CS2030S

Programming Methodology II

Lecture 08: Lambda and Lazy

Errata

Errata

ImmutableArray

ImmutableArray

Code

ImmutableArray_v1.java

```
class ImmutableArray<T> {
  private final T[] array;
  private final int start;
  private final int end;

  @SafeVarargs
  public static <T> ImmutableArray<T> of(T... items) {
    // We need to copy to ensure that it is truly immutable
    @SuppressWarnings("unchecked");
    T[] arr = (T[]) new Object[items.length];
    for (int i=0; i<items.length; i++) {
      arr[i] = items[i];
    }
    return new ImmutableArray<>(arr);
  }
    :
}
```

Notes

To truly make it immutable, we need to *copy* the items in the factory method of. Otherwise, there may still mutability.

Lambda

Lambda

Pure
- Definition
- Properties
- Side-Effects
First-Class
Functional
Lambda
Curry
Closure
Barrier

Pure Functions

Definition

A pure function *(denoted mathematically as $f : X \to Y$)* is a *mapping* from the domain X to the codomain Y. For each $x \in X$, there is $y \in Y$ such that $y = f(x)$. Additionally, the operation must be *deterministic*, without *side-effects*, and *referentially transparent*.

Lambda

Pure
- Definition
- Properties
- Side-Effects
First-Class
Functional
Lambda
Curry
Closure
Barrier

Pure Functions

Definition

A pure function *(denoted mathematically as $f : X \rightarrow Y$)* is a *mapping* from the domain X to the codomain Y. For each $x \in X$, there is a $y \in Y$ such that $y = f(x)$. Additionally, the operation must be *deterministic*, without *side-effects*, and *referentially transparent*.

Properties

- Deterministic

  for the same x, $f(x)$ must always return the same y

- Referentially Transparent

  any time we have $f(x)$ we can replace it with y and any time we have y we can replace it with $f(x)$ *(minus the actual computation performed as we are only interested in the result)*

Lambda

Pure
- Definition
- Properties
- Side-Effects
First-Class
Functional
Lambda
Curry
Closure
Barrier

Pure Functions

Side-Effects

> We say that the ***return value*** of the function is the ***"main"*** effect. Any other effects are side-effects.

Possible Side-Effects

1. Print to monitor
2. Write to files
3. Throw exceptions
4. Assign or mutate fields
5. *... any other effects visible by the caller*

Lambda

Pure
- Definition
- Properties
- Side-Effects
First-Class
Functional
Lambda
Curry
Closure
Barrier

Pure Functions

Side-Effects

> **Question**
>
> Consider the following functions?
>
> ```
> int incr(int i) {
>   return this.count + i;
> }
> ```
>
> ```
> int add(int i, int j) {
>   return i + j;
> }
> ```
>
> Which of the functions above are pure functions?

	Choice	Comment	
A	incr	*NO: return value depends on* `this.count`	✗
B	add	*YES: overflow is not an error!*	✓

Lambda

Pure
- *Definition*
- *Properties*
- ***Side-Effects***

First-Class
Functional
Lambda
Curry
Closure
Barrier

Pure Functions

Side-Effects

> **Question**
>
> Consider the following functions?
>
> ```
> int div(int i, int j) {
>   return i / j;
> }
> ```
>
> ```
> int square(int i) {
>   return i * i;
> }
> ```
>
> Which of the functions above are pure functions?

	Choice	Comment	
A	div	*NO: cannot divide by zero*	✗
B	square	*YES: overflow is not an error!*	✓

Lambda

Pure
- Definition
- Properties
- Side-Effects
First-Class
Functional
Lambda
Curry
Closure
Barrier

Pure Functions

Side-Effects

> **Question**
>
> Consider the following functions?
>
> ```java
> int dice() {
>   return rand.nextInt(6) + 1;
> } // returns 1 to 6
> ```
>
> ```java
> void incrCount(int i) {
>   this.count += 1;
> }
> ```
>
> Which of the functions above are pure functions?

	Choice	Comment	
A	dice	*NO: non-deterministic*	✖
B	incrCount	*NO: no return value + side effect*	✖

Lambda

Pure
First-Class
- Definition
- Method
- Towards
- Function
Functional
Lambda
Curry
Closure
Barrier

Function as First-Class Citizens

Definition

A programming language is said to have **first-class functions** when functions in that language are *treated like any other variables*.

Question

Which operation below still cannot be done if functions behave like any other variable?

	Choice	Comment	
A	Assign functions to variables	*NO: a variable can be assigned to other variables*	✗
B	Add two functions	*YES: this is only possible on numbers*	✓
C	Pass functions as arguments	*NO: a variable can be passed as arguments*	✗
D	Return as return value	*NO: a variable can be used as return value*	✗
E	Put into array	*NO: a variable can be put into array of correct type*	✗

Lambda

Pure
First-Class
- Definition
- Method
- Towards
- Function
Functional
Lambda
Curry
Closure
Barrier

Function as First-Class Citizens

Java Method

> Unfortunately, methods in Java are NOT first-class.

Not Allowed

```
int inc(int x) {
  return x + 1;
}

int apply(??? f, int x) {
  return f(x);
}
```

```
jshell> apply(inc, 2)
```

Lambda

Pure
First-Class
- Definition
- Method
- **Towards**
- Function
Functional
Lambda
Curry
Closure
Barrier

Function as First-Class Citizens

Towards First-Class Functions in Java

> 1. Create a class *(object is first-class citizen!)*.

Allowed

```
class Inc {
  int call(int x) {
    return x + 1;
  }
}
int apply(Inc f, int x) {
  return f.call(x);
}
```

```
jshell> Inc inc = new Inc()
inc ==> Inc@6e8cf4c6
jshell> apply(inc, 2)
$.. ==> 3
```

Notes

We need to create a new class for each first-class function we want to use.

Lambda

Pure
First-Class
- Definition
- Method
- Towards
- Function
Functional
Lambda
Curry
Closure
Barrier

Function as First-Class Citizens

Towards First-Class Functions in Java

> 2. Abstract into an interface *(now there can be many classes!)*.

Allowed

```
interface Fun {
  int call(int x);
  // the implementation is
  // given by the user
}
int apply(Fun f, int x) {
  return f.call(x);
}
```

```
class Inc implements Fun {
  @Override
  public int call(int x) {
    return x + 1;
  }
}
class Sqr implements Fun {
  @Override
  public int call(int x) {
    return x * x;
  }
}
```

Lambda

Pure
First-Class
- Definition
- Method
- Towards
- Function
Functional
Lambda
Curry
Closure
Barrier

Function as First-Class Citizens

Towards First-Class Functions in Java

> 3. Generalize types using generic *(now we don't just have to work with int!)*.

Allowed

```
interface Fun<T, R> {
  R call(T x);
  // T: argument
  // R: return
} // Fun : T -> R
<T,R> R apply(Fun<? extends T, ? super R> f, T x) {
  return f.call(x);
}
```

Lambda

Pure
First-Class
- Definition
- Method
- Towards
- Function
Functional
Lambda
Curry
Closure
Barrier

Function as First-Class Citizens

Function <T,R>

Lambda

Pure
First-Class
- Definition
- Method
- Towards
- Function
Functional
Lambda
Curry
Closure
Barrier

Function as First-Class Citizens

Function <T,R>

> **Question**
>
> If Fun<T,R> is an interface for a function with one parameter, what is the interface for function with *two* parameters?
>
> Use the generic types T1 for the first parameter, T2 for the second parameter, and R for the return type.

	Choice	Comment	
A	Fun<T1, R extends <Fun<T2, R2>>>	*NO: what are T2 and R2?*	✗
B	Fun<T1, Fun<T2, R>>	*NO: what are T2 and R?*	✗
C	Fun<T1, T2, R>	*YES: all three are type parameters*	✓
D	Fun<<T1, T2>, R>	*NO: this is a syntax error*	✗

Lambda

Pure
First-Class
Functional
- Definition
Lambda
Curry
Closure
Barrier

Functional Interface

Definition

A functional interface is an interface with a *single abstract method*.

Annotation

- We can annotate functional interface with `@FunctionalInterface` annotation
- A functional interface can be used as the assignment target for a *lambda expression* or *method reference*

Transformer

```
@FunctionalInterface
interface Transformer<T,U> {
  U transform(T t);
}
```

Notes

From now on, the lecture notes will be using `Function` as defined in Java.

Lambda

Pure
First-Class
Functional
Lambda
- Definition
- Towards
- Syntax
Curry
Closure
Barrier

Lambda Expression

Definition

A lambda expression is an *anonymous* function.

Lambda

Pure
First-Class
Functional
Lambda
- *Definition*
- ***Towards***
- *Syntax*
Curry
Closure
Barrier

Lambda Expression

Definition

A lambda expression is an *anonymous* function.

Towards Lambda

Named Class

```
class Inc implements
    Function<Integer, Integer> {
  @Override
  public int call(int x) {
    return x + 1;
  }
}
```

```
Inc f = new Inc();
```

Anonymous Class

```
Function<Integer, Integer> f =
  new Function<>() {
    @Override
    Integer call(Integer x) {
      return x + 1;
    }
  };
```

```
// Any shorthand?
```

Lambda

Pure
First-Class
Functional
Lambda
- Definition
- Towards
- Syntax
Curry
Closure
Barrier

Lambda Expression

Syntax

Single Parameter

```
param -> expr
```

No Parameter

```
() -> expr
```

Multiple Parameters

```
(param1, param2) -> expr  // can have as many param as needed
```

Multiple Statements

```
(param1, param2, param3) -> { body; return expr; }
```

Lambda

Pure
First-Class
Functional
Lambda
Curry
- Motivation
- Definition
- Example
Closure
Barrier

Curried Functions

Motivation

> Consider functions that return a value. How do we create an interface for functions with
>
> 1 parameter

Lambda

Pure
First-Class
Functional
Lambda
Curry
- Motivation
- Definition
- Example
Closure
Barrier

Curried Functions

Motivation

> Consider functions that return a value. How do we create an interface for functions with
>
> | | |
> |---|---|
> | 1 parameter | Function1<T, R> |
> | 2 parameters | |

Lambda

Pure
First-Class
Functional
Lambda
Curry
- Motivation
- Definition
- Example
Closure
Barrier

Curried Functions

Motivation

> Consider functions that return a value. How do we create an interface for functions with
>
> | | |
> |---|---|
> | 1 parameter | Function1<T, R> |
> | 2 parameters | Function2<T1, T2, R> |
> | 3 parameters | |

Lambda

Pure
First-Class
Functional
Lambda
Curry
- Motivation
- Definition
- Example
Closure
Barrier

Curried Functions

Motivation

> Consider functions that return a value. How do we create an interface for functions with
>
> | | |
> |---|---|
> | 1 parameter | Function1<T, R> |
> | 2 parameters | Function2<T1, T2, R> |
> | 3 parameters | Function3<T1, T2, T3, R> |
>
> Is there a limit? Can we not make a *general* interface for all possible number of parameters?

Lambda

Pure
First-Class
Functional
Lambda
Curry
- Motivation
- Definition
- Example
Closure
Barrier

Curried Functions

Definition

Currying is a technique to convert a function that takes multiple arguments into a *sequence* of functions that each *takes a single argument*.

Example

Two Arguments

```
BiFunction<Integer,Integer,Integer> f = (x, y) -> x + y;
f.apply(1, 2);
```

One Argument

```
Function<???, ???> f = x -> ???;
f.apply(1); // then what?
```

Lambda

Pure
First-Class
Functional
Lambda
Curry
- Motivation
- Definition
- Example
Closure
Barrier

Curried Functions

Definition

Currying is a technique to convert a function that takes multiple arguments into a *sequence* of functions that each *takes a single argument*.

Example

Two Arguments

```
BiFunction<Integer,Integer,Integer> f = (x, y) -> x + y;
f.apply(1, 2);
```

One Argument

```
Function<Integer, Function<Integer, Integer>> f = x -> y -> x + y;
f.apply(1).apply(2);
```

#The lambda expression is read from right-to-left so it is equivalent to x -> (y -> (x + y)).

Lambda

Pure
First-Class
Functional
Lambda
Curry
Closure
Barrier

Lambda as Closure

Code

Point

```
class Point {
  // code omitted
  public double distance(Point p) {
    // code omitted
  }
}
```

Transformer

```
Point origin = new Point(0, 0);
Function<Point, Double> dist =
  p -> origin.distance(p);

// Recap, 'origin' needs to be
//   either final or effectively final
```

Method Reference

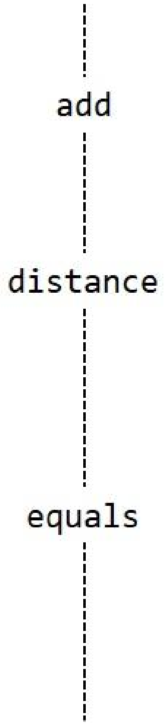
```
Point origin = new Point(0, 0);
Function<Point, Double> dist = origin::distance;
```

Lambda

Pure
First-Class
Functional
Lambda
Curry
Closure
Barrier
- Previously
- Now

Lambda as Abstraction Barrier

Previously

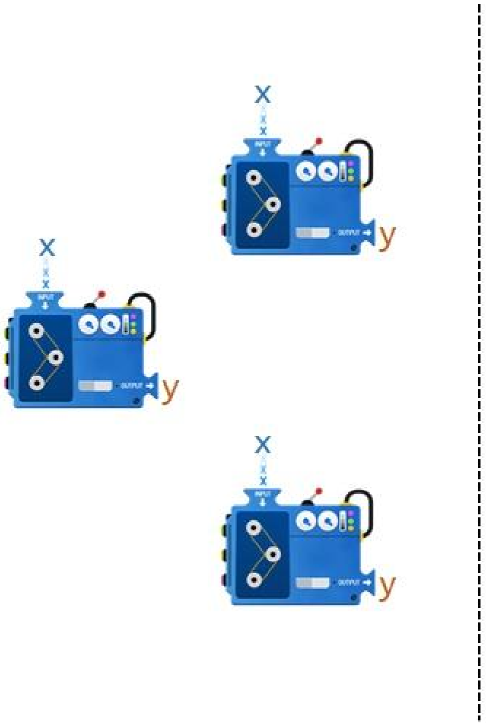


Lambda

Pure
First-Class
Functional
Lambda
Curry
Closure
Barrier
- Previously
- Now

Lambda as Abstraction Barrier

Now



Lazy

Lazy

Eager
Lazy
Lazy<T>

Eager Evaluation

Logger

Code

Logger_v0.java

```
class Logger {
  enum LogLevel { INFO, WARNING, ERROR };
  public static LogLevel currLogLevel = LogLevel.WARNING;
  static void log(LogLevel level, String msg) {
    if (level.compareTo(Logger.currLogLevel) >= 0) {
      System.out.println(" [" + level + "] " + msg);
    }
  }
}
```

Other Interface

- Producer<T>
- Consumer<T>
- Task

Notes

Producer<T> is as defined in Lab 5.

Lazy

Eager
Lazy
Lazy<T>

Eager Evaluation

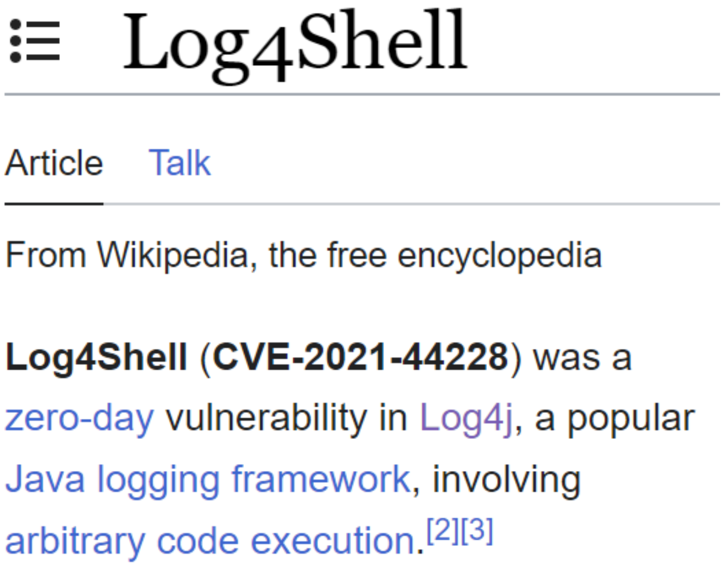
Logger

Lazy

Eager
Lazy
Lazy<T>

Eager Evaluation

Logger



Log4Shell

Article Talk

From Wikipedia, the free encyclopedia

Log4Shell (**CVE-2021-44228**) was a zero-day vulnerability in Log4j, a popular Java logging framework, involving arbitrary code execution.[2][3]

Status: Installing Java

Java Setup - Progress
Java
ORACLE
3 Billion Devices Run Java
Computers, Printers, Routers, Cell Phones, BlackBerry, Kindle, Parking Meters, Public Transportation Passes, ATMs, Credit Cards, Home Security Systems, Cable Boxes, TVs...
ORACLE

Lazy

Eager
Lazy
- How
- Procrastinate
- Never Repeat
Lazy<T>

Lazy Evaluation

How to be Lazy

Procrastinate Until the Last Minute

> Do not perform the computation. *Produce* it when needed.

Never Repeat Yourself

> Do not perform the computation twice. Also known as *memoization*.

Lazy

Eager
Lazy
- How
- Procrastinate
- Never Repeat
Lazy<T>

Lazy Evaluation

How to be Lazy

Procrastinate Until the Last Minute

> Do not perform the computation. *Produce* it when needed.

Code

Logger_v1.java

```
class Logger {
  enum LogLevel { INFO, WARNING, ERROR };
  public static LogLevel currLogLevel = LogLevel.WARNING;
  static void log(LogLevel level, Producer<String> msg) {
    if (level.compareTo(Logger.currLogLevel) >= 0) {
      System.out.println(" [" + level + "] " + msg.produce());
    }
  }
}
```

Lazy

Eager
Lazy
- How
- Procrastinate
- **Never Repeat**
Lazy<T>

Lazy Evaluation

How to be Lazy

Never Repeat Yourself

> Do not perform the computation twice. Also known as *memoization*.

Code

Logger_v2.java

```java
class Logger {
  enum LogLevel { INFO, WARNING, ERROR };
  public static LogLevel currLogLevel = LogLevel.WARNING;
  static void log(LogLevel level, Lazy<String> msg) {
    if (level.compareTo(Logger.currLogLevel) >= 0) {
      System.out.println(" [" + level + "] " + msg.get());
    }
  }
}
```

Lazy

Eager
Lazy
Lazy<T>
- Idea
- Bad Code

Lazy<T>

Idea

Fields

- T
- Producer<T>

Convention

- Producer is one-time use
  - If the value is non-null, it can be used
  - If the value is null, it is already used
- Once used, set this to null

Caution

This convention is different from the notes due to space limitation of the slide. The convention in the note is a better convention.

In fact, you will create an even better one in Lab 6!

Lazy

Eager
Lazy
Lazy<T>
- Idea
- Bad Code

Lazy<T>

Bad Code

Lazy.java

```java
class Lazy<T> {
  T value;
  Producer<T> producer;
  // the better approach is to use boolean isAvailable like in the notes
  public Lazy(Producer<T> producer) {
    this.producer = producer;
    this.value = null;
  }
  public T get() {
    if (this.producer != null) {        // can be used!
      this.value = producer.produce();  // use it
      this.producer = null;             // prevent other uses
    }
    return this.value;
  }
}
```

```
jshell> /exit
|       Goodbye
```