7. Functional programming in Java 8

Slides:

- *22 Java 8 Lambdas*
- *23 Java Stream API*

Java 8

Java 8 is the biggest change to Java since its creation. The eighth version introduces functional programming through Lambda expressions.

Concurrently to lambdas, they releases also the Stream API, that helps in using lambdas without recompiling existing binaries.

Lambda expression

Benefits of Lambdas

- Enables functional programming in Java
- Allows to pass behaviors as well as data to functions
- Allows laziness in stream processing
- Facilitates parallel programming
- Provides more generic, reusable and flexible APIs

Examples:

- lambda: `list.forEach(x -> System.out.println(x));`
- method reference: `list.forEach(System.out::println);`
- block/multiline: `list.forEach(x -> { System.out.println(x); });`

Things to know:

All the variable used in a lambda must be final (by declaration) or effectively final.

Can also be static field of the class, but for use them we need to be in a static method (e.g. main).

Lambda implementation

The compiler converts a lambda expression into a function, compiling its code.
Then generates the code to call the compiled function where needed.
Lambdas are instances of Functional Interface, that is a Java interface with exactly one abstract method (the converted lambda).
For that reason, all the Functional Interfaces can be used as target of lambda expression. For example is possible to pass a method reference, that is a Functional Interface, instead of a lambda.
The compiler uses type inference based on the target type.
Arguments and result types of the lambda must match those of the abstract method of the Functional Interface.
Lambdas can be interpreted as instances of anonymous inner classes implementing the Functional Interface, and are invoked by calling the only abstract method.
As for the method references, lambdas can be assigned to variables: `Runnable runnable = () -> { System.out.println("Hello Lambda!"); };`

Examples of functional interfaces

```
// function with parameters without return value
public interface Consumer<T> {  // and BiConsumer
  void accept(T t);
}
// function with no parameters but with return value
public interface Supplier<T> {
  T get();
}
// function with both parameters and return value (of common type)
public interface Predicate<T> {
  boolean test(T t);
}
// function with both parameters and return value (of specified class)
public interface Function <T,R> {
  R apply(T t);
}
```

Method references

Method reference type	**Syntax**
Static method	ClassName::StaticMethodName
Constructor	ClassName::new
Method of an object instance	objectReference::MethodName

Backward compatibility

To maintain backward compatibility is not possible to add new abstract methods to an interface.
For that reason, Java 8 allows existent interfaces to include abstract methods, static methods and default methods defined in term of other abstract methods.
Hence, the backward compatibility for the Java Collections Framework is granted through lambda expressions and default methods.

Stream API

The `java.util.stream` package provides utilities to support functional-style operation on streams of values.

Stream vs Collection

A stream is not a data structure, hence doesn't store elements. It conveys elements from a source (data structure, generator function, I/O) through a pipeline of operations. It produces a result without modifying its source.

Stream properties

- Laziness-seeking: operations are divided into intermediate (stream-producing) and terminal (value-producing). All the intermediate function are lazy, and are evaluated only when a result is needed by the following operation.
- Unbounded: while collections are finite, streams are possibly not.
- Consumable: stream's elements are visited only once. Then another stream must be generated from the same source, as for the iterators.

Stream pipeline

Consists in:

- a source
- zero or more intermediate operations
- a terminal operation producing a non-stream value

Intermediate methods are not performed until the terminal method is called. At this point the stream is consumed and is not possible to execute other operations on that.

Short-circuit intermediate methods can cause the earlier intermediate methods to be processed until is not possible to process the short-circuit method.

An example of short-circuit method is a filter, that cause the procession of items until the current one is not a suitable output.

Intermediate operations

```
Stream<T> filter(Predicate<T> predicate)     // filter
IntStream mapToInt(ToIntFunction<T> mapper)  // map f:T -> int
<R> Stream<R> map(Function<T,R> mapper)      // map f:T->R
Stream<T> peek(Consumer<T> action)           // performs action on elements
without affective them (e.g. a debug print)
Stream<T> distinct()                         // remove duplicates, stateful
Stream<T> sorted()                           // sort elements, stateful
Stream<T> limit(long maxSize)                // truncate
Stream<T> skip(long n)                       // skips first n elements
```

Terminal operations

```
void forEach(Consumer<T> action)                         // for-iterator
Object[] toArray()                                       // accumulator
T reduce(T identity, BinaryOperator<T> accumulator)      // fold
Optional<T> reduce(BinaryOperator<T> accumulator)        // fold
R collect(Supplier<R> supplier,
    BiConsumer<R, T> accumulator, BiConsumer<R, R> combiner);  // collector
(accumulator)
Optional<T> min(Comparator<T> comparator)  // return the min (or max)
boolean allMatch(Predicate<T> predicate)   // check predicate (bool),
short-circuiting
boolean anyMatch(Predicate< T> predicate)  // check predicate
(bool),short-circuiting
Optional<T> findAny()                      // return a value, short-circuiting
```

Iterator and generators

Generates infinite streams.

```
// Iterator
static <T> Stream<T> iterate(T seed, UnaryOperator<T> f)

// Example: summing first 10 elements of an infinite stream
int sum = Stream.iterate(0,x -> x+1).limit(10).reduce(0,(x,s) -> x+s);


// Generator
static <T> Stream<T> generate(Supplier<T> s)

// Example: printing 10 random numbers
Stream.generate(Math::random).limit(10).forEach(System.out::println);
```

Parallel streams

Stream operations can be executed in serial (default) or in parallel using `.parallelStream()`.

The Runtime Support takes care of using multithreading in a transparent way.

Thread safety is guaranteed also on non-thread-safe collections, unless some operations have side-effects.

To process items in parallel the source must be efficiently splittable. Most of Collections are, but I/O is not.

Moreover, intermediate operations should be stateless and not affect the source (non-interfering behaviour).

For parallel streams with side-effects, ensuring thread safety is the programmers' responsibility.

Stream implementation

Java 8 Streams are implemented with the Spliterator, that is the parallel analogue of an Iterator.

As an iterator it has methods to sequentially advancing and to apply an action to the next item, but has also the ability of splitting off some portion of the input into another spliterator which can be processed in parallel.

Monads in Java

Monads in Java are implemented with Optional value and Streams.

An Optional value is an object that can or not have a value, e.g. `Optional<Integer>` can have an integer value.

The `isPresent()` method returns true if the object has a value. The `get()` method returns the value if it is present, otherwise throws `NoSuchElementException`.

In a similar way, is possible to obtain stream with one single element, or empty.

The `flatMap` method allows to map elements in a Stream on in an Optional with other elements, returning a new Stream or Optional.