Wycieczka po Javie 7 i 8

24-03-2015

What's new in Java SE 7

- Swing
- IO and New IO
- Networking
- Security
- Concurrency Utilities
- Rich Internet Apps development
- Java 2D
- Java XML - JAXP, JAXB and JAX-WS
- Internalization
- Multithreaded custom class loaders

- Binary literals
- Strings in switch statements
- try-with-resources
- Catching multiple exceptions and improved type checking
- Underscores in numeric literals
- Type inference for generic instance creation
- `@SafeVarargs`
- `invokedynamic`
- G1 Collector
- JDBC 4.1

What's new in Java SE 7

- Swing
- IO and New IO
- Networking
- Security
- Concurrency Utilities
- Rich Internet Apps development
- Java 2D
- Java XML - JAXP, JAXB and JAX-WS
- Internalization
- Multithreaded custom class loaders

- Binary literals
- **Strings in switch statements**
- **try-with-resources**
- **Catching multiple exceptions and improved type checking**
- Underscores in numeric literals
- **Type inference for generic instance creation**
- `@SafeVarargs`
- `invokedynamic`
- G1 Collector
- JDBC 4.1

What's new in Java SE 8

- lambda expressions
- method references
- interface default methods
- interface static methods
- repeatable annotations
- improved type inference
- method parameter reflection
- `CompletableFuture`
- `java.util.stream`
- improved `java.util.HashMap`
- Java RX
- JavaDoc tool
- Date-Time package
- Nashorn Javascript engine
- IO and NIO improvements
- parallel array sorting
- standard en/decoding Base64
- unsigned arithmetic support
- JDBC 4.2
- Java DB 10.10
- Java XML - JAXP
- removal of PermGen

What's new in Java SE 8

- **lambda expressions**
- **method references**
- **interface default methods**
- **interface static methods**
- repeatable annotations
- improved type inference
- method parameter reflection
- `CompletableFuture`
- **`java.util.stream`**
- improved `java.util.HashMap`
- Java RX
- JavaDoc tool
- **Date-Time package**
- Nashorn Javascript engine
- IO and NIO improvements
- parallel array sorting
- standard en/decoding Base64
- unsigned arithmetic support
- JDBC 4.2
- Java DB 10.10
- Java XML - JAXP
- **removal of PermGen**

Better date/time API

Basic date/time types to begin with

- `Instant` - a timestamp, point on the timeline
- `LocalDate` - a date without a time, or any reference to an offset or time-zone
- `LocalTime` - a time without a date, or any reference to an offset or time-zone
- `LocalDateTime` - combines date and time, but still without any offset or time-zone
- `ZonedDateTime` - a "full" date-time with time-zone and resolved offset from UTC/Greenwich

Better date/time API

Naming conventions

- `of` - static factory method
- `parse` - static factory method focussed on parsing
- `get` - gets the value of something
- `is` - checks if something is true
- `with` - the immutable equivalent of a setter
- `plus` - adds an amount to an object
- `minus` - subtracts an amount from an object
- `to` - converts this object to another type
- `at` - combines this object with another, such as `date.atTime(time)`

Better date/time API

- `TemporalAdjuster` - temporal objects modifiers (*firstDayOfNextMonth*, *lastDayOfYear* etc)

Programming paradigms in Java 8

Java is:

- imperative,
- object-oriented,
- declarative (annotations),
- functional,
- concurrent,
- generic,
- statically typed.

Programming paradigms in Java 8

Java is:

- imperative,
- object-oriented,
- declarative (annotations),
- **functional,**
- concurrent,
- generic,
- statically typed.

Functional Programming paradigm

Functional Programming

- functions are first-class citizens (*code as data*)
- higher order functions
- there is no state (immutability)
- there are no side effects
- substitution model
- *persistent* data structures
- tail recursion (no loops)
- lazy evaluation
- reactive programming
- monads

Introducing lambda expression

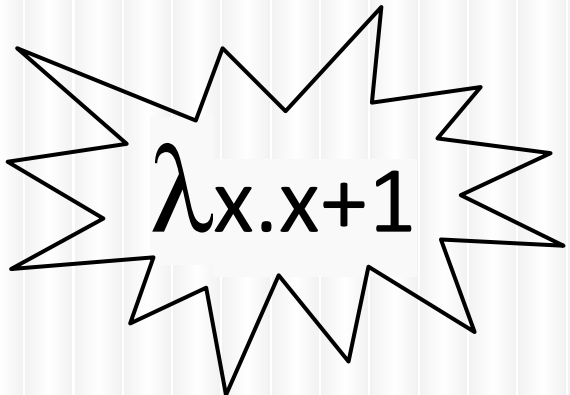
- numeric literals: `1, 3.14f, 5000L, 1_000_000, 0xCafe`
- binary literals (since Java 7): `0b10101`
- boolean literals: `true, false`
- String literal: `"Java 8"`
- array literals: `{ 1, 2, 3 }, { {x, y}, {1, 2} }`

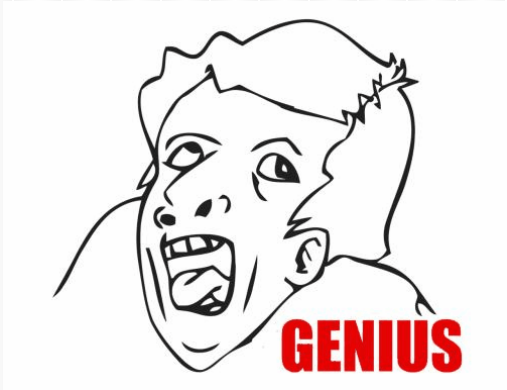
Introducing lambda expression

- numeric literals: `1, 3.14f, 5000L, 1_000_000, 0xCafe`
- binary literals (since Java 7): `0b10101`
- boolean literals: `true, false`
- String literal: `"Java 8"`
- array literals: `{ 1, 2, 3 }, { {x, y}, {1, 2} }`
- function literal: `(int x) -> x+1`

Introducing lambda expression

- numeric literals: `1, 3.14f, 5000L, 1_000_000, 0xCafe`
- binary literals (since Java 7): `0b10101`
- boolean literals: `true, false`
- String literal: `"Java 8"`
- array literals: `{ 1, 2, 3 }, { {x, y}, {1, 2} }`
- function literal: `(int x) -> x+1`





Syntax

```
Function<String, Integer> f =
(String s) -> { return s.length; };
```

Syntax

```
Function<String, Integer> f =
(String s) ->           s.length;
```

Syntax

```
Function<String, Integer> f =
        s  ->           s.length;
```

Syntax

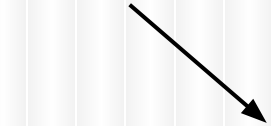
```
Function<String, Integer> f =
                    String::length;
```

Syntax

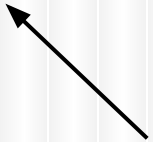
```
BiFunction<String, String, Integer> compare =
(x, y) -> x.compareTo(y);

Function<String, Integer> f = String::length;

Supplier<Integer> random = () -> r.nextInt();

Consumer<Person> introducer =
p -> println("My name is " + p.name);

IntFunction<Integer> incrementator = x -> x+1;
```

Syntax

Unit type



```
Supplier<Integer> random = () -> r.nextInt();
```



Unit => Integer

`@FunctionalInterface`

- Every lambda expression is compatible to *SAM interfaces* with method of matching type
- We can annotate *SAM interface* with `@FunctionalInterface`
- … but we don't have to (`Function` from *Guava*).

Closures

- Lambda expressions can access **effectively final variables** from the enclosing scope

4 flavours of *Method References*

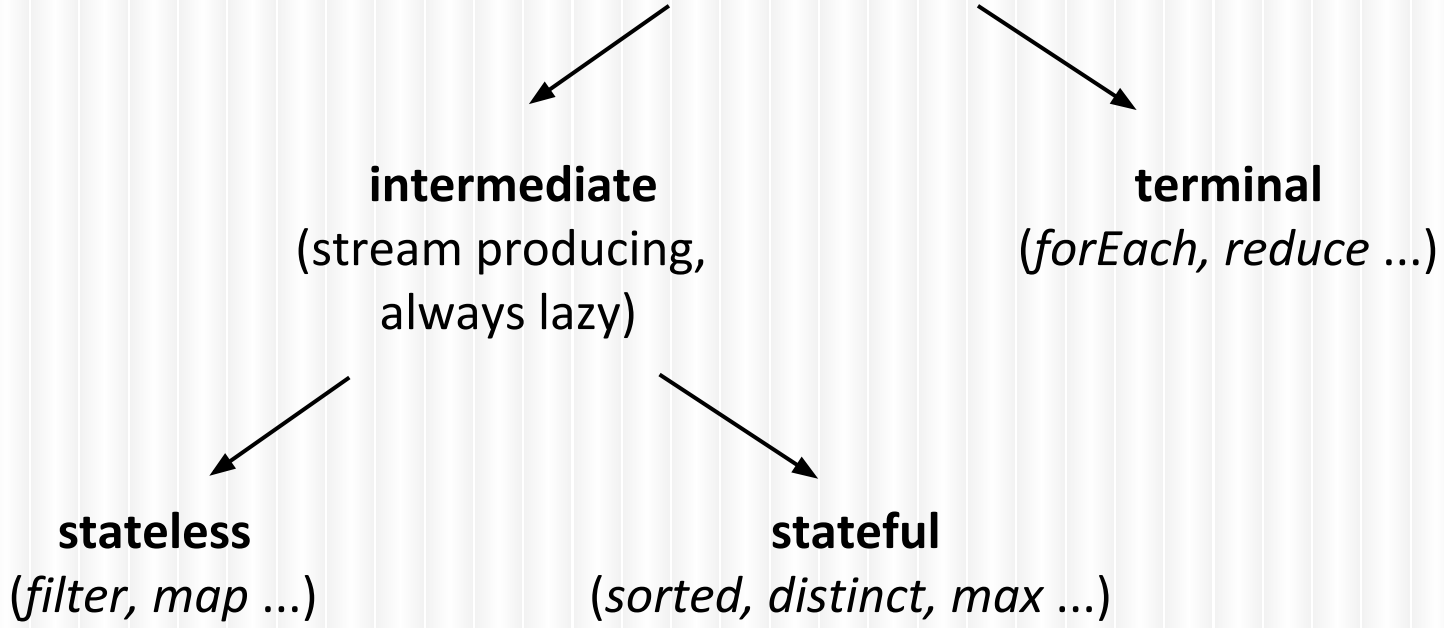
- instance method of given object: `p::getName`
- instance method of given type: `String::compareTo`
- static method: `Math::max`
- constructor: `HashSet<>::new`

Stream API

Stream - the key abstraction

- a stream is **not a data structure** - it conveys elements from a source through a pipeline of computational operations,
- **functional** in nature - an operation on a stream produces a result, but does not modify its source,
- many stream operations (filtering, mapping, or duplicate removal) can be implemented **lazily**,
- possibly **unbounded,**
- elements of a stream are **only visited once** during the life of a stream,
- may be effectively **parallelized.**

Operations on streams



java.util.stream

- `Stream<T>`
- `Collectors`

- `IntStream, boxed()` - primitives support

Consuming stream

- `collect()` - consume stream using Collector that wraps mutable result
- `reduce()` - consuming a stream in pure functional way, always create new element when it process an element

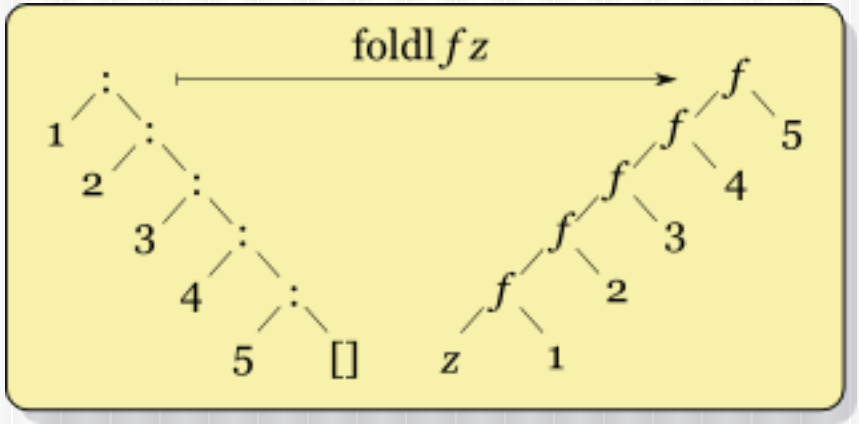
Functional loops

- `for` -> `forEach`
- `while` -> no equivalent, may use `limit(long)`

- `reduce` (aka *fold*)

- no mutable state! (otherwise can’t be parallelized)

reduce()



Parallelism

- `parallelStream()`
- `ForkJoinPool.getCommonPoolParallelism()`
- //TODO: performance
- //TODO: parallel stream processing traps

Old classes, new methods

- `Iterable - forEach`
- `Comparator` - many utility methods (`comparing, reversed` etc)
- `Map` - default methods
- `Map.Entry` - `comparingByKey, comparingByValue`

Higher order functions

$$\frac{\partial}{\partial x}\left(\frac{\partial f}{\partial x}\right) = \frac{\partial^2 f}{\partial x^2} = f_{xx} = D^2{}_{xx} f,$$

$$\frac{\partial}{\partial x}\left(\frac{\partial f}{\partial y}\right) = \frac{\partial^2 f}{\partial x\,\partial y} = f_{xy} = D^2{}_{xy} f,$$

$$\frac{\partial}{\partial y}\left(\frac{\partial f}{\partial x}\right) = \frac{\partial^2 f}{\partial y\,\partial x} = f_{yx} = D^2{}_{yx} f,$$

$$\frac{\partial}{\partial y}\left(\frac{\partial f}{\partial y}\right) = \frac{\partial^2 f}{\partial y^2} = f_{yy} = D^2{}_{yy} f.$$

Higher order functions

- composing functions
- currying and partial argument application

Composing functions

Given functions $f : X \to Y$ and $g : Y \to Z$
one can define a new function $(g \circ f) : X \to Z$,
such that $(g \circ f)(x) = g(f(x))$.

Composing functions

Given functions $f : X \to Y$ and $g : Y \to Z$
one can define a new function $(g \circ f) : X \to Z$,
such that $(g \circ f)(x) = g(f(x))$.

```
default <V> Function<V,R> compose(Function<? super V,? extends T>
before)
```

Currying

- `(X, Y) -> Z` can be transformed to `X -> Y -> Z`, which is same as `X -> (Y -> Z)`
- now we can apply arguments one by one, each time constructing a new function

Introduction to Monads

without category theory

Monads

- *“a **monad** is a structure that represents computations defined as sequences of steps: a type with a monad structure defines what it means to chain operations, or nest functions of that type together.”*
  - Wikipedia

Monads

- *“a **monad** is a structure that represents computations defined as sequences of steps: a type with a monad structure defines what it means to chain operations, or nest functions of that type together.”*
  - Wikipedia
- monad hides side effects (exceptions, change of state, concurrency, I/O etc)

Monads

- *“a **monad** is a structure that represents computations defined as sequences of steps: a type with a monad structure defines what it means to chain operations, or nest functions of that type together.”*
  - Wikipedia
- monad hides side effects (exceptions, change of state, concurrency, I/O etc)
- parametrized structure *M<T>* with at least two operations:
  - ➢ `bind: M<T1> -> M<T2>` (aka *flatMap*)
  - ➢ `unit: T -> M<T>` (aka *of*)

Monads

- *“a **monad** is a structure that represents computations defined as sequences of steps: a type with a monad structure defines what it means to chain operations, or nest functions of that type together.”*
  - Wikipedia
- monad hides side effects (exceptions, change of state, concurrency, I/O etc)
- parametrized structure *M<T>* with at least two operations:
  - ➢ *`bind: M<T1> -> M<T2>`* (aka *flatMap*)
  - ➢ *`unit: T -> M<T>`* (aka *of*)
- important monads: *List, Optional, Promise, Try*

map vs *flatMap*

- *map*: (T -> U) -> M<U>
- *flatMap*: (T -> M<U>) -> M<U>

map vs *flatMap*

- *map*: (**T -> U**) -> **M<U>**
- *flatMap*: (**T -> M<U>**) -> **M<U>**

- inappropriate usage of *map* may cause monad to become nested:

```
Stream<Person> people = ...

people.map(Person::getPhones)     // Stream<Stream<Phone>>

people.flatMap(Person::getPhones) // Stream<Phone>
```

- types will always guide you!

Metaspace

“Java HotSpot(TM) 64-Bit Server VM warning: ignoring option MaxPermSize=256m; support was removed in 8.0”

Metaspace

“Java HotSpot(TM) 64-Bit Server VM warning: ignoring option MaxPermSize=256m; support was removed in 8.0”

- *PermGen* space is completely removed, JVM arguments are ignored + warning
- most allocations for the class metadata are allocated out of native memory
- Soma data has been moved to Heap space
- new flags: MetaspaceSize, `MaxMetaspaceSize`, `MinMetaspaceFreeRatio`, `MaxMetaspaceFreeRatio`

Metaspace

“Java HotSpot(TM) 64-Bit Server VM warning: ignoring option MaxPermSize=256m; support was removed in 8.0”

- *PermGen* space is completely removed, JVM arguments are ignored + warning
- most allocations for the class metadata are allocated out of native memory
- Soma data has been moved to Heap space
- new flags: MetaspaceSize, `MaxMetaspaceSize, MinMetaspaceFreeRatio, MaxMetaspaceFreeRatio`
- *ERROR: java.lang.OutOfMemoryError: Metadata space*

Where to go from here

- [Code] https://github.com/mszarlinski/java-8-talk
- [Code] https://github.com/nurkiewicz/java8-workshop
- [LocalDate] http://www.journaldev.com/2800/java-8-date-time-api-example-tutorial-localdate-instant-localdatetime-parse-and-format
- [Monads] http://java.dzone.com/articles/whats-wrong-java-8-part-iv
- [Monads] https://www.youtube.com/watch?v=ZhuHCtR3xq8
- [Metaspace] http://mail.openjdk.java.net/pipermail/hotspot-dev/2012-September/006679.html
- [Metaspace] http://javaeesupportpatterns.blogspot.ie/2013/02/java-8-from-permgen-to-metaspace.html
- [Scala] https://www.coursera.org/course/progfun