11 Lambda Expressions and Streams

Instructor: Ke Wei（柯韋）

A319 ✆ Ext. 6452 ✉ wke@ipm.edu.mo

http://brouwer.ipm.edu.mo/COMP212/19/

Bachelor of Science in Computing, School of Applied Sciences, Macao Polytechnic Institute

November 19(21), 2019

Outline

Callback Functions and Event Handlers

- A callback function is a function that is passed as an argument to another function, to be "called back" at a later time. For example, when calling a general sorting method, you may want to specify a callback function to compare two objects.

  $$\texttt{static}\ \langle T\rangle\ \texttt{void}\ \mathit{sort}(T\texttt{[]}\ a\texttt{,}\ \mathit{callback\_function\_to\_compare\_two\_objects}\ f)$$

- In a typical GUI program, the GUI framework is responsible for the interaction between graphical components, such as buttons and menu items, and the user. When a command is requested, the GUI framework need a callback function to handle this event.

  $$\texttt{void}\ \mathit{addActionListener}(\mathit{callback\_function\_to\_handle\_an\_event}\ h)$$

  Such a callback function is often called an event handler.

Functional Interfaces

- The closest thing in Java to a callback function is a method that belongs to an object. By passing and returning this object, the method can be carried with.
- For a standalone callback function, the minimal type of such a carrying object is an interface with only one abstract method.

```
interface Comparator<T> {
    int compare(T x, T y);
}

interface ActionListener {
    void actionPerformed(ActionEvent e);
}
```

- An interface with only one abstract method is thus called a *functional interface*.

Anonymous Interface Implementation

- Before Java 8, a callback function wrapped in a functional interface is often carried by an object of an anonymous class.

```
sort(a, new Comparator<Student>() {
    @Override int compare(Student x, Student y) {
        return Double.compare(y.mark, x.mark);
    }
});
```

  This sorts a array of students from higher marks to lower marks.
- An anonymous class can do all that we need to wrap a callback function.
- However, the syntax is more like a class definition than an expression.
- Given the parameter type *Comparator*⟨*T*⟩ and *a* as an array of students, the *Comparator*⟨*Student*⟩ can be inferred.
- For a functional interface having only one method, mentioning the method signature of which to override is completely not necessary.

Lambda Expressions

- With Java 8, a functional interface can be implemented by a lambda expression, which is a stateless object of an anonymous class.

  ```
  sort(a, (x, y) -> Double.compare(y.mark, x.mark));

  exitButton.addActionListener(e -> { frame.dispose(); });
  ```

- A lambda expression has the following form,

  (parameter list) `->` expression or `{` statements `}`

  A lambda expression defines an anonymous method.
- The name of the method and the types of the parameters and return value are inferred from the functional interface which the lambda expression implements.
- More important, usually, a callback function does not need to have a state.
- The "lambda" is after Alonzo Church's λ-calculus, where λ is the symbol to introduce anonymous functions. For example, $\lambda x.x^2$.

Lambda Parameters

- Lambda expression can take parameters just like methods. The parameters must match the method signature of the functional interface the lambda expression implements.
- Often the types of the parameters can be inferred from the context, thus can be omitted.

```
Comparator<String> sc =
        (x, y) -> x.toUpperCase().compareTo(y.toUpperCase());
```

- If the lambda expression is matching a method that takes no parameter, then you can write the lambda expression like this,

```
Runnable hello = () -> System.out.println("Hello world!");
```

- When a lambda expression takes a single parameter, you can also omit the parentheses, like this,

```
ActionListener echo =
        e -> System.out.println("Command: "+e.getActionCommand());
```

Lambda Bodies

- The body of a lambda expression, and thus the body of the method it represents, is specified to the right of the ->.
- If the lambda body consists of multiple lines, you can enclose the lines inside { }.

```
Comparator<String> sc = (x, y) -> {
        String u = x.toUpperCase(), v = y.toUpperCase();
        return u.compareTo(v);
}
```

- If the lambda body contains only a single `return` statement, you can only write the expression to return as the lambda body.

```
map(a, x -> x*x)
```

- If a lambda expression only calls another method on exactly the lambda parameter list, the lambda expression can be simplified as a method reference.

`s -> System.out.println(s)` becomes `System.out::println`

Functional Programming

- Computers typically implement the Von Neumann architecture, which is a widely-used computer architecture based on a 1945 description by the mathematician and physicist John von Neumann.
- This architecture is biased toward *imperative programming*, which is a programming paradigm that uses statements to change a program's state. C, C++, and Java are all imperative programming languages.
- In *functional programming*, computations are codified as *functions*. These are *mathematical function*-like constructs that are evaluated in *expression contexts*. Such a function produces an output depending only on its arguments.
- Functional programming languages are declarative, meaning that a computation's logic is expressed without describing its control flow.
- Functional programming originated in λ-calculus, which was introduced by Alonzo Church. Another origin is combinatory logic, which was introduced by Moses Schönfinkel and subsequently developed by Haskell Curry.

Key Concepts of Functional Programming

- Functions are *first class objects*. Functions can be referenced by variables, passed to other functions as arguments, and returned from other functions as results.
- There are *pure functions*. A function is pure if the execution of the function has no side effects, and the return value of the function depends only on the input parameters passed to the function.
- There are *higher order functions*. A function is higher order if at least one of the function takes one or more functions as parameters, and/or the function returns another function as result.

```
static <T> ArrayList<T> filter(List<T> s, Predicate<T> p) {
    ArrayList<T> r = new ArrayList<T>();
    for ( T x : s )
        if ( p.test(x) ) r.add(x);
    return r;
}
```

Higher Order Function — *forEach*

- The Java *Consumer* interface is defined below.

```
interface Consumer<T> { void accept(T x); }
```

- The *forEach* method apply the *accept* method on each of the elements in an *Iterable*.

```
static <T> void forEach(Iterable<T> s, Consumer<? super T> c) {
    for ( T x : s )
        c.accept(x);
}
```

- This is the functional style `for` loop. For example, to print all the elements, we write

```
forEach(s, System.out::println);
```

- In fact, *forEach* is a default method of *Iterable*, we may also write

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

Covariance and Contravariance

- Parameterized types are *invariant*. In other words, for any two distinct types S and T, $Node\langle S\rangle$ is neither a subclass nor a superclass of $Node\langle T\rangle$.
- It seems intuitive that $Node\langle Circle\rangle$ is a subclass of $Node\langle Shape\rangle$.

```
void getFromShapeNode(Node<Shape> n) {
    Shape s = n.getElem(); ...
}
```

- However there are also cases that $Node\langle Shape\rangle$ is a subclass of $Node\langle Circle\rangle$.

```
void setToCircleNode(Node<Circle> n) {
    Circle c; ... n.setElem(c); ...
}
```

- A **factory** that produces better **cars** is a better factory (covariance).
- A **driver** that drives worse **cars** is a better driver (contravariance).

Covariance and Contravariance

- Parameterized types are *invariant*. In other words, for any two distinct types S and T, $Node\langle S\rangle$ is neither a subclass nor a superclass of $Node\langle T\rangle$.
- It seems intuitive that $Node\langle Circle\rangle$ is a subclass of $Node\langle Shape\rangle$.

```
void getFromShapeNode(Node<Circle> n) {
    Shape s = n.getElem(); ...
}
```

- However there are also cases that $Node\langle Shape\rangle$ is a subclass of $Node\langle Circle\rangle$.

```
void setToCircleNode(Node<Circle> n) {
    Circle c; ... n.setElem(c); ...
}
```

- A **factory** that produces better **cars** is a better factory (covariance).
- A **driver** that drives worse **cars** is a better driver (contravariance).

Covariance and Contravariance

- Parameterized types are *invariant*. In other words, for any two distinct types S and T, $Node\langle S\rangle$ is neither a subclass nor a superclass of $Node\langle T\rangle$.
- It seems intuitive that $Node\langle Circle\rangle$ is a subclass of $Node\langle Shape\rangle$.

```
void getFromShapeNode(Node<Circle> n) {
    Shape s = n.getElem(); ...
}
```

- However there are also cases that $Node\langle Shape\rangle$ is a subclass of $Node\langle Circle\rangle$.

```
void setToCircleNode(Node<Circle> n) {
    Circle c; ... n.setElem(c); ...
}
```

- A **factory** that produces better **cars** is a better factory (covariance).
- A **driver** that drives worse **cars** is a better driver (contravariance).

Covariance and Contravariance

- Parameterized types are *invariant*. In other words, for any two distinct types S and T, $Node\langle S\rangle$ is neither a subclass nor a superclass of $Node\langle T\rangle$.
- It seems intuitive that $Node\langle Circle\rangle$ is a subclass of $Node\langle Shape\rangle$.

```
void getFromShapeNode(Node<Circle> n) {
    Shape s = n.getElem(); ...
}
```

- However there are also cases that $Node\langle Shape\rangle$ is a subclass of $Node\langle Circle\rangle$.

```
void setToCircleNode(Node<Shape> n) {
    Circle c; ... n.setElem(c); ...
}
```

- A **factory** that produces better **cars** is a better factory (covariance).
- A **driver** that drives worse **cars** is a better driver (contravariance).

Bounded Wildcard Types

- Upper bound: *Node*⟨? `extends` *Shape*⟩ — *Node*⟨ any subclass of *Shape*⟩.

  ```
  Shape getFromNode(Node<? extends Shape> n) {return n.getElem();}
  ```

- Lower bound: *Node*⟨? `super` *Circle*⟩ — *Node*⟨ any superclass of *Circle*⟩.

  ```
  void setToNode(Node<? super Circle> n, Circle c) {n.setElem(c);}
  ```

- We can also use them together to write a read-use-write example:

```
void readUseWrite(Node<? super Rectangle> dest,
                  Node<? extends Rectangle> src) {
    Rectangle r = src.getElem();
    System.out.println(r.getWidth()+","+r.getHeight());
    dest.setElem(r);
}
```

Higher Order Function — *map*

- The Java *Function* interface is defined below.

```
interface Function<T, R> { R apply(T x); }
```

- The *map* method apply the *apply* method on each of the elements in an *Iterable*.

```
static <T, R>
ArrayList<R> map(Iterable<T> s, Function<? super T, ? extends R> m) {
    ArrayList<T> r = new ArrayList<T>();
    for ( T x : s )
        r.add(m.apply(x));
    return r;
}
```

- For example, to get a list of strings from the elements, we write

```
List<String> ls = map(s, x -> x.toString());
```

Higher Order Function — *reduce*

- The Java *BiFunction* interface is defined below.

```
interface BiFunction<S, T, R> { R apply(S x, T y); }
```

- The *reduce* method apply the *apply* method on each of the elements in an *Iterable* to accumulate them into an identity element.

```
static <T, R>
R reduce(Iterable<T> s, R id, BiFunction<R, ? super T, R> acc) {
    R r = id;
    for ( T x : s ) r = acc.apply(r, x);
    return r;
}
```

- For example, to get a list of strings from the elements, we write

```
List<String> ls = reduce(s, new ArrayList<String>(),
                         (l, x) -> {l.add(x.toString()); return l;});
```

Java *Stream* API

- The Java Stream API provides a more functional programming approach to iterating and processing elements of a collection.
- You obtain a stream from a collection by calling the *stream*() method of the given collection.

```
List<String> items = new ArrayList<String>();
items.add("one"); items.add("two"); items.add("three");
Stream<String> stream = items.stream();
```

- Once you have obtained a *Stream* instance from a *Collection* instance, you use that stream to process the elements in the collection.
- Processing the elements happens in two phases: *configuration* and *processing*.
- First the stream is configured. The configuration can consist of filters and mappings. These are also referred to as *non-terminal* operations.
- Second, the stream is processed. The processing consists of doing something to the filtered and mapped objects.

Java *Stream* API — Common Functions

- You filter a stream using the *filter*() method.

```
stream.filter( item -> item.startsWith("o") );
```

- You can map the items in a collection to other objects using the *map*() method.

```
stream.map( item -> item.toUpperCase() )
```

- When the *collect*() method is invoked, the filtering and mapping will take place and the object resulting from those actions will be collected.

```
List<String> filtered = stream
        .filter( item -> item.startsWith("o") )
        .collect(Collectors.toList());
```

- The *count*() method simply returns the number of elements in the stream after filtering has been applied.

```
long count = stream.filter( item -> item.startsWith("t")).count();
```

Java *Stream* API — *reduce*

- The *reduce*() method can reduce the elements of a stream to a single value.

```
String reduced = stream.reduce((acc, item) -> acc + " " + item).get();
```

- This *reduce*() method takes a BinaryOperator as parameter and returns an *Optional* In case the stream contains no elements, the *Optional*.*get*() returns `null`.
- There is another *reduce*() method which takes two parameters. It takes an initial value for the accumulated value, and then a *BinaryOperator*.

```
String reduced2 = stream.reduce("", (acc, item) -> acc + " " + item);
```

- A stream can be reduced in parallel internally into several accumulators and then combined together for the final result.
- If combining two accumulations is different from accumulating a single element, we need to specify another binary operator for the combiner.

```
String reduced3 = stream.reduce("", (acc, item) -> acc + " [" + item +"]",
                                (acc1, acc2) -> acc1 + " " + acc2);
```