jax® 2012

Business Technology | Days

Lambda expressions in Java: a compiler writer's perspective

Maurizio Cimadamore
Type-system engineer, Oracle Corporation

The following is intended to outline our general product direction. It is intended for information purposes only, and may not be incorporated into any contract. It is not a commitment to deliver any material, code, or functionality, and should not be relied upon in making purchasing decisions.
The development, release, and timing of any features or functionality described for Oracle's products remains at the sole discretion of Oracle.

BACKGROUND

Where are we?

- Multicore hardware is now the default
  - Moore’s law delivering more cores, not faster cores
- We must learn to write software that parallelizes gracefully
  - Right now, the serial code and the parallel code for a given operation don’t look anything like each other
  - Fork-join (added in Java SE 7) is a good start, but not enough

Problem: external iteration

```
List<Student> students = ...
double highestScore = 0.0;
for (Student s : students) {
  if (s.gradYear == 2011) {
    if (s.score > highestScore) {
      highestScore = s.score;
    }
  }
}
```

☹	Client controls iteration
☹	*Inherently serial:* iterate from beginning to end
☹	Not thread-safe (shared mutable variable)

Internal iteration w/ lambdas

```
SomeCoolList<Student> students = ...
double highestScore =
  students.filter(Student s -> s.getGradYear() == 2011)
          .map(Student s -> s.getScore())
          .max();
```

☺	Library-based iteration
☺	Traversal might be done *in parallel*
☺	Thread-safe (stateless)
☺	More readable and less error prone!

Why closures for Java?

- Provide libraries a path to multicore
  - Today, developer's primary tool for computing over aggregates is the for loop – which is fundamentally serial
  - Parallel-friendly APIs need internal iteration
  - Internal iteration needs a concise code-as-data mechanism
- Empower library developers
  - Closures are useful for all kinds of libraries, serial or parallel
  - Enable a higher degree of cooperation between libraries and client code
- It's about time!
  - Java is the lone holdout among mainstream OO languages at this point to not have closures
  - Adding closures to Java is no longer a radical idea

LAMBDA EXPRESSIONS

Lambda Expressions

- One construct, several syntactic forms:

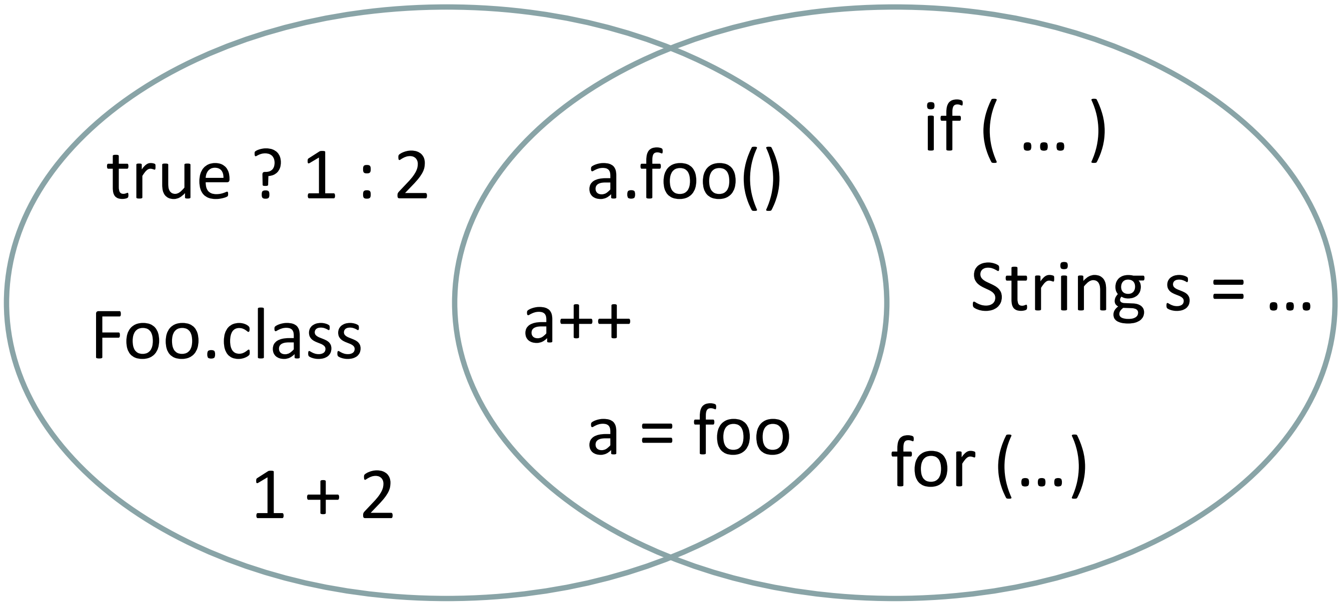
```
LambdaExpression:
  TypeParametersopt LambdaParameters '->' LambdaBody

LambdaExpressionAfterCast:
  LambdaParameters '->' LambdaBody

LambdaParameters:
  Identifier
  '(' InferredFormalParameterList ')'
  '(' FormalParameterListopt ')'

InferredFormalParameterList:
  Identifier
  InferredFormalParameterList ',' Identifier

LambdaBody:
  Expression
  Block
```

Explicit vs. Implicit parameters

- Lambda parameter types can be omitted
  - Compiler to the rescue: implicit parameters will be inferred from the context
  - Keywords only allowed on explicit parameters

explicit	*implicit*
`(int x) -> x+1`	`x -> x+1`
`(int x, int y) -> x+y`	`(x,y) -> x+y`
`(final String msg) -> { log(msg); }`	`msg -> { log(msg); }`

Expression vs. Statement



Expression lambdas

- Expression lambda
  - Body is an expression which is also the return value of the lambda

expr	`int x -> x*x`
	`(int x,int y)-> x + y`
	`Object o -> o.toString()`

Statement lambdas

- Statement lambda
  - Body is an ordinary block
  - A statement lambda can return (where return means ‘local’ return)

stmt	`boolean c-> { assert c; }`
	`(int x, int y)-> { if (cond) return x; return y; }`
	`Object o-> { System.out.println(o); }`

Expression or statement?

- An expression statement can be used in both statement and expression lambdas
  - Semantics could depend on the context
  - Syntax helps to disambiguate (no semi-colon tricks!)

expr	`(Collection<?> c, Object o) -> c.add(o);`
stmt	`(Collection<?> c, Object o) -> { c.add(o); }`

To return or not to return?

- A lambda might/might not have a return value
  - A statement lambda that does not return is *void-compatible*
  - A statement lambda that has one or more return values is *value-compatible*
  - An expression lambda is *always* value-compatible

void	`(String s) -> { System.out.println(s); }` `(boolean cond) -> { if (cond) return; }`
value	`(String s)-> s` `(String s)-> { return s; }`

Don't call me, I'll call you!

- A lambda expression can be executed when the scope in which it has been created is no longer available in the execution stack
  - This can happen if the lambda is saved and then executed at a later stage (laziness)
- Implications:
  - Restrictions on local variable capture
  - Restrictions on jumps

Local variable capture

- Lambda expressions can refer to *effectively final* local variables from the enclosing scope
  - An effectively final variable meets the requirements for final variables (i.e. assigned once), even if not declared as such
  - Close over *values*, not over *variables*!

✗	`long before = 1;`
	`... (File p) -> p.lastModified() <= before;`
	`before = 3;`

✓	`long before = 1;` `... (File p) -> p.lastModified() <= before;`

Jumps

- **`break`/`continue`** are allowed if the target is within the lambda expression
  - Non-local jumps are disallowed

✗

```
for (Object o : elems) {
  … ()-> { break; };
}
```

✓

```
()-> switch(s) {
        case "Hello!": break;
}
```

Correspondence Principle

[…] the underlying semantic notions for both parameter and definition mechanisms are simply expression evaluation (of an actual parameter or the right-hand side of a definition) and identifier binding (of a formal parameter or the left-hand side of a definition.) […] For any parameter mechanism, an analogous definition mechanism is possible, and vice versa. This is known as the principle of correspondence.

from R. D. Tennent's Principle's of Programming Languages (1981)

- For a given expression **`expr`**, **`lambda expr`** *should be semantically equivalent*.
- Implications:
  - Shadowing
  - Meaning of names (i.e. **`this`**)

Scope of lambda parameters

- Lambda parameters share same scope with locals variables defined in the enclosing scope
  - Error when lambda declares parameter with same name as a local in the enclosing scope

✗	`void shadowTest(Object i) {`
	`... (int i) -> i*i;`
	`}`

✓	`void shadowTest(Object i) {`
	`... (int i2) -> i2*i2;`
	`}`

Meaning of names

- The meaning of names are the same inside the lambda as outside
  - **`this`** refers to the enclosing object, not the lambda itself
  - Easier than inner classes:
    no ambiguity between enclosing vs. inherithed symbols

✓	`class Foo {`
	`    ... () -> { Foo f = this; };`
	`}`

✓	`class Foo {`
	`    ... () -> toString();`
	`}`

METHOD REFERENCES

Internal iteration w/ lambdas (reloaded)

```
SomeCoolList<Student> students = ...
double highestScore =
  students.filter(Student s -> s.getGradYear() == 2011)
          .map(Student s -> s.getScore())
          .max();
```

☺	Library-based iteration
☺	Traversal might be done *in parallel*
☺	Thread-safe (stateless)
☺	More readable and less error prone!
😐	Accidental horizontal verbosity

Internal iteration w/ method references

```
SomeCoolList<Student> students = ...
double highestScore =
  students.filter(Student s -> s.getGradYear() == 2011)
          .map(Student::getScore)
          .max();
```

☺	Library-based iteration
☺	Traversal might be done *in parallel*
☺	Thread-safe (stateless)
☺	More readable and less error prone!
☺	Reuse of existing code!

Method References

- Syntactic shortcut for creating a lambda expression out of an existing method/constructor
  - Many flavors of method references:

```
MethodReference:
 ExpressionName '::' NonWildTypeArgumentsopt Identifier
 Primary '::' NonWildTypeArgumentsopt Identifier
 ReferenceType '::' NonWildTypeArgumentsopt Identifier

ConstructorReference:
 ClassType '::' NonWildTypeArgumentsopt 'new'
```

Overview of Method References

	Qualifier Expression toplevel type	inner type	expression
Static identifier	static method reference	static method reference	static method reference
Instance identifier	unbound method reference	unbound method reference	bound method reference
`new`	toplevel constructor reference	inner constructor reference	N/A

Name

Desugaring method references

- Three kinds of method references
  - *Static* - access static methods
  - *Bound* - access instance method *explicitly*
  - *Unbound* - access instance method *implicitly*

	::	λ
static	`Logger::log`	`(String msg) -> Logger.log(msg)`
bound	`getPerson()::name`	`final Person p = getPerson();` `() -> p.name()`
unbound	`Person::name`	`(Person p) -> p.name()`

Desugaring constructor references

- Two kinds of method references
  - *Toplevel* - access toplevel constructor
  - *Inner* - access inner constructor *implicitly*
- Desugaring of inner constructor reference depends on whether an enclosing instance is in scope!

	::	λ
toplevel	`Person::new`	`(String name) -> new Person(name)`
inner$_1$	`Inner::new`	`() -> Outer.this.new Inner();`
inner$_2$	`Inner::new`	`(Outer o) -> o.new Inner();`

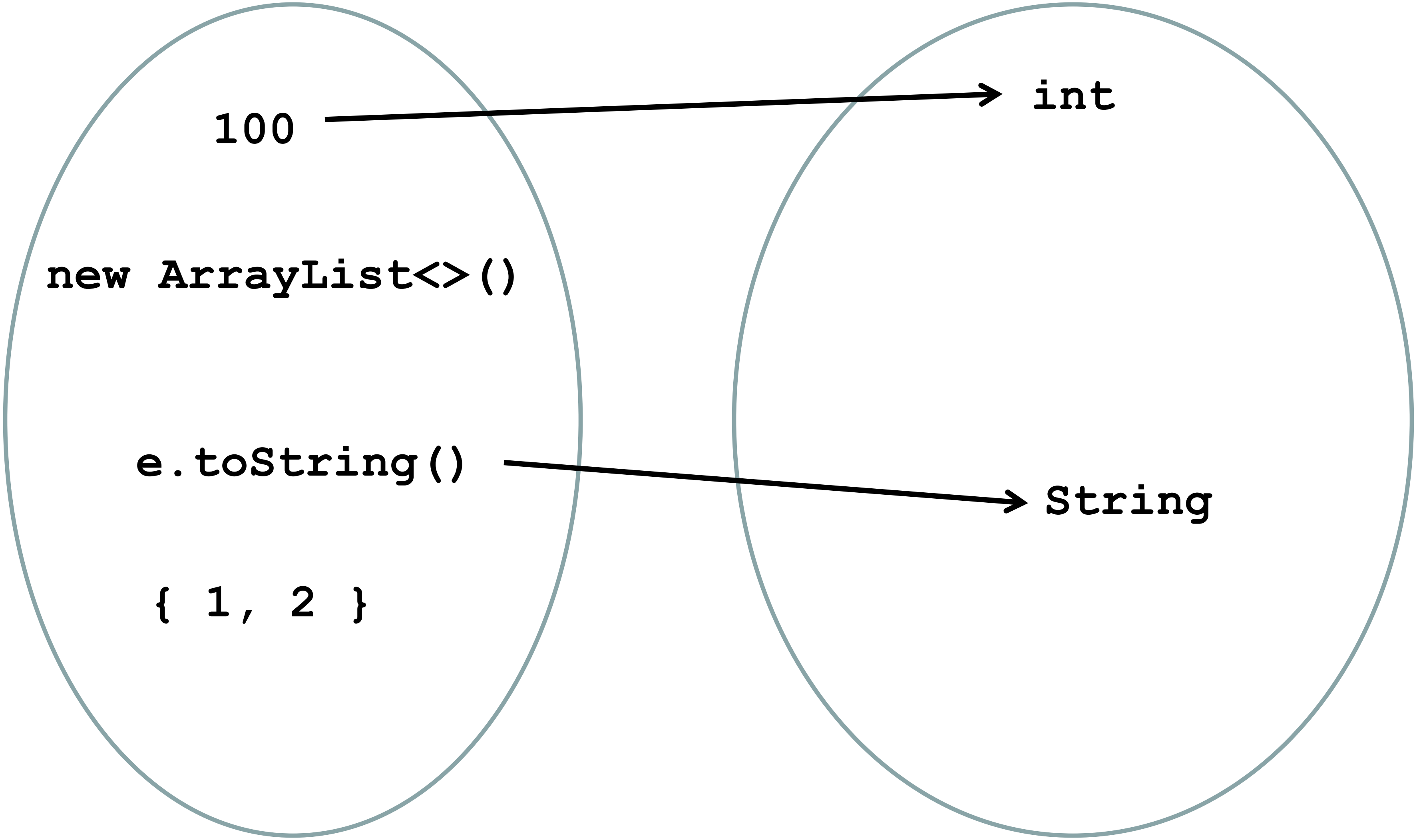
FUNCTIONAL INTERFACES

Not all expressions are created equal!

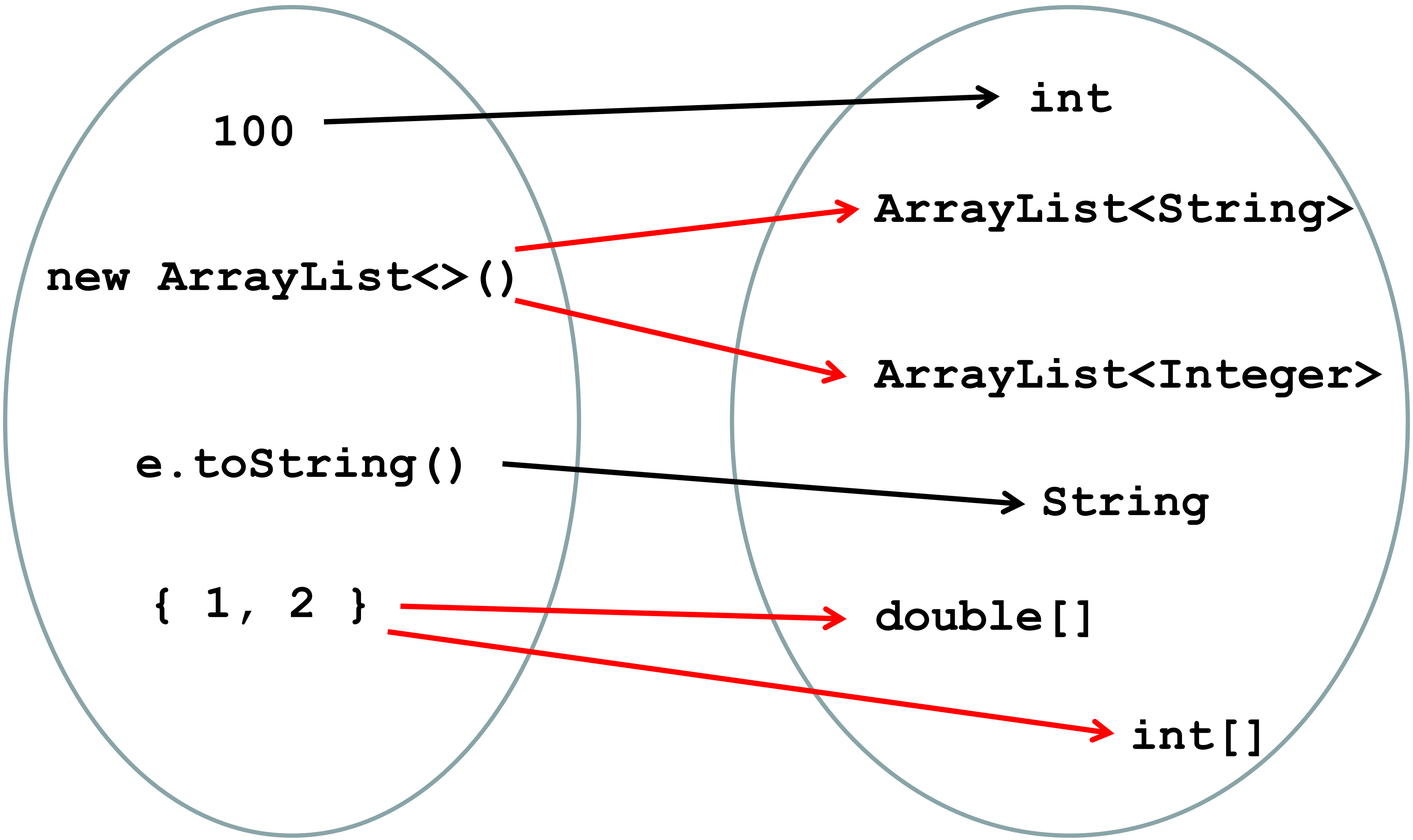
```
100

new ArrayList<>()

e.toString()

{ 1, 2 }
```

?

Standalone expressions



Poly expressions



Lambda and method references as poly expressions

- Lambda expression/method references are *just* new kinds of poly expressions
  - The type of lambda/method reference cannot be computed in isolation (i.e. w/o a target type)
  - A lambda/method reference can be used whenever a *compatible* functional interface is expected!

λ	`Runnable r = () -> { System.out.println("hi"); };`
::	`Runnable r = System::gc;`

Functional interfaces

A functional interface is an interface that has just one abstract method, and thus represents a single function contract. In some cases, this "single" method may take the form of multiple abstract methods with override-equivalent signatures inherited from superinterfaces; in this case, the inherited methods logically represent a single method.

from the Project Lambda EDR

Where are my arrow types?

- For years, we’ve used single-method interfaces to represent functions and callbacks
  - A *functional interface* is an interface with one method
- Functional interfaces provide an hook to switch between nominal/structural type information:
  - A functional interface is just a (nominal) interface type…
  - Each functional interface is associated with a functional descriptors that carries *structural* type information:
    - Argument types
    - Return type
    - Thrown types

Functional interfaces in the JDK

Interface	*Descriptor*
`Comparator<T>`	`boolean compare(T x, T y);`
`FileFilter`	`boolean accept(File x);`
`Callable<T>`	`T call();`
`Runnable`	`void run();`
`ActionListener`	`void actionPerformed(ActionEvent e);`

Lambda as functional descriptors

- A lambda λ is said to be compatible with a functional descriptor *F* iff:
  - Parameter types in λ matches the parameter types in *F*
  - Return value(s) in λ is *compatible* with the return type of *F*
  - The checked exceptions thrown by λ are a subset of the exceptions declared by *F*
- Implicit lambda parameters are inferred from argument types in *F*

Lambda as functional descriptors

✓	`FileFilter ff = (File f) -> f.isDirectory();`
✓	`Comparator<String> cs = (s1,s2) -> s1.length() - s2.length();`
✗	`Runnable r = () -> { throw new Exception(); }`
✗	`Comparator<String> cs = (s1,s2)-> true;`
✗	`FileFilter ff = (String s)-> s.endsWith("Hello!");`

The long arm of void-compatibility

- If return type of the functional descriptor is **`void`**, the lambda must be void-compatible!

✓	`Runnable r = () -> { System.out.println("Hello!"); }`
✗	`Runnable r = () -> System.out.println("Hello")!`

TARGET-TYPING IN METHOD CONTEXT

Internal iteration w/ lambdas and method references (reloaded)

```
SomeCoolList<Student> students = ...
double highestScore =
  students.filter(Student s -> s.getGradYear() == 2011)
          .map(Student::getScore)
          .max();
```

☺	Library-based iteration
☺	Traversal might be done *in parallel*
☺	Thread-safe (stateless)
☺	More readable and less error prone!
☺	Reuse of existing code!
😐	Redundant type-information

Internal iteration w/ lambdas, method references and target-typing

```
SomeCoolList<Student> students = ...
double highestScore =
  students.filter(s -> s.getGradYear() == 2011)
            .map(Student::getScore)
            .max();
```

☺	Library-based iteration
☺	Traversal might be done *in parallel*
☺	Thread-safe (stateless)
☺	More readable and less error prone!
☺	Reuse of existing code!
☺	Don't repeat yourself!

Target-typing in Java

- Lambda expressions/method references are a new form of poly expressions
  - They cannot be type-checked w/o a target type
- This is a problem as the target-type information is not always propagated (as in JDK 7)

✓

```java
List<String> ls = new ArrayList<>();
```

✗

```java
List<String> ls = true ? new ArrayList<>() : new ArrayList<>();
```

✗

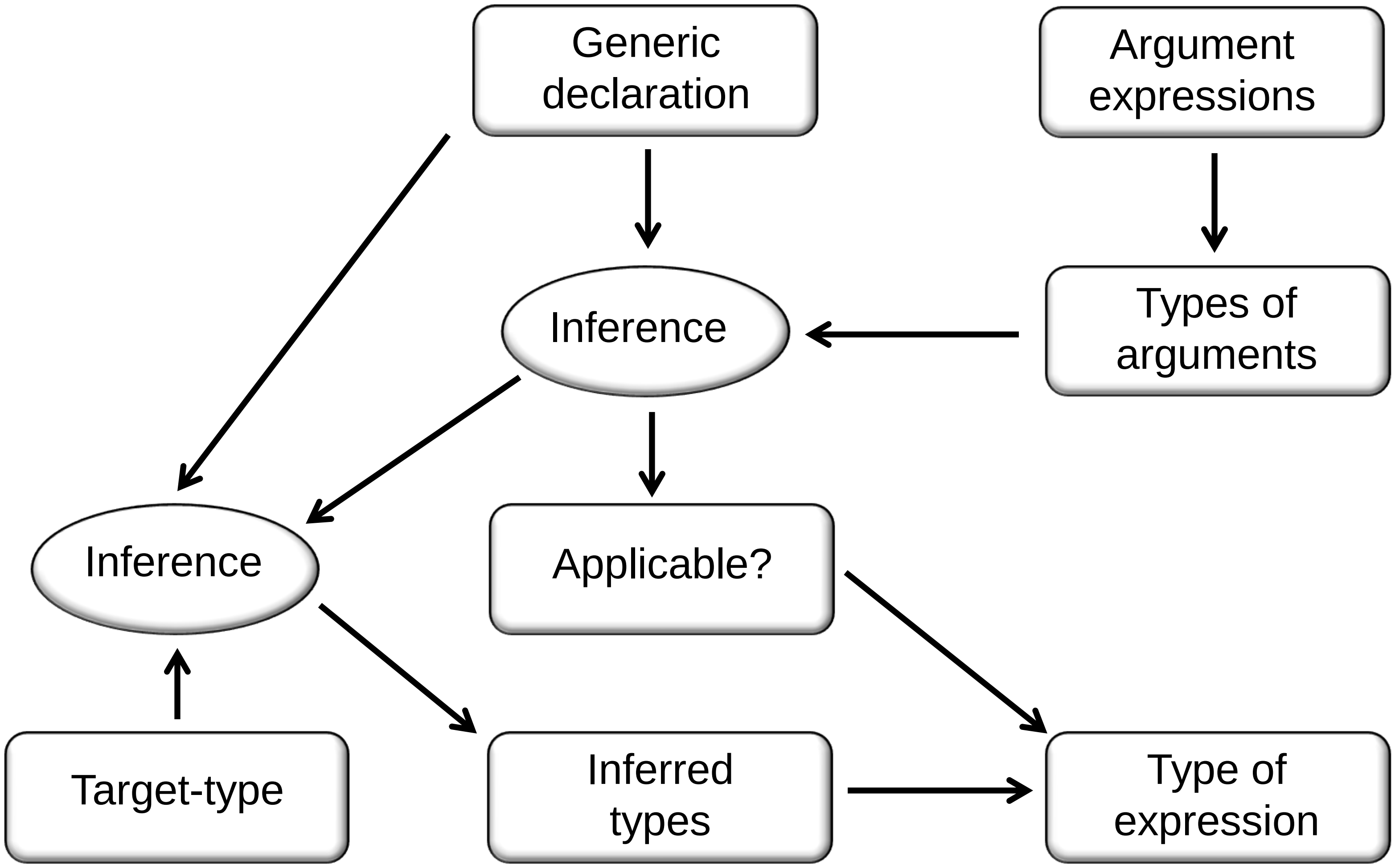
```java
void m(List<String> ls) { ... }
m(new ArrayList<>());
```

Consistent use of target-typing

- Goal: it should be possible to use a lambda/method reference in all contexts where a target-type exists

`=`	`squareMapper = x -> x*x;`
`[]`	`Mapper<Integer>[] mappers = { x -> x*x, x -> x+x; }`
`return`	`return x -> x*x;`
`()`	`numbers.map(x -> x*x)`
`?`	`square ? x -> x*x : x -> x+x;`

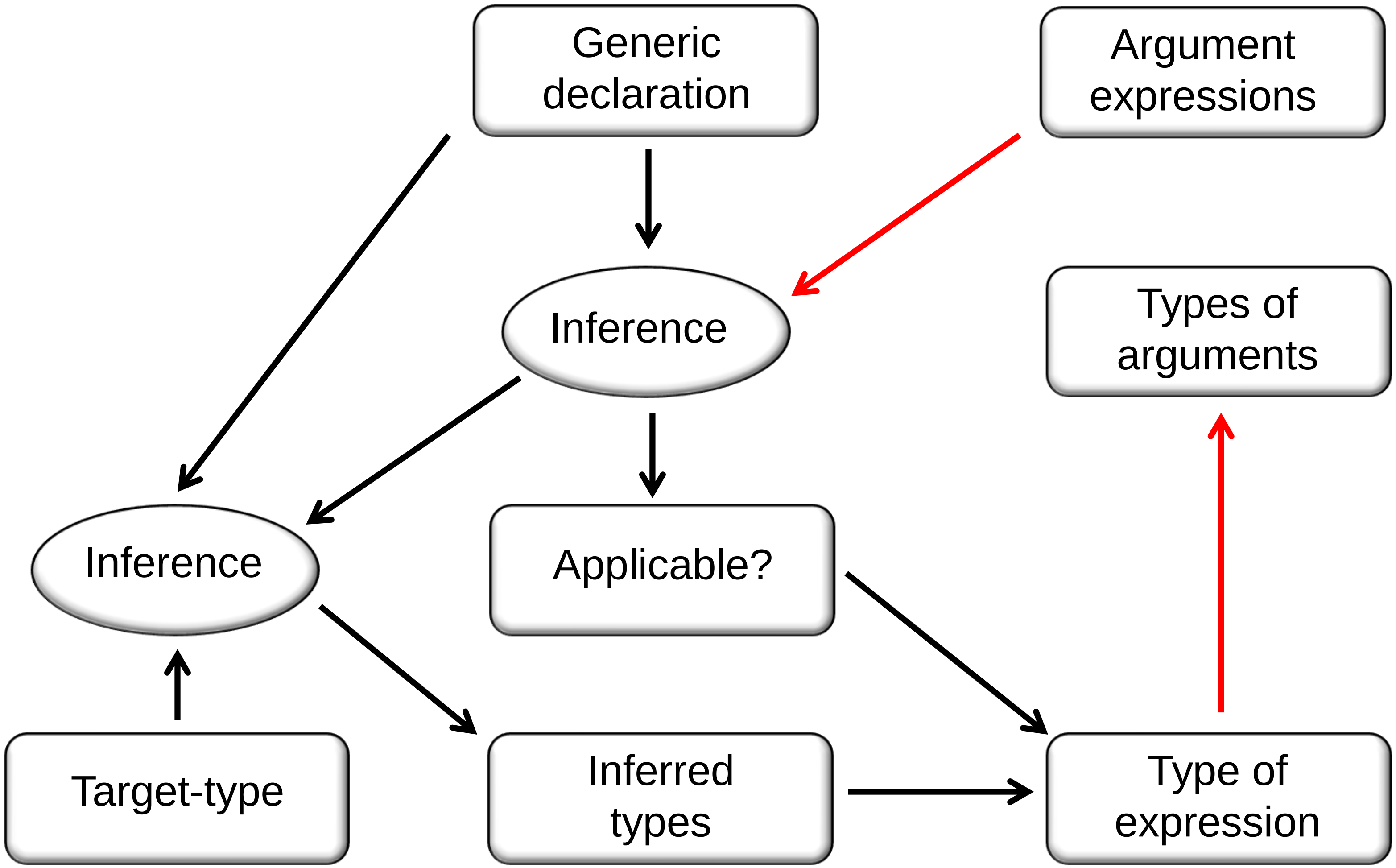
Overload resolution in JDK 7



Target-typing and overload resolution

- Lambda/method reference in method context can be checked more than once
  - Speculative type-checking - the lambda/method reference is type-checked against each possible target-type
- Presence of multiple overload candidates
  - checking of lambda/method reference is used to discard some candidates
- Structural most specific check
  - When multiple compatible functional interfaces found
- Let go of the assumption that we must know argument types *ahead* of overload resolution!

Overload resolution revisited



Speculative type-checking

- Lambda/method reference in method context can be checked more than once
  - Speculative type-checking - the lambda/method reference is type-checked against each possible target-type

`m(x -> x.toString());`		
?	`void m(SAM1 s1)`	`String apply1(Integer i);`
?	`void m(SAM2 s2)`	`Integer apply2(Integer i);`
?	`void m(SAM3 s3)`	`Object apply3(Integer i);`

The art of pruning

- Overload candidates are *filtered* using the information derived when checking lambda/method reference
  - If an overloaded method does not satisfy those constraints, it is dropped from the applicable set

`m(x -> x.toString());`		
?	`void m(SAM1 s1)`	`String apply1(Integer i);`
✗	`void m(SAM2 s2)`	`Integer apply2(Integer i);`
?	`void m(SAM3 s3)`	`Object apply3(Integer i);`

Structural most specific

- When multiple overload candidates are available, a most specific signature is selected
  - If formals are functional interfaces, a full structural check is performed on the underlying descriptors

`m(x -> x.toString());`		
✓	`void m(SAM1 s1)`	`String apply1(Integer i);`
✗	`void m(SAM2 s2)`	`Integer apply2(Integer i);`
✗	`void m(SAM3 s3)`	`Object apply3(Integer i);`

Target-typing and generic methods

- An overload candidate could be a generic method
  - The target-type might depend on yet-to-infer inference variables!
- Lambda expressions in method context cannot be type-checked because of the presence of inference variables in the target-type
  - Such lambdas are said to be *stuck*
- Possible solutions (still under consideration):
  - Inference errors
  - Wait for lambda to become unstuck

A cycle too far?

- It looks like there is cycle in the inference machinery:
  - The compiler needs type-information on actual arguments in order to proceed with method type-inference
  - The compiler needs a fully instantiated target-type in order to type-check certain actual arguments

```
m(x->1, "Hello!");
```

✘	`<Z> void m(SAM<Z> s, Z z)`	`interface SAM<X> {`
		`   int apply(X i);`
		`}`

Out-of-order method checking

- Alternatively, let go of the assumption that arguments are checked from left to right
  - i.e. instantiate all inference variables in the target-types first using constraints derived from unstuck arguments
  - As arguments become unstuck, type-check them until no further progress can be made

```
m(x->1, "Hello!");
```

✓	`<Z> void m(SAM<Z> s, Z z)`	`interface SAM<X> { int apply(X i); }`

WRAP UP

Project Lambda / JSR-335 Status

- Project Lambda started December 2009
  - Explorations done through OpenJDK
- JSR-335 filed November 2010
  - Prototype compiler developed in OpenJDK
- Current status
  - EDR draft #1 now public, available at http://www.jcp.org/en/jsr/summary?id=335
  - Compiler prototype binaries available at http://jdk8.java.net/lambda/
    - Includes VM support for extension method dispatch!
- Feedback is welcome!

Conclusion

- Adding closures to Java is key to promote fluent, functional-oriented code that is multicore-ready
- One goal, two constructs
  - Lambda expressions
  - Method references
- Functional interface allows smooth interoperability with existing code
- The devil is in the details
  - Evolving an existing language is always hard – lots of interactions with existing features
  - Novel target-typing support requires overload resolution/inference overhaul

Maurizio Cimadamore
Type-system engineer, Oracle Corporation