Comments on CS1020 Mid-term Test (AY2015/6 Semester 2)

1. Section A: MCQs

The bonus question asks what the story Prof Tan told the class in his first lecture is. The correct answer is (A) Baby. Prof Tan specifically wanted to deliver the first lecture to share his favourite 'baby story', so it must have been quite a disappointment to him that many students answered wrongly.

A number of students answered (C) James Gosling. Prof Tan did mention Games Gosling in slide 6 of lecture #1 as the creator of Java, but it wasn't a story. However, we accepted this as the correct answer as well.

For the other 5 MCQs, the table below shows the percentage of students who chose the correct answers, and of those who chose the most popular wrong answers:

	Q1	Q2	Q3	Q4	Q5
%students who chose the correct answer	A (55.0%)	D (57.6%)	C (71.8%) Easiest	E (64.4%)	B (51.8%) Hardest
%students who chose the most popular wrong answer	C (16.2%)	E (28.8%)	B (16.2%)	B (21.5%)	E (34.7%)

The percentage of students who got the answers right is a bit on the low side. The easiest question is Q3, and the hardest seems to be Q5, with only half of the class got it right. Q5 is on basic OOP, so it is disappointing that many students didn't get it right. Please go through Q5 again to understand it if you have got it wrong.

2. Section B: Common mistakes

I will share the common mistakes we encountered in our marking.

Q6.

(i) ***Incorrect output format***

A number of students wrote only one integer. They may have missed out the method call to `print()` within `D2.f()`. Others wrote the output on two lines, failing to distinguish between `print()` and `println()`.

It is sad to see students adding the two integers to be printed together, or writing the '+' character in the output. There is a need to understand what the '+' **operator** does, for **different types of operands**.

(ii) ***Default constructor***

A number of students are not aware that Java **implicitly creates a default constructor** when there is **NO** constructor declared within a class. This constructor implicitly calls `super()` as its first and only statement. There is no error when we compile the code, and subsequently run it.

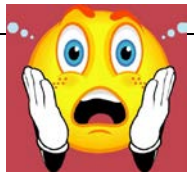
(iii) ***Failing to trace through*** **`D2.f()`** ***execution***

Many students do not understand pass-by-value, reference versus object, and what the `new` keyword does.

A good number printed "3 3" as the output, either wrongly thinking that there is only one reference `d1`, or one common object pointed to by each `d1` in the two different methods by the time we call the `print()` statement in `D2.f()`.

Tracing through program execution line-by-line, **drawing** the references in both methods and the objects they point to, **on paper**, will help greatly in ensuring that the values are correctly assigned and accessed.

If you do not understand references and objects well, how to draw them on paper, or how to trace through what a statement does using your diagram, please clarify your doubts with your tutor ASAP, or enquire on the IVLE forums. These are **fundamental** to Object Oriented Programming!



Q7i.

(i) ***"`m()` does not exist in class A"***

A number of students did not remember that B and C will **inherit public members from A**. Some students mixed up the direction in which methods are inherited.

A B object has `m()` because class A has `m()` defined, and class B inherits it from A.
A C object has `m()` because it inherits `m()` from B, and hence from A.
Every C is a B , which in turn is an A, so it has the `m()` behaviour.

"B inherits public members from A" is just a concept. How does Java implement it? Here, during compile-time:

Java checks whether `m()` exists in B, the datatype of the reference named `b`.
As m() does not exist in B, Java moves up the inheritance hierarchy.
If the method is still not found in `java.lang.Object`, compilation fails, as in Q7iii.

(ii) ***Confusion over inheriting from a class far up the inheritance hierarchy***

The `super` keyword cannot be used to invoke an implementation of a method two levels up the inheritance hierarchy, when the method has already been overridden one level up. A number of students confused that fact with a class not being able to inherit members of classes two levels up.

Within `C.n()`, we **cannot** write a statement to invoke `A.n()`.

However, **outside** class `C`, in `Q7.main()`, we **CAN** invoke `A.m()` using the reference `b`.

Therefore, the `super` keyword is used within the subclass to access a superclass' member, not from OUTSIDE the subclass. It is also important to distinguish between the two **different concepts**: inheritance versus overriding.

C extends B extends A is also **NOT a case of multiple inheritances**. C directly extends only one class. An example of multiple inheritances, which Java does NOT support, is

"Superman extends BOTH Bird AND Human".

(iii) *Runtime error versus compile-time error*

A call to `b.m()` in line 31 is valid (we are still at part i). However, some students mistakenly thought that method `m()` does not exist. **Even if** that were the case, we would encounter a compile-time error, rather than a runtime error. The **compiler** will ensure that a method call refers to a **matching method** that **exists in the datatype of the reference**, or above.

Q7ii.

(i) *Polymorphic behaviour*

Many students thought that `B.n()` will be executed. It is true that `b` is declared as a `B` reference, and that at compile-time, Java 'thinks' the program will execute `B.n()`. However, the method actually executed at **runtime** depends on the **datatype of the object**, NOT that of the reference. Since the object's datatype is `C`, `C.n()` will be executed at runtime.

(ii) `this.getClass().getName()`

As mentioned in the question, `this.getClass().getName()` returns the **datatype of the object**, not that of the reference! Again, as with `super`, the keyword `this` is only used within the service classes A, B and C to refer to the current instance, and not from the driver class Q7.

Q7iii.

(i) `"C.p in C"`

As explained in Q7i. point (i), at **compile-time**, Java finds `p()` based on the **datatype of the reference**, NOT that of the object.

(ii) "*Error as* C *cannot be accessed from* B"

Here, invoking `b.p()` from outside `C` has **nothing to do with accessibility or encapsulation**.

Accessibility - Within `C.n()`:

No access to a private variable in A, not even on the same object using `this`

Symbol not found - Outside of class `C`, in `Q7.main()`:

The compiler says there is no `p()` in B and above, because of the reference's datatype

(iii) "`b` *is a* `B` *object/instance, though instantiated with the* `C()` *constructor*"

This is one of the two most common mistakes in the entire Q7.

To be precise, `b` is a reference, not an object; hence `b` points (refers) to an object. Understand the difference between them.

- The datatype of the object (instance), pointed (referred) to by `b`, is `C`.
- The datatype of the reference is B.
- Even if we cast the reference, the datatype of `b` is still B, and that of the object is still `C`.

Q8.

(i) "F*'s next points to* G"

A number of students either traced through the code wrongly, or did not express the result properly on the diagram. Although there is no "standard" notation, F's next attribute pointing to G or G's next attribute or its value within confuses others and possibly yourself.

Ask yourself, should a **reference point to** an object or an attribute**?**

(ii) "null *object*"

null is a special value that can be assigned only to reference-typed variables. It means that the reference **does NOT point to any object**. There is **no "null object" in Java** (unlike Python).

(iii) "`curr2.getNext()` *is* null*, so* NullPointerException *will be thrown*"

A method returning a null is not a problem. **null is a valid value** for a **reference-typed** variable, and null can be assigned to `jump`. The exception occurs when we try to **de-reference null** using "`.`" or "`[`". Here, at line 7, there is no object to call the instance method getNext() on, since `jump` is null.

(iv) *Terminating loop before line* 6 / *after line* 7

To avoid encountering a NullPointerException when there are an even number of nodes, many students broke out of the loop just before line 5 when `curr2.getNext()` is null, or just before line 6 when `jump` is null, on the last iteration. This results in the tail node of the first list not being properly separated from the second list.

Performing checks on `jump` after line 7 will not solve the problem, as line 7 fails while running. Some students inserted code after line 9, but their code did not work for both even and odd sized lists.

(v) *Infinite loop*

Many students identified the problem that `getNext()` is being called using a null reference, the fact that the problem occurs when there are an even number of nodes, and tested whether `jump` is null after ensuring that the first list has its references properly set at line 6.

However, if lines 8 and 9 are only executed when `jump` is not null, then the program will not terminate when jump is null, because both `curr` and `curr2` are unable to advance past the last nodes in their respective lists.

Q9.

(i) ***The use of cast*** **(Integer)**

There are several possible answers for this question. Two versions are given in the Answers document, one of which is the following:

```
for (Integer num: numbers) {            // line 1
   while (num > 0) {                    // line 2
      int digit = num % 10;             // line 3
      digits.remove((Integer) digit);   // line 4
      num /= 10;                        // line 5
   }
}
```

Many students wrote a code similar to the above but did not include the type cast (Integer), that is, their line 4 looks like:

```
      digits.remove(digit);             // line 4
```

This does not work. Why?

Note that in the **ArrayList** API there are two overloaded **remove()** methods:

- remove(int index): Removes the element at the specified position in this list.
- remove(Object o): Removes the first occurrence of the specified element from this list, if it is present.

For example, if digit is 6, **`digits.remove(digit)`** will remove the element at position 6 in digits! That element may not be 6 as some elements might have been removed from digits by then.

Alternatively, we may declare **digit** as an Integer reference instead of int type in line 3:

```
Integer digit = num % 10;
```

Doing so would then eliminate the need of the type cast (Integer) in line 4.

(ii) ***Calling the same method several times and assigning a value to an accessor***

Quite a number of students wrote this:

```
   while (numbers.get(i) > 0) {             // line 1
      int digit = numbers.get(i) % 10;      // line 2
      digits.remove((Integer) digit);       // line 3
      numbers.get(i) = numbers.get(i)/10;   // line 4
   }
```

Note that **`numbers.get(i)`** is being called a few times in each iteration. This kind of code, as you should have learned in CS1010 (or equivalent), is bad because each time you call a method there is overhead incurred. You should call it once instead, assign the value to a variable, and use the variable thereafter, as shown in the Answers document.

The other more serious mistake is in line 4. It attempts to assign a value to **`numbers.get(i)`** which is an accessor! This is a compilation error. You should use a

mutator (eg: the **`set()`** method in **ArrayList**) to do this. However, if you had assigned **numbers.get(i)** to a variable as suggested above, this would not be an issue.

(iii) ***Using the wrong syntax on* ArrayList*, treating it like an array***

I saw a number of answers that look like this:

```
while (numbers[i] > 0) {
   int digit = numbers[i] % 10;
   digits.remove((Integer) digit);
   numbers[i] /= 10;
}
```

Students mixed up the syntax of ArrayList with array. We write **`arr[i]`** to access the i^{th} element of an array arr, but we write **`arrList.get(i)`** to access the i^{th} element of an ArrayList arrList.

The same mistake is encountered in Q10.

(iv) ***Using // for division***

I saw **`num = num//10;`** a number of times while marking, something which I have never seen before. Only later then I found out that // is integer division in Python, so this must have been written by CS1010S students. In Java, / is integer division if both the operands are integers, and // would make the rest of the line a comment.

(v) ***Using* String**

A number of students used String to solve this question. There is no need to use String here. Moreover, using String would incur overhead such as converting the integer into a string, and it would take up more space to hold the characters in the string. Note that in general, you use something only if its benefits outweigh the cost. In this case, the problem can be easily solved by sticking to ArrayList.

Q10 (a)

(i) ***Using the wrong syntax on* ArrayList*, treating it like an array***

As in Q9 point (iii), a number of students wrote **_vertices[i]** to access the i^{th} element in _vertices.

(ii) ***Missing one side or getting out of range***

Some students processed only n – 1 sides of an n-sided polygon as their 'for' loop runs from 0 to n – 2 as follows:

```
for (int i=0; i<_vertices.size()-1; i++)
```

Some students included all the sides, but the value (i+1) in **_vertices.get(i+1)** will be out of range when it comes to the last iteration (i.e. when i = n-1). Some students solved this by having an 'if' statement to separate the two cases, but this can be solved by using the mod operator % as shown in the Answers document.

One other solution is to add the first vertex into **_vertices** as the last vertex, so that now **_vertices** contains *n*+1 vertices, with the same first and last vertex. You can then run through the loop for i from 0 to *n*-1 without having to worry that (i+1) in the last iteration goes out of range. Remember to remove the last vertex from **_vertices** after that.

Q10 (b)

Many students made the mistake of writing:

```
int size = pts.size();
for (int i=0; i<size; i++) {
   for (int j=0; j<size; j++) {
      if (pts.get(i).equals(pts.get(j)))
         return false;
   }
}
```

Such issue should have been addressed in CS1010 (or equivalent). Here, when i==j, the points are the same and hence the code will always return false!

Besides the one shown in the Answers document, here is one acceptable algorithm which a few students used:

- Instead of using a nested loop, use a single loop with **indexOf()** and **lastIndexOf()** to check if they return different values; if so, then there must be duplicates.

```
for (Point pt: pts) {
   if (pts.indexOf(pt) != pts.lastIndexOf(pt))
      return false;
}
```

Ivan Chew and Aaron Tan
10 March 2016