Ausblick auf Java 8

Martin Plümicke

Baden-Wuerttemberg Cooperative State University
Stuttgart/Horb

25. Mai 2012

Overview

Introduction
- Closures
- Java's motivation

Java 8 features
- λ–expressions
- Functional interfaces
- Type inference
- Method references
- Default methods

Problems of the functional interface approach
- Subtyping
- Higher-order functions

Type–inference

Conclusion and Outlook

Base

- Brian Goetz: Project Lambda 2011, version 0.4.2, 2011
- Brian Goetz: Language / Library / VM co-evolution in Java SE 8, November 17, 2011
- Martin Plümicke: Brian's approach vs. Martin's approach, functional interfaces vs. function types in Java 8, Bad Honnef, May 3, 2012

History of Java type system

Version 1:

- Subtyping on classes (without parameters)
  Ex.: $\texttt{Integer} \leq^* \texttt{Object}$

History of Java type system

Version 1:

- Subtyping on classes (without parameters)
  Ex.: `Integer` $\leq^*$ `Object`

Version 5:

- Parametrized classes (type constructors)
  Ex.: `Vector<X>`
- Subtyping extension
  Ex.: `Matrix` $\leq^*$ `Vector<Vector<Integer>>`
- Wildcards (with lower and upper bounds)
  Ex.: `Vector<? extends Integer>`

History of Java type system

Version 1:

- Subtyping on classes (without parameters)
  Ex.: `Integer` $\leq^*$ `Object`

Version 5:

- Parametrized classes (type constructors)
  Ex.: `Vector<X>`
- Subtyping extension
  Ex.: `Matrix` $\leq^*$ `Vector<Vector<Integer>>`
- Wildcards (with lower and upper bounds)
  Ex.: `Vector<? extends Integer>`

Version 8:

- Closures (λ–expressions)
- Functional interfaces, but no function types

Closures, λ–expression, Nameless functions

Well-known from functional programming languages.

Example: Identity–function

```
\x -> x                    -- HASKELL

fn x => x                  -- SML

(lambda (x) x)             -- SCHEME
```

Closures, λ–expression, Nameless functions

Well-known from functional programming languages.

Example: Identity–function

```
\x -> x                    -- HASKELL

fn x => x                  -- SML

(lambda (x) x)             -- SCHEME
```

Application: `(\x -> x) 1 = 1`

Closures, λ–expression, Nameless functions

Well-known from functional programming languages.

Example: Identity–function

```
\x -> x                    -- HASKELL

fn x => x                  -- SML

(lambda (x) x)             -- SCHEME
```

Application: `(\x -> x) 1 = 1`

Function-types: `(\x -> x) :: a -> a`

Higher–order functions

Argument types and result types of functions can also be function types.

Example:

```
map :: (a -> b) -> [a] -> [b]
```

Higher–order functions

Argument types and result types of functions can also be function types.

Example:

```
map :: (a -> b) -> [a] -> [b]
```

Application:

```
map (\x -> x+1) [1,2,3,4,5] = [2,3,4,5,6]
```

Motivating Example: *External Iteration*

```
public class Student {
    String name;
    int graduationYear;
    double score; }
```

Motivating Example: *External Iteration*

```
public class Student {
    String name;
    int graduationYear;
    double score; }

List<Student> students = ...
double highestScore = 0.0;
for (Student s : students) {
  if (s.gradYear == 2012) {
    if (s.score > highestScore) {
      highestScore = s.score;
    }
  }
}
```

Motivating Example: *External Iteration*

```
public class Student {
    String name;
    int graduationYear;
    double score; }

List<Student> students = ...
double highestScore = 0.0;
for (Student s : students) {
  if (s.gradYear == 2012) {
    if (s.score > highestScore) {
      highestScore = s.score;
    }
  }
}
```

- external iteration
- serial iteration
- not thread-safe

Motivating Example: *Inner classes*

```
public class Student {
    String name;
    int graduationYear;
    double score; }

SomeCoolList<Student> students = ...
double highestScore =
  students.filter(new Predicate<Student>() {
    public boolean op(Student s) {
      return s.getGradYear() == 2011; }
  })
```

Motivating Example: *Inner classes*

```
public class Student {
    String name;
    int graduationYear;
    double score; }

SomeCoolList<Student> students = ...
double highestScore =
  students.filter(new Predicate<Student>() {
    public boolean op(Student s) {
      return s.getGradYear() == 2011; }
  }).map(new Mapper<Student,Double>() {
    public Double extract(Student s) {
      return s.getScore(); }
})
```

Motivating Example: *Inner classes*

```
public class Student {
    String name;
    int graduationYear;
    double score; }

SomeCoolList<Student> students = ...
double highestScore =
  students.filter(new Predicate<Student>() {
    public boolean op(Student s) {
      return s.getGradYear() == 2011; }
  }).map(new Mapper<Student,Double>() {
    public Double extract(Student s) {
      return s.getScore(); }
}).max();
```

Motivating Example: *Inner classes*

```
public class Student {
    String name;
    int graduationYear;
    double score; }

SomeCoolList<Student> students = ...
double highestScore =
  students.filter(new Predicate<Student>() {
    public boolean op(Student s) {
      return s.getGradYear() == 2011; }
  }).map(new Mapper<Student,Double>() {
    public Double extract(Student s) {
      return s.getScore(); }
}).max();
```

- internal iteration by inner classes
- traversal may be done in parallel
- but ugly syntax

Motivating Example: *Closures*

```
public class Student {
    String name;
    int graduationYear;
    double score; }

SomeCoolList<Student> students = ...
double highestScore =
  students.filter(Student s -> s.getGradYear() == 2011)
          .map(Student s -> s.getScore())
          .max();
```

Motivating Example: *Closures*

```
public class Student {
    String name;
    int graduationYear;
    double score; }

SomeCoolList<Student> students = ...
double highestScore =
  students.filter(Student s -> s.getGradYear() == 2011)
          .map(Student s -> s.getScore())
          .max();
```

- closures
- more readable

Java 8 features

- λ–expressions, closures
- functional interfaces
- restricted type inference
- method references
- default methods

Today, Java 7

```
Collections.sort(List<T> list, Comparator<? super T> c)

interface Comparator<T> {
    int compare(T o1, T o2)
}
```

Today, Java 7

```
Collections.sort(List<T> list, Comparator<? super T> c)

interface Comparator<T> {
    int compare(T o1, T o2)
}


Collections.sort(people,
    new Comparator<Person>() {
      public int compare(Person x, Person y) {
        return x.getLastName().compareTo(y.getLastName());
      }
    });
```

Lambda–expressions in Java 8

```
Collections.sort(people,
      new Comparator<Person>() {
        public int compare(Person x, Person y) {
          return x.getLastName().compareTo(y.getLastName());
        }
      });
```

```
Collections.sort(people,
    (Person x, Person y) ->
        x.getLastName().compareTo(y.getLastName()));
```

Types of λ–expressions in Java 8

Functional interfaces: Interfaces with a single method (SAM–types) as target types for lambda expressions.

Types of λ–expressions in Java 8

Functional interfaces: Interfaces with a single method (SAM–types) as target types for lambda expressions.

E.g.

```
interface Comparator<T>  { int compare(T x, T y); }
interface FileFilter     { boolean accept(File x); }
interface DirectoryStream.Filter<T> { boolean accept(T x); }
interface Runnable       { void run(); }
interface ActionListener { void actionPerformed(...); }
interface Callable<T>    { T call(); }
```

Types of λ–expressions in Java 8

Functional interfaces: Interfaces with a single method (SAM–types) as target types for lambda expressions.

E.g.

```
interface Comparator<T>  { int compare(T x, T y); }
interface FileFilter     { boolean accept(File x); }
interface DirectoryStream.Filter<T> { boolean accept(T x); }
interface Runnable       { void run(); }
interface ActionListener { void actionPerformed(...); }
interface Callable<T>    { T call(); }
```

but: no function types!!!

Functional interfaces as compatible target types

A lambda expression is *compatible* with a type T, if

- T is a functional interface type
- The lambda expression has the same number of parameters as T's method, and those parameters' types are the same
- Each expression returned by the lambda body is compatible with T's method's return type
- (Each exception thrown by the lambda body is allowed by T's method's throws clause)

Ambiguous target types

Target type of

```
(Person x, Person y) ->
     x.getLastName().compareTo(y.getLastName()));
```

```
Comparator<Person>
(interface Comparator<T> { int compare(T o1, T o2) })
```

Ambiguous target types

Target type of

```
(Person x, Person y) ->
     x.getLastName().compareTo(y.getLastName()));
```

```
Comparator<Person>
(interface Comparator<T> { int compare(T o1, T o2) })
```

and also

```
interface PersonComparator { int compare(Person o1, Person o2) }
```

Ambiguous target types

Target type of

```
(Person x, Person y) ->
     x.getLastName().compareTo(y.getLastName()));
```

```
Comparator<Person>
(interface Comparator<T> { int compare(T o1, T o2) })
```

and also

```
interface PersonComparator { int compare(Person o1, Person o2) }
```

but also

```
interface funny { int somename(Object o1, Object o2) }
```

Equivalence class and canonical representation

Lemma: There is an equivalence class of compatible target types for a lambda expression.

Equivalence class and canonical representation

Lemma: There is an equivalence class of compatible target types for a lambda expression.

For the equivalence class of the compatible target types of a lambda expression, there is a canonical representation

$$\texttt{Fun}N\texttt{<}R, T1, \ldots, TN\texttt{>}$$

with

```
interface FunN<R,T1, ..., TN>
      { R apply(T1 arg1 , ..., TN argN); }
```

if the type of the single method of a compatible target type is

$$(T1, \ldots, TN) \rightarrow R$$

Function-types in Java 8

The canonical representations

$$\texttt{Fun}N\texttt{<}R, T1, \ldots, TN\texttt{>}$$

can play in Java 8 the role as function-types

$$(T1, \ldots, TN) \rightarrow R$$

in functional programming languages.

Parameter type–inference, Java 8

```
Collections.sort(people,
    (Person x, Person y) ->
        x.getLastName().compareTo(y.getLastName()));
```

```
Collections.sort(people,
    (x, y) ->
        x.getLastName().compareTo(y.getLastName()));
```

Type–inference in Java 8

- Type parameter instatiation (Java 5.0):
  `id:` a → a
  `id(1)` : for a the type `Integer` is inferred.

- Diamond–operator (Java 7):
  `Vector <Integer> v = new Vector <>`

- Parameter's type–inference in lambda expressions (Java 8):

  `(T1 x1, ..., TN xN)` → `h(x1, ..., XN)`

  The types `T1, ..., TN` can be inferred:

  `(x1, ..., xN)` → `h(x1, ..., XN)`

  is a correct lambda expression in Java 8.

Hide comparing function

```
Collections.sort(people,
    (x, y) ->
        x.getLastName().compareTo(y.getLastName()));
```

```
public <T, U extends Comparable<? super U>>
  Comparator<T> comparing(Mapper<T, ? extends U> mapper)
  { ... }

interface Mapper<T,U> { public U map(T t); }

Collections.sort(people, comparing(p -> p.getLastName()));
```

Method reference, Java 8

```
Collections.sort(people, comparing(p -> p.getLastName()));
```

```
Collections.sort(people, comparing(Person::getLastName));
```

Considering *SomeCoolList*

```
SomeCoolList<Student> students = ...
double highestScore =
  students.filter(Student s -> s.getGradYear() == 2011)
          .map(Student s -> s.getScore())
          .max();
```

Extension of `List` by:

- `filter`
- `map`
- `max`

The interface iterable

```
interface List<T> implements Iterable<T>, ...  { ... }
```

The interface iterable

```
interface List<T> implements Iterable<T>, ...  { ... }

interface Iterable<T> { Iterator<T> iterator(); }
```

The interface iterable

```
interface List<T> implements Iterable<T>, ...  { ... }

interface Iterable<T> { Iterator<T> iterator(); }

interface Iterator<T> {
  boolean hasNext();
  E next();
  void remove();
}
```

Idea: Extension of Iterable

```
interface Iterable<T> {

  Iterator<T>     iterator();

  Iterable<T>     filter(Predicate<? super T> predicate);

  <U> Iterable<U> map(Mapper<? super T, ? extends U> mapper);

  Iterable<T>     sorted(Comparator<? super T> comp);

  // and more
}
```

Idea: Extension of Iterable

```
interface Iterable<T> {

  Iterator<T>     iterator();

  Iterable<T>     filter(Predicate<? super T> predicate);

  <U> Iterable<U> map(Mapper<? super T, ? extends U> mapper);

  Iterable<T>     sorted(Comparator<? super T> comp);

  // and more
}
```

Problem: old implementations of Iterable<T>

Default methods

Idea: For all new methods a default implementation is given.

Example:

```
Iterable<T> sorted(Comparator<? super T> comp)
    default { throw new noIterableObjectException(); };
```

Multiple inheritance (method resolution)

```
interface Collection<T> {
  public Collection<T> filter(Predicate<T> p) default ...;
}

interface Set<T> extends Collection<T> {
  public Set<T> filter(Predicate<T> p) default ...;
}

class D<T> implements Set<T> { ... }

class C<T> extends D<T> implements Collection<T> { ... }
```

In `C<T>`:default method from `Set<T>` or `Collection<T>`?

- `Set<T>` is more specific, so it wins over `Collection<T>`.

Multiple inheritance (ambiguity)

Example:

```
interface A {
  void m() default { System.out.println("A"); } ;
}

interface B {
  void m() default { System.out.println("B"); } ;
}

class C implements A, B {

  public void m() { A.super.m(); }  //ambiguous resolution
}
```

Default methods is also a VM feature

Old `.class`–file implementation (without realizations of new methods) can implement the new interfaces.

The VM add the default methods!

Subtyping

$$(T'_1, \ldots, T'_N) \to T_0 \leq^* (T_1, \ldots, T_N) \to T'_0, \quad \text{iff } T_i \leq^* T'_i$$

Subtyping

$$(T'_1, \ldots, T'_N) \to T_0 \leq^* (T_1, \ldots, T_N) \to T'_0, \quad \text{iff } T_i \leq^* T'_i$$

$$\texttt{Fun}N\texttt{<}T_0, T'_1, \ldots, T'_N\texttt{>} \not\leq^* \texttt{Fun}N\texttt{<}T'_0, T_1, \ldots, T_N\texttt{>}, \quad \text{for } T_i \not\leq^* T'_i$$

Subtyping

$$(T'_1, \ldots, T'_N) \to T_0 \leq^* (T_1, \ldots, T_N) \to T'_0, \quad \text{iff } T_i \leq^* T'_i$$

$$\texttt{Fun}N\texttt{<}T_0, T'_1, \ldots, T'_N\texttt{>} \not\leq^* \texttt{Fun}N\texttt{<}T'_0, T_1, \ldots, T_N\texttt{>}, \quad \text{for } T_i \not\leq^* T'_i$$

Example:

$$\texttt{Integer} \to \texttt{Integer} \quad \leq^* \quad \texttt{Integer} \to \texttt{Object}$$

Subtyping

$$(T'_1, \ldots, T'_N) \to T_0 \leq^* (T_1, \ldots, T_N) \to T'_0, \quad \text{iff } T_i \leq^* T'_i$$

$$\texttt{Fun}N\texttt{<}T_0, T'_1, \ldots, T'_N\texttt{>} \not\leq^* \texttt{Fun}N\texttt{<}T'_0, T_1, \ldots, T_N\texttt{>}, \quad \text{for } T_i \leq^* T'_i$$

Example:

$$\texttt{Integer} \to \texttt{Integer} \quad \leq^* \quad \texttt{Integer} \to \texttt{Object}$$

but

```
Fun1<Integer,Integer> idIntInt = (Integer x) -> x
Fun1<Object,Integer> idIntObj = idIntInt
```

is **wrong!**, as

$$\texttt{Fun1<Integer,Integer>} \quad \not\leq^* \quad \texttt{Fun1<Object,Integer>}$$

Subtyping with wildcards I

```
Fun1<Integer,Integer> idIntInt = (Integer x) -> x;
Fun1<? extends Object, Integer> idExtSup = idIntInt;
```

as

`Fun1<Integer,Integer>` $\leq^*$ `Fun1<? extends Object, Integer>`

Subtyping with wildcards I

```
Fun1<Integer,Integer> idIntInt = (Integer x) -> x;
Fun1<? extends Object, Integer> idExtSup = idIntInt;
```

as

Fun1<Integer,Integer> $\leq^*$ Fun1<? extends Object, Integer>

```
Fun1<Integer, Number> idNumInt = (x) -> (Integer)x;
Fun1<? extends Object, ? super Integer> idExtSup2 = idNumInt;
```

as

Fun1<Integer, Number> $\leq^*$ Fun1<? extends Object, ? super Integer>

Subtyping with wildcards II

Example:

```
Object m(Integer x, Fun1<? extends Object, ? super Integer> f) {
  return f.apply(x);
}
```

Subtyping with wildcards II

Example:

```
Object m(Integer x, Fun1<? extends Object, ? super Integer> f) {
  return f.apply(x);
}

Fun1<Integer,Integer> idIntInt = (Integer x) -> (Integer)x;
Object x1 = m(2, idIntInt);
```

Subtyping with wildcards II

Example:

```
Object m(Integer x, Fun1<? extends Object, ? super Integer> f) {
  return f.apply(x);
}

Fun1<Integer,Integer> idIntInt = (Integer x) -> (Integer)x;
Object x1 = m(2, idIntInt);

Fun1<Object,Integer> idIntObj = (Integer x) -> (Object)x;
Object x2 = m(2, idIntObj);
```

Subtyping with wildcards II

Example:

```
Object m(Integer x, Fun1<? extends Object, ? super Integer> f) {
  return f.apply(x);
}

Fun1<Integer,Integer> idIntInt = (Integer x) -> (Integer)x;
Object x1 = m(2, idIntInt);

Fun1<Object,Integer> idIntObj = (Integer x) -> (Object)x;
Object x2 = m(2, idIntObj);

Fun1<Integer, Number> idNumInt = (Number x) -> (Integer)x;
Object x3 = m(2, idNumInt);
```

Function-application

Functional programming languages:

```
(\(x1, ..., xN) -> h(x1, ..., xN)) (arg1, ..., argN)
```

Function-application

Functional programming languages:

```
(\(x1, ..., xN) -> h(x1, ..., xN)) (arg1, ..., argN)
```

Goal in Java:

```
((x1,...,  xN) -> h(x1,..., xN)).(arg1,...,argN);
```

Function-application

Functional programming languages:

```
(\(x1, ..., xN) -> h(x1, ..., xN)) (arg1, ..., argN)
```

Goal in Java:

```
((x1,...,  xN) -> h(x1,..., xN)).(arg1,...,argN);
```

in Java 8:

```
((x1,..., xN) -> h(x1,..., xN)).apply(arg1....,argN);
```

wrong!, no lambda expression is allowed as receiver

Function-application

Functional programming languages:

```
(\(x1, ..., xN) -> h(x1, ..., xN)) (arg1, ..., argN)
```

Goal in Java:

```
((x1,...,  xN) -> h(x1,..., xN)).(arg1,...,argN);
```

in Java 8:

```
((x1,..., xN) -> h(x1,..., xN)).apply(arg1....,argN);
```

wrong!, no lambda expression is allowed as receiver

```
((FunN<T0, T1,..., TN> )
 (x1,..., xN) -> h(x1,..., xN)).apply(arg1,...,argN);
```

ok!, as there is cast–expression

Currying $f : T_1 \longrightarrow T_2 \longrightarrow \ldots \longrightarrow T_N \longrightarrow T_0$

Application:

in Java 8:

```
((Fun1<Fun1<Fun1<...Fun1<T0,TN>,...>,T2>,T1> )
 (x1) -> (x2) ->...-> (xN) -> h(x1,...,XN))
 .apply(a1).apply(a2).....apply(aN))
```

Currying $f : T_1 \longrightarrow T_2 \longrightarrow \ldots \longrightarrow T_N \longrightarrow T_0$

Application:

in Java 8:

```
((Fun1<Fun1<Fun1<...Fun1<T0,TN>,...>,T2>,T1> )
 (x1)->(x2)->...->(xN)->h(x1,...,XN))
 .apply(a1).apply(a2).....apply(aN))
```

Goal in Java:

```
((x1)->(x2)->...->(xN)->h(x1,...,XN)).a1.a2....aN
```

Currying $f : T_1 \longrightarrow T_2 \longrightarrow \ldots \longrightarrow T_N \longrightarrow T_0$

Application:

in Java 8:

```
((Fun1<Fun1<Fun1<...Fun1<T0,TN>,...>,T2>,T1> )
 (x1)->(x2)->...->(xN)->h(x1,...,XN))
 .apply(a1).apply(a2).....apply(aN))
```

Goal in Java:

```
((x1)->(x2)->...->(xN)->h(x1,...,XN)).a1.a2....aN
```

Extension of type–inference: Types of complete λ–expressions must be inferable.

Type–inference for Java with *function–types* [Plümicke, PPPJ 2011]

- Type-inference of parameter's type in lambda expressions (Java 8)
- Type-inference of complete lambda expressions
- Type-inference of local variables

Results: Well-typings [Fuh/Mishra 88]

Type–inference for Java with *function–types* [Plümicke, PPPJ 2011]

- Type-inference of parameter's type in lambda expressions (Java 8)
- Type-inference of complete lambda expressions
- Type-inference of local variables

Results: Well-typings [Fuh/Mishra 88]

Example

```
class Example {

  fac = (n) -> {
    ret = 1;
    for(i=2; i <= n; i++) ret = ret * i;
    return ret;
  }}
```

Type–inference for Java with *function–types* [Plümicke, PPPJ 2011]

- Type-inference of parameter's type in lambda expressions (Java 8)
- Type-inference of complete lambda expressions
- Type-inference of local variables

Results: Well-typings [Fuh/Mishra 88]

Example

```
class Example {

  #Integer(Integer) fac = (Integer n) -> {
    Integer ret = 1;
    for (Integer i=2; i <= n; i++) ret = ret * i;
    return ret;
  }}
```

Type-inference for Java 8

- **Type–inference for functional interfaces**
  Extension of type–inference for Java 8 by type–unification [Pluemicke 2007]
- **Type–inference for methods**
  Introduction of overloading and overriding

Conclusion and Outlook

Conclusion

- Extensions of Java 8
  (λ–expressions/closures, functional interfaces, restricted type inference, method references, default methods)
- Function applications
- Type–inference

Conclusion and Outlook

Conclusion

- Extensions of Java 8
  (λ–expressions/closures, functional interfaces, restricted type inference, method references, default methods)
- Function applications
- Type–inference

Outlook

- Discussion, if function-types are better
- Type–inference for Java with function-types by type–unification

No well-typings, but type constraints on type variables

Java 9 or later

```
List<Student> students = new ArrayList<>(...);
...
double highestScore =
students.parallel()
        .filter(s -> s.getGradYear() == 2011)
        .map(s -> s.getScore())
        .max();
```

- Java 7 provides general-purpose Fork/Join
- Fork-join parallelism is powerful and efficient
- But, want to protect users from having to actually write fork-join code

Project Lambda

Project Web-Site:

`http://openjdk.java.net/projects/lambda/`

Binary code (Windows, Linux, Mac);

`http://jdk8.java.net/lambda/`