Brian's approach vs. Martin's approach
Functional Interfaces vs. function types in Java 8

Martin Plümicke

Baden-Wuerttemberg Cooperative State University
Stuttgart/Horb

3. Mai 2012

Overview

Two approaches

- Brian's approach: Java 8 [Project Lambda 2011 version 0.4.2, Brian Goetz 2011]
  - closures
  - functional interfaces
  - restricted type inference
  - method references
  - default methods

Two approaches

- Brian's approach: Java 8 [Project Lambda 2011 version 0.4.2, Brian Goetz 2011]
  - closures
  - functional interfaces
  - restricted type inference
  - method references
  - default methods
- Martin's approach: Java$_\lambda$ [Project Lambda 2010, version 0.1.5; Martin Plümicke, PPPJ 2011]
  - closures
  - function types
  - higher-order functions
  - complete type inference

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()));
```

Typen von Lambda–expressions in Java 8

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

Typen von Lambda–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(); }
```

Typen von Lambda–expressions in Java$_\lambda$

Function-types: $(\tau_1, \ldots, \tau_n) \longrightarrow \tau$ (**syntax:** $\#\,\tau\,(\tau_1, \ldots, \tau_n)$)

E.g.

Lambda–expression:

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

has the type:

$$(\texttt{Person}, \texttt{Person}) \longrightarrow \texttt{int}$$

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()));
```

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));
```

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

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

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

Canonical representation

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$_\lambda$

#T0(T1, ... , TN) op;

- The function op has the type (T1, ... , TN) $\longrightarrow$ T0.

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

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

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

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

$$\texttt{Fun1<Integer,Integer>} \quad \leq^* \quad \texttt{Fun1<? extends Object, Integer>}$$

Subtyping with wildcards I

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

as

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

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

as

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

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

in Java$_\lambda$ (with type–inference):

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

Function-application

in Java$_\lambda$ (with type–inference):

```
((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

in Java$_\lambda$ (with type–inference):

```
((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 $f : T_1 \to T_2 \to \ldots \to T_N \to T_0$

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 $f : T_1 \to T_2 \to \ldots \to T_N \to T_0$

Application:

in Java 8:

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

in Java$_\lambda$: (with type–inference):

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

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.

Type–inference in Java$_\lambda$

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

Results: Well-typings [Fuh/Mishra 88]

Type–inference in Java$_\lambda$

- Type-inference of parameter's type in lambda expressions, as in 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 in Java$_\lambda$

- Type-inference of parameter's type in lambda expressions, as in 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
- Function-types vs. functional interfaces
- Function applications
- Type–inference

Conclusion and Outlook

Conclusion

- Extensions of Java 8
- Function-types vs. functional interfaces
- Function applications
- Type–inference

Outlook

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

No well-typings, but type constraints on type variables