index 0 and index 1.



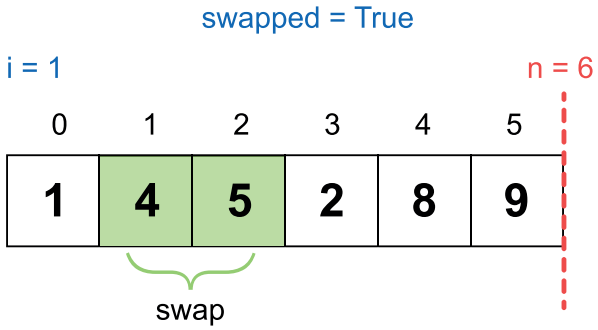
Index 0 value of 5 is greater than index 1 value of 1, so swap the items and record that a swap has occurred by setting `swapped` to TRUE.



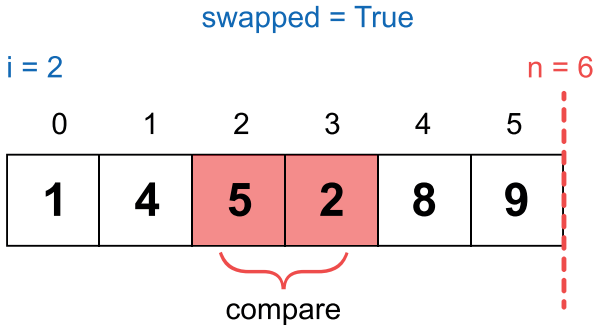
2. Compare the items at index 1 and index 2.



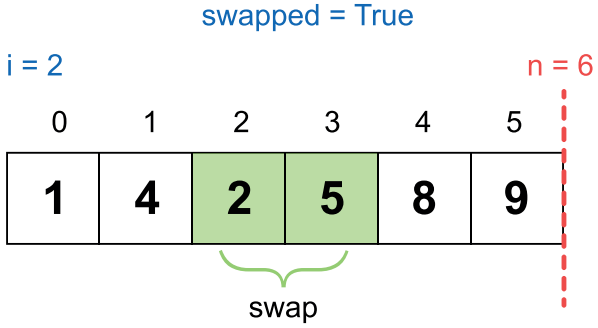
Because index 1 value of 5 is greater than index 2 value of 4, we swap the items, and record that a swap has occurred.



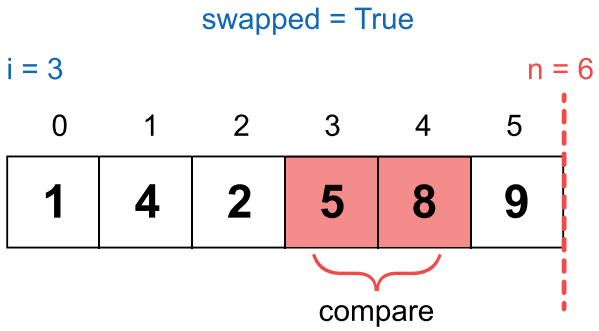
3. Compare the items at index 2 and index 3.



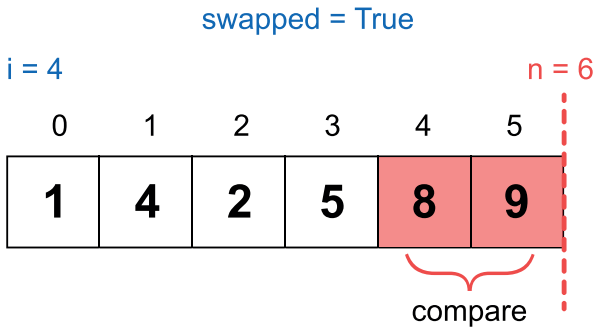
Because index 2 value of 5 is greater than index 3 value of 2, we swap the items, and record that a swap has occurred.



4. Compare the items at index 3 and index 4. Because index 3 value of 5 is not greater than index 4 value of 8 there is no swap to do.



5. Compare the items at index 4 and index 5. Because index 4 value of 8 is not greater than index 5 value of 9 there is no swap to do.



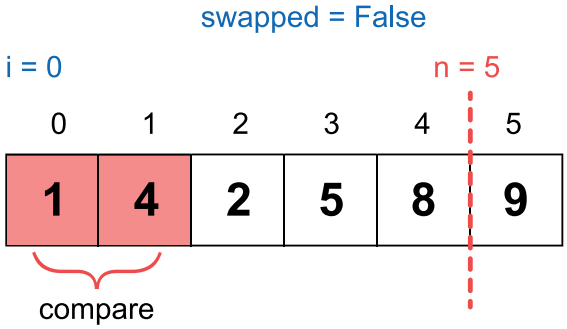
At the end of the first pass through, the item at index 5 is correctly sorted.

`n` is reduced by 1 so that this item will be ignored in the next pass because it is already in the correct position.

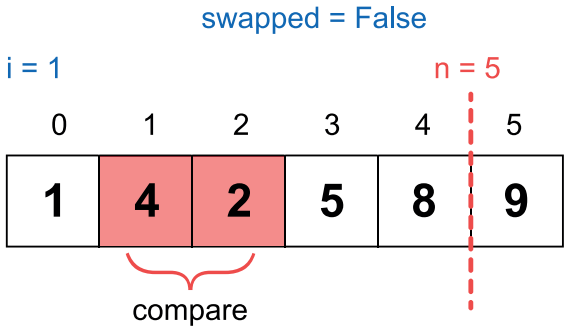
Second pass:

The second pass starts at the beginning of the list again.

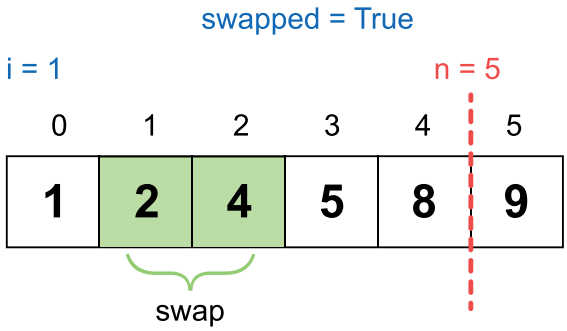
1. `swapped` is set to FALSE. Compare the items at index 0 and index 1. Because index 0 value of 1 is not greater than index 1 value of 4, there is no swap to do.



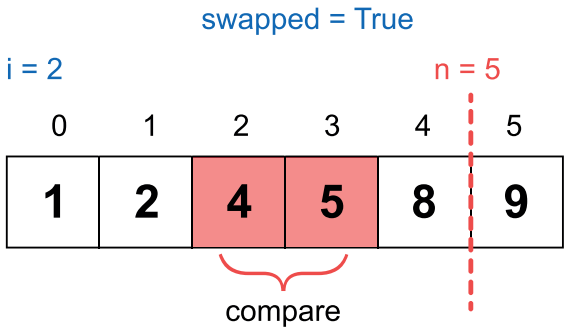
2. Compare the items at index 1 and index 2.



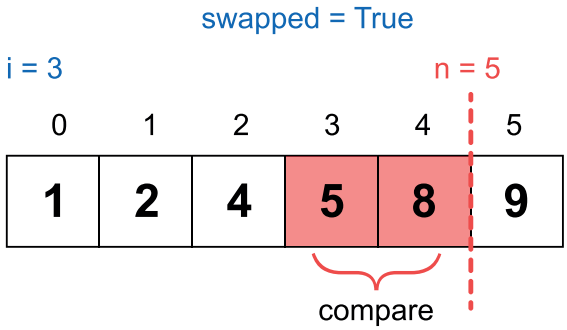
Because index 1 value of 4 is greater than index 2 value of 2, we swap the items, and record that a swap has occurred.



3. Compare the items at index 2 and index 3. Because index 2 value of 4 is not greater than index 3 value of 5, there is no swap to do.



4. Compare the items at index 3 and index 4. Because index 3 value of 5 is not greater than index 4 value of 8, there is no swap to do.



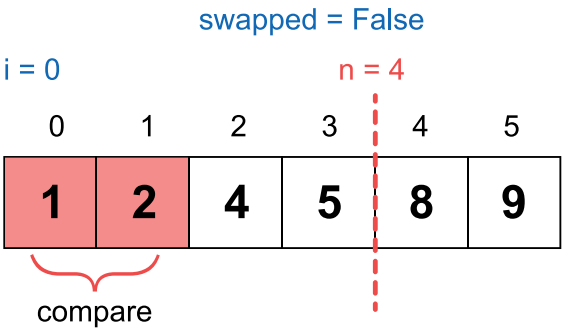
At the end of the second pass the list is sorted but in order to exit the WHILE loop, the algorithm needs to go through a whole pass without any swaps.

`n` is reduced by 1.

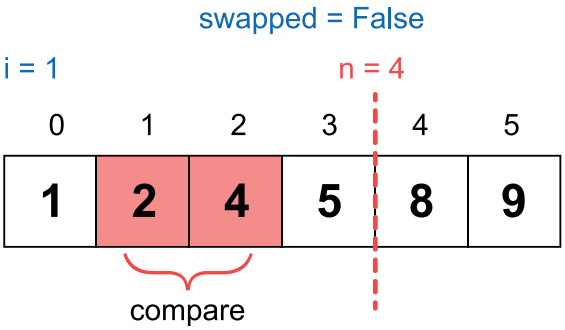
Final pass without any swaps:

The final pass starts at the beginning of the list again.

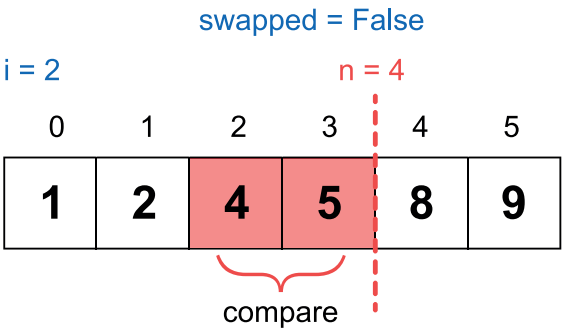
1. `swapped` is set to FALSE. Compare the items at index 0 and index 1. No swap to do.



2. Compare the items at index 1 and index 2. No swap to do.



3. Compare the items at index 2 and index 3. No swap to do.



At the end of the third pass, there have been no swaps so we know that the list is now correctly sorted and can exit the algorithm.

Bubble sort (50 min) Go online

Q18: Implement the bubble sort for a list of 6 numbers. Test it initially using the values given in the previous example.

Here is a reminder of the values we used in the example: Figure 3.7

. . . . . .

Q19: Implement the bubble sort for a list/array of 20 randomly generated numbers.

Your answer should include the following steps:

a) Write code to randomly generate the 20 numbers in the list/array.
b) Then carry out the bubble sort.

3.6.2 Insertion sort

The insertion sort is similar to how we would sort a small set of items such as a hand of cards. 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.

The insertion sort algorithm starts at one end of the array, dividing it into two sections, a sorted one and an unsorted one. The first item in the array can be treated as a sorted array containing only one item, so it starts comparisons with the second item in the array. The algorithm then progressively moves each subsequent item into the sorted section until it reaches the last item.

```
1  PROCEDURE insertion_sort(list)
2    DECLARE value INITIALLY 0
3    DECLARE index INITIALLY 0
4    FOR i = 1 to length(list)-1 DO
5      SET value TO list[i]
6      SET index TO i
7      WHILE (index > 0) AND (value < list[index-1]) DO
8        SET list[index] TO list[index-1]
9        SET index TO index - 1
10     END WHILE
11     SET list[index] TO value
12   END FOR
13 END PROCEDURE
```

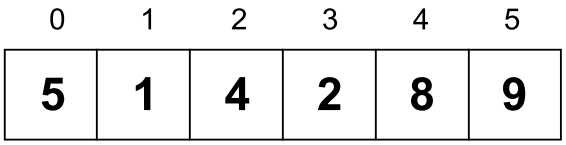
In the insertion sort, the list is split into an unsorted part and a sorted part. Each pass through the list, adds one item to the sorted part.

The algorithm uses a fixed loop to run through the array, starting from the item at position 1, keeping track of the index position of the current item being sorted and its value. A conditional loop then checks to see if the item in the previous index position is greater than the current one and if so moves it into that place. Before the fixed loop completes, the current item is placed into the gap left by the move.

Example : Insertion sort

Let's look at an insertion sort example using the same list of values to be sorted as follows:

Figure 3.8: Insertion sort: list of values to be sorted



The initial values of the variables are set as follows:

- `value` is set to 0.
- `index` is set to 0.

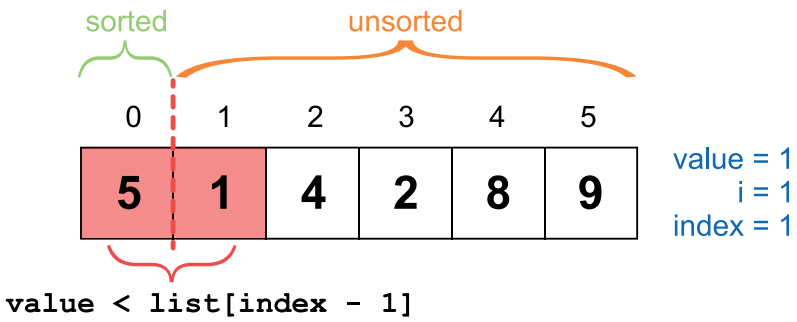
First pass:

The first pass through will look at index 1 and index 0 and place the lowest value in index 0. The pass will proceed as follows:

1. `FOR i = 1 to length(list)-1`, so `i` will start at 1 and finish at 5 (given that the list is 6 long).

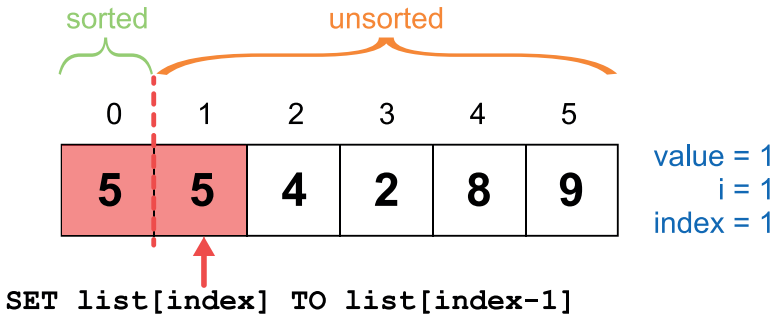
   Initially at the start of the FOR loop `SET value TO list[i]` which is `list[1]` giving us a `value` of 1.

   `SET index TO i`, which is 1.



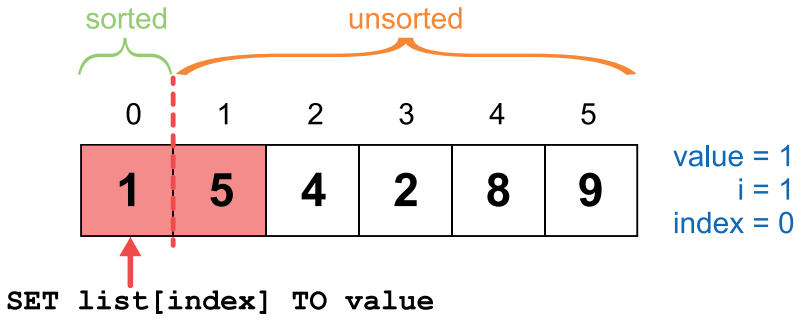
2. Both conditions of the WHILE loop are true to continue into the loop: `(index > 0) AND (value < list[index-1])` as `value` of 1 is less than index 0 value of 5.

   In the WHILE loop we `SET list[index] TO list[index - 1]`. So we write the content of index position 0 to index position 1 which is 5.



3. Then `SET index TO index - 1` so `index` will now be 0. This means one of the conditions for the WHILE loop `index > 0` is now false, so we exit the WHILE loop.

   Now `SET list[index] TO value` writes `value` into the current content of the `index`, which is index 0. So the content of index 0 becomes 1.



This completes the first sorted item in the list and the first pass through.

Second pass:

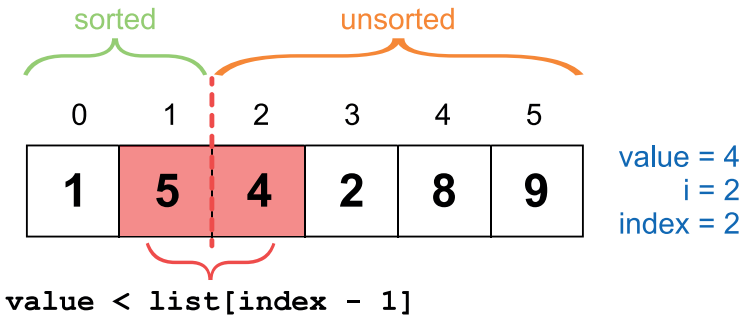
The second pass through will look at index 2 and index 1 and place the lowest value in index 1. The pass will proceed as follows:

1. We go back to the start of the FOR loop and `i` increments by 1 to become 2.

   `SET value TO list[i]` sets `value` to 4.

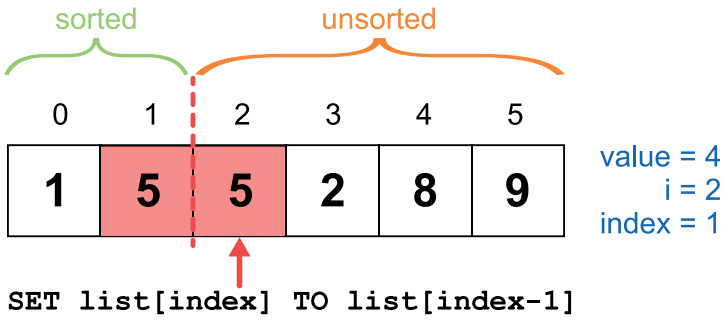
   `SET index TO i` sets `index` to 2.

   We enter the WHILE loop because both of the conditions are true: `(index > 0) AND (value < list[index-1])` as `value` of 4 is less than index 1 value of 5.



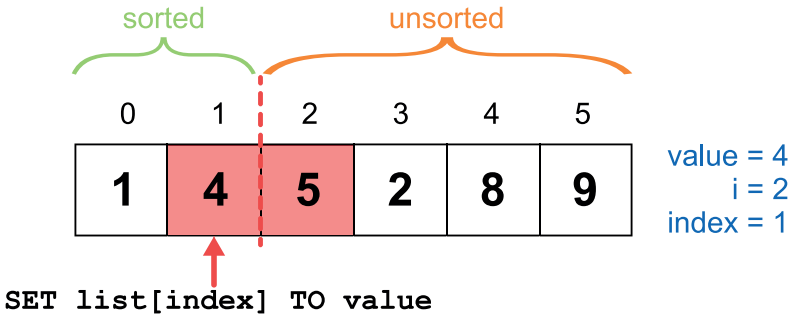
2. In the WHILE loop we `SET list[index] TO list[index - 1]`. So we write the content of index position 1 to index position 2 which is 5.

   Then we `SET index TO index - 1` so `index` will now be 1.



3. Return to the start of the WHILE loop and check if `index > 0`, which it is, and `value < list[index-1]` which it is not, because 4 is not less than 1.

   So we exit the WHILE loop and `SET list[index] to value`. So the content of index 1 becomes 4.



   This completes the second pass through and now the first two positions in the list are sorted.

Third pass:

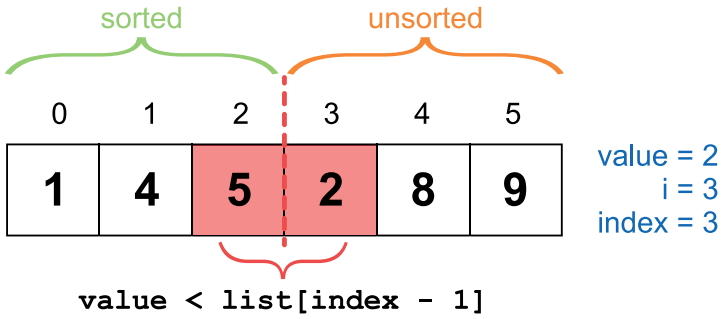
The third pass through will look at index 3 and index 2 and place the lowest value in index 2. The pass will proceed as follows:

1. We go back to the start of the FOR loop and `i` increments by 1 to become 3.

   `SET value TO list[i]` sets `value` to 2.

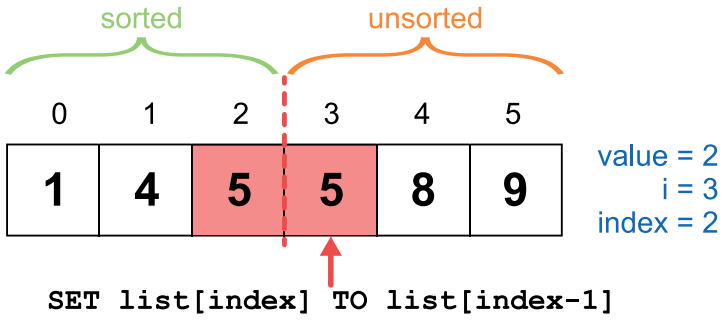
   `SET index TO i` sets `index` to 3.

   We enter the WHILE loop because both of the conditions are true: `(index > 0) AND (value < list[index-1])` as `value` of 2 is less than index 2 value of 5



2. In the WHILE loop we `SET list[index] TO list[index - 1]` so the algorithm writes the content of index position 2 to index position 3 which is 5.

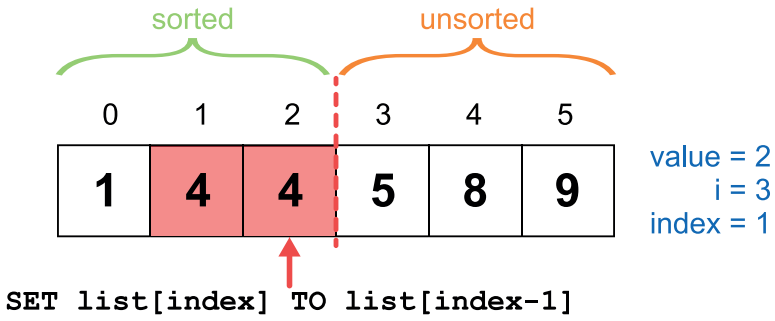
   Then we `SET index TO index - 1` so `index` will now be 2.



3. The WHILE loop continues because both of the conditions are true: `(index > 0) AND (value < list[index-1])` as `value` of 2 is less than index 1 value of 4.

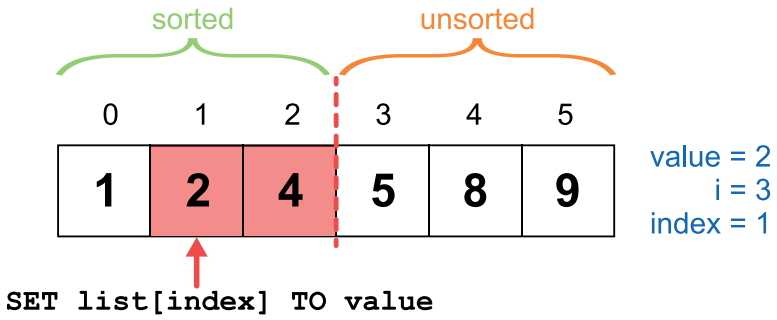
   In the WHILE loop we `SET list[index] TO list[index - 1]` so the algorithm writes the content of index position 1 to index position 2 which is 4.

   Then we `SET index TO index - 1` so `index` will now be 1.



4. Return to the start of the WHILE loop and check if `index > 0`, which it is, and `value < list[index-1]` which it is not, because 2 is not less than 1.

   So we exit the WHILE loop and `SET list[index] to value`. So the content of index 1 becomes 2.



   This completes the third pass through and now the first three positions in the list are sorted.

5. At the end of the third pass the remainder of the list is now also sorted. However the insertion sort would still loop through to the length of the list-1 (until it had processed position 5).

Insertion sort (50 min) Go online

Q20: Implement the insertion sort for a list of 6 numbers. Test it initially using the values given in the previous example.

Here is a reminder of the values we used in the example: Figure 3.8

. . . . . .

Q21: Implement the insertion sort for a list/array of 20 randomly generated numbers.

Your answer should include the following steps:

a) Write code to randomly generate the 20 numbers in the list/array.

b) Then carry out the insertion sort.

3.6.3 Binary search

From your studies at Higher you should be familiar with the linear search algorithm. This is an algorithm which searches through a list or array of items to find a specific value or values.

The binary search algorithm is much more efficient than a linear search. A linear search begins at the start of a list and works its way through to the end which is not a very efficient means of searching data.

The binary search algorithm works by looking at the middle point of the list and working out if the number we are looking for is higher or lower than the number in the middle position. This information is then used to exclude half of the list from the remaining comparisons. The size of the list keeps being halved until the item is found or we run out of list.

Key point

A requirement of the binary search algorithm is that the numbers in the list must be sorted before the algorithm can work.

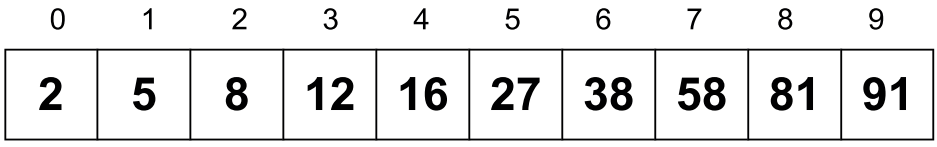
The algorithm for binary search is:

```
1  PROCEDURE binary_search(list,target)
2    DECLARE low INITIALLY 0
3    DECLARE high INITIALLY length(list)-1
4    DECLARE mid INITIALLY 0
5    DECLARE found INITIALLY FALSE
6    WHILE NOT found AND low <= high
7      SET mid TO (low+high)/2
8      IF target = list[mid] THEN
9        SEND "Found at position " & mid TO DISPLAY
10       SET found TO TRUE
11     ELSE IF target > list[mid] THEN
12       SET low TO mid+1
13     ELSE
14       SET high TO mid-1
15     END IF
16   END WHILE
17   IF found = FALSE THEN
18     SEND "Target not found" TO DISPLAY
19   END IF
20 END PROCEDURE
```

Example : Binary search

Let's look at a binary search example where the value we are looking for is 27 (the search key) in the following list of 10 elements:

Figure 3.9: Binary search: list of values to be searched



You can see that the list has already been sorted into ascending order before we start the binary search.

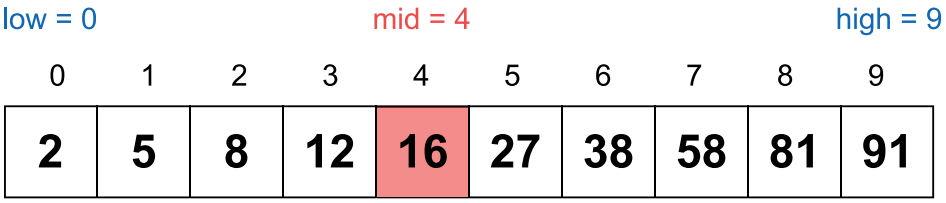
The value (search key) we are looking for is 27. This is our `target` variable.

We use three important variables during the search, these are `low`, `high` and `mid`.

- `low`: is set to the lowest index for the list, in this case 0.
- `high`: is set to the highest index for the list, in this case `length(list)-1` which is 9.

- `mid`: is the position which is in the middle of the list. `mid` will start as 4 in this case which is `(low + high)/2`. (0 + 9)/2 actually gives us 4.5 but we chop off the 0.5 decimal part as we are using integer values to hold the index.

So, the list starts with the `low`, `mid` and `high` variables set as follows:

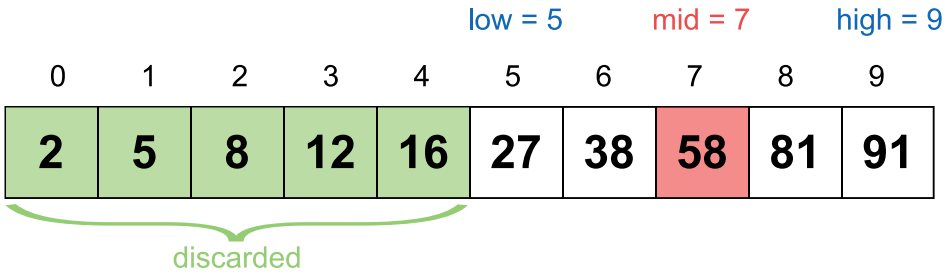


If our search key is 27, the following will happen:

1. Compare the search key of 27 with the item at position `mid`. The value of 27 that we are looking for is bigger than the value of 16 in the `mid` position so we know that everything from the `mid` position to the `low` position is not going to contain what we are looking for. We can therefore discard this part of the list from any remaining processing.

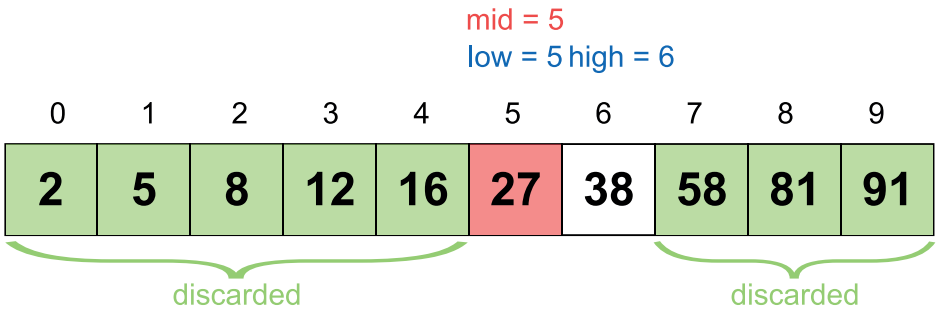
   Discarding part of the list is done by moving the `low` or the `high` position. In this case we set our new `low` to `mid + 1`, so `low` becomes 5.

2. Repeat the loop and create a new `mid` location between `low` of 5 and `high` of 9 which is (5 + 9)/2 giving us 7.



3. Compare our search key of 27 with the new `mid` value. This time the `mid` value at position 7 is 58 which is higher than the 27 we are trying to find. This means that we know that anything from the current `mid` position to the end of the list is not going to contain the required value and can also be discarded. This is done by setting `high` to `mid - 1`, so `high` becomes 6.

4. Repeat the loop again and create a new `mid` location between `low` of 5 and `high` of 6 which is (5 + 6)/2 giving us 5 (remember we are working with integers so we chop off the 0.5).



5. Compare our search key of 27 with the new `mid` value of 27. This time the value at the `mid` position is what we were searching for, so `found` is set to TRUE and we can exit the WHILE loop. A message is sent to the display to say where the value was found in the list.

If the value we were looking for was not found in the list we would eventually get to a point where `low` was greater than `high`. This would terminate the WHILE loop and if `found` was FALSE, we can write a message "Target not found" to explain that the item was not found.

As you can see, this algorithm quickly found a value in the list in relatively few comparisons. This algorithm is much more efficient than the linear search that you will have used previously.

Binary search (50 min) Go online

Q22: Implement the binary search for a list of 10 numbers. Test it initially using the values given in the previous example.

Here is a reminder of the values we used in the example: Figure 3.9

. . . . . .

Q23: Implement the binary search for a list/array of 20 randomly generated numbers.

Your answer should include the following steps:

a) Write code to randomly generate the 20 numbers in the list/array.
b) Then carry out an insertion sort to sort the numbers.
c) Display the list and allow a value to be entered to search for.
d) Use the binary search to search through the list and display the position of an element (or display item not found if it is not in the list).

3.7 Learning points

Summary

You should now know how to implement the following data structures in your chosen programming language:

- understand the features of object oriented languages:
  - classes;
  - properties;
  - methods;
  - constructors;
  - instantiation;
  - object;
  - inheritance;
  - subclasses;
  - polymorphism.
- records and arrays of records;
- parallel 1D arrays;
- 2D arrays;
- arrays of objects;
- single and double linked lists;
- database connectivity using PHP and SQL;
- algorithm specifications for bubble sort, insertion sort and binary search.

3.8 End of topic test

End of topic test: Implementation Go online

Q24: The class definition for Magazine is as follows:

```
1  CLASS Magazine INHERITS Publication WITH {INTEGER edition, INTEGER
       year}
2
3  METHODS
4
5    FUNCTION publishEdition(INTEGER edval, INTEGER yrval) RETURNS
         BOOLEAN
6      SET THIS.edition TO edval
7      SET THIS.year TO yrval
8      RETURN true
9    END FUNCTION
10
11 END CLASS
```

This code includes which of the following features of object orientated programming?

a) Object, encapsulation, constructor.
b) Inheritance, polymorphism, encapsulation.
c) Getters, setters, constructor, inheritance.
d) Methods, object, encapsulation.

.....

Q25: A program uses the following method within a class for a user:

```
1  PROCEDURE updateEmail (STRING newEmail)
2    SET THIS.email to newEmail
3  END PROCEDURE
```

What is the correct code to use to update the email data of an object called **currentUser** to scholar@hw.com?

a) `SET currentUser TO "scholar@hw.com"`
b) `SET currentUser.email TO "scholar@hw.com"`
c) `currentUser.updateEmail("scholar@hw.com")`
d) `currentUser.email(email = "scholar@hw.com")`

.....

Q26: Getters and setters are required in a class because of:

a) encapsulation.
b) inheritance.
c) polymorphism.
d) compilation.

.....

Q27: A record structure is set up as:

```
RECORD gameSale IS {STRING title, STRING rating, REAL price}
```

An array of records is declared using this record:

```
DECLARE basket AS ARRAY of gameSale INITIALLY [] * 20
```

The correct notation to reference the `title` of the game with an index of 6 would be:

a) `gameSale[6].basket.title`
b) `basket[6].title`
c) `gameSale[6].title`
d) `title.gameSale[6]`

.....

Q28: Here is a 2D array:

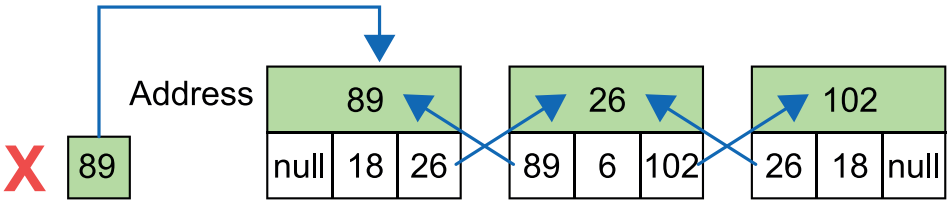
12.3	16.2	17.0
45.6	23.2	19.2
10.0	9.2	9.2
17.9	65.0	29.2

What is the correct statement to declare this 2D array?

a) `DECLARE myArray AS ARRAY of [] INITIALLY`
b) `DECLARE myArray AS ARRAY of ARRAY OF REAL`
c) `SET myArray AS [[]] OF REAL`
d) `DECLARE myArray AS ARRAY OF REAL`

.....

Q29:



The X in the diagram represents the:

a) tail.
b) head.
c) next pointer.
d) previous pointer.

.....

Q30: An advantage of a double linked list when compared with a single linked list is that:

a) insertion can be carried out at any point in the list.
b) deletion can be made at the head of the list.
c) the end of the list is terminated with a null value.
d) the list can be traversed in either direction.

.....

Q31: Which four parameters are required to connect to a database server and select a database using the mysqli API?

a) Client, server, username, password.
b) Username, password, database, tablename.
c) Username, server, password, database.
d) Client, tablename, username, server.

.....

Q32: Which sorting algorithm is being used here?

```
1  PROCEDURE sort(list)
2    DECLARE value INITIALLY 0
3    DECLARE index INITIALLY 0
4    FOR i = 1 to length(list)-1 DO
5      SET value TO list[i]
6      SET index TO i
7      WHILE (index > 0) AND (value < list[index-1]) DO
8        SET list[index] TO list[index-1]
9        SET index TO index - 1
10     END WHILE
11     SET list[index] TO value
12   END FOR
13 END PROCEDURE
```

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

.....

Q33: This version of the bubble sort algorithm has some lines of code missing from the `IF` statement.

```
1  PROCEDURE bubble_sort(list)
2    DECLARE n INITIALLY length(list)
3    DECLARE swapped INITIALLY TRUE
4    WHILE swapped AND n >= 0
5      SET swapped TO False
6      FOR i = 0 to n-2 DO
7        IF list[i] > list[i+1] THEN
8          ????
9          ????
10         ????
11         ????
12       END IF
13     END FOR
14     SET n TO n - 1
15   END WHILE
16 END PROCEDURE
```

Select the missing code from the following:

a)

```
1 SET list[i] TO list[i+1]
2 SET temp TO list[i]
3 SET list[i+1] TO temp
4 SET swapped TO TRUE
```

b)

```
1 SET temp TO list[i]
2 SET list[i] TO list[i+1]
3 SET list[i+1] TO temp
4 SET swapped TO FALSE
```

c)

```
1 SET temp TO list[i]
2 SET list[i] TO list[i+1]
3 SET list[i+1] TO temp
4 SET swapped TO TRUE
```

d)

```
1 SET temp TO list[i]
2 SET list[i] TO list[i+1]
3 SET list[i+1] TO temp
4 SET n TO n+1
```

Topic 4

Testing and evaluation

Contents

Prerequisites

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

- describe, create and implement a comprehensive final test plan to ensure that a program does what it is intended to do;
- identify errors in syntax, execution and logic appropriate to Higher level;
- describe and provide examples of the following debugging techniques;
  - dry runs;
  - trace tables/tools;
  - watchpoints;
  - breakpoints.
- describe, identify and exemplify the evaluation of a solution in terms of:
  - fitness for purpose;
  - efficient use of coding constructs;
  - usability;
  - maintainability;
  - robustness.

Learning objective

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

- describe, implement and give examples of component testing during the development of a solution;
- describe, explain and implement the following approaches to testing:
  - integrative testing;
  - final testing;
  - end user testing;
  - usability testing based on prototypes.
- evaluate solutions in terms of:
  - fitness for purpose;
  - efficiency;
  - usability:
  - maintainability with consideration for perfective, corrective and adaptive maintenance;
  - robustness.

4.1 Revision

Quiz: Revision Go online

Q1: Fill in the missing values in the trace table for the following program:

```
1  DECLARE numbers INITIALLY [17, 15, 4, 7, 8]
2
3  PROCEDURE findMinimum(ARRAY OF INTEGER numbers)
4    DECLARE minimumValue INITIALLY numbers[0]
5    FOR counter FROM 1 TO 4 DO
6      IF minimumValue > numbers[counter] THEN
7        SET minimumValue TO numbers[counter]
8      END IF
9    END FOR
10   SEND ["The smallest value was "& minimumValue TO DISPLAY
11 END PROCEDURE
12
13 findMinimum(numbers)
```

minimumValue	counter	numbers[counter]
	1	
	2	
	3	
	4	

. . . . . .

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: Breakpoints are set in a program to:

a) terminate execution when resources are low.
b) stop compilation when an error occurs.
c) allow program execution to be halted for debugging at a particular point.
d) pass data between components.

. . . . . .

Q4: Component testing is used to:

a) test how a number of modules of the program function together.
b) test the final program meets the requirements.
c) ensure that the program is accessible.
d) test that individual modules of code function as expected.

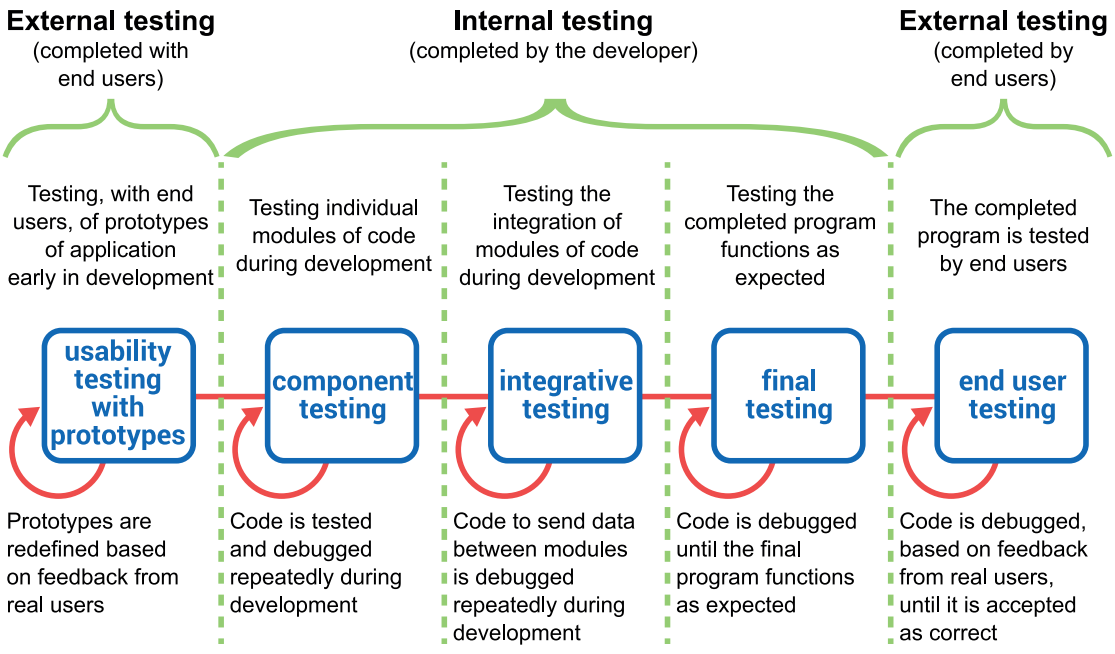
4.2 Introduction

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 revisit a previous stage in the light of subsequent previously undetected problems.

Figure 4.1: Types of testing used through development



4.3 Component testing

Component testing, which is also sometimes referred to as **unit testing**, involves testing individual modules of code to ensure that they function as expected. The purpose of component testing is to ensure that the smallest testable elements of the program function correctly.

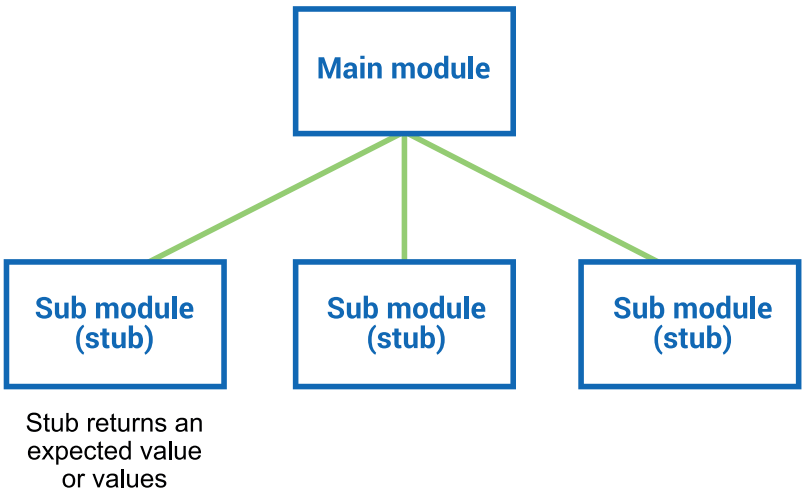
Component testing can take place during the development process and does not need every part of the project to have been completed before the testing takes place. It makes sense to test individual components of a project while they are being written rather than waiting until they are combined together making errors difficult to spot, especially 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**. These are temporary modules which allow the testing to be completed when the whole program is not ready.

Stubs

Stubs are used in a top down testing approach. When a major module is ready to test, but the sub modules are not ready, stubs are created to return expected values as dummy sub modules. A stub may simulate the behavior or be a temporary substitute for code that is still to be developed.

Figure 4.2: Stubs used to test a component



Example : Stub for a sorting procedure

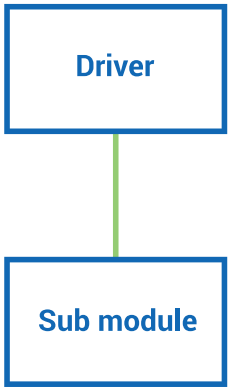
The following stub could be used when a sorting procedure is required in a program but has yet to be written:

```
1 PROCEDURE Sort (ARRAY OF INTEGER numbers)
2   RETURN [0,1,2,3,4,5,6,7,8,9,10]
3 END PROCEDURE
```

Drivers

Drivers are used in a bottom up testing approach. Drivers are used when a sub module is ready to be tested but the main module that it returns values to is not ready. The driver is a program written to call the sub module and display the values it returns.

Figure 4.3: Driver used to test a sub module



Example : Driver used to test a sort procedure

The following driver could be used to test the sort procedure before the rest of the program has been completed:

```
1 PROCEDURE TestSort()
2   DECLARE numbers INITIALLY [4,78,2,67,3,156,5,987,3,23]
3   Sort (numbers)
4   FOR counter FROM 0 TO 9 DO
5     SEND numbers [counter] TO DISPLAY
6   END FOR
7 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 revisited in the light of problems encountered at a later stage.

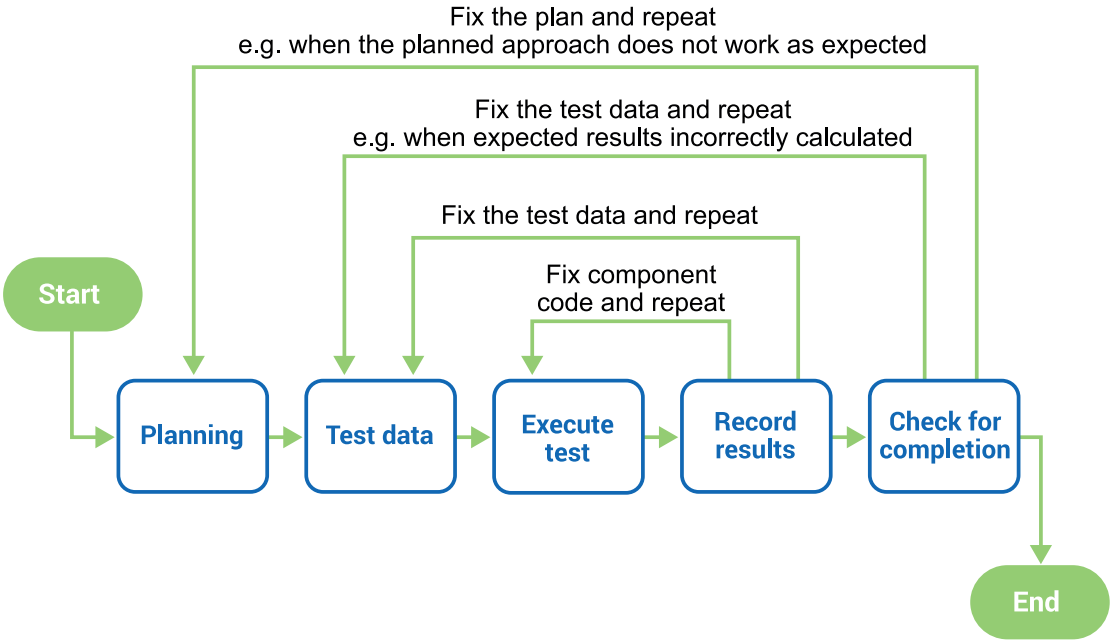
The component testing test process consists of:

1. **Planning:** Planning the test. How will it be carried out, will it use drivers/stubs, what does successful completion look like?
2. **Test data:** Design the test data. Design specific test cases to use with the program. What inputs will be used and what will the expected output be?
3. **Execute test:** Run the test using the planned approach and test data. This will include writing any temporary code to allow the test to be completed.
4. **Record results:** Record the output of the test so that actual output can be compared to expected output.
5. **Check for completion:** Check that the actual output matches that expected. If not identify

why not and repeat the testing process from the appropriate point. If it does, then test is complete.

The processes of running component testing is shown in the following diagram.

Figure 4.4: Carrying out component testing



Once modules have been through component testing, and meet the requirements, it is time to move on to testing how the components all function together.

4.4 Integrative testing

Once component testing is complete, the next stage is to make sure that the components 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. This is known as **integrative testing**.

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 together, then combined with others. Alternatively, if the project is a relatively small one, all the modules may be combined and the system tested as a whole.

Integrative testing follows a similar approach to component testing:

1. **Planning:** Planning the test. How will it be carried out, will it use drivers/stubs, what does successful completion look like?
2. **Test data:** Design the test data. Design specific test cases to use with the program. What inputs will be used and what will the expected output be?

3. **Execute test:** Run the test using the planned approach and test data. This will include writing any temporary code to allow the test to be completed.
4. **Record results:** Record the output of the test so that actual output can be compared to expected output.
5. **Check for completion:** Check that the actual output matches that expected. If not identify why not and repeat the testing process from the appropriate point. If it does, then test is complete.

A test plan for integrative testing 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 testing starts;
- details of what constitutes a successful test.

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

Example : Documentation for an integrative test

Example documentation for an integrative test would be:

Specification: A payroll program reads the details of hours worked and individual payrates from a file. The file contains the employee number, hours worked and payrate for each employee. Once read into the program, pay for each employee is calculated and written to a file.

Test case ID	Test case objective	Test case description	Expected result
1	Check values correctly read from file and stored in array of records data structure.	Use code to the read the file and display the results to screen.	Results to match the content of the file.
2	Check calculation of pay from the data read.	Complete calculations based on specific know values and write results to screen.	Payroll calculated for each employee is stored in an array of records and displayed successfully.
3	Check data is written to file following calculation of payroll.	Program opens file and writes the payroll number and calculated pay to the file.	New records are added to the file which match the expected results.

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

4.5 Final testing

Final testing, also known as system testing, is the final internal check of the assembled program. Final testing is carried out on the completely integrated program to ensure that the program meets the specified requirements.

Final testing is completed in house before the program is made available for any end user testing and is typically carried out independently of the development team that produced the system. Using an independent testing team to complete the final testing removes any bias that the development team might have in how they understand and use the system they have developed. This bias can prevent the discovery of errors in the software. An independent test team, (independent of the development, not necessarily independent of the development company) does not have this bias as they have not been involved in the development.

Typically, final testing involves the creation of a system test plan as follows:

1. **Planning:** Planning the test. How will it be carried out, what does successful completion look like?
2. **Test data:** Design the test data. Design specific test cases to use with the program. What inputs will be used and what will the expected output be? These will link to the initial requirements for the program.
3. **Execute test:** Run the test using the planned approach and test data.
4. **Record results:** Record the output of the test so that actual output can be compared to expected output.
5. **Check for completion:** Check that the actual output matches that expected. If not identify why not and repeat the testing process from the appropriate point. If it does, then test is complete.

4.6 End user testing

End user testing, often called 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 end user testing is that the client accepts delivery of the software, and checks that it is fit for purpose.

End user 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.

Beta testing, because it is done by people who will actually be using the software, can highlight

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.

4.7 Usability testing based on prototypes

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.

Usability testing based on a prototype Go online

If you have access to the internet you can see an excellent example of usability testing based on a prototype, in this video here: https://vimeo.com/114778650

In the video you can see that the prototypes of the interface are tested and then the results of the testing are used to inform further prototypes or the development of the prototype is acceptable.

The following criteria can be used to test and evaluate a user interface for 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.

History of user interfaces Go online

If you have access to the internet why not watch this video on the history of user interfaces:
https://www.youtube.com/watch?v=U1Oy4X5Ni8Y

4.8 Evaluation

An **evaluation** of the software is used to identify areas for improvement, perhaps for future maintenance.

Evaluation should consider the following aspects of the program:

- **Fitness for purpose:** does it do everything as specified? if not, why not? does it include any extra features?
- **Efficiency:** how efficient are the coding constructs used? are modules used efficiently?
- **Usability:** is it easy to use and learn? could it be improved in any way?
- **Maintainability:** what steps have been taken to ensure it is easily maintained?
- **Robustness:** does the program cope with external errors and unforeseen mishaps during execution?

4.8.1 Fitness for purpose

The software is **fit for purpose** if it does what it was initially required to do. Fitness for purpose can be included in an evaluation of the software by confirming that it delivers each part of the initial specification. If the software is enhanced beyond the initial specification this should be recorded here in the evaluation.

The results of testing are key to evaluating the fitness for purpose of the application. While creating the test data, you will have calculated the expected results. If the output matches what was expected then the solution is fit for purpose.

4.8.2 Efficiency

Efficiency relates to how the software uses system resources such as processor time, memory and storage. Typically, this would be assessed at this level by reviewing the efficiency of the code within the program.

To make it more efficient code should be reused where appropriate. Code reuse is achieved by using modular code; where modules are reused to reduce the amount of code needed.

Consider the following two programs which both validate numeric input:

Program 1:

```
DO
  RECEIVE value1 FROM KEYBOARD
  IF value1 < 10 OR value1 > 20 THEN
    SEND "Error, please enter value from 10 to 20"
  END IF
UNTIL value1 >= 10 AND value1 <= 20
DO
  RECEIVE value2 FROM KEYBOARD
  IF value2 < 0 OR value2 > 30 THEN
    SEND "Error, please enter value from 0 to 30"
  END IF
UNTIL value2 >= 0 AND value2 <= 30
DO
  RECEIVE value3 FROM KEYBOARD
  IF value3 < 8 OR value3 > 17 THEN
    SEND "Error, please enter value from 0 to 30"
  END IF
UNTIL value3 >= 8 AND value3 <= 17
```

Program 2:

```
FUNCTION validateinput(INTEGER min, INTEGER max) RETURNS INTEGER
  DO
    RECEIVE value FROM KEYBOARD
    IF value <min OR value > max THEN
      SEND "Error, please enter value from " & min & " to " & max
    END IF
  UNTIL value >= min AND value <= max
  RETURN value
END FUNCTION

SET value1 TO validateinput(10,20)
SET value2 TO validateinput(0,30)
SET value3 TO validateinput(8,17)
```

Program 1 uses multiple modules, or lines within a module, while Program 2 is much more efficient because it uses one routine with parameters to validate the numeric input.

4.8.3 Usability

Usability is defined, in the international standard, as:

> *the extent to which a product can be used by specified users to achieve specified goals with effectiveness, efficiency and satisfaction in a specified context of use.*

A program is usable if the user interface meets these three outcomes:

1. It should be easy for the user to learn and become competent in using the interface from initial use.
2. The interface should make it simple to achieve the intended purpose of the program. A good interface will guide the user to complete the task that the program is intended to perform.
3. The user interface should be so simple to use that once the user has used it, he or she can recall the interface on future visits and how to use it. This means that the interface becomes increasingly familiar over time.

4.8.4 Maintainability

Maintenance is the process of changing a program after it has been written. In order to carry out maintenance the following areas of good practice should be implemented in the program:

- modularity;
- effective use of internal commentary;
- meaningful variable and module names (procedures/functions);
- effective use of white space within code.

We will look at each of these areas in turn.

4.8.4.1 Modularity

Modularity means that the program is divided into separate subprograms. These subprograms can, where appropriate, be reused to ensure that the code is efficient.

An advantage of modularity is that maintenance can be carried out on one area of the program without negatively affecting others or requiring large portions of code to be rewritten.

4.8.4.2 Effective use of internal commentary

The programmer who writes the program may not be the person who maintains it. There also may be some time between writing code and then maintaining it. Effective internal commentary clearly explains the operation of the code so that programmers maintaining the program in the future understand what different sections of the code do.

A partial example of effective use of internal commentary is shown below:

```
function listClasses() {
  //get setup. start with the date
  var now = new Date();
  //get the data as a string for the filename later
  var endTime = now.toISOString();
  //we're going to grab classroom details in batches of 500 at a time
  var pageSizeValue = 500;
  //setup an array to dump the class list into, we won't write it to
     the sheet until we are done as its faster to use memory
  var rows = [];
  //first page being read so this is empty -
  var nextPageToken = '';
  //First record about a classroom is at Row 1
  var startRow = 1;
  //We're going to write the teacher and OU details into a separate
     array as a quick lookup of teacher details
  //(so we only have to read each teacher only once from the Admin
     panel - so it's faster!)
  var ownerArray = [];
  //start a loop which will keep going until all the Classrooms are
     listed
  do {
    //Let's setup the arguments so that we can run the query to get 500
       classroom records
    var optionalArgs = {
      pageSize: pageSizeValue,
      pageToken: nextPageToken,
      fields: "nextPageToken,courses(id, name, ownerId, courseState,
         creationTime, updateTime, section, enrollmentCode)",
    };
    //write the results into an array object cls
    var cls = Classroom.Courses.list(optionalArgs);
    //point to the next 500 to get, nextPageToken bookmarks the next
       set of records to get.
    var nextPageToken = cls.nextPageToken;
    // loop round the result page
    // go through all the results in the array object to grab each
       classroom list and write it to an array
    for (var i = 0, len = cls.courses.length; i < len; i++) {
...
```

4.8.4.3 Meaningful variable and module names

Code is said to be readable if it makes use of meaningful variable and module names that help the programmer understand what the program does. If the contents of a variable are clearly identified by the name of the variable, then this helps ensure the program is maintainable.

4.8.4.4 Effective use of white space within code

Using white space within code helps to make the code more readable by defining the structure of the code.

Consider the following two examples of the use of indentation to show structure:

Code without the use of white space

```
PROCEDURE bubble_sort(list)
DECLARE n INITIALLY length(list)
DECLARE swapped INITIALLY TRUE
WHILE swapped AND n >= 0
SET swapped TO False
FOR i = 0 to n-2 DO
IF list[i] > list[i+1] THEN
SET temp TO list[i]
SET list[i] TO list[i+1]
SET list[i+1] TO temp
SET swapped TO TRUE
END IF
END FOR
SET n TO n - 1
END WHILE
END PROCEDURE
```

This code is difficult to read. If the same code is presented with white space / indentation to show the structure it becomes much easier to read.

Code using white space / indentation

```
PROCEDURE bubble_sort(list)

  DECLARE n INITIALLY length(list)
  DECLARE swapped INITIALLY TRUE

  WHILE swapped AND n >= 0

    SET swapped TO False
    FOR i = 0 to n-2 DO
      IF list[i] > list[i+1] THEN
        SET temp TO list[i]
        SET list[i] TO list[i+1]
        SET list[i+1] TO temp
        SET swapped TO TRUE
      END IF
    END FOR
    SET n TO n - 1

  END WHILE

END PROCEDURE
```

This code uses indentation to align open and closing structures within the code. Blank lines are also used to separate sections of code to assist with readability. This use of white space / indentation makes amending the program much easier as the structure is clearer to the programmer making the changes.

4.8.4.5 Types of computer maintenance

There are three types of computer maintenance. These are:

- corrective;
- adaptive;
- perfective.

Corrective maintenance

Corrective maintenance is the fixing of bugs in the software. Creating computer software is a complex process and, despite rigorous testing, often errors in coding will make their way through to the finished program.

Corrective maintenance corrects errors in the program so that it adheres to the original specification.

Adaptive maintenance

Adaptive maintenance amends the program in response to changes in the operating environment. For example, an organisation changes the operating system that their computers run on. This change means that a program that is used in the organisation stops working. Adaptive maintenance is then carried out to allow the program to run on the new operating system.

Perfective maintenance

Perfective maintenance is carried out to improve the program. Over time users often find that a program written for them does not do everything they would want it to. Perfective maintenance often adds new features to a program to allow it to better meet user requirements.

4.8.5 Robustness

Robustness is the program's ability to cope with errors during execution without failing. Evaluation of robustness will depend on the results of testing. It is possible for a program to fully meet its requirements but to not be particularly robust.

Programmers often refer to a program as being able to gracefully handle errors when they are encountered. This means that the program should not crash out (cease running with little or no feedback to the user) but should notify the user of the error and provide a point from which the user can continue to use the program.

4.9 Learning points

Summary

You should now be able to:

- describe, implement and give examples of component testing during the development of a solution;
- describe, explain and implement the following approaches to testing:
  - integrative testing;
  - final testing;
  - end user testing;
  - usability testing based on prototypes.
- evaluate solutions in terms of:
  - fitness for purpose;
  - efficiency;
  - usability:
  - maintainability with consideration for perfective, corrective and adaptive maintenance;
  - robustness.

4.10 End of topic test

End of topic test: Testing and evaluation Go online

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? (Choose all that apply)

a) Documentation is required by the client.
b) Documentation will aid anyone who needs to revisit a previous stage in the light of subsequent previously undetected problems.
c) Documentation will be invaluable to those testing subsequent stages.
d) Documentation will reduce the cost of developing the software.

. . . . . .

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 integrative testing? (Choose all that apply)

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) Integrative testing
b) Component testing
c) Final testing
d) Usability 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: When evaluating the efficiency of a program it is important to consider whether the program:

a) has meaningful variable names and module identifiers.
b) includes internal commentary.
c) contains modular code which effectively reuses modules.
d) graciously traps errors when the program fails.

. . . . . .

Q13: What is the purpose of internal commentary within program code?

a) To provide user documentation for end users of the program.
b) To explain the function of the code for maintenance.
c) To provide a usable interface for the program.
d) To provide guidance for end user testing.

. . . . . .

Q14: A program is robust if:

a) it meets the requirements of the specification.
b) it can be compiled using either an interpreter and a compiler.
c) it uses stubs and drivers as part of testing.
d) it provides feedback and the opportunity to recover if an error is encountered during execution.

Topic 5

Software design and development test

Software design and development test Go online

Q1: Which of the following statements are true of maintainability?

a) Maintainability is improved by having readable code.
b) Global variables improve maintainability.
c) Maintainability is improved through having effective internal commentary.
d) Maintainability 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 basing a new class, a subclass, on another class.

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

. . . . . .

Q5: Which four items of data are required to open a connection to a database server using PHP?

a) Server name / location, access ports, connection string, username.
b) Server name / location, database name, database table, query.
c) Server name / location, username, password, database table.
d) Server name / location, username, database name, database query.

. . . . . .

Q6: What are ***instance variables*** in object oriented programming?

a) Local variables declared within the methods defined in a class.
b) Static class variables, declared outside the methods of the class, which only exist once in memory across the class and all subclasses.
c) Methods declared within the class to process values when the class is instantiated.
d) Variables, declared outside the methods of the class, that are allocated memory when the class is instantiated, and which record the state of the instantiated object.

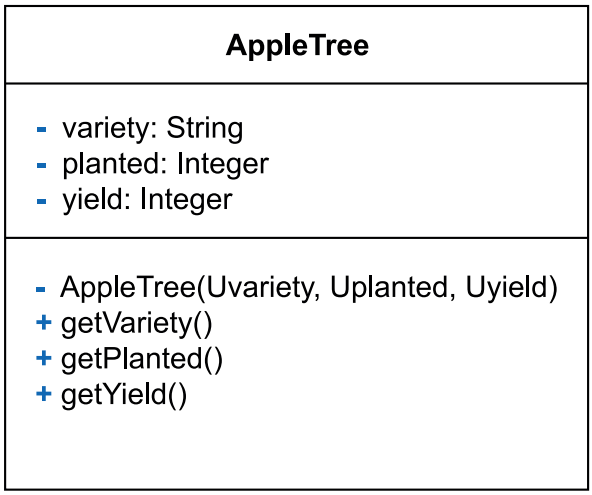
. . . . . .

Q7: Which method is called when a class is instantiated to create an object?

a) Assembler
b) Setter
c) Getter
d) Constructor

. . . . . .

Q8: Review the following UML class diagram.

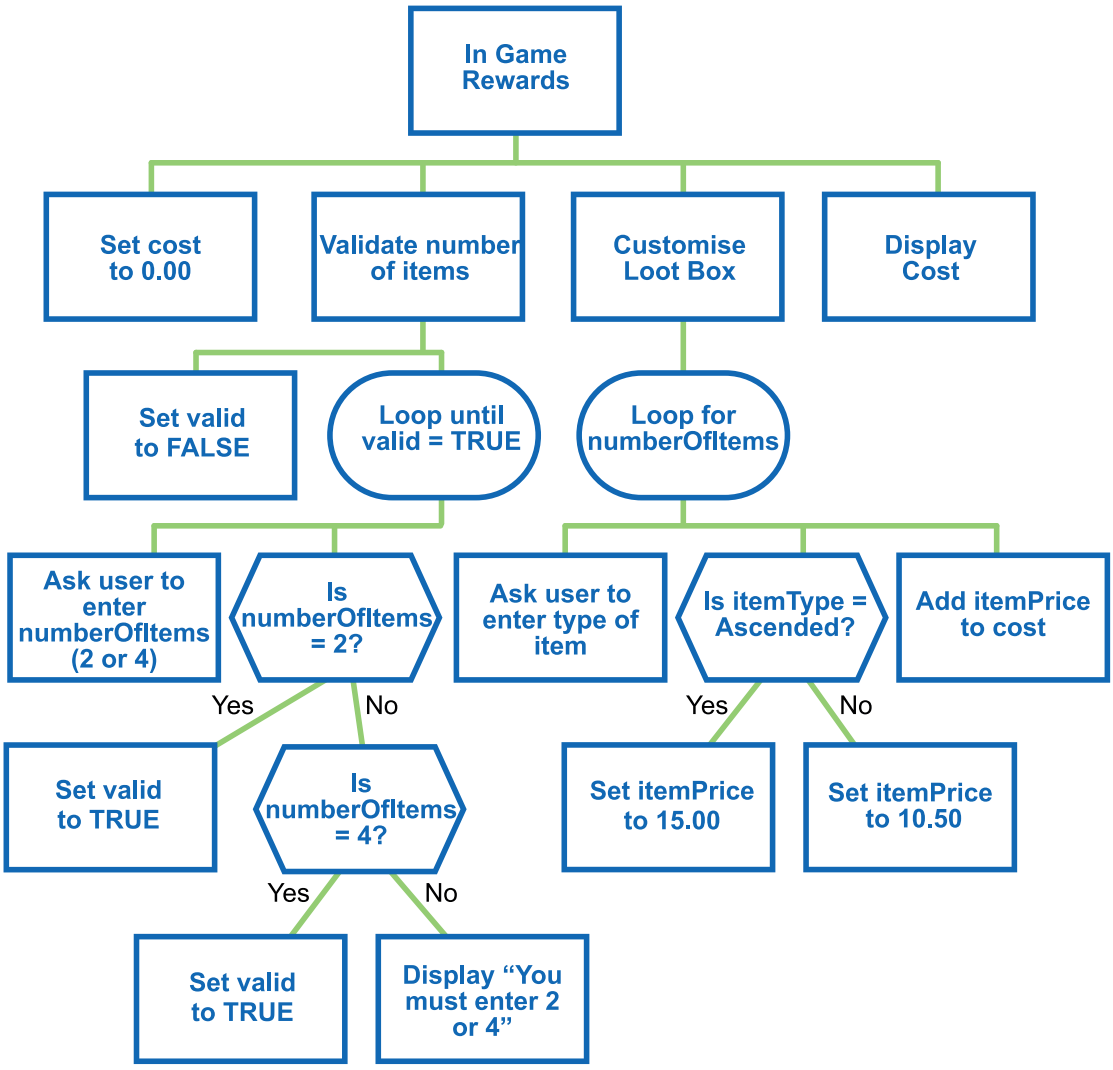


Which of the following descriptions of this class is true?

a) The constructor can be accessed directly from within the program.
b) Once an object is created based on this class, the ***variety*** and ***yield*** instance variables cannot be updated.
c) Setters and getters are presented for each instance variable.
d) The ***variety*** instance variable can be accessed directly.

. . . . . .

Q9: Review the following structure diagram.



How many selection constructs are in this diagram?

a) 2
b) 3
c) 4
d) 6

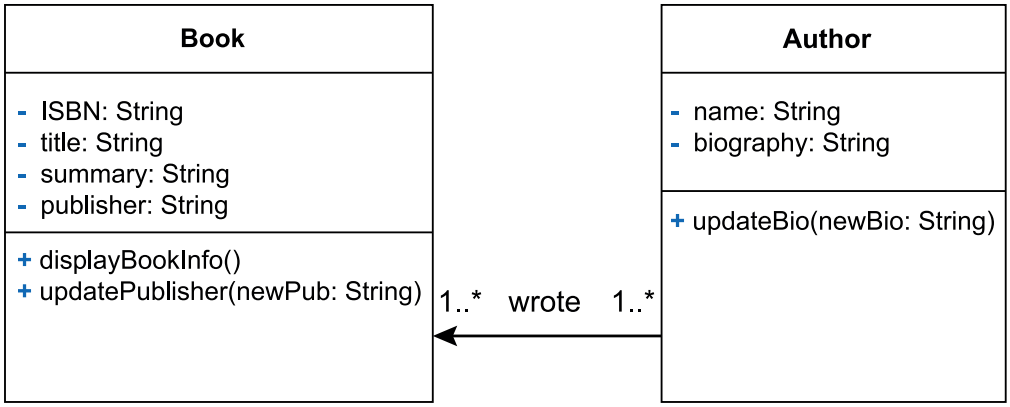
. . . . . .

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 by end users.
d) A scale model of a project.

. . . . . .

Q11: Review the association between these two classes in this UML diagram.

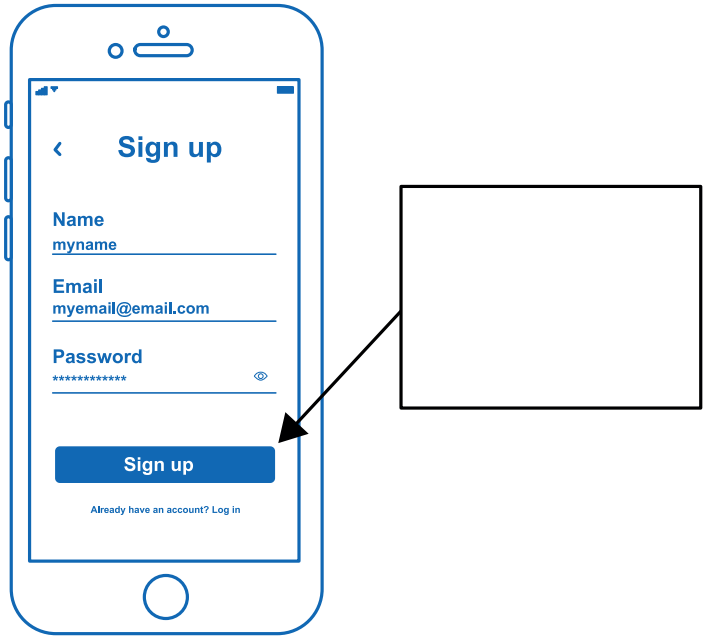


Which one of the following statements applies to this diagram?

a) All books must have one or more authors and each author must have more than one book.
b) An author can exist without an associated book.
c) An author can write any number of books, but a book can have only one author.
d) A book must have one or more authors and each author must have one or more books.

. . . . . .

Q12: Review this wireframe for an application.



Identify the most suitable text to include in the wireframe to design the underlying process of the "**Sign up**" button.

a) Creates a user account based on the details entered in this page.
b) Logs user in using the entered credentials.
c) Validates the information against known users.
d) Adds an item to the user's shopping cart.

.....

Q13: In the SQA reference language, the constructor of a class is defined using which of the following syntax?

a) `CONSTRUCTOR METHOD (DATATYPE property, ...)`
b) `CONSTRUCTOR FUNCTION (DATATYPE property, ...) RETURNS DATATYPE`
c) `CONSTRUCTOR (DATATYPE property, ...)`
d) `CLASS CONSTRUCTOR (DATATYPE property, ...)`

.....

Q14:

```
1  CLASS sheepDog INHERITS dog WITH (OVERRIDE BOOLEAN breed)
2
3  METHODS
4    ...
5    OVERRIDE FUNCTION getBreed()
6      IF THIS.breed = TRUE THEN
7        RETURN "Border Collie"
8      ELSE
9        RETURN "Unknown"
10     END IF
11   END FUNCTION
12   ...
13 END CLASS
```

Which object oriented programming concept is being implemented by this code?

a) Inheritance
b) Polymorphism
c) Instantiation
d) Constructor

.....

Q15:

An array of records is required to store the details of pupils placed on a waiting list for a school. The waiting list stores, the pupil first and lastname, parent first and last name and parent phone number. The waiting list has a maximum of 50 places. Which is the correct code to define an array of records for this purpose?

a)

```
1  DECLARE waitingList ARRAY AS {STRING ALL AS pupilFname,
     pupilLname, parentFname, parentLname, phoneNumber} * 50
```

b)

```
1  RECORD waitingRow IS {STRING pupilFname, STRING pupilLname,
     STRING parentFname, STRING parentLname, STRING phoneNumber}
2  DECLARE waitingList AS ARRAY of waitingRow INITIALLY [] * 50
```

c)

```
1 RECORD waitingRow IS {STRING pupilFname, STRING pupilLname,
     STRING parentFname, STRING parentLname, STRING phoneNumber}
     * 50
2 DECLARE waitingList AS ARRAY of waitingRow
```

d)

```
1 DECLARE waitingList AS [] * 50 INITIALLY {STRING pupilFname,
     STRING pupilLname, STRING parentFname, STRING parentLname,
     STRING phoneNumber}
```

.....

Q16: Which sorting algorithm is being used here?

```
 1 PROCEDURE someCode(list)
 2   DECLARE value INITIALLY 0
 3   DECLARE index INITIALLY 0
 4   FOR i = 1 to length(list)-1 DO
 5     SET value TO list[i]
 6     SET index TO i
 7     WHILE (index > 0) AND (value < list[index-1]) DO
 8       SET list[index] TO list[index-1]
 9       SET index TO index - 1
10     END WHILE
11     SET list[index] TO value
12   END FOR
13 END PROCEDURE
```

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

.....

Q17: A program will record a time for each athlete. There are 10 athletes and they will be recorded in order of their training run. Which data structure is most appropriate to use for this problem?

a) Array of records
b) Parallel 1D arrays
c) 2D array
d) Array of objects

.....

Q18: 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.

. . . . . .

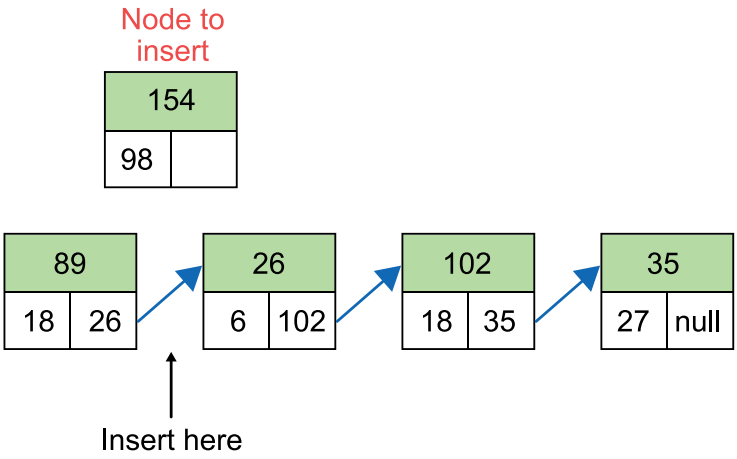
Q19:

In component testing, a section of code designed to test a function or procedure is a:

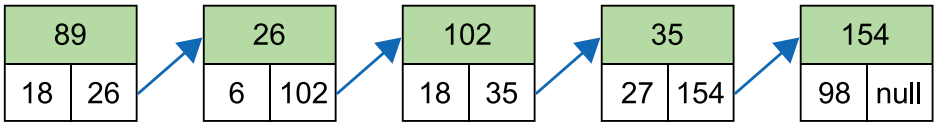
a) stub.
b) driver.
c) sub procedure.
d) code block.

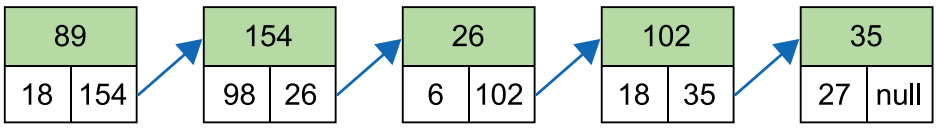
. . . . . .

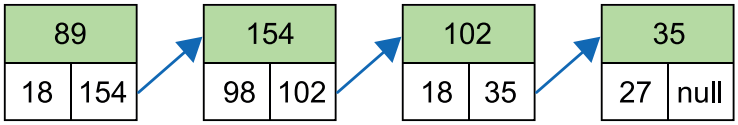
Q20: A single linked list is shown below.

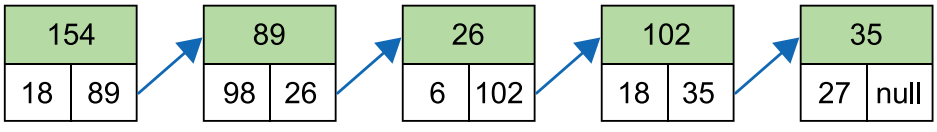


Which diagram shows the correct state of the list after inserting the item shown?

a)


b)


c)


d)


.....

Q21: What is the purpose of null values in a double linked list?

a) To be a place holder for new items to be inserted from the list.
b) To be written into the list to delete items from it.
c) To be stored in the next or previous pointers at the beginning/end of the list.
d) To record the state of an empty list.

.....

Q22: Which disadvantage of a single linked list is resolved by using a double linked list?

a) Limits on the maximum about of memory available to the list.
b) Errors in locating the insertion point within the list.
c) Recording the head of the list.
d) Being able to traverse the list in both directions.

.....

Q23: The following algorithm for a binary search is incomplete:

```
 1 Line 1.PROCEDURE binary_search(list,target)
 2 Line 2.    DECLARE low INITIALLY 0
 3 Line 3.    DECLARE high INITIALLY length(list)-1
 4 Line 4.    DECLARE mid INITIALLY 0
 5 Line 5.    DECLARE found INITIALLY FALSE
 6 Line 6.    WHILE NOT found AND low <= high
 7 Line 7.      SET mid TO (low+high)/2
 8 Line 8.      IF target = list[mid] THEN
 9 Line 9.        SEND "Found at position " & mid TO DISPLAY
10 Line 10.       SET found TO TRUE
11 Line 11.     ELSE IF target > list[mid] THEN
12 Line 12.       ???
13 Line 13.     ELSE
14 Line 14.       ???
15 Line 15.     END IF
16 Line 16.   END WHILE
17 Line 17.   IF found = FALSE THEN
18 Line 18.     SEND "Target not found" TO DISPLAY
19 Line 19.   END IF
20 Line 20.END PROCEDURE
```

Select the correct code for Lines 12 and 14 from the following:

a) `Line 12 SET low TO mid+1`
   `Line 14 SET high TO mid+1`
b) `Line 12 SET low TO mid-1`
   `Line 14 SET high TO mid+1`
c) `Line 12 SET high TO mid-1`
   `Line 14 SET low TO mid+s`
d) `Line 12 SET low TO mid+1`
   `Line 14 SET high TO mid-1`

.....

Q24: An algorithm for the bubble sort is shown below:

```
1  PROCEDURE bubble_sort(list)
2    DECLARE n INITIALLY length(list)
3    DECLARE swapped INITIALLY TRUE
4    WHILE swapped AND n >= 0
5      SET swapped TO False
6      FOR i = 0 to n-2 DO
7        IF list[i] > list[i+1] THEN
8          SET temp TO list[i]
9          SET list[i] TO list[i+1]
10         SET list[i+1] TO temp
11         SET swapped TO TRUE
12       END IF
13     END FOR
14     SET n TO n - 1
15   END WHILE
16 END PROCEDURE
```

What is the purpose of the variable *swapped* in this algorithm?

a) To count the number of swaps in each pass through the array.
b) To act as temporary store for values when they are swapped.
c) To detect if a swap has not occurred allowing the loop to terminate when the list is fully sorted.
d) To store the length of the list.

.....

Q25: This code populates a 2D integer array:

```
1  PROCEDURE Fillarray(ARRAY OF INTEGER myArray)
2    FOR row FROM 0 TO 4 DO
3      FOR column FROM 0 TO 4 DO
4        SET  myArray[row][column] TO row + column + 1
5      END FOR
6    END FOR
7  END PROCEDURE
```

What value would the following command send to the screen?

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

a) 2
b) 4
c) 5
d) 6

.....

Q26: Here is a 2D array:

12.3	16.2	17.0
45.6	23.2	19.2
10.0	9.2	9.2
17.9	65.0	29.2

What is the correct statement to declare this 2D array?

a) `DECLARE myArray AS ARRAY of [] INITIALLY`
b) `DECLARE myArray AS ARRAY of ARRAY OF REAL`
c) `SET myArray AS [[]] OF REAL`
d) `DECLARE myArray AS ARRAY OF REAL`

. . . . . .

Q27: Which of these groups of people will **not** be involved in integrative testing? Choose **two** options.

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

. . . . . .

Q28: What type of testing is performed on the finished product prior to acceptance by the client?

a) Integrative testing
b) Component Testing
c) Final testing
d) Usability testing

. . . . . .

Q29: What is the purpose of internal commentary within program code?

a) To provide user documentation for end users of the program.
b) To explain the function of the code for maintenance.
c) To provide a usable interface for the program.
d) To provide guidance for end-user testing.

. . . . . .

Q30: What is meant by efficiency when evaluating a program?

a) Code is reused and modules used to reduce the amount and complexity of code. It makes good use of system resources.
b) The code is constructed within the require timescale by the development team, making efficient use of the resources.
c) The program performs as expected and does what it was initially required to do.
d) The interface of the program is designed for the intended audience.

Glossary

1D array

a type of linear array which is accessed using a single index e.g. myArray[56]

2D array

a type of array which is accessed using a two index e.g. myArray[7,10] because the array has 2 dimensions

Acceptance criteria

the agreed critera which have to be met before the product can be handed over to the client.

Actor

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

Agile

an approach to software development that is flexible and focuses on working software over documentation and complete sets of requirements.

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

Budget

the amount of money available to complete a project.

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)

Component testing

the testing of modules of code without integration with other modules

Constraints

the variables that limit the activities of the project: scope, time, cost and quality.

Constructor

a default method for a class which is used when an object is created. The constructor may give values to none, some or all of the instance variables for the object when it is created from the class

Cost

the financial value of a resource used to deliver part of a project (may also apply to the whole project).

Data integrity

when data transfer between hardware or software components occurs without errors

Documentation

anything written or recorded about a program. This can take the form of internal commentary, user guides, the result of testing at any stage and evaluative commentary on the program

Dot notation

using dot notation an object can be specified and then a property of that object can be referenced. e.g. The value of the property "name" for the object "Rex" can be referenced using Rex.name. Similarly, in SQL dot notation is used to reference TABLES and then COLUMN within the table e.g. customer.name (where customer is the database table and name is the column holding the data)

Driver

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

Encapsulation

a characteristic of objects in object oriented programming which means that objects are closed systems which cannot be altered from outside

Evaluation

an evaluation of software is used to identify areas for improvement, perhaps for future maintenance

Fitness for purpose

a program is fit for purpose if it meets the requirements of the specification

Getter

a method in a class for retrieving the value of an instance variable

Inheritance

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

Instance variable

in object oriented programming the instance variables of an object are the data items which that object uses

Instance variables

In object oriented programming the instance variables of an object are the data items which that object uses

Instantiation

the process of creating an object from a class. An object is a ***real*** item rather than an abstract definition (a class)

Integrative testing

testing how modules of the program work together. It is focused on the interfacing between the modules i.e. the passing and returning of parameters

Linked list

a data structure that allows sequences of values to be chained together

Method

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

Milestones

a significant stage or event in the development of a project.

MySQL

a popular open source database

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

Objectives

what the project is intended to achieve.

PHP

a server-side scripting language designed for web development but also used as a general purpose programming language

Polymorphism

in object oriented programming, this refers to a programming language's ability to redefine methods for derived classes

Private

in object oriented programming an instance variable or method that is private can only be accessed by a method of the class in which it was defined

Product backlog

is the single most important artifact in the Scrum approach to agile development. The product backlog is, in essence, an incredibly detailed analysis document, which outlines every requirement for a system, project, or product.

Project manager

the person in overall charge of the planning and execution of a particular project.

Property

an instance variable, a value within the class

Pseudocode

an informal high level description of how a computer program functions

Public

in object oriented programming an instance variable or method that is public can be accessed directly by any method within any other object in the program

Quality

the standard of something as measured against other things of a similar kind

Record

a collection of fields, possibly of different data types, typically in fixed number and sequence

Scope

the detail of what a project is to achieve. Changes to the scope of the project have a direct impact on the time and cost of the project.

SCRUM

is an iterative and incremental agile software development framework for managing product development.

Setter

a method in a class for establishing the value of an instance variable

Skeuomorphism

when an interface mimics the hardware which an application is replacing

Sprint

a period of development when a fixed set of product items (the sprint backlog) is developed.

Structure diagram

a modelling tool used to document the structures that make up a program. It shows the hierarchy, or structure, of the different components, or modules, of a system and shows how they connect and interact with each other. It includes structures for selection, iteration and sequence

Stub

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

Subclass

in object oriented programming a class that is declared and inherits from another class

Timescale

a period of time to complete an activity.

Top down analysis

breaking a problem down into smaller sub problems in order to make it easier to build a solution

Unified Modelling Language (UML)

a standardised modelling language used by programmers to define various aspects of an application's design. Both Class diagrams and Use Case diagrams are available within the UML set of tools

Use cases

is an approach used in system analysis to identify, clarify, and organize system requirements. Use cases are made up of sets of possible sequences of interactions between systems and users in a particular environment and related to a particular goal.

User persona

is a representation of the goals and behavior of a hypothesized group of users. In most cases, personas are created from data collected from interviews with users.

User scenarios

describe the stories and context behind why a specific user or user group comes to your site or use your application. They note the goals and questions to be achieved and sometimes define the possibilities of how the user(s) can achieve them on the site.

User stories

are short, simple description of a feature told from the perspective of the person who desires the new capability, usually a user or customer of the system.

Waterfall model

is a sequential design process, used in software development processes, in which progress is seen as flowing steadily downwards (like a waterfall) through the phases of Conception, Initiation, Analysis, Design, Construction, Testing, Production/Implementation and Maintenance.

Wireframe

a two-dimensional illustration of an application's interface that specifically focuses on the visual layout, inputs, validation, underlying processes and outputs of the application

Answers to questions and activities

Topic 1: Analysis

Activity: Types of feasibility (page 12)

Q1:

A project that cannot be delivered in the time available because of the complexity of the task	has an issue with schedule feasibility.
A project which has insufficient finance to hire the necessary staff to complete the project when required	has an economic feasibility issue.
A project that processes banking transactions to indicate spending habits in contravention of data protection laws	has an issue with legal feasibility.
A project which cannot proceed because the software to link two data sources into the project is not available	has an issue with technical feasibility.

Review of Use Case Diagrams (page 19)

Q2:

Use Case diagrams are made using **Actors** that will interact with the system. The system is indicated by a **system boundary** where all the use cases will be placed. A Use Case in the diagram will be represented by an **ellipse**.

There are 5 relationships we need to use in a Use Case Diagram:

Association

Include — this is a use case that is always **actioned** but not directly by the Actor.

Extend — This Use Case is not directly accessed by the Actor and doesn't always run.

Generalisation of an **Actor**

Generalisation of a Use Case

End of topic test: Analysis (page 19)

Q3: a) A person or entity that interacts with the system.

Q4: c) how actors interact with the system.

Q5: c) economic and technical

Q6:

- Time
- Scope
- Legal
- Cost

Q7:

- End User Requirements
- Scope
- Boundaries
- Constraints

Topic 2: Design

Quiz: Revision (page 23)

Q1: b) Actual parameters

Q2: a) Formal parameters

Q3: b) The modules in a structure chart will become modules in the finished program.

Q4: d) Pseudocode

Q5: b) Pseudocode

Q6: d) top down design.

Q7: a) creating pseudocode from a structure diagram and data flow diagram.

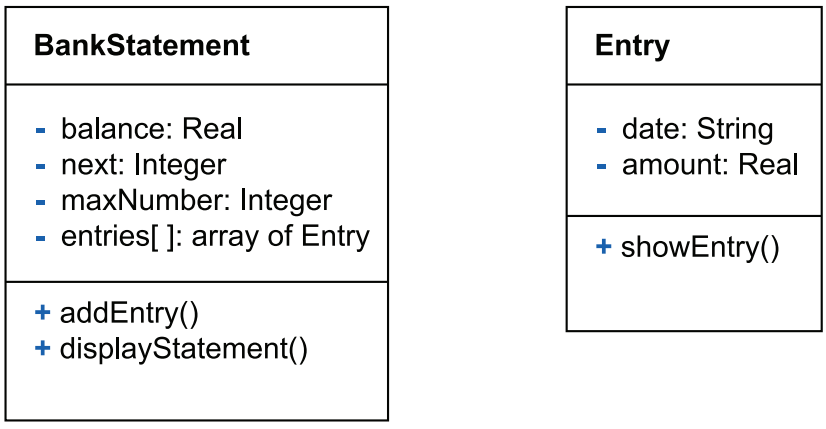
Pseudocode: shopping algorithm (page 42)

Q8: Possible answer:

```
1  PROCEDURE  GetCustomerChoice()
2    DECLARE answer AS STRING INITIALLY ""
3    FOR Counter FROM 0 TO 2 DO
4      SEND "Do you wish to purchase a "& shoppingBasket[counter].
           productName & " Enter Y or N" TO DISPLAY
5       RECEIVE answer FROM KEYBOARD
6      IF answer = "Y" OR answer = "y" THEN
7        SET shoppingBasket[counter].purchased TO true
8        SEND shoppingBasket[counter].productName & " added to shopping
             basket" TO DISPLAY
9      END IF
10   END FOR
11 END PROCEDURE
```

Class: bank statement (page 46)

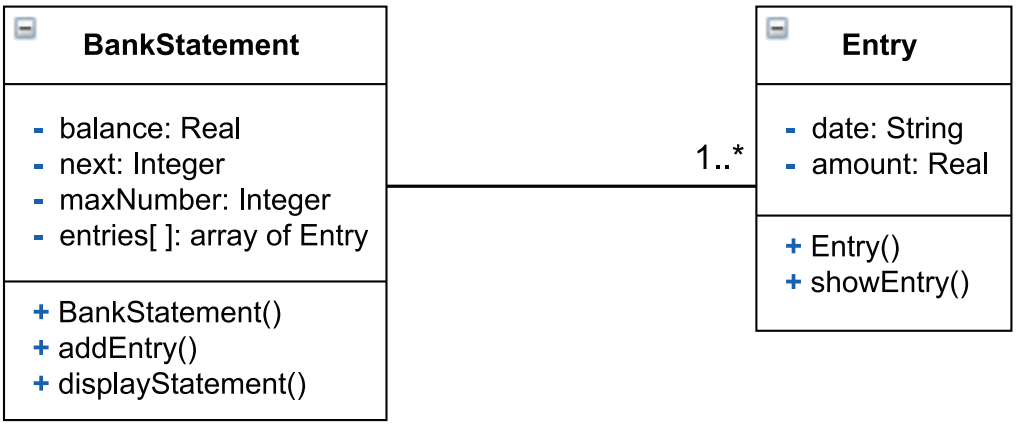
Q9: Possible answer:



Q10: Possible answer:

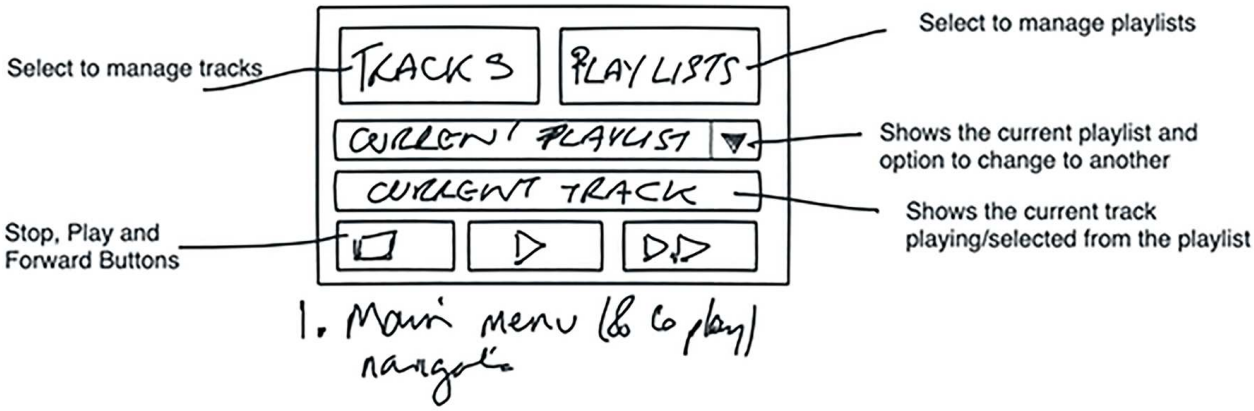
```
1  CLASS Entry IS {STRING date, REAL amount}
2  METHODS
3
4  PROCEDURE ShowEntry()
5    SEND THIS.date & " " & THIS.amount TO DISPLAY
6  END PROCEDURE
7
8  END CLASS
9
10 CLASS BankStatement IS {ARRAY OF Entry entries,INTEGER next, INTEGER
       maxnumber, REAL balance}
11 METHODS
12
13 CONSTRUCTOR BankStatement()
14   DECLARE THIS.balance INITIALLY 0
15   DECLARE THIS.next INITIALLY 0
16 END CONSTRUCTOR
17
18 PROCEDURE AddEntry (Entry newEntry)
19   IF THIS.next= THIS.maxnumber THEN
20     <error - too many entries>
21   ELSE
22     SET THIS.entries[next] TO newEntry
23     SET THIS.next TO THIS.next + 1
24     SET THIS.balance TO THIS.balance + newEntry.amount
25   END IF
26 END PROCEDURE
27
28 PROCEDURE DisplayStatement()
29   FOR counter FROM 0 TO next - 1 DO
30     THIS.entries[counter].showEntry()
31   END FOR
32 END PROCEDURE
33
34 END CLASS
```

Q11: Possible answer:

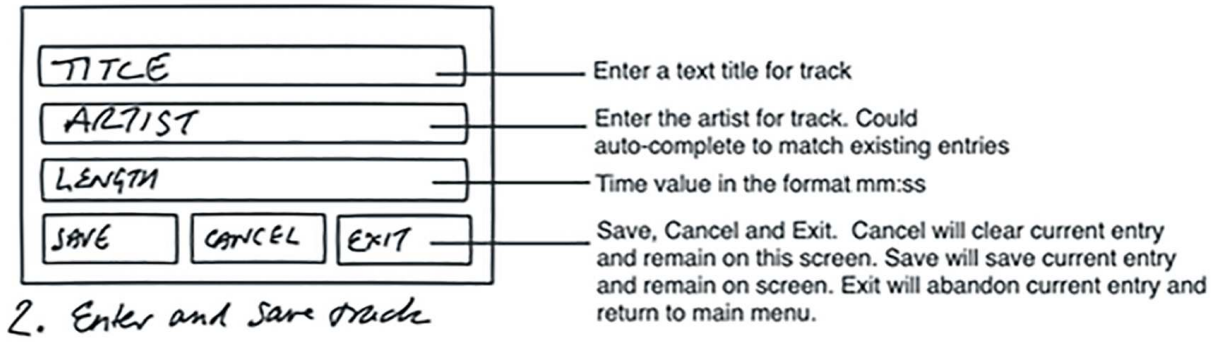


Create wireframes for the PlayList system (page 52)

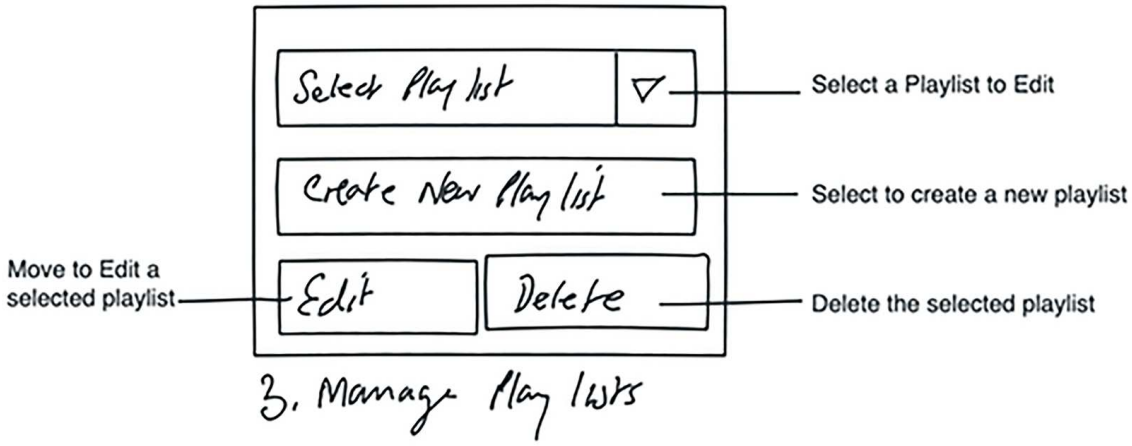
Q12: Possible answer:



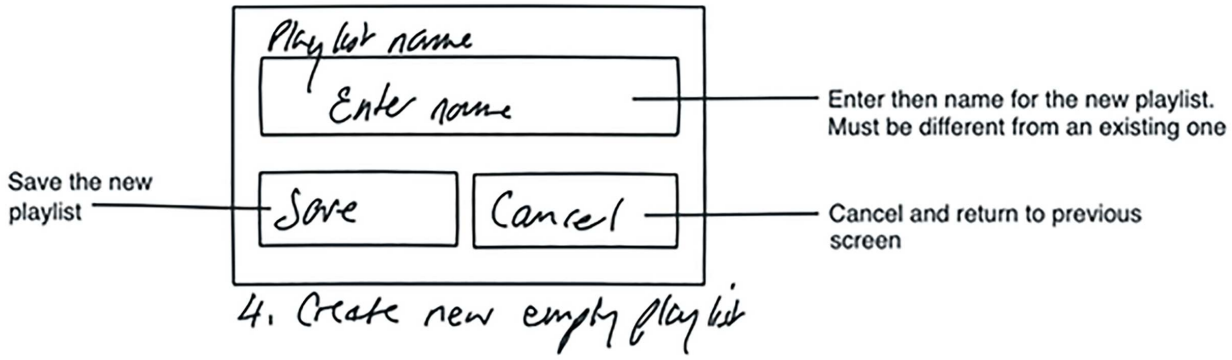
Q13: Possible answer:



Q14: Possible answer:



Q15: Possible answer:



Q16: Possible answer:

End of topic test: Design (page 54)

Q17: a) encapsulation

Q18: a) a class

Q19: b) inheritance

Q20: c) arrows below the process showing the data flowing in and out with appropriate labels.

Q21:

a) It shows the instance variables and methods for a class and the nature of the encapsulation (private/public) for each.

b) It shows the association between classes.

c) It shows how one class may inherit the details of another class.

Topic 3: Implementation

Quiz: Revision (page 57)

Q1: b) input validation.

Q2: a) counting occurrences.

Q3: a) a fixed loop and a boolean variable.

Q4: a) a fixed loop.

Q5: c) linear search.

Create a dog class (page 64)

Q6: Possible answer:

```
1  CLASS dog IS {STRING name * 30, STRING colour * 12, REAL height, REAL
       length, REAL weight, STRING breed}
2
3  METHOD
4
5    CONSTRUCTOR (STRING dogName * 30, STRING dogColour * 12, REAL
         dheight, REAL dlength, REAL dweight, STRING dbreed)
6
7    DECLARE THIS.name INITIALLY dogName
8    DECLARE THIS.colour INITIALLY dogColour
9    DECLARE THIS.height INITIALLY dheight
10   DECLARE THIS.length INITIALLY dlength
11   DECLARE THIS.weight INITIALLY dweight
12   DECLARE THIS.breed INITIALLY dbreed
13
14   END CONSTRUCTOR
15
16   FUNCTION getName() RETURNS STRING
17     RETURN THIS.name
18   END FUNCTION
19
20   PROCEDURE setName(STRING dogName)
21     SET THIS.name TO dogName
22   END PROCEDURE
23
24   FUNCTION getColour() RETURNS STRING
25     RETURN THIS.colour
26   END FUNCTION
27
28   PROCEDURE setColour(STRING dogColour)
29     SET THIS.colour TO dogColour
30   END PROCEDURE
31
32   FUNCTION getHeight() RETURNS REAL
33     RETURN THIS.height
34   END FUNCTION
```

```
35
36    PROCEDURE setHeight(REAL dheight)
37      SET THIS.height TO dheight
38    END PROCEDURE
39
40    FUNCTION getLength() RETURNS REAL
41      RETURN THIS.length
42    END FUNCTION
43
44    PROCEDURE setLength(REAL dlength)
45      SET THIS.length TO dlength
46    END PROCEDURE
47
48    FUNCTION getWeight() RETURNS REAL
49      RETURN THIS.weight
50    END FUNCTION
51
52    PROCEDURE setWeight(REAL dweight)
53      SET THIS.weight TO dweight
54    END PROCEDURE
55
56    FUNCTION getBreed() RETURNS STRING
57      RETURN THIS.breed
58    END FUNCTION
59
60    PROCEDURE setBreed(STRING dbreed)
61      SET THIS.breed TO dbreed
62    END PROCEDURE
63
64  END CLASS
```

Implement the policeDog subclass (page 65)

Q7: Possible answer:

```
1  CLASS policeDog INHERITS dog WITH {INTEGER policeID, STRING handler}
2
3  METHODS
4
5    CONSTRUCTOR (STRING name, INTEGER age. STRING dogName * 30, STRING
         dogColour * 12, REAL dheight, REAL dlength, REAL dweight, STRING
         dbreed)
6
7    DECLARE THIS.name INITIALLY dogName
8    DECLARE THIS.colour INITIALLY dogColour
9    DECLARE THIS.height INITIALLY dheight
10   DECLARE THIS.length INITIALLY dlength
11   DECLARE THIS.weight INITIALLY dweight
12   DECLARE THIS.breed INITIALLY dbreed
13   DECLARE THIS.policeID INITIALLY policeID
14   DECLARE THIS.handler INIITIALLY handler
15
16   END CONSTRUCTOR
17
18   FUNCTION getPoliceID() RETURNS INTEGER
19     RETURN THIS.policeID
20   END FUNCTION
21
22   PROCEDURE setPoliceID(INTEGER pID)
23     SET THIS.policeID TO pID
24   END PROCEDURE
25
26   FUNCTION getHandler() RETURNS STRING
27     RETURN THIS.handler
28   END FUNCTION
29
30   PROCEDURE setHandler(STRING pHandler)
31     SET THIS.handler TO pHandler
32   END PROCEDURE
33
34 END CLASS
```

Implement the sheepDog subclass using polymorphism (page 66)

Q8: Possible answer:

```
CLASS sheepDog INHERITS dog WITH (OVERRIDE BOOLEAN breed)

METHODS

  OVERRIDE FUNCTION getBreed()
    IF THIS.breed = TRUE THEN
      RETURN "Border Collie"
    ELSE
      RETURN "Unknown"
    END IF
  END FUNCTION

END CLASS
```

Implement a record structure (page 68)

Q9: Possible answer:

```
RECORD tweet IS {STRING handle, STRING message, INTEGER views}

SET tweet.handle TO "compednet"
SET tweet.message TO "This is a great example of a record"
SET tweet.views TO 178

SEND "Handle:" & tweet.handle TO DISPLAY
SEND "Message:" & tweet.message TO DISPLAY
SEND "Views:" & tweet.views TO DISPLAY
```

Arrays of records (page 70)

Q10: Possible partial answer:

```
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

END PROCEDURE
```

Song, artist, number of plays program (page 72)

Q11: Possible answer:

```
DECLARE song AS ARRAY OF STRING INITITALLY ("Xanadu", "I'm Not
   Okay","Kiss","Light My Fire","Barcelona")
DECLARE artist AS ARRAY OF STRING INITIALLY ("Rush","My Chemical
   Romance","Prince","The Doors","Twin Atlantic")
DECLARE plays AS ARRAY OF INTEGER INITIALLY (3092, 2980, 3010,1893,429)

FOR element FROM 0 to 4 DO
  SEND song(element), artist(element), plays(element) TO DISPLAY
END FOR
```

3 x 3 maze using a 2D array (page 75)

Q12: Possible answer:

```
FUNCTION setupMaze (maze)

  FOR column FROM 0 TO 8 DO
    FOR row FROM 0 TO 3 DO
      SET maze[column , row] TO -1
    END FOR
  END FOR

  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
  SET maze [3 ,1] TO 4
  SET maze [3 ,2] TO 6
  SET maze [4 ,0] TO 1
  SET maze [4 ,3] TO 3
  SET maze [5 ,0] TO 2
  SET maze [6 ,0] TO 3
  SET maze [6 ,1] TO 7
  SET maze [7 ,1] TO 8
  SET maze [7 ,3] TO 6
  SET maze [8 ,3] TO 7

END FUNCTION

PROCEDURE ChangeRoom (REF room, direction)
  IF maze[room , direction = -1 THEN
    SEND "You have hit a wall" TO DISPLAY
  ELSE
    newRoom = maze[room , direction]
    SEND "You are now in room "& newRoom
  END IF
  room = newRoom
END PROCEDURE

FUNCTION getDirection()
  SET valid TO FALSE
  DO
    RECEIVE direction FROM KEYBOARD
    IF direction >= 0 AND direction <=3 THEN
      SET valid TO TRUE
    ELSE
      DISPLAY "Please try again, enter 0, 1, 2 or 3"
    END IF
  WHILE direction < 0 OR direction > 3
  RETURN direction
END FUNCTION

DECLARE maze[9,4] AS INTEGER
maze = Setup(maze)
```

```
52 room = 0
53 DO
54   SEND "You are in room " & room TO DISPLAY
55   SEND "Enter 0 to go North" TO DISPLAY
56   SEND "Enter 1 to go East" TO DISPLAY
57   SEND "Enter 2 to go South" TO DISPLAY
58   SEND "Enter 3 to go West" TO DISPLAY
59   direction = getDirection()
60   ChangeRoom(room, direction)
61 LOOP UNTIL room = 8
```

Music player: array of objects (page 76)

Q13: Possible answer:

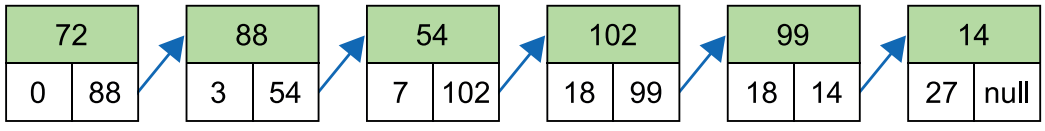
```
CLASS Track IS {STRING title, STRING artist}

METHODS

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

END CLASS

CLASS Collection IS {ARRAY OF Track tracks}

METHODS

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

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

END CLASS

PROCEDURE Setup()
  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"}
END PROCEDURE

DECLARE sixtiesTracks INITIALLY Collection(10)
Setup(sixtiesTracks)
sixtiesTracks.showTracks
```

Linked lists (page 92)

Q14: c) a linked list can increase and decrease in size as nodes are added or removed but an array is declared with a fixed size.

Q15: a) Points to the start of the list (the first node).

Q16: c)



Execute and display results (page 104)

Q17: Possible answer:

```
<?php
  $host="localhost";
  $username="root";
  $pword="root";
  $database="esports";

  require("db_connection.php");

?>
<html>
  <head>
    <title>Database Query Test</title>
    <style>
      table, td, th {border: 1px solid black;}
    </style>
  </head>
  <body>
  <?php
    //set up a query to use
    $query_string = "SELECT * FROM game";
    ?>
    <table>
      <thead>
        <th class="outline">Game</th><th>Platform</th><th> </th><th>
            </th>
      </thead>
      <tbody>
    <?php

    if ($result = $mysqli->query($query_string)) {  //run the query

      while ( $row = $result_fetch_assoc( $result) {
        echo "<tr ><td >" . $row['game ']. "</td >";
        echo "<td >" . $row[' platform '] . "</td >";
        echo "<td >" . $row[' playername '] . "</td >";
        echo "<td >" . $row['score '] . "</td ></tr >";
      }

      $result->close();  //free the $result set (clear it)
    }

    $mysqli->close();
    ?>
      </tbody>
    </table>
    <a href="add-game-row.php">Insert a new row</a>
  </body>
</html>
```

Bubble sort (page 111)

Q18: Possible answer:

```
PROCEDURE bubble_sort(list)
  DECLARE n INITIALLY length(list)
  DECLARE swapped INITIALLY TRUE
  WHILE swapped AND n >= 0
    SET swapped TO False
    FOR i = 0 to n-2 DO
      IF list[i] > list[i+1] THEN
        SET temp TO list[i]
        SET list[i] TO list[i+1]
        SET list[i+1] TO temp
        SET swapped TO TRUE
      END IF
    END FOR
    SET n TO n - 1
  END WHILE
END PROCEDURE

PROCEDURE display_list(list)
  DECLARE n INITIALLY length(list)
  FOR i = 0 to n DO
    SEND list[i] TO DISPLAY
  END FOR
END PROCEDURE

DECLARE List AS ARRAY OF INTEGER INITIALLY []*6
List = [ 5, 1, 4, 2, 8, 9 ]
bubble_sort(List)
display_list(List)
```

Q19: Possible answer:

```
PROCEDURE fill_list(list)
  DECLARE n INITIALLY length(list)
  FOR i = 0 to n DO
    SET list[i] = <random number between 1 and 99>
  END FOR
END PROCEDURE

PROCEDURE bubble_sort(list)
  DECLARE n INITIALLY length(list)
  DECLARE swapped INITIALLY TRUE
  WHILE swapped AND n >= 0
    SET swapped TO False
    FOR i = 0 to n-2 DO
      IF list[i] > list[i+1] THEN
        SET temp TO list[i]
        SET list[i] TO list[i+1]
        SET list[i+1] TO temp
        SET swapped TO TRUE
      END IF
    END FOR
    SET n TO n - 1
  END WHILE
END PROCEDURE

PROCEDURE display_list(list)
  DECLARE n INITIALLY length(list)
  FOR i = 0 to n DO
    SEND list[i] TO DISPLAY
  END FOR
END PROCEDURE

DECLARE List AS ARRAY OF INTEGER INITIALLY []*20
fill_list(List)
bubble_sort(List)
display_list(List)
```

Insertion sort (page 116)

Q20: Possible answer:

```
PROCEDURE insertion_sort(list)
  DECLARE value INITIALLY 0
  DECLARE index INITIALLY 0
  FOR i = 1 to length(list)-1 DO
    SET value TO list[i]
    SET index TO i
    WHILE (index > 0) AND (value < list[index-1]) DO
      SET list[index] TO list[index-1]
      SET index TO index - 1
    END WHILE
    SET list[index] TO value
  END FOR
END PROCEDURE

PROCEDURE display_list(list)
  DECLARE n INITIALLY length(list)
  FOR i = 0 to n DO
    SEND list[i] TO DISPLAY
  END FOR
END PROCEDURE

DECLARE List AS ARRAY OF INTEGER INITIALLY []*6
List = [ 5, 1, 4, 2, 8, 9 ]
insertion_sort(List)
display_list(List)
```

Q21: Possible answer:

```
PROCEDURE fill_list(list)
  DECLARE n INITIALLY length(list)
  FOR i = 0 to n DO
    SET list[i] = <random number between 1 and 99>
  END FOR
END PROCEDURE

PROCEDURE insertion_sort(list)
  DECLARE value INITIALLY 0
  DECLARE index INITIALLY 0
  FOR i = 1 to length(list)-1 DO
    SET value TO list[i]
    SET index TO i
    WHILE (index > 0) AND (value < list[index-1]) DO
      SET list[index] TO list[index-1]
      SET index TO index - 1
    END WHILE
    SET list[index] TO value
  END FOR
END PROCEDURE

PROCEDURE display_list(list)
  DECLARE n INITIALLY length(list)
  FOR i = 0 to n DO
    SEND list[i] TO DISPLAY
  END FOR
END PROCEDURE

DECLARE List AS ARRAY OF INTEGER INITIALLY []*20
fill_list(List)
insertion_sort(List)
display_list(List)
```

Binary search (page 119)

Q22: Possible answer:

```
PROCEDURE display_list(list)
  DECLARE n INITIALLY length(list)
  FOR i = 0 to n DO
    SEND list[i] TO DISPLAY
  END FOR
END PROCEDURE

PROCEDURE binary_search(list,target)
  DECLARE low INITIALLY 0
  DECLARE high INITIALLY length(list)-1
  DECLARE mid INITIALLY 0
  DECLARE found INITIALLY FALSE
  WHILE NOT found AND low <= high
    SET mid TO (low+high)/2
    IF target = list[mid] THEN
      SEND "Found at position " & mid TO DISPLAY
      SET found TO TRUE
    ELSE IF target > list[mid] THEN
      SET low TO mid+1
    ELSE
      SET high TO mid-1
    END IF
  END WHILE
  IF found = FALSE THEN
    SEND "Target not found" TO DISPLAY
  END IF
END PROCEDURE

DECLARE List AS ARRAY OF INTEGER INITIALLY []*10
List = [ 2, 5, 8, 12, 16, 27, 38, 58, 81, 91 ]
display_list(List)
binary_search(List, 27)
```

Q23: Possible answer:

```
1  PROCEDURE fill_list(list)
2    DECLARE n INITIALLY length(list)
3    FOR i = 0 to n DO
4      SET list[i] = <random number between 1 and 99>
5    END FOR
6  END PROCEDURE
7
8  PROCEDURE insertion_sort(list)
9    DECLARE value INITIALLY 0
10   DECLARE index INITIALLY 0
11   FOR i = 1 to length(list)-1 DO
12     SET value TO list[i]
13     SET index TO i
14     WHILE (index > 0) AND (value < list[index-1]) DO
15       SET list[index] TO list[index-1]
16       SET index TO index - 1
17     END WHILE
18     SET list[index] TO value
19   END FOR
20 END PROCEDURE
21
22 PROCEDURE display_list(list)
23   DECLARE n INITIALLY length(list)
24   FOR i = 0 to n DO
25     SEND list[i] TO DISPLAY
26   END FOR
27 END PROCEDURE
28
29 PROCEDURE binary_search(list,target)
30   DECLARE low INITIALLY 0
31   DECLARE high INITIALLY length(list)-1
32   DECLARE mid INITIALLY 0
33   DECLARE found INITIALLY FALSE
34   WHILE NOT found AND low <= high
35     SET mid TO (low+high)/2
36     IF target = list[mid] THEN
37       SEND "Found at position " & mid TO DISPLAY
38       SET found TO TRUE
39     ELSE IF target > list[mid] THEN
40       SET low TO mid+1
41     ELSE
42       SET high TO mid-1
43     END IF
44   END WHILE
45   IF found = FALSE THEN
46     SEND "Target not found" TO DISPLAY
47   END IF
48 END PROCEDURE
49
50 DECLARE List AS ARRAY OF INTEGER INITIALLY []*20
51 fill_list(List)
52 insertion_sort(List)
53 RECEIVE target FROM KEYBOARD
```

```
54 display_list(List)
55 binary_search(List,target)
```

End of topic test: Implementation (page 121)

Q24: b) Inheritance, polymorphism, encapsulation.

Q25: c) `currentUser.updateEmail("scholar@hw.com")`

Q26: a) encapsulation.

Q27: b) `basket[6].title`

Q28: b) `DECLARE myArray AS ARRAY of ARRAY OF REAL`

Q29: b) head.

Q30: d) the list can be traversed in either direction.

Q31: c) Username, server, password, database.

Q32: d) Insertion sort

Q33: c)

```
1 SET temp TO list[i]
2 SET list[i] TO list[i+1]
3 SET list[i+1] TO temp
4 SET swapped TO TRUE
```

Topic 4: Testing and evaluation

Quiz: Revision (page 127)

Q1:

minimumValue	counter	numbers[counter]
17	1	15
15	2	4
4	3	7
4	4	8

Output: The smallest value was 4.

Q2: c) extreme data.

Q3: c) allow program execution to be halted for debugging at a particular point.

Q4: d) test that individual modules of code function as expected.

End of topic test: Testing and evaluation (page 141)

Q5: b) The test data used.

Q6:

b) Documentation will aid anyone who needs to revisit a previous stage in the light of subsequent previously undetected problems.

c) Documentation will be invaluable to those testing subsequent stages.

Q7: a) A dummy function or procedure.

Q8: b) Code designed to test a function or procedure.

Q9:

c) Clients

d) Beta testers

Q10: c) Final testing

Q11: a) There should always be an option to escape from a crucial operation.

Q12: c) contains modular code which effectively reuses modules.

Q13: b) To explain the function of the code for maintenance.

Q14: d) it provides feedback and the opportunity to recover if an error is encountered during execution.

Topic 5: Software design and development test

Software design and development test (page 146)

Q1:

a) Maintainability is improved by having readable code.

c) Maintainability is improved through having effective internal commentary.

Q2: a) encapsulation

Q3: a) a class

Q4: a) inheritance

Q5: c) Server name / location, username, password, database table.

Q6: d) Variables, declared outside the methods of the class, that are allocated memory when the class is instantiated, and which record the state of the instantiated object.

Q7: d) Constructor

Q8: b) Once an object is created based on this class, the ***variety*** and ***yield*** instance variables cannot be updated.

Q9: b) 3

Q10: c) A working model designed for the purpose of evaluation by end users.

Q11: d) A book must have one or more authors and each author must have one or more books.

Q12: a) Creates a user account based on the details entered in this page.

Q13: c) `CONSTRUCTOR (DATATYPE property, ...)`

Q14: b) Polymorphism

Q15: b)

```
1 RECORD waitingRow IS {STRING pupilFname, STRING pupilLname, STRING
     parentFname, STRING parentLname, STRING phoneNumber}
2 DECLARE waitingList AS ARRAY of waitingRow INITIALLY [] * 50
```

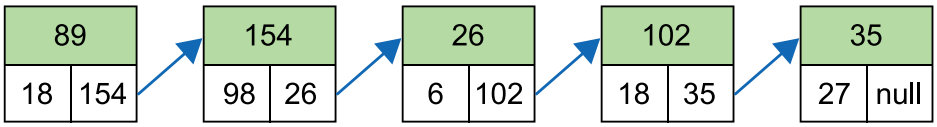
Q16: d) Insertion sort

Q17: b) Parallel 1D arrays

Q18: a) stub.

Q19: b) driver.

Q20: b)



Q21: c) To be stored in the next or previous pointers at the beginning/end of the list.

Q22: d) Being able to traverse the list in both directions.

Q23: d)

```
Line 12 SET low TO mid+1
Line 14 SET high TO mid-1
```

Q24: c) To detect if a swap has not occurred allowing the loop to terminate when the list is fully sorted.

Q25: c) 5

Q26: b) `DECLARE myArray AS ARRAY of ARRAY OF REAL`

Q27:

c) Clients

d) Beta testers

Q28: c) Final testing

Q29: b) To explain the function of the code for maintenance.

Q30: a) Code is reused and modules used to reduce the amount and complexity of code. It makes good use of system resources.