

Java gets a closure

Tomasz Kowalczewski



Agenda

- Lambdas and closures
- Java syntax
- Functional interfaces
- Implementation notes
- Defender methods and interface evolution
- Library changes
- Lambda related language changes

Lambda expression

- Lambda expression is another name for an anonymous function e.g. function not bound to an identifier.
- In Java it includes an optional list of type parameters, a list of formal parameters, and an expression or block expressed in terms of those parameters.

Example

```
char[] string = „jdd 2011”
```

```
(int count) -> {  
    for(int i = 0; i < count; i++) {  
        string[i] = Character.toUpperCase(string[i]);  
    }  
}
```

free variable



Why we hate anonymous inner classes?

- Java already has a form of closures - anonymous inner classes

```
List<Person> list = new ArrayList<>();
Collections.sort(list, new Comparator<Person>() {
    @Override
    public int compare(Person p1, Person p2) {
        return p1.getName().compareTo(p2.getName());
    }
});
```



Why we hate anonymous inner classes?

- Bulky syntax
- Inability to capture non-final local variables
- Transparency issues surrounding the meaning of return, break, continue, and 'this'
- No nonlocal control flow operators

Will we love lambda expressions?

- ~~Bulky syntax~~
- ~~Inability to capture non-final local variables~~
- Transparency issues surrounding the meaning of return, break, continue, ~~and 'this'~~
- No nonlocal control flow operators

Lambda syntax

```
List<Person> list = new ArrayList<>();  
Collections.sort(list, (p1, p2) ->  
    p1.getName().compareTo(p2.getName())  
);
```



```
List<Person> list = new ArrayList<>();  
Collections.sort(list, new Comparator<Person>() {  
    @Override  
    public int compare(Person p1, Person p2) {  
        return p1.getName().compareTo(p2.getName());  
    }  
});
```



Lambda syntax

LambdaExpression:

TypeParameters_{opt} LambdaParameters '->' LambdaBody

LambdaParameters:

Identifier

(' InferredFormalParameterList ')'

(' FormalParameterList_{opt} ')'

InferredFormalParameterList:

Identifier

InferredFormalParameterList ', ' Identifier

LambdaBody:

Expression

Block



Lambda syntax examples

```
() -> {}
```

```
() -> 3.14
```

```
() -> { System.out.println("lambda"); }
```

```
() -> { return 3.14; }
```

```
i -> i*2
```

```
(int i) -> i+1
```

```
(i) -> i+1
```

```
(List<Person> people) -> {
```

```
    for(Person person : people) {
```

```
        System.out.println(person.getFirstName());
```

```
    }
```



Invalid syntax examples

- Either types/modifiers are inferred or we need to specify all of them:

```
(final x) -> {}
```

```
(int x, y) -> 3.14
```

- Lambda body is either an expression (no brackets) or a block (brackets and return statement needed):

```
() -> { 3.14; }
```

- Only unary lambda may omit parentheses:

```
i, j -> i*j
```

What is the type of a lambda?

- There will be no function types in Java 8
- We need to define how lambdas interact with other parts of Java
- We need ways to store, pass and execute them

Functional interfaces

- Previously known as SAM interfaces
 - SAM = Single Abstract Method
- Interfaces that define one method
 - Public methods defined by Object do not count

```
interface Runnable { void run(); }
```

```
interface Callable<T> { T call(); }
```

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



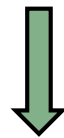
Closure conversion

- Lambda expression is converted to an instance of a functional interface
- It can appear in assignment, invocation and casting contexts
- The type lambda is converted to is inferred from the context
 - Target typing
- Parameter types of the lambda are also inferred

Type inference

```
interface Processor<T> {  
    void process(Context c, T item);  
}
```

```
Processor<Integer> p = (Context c, Integer i) ->  
    { ... };
```



```
Processor<Integer> p = (c, i) -> { ... };
```

Local variable capture

- Lambda expressions can refer to effectively final local variables from the enclosing scope

```
int getAdultCount(List<Person> people, int threshold) {  
    return Collections.filter(people, p -> p.getAge()  
        > threshold).size();  
}
```

```
void registerNotifier(JButton button, JLabel label) {  
    button.addActionListener(e -> label.setText(""));  
}
```



Local variable capture

- Cannot do this:

```
int accumulatedExperience = 0;
people.forEach(p -> accumulatedExperience += p.getEmploymentYears());
```

- Don't even try this:

```
/* final */ int[] accumulatedExperience = { 0 };
people.forEach(p -> accumulatedExperience[0] += p.getEmploymentYears());
```

- Try this:

```
/* final */ AtomicInteger accumulatedExperience = new AtomicInteger();
people.forEach(p -> accumulatedExperience.addAndGet(p.getEmploymentYears()));
```

- Best solution:

```
people.reduce(0, (value, p) -> value + p.getEmploymentYears());
people.reduce(0, Math::add);
```



Lexical scoping

- All identifiers retain their meaning inside lambda expression
- The meaning of **this** does not change.
- Lambda formals cannot shadow variables from enclosing scope.
- There is no non-local control flow

Lexical scoping

```
interface ConfigFilter {  
    String KEY = "configuration.source";  
    boolean matches(Properties properties);  
}
```

```
class FileConfigurator {  
    String VALUE = „file://“;  
  
    public void configure(List<Properties> list) {  
        configure(list, p ->  
            p.getProperty(ConfigFilter.KEY).startsWith(VALUE));  
    }  
  
    void configure(List<Properties> l, ConfigurationFilter f) { ... }
```



Lambda self reference

- Definite assignment rules say when a variable is definitely assigned
 - e.g. if it is a local variable it can be safely used
- Definite unassignment say when a variable is definitely unassigned
 - e.g. so that it can be assigned if it is final
- Function `fib = n -> { n == 0 ? 1 : n * fib.apply(n - 1) };`

Method references

- Method reference is a shorthand for a lambda invoking just that method
- Static methods simply translate like lambda with same arguments and return type:

```
class Integer {  
    public static String toString(int i) { ... }  
}
```

```
Mapper<Integer, String> a = i -> Integer.toString(i);  
Mapper<Integer, String> b = Integer::toString;
```



Method references

- Non static method reference of type T translates like lambda with an additional argument of type T:

```
Mapper<String, Integer> a = s -> s.length;  
Mapper<String, Integer> b = String::length;
```

- Instance method reference translates like lambda with same arguments and return type (and implicit receiver):

```
Callable<Integer> c = "test"::length;
```



Constructor references

- Basically same as method references:

```
Callable<Person> p = Person::new;
```

- **Generic type constructor:**

```
LinkedList<Integer>::new
```

- **Raw type constructor :**

```
LinkedList::new
```

- **Generic constructor with type argument:**

```
Foo::<Integer>new
```



Exception transparency?

- Generic exceptions are cumbersome to use
- Libraries either use interface with methods that throw Exception or throw nothing:

```
interface Runnable {  
}
```

```
interface Callable<V> {  
    V call() throws Exception  
}
```

```
interface Callable<V, E extends Exception> {  
    V call() throws E  
}
```



Exception transparency?

```
interface Callable<V, throws E> {  
    V call() throws E  
}
```

```
class Executor {  
    static <V, throws E> execute(Callable<V, E> throws E;  
}
```

```
Callable<Integer, IOException> c = ...  
Executor.execute(c); // Throws IOException
```

```
Callable<Integer> d = ...  
Executor.execute(d); // Throws nothing
```

```
Callable<Integer, ExecutionException, BadBreathException> e = ...  
Executor.execute(e); // Throws ExecutionException, BadBreathException
```



Translation to inner classes

```
public class SimpleLambda {  
    Runnable r = new Runnable() {  
        public void run() { }  
    };  
}
```

```
public class SimpleLambda {  
    Runnable r = () -> {};  
}
```

- Compiling this source files will each time generate two classes:
 - SimpleLambda.class
 - SimpleLambda\$1.class
- In both situations classes are almost identical

Translation to method handles

- Try compiling with option `-XDlambdaToMethod`
- This will generate one class file.
- It will contain additional elements:
 - Static method called `lambda$0` with compiled lambda body
 - Bootstrap method that will call `ProxyHelper` to generate interface proxy
 - Calls to `Runnable.run()` will be dispatched through `MethodHandle`
 - `InvokeDynamic` instruction at instantiation callsite

Extension Methods



Extension methods

- Programming style would be different if Java had closures from day one
- Different library designs, different collection classes
- Adding closures without library support would be dissapointing
- When interface is published it is effectively freezed – adding new methods will break existing implementations

Extension methods

- We would like to be able to write:

```
list.sortBy(Person::getFirstName).filter(Person::
  isEmployed).filter(p -> p.getAge()
  <30).forEach(System.out::println);
```

```
students.map(s -> s.getScore()).reduce(0.0,
  Math::max);
```

Static extension methods

- In C# they enable adding methods to existing types without recompiling or creating a new type.
- Special kind of static method, called as if it was an instance method on the extended type.
- Use site extension – user is in control
- Create illusion of adding a method
- Not reflectively discoverable
- No way to override

Extension methods

- Mechanism for interface evolution
- Library developer is in control
- Main use case: java collections
- A.k.a. public defender methods:

“if you cannot afford an implementation of this method, one will be provided for you”



Virtual extension methods

```
public interface List<E> extends Collections<E> {  
public void sort(Comparator<? super E> c) default  
    Collections.<E>sort;  
}
```

```
public class Collections {  
public static <T extends Comparable<? super T>> void  
    sort(List<T> list) { ... }  
}
```

```
List<Person> jddAttendee = ...  
jddAttendee.sort(...);
```



Virtual extension methods

- Compiled to regular invokeinterface call
- Caller is unaware of special call dispatch
- Target method is resolved by JVM at runtime
- Adds multiple inheritance of behavior not state!
- Changes are (mostly) source and binary compatible

Method resolution

- Inheritance of methods from classes and interfaces is treated separately
- First JVM performs standard method implementation search – from the receiver class upwards through inheritance hierarchy to Object

Method resolution

```
interface List
void foo() default Collections.foo;
```



```
class D implements List
void foo() { ... }
```



```
class C extends D
implements List
```

Which method is called?

```
new C().foo();
```



Method resolution

- List interfaces implemented by C (directly or indirectly) which provide a default for the method in question
- Remove all items that are superinterface of any other one from this list
- Create a set of distinct defaults provided by interfaces on the list
- Resolution is successful if this set has a single item



Throw a linkage exception otherwise

Method resolution

```
interface Collection  
void foo() default Collections.foo;
```



```
interface List implements  
Collection  
void foo() default Lists.foo;
```



```
class D implements List
```

Which method is called?

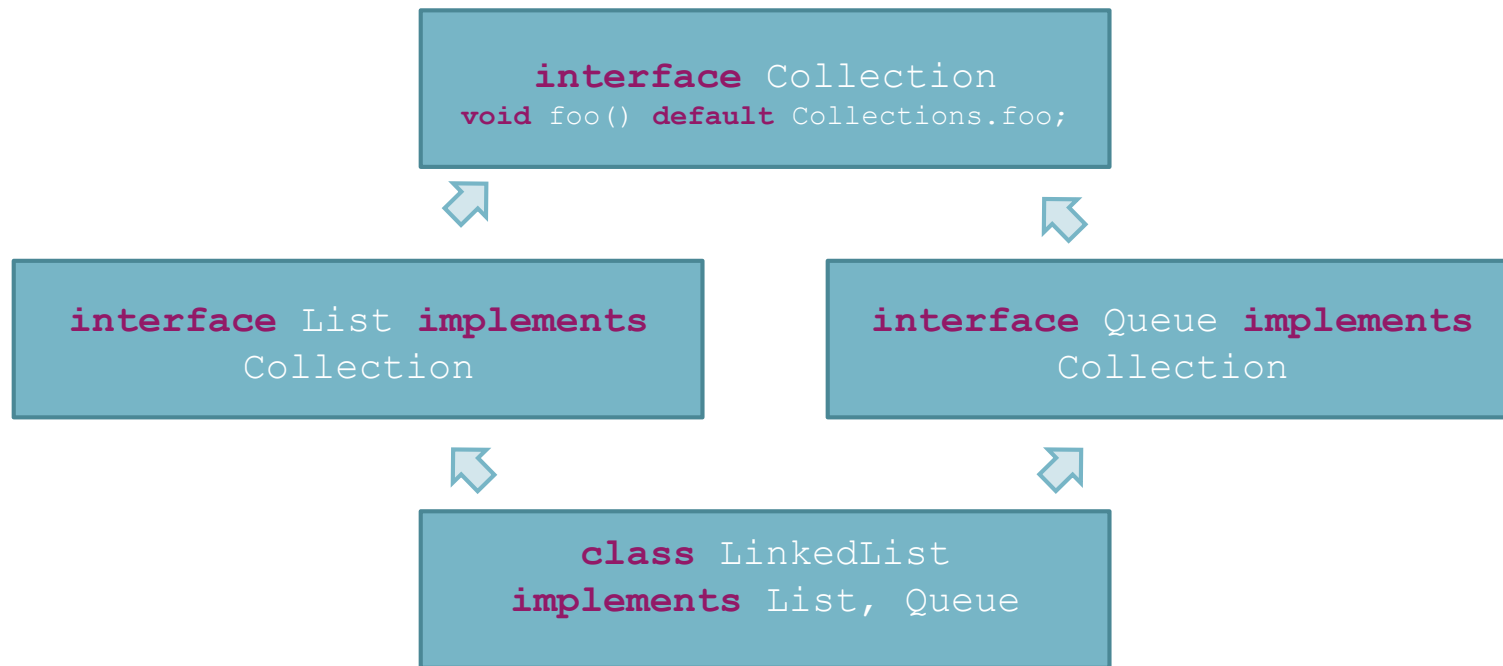
```
new C().foo();
```



```
class C extends D  
implements Collection
```



Method resolution



- When using this method resolution rules diamonds are not a problem (there is no state inheritance!)

Manual method disambiguation

- Compile time disambiguation of conflicting defaults is done using new syntax: `A.super.method()`

```
interface A {  
    void foo() default As.foo;  
}
```

```
interface B {  
    void foo() default Bs.foo;  
}
```

```
class C implements A, B {  
    public void foo() { A.super.foo(); }  
}
```



Collection enhancements



Collection enhancements

```
public interface Predicate<T> {  
    boolean eval(T t);  
}
```

```
public interface Block<T> {  
    void apply(T t);  
}
```

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



Iterable

```
public interface Iterable<T>{  
    Iterable<T> filter(Predicate<? super T> predicate)  
        default Iterables.filter;  
  
    Iterable<T> forEach(Block<? super T> block)  
        default Iterables.forEach;  
  
    <U> Iterable<U> map(Mapper<? super T, ? extends U>  
        mapper) default Iterables.map;  
  
    T reduce(T base, Operator<T> reducer)  
        default Iterables.reduce;  
}
```



Collection

```
public interface Collection<E> extends Iterable<E> {  
  
    boolean retainAll(Predicate<? super E> filter)  
        default CollectionHelpers.retainAll;  
  
    boolean removeAll(Predicate<? super E> filter)  
        default CollectionHelpers.removeAll;  
  
    void addAll(Iterable<? extends E> source)  
        default CollectionHelpers.addAll;  
  
}
```



Iterator

```
public interface Iterator<E> {
    boolean hasNext();
    E next();
    void remove() default Iterators.removeUnsupported;
}

public final class Iterators {
    public static <T> void removeUnsupported(Iterator<T> t) {
        throw new UnsupportedOperationException("remove");
    }
}
```



Enumeration

```
interface Enumeration<E> extends Iterator<E> {  
    boolean hasMoreElements();  
  
    E nextElement();  
  
    // { return hasMoreElements(); }  
    boolean hasNext() default Enumerations.hasNext;  
  
    // { return nextElement(); }  
    E next() default Enumerations.next;  
}
```

Collections enhancements

```
Collections.sort(list, (p1, p2) ->  
    p1.getName().compareTo(p2.getName())  
);
```



```
interface Extractor<T,U> { public U extract(T t); }
```

```
public void<T, U extends Comparable<? super U>>  
    sortBy(Collection<T> coll, Extractor<T, U> ext);
```



```
Collections.sortBy(people, Person::getName);
```



```
people.sortBy(Person::getName);
```



Development status

- Started in december 2009
- Compiler implemented as part of an OpenJDK
- Early Draft Review recently published
 - <http://www.jcp.org/en/jsr/summary?id=335>
- No VM support for extension methods yet



Resources

- Compiler prototype binaries available at:
 - <http://jdk8.java.net/lambda/>
- State of the lambda document:
 - <http://cr.openjdk.java.net/~briangoetz/lambda/lambda-state-3.html>
- Extension methods draft:
 - <http://cr.openjdk.java.net/~briangoetz/lambda/Defender%20Methods%20v3.pdf>
- Extension methods weaving agent available at:
 - hg.openjdk.java.net/lambda/defender-prototype/

Resources

- Brian Goetz's Oracle Blog
 - <http://blogs.oracle.com/briangoetz>
- Neal Gafter's blog (interesting for historical purposes)
 - <http://gafter.blogspot.com>
- lambda-dev discussion list
 - <http://mail.openjdk.java.net/pipermail/lambda-dev>