Lambdas I

Funktionale Programmierung in Java mit Lambdas

Angelika Langer

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 ~20 years
  - curriculum of a couple of challenging courses
  - JCP observer and Java champion since 2005

  - co-author of "Effective Java" column
    (`Angelikalanger.com/Articles/EffectiveJava.html`)

  - author of Java Generics FAQ online
    (`AngelikaLanger.com/GenericsFAQ/JavaGenericsFAQ.html`)

  - author of Lambda Tutorial & Reference
    (`AngelikaLanger.com/Lambdas/Lambdas.html`)

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 (Java 7)
  - 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)

today

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

- `for`-loop uses an `iterator`:

```
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

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

```
public interface Consumer<T> {
  void accept(T t)
  …
}
```

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

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

```
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 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, …

lambda expressions & functional interfaces

- functional interfaces

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

- conversions

```
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

error: cannot infer target type

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

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

agenda

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

- **functional patterns**

formal description

```
lambda = ArgList "->" Body

ArgList = Identifier
          | "(" Identifier [ "," Identifier ]* ")"
          | "(" Type Identifier [ "," Type Identifier ]* ")“

Body = Expression
       | "{" [ Statement ";" ]+ "}"
```

syntax samples

argument list

```
(int x, int y) -> { return x+y; }
        (x, y) -> { return x+y; }
            x  -> { return x+1; }

() -> { System.out.println(“I am a Runnable”); }
```

body

```
// single statement or list of statements
a -> {
       if (a.balance() < threshold) a.alert();
     }
// single expression
a -> (a.balance() < threshold) ? a.alert() : a.okay()
```

return type (is always inferred)

```
(Account a) -> { return a; }           // returns  Account
()          ->          5              // returns  int
```

local variable capture

```
int cnt = 16;

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

cnt++;
```

error: cnt is read-only

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

reason for "effectively final"

```
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”);
```

error

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

```
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

lambda

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

error

- different from inner classes
  - inner class body is a scope of its own

inner class

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

fine

lambda body lexically scoped, pt. 2

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

lambda

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

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

inner class

```
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 "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

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

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

- interface `Comparator` is a functional interface

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

inherited from `Object`

- sort a collection/array of `Persons`

```
Stream<Person> psp = Arrays.parallelStream(personArray);
…
psp.sorted((Person a, Person b) -> Person.compareByName(a,b));
```

example (cont.)

- used a wrapper that invokes `compareByName()`

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

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

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

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

agenda

- lambda expression
- functional patterns

patterns + idioms

- sometimes hard to do it right
  - coming from an imperative programming approach

- even with a background in functional programming
  - Java has its own way, with respect to the existing language

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

```
Collection<String> c = ...

Iterator<String> iter = c.iterator();
while (iter.hasNext())
   System.out.println(iter.next() + ' ');


for(String s : c)
   System.out.println(s + ' ');


c.forEach(s -> System.out.println(s) + ' ');
```

< Java 5

Java 5

Java 8

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

external iteration & performance

- Java 5 for-each loop is just syntactical sugar
  - still uses iterator's `hasNext()` and `next()`
- redundancy: often `next()` calls `hasNext()` again
  - excerpt from `LinkedList<E>.ListItr<E>`

```
public E next() {
  if (!hasNext())
    throw new NoSuchElementException();
  ...
}
```

- with `forEach()` no need to perform `hasNext()` twice

iteration in Java 8

- `java.lang.Iterable<T>` contains two methods now

  `Iterator<T> iterator()`

  - for external iteration

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

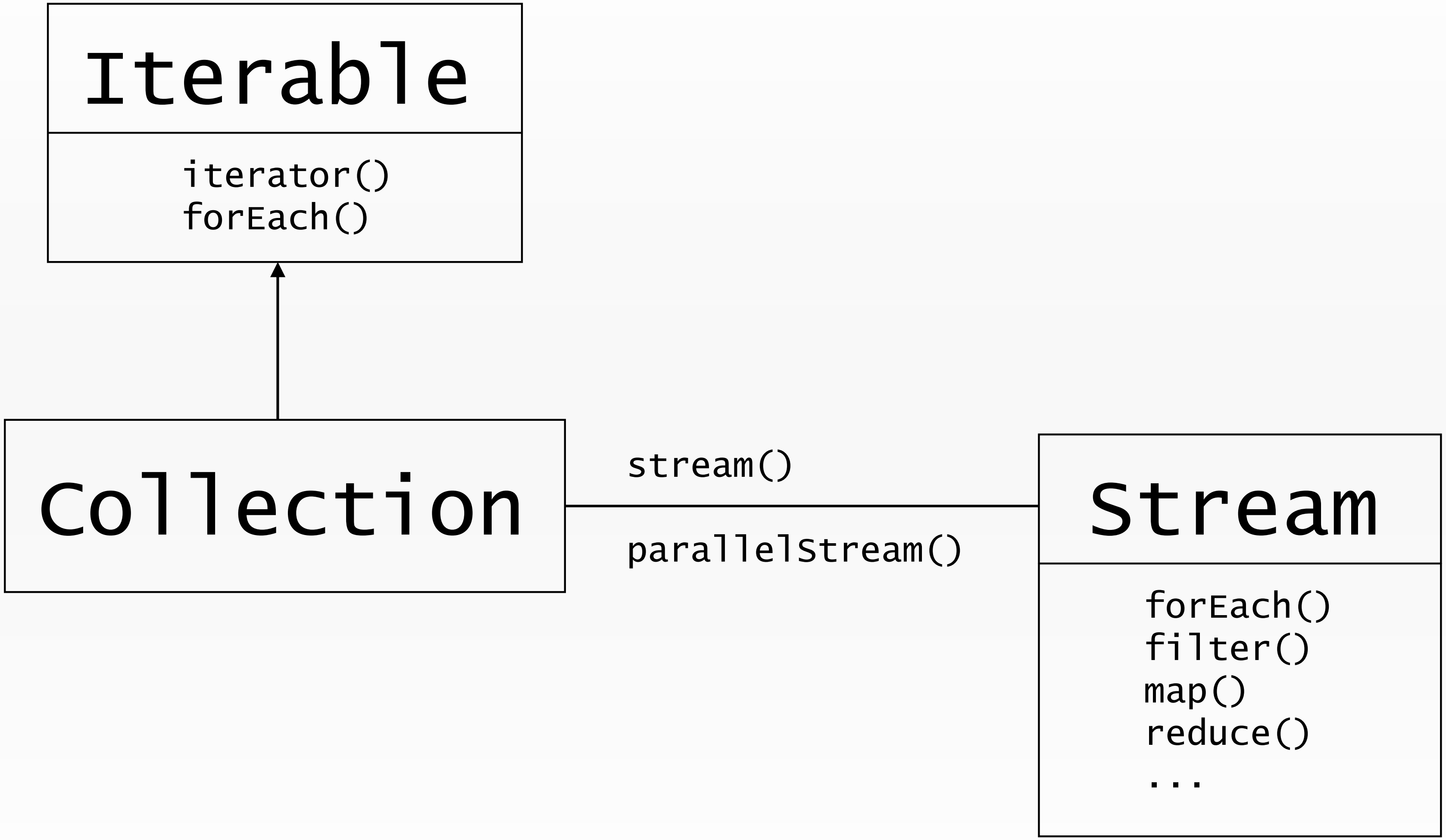
  - for internal iteration

- your choice which to use
  - external or internal iteration

streams in Java 8

- interface `java.util.stream.Stream<E>`
  - supports `forEach`, `filter`, `map`, `reduce`, and more
- two new methods in `java.util.Collection<E>`
  - `Stream<E> stream()`, sequential functionality
  - `Stream<E> parallelStream()`, parallel functionality

Java 8 design (diagram)



Iterable

iterator()
forEach()

Collection

stream()

parallelStream()

Stream

forEach()
filter()
map()
reduce()
...

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

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

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

```
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

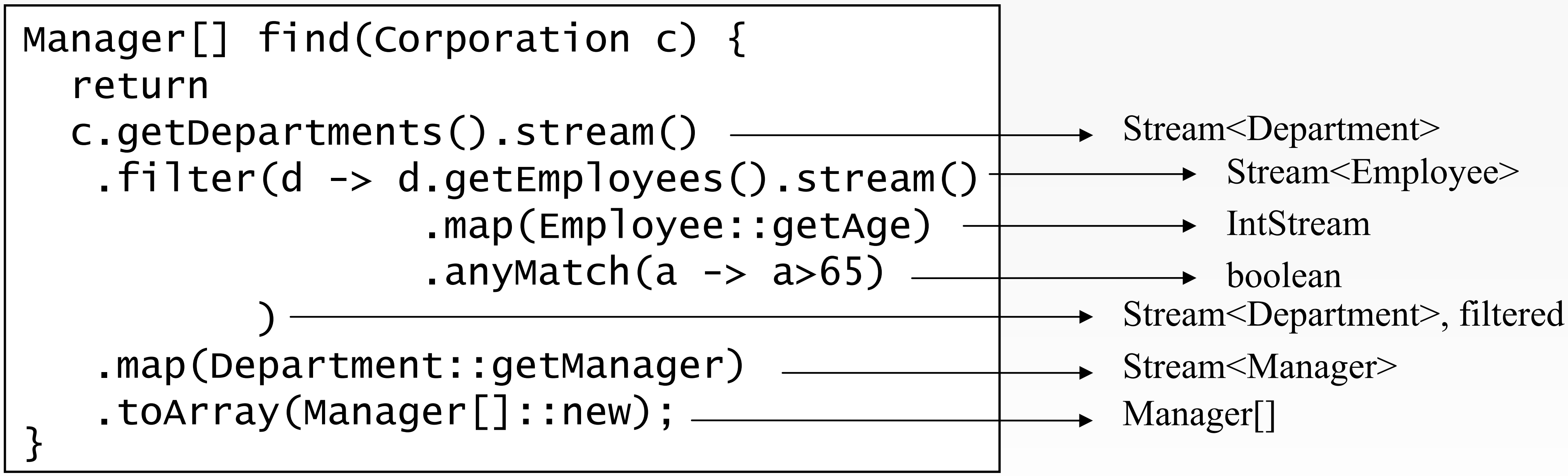
fluent programming

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

```
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 older than 65



```
Manager[] find(Corporation c) {
  return
  c.getDepartments().stream()
   .filter(d -> d.getEmployees().stream()
                 .map(Employee::getAge)
                 .anyMatch(a -> a>65)
          )
   .map(Department::getManager)
   .toArray(Manager[]::new);
}
```

Stream<Department>

Stream<Employee>

IntStream

boolean

Stream<Department>, filtered

Stream<Manager>

Manager[]

pitfalls - example: "add 5"

No!

- situation:
  - `ArrayList<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:

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

no effect !!!

- alternative, functional way:
  - don’t alter existing data, produce a new result

```
IntStream ints5Added =
                ints.stream().mapToInt(i -> i + 5);
```

fine!

intermediate result / lazy operation

- bulk operations that return a stream are intermediate / lazy

```
IntStream ints5Added = ints.stream().mapToInt(i -> i + 5);
```

- resulting `IntStream` contains references to
  - original `ArrayList` `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

Yes!

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

```
IntStream ints5Added = ints.stream().mapToInt(i -> i + 5);

System.out.println("sum is: " +
                    ints5Added.reduce(0, (i, j) -> i+j));
```

```
sum is: 76
```

agenda

- **lambda expression**
- **functional patterns**
  - internal iteration
  - execute around

execute-around (method) pattern/idiom

- situation

```
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

```
Object lock = new Object();

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

explicit lock

```
Lock lock = new ReentrantLock();

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

helper class `Utils`

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

```
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*

```
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
`CriticalRegion`

example : thread-safe `MyIntStack` (cont.)

- more user code

```
  ...

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

local return from lambda

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

example : thread-safe `MyIntStack` (cont.)

No!

- more user code

```
  ...

  public int pop() {
    int[] retVal = { -1 };
    withLock(lock, () -> {
      if (sp < 0)
        throw new NoSuchElementException();
      else
        retVal[0] = array[sp--];
     });
     return retVal[0];
  }
}
```

array boxing hack

- implementation uses dubious array boxing hack

signature `of CriticalRegion`

- `CriticalRegion` has signature:

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

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

```
public interface CriticalRegion<T> {
  T apply();
}
```

signature of `CriticalRegion` (cont.)

- which requires an corresponding `withLock()` helper

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

- which simplifies the `pop()` method

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

    return (array[sp--]);
  });
}
```

no array boxing hack needed

signature of `CriticalRegion` (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:

```
    void  withLock(Lock l, VoidRegion        cr)
<T> T withLock(Lock l, GenericRegion<T> cr)
```

arguments are no problem

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

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

    array[sp] = e;
  });
}
```

method argument is captured

checked exceptions are a problem

- only runtime exceptions are fine

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

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

- for checked exceptions we need *exception transparency*

exception transparency

- exception propagation:
  - how can we propagate checked exception thrown by lambda back to surrounding user code ?
- 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

```
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;
  }
}
```

wrap

unwrap

self-made exception transparency

- declare functional interfaces with checked exceptions
  - reduces user-side effort significantly
  - functional type declares the checked exception(s):

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

  - helper method declares the checked exception(s):

```
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

```
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

www.AngelikaLanger.com

Lambda Expressions

Q & A

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