Lambda expressions and streams

Outline

- Lambda expressions
  - The lambda calculus and lambda expressions
  - Functional programming
  - Java 8 lambdas

- Streams
  - Obtaining streams
  - Stream operations, structure, lifecycle
  - Parallel programming with streams

Outline

- Lambda expressions
  - The lambda calculus and lambda expressions
  - Functional programming
  - Java 8 lambdas

- Streams
  - Obtaining streams
  - Stream operations, structure, lifecycle
  - Parallel programming with streams

The lambda calculus

- The lambda calculus was introduced in the 1930s by Alonzo Church as a mathematical system for defining computable functions
- The lambda calculus is equivalent in definitional power to that of Turing machines
- The lambda calculus serves as the computational model underlying functional programming languages such as Lisp, Haskell, and Ocaml
- Features from the lambda calculus such as lambda expressions have been incorporated into many widely used programming languages like C++, C# and (finally) Java 8

Example of a lambda expression

- The lambda expression

  $$\lambda x . (+ x 1) 2$$

  represents the application of a function λ x . (+ x 1) with a formal parameter x and a body + x 1 to the argument 2. Notice that the function definition λ x . (+ x 1) has no name; it is an *anonymous function*

- In Java 8, we would represent this function definition by the Java 8 lambda expression `x -> x + 1`

Lambdas in Java 8

- A Java 8 lambda is basically a method in Java without a declaration, usually written as (parameters) -> { body }. Examples:
  1. `(int x, int y) -> { return x + y; }`
  2. `x -> x * x`
  3. `() -> x`
- A lambda can have zero or more parameters separated by commas and their type can be explicitly declared or inferred from the context
- Parenthesis are not needed around a single parameter
- `()` is used to denote zero parameters
- The body can contain zero or more statements
- Braces are not needed around a single-statement body

Outline

- Lambda expressions
  - The lambda calculus and lambda expressions
  - Functional programming
  - Java 8 lambdas

- Streams
  - Obtaining streams
  - Stream operations, structure, lifecycle
  - Parallel programming with streams

Functional programming

- A style of programming that treats computation as the evaluation of mathematical functions
- Eliminates side effects
- Treats data as being immutable
- Expressions have referential transparency
- Functions can take functions as arguments and return functions as results
- Prefers recursion over explicit loops

Why do functional programming?

- Write easier-to-understand, more declarative, more concise programs than imperative programming
- Focus on the problem rather than the code
- Get rid of boilerplate / syntactic sugar
- Parallel programming is easier
- Write cleaner APIs

Functional programming and Java 8

- Java 8 is the biggest change to Java since the inception of the language (or, perhaps, since Java 5)
- Lambdas are the most important new addition
- Java was playing catch-up: most major programming languages already had support for lambda expressions
- A big challenge was to introduce lambdas without requiring recompilation of existing binaries

Outline

- Lambda expressions
  - The lambda calculus and lambda expressions
  - Functional programming
  - Java 8 lambdas

- Streams
  - Obtaining streams
  - Stream operations, structure, lifecycle
  - Parallel programming with streams

Java 8 lambdas

- Syntax of Java 8 lambda expressions
- Functional interfaces
- Variable capture
- Method references
- Default methods

Example 1:
Print a list of integers with a lambda

```
List<Integer> intList = Arrays.asList(1,2,3);

intList.forEach(x -> System.out.println(x));
```

- `x -> System.out.println(x)` is a lambda expression that defines an anonymous function with one parameter named `x` of type `Integer`
  - The compiler infers the type of `x` from the context

Example 2:
A multiline lambda

```
List<Integer> intList = Arrays.asList(1,2,3);

intList.forEach(x -> {
   x += 2;
   System.out.println(x);
});
```

- Braces are needed to enclose a multiline body in a lambda expression

Example 3:
A lambda with a defined local variable

```
List<Integer> intList = Arrays.asList(1,2,3);

intList.forEach(x -> {
    int y = x * 2;
    System.out.println(y);
});
```

- Just as with ordinary methods, you can define local variables inside the body of a lambda expression

Example 4:
A lambda with a declared parameter type

```
List<Integer> intList = Arrays.asList(1,2,3);

intList.forEach((Integer x) -> {
    x += 2;
    System.out.println(x);
});
```

- You can, if you wish, specify the parameter type

Implementation of Java 8 lambdas

- The Java 8 compiler first converts a lambda expression into a method
- The generated method is called when needed
- For example, `x -> System.out.println(x)` could be converted into a generated static method

```
public static void genName(Integer x) {
    System.out.println(x);
}
```

- But what type should be generated for this method? How should it be called? What class should it go in?

Functional interfaces

- Design decision: Java 8 lambdas are assigned to functional interfaces
- A functional interface is a Java interface with exactly one non-default method, for example:

```
public interface StringTrans {
    String doIt(String s);
}
```

- Method name is not important, only signatures of lambda expression and method need to match
  - signature = parameter number and types, return type, exceptions thrown, but **<u>not</u>** method name
- Optional annotation `@FunctionalInterface` emphasizes that the interface will be used this way

Properties of the generated method

- The method generated from a Java 8 lambda expression has the same signature as the method in the functional interface
- The type is the same as that of the functional interface to which the lambda expression is assigned
- The lambda expression body becomes the body of the method in the interface

Available functional interfaces

The package *java.util.function* defines many new useful functional interfaces, for example:

- `Predicate<T>`
  - Represents a predicate (boolean-valued function) of one argument
  - Functional method is `boolean test(T t)`
    - Evaluates this Predicate on the given input argument (`T t`)
    - Returns `true` if the input argument matches the predicate, otherwise `false`

Available functional interfaces

- `Supplier<T>`
  - Represents a supplier of results
  - Functional method is `T get()`
    - Returns a result of type `T`

- `Consumer<T>`
  - Represents an operation that accepts a single input and returns no result
  - Functional method is `void accept(T t)`
    - Performs this operation on the given argument (`T t`)

Available functional interfaces

- `Function<T,R>`
  - Represents a function that accepts one argument and produces a result
  - Functional method is `R apply(T t)`
    - Applies this function to the given argument (`T t`)
    - Returns the result of type `R`
- `UnaryOperator<T>`
  - Extends `Function`
  - Represents an operation on a single operand that produces a result of the same type as its operand
  - Functional method is `T apply(T t)`
    - Applies this function to the given argument (`T t`)
    - Returns the result of type `T`

Available functional interfaces

Single-method interfaces were a common idiom in Java even before Java 8, all of which can now be regarded as functional interfaces, for example:

- `Comparator<T>`
  - Compares its two arguments for order
  - Functional method is `int compare(T o1, T o2)`
    - Returns a negative integer, zero, or a positive integer as the first argument is less than, equal to, or greater than the second
- `Runnable`
  - Used for defining threads
  - Functional method is `void run()`

Assigning lambdas to variables

- A lambda expression can be assigned to a variable of a functional interface type if the signatures of the lambda expression and functional method are compatible
- The compiler will “fill in the blanks” if argument types in the lambda are not specified
- Example:

```
List<Integer> intList = Arrays.asList(1,2,3);
Consumer<Integer> p = x -> System.out.println(x);
intList.forEach(p);
```

Variable capture

- Lambdas can interact with variables defined outside the body of the lambda

- Using these variables is called variable capture

Local variable capture example

```
import java.util.*;

public class LVCExample {
  public static void main(String[] args) {
    List<Integer> intList = Arrays.asList(1,2,3);
    int var = 10;
    intList.forEach(x -> System.out.println(x + var));
    // var = 40; // error
  }
}
```

- Note: local variables (including method parameters) used inside the body of a lambda expression must be final or effectively final

Field capture example

```
import java.util.*;

class FCExample {
  private int f = 10;
  private static int sf = 20;
  void doIt() {
    List<Integer> intList = Arrays.asList(1,2,3);
    intList.forEach(x -> System.out.println(x + f + sf));
  }
  public static void main(String[] args) {
    FCExample x = new FCExample();
    x.doIt();
  }
}
```

Method references

- Another new feature in Java 8
- Method references can be used to pass an existing function in places where a lambda (i.e. functional interface) is expected
- The signature of the referenced method needs to match the signature of the functional interface method

Summary of method references

Method Reference Type	Syntax	Example
static	ClassName::StaticMethodName	String::valueOf
constructor	ClassName::new	ArrayList::new
specific object instance	objectReference::MethodName	x::toString
arbitrary object of a given type	ClassName::InstanceMethodName	Object::toString

Conciseness with method references

We can rewrite the statement

```
intList.forEach(x -> System.out.println(x));
```

more concisely using a method reference:

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

Default methods

- With lambda expressions there came the need to considerably extend the standard library
  - For example, add `forEach` method to collections

- However, introducing new methods to interfaces would break existing code, since it must implement the new methods (but it could not have known about the new methods in advance!)

- This was the primary reason for introducing default methods in Java 8

Default methods

- For example, this would break existing code:

```
public interface Iterable<T> {
  public Iterator<T> iterator();
  public void forEach(Consumer<? super T> consumer);
}
```

- But, using a default method would not:

```
public interface Iterable<T> {
  public Iterator<T> iterator();
  public default void forEach(Consumer<? super T>
                                consumer) {
    for (T t : this) consumer.accept(t);
  }
}
```

Outline

- Lambda expressions
  - The lambda calculus and lambda expressions
  - Functional programming
  - Java 8 lambdas

- Streams
  - Obtaining streams
  - Stream operations, structure, lifecycle
  - Parallel programming with streams

Stream API

- The new *java.util.stream* package provides utilities to support functional-style operations on streams of values
  - Streams are a functional-style design pattern (i.e. monads)
  - Represented by Java interface `Stream<T>`, as well as `IntStream`, `LongStream`, `DoubleStream` for primitive types

- Streams can be obtained from practically any meaningful source in the Java API: collections, arrays, strings, files, ...

- Streams can be sequential or parallel

- Streams are useful for selecting values and performing actions on the results, especially when combining different selections and actions

Outline

- Lambda expressions
  - The lambda calculus and lambda expressions
  - Functional programming
  - Java 8 lambdas

- Streams
  - Obtaining streams
  - Stream operations, structure, lifecycle
  - Parallel programming with streams

Obtaining streams

- From individual values
  - **Stream.of(val1, val2, …):** stream of given values
- From ranges
  - **IntStream.range(n1, n2):** stream of ints from n1 to n2–1
- From an array
  - **Arrays.stream(someArray):** stream of array elements
- From a Collection
  - **someCollection.stream():** stream of collection elements
- From a String
  - **someString.chars():** stream of characters (as ints)
- From a file (NIO API)
  - **Files.line(path):** stream of file lines (as Strings)

Outline

- Lambda expressions
  - The lambda calculus and lambda expressions
  - Functional programming
  - Java 8 lambdas

- Streams
  - Obtaining streams
  - Stream operations, structure, lifecycle
  - Parallel programming with streams

Stream operations

- An **intermediate operation** keeps a stream open for further operations. Intermediate operations are "lazy"
- A **terminal operation** must be the final stated operation on a stream. Once a terminal operation is invoked, the stream is consumed and is no longer usable
- Notes: the terminal operation is stated last, but "starts executing" first – its execution triggers the execution of intermediate operations and input of elements from the source. Stream operations do not modify the source

Stream structure

A stream has three components:

1. A **source** such as a collection, an array, a generator function, etc.
2. A **pipeline** of zero or more intermediate operations; and
3. A **terminal operation**

Stream lifecycle

- **Creation**
  - Create stream from some source
- **Configuration**
  - Configure stream with a sequence (pipeline) of intermediate operations
- **Execution**
  - Execute stream by invoking the terminal operation
- **Cleanup**
  - Once executed, the stream is “consumed” and discarded; in order to execute again, a fresh stream needs to be made

Terminal operations

- **Reduction terminal operations:** return a single result
  - count, min, max, reduce
- **Mutable reduction terminal operations:** return multiple results in a container data structure
  - collect, toArray
- **Search terminal operations:** return a result as soon as a match is found
  - findFirst, findAny, anyMatch, allMatch
- **Generic terminal operation:** do any kind of processing you want on each stream element
  - forEach
- Nothing happens until the terminal operation is invoked!

Class Optional<T>

- Package *java.util*, new classes in Java 8
- **Optional<T>:** A container which may or may not contain a non-null value of type T
- **OptionalInt**, **OptionalLong**, **OptionalDouble**: primitive-type variants
- Common methods:
  - **isPresent()** – returns true if value is present
  - **get()** – returns value if present, otherwise throws NoSuchElementException
  - **orElse(T other)** – returns value if present, otherwise returns other
  - **ifPresent(Consumer)** – runs the consumer function on the value if it is present

Reduction terminal operations

```
List<Integer> integers = Arrays.asList(1, 2, 3, 4, 5);

Long count = integers.stream().count();
System.out.println(count); // 5

Optional<Integer> result;
result = integers.stream().min((x, y) -> x - y);
System.out.println(result.get()); // 1

result = integers.stream().max(Comparator.naturalOrder());
System.out.println(result.get()); // 5

Integer reduct = integers.stream().reduce(0, (x,y) -> x + y);
System.out.println(reduct); // 15
```

Mutable reduction terminal operations

```
Set<Integer> s = integers.stream().collect(Collectors.toSet());
System.out.println(s); // [1, 2, 3, 4, 5]

Integer[] a = integers.stream().toArray(Integer[]::new);
Arrays.stream(a).forEach(System.out::println);
```

- Class Collectors defines many useful collectors for passing as arguments to collect():
  - toList
  - toSet
  - toMap

Search terminal operations

```
List<Integer> integers = Arrays.asList(1, 2, 3, 4, 5, 5);

Optional<Integer> result = integers.stream().findFirst();
System.out.println(result.get()); // 1

result = integers.stream().findAny();
System.out.println(result.get()); // 1 (generally unpredictable)

boolean match = integers.stream().anyMatch(x -> x == 5);
System.out.println(match); // true

match = integers.stream().allMatch(x -> x > 3);
System.out.println(match); // false
```

Intermediate operations

- Intermediate operations can be **stateless** and **stateful**

- **Stateless intermediate operations:** do not need to know anything about results from previous steps in the pipeline
  - **filter:** excludes all elements that do not match a given Predicate
  - **map:** performs a one-to-one transformation of elements using a given Function
  - **mapToInt, mapToLong, mapToDouble:** map to a primitive stream using an appropriate Function
  - **peek:** execute a Consumer for each element, useful for debugging

Stateless intermediate operations

```
List<Integer> integers = Arrays.asList(1, 2, 3, 4, 5, 5);

integers.stream()
   .filter(x -> x < 4)
   .forEach(System.out::println); // 1 \n 2 \n 3

List<Integer> r = integers.stream()
   .peek(x -> System.out.println("Old value: " + x))
   .map(x -> x + 1)
   .peek(x -> System.out.println("New value: " + x))
   .collect(Collectors.toList());
r.forEach(System.out::println); // 2 \n 3 \n 4 \n 5 \n 6 \n 6

int sum = integers.stream()
   .mapToInt(x -> x)
   .sum(); // also average(), min(), max()
System.out.println(sum); // 20
```

Intermediate operations

- Intermediate operations can be **stateless** and **stateful**

- **Stateful intermediate operations:** need to know something about results from previous steps in the pipeline
  - **distinct:** removes duplicate elements from the stream
  - **limit:** leaves only the first *n* elements in the stream
  - **skip:** removes the first *n* elements from the stream
  - **sorted:** sorts the stream

Stateful intermediate operations

```
List<Integer> integers = Arrays.asList(1, 2, 3, 4, 5, 5);

integers.stream().distinct()
  .forEach(System.out::println); // 1 \n 2 \n 3 \n 4 \n 5

integers.stream().limit(3)
  .forEach(System.out::println); // 1 \n 2 \n 3

integers.stream().skip(3)
  .forEach(System.out::println); // 4 \n 5 \n 5

integers = Arrays.asList(7, 1, 2, 3, 4, 5, 5);
integers.stream().sorted()
  .forEach(System.out::println);
  // 1 \n 2 \n 3 \n 4 \n 5 \n 5 \n 7
```

Example: Using map and reduce

```
List<Integer> list = Arrays.asList(1,2,3);
int sum = list.stream()
              .map(x -> x*x)
              .reduce((x,y) -> x + y)
              .get();
System.out.println(sum);
```

- Here, `map(x -> x*x)` squares each element and then `reduce((x,y) -> x + y)` reduces all elements to a single number
- Such combinations of map and reduce are a common idiom for facilitating parallel and distributed programming

Outline

- Lambda expressions
  - The lambda calculus and lambda expressions
  - Functional programming
  - Java 8 lambdas

- Streams
  - Obtaining streams
  - Stream operations, structure, lifecycle
  - Parallel programming with streams

Parallel programming with streams

- Java streams can be executed in either sequential or parallel mode
- Up to now we worked with sequential streams
- A parallel stream can be obtained in several ways:
  - **someCollection.parallelStream()** (instead of **.stream()**)
  - **someStream.parallel()**
- Syntactically, everything else is the same – parallelism is achieved automatically!

References

- Alfred V. Aho, Design and Implementation of Lambda Expressions in Java 8: http://www.cs.columbia.edu/~aho/cs6998/Lectures/14-09-15.pptx
- Adib Saikali, Java 8 Lambda Expressions and Streams: www.youtube.com/watch?v=8pDm_kH4YKY
- Java Tutorials - Lambda Expressions: http://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html
- Viral Patel, Java 8 Lambda Expressions Tutorial with Examples: http://viralpatel.net/blogs/lambda-expressions-java-tutorial/
- Brian Goetz, Lambdas in Java: A peek under the hood: https://www.youtube.com/watch?v=MLksirK9nnE