JDT Embraces Lambda Expressions

Srikanth Sankaran
IBM India
Stephan Herrmann
GK Software
Noopur Gupta
IBM India
Java 8 features:

- JSR335 - Project Lambda
  - Lambda Expressions & Method References
  - Extensions to
    Overload resolution & Type inference
  - Support for “code carrying” interface methods.
- JSR308 - Type Annotations
- JEP120 - Repeating Annotations
- JEP118 - Method Parameter Reflection
- JSR269 - Pluggable Annotation Processor API & javax.lang.model API enhancements for Java 8
Eclipse Luna I-build (with Java 8 support)
[http://download.eclipse.org/eclipse/downloads/]

Eclipse Kepler feature patch for Java 8
[http://download.eclipse.org/eclipse/updates/4.3-P-builds/]

Eclipse features for Java 8:
- Compiler & Incremental Builder
- Code completion
- Code selection/navigation
- Search engine
- Code formatter
- Code reconciler
- Type hierarchy viewer
- Type annotations based static null analysis
- AST APIs
- Programmatic AST rewriting API support
- Java Model Support
Java 8 effort in numbers:

- Number of commits: 1400+
- Number of authors: 31
- First commit: 25th May 2012
- Last commit: 16th March 2014
- Tasks completed: 872
- Alpha testing since 21st October 2013
- Beta testing since 1st February 2014
- Defects reported by user community: 140+
  - Resolved: 132
  - Deferred: 8 (most with workarounds)
- Major projects verified to build successfully: JRE8, OpenJFX, Eclipse SDK...
A sneak peek:
- Technical Challenges
- Demo
- Process Challenges
Lambda Expressions

- Lambda expression is basically
  - an anonymous method
  - with possibly "captured" enclosing state
  - enables paradigm of “code-as-data”

- At the grammar level, lambdas are expressions
  - Can occur anywhere an expression can

- Legally however,
  - Lambda needs a context that defines a "target type"
  - Everywhere else will have to be rejected by the compiler
  - Contexts that provide target type:
    Assignment, initialization, casts, method/ctor invocation, lambda expression body, conditional expressions and return statements
Target Typing Examples

```java
IResourceChangeListener listener = (event) -> handleCheeseMove(event);

listener = (event) -> { handleEvent(event); };

Object o = (IResourceChangeListener) (event) -> { handleEvent(event); };

workspace.addResourceChangeListener(
    (event) -> handleCheeseMove(event), IResourceChangeEvent.CHEESE_MOVE);

return (event) -> handleCheeseMove(event);

IResourceChangeListener listener = swallowEvent ?
    (event) -> {} : (event) -> handleCheeseMove(event);

Callable<Runnable> callable = () -> () -> System.out.println("Lambda chained");
```
Functional Interfaces

- The target type must be a functional interface

- Functional interface
  - Declares a single abstract method
  - Some methods don't count:
    - Default & static methods - they are not abstract
    - May freely redefine java.lang.Object's methods
  - May be tagged with `@FunctionalInterface` to express intent

- So the lambda object implements the functional interface

- Another way of constructing objects
  - apart from the SE7 ways
    (new, reflection, serialization, clone ...)
Challenges in Parsing Java 8

- New and significant parsing challenges.
- Changes to very "busy" productions.
- Tokens following '(' can be:
  - a type, an expression or a lambda parameter list
- Ambiguity cannot always be resolved until -> is seen
- Makes it hard to parse (impossible ?) with a fixed look ahead parser.

- JDT parser
  - LALR(1) parser generated by Jikes parser generator
  - At the bottom of a very tall & broad tool stack
  - Parser, CompletionParser, SelectionParser, MatchLocatorParser, IndexingParser, SourceElementParser, CommentRecorderParser, CodeSnippetParser ...

- Replacing parser technology is highly disruptive, expensive, destabilizing.
Challenges in Parsing Java 8 - Solution

- Grammar refactoring/restructuring proved futile.
- Shift/reduce and/or reduce/reduce conflicts

- Rather than build a parser for $G(L)$, build one for $G'(L')$ such that
  - $G'$ is LALR(1)
  - $L'$ - $L$ is {} for "user encodable" programs.

- Accomplished via injection of "synthetic" tokens

```
LambdaExpression ::= LambdaParameters '->' LambdaBody
LambdaParameters  -> BeginLambda LambdaParameterList
LambdaParameterList -> '(' FormalParameterListopt ')'`
LambdaBody  -> Expression
LambdaBody  -> Block
```

- What happens if a program contains

```
I i = BeginLambda (x) -> {}; ??
```
Challenges in Parsing Java 8 – Solution (contd)

- JDT infrastructure supports mini/micro parsers on grammar “fragments”

- Upon scanning a '(' the scanner
  - halts the main parser,
  - runs a special micro parser on a "Reconnaissance mission"
  - injects the synthetic token to steer the parser if "VanguardParser" spots a lambda parameter list

- Many smarts exist to minimize such lookahead
Overload Resolution

- Chicken and egg situations.
- Overload resolution influenced by argument types.
- For poly expression arguments, type cannot be ascertained until target type is known.
- Target type cannot be known until target method is known!
Overload Resolution

- Chicken and egg situations.
- Overload resolution influenced by argument types.
- For poly expression arguments, type cannot be ascertained until target type is known.
- Target type cannot be known until target method is known!

- "All problems in computer science can be solved by another level of indirection"
Overload Resolution

- Chicken and egg situations.
- Overload resolution influenced by argument types.
- For poly expression arguments, type cannot be ascertained until target type is known.
- Target type cannot be known until target method is known!

"All problems in computer science can be solved by another level of indirection"

- Lambda expression in the absence of a target type resolves to a PolyTypeBinding
- PolyTypeBinding's encode enough information about the lambda expression to serve as a proxy type until eventual resolution.
Convolutions to Compiler Architecture

- **Java 7 world allows compiler phases to be partitioned neatly:**
  - Scanning + Parsing => Parse tree
  - + Resolution => Resolved parse tree
  - + Data/control flow analysis => Flow analyzed parse tree
  - + Code generation => Class files.

- **Optimal CPU cycles and memory management per client needs.**

- **Java 8: Need information from the traditional phase N+1 right during phase N.**
  - value - void compatibility: not determinable via AST visitors
    - () -> { throw new Exception(); } is **both** void and value compatible.
    - Thrown but not caught exceptions required for type inference.

- **JDT infrastructure unprepared for such convolutions.**

- **Still an unsolved problem.**

- **GA release makes a strong guarantee that**
  - No erroneous program would be accepted and vice versa, but ... Error reporting can be suboptimal.
Changes to Indexer & Search Engine

- JDT indexer is “word based”
  
  E.g: For a method invocation, index file encodes method name and parameter count.

- Upon search, index query pulls up the documents with potential matches.

- Documents resolved and searched for actual matches.

- For a lambda there is nothing to encode!

- Until resolution that is - i.e indexer should first resolve the document.

- Some times, even unmodified files should be scheduled for reindexing.
Changes to Type System

- While not directly related to JSR 335, Java 8 in general
  - Changes the very notion of a type.
  - Some subsystems should view `@NonNull String != @ReadOnly String`
  - while others should view them as being the same.
  - Abstractions need to support either world view efficiently.

- In JDT prior to Java 8, types are "interned"
  - `String is a String is a String`

- `==` and `!=` determine type equality - all over the place

- Everyone of those is places is a potential bug now

- Searching for needles in haystack ?

  Solution: Let the compiler find the needles.
Entire lambda body must be recovered - not just declarations.

Body can influence overload resolution which can influence elided parameter type.
Type Inference
Goal: Brevity

- What we do in Java 7 (instance of an anonymous class):

```java
new BinaryOperator<Integer>() {
    @Override
    public Integer apply(Integer i1, Integer i2) {
        return i1 + i2;
    }
};
```

- What we want in Java 8 (lambda):

```java
(i1, i2) -> i1+i2;
```

- What can be omitted:
  - the functional interface
  - its generic type arguments
  - the function signature
  - ...

- How come this works?
Goal: Reuse

- Lambdas are typically arguments to generic library functions

```java
public static <T, K, U, M extends Map<K, U>>
Collector<T, ?, M> toMap(Function<K, ? super T, ? extends K> keyMapper,
Function<U, ? super T, ? extends U> valueMapper,
BinaryOperator<U> mergeFunction)
```

- Would you prefer to call:
  ```java
  Collectors.<Person, String, Integer>toMap(..)
  ```
- Or:
  ```java
  Collectors.toMap(..)
  ```
- Again: omitting “unnecessary” boiler plate
- Similar for instantiation of generic class (Java 7)
  ```java
  Collector<String, Integer> coll = new MyCollector<>();
  ```
- How does it work?
Solution: Type Inference

- Let the compiler collect “all available” type information
- Let it deduce the types
  - instantiation of type parameters of
    - generic methods
    - diamond instantiations
    - functional interface
  - lambda parameters
- If inference finds a solution, it is type safe by definition
- If it doesn't find a solution ...? ...
What you don’t want to know

- Solving this little example

```java
Map<String, Integer> test3(Stream<Person> persons) {
    return persons.collect(Collectors.toMap(
        p -> p.getLastName(),
        p -> p.getAge(),
        (i1, i2) -> i1 + i2));
}
```
What you don’t want to know

- Produces these constraints:

```java
TypeBound K#3 :> java.lang.String
Dependency K#3 = K#3
TypeBound K#3 = java.lang.String
TypeBound K#3 = java.lang.Object
Dependency R#0 = java.util.Map<K#3, U#4>
Dependency R#0 = java.util.Map<java.lang.String, U#4>
Dependency R#0 = java.util.Map<K#3, java.lang.Integer>
TypeBound R#0 = java.util.Map<java.lang.String, java.lang.Integer>
TypeBound R#0 = java.lang.Object
TypeBound T#2 :> Person
Dependency T#2 = T#2
TypeBound T#2 = Person
TypeBound T#2 = java.lang.Object
Dependency Map<K#3, U#4>#6 = java.util.Map<K#3, U#4>
Dependency Map<K#3, U#4>#6 = java.util.Map<java.lang.String, U#4>
Dependency Map<K#3, U#4>#6 = java.util.Map<K#3, java.lang.Integer>
TypeBound Map<K#3, U#4>#6 = java.util.Map<java.lang.String, java.lang.Integer>
Dependency Map<K#3, U#4>#6 = R#0
TypeBound Map<K#3, U#4>#6 = java.lang.Object
TypeBound Map<K#3, U#4>#6 = java.util.Map<java.lang.String, java.lang.Integer>
TypeBound A#1 = java.lang.Object
TypeBound A#1 = java.lang.Object
TypeBound U#4 :> java.lang.Integer
TypeBound U#4 = java.lang.Integer
Dependency U#4 = U#4
TypeBound U#4 = java.lang.Object
TypeBound ?#5 = java.lang.Object
Dependency ?#5 = A#1
TypeBound ?#5 = java.lang.Object
```
What you might want to know about Inference

- **Information considered:**
  - During overload resolution:
    - receiver type
    - argument types
  - Eventually
    - target type
  - Never
    - what's right of the next dot
  - Nested inference
    - expressions nested as call arguments
    - combine inner and outer to one big inference
What you might want to know about Inference

- **Information considered:**
  - During overload resolution:
    - receiver type
    - argument types
  - Eventually
    - target type
  - Never
    - what's right of the next dot
  - Nested inference
    - expressions nested as call arguments
    - combine inner and outer to one big inference

```java
Map<String, Integer> test3(Stream<Person> persons) {
    return persons.collect(Collectors.toMap(
        p -> p.getLastName(),
        p -> p.getAge(),
        (i1, i2) -> i1 + i2));
}
```
What if inferences doesn't see that ...?

- To make sure you know what you're doing:
  - insert explicit types

- There's no guarantee that inference finds every solution
  - “reduction is not completeness-preserving”
  - incompleteness is part of the specification

- Inference is constraint solving
  - unlike common sense reasoning
  - if it fails, it's hard to say, why

The method foo(T1) in the type C is not applicable for the arguments (..)

Type mismatch: cannot convert from Foo to Bar

Better error reporting: future work

- Simplification for users
  - bought by a tremendous complexity in the compiler
Default Methods - Intention

- **Intention**
  - support evolution of interfaces in libraries: add methods to an interface w/o breaking implementors

- still legal:
Default Methods - Compatibility

- Designed for compatibility during evolution
- For the price of compatibility
  - Recipe for disaster:
    - implement java.util.List → compile against JRE 7 → OK
    - upgrade to JRE 8 but compile as 1.7
  - Undefined compiler behavior
    - 1.7 compiler cannot handle default methods (from .class)
    - exact answer depends on compiler implementation details
    - there is no specification
    - ecj and javac differ
    - assimilation is difficult

x The type MyList<E> must implement the inherited abstract method Collection<E>.stream()
Java 8 in Eclipse - Demo
Getting Started

- Download Eclipse (with Java 8 support)

- Add a JRE for JDK 1.8
  \((Windows > Preferences > Java > Installed JREs)\)

- Set Compiler compliance level to 1.8
  \((Windows > Preferences > Java > Compiler)\)

- Create a Java Project with JDK 1.8 JRE
Convert anonymous class to lambda expression

String[] names = { "Bob", "Dave", "Charlie", "Alice" };

Arrays.sort(names, new Comparator<String>() {
    @Override
    public int compare(String o1, String o2) {
        return o1.compareTo(o2);
    }
});

Ctrl + 1

String[] names = { "Bob", "Dave", "Charlie", "Alice" };

Arrays.sort(names, new Comparator<String>() {
    @Override
    public int compare(String o1, String o2) {
        return o1.compareTo(o2);
    }
});

Convert to lambda expression
- Extract to local variable (replace all occurrences)
- Extract to local variable

Press 'Ctrl+Enter' to fix all problems of same category

Enter

String[] names = { "Bob", "Dave", "Charlie", "Alice" };

Arrays.sort(names, (o1, o2) -> o1.compareTo(o2));
Change lambda body to block

String[] names = { "Bob", "Dave", "Charlie", "Alice" };
Arrays.sort(names, (o1, o2) -> o1.compareTo(o2));

Ctrl + 1

String[] names = { "Bob", "Dave", "Charlie", "Alice" };
Arrays.sort(names, (o1, o2) -> o1.compareTo(o2));

Enter

String[] names = { "Bob", "Dave", "Charlie", "Alice" };
Arrays.sort(names, (o1, o2) -> {
    return o1.compareTo(o2);
});
Change lambda body to expression

```java
String[] names = { "Bob", "Dave", "Charlie", "Alice" };  
Arrays.sort(names, (o1, o2) -> {
    return o1.compareTo(o2);
});
```

Press Ctrl + 1 to change the body block to an expression:

```
String[] names = { "Bob", "Dave", "Charlie", "Alice" };  
Arrays.sort(names, (o1, o2) -> o1.compareTo(o2));
```
Convert lambda expression to anonymous class

```java
String[] names = { "Bob", "Dave", "Charlie", "Alice" };
Arrays.sort(names, (o1, o2) -> {
    System.out.println(o1 + " , " + o2);
    return o1.compareTo(o2);
});
```

Press `Ctrl+Enter` to fix all problems of same category.

```java
String[] names = { "Bob", "Dave", "Charlie", "Alice" };
Arrays.sort(names, new Comparator<String>() {
    @Override
    public int compare(String o1, String o2) {
        System.out.println(o1 + " , " + o2);
        return o1.compareTo(o2);
    }
});
```
Clean Up – Functional interface instances
Lambda Expression:
```
Arrays.sort(names, (o1, o2) -> o1.compareTo(o2));
```

Method Reference:
```
Arrays.sort(names, String::compareTo);
```

Hover to see the functional method
Refactorings - updated to support Java 8

Demo Examples:

- Change method signature
- Rename
- Extract method
- Extract method in interfaces

Adornments for default, static and abstract methods in an interface

‘default’ access modifier replaced with ‘package’ access modifier in UI
Type hierarchy

Search

Content assist

New Formatter options for lambda:

- Braces
- Whitespace

Debug support
The Path Behind
Game of Quality Assurance

abstractly capture the concepts

spec

javac
Game of Quality Assurance

spec

abstractly capture the concepts

javac

ecj

implement the spec, the full spec, and nothing but the spec
Game of Quality Assurance

abstractly capture the concepts

spec

implement the spec, the full spec, and nothing but the spec

spec is not implementable (incomplete) (Apr – Sep 2013)

javac

ecj
Game of Quality Assurance

spec

abstractly capture the concepts

javadoc

implement the spec, the full spec, and nothing but the spec

ecj

compare behavior
Game of Quality Assurance

spec

abstractly capture the concepts

javac

ecj

implement the spec, the full spec, and nothing but the spec

deviation

compare behavior
Game of Quality Assurance

abstractly capture the concepts

spec

implement the spec, the full spec, and nothing but the spec

bug?

bug?

bug?

deviation

compare behavior

javac

ecj
Game of Quality Assurance

abstractly capture the concepts

implement the spec, the full spec, and nothing but the spec

compare behavior
Game of Quality Assurance

abstractly capture the concepts

Far too risky to cram late into 8 (Nov 2013)

Far too risky to cram late into 8 (Nov 2013)

implement the spec, the full spec, and nothing but the spec

compare behavior
Raw Types in Type Inference

Where are unchecked conversions allowed?

- List <: List<String> ?
  - spec: NO
  - javac: maybe, sometimes
  - ecj ?

Raw types

- supported in Java 5 to allow migration
- some programs using raw types will eventually be rejected

Serving two masters – JLS & javac?

- should we exactly copy javac behavior?
- is precisely implementing the spec better?
- competition helps improve quality
Status Reached

- New spec is more precise than previous versions
  - will be a good judge in case of doubt

- Ecj implementation is very close to the spec
  - bugs can be easily identified and fixed
  - 1 unimplemented branch, if you see:
    “Problem detected during type inference: ...”

you win! :)

The Path Behind
Why am I doing this?

- **2002: Research on programming languages beyond OO**
  - Object Teams, realized as an extension of Java / JDT
Why am I doing this?

- **2002:** Research on programming languages beyond OO
  - Object Teams, realized as an extension of Java / JDT

- **2010:** Object Teams becomes an Eclipse project
Why am I doing this?

- **2002**: Research on programming languages beyond OO
  - Object Teams, realized as an extension of Java / JDT

- **2010**: Object Teams becomes an Eclipse project

- **2010**: I joined the JDT team
Why am I doing this?

• **2002**: Research on programming languages beyond OO  
  – Object Teams, realized as an extension of Java / JDT

• **2010**: Object Teams becomes an Eclipse project

• **2010**: I joined the JDT team

• **2010**: I joined GK Software
Why am I doing this?

● 2002: Research on programming languages beyond OO
   – Object Teams, realized as an extension of Java / JDT

● 2010: Object Teams becomes an Eclipse project
   – programming language ahead of the mainstream

● 2010: I joined the JDT team

● 2010: I joined GK Software
Why am I doing this?

- **2002:** Research on programming languages beyond OO
  - Object Teams, realized as an extension of Java / JDT

- **2010:** Object Teams becomes an Eclipse project
  - programming language ahead of the mainstream

- **2010:** I joined the JDT team
  - any improvement affects a huge number of developers

- **2010:** I joined GK Software
Why am I doing this?

- **2002**: Research on programming languages beyond OO
  - Object Teams, realized as an extension of Java / JDT

- **2010**: Object Teams becomes an Eclipse project
  - programming language ahead of the mainstream

- **2010**: I joined the JDT team
  - any improvement affects a huge number of developers

- **2010**: I joined GK Software
  - company supports my involvement
What's cool about working on JDT?

• **Java 8 special**
  - working with leading Java experts

• **“Normal” operation**
  - get your free daily Java puzzler, via bugzilla
  - working on a component of high quality standards
  - working in a team with a good tradition of reviewing
  - giving back to the Eclipse community

• **Is it just me (crazy)?**
  - a few minutes of sporadic involvement doesn't cut it
  - many (academics) have the know how
  - many are actually working on / with this code
Dramatis personæ - The usual suspects

- Andy Clement
- Steve Francisco
- Michael Rennie
- Olivier Thomann
- Curtis Windatt
- Jesper S. Møller
- Stephan Herrmann
- Markus Keller
- Dani Megert
- Jay Arthanareeswaran
- Deepak Azad
- Shankha Banerjee
- Anirban Chakarborty
- Vikas Chandra
- Noopur Gupta
- Ayushman Jain
- Manju Mathew
- Manoj Palat
- Srikanth Sankaran
- Sarika Sinha
- Walter Harley
- David Williams
- Jesper S. Møller
- Stephan Herrmann
- Markus Keller
- Dani Megert
- Jay Arthanareeswaran
- Deepak Azad
- Shankha Banerjee
- Anirban Chakarborty
- Vikas Chandra
- Noopur Gupta
- Ayushman Jain
- Manju Mathew
- Manoj Palat
- Srikanth Sankaran
- Sarika Sinha
Thank you
Evaluate this session

1. Sign-in: [www.eclipsecon.org](http://www.eclipsecon.org)

2. Select session from schedule

3. Evaluate: +1  0  -1