Java 8

ANIL JOSEPH

Agenda

- Interfaces
- Lambda Expressions
- Streams
- Collectors
- Parallelism

Java 8

- More concise code.
- Extensibility.
- Simplified coding on multi core systems.

Interfaces

- Interfaces in Java 8 introduces 2 new types of methods apart from abstract methods
- Two new types
  - Default
  - Static

Interfaces: Default methods

- Default methods are methods in an interface with an implementation(can be overridden in the implementing class).
- Provides a mechanism to add new methods to existing interfaces
- It doesn't break backwards compatibility
- Gives Java multiple inheritance of behavior, as well as types
  - but not state!

Interfaces: Default methods

```
public interface InterfaceA {

    public void doSomething();

    default void buildTask(){
        System.out.println("InterfaceA buildTask");
    }

}
```

Interfaces: Default methods Conflict Resolution

- The Java compiler follows inheritance rules to resolve the name conflict.
- Instance methods are preferred over interface default methods.
- Methods that are already overridden by other candidates are ignored.
  - This circumstance can arise when supertypes share a common ancestor.
- If two or more independently defined default methods conflict, or a default method conflicts with an abstract method, then the Java compiler produces a compiler error.
  - You must explicitly override the supertype methods.

Interfaces: static methods

- Java 8 static methods in an interface
- This is linked to default(extension) methods in that interfaces can now include behavior.
- Static methods, by definition, are not abstract

Interfaces: static methods

```
public interface InterfaceB {

    void doWork();

    default void defaultTask(){

        System.out.println("InterfaceB defaultTask...");

    }

    static void buildTask(){
        System.out.println("InterfaceB static buildTask");
    }
}
```

Functional Interfaces

- Many interfaces in the Java library declare only one method.
- Examples are java.lang.Runnable, java.awt.event.ActionListener, java.util.Comparator etc
- These interfaces are also called Single Abstract Method interfaces (SAM Interfaces).
- ***In Java 8 the same concept of SAM interfaces is recreated and are called Functional interfaces.***
- ***Functional interfaces*** can be represented using Lambda expressions, Method reference and constructor references.
- ***@FunctionalInterface*** *annotation* can be used for compiler level errors.

Functional Interfaces

```
@FunctionalInterface
public interface Simple {

    void doSomething();
}
```

Lambda Expressions

- Lambda expressions represent anonymous functions
  - Like a method, has a typed argument list, a return type, a set of thrown exceptions, and a body
  - Not associated with a class
- Lambda expressions enable you to treat functionality as method argument, or code as data.
- Syntax
  - (parameters) -> {body}

Lambda Expressions

- **The Functional Interface**

```
@FunctionalInterface
public interface Simple {

    void doSomething();
}
```

- **The Lambda Expression.**

```
Simple simple = () -> System.out.println("Implemented using lambda");
simple.doSomething();
```

Lambda Expressions

- **The Functional Interface**

```
@FunctionalInterface
public interface Calculator {

    int calculate(int x, int y);

}
```

- **The Lambda Expression.**

```
Calculator calculator = (x, y) -> x + y;
System.out.println(calculator.calculate(10, 20));
calculator = (x, y) -> x - y;
System.out.println(calculator.calculate(10, 20));
calculator = (x, y) -> {

            System.out.println("Calculating");
            return x * y;

        };
System.out.println(calculator.calculate(10, 20));
```

Lambda Expressions

- Almost all machines now are multi-core, multi-processor, or both
- We need to make it simpler to write multi-threaded Java code
  - Java has always had the concept of threads
  - Even using the concurrency utilities and fork-join framework this is hard
- Lambda expressions and the streams API simplify the threading / parallelism code.

Method References

- Method references let us reuse a method as a lambda expression
- Four Types of references
- Reference to a static method:
  - **ContainingClass::staticMethodName**
- Reference to an instance method of a particular object:
  - **ContainingObject::instanceMethodName**
- Reference to an instance method of an arbitrary object of a particular type:
  - **ContainingType::methodName**
- Reference to a constructor:
  - **ClassName::new**

New Date and Time API

- Java 8 introduces a new set of classes to work with Date and Time
- The new classes are in the package java.time
- New Classes
  - LocalDate
  - LocalTime
  - LocalDateTime
  - Instant
  - Duration
  - ZonedDateTime

Stream API's

- A ***stream*** represents a sequence of elements supporting sequential and parallel aggregate operations.
- A stream is like a pipeline with three parts
  - A source
  - Zero or more intermediate operations
  - A terminal operation
    - Producing a result or a side-effect

```
int sum = transactions.stream().
  filter(t -> t.getBuyer().getCity().equals("London")).
  mapToInt(Transaction::getPrice).
  sum();
```

Stream API's Sources

- There are many ways to create a Stream.
- From collections and arrays
  - `Collection.stream()`
  - `Collection.parallelStream()`
  - `Arrays.stream(T array)` or `Stream.of()`
- Static factories
  - `IntStream.range()`
  - `Files.walk()`
- Roll your own
  - `java.util.Spliterator()`

Stream API's Intermediate Operations

- Intermediate Operations are executed lazily.
- The internal processing model of streams is designed in order to optimize the processing flow.
- Intermediate Operations available
  - Mapping
    - Mapping is a process of changing the form of the elements in a *stream*.
  - Filtering
    - Filtering is a process of selecting items depending upon some condition.
  - Unique Elements
    - A process to ensure there are a no duplicate elements. It's a type of filter
  - Skipping
    - A process to skip a number of elements. It's a type of filter
  - Sorting

Stream API's Terminal methods

- Invoking a terminal operation executes the pipeline
- All operations can execute sequentially or in parallel
- Terminal operations can take advantage of pipeline characteristics

Collectors

- The collect method the Stream of is a terminal operation, that allows to transform a stream into another type(possibly a list or set).
- The argument passed to the collect method is an instance of java.util.stream.Collector.
- The Collector essentially describes a recipe for accumulating the elements of a stream into a final result.
- Collector operations
  - Grouping
  - Partitioning

Optional

- Java 8 introduces a new class called java.util.Optional<T>.
- Used to indicate that reference may, or may not have a value.

Parallel Processing

- Java 8 Streams supports parallel processing.
- A collection can be turned into a parallel stream by invoking the method parallelStream on the collection source.
- A *parallel* stream is a stream that splits its elements into multiple chunks, processing each chunk with a different thread.
- It can automatically partition the workload of a given operation on all the cores of your multicore processor and keep all of them equally busy.
- Parallel streams use the threads for the ThreadPool
  - The default size is the equivalent to the number of processors.
  - System.setProperty("java.util.concurrent.ForkJoinPool.common.parallelism", "12");