Informatics 1B: Object Oriented Programming in Java

Code Quality & Coding Conventions

Original author: David Symons
Tweaking: Brian Mitchell
Version: 0.9 (light)
(known bugs in page headers,
some wording may need revising)

School of Informatics
The University of Edinburgh

This document forms part of the 2023 Inf1B assignments. You are expected not just to read it but to make use of it. It takes time and repetition to make the advice second-nature, so keep referring to this document when you are coding or designing. Knowledge of information from specific parts of this document might be examined in Assignment 1, whose instructions will say which parts. The whole document helps with Assignments 2 and 3, and with programming generally.

This document is available in different colour schemes in the assignments area of the Inf1B Learn course.

If you need a version with a different layout please post a private question on Piazza explaining your needs such as font choice, font size, line spacing, or no hyphenation.

Some time after this document is released there will be a single official thread on Piazza for asking for clarifications and reporting possible errors (please quote the line number).

Table of contents

1 Introduction 1

2 Naming done properly 2
2.1 Descriptive naming . . . 4
2.2 Naming conventions . . . 5

3 Layout 6
3.1 Indentation rule . . . 8
3.2 Braces . . . 8
3.3 Parentheses . . . 9
3.4 Whitespace . . . 9
3.5 Width . . . 10

4 Code documentation and commenting 10
4.1 What to comment . . . 10
4.2 How much to write . . . 11
4.3 Comment quality . . . 11

5 Program design 13
5.1 Structure . . . 13
5.2 Data representation . . . 13
5.3 Modularity . . . 14
5.4 Avoid code duplication . . . 15

6 Robustness 15
6.1 Validation . . . 16
6.2 Access modifiers . . . 17

7 Technical/language proficiency 18
7.1 Use appropriate language features . . . 18
7.2 Avoid wasting resources . . . 19
7.3 Know the API . . . 19

8 Code idioms 19
8.1 Initialise class fields . . . 19
8.2 Magic numbers . . . 20
8.3 Magic strings and the end-of-line . . . 25
8.4 Importing classes . . . 27
8.5 Ordering of declarations . . . 27

List of tables

1 Orthographic conventions for Java identifiers . . . 6

A note on the text

It is possible, even likely, that some of the advice, recommendations, or programming style in this document differ from those you encounter elsewhere. Do not worry about this. Keep an open mind, update your opinion, and use your judgement.

1 Introduction

Wherever more than one person works on a code base (basically in every programming job) all involved will have to agree in advance on specific coding conventions for how to write code, **and** then **adhere** to those conventions — in some cases keeping your job may depend on it. This coding conventions aspect of programming is separate from writing functionally correct code — that is enforced by the programming language and its compiler — but it is perhaps equally important.

Coding conventions comprise a set of guidelines governing programming style, including but not limited to white space generally and indentation specifically as well as comments, declarations, statements, naming conventions, and error handling. The exact specifications of all these are debated but to ensure consistency between different people working on the same source code, a common ground must be specified. Besides any given organisation's in-house style, there are also well-known conventions which are widely followed, for example from Google[1] or Oracle.[2] For Inf1B the default formatting provided by IntelliJ is fine: you are also allowed to customise the settings. The main aim for now is internal consistency within a single code project, no matter its size, from the trivially short to the impressively long.

This document contains coding conventions used for code you produce during the Inf1B course, particularly for the assignments. The style is based on the guide in the "Objects First with Java" text book.[3] On Inf1B we are not overly prescriptive about indentation providing the indentation makes the code easy to read. On Inf1B we are much more concerned that you are naming things clearly and usefully and that your code is formatted consistently. However not all the code that you encounter will be within our guidelines, some deliberately so. Therefore be prepared to reformat and maybe rewrite code.

1 https://google.github.io/styleguide/javaguide.html

2 https://www.oracle.com/technetwork/java/codeconvtoc-136057.html

3 David J. Barnes & Michael Kölling, Objects First with Java, Sixth Edition, Pearson, 2016, ISBN (Global Edition): 978-1-292-15904-1

2 Naming done properly

Keywords such as **`int`**, **`for`**, **`while`**, **`if`**, and **`return`** must always be written exactly as specified by the language you are using. This is non-negotiable because the programming language has the final say. There is more leeway with identifiers such as the names of classes, methods variables, and constants. These can be named freely within the constraints of the programming language. However to be a good programmer it is not enough for identifiers merely to comply with the language's rules: as a minimum they must also be informative and accurate; within a professional environment they must also be compliant with the coding style rules — something you should eventually aim for in your own code, especially in group work.

Code is called self-documenting when the names we choose are insightful and make the code easier to read. There are different ways a good programmer can give hints about what classes or methods do and what a variable is used for.

Generally you should avoid negative names, for example a function called `hasNoData()` or `isNotEmpty()`. Negative names can be harder to understand than the positive equivalent, especially when the named item is associated with a boolean value because it can easily become counter-intuitive in the case of a double negative which is when a boolean with a negative name is set to false. You have probably seen negatively worded options in dialogue boxes such as "Don't show full menus". This might translate to a boolean variable `dontShowFullMenus` which when set to **`false`** means "do show full menus." It is far clearer to use the positive version: "Show full menus" → `showFullMenus`. When you want the negative, for example when there is no data, just use `!` the logical not operator: **`if`** `(!hasData())` or **`if`** `(!isEmpty())` instead of something with a negative name like `isNotEmpty()`.

Struggling to choose a suitable name for a variable, method, class, or other item is a common problem. When this happens, it often means you are unclear exactly what the purpose of that whatever-it-is should be. This strongly suggests you need to clarify the design before you type any more code. If you find yourself needing to use And in the name — `saveAndPrintAndCloseDocument()` — this suggests you have not refined the design sufficiently. While you very often need methods and functions to perform more than one step, the identifier should reflect a single, coherent name appropriate for the entire logical content such as `makeTea()` (see Listing 1). This is not cheating! It is good design: indeed it probably reflects suitable application of stepwise refinement, see section 5.1. The overall accomplishment of the variable, method, or class is reflected in its name and this should be as independent as possible of the actual steps involved. This is because those steps might need to change in future even though the aim remains unchanged.

I could not easily think of a more appropriate name for the method `saveAndPrintAndCloseDocument()` which suggests saving and printing and closing should probably be three separate methods. However if you feel strongly that it would be useful to have the word processor you are writing do those three actions together in order to finalise a document in a single user command, then you could call the method something like `doFinalDocumentActions()` and have a brief comment that outlines what it does. Do that preferably as JavaDoc so it can appear in the automatically generated documentation of your program code.

Hopefully during most of the previous paragraph you were thinking saving and printing and closing a document should be three separate methods anyway. You are correct. If this was not your experience, go back and have another try. Now we are all agreed it is good design to have those three methods, the question remains whether to have an additional method that does all three with a single method call. If so, then its body would mainly be just calling the other methods in the appropriate order.

```java
/**
 * Simulate making a cup of ("English") tea with additions.
 * Quantities are ml for liquids, teaspoons for sugar, and tablets for sweeteners.
 *
 * @param plantMilkVolume (vegan)
 * @param cowMilkVolume
 * @param sugars
 * @param sweeteners
 * @param cupVolume
 */
public static void makeTea(int plantMilkVolume,
                           int cowMilkVolume,
                           int sugars,
                           int sweeteners,
                           int cupVolume) {
}
```

Listing 1: One way to specify a fuller, flexible version of `makeTea`

It is also common to have to rename one or more identifiers part way through development once the exact usage becomes clear, or perhaps just clearer for now. Use the Refactor > Rename facility of IntelliJ so that all the relevant instances are renamed consistently without renaming other things with the same name and risking introducing inconsistencies or even errors. For example it is common to use `i` as the loop variable in different loops throughout the code but renaming `i` in one loop should not unintentionally rename different variables with the same name in other loops, or worse change every occurrence of the character `i` wherever is appears. Nonetheless a more descriptive name for the loop variable is encouraged on this course. For example if you were iterating over an array of numbers that represent grades, then the loop variable could be `grade` in an enhanced **for** loop, or perhaps `index` or `gradeIndex` in a traditional **for** loop. It is possible in a real world situation that the coding convention you must work to encourages or even requires loop variables to be a single letter.

2.1 Descriptive naming

Use descriptive but concise names for all identifiers (variables, constants, methods, classes, interfaces, and enumerations). Only use abbreviations if they are extremely well known, for example "nr" for "number" or "max" for "maximum".

Simple mutator methods ("setters") should be named `setSomething()` while simple accessor methods ("getters") should be named `getSomething()`. Accessor methods with boolean return values often start `is` or `has`, for example `isEmpty()` or `hasData()`. IntelliJ can create getter and setter methods (and constructors) for you but only for field variables and class variables that already exist.

Single-letter variables names should not be used, but there are some exceptions from sheer weight of historical usage. You can use `i`, `j` and `k` as integer counters in loops:

```
for (int i = 0; i < 10; i++) {
    for (int j = 0; j < 10; j++) {
        // do something with i and j
    }
}
```

You can use `x`, `y` and `z` for coordinates as these match the conventional names used in geometry. If you find it difficult to remember the exact purpose of any identifier, especially a single-letter variable, then you could temporarily rename it while you are working and restore the original name afterwards. You will need the Refactor › Rename command to do this safely.

2.2 Naming conventions

Generally in programming, largely irrespective of the programming language, it is vital to be able to recognise the difference between lowercase and uppercase letters from the Roman alphabet. If your main natural (human) language does not use the Roman alphabet and especially if your main language does not use an alphabet at all, then you must make extra effort both to passively recognise and actively use upper- and lowercase letters. Many programming languages are case sensitive so an identifier called `width` might be considered different from identifiers called `Width` and `WIDTH`. The way a word is written using upper- or lowercase is its orthography. This is different from the spelling of a word, which is the choice of letters (irrespective of case) to represent sounds.

Because identifiers frequently need to encapsulate a non-trivial concept, they would frequently be described in a natural language such as English using multiple words — if you are a native German speaker, please stop looking smug. It is rare for programming languages to allow multi-word identifiers, so the typical solution is to delete all the white space and punctuation between the natural language description. The effect is to smash all the component words into something that by default is often at best hard to read; though if you are lucky also unintentionally funny. In Java, the way to make the single word identifier more human-friendly[4] is to adapt the orthography. This adaptation must meet the programming language's conventions.

4 The compiler is unaffected because it is not trying to understand them.

Class and interface names are written in **PascalCase**.[5] That means that every word starts with a capital letter and all other characters are lower case. Examples: `MenuBar`, `GameSettings`

5 Named after the convention in the programming language of the same name, itself named after Blaise Pascal.

Method and variable names are written in **camelCase**, alluding to the hump in the middle of a camel's back. Camel case is like PascalCase except that the first letter is always in lower case. This small but extremely important difference is, for humans, the primary way to distinguish classes from class instances. Compare `menuBar` and `gameSettings` with the identifiers in the previous paragraph that

Name(s)	Description	Usage	Examples
PascalCase	Every internal “word” starts with a capital letter and all other characters are lower case	class names interface names	`MenuBar` `GameSettings`

camelCase	As PascalCase except that the first letter is lower case	method names
		variable names
			`menuBar`
			`gameSettings`
			`defaultSettings`
			`highScore`
			`totalCostOfShopping`

SHOUT_CASE
SCREAMING_SNAKE_CASE
	All capitals with words separated by an underscore	constants	`MAX_WIDTH`
			`NUMBER_OF_ROWS`
			`NR_OF_ROWS`

Table 1: Orthographic conventions for Java identifiers

have the same spelling but different orthography. The identifiers starting with a lowercase letter variables are probably variables that use an instance of a class of the same spelling. Other examples: `highScore`, `totalCostOfShoppingBasket`.

Constants (**`final`** “variables” whose value can never vary once it has been set) are an exception. Constants are written in **SHOUT_CASE**, also called **SCREAMING_SNAKE_CASE**,[6] and those two names are examples of names written in that style: all capitals with underscores separating words to enhance legibility. Other examples: `MAX_WIDTH`, `NUMBER_OF_ROWS`, `NR_OF_ROWS`. Note that an identifier written in English as “row number” seems much more likely to be a variable (probably as a parameter or loop index) than a constant. If a variable, it would be written `rowNumber` (camelCase) whereas the constant would be `NUMBER_OF_ROWS`. Note the differences in terms of orthography and meaning and how orthography and meaning are interlinked in naming identifiers. This can be extra difficult for non-native speakers of English.

6 For example in the String Manipulation plugin.

3 Layout

Consistent formatting throughout the entire project is essential, even indeed especially when there are multiple authors whether concurrently or over time. Indentation, braces and parentheses can help the reader understand the code. Code is read (much) more often than it is written. That might not be the case when you are developing your own code for a university assignment done on your own, but it will be the case for a university group project (you hope) and throughout a professional programming career (you hope). Easily readable code is critical for success. Not only does it help someone else debug or extend your code at a later date, it helps you write better code in the first place. Clean, well-written code saves everyone involved (whether now or later) time in the medium and long term, and often in the short term.

Layout done wrongly can be misleading. Consider the following example, known as the dangling else problem:

```
if (condition1)
    if (condition2)
        System.out.println("A");
else
    System.out.println("B");
```

The indentation in the above snippet suggests that `"B"` is printed if `condition1` is **`false`**. However the way Java is implemented, the **`else`** clause belongs to the last **`if`** statement. So, if `condition1` is **`false`**, nothing is printed. This is a simple example to highlight the point. Imagine if that happened in a safety critical where `condition1` one is supposed to result in a message saying that things are about to get "interesting."

The following shows the exact same code with different indentation. Unlike some other languages such as Python and Haskell, Java does not infer meaning from indentation (it actually ignores it), meaning that the Java code below will give the same result as the code above (whereas the Python or Haskell equivalent would not).

```
if (condition1)
    if (condition2)
        System.out.println("A");
    else
        System.out.println("B");
```

For a human reader, the second version is much clearer.

Whereas Python and Haskell rely on indention to group code into logical blocks, Java uses braces, also known as curly brackets or tautologously as curly braces. In Java braces are required to group multiple statements together. Technically braces are not needed for single statements but the use of braces around single statements can massively improve readability and thus reduce misunderstanding and error. The following code is unambiguous and can be read much more easily:

```
if (condition1) {
    if (condition2)
        System.out.println("A");
    else
        System.out.println("B");
}
```

or if you prefer the other style of brace placement:

```
if (condition1)
{
    if (condition2)
        System.out.println("A");
    else
        System.out.println("B");
}
```

IntelliJ has a built-in command to reformat your code to a preset configuration, which can be customised. It can also put braces or parentheses around source code that has been selected: select some source code and type an opening brace or parenthesis.

3.1 Indentation rule

All statements within a block are indented one level deeper than the block itself. Use tabs for indentation.[7] If you use multiples spaces, you can easily miscount or delete one, leading to statements not being properly aligned. It is probably also slower to edit. A decent editor or IDE can automatically convert freely between spaces and tabs, so you can configure it for your convenience and to comply with coding conventions. The IntelliJ plugin IndentRainbow can help you visualise correct indentation alignment.

7 Tabs versus spaces is a common and often tribal discussion point among programmers.

3.2 Braces

There are two common styles for where to open and close braces. The one shown on the left aligns opening and closing braces vertically. This way, you can visually match pairs of braces. The style on the right is more vertically compact, but braces are not aligned and it can look as though there are more closing braces than opening braces.[8]

8 TODO code

If working for a company, you will probably be told which style to use, but either are fine for this course. Note that's either (one at a time) not both (in the same project). Once you have settled on a style, stick to it and apply it consistently. Configure IntelliJ according to your wishes modulo the coding conventions. Often, consistency is more valuable than any other attribute of the code quality except of course correctness and fulfilling the task requirements.

3.3 Parentheses

Java "knows" that multiplication and division have higher preference than addition and subtraction, but it is often helpful to use explicit parentheses to improve readability. For example:

```
int result = (5 * 4) + (2 * 11);
```

is clearer than:

```
int result = 5 * 4 + 2 * 11;
```

IntelliJ's context-sensitive suggestions (the light-bulb, or Alt+Enter) can help you. This clarity of this example code can be greatly increased further by avoiding magic numbers, section 8.2.

3.4 Whitespace

Use a space either side of operators to make them easier to see:

```
int result = (5 + 2) * 6;
```

and not

```
int result=(5+2)*6;
```

Do not leave any large gaps of consecutive blank lines in your code. Instead use exactly one blank line to separate logical blocks of code:

- use one blank line between methods (and constructors);
- use a blank line to separate logical parts within a method: think of this as separating the method into paragraphs (self-coherent sub-steps).

In order to learn how to paragraph your code well, you need to read a lot of quality code written by others. This is exactly the same as reading to improve your knowledge of a foreign language. Study others' code for the formatting, the paragraphing, the naming, and the structuring. This is a major way for you to improve your own practice. You can do this in explicit steps: for example first look at indentation, then look at paragraphing, then at naming. The order is your personal choice and may vary according to your changing needs. When you are a beginner or novice programmer, you must read a lot of code to expand your knowledge, just as you must write a lot of code to retain that knowledge.

3.5 Width

First it is important to distinguish a line from a Java statement. A statement can occupy multiple lines. Lines in IntelliJ are numbered down the left-hand side of the source code.

Whereas a statement is as long or short as it needs to be, conventionally a line of code should be at most 80 characters wide. This is a tradition stemming from the days when 80 characters was the maximum that typical computer screens could display. Although that technical limitation has long since been overcome, the convention persists. One major reason for this is that 80 characters is comfortable for human reading, especially as leading white space typically makes the actual text of the code narrower.

You are expected to keep your code to a maximum width of 80 characters. You can (and probably should) configure IntelliJ to do a hard wrap at column 80. Learn to use the IntelliJ command, preferably by its shortcut key, that can join lines. This speeds up handling situations where the wrapping is not what you want.

4 Code documentation and commenting

4.1 What to comment

Comments should be used wherever you are able to provide additional information that is not easily seen from the code itself. The following should always be commented, preferably with JavaDoc comments:

- every class should have a (JavaDoc) comment at the top, explaining its (general) purpose;
- all static constants and (non-static) class fields should have a (JavaDoc) comment;
- every method should have a (JavaDoc) comment stating what it does.

In some cases, you may want to add single-line or block comments (not JavaDoc) to explain individual lines within a method.

This is useful to clarify more complicated bits of code or to state what is being calculated. For example:

```
// Compute the sum of potential and kinetic energy.
float totalEnergy = mass * gravitationalAcceleration * height
                    + 0.5 * m * velocity * velocity;
```

In the example, the identifier names alone should suffice for another programmer knowledgeable about the topic. However not all programmers are physicists or mechanical engineers. So it might be useful to put a version of the equation in the comment or a link to an external reference that explains it. If the latter, where possible try to use the internet archive (“the way back machine”) for an archived version of the link so that the link has an increased chance of still being available in future. Direct links seem to expire one week after the original programmer leaves the company.

4.2 How much to write

Comment as much as you feel is needed for another intelligent programmer to be able to understand your code. It is safer to slightly underestimate your potential readers but you can assume that anyone inspecting your source code to understand it has a reasonable ability level. Comments also serve to ensure you yourself can understand your own code when you come back to it years later. Ideally, there should be a lot of information in a small amount of text. Get straight to the point and condense the essential information into a clear comment. As with writing in any language, including natural languages, you should probably have at least one attempt at rewriting the comment (meaning your second draft) to improve it.

4.3 Comment quality

Comments must provide additional insight into what is happening in the code. They should not just re-state what the code is doing. In the example with the potential and kinetic energy above, it would be pointless to comment “Mass is multiplied by the gravitational acceleration and the result is again multiplied by the height. Then we add half the mass multiplied by the velocity squared.” All this can be seen immediately from the code, thanks to an excellent choice of identifier names. What is not necessarily clear is what that equation is actually computing. The comment in the original examples provides that additional information.

Do not clutter simple bits of code (that are self-documenting) with comments. Plastering code with comments is a waste of everyone's time: yours writing them and everyone else's reading them. An example of such clutter would be to comment a method `setValue(int value)` with "Sets the value" or, even worse "Sets the value of Value to value."

Ensure comments are not misleading, distracting, or ambiguous. Clearly state the main points or insights. Avoid long, convoluted explanations of your thought process when writing the code.

Listing 2 shows a snippet from a real example.[9]

9 https://github.com/Waikato/weka-3.8/blob/master/weka/src/main/java/weka/classifiers/evaluation/Evaluation.java

You can safely assume the method in Listing 2 returns the kappa statistic... Unnecessary, tautologous explanation is not the only "crime" here. A quick comment about what the kappa statistic actually is might help, or perhaps a link to a document explaining it in more detail because Javadoc comments support hyperlinking. Note that is a comment about what the kappa statistic is and not how the method to calculate it works. When describing code, sometimes you need to describe how it works but sometimes you need to describe what it does without really explaining how it does it. The latter case is common because the procedure might need to change to yield the same result.

The Listing 2 crime scene yields more evidence. If you know about this aspect of statistics then you might know there are two distinct kappa statistics used in different circumstances. If you did not know this then you did not even know you might have been misled. Ignorance might be bliss but it might also be dangerous: "we used statistical measures to evaluate the safety of our system." Using the wrong kappa will probably still give you a result rather than an error but that result will probably be at best meaningless and at worst misleading; and yet you might not realise. So is the function

```java
/**
 * Returns value of kappa statistic if class is nominal.
 *
 * @return the value of the kappa statistic
 */
public final double kappa() {
```

Listing 2: A good example of commenting done badly

in the example returning Cohen’s kappa or Fleiss’ kappa? Specifying which would have been a rather good use of a comment. As it is, you have to inspect the code (if you can find the file 6 levels down in the repository), interpret the calculations it performs, and then retrofit what you think it does — and you could be wrong — to the equations of Cohen’s and Fleiss’s kappa statistics (that you have had to research) to see which it is. Spoiler: we are not telling you.

5 Program design

A good design distinguishes good code from bad. It takes time and observation and practice to become a good architect. The following guidelines are a useful start.

5.1 Structure

Programs should be designed to divide responsibilities structurally. A good design should split the task into more manageable chunks. Think “divide and conquer” which in software development is step-wise refinement. One way to do this is create a method with a name reflecting a high-level task (such as `makeTea`), populate that with sub-tasks such as `prepareKettle` and `prepareCup`, then create methods with those names and repeat by refining those methods, repeating the refinement until it becomes a relatively simple instruction, possibly involving a call to some library code. One object-oriented example of “divide and conquer” is the meaningful division of the code into multiple classes.

5.2 Data representation

Use the appropriate data structures for the task at hand. This is the programming version of the saying “using the right tool for the job.” Good data representation can lead to much simpler and often more efficient solutions. The more of the Java collections you are familiar with, the better your chances of choosing sensibly. Do not worry about making mistakes in the early days. Correcting a mistake is one of the most informative ways to learn.

Planning ahead will save time in the end. Suppose part of the code you are writing needs to use calendar dates. There is a choice of suitable data structures and we leave you to think about those.

Right now we want to concentrate on another aspect. Thinking about the use your code will eventually have, it helps to decide at the design stage how to represent data. In our imaginary program that processes dates then we have two main choices: calendar representation or ordinal representation — both of these are independent of the actual data structures used in the code. If you know your program will mainly be displaying dates, then you want a representation that facilitates handling calendar dates (29 February 2144, 30 November 2145). However if you know your program will mainly be calculating dates, then an ordinal representation (34th day of the year 2143, perhaps written as 2143-34) is convenient. If you want to know the number of days from 20 January 2144 to 6 March 2144 then in a calendar representation you have to do more processing than in an ordinal representation where you simply substract one date from the other (and remember the off-by-one error).

5.3 Modularity

Classes and methods should perform well-defined tasks. They can be seen as building blocks for solving many different problems, not just the task at hand. Long methods are a symptom of a too problem-specific design. A lack of modularity makes it hard to extend a program, because adding new features requires the specific methods to be patched or completely rewritten. A rough guideline is that methods should fit on one “traditional” screen. That means a maximum of about 30 lines (and 80 characters wide). However 30 lines is already considered long. Most methods should be considerably shorter. You achieve this by stepwise refinement: breaking the code into named chunks which can be methods, sometimes variables, and in Object-Orient Programming, objects or interfaces.

Another huge advantage of the modular approach is that you can test each part separately. That means you are building a larger system out of smaller components which you believe already work perfectly (assuming your testing was thorough and your design flexible). The best way to go about coding this is baby steps: write a couple of lines and test them. This is far, far faster than writing a big chunk of code and trying to discover what is wrong with it this time.

Knowing the paradigm of the programming language you will use (Object-Oriented or Functional or Logic or the traditional Imperative) then that influences how you design as you play to the strengths of the paradigm. Moreover, if you know the actual programming language you will use, then you can play to its strengths too, whether its in-built features (syntactic sugar for example) and characteristics (such as speed or portability), or its libraries (Java's collections for example).

5.4 Avoid code duplication

If a program has similar or identical code repeated in different places, there is almost certainly a cleaner design. This is "DRY" versus "WET" programming: "Don't Repeat Yourself" versus "Write Everything Twice" (or even Thrice).[10] Duplicated code makes the program longer than it needs to be and makes it much more difficult and time-consuming to maintain for consistency and reliability. When you update a method in one class, you probably need to make the same change to the copied version of that method in another class. This is more work and there is a risk of the methods becoming inconsistent. A good, modular design is the cure for this because everything is written once and only once.

10 Also criticised as "We Enjoy Typing."

Or more accurately, everything appears only once in the final design. It may have been written and rewritten more than once, partially or wholly: it is possible there was some WET programming along the way. This was probably due to not fully understanding the design you needed or how some aspects of the language or a library are best utilised. This is common when developing. The interim versions are for making improvements: it is the final version that has been DRY'd out that counts.

When you look at someone's finished code you have no idea of the changes made during its development. It may have been a complete mess originally. A well-written final program was probably always based on a good design, even if some of the implementation was temporarily messy. Therefore please do not expect to product perfect code first time, especially for a non-trivial task, and especially when you only have a few years of programming experience.

6 Robustness

Programs should be able cope with incorrect usage, or preferably not allow it in the first place.

6.1 Validation

Catch invalid and unexpected inputs or parameter values, and handle them gracefully. The minimum acceptable is a meaningful error message. Either throw an `Exception` or print something useful to `System.err.println()` — this prints to Standard Error, one of the three "standard" streams that operating systems use. If you can recover from the error, perhaps by asking the user to correct their input, this is generally a good solution. `Exception` handling is a very Object-Oriented solution.

Even better is if your design, especially the interface, can minimise the possibilities for users going wrong in the first place.[11] If your program needs a number from the user, it must ensure that the user's input is a number. Except of course the user is typing a string (or in some programming languages multiple consecutive characters) that looks like a number: "87" for example. This then needs to be converted to the numeric value 87. The user could have typed "eighty-seven" which a person could handle but your program might not. One way to avoid this would be if your program either only displays digits on the screen to touch or click, or it immediately disqualifies non-digits so when someone touches a letter nothing happens.

11 Though people do have endless creativity for misusing, misunderstanding, and maximising damage.

Anywhere a program allows input, whether from a user or a file, that program should assume the input is not just possibly wrong but actively dangerous. It usually needs several "layers" of code to handle input fully. If you think of a program as a series of concentric circles, then the innermost circle is the purest whereas outside the outermost circle is chaos and trouble and dirt and pollution. The program needs to filter the input through the concentric circles, which can be considered as membranes. The order of these is vitally important but rather than tell you, think about it and read others' code for ideas, and most importantly try it. This filtration, when done correctly and fully — you need both — ensures the input is "safe and sensible" by the time it reaches the "pure" inner part of the code, depending on what is appropriate.

The source of potential problems is impressively large. Assume we need to read from a file, then the potential problems include but are certainly not limited to: the file does not exist; the file exists but you do not have appropriate access; the file is locked by another user or process; the file exists but is empty; the file did exist but somehow vanished while it was being read; the file contains inappropriate data in an inappropriate format; only one of the data or the format is appropriate; the data are correct and in an appropriate format but not in the expected order. At some point the program also needs to decide whether any individual datum[12] is actually meaningful. Take our earlier case of the user typing: even when your program has finally computed the user's input value as 87, it is simply not a sensible answer to the prompt of "How many teeth do you have left?"

12 "Data" is the grammatical plural, "datum" the singular; but many do treat "data" as grammatically singular

6.2 Access modifiers

By using restrictive access modifiers, you can prevent access to fields and methods that other parts of the code should not be using. In this way you can prevent incorrect usage in the first place. This is also an important measure in preventing the programming crime of "close coupling." This is where one part of the code "knows" the inner workings of another part of the code and makes use of that knowledge. This may seem clever to an inexperienced programmer but it is a nightmare for altering or maintaining code. Close coupling means you cannot make serious changes to one part of the code without breaking another part that has no business knowing how the first part works.

Imagine a person delivering letters to houses. In a loosely coupled system, the letters can be pushed through your door's letterbox no matter where that letterbox is located on the door: high up, low down, dead centre. Changing your front door for one with a letterbox in a different position would not stop the letters being delivered because the person delivering the post knows to use the letterbox, no matter its location. A closely coupled system would "know" the position of your letterbox and always attempt to push the letter at those very specific coordinates. This system would only continue to work if your new front door had the letter box in exactly the same position as your old front door. This may seem like a silly example but that is precisely what close coupling looks like. However close coupling is not just silly and poor practice, it can be actively dangerous: imagine any kind of safety-critical system you wish and how important it is that changing one component does not cause something else to misfunction.

Please realise that there are always interfaces between compontents in program code: whether methods (and their parameters) or classes or any other programming construct. The distinguishing feature is that internal changes, for example within a class, should not (usually) affect other classes. So if you decided to change the way your class in a larger program interally represents dates, then that should have no effect on any other classes because they should not know how your class represents dates. Referrnig to our earlier example, it should not effect any other part of the software system if you change from a calendar representation of dates to an ordinal one. The exchange of information should be through the provided methods whose signatures have almost certainly not changed.

Always specify an access modifier and consider them in this order: **`private`**, **`protected`**, or **`public`**. When left unspecified, access defaults to **`package private`** which is rarely what you want. You should know from your design what level of access other components should be allowed. If you do not: either you have not understood your design (or the scenario) well enough yet, or the design is underspecified.

Fields should not be **`public`** (except for some **`final`** fields). Use getters and setters (section 2.1) if other classes need access. This way the class retains some control when a change is made. For example it can check the new, incoming value is actually sensible. It could also record that a change has happened, which could be useful as it records accountability and possibly allows backtracking.

7 Technical/language proficiency

This is a broad area but in essence “using the right tools for the job.”

7.1 Use appropriate language features

Make use of the full potential of your programming language. This means not restricting yourself to building everything using a few keywords when there is a more elegant solution. It also means not using a needlessly complex mechanism for solving a simple task.

To become good at making the most of a language, you need to practice writing it and more importantly practice rewriting it to a different design. Actually this starts by creating different designs initially and implementing them.

You also need to "read" others' code and analyse it for how they accomplish tasks elegantly. To become good at reading, you need to develop a sense of "taste" (or "smell") for what is good quality code and what is not. There is no shortcut to this: to understand you have to try it for yourself, preferably by writing it yourself, even if only in a basic way. This is one reason why your teachers and tutors insist you write your own code for assignments and not just copy and paste it. It is important for you to realise our own current strengths and weaknessse. You do improve with practice but you have to put in the hard yards. This is how labs and tutorials contribute to your success.

7.2 Avoid wasting resources

Consider time and space complexity. For example do not copy a large array when you could have just used the original. Do not concatenate long strings in a loop: use a `StringBuilder` instead. To understand why, look at the source code of Java[13] itself: the `String` class is declared **`final`**. IntelliJ is excellent at finding such optimisations. Use the `Problems` tab as a guide. There is also an `Inspect code` menu option. Helping you improve your code is just one reason why we use complex tools like IntelliJ.

13 The file `src.zip` in your JDK directory.

7.3 Know the API

Do not reinvent the wheel every time: or preferably at all. Know what the (standard) library provides and make use of it. **Note:** some assignments may restrict use of libraries or features in order to test that you can write the code yourself and because submitting someone else's code makes the marks unfair.

8 Code idioms

8.1 Initialise class fields

Initialise all class fields in the declaration or by using the constructor. Often this is done by assigning a parameter on the constructor directly to the class field, Listing 3. However if logic is needed, in this case to ensure a minimum number of sides, then that should go via a setter method (section 2.1) which the constructor can make use of to avoid WET programming (section 5.4), Listing 4. Depending on what is appropriate, the logic can go directly in the setter or in another (possibly **private**) method that the setter (and other methods) can make use of. If the parameter and the class field have the same name — and it can be hard to think of a meaningful alternative name — then they are distinguished by prefixing **this**. (with the dot) to the class field, Listing 5.

8.2 Magic numbers

“Magic numbers” (they really are called that) are where actual numbers are harded-coded into the source. Look at this code and decide what the numbers 1 and 3 mean:

```
if ((x == 1) && (y == 3)) {
    return 1;
}
```

At best, you can say what the code does (returns 1 if x is 1 and y is 3) but not what the code means. Would it help if we told you this code snippet is part of a function which decides the winner in a game of rock paper scissors?[14]

14 🪨 📜 ✂️

Even then, what is 1 and what is 3 and why does 1 beat 3? From this snippet you simply cannot say with certainty: you can guess but guessing is potentially dangerous — code for a safety-critical system is no place to guess and hope. Moreover you might think that the 1 being returned is the same as the 1 in the condition of the **if** statement. It could be in some other situation (same code fragment, different scenario) but in this case it is not. The 1 being returned is saying that Player 1 has won whereas the 1 in the condition represents Player 1’s choice in the game. Moreover, neither of those 1s are related to the 1 in Player 1’s moniker.

Compare the magic number version with this one:

```
if ((x == ROCK) && (y == SCISSORS)) {
    return PLAYER_1;
}
```

This version is an improvement but we should of course use more meaningful names:

```
if ((player1Choice == ROCK) && (player2Choice == SCISSORS)) {
    return PLAYER_1;
}
```

```java
public class Shape {

    public final static int MINIMUM_SIDES = 3;

    private int numberOfSides;

    public Shape(int sides) {
        numberOfSides = sides;
    }
}
```

Listing 3: Using a constructor for initialisation. JavaDoc comments omitted for brevity.

```java
public class Shape {

    public final static int MINIMUM_SIDES = 3;

    private int numberOfSides;

    public Shape(int sides) {
        setSides(sides);
    }

    public setSides(int sides) {
        if (sides < MINIMUM_SIDES)
            numberOfSides = MINIMUM_SIDES;
        else
            numberOfSides = sides;
    }
}
```

Listing 4: Using a setter directly to enforce consistency. JavaDoc comments omitted for brevity.

```java
public class Shape {

    private int sides;

    public Shape(int sides) {
        this.sides = sides;
    }
}
```

Listing 5: Using `this.` for initialisation. JavaDoc comments omitted for brevity.

We can agree this code snippet is completely understandable, if you know the game of rock paper scissors, even when all the rest of the code for the game is not present. To complete your understanding, imagine that somewhere appropriate there is a series of definitions that associate well-formed meaningful identifier names with values, see section 2. There are different mechanisms for doing this in Java. One way (not necessarily the best but more than good enough to make our point) is to use constants, which by convention are written in SCREAMING_SNAKE_CASE:

```
final static int ROCK     = 1;
final static int PAPER    = 2;
final static int SCISSORS = 3;

final static int PLAYER_1 = 1;
final static int PLAYER_2 = 2;
```

Note that declaring the players this way soon becomes clumsy as the number of players increases and it becomes unsustainable for large numbers of players. Using some form of data structure like an array would be neater, easier to maintain, and more professional because all the players, however many, can be held in a single data structure or in parallel data structures such as a series of arrays where the same index in each array refers to different aspects of the same player. Your ideas for how to do this will change as you become more experienced with Java: so design something now and then redesign it at the end of the course and compare.

The variables about choices probably cannot be constants because each player's choice will probably differ per round of the game: a player might choose PAPER every round but they should not be forced to because the game is buggy and only lets you ever choose once. For this code snippet the players' choices look like parameters on the method whose signature might be:

```
private int decideWinner(int player1Choice, int player2Choice) {
// code to determine who wins
}
```

There are other possibilities for the method signature depending on the design and implementation choices. It is a useful paper exercise to design the game (or part of it) multiple times with different designs each time and compare their relative strengths and weaknesses. Again, do this now and repeat at the end of the course.

It is important to learn from the rock-paper-scissors example not just how to improve the code but also to understand the problems that magic numbers cause. So far you have seen that they cause instant confusion about their meaning, and that not all instances of a literal number are necessarily the same magic number (in our example the 1s with different meanings). But this is by no means the end of list of problems caused by magic numbers.

Using magic numbers also causes other programmers to doubt your competence as a programmer. Someone else looking at your code and seeing magic numbers would immediately think “Well, if you can’t do the easy stuff properly, how can we trust you have done the difficult stuff properly?” I have a manual for a device that insists the USB port is under a flap on the front when in fact it is on the back and definitely not under a flap. It does not matter technically but it makes the company look inept. More seriously, you are about to board a plane and notice that only half of it is painted and that the name of the airline is spelt wrong. In theory these are only superficial, inconsequential mistakes. But they would make you doubt whether a company so careless can maintain its planes safely. Also, did the airline not notice because they do not check their work carefully, do they simply not care?

In many cases (not necessarily in our rock-paper-scissors example) it can be unclear what changing the value of the magic number will do. You might think “just try it and see” but suppose you were working on code that many others rely on, or something that is safety-critical. Randomly changing dials on a control panel in an aircraft or nuclear power station “to see what they do” is unwise. It can help your programming if you try to imagine the code you are writing (or reading) in a real world context.

This hints at another serious problem with magic numbers. Suppose you do need to change something. Perhaps you have worked out that a magic number in some code (that you definitely did not write) is referring to the current rate of Value Added Tax (VAT, the equivalent of GST in some countries). The rate can be changed arbitrarily by the government and is therefore outside of your company’s control but you need your code to be always up to date. Suppose the rate changes from 5 % to 7 %: as the maintainer of poorly written source code, you need to find the instances of the number 5 and change them. Except of course you must not assume that all instances of the number 5 refer to the VAT rate:

some number 5s might refer to the number of working days in a week and changing those to the new VAT rate (7 % but written just as 7) might cause many problems, not only in the software but in the real world when employees leave because they do not want to work the 7-day week the updated software now insists on.

This should remind you of the differing meanings of the different number 1s in our rock-paper-scissors example, as well as showing you that even simple examples can be closely related to real-world situations. Finding and replacing all the appropriate instances and only the appropriate instances is potentially time-consuming — real systems are often measured in tens of thousands of lines of code across multiple files — and doing this manually is always error-prone because it is too easy to miss instances or to change unrelated instances. It may be even worse if you try to be clever and use some form of automated find and replace as you may not see some lines being changed.

One advantage of associating a value with an identifier is that if the value needs to be changed, you only need to change it once and in one place only: remember DRY programming (section 5.4). So when the VAT rate changes, you only need to update one definition and recompile. In real life you would of course also do some rigorous testing. If you wanted to be really clever, you would accept the VAT rate from the user via prompted input or read the VAT rate from a configuration file or command-line parameter. That way the source code would not even need changing.

Each line of code should make sense on its own wherever possible — and it is much more possible than it might first seem when you are inexperienced. A comment explaining something is a poor substitute for clear code, not least because a comment barely addresses the problem of the magic number in that line and does nothing for other occurrences. Even if the source code is short, classes are used by other programs which are written completely separately, possibly years later, and probably by someone else. This is one of the key characteristics of Object-Oriented Programming: extensibility. This makes it hard for another programmer to know whether this value can be changed safely unless the original programmer uses self-documenting or properly commented code.

One place where a magic number is usually acceptable is compensating for the off-by-one error. A constant of the valuef +1 or -1 is unnecessary where you just need to subtract 1 from or add 1 to a variable (typically a loop index) to obtain the correct number, for example printing the current loop iteration count. You saw an example of an acceptable magic number in the calculation of `totalEnergy`, page 11.

8.3 Magic strings and the end-of-line

Unsurprisingly, magic strings are the same as magic numbers (section 8.2) but with `String`s and **`char`**acters.

Text that forms parts of messages is often written with magic strings. For one-off uses this can be ok but perhaps not ideal, so always consider defining a constant: just because you have only one use for that `String` in this part of the program does not prevent another part or another programmer needing it. In such cases consider the visibility of the `String` by its access modifiers, section 6.2, and whether it is declared in a class or a method.

For genuine one-off uses consider a formatted string: check the official Java documentation on `String.format()` and the related `System.out.printf()`.

`String.format()` returns a new[15] `String` that has been formatted which means it applies formatting and gives you the result. Formatting can for example be displaying a floating point number to a specific number of decimal places (with automatic rounding); or padding a number with a specified amount of leading zeros or spaces; or displaying the contents of a `String` variable in uppercase. None of those formatting examples affect the original value, so the variable containing the floating point is unchanged as is the `String` printed in uppercase. Some examples are in Listing 6.

15 Remember `Strings` in Java are immutable.

You use `String.format()` when you want to format a `String` and pass it as an argument or save it for repeated use. If you know you definitely only need to print the formatted String, then go straight to `System.out.printf()`. You can short-cut this in IntelliJ by typing `souf` (all lowercase) and pressing ⭾ (TAB) once the list of auto-completions pops up. If your computer is running IntelliJ slowly there might be a 2-second delay.

Looking at Listing 6, note how the formatted `String` (especially the first one) can make it easier to read the message text than the concatenated Strings in the `println` statement. Play with the example in conjunction with reading the official guide to `String.format`. As a useful exercise, create data structures that hold the names of those participating in a fictional election along with everyone's share of the votes. Use a loop to print the name and result per party on a separate line each. Play with the formatting to change the column widths and the printed precision of the numbers.

Note that %n is used to guarantee using the correct way to end a line on the computer that the programming is running on. Remember that Java is platform-independent and so a compiled Java program (the `.class` files) can be taken and run on any other computer irrespective of operating system that Java supports. This is different from taking the source code (`.java` files) and compiling them on another computer. Windows uses 2 characters to end a line (Carriage Return then Line Feed, often written CRLF) whereas Linux and Mac use just the Line Feed character. There are historical reasons for this based on manual typewriters. Carriage return sent the carriage (the bit holding the paper) back to the first column. This allowed you to type over any existing text on that line and was how you got **bold** letters or created symbols that were not on the actual keyboard such as plus-minus (±) (type the + then go back and type an underscore). Line feed advanced the paper by one line so you could type a fresh line.

```java
System.out.printf("Share of votes for Green Party: %3.1f%n",
                  48.071824744198);

double greenShare = 48.071_824_744_198;
double goldShare  = 51.852_651_112_1696;

long totalNumberOfVotes = 33_577_342L;

/* The formatting of the goldShare is deliberately different to highlight
   padding with zeros from padding with spaces */
System.out.printf("Greens:\t%3.1f (%10d votes)%nGolds:\t%3.1f (%010d votes)%n",
                  greenShare, calculateVotes(totalNumberOfVotes, greenShare),
                  goldShare,  calculateVotes(totalNumberOfVotes, goldShare));
/* int calculateVotes(int totalVotes, double voteShare)
   is missing but is the formula:
   totalVotes / 100 * voteShare  (it might need rounding)
   The formula is another case where a magic number might be acceptable */

// Compare formatted Strings with concatenated Strings:
System.out.println("Proof that the shares were not changed from their declarations
   "
                   + " greens " + greenShare + " golds " + goldShare);
```

Listing 6: Examples of formatted `Strings`

Many programming languages use `\n` to represent the end-of-line. Java can do this but be warned that it is only inserting a Line Feed character and thus your program might not work properly on Windows. Using `println` %n in `printf` makes the code platform independent. For reference `\r` is the Carriage Return character.

`\t` prints a tab which can help with alignment, but using %s gives more control because you can specify the width of the column, such as `%15s`, and even control whether it is left-justified or right-justified (`%-15s`), and you can make a `String` uppercase just by using %S.

8.4 Importing classes

Import classes separately. Instead of:

```
import java.util.*;
```

write:

```
import java.util.ArrayList;
import java.util.HashSet;
```

Although the first example seems more convenient, it suggests the programmer is either lazy (could not be bothered to specify the exact imports) or — and these are inclusive-or — is disorganised (did not know what to import due to a vague design) or was using a dumb tool for writing code (and thus looking unprofessional). IntelliJ can optimise imports for you with a short-cut key. IntelliJ can also create the **`import`** statements for you when you type the name of a class or interface or enumeration that is not currently available.

8.5 Ordering of declarations

There is a Java convention for the order in which declarations should appear in a class:

1. fields
2. constructors
3. methods

IntelliJ can reorder your code for you with a single command: Code 〉 Rearrange Code.