COMP6700/2140 Functional Interfaces

Alexei B Khorev

Research School of Computer Science, ANU

March/April 2017

Topics

1. Again: λ-expression $\overset{\text{sort of}}{\equiv}$ instance of FI
2. Predefined Functional Interfaces
3. Method References: reuse of pre-existing code
4. Default Methods
   - Default Methods in Functional Interfaces
   - Extending Interfaces with Default Methods
5. Closures
6. Identity, state and behaviour and λ-s
7. Exceptions in λ-expression's
8. Exceptions vs Optional
9. More on Interfaces:
   - Mixins
   - Traits

Lambda Expressions: Recap and Some Subtleties

λ expression syntax

```
(parameters) -> expression // for simple codes

(parameters) -> { statements; } // for multi-statement codes
```

1. λ-expression's allow to pass code to defined methods for later execution.
2. λ-expression can be assigned to a variable of a functional interface type.
3. Often, λ-expression's are treated as literal data, objects which represent them in JVM do not have identity (equlas-method does not have consistent semantics — you can experiment with this), and their bytecode is not stored in classes, but instead is executed by the `invokedynamic` JVM instruction.
4. A λ-expression can be recursive: make a reference to itself (via an assigned reference) inside the λ-body (on RHS of `"->"`).
5. A λ-expression can throw an exception like an ordinary exception or constructor (Java API functional interfaces do not throw, you may need to define your own FI, or catch an exception when assigning to or passing λ).
6. The most prolific use of λ's has been in the new style of collection processing which relies on *internal iteration* (without explicit reference to an iterator). On the level of API, such processing is implemented in `java.util.stream` package and various default methods added to pre-existed interfaces (like `java.util.List` and many others).

Functional Interfaces API

Most often used functional interfaces, defined by the combination of `parameters -> value`, have been defined in a new package `java.util.function`. It includes interfaces with type values being standard Java types (like Double, Integer, Boolean etc), as well as generic ones

- `Consumer<T>` — represents a function object that accepts a single input argument and returns no result, `T -> void`, *eg*, prints the object of `T`; side-effects are expected (method's name `accept`)
- `Function<T,R>` — a function object `T -> R` (used in maps, method's name `apply`);
- `Predicate<T>` — a function object `T -> Boolean` (used in filters, method's name `test`);
- `Supplier<T>` — a function object `() -> T`, can be used to create instances of `T` on demand, object-equivalent to `T::new` (method's name `get`)

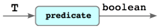
All of them can be used as assignment targets for a lambda expression or method reference.

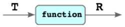
```
Predicate<Account> accSelector = a -> a.isCheckAccount();
Consumer<Account> accProcessor = a -> a.processChecks();
for (Account a: accounts) {
   if (accSelector.test(a)) accProcessor.accept(a);
}
accSelector = a -> isTermDeposit();
accProcessor = a -> accrueInterests();
...
```

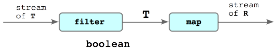
Streams: a sneak preview

This is good already, but with *streams* the processing functional interfaces can be put to use in even more nimble and elegant way.

The functional interfaces **have been predefined because** they are used as parameters of *stream operations* — methods which are called on *stream objects*. A great feature of stream processing is that operations can chained together (lazy computations):

- `stream.filter(predicate)` 

- `stream.map(transform)` 

- `stream.filter(predicate).map(transform)` 

Here, `predicate` and `transform` are instances of `java.util.function.Predicate` and `java.util.function.Function` interfaces.

```
acc.processChecks();
// if some checks were not cleared (due to insufficient funds)
double totalDishonoured = acc.outstandingChecks
   .stream() // converts a collection into a stream
   .reduce((a,b) -> a + b) // total unsettled amount wrapped in Optional
   .get(); // returns outstanding due amount out of Optional object
```

Method References

So much code is already defined as methods. Can it be (re-)used to avoid lambda-repetitions? Yes (with some restrictions)!

```
button.setOnAction(e -> System.out.println(e)); // instead can be written
button.setOnAction(System.out::println);         // like this
books.sort(Book::compareTitleLength); // assuming Book has compareTitleLength
```

`System.out::println` is a *method reference*: the `setOnAction` method supplies an argument (event), if the method reference signature matches — there is ambiguity how to use the argument and what to do.

Three cases

1. `object::instanceMethod` — equivalent to λ-expression which supplies the parameters
2. `Class::staticMethod` — ditto (eg, `Math::pow` is the same as `(x,y) -> Math.pow(x,y)`)
3. `Class::instanceMethod` — the first parameter is the *target of the method*: `String::compareToIgnoreCase` is equivalent to `(x,y) -> x.compareToIgnoreCase(y)`

Constructor References

`String s -> new Label(s)` is the same as `Label::new`. Works with arrays, too: `int[]::new` can be passed an integer `n` (an array of length `n` will be returned).

Default Methods

To perform an operation on a collection is very frequently occurring task. The functional approach, when a function (code) which performs the operation is passed to a traversing method invoked on the collection could be quite elegant (shorter) solution compared to a traditional one:

```
for (int i = 0; i < list.size(); i++)
    System.out.println(list.get(i));
```

But

```
list.forEach(System.out::println);
```

cannot be used until `forEach` is added to the JCF, yet this would break backward compatibility (all existing code which implements `Collection` will be broken — “no can do”).

The trick is to **add a concrete method** with such name **to** the Collection **interface** (give the same treatment to other library interfaces). Sounds impossible (ridiculous)? Maybe, but it is not wrong, and it is done! Default methods are marked by qualifier... (surprise!) `default`.

```
interface Person {
    long getId(); // usual abstract method
    default String getName() {
        return "John Doe, NSA analyst";
    }
}
```

Default Methods: “Grand Piano in the bushes”

Revisit the *Book* class, which (imagine!) has been augmented with the method:

```
public int titleLength() { return this.title.length(); }
```

but not with `compareTitleLength()` (used above as a method reference in sorting). Can we *easily* sort the list against the title length? We cannot use `titleLength` for the method reference in `sort(Book::titleLength)` because `sort` needs a *Comparator* object. *Comparator* is a functional interface of the type `(T,T) -> int`. Does *Comparator* have a default method which returns an object of this type?

```
% javap java.util.Comparator | grep compar
public abstract int compare(T, T);
... ...
public static <T> java.util.Comparator<T>
         java.util.function.comparingInt(ToIntFunction<? super T>);
... ...
```

`ToIntFunction` is a functional interface of the type we need — our task is easily accomplished:

```
books.sort(comparingInt(Book::titleLength));
```

Note: a more general default method `comparing` which returns *Comparator* also exists and can be used. (A free hand in adding default methods?)

Default Methods in Functional Interfaces

Two FI (as example) have the default methods:

- `java.util.function.Predicate`
  1. `Predicate<T> and(java.util.function.Predicate<? super T>);`
  2. `Predicate<T> negate();`
  3. `Predicate<T> or(java.util.function.Predicate<? super T>);`
  4. `static <T> Predicate<T> isEqual(java.lang.Object);`
- `java.util.function.Function` (super and extends in arg type declaration are omitted)
  1. `Function<V, R> compose(java.util.function.Function<V, T>)`
  2. `Function<T, V> andThen(java.util.function.Function<V, T>)`
  3. `static <T> Function<T, T> identity()`

What is their purpose? Notice, that they return the same type of FI in which they are defined. They are used for composing two or more lambdas (for function composition, see an example PseudoRandomGenerator.java)

- composition functions: $(f \circ g)(x) \equiv f(g(x)) = x \xrightarrow{g} y \xrightarrow{f} z$
- composing predicates:

```
inventory.sort(comparing(Book::price)             // sorting by price
               .reversed()                        // from dear to cheap
               .thenComparing(Book::getCountry)) // and then by country
```

Inheritance with Default Methods

Interfaces with implementation change the formerly simple business of single-only inheritance.

```
public interface A {
    default void hello() { System.out.println("Hello from A"); }
}
public interface B extends A {
   default void hello() { System.out.println("Hello from B"); }
}
public class C implements B, A {
   public static void main(String... args) { new C().hello(); }
}
```

Three rules of resolution

1. Classes always win. A method declaration in the class or a superclass takes priority over any default method declaration.
2. Otherwise, sub-interfaces win: the method with the same signature in the most specific default-providing interface is selected. (If *B* extends *A*, *B* is more specific than *A*).
3. If the choice is still ambiguous, the class inheriting from multiple interfaces has to explicitly select which default method implementation to use by overriding it and calling the desired method explicitly.

The method dispatch now is more complex — it is both vertical and horizontal.

“DDDD”: Default Dreadful Diamond-of-Death (resolution?)

There is a new syntax feature which concerns the selection of implementation during a multiple implementation of default methods (I assumed the static import of `java.lang.System.out` to cut on the line length, and used (syntactically illegal) acronym `p.s.v` for `public static void`):

```java
interface A {
  default void foo () {
    out.println("I am from A");
  }
}

interface C extends A, B {
  @Override
  default void foo () {
    B.super.foo();   // case A
    out.println("Override from C");
    A.super.foo(); // case B
  }
}
```

```java
interface B {
  default void foo () {
    out.println("I am from B");
  }
}

class InhTest implements C {
   p.s.v. main(String... args) {
    new InheritanceTest().foo();
}
// output
I am from B
Override from C
I am from A
```

Notice the parent super-reference in *C*: `A.super.foo()` — something we haven't seen before because there was no multiple inheritance by implementation in Java. We can even have *both* super's in overriding `C.foo()`!

Default Methods: not quite new feature!

Declare an interface (functional or not):

```
interface A { void sneeze(); }
```

Declare variables of type A and assign them to suitable λ-s:

```
A a = () -> System.out.printf("Usually I sneeze 2 times%n");
A b = () -> System.out.printf("Usually I sneeze 3 times%n");
```

Now, ask yourself: Is a an object? **Yes**. Is *every* object an instance of the class *Object*. **Yes**. Can we call equals, or `hashCode` on it? **Of course!**

```
jshell> a.toString()
$6 ==> "$Lambda$17/1401132667@2ac273d3"
...
jshell> a.equals(b)
$8 ==> false
jshell> a.hashCode()
$9 ==> 717386707
```

Why would it be wrong if we call own methods on a? To deny it would be "undemocratic". The morale — this was already happening before Java 8!

Default Methods: Caveat

[This consideration is due to Rafael Winterhalter Java 8 default methods can break your (users') code]

Default methods (also called *virtual extension* methods, or *defender* methods) are a smart invention, but care should be exercised when using them (for your own API).

Imaging the following — we wrote a class (call it *MyInput*) which inherited from *ThirdPartyClass* and implemented *SimpleInput* from some API. The default method `bar()` from that interface has been overridden with the super-reference to the original method. *Then*, the API authors extended *SimpleInput* into *ComplexInput*, with a would be better implementation of the default method `bar()`, and decided to make *ThirdPartyClass* implement *ComplexInput* and override `ComplexInput.bar()`. The client, *MyInput* has no change and still uses *SimpleInput* (hasn't change) and *ThirdPartyClass* (changed, *SimpleInput* now is its "grand parent interface"). The attempt to recompile `MyInput.java` will fail (the source code for this example is in two directories in "Examples" — before and after):

```
% javac MyInput.java
error: bad type qualifier SimpleInput in default super call
SimpleInput.super.bar(); // implementation from implemented interface
                      ^
method bar() is overridden in ThirdPartyClass
```

Yet, surprisingly, the JVM will not see anything wrong with the old *MyInput* bytecode (copy `MyInput.class` from "before" to "after", and run it there).

Closures

The feature used in *language holy wars*: "Does your language support closures?"

It's simple: a closure is code which *captures* variables from an enclosing scope. In case of λ's:

```
public static void repeatMessage(String text, int count) {
   Runnable r = () -> {
      for (int i = 0; i < count; i++) {
         System.out.println(text);
         Thread.yield();
      }
   };
   new Thread(r).start();
}
```

The variables from the enclosing method which are *not* lambda's parameters (so called *free variables*) `text` and `count` can be long gone after the `repeatMessage()` has returned (because the thread started by it may go for longer, it's independent), but the actual parameter values will be still in use.

Limitations: cannot alter *free variables* from the outer scope when the closure executes: they have to be *effectively final* — no `final` modifier is necessary but an attempt to change them will result in compile error (for "proto-closures" — inner classes — `final` was obligatory).

Mutating Closure

If a free variable is a reference of a mutable object, the closure *can* change (mutate) such object from the outer scope:

```
@FunctionalInterface
interface Converter<F, T> {
    T convert(F from);
}
int num = 1;
int[] values = new int[1]; values[0] = num;
System.out.println(values[0]);
Converter<Integer, String> stringConverter =
    (from) -> {values[0] *= 2; return String.valueOf(from + num);};
//num = num * 10; // this would be a compile error
System.out.println(values[0]);
```

An attempt to change `num` *within* the λ-expression, or even *outside* would cause a compile error. An example which tests these attempts is OuterScope.java.

This mutating trick may seem clever, but it suffers from performance problems and race conditions (in multi-threaded environment). There are better ways than utilise a captured variable (see **F5**).

Iterator, Iterable and λ-expression

Iterator allows an access to a container internals through methods `hasNext()` and `next()`; if `iter` is an iterator to a container `coll`, the latter can be traversed with the former:

```
for ( ; iter.hasNext(); ) {
   T t = iter.next();
   // do something with t
}
```

Iterable is an object whose class implemented this interface, its method `iterator()`; an iterable container `coll` can be traversed with a `"foreach"`-loop:

```
for (T t: coll)  {
   // do something with t
}
```

An iterable provides the iterator as a return value of its `iterator()` method. What about reversing this operation? λ-expression can do the trick:

`iterable` $\rightarrow$ `iterator`

```
Iterator<T> iter = coll.iterator();
```

To traverse with `"foreach"`-loop by having an iterator one has to “help” the compiler to set the target type of the λ-expression:

`iterator` $\rightarrow$ `iterable`

```
Iterable<T> iterable =
             () -> iterator;
for (String s :
  (Iterable<String>)() -> iterator){
  ...
}
```

Recursive λ-expression's

A λ-expression can contain reference to itself inside the λ-body, *ie* be *recursive*. A popular (corny) example is Fibonacci numbers:

```
public IntUnaryOperator fib = n -> (n < 2) ? n :
                        fib.applyAsInt(n-1) + fib.applyAsInt(n-2);
```

(The functional interface `java.util.function.IntUnaryOperator` declares an abstract method `applyAsInt(int x)` which returns `int`).

An attempt to have `fib` as a field initialised outside a constructor, or as a local variable inside `main` would fail (by the rules restricting forward references in initialisers for local or instance variables). However, by providing the lambda with an identity, *ie* by making it an object — via explicit constructor declaration and invocation — defining a recursive λ-expression becomes possible. Two ways to make it an object:

1. RecursiveLambda.java — to define a constructor which initialises `fib`, or
2. StaticRecursiveLambda.java — to declare `static fib` and initialise it with a static block.

In general (according to *Java Specification*), the issue whether a λ-expression is an object (has an identity) or not is undetermined (it is, of course, is defined in platform implementations and inside API, but it is not exposed to API clients).

Identity, state and behaviour

Semantic Difference between λ-expressions and inner classes

- λ-declarations do not introduce a naming scope, the keyword `this` (and `super`) has the same meaning as in the enclosing environment — the enclosing object (and its superclass object). The following program prints “Hello, world!” twice (`this` is a self-reference of an object with already overridden `toString()` method).

```
public class Hello {
   Runnable r1 = () -> System.out.println(this);
   Runnable r2 = () -> System.out.println(toString());
   public String toString() { return "Hello, world!"; }
   public static void main(String... args) {
       new Hello().r1.run();
       new Hello().r2.run();
   }
}
```

- An inner class does create a naming scope: The *would be same* version with anonymous inner classes with which `r1` and `r2` are instantiated, executes a non-overridden version of `Object.toString()` (since `r1` and `r2` extend *Object* directly, and `this` and `toString()` are “theirs”, not the outer class’s). Think this over! — Hello.java and OldHello.java.

Of three aspects of an object — identity, state and behaviour — λ-s represent only behaviour (“code”), their state is subsumed into the outer object; since they have no (own) state, lambdas do not need identity.

Exceptions in lambdas

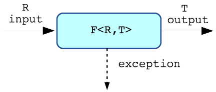
What if the body of a lambda-expression throws an exception? Nothing special — deal with it (by catch-ing) in a usual manner (from the Assignment One sample program JacksonSamplerStream.java):

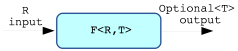
```
Stream<String> lines = Files.lines(Paths.get(args[0]),
            StandardCharsets.UTF_8);
    lines.forEach(s -> {
        try {
            tweet.putAll(mapper.readValue(s, Map.class));
            tweet.forEach((k,v) ->
                    System.out.printf("%s: %s%n", k, v));
            System.out.println("=================");
        tweet.clear();
        } catch (IOException ioe) {
            System.out.printf("Bad json record %s%n", ioe);
        }
    });
```

Catching and not re-throwing but returning the same type as the try-branch is one possible way. The result can also be wrapped inside an *Optional*. Lambda expression are not allowed to throw exceptions (and they need not to).

Exceptions vs Optional

λ-expressions represent a feature of functional programming style, in which methods are regarded as faithful implementation of ideal mathematical functions: they take an input (the argument) and they return a value *without any other effect*. Yet, mathematical functions often do not produce a value (division by zero, square root of a negative number...) — one can argue that mathematics always tries to resolve this by extending the set of values such that these "non-return" cases get eliminated. In programming, there are two possibilities:





1. Make methods to throw exceptions when the return value cannot be correctly computed

For functional programming, the presence of the second (exception) channel is very foreign. It is desirable to have only one output which *always* returns.

2. Box the return value, and leave it to the caller to deal with possibility that the box may not contain the value at all — this is an approach when the method is defined to return an *Optional* ("the box") object. We slight touched upon this in P5 lecture. The functional style is better suited for the second approach. That's why most of functional interfaces in the standard API do not throw exceptions. That's why *some* stream methods (`reduce` *etc*) return `java.util.Optional<T>` instead of `T`.

Recap on Interfaces

`interface` declaration was introduced in Java to retain multiple inheritance by contract, but to avoid (perhaps, misperceived) complexity of multiple inheritance by implementation.

At the same time, `interface` represents most flexible *type definition* which is the key to exercising the "program to interface" principle of OOP. The possibility to define an object type by an interface, which does not have any data associated with it, indicates that the essence of object-oriented programming is **not** state representing data of an object, but object's behaviour defined by methods declared in its interface.

A useful application of `interface`'s is to have them as *mixin*'s — types which a class can implement additionally to its "main" type. Your class is defined by whatever you include in its declaration *and* it also is *comparable* (via implementing *Comparable*), or *cloneable*, or *serialisable*, or *runnable*, or all those things together (if this makes sense, sometimes it doesn't). Extending a class is specialisation, implementing (mixing-in) an interface is composition.

Using the words of Brian Goetz (in his answer to StackOverflow in Feb 2015):

> *...from the perspective of the client of an interface, default methods should be indistinguishable from "regular" interface methods. The default-ness of a method, therefore, is only interesting to the designer and implementor of the interface.*

Interfaces and Traits

The Java SE 8 language rules enhancements have changed the nature of `interface` construct. An interface with default methods (which may contain no abstract methods at all) is what now can be called a *trait* — a unit in a programming language type system, which may be *stateful* (have fields, like in *Scala*), or stateless (like in *Java*; *Rust* has an interesting hybrid model).

What is trait?

More precisely, a trait is a collection of methods that implement behaviour to a class, defined from the component traits; the trait mechanism requires that the thus defined class implements a set of methods that parameterise the behaviour. The key point here (which makes the crucial distinction from a plain inheritance-based class definition model like Java's) is algebraic trait composition rules in a class definition. The traits featured in a class definition can be:

- combined — a *symmetric sum* rule
- overridden — an *asymmetric sum* rule
- expanded — an *alias* rule
- excluded — an *exclusion* rule

These rules can be implemented at the syntax level as value type operators, similar to set theory operations or Boolean Calculus. For example, the symmetric sum can be represented by the operator `&`. Java can emulate all but the exclusion (removing a method from inheritance; a child class can always access a method defined in an implemented interface/trait). An example of such emulation can be found in Oracle's Java Tutorial Default Methods.

Where to look for this topic in the textbook?

- Hortsmann's Core Java for the Impatient, Ch. 3.4–3.6
- Oracle's Java Tutorial Lambda Expressions
- Oracle's Java Tutorial List Interfaces
- Oracle's Java Tutorial When to Use Nested Classes, Local Classes, Anonymous Classes, and Lambda Expressions
- Oracle's Java Tutorial Default Methods
- Maurice Naftalin's Lambda FAQ has sections "Fundamentals" and "Advanced Questions", which deal with some subtle issues of lambdas, closures and default methods (very useful)
- Traits are an alternative attempt at realising the OO programming paradigm in which problems (related to the standard inheritance) of duplicated features, bloated and inappropriate hierarchies and member conflicts do not arise. The seminal paper on this subject is Traits: composable units of behavior.