Fundamentals of Programming Coursework Specification 2025-26

Important Information

This coursework is an individual assignment measuring your attainment of the learning objectives of this module. You must complete the work yourself, without assistance from other students, and you must not use materials that you find online without proper citation and referencing. Do not ask other students to help you, and do not share your code or report with other students. All parts of your submission will be checked for similarity to other student submissions, including your Java code, as well as against online sources. If you try and work together, use the work of others without citations, or share your code or report, this will be detected and you will be subject to the formal plagiarism process as part of University regulations, with penalties up to and including expulsion from the institution.

AI use in this assessment

The use of artificial intelligence tools, generative or otherwise, is not permitted for this assessment. You must produce all the code yourself, and you must write all of the content for the report yourself. The use of any AI tool will be considered academic misconduct. You may not use AI for ideation, research purposes or summarising ideas. You may not use AI to produce, enhance or change any aspect of the work in the assessment. If your work appears to contain AI generated content you may be asked to explain the content of your submission and the process used to produce it as part of an academic misconduct investigation.

Deadline: 15:30 Friday 12th December 2025

Late submissions without an approved extension will receive a mark of zero **with no exceptions**. Extension requests need to be submitted formally using the procedure described at: https://unibradfordac.sharepoint.com/sites/registry-and-student-admin-intranet/SitePages/Extenuating-Circumstances.aspx

Please do not e-mail the module co-ordinator asking for an extension. Module co-ordinators are not permitted to grant extensions outside of the formal process linked above.

Introduction

The coursework assignment for this module requires you to work on a software project provided by the module co-ordinator by completing a series of tasks described in this document. The tasks ask you to complete Java methods to add functionality to a basic 2D game. You will also write and submit a report on your submission, describing how your code works for each of the tasks.

The software provided to you is partially complete – the code to draw graphics to the screen and to handle input from a user are already working and do not require any modification. You will complete methods in a specific class to build key parts of the "game engine" – the code that implements the rules of the game, specifying what is drawn to the screen and what happens when a user interacts with the system using simple keyboard inputs.

The software is fully documented and the methods that you need to complete have both a **task description** in this document and **documentation within the code** describing what they should do. You must use what you have learned during the module to write the code to complete these methods. As you complete each task you will find the program grows more functional: each task, when completed, should produce some new visible output or keyboard-driven behaviour on the screen for you to test and assess against the task description and method documentation.

The software

The coursework software is provided to you as a NetBeans project (a ZIP file) and can be found in the Assignment section of Canvas. You must download this project and import it into NetBeans. Instructions to do this are given below.

The software is comprised of ten Java source code files and a number of graphical image files inside the project folder, as well as other files created by NetBeans related to the project configuration. **You must review and examine each class**, reading the documentation for each and **understand how the software works before you start working**. The modifications that you will make to the software are limited to the `GameEngine` class for almost all tasks, which you will see has a number of empty or partially complete methods for you to work on. It is strongly recommended that you do not modify any other classes without speaking to the module co-ordinator first, unless a task specifically mentions the need to modify other classes.

Importing and opening the software

1) Download the ZIP file containing the software project from the Canvas Assignment area and save it to a location of your choice (strongly recommended that you use your University OneDrive account).
2) **Extract** the contents of the ZIP file to a location of your choice by right-clicking on it and selecting the appropriate option. **Do not just double click to open the ZIP folder!** You should now have a folder called `FoPCW2025`.
3) Rename the extracted folder to **add your username** to the end of the folder name - e.g. if your username is `jsmith11` then you should rename it `FoPCW2025jsmith11`
4) Start the Apache NetBeans application
5) From the File menu select Open Project
6) Navigate to the location of the folder you just renamed and select/open it
7) You should now have the project opened in Apache NetBeans ready for you to work on

Please discuss any problems opening the software with the module staff during the first lab session!

Submission Instructions

You **must submit two separate files**: the written report for this assignment in Word format and a separate ZIP file containing the exported software. Both files must be submitted at the same time through the submission link on Canvas. **Do not ZIP both parts** of your submission together into one ZIP file. Keep them separate and attach/upload **both, one after the other**, as part of the same submission.

If you do not submit both a report and a software solution containing your Java code then **you will receive a mark of zero,** and if you ZIP both required files together and submit just a single ZIP file **you will also receive a mark of zero.** The code and the report together demonstrate your attainment of the learning objectives for the module and submitting only one is not sufficient to demonstrate your understanding of the module content.

The Report

Part of your submission for this coursework is a written report that describes how your code solution for each task works. The report should be presented professionally and conform to the guidance below. The report **must** be submitted as a Microsoft Word document. **It is strongly recommended that you write up the relevant report section for each task you complete after you complete it and save it to your OneDrive, and seek feedback on your writing style during timetabled lab sessions to avoid losing marks for poorly written and non-technical content.**

The report must be structured and have content as follows:

- A single cover page that incudes your UB number, the module code and the module title only
- One section for each task that begins with the task number and name, comprised of **at most two paragraphs of text** that concisely describe how your code for each task works. The text should be focussed on explaining how your code works, and there **should be between 25 and 300 words** describing your solution for each task, depending on the complexity of the code. **300 is a maximum** and some tasks should use less than 300 words if they are simpler to explain. Do not write about how difficult you found a task. Do not write your personal thoughts and feelings about the task or your code. Do not write in a "diary" style describing what **you** did, and do not write using "I…" or "My…." to describe the code. Write only about the code (e.g. "The solution uses two loops…" rather than "I wrote two loops") and how the solution works.
- Only describe **how the code works and avoid generic, non-technical content such as explaining why the code exists, why the feature it implements is used in the game, how it affects player enjoyment etc.** You should seek feedback from the module co-ordinator during lab sessions on your written drafts to check that you are using the right technical style. Failure to do so may significantly reduce your marks for tasks, as the clarity of your explanations is used to assess your attainment of the learning outcomes; it is not enough just to have some code that runs if the report does not explain how it works clearly and in your own words.
- A final section describing any tasks with partial solutions that you partly developed but did not submit, or which were left as comments in the code. Describe the algorithm or partial code and any parts that worked. **Do not write about why you failed to complete tasks or that you ran out of time.**

The report must be submitted through Canvas, using the TurnItIn link in the Assignment area. If you put the report into a ZIP file it cannot be scanned by TurnItIn and **you will receive a mark of zero.**

Software submission and export instructions

You must also submit the software through Canvas. You must submit your final version before the deadline, and **submit a version that works** i.e. that compiles and is testable by the module staff by running the project **using Apache NetBeans 26 or 27 and JDK 24**, demonstrating the completed tasks.

You must submit the ZIP file containing your software solution that Apache NetBeans creates when you Export it from the File menu as a ZIP file. To do this click the File menu and select export Project -> To ZIP. Select the location to build the ZIP file to and verify that the ZIP file has been added to the chosen folder. Note that if you change the save location you may need to add the .zip extension back onto the file name. **You must check the content of the ZIP file before submission to ensure it contains the correct source code files for the latest working version of your code**.

It is very strongly recommended that you export your project when you complete each task, keeping a backup of every version stored in your University OneDrive account. This will avoid any problems with "lost" coursework and allow you to submit the latest version that compiles and runs. It is vital that your final submission runs successfully when module staff test it by running the complete project using Apache NetBeans 26 or 27 and JDK 24 installed on the University Horizon platform – therefore you should test it before submission using Horizon.

Deadline

The deadline for submitting your software and report is 15:30 Friday 12th December 2025. Any work submitted after the deadline will result in a mark of zero for your entire coursework, effectively meaning that you fail the entire module.

This includes work submitted at any time after 15:30:00 on Friday the 12th of December 2025. To be explicitly clear, submitting as little as one second after 15:30:00 means you get a mark of zero. Do not leave your submission to the last few minutes or you risk significant damage to your degree progression. Deadlines are not flexible – they are the absolute latest and final permitted submission time and it is your responsibility to submit your work in time.

Task specifications

To complete this coursework you must work through the tasks specified below. Each Task has an associated number of marks for completing it. Solutions that efficiently implement more complex ideas for solving the task, that describe the solution more clearly in the report using concise technical descriptions, and that follow Java coding conventions more closely will gain more marks. You may notice that the total marks for all tasks sum to more than 100. You can only achieve a maximum capped mark of 100 in this coursework, allowing you to achieve the highest marks without having to have a "perfect" solution for each task; you will simply need to complete as many tasks as you can, documenting and formatting the code for them properly.

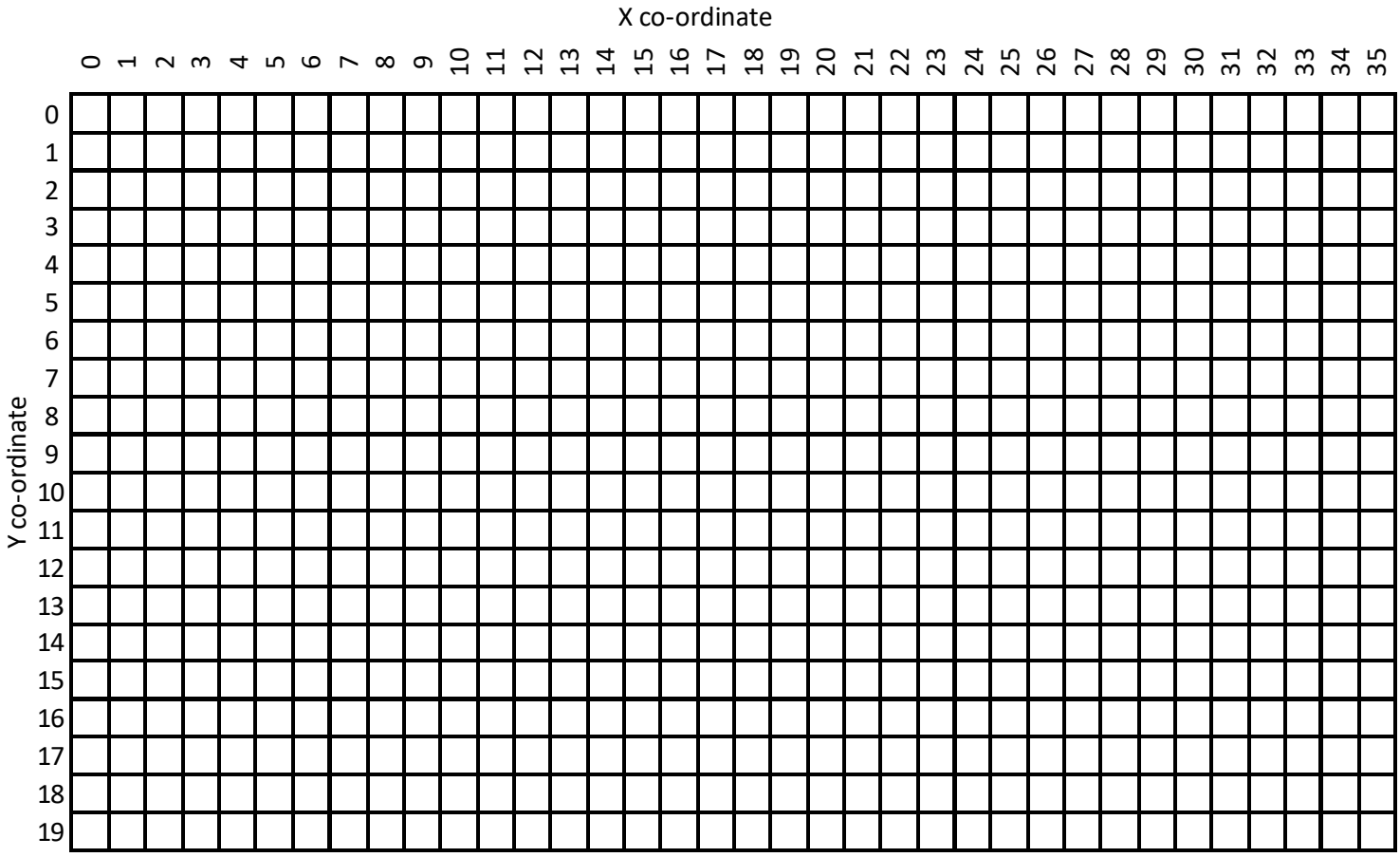
Although in theory you can complete the tasks in a different order, **it is strongly recommended that you complete tasks in the order suggested**, as they build on top of one another, and later tasks may not make sense if the earlier tasks are not completed. You may like to return to a task after you have completed it to add more features, or to increase the complexity of your solution/algorithm; more complete, challenging, efficient and elegant solutions will be awarded more marks.

Once again: It is strongly recommended that you "Export" your project after you complete every task so that you have a working copy ready to submit at every stage. You should submit the latest **working** version of your solution – **your code must compile and run when it is tested** for marking. Submitted solutions with compiler errors will have a very heavy penalty to the marks awarded and are likely to result in failing the module. Get help in the labs early and don't be afraid to ask questions.

Unless otherwise specified, all methods described below are in the `GameEngine` class, and the documentation in the code for each method describes the expected behaviour.

Co-ordinate system details

The game uses numerical X and Y co-ordinate values to represent specific tiles in a level, with a level being made up of a grid with size 36 by 20. The image below may help you to understand the co-ordinate system for generating levels and placing the player in the game. **X co-ordinate values are first, Y co-ordinate values are second**, much like graphs in mathematics with the style (X,Y), **but the origin (0,0) is in the top left**.



Task 1 – Create the player object (5 marks)

You must complete the code for the `createPlayer()` method. The method must instantiate a `Player` object that represents the player in the game and assign the object to the GameEngine class attribute called `player`. The `Player` class documentation describes how to create a `Player` object using the appropriate constructor. You must set `X` and `Y` positions corresponding to a tile position in the current level; for now any position within the grid shown above is fine. You will know your solution works when you see an icon for the player appear in the game window.

Task 2 – Move the player (10 marks)

The software already detects when the user presses a key on the keyboard. The `InputHandler` class handles the key press and calls the `doTurn()` method of the `GameEngine` class every time the user presses any key on the keyboard – do not try to call this method yourself in any other part of the code or it will make multiple turns pass with every key press. If the key pressed was one of the arrow keys on the keyboard, it also calls a method in the `GameEngine` class to move the player in the appropriate direction. Your task is to complete the code for the `movePlayer()` method so that the player moves when an arrow key is pressed. Do not write code in the other movement methods yet; they will be used later.

For your first attempt at this task it is suggested that you complete the code for one movement direction, test it and then copy-paste-modify the code for the other directions. Design a brief algorithm first that specifies what should happen for a single direction with respect to the X and Y attribute values for the `player` object and how they should change to make the player move. You will need to retrieve (via a method in the `Player` and/or `Entity` class) the current player object's position, then set a new position using a method and new X or Y values based on the previous position values.

For this task you should not worry about the player moving off the edges of the map or what happens if you add tiles to the map – these will be handled in a later task.

Task 3 – Creating a farm (10 marks)

You must complete the code for the `generateFarm()` method. The method must instantiate and then fill the two dimensional (2D) array called `level` , an attribute of the `GameEngine` class. The `level` array must be filled with `Tile` objects, one for every element of the 2D array. The contents of the array you create and fill (i.e. the `Tile` objects) is used to draw tiles to the screen in the `GameGUI` class. Your code must create (instantiate) a new `Tile` object for every element in the level array by calling the appropriate constructor from the `Tile` class inside a pair of nested for loops. You must specify the type of `Tile` for every object you create and place into the array. The `Tile` constructor requires you to pass a `TileType` value to it when called. `TileType` is an enumeration in the `Tile` class. For example, a 2D array of `Tiles` using only `TileType.STONE_GROUND` tiles will draw a full level of "empty" tiles. It is recommended that you begin your solution to this task by filling the array entirely with `TileType.STONE_GROUND` tiles.

Task 4 – Better player movement (5 marks)

You must complete the code for the `betterMovePlayer()` method to improve the movement logic for the player object. Check the documentation for this method now. Leave the original movement code you wrote for Task 2 unchanged or you will lose marks for that task. This new method should check if the new X and/or Y co-ordinates for the player would place them outside the level boundaries after

moving, and prevent any movement if it would leave the player in an invalid position. You will find this task easier if you create an algorithm design first that helps you understand the logic and processes/decisions to implement the new movement concept. You should use decision making statements in the code that check the X and Y values to decide whether to move the player or not.

Task 5 – A better farm (5 marks)

You must complete the code for the `generateBetterFarm()` method to create an improved farm layout. Check the documentation for this method now. You must leave the code for your original `generateFarm()` method unchanged, or you will lose marks for that task. The new method requires you to add a patch of `TileType.DIRT` tiles to the level that can be farmed in a later task. For higher marks you should generate this dirt patch in a dynamic way, so that its size and/or position changes each time the game is started.

Task 6 – Tilling dirt (5 marks)

You must modify your latest level generation code so that a tile with the type `HOE_BOX` is added to the level in a sensible place. You must also modify your player movement code in two ways. Firstly, if the player attempts to move into the `HOE_BOX` tile then the player should not move, but the relevant attribute of the player object should be changed to indicate that they are now holding the hoe tool. You will know if this works as the player image will change to show it holding a hoe with a brown handle and blue head (make sure it is not an axe, seed bag or pickaxe!). Secondly, you must modify the player movement code so that if the player moves into a dirt tile while holding the hoe, the dirt tile is changed to a `TILLED_DIRT` tile. You can achieve this by creating a new `Tile` object with the right type and writing it into the `level` array in the correct position.

Task 7 – Sowing seeds (5 marks)

You must modify the player movement and level generation code, similar to the previous task, to implement the concept of seeds and sowing seeds on tilled dirt tiles. Add a seed box tile to the level, and change the player's held item if the player attempts to move into this tile (a green bag should appear in the player's hand if this works correctly). If the player is holding the seed bag and moves into a tilled dirt tile, the tile should be changed to a sowed dirt tile.

Task 8 – Even better farms (5 marks)

You must complete the code for the `generateEvenBetterFarm()` method, and once again must leave the code for previous level generation methods unchanged. Check the documentation for this method now. Your newly improved code must add a "farmhouse" to the level comprised of some house floor tiles and one bed tile. Ideally you should also use wall tiles to create a more realistic house, remembering to leave an open tile for exiting the house. You must also modify the player movement code to prevent movement through wall tiles. Add or modify code in an appropriate place so that the player is placed in the bed tile after the level is generated. For top marks in this task your code should generate the farmhouse dynamically and should avoid overwriting the dirt tiles placed in the previous level generation task.

Task 9 – Growth and even better movement (5 marks)

You must add code to implement the `growCrops()` method, and the `evenBetterMovePlayer()` method. The `growCrops()` method must traverse the `level` array and change any sowed dirt tiles into crop tiles. The `evenBetterMovePlayer()` method must be completed (again leave the previous move methods unchanged or you will lose marks for those tasks) to implement a new check for

the player moving into a bed tile. If the player moves into a bed tile then you must call the `triggerNight()` method. For the highest marks, the `evenBetterMovePlayer()` method should be modified to improve the efficiency of the movement code, e.g. to remove duplicated code and create new methods with re-usable functionality that improve the clarity and readability of the movement code (which by now is likely to be quite complex).

Task 10 – Harvest (5 marks)

You must modify the latest player movement code so that if the player moves into a crop tile, that tile is changed to a dirt tile and the value of the `money` attribute of the GameEngine class is increased. Print a message to the standard output informing the user of the new value for the money attribute using an appropriate and user friendly format for the text.

Task 11 – A pest appears (5 marks)

Modify your `growCrops()` method so that if one or more crop tiles are added to the level, then a `Pest` object is created, and then assigned to the `GameEngine` `pest` attribute (and therefore is placed in the level when displayed; this happens automatically if the attribute is assigned a `Pest` object). Implement the `movePest()` method so that the pest moves towards the nearest crop tile whenever the method is called. You will need to check if the `GameEngine` pest attribute is `null` or not before attempting to call methods on it (e.g. to get its position). If the pest moves into a crop tile, that tile should be replaced with a dirt tile. Modify the player movement code so that if the player moves into the same tile as the pest, the pest is removed from the game (by setting the `pest` attribute to `null`).

Task 12 – Realistic farmland (5 marks)

Modify the latest level generation code so that a more interesting and dynamic level is generated each time the game is started. You can add different tiles to the level and should aim to implement interesting and efficient solutions that make the farm look more realistic. Add `Tree` and `Rock` objects to the level by using the `debris` array; check the documentation for the array and the two classes. Modify the player movement code so that the player cannot move through rocks or trees. You will need to traverse the debris array to check for tree/rock positions when moving the player and prevent movement if one exists in the tile the player is trying to move into. Modify the level generation code to add the axe and pickaxe objects to the game using the appropriate tile types, which (when held by the player) allow them to cut trees or break rocks when attempting to enter tiles that contain a tree or rock.

Task 13 – Realistic agriculture (5 marks)

You must modify the relevant method(s) so that crops may fail to grow from seeds overnight, and instead will revert to tilled dirt tiles. Tilled dirt tile should also have a chance to degrade to normal dirt tiles overnight if no seeds were planted in them. Pick sensible odds (the chance for this to happen) for these effects so that they can be easily observed during testing without making the game overly difficult.

TASKS 14-19 CAN BE COMPLETED IN ANY ORDER AND ARE MORE COMPLEX AND DIFFICULT THAN PREVIOUS TASKS!

Task 14 – Durability problems (5 marks)

Modify your code so that tools have a chance of breaking when used, and the seed bag begins to empty each time a seed is planted. Trees and rocks should now take several "hits" to break. Modify the game interface and the relevant classes so that a visual effect can be seen for partly cut trees and partly broken rocks, and the tools/seed bag change appearance depending on how full or broken they are. You may need to add attributes for trees and rocks to store their durability. New seeds and tools can be collected in the normal way.

Task 15 – Non-player characters (5 marks)

Add non-player characters (NPCs) to the game that can appear each morning. Write your code so that one is guaranteed to appear on the second day. NPCs should slowly move around the farm. When the player attempts to move into the same tile as one, a graphical user interface dialogue box appears with some conversation and response choice(s) for the player to select. The choice(s) should have some impact on the game, perhaps modifying the farm or some attributes. **You may use the standard output (i.e. println statements) to implement this feature, but will get less marks than using GUI components.**

Task 16 – Weather and water (5 marks)

Add a watering can and the requirement that sowed tiles be watered (with visual changes) before they can grow into crops. Add two or three types of weather to the game that have an effect on the game mechanics, with one type of weather being rain that automatically waters sowed dirt tiles without the player having to do this.

Task 17 – Farmhand (5 marks)

Add an NPC to the game that works on the farm. This farmhand should move around the farm to till all untilled dirt tiles. If all dirt tiles are tilled it should begin sowing seeds in tilled dirt tiles. If all tilled dirt tiles have been seeded it should begin harvesting crops. Attempting to move into the same tile as the farmhand should produce an interesting effect with feedback to the user via a sensible mechanism.

Task 18 – Shopping (5 marks)

Add a new tile type to the game – a signpost. When the player moves into the signpost a new level array is generated that contains a shop. You will need to save the original level array to re-load it later. The shop should sell things that add to the complexity and depth of the game: new items, tiles, crop types, farmhands (if implemented) etc. These should cost a varying amount of money that is deducted from the players available money. New items should be added to the farm when the player returns by loading and modifying the originally saved level array.

Task 19 – Expanded farm (5 marks)

Modify the game code so that moving off the edge of a level loads a new level with dynamically generated content. The original level, and any generated levels, should be saved/stored in a sensible way so they can be reloaded when the player attempts to return to them by moving off the edge of the current level. The player should be able to farm dirt in the new levels, and any farming effects (crop growth etc) should be applied to all levels that have been generated so far. For maximum marks, pests should be able to move across levels to the nearest crop and NPCs should be able to move between levels as well (this part is hard; make a plan and an algorithm design first).

Task X – Adde plura, plus mereberis (10 marks)

Note: You should only attempt this task when you have completed most, or all, of the earlier tasks.

Add new features to the game that implement interesting, complex, fun or unusual ideas. You are encouraged to discuss these with the module co-ordinator. Remember that this is **not** a coursework about creating a game, it is a coursework about **computer programming**. Ensure that your ideas demonstrate more advanced and complex programming skills to gain more marks. For example, changing the images (tile pictures) in the game demonstrates only very limited coding skill at best, so is unlikely to gain you many marks for this task. Achieving high marks in this task will require demonstration of complex and well executed programming concepts, either through great depth/complexity of the feature(s) added and/or use of advanced programming techniques and language features.