JAVA 8

HIPSTER SLIDES

NEW JAVA VERSION FROM FUNCTIONAL PROGRAMMER PERSPECTIVE

Created by Oleg Prophet / @oregu_desu

JAVA 8

03/18/2014
Last update: 5

Unofficial tagline:

YOU CAN BE HIPSTER TOO.

FEATURES

- Mixins, aka default methods
- Collection goodies
- More type inference
- Project Lambda
- Streams
- No Permgen. No OOME: permgen space errors*

DEFAULT METHODS

- Known as Defender Methods
- Implementation methods in interfaces
- Poor man's Mixins
- Multiple inheritance
- (With ambiguity resolving mechanism!)
- Reduce abstract classes
- Utility methods
- “Adding a method to an interface is ~~not~~ now a source-compatible change”

DEFAULT METHODS EXAMPLE

```
public interface Sized {
  default boolean isEmpty() {
    return size() == 0;
  }
  int size();
}
```

À-LA MIXINS EXAMPLE

```
class VeryFastCar extends ACar implements IFastCar, IFastSteerCar {}
class VerySlowCar extends ACar implements ISlowCar, ISlowSteerCar {}

// Even better would be (you can in Scala)
ICar car = new ACar() with ISlowCar, IFastSteerCar;
```

MORE POWER TO INTERFACES

Finally! Define static methods **right** in the interfaces.

How that makes you feel, huh?

Remove your Collections, Arrays, Paths now.

COLLECTION GOODIES

MAPS:

- getOrDefault(K, V) \m/
- putIfAbsent(K, V)
- replace(K, V new)
- replace(K, V old, V new)
- compute(K, BiFunction) *
- computeIfAbsent(K, Function) *
- computeIfPresent(K, BiFunction) *
- merge(T, V, BiFunction) *

Reduce your boilerplate.

COLLECTION GOODIES

Set and List didn't change interface much, but let's lookup Collection and Iterable.

- spliterator() *
- removeIf(Predicate) *
- stream() *
- parallelStream() *
- (Iterable).forEach(Consumer) *

* We'll get to them in a moment.

DATE/TIME GOODIES

Since mutability is evil, we replaced java.util.Date class with a bunch of immutable java.time.* classes!

“All the classes are immutable and thread-safe.”

TYPE INFERENCE

JAVA 7

```
void processStringLst(List<String> l) { ... }

Lst.processStringLst(List.<String>empty());
```

JAVA 8

```
Lst.processStringLst(List.empty());
```

BUT STILL

```
String s = Lst.<String>singleton().head();
```

Meh...

TYPE INFERENCE

More we'll see in lambda slides

LAMBDA SLIDES

() → {}

() → {}

- Project Lambda (JSR #335)
- Initiated in December 2009 as Straw-Man proposal
- Loooong awaited
- Full class support
- Not a syntactic sugar for an anonymous inner class
- Even though it can appear so.
- They are not even objects.

WITHOUT () → {}

```java
List<String> names = new ArrayList<String>();
for (int i = 0; i < fields.size(); i++) {
  Field fld = fields.get(i);
  names.add(fld.getName());
}
for (int i = 0; i < names.size(); i++) {
  String name = names.get(i);
  System.out.println(name);
}
```

() → {}

```
names = fields.stream().map(Field::getName).collect(toList());
names.forEach(System.out::println);
```

() → {}

```
names.map((String s) -> { return s.length(); });
```

We know it's a collection of strings!

```
names.map((s) -> s.length());
```

That's not a LISP! Who likes parentheses anyway?

```
names.map(s -> s.length());
```

Can I have a method reference, please?

```
names.map(String::length);
```

Thank you, Java 8.

() → {}

METHOD REFERENCES

Object::toString

Field::create

Field::new

this::processField

a::process (a is some object in scope)

MORE () → {} EXAMPLES

```
// Group employees by department
Map<Department, List<Employee>> byDept
    = employees.stream().collect(groupingBy(Employee::getDepartment));
```

```
// Partition students into passing and failing
Map<Boolean, List<Student>> passingFailing
  = students.stream().collect(partitioningBy(
                                s -> s.getGrade() >= PASS_THRESHOLD));
```

```
// Classify people by state and city
Map<String, Map<String, List<Person>>> peopleByStateAndCity
  = personStream.collect(groupingBy(Person::getState,
                                    groupingBy(Person::getCity)))
```

FUNCTIONAL INTERFACES

```
@FunctionalInterface
public interface Function<T, R> {
  R apply(T t);
}
```

```
Function<String, String> m = s -> s.toUpperCase();
Function<String, Integer> f = String::length;
Function g = f.andThen(Integer::reverse);
```

```
Function id = Function.identity();
```

COMPOSE LIKE A PRO

Function composition

$$f : X \to Y$$

$$g : Y \to Z$$

$$g \circ f : X \to Z$$

```java
Function<String, Integer> f = String::length;
Function<Integer, Float> g = Integer::floatValue;
Function h = g.compose(f);
```

CURRY LIKE A PRO

```java
Function<String, UnaryOperator<String>> curried =
                          s1 -> s2 -> s1.concat(" ").concat(s2);

// Partial application
UnaryOperator<String> hask = curried.apply("Haskell");

out.println(hask.apply("Curry"));
out.println(hask.apply("Wexler"));
```

* Currying is a fancy name for schönfinkeling

CURRY LIKE A SEMI-PRO

Can't curry any function like (a, b) → a + b;

But we have tools:

```
public interface Curry {
  static <T,U,R> Function<U, R> curry(BiFunction<T, U, R> bi, T t) {
    return u -> bi.apply(t ,u);
  }
}
```

```
BiFunction<String, Integer, Float> bi = (s, i) -> (s.length() + i)/2.0f;
// Can't do bi.curry("hello") for any bi

Function<Integer, Float> part = Curry.curry(bi, "hello");

// Will we be able call part(10) someday?
out.println(part.apply(10));
out.println(part.apply(22));
```

JAVA.UTIL.FUNCTION.*

- Function<T, R>
- BiFunction<T, U, R>
- Predicate<T>
- Supplier<T>
- Consumer<T>
- BiConsumer<T, U>
- UnaryOperator<T> : Function<T, T>

STREAMS

- filter
- map
- flatMap
- distinct
- sorted
- limit

These are intermediate operations

They are all lazy.

STREAMS

- forEach *
- forEachOrdered
- toArray
- reduce
- collect *
- min, max, count, sum
- (any|all|none)Match
- findAny *

These are terminal operations

They are not lazy at all.

No element will be produced until you call one of these.

* Collectors api: toList(), counting(), joining(), etc.

PARALLEL STREAMS

From sequential to parallel in the blink of an eye

```
lns = names.parallelStream().collect(groupingBy(String::length));
lns.forEach((key, ns) -> out.println(key + ":\t" +
                                        ns.stream().collect(joining(", "))));
```

FAILED COMPUTATION?

`findAny()` returns special container object `Optional`

- `isPresent, ifPresent(Consumer)`
- `orElse, orElse(Supplier), orElseThrow`
- Treat like collection: `map, flatMap, filter`
- Create Optional: `empty, of(val), ofNullable(val)`

A convenient way to represent result absence.

(And reduce NPE count.)

NO MORE PERMGEN

No more PermGen space errors and PermGen tuning.

Java says:
VM warning: ignoring option MaxPermSize=512m;
support was removed in 8.0

Jon Masamitsu:

> *A goal for removing perm gen was so that users do not have to think about correctly sizing it.*

— But where are my class instances?

METASPACE!

METASPACE!

```
java.lang.OutOfMemoryError: Metadata space
```

METASPACE

Native memory region for class data.

Grow automatically by default.

Garbage collected.

-XX:MetaspaceSize -XX:MaxMetaspaceSize

Transition to Java 8: `e/Perm/Metaspace/g`

BUT, OLEG, WAIT!

— You said this is Hipster slides, but you didn't even mention a monad!

— Sorry guys. No monads until we'll have Higher Kinded Polymorphism in Java!

THE END

BY OLEG PROPHET / HAKUTAKU.ME

Source: slides, java samples
Thank you!