Functional Interfaces

Terence Sim

Motivation

```
<T> T max3(T a, T b, T c, compare) {
    T max = a;
    if (compare(b, max) > 0) {
        max = b;
    }
    if (compare(c, max) > 0) {
        max = c;
    }
    return max;
}
```

`max3` is a generic method to find the largest of its 3 inputs, but it needs a `compare` function that works for the generic type `T`. But this code won't compile.

Motivation

```
<T> T max3(T a, T b, T c, Comparator<T> ct) {
    T max = a;
    if (ct.compare(b, max) > 0) {
        max = b;
    }
    if (ct.compare(c, max) > 0) {
        max = c;
    }
    return max;
}
```

Java doesn't accept functions as inputs. The next best thing is to wrap the function in a class or interface, and pass that in. Now the code compiles correctly.

Functional Interface

It is an interface that has a *single abstract method* (SAM), which must be overridden by the user.

Two purposes:

1. As a way of passing functions into other functions, or returning a function as a value, or for assigning a function to a variable.

   Functions thus become *first-class objects*, like other objects.

2. For the compiler to check that the caller of `max3` is indeed using a type that has a `compare` method.

   The compiler checks that the type *implements* that interface.

Calling `max3`

```
class IntegerComparison implements
      Comparator<Integer> {
    public int compare(Integer x, Integer y) {
        return x-y;
    }
}

int foo() {
    int largest = max3(-1, 2, -3, new
                        IntegerComparison());
}
```

This is one way to call `max3`. We will see easier ways…

Many pre-defined functional interfaces

```
Comparator<T>:  int compare(T x, T y)
```

In `java.util.function`

https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/function/package-summary.html

```
Predicate<T>:   boolean test(T x)
Function<T,R>:  R apply(T x)
Supplier<T>:    T get()
Consumer<T>:    void accept(T x)
```

eg: showing pass/fail

```java
import java.util.function.Predicate;

<T> void showPassFail(T thing, Predicate<T> isPass) {
    if (isPass.test(thing))
        System.out.println("Pass");
    else
        System.out.println("Fail");
}
```

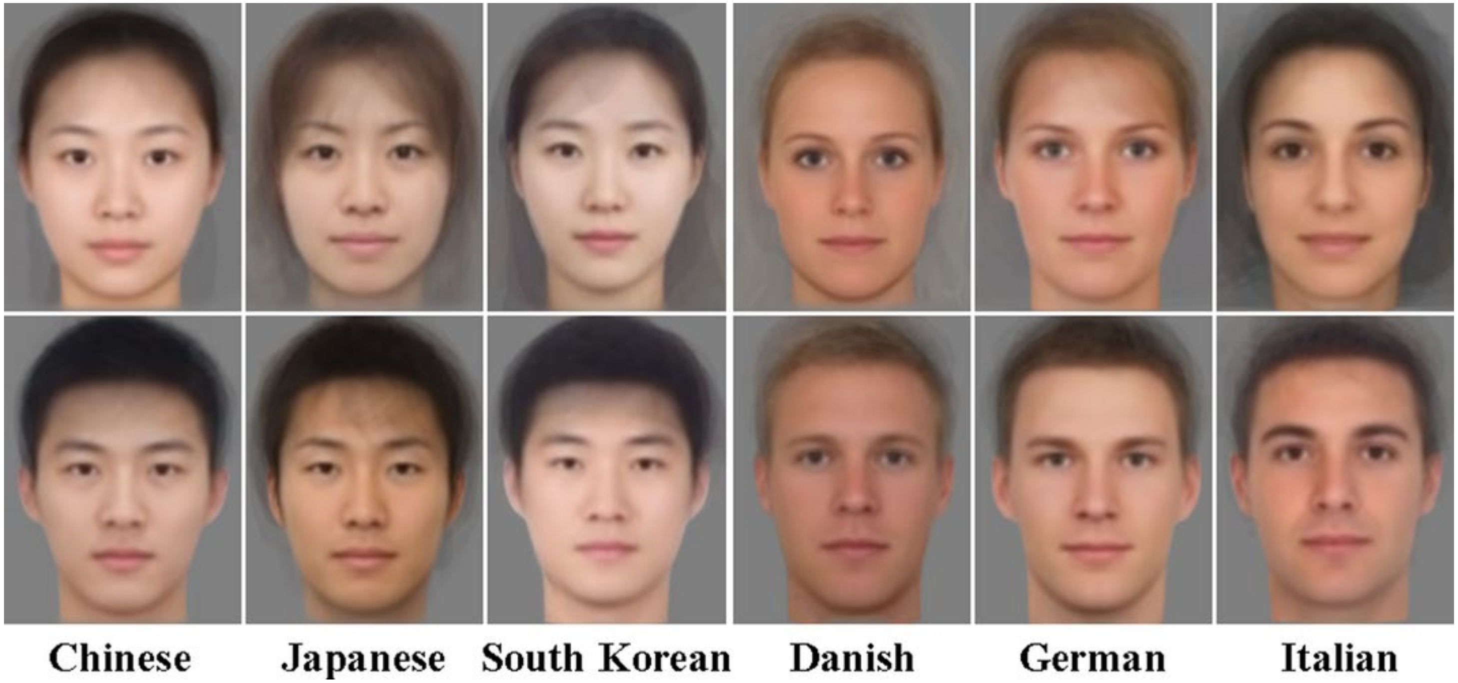
eg: averaging a list of things

$$\overline{X} = \frac{1}{N} \sum_{i=1}^{N} X_i$$

```
import java.util.function.BinaryOperator;
import java.util.function.BiFunction;

<T> T average(List<T> list, BinaryOperator<T> add,
              BiFunction<T,Double,T> scale) {
    // assume list is non-empty
    T sum = list.get(0);
    int size = list.size();
    for (T item : list.subList(1, size))
        sum = add.apply(sum, item);
    return scale.apply(sum, new Double(1.0/size));
}
```

Average male and female faces



It is possible to average face images!

Some terminology

Higher-order function

The function that accepts the functional interface as input, and uses the interface in its body. eg: `max3`

Caller

The one who calls the higher-order function. eg: `foo`

Is there a better way?

```
class IntegerComparison implements
Comparator<Integer> {
    public int compare(Integer x, Integer y) {
        return x-y;
    }
}
max3(-1, 2, -3, new IntegerComparison() );
```

Notice that the caller has to

- create a single-use class just to implement the interface,
- create an instance of the class to pass into the higher-order function.

Is there a better way?

```
max3(-1, 2, -3, new Comparator<Integer> () {
    public int compare(Integer x, Integer y) {
        return x-y;
    }
})
```

Use anonymous class instead of single-use class.

Lambda expressions

```
max3(-1, 2, -3,
    (Integer x, Integer y) -> {
        return x-y;
    })
```

Use lambda expression! Think of it as an anonymous function.

Named after λ-calculus by Alonzo Church in the 1930s.

https://en.wikipedia.org/wiki/Lambda_calculus

Lambda expression: syntax

```
parameters -> statement-body
```

```
(Integer x, Integer y) -> {return x-y;}
```

OR

```
(x,y) -> x-y
```

Compiler infers the type of `x,y` from usage.

Omit `return` if body has single expression.

Single parameter:

```
x -> x*x  //drop the () if single parameter
```

no parameter:

```
() -> System.out.println("CS2030 is fun!");
```

Averaging

```
BinaryOperator<Double> addDouble = (x,y)->x+y;

BinaryOperator<Double> multDouble =
      (x,s)->x*s;

List<Double> numberList =
Arrays.asList(1.0,2.0,3.0,4.0,5.0,6.0,7.0,8.0,9.
      0,10.0);


average(numberList, addDouble, multDouble);

=> 5.5
```

Note the assignment of lambdas to variables.

Note also that
`BinaryOperator<T> extends BiFunction<T,T,T>`

Method reference

If all your lambda does is to call another method, you can directly pass the method instead:

class-name :: *method-name*

The code below uses the `compareTo` method to perform the actual comparison of strings.

There are a few variants of method references, see

https://docs.oracle.com/javase/tutorial/java/javaOO/methodreferences.html

```
max3("hello","goodbye","zoo",(x,y)->x.compareTo(y));
      is equivalent to:
max3("hello","goodbye","zoo",String::compareTo);
```

showPassFail version 2.0

Instead of fixing `System.out.println` in our code, we can defer display to another function, via the `Consumer` interface.

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


<T> void showPassFail(T thing, Predicate<T> isPass,
      Consumer<Boolean> display) {
    display.accept(isPass.test(thing));
}


showPassFail(5, (x -> x>10), System.out::println);
=> false
```

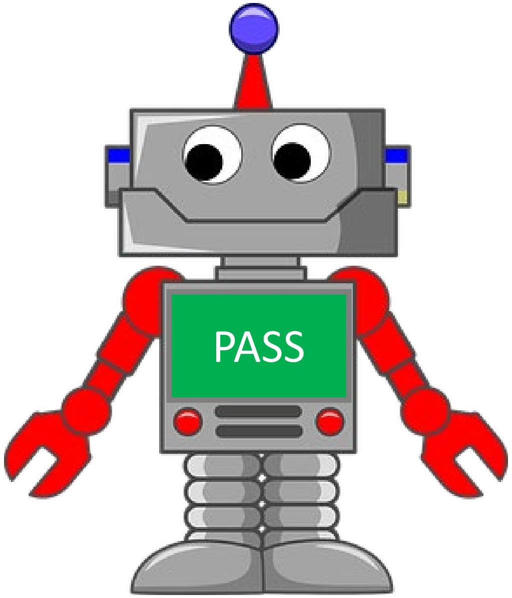
showPassFail version 2.0

```
class CabDriver {
    private int drivingExperience = 0;

    public CabDriver(int years) {
        this.drivingExperience = years;}

    public int drivingExperience() {
        return this.drivingExperience;}
}

showPassFail(new CabDriver(20),
        x-> x.drivingExperience() > 10,
         Robot::display));
```



Lambda closure

Lambda is not just syntactic sugar. It captures the enclosing scope where it was defined (called *lexical scoping*).

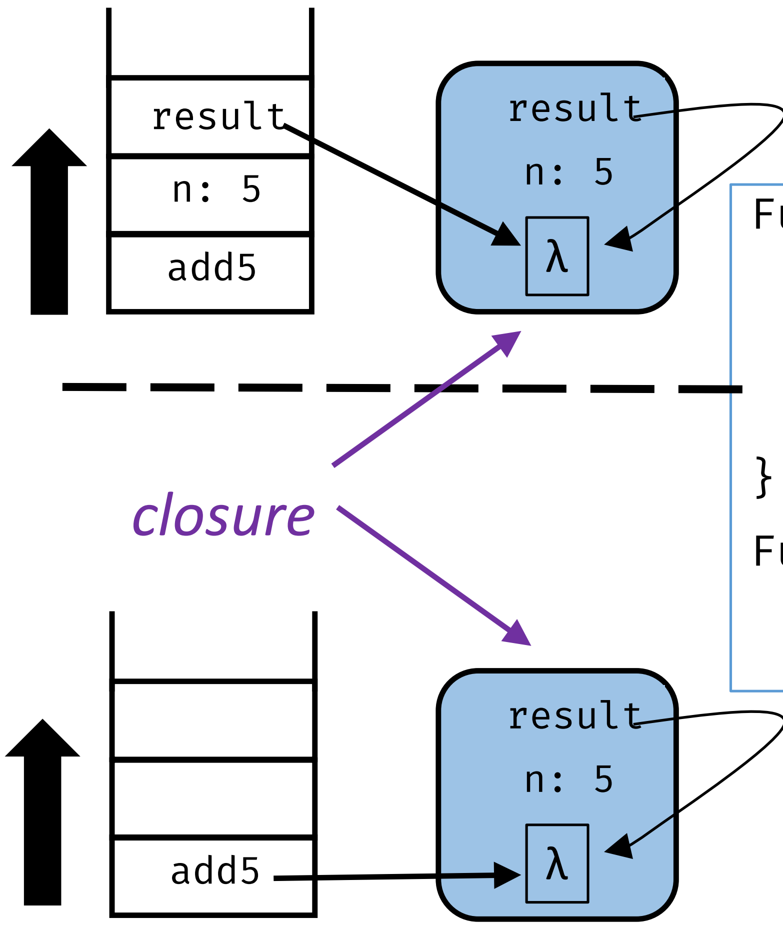
```
Function<Integer, Integer> addN(int n) {
    Function<Integer,Integer> result = x -> x+n;
    return result;
}
Function<Integer, Integer> add5 = addN(5);
add5.apply(7);
=> 12
```

The variables `n` and `result` are copied from the stack into a *closure* object (in the heap).

These variables must be final or *effectively final* (compiler will flag error if they are changed.)

Memory model

Just before `addN` returns




```
Function<Integer, Integer> addN(int n) {
    Function<Integer,Integer> result =
        x -> x+n;
    return result;
}
Function<Integer, Integer> add5 =
        addN(5);
```

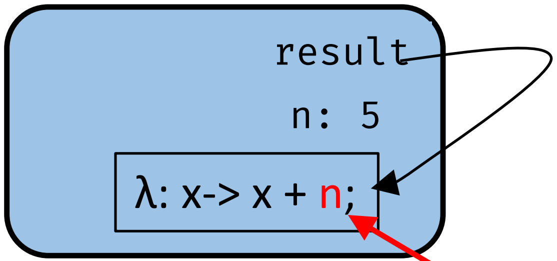
After `addN` returns

Page added

Lambda closure

When the lambda is called, its arguments (if any) are pushed onto the stack just like a normal function call*, but its body is evaluated in the closure.*

That is, any free variables (ie. non-parameter variables) in the body are looked up in the closure.



When `add5.apply(7)` is called, the free variable n is looked up in the closure, which has the value of 5.

Lambda closure

```
Function<Integer, Integer> addN(int n) {
    Function<Integer,Integer> result = x -> x+n;
    n++; //Compile error: variable changed.
    return result;
}
```

```
Function<Integer, Integer> addN(int n) {
   return n -> 2+n;
//Compile error: lambda parameter has same name as
variable in enclosing scope.
}
```

We will use closures for something even more cool in the future! 😉

Function composition

$$f^2(x) \equiv (f \circ f)(x) \equiv f(f(x))$$

```
Function<Integer, Integer> twice(Function<Integer,
      Integer> f) {
    return f.andThen(f);
}


Function<Integer, Integer> mystery =
      twice(twice(x->x*x));
mystery.apply(2);
=> ??
```

https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/function/Function.html#andThen(java.util.function.Function)

Application
Simple command-line calculator

Select Ubuntu 18.04 LTS

```
tsim@DESKTO-TD4SNLQ:~/java=> java Calc
[1]=> add 3 5
8.000000
[2]=> mult -2 38
-76.000000
[3]=> % 25 80
20.000000
[4]=> recip 10
0.100000
[5]=>
```

The Read-Eval-Print Loop (REPL)

```
public class Calc {
    public static void main(String[] args) {
        Statement.initialize();

        try {
            while (true) {
                Input input = Input.readInput();
                Statement statement = Statement.parse(input);
                Result result = statement.evaluate();
                result.display();
            }
        } catch (NoSuchElementException e) {
            //break out of while loop; end program
        }
    }
}
```

Key idea: data-directed programming

key	value
`“add”`	`x -> x[0] + x[1]`
`“mult”`	`x -> x[0] * x[1]`
`“recip”`	`x -> 1.0 / x[0]`
`“%”`	`x -> x[0] * x[1] / 100.0`

We use a table to associate a lambda (that performs the actual calculation) with the name of the calculation.

The `Hashtable<K,V>` class stores entries of key-value pairs, along with `get()` and `put()` to retrieve and add entries, respectively.

also called *dictionary* in other languages, eg. Python

Think of `Hashtable` as an array, where the index can be another data type, not just int. Getting and putting can be done in $O(1)$ time.

https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/Hashtable.html

Initializing the Hashtable

```
private static Hashtable<String,
       Function<Double[],Double>> commandTable;

private static Function<Double[],Double>
        addition = x -> x[0] + x[1];

private static Function<Double[],Double>
        multiplication = x -> x[0] * x[1];

private static Function<Double[],Double>
        reciprocal = x -> 1.0 / x[0];

private static Function<Double[],Double>
        percentage = x -> x[0] * x[1] / 100.0;
```

Initializing the Hashtable

```
public static void initialize() {
	commandTable = new Hashtable<>();
	commandTable.put("add", addition);
	commandTable.put("mult", multiplication);
	commandTable.put("recip", reciprocal);
	commandTable.put("%", percentage);
}
```

Evaluating the input

```
//Members in the Statement class
private String command;
private Double[] arguments;
private String message;

public Result evaluate() {
       double r = commandTable.get(this.command)
                     .apply(this.arguments);
       this.message = String.format("%f", r);

       return new Result(this.message);
}
```

The user's input is parsed to store the command in `command`, and the args in `arguments`. Then `evaluate()` is called to get the corresponding function from the `Hashtable` and apply it to the arguments.

Remarks

This style of *data-directed programming*, by using a table to select the correct function, is a powerful one.

New commands can be easily added, and old ones removed or modified, just by changing the table entry.

> Think of how to add a “sub” command for subtraction.

Full executable code is available in Luminus Files.

> But the code contains a lot of error checking, which makes it messy. 😩
>
> We will clean up the code in future.

Lecture Summary

- Functional Interfaces are the means by which Java allows functions to be first-class objects.
- There are many functional interfaces in the Java API.
  - Use them where appropriate. Create your own only when necessary.
- Lambda expressions are anonymous functions that create closures.
- Method references allow any method to be passed.
- Data-directed programming is a powerful strategy that permits a datum to select the appropriate function to be executed.