Welcome to CS2030S Lab 8!

29 October 2021 [16A]

Please login to your pe node once it's 4pm.

DO NOT SCAN IF YOU'RE NOT HERE.

PA2 Alternative Arrangements

- PA2 falls on the Friday of Week 13 (**12/11/2021**) *Yay well-being!*
- Please let us know if you have to take tests immediately **before** or **after** your CS2030S lab (e.g. if you have lab from 1000-1200 and your test is at 1400 then it's not counted)
- We will book venues for you to take your test **if you need them** (PA2 will end around 10-15 minutes before the end of your lab slot so you can go elsewhere to take the test if you prefer)

TA Interest 😎

Please refer to the email sent out by Ivy Ng on 12/10/2021 for steps to follow if you're interested in becoming a TA for this mod next semester.

Don't worry about your grades at the moment.

Infinite List 🤯

1. Stream concepts
2. Lambda functions
3. Variable capture and closures
4. Immutability

Some Functional Interfaces

Observe the input/output carefully

Interface	Function Signature	Example
`Predicate<T>`	`boolean test(T t)`	`Collection::isEmpty`
`Function<T,R>`	`R apply(T t)`	`Arrays::asList`
`Supplier<T>`	`T get()`	`Math::random`
`Consumer<T>`	`void accept(T t)`	`System.out::println`
`UnaryOperator<T>`	`T apply(T t)`	`String::toLowerCase`
`BinaryOperator<T>`	`T apply(T t1, T t2)`	`BigInteger::add`

The Spirit of Laziness 💤

⚠️ Method references are **not lazy**! Stick to lambda expressions.

Postponing the creation of objects and sequence of evaluations (in the case of streams) for greater **efficiency**.

- Suppliers

  `() → t`

- Streams
- *Can get very complicated with multi-threading.

Variable capture and closures

```
jshell> void printMessage(String text, int repeat) {
   ...> Consumer consumer = whatever -> System.out.println(text);
   ...> for (int i = 0; i < repeat; i++)
   ...> consumer.accept(-99);
   ...> }
|  Warning:
|  unchecked call to accept(T) as a member of the raw type java.util.function.Consumer
|  consumer.accept(-99);
|  ^------------------^
|  created method printMessage(String,int)

jshell> printMessage("capturedText",3);
capturedText
capturedText
capturedText
```

How does `text` stay around when `consumer` is invoked?

Variable capture and closures

A lambda expression has three ingredients:

1. A block of code
2. Parameters
3. Values for the *free* variables—variables that are **not parameters** and **not defined inside the code**.

```
jshell> void printMessage(String text, int repeat) {
   ...> Consumer consumer = whatever -> System.out.println(text);
   ...> for (int i = 0; i < repeat; i++)
   ...> consumer.accept(-99);
   ...> }
```

We say `"capturedText"` has been *captured* by the lambda expression.

Variable capture and closures

In a lambda expression, you can only reference variables whose value doesn't change. Mutating variables in a lambda expression is not safe when multiple actions are executed concurrently.

```
jshell> void countDown(int start) {
   ...> Consumer consumer = whatever -> System.out.println(start--); ❌
   ...> for (int i = 0; i < start; i++)
   ...> consumer.accept(69);
   ...> }
|  Warning:
|  unchecked call to accept(T) as a member of the raw type java.util.function.Consumer
|  consumer.accept(69);
|  ^-----------------^
|  Error:
|  local variables referenced from a lambda expression must be final or effectively final
|  Consumer consumer = whatever -> System.out.println(start--);
|
```

An effectively final variable is a variable that is never assigned a new value after it has been initialised.

Variable capture and closures

```
jshell> void lambdaScopeDemo() {
   ...> String first = "I'm first!";
   ...> Comparator<String> comp = (first, second) -> first.length() - second.length();
   ...> }
|  Error:
|  variable first is already defined in method lambdaScopeDemo()
|  Comparator<String> comp = (first, second) -> first.length() - second.length();
|
```

Beware of using the same name in lambda expressions.

The technical term for a block of code together with the values of the free variables is a *closure*. In Java, lambda expressions are closures.

Anonymous (Inner) Classes

- You can instantiate a class without defining it proper.
- Eg. Anonymous class that implements `Function`:

```
Function<Integer, Integer> addOne = new Function<>() {
    @Override
    public Integer apply(Integer x) {
        return Integer.valueOf(x.intValue() + 1);
    }
};
```

Anonymous Inner Classes

```
abstract class A {
    abstract int get();
    static A get1() {
        return new A() {
            @Override
            int get() {return 99;}
        };
    }
    static A get2() {
        return new A() {
            @Override
            int get() {return 100;}
        };
    }
}
```

```
jshell> A.get1().get()
$1 ==> 99

jshell> A.get2().get()
$2 ==> 100
```

Treat today as a mock test!

Last chance to ask questions!