1 Introduction: what is a report?

As you know, in this School we ask you to do a lot of programming: exploring and experimenting with different aspects of coding. Along with these, we ask you to write reports: brief documents to go along with the code and **explain** various aspects. They should be **a few pages** long - anywhere from one to a dozen pages depending on the practical and your own style, and should be anonymised just like the code.

The contents of your report are **flexible**, and the format is largely up to you. Different tutors will have different preferences for what they expect, so here we will simply discuss the essential features which most agree a good report should have.

2 Motivation: why write reports?

This is a degree in Computer Science, so why are we asking you to write reports?

Well Computer Science **isn't just programming**: in the real world, you'll often need to interact with customers, teammates, supervisors and others in a variety of ways. This can range from explaining how exactly to interact with different features, to more mundanely explaining concisely what work has been completed. During this degree, we hope you will learn and demonstrate these skills while also giving your markers an insight into how you see your code.

Helping your markers to understand how you work with code is itself very **important both for them and for you**. A marker knowing why you made a given choice / mistake / flourish can influence whether it has any impact on your mark. For example, a marker will generally frown on extremely long classes that could be split into two or more, but if you mentioning that keeping the class together was a deliberate decision for solid reasons - to allow, for example, for large amounts of internal data to be easily shared by all methods - could mitigate any penalty. On the other hand, if the deliberate decision you took was based on a misunderstanding - maybe you missed an easy way to share data between the classes - then your marker may be able to change their feedback to more helpfully and specifically address that.

3 Presentation: how should it look?

When writing your report - whether for a marker, client, teammate or anyone else - bear in mind it will be read by someone to whom you want to **give a good impression**. A marker who can't easily find the relevant details because of unreadable grammar or unstructured walls of text may assume those details aren't there. A client who believes that you've put no effort into a report - which may be the only thing they understand - may simply not want to hire you. As such, it's important that your report looks professional.

There are many simple ways to achieve this. One of the very simplest is to provide the **basic information** on what your piece of work is. This obviously means including module code, practical number and date for each piece of work, but some practicals will have extra details. If you're working in a team, for example, remember to list the team name/number. This avoids confusion when a marker has many different submissions to sort through, or when you look back later in the semester at work you're doing now.

Remember when doing this to follow any **additional instructions** you've been given - most notably, the rule of **anonymisation**. It's easy to lose a lot of respect from your reader if the first thing they see is your name or username: the first impression they get is of a student who can't follow the most basic of instructions.

One easy way to make your report much clearer is to split it up into **sections**. The work you've done on designing your code is probably entirely unrelated to the testing you've carried out, so why jumble it all together in one long and confusing block? If the marker wants to check for a specific detail, such as how you tested a particular feature, they shouldn't need to read through your entire report to find it. In practice, if they do, they might simply miss the place where you mention it and assume you missed it entirely.

To make your report appear professional, try **formatting it appropriately**. This can have a significant effect on the reader's general impressions, and is particularly easy to accomplish. Most word processing software will do it for you: Microsoft Word, Pages, LibreOffice and LaTeX all offer styling to make your headings and metadata stand out - and even tables of contents in case your report gets especially long.

Finally, and most importantly: **proofread your report**. When noting down a feature mid-coding, or simply writing down whatever's in your head when everything's done, you may well miss important details or explain in a way that doesn't actually make as much sense as you think. These problems can be big or small, from entire unintelligible sections to grammatical ambiguities that make your point unclear, but they always harm your report and are very easy to fix. Bottom line: regardless of how perfect you think your report is initially, it's almost always worth the effort of looking over it all again.

4 Timing: when should I write?

It can be very tempting to leave your report to the very **last minute. Don't** give in to that temptation, as it will almost always damage your report. Coding is notorious for taking longer than expected, especially for inexperienced programmers, and if you plan to start writing your report a few hours before the deadline, it may well end up being done in the last few minutes. It is very difficult, if not impossible, to write a good report in those circumstances.

Even without time constraints, it's generally far easier to write your report **alongside your design & coding** than all in one block at the end. Quite simply, it's easier to note down the structures, extensions, decisions and difficulties you encounter when you encounter them, rather than hours or days later. You may also find that writing down your **design decisions early on** makes them clearer in your own mind, making the task of implementing them noticeably easier and improving the quality of your resulting code.

Not everything will be worked on throughout your project, of course. Any **overview** of your code, or broad impressions of the practical as a whole, must necessarily be written once you've finished working on the specific details. Some things, also, are best dealt with early on so you don't have to think about them at more time-critical later stages: **basic information**, for example, such as matriculation number, module and so on. You could even write a **template** structure at the start of a semester which will save you having to write these each time you start a new piece of work.

5 Content: what needs to be included?

There are lots of things to think about when writing your report, but the most obvious is: what should it say? Let's start with what it shouldn't say: your report **should not directly explain your code**. When trying to explain design decisions, it can be tempting to explain how method X takes two parameters and calls method Y which then uses class Z to... and so on. This is not helpful for a marker simply because your code is attached in any submission, and will provide a more detailed, concise and accurate description than a report ever could. If it isn't entirely clear what a method/class does, you can address this with improved naming or by adding comments to the code, rather than relying on the report.

While it isn't helpful to simply paraphrase your code, **important choices** about why the code does what it does can be very helpful. If you're implementing a game of Chess, did you choose to represent pieces as classes (e.g. Pawn), strings ("pawn") or integers (3)? Why did you decide that was best, and would there have been merits in choosing a different system? These **design decisions** allow you to demonstrate your understanding and analysis of the various possible solutions to the problems you encounter. Note that **not *every* decision** needs to be mentioned: depending on the practical and the complexity of the code, many may be unimportant. For example, if asking the user to make a choice, you can ask them to enter a number (e.g. 5 for Exit) or a string ("quit"). This may be important in early practicals or when focusing on the user experience, but otherwise it may not affect the program significantly.

Bear in mind that your code needs to be run by someone. As such, if you have any unusual features about how your program is run - perhaps there are specific parameters that need to be passed, or simply lots of files but only one that should be executed - then give all details necessary on **how to run** your program.

Of course, while your program must be run by someone, that person isn't perfect. It's always possible that a marker will miss an aspect of your code that you wanted them to see. As such, it's always sensible to highlight in your report the **extension work** you've done, to be sure this will be fully taken into account.

Just as the marker is not perfect, chances are that neither are you. If you had any **difficulties** with the practical, mentioning these can be helpful in a variety of ways. The clearest is that the feedback you get on your report can then specifically address the areas you most need it to. This kind of feedback can be some of your most useful input, as it's tailored specifically for you. If you didn't manage to solve any particular problem, then maybe your tutor will suggest ways to do so next time - and even if not, they may be more lenient on any lack of functionality if it's clear you tried to find a way through it. Even if you did manage to solve the problem, it may be that your solution - which they might have overlooked if it wasn't mentioned - impresses them.

Last but not least, one aspect of your program can be especially difficult to demonstrate within the code: **testing**. While you may be confident that your program does what you intended it to do, how have you made sure of that? Testing can be at any level of your program, from making sure individual methods return what you expected through to playing with the user interface to check it all makes sense. Remember to check for edge cases: for example, if you're asking for the size of an array, what if the user enters a string? Or a negative number?

The **way you show** your testing is up to you: screenshots are favoured by some, or console copy/pastes, or even full frameworks like JUnit. For some programs, a table structure may be helpful, with rows for methods/features and columns for input, expected results and actual results. Note that while it may be difficult to think of **things to test** in simple early practicals, later on this will become one of the most important features of your report.

6 Marking: how is the report weighted?

The report is an integral part of the coursework you submit, but **does not have** a single, simple importance. There are a number of ways a good report can directly affect the view a grader has on your work, both positively and negatively. However, even apart from these a report can have a broader effect on your mark: both limiting it in the case of a report of general poor quality - *even if* your code is excellent - or slightly boosting it in the case of an exemplary one.

7 Sample structure

As stated earlier, the structure and style of a report is primarily up to you. However, for most practicals there are a few **fundamental features** which the report should contain. While in many cases - especially early in the degree - it will be possible to merge some of these together, you should probably have at least two or three separate sections such as:

Introduction - what was the broad outline of the practical?
Design - how did you approach the practical and what high-level choices did you make?
Implementation - any lower-level details to do with your coding.
Testing - how did you make sure the program does what you think it does?
Difficulties - what went wrong or was unexpected?
Conclusion - any general thoughts on the practical and how you approached it.