University of Brighton

School of Computing, Engineering and Mathematics

Assessment Brief Form

Module Title:	Introduction to programming
Module Code:	CI401
Author(s)/Marker(s) of Assignment	Dr Roger Evans

Assignment No:	1
Assignment Title:	Coursework: Simple Application
Percentage contribution to module mark:	50%
Weighting of component assessments within this assignment:	N/A
Module Learning Outcome/s Covered: (Refer to module syllabus)	**LO1** Use sequence, selection and iteration to develop simple applications **LO2** Utilise a variety of data types and collections **LO3** Detect and correct errors in both logic and syntax **LO4** Test and debug simple programs **LO5** Demonstrate a practical understanding of Object Oriented programming techniques

Assignment Brief and Assessment Criteria:
See slide-set below

Date of issue:	8th February 2021
Deadline for submission:	30th April 2021 15:00
Method of submission:	Online
Date feedback will be provided	2nd June 2021

1. *A copy of your coursework submission may be made as part of the University of Brighton's and School of Computing, Engineering & Mathematics procedures which aim to monitor and improve quality of teaching. You should refer to your student handbook for details.*
2. *All work submitted must be your own (or your team's for an assignment which has been specified as a group submission) and all sources which do not fall into that category must be correctly attributed. The markers may submit the whole set of submissions to the JISC Plagiarism Detection Service.*

2020 CI401
Introduction to programming

The coursework assessment

Semester 2 2020/2021

Roger Evans

Module leader

Introduction

- The coursework assessment task requires you to submit some code and documentation to demonstrate your learning in CI401
- We provide starter projects which you can use as a baseline, adding your own extensions and documentation. Or you can create a project yourself.
- This slide set provides information about the task including:
  - The formal specification of the assessment task (from the Module Specification)
  - Projects - choice of starter projects, or your own topic
  - How your project will be assessed – marking scheme and grading etc.
  - Details of the submission process

The Module Specification

The formal description of this assessment task is in the Module Specification document as follows:

Task 1 (50%) Simple Application (LO 1,2,3,4,5).

- As part of their guided independent study and weekly lab classes, students will create a simple application.
- This coursework requires each student to submit their application and a technical report which explains how their code works.
- (17.5 hours)

Learning outcomes assessed

- LO1: Use sequence, selection and iteration to develop simple applications
- LO2: Utilise a variety of data types and collections
- LO3: Detect and correct errors in both logic and syntax
- LO4: Test and debug simple programs
- LO5: Demonstrate a practical understanding of Object Oriented programming techniques

Projects

Projects

- You will have a choice of starter projects or you can create a project from scratch
- You only need to do ONE of these options for your assessment!

- For the starter projects, you are provided with a set of initial classes that will run the application
- Your task is to add some additional features to your chosen project.
- We will work with these projects during the term, to learn more about them, and some new programming techniques you can use in your assessment

Help with project work

- Week 2.01-2.03 – Starter projects provided with some parts ‘missing’
- Weeks 2.01-2.04 – Lab work is to complete these missing parts, with the help of tutors if required (not assessed)
- Week 2.04 – new versions of projects available with solutions for these lab parts included (if you need them). You will be assessed on what else you add beyond these solutions
- Week 2.05-2.09 - work independently on project in lab sessions, but tutors can only give you general guidance. Additional lab examples of new techniques (inheritance, encapsulation, documentation, testing, collections) will also be provided which we can help you work through in more detail

General enhancements

- As you develop your projects, we will cover additional topics in lectures and labs that you can incorporate into your solutions
- These will include
  - Using simple inheritance
  - Encapsulation and variable scope
  - Testing (with Junit)
  - Documenting code using Javadoc
  - ‘Collection’ types
  - Files, input and output

Project preparation timeline

Week	Date (w/b)	Project	Related topic/lab exercise
2.01	08-Feb-2021	Starter projects and topics announced	Introduction to coursework
2.02	15-Feb-2021	Supported project lab work	Simple Inheritance
2.03	22-Feb-2021	Supported project lab work	Scope, Visibility and Encapsulation
2.04	01-Mar-2021	Assessment baseline starter projects	Testing - JUnit
2.05	08-Mar-2021	Independent project work	Documentation - Javadoc
2.06	15-Mar-2021	Independent project work	Collections and generic types
2.07	22-Mar-2021	Independent project work	IO: Files and Streams
2.08	19-Apr-2021	Independent project work	
2.09	26-Apr-2021	Independent project work	
		Project deadline – 30-April-2021, 3pm	

Choosing a project

- The starter projects have different elements, some more challenging than others, some more interesting than others
- You will be assessed on your general coding skills, not on how well you enhance the particular project you choose
- So you don't have to choose the 'hardest' one – you can get good marks doing any of them well, and pass quite comfortably doing just the basic things
- Look at the marking scheme to understand what you get marks for, and focus on those features.

Choosing your own additions

- We will provide some ideas for extensions, but these are only suggestions.
- You don't have to do them exactly, and you certainly don't have to do them all!
- You can make your own additions, as long as they are broadly within the theme of one of the projects
- Remember the important thing is to show what you can do according to the marking criteria (which are general, not project-specific)
- If you are not sure, ask for advice about an addition.

Project topics

Starter project - Breakout

- Breakout is a classic computer arcade game
- Here's a video to remind you:
  - https://www.youtube.com/watch?v=AMUv8KvVt08
- In this project you will be provided with a basic shell, and you need to add features in two stages:
  - Making the game work (the shell does not include everything) Not assessed
  - Adding new features (we provide a list of suggestions) Assessed
- You will also need to document your code, and write a short technical report Assessed

Making Breakout work (not assessed)

- Model class
  - add bricks to the model (the shell has no bricks!) - code point [1]
- View class
  - display the bricks on screen - code point [2]
- Model class
  - if the ball hits a brick, the brick is destroyed – code point [3]

Suggestions for adding features to Breakout (assessed)

- Adding effects like colour (sound?)
- Adding more bricks and brick layouts
- Losing ‘lives’ if the ball goes off the bottom
- Adding more complexity to the game – bricks requiring multiple impacts (changing colour?), increasing speed, multiple balls
- Levels of play
- Competitive play and high score table

Starter project - ATM

- The ATM project simulates a cashpoint (except that it doesn't give you any money ☹)
- As with the Breakout project, you will be provided with a basic shell, and you need to add features in two stages:
  - Making the ATM work (the shell does not include everything) Not assessed
  - Adding new features (we provide a list of suggestions) Assessed
- You will also need to document your code, and write a short technical report Assessed

Making the ATM work (not assessed)

- The ATM interface does all the right things, calling methods in the LocalBank object to log in and work with an account.
- LocalBank has a couple of accounts in it.
- But the methods in LocalBank don't do anything!
- You need to change them so that they do.

- The code in LocalBank includes comments to tell you where you need to make changes

Suggestions for adding features to ATM (assessed)

- Adding effects like colour and CSS styling
- Add the ability to change your password
- Offer to check the balance before withdrawing money
- Restyle the GUI to be more like modern ATMs (eg options displayed on screen, and generic buttons to select them etc)
- Add a mini-statement option
- Add different account types with different functions (eg overdrafts)
- Connect the ATM to a real database (advanced - discuss in labs)
- ‘Encrypting’ the connection to the bank (advanced - discuss in labs)

Starter project - Maze game

- This is a slightly more advanced starter project based on a maze exploration game, searching for treasure.
- It will not be supported so much in the lectures, but we are happy to help in the labs
- The initial lab exercise will ask you to add some basic game functionality (a ghost chasing you through the maze) Not assessed
- You are then free to add more interesting extensions (suggestions will be provided) Assessed
- You will also need to document your code, and write a short technical report Assessed

Your own project

- You can choose a project of your own if you wish – simple games are always good, but other applications are fine too
- Things to think about
  - It does not need to be too big/complex – the starter project all have at most 5-6 classes and that's really all you need
  - Make sure you understand the marking scheme, and focus on a project which lets you show off your skills in all the criteria
  - Use an extra page in your report to give a more thorough explanation of what you have done. This is important – you must show that the code is your own and that you understand it.
- Make sure you discuss it with a tutor in labs by week 2.05, to confirm that it is a suitable idea

Assessment

Project marking criteria

	Learning outcome	Shorthand	Marking criterion	%
LO1	Use sequence, selection and iteration to develop simple applications	Coding	Statements, method calls, sequences, if statements, loops	20%
LO2	Utilise a variety of data types and collections	Data	Variables, types, arrays, classes, collections	20%
LO3	Detect and correct errors in both logic and syntax	Development	Program design and development, detecting and correcting logic and syntax errors	20%
LO4	Test and debug simple programs	Testing	Testing and debugging	10%
LO5	Demonstrate a practical understanding of Object Oriented programming techniques	OO	Inheritance, encapsulation, overloading, overriding	20%
		Documentation	Comments and report	10%

Project grades (approximate)

- 20% – Limited understanding and/or significant errors
- 30% – Some correct examples based on course materials
- 40% (Pass) – Able to use the basic programming techniques and examples learnt in semester 1, mostly correctly
- 60% – Able to use and explain basic techniques, and extend examples provided
- 80%– Able to incorporate more advanced techniques developed in semester 2
- 100% – Able to show clear understanding and application of advanced techniques in new ways

Marking scheme (Rubric)

	100.00	80.00	60.00	40.00	30.00	20.00	0.00
Coding 20% Statements, method calls. sequences, if statements, loops	Full range of control structures used; efficient, clear and robust code.	Method calls, arguments and results; explain control and data flow.	Combine sequences, if statements and loops; explain control flow.	Mostly correct usage of simple sequences, if statements and loops.	Some correct examples of sequences, if statements and/or loops, based on course materials.	Limited understanding of course material examples, and/or significant errors	Minimal evidence/ understanding
Data 20% Variables, types, arrays, classes, collections	Clear understanding of datatypes, effectively used in original coding solutions.	More advanced types (simple inheritance, collection classes), going beyond course examples.	Explain/use basic types and arrays; define and use simple classes.	Mostly correct use of basic types, arrays and variable declarations.	Some correct examples of types, arrays and/or variables based on course materials.	Limited understanding of course material examples, and/or significant errors.	Minimal evidence/ understanding
Development 20% Program design and development, detecting and correcting logic and syntax errors	Clear code design and development. Good understanding of logic and syntax errors, with strategies to minimise and resolve them.	Good code design and development. Able to distinguish logic and syntax errors, and have strategies to fix them.	Describe code design and development, with examples of logic and syntax errors and how you fixed them.	Describe code design, with simple examples of any problems you had to solve	Some evidence of design and detecting and correcting logic and/or syntax errors.	Limited understanding of code design and/or error correction	Minimal evidence/ understanding
Testing 10% Testing and debugging	Effective, appropriate testing , for example using JUnit, and examples of bugs revealed and corrected.	Systematic testing at class level, use of debugging tools, discussion of the benefits of systematic approach.	Describe approach to testing, provide evidence, and discuss bugs found and fixed.	Evidence of testing code, and discussing testing/ debugging process	Some evidence of testing and/or discussion of testing and debugging.	Limited understanding/ evidence of testing and debugging.	Minimal evidence/ understanding
OO 20% Inheritance, encapsulation, overloading, overriding	Clear understanding of key OO techniques, appropriate use in a more advanced task.	Use of more advanced OO techniques in more sophisticated/ modular code for a moderate programming task.	Analyse a simple task using objects, create classes, with variables, constructors and methods, to implement it.	Mostly correct use of classes from course materials, creating objects and calling methods to achieve a simple task.	Some examples using basic OO techniques based on course materials.	Limited understanding of course material examples, and/or significant errors.	Minimal evidence/ understanding
Documentation 10% Comments and report	Comprehensive technical documentation, and Javadoc documentation demonstrating use of Javadoc features.	Clear instructions, detailed technical description, and Javadoc documentation of key aspects, using a range of Javadoc features.	Technical instructions/ description, informative code comments , for example using Javadoc (/** */) comments.	Basic technical description in report and basic end-of-line (//) and/or block (/* */) comments.	Some correct technical descriptions and/or code comments.	Limited documentation, and/or significant errors or misunderstandings.	Minimal evidence/ understanding

How we use the marking scheme

- You get graded for each criterion
- No half marks – just one of these 7 values
- Mark calculated by multiplying column score by row weight, and adding them all together
- For example:
  - 60*20% + 40*20% + 60*20% + 60*10% + 80*20% + 60*10% =
  - 12+8+12+6+16+6 =
  - 60

Pass level – 40%

	100.00	80.00	60.00	40.00	30.00	20.00	0.00
Coding 20% Statements, method calls. sequences, if statements, loops	Full range of control structures used; efficient, clear and robust code.	Method calls, arguments and results; explain control and data flow.	Combine sequences, if statements and loops; explain control flow.	Mostly correct usage of simple sequences, if statements and loops.	Some correct examples of sequences, if statements and/or loops, based on course materials.	Limited understanding of course material examples, and/or significant errors	Minimal evidence/ understanding
Data 20% Variables, types, arrays, classes, collections	Clear understanding of datatypes, effectively used in original coding solutions.	More advanced types (simple inheritance, collection classes), going beyond course examples.	Explain/use basic types and arrays; define and use simple classes.	Mostly correct use of basic types, arrays and variable declarations.	Some correct examples of types, arrays and/or variables based on course materials.	Limited understanding of course material examples, and/or significant errors.	Minimal evidence/ understanding
Development 20% Program design and development, detecting and correcting logic and syntax errors	Clear code design and development. Good understanding of logic and syntax errors, with strategies to minimise and resolve them.	Good code design and development. Able to distinguish logic and syntax errors, and have strategies to fix them.	Describe code design and development, with examples of logic and syntax errors and how you fixed them.	Describe code design, with simple examples of any problems you had to solve	Some evidence of design and detecting and correcting logic and/or syntax errors.	Limited understanding of code design and/or error correction	Minimal evidence/ understanding
Testing 20% Testing and debugging	Effective, appropriate testing , for example using JUnit, and examples of bugs revealed and corrected.	Systematic testing at class level, use of debugging tools, discussion of the benefits of systematic approach.	Describe approach to testing, provide evidence, and discuss bugs found and fixed.	Evidence of testing code, and discussing testing/ debugging process	Some evidence of testing and/or discussion of testing and debugging.	Limited understanding/ evidence of testing and debugging.	Minimal evidence/ understanding
OO 20% Inheritance, encapsulation, overloading, overriding	Clear understanding of key OO techniques, appropriate use in a more advanced task.	Use of more advanced OO techniques in more sophisticated/ modular code for a moderate programming task.	Analyse a simple task using objects, create classes, with variables, constructors and methods, to implement it.	Mostly correct use of classes from course materials, creating objects and calling methods to achieve a simple task.	Some examples using basic OO techniques based on course materials.	Limited understanding of course material examples, and/or significant errors.	Minimal evidence/ understanding
Documentation 10% Comments and report	Comprehensive technical documentation, and Javadoc documentation demonstrating use of Javadoc features.	Clear instructions, detailed technical description, and Javadoc documentation of key aspects, using a range of Javadoc features.	Technical instructions/ description, informative code comments , for example using Javadoc (/** */) comments.	Basic technical description in report and basic end-of-line (//) and/or block (/* */) comments.	Some correct technical descriptions and/or code comments.	Limited documentation, and/or significant errors or misunderstandings.	Minimal evidence/ understanding

Example – 56% (12+16+8+4+12+4)

	100.00	80.00	60.00	40.00	30.00	20.00	0.00
Coding 20% Statements, method calls. sequences, if statements, loops	Full range of control structures used; efficient, clear and robust code.	Method calls, arguments and results; explain control and data flow.	Combine sequences, if statements and loops; explain control flow.	Mostly correct usage of simple sequences, if statements and loops.	Some correct examples of sequences, if statements and/or loops, based on course materials.	Limited understanding of course material examples, and/or significant errors	Minimal evidence/ understanding
Data 20% Variables, types, arrays, classes, collections	Clear understanding of datatypes, effectively used in original coding solutions.	More advanced types (simple inheritance, collection classes), going beyond course examples.	Explain/use basic types and arrays; define and use simple classes.	Mostly correct use of basic types, arrays and variable declarations.	Some correct examples of types, arrays and/or variables based on course materials.	Limited understanding of course material examples, and/or significant errors.	Minimal evidence/ understanding
Development 20% Program design and development, detecting and correcting logic and syntax errors	Clear code design and development. Good understanding of logic and syntax errors, with strategies to minimise and resolve them.	Good code design and development. Able to distinguish logic and syntax errors, and have strategies to fix them.	Describe code design and development, with examples of logic and syntax errors and how you fixed them.	Describe code design, with simple examples of any problems you had to solve	Some evidence of design and detecting and correcting logic and/or syntax errors.	Limited understanding of code design and/or error correction	Minimal evidence/ understanding
Testing 10% Testing and debugging	Effective, appropriate testing , for example using JUnit, and examples of bugs revealed and corrected.	Systematic testing at class level, use of debugging tools, discussion of the benefits of systematic approach.	Describe approach to testing, provide evidence, and discuss bugs found and fixed.	Evidence of testing code, and discussing testing/ debugging process	Some evidence of testing and/or discussion of testing and debugging.	Limited understanding/ evidence of testing and debugging.	Minimal evidence/ understanding
OO 20% Inheritance, encapsulation, overloading, overriding	Clear understanding of key OO techniques, appropriate use in a more advanced task.	Use of more advanced OO techniques in more sophisticated/ modular code for a moderate programming task.	Analyse a simple task using objects, create classes, with variables, constructors and methods, to implement it.	Mostly correct use of classes from course materials, creating objects and calling methods to achieve a simple task.	Some examples using basic OO techniques based on course materials.	Limited understanding of course material examples, and/or significant errors.	Minimal evidence/ understanding
Documentation 10% Comments and report	Comprehensive technical documentation, and Javadoc documentation demonstrating use of Javadoc features.	Clear instructions, detailed technical description, and Javadoc documentation of key aspects, using a range of Javadoc features.	Technical instructions/ description, informative code comments , for example using Javadoc (/** */) comments.	Basic technical description in report and basic end-of-line (//) and/or block (/* */) comments.	Some correct technical descriptions and/or code comments.	Limited documentation, and/or significant errors or misunderstandings.	Minimal evidence/ understanding

Example – 72% (12+16+12+10+16+6)

	100.00	80.00	60.00	40.00	30.00	20.00	0.00
Coding 20% Statements, method calls. sequences, if statements, loops	Full range of control structures used; efficient, clear and robust code.	Method calls, arguments and results; explain control and data flow.	Combine sequences, if statements and loops; explain control flow.	Mostly correct usage of simple sequences, if statements and loops.	Some correct examples of sequences, if statements and/or loops, based on course materials.	Limited understanding of course material examples, and/or significant errors	Minimal evidence/ understanding
Data 20% Variables, types, arrays, classes, collections	Clear understanding of datatypes, effectively used in original coding solutions.	More advanced types (simple inheritance, collection classes), going beyond course examples.	Explain/use basic types and arrays; define and use simple classes.	Mostly correct use of basic types, arrays and variable declarations.	Some correct examples of types, arrays and/or variables based on course materials.	Limited understanding of course material examples, and/or significant errors.	Minimal evidence/ understanding
Development 20% Program design and development, detecting and correcting logic and syntax errors	Clear code design and development. Good understanding of logic and syntax errors, with strategies to minimise and resolve them.	Good code design and development. Able to distinguish logic and syntax errors, and have strategies to fix them.	Describe code design and development, with examples of logic and syntax errors and how you fixed them.	Describe code design, with simple examples of any problems you had to solve	Some evidence of design and detecting and correcting logic and/or syntax errors.	Limited understanding of code design and/or error correction	Minimal evidence/ understanding
Testing 10% Testing and debugging	Effective, appropriate testing , for example using JUnit, and examples of bugs revealed and corrected.	Systematic testing at class level, use of debugging tools, discussion of the benefits of systematic approach.	Describe approach to testing, provide evidence, and discuss bugs found and fixed.	Evidence of testing code, and discussing testing/ debugging process	Some evidence of testing and/or discussion of testing and debugging.	Limited understanding/ evidence of testing and debugging.	Minimal evidence/ understanding
OO 20% Inheritance, encapsulation, overloading, overriding	Clear understanding of key OO techniques, appropriate use in a more advanced task.	Use of more advanced OO techniques in more sophisticated/ modular code for a moderate programming task.	Analyse a simple task using objects, create classes, with variables, constructors and methods, to implement it.	Mostly correct use of classes from course materials, creating objects and calling methods to achieve a simple task.	Some examples using basic OO techniques based on course materials.	Limited understanding of course material examples, and/or significant errors.	Minimal evidence/ understanding
Documentation 10% Comments and report	Comprehensive technical documentation, and Javadoc documentation demonstrating use of Javadoc features.	Clear instructions, detailed technical description, and Javadoc documentation of key aspects, using a range of Javadoc features.	Technical instructions/ description, informative code comments , for example using Javadoc (/** */) comments.	Basic technical description in report and basic end-of-line (//) and/or block (/* */) comments.	Some correct technical descriptions and/or code comments.	Limited documentation, and/or significant errors or misunderstandings.	Minimal evidence/ understanding

How to set yourself targets

- Identify the column that you think best reflects your level of confidence overall – your ‘comfort zone’
- Pick one or two rows corresponding to skills you think you are good at
- On those rows, look to the left of your comfort zone to see what extra bits you need to do to achieve that mark
- Focus on a project extension that will give you scope to do those extra bits
- Ask for advice in labs if you’re not sure

If things don't quite work out

- Having a perfect working program is always nice, but it is not one of the marking criteria!
- If you find it hard to get something working, ask for advice in labs – we can't help you directly on your code, but we can help work out for yourself what to do
- Don't throw away code that isn't working, or has bugs – you can still get marks for what you have done. But talk about the problems in your report.

Submission

Coursework submission

- Submission deadline: 3pm – 30th April 2021
- Marks and feedback: 2nd June 2021

You need to submit two items to different submission links:

1. The code and documentation of your project – a BlueJ or Eclipse project folder, as a zip file
2. Your project technical report, as a docx or pdf file

The submission links are in the module area in the Assessment page:

1. Coursework project code (zip file)
2. Coursework technical report (docx or pdf file)

Make sure you submit the right way round or you will get submission errors.

What to submit - project code

- The first part of this coursework is the project code itself.
- This should be a zip file containing all the project files (java files, class files, css files etc.) as well as Javadoc documentation (which you can generate automatically from the code – we will learn about this).
- It can be a BlueJ project or an Eclipse project

What to submit - technical report

- The second part of this coursework is a short technical report with the following sections:
  - Description of the project and what you aimed to do
  - Brief instructions to run the code
  - How the code works – the main classes, and how they work together to make the app run
  - Testing – how did you check everything worked properly?
  - Reflection – your own thoughts on how well the project went, what went well, and what didn't go so well, how you solved problems and tested your code, things you tried to do, or would like to do to make it better

Technical report – additional tips

- Use the report to draw attention to things you did that meet the marking scheme – it is much easier for us to mark based on the report than trying to pick things out of the code.
- Don't make the report too long – 3-5 pages is fine, and no more than 8. Include an extra page if you implemented your own project from scratch, to demonstrate your understanding of the code. Don't count the title, contents or references when counting pages
- Include screenshots, small code listings (to show a particular feature) if you wish, and references to websites or books that you used
- If you use actual code examples or libraries from the web, make sure you include a reference to them in the code (as a comment) and in the report
- You can use Javadoc to include more details about the code itself as part of the code submission, so you only need to give a high level view of the code in the report.

Extensions

You can get an Extension (normally 2 weeks) if you have a Learning Support Plan that says so, or if you have a good reason (illness etc). But it is not automatic, and you must apply for it before the deadline.

You can also apply for Mitigating Circumstances if something more serious has happened that will affect your performance in an assessment.

Contact the School Office for the forms for these processes.

Submission problems

- The submission deadline is 3pm on 30th Apr 2021
- If you have problems with the submission system near the deadline, don't panic! But do something!
  - Take a screenshot (including the computer date and time) as evidence that you are trying before the deadline.
  - Contact the service desk for help: servicedesk@brighton.ac.uk, or 01273 644444 (or just ext 4444 from a uni phone)
  - Contact the School Office with evidence of your submission as quickly as possible after the deadline
  - Don't contact the module team – we can't help.

Plagiarism

Plagiarism

- Plagiarism is trying to take credit (in particular, trying to get marks) for work that was done by someone else
- This includes copying from websites or other people's essays, without saying that you are doing so.
- It is ok to have a quote or a diagram from somewhere else, if you provide a reference to where you got it from (but you don't get as many marks for doing that as you would for writing it yourself)

Plagiarism and TurnItIn

- We use a system call TurnItIn to check for plagiarism automatically when you submit something
- For CI401, you will be submitting your technical report using TurnItIn.
- TurnItIn automatically checks your report and gives you a similarity score – a measure of how much of your document is similar to things TurnItIn has seen before (websites, other student submissions from Brighton and other universities etc)
- It is important to learn to write things in your own words, rather than copy-pasting from things you have read.
- And in serious cases of plagiarism, we have a formal process for dealing with Academic Misconduct ('cheating').

Plagiarism and TurnItIn

- You can submit your report early, and get a TurnItIn score from it.
- The TurnItIn score is out of 100, and it is perfectly possible to get a score below 5. If you get a score above 20 you should definitely check to see why.
- TurnItIn will give you a report showing in colour the parts it thinks come from somewhere else. It does get confused, especially by code, so if your score is higher than you expect, it is worth checking what is going on.
- You can submit your report as many times as you like, so if you submit it once and the score is high, you can change the paper and submit it again.
- Ask for help in labs if you are unsure.