CM 10227: Lecture 9

Dr Rachid Hourizi and Dr Michael Wright

December 1, 2016

Resources

- More help with this course
  - Moodle
  - E-mail - programming1@lists.bath.ac.uk
- Online C and Java IDE
  - https://www.codechef.com/ide
  - Remember to select Java from the drop down menu.
- Books
  - Java by Dissection (Free pdf online)
  - The Java Tutorial (https://docs.oracle.com/javase/tutorial/)

Resources

- The places that you can get additional support if you are finding the pace of the course a little fast now include
  - A labs (Continued from week 1)
  - B labs
  - ... Wednesday 11:15-13:05 EB0.7
  - ... Fridays 17:15 to 19:15 in CB 5.13)
  - The Drop in Session
  - ... booked 20 min appointments
  - ... Friday 11.15-13.05 1E 3.9
  - PAL sessions (Mondays 14:15 to 15:05 1E 3.9)

This week

- Errors
- Exceptions
- Style : Writing Better Code

Style : Writing Better Code

Reading from a File

- We know how to read from the standard in...
- ... using the Scanner class
- ... using the BufferedReader class

BufferedReader : To Read From Standard In

```
BufferedReader input = new BufferedReader(new
    InputStreamReader(System.in));

try {
     String userInput = input.readLine();
     ...
}
catch (IOException e) {
     e.printStackTrace();
}
```

InputStreamReader

- From the Java Doc...
- An InputStreamReader is a bridge from byte streams to character streams
- It reads bytes and decodes them into characters using a specified charset
- The charset that it uses may be specified by name..
- ... or may be given explicitly
- ... or the platform's default charset may be accepted
- Subclass of Reader which is an abstract class for reading character streams

BufferedReader

- From the Java Doc...
- Reads text from a character-input stream...
- ... buffering characters so as to provide for the efficient reading of characters, arrays, and lines

BufferedReader

- So if BufferedReader reads text from a character-input stream...
- ... and we know its constructor takes an object of type Reader as it parameter
- ... and we know that InputStreamReader is a subclass of Reader
- ... and we know that another subclass of Reader is FileReader
- ... then (hopefully) the following code is not too surprising

BufferedReader : To Read From a File

```
File myFile = new File("example.txt");
FileReader fileIn = new FileReader(myFile);
BufferedReader input = new BufferedReader(fileIn);

try {
    String line = input.readLine();
    ...
}
catch (IOException e) {
    e.printStackTrace();
}
```

Style : Writing Better Code

Style : Writing Better Code

- Maintainance
- Coupling - Cohesion
- Improving Code

Software Changes

- Software is not like a novel that is written once and then remains unchanged.
- Software is extended, corrected, maintained, ported, adapted
- The work is done by different people over time (often decades).

- Change or Become Useless!
- There are only two options for software:
  - Either it is continuously maintained
  - Or it becomes useless.
- Software that cannot be maintained will be thrown away.

Code Quality

- Two important concepts for quality of evolving code:

- Coupling
- Cohesion

Coupling

- Coupling refers to links between separate units of a program.
- If two classes depend closely on many details of each other, we say they are tightly coupled.
- Aim for loose coupling.

Loose Coupling

- Loose coupling makes it possible to:
  - understand one class without reading others
  - change one class without affecting others
- Thus: improves maintainability.

Cohesion

- Cohesion refers to the the number and diversity of tasks that a single unit is responsible for.
- If each unit is responsible for one single logical task, we say it has high cohesion.
- Cohesion applies to classes and methods.
- Aim for high cohesion.

High Cohesion

- High cohesion makes it easier to:

- Understand what a class or method does
- Use descriptive names
- Reuse classes or methods

Cohesion of Methods

- A method should be responsible for one and only one well defined task

Cohesion of Classes

- Classes should represent one single, well defined entity.

Design Questions

- Common questions
  - How long should a class be?
  - How long should a method be?
- Can now be answered in terms of cohesion and coupling

Design Guidelines

- A method is too long if it does more then one logical task.
- A class is too complex if it represents more than one logical entity.
- Note these are guidelines. They still leave much open to the designer

Code Duplication

- Is an indicator of bad design,
- Makes maintenance harder,
- Can lead to introduction of errors during maintenance.

Responsibility-Driven Design

- Question: where should we add a new method (which class)?
- Each class should be responsible for manipulating its own data.
- The class that owns the data should be responsible for processing it.
- Responsibility-driven design leads to low coupling.

Localising Change

- One aim of reducing coupling and responsibility-driven design is to localize change.
- When a change is needed, as few classes as possible should be affected.

Thinking Ahead

- When designing a class, we try to think what changes are likely to be made in the future.
- We aim to make those changes easy.

Refactoring

- When classes are maintained, often code is added.
- Classes and methods tend to become longer.
- Every now and then, classes and methods should be refactored to maintain cohesion and low coupling.

Refactoring and Testing

- When refactoring code, separate the refactoring from making other changes.
- First do the refactoring only, without changing the functionality.
- Test before and after refactoring to ensure that nothing was broken.