CSSE2002/7023

Programming in the Large

Week 11.1: Lambda Expressions

In this Session

- Lambda Expressions
- Functional Interfaces
- Method References
- Streams

Lambda Expressions

Lambda expressions are anonymous methods

- don't have a name
- don't belong to a class

Input parameters (if they exist) are specified on the left side of the lambda operator ($->$) and the functionality on the right side

parameter(s) $->$ *expressionbody*

See: `LambdaExamples.java`

Methods vs. Lambda Expressions

A method has:

- Name
- Parameter List
- Body
- Return Type

A lambda expression has:

- No Name
- Parameter List
- Body
- No Return Type

Lambda Expressions – Important Points

Define inline implementation of a functional interface

- i.e. interface with a single method

Reduce the need for anonymous classes

- many common uses in Java were to implement functional interfaces

See: `ButtonDemo8.java`

Functional Interfaces

Interface defining a single abstract method

- a function

May contain default methods and static methods[1]

- or overridden methods from Object

Because there is only one abstract method, its name can be omitted when it's implemented

- anonymous class expression
- lambda expression

Allows functionality (method logic) to be passed as data to other methods

- functionality can be used in the method

Revisit: `LambdaExamples.java`

[1] https://docs.oracle.com/javase/tutorial/java/IandI/defaultmethods.html

Lambda Expressions – For Each Loop

Logic of a foreach loop can be replaced by a lambda expression

- passed to `forEach` method of the `Iterable`[2]
  - applies function to each element of the collection

See: `LambdaIterate.java`

[2] e.g. `List`

Method Reference

Used to refer to a method of a functional interface

- easy form of lambda expression

May be a

- reference to a static method
- reference to an instance method
- reference to a constructor

See: `MethodReferenceExample.java` and
`MethodRefConstructor.java`

Standard Functional Interfaces[3]

`Consumer<T>` :: `void accept(T t)`

`Function<T, R>` :: `R apply(T t)`

`Predicate<T>` :: `boolean test(T t)`

`Supplier<T>` :: `T get()`

`UnaryOperator<T>` :: `T apply(T t)`

Binary versions of functional interfaces

- e.g. `BiFunction<T, U, R>` – two parameters and return

Primitive specialisations of functional interfaces

- e.g. `IntPredicate` – for efficiency – no autoboxing

`java.util.Comparator<T>` :: `int compare(T o1, To2)`

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

Streams[4]

Sequence of elements from a source

- e.g. Collections, Arrays, I/O Resources, ...
  - stream doesn't hold any data
  - stream doesn't modify source

Aggregate operations

- functions that make use of the stream contents

Pipelining

- intermediate operations return the stream
  - operations can be joined together

Automatic iteration

- iteration performed internally over the source
  - can process streams from sources that won't fit in memory
  - enables lazy invokation
    - intermediate operations only called when needed

[4] https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/stream/package-summary.html

Stream Operations

Most accept parameters for user defined behaviour

- functional interface

Intermediate

- process elements in stream
- return a stream – allow pipelining
- e.g. `map`, `filter`, `sorted`, ...

Terminal

- last operation on a stream
- returns a result
- e.g. `forEach`, `collect`, `match`, `reduce`, ...

See: `PredicateExample.java`

Streams – Important Points

Pipeline may execute sequentially or in parallel

- property of the stream
  - can be modified at run-time

Order of intermediate operation is important

- operations that reduce the size of the stream should occur before operations that are applied to each element
  - e.g. `filter` before `map`

```
List<X> result = list.stream()
                     .map(...)
                     .filter(...)
                     .collect(Collectors.toList());
```

Applies the map function to **all** elements in the list

Then filters out the unneeded elements and returns a list of them

- do `filter` before `map`

Further Reading

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

- Simple example – looks at multiple ways to solve a problem and the benefit of using lambda expressions.

https://www.javaworld.com/article/3452018/get-started-with-lambda-expressions.html

- Overview of features and techniques of using lambda expressions in Java.

https://www.javaworld.com/article/3453296/get-started-with-method-references-in-java.html

- Overview of features and techniques of method and constructor references in Java.