CS2030 Programming Methodology
Semester 1 2023/2024

18 & 19 October 2023
Problem Set #7 Suggested Guidance
Variable Capture

1. Study the following program fragment.

```
 1 abstract class A {
 2     abstract void g();
 3 }
 4
 5 class B {
 6     int x = 1;
 7
 8     void f() {
 9         int y = 2;
10
11         A a = new A() {
12             void g() {
13                 x = y;
14             }
15         };
16
17         a.g();
18     }
19 }
```

   Now suppose the following is invoked:

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

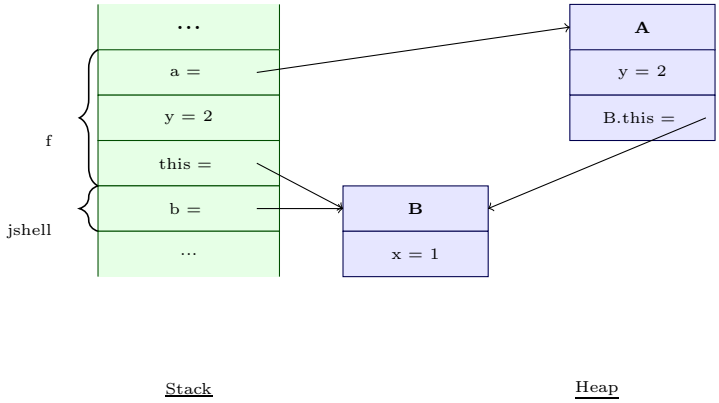
   Sketch the content of the stack and heap *just before* the statement in line 17 is executed. Label the values and variables/fields clearly. You can assume `b` is already on the heap and you can ignore all other content of the stack and the heap before `b.f()` is called.

   Line 13 should preferably be written as

```
B.this.x = y;
```

   The following will not work:

```
this.x = y;
```



The anonymous inner class (local class) captures the following:

- *a copy of variables of the enclosing method that it uses; and*
- *reference to the enclosing class via a qualified this, e.g.* `B.this`

Also note that variables of the enclosing method that is captured cannot be modified, i.e. it has to be final or effectively final.

```
class B {
    int x = 1;

    void f() {
        int y = 2;

        y = 10; // cannot be changed here

        A a = new A() {
            void g() {
                y = 10; // cannot be changed here
                x = y;
            }
        };

        y = 10; // cannot be changed here

        a.g();

        y = 10; // cannot be changed here
    }
}
```

2. You are given two functions $f(x) = 2 \times x$ and $g(x) = 2 + x$.

   (a) By creating an abstract class `Func` with an abstract method `apply`, evaluate $f(10)$ and $g(10)$.

```
abstract class Func {
    abstract int apply(int x);
}

Func f = new Func() {
    int apply(int x) {
        return 2 * x;
    }
}

Func g = new Func() {
    int apply(int x) {
        return 2 + x;
    }
}

f.apply(10)
g.apply(10)
```

```
jshell> Func f = new Func() {
   ...>     int apply(int x) {
   ...>         return 2 * x;
   ...>     }
   ...> }
f ==> 1@52cc8049

jshell> Func g = new Func() {
   ...>     int apply(int x) {
   ...>         return 2 + x;
   ...>     }
   ...> }
g ==> 1@312b1dae

jshell> f.apply(10)
$.. ==> 20

jshell> g.apply(10)
$.. ==> 12
```

   We cannot use a lambda here since `Func` *is not a functional interface.*

```
jshell> interface Func {
   ...>     int apply(int a)
   ...> }
|  created interface Func
```

```
jshell> Func f = x -> 2 * x;
f ==> $Lambda$20/0x0000000800c0a000@52cc8049

jshell> Func g = x -> 2 + x;
g ==> $Lambda$21/0x0000000800c0a428@312b1dae

jshell> f.apply(10)
$.. ==> 20

jshell> g.apply(10)
$.. ==> 12
```

(b) The composition of two functions is given by $f \circ g(x) = f(g(x))$. As an example, $f \circ g(10) = f(2+10) = (2+10) * 2 = 24$. Extend the abstract class in question 2a so as to support composition, i.e. `f.compose(g).apply(10)` will give 24.

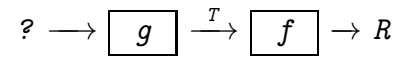
```
abstract class Func {
    abstract int apply(int a);

    Func compose(Func other) {
        return new Func() {
            int apply(int x) {
                return Func.this.apply(other.apply(x)); // <-- take note!
            }
        };
    }
}
```

```
jshell> Func f = new Func() {
   ...>     int apply(int x) {
   ...>         return 2 * x;
   ...>     }
   ...> }
f ==> 1@5b6f7412

jshell> Func g = new Func() {
   ...>     int apply(int x) {
   ...>         return 2 + x;
   ...>     }
   ...> }
g ==> 1@7530d0a

jshell> f.compose(g).apply(10)
$.. ==> 24
```

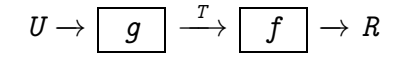
What happens if we replace the statement `return Func.this.apply(other.apply(x))` *with* `return this.apply(other.apply(x))` *instead? The* `apply` *method will recursive call itself! The* `this` *in* `Func.this` *is known as a "qualified this" and it refers not to it's own object, but the enclosing object. Here, the enclosing object's* `apply` *method is the one that returns* `2 * x`.

(c) Now re-implement the `Func` abstract class as generic abstract class `Func<T,R>` with the corresponding re-definitions of `apply` and `compose` methods.

Let's assume that `f.compose(g)` *where f is* `Func<T,R>`*, you can visualize the composition as* `g` *and then* `f`*:*



Since `g` *comes before* `f`*, it's output type must match the input type of* `f`*; the input type of* `g` *can be anything other than* `T` *or* `R`*, say* `U`*:*



Hence `other` *should be declared* `Func<U,T>`*, or more generally* `Func<? super U, ? extends T>`*.*

Finally, the return type of `compose` *should be* `Func<U,R>`*.*

```
abstract class Func<T,R> {
    abstract R apply(T t);

    <U> Func<U,R> compose(Func<? super U, ? extends T> other) {
        return new Func<U,R> () {
            R apply(U x) {
                return Func.this.apply(other.apply(x));
            }
        };
    }
}
```

```
jshell> Func<String, Integer> f = new Func<String, Integer>() {
   ...>     @Override
   ...>     Integer apply(String s) {
   ...>         return s.length();
   ...>     }
   ...> }
f ==> 1@5b6f7412

jshell> Func<Integer, String> g = new Func<Integer, String>() {
   ...>     @Override
   ...>     String apply(Integer x) {
   ...>         return x + "";
   ...>     }
   ...> }
g ==> 1@7530d0a

jshell> g.compose(f).apply("this") +
   ...> g.compose(f).apply("is") +
   ...> g.compose(f).apply("fun!!!")
$.. ==> "426"
```