Project Lambda

JSR 335

22 November 2013

Martin Škurla

Who is this guy?

- Martin Škurla
  - Certifications
    - Sun Certified Programmer for the Java 2 Platform, Standard Edition 6.0
    - Oracle Certified Professional, Java EE 5 Web Component Developer
  - Open Source Committer
    - Gephi (GSoC 2010), NetBeans (NetCAT), Abego Tree Layout, AST Visualizer
  - Conference Speaker
    - Fosdem 2012
  - Technical Reviewer
    - CPress, Grada, Manning
  - Previous work experience
    - Sors Technology, ESET, Celum, Atlassian
  - Currently
    - PSEC at Barclays (Exchange connectivity)

“Mostly, when you see programmers, they aren’t doing anything. One of the attractive things about programmers is that you cannot tell whether or not they are working simply by looking at them. Very often they’re sitting there seemingly drinking coffee and gossiping, or just staring into space. What the programmer is trying to do is get a handle on all the individual and unrelated ideas that are scampering around in his head.”

Charles M. Strauss

Agenda

- Reasons to introduce lambda expressions
- What are lambda expressions, comparison with closures
- Short history (BGGA, CICE, FCM)
- Project Lambda
- Syntax and semantics
  - Lambda expressions and Function types
  - Functional interfaces
  - Target typing
  - Lexical scoping
  - Capturing variables
- Default and static interface methods
- Stream API

Reasons to introduce lambda expressions

- To address concurrency more efficiently

Reasons to introduce lambda expressions

- To address concurrency more efficiently
- To be more functional friendly

Reasons to introduce lambda expressions

- To address concurrency more efficiently
- To be more functional friendly
- To be able to model code as data

Reasons to introduce lambda expressions

- To address concurrency more efficiently
- To be more functional friendly
- To be able to model code as data
- To address issues of anonymous inner classes:
  - Bulky syntax
  - Confusion surrounding the meaning of names and this
  - Inflexible class-loading and instance creation semantics
  - Inability to capture non-final local variables
  - Inability to abstract over control flow

Reasons to introduce lambda expressions

- To address concurrency more efficiently
- To be more functional friendly
- To be able to model code as data
- To address issues of anonymous inner classes:
  - Bulky syntax
  - Confusion surrounding the meaning of names and this
  - Inflexible class-loading and instance creation semantics
  - Inability to capture non-final local variables
  - Inability to abstract over control flow
- It's about time

Lambda expressions

- Lambda expression
  - also called lambda function, function literal or function constant

Lambda expressions

- Lambda expression
  - also called lambda function, function literal or function constant
  - an anonymous function defined and possibly called without being bound to an identifier

Lambda expressions

- Lambda expression
  - also called lambda function, function literal or function constant
  - an anonymous function defined and possibly called without being bound to an identifier
  - originated in λ-calculus by Alonso Church in the 1930s

Lambda expressions

- Lambda expression
  - also called lambda function, function literal or function constant
  - an anonymous function defined and possibly called without being bound to an identifier
  - originated in λ-calculus by Alonso Church in the 1930s
- Closure
  - also called function closure or lexical closure

Lambda expressions

- Lambda expression
  - also called lambda function, function literal or function constant
  - an anonymous function defined and possibly called without being bound to an identifier
  - originated in λ-calculus by Alonso Church in the 1930s
- Closure
  - also called function closure or lexical closure
  - a function or reference to a function together with a referencing environment

Short history

- 3 main proposals filled in 2006

Short history

- 3 main proposals filled in 2006
- BGGA
  - The most heavyweight proposal (non local return, function types, …)
  - Lead by Gilad Bracha, Neal Gafter, James Gosling

Short history

- 3 main proposals filled in 2006
- BGGA
  - The most heavyweight proposal (non local return, function types, …)
  - Lead by Gilad Bracha, Neal Gafter, James Gosling
- CICE
  - “Concise Instance Creation Expressions”
  - Mostly syntactic sugar over anonymous inner classes
  - Lead by Josh Bloch

Short history

- 3 main proposals filled in 2006
- BGGA
  - The most heavyweight proposal (non local return, function types, …)
  - Lead by Gilad Bracha, Neal Gafter, James Gosling
- CICE
  - “Concise Instance Creation Expressions”
  - Mostly syntactic sugar over anonymous inner classes
  - Lead by Josh Bloch
- FCM
  - In between of previous two proposals (method references)

Short history

- 3 main proposals filled in 2006
- BGGA
  - The most heavyweight proposal (non local return, function types, …)
  - Lead by Gilad Bracha, Neal Gafter, James Gosling
- CICE
  - “Concise Instance Creation Expressions”
  - Mostly syntactic sugar over anonymous inner classes
  - Lead by Josh Bloch
- FCM
  - In between of previous two proposals (method references)
- … and all of them have failed

Project Lambda formally

- Formally
  - JSR 335: Lambda Expressions for the Java Programming Language
  - Part of JSR 337: Java SE 8

Project Lambda formally

- Formally
  - JSR 335: Lambda Expressions for the Java Programming Language
  - Part of JSR 337: Java SE 8
- Consists of
  - Functional Interfaces
  - Lambda Expressions
  - Method and Constructor References
  - Default Methods
  - Typing, Evaluation and Overload Resolution
  - Type Inference

Syntax and semantics (part 1)

- Lambda expressions
  - Aimed to address the “vertical problem”

Syntax and semantics (part 1)

- Lambda expressions
  - Aimed to address the “vertical problem”
  - Syntax: ‘(’ `optionalArgumentList` ‘)’ ‘->’ `expression` | ‘{’ `codeBlock` ‘}’

Syntax and semantics (part 1)

- Lambda expressions
  - Aimed to address the “vertical problem”
  - Syntax: `'(' optionalArgumentList ')' '->' expression | '{' codeBlock '}'`
- Function types

Syntax and semantics (part 1)

- Lambda expressions
  - Aimed to address the “vertical problem”
  - Syntax: ‘(’ `optionalArgumentList` ‘)’ ‘->’ `expression` | ‘{’ `codeBlock` ‘}’
- Function types
  - Examples:
    - Function types in Scala: `(Int,Int) => Boolean`
    - Delegates in C#: `public delegate void deleg(Object p1, Object p2);`

Syntax and semantics (part 1)

- Lambda expressions
  - Aimed to address the “vertical problem”
  - Syntax: ‘(’ `optionalArgumentList` ‘)’ ‘->’ `expression` | ‘{’ `codeBlock` ‘}’
- Function types
  - Examples:
    - Function types in Scala: `(Int,Int) => Boolean`
    - Delegates in C#: `public delegate void deleg(Object p1, Object p2);`
  - We don’t have function types in Java

Syntax and semantics (part 1)

- Lambda expressions
  - Aimed to address the “vertical problem”
  - Syntax: ‘(’ `optionalArgumentList` ‘)’ ‘->’ `expression` | ‘{’ `codeBlock` ‘}’
- Function types
  - Examples:
    - Function types in Scala: `(Int,Int) => Boolean`
    - Delegates in C#: `public delegate void deleg(Object p1, Object p2);`
  - We don’t have function types in Java
  - They would arguably be ineffective
    - Because method dispatch is already quite complex and function types would make it even worse (type signatures, type equality, …)
    - Because API designers would need to maintain 2 types of incompatible APIs
    - Because teaching JVM about function types would be huge effort (new type signatures, new bytecode for invocation, new verification rules, …)

Syntax and semantics (part 2)

- Functional interfaces
  - Previously called SAM types

Syntax and semantics (part 2)

- Functional interfaces
  - Previously called SAM types
  - Interfaces with just one method

Syntax and semantics (part 2)

- Functional interfaces
  - Previously called SAM types
  - Interfaces with just one method
  - Compiler identifies interfaces as such based on its structure

Syntax and semantics (part 2)

- Functional interfaces
  - Previously called SAM types
  - Interfaces with just one method
  - Compiler identifies interfaces as such based on its structure
  - Able to manually mark them by `@java.lang.FunctionalInterface`

Syntax and semantics (part 2)

- Functional interfaces
  - Previously called SAM types
  - Interfaces with just one method
  - Compiler identifies interfaces as such based on its structure
  - Able to manually mark them by `@java.lang.FunctionalInterface`
  - Existing examples: `Runnable`, `Callable<T>`, `Comparator<T>`, `ActionListener`, `PropertyChangeListener`, …
  - Addition of new ones: `Predicate<T>`, `Consumer<T>`, `Function<T,R>`, `Supplier<T>`, `UnaryOperator<T>`, `BinaryOperator<T>`

Syntax and semantics (part 2)

- Functional interfaces
  - Previously called SAM types
  - Interfaces with just one method
  - Compiler identifies interfaces as such based on its structure
  - Able to manually mark them by `@java.lang.FunctionalInterface`
  - Existing examples: `Runnable`, `Callable<T>`, `Comparator<T>`, `ActionListener`, `PropertyChangeListener`, ...
  - Addition of new ones: `Predicate<T>`, `Consumer<T>`, `Function<T,R>`, `Supplier<T>`, `UnaryOperator<T>`, `BinaryOperator<T>`
- Target typing
  - Determining the type of lambda expression

Syntax and semantics (part 2)

- Functional interfaces
  - Previously called SAM types
  - Interfaces with just one method
  - Compiler identifies interfaces as such based on its structure
  - Able to manually mark them by `@java.lang.FunctionalInterface`
  - Existing examples: `Runnable`, `Callable<T>`, `Comparator<T>`, `ActionListener`, `PropertyChangeListener`, …
  - Addition of new ones: `Predicate<T>`, `Consumer<T>`, `Function<T,R>`, `Supplier<T>`, `UnaryOperator<T>`, `BinaryOperator<T>`
- Target typing
  - Determining the type of lambda expression
  - Design goal: *"Don't turn a vertical problem into a horizontal problem."*

Syntax and semantics (part 2)

- Functional interfaces
  - Previously called SAM types
  - Interfaces with just one method
  - Compiler identifies interfaces as such based on its structure
  - Able to manually mark them by `@java.lang.FunctionalInterface`
  - Existing examples: `Runnable`, `Callable<T>`, `Comparator<T>`, `ActionListener`, `PropertyChangeListener`, …
  - Addition of new ones: `Predicate<T>`, `Consumer<T>`, `Function<T,R>`, `Supplier<T>`, `UnaryOperator<T>`, `BinaryOperator<T>`
- Target typing
  - Determining the type of lambda expression
  - Design goal: *"Don't turn a vertical problem into a horizontal problem."*
  - Poly expressions

Syntax and semantics (part 3)

- Lexical scoping
  - Determining the meaning of names and `this`

Syntax and semantics (part 3)

- Lexical scoping
  - Determining the meaning of names and `this`
  - Inherited members in inner classes can shadow outer declarations

Syntax and semantics (part 3)

- Lexical scoping
  - Determining the meaning of names and `this`
  - Inherited members in inner classes can shadow outer declarations
  - Lambda expressions do not inherit any names from its supertype

Syntax and semantics (part 3)

- Lexical scoping
  - Determining the meaning of names and `this`
  - Inherited members in inner classes can shadow outer declarations
  - Lambda expressions do not inherit any names from its supertype
  - Lambda expressions do not introduce a new level of scoping, they are lexically scoped

Syntax and semantics (part 3)

- Lexical scoping
  - Determining the meaning of names and `this`
  - Inherited members in inner classes can shadow outer declarations
  - Lambda expressions do not inherit any names from its supertype
  - Lambda expressions do not introduce a new level of scoping, they are lexically scoped
  - Anonymous inner classes still make sense and are essential in some scenarios

Syntax and semantics (part 3)

- Lexical scoping
  - Determining the meaning of names and `this`
  - Inherited members in inner classes can shadow outer declarations
  - Lambda expressions do not inherit any names from its supertype
  - Lambda expressions do not introduce a new level of scoping, they are lexically scoped
  - Anonymous inner classes still make sense and are essential in some scenarios
- Capturing variables
  - Lambda expressions and anonymous inner class now able to capture “effectively final” local variables

Syntax and semantics (part 3)

- Lexical scoping
  - Determining the meaning of names and `this`
  - Inherited members in inner classes can shadow outer declarations
  - Lambda expressions do not inherit any names from its supertype
  - Lambda expressions do not introduce a new level of scoping, they are lexically scoped
  - Anonymous inner classes still make sense and are essential in some scenarios
- Capturing variables
  - Lambda expressions and anonymous inner class now able to capture “effectively final” local variables
  - Local variable is effectively final if its initial value is never changed

Syntax and semantics (part 3)

- Lexical scoping
  - Determining the meaning of names and `this`
  - Inherited members in inner classes can shadow outer declarations
  - Lambda expressions do not inherit any names from its supertype
  - Lambda expressions do not introduce a new level of scoping, they are lexically scoped
  - Anonymous inner classes still make sense and are essential in some scenarios
- Capturing variables
  - Lambda expressions and anonymous inner class now able to capture “effectively final” local variables
  - Local variable is effectively final if its initial value is never changed
  - Lambda expressions close over values, not variables

Syntax and semantics (part 3)

- Lexical scoping
  - Determining the meaning of names and `this`
  - Inherited members in inner classes can shadow outer declarations
  - Lambda expressions do not inherit any names from its supertype
  - Lambda expressions do not introduce a new level of scoping, they are lexically scoped
  - Anonymous inner classes still make sense and are essential in some scenarios
- Capturing variables
  - Lambda expressions and anonymous inner class now able to capture “effectively final” local variables
  - Local variable is effectively final if its initial value is never changed
  - Lambda expressions close over values, not variables
  - Implications on memory management

Syntax and semantics (part 3)

- Lexical scoping
  - Determining the meaning of names and `this`
  - Inherited members in inner classes can shadow outer declarations
  - Lambda expressions do not inherit any names from its supertype
  - Lambda expressions do not introduce a new level of scoping, they are lexically scoped
  - Anonymous inner classes still make sense and are essential in some scenarios
- Capturing variables
  - Lambda expressions and anonymous inner class now able to capture “effectively final” local variables
  - Local variable is effectively final if its initial value is never changed
  - Lambda expressions close over values, not variables
  - Implications on memory management
  - Beneficial for functional style of programming (immutability and reduction)

Default and static interface methods

- *Solving the interface evolution problem*

Default and static interface methods

- *Solving the interface evolution problem*
- *“APIs are like stars; once introduces, they stay with us forever.”*
  - Practical API Design, Jaroslav Tulach, Apress, 2008

Default and static interface methods

- *Solving the interface evolution problem*
- *“APIs are like stars; once introduces, they stay with us forever.”*
  - Practical API Design, Jaroslav Tulach, Apress, 2008
- Default methods
  - Previously called virtual extension methods or defender methods

Default and static interface methods

- *Solving the interface evolution problem*
- *“APIs are like stars; once introduces, they stay with us forever.”*
  - Practical API Design, Jaroslav Tulach, Apress, 2008
- Default methods
  - Previously called virtual extension methods or defender methods
  - Default methods have an implementation that is inherited by classes that do not override it

Default and static interface methods

- *Solving the interface evolution problem*
- *“APIs are like stars; once introduces, they stay with us forever.”*
  - Practical API Design, Jaroslav Tulach, Apress, 2008
- Default methods
  - Previously called virtual extension methods or defender methods
  - Default methods have an implementation that is inherited by classes that do not override it
  - Do not count in functional interfaces as abstract methods

Default and static interface methods

- *Solving the interface evolution problem*
- *“APIs are like stars; once introduces, they stay with us forever.”*
  - Practical API Design, Jaroslav Tulach, Apress, 2008
- Default methods
  - Previously called virtual extension methods or defender methods
  - Default methods have an implementation that is inherited by classes that do not override it
  - Do not count in functional interfaces as abstract methods
  - Changing the rules of API design

Default and static interface methods

- *Solving the interface evolution problem*
- *“APIs are like stars; once introduces, they stay with us forever.”*
  - Practical API Design, Jaroslav Tulach, Apress, 2008
- Default methods
  - Previously called virtual extension methods or defender methods
  - Default methods have an implementation that is inherited by classes that do not override it
  - Do not count in functional interfaces as abstract methods
  - Changing the rules of API design
- Static interface methods
  - Allow helper methods that are specified to an interface to live with the interface rather that in a side class

Default and static interface methods

- *Solving the interface evolution problem*
- *“APIs are like stars; once introduces, they stay with us forever.”*
  - Practical API Design, Jaroslav Tulach, Apress, 2008
- Default methods
  - Previously called virtual extension methods or defender methods
  - Default methods have an implementation that is inherited by classes that do not override it
  - Do not count in functional interfaces as abstract methods
  - Changing the rules of API design
- Static interface methods
  - Allow helper methods that are specified to an interface to live with the interface rather that in a side class
  - E.g. factory methods for `Comparator<T>`

Stream API

- Stream
  - Represents a sequence of values
  - Exposes a set of aggregate operations on those values

Stream API

- Stream
  - Represents a sequence of values
  - Exposes a set of aggregate operations on those values
- Comparison with collections:
  - No storage
  - Functional nature
  - Laziness-seeking
  - Bounds optional

Stream API

- Stream
  - Represents a sequence of values
  - Exposes a set of aggregate operations on those values
- Comparison with collections:
  - No storage
  - Functional nature
  - Laziness-seeking
  - Bounds optional
- Laziness
  - Stream supports eager and lazy operations
  - Can use short-circuiting to stop processing once they can determine the final result

Stream API

- Stream
  - Represents a sequence of values
  - Exposes a set of aggregate operations on those values
- Comparison with collections:
  - No storage
  - Functional nature
  - Laziness-seeking
  - Bounds optional
- Laziness
  - Stream supports eager and lazy operations
  - Can use short-circuiting to stop processing once they can determine the final result
- Parallelism
  - Easy, based on fork-join framework

Performance

- Lambda expressions vs anonymous inner classes
  - Linkage vs. Class loading
  - Capture vs. Instantiation
  - Invocation vs. Invocation

Performance

- Lambda expressions vs anonymous inner classes
  - Linkage vs. Class loading
  - Capture vs. Instantiation
  - Invocation vs. Invocation
- JDK 8 b115
  - Lambda linkage performance
    - 3-10%: use reflection instead of ASM to manipulate parameter types
    - *6-10%:* initialize generated class earlier
- JDK 8 b 117
  - Lambda linkage performance
    - 5-10%: caching `MethodType`'s descriptor string

More on Project Lambda

- We did not cover:
  - Method references
  - Translation of lambda expressions
- Links:
  - http://openjdk.java.net/projects/lambda/
  - http://www.jcp.org/en/jsr/detail?id=335
  - https://jdk8.java.net/download.html
  - http://www.jcp.org/en/jsr/detail?id=337
  - http://cr.openjdk.java.net/~briangoetz/lambda/lambda-state-final.html
  - http://cr.openjdk.java.net/~briangoetz/lambda/lambda-libraries-final.html
  - https://github.com/orfjackal/retrolambda

Q & A

BARCLAYS

Wrap up

- Project Lambda is about:
  - Language changes
    - Lambda Expressions
    - Functional Interfaces
    - Target Typing
    - Lexical Scoping
    - Capturing Variables
    - Default and Static Interface Methods
    - Method and Constructor References
  - API changes
    - Stream API
  - Mindset changes
    - Moving to functional style of programming

Thank you for attention

BARCLAYS

Disclaimer

CONFLICTS OF INTEREST BARCLAYS IS A FULL SERVICE INVESTMENT BANK. In the normal course of offering investment banking products and services to clients. Barclays may act in several capacities (including issuer, market maker, underwriter, distributor, index sponsor, swap counterparty and calculation agent) simultaneously with respect to a product, giving rise to potential conflicts of interest which may impact the performance of a product.

NOT RESEARCH This document is from a Barclays Trading and/or Distribution desk and is not a product of the Barclays Research department. Any views expressed may differ from those of Barclays Research.

BARCLAYS POSITIONS Barclays, its affiliates and associated personnel may at any time acquire, hold or dispose of long or short positions (including hedging and trading positions) which may impact the performance of a product.

FOR INFORMATION ONLY THIS DOCUMENT IS PROVIDED FOR INFORMATION PURPOSES ONLY AND IT IS SUBJECT TO CHANGE. IT IS INDICATIVE ONLY AND IS NOT BINDING.

NO OFFER Barclays is not offering to sell or seeking offers to buy any product or enter into any transaction. Any transaction requires Barclays' subsequent formal agreement which will be subject to internal approvals and binding transaction documents.

NO LIABILITY Barclays is not responsible for the use made of this document other than the purpose for which it is intended, except to the extent this would be prohibited by law or regulation.

NO ADVICE OBTAIN INDEPENDENT PROFESSIONAL ADVICE BEFORE INVESTING OR TRANSACTING. Barclays is not an advisor and will not provide any advice relating to a product. Before making an investment decision, investors should ensure they have sufficient information to ascertain the legal, financial, tax and regulatory consequences of an investment to enable them to make an informed investment decision.

THIRD PARTY INFORMATION Barclays is not responsible for information stated to be obtained or derived from third party sources or statistical services.

PAST & SIMULATED PAST PERFORMANCE Any past or simulated past performance (including back-testing) contained herein is no indication as to future performance.

OPINIONS SUBJECT TO CHANGE All opinions and estimates are given as of the date hereof and are subject to change. Barclays is not obliged to inform investors of any change to such opinions or estimates.

NOT FOR RETAIL This document is being directed at persons who are professionals and is not intended for retail customer use.

IMPORTANT DISCLOSURES For important regional disclosures you must read, click on the link relevant to your region. Please contact your Barclays representative if you are unable to access.

EMEA EMEA Disclosures APAC APAC Disclosures U.S. US Disclosures

IRS CIRCULAR 230 DISCLOSURE: Barclays does not provide tax advice. Please note that (i) any discussion of US tax matters contained in this communication (including any attachments) cannot be used by you for the purpose of avoiding tax penalties; (ii) this communication was written to support the promotion or marketing of the matters addressed herein; and (iii) you should seek advice based on your particular circumstances from an independent tax advisor.

CONFIDENTIAL This document is confidential and no part of it may be reproduced, distributed or transmitted without the prior written permission of Barclays.

ABOUT BARCLAYS Barclays Bank PLC offers premier investment banking products and services to its clients through Barclays Bank PLC. Barclays Bank PLC is authorised by the Prudential Regulation Authority and regulated by the Financial Conduct Authority and the Prudential Regulation Authority and is a member of the London Stock Exchange. Barclays Bank PLC is registered in England No. 1026167 with its registered office at 1 Churchill Place, London E14 5HP.