Java 8

Functional Programming with Lambdas

Angelika Langer
Training/Consulting

http://www.AngelikaLanger.com
objective

• learn about lambda expressions in Java
• know the syntax elements
• understand typical uses
speaker's relationship to topic

- independent trainer / consultant / author
  - teaching C++ and Java for >15 years
  - curriculum of a couple of challenging courses
  - JCP observer and Java champion since 2005
  - co-author of "Effective Java" column
  - author of Java Generics FAQ online
  - author of Lambda Tutorial & Reference
agenda

• lambda expression
• functional patterns
lambda expressions in Java

• lambda expressions
  ‣ formerly known as closures

• concept from functional programming languages
  – anonymous method
    ‣ “ad hoc” implementation of functionality
  – code-as-data
    ‣ pass functionality around (as parameter or return value)
  – superior to (anonymous) inner classes
    ‣ concise syntax + less code + more readable + “more functional”
key goal

• **build better (JDK) libraries**
  – e.g. for easy parallelization on multi core platforms

• **collections shall have parallel bulk operations**
  – based on fork-join-framework
  – execute functionality on a collection in parallel

• **separation between "what to do" & "how to do"**
  – user  =>  *what* functionality to apply
  – library  =>  *how* to apply functionality
    (parallel/sequential, lazy/eager, out-of-order)
private static void checkBalance(List<Account> accList) {
    for (Account a : accList)
        if (a.balance() < threshold) a.alert();
}

- **for-loop uses an iterator**:

```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 not inherently serial
  - traversal order defined by forEach()’s implementation
  - burden of parallelization put on library developer

```java
public interface Stream<T> {
    ...  
    void forEach(Consumer<? super T> consumer);
    ...
}
```

```java
public interface Consumer<T> {
    void accept(T t)
    ...
}
```
Stream.forEach() - example

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

// with anonymous inner class
pAccs.forEach( new Consumer<Account>() {
    void accept(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
agenda

• lambda expression
  – functional interfaces
  – lambda expressions (syntax)
  – method references

• functional patterns
is a lambda an object?

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

• right side: lambda expression

• intuitively
  – a lambda is "something functional"
    ‣ takes an Account
    ‣ returns nothing (void)
    ‣ throws no checked exception
    ‣ has an implementation {body}
  – kind of a function type: (Account) -> void

• Java's type system does not have function types
functional interface = target type of a lambda

```java
interface Consumer<T> { public void accept(T a); }

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

- lambdas are converted to *functional interfaces*
  - function interface := interface with one abstract method

- compiler infers target type
  - relevant: parameter type(s), return type, checked exception(s)
  - irrelevant: interface name + method name

- lambdas need a *type inference* context
  - e.g. assignment, method/constructor arguments, return statements, cast expression, …
### lambda expressions & functional interfaces

- **functional interfaces**

```java
interface Consumer<T> { void accept(T a); }
interface MyInterface { void doWithAccount(Account a); }
```

- **conversions**

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

---

© Copyright 2003-2013 by Angelika Langer & Klaus Kreft. All Rights Reserved.
http://www.AngelikaLanger.com/
last update: 11/4/2013, 11:16
agenda

- lambda expression
  - functional interfaces
  - lambda expressions (syntax)
  - method references

- functional patterns
formal description

\[
\begin{align*}
\text{lambda} &= \text{ArgList} \rightarrow \text{Body} \\
\text{ArgList} &= \text{Identifier} \\
&\quad | ("\text{Identifier} [",\text{Identifier}]*)" \\
&\quad | ("\text{Type}\text{Identifier} [",\text{Type}\text{Identifier}]*)" \\
\text{Body} &= \text{Expression} \\
&\quad | "{\text{Statement};\}+"}
\end{align*}
\]
syntax samples

argument list

(i nt  x,  i nt  y)  -&gt; {  r etur n  x+y;  }
(x, y) -&gt; { return x+y; }
x  -&gt; { return x+1; }

()  -&gt; {  S yste m  o ut .  p rin t l n( "I  a m  a  R unnabl e" );  }

body

// single statement or list of statements
a  -&gt; {
   i f  (a. bal an ce( )  &lt;  t hres h ol d)  a. a l e r t( );
}

// single expression
a  -&gt; (a. bal an ce( )  &lt;  t hres h ol d)  ?  a. a l e r t( )  :  a. ok a y() 

return type (is always inferred)

(Account a)  -&gt; {  r etur n a;  }                           // returns Account
()  -&gt;  5                                                // returns int
local variable capture

- local variable capture
  - similar to anonymous inner classes
- no explicit `final` required
  - implicitly `final`, i.e. read-only

```java
int cnt = 16;

Runnable r = () -> { System.out.println("count: " + cnt); };

cnt++;  // error: cnt is read-only
```
reason for "effectively final"

```java
int cnt = 0;

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

// start Runnable r in another thread
threadPool.submit(r);
...

while (cnt <= 16) /* NOP */;
System.out.println("cnt is now greater than 16");
```

problems:
- unsynchronized concurrent access
  - lack of memory model guaranties
- lifetime of local objects
  - locals are no longer "local"
the dubious "array boxing" hack

• to work around "effectively final" add another level of indirection
  – i.e. use an effectively final reference to a mutable object

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

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

System.out.println("contains " + r_cnt[0] + "exe-files");
```

• no problem, if everything is executed sequentially
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

inner class
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 "invokedynamic" from JSR 292)

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

• lambda expression
  – functional interfaces
  – lambda expressions (syntax)
  – method references

• functional patterns
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

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

- Stream\(\langle T\rangle\) has a `sorted()` method

  ```java
  Stream\(\langle T\rangle\) sorted(Comparator\(<\text{? super } T\rangle\) 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 `Person`

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

- used a wrapper that invokes `compareToName()`

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

- specify `compareToName()` directly (method reference)

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

- method references need context for type inference
  - conversion to a functional interface, similar to lambda expressions
agenda

- lambda expression
- functional patterns
  - internal iteration
  - execute around
external vs. internal iteration

• iterator pattern from GOF book
  – distinguishes between external and internal iteration
  – who controls the iteration?

• in Java, iterators are external
  – collection user controls the iteration

• in functional languages, iterators are internal
  – the collection itself controls the iteration
  – with Java 8 collections will provide internal iteration

GOF (Gang of Four) book:
"Design Patterns: Elements of Reusable Object-Oriented Software", by Gamma, Helm, Johnson, Vlissides, Addison-Wesley 1994
various ways of iterating

- internal iteration in Java 8
  - separates iteration from applied functionality
  - Java 5 for-each loop already comes close to it
Java 8 design (diagram)

Iterable
- iterator()
- forEach()

Collection

Stream
- forEach()
- filter()
- map()
- reduce()
- ...

parallelStream()
filter/map/reduce in Java 8

• for-each
  apply a certain functionality to each element of the collection

  accounts.forEach(a -> a.addInterest());

• filter
  build a new collection that is the result of a filter applied to each element in the original collection

  Stream<Account> result =
  accounts.filter(a -> a.balance() > 1000000?true:false);
filter/map/reduce (cont.)

- **map**
  build a new collection, where each element is the result of a mapping from an element of the original collection

  ```java
  IntStream result = accounts.map(a -> a.balance());
  ```

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

  ```java
  int sum = accounts.map(a -> a.balance())
  .reduce(0, (b1, b2) -> b1 + b2);
  ```

- **and many more**: `sorted()`, `anyMatch()`, `flatMap()`, ...
what is a stream?

• view/adaptor of a data source (collection, array, …)
  – class java.util.stream.Stream<T>
  – class java.util.stream.IntStream

• a stream has no storage => a stream is not a collection
  – supports forEach/filter/map/reduce functionality as shown before

• stream operations are "functional"
  – produce a result
  – do not alter the underlying collection
fluent programming

- streams support fluent programming
  - operations return objects on which further operations are invoked
  - e.g. stream operations return a stream

```java
interface Stream<T> {
    Stream<T> filter (Predicate<? super T> predicate);
    <R> Stream<R> map (Function<? super T, ? extends R> mapper);
    ...
}
```
fluent programming

- example:
  - find all managers of all departments with an employee younger than 25

```java
Manager[] find(Corporation c) {
    return c.getDepartments().stream()
        .filter(d -> d.getEmployees().stream()
                .map(Employee::getAge)
                .anyMatch(a -> a<25)
        )
        .map(Department::getManager)
        .toArray(Manager[]::new)
}
```
pitfalls - example: "add 5"

• situation:
  - List<Integer> ints containing some numbers
  - want to add 5 to each element

• first try:

  ints.stream().forEach(i -> { i += 5; });

  no effect !!!
pitfalls - example: "add 5" (cont.)

• remember trying this with for-each loop:

```java
for (int i : ints) {
    i += 5;
}
```

no effect !!!

• alternative, imperative way:

```java
for (int i; ints.size(); i++) {
    ints.set(i, ints.get(i) + 5);
}
```

okay!

• works
  – but iteration and applied functionality are intermingled
pitfalls - example: "add 5" (cont.)

• the functional way
  – don’t think about altering existing data
  – apply functionality to produce a new result

```java
Stream<Integer> ints5Added = ints.stream().map(i -> i + 5);
```

fine!
intermediate result / lazy operation

- bulk operations that return a stream are intermediate / lazy

```java
Stream<Integer> ints5Added = ints.stream().map(i -> i + 5);
```

- resulting `Stream` contains references to
  - original `List` `ints`, and
  - a `MapOp` operation object
    - together with its parameter (the lambda expression)

- the operation is applied later
  - when a terminal operation occurs
terminal operation

• a terminal operation does not return a stream
  – triggers evaluation of the intermediate stream

```java
Stream<Integer> ints5Added = ints.stream().map(i -> i + 5);
List<Integer> result = ints5Added.collect(Collectors.toList());
```

– or in fluent programming notation:

```java
List<Integer> result = ints.stream()
  .map(i -> i + 5)
  .collect(Collectors.toList());
```
more pitfalls - one-pass

```java
Stream<Integer> ints5Added = ints.stream().map(i -> i + 5);
ints5Added.forEach(i -> System.out.print(i + " 
 System.out.println("sum is: ");

ints5Added.reduce(0, (i, j) -> i + j));
```

```
6 7 8 9 10 11 12 13
Exception in thread "main"
java.lang.IllegalStateException: Stream source is already consumed

• stream elements can only be consumed once
  – like bytes from an input stream
```
fluent approach

```java
System.out.println("sum is: "+
    ints.stream()
    .map(i -> i + 5);
    .peek(i -> System.out.print(i +" "))
    .reduce(0, (i, j) -> i+j)
);
```

6 7 8 9 10 11 12 13 sum is: 76

- use intermediate `peek` operation
  - instead of a terminal `forEach` operation
agenda

- lambda expression
- functional patterns
  - internal iteration
  - execute around
execute-around (method) pattern/idiom

• situation

```java
public void handleInput(String fileName) throws IOException {
    InputStream is = new FileInputStream(fileName);
    try {
        ... use file stream ...
    } finally {
        is.close();
    }
}
```

• factor the code into two parts
  – the general "around" part
  – the specific functionality
    › passed in as lambda parameter
**execute-around pattern (cont.)**

- clumsy to achieve with procedural programming
  - maybe with reflection, but feels awkward

- typical examples
  - acquisition + release
  - using the methods of an API/service (+error handling)
  - ...

- blends into: *user defined control structures*
object monitor lock vs. explicit lock

**implicit lock**

```java
Object lock = new Object();

synchronized (lock) {
    ... critical region ...
}
```

**explicit lock**

```java
Lock lock = new ReentrantLock();

lock.lock();
try {
    ... critical region ...
} finally {
    lock.unlock();
}
```
helper class `Utils`

- split into a `functional type` and a `helper method`

```java
public class Utils {
    @FunctionalInterface
    public interface CriticalRegion {
        void apply();
    }

    public static void withLock(Lock lock, CriticalRegion cr) {
        lock.lock();
        try {
            cr.apply();
        } finally {
            lock.unlock();
        }
    }
}
```
example: thread-safe MyIntStack

- user code

```java
private class MyIntStack {
    private Lock lock = new ReentrantLock();
    private int[] array = new int[16];
    private int sp = -1;

    public void push(int e) {
        withLock(lock, () -> {
            if (++sp >= array.length)
                resize();
            array[sp] = e;
        });
    }

    ...
}
```

lambda converted to functional type

Critical Region

```
if (++sp >= array.length)
    resize();
array[sp] = e;
```
example: thread-safe MyIntStack (cont.)

- more user code

```java
...public int pop() {
    withLock(lock, () -> {
        if (sp < 0)
            throw new NoSuchElementException();
        else
            return array[sp--];
    });
}
```

- error:
  - `CriticalRegion::apply` does not permit return value
  - return in lambda is local, i.e., returns from lambda, not from `pop`
signature of CriticalRegion

- CriticalRegion has signature:

```java
public interface CriticalRegion {
    void apply();
}
```

- but we also need this signature
  - in order to avoid array boxing hack

```java
public interface CriticalRegion<T> {
    T apply();
}
```
signature of Critical Region (cont.)

• which requires an corresponding `withLock()` helper

```java
public static <T> T withLock(Lock lock, CriticalRegion<? extends T> cr) {
    lock.lock();
    try {
        return cr.apply();
    } finally {
        lock.unlock();
    }
}
```

• which simplifies the `pop()` method

```java
public int pop() {
    return withLock(lock, () -> {
        if (sp < 0)
            throw new NoSuchElementException();
        return (array[sp--]);
    });
}
```
signature of Critical Region (cont.)

- but creates problems for the `push()` method
  - which originally returns `void`
  - and now must return a ‘fake’ value from it’s critical region

- best solution (for the user code):
  - two interfaces: `VoidRegion`, `GenericRegion<T>`
  - plus two overloaded methods:
    
    ```java
    void withLock(Lock l, VoidRegion cr)
    <T> T withLock(Lock l, GenericRegion<? extends T> cr)
    ```
arguments are no problem

• input data can be captured
  – independent of number and type
  – reason: read-only

```java
public void push(final int e) {
    withLock(lock, () -> {
        if (++sp >= array.length)
            resize();
        array[sp] = e;
    });
}
```
coping with exceptions

- only runtime exceptions are fine

```java
public int pop() {
    return withLock(lock, () -> {
        if (sp < 0)
            throw new NoSuchElementException();
        return (array[sp--]);
    });
}
```

- exactly what we want:
  ```java
  pop() throws NoSuchElementException when stack is empty
  ```
checked exception problem

• checked exceptions cause trouble
  – Critical Region's method must not throw
    ```java
    void myMethod() throws IOException {
        withLock(lock, () -> {
            ... throws IOException ...
        });
    }
    ```

  – how can we propagate checked exception thrown by lambda back to surrounding user code?
tunnelling vs. transparency

- two options for propagation:
  - wrap it in a `RuntimeException` (a kind of "tunnelling"), or
  - transparently pass it back as is => `exception transparency`
"tunnelling"

- wrap checked exception into unchecked exception
  - messes up the user code

```java
void myMethod() throws IOException {
    try {
        withLock(lock, () -> {
            try {
                // ... throws IOException ...
            } catch (IOException ioe) {
                throw new RuntimeException(ioe);
            }
        });
    } catch (RuntimeException re) {
        Throwable cause = re.getCause();
        if (cause instanceof IOException) {
            throw ((IOException) cause);
        } else {
            throw re;
        }
    }
}
```
self-made exception transparency

• declare functional interfaces with checked exceptions
  – reduces user-side effort significantly

  – functional type declares the checked exception(s):

    ```java
    public interface VoidIOERegion {
        void apply() throws IOException;
    }
    ```

  – helper method declares the checked exception(s):

    ```java
    public static void withLockAndIOException
        (Lock lock, VoidIOERegion cr) throws IOException {
        lock.lock();
        try {
            cr.apply();
        } finally {
            lock.unlock();
        }
    }
    ```
self-made exception transparency (cont.)

- user code simply throws checked exception

```java
void myMethod() throws IOException {
    withLockAndIOException(lock, () -> {
        ... throws IOException ...
    });
}
```

caveat:
- only reasonable, when exception closely related to functional type
  - closely related = is typically thrown from the code block
  - not true in our example
  - just for illustration of the principle
wrap-up execute around / control structures

• factor code into
  – the general around part, and
  – the specific functionality
    › passed in as lambda parameter

• limitations
  – regarding checked exceptions & return type
    › due to strong typing in Java
  – is not the primary goal for lambdas in Java 8
  – nonetheless quite useful in certain situations
authors

Angelika Langer
Training & Consulting

Klaus Kreft
Consultant / Performance Expert, Germany

www.AngelikaLanger.com
Lambda Expressions

Q & A

Lambda Tutorial: AngelikaLanger.com/Lambdas/Lambdas.html