CS2030 Programming Methodology

Semester 1 2019/2020

4 October 2019

Problem Set #5 Suggested Guidance

Local classes and Lambda Expressions

1. For each of the questions below, suppose the following is invoked:

```
B b = new B();
b.f();
```

Sketch the content of the stack, heap and metaspace *immediately after* the line

```
A a = new A();
```

is executed. Label the values and variables/fields clearly. You can assume `b` is already on the heap and you can ignore all other content of the stack and the heap before `b.f()` is called.

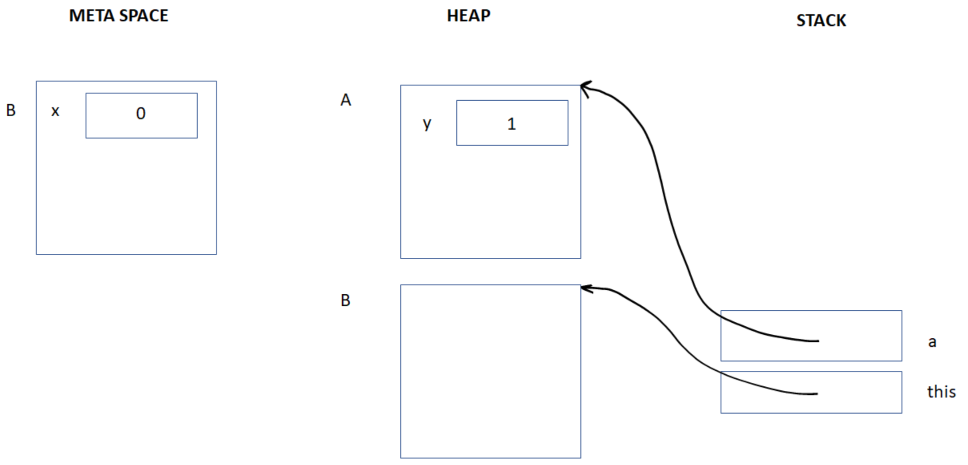
(a)

```
class B {
    static int x = 0;

    void f() {
        A a = new A();
    }

    static class A {
        int y = 0;

        A() {
            y = x + 1;
        }
    }
}
```

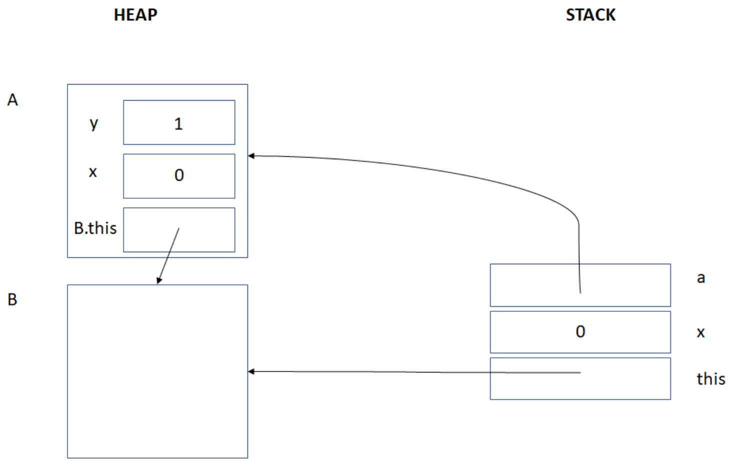


(b)

```
class B {
    void f() {
        int x = 0;

        class A {
            int y = 0;
            A() {
                y = x + 1;
            }
        }

        A a = new A();
    }
}
```



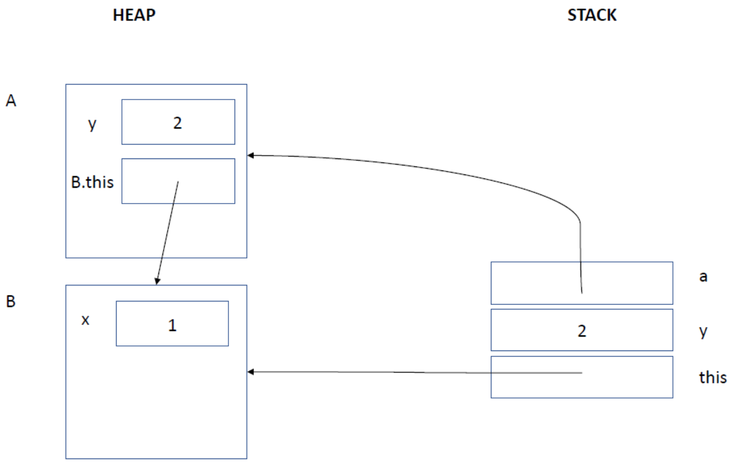
(c)

```
class B {
    int x = 1;

    void f() {
        int y = 2;

        class A {
            void g() {
                x = y;
            }
        }

        A a = new A();
        a.g();
    }
}
```



2. Java implements lambda expressions as anonymous classes. Suppose we have the following lambda expression `Function<String,Integer>`:

```
Function<String,Integer> findFirstSpace = str -> str.indexOf(' ');
```

Write the equivalent anonymous class for the expression above.

```
Function<String,Integer> findFirstSpace = new Function<>() {
    @Override
    public Integer apply(String str) {
        return str.indexOf(' ');
    }};
```

3. Suppose we have a class `A` that implements the following methods:

```
class A {
    int x;
    boolean isPositive;

    static A of(int x) {
        A a = new A();
        a.x = x;
        a.isPositive = (x > 0);
        return a;
    }

    A foo(Function<Integer, A> map) {
        return map.apply(this.x);
    }

    A bar(Function<Integer, A> map) {
        if (this.isPositive) {
            return map.apply(this.x);
        } else {
            return A.of(this.x);
        }
    }
}
```

Which of the following conditions hold for `A` for all values of `x`? `f` and `g` are both variables of type `Function<Integer,A>`; `a` is an object of type `A`.

(a) `A.of(x).foo(f)` always returns `f.apply(x)`

(b) `a.foo(f).bar(g)` equals to `a.foo(x -> f.apply(x).bar(g))`

(c) `a.bar(f).bar(g)` equals to `a.bar(x -> f.apply(x).bar(g))`

Solution: All of them.

First, let's understand what `A` does. `A` wraps around an `int` value `x`, along with the *context* of whether the value is positive or not.

The method `foo` simply applies the given method `f` on the internal value `x` and returns `f.apply(x)` (`map` is `f`). So (a) is true.

The method bar is a bit controversial — it selectively applies `f` on the internal value `x`. It applies `f` only if the value is positive, and leave the value untouched otherwise. To check if (b) and (c) hold, we can systematically analyze the cases.

Let `x` be the value contained in `a`. Let's check for (b).

- `a.foo(f)` is just `f.apply(x)`
- If `f.apply(x)` is not positive, `a.foo(f).bar(g)` is just `a.foo(f)`.
  `a.foo(x -> f.apply(x).bar(g))` is just `a.foo(x -> f.apply(x))`, which is just `a.foo(f)`.
- What if `f.apply(x)` is positive?
  Then `a.foo(f).bar(g)` is `g.apply(f.apply(x))` wrapped in `A`.
  `a.foo(x -> f.apply(x).bar(g))` is also `a.foo(x -> g.apply(f.apply(x)))`.
  Since `foo` applies the given lambda unconditionally, we get `g.apply(f.apply(x))` wrapped in `A` as well.

So (b) holds. Now let's check for (c).

- Suppose `x` is positive, then `a.bar(f)` is no different from `a.foo(f)`, the same argument above holds.
- Suppose `x` is non-positive, then `a.bar(f).bar(g)` is just `a`.
  `a.bar(x -> f.apply(x).bar(g))` is also just `a` (since `a.bar(...)` is `a`).

So (c) holds.

Question: Does `a.bar(f).foo(g) == a.bar(x -> f.apply(x).foo(g))` hold?

4. Write your own `Optional` class with the following skeleton:

```
class Optional<T> {
    private final T value;

    public static <T> Optional<T> of(T v) {
        :
    }

    public static <T> Optional<T> ofNullable(T v) {
        :
    }

    public static <T> Optional<T> empty(T v) {
        :
    }

    public void ifPresent(Consumer<? super T> consumer) {
        :
    }

    public Optional<T> filter(Predicate<? super T> predicate) {
        :
    }

    public <U> Optional<U> map(Function<? super T, ? extends U> mapper) {
        :
    }

    public<U> Optional<U> flatMap(Function<? super T, Optional<U>> mapper) {
        :
    }

    public T orElseGet(Supplier<? extends T> other) {
        :
    }
}
```

```java
import java.util.function.Consumer;
import java.util.function.Function;
import java.util.function.Supplier;
import java.util.function.Predicate;

class Optional<T> {
  private final T value;
  private static final Optional<?> EMPTY = new Optional<>(null);

  private Optional(T v) {
    this.value = v;
  }

  public static <T> Optional<T> of(T v) {
    if (v == null) {
      throw new NullPointerException();
    }
    return new Optional<>(v);
  }

  public static <T> Optional<T> ofNullable(T v) {
    if (v == null) {
      return empty();
    }
    return Optional.of(v);
  }

  public static <T> Optional<T> empty() {
    @SuppressWarnings("unchecked")
    Optional<T> o = (Optional<T>) EMPTY;
    return o;
  }

  public boolean isPresent() {
    return this.value != null;
  }

  public void ifPresent(Consumer<? super T> consumer) {
    if (isPresent()) {
      consumer.accept(this.value);
    }
  }

  public Optional<T> filter(Predicate<? super T> predicate) {
    if (!isPresent()) {
      return empty();
    }
    if (predicate.test(this.value)) {
      return this;
    }
    return empty();
  }

  public <U> Optional<U> map(Function<? super T, ? extends U> mapper) {
    if (!isPresent()) {
      return empty();
    }
    return Optional.ofNullable(mapper.apply(this.value));
  }

  public<U> Optional<U> flatMap(Function<? super T, Optional<U>> mapper) {
    if (!isPresent()) {
      return Optional.empty();
    }
    return mapper.apply(this.value);
  }

  public T orElseGet(Supplier<? extends T> other) {
    if (!isPresent()) {
      return other.get();
    } else {
      return this.value;
    }
  }

  public String toString() {
    return "Optional[" + this.value + "]";
  }
}
```