CS2030 Programming Methodology
Semester 2 2019/2020

13 February 2020
Problem Set #4 Suggested Guidance

1. Consider a generic class `A<T>` with a type parameter `T` having a constructor with no argument. Which of the following expressions are valid (with no compilation error) ways of creating a new object of type `A`? We still consider the expression as valid if the Java compiler produces a warning.

   (a) `new A<int>()`

   (b) `new A<>()`

   (c) `new A()`

   (a) Error. A generic type cannot be a primitive type. You need to use a wrapper class, as is covered in Q2.

   (b) Valid. Java will create a new class replacing `T` *with Object.*

   (c) Valid as well. Same behaviour as above, but using raw type (for backwards compatibility) instead. Should be avoided in our class!

2. Given the following Java program fragment,

```
class Main {
    public static void main(String[] args) {
        double sum = 0.0;

        for (int i = 0; i < Integer.MAX_VALUE; i++) {
            sum += i;
        }
    }
}
```

   you can determine how long it takes to run the program using the `time` utility

```
$time java Main
```

   Now, replace `double` with the wrapper class `Double` instead. Determine how long it takes to run the program now. What inferences can you make?

   Despite its conveniences, there is an associated overhead in the use of autoboxing. In addition, due to the immutability of the wrapper class, many objects are created on the heap. So don't use the wrapper class for primitives unnecessarily!

3. Recall that the == operator compares only references, i.e. whether the two references are pointing to the same object. On the other hand, the `equals` method is more flexible in that it can override the method specified in the `Object` class.

   In particular, for the `Integer` class, the `equals` method has been overridden to compare if the corresponding `int` values are the same or otherwise.

   What do you think is the outcome of the following program fragment?

```
Integer x = 1;
Integer y = 1;
x == y

x = 1000;
y = 1000;
x == y
```

   Why do you think this happens? *Hint: check out Integer caching*

   We would expect the top fragment to be false since we are comparing object references. Since integers within a small range are very often used, it makes sense for the `Integer` *class to keep a cache of Integer objects within this range (-128 to 127) such that autoboxing, literals and uses of Integer.valueOf() will return instances from that cache instead.*

   Rather than concern oneself with the effects of caching or otherwise, the bottomline is to always use equals to compare two reference variables.

4. In the Java Collections Framework, `List` is an interface that is implemented by `ArrayList`. For each of the statements below, indicate if it is a valid statement with no compilation error. Explain why.

   (a)
```
void foo(List<?> list) { }

foo(new ArrayList<String>());
```

   (b)
```
void foo(List<? super Integer> list) { }

foo(new List<Object>());
```

   (c)
```
void foo(List<? extends Object> list) { }

foo(new ArrayList<Object>());
```

   (d)
```
void foo(List<? super Integer> list) { }

foo(new ArrayList<int>());
```

(e)
```
void foo(List<? super Integer> list) { }

foo(new ArrayList());
```

(a) Yes, since `ArrayList<String> <: List<String> <: List<?>`

(b) No, `List` *is an interface. It would be fine if we changed it to* `ArrayList<Object>` *since*
`ArrayList<Object> <: List<Object> <: List<? super Object> <: List<? super Integer>`

(c) Yes, since
`ArrayList<Object> <: ArrayList<? extends Object> <: List<? extends Object>`

(d) Error. A generic type cannot be primitive type.

(e) Compiles, but with a unchecked conversion warning. The use of raw type should also be generally be avoided.