New Features in Java 8

Alexander Gehrke

November 30, 2016

Übersicht - WIP

New Features of Interfaces

λ Lambdas λ

- Anonymous Functions and Method References
- java.util.stream
- java.util.Optional
- Parallelization with Streams

Other News

New Features of Interfaces

Static Methods

- Interfaces can now have static methods
- Can replace utility classes $\implies$ not instantiable
- Several new utility functions in the standard library, e.g. in Comparator:

```
static <T extends Comparable<? super T>>
↪  Comparator<T> naturalOrder()
```

Default Methods

New keyword `default` allows implementation of methods in interfaces.

Use cases:

- Allow extension of interfaces without breaking existing implementations
- Reuse methods on several classes, with less limitations than abstract base class (i.e. a limited form of traits)
- Mark optional methods (e.g. `Iterator.remove()`)

Default Methods - Properties and limitations

```
interface DefaultExample {
  int foo();
  int bar();
  default int sum() {
    return foo() + bar();
  }
}
```

- No state, i.e. no fields or static vars in interfaces
- May access other methods defined in the interface
- when implementing two interfaces with same default method: override required

Default Methods - Properties and limitations

An implementation in a class always takes precedence

- includes methods in implementing class and methods inherited from extending other classes
- therefore can not override methods from Object (e.g. `boolean equals(Object)`, `int hashCode()`, `String toString()`)
- But *can* override default method of parent interface

@FunctionalInterface

```java
@FunctionalInterface
interface Comparator {
  int compare(T o1, T o2);
  /* ... */
}
```

An interface annotated with `@FunctionalInterface` …

- may only have one abstract method (otherwise: compiler error)
- can contain static and default methods
- can be instantiated with a lambda (although not required)

λ Lambdas λ

Outline

Anonymous Functions (Lambdas)

New syntax for defining inline functions:

```
// from int to int
(int x) -> x * x

// more than one argument
(int a, int b) -> a * b
```

Anonymous Functions (Lambdas)

Types can be inferred

```
// single argument with inferred type
x -> x * x

// with multiple arguments parentheses are required
(a, b) -> a * b

// function without parameters
() -> new Foo()
```

Anonymous Functions (Lambdas)

Lambda may take a multiline block:

```
// single expression
(Foo arg) -> arg.doSomething()

// block, requires 'return'
(Foo arg) -> {
  String res = arg.doSomething();
  Bar finalRes = ...  /* do more stuff */
  return finalRes;
}
```

Anonymous Functions (Lambdas)

Types of Lambdas

A lambda takes the type of an interface with exactly one abstract method, which matches its signature.

Anonymous Functions (Lambdas) - Types

```
class Foo { String doSomething() { /*...*/ } }

// single expression
(Foo arg) -> arg.doSomething()
```

- takes one argument of type `Foo`
- returns the result of the expression, no `return` needed
- returns a `String`, the type of the expression

$\Longrightarrow$ matching signature: `String someName(Foo theFoo)`

Anonymous Functions (Lambdas) - Types

```
// block
(Foo arg) -> {
  String res = arg.doSomething();
  Bar finalRes = ...  /* do more stuff */
  return finalRes;
}
```

- takes one argument of type `Foo`
- evaluates the block after `->`
- returns a `Bar` (type of `finalRes`)
- `return` required, otherwise return type is void

$\Longrightarrow$ matching signature: `Bar someName(Foo theFoo)`

Standard Library Types

```java
@FunctionalInterface
public interface Function<T, R> {

    /**
     * Applies this function to the given argument.
     *
     * @param t the function argument
     * @return the function result
     */
    R apply(T t);
```

Back to our example...

```
Function<Foo, String> myFooFunc =
(Foo arg) -> arg.doSomething();

Foo myFoo = new Foo();

String res = myFooFunc.apply(myFoo);
```

Back to our example... with type inferrence

```
Function<Foo, String> myFooFunc =
(arg) -> arg.doSomething();

Foo myFoo = new Foo();

String res = myFooFunc.apply(myFoo);
```

Standard Library Types - java.util.function

`Function<T,R>`	takes a `T`, returns an `R`
`BiFunction<T,U,R>`	takes a `T` and a `U`, returns an `R`
`UnaryOperator<T>`	like above, all types identical
`BinaryOperator<T>`	
`Predicate<T>`	like above, returns a `boolean`
`BiPredicate<T,U>`	
`Consumer<T>`	takes a `T`, returns nothing
`Supplier<T>`	takes no arguments, returns a `T`

Capturing

Lambdas may refer to variables outside the lambda:

- local variables, if they are (effectively) final
- fields
- static variables

```
int k = 2;
return x -> x * k;
```

Lambdas - Caveats

What you cannot do with lambdas:

- modify captured local variables (that would make them non-final)
  - but captured fields and static variables can be changed
- throw checked exceptions not declared by the functional interface
- use `break` or `return` to break out of surrounding loops or return from the calling function

More examples

Lambdas may instantiate any single-abstract-method (SAM) interface, for example:

- Comparators:

```
List<Person> people = ...;
people.sort((a,b) ->
↪  a.getName().compareTo(b.getName()));
```

- GUI Event handlers:

```
button.setOnAction((event) -> log("Button
↪  clicked"));
```

Method References

What if we want to call an existing method, that already has the right signature?

- Existing methods can be referenced with new `::` operator
- Can reference static methods, instance methods and constructors
- instance methods either bound to an instance, or take an additional parameter for `this`

Method References - Referencing static methods

Static methods are referenced by `ClassName::methodName`:

```
Function<String, Integer> strToInt = Integer::parseInt;
int answer = strToInt.apply("42");
```

Method References - Referencing instance methods

With `instance::methodName`, reference is bound to an object:

```
Predicate<Object> inSet = mySet::contains;
if(inSet.test(someObject)) { ... }
```

When using `ClassName::methodName`, add instance as first parameter

```
BiPredicate<Set<?>, Object> inSet = Set::contains;
if(inSet.test(mySet, someObject)) { ... }
```

Method References - Referencing constructors

Reference constructors with `ClassName::new`:

```
Supplier<ArrayList<String>> listFactory =
↪  ArrayList::new;
ArrayList<String> = listFactory.get();
```

Method References - Overloaded methods

When a method or constructor is overloaded, select the one matching the used interface:

```
IntFunction<ArrayList<String>> listWithCapacity =
↪  ArrayList::new;
ArrayList<String> = listFactory.apply(5);
```

Method References - Overloaded methods

Caveat: static method and instance method with same signature cannot be handled:

```
// compile error: ambiguous
Function<Integer, String> i2s = Integer::toString;

// could be
static Integer.toString(int a);
Integer.toString();
```

Method References - comparison to lambdas

Method Reference	equivalent lambda
`Integer::parseInt`	`str -> Integer.parseInt(str)`
`Set::contains`	`(mySet, e) -> mySet.contains(e)`
`mySet::contains`	`(e) -> mySet.contains(e)`
`ArrayList::new`	`() -> new ArrayList()`

A typical task in the JPP

Task:

You have a class `Person` with first name, last name, age and an address. Write a comparator for sorting Persons by last name, then by first name, then by age and then by address.

The old way to write comparators

```
class PersonComparator implements Comparator<Person> {
  int compare(Person a, Person b) {
    int result = a.getLastName()
        .compareTo(b.getLastName());
    if(result == 0) {
      result = a.getFirstName()
          .compareTo(b.getFirstName());
      if(result == 0) {
        result = Integer.compare(...
        if(result == 0) {
          ...
        }
      }
    }
    return result;
  }
}
```

Comparators with new API - leveraging method references

```
Comparator<Person> myPersonComparator =
Comparator.comparing(Person::getLastName)
          .thenComparing(Person::getFirstName)
          .thenComparingInt(Person::getAge)
          .thenComparing(Person::getAddress);
```

- much shorter (fits on the slide ;-)
- describes what is happening, not how
- small enough so that a separate class is usually not needed

Comparators with new API - leveraging method references

Several new utility methods in `Comparator`:

- `comparing` / `thenComparing` for Comparables, primitives or other types with an additional comparator
- instance method `reversed()`
- `Comparator.naturalOrder()`, when an API needs a comparator, but your type is already `Comparable`
- `Comparator.nullsFirst(Comparator c)` for sorting collections containing nulls.
  If you need this, there's something wrong with your collection.

Outline

Iterating over collections

One of the most common tasks: do something with several input objects:

- `for` -loop with indices
- `Iterator` with `hasNext()`, `next()`
- enhanced `for` -loop (`for(T elem : coll) ...`)

Iterating over collections

One of the most common tasks: do something with several input objects:

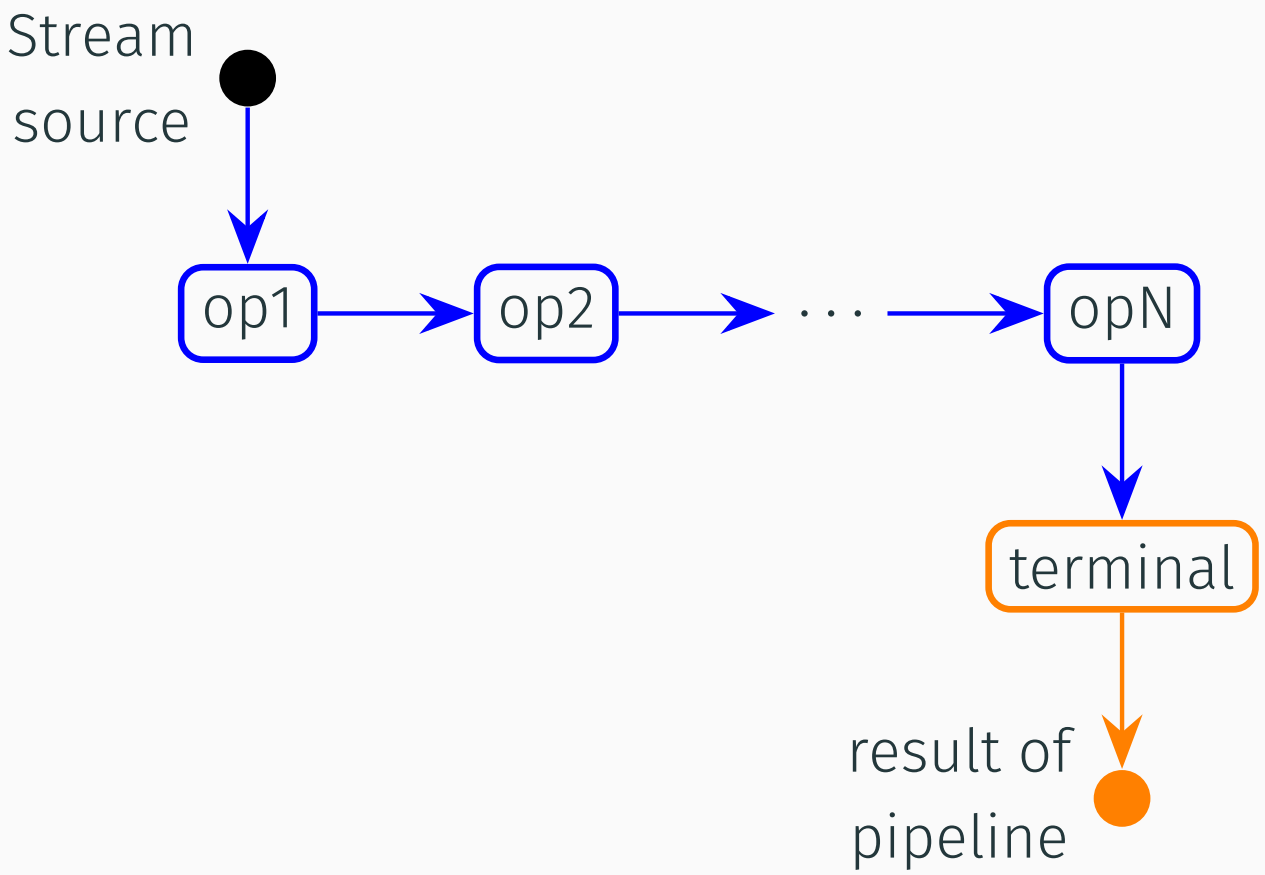
- `for` -loop with indices
- `Iterator` with `hasNext()`, `next()`
- enhanced `for` -loop (`for(T elem : coll) ...`)
- *New:* `Collection<T>.forEach(Consumer<T>)`
- *New:* Element Streams

Streams - Handling collections with Lambdas

- new package `java.util.stream` contains Classes for handling collection elements in a more functional way
- `Stream<T>` similar to iterator: can only be used once, returns elements, may be infinite
- provides several operations on stream elements
- operations are chained and result in a pipeline
- can be parallelized
- primitive variants for `double`, `int` and `long`

Stream operations

Streams can have several intermediate operations, that return a Stream, and a single terminal operation, that can return any type. A stream pipeline:



Intermediate Operations on Streams - stateless

An intermediate operation transforms the stream into another stream.

- `map(Function<? super T, ? extends R> f)`:
  apply `f` to each element, return stream of results
- `filter(Predicate<? super T> p)`:
  apply `p` to each element, return stream of all elements, for which it returns true
- `flatMap(Function<...Stream<R>> f)`:
  apply a function that returns a stream to each element.
  return stream of all elements in the results.

Intermediate Operations on Streams - stateful

Stateful intermediate operations keep some kind of state and may be slow in parallel streams

- `limit(int n)`:
  return a stream of the first `n` elements
- `skip(int n)`:
  return the stream without the first `n` elements
- `sorted()`:
  returns a stream in sorted order (optionally with comparator)
- `distinct()`:
  returns a stream without duplicated elements (first occurrence only)

Intermediate Operations on Streams - Example

```
Stream<String> lowercasedLongStrings =
    someStringCollection.stream()          //(1)
      .filter(str -> str.length() > 10)    //(2)
      .map(String::toLowerCase)            //(3)
      .flatMap(str -> {                    //(4)
        String[] segments = str.split(",");
        return Arrays.stream(segments)
      });
```

1. start with a stream of strings
2. check for each string, if it is longer than 10 chars
3. convert remaining strings to lower case
4. split each element at comma and turn into stream
   flatMap concatenates $\Rightarrow$ no `Stream<Stream<String>>`

Terminal Operations on Streams

A terminal operation produces a result or side-effect. Streams are lazy, intermediates not applied until terminal operation is called.

- `forEach(Consumer c)` / `forEachOrdered(Consumer c)`: pass all elements to `c`, optionally preserving order
- `reduce(T identity, BinaryOperator<T> accumulator)`: combines all elements with associative accumulator, resulting in a single `T`. Omitting the identity object makes result type `Optional<T>` (next chapter)
- `collect(Collector<? super T,A,R> collector)`: similar to reduce, but with mutable accumulator. Predefined collectors in `java.util.stream.Collectors`

Terminal Operations on Streams - boolean terminals

These are *short-circuiting*, i.e. may not need to look at all elements. Test given predicate against elements and return true, if any element / all elements / no element matches:

- `anyMatch(Predicate<? super T> pred)`
- `allMatch(Predicate<? super T> pred)`
- `noneMatch(Predicate<? super T> pred)`

Terminal Operations on Streams - more terminals

- `count()`:
  number of elements (useful in combination with filter)
- `min/max(Comparator<? super T> comp)`:
  the minimum / maximum element of the stream by the given comparator
- `findAny() / findFirst()` returns some / the first element of the stream. `findAny()` is non-deterministic, but faster in parallel streams.

Additional operations on primitive streams:

- `sum()`: sum of all elements
- `average()`: arithmetic mean over all elements

Collectors

Collectors accept elements from a stream and combine them in some way. Most common operation: pack them into a collection:

```
Stream<String> lowercasedLongStrings =
  someStringCollection.stream()
    .filter(str -> str.length() > 10)
    .map(String::toLowerCase)
    .flatMap(str -> {
      String[] segments = str.split(",");
      return Arrays.stream(segments)
    });
```

Collectors

Collectors accept elements from a stream and combine them in some way. Most common operation: pack them into a collection:

```
List<String> listOfStrings =
  someStringCollection.stream()
    .map(...).filter(...).flatMap(...)
    .???
```

Collectors

Collectors accept elements from a stream and combine them in some way. Most common operation: pack them into a collection:

```
List<String> listOfStrings =
  someStringCollection.stream()
    .map(...).filter(...).flatMap(...)
    .collect(Collectors.toList());
```

Collectors

Collectors accept elements from a stream and combine them in some way. Most common operation: pack them into a collection:

```
Set<String> setOfStrings =
  someStringCollection.stream()
    .map(...).filter(...).flatMap(...)
    .collect(Collectors.toSet());
```

Collectors

Collectors accept elements from a stream and combine them in some way. Most common operation: pack them into a collection:

```
String concatenatedString =
  someStringCollection.stream()
    .map(...).filter(...).flatMap(...)
    .collect(Collectors.joining(", "));
```

Collectors - Maps

Several collectors for maps:

```
// calculating a key for each entry
Map<Integer, Person> personsById =
  persons.stream().collect(
    Collectors.toMap(
      // calculate map key
      Person::getId,
      // calculate map value
      Function.identity()
      // optionally: merge function on duplicate keys
    )
  );
```

Collectors - Maps

Several collectors for maps:

```
// when we expect duplicates we want to keep
Map<String, List<Person>> personsByLastName =
  persons.stream().collect(
    Collectors.groupingBy(Person::getLastName)
  );
```

Collectors - Maps

Several collectors for maps:

```
// very similar, like groupingBy but with a predicate
Map<Boolean, List<Person>> admins =
  persons.stream().collect(
    Collectors.partitioningBy(Person::isAdmin)
  );
```

Other Stream souces

Streams don't have to come from a collection:

- infinite Streams: `Stream.generate(Supplier<T> s)`, `Stream.iterate(T seed, UnaryOperator<T> f)`
- initialize with elements: `Stream.of(T... elem)`
- Streaming with arrays: `Arrays.stream(T[] a)`

Outline

Optional

- Represents a value that may or may not be there
  `Optional.of(value)`, `Optional.empty()`
- can perform operations on value without explicit check
- if operation on value can also return `Optional`, no nested checks are required

Optional

To retrieve the value:

```
// get() throws exception for empty optional.
// much like using nulls, but fails early
if(optional.isPresent()) optional.get();

// default value, if the optional is empty
optional.orElse(theDefault);

// throw custom exception if empty
optional.orElseThrow(IllegalArgumentException::new);
optional.orElseThrow(() -> new NoSuchElementException(
    "important stuff missing!"
));
```

Optional

Or just pass it to a consumer:

```
optional.ifPresent(val -> {
  System.out.println(val);
});

// shorter
optional.ifPresent(System.out::println);
```

Optional

Working on optionals without ”unboxing”:

```
Optional<String> optStr = stringStream.findAny();

// if Optional has value: apply operation
optStr = optStr.map(String::trim);

// if Optional has value: check predicate.
// if that returns false, result is empty optional
optStr = optStr.filter(str -> str.contains("foo"));

// flatMap handles operations returning optionals
optStr = optStr.flatMap(str -> findMatch(str));
```

All operations called on the empty Optional return it unchanged.

Optional - Usage as return type

Typical example: method that finds value by some property

```
public Part findPart(String id) {
  for(Part p : this.parts) {
    if(p.getId().equals(id)) {
      return p;
    }
  }
  return null;
}
```

Problems:

- caller must look into method, to see if there can be no result
- caller can easily forget to check for nulls
- with several such methods, lots of null checks required

Optional - Usage as return type

Same method using streams and `Optional<Part>`

```
public Optional<Part> findPart(String id) {
  return this.parts.stream()
    .filter(p -> p.getId().equals(id))
    .findAny();
}
```

- `filter` only passes matching elements
- element passed to `findAny()` ⇒ return optional of element
- no element matches ⇒ filtered stream is empty. `findAny()` on empty stream ⇒ return empty optional

Optional - Usage as return type

For results not from a stream, use:

```
public Optional<Foo> getFoo() {
  if(someBool) return Optional.of(buildTheFoo());
  else return Optional.empty();
}
```

When using an API that may return nulls, but cannot be changed:

```
public Optional<Foo> getFoo() {
  // returns empty optional if argument is null
  return Optional.ofNullable(getFooOrNull());
}
```

Outline

Parallelization with Streams

Using `parallelStream()` or calling `parallel()` will not magically make your code work in parallel.

- parallelization done by splitting the stream
- read the javadocs for stream operations for behaviour in parallel execution
- take special care of not breaking contracts (e.g. don't use stateful lambdas)
- choose terminal and stateful operation carefully (again: read the docs)
  - `findAny` instead of `findFirst` if possible
  - use collectors made for parallel operation
  - `distinct` expensive on ordered steams

Other News

More API Additions

- New date API based on joda-time: `java.time`
- Several new methods on collections and maps (e.g. `computeIfAbsent`)
- Concurrency API extended for use with lambdas (e.g. `CompletableFuture`)
- IO/NIO API has several new methods returning streams (e.g. `Files.lines`)

Next Year: Java 9

Features (probably) coming in Java 9:

- Module System
- Java REPL
- Micro Benchmarking Suite