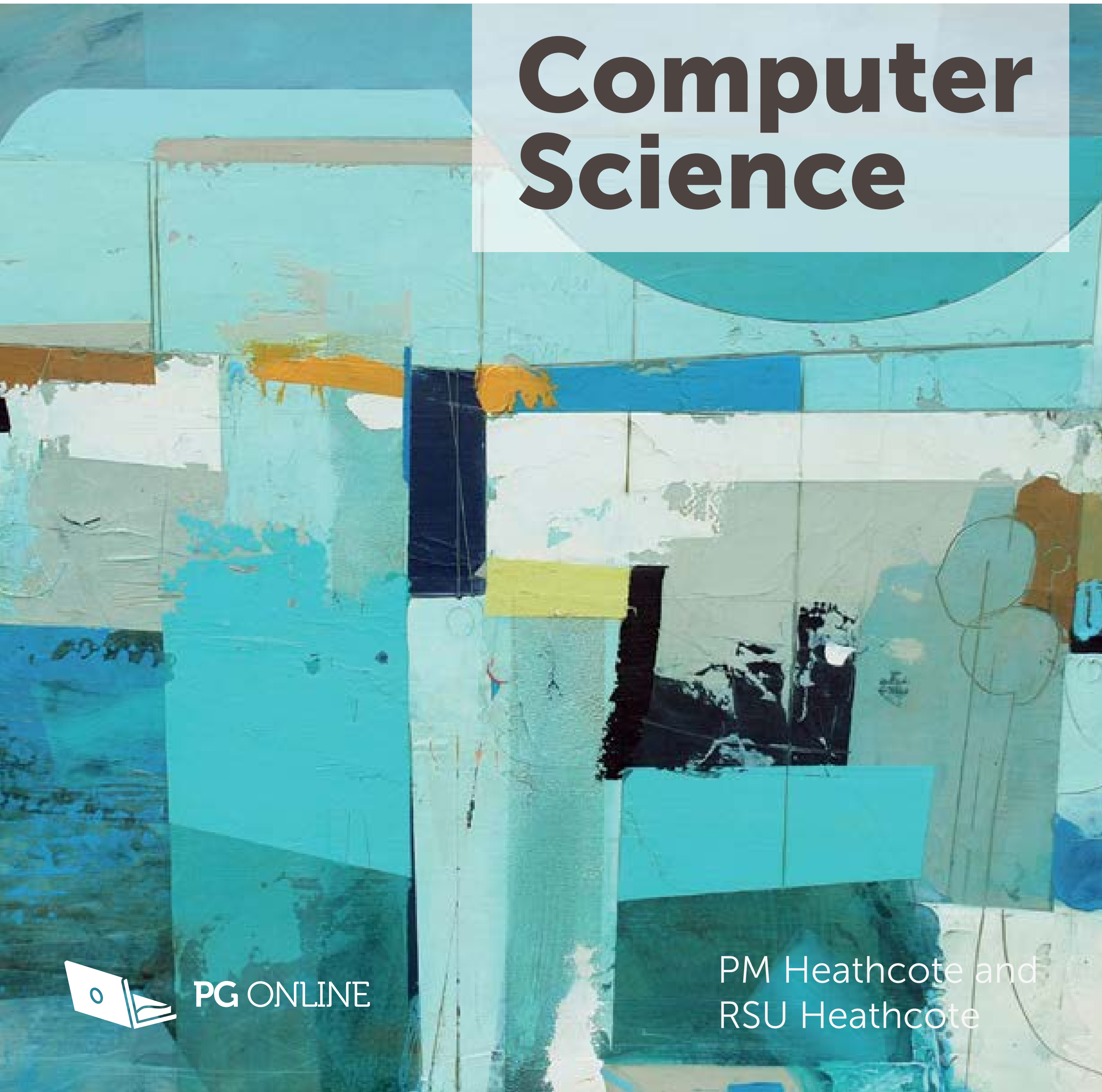
AQA Approved

AQA A Level Year 2 Computer Science



PM Heathcote and
RSU Heathcote

PG ONLINE

AQA A Level Year 2 Computer Science

P.M. Heathcote

R.S.U. Heathcote

Published by
PG Online Limited
The Old Coach House
35 Main Road
Tolpuddle
Dorset
DT2 7EW
United Kingdom
sales@pgonline.co.uk
www.pgonline.co.uk
2016

PG ONLINE

Acknowledgements

We are grateful to the AQA Examination Board for permission to use questions from past papers.

The answers in the Teacher's Supplement are the sole responsibility of the authors and have neither been provided nor approved by the examination board.

We would also like to thank the following for permission to reproduce copyright photographs:

Screenshot from Roboform website © Roboform
Alan Turing © By kind permission of the Provost and Fellows, King's College, Cambridge
from Archives Centre, King's College, Cambridge. AMT/K/7/12
Trans-continental Internet connections © Telegeography
Internet registries map © Ripe NCC
Other photographic images © Shutterstock

Graphics: Rob Heathcote

Cover picture © 'Blue Day'
Acrylic on board
Reproduced with the kind permission of Andrew Bird
www.abirdart.co.uk

Design and artwork by OnThree
www.on-three.com

Typeset by Ian Kingston

First edition 2016, reprinted 2016

A catalogue entry for this book is available from the British Library

ISBN: 978-1-910523-03-2

Printed and bound in Great Britain by Lightning Source Inc., Milton Keynes

Preface

The aim of this textbook is to provide detailed coverage of the topics in the new AQA A Level Computer Science specification.

The book is divided into six sections and within each section, each chapter covers material that can comfortably be taught in one or two lessons.

In the second year of this course there is a strong emphasis on algorithms and data structures, and these are covered in the first two sections of the book. These are followed by sections on regular languages, the Internet and databases.

Object Oriented Programming and functional programming are covered in the final section, which describes basic theoretical concepts in OOP, as well as providing some practical exercises using the functional programming language Haskell. Lists, the fact-based model and 'Big Data' are all described and explained.

Two short appendices contain A Level content that could be taught in the first year of the course as an extension to related AS topics.

The OOP concepts covered may also be helpful in the coursework element of the A Level course.

Each chapter contains exercises and questions, some new and some from past examination papers. Answers to all these are available to teachers only in a Teacher's Supplement which can be ordered from our website **www.pgonline.co.uk**.

Approval message from AQA

This textbook has been approved by AQA for use with our qualification. This means that we have checked that it broadly covers the specification and we are satisfied with the overall quality. Full details of our approval process can be found on our website.

We approve textbooks because we know how important it is for teachers and students to have the right resources to support their teaching and learning. However, the publisher is ultimately responsible for the editorial control and quality of this book.

Please note that when teaching the A Level Computer Science course, you must refer to AQA's specification as your definitive source of information. While this book has been written to match the specification, it cannot provide complete coverage of every aspect of the course.

A wide range of other useful resources can be found on the relevant subject pages of our website: www.aqa.org.uk.

Contents

Chapter 41 – Graphs

Objectives

- Be aware of a graph as a data structure used to represent complex relationships
- Be familiar with typical uses for graphs
- Be able to explain the terms: graph, weighted graph, vertex/node, edge/arc, undirected graph, directed graph
- Know how an adjacency matrix and an adjacency list may be used to represent a graph
- Be able to compare the use of adjacency matrices and adjacency lists

Definition of a graph

A graph is a set of **vertices** or **nodes** connected by **edges** or **arcs**. The edges may be one-way or two way. In an **undirected graph**, all edges are bidirectional. If the edges in a graph are all one-way, the graph is said to be a **directed graph** or **digraph.**

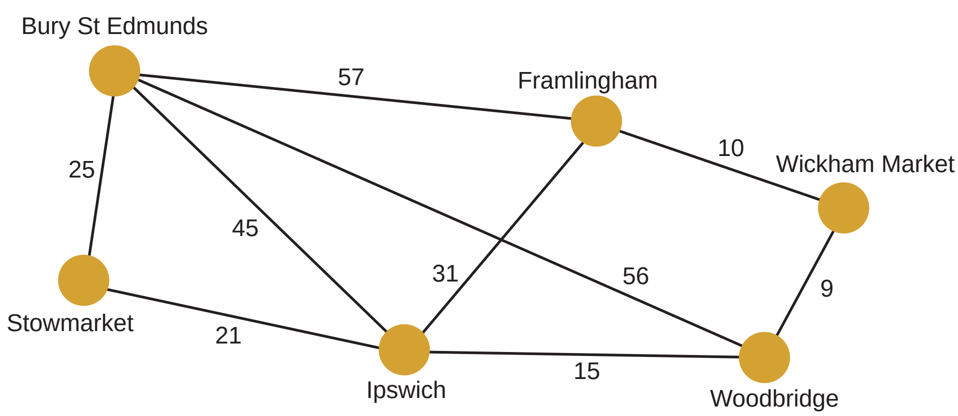


Figure 41.1: An undirected graph with weighted edges

The edges may be **weighted** to show there is a cost to go from one vertex to another as in Figure 41.1. The weights in this example represent distances between towns. A human driver can find their way from one town to another by following a map, but a computer needs to represent the information about distances and connections in a structured, numerical representation.

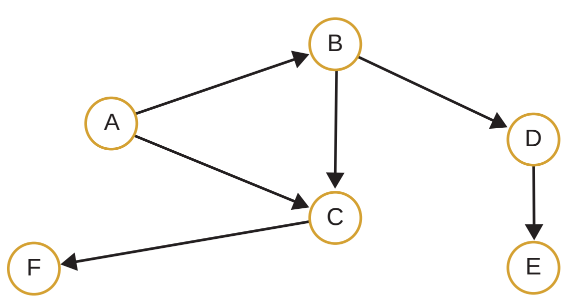


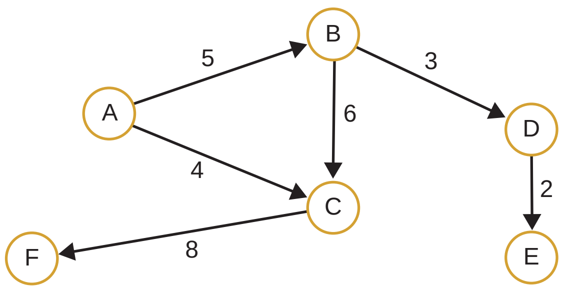
Figure 41.2: A directed, unweighted graph

Implementing a graph

Two possible implementations of a graph are the **adjacency matrix** and the **adjacency list**.

The adjacency matrix

A two-dimensional array can be used to store information about a directed or undirected graph. Each of the rows and columns represents a node, and a value stored in the cell at the intersection of row i, column j indicates that there is an edge connecting node i and node j.



	A	B	C	D	E	F
A		5	4			
B			6	3		
C						8
D					2	
E						
F						

In the case of an **undirected graph**, the adjacency matrix will be symmetric, with the same entry in row 0 column 1 as in row 1 column 0, for example.

An unweighted graph may be represented with 1s instead of weights, in the relevant cells.

Q1: Draw an adjacency matrix to represent the weighted graph shown in Figure 41.1.

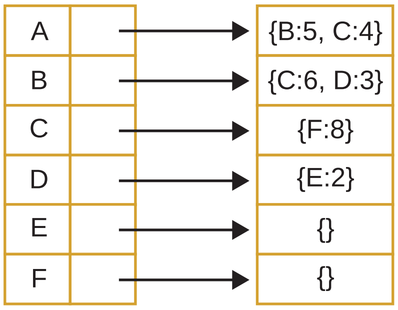
Advantages and disadvantages of the adjacency matrix

An adjacency matrix is very convenient to work with, and adding an edge or testing for the presence of an edge is very simple and quick. However, a sparse graph with many nodes but not many edges will leave most of the cells empty, and the larger the graph, the more memory space will be wasted. Another consideration is that using a static two-dimensional array, it is harder to add or delete nodes.

The adjacency list

An adjacency list is a more space-efficient way to implement a sparsely connected graph. A list of all the nodes is created, and each node points to a list of all the adjacent nodes to which it is directly linked. The adjacency list can be implemented as a list of dictionaries, with the key in each dictionary being the node and the value, the edge weight.

The graph above would be represented as follows:



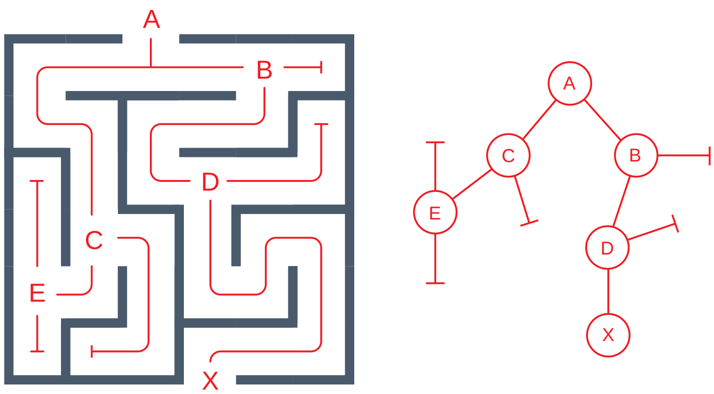
Applications of depth-first search

Applications of the depth-first search include the following:

- In scheduling jobs where a series of tasks is to be performed, and certain tasks must be completed before the next one begins.
- In solving problems such as mazes, which can be represented as a graph

Finding a way through a maze

A depth-first search can be used to find a way out of a maze. Junctions where there is a choice of route in the maze are represented as nodes on a graph.



8-47

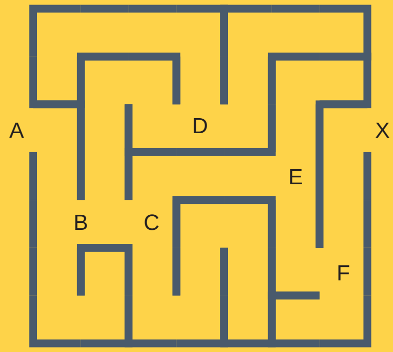
Q1: (a) Redraw the graph without showing the dead ends.

(b) State the properties of this graph that makes it a tree.

(c) Complete the table below to show how the graph would be represented using an adjacency matrix.

	A	B	C	D	E	X
A						
B						
C						
D						
E						
X						

Q2: Draw a graph representing the following maze. Show the dead ends on your graph.



Chapter 53 – The Turing machine

Objectives

- Know that a Turing machine can be viewed as a computer with a single fixed program, expressed using
    - a finite set of states in a state transition diagram
    - a finite alphabet of symbols
    - an infinite tape with marked off squares
    - a sensing read-write head that can travel along the tape, one square at a time
- Understand the equivalence between a transition function and a state transition diagram
- Be able to:
    - represent transition rules using a transition function
    - represent transition rules using a state transition diagram
    - hand-trace simple Turing machines
- Explain the importance of Turing machines and the Universal Turing machine to the subject of computation

Alan Turing

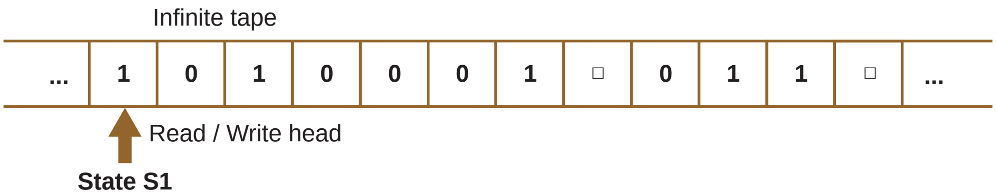
Alan Turing (1912–1954) was a British computer scientist and mathematician, best known for his work at Bletchley Park during the Second World War. While working there, he devised an early computer for breaking German ciphers, work which probably shortened the war by two or more years and saved countless lives.

Turing was interested in the question of **computability**, and the answer to the question "Is every mathematical task computable?" In 1936 he invented a theoretical machine, which became known as the **Turing machine**, to answer this question.

The Turing machine

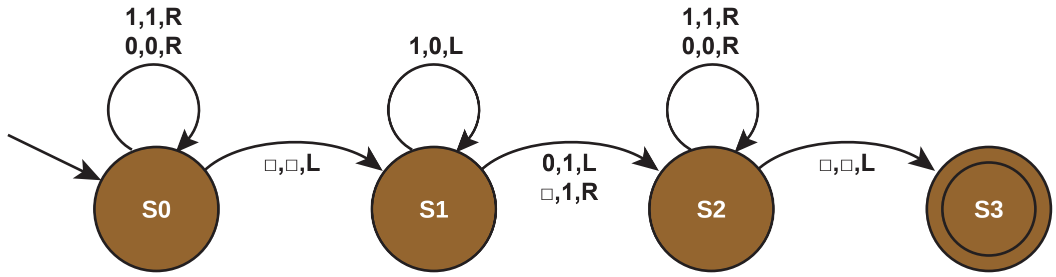
The Turing machine consists of an infinitely long strip of tape divided into squares. It has a read/write head that can read symbols from the tape and make decisions about what to do based on the contents of the cell and its current state.

Essentially, this is a finite state machine with the addition of an infinite memory on tape. The FSM specifies the task to be performed; it can erase or write a different symbol in the current cell, and it can move the read/write head either left or right.

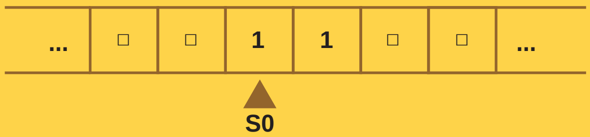


The Turing machine is an early precursor of the modern computer, with input, output and a program which describes its behaviour. Any alphabet may be defined for the Turing machine; for example a binary alphabet of 0, 1 and □ (representing a blank), as shown in the diagram above.

The finite state machine corresponding to the state transition diagram is given below.



Q1: Trace the computation of the Turing machine if the tape starts with the data 11 as shown below.



(You will need to draw ten representations of the tape to complete the computation.)

Transition functions

The transition rules for any Turing machine can be expressed as a **transition function** δ. The rules are written in the form

δ (Current State, Input symbol) = (Next State, Output symbol, Movement).

Thus the rule

δ (S1, 0) = (S2, 1, L)

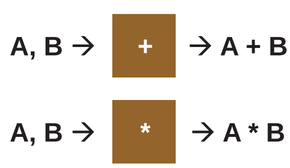
means "IF the machine is currently in state S1 and the input symbol read from the tape is 0, THEN write a 1 to the tape, and move left and change state to S2".

Q2: Looking at the state transition diagram above, write the transition rules for inputs of 0, 1 and □ when the machine is in state S0.

9-53

The universal Turing machine

A Turing machine can theoretically represent any computation.



Each machine has a different program to compute the desired operation. However, the obvious problem with this is that a different machine has to be created for each operation, which is clearly impractical.

Turing therefore came up with the idea of the **Universal Turing machine**, which could be used to compute any computable sequence. He wrote: "If this machine **U** is supplied with the tape on the beginning of which is written the string of quintuples separated by semicolons of some computing machine **M**, then **U** will compute the same sequence as **M**."

Chapter 68 – Object-oriented design principles

Objectives

- Understand concepts of association, composition and aggregation
- Understand the use of polymorphism and overriding
- Be aware of object-oriented design principles:
    - encapsulate what varies
    - favour composition over inheritance
    - program to interfaces, not implementation
- Be able to draw and interpret class diagrams

Association, aggregation and composition

Recall that inheritance is based on an "is a" relationship between two classes. For example, a cat "is a(n)" animal, a car "is a" vehicle. In a similar fashion, **association** may be loosely described as a "**has a**" relationship between classes. Thus a railway company may be associated with the engines and carriages it owns, or the track that it maintains. A teacher may be associated with a form bi-directionally – a teacher "has a" student, and a student "has a" teacher. However, there is no **ownership** between objects and each has their own lifecycle, and can be created and deleted independently.

Association aggregation, or simply **aggregation**, is a special type of more specific association. It can occur when a class is a collection or container of other classes, but the contained classes do not have a strong lifecycle dependency on the container. For example, a player who is part of a team does not cease to exist if the team is disbanded.

12-68

Aggregation may be shown in class diagrams using a hollow diamond shape between the two classes.

Class diagram showing association aggregation

Composition aggregation, or simply **composition**, is a stronger form of aggregation. If the container is destroyed, every instance of the contained class is also destroyed. For example if a hotel is destroyed, every room in the hotel is destroyed.

Composition may be shown in class diagrams using a filled diamond shape. The diamond is at the end of the class that owns the creational responsibility.

Class diagram showing composition aggregation

Q1: Specify whether each of the following describe **association aggregation** or **composition aggregation**.

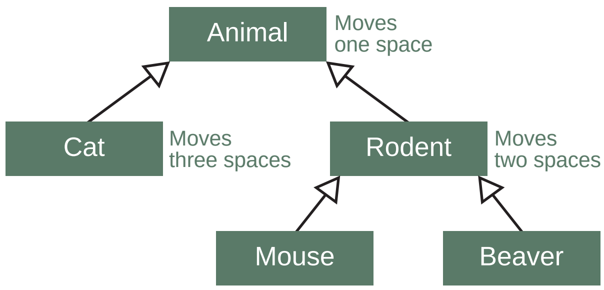
(a) Zoo and ZooAnimal

(b) RaceTrack and TrackSection

(c) Department and Teacher

Polymorphism

Polymorphism refers to a programming language's ability to process objects differently depending on their class. For example, in the last chapter we looked at an application that had a superclass `Animal`, and subclasses `Cat` and `Rodent`. All objects in subclasses of `Animal` can execute the methods `moveLeft`, `moveRight`, which will cause the animal to move one space left or right.



We might decide that a `cat` should move three spaces when a `moveLeft` or `moveRight` message is received, and a `Rodent` should move two spaces. We can define different methods within each of the classes to implement these moves, but keep the same method name for each class.

Defining a method with the same name and formal argument types as a method inherited from a superclass is called **overriding**. In the example above, the `moveLeft` method in each of the `Cat` and `Rodent` classes overrides the method in the superclass `Animal`.

12-68

Q2: Suppose that `tom` is an instance of the `Cat` class, and `jerry` is an instance of the `Mouse` class. What will happen when each of these statements is executed?

tom.moveRight()

jerry.moveRight()

Q3: Looking at the diagram above, what changes do you need to make so that `bertie`, an instance of the `Beaver` class, moves only one space when given a `moveRight()` message?

Class definition including override

Class definitions for the classes `Animal` and `Cat` will be something like this:

```
Animal = Class
        Public
           Procedure moveLeft
           Procedure moveRight
        Protected
           Position: Integer
        End
Cat = Subclass (Animal)
        Public
           Procedure moveLeft (Override)
           Procedure moveRight (Override)
           Procedure pounce
        Private
           Name: String
        End
```

Note: The 'Protected' access modifier is described on page 356.

Index

Index

AQA A Level **Year 2**

Computer Science

The aim of this textbook is to provide a detailed understanding of each topic in the second year of the new AQA A Level Computer Science specification. It is presented in an accessible and interesting way, with many in-text questions to test students' understanding of the material and ability to apply it.

The book is divided into six sections, each containing roughly six chapters. Each chapter covers material that can comfortably be taught in one or two lessons. It will also be a useful reference and revision guide for students throughout the A Level course. Two short appendices contain A Level content that could be taught in the first year of the course as an extension to related AS topics.

Each chapter contains exercises, some new and some from past examination papers, which can be set as homework. Answers to all these are available to teachers only, in a Teachers Supplement which can be ordered from our website **www.pgonline.co.uk**

About the authors

Pat Heathcote is a wellknown and successful author of Computing textbooks. She has spent many years as a teacher of A Level Computing courses with significant examining experience. She has also worked as a programmer and systems analyst, and was Managing Director of Payne-Gallway Publishers until 2005.

Rob Heathcote has many years of experience teaching Computer Science and is the author of several popular textbooks on Computing. He is now Managing Director of PG Online, and writes and edits a substantial number of the online teaching materials published by the company.

Cover picture:

'Blue Day'
Acrylic on board 51x61cm
© Andrew Bird
www.abirdart.co.uk

This book has been approved by AQA.

PG ONLINE

ISBN 978-1-910523-03-2
90000
9 781910 523032