

Welcome to CS2030 Lab 8!

32 March 2022 [10J]



Only scan if you're physically
present!
LOGIN TO PLAB NOW!

PA2 Alternative Arrangements

- PA2 falls on the Friday of Week 12 (**08/04/2022**)
- Please let us know if you have to take tests immediately **before** or **after** your CS2030 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)

Lab 8

Actually no new content...

You can treat today's lab as a mock assessment to prepare you for PA2...

However, I have thrown in some of my old materials that may help a little bit?

TA Interest 😎

Please let me know if you're interested in becoming a TA next semester.

Don't worry about your grades at the moment.

Some Functional Interfaces

Observe the input/output carefully

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

The Spirit of Laziness z^z^z

⚠ 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

$() \rightarrow t$

- 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**.

```
jsell> 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
```



`Function<T, T>`



`UnaryOperator<T>`