2020 CI401
Introduction to programming

Week 2.01
Introduction and Project

8th February 2021

Roger Evans

Module leader

Lecture recording and attendance

- This lecture will be recorded and published in the module area
- The focus of recording is on the lecturer, not the audience
- If you are particularly concerned not to be part of the recording, it is ok to turn off your microphone and camera

- In addition, lecture attendance is now being routinely recorded (in all modules) to help the School Office monitor engagement

- (This slide is really a reminder to me to start recording and record attendance!)

Module team contacts (semester 2)

Name	Role	Contact
Roger Evans	**Module leader** Lectures, seminars, labs	C507 (Cockcroft 5th floor) R.P.Evans@brighton.ac.uk (email) @Roger Evans (teams) @rogerevansbton (twitter)
Course administrators	**General questions**	CEMUGComputing@brighton.ac.uk
Stelios Kapetanakis	Seminars, labs	
Karina Rodriguez	Seminars, labs	
Goran Soldar	Seminars	
Ali Hamie	Labs	
Jim Burton	Labs	

Module timetable – Semester 2

	Monday	Tuesday	Wednesday	Thursday	Friday
0900				Lab Online (H401)	
1000					Labs Online (W201,202)
1100	Lecture Online				
1200					
1300	Seminar Online			Lab Online (C107)	
1400				Lab Online (C107)	Labs Online (W201,202)
1500	Seminars Online			Lab Online (C107)	
1600					

Online and on campus teaching

- Online teaching for all sessions until at least 5th March
- From 8th March, your timetable will show rooms for labs only – online continues for lecture and seminars, and is still available for labs as well
- This date may be delayed further, depending on future Government guidance
- Timetables only published up to Easter vacation for now

Module structure (version 4)

Semester 1

Week	Topic	Theme
1.01	Introduction / Hello World	Coding
1.02	Variables, loops and choices	Coding
1.03	Input, more loops and choices	Coding
1.04	Variables and expressions	Coding
1.05	Types, assignment and arrays	Data
1.06	Let's play Top Trumps!	Data
1.07	Objects and methods	OO
1.08	Working with numbers	Data
1.09	Simple Algorithms	Dvp
1.10	Introduction to JavaFX	Dvp
1.11	No lecture (Strike)	
	Xmas vacation 21 Dec – 8 Jan	
1.12	Simple Animation	Dvp
1.13	Semester 1 Review	

Semester 2

Week	Topic	Theme	Project
2.01	Project topics and assessment	Project	Set
2.02	Simple Inheritance	OO	Lab
2.03	Scope, Visibility and Encapsulation	OO	Lab
2.04	Testing - JUnit	Testing	Lab
2.05	Documentation - Javadoc	Doc	Study
2.06	Collections and generic types	Data	Study
2.07	IO: files and streams	Dvp	Study
	Easter Vacation 29 Mar – 16 Apr		
2.08	Numbers – the computer's view	Data	Study
2.09	Java vs Python		Submit
2.10	More algorithms – search and sort	Dvp	
2.11	How fast is my code?	Dvp	
2.12	Java 'under the hood'		
2.13	Revision week		Exam ↓

Assessment

	What	Set	Deadline	Marks
Task 1	Coursework project report	8th February 2021 (Start of semester 2)	30th April 2021 (End of week 2.09)	50%
Task 2	‘Open book’ exam 1.5 hours (2.25 hours if online)	During exam weeks 30th May – 11th June 2020		50%

The coursework project

Overview of the project

- The project runs from week 2.01 to week 2.09 (8th Feb to 30th Apr)
- Aim is to show your learning in several aspects of programming
- You will complete your project in lab sessions and private study
- We will help you with lab exercises, but not detailed project work
- In lectures we will look at new techniques you can incorporate into your project as you go along
- You will have a choice of four project options

Project options

- Either use a ‘starter’ project that we provide and add features to it
- Or write your own project from scratch
- Starter projects
  - Breakout game (classic arcade game)
  - ATM (cashpoint) interface
  - Maze game (finding treasure/avoiding ghosts/pac-man style game)
- We will look at these three starters over the first three weeks, then you decide which you want to focus on.

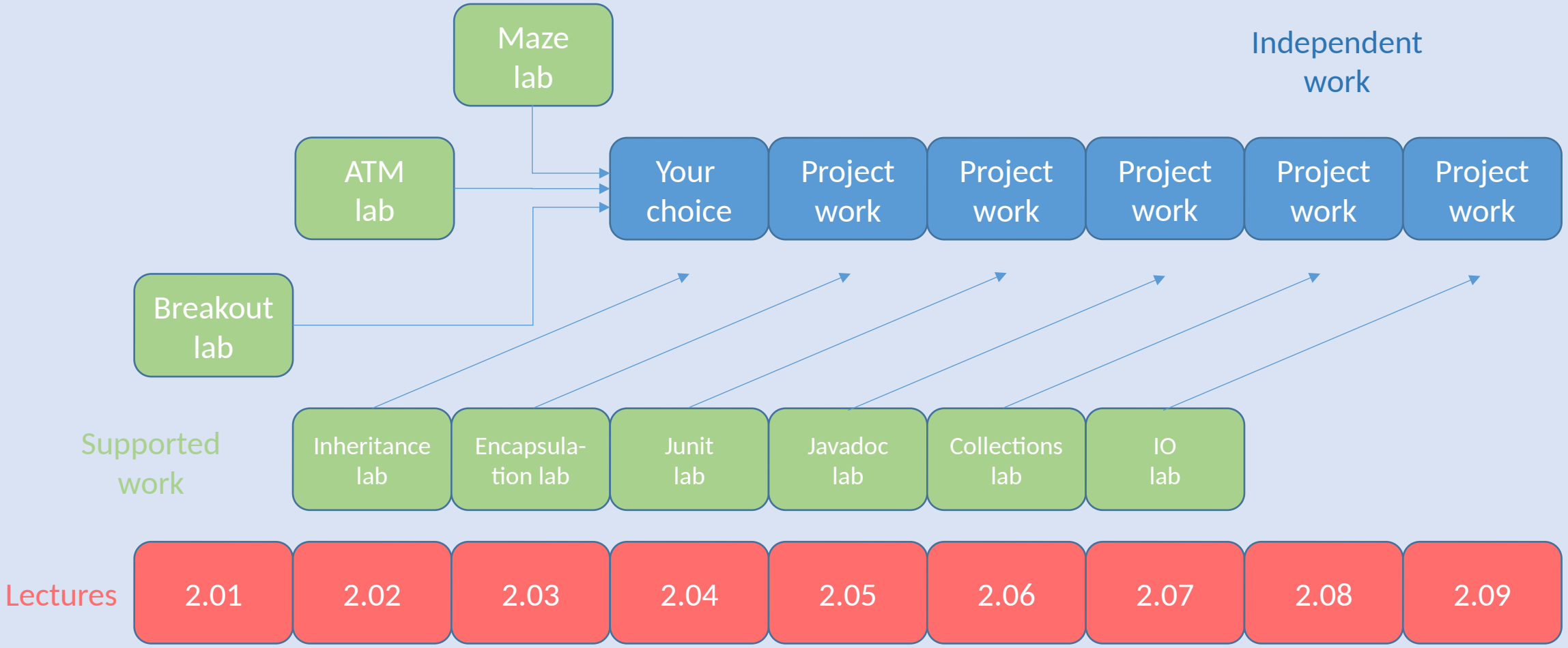
What you get marks for

- **Coding** – using different statements, method calls etc
- **Data** – using variables, arrays, types etc
- **Development** – program design and development, code organisation and style
- **Testing** – techniques for testing and debugging your code
- **Object-oriented features** – using simple OO programming techniques (inheritance, encapsulation etc.)
- **Documentation** – documenting your code AND report writing

What you learn about alongside

- **Inheritance** – a fundamental concept of object oriented programming
- **Scope, visibility and encapsulation** – how (and why) to keep things private or public in your code
- The **JUnit** testing library
- The **Javadoc** code documentation system
- **‘Collections’** – a fancier version of arrays for organising data
- **Files and streams** – storing and loading data from files

How it fits together



How the project is marked

- Marking criteria each have a fixed amount of the mark:
  - Coding 20%
  - Data 20%
  - Development 20%
  - Object oriented elements 20%
  - Testing 10%
  - Documentation 10%
- You need to do all of these well to get a good mark
- You only need to do each of these ‘ok’ to pass

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

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

Further information

- More details in the coursework guide (which is also part of the assignment brief)
- We will come back to different parts of the process later in the semester

Introduction to Breakout

Introduction to Breakout

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 week's lab exercise, you are given a version of Breakout that does almost nothing.
- The lab exercise is to turn it into a functional game (with help from tutors if you need it)
- The solution to this lab is the starting point for your independent project work, if you choose to do Breakout.

Demonstrating Breakout

- The solution system –a simple, working, Breakout game, with a row of bricks which disappear correctly when the ball bounces off them

- The lab exercise system – without any bricks! But it does have a ball, a working bat and a score function

What's actually happening in the code of this game?

- On the screen, there's a score, some bricks, the ball and the bat
- The screen appears to change – the ball moves, the score updates, bricks disappear, the bat moves when you tell it to
- But as we saw with our JavaFX animation, nothing is really moving – what's happening is the image on the screen is being changed very quickly (50 times a second)
- What our program is doing is working out what should appear on the screen at each point, and then displaying it

How does the program decide what to display?

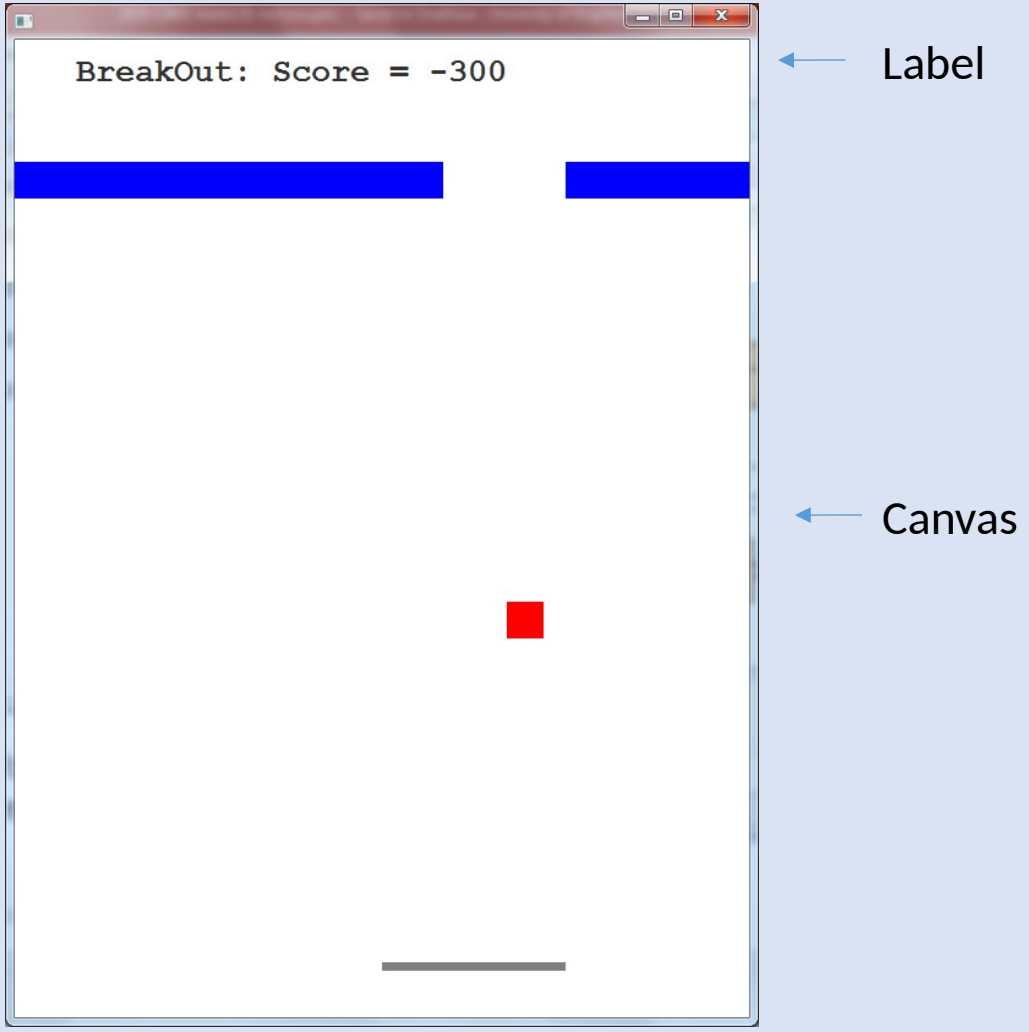
- The ball is ‘moving’. So for each image (50 times a second), the program needs to work out where the ball has moved to (so it needs to know where it is, what direction it is going in, and how fast it is supposed to be ‘moving’)
- The bat also ‘moves’, but only when a key is pressed. So the program is watching for keys to be pressed, and changing the position of the bat when that happens
- Whenever the ball moves the program also checks whether it has hit anything:
  - The side or top – the ball needs to change direction to bounce off
  - The bottom – the ball bounces off and there is a score penalty
  - A brick – the ball bounces off, the brick disappears, and the score increases
  - The bat – the ball bounces off

How is the program organised?

- The program follows a very common pattern for writing GUI programs called model-view-controller (or MVC)
- We will see MVC a few more times this semester. Today we just need to get an idea of the basics
- In MVC the main structure of the program is divided into three classes
  - The Model class – where all the calculations about what is happening in the game are stored
  - The View class – which manages what you actually see on the screen
  - The Controller class – which decides what happens when the user presses keys etc.

The View class

- We start with the View, because it is the most ‘visible’
- This is a JavaFX window like the ones we have seen before
- In fact it only has two visual components:
  - A Label, showing the score
  - A Canvas, on which it ‘paints’ the current game state (repainting the whole thing 50 times a second)
- In addition it has an event handler, which detects keyboard presses



The Model class

- This is where the 'game logic' is
- It has a loop, running 50 times a second, when it moves the ball and checks whether it has hit anything
- It also responds to commands to move the bat (and a few other things)
- It uses a set of 'Top Trumps' cards (the GameObj class)for all the elements of the game – bat, ball, bricks
- All the mode is doing is changing numbers on the cards

boolean visible	true
int topX	0
int topY	0
int width	0
int height	0
Color colour	
int dirX	1
int dir Y	1

← GameObj Class

boolean visible	true
int topX	200
int topY	350
int width	30
int height	30
lor colour	red
dirX	1
dir Y	1

↑ Ball

boolean visible	true
int topX	200
int topY	650
int width	150
int height	10
Color colour	gray
int dirX	1
int dir Y	0

← Bat

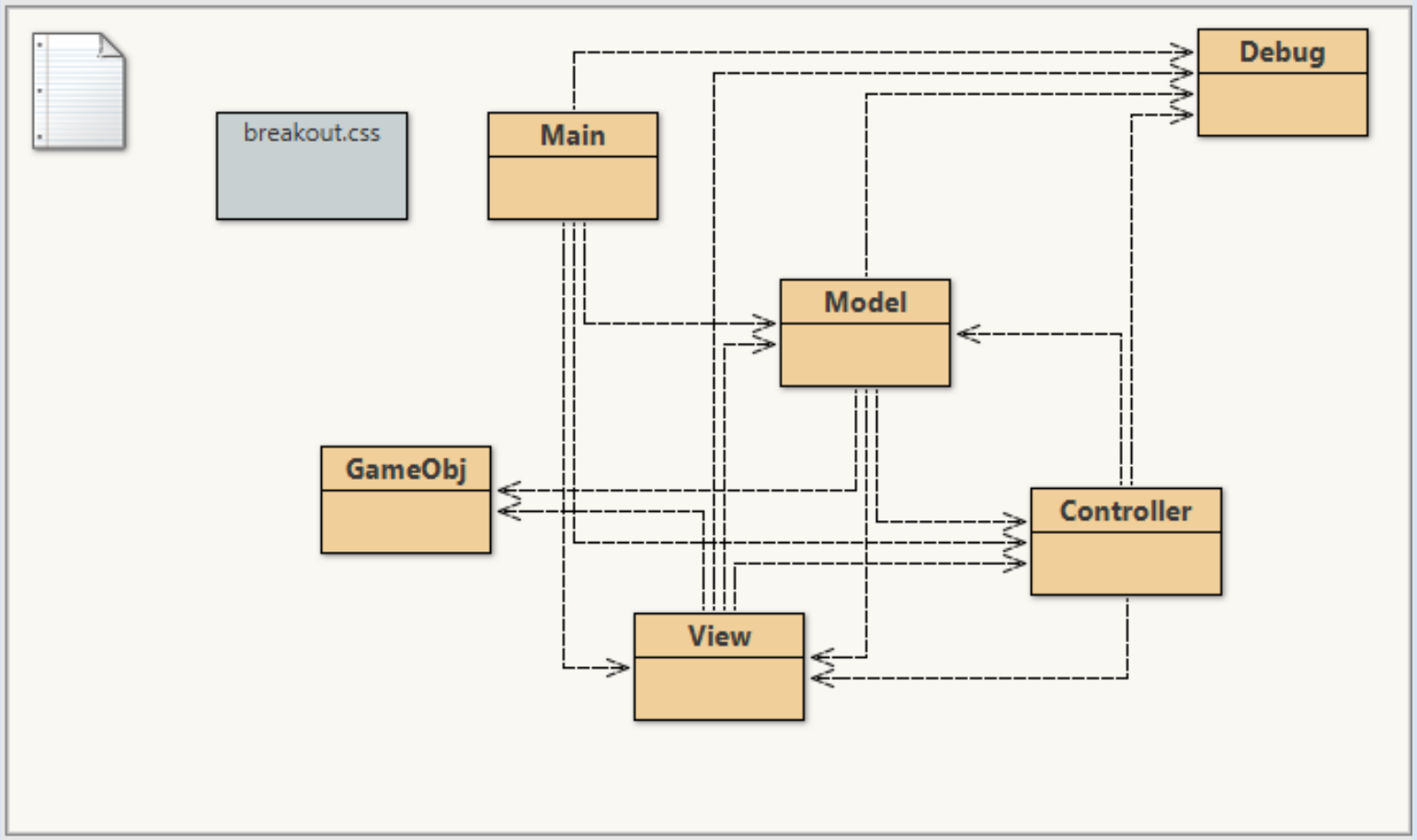
The Controller class

- The simplest of the three classes
- This provides the event handler function that the View uses
- Whenever the user presses a key, the controller decides what the game should do and tells the the Model to do it
- For example
  - When the user presses ‘<‘, it tells the Model to move the bat left
  - When the user presses ‘>’, it tells the model to move the bat right
- You can change the way keys map into commands without changing the View or the Model
  - Make a left-handed version (using , say ‘Z’ and ‘X’ instead of ‘<‘, ‘>’)
  - Add functions for a two player version (on one keyboard)

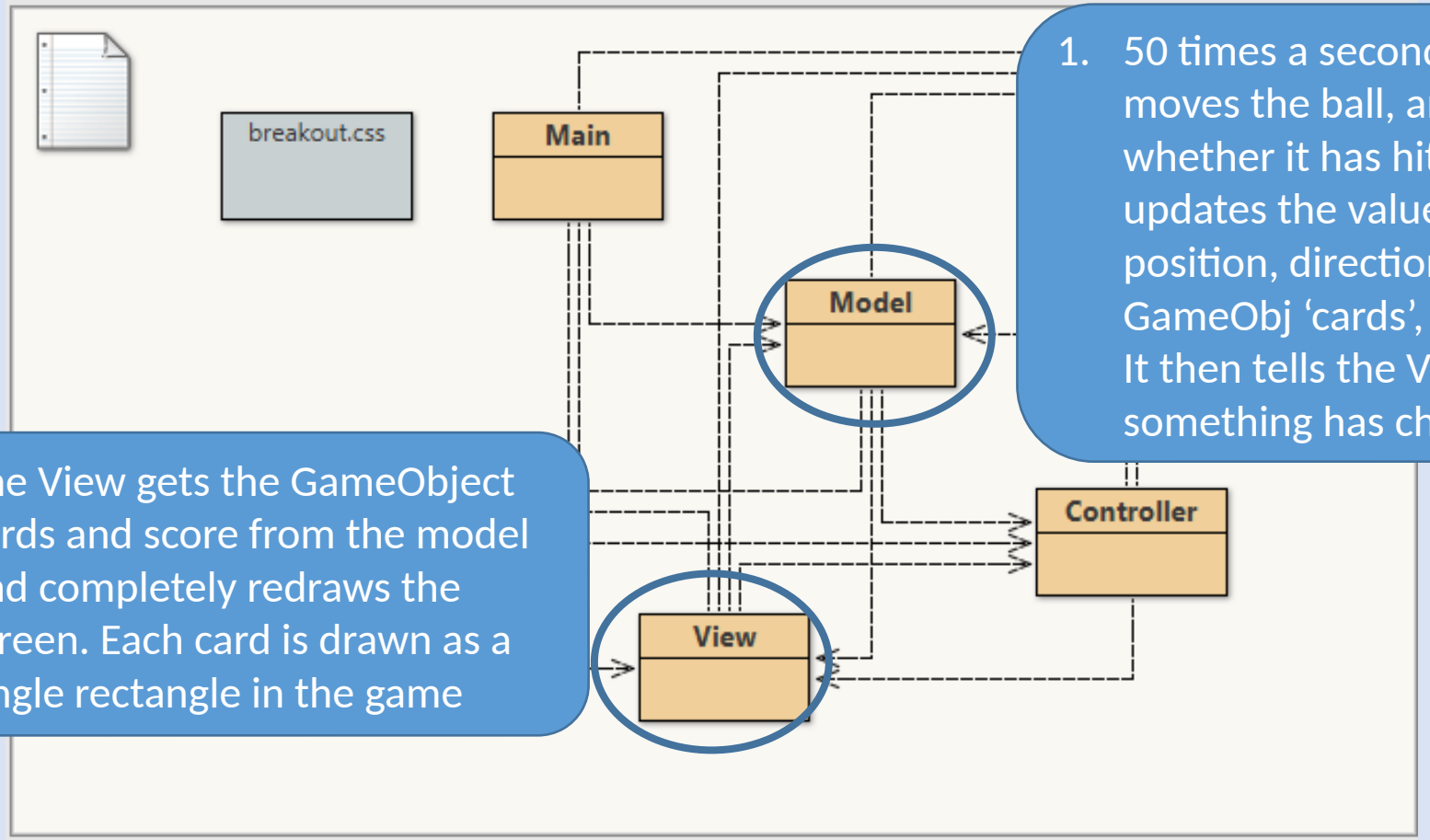
Other classes

- The Main class – starts the program, creates the Model, View and Controller objects and ‘joins them together’
- The GameObj class – the main data class for the things in the game (this is the ‘Top Trumps deck’ the model uses)
- The Debug class – prints out messages about what is going on to help you debug the program

Breakout classes



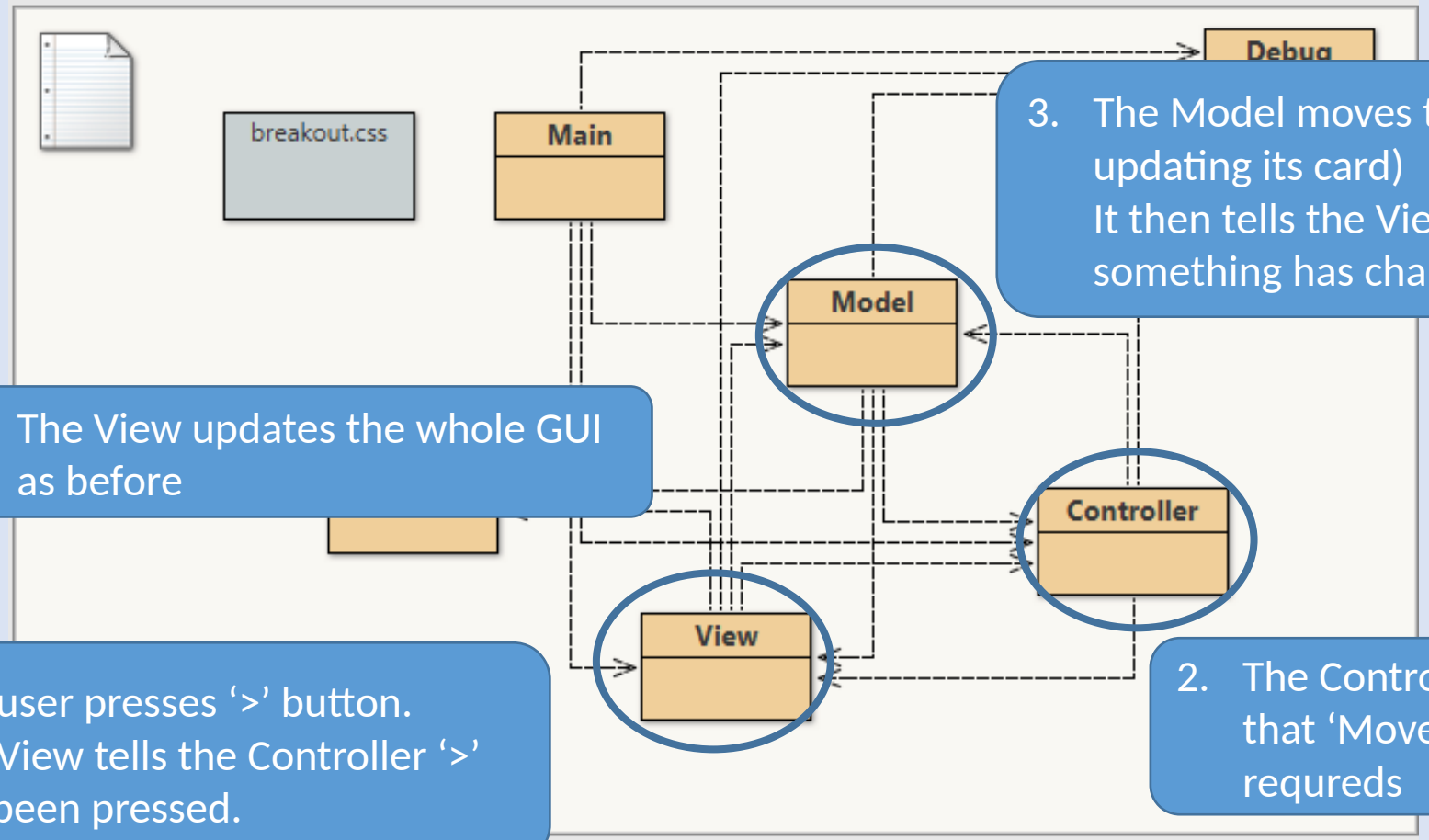
How the components talk to each other – main loop



1. 50 times a second, the model moves the ball, and checks whether it has hit anything. It updates the values (visible, position, direction etc) on the GameObj 'cards', and the score. It then tells the View that something has changed

2. The View gets the GameObject cards and score from the model and completely redraws the screen. Each card is drawn as a single rectangle in the game

How the components talk to each other – key presses



1. The user presses '>' button.
   The View tells the Controller '>' has been pressed.

2. The Controller tells the Model that 'Move bat left' has been requreds

3. The Model moves the bat (by updating its card)
   It then tells the View that something has changed

4. The View updates the whole GUI as before

More details about the code

- Some slides looking at the code details are coming very soon (before the labs!)
- We may show you a bit from last year’s version in the Seminars today
- But I want to improve the game a little, and so I don’t want to confuse you by distributing the old code just yet!

Making Breakout work – the lab exercise (not assessed)

- Model class
  - add bricks to the model (the shell has no bricks!)
- View class
  - display the bricks on screen
- Model class
  - if the ball hits a brick, the brick is destroyed

Suggestions for adding features to Breakout (can be assessed)

- Adding effects like colour or sound
- Adding more bricks and brick layouts
- Stopping the bat from going off the sides
- Losing ‘lives’ if the ball goes off the bottom
- Adding a ‘Game over’ screen
- 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

Lab exercises
Week 2.01

Breakout lab exercises

- Download the Breakout game as a BlueJ (or Eclipse) project
- Run it (as a JavaFX application)
  - you will see a simple game screen with a moving ball, a score
  - Also you can use the < and > to move the bat at the bottom of the screen
  - But there are no bricks!
- The lab exercise for this week is:
  - to get a general idea of how the Breakout program works
  - add code to it for a single row of the bricks, which disappear when the ball bounces off them

Breakout – adding bricks

- There are three places in the program where you need to add code:
  - 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]
- There are big comments in the code to show you where you need to make these changes

JavaFX Notes

JavaFX, BlueJ and Eclipse

- Breakout is implemented using JavaFX
- JavaFX is a standard Java library, so much of the code development etc., is completely standard and supported by standard tools.
- However, because JavaFX provides GUI support, there are some points at which it may interact or interfere with the GUI support of the tools used to develop systems, such as BlueJ or Eclipse

JavaFX and BlueJ - 3 useful points

1. When creating a new top level JavaFx class, you should select the class type JavaFX Class, rather than just Class. This loads a convenient template for JavaFX classes, and also allows BlueJ to handle them differently when running a class.
2. A class which has been created as a JavaFX Class has a new option on the class menu: Run JavaFX Application. This is the option to use to test a class you have created.
3. JavaFX application classes are created with a 'main' method. This method allows your code to run as a standalone Java program. However, if you try and run the main method from BlueJ (as you would any other Java program class), it will generate an error. Instead use option 2 (above).

JavaFX and Eclipse

- The standard installation of Eclipse for Java does not automatically include the JavaFX libraries in new Java projects. This means your JavaFX classes will not compile.
- To create a project which does include JavaFX, you need to change the JRE setting when you create the project – see next slide for details

- If you have your own eclipse installation, you might like to install the e(fx)clise plugin (https://www.eclipse.org/efxclipse/index.html), which provides many more JavaFX-related tools. I've not played with this yet though, so you are on your own with it (though I'm always happy to try and help).

Creating a JavaFX project in Eclipse

To create an Eclipse project which does include JavaFX, select the third JRE option ('Use default JRE')

(A standard Java project uses the first option, which does not include JavaFX)