Coursework Feedback Comments

Below a list of coursework feedback comments, covering the typical issues raised by Java courseworks seen over a number of years.

Code

1. Code is not well structured or formatted. The IDE will automatically format source code, so take advantage of that.

2. Code and comments mostly reasonable but could be tidied up in some places. For example, some lines may be too long and should be broken up with new lines to maintain the formatting, or there are unnecessary blank lines, or the comment is difficult to read.

3. Don't use tab characters in source code. Don't leave commented-out lines of source code, if the code is not used remove it before submission!

4. Pointless comments. Comments should add information to the code, not state what is obvious from reading the code. For example, the comment 'Initialise the instance variables' in a constructor adds nothing. A comment should explain 'why' not 'what', adding value to the code.

   'Why' means explain the rationale of the code, why is it this way, rules, logic, the underlying reason for the code or design. These are things that cannot be determined by reading the code.

   'What' means an explanation of the code itself, which should be obvious by reading the code if it is properly written, making the comment redundant.

   Also remove default comments added by IDEs, such as this example:

   ```
   /*
    * To change this template, choose Tools | Templates
    * and open the template in the editor.
    */
   ```

5. Code is OK but commenting is poor, and reduces the readability of the code. Always remember to make your code readable by other programmers (or yourself in six months time!).

6. Code compiles but there are issues and it doesn't always run correctly and/or do anything useful. Review your code carefully, just because it compiles doesn't mean it is well written.

7. Code won't compile. Don't try and write a lot of code before compiling it. Work in small steps so that when you get a compilation error it is obvious what you last did to cause the error. Take note of all the help and options the IDE gives you.

8. Check your spelling! For variable, class and method names and also in comments and data. For example: Object-Oriented not Object-Orientated. Its versus it's. Punctuation.

   Doing good quality work is important and that includes the spelling.

9. Beware cast expressions. With generics, cast expressions are needed much less frequently, especially when getting values from data structures. If you find yourself using a cast expression take a good look at the design to see if it can be eliminated. If you have a reference of superclass type avoid casting it to a subclass type. This is known as *downcasting* and is frequently an indication of a serious design problem.

   For example, if you have a class Person and a subclass Technician you can have a variable of type Person referring to a Technician object. If you try to cast the Person reference to Technician:
   ```
   Person aPerson = new Technician(...);
   ...
   Technician t = (Technician)aPerson;
   ```
   then you need to ask why you need to do this. The Person class should either declare all the methods that will be called on the Technician object, so the cast is not needed, or the design should be changed so that the Technician object is referred to only by references of type Technician. Remember also that a cast expression can cause a runtime error if the object referred to is not of the type expected.

10. No public instance variables and/or don't leave out public/protected/private altogether! Declare all instance variables private unless you have a very good reason not to (there is no good reason!).

    Definitely don't declare a public variable (either static or instance) and then access it from another class using an expression like: obj.someVariable = 10;.

11. Class names should start with a capital letter. This is a usage convention rather than a language rule, so has no affect on whether a class compiles or not. However, programmers expect class names to start with a capital and it helps identify a name as a class name.

12. Don't use a public static variable as a global data structure. Use a private instance variable in class and create an instance of the class with getter/setter methods. Or use the Singleton Pattern.

13. Don't have empty compound statements. For example, this code:
    ```
    if (someTest())
    { }
    else
    { return aValue; }
    ```

    can be rewritten as:
    ```
    if (!someTest())
    { return aValue; }
    ```

    This uses the not operator (!) to reverse the boolean expression.

14. Other than main, don't write static methods unless you understand what they are meant to be used for.

15. Beware of extra semi-colons. For example, writing:
    ```
    if (a == b);
    {
    ```

```
    System.out.print(a);
}
```

is valid code that will compile but note the semi-colon following the boolean expression. This will terminate the if statement and the following compound statement will always be executed.

16. Don't name all your methods do<Something>. The 'do' part is redundant and the naming also implies that you are not following an OO approach.

17. A .java file should contain one Java class only (excluding nested classes). Putting multiple classes into the same file will cause problems with the compiler.

18. Don't use static variables in place of instance variables. Doing so will sooner or later cause major problems.

19. Split up monster methods. If a method has more than 15-20 lines it most likely needs to be split into several smaller methods. For example:

```
public void myMethod()
{
    first part
    … lots of code
    second part
     … lots more code
    third part
     .. even more code
}
```

then each part can become a method and the code replace with a method call:

```
public void myMethod()
{
   call firstMethod()
   call secondMethod()
   call thirdMethod()
}
```

The original method calls the new methods and becomes a control or manager method. The new methods contain sections of code extracted from the original method. Parameters and/or return values may need to be added. The IDE refactoring called Extract Method will make this easy to do, and undo if needed.

The new method names, hopefully chosen well to make the intent of each method clear, will remove the need for comments and make the code more readable.

20. Following from the previous item, if a method is structured like this:

```
if ( something) { ... }
else
if (somethingelse) { ... }
else
if (anotherthing) { ... }
```

split the method up to extract the if statement bodies into separate methods.

21. If a method returns a value using an if statement, *don't* write this:

    ```
    if (boolean exp)
    {  return true; }
    else
    { return false; }
    ```

    Instead use this:

    ```
    return boolean exp;
    ```

    The if statement is unnecessary and just clutters up the code!

22. When using pathnames to access files or directories, don't assume the program will be run on a particular operating system, as the file name formats differ between Windows and Unix. Windows uses drive letters and '\' as a name separator, for example c:\someplace\myfile.txt. Unix doesn't use drive letters and uses '/' as a name separator, for example /users/home/someplace/myfile.txt.

    Class File provides a way of getting the right character to use: File.separator will return the character for the operating system the program is currently running on. Don't use drive letters at all. And stop using Windows.

23. Pathnames for data files should be relative not specified in a specific place in the filestore using an absolute pathname. An absolute name gives the exact location starting from the root directory, while a relative name give the path based on where the program actually is in the filestore. An absolute name won't work on a different machine.

24. When using generic class like ArrayList or HashMap make sure the parameter type is specified in all the places it is needed. For example:

    ```
    ArrayList<String> a = new ArrayList();
    ```
    should be:
    ```
    ArrayList<String> a = new ArrayList<>();
    ```
    Note that the angle brackets on the right hand side can be empty as the parameter type String will be inferred. You can still include String if you want:
    ```
    ArrayList<String> a = new ArrayList<String>();
    ```

    If you don't do this the compiler will display a message like this:
    Note: Some input files use unchecked or unsafe operations.
    Note: Recompile with -Xlint:unchecked for details.

25. Don't leave a method with an empty method body. Either remove it or fill in the method body.

26. Don't leave an empty catch block in a try-catch statement. If there isn't anything useful you can put in the catch block it is either in the wrong method, or you need to rethink why you are using the exception handling mechanism in the section of code.

27. Only one Input object to read from the keyboard should be created when a program is run. If more than one is created, all the objects will be trying to read from the same

input and will not work properly. The same issue can arise when using a Scanner or other Java library class when inputting from the keyboard.

28. Don't use class Vector. Although it is still included in the class libraries you should be using ArrayList or one of the other generic data structure classes.

29. Don't use cryptic variable and method names. A name should be *pronounceable* and reflect the use of the variable or method.

30. Don't use == to compare Strings. The == operator will compare object references and tell you if the two String objects are the same object. To compare the values of two different String objects to see if they represent the same string you should use the equals or compareTo methods.

31. Learn how to use your IDE, especially IntelliJ IDEA. It will have many features and short cuts that will make your programming more efficient (and fun!).

32. The Intellij IDE can analyse your code (see the Inspect Code item in the Analyse menu). Make use of this to identify problems with your code, for example a variable that has been defined but never used, or methods that are never called. Check for and remove unused code, variables and methods.

Design

1. Less than 3 classes. The classes are probably not good abstractions.

2. The classes chosen do not represent a good set of abstractions for the problem being solved. Keep your classes cohesive, focussed on representing one abstraction well. A class should only have one reason for changing, to improve the implementation of the abstraction it represents. If a class has to change for several unrelated reasons then it a poor abstraction and needs splitting up into two or more classes.

3. One or more classes are getting too large and less cohesive. If a class contains a larger number of methods (e.g., greater than 10) consider splitting it into several smaller classes.

4. You have written a one class program. You have essentially written a procedural program and put all the methods into one class. This really misses the point, so you need to look again at classes and objects, and the idea of object-oriented programming.

5. Mostly a one class program with one or two other minor classes. Also known as a 'Monster class' - one class holds most of the methods, any other classes are much smaller, and may just hold a few methods without instance variables. The design needs to be re-worked to identify a better set of classes.

6. Don't write a procedural program (like a C program) and then spread out all the methods into classes to try and make the program look object-oriented.

7. Don't identify a collection of procedures (methods) and then try to create a class out of each procedure so you end up with a collection of classes containing one method.

8. If you have a class which is just a collection of unrelated methods with no instance variables, then it is probably not a good class. In the worst case it is a dumping ground for methods you can't fit it another class!

   You can have a utility class with a collection of related methods, but you do need to make sure the methods are clearly related. For example, class Math in the standard Java library is a utility class containing the set of methods implementing mathematical operations.

9. The design is confused and needs re-thinking. Will anyone else understand your design? If you don't understand it nobody will!

10. Better class names could have been chosen. Use a name that makes clear the role or intent of the class. You can rename classes as you find better names, and it is often the case that you don't find a better name until well into the coding work when you better understand the design and abstractions. The IntelliJ IDE makes it easy to change class names using the rename refactoring.

11. Don't use plural class names. If an object of a class is intended to hold a collection of other objects (e.g., a Library holds a list of Books), use a singular name for the class. For example, use FileCollection or FileStore for a class that holds a collection of files.

    Don't make the mistake of creating, for example, a Book class and try to use it to represent a collection of books as well as a single book. This is muddling up the ideas of an object to represent a book with a collection of books.

12. Don't have a class name which is a verb, find a noun instead. For example, you might name a class to manage a search as Searching, where it would be better to use a name like SearchController or SearchManager. Or you might use ListLoader rather than LoadLists.

    If you find yourself using verbs as names, beware of creating classes just to wrap single methods as you a probably using a procedural design and wrapping the code in classes.

13. Don't use public static variables unless they are declared as final and treated as constants. Using non-final public static variables and accessing them directly from other classes is a poor design strategy, resulting in poor encapsulation and increased coupling (dependencies between classes).

    If you must have variables accessible to multiple parts of a program then create a class using private instance variables along with setters and getters, and use the Singleton pattern to allow only one object of the class to be created. Note, however, that this will increase the dependencies between the different parts.

    If possible, modify the program design to avoid the need for such "global" variables.

14. Beware unintended recursion. In particular when displaying menus for a text-based interface to your program.

    For example, if you have a run method that contains a loop to display a menu and run a command, make sure the menu display or command methods do not recursively call the run method. They should simply return so the next iteration of the run loop displays the menu again.

15. No working load and/or save to file option. Important functionality is missing as the program cannot store data between runs.

16. Creating a subclass of ArrayList to get a list of something class (e.g., list of items) is a bad design strategy as the subclass will inherit all the public methods of ArrayList that are not appropriate for the subclass. Use association instead so that the list of something class uses an ArrayList via a private instance variable.

17. Remove duplicate or similar code. If you have code in two or more methods or classes that looks similar or is the same (copy and paste code), then try to remove the duplication. Use techniques like extracting shared code into a single method or creating a superclass to hold shared code used by subclasses.

UML Class Diagram

1. The diagram is OK as far as it gets but there are inaccuracies such as incorrect associations or missing information.

2. The diagram is not a plausible and/or useful UML class diagram. It doesn't follow UML diagram syntax and is an arbitrary collection of lines and boxes that doesn't make sense. If you don't know what it means, nobody will!

3. The diagram is not consistent with the source code. Extra classes have been added or classes are missing.

4. One or more associations are missing or incorrect. An association is shown as a line between two class icons showing that one class has an *instance variable* referring to an object of the other class. Don't mix up inheritance with association.

5. Check associations and association annotations. Multiplicity should appear on all associations. Arrow heads should be on the right end and have the same shape. Don't use bi-directional associations with an arrow head at each end.

6. Beware of tools that automatically generate UML class diagrams from source code. They don't get all the associations and dependencies correct, and can include library classes you don't want on the diagram.

7. Don't confuse associations and dependencies. A class has an association to another class if an *instance variable* is used to refer to an object of the other class. A dependency is used when one class uses another class temporarily, for example via a *local or parameter variable* within a single method. A dependency is shown as a dashed line.

8. Don't branch or join associations. Use separate association lines.

9. A class icon must be connected to at least one other icon via an association or dependency. Otherwise, it can’t be part of the design as nothing can use it!

10. A class diagram is not a flow chart. It does not show the sequence of method calls between objects of the classes, or show that classes ‘call’ each other in a particular sequence.