CODE STRUCTURING & CODING STRATEGY

HOW TO STRUCTURE YOUR WORK?

- This is for guidance only and I will not go into great detail, to avoid seeing identically structured solutions.
- Part of the practical is structuring it yourself. However, it is likely you will want at least the following **components**:
  - A parser,
  - A representation of the states of a simulation,
    - Operations over that state
  - The simulation algorithm,
  - Something to handle output,
  - Something to analyse results,
  - **A test suite**.

SOME OBVIOUS DECISIONS

- Do you want to parse into some *abstract syntax* data structure and then convert that into a representation of the initial state,
  - Or could you parse directly into the representation of the initial state?
- Do you wish to print out events as they occur during the simulation,
  - Or record them and print them out later?
- Do you wish to analyse the simulation events as the simulation proceeds,
  - Or analyse the events afterwards?

PARSING

- You do not necessarily need to start with the parser.
- The parser produces some kind of data structure. You could instead start by hard coding your examples in your source code.
- Though the parsing for this project is pretty simple.
- Hence you could start with the parser, even if not **complete**, but remember your parser is marked at Part 2.
    - Hard coding data structure instances could prove laborious,
    - but doing so would ensure your simulator code is not heavily coupled with your parser code.

SOFTWARE CONSTRUCTION

- Software construction is relatively unique in the world of large projects in that it allows a great deal of *back tracking*.
- Many other forms of projects, such as construction, event planning, and manufacturing, only allow for backtracking in the design phase.
- The design phase consists of building the object **virtually** (on paper, on a computer) when back tracking is inexpensive.
- Software projects do not produce physical artefacts, so the construction of the software is mostly the design.

REFACTORING

- Refactoring is the process of restructuring code while achieving exactly the same functionality, but with a better design.
- This is powerful, because it allows trying out various designs, rather than guessing which one is the best.
- It allows the programmer to design retrospectively once significant details are known about the problem at hand.
- It allows avoiding the cost of full commitment to a particular solution which, ultimately, may fail.

MORE ABOUT REFACTORING

- Refactoring is a term which encompasses both factoring and defactoring.
- Generally the principle is to make sure that code is written exactly once.
- We hope for zero duplication.
- However, we would also like for our code to be as simple and comprehensible as possible.

FACTORING AND DEFACTORING

- We avoid duplication by writing re-usable code.
- Re-usable code is generalised.
- Unfortunately, this often means it is more complicated
- *Factoring* is the process of removing common or replaceable units of code, usually in an attempt to make the code more general.
- *Defactoring* is the opposite process specialising a unit of code usually in an attempt to make it more comprehensible.

FACTORING EXAMPLE

```
void primes(int limit){
    integer x = 2;
    while (x <= limit){
        boolean prime = true;
        for (i = 2; i < x; i++){
            if (x % 2 == 0){ prime = false; break; }
        }
        if (prime){ System.out.println(x + " is prime"); }
    }
}
```

A very naive but perfectly reasonable bit of code to print out a set of prime numbers up to a particular limit.

FACTORING EXAMPLE

```
void print_prime(int x){
    System.out.println(x + " is prime");
}
void primes(int limit){
    x = 2;
    while (x <= limit){
        ... // as before
        if (prime){ print_prime(x); }
    }
}
```

Here we have “factored out” the code to print the prime number to the screen. This **may** make it more readable, but the code is not more general.

FACTORING EXAMPLE

To make it more general we have to actually parametrise what we do with the primes once we have found them.

```
interface PrimeProcessor{
    void process_prime(int x);
}
class PrimePrinter implements PrimeProcessor{
    public void process_prime(int x){
        System.out.println(x + " is prime");
    }
}
void primes(int limit, PrimeProcessor p){
    x = 2;
    while (x <= limit){
        ... // as before
        if (prime){ p.process_prime(x); }
    }
}
```

You can now use different functions to display, store, etc. the prime numbers.

FACTORING

We can go further and factor out the testing as well:

```
interface PrimeTester{
    boolean is_primes(int x);
}
class NaivePrimeTester implements PrimeTester{
    public boolean is_prime(int x){
        for (i = 2; i < x; i++){
            if (x % 2 == 0){ return false; }
        }
        return true;
    }
}
void primes(int limit, PrimeTester t, PrimeProcessor p){
    x = 2;
    while (x <= limit){
        if (t.is_prime(p)){ p.process_prime(x); }
    }
}
```

FACTORING

Now that testing is factored out, it does not have to be used solely for primes.

```
interface IntTester{
    boolean property_holds(int x);
}
class NaivePrimeTester implements IntTester{
    public boolean property_holds(int x){
        for (i = 2; i < x; i++){
            if (x % 2 == 0){ return false; }
        }
        return true;
    }
} // Similarly for PrimeProcessor to IntProcessor
void number_seive(int limit, IntTester t, IntProcessor p){
    x = 0;
    while (x <= limit){
        if (t.property_holds(p)){ p.process_integer(x); }
    }
}
```

FACTORING

Print the ***perfect*** numbers:

```
interface IntTester{
    boolean property_holds(int x);
}
class PerfectTester implements IntTester{
    public boolean property_holds(int x){
        return (sum(factors(x)) == x);
    }
} // Similarly for PerfectProcessor
void number_seive(int limit, IntTester t, IntProcessor p){
    x = 0;
    while (x <= limit){
        if (t.property_holds(p)){ p.process_integer(x); }
    }
}
```

FACTORING

So which version do we prefer? This one:

```
public abstract class NumberSeive{
    abstract boolean property_holds(int x);
    abstract void process_integer(int x);
    abstract int start_number;
    void number_seive(int limit){
        x = self.start_number;
        while (x <= limit){
            if (self.property_holds(p)){ self.process_integer(x); }
        }}} // Close all the scopes
public class PrimeSeive inherits NumberSeive{
    public boolean property_holds(int x){
        for (i = 2; i < x; i++){
            if (x % 2 == 0){ return false; }
        }        return true;  }
    void process_integer(int x) { System.out.println (x + " is prime!"); }
    int start_number = 2;
}
```

FACTORING

Or the original version?

```
void primes(int limit){
    integer x = 2;
    while (x <= limit){
        boolean prime = true;
        for (i = 2; i < x; i++){
            if (x % 2 == 0){ prime = false; break; }
        }
        if (prime){ System.out.println(x + " is prime"); }
    }
}
```

FACTORING

Something in between?

```
LinkedList get_primes(int limit){
    int x = 2; LinkedList results = new LinkedList();
    while (x <= limit){
        boolean prime = true;
        for (i = 2; i < x; i++){
            if (x % 2 == 0){ prime = false; break; }
        }
        if (prime){ results.append(x); }
    }
}
void primes(int limit){
    for x in get_primes(limit){
        System.out.println(x + " is prime");
    }
}
```

FACTORING

- What you should factor depends on the context.
- How likely am I to need more number seives?
- How likely am I to do something other than print the primes?
- Try to find the right re-usability/time trade-off.

DEFACTORING

- Numbers such as the number 20 can be factored in different ways
  - 2,10
  - 4,5
  - 2,2,5
- If we have the factors 2 and 10, and realise that we want the number 4 included in the factorisation we can either:
  - Try to go directly by multiplying one factor and dividing the other, or
  - Defactor 2 and 10 back into 20, then divide 20 by 4.

DEFACTORING

- Similarly, your code is factored in some way.
- In order to obtain the factorisation that you desire, you may have to first **de**factor some of your code.
- This allows you to factor down into the desired components.
- This is often easier than trying to short-cut across factorisations.

SIEVE OF ERATOSTHENES

1. Create a list of consecutive integers from 2 to n: (2, 3, ..., n)
2. Initially, let p equal 2, the first prime number
3. Starting from p, count up in increments of p and mark each of these numbers greater than p itself in the list
   - These will be multiples of p: 2p, 3p, 4p, etc.; note that some of them may have already been marked.
4. Find the first number greater than p in the list that is not marked
   - If there was no such number, stop
   - Otherwise, let p now equal this number (which is the next prime), and repeat from step 3

SIEVE OF ERATOSTHENES

```
void primes(int limit){
    LinkedList prime_numbers = new LinkedList();
    boolean[] is_prime = new Array(limit, true);
    for (int i = 2; i ‹ Math.sqrt(limit); i++){
        if (is_prime[i]){
            prime_numbers.append(i);
            for (j = i * i; j ‹ limit; j += i){
                is_prime[j] = false;
    }
```

You can probably do this via our abstract number sieve class, but likely you don't want to. The alternative is to defactor back to close to our original version and then factor the way we want it.

DEFACTORING

- Flexibility is great, but it is generally not without cost.
  - The cognitive cost associated with understanding the more abstract code.
- If the flexibility is not now or unlikely to become *required* then it might be worthwhile defactoring.
- It is appropriate to explain your reasoning in comments.

REFACTORING SUMMARY

- Code should be factored into multiple components.
- Refactoring is the process of changing the division of components.
- Defactoring can help the process of changing the way the code is factored.
- Well factored code will be easier to understand.
- Do not update functionality at the same time.

SUGGESTED STRATEGY

- Note that this is merely a **suggested** strategy.
- Start with the simplest program possible.
- Incrementally add features based on the requirements.
- After each feature is added, refactor your code.
  - This step is important, it helps to avoid the risk of developing an unmaintainable mess.
  - Additionally it should be done with the goal of making future feature implementations easier.
  - This step includes janitorial work (discussed later).

SUGGESTED STRATEGY

- At each stage, you always have *something* that works.
- Although you need not specifically design for later features, you do at least know of them, and hence can avoid doing anything which will make those features particularly difficult.

ALTERNATIVE STRATEGY

- Design the whole system before you start.
- Work out all components and sub-components needed.
- Start with the sub-components which have no dependencies.
- Complete each sub-component at a time.
- Once all the dependencies of a component have been developed, choose that component to develop.
- Finally, put everything together to obtain the entire system, then test the entire system.

JANITORIAL WORK

- *Janitorial work* consists mainly of the following:
    - Reformatting,
    - Commenting,
    - Changing Names,
    - Tightening.

JANITORIAL WORK

REFORMATING

```
void method_name (int x)
{
  return x + 10;
}
```

Becomes:

```
void method_name(int x) {
  return x + 10;
}
```

There is plenty of software which will do this work for you as well.

JANITORIAL WORK

REFORMATTING

- Reformatting is entirely superficial.
- It is important to consider **when** you apply this.
- This may well conflict with other work performed concurrently.
- Reformatting should be largely unnecessary, if you keep your code formatting correctly in the first place.
  - More commonly required on group projects.

JANITORIAL WORK

COMMENTING

- Writing good comments in your code is **essential**.
- When done as janitorial work this can be particularly useful.
  - You can comment on the stuff that is not obvious even to yourself as you read it.
- The important thing to comment is not **what** or **how** but **why**.
- Try not to have redundant/obvious information in your comments:

```
// 'x' is the first integer argument
int leastCommonMultiple(int x, int y)
```

JANITORIAL WORK

COMMENTING

Ultra bad:

```
// increment x
x += 1;
```

Better:

```
// Since we now have an extra element to consider
// the count must be incremented
x += 1;
```

JANITORIAL WORK

CHANGING NAMES

- The previous example used **`x`** as a variable name.
- Unless it really is the x-axis of a graph, choose a better name.
- This is of course better to do the first time around.
- However as with commenting, unclear code can often be more obvious to its author upon later reading it.

JANITORIAL WORK

TIGHTENING

```
void main(...){
  run_simulation();
}
```

Tightened to become:

```
void main(...){
  try{
    run_simulation();
  } catch (FileNotFoundException e) {
    // Explain to the user ..
  }
}
```

JANITORIAL WORK

TIGHTENING

- For some developers this is not janitorial work, since it actually changes in a non-superficial way the function of the code.
- However, similar to other forms, it is often caused by being unable to think of every aspect involved when writing new code.

JANITORIAL WORK

- Most of this work is work that arguably could have been done right the first time around when the code was developed.
- However, when developing new code, you have limited cognitive capacity.
- You cannot think of everything when you develop new code. Janitorial work is your time to rectify the *minor* stuff you forgot.
- Better than trying to get it right first time is making sure you later review your code.

JANITORIAL WORK

- *Remember, refactoring is the process of changing code without changing its functionality, whilst improving design*.
- Strictly speaking *janitorial work* is **not** refactoring.
  - It should not change the function of the code,
    - (*Tightening* might, but generally for exceptional input only.)
  - but neither does it make the design any better.
- In common with refactoring you should not perform janitorial work on pre-existing code whilst developing new code.

COMMON APPROACH

- There is a common approach to developing applications
    1. Start with the main method
    2. Write some code, for example to parse the input
    3. Write (or update) a test input file
    4. Run your current application
    5. See if the output is what you expect
    6. Go back to step 2.

DO NOT START WITH MAIN

- A better place to start is with a test suite.
- This doesn't have to mean you cannot start coding.
- Write a couple of test inputs.
- Create a skeleton *“do nothing”* parse function.
- Create an entry point which simply calls your parse function on your test inputs (all of them).
- Watch them fail.

DO NOT START WITH MAIN

```
DataStructure parse_method(String input_string){
    return null;
}
void run_test(input){
    try { result = parse_method(input);
          if (result == null){
            System.out.println("Test failed by producing null");
          } else { System.out.println("Test passed"); }
    }
    catch (Exception e){
        System.out.println("Test raised an exception!");
    }}
test_input_one = "...";
test_input_two = "...";
void test_main(){
    run_test(test_input_one);
    run_test(test_input_two);
    ...
}
```

DO NOT START WITH MAIN

1. Code until those tests are green
    - Including possibly refactoring
2. Without forgetting to commit to git as appropriate
3. Consider new functionality
    - Write a method that tests for that new functionality
    - Watch it fail, whether by raising an exception or simply not producing the results required
    - Return to step 1.
4. You can write your `main` method any time you like
    - It should be very simple, as it simply calls all of your fully tested functionality

DO NOT START WITH MAIN

- Any time you run your code and examine the results, you should be examining output of tests
- If you are examining the output of your program ask yourself:
  - Why am I examining this output by hand and not automatically?
  - If I fix whatever is strange about the output can I be certain that I will never have to fix this again?
- Of course sometimes you need to examine the output of your program to determine **why** it is failing a test. This is just semantics (it is still the output of some test)

SUMMARY

- Everything your program outputs should be tested.
- Intermediate results that you might not output can still be tested as well.
- Run all of your tests, all of the time
    - It may take too long to run them all for each development run,
    - In which case, run them all before and after each commit.