Java 8

Lambda Expressions

Angelika Langer
Training/Consulting

http://www.AngelikaLanger.com
objective

- explain the new language feature of lambda expressions
- what is the intent?
- which problem do they solve?
- what are the syntax elements?
- how will they be used in the JDK?
speaker's relationship to topic

• independent trainer / consultant / author
  – teaching C++ and Java for 15+ years
  – curriculum of a couple of challenging courses
  – co-author of "Effective Java" column
  – author of Java Generics FAQ online
  – JCP member and Java champion since 2005
agenda

• **introduction**
• functional interfaces
• lambda expressions (the details)
• method references
• extension methods
• ‘lambdafication’ of the JDK
lambda expressions in Java

• *lambda expressions*
  – aka *lamdas*; formerly known as *closures*

• concept from functional programming
  – “anonymous method” / “code-as-data”
    › ‘ad hoc’ implementation of functionality
    › pass functionality around (parameter, return value)

  – similar to (anonymous) inner classes
    › advantage of lambda expressions: concise syntax + less code
    › “more functional”
history

• 2006 – 2009
  – several proposals for ‘closures in Java’
  – no convergence; none fully supported by Sun / Oracle

• since 2010
  – OpenJDK Project Lambda; tech lead Brian Goetz
  – JSR 335 (Nov. 2010)
    "Lambda Expressions for the Java Programming Language"
  – JEP 126 (Nov. 2011)
    "Lambda Expressions and Virtual Extension Methods"
Oracle’s design guideline

- aid usage of libraries that …
  - make use of parallelization on multi core platforms
  - special focus: JDK

- rules out
  - which features are relevant?
  - how complex can they be?

- general guideline: "as simple as possible"
  - several (previously discussed) features were dropped
  - e.g. function types, exception transparency, …
agenda

- introduction
- functional interfaces
- lambda expressions (the details)
- method references
- extension methods
- ‘lambdafication’ of the JDK
key goal

• support JDK abstractions that …
  – make use of parallelization on multi core platforms

• collections shall have parallel bulk operations
  – based on fork-join-framework (Java 7)
  – execute functionality on a collection in parallel
    › i.e. access multiple elements simultaneously
  – specified as: JEP 107
    › details later
today

```java
private static void checkBalance(List<Account> accList) {
    for (Account a : accList)
        if (a.balance() < threshold) a.alert();
}
```

- **new for-loop style**
  - actually an external `Iterator` object is used:

```java
Iterator iter = accList.iterator();
while (iter.hasNext()) {
    Account a = iter.next();
    if (a.balance() < threshold) a.alert();
}
```

- **code is inherently serial**
  - traversal logic is fixed
  - iterate from beginning to end
Stream forEach() - definition

- `forEach()`'s iteration is not inherently serial
  - traversal order is defined by `forEach()`'s implementation
  - burden of parallelization is put on the library developer
    - not on the library user

```java
public interface Stream<T> {
    ...
    void forEach(Block<? super T> sink);  // ...}
```

```java
public interface Block<A> {
    void apply(A a);
    ...
}
```
Stream.forEach() - example

Stream<Account> pAccs = accList.parallel();

// with anonymous inner class
pAccs.forEach(new Block<Account>() {
    void apply(Account a) {
        if (a.balance() < threshold) a.alert();
    }
});

// with lambda expression
pAccs.forEach((Account a) -> {
    if (a.balance() < threshold) a.alert();
});

- lambda expression
  - less code (overhead)
  - only actual functionality
    - easier to read
**lambda expression** a Block<Account>?

Block<Account> block =

(Account a) -> {
    if (a.balance() < threshold) a.alert();
};

- right side: lambda expression
- intuitively
  - ‘something functional’
    - takes an Account
    - returns nothing (void)
    - throws no checked exception
- nothing in terms of the Java type system
  - just some code / functionality / implementation
**functional interface = target type of a lambda**

```java
interface Block<A> { public void apply(A a); }

Block<Account> pAccs =
    (Account a) -> { if (a.balance() < threshold) a.alert(); };
```

- **lambdas are converted to functional interfaces**
  - function interface ≈ interface with one method
  - parameter type(s), return type, checked exception(s) must match
  - functional interface’s name + method name are irrelevant

- **conversion requires type inference**
  - lambdas may only appear where target type can be inferred from enclosing context
  - e.g. variable declaration, assignment, method/constructor arguments, return statements, cast expression, ...
idea behind functional interfaces

- interfaces with one method have been the ‘most functional things’ in Java already:

```
interface Runnable { void run(); }
interface Callable<T> { T call(); }
interface Comparator<T> { boolean compare(T x, T y); }
...
```

- "as simple as possible"
- reuse existing interface types as target types for lambda expressions
lambda expressions & functional interfaces

• functional interfaces

```java
interface Block<A> { void apply(A a); }
interface MyInterface { void doWithAccount(Account a); }
```

```java
Block<Account> block = 
(Account a) -> { if (a.balance() < threshold) a.alert(); };
MyInterface mi = 
(Account a) -> { if (a.balance() < threshold) a.alert(); };
mi = block;  // error: types are not compatible
```

• conversions

```java
Block<Account> block = 
(Account a) -> { if (a.balance() < threshold) a.alert(); };
MyInterface mi = 
(Account a) -> { if (a.balance() < threshold) a.alert(); };
mi = block;  // error: types are not compatible
```

• problems

```java
Object o1 = 
(Account a) -> { if (a.balance() < threshold) a.alert(); };
Object o2 = (Block<Account>)
(Account a) -> { if (a.balance() < threshold) a.alert(); };
```

error: cannot infer target type
evaluation

- **lambda expression**
  - easy and convenient way to implement ad-hoc functionality

- **functional interfaces**
  - serve as target types for lambda expressions
  - integrate lambda expressions into Java type system

- **advantages**
  - simple: no new elements in Java type system
    - good for language designers and users
  - built-in backward compatibility
    - e.g. can provide a lambda where a `Runnable` (JDK 1.0) is needed
evaluation (cont.)

• down-side
  – must define `Block<A>` to describe parameter type:

    ```java
    public void forEach(Block<? super T> sink) ...
    public interface Block<A> { void apply(A A); }
    ```

  – code overhead, no explicit function type: `<T> -> void`

• justification: overhead is acceptable
  – explicit function types add many more complications
  – "we (the library providers) do it for you (the library users)"

  – may be added later
    · JSR 335 (lambda spec) mentions function types as potential future enhancement
agenda

- introduction
- functional interfaces
- **lambda expressions (the details)**
- method references
- extension methods
- ‘lambdafication’ of the JDK
lambda expression syntax

• since September 2011:

```java
(Account a) -> { if (a.balance() < threshold) a.alert(); }
```

• previously:

```java
# { Account a -> if (a.balance() < threshold) a.alert(); }
```

• syntax: C#, with ‘–>’ instead of ‘=>’
  – proven concept
  – quite similar to Scala’s closure syntax, too

  – ‘–>’ instead of ‘=>’ to avoid *dueling arrows*

```java
foo (x => x.size <= 10);
```
formal description

\[
\text{lambda} = \text{ArgList} \rightarrow \text{Body}
\]

\[
\text{ArgList} = \text{Identifier} \\
\mid \left( \text{ Identifier } [ \ , \ , \text{ Identifier } ]^{*} \right) \\
\mid \left( \text{ Type Identifier } [ \ , \ , \text{ Type Identifier } ]^{*} \right)
\]

\[
\text{Body} = \text{Expression} \\
\mid \left\{ \text{ Statement } ; ; \right\}^{+}
\]

- options related to
  - argument list, and
  - body
argument list, pt. 1

• if possible, compiler infers parameter type
  – inference based on target type, not lambda body

• if not possible, parameter type must be specified
  – parameter type can always be specified

• multiple parameters
  – all parameters either declared or inferred (no mix possible)
argument list, pt. 2

- omission of parentheses in case of one argument without type identifier possible

- examples:

```java
a -> { if (a.balance() < threshold) a.alert(); }

(int x) -> { return x+1; }
x -> { return x+1; }  // omit parentheses

(int x, int y) -> { return x+y; }
(x, y) -> { return x+y; }  // can’t omit parentheses

// no special nilary syntax
() -> { System.out.println(“I am a Runnable”); }
```
Body = Expression | "{" [ Statement ; ]+ "}"
return, pt. 1

- return type is always inferred
  - i.e. cannot be specified explicitly
  - you might consider to cast the return value

```
( )        -> { return 21; }       // returns int
(Account a) -> { return a; }       // returns Account
( )        -> { return (long)21; }  // returns long
```
return, pt. 2

- no `return` with single expression
  - use of `return` is an error

- `return` used with list of statements
  - when using multiple `return`s:
    programmer responsible, that the return type can be inferred

```
() -> { return 21; }
() -> return 21
() -> 21
```

error !!!!
• local variable capture
  – important feature
  – similar to anonymous inner classes
    ‣ but no explicit `final`
    ‣ but still only read access

```java
int i = 16;
Runnable r = () -> { System.out.println("i is: " + i); };```

© Copyright 2003-2012 by Angelika Langer & Klaus Kreft. All Rights Reserved.
http://www.AngelikaLanger.com/
last update: 12/6/2013, 10:24

Lambda Expressions in Java (27)
local variable capture (cont.)

• effectively final variables
  – same as with anonymous inner classes, but
    › you do not have to use the final identifier explicitly

```java
int i = 16;
Runnable r = () -> { System.out.println("i is: " + i); };
i++; // error because
```
effectively final

- local variable capture not much different from inner classes

- but caused a lot of discussion
  - *I want write access!*

- a look at the details
  - shows the limitations
reason for effectively final

```java
int i = 0;

Runnable r = () -> { for (int j=0; j < 32; j++) i = j; };

// start Runnable r in another thread
...

while (i <= 16) /* NOP */;

System.out.println("i is now greater than 16");
```

- problem: unsynchronized concurrent access
- no guarantees from the memory model
but the effectively final does not prevent ...

... all evil in the world

• consider a mutable object referred to by an effectively final reference

```java
int[] ia = new int[1];

Runnable r = () -> {
    for (int j = 0; j < 32; j++)
        ia[0] = j;
};

// start Runnable r in another thread
...

while (ia[0] <= 16) /* NOP */;

System.out.println("ia[0] is now greater than 16");
```
I want write access! – idioms to come?

```java
File myDir = ....
int[] ia = new int[1];

File[] fs = myDir.listFiles( f -> {
    if (f.isFile()) {
        n = f.getName();
        if (n.lastIndexOf(".exe") == n.length()-4)
            ia[0]++;
        return true;
    }
    return false;
};);

System.out.println("contains "+fs.size+" files, "+ia[0]+" are exe-files");
```

• no problem, if everything is executed sequentially
no transparent parallelism!

```java
int[] ia = new int[1];
pAccs.forEach( (Account a) -> {
    if (a.balance() < threshold) {
        a.alert();
        ia[0] ++;
    }
});
System.out.println(ia[0] + " alerts !!!");
```

- need to know whether …
  - methods that take lambda uses multiple threads or not
  - `Stream.forEach()` VS. `File.list()`

- currently not expressed in Java syntax
  - JavaDoc, comment, …
lambda body lexically scoped, pt. 1

- lambda body scoped in enclosing method
- effect on local variables:
  - capture works as shown before
  - no shadowing of lexical scope

```java
int i = 16;
Runnable r = () -> { int i = 0;
    System.out.println("i is: " + i);
};
```

error

```java
final int i = 16;
Runnable r = new Runnable() {
    public void run() { int i = 0;
        System.out.println("i is: " + i);
    }
};
```

fine

- different from inner classes
  - inner class body is a scope of its own
lambda body lexically scoped, pt. 2

- `this` refers to the enclosing object, not the lambda
  - due to lexical scope, unlike with inner classes

```java
public class MyClass {
    private int i = 100;

    public void foo() {
        Runnable r = () -> {System.out.println("i is: " + this.i);};
    }
}
```

```java
public class MyClass {
    private int i = 100;

    public void foo() {
        Runnable r = new Runnable() {
            private int i = 200;
            public void run() {System.out.println("i is: " + this.i);}
        };
    }
}
```
lambdas vs. inner classes - differences

- **local variable capture:**
  - implicitly final vs. explicitly final

- **different scoping:**
  - `this` means different things

- **verbosity:**
  - concise lambda syntax vs. inner classes' syntax overhead

- **performance:**
  - lambdas slightly faster (use `MethodHandle` from JSR 292 ("invokedynamic"))

- **bottom line:**
  - lambdas better than inner classes for functional types

- **but what if you add a second method to a functional interface**
  - and turn it into a regular non-functional type ??
agenda

• introduction
• functional interfaces
• lambda expressions (the details)
• method references
• extension methods
• ‘lambdafication’ of the JDK
an example

- want to sort a collection of Person objects
  - using the JDK's new function-style bulk operations and
  - a method from class Person for the sorting order

element type Person

class Person {
    private final String name;
    private final int age;
    ...
    public static int compareByName(Person a, Person b) { ... }
}
example (cont.)

- Stream\<T\> has a `sorted()` method

  ```java
  Stream\<T\> sorted(Comparator<? super T> comp)
  ```

- interface `Comparator` is a functional interface

  ```java
  public interface Comparator\<T\> {
    int compare(T o1, T o2);
    boolean equals(Object obj);
  }
  ```

  inherited from `Object`

- sort a collection/array of Persons

  ```java
  Stream\<Person\> psp = Arrays.parallel(personArray);
  ...
  psp.sorted((Person a, Person b) -> Person.compareByName(a, b));
  ```
example (cont.)

• used a wrapper that invokes `compareByName()`

```java
psp.sorted((Person a, Person b) -> Person.compareByName(a, b));
```

• specify `compareByName()` directly *(method reference)*

```java
psp.sorted(Person::compareByName);
```

- reuse existing implementation
- less code

• syntax not final, but very likely: “::”
idea ...

... behind method references

• take an existing method from some class, and make it the implementation of a functional interface
  – similar to lambda expressions

• need context that allows conversion to a target type
  – similar to lambda expressions

• method handles are included in JSR 335
agenda

• introduction
• functional interfaces
• lambda expressions (the details)
• method references
• extension methods
• ‘lambdafication’ of the JDK
problem

• no (good) support for interface evolution in Java today

• interface evolution
  = add a new method to an existing interface
  – problem: all existing implementations of the interface break

• concrete situation
  – extend existing collection interfaces for functional programming
  – e.g. to interface `java.util.Collection<T>` add method:

```java
void forEach(Block<? super T> block)
```
solution

• extension method

  – add a new method to an interface together with a default implementation

  – implementations of the interface are free to override it, but don’t have to
example

from package java.util:

```java
public interface Collection<T>
    extends ...
{
    ...
    everything as before ...

    public void forEach(Block<? super T> block)
        default {
        for (T each : this)
            block.apply(each);
    }
}

• no additional state / no instance fields
• implementation based on the functionality of the other methods +
  the additional parameter(s) from the new method
```
extension methods are included in JSR 335

name:
- also know as: defender methods and default methods
- in JSR 335 called virtual extension methods
  - as opposed to C#'s (non-virtual) extension methods
    (which cannot be overridden)
extensions methods vs. OO concepts

- Java interfaces are not really interfaces anymore
- they (can) provide implementation
- dilutes the "interface" concept somewhat
extensions methods vs. OO concepts

• Java provides multiple inheritance of functionality now

\[
\text{class A extends B implements I, J, K, L {}
}\]

  - A inherits functionality from B
  - A might inherit functionality from I, J, K, L
    because these interfaces might provide extensions methods

• is it a problem? - NO!
  - relatively safe, no additional state inherited from interfaces
language evolution

C++:
• multiple inheritance of functionality
  – considered dangerous

classic Java:
• single inheritance of functionality + multiple inheritance only for interfaces
  – problem: interface evolution => where to provide new functionality?

new languages:
• mixins (Ruby) / traits (Scala)
  – to solve the problem

Java 8:
• extension methods
  – fit into existing language
  – not too different from ‘stateless’ traits in Scala
    (but without linearization to resolve defaults)
problem with multiple inheritance

the diamond:

• how is A’s state inherited to D?
  – once, or
  – twice (via B and via C)

• there is no ‘right’ answer to this question
  – C++ gives you the option: virtual / non-virtual inheritance
  – makes it more complicated
Java 8

the diamond:

- **A** can only be an *interface* (not a *class*)
  - can have an implementation, but
  - no state (no instance fields)

- no state means no (diamond) problem!
  - no issue regarding "how many A parts does D have?"
still conflicts – ambiguity #1

- inherit the **same method** from a **class** and an **interface**
  - extends dominates implements
  - sub-class inherits super-class’s method (not interface’s method)

```java
l   foo() default { ... }
    
Cl  foo() { ... }
    
C2  inherits foo() from Cl
```
ambiguity #2

- inherit the **same method** from different **interfaces**
  - sub-interfaces shadow super-interfaces
  - if the interfaces are unrelated \( \rightarrow \) no default at all
    \( \rightarrow \) results in a compile error

```
I1 foo() default { ... }
I2 foo() default { ... }
C inherits foo() from I2
```

```
I1 foo() default { ... }
I2 foo() default { ... }
C
```

**compile-time error!**
ambiguity #2 (cont.)

- can address ambiguity explicitly when implementing class `C`

```java
class C implements I1, I2 {
    public void foo() { I1.super.foo(); }
}
```

- new syntax to qualify the super-interface
implementation techniques in collection framework

- until now:
  - interfaces for types
  - skeletal behavior with abstract classes

<table>
<thead>
<tr>
<th>interfaces</th>
<th>abstract classes</th>
<th>concrete classes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Iterable&lt;E&gt;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Collection&lt;E&gt;</td>
<td>AbstractCollection&lt;E&gt;</td>
<td></td>
</tr>
<tr>
<td>List&lt;E&gt;</td>
<td>AbstractList&lt;E&gt;</td>
<td>ArrayList&lt;E&gt;</td>
</tr>
</tbody>
</table>
beyond interface evolution

• interface evolution is primary motivation, but extension methods are useful in themselves

• approach:
  – define interface as always
  – provide default implementation for those methods that …
    › don’t need state, but
    › can be based on functionality of other (really abstract) methods
  – provide implementation of (really abstract) methods
    › by abstract classes (if functionality can be factored out)
    › by concrete classes
extension methods and retrofits

- JDK 1.0 introduced `Enumeration`
- JDK 1.2 replaced it with `Iterator`
- conceivable in Java 8

```java
interface Enumeration<E> extends Iterator<E> {
    boolean hasMoreElements();
    E nextElement();

    boolean hasNext() default { return hasMoreElements(); }
    E next() default { return nextElement(); }
    void remove() default {
        throw new UnsupportedOperationException();
    }
}
```
agenda

• introduction
• functional interfaces
• lambda expressions (the details)
• method references
• extension methods
• ‘lambdafication’ of the JDK
JEP 107: Bulk Data Operations for Collections

- JEP = JDK enhancement proposal, for Java 8
- also know as: "for-each/filter/map/reduce for Java"
  - for-each
    apply a certain functionality to each element of the collection
    
    ```java
    accountCol.forEach(a -> a.addInterest());
    ```
  - filter
    build a new collection that is the result of a filter applied to each element in the original collection
    
    ```java
    Stream<Account> result =
    accountCol.filter(a -> a.balance() > 1000000 ? true : false);
    ```
– **map**
  build a new collection, where each element is the result of a mapping from an element of the original collection

```
Stream<Integer> result = accountCol.map(a -> a.balance());
```

– **reduce**
  produce a single result from all elements of the collection

```
accountCol.map(a -> a.balance())
  .reduce(new Integer(0),
      (b1, b2) -> new Integer(b1.intValue()+b2.intValue()));
```
... additional methods

- `sorted()`, `anyMatch()`, `findFirst()`, ...
  - see JavaDoc for yourself
serial vs. parallel & eager vs. lazy

• provides
  – serial processing, i.e. performed by single thread
  – parallel processing, i.e. using multiple parallel threads

• provides
  – eager processing, i.e. produce a result or side effect
  – lazy processing, i.e. creates a stream
eager vs. lazy - example

- consider a sequence of operations
  - filter with a predicate, map to long value, and apply printing

\[
\text{myCol.filter((Account a) -> a.balance() > 1000000)}
  \text{.map((Account a) -> a.balance())}
  \text{.forEach((long l) -> System.out.format("%d\t", l));}
\]

- eager
  - each operation is executed when it is applied

- lazy
  - execute everything after the last operation has been applied
  - optimize this execution
    - e.g. call `a.balance()` only once for each account and print it directly without any intermediate collection of balances
**streams**

- **provides**
  - **eager** processing, i.e. *produce a result or side effect*
  - **lazy** processing, i.e. *creates a stream*

- **stream**
  - not a storage of values
    - i.e. no collection
    - view/adaptor of a data source (collection, array, …)
  - (mostly) functional
    - does not alter the underlying data source
    - produces a result (or side effect)
prototype available of implementation

• e.g. `java.lang.Stream<T>`
  – extended with foreach/filter/map/reduce operations

• design/functionality shown
  – based on the current OpenJDK implementation

• might change until Java 8 is final (September 2013)
wrap-up

• lambda expressions
  – new functional elements for Java
  – similarities with anonymous inner classes
    › advantages: less code, ‘more functional’, faster

• additionally in JSR 335 / JEP 126
  – method handles
  – extension methods

• JDK changes
  – JEP 107: for-each, filter, map, reduce for collections and arrays
  – JEP 109: additional smaller changes
resources

• Oracle's Java Tutorial: Section on "Lambda Expressions"
  http://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html

• JSR 335 "Project Lambda" - The official OpenJDK project page
  http://openjdk.java.net/projects/lambda/

  http://cr.openjdk.java.net/~briangoetz/lambda/lambda-state-4.html

• Angelika Langer: Lambda/Streams Tutorial & Reference
  http://www.angelikalanger.com/Lambdas/Lambdas.html

• Maurizio Cimadamore: Lambda expressions in Java - a compiler writer's perspective, JAX 2012
authors

Angelika Langer
Training & Consulting

Klaus Kreft
Performance Expert, Germany

http://www.AngelikaLanger.com