Universal Task Requirements

This page covers general requirements which apply to all tasks in the unit. If your submission does not meet these requirements you will be asked to fix and resubmit the task.

Why do we require this? Being able to follow a coding, writing, or other standard is an important skill. Different projects and teams will have different rules, and although you may not always agree with them, the benefits of consistency far outweigh any drawbacks. Even if you are working on your own, future you will thank you! We have also embedded good practices and other skills expected by industry in these requirements to encourage you to get into some good coding habits.

If you have any questions or clarifications, please do not hesitate to talk to your tutor.

Submissions

- All tasks must be submitted as a PDF file. All students at Swinburne have access to the Adobe tools, which includes Adobe Acrobat that also you to combine a set of PDF files to into a single PDF document.
- We regularly ask you to provide screenshots of your code, output, and working environment. On Windows, use the combination **Windows key + Shift + S** to activate the snipping tool and choose the desired mode (i.e., rectangular mode or window mode). On macOS, use the combination **Shift - Cmd - 5** for window mode or **Shift - Cmd - 4** for rectangular mode.
- Screenshots provided in pass and credit level task submissions need to show all relevant functionality of the task, working on your machine.
- All portions of your submission (written answers, code, comments, file names etc.) must be written in English.
- Each submission must include everything relevant. We will not look through past submissions. So, if you are asked to fix and resubmit something, do not just re-upload the things you had to fix.
- All code must compile and run.
- Task submissions must demonstrate that every instruction in the task PDF has been correctly followed (e.g., logic matches pseudocode, classes and collaborates match provided designs, all questions are answered correctly etc.).
- All check-in tasks must be demonstrated in person to your tutor, during the semester. Other tasks can be signed off without a demonstration, unless one is requested by your tutor.
- Tutors may refuse to sign off a task as Complete if you cannot appropriately explain your work. This is particularly important for the “In Person Check-in” tasks.

Code Style

These requirements relate to the layout of the syntax and logic in your code.

Files

- Use PascalCase when naming files and folders - e.g., “MyFolder”, “SomeClassFile.cs”.
- Wherever possible, one class definition per file.
- File names should match the name of the class contained within — e.g., a file containing the class `MyClass` should be named “MyClass.cs”.

Code Layout

- One statement per line.
- One blank line between method and property definitions.
- Use Allman style braces, where each curly bracket is on its own line and statements within curly braces are indented one more level.
- Preferred indentation is 4 spaces. You can use any sized indentation you like (within reason) as long as it is applied consistently.
- `using` declarations at the top of the file, before the namespace declaration.
- Use brackets to clarify the intended order of operations in expressions — e.g., `if ((a > b) && (c < d)).`
- Organize the elements of a class in this order:
  1. static, const, or readonly fields
  2. other fields
  3. properties
  4. constructors
  5. methods
- Modifiers are written in the following order: `public protected private new abstract virtual override static.`

Naming Conventions

- Use PascalCase when naming a class, record, or struct — e.g., `ThisIsAName.`
- When naming an interface, use PascalCase but add an “I” at the start — e.g., `IMyInterface.`
- When naming any methods, or public fields and properties, use PascalCase — e.g., `public bool MyBool.`
- When naming private fields, use lower camelCase, and add an underscore at the start — e.g., `private bool _myBool.`
- When naming parameters, use lower camelCase — e.g., `public void MyMethod(int someValue).`
- When naming constants, use ALL_UPPER_CASE.
- Always specify an element’s access modifier — e.g., `public int x`, not `int x.`

Choice of Abstraction

- Use `List` over arrays, unless the size of the container is both fixed and known at the time of construction.
- Use the most appropriate type of loop for the context — e.g., always use a `foreach` loop to iterate over a collection, unless you specifically need to know the positions of the elements.
- Avoid unnecessary use of `== true` in boolean conditions — e.g., `if (a > b)`, not `if ((a > b) == true).`

Development Principles

- Follow the DRY principle, by avoiding duplication of logic and data where practical.
- Methods should not have hidden or unnecessary side effects. That is, a method should only modify data or change state that needs to be changed. For example, a method for calculating a value should calculate and return that value, not also print it out.
- All identifiers should be meaningful and concise.

Unit Tests

- All unit tests must pass.
- Each unit test file should contain one class, which contains tests for one class from the core project.
- Name unit test files and classes according to the class they are testing — e.g., a class containing tests for “MyClass” should be named “MyClassTests” or “TestMyClass”.
- Unit tests must show an appropriate use of the `Setup` method.
- All unit tests must demonstrate appropriate selection of `Assert` statements — e.g., if you are testing that two values are equal, use `Assert.AreEqual(a, b)`, not `Assert.IsTrue(a == b).`
- Unit tests need to demonstrate appropriate coverage of the functionality of the class being tested.
- Each unit test should test one small piece of functionality, or one “use case” of a method.

Unit-Specific Requirements

These requirements are not likely to be seen in industry. We have added them because we need you to demonstrate your learning and understanding of certain concepts. Be aware that examples you find online are unlikely to adhere to these requirements.

- Use of `var` is not allowed — only use concrete data types — e.g., `int x = 7`, not `var x = 7.`
- Use of lambda expressions is not allowed in the pass and credit level work.
- Use of ternery statements is not allowed in the pass and credit level work.
- Initialize new objects using long-form syntax — e.g., `List<int> myList = new List<int>()`, not `List<int> myList = new().`