JDK 9 Language and Tooling Features

Joseph D. Darcy (@jddarcy)
Vicente A. Romero
Java Platform Group, Oracle
October 2015

Safe Harbor Statement

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.

Survey: Which JDK train are you using in production?

- JDK 9 EA with Jigsaw
- JDK 9 EA
- JDK 8
- JDK 7
- JDK 6
- JDK 5.0
- JDK 1.4.2 or earlier

JDK 9 Overview

JDK 9 Information

- Current schedule: GA September 22, 2016
- Early access binaries + docs, updated weekly: https://jdk9.java.net/
- Early access binaries with Jigsaw: https://jdk9.java.net/jigsaw/
- OpenJDK:
  - Project: http://openjdk.java.net/projects/jdk9/
  - Mailing list: http://mail.openjdk.java.net/mailman/listinfo/jdk9-dev
  - Source code: http://hg.openjdk.java.net/jdk9/dev/
  - Adoption: http://mail.openjdk.java.net/pipermail/adoption-discuss/
  - JEPs (JDK Enhancement Proposals) used for project tracking

I NEED
YOU
to try
JDK 9
EA builds

Capsule summary of modularity language impact in JDK 9

- Modules bundle together one or more packages and offer stronger encapsulation than jars
  - `module-info.java` files to declare dependencies between modules
  - Changes to `javac` command line to find types in modules as well as in jars
- For (much) more information see sessions:
  - *Introduction to Modular Development* [CON5118], Wednesday, Oct 28, 1:00 pm
  - *Advanced Modular Development* [CON6821], Thursday, Oct 29, 2:30 pm
  - *Prepare for JDK 9* [CON5107], Monday
  - *Project Jigsaw: Under the Hood* [CON6823], Monday

Outline

- User-facing features
  - `jshell`
  - JavaDoc.Next
  - Cross compiling and the `javac -release` flag
  - Milling Project Coin (five tiny language improvements)
  - Deprecation and imports
- Reengineering `javac`
  - Tiered Attribution
- Q & A

jshell

JEP 222: jshell: The Java Shell (Read-Eval-Print Loop)

A historical perspective

- Lisp systems decades back provided a read-evaluate-print-loop (REPL) to interact with the language
- Similar functionality is found in Ruby and Python as well as Scala, Groovy, and Clojure
- What about Java?

jshell – new command in $JDK/bin in JDK 9

Project Kulla: http://openjdk.java.net/projects/kulla/

- Less ceremony for students learning Java
- Less formal way for experienced developers to
  - Explore using a new API
  - Experiment with new language features
- Leverages many existing JDK technologies
- Dedicated session: *jshell*: The New Interactive Java Language Shell for JDK 9 [CON7823], Wednesday, Oct 28, 8:30 a.m.

Teaser DEMO

JavaDoc.Next

http://openjdk.java.net/projects/javadoc-next/

HTML 5

JEP 224: HTML5 Javadoc

- The javadoc tool has historically generated HTML 4.01 output
- HTML 5 is a new HTML standard, finalized October 2014
  - Richer semantic structure
  - New attributes defined
- Use

  ```
  javadoc -html5 …
  ```

  to opt-into the new output
- (Still uses frames-style layout)

HTML 5 logo from www.w3.org/html/logo/index.html

Doclint and package filtering

- Doclint, added in JDK 8, performs various structural and semantics checks on javadoc tags in various categories:
  - **Accessibility**: conditions that may be flagged by an accessibility checker, such as obsolete attributes or headers out of sequence
  - **HTML**: HTML conformance issues (vary with DOCTYPE), e.g. anchor already defined, bad start-end tag matching
  - **Missing**: missing comment or missing tag within a comment (e.g. not all parameters have a @param tag)
  - **Reference**: the target of an @see, @link, @throws, etc. is invalid
  - **Syntax**: malformed HTML in comments

Configuring doclint

- Doclint checks can be run as part of `javadoc` or as part of `javac`; (both used in JDK build)
- HTML checks are appropriate for chosen version of HTML output
- Checks can be limited to chosen language accessibility (`public`, `protected`, …)
- Checks can be enabled / disabled per category
  - Example, for public and protected types, do all checks except for reference: `-Xdoclint:all/protected,-reference`
- Checks can be limited to selected packages (new in JDK 9):
  - To check `java.*` and `javax.*`: `'-Xdoclint/package:java.*,javax.*'`
  - To exclude `example.com`: `'-Xdoclint/package:-example.com.*'`

Simplified Doclet API

JEP 221: Simplified Doclet API

- Doclets are plugins to `javadoc`; standard doclet most commonly used
- Doclet APIs
  - Don’t follow current best-practices for API design
  - Uses an inaccurate and hard to evolve language model
- Replace old Doclet APIs with newer APIs with better functionality
  - Use `javax.lang.model` API for language model
  - Use DocTree API (`com.sun.source.doctree`)
- Simpler and more compact

javadoc search: coming soon to a future JDK 9 build…

JEP 225: Javadoc Search

- By default javadoc output will have a search box
  - Client-side implementation in JavaScript
  - Indexes package names, type names, member names
  - New tag to add other index items

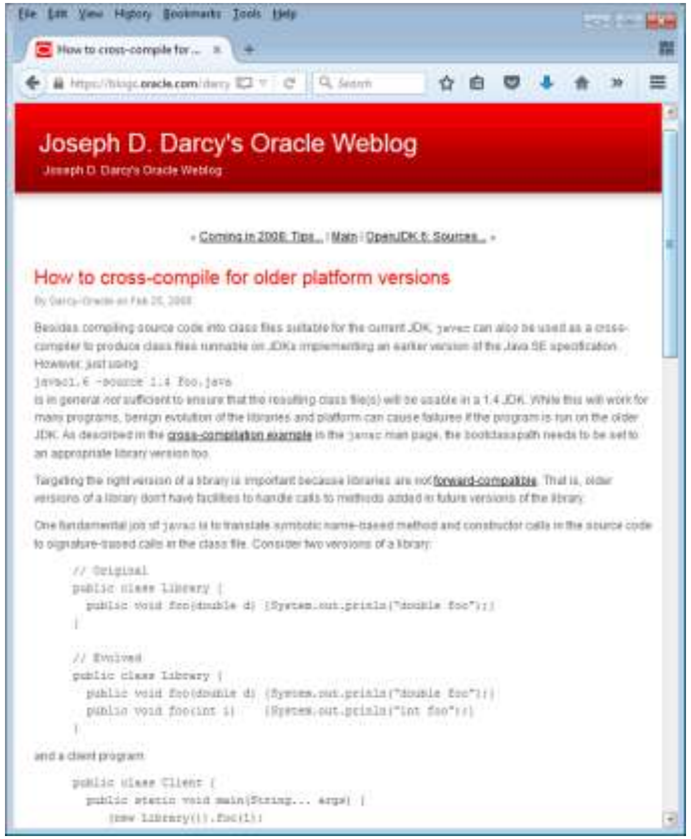
DEMO

Cross-compiling and the `–release` flag

Cross-compiling to older releases

https://blogs.oracle.com/darcy/entry/how_to_cross_compile_for

- To use `javac` to compile to older release *N*, must set
  - -source *N*
  - -target *N*
  - -bootclasspath jdk*N*/lib/rt.jar


File Edit View History Bookmarks Tools Help
How to cross-compile for ...
https://blogs.oracle.com/darcy
Search

Joseph D. Darcy's Oracle Weblog

Joseph D. Darcy's Oracle Weblog

« Coming in 2008: Tips... | Main | OpenJDK 6: Sources... »

How to cross-compile for older platform versions

By Darcy-Oracle on Feb 25, 2008

Besides compiling source code into class files suitable for the current JDK, `javac` can also be used as a cross-compiler to produce class files runnable on JDKs implementing an earlier version of the Java SE specification. However, just using

`javac1.6 -source 1.4 Foo.java`

is in general *not* sufficient to ensure that the resulting class file(s) will be usable in a 1.4 JDK. While this will work for many programs, benign evolution of the libraries and platform can cause failures if the program is run on the older JDK. As described in the cross-compilation example in the `javac` man page, the bootclasspath needs to be set to an appropriate library version too.

Targeting the right version of a library is important because libraries are not forward-compatible. That is, older versions of a library don't have facilities to handle calls to methods added in future versions of the library.

One fundamental job of `javac` is to translate symbolic name-based method and constructor calls in the source code to signature-based calls in the class file. Consider two versions of a library:

```
// Original
public class Library {
  public void foo(double d) {System.out.println("double foo");}
}

// Evolved
public class Library {
  public void foo(double d) {System.out.println("double foo");}
  public void foo(int i)    {System.out.println("int foo");}
}
```

and a client program

```
public class Client {
  public static void main(String... args) {
    (new Library()).foo(1);
```

Policy on older releases: 1 + 3 back

JEP 182: Policy for Retiring javac -source and -target Options

- Started implementing in JDK 8
- In JDK 9, -source/-target supported values:
  - 9 (the default)
  - 8
  - 7
  - 6 (deprecated, warning issued when used)

Why the bootclasspath needs to be set

```
// JDK N
public class Library {
  public void foo(double d) {…}
}

// JDK N+1
public class Library {
  public void foo(double d) {…}
  public void foo(int i) {…}
}
```

Why the bootclasspath needs to be set

```
// JDK N
public class Library {
  public void foo(double d) {…}
}

// JDK N+1
public class Library {
  public void foo(double d) {…}
  public void foo(int i) {…}
}
```

```
public class Client {
  public static void main(String… args) {
     (new Library()).foo(1234);
  }
}
```

Why the bootclasspath needs to be set

```
// JDK N
public class Library {
  public void foo(double d) {…}
}

// JDK N+1
public class Library {
  public void foo(double d) {…}
  public void foo(int i) {…}
}
```

```
public class Client {
  public static void main(String… args) {
     (new Library()).foo(1234);
  }
}
```

If Client is compiled against JDK *N*+1 but run against JDK *N*:

Why the bootclasspath needs to be set

```
// JDK N
public class Library {
  public void foo(double d) {…}
}

// JDK N+1
public class Library {
  public void foo(double d) {…}
  public void foo(int i) {…}
}
```

```
public class Client {
  public static void main(String… args) {
     (new Library()).foo(1234);
  }
}
```

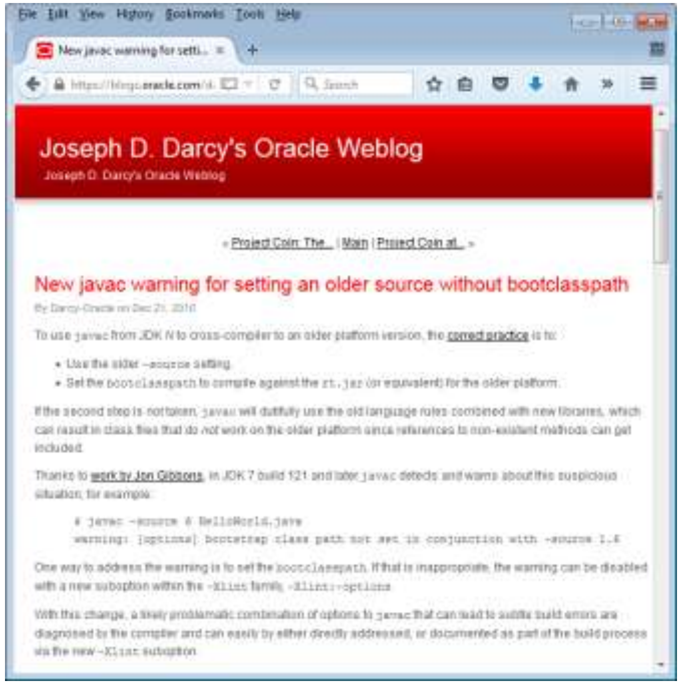
If Client is compiled against JDK *N*+1 but run against JDK *N*:

```
Exception in thread "main"
java.lang.NoSuchMethodError: Library.foo(I)V
        at Client.main(Client.java:3)
```

An explicit warning as of JDK 7

https://blogs.oracle.com/darcy/entry/bootclasspath_older_source

- `javac –cp … -source 6 -target 6 Client.java`
  `warning: [options] bootstrap class path not set in conjunction with -source 1.6`
- The warning can be suppressed with
  `-Xlint:-options`



File Edit View History Bookmarks Tools Help
New javac warning for setti...
https://blogs.oracle.com/d
Search

Joseph D. Darcy's Oracle Weblog

Joseph D. Darcy's Oracle Weblog

« Project Coin: The... | Main | Project Coin at... »

New javac warning for setting an older source without bootclasspath

By Darcy-Oracle on Dec 21, 2010

To use `javac` from JDK *N* to cross-compiler to an older platform version, the correct practice is to:

- Use the older `-source` setting.
- Set the `bootclasspath` to compile against the `rt.jar` (or equivalent) for the older platform.

If the second step is not taken, `javac` will dutifully use the old language rules combined with new libraries, which can result in class files that do *not* work on the older platform since references to non-existent methods can get included.

Thanks to work by Jon Gibbons, in JDK 7 build 121 and later `javac` detects and warns about this suspicious situation; for example:

```
# javac -source 6 HelloWorld.java
warning: [options] bootstrap class path not set in conjunction with -source 1.6
```

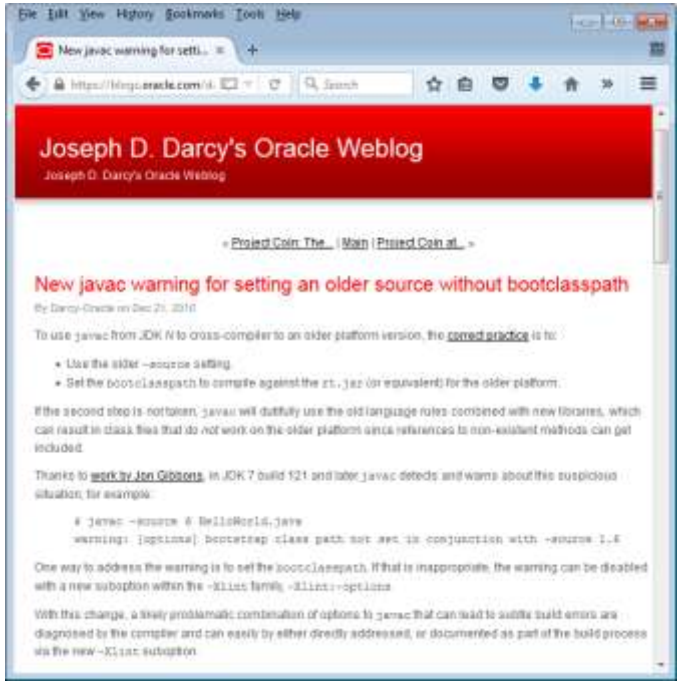
One way to address the warning is to set the `bootclasspath`. If that is inappropriate, the warning can be disabled with a new suboption within the `-Xlint` family, `-Xlint:-options`.

With this change, a likely problematic combination of options to `javac` that can lead to subtle build errors are diagnosed by the compiler and can easily by either directly addressed, or documented as part of the build process via the new `-Xlint` suboption.

An explicit warning as of JDK 7

https://blogs.oracle.com/darcy/entry/bootclasspath_older_source

- `javac –cp … -source 6 -target 6 Client.java`
  `warning: [options] bootstrap class path not set in conjunction with -source 1.6`
- The warning can be suppressed with
  `-Xlint:-options`
- But despite the warning, bugs still came in…


File Edit View History Bookmarks Tools Help
New javac warning for setti...
https://blogs.oracle.com/d
Search

Joseph D. Darcy's Oracle Weblog

Joseph D. Darcy's Oracle Weblog

« Project Coin: The... | Main | Project Coin at... »

New javac warning for setting an older source without bootclasspath

By Darcy-Oracle on Dec 21, 2010

To use `javac` from JDK *N* to cross-compiler to an older platform version, the correct practice is to:

- Use the older `-source` setting.
- Set the `bootclasspath` to compile against the `rt.jar` (or equivalent) for the older platform.

If the second step is not taken, `javac` will dutifully use the old language rules combined with new libraries, which can result in class files that do *not* work on the older platform since references to non-existent methods can get included.

Thanks to work by Jon Gibbons, in JDK 7 build 121 and later `javac` detects and warns about this suspicious situation; for example:

```
# javac -source 6 HelloWorld.java
warning: [options] bootstrap class path not set in conjunction with -source 1.6
```

One way to address the warning is to set the `bootclasspath`. If that is inappropriate, the warning can be disabled with a new suboption within the `-Xlint` family, `-Xlint:-options`.

With this change, a likely problematic combination of options to `javac` that can lead to subtle build errors are diagnosed by the compiler and can easily by either directly addressed, or documented as part of the build process via the new `-Xlint` suboption.

JDK 9’s do-what-I-want approach: -release *N*

JEP 247: Compile for Older Platform Versions

- `javac -release` *`N`* `…`
  is equivalent to
  `javac -source` *`N`* `-target` *`N`* `–bootclasspath rt`*`N`*`.jar…`
- Information about APIs of earlier releases available to `javac`
  - Stored in a compressed fashion
  - Only provide Java SE *N* and JDK *N*-exported APIs that are platform neutral
- Same set of release values *N* as for -source / -target
- Incompatible combinations of options rejected

Advantages of -release *N*

- No user need to manage artifacts storing old API information
- Should remove need to use tools like Animal Sniffer
- May use newer compilation idioms than the `javac` in older releases
  - Bug fixes
  - Speed improvements

Milling Project Coin in JDK 9

http://openjdk.java.net/jeps/213

Photo copyright 2011 Joseph D. Darcy. All rights reserved.

Project Coin features in JDK 7

Standardized under JSR 334

- Binary literals and underscores in literals
- Strings in switch
- Diamond
- Multi-catch and more precise rethrow
- `try`-with-resources
- Varargs warnings

Project Coin features in JDK 7

Standardized under JSR 334

- Binary literals and underscores in literals
- Strings in switch
- ***Diamond***
- Multi-catch and more precise rethrow
- ***`try`-with-resources***
- ***Varargs warnings***

A few small amendments in JDK 9

- Allow @SafeVargs on private instance methods.
- Allow effectively-final variables to be used as resources in the try-with-resources statement.
- Allow diamond with anonymous classes if all type arguments of the inferred types are denotable.
- Complete the removal, begun in Java SE 8, of underscore from the set of legal identifier names.
- Source-level private interface methods.

A few small amendments in JDK 9

- Allow @SafeVargs on private instance methods.
- Allow effectively-final variables to be used as resources in the try-with-resources statement.
- Allow diamond with anonymous classes if all type arguments of the inferred types are denotable.
- Complete the removal, begun in Java SE 8, of underscore from the set of legal identifier names.
- Source-level private interface methods.

Project Lambda amendments

`@SafeVarargs` on `private` instance methods

Example from JDK 5.0 and 6

See JLSv3 §4.12.2.1 – Heap Pollution

```
List<List<String>> monthsInTwoLanguages =
     Arrays.asList(Arrays.asList("January", "February"),
                   Arrays.asList("Gennaio", "Febbraio"));
```

Example from JDK 5.0 and 6

See JLSv3 §4.12.2.1 – Heap Pollution

```
List<List<String>> monthsInTwoLanguages =
     Arrays.asList(Arrays.asList("January", "February"),
                   Arrays.asList("Gennaio", "Febbraio"));
```

```
 warning: [unchecked] unchecked generic array creation
for varargs parameter of type List<String>[]
      Arrays.asList(Arrays.asList("January",
                   ^
```

Example from JDK 5.0 and 6

See JLSv3 §4.12.2.1 – Heap Pollution

```
List<List<String>> monthsInTwoLanguages =
    Arrays.asList(Arrays.asList("January", "February"),
                  Arrays.asList("Gennaio", "Febbraio"));
```

```
 warning: [unchecked] unchecked generic array creation
for varargs parameter of type List<String>[]
      Arrays.asList(Arrays.asList("January",
                     ^
```

But nothing bad actual happens *in this case*, so the error is uninformative.

Reminder: @SafeVarargs from Project Coin

- New annotation type `java.lang.SafeVarargs` and application to libs
- Summary: no longer receive uninformative unchecked compiler warnings from calling platform library methods:
  - `<T> List<T> Arrays.asList(T... a)`

  -
    ```
    <T> boolean Collections.addAll(Collection<? super T> c,
                                   T... elements)
    ```

  -
    ```
    <E extends Enum<E>> EnumSet<E> EnumSet.of(E first,
                                              E... rest)
    ```
  - `void javax.swing.SwingWorker.publish(V... chunks)`
- Removes warnings from method *call* sites

Annotation Properties

- Annotations are only inherited on *classes*, not on interfaces or methods
- Therefore, a `@SafeVarargs` annotation can only be used on methods that cannot be overridden:

Annotation Properties

- Annotations are only inherited on *classes*, not on interfaces or methods
- Therefore, a `@SafeVarargs` annotation can only be used on methods that cannot be overridden:
  - Constructors (from a certain point of view, special `static` methods)

Annotation Properties

- Annotations are only inherited on *classes*, not on interfaces or methods
- Therefore, a `@SafeVarargs` annotation can only be used on methods that cannot be overridden:
  - Constructors (from a certain point of view, special `static` methods)
  - `static` methods

Annotation Properties

- Annotations are only inherited on *classes*, not on interfaces or methods
- Therefore, a `@SafeVarargs` annotation can only be used on methods that cannot be overridden:
  - Constructors (from a certain point of view, special `static` methods)
  - `static` methods
  - `final` methods

Annotation Properties

- Annotations are only inherited on *classes*, not on interfaces or methods
- Therefore, a `@SafeVarargs` annotation can only be used on methods that cannot be overridden:
  - Constructors (from a certain point of view, special `static` methods)
  - `static` methods
  - `final` methods
  - `private` methods (omitted in JDK 7)

Language Specification updates

- JLS §9.6.4.7. @SafeVarargs
  It is a compile-time error if a variable arity method declaration that is neither `static` nor `final` is annotated with the annotation `@SafeVarargs`.

Language Specification updates

- JLS §9.6.4.7. @SafeVarargs
  It is a compile-time error if a variable arity method declaration that is neither `static` nor `final` nor `private` is annotated with the annotation `@SafeVarargs`.

Effectively final variables & `try`-with-resources

Refresher: try-with-resources

You type this:

```
try (Resource r = aa()) {
    bb(r);
} catch (Exception e) {
    cc();
} finally {
    dd();
}
```

Refresher: `try-with-resources`

You type this:

```
try (Resource r = aa()) {
    bb(r);
} catch (Exception e) {
    cc();
} finally {
    dd();
}
```

⇒

Compiler generates (approximately) this:

```
try {
    Resource r = null;
    try {
        r = aa();
        bb(r);
    } finally {
        if (r != null)
            r.close();
    }
} catch (Exception e) {
    cc();
} finally {
    dd();
}
```

Refresher: try-with-resources

You type this:

```
try (Resource r = aa()) {
    bb(r);
} catch (Exception e) {
    cc();
} finally {
    dd();
}
```

⇒

Compiler generates (approximately) this:

```
try {
    Resource r = null;
    try {
        r = aa();
        bb(r);
    } finally {
        if (r != null)
            r.close();
    }
} catch (Exception e) {
    cc();
} finally {
    dd();
}
```

It's actually more complicated because of the way exceptions from `close()` are handled.

Full disclosure: `try`-with-resources actual desugaring

```
try ResourceSpecification
  Block
```

Full disclosure: try-with-resources actual desugaring

```
try ResourceSpecification
  Block
```

⇒

```
{
  final VariableModifiers_minus_final R #resource = Expression;
  Throwable #primaryException = null;

  try ResourceSpecification_tail
    Block
  catch (Throwable #t) {
    #primaryException = t;
    throw #t;
  } finally {
    if (#resource != null) {
      if (#primaryException != null) {
        try {
          #resource.close();
        } catch(Throwable #suppressedException) {
          #primaryException.addSuppressed(#suppressedException);
        }
      } else {
        #resource.close();
      }
    }
  }
}
```

Structure of the resources to be managed

- In Java SE 7, the resources to be managed by a `try`-with-resources statement must be fresh variables declared in the statement:
  ```
  try (Resource r = …)
  ```
- The original proposal allowed general expressions that were `AutoCloseable`, but this was found to be problematic for a variety of reasons such as:
  ```
  Resource r = new Resource1(); // First resource object
  try (r) {
    r = new Resource2();         // Second resource object
  }
  // Which object(s) have had their close method called?
  ```

Refined proposal: reuse final or effectively final resources

- Instead of general expressions, just allows variables that are either:
  - `final` or
  - effectively `final`

  to be used as resources.
- Effectively `final` variables are not explicitly declared as `final`, but could be and still have the program compile, see JLS §4.12.4.
- Variables have an explicit type, avoiding the need for type inference.

Example

Instead of

```
final Resource r = new Resource();
try (Resource r2 = r) {
   …
}
```

can just use

```
final Resource r = new Resource();
try (r) {
   … // Cannot mutate r
}
```

Grammar of the specification change, JLS §14.20.3

Resource:

{VariableModifier} UnannType VariableDeclaratorId = *Expression*

Grammar of the specification change, JLS §14.20.3

Resource:

{VariableModifier} UnannType VariableDeclaratorId = Expression

VariableAccess

VariableAccess:

ExpressionName

FieldAccess

Core of the semantics of the specification change

An operand to try-with-resources statement may be a *VariableAccess* expression, which is an expression name (6.5.6) or a field access expression (15.11). The name or expression must denote a `final` or effectively `final` variable of type `AutoCloseable` which is definitely assigned before the `try`-with-resources statement; otherwise, a compile-time error occurs.

Core of the semantics of the specification change

An operand to try-with-resources statement may be a *VariableAccess* expression, which is an expression name (6.5.6) or a field access expression (15.11). The name or expression must denote a `final` or effectively `final` variable of type `AutoCloseable` which is definitely assigned before the `try`-with-resources statement; otherwise, a compile-time error occurs.

(Besides a direct construct like a local variable or method parameter, a resource to be managed could be a `final` *field in an object or a* `final static` *field of a nested class, including a construct like* `(new Foo()).finalResourceField`*).*

Diamond with anonymous classes, redux

Diamond and anonymous classes, review

```
Object o;
List<?> arg = ...;
o = new Box<>(arg);
```

```
public class Box<T> {
    private T value;

    public Box(T value) {
        this.value = value;
    }

    T getValue() {
        return value;
    }
}
```

Diamond and anonymous classes, review

```
Object o;
List<?> arg = ...;
o = new Box<>(arg);
```

The compiler needs to *infer* the type to use for Box. The language is still *statically* typed even though it is not always *explicitly* typed.

```
public class Box<T> {
    private T value;

    public Box(T value) {
        this.value = value;
    }

    T getValue() {
        return value;
    }
}
```

Inside the compiler…

```
Object o;
List<?> arg = ...;
o = new Box<>(arg);
```

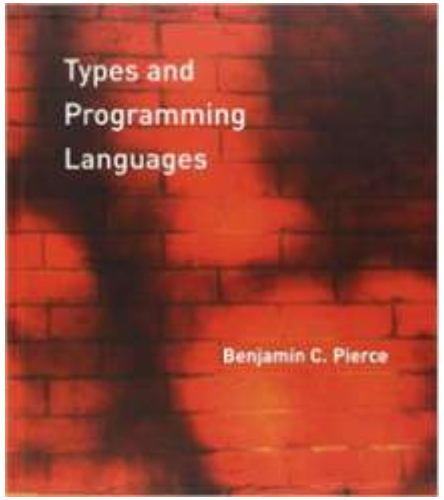
Inside the compiler…

```
Object o;
List<?> arg = ...;
o = new Box<List<capture of ?>>(arg);
```

Inside the compiler…

```
Object o;
List<?> arg = ...;
o = new Box<List<capture of ?>>(arg);
```

Capture conversion;
results in a ***non-denotable*** *type*
(For details see [JLS §5.1.10](JLS §5.1.10))



A Box can be subclassed, including anonymously

```
Object o;
List<?> arg = ...;
o = new Box<List<capture of ?>>(arg){…};
```

A Box can be subclassed, including anonymously

```
Object o;
List<?> arg = ...;
o = new Box<List<capture of ?>>(arg){…};
```

Class file expressed as lower-level source code

```
Object o;
List<?> arg = ...;
o = new Box$1(arg);

class Box$1 extends Box<List<capture of ?>>{…}
```

Class file expressed as lower-level source code

```
Object o;
List<?> arg = ...;
o = new Box$1(arg);

class Box$1 extends Box<List<capture of ?>>{…}
```

Anonymous classes translate into a new class file with a full set of attributes.

Class file expressed as lower-level source code

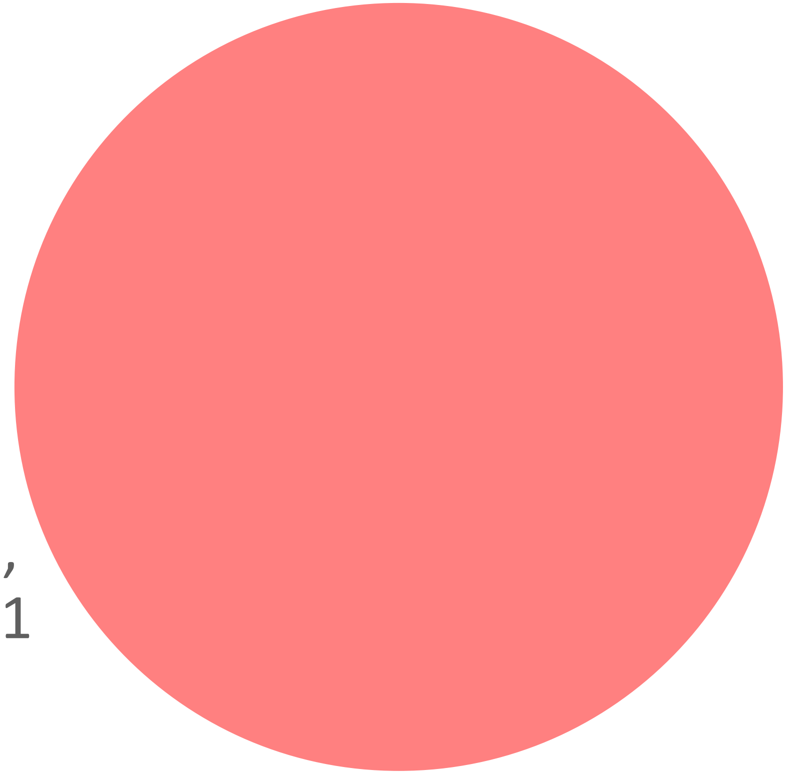
```
Object o;
List<?> arg = ...;
o = new Box$1(arg);

class Box$1 extends Box<List<capture of ?>>{…}
```

Anonymous classes translate into a new class file with a full set of attributes.

Type info needs to be stored in a `Signature` *attribute (JVMS §4.7.9)*

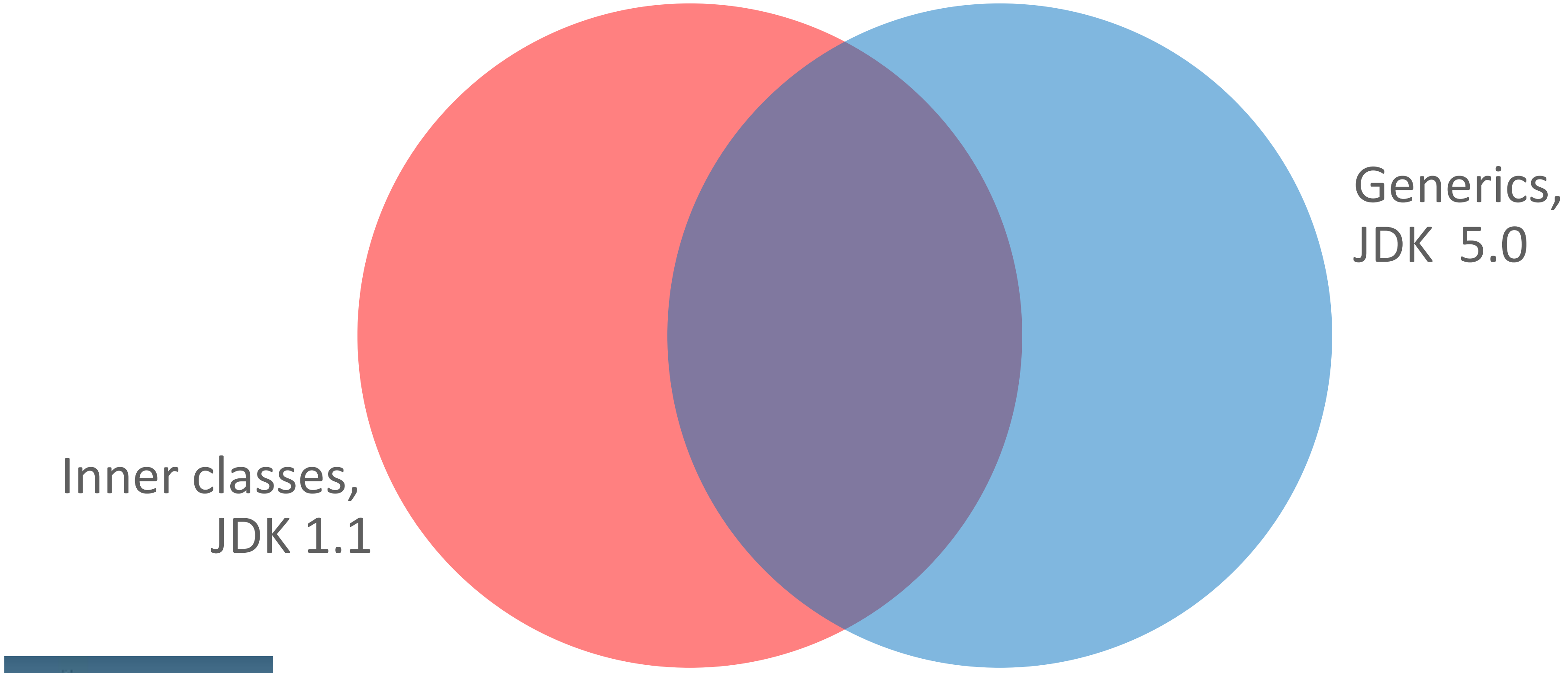
Language feature interactions over time

Language feature interactions over time

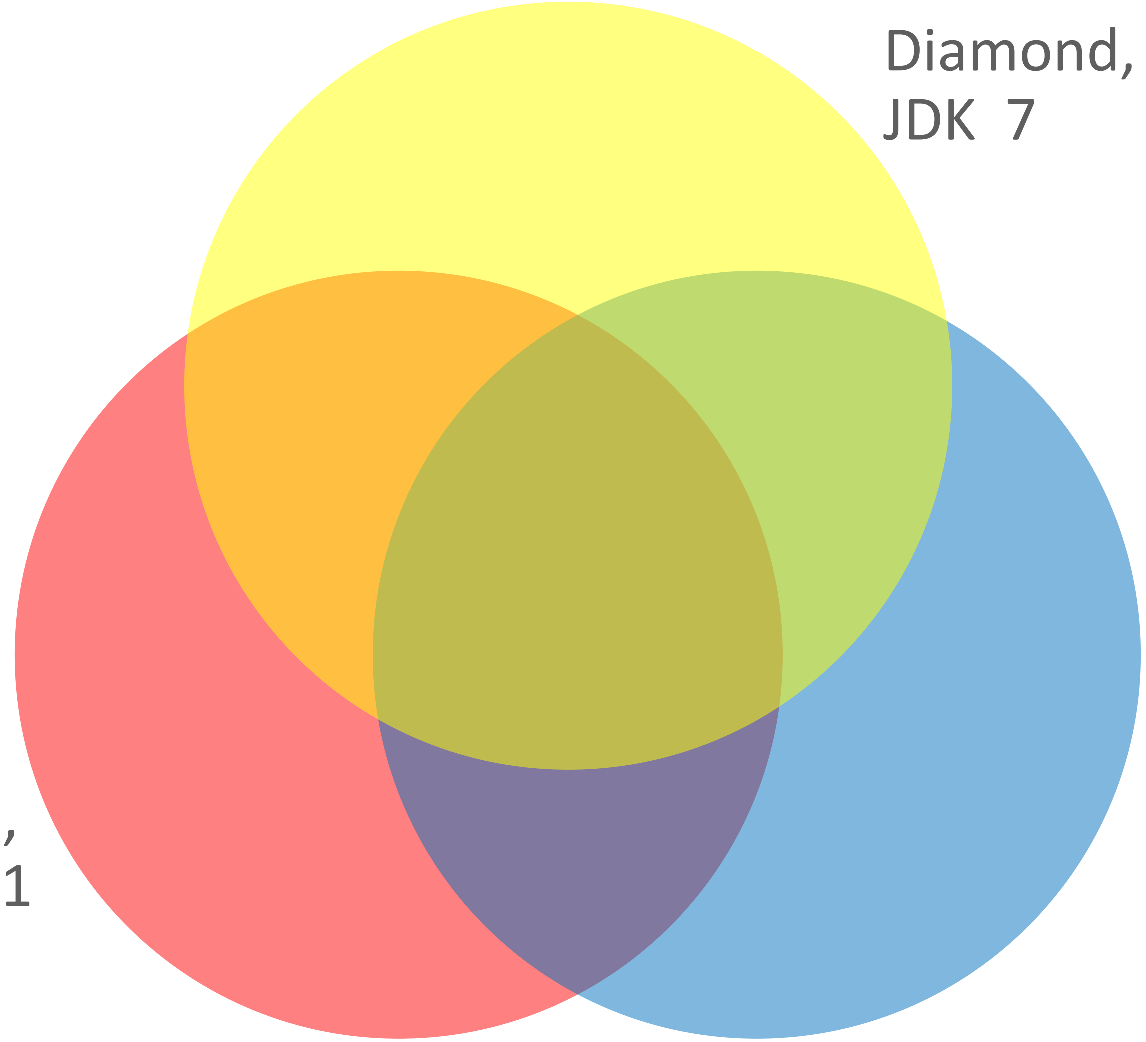


Inner classes,
JDK 1.1

Language feature interactions over time



Language feature interactions over time

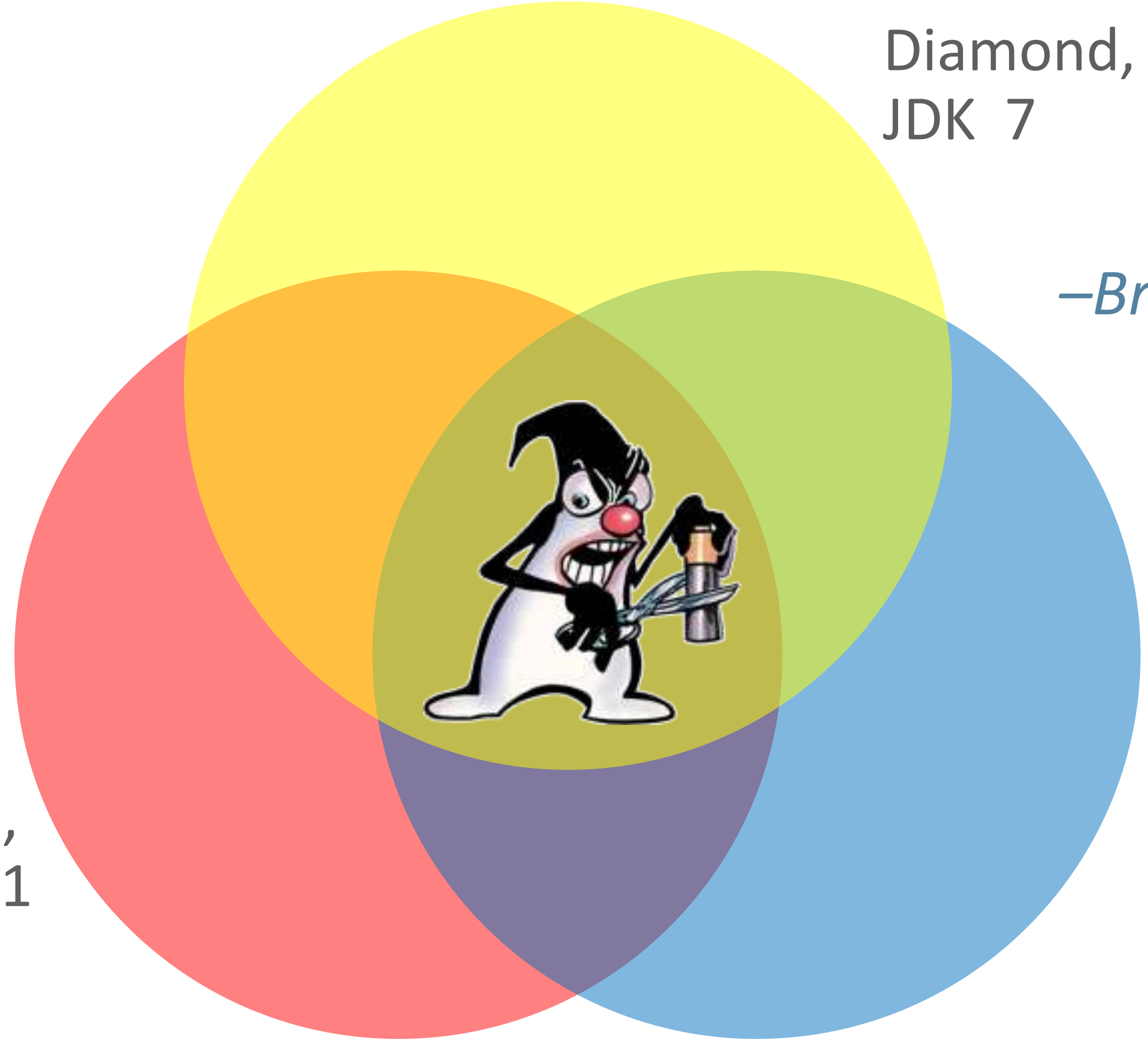


Diamond,
JDK 7

Generics,
JDK 5.0

Inner classes,
JDK 1.1

Language feature interactions over time



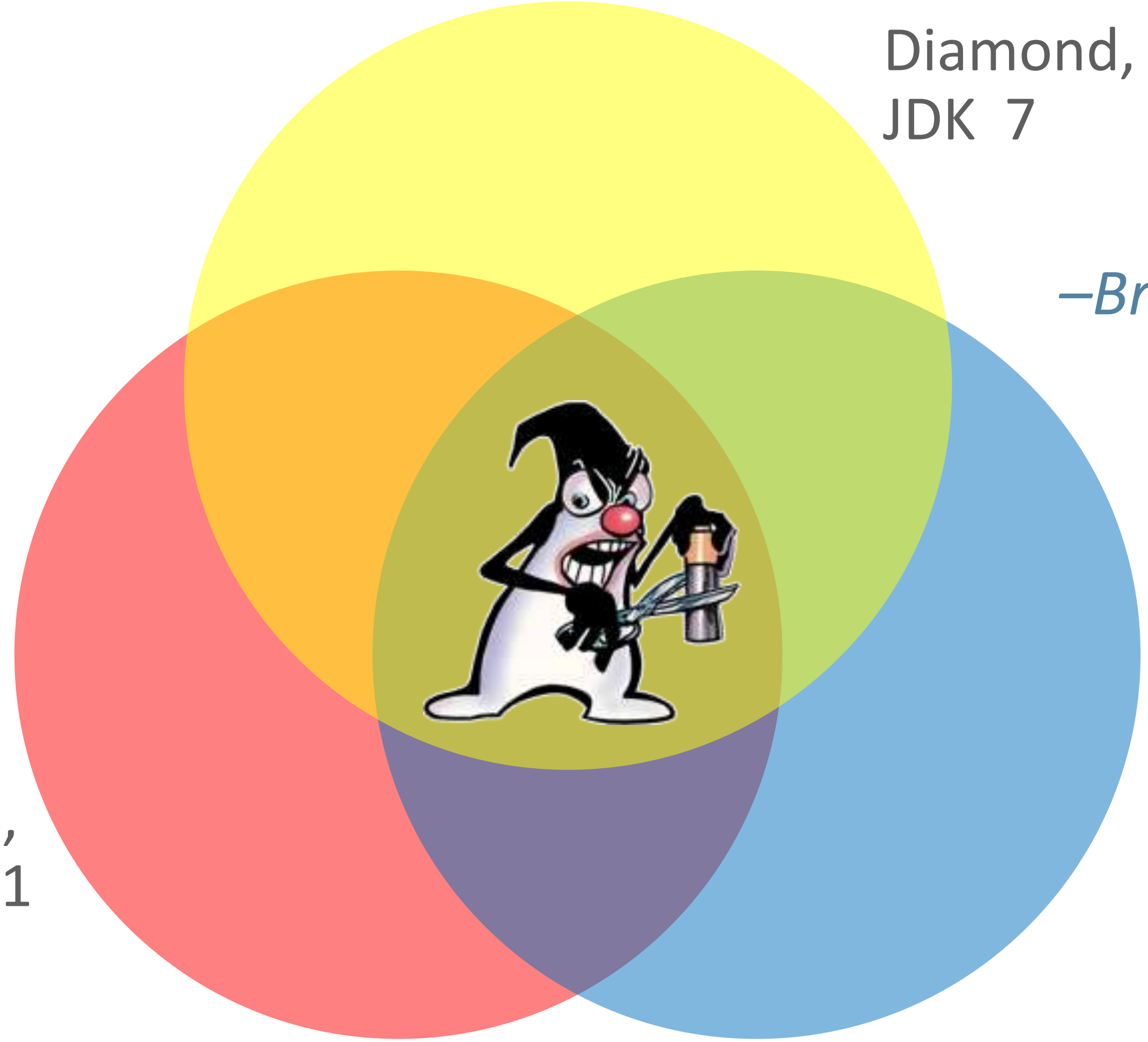
Diamond,
JDK 7

Generics,
JDK 5.0

Inner classes,
JDK 1.1

“Today’s problems come from yesterday’s solutions.”
–Brian Goetz via Peter Senge via …

Language feature interactions over time



> *"Today's problems come from yesterday's solutions."*
> *–Brian Goetz via Peter Senge via …*

JDK 7 Result:
Cannot use diamond with anonymous classes in any circumstances.

A forward-looking statement from the JSR 334 Expert Group

JSR 334 Proposed Final Draft, June 2011

"Internally, a Java compiler operates over a richer set of types than those that can be written down explicitly in a Java program. The compiler-internal types which cannot be written in a Java program are called *non-denotable* types. Non-denotable types can occur as the result of the inference used by diamond. Therefore, using diamond with anonymous inner classes is *not* supported since doing so in general would require extensions to the class file signature attribute to represent non-denotable types, a de facto JVM change. *It is feasible that future platform versions could allow use of diamond when creating an anonymous inner class as long as the inferred type was denotable*."

Java SE 9 allows use of diamond when creating an anonymous inner class as long as the inferred type is denotable. (JDK-8073593)

Partial specification updates:
JLS §15.9.3 Choosing the Constructor and its Arguments

It is a compile-time error if the superclass or superinterface type of the anonymous class, T, or any subexpression of T, has one of the following forms:

- A type variable (4.4) that was not declared as a type parameter (such as a type variable produced by capture conversion (5.1.10))
- An intersection type (4.9)
- A class or interface type, where the class or interface declaration is not accessible from the class or interface in which the expression appears.

The term “subexpression” includes type arguments of parameterized types (4.5), bounds of wildcards (4.5.1), and element types of array types (10.1). It excludes bounds of type variables.

Partial specification updates:
JLS §15.9.5 Anonymous Class Declarations

The superclass or superinterface type is given by the class instance creation expression (15.9.1); if this type is generic and its type arguments are elided using '`<>`', the type arguments are inferred while choosing a constructor (15.9.3).

If the class instance creation expression elides the supertype's type arguments using '`<>`', then for all non-private methods declared in the class body, it is as if the method declaration is annotated with `@Override` (9.6.4.4).

[Note:] When the diamond form is used, the inferred type arguments may not be as anticipated by the programmer. Consequently, the supertype of the anonymous class may not be as anticipated, and methods declared in the anonymous class may not override supertype methods as intended. Treating such methods as if annotated with @Override (if they are not actually annotated with `@Override`) helps avoid silently incorrect programs.

Effectiveness

- In JDK 7, diamond could eliminate explicit type arguments at ≈90% of constructor call sites
- Diamond with anonymous classes should allow removal of a large fraction of the remaining 10%.
  - JDK code base updated to use this feature, hundreds of call sites (JDK-8078467)
  - Main beneficiaries were uses of `java.security.PrivilegedAction`

An underscore is no longer an identifier name

JDK-8061549: Disallow ‘_‘ as a one-character identifier

Starting in Java SE 8

Discussed in http://mail.openjdk.java.net/pipermail/lambda-dev/2013-July/010661.html

- Part of Project Lambda / JSR 335
- Cannot use '_' as an identifier for a lambda parameter; avoid confusion with "wunderbar" from other languages
- If used elsewhere legal, but generates a warning; from `javac`:
  warning: '_' used as an identifier (use of '_' as an identifier might not be supported in releases after Java SE 8)
  - Only a single underscore as an identifier generates a warning, use as a separator inside a longer identifier is fine

Future possibilities

Claim back the name real estate

- Partial diamond:
  `new Foo<String, _>();`
- Partial witnesses:
  `foo.<String, _>bar()`
- “Don’t care” parameter names, suppress “var not used” warnings
- …

`private` interface methods

Background: default methods

- Starting in Java SE 8, as part of Project Lambda interfaces can have *default methods*.
- Default methods are non-`abstract` and thus have a method body.
- At the JVM level, interfaces could have `private` methods;
  helpful to implement lambdas, etc.
- However, `private` interface methods were *not* valid in the source language.

Now in Java SE 9...

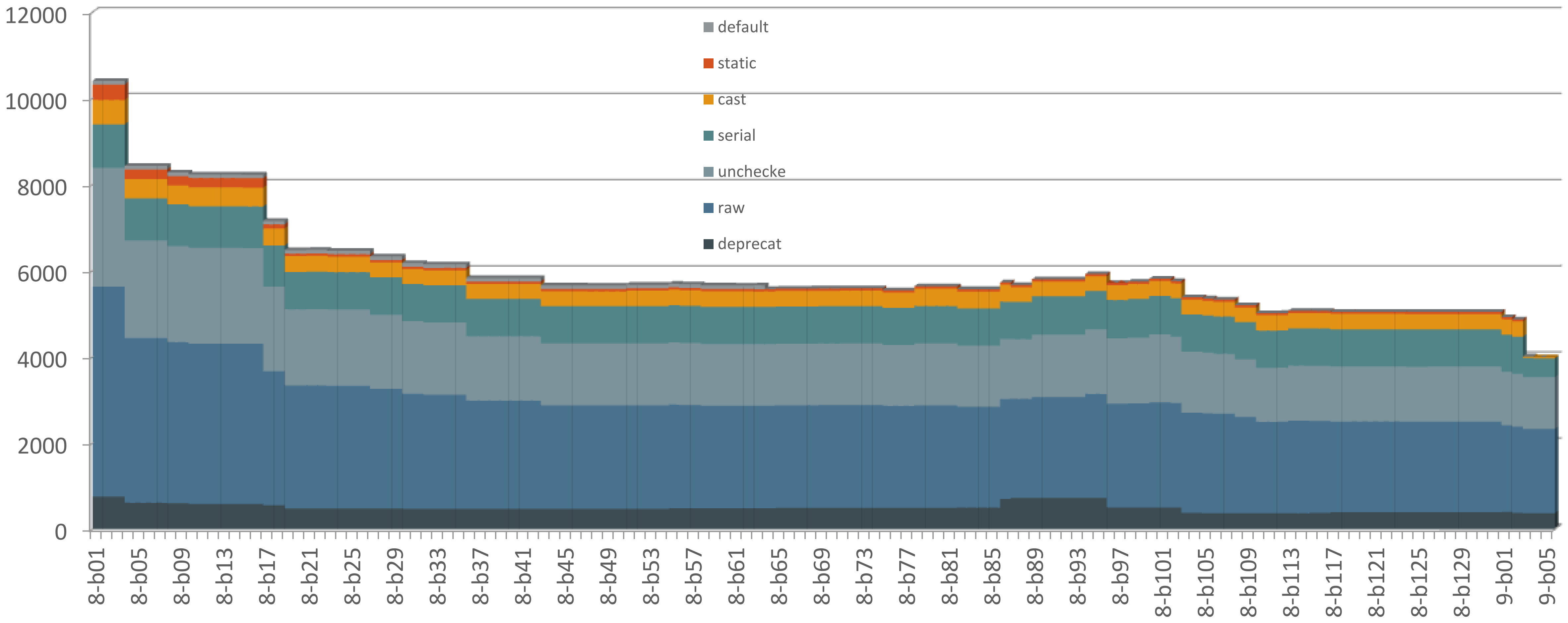
JDK-8071453: Allow interface methods to be private

- As planned, now in Java SE 9 interfaces can have `private` methods in the source level too:
  - `static` methods
  - instance methods
- Sample usage: helper methods to implement default methods

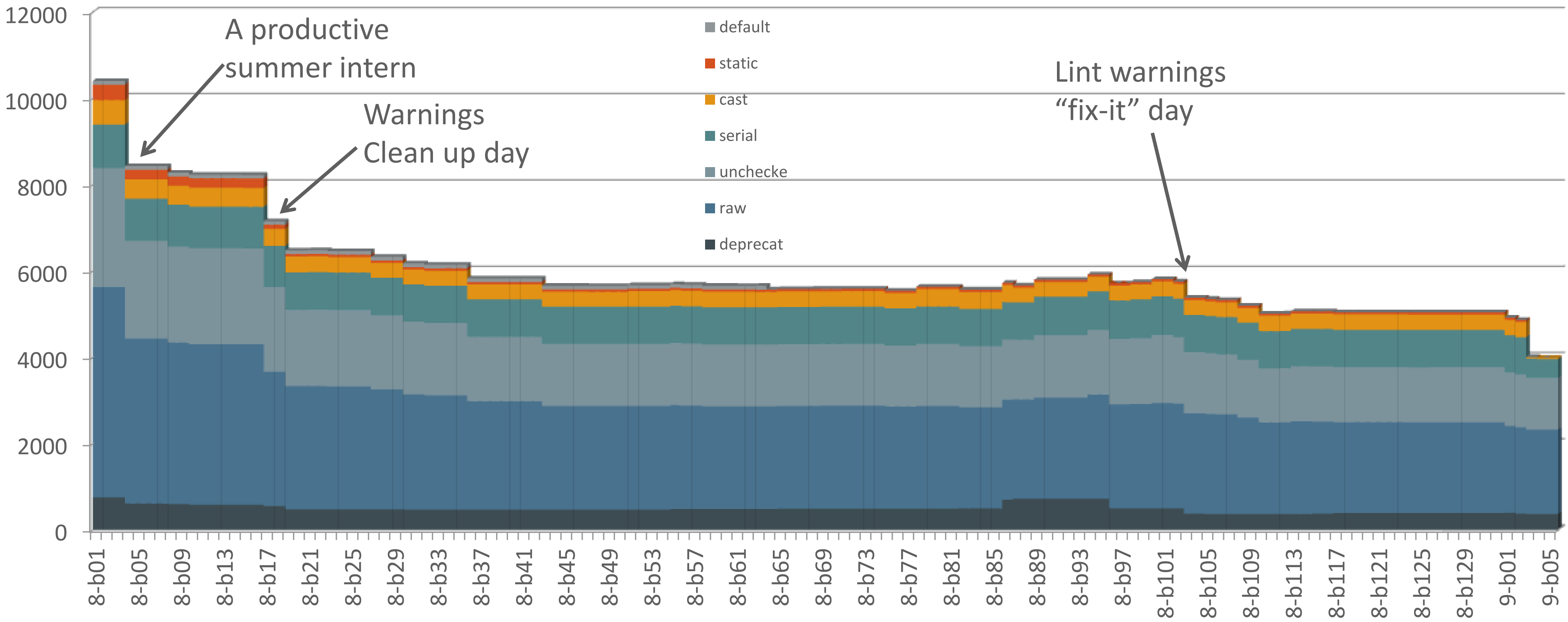
Deprecation and imports

JEP 211: Elide Deprecation Warnings on Import Statements

Background: warnings, warnings, everywhere



Background: warnings, warnings, everywhere

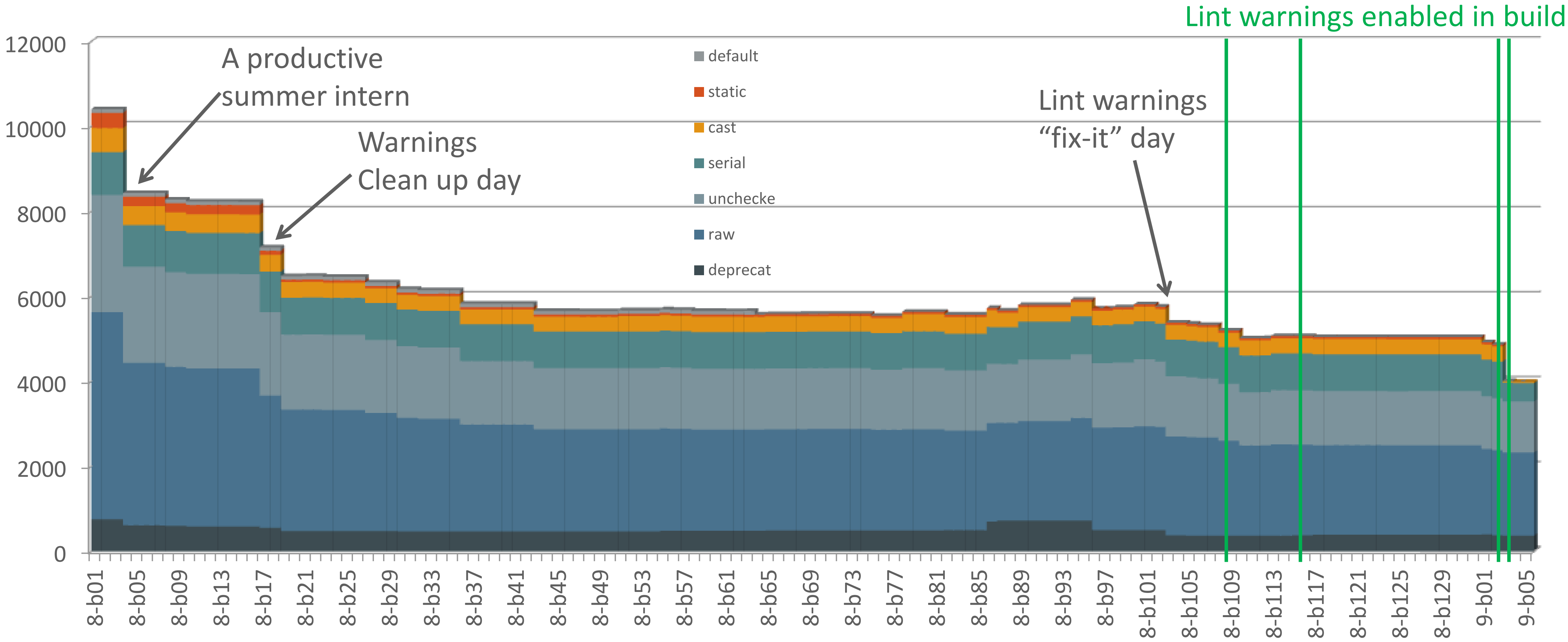


A productive summer intern

Warnings Clean up day

Lint warnings “fix-it” day

Background: warnings, warnings, everywhere



A productive summer intern

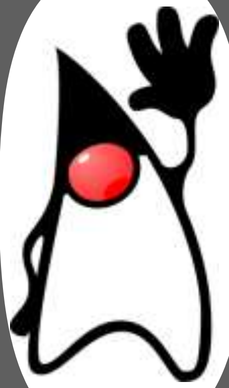
Warnings Clean up day

Lint warnings “fix-it” day

Lint warnings enabled in build

Number of `javac` lint warnings in JDK 9 `jdk` repo today

0

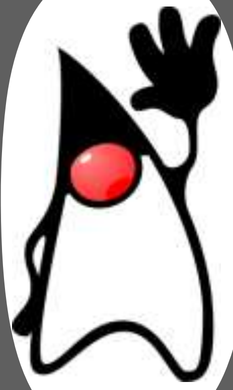


Number of `javac` lint warnings in JDK 9 `jdk` repo today

Covers all of the following:

- Open code
- Closed code
- Platform-specific code
- Generated code
- `-Xlint:all –Werror` in the build

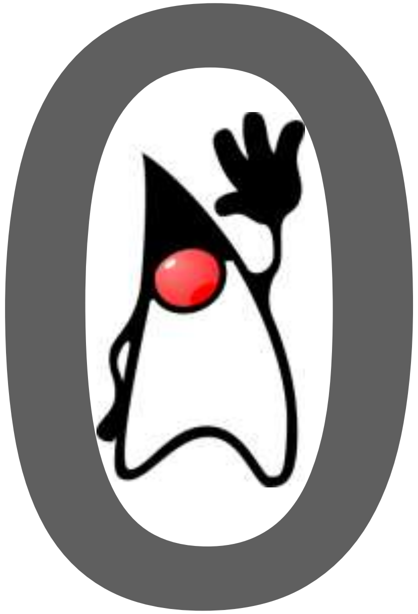
0



Number of `javac` lint warnings in JDK 9 `jdk` repo today

Covers all of the following:

- Open code
- Closed code
- Platform-specific code
- Generated code
- `-Xlint:all –Werror` in the build



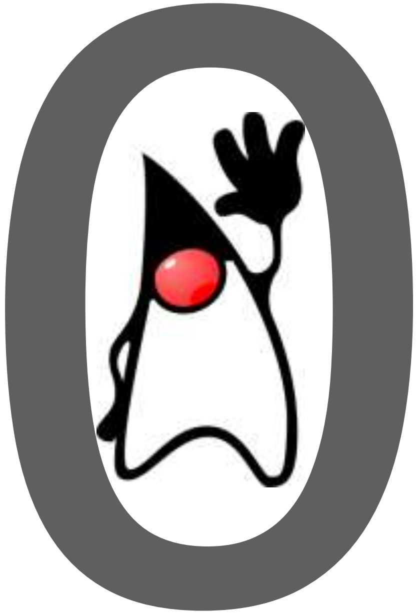
Warnings-free modules include:

- `java.base`: packages `java.lang`, `java.util`, `java.math`, ….
- `java.desktop`: packages `java.awt`, `javax.swing`, …
- `java.compiler`: packages `javax.lang.model`, …

Number of `javac` lint warnings in JDK 9 `jdk` repo today

Covers all of the following:

- Open code
- Closed code
- Platform-specific code
- Generated code
- `-Xlint:all –Werror` in the build



Warnings-free modules include:[†]

- `java.base`: packages `java.lang`, `java.util`, `java.math`, ….
- `java.desktop`: packages `java.awt`, `javax.swing`, …
- `java.compiler`: packages `javax.lang.model`, …

[†] corba, jaxp, jax-ws, not included

Technical dividends of going warnings free

`javac` as the first-line static analysis tool

- Did you declare a `serialVersionUID` in your exception type?
  (All exceptions are serializable; thanks RMI!)
- Did you override `equals` and `hashCode` correctly?
  (Nuanced and accurate check.)
- Is that fallthrough in a switch intentional?
  (Possible security issue; c.f. "`goto fail; goto fail;`")
- Is it really acceptable to have a new use of that deprecated item?

Technical dividends of going warnings free

javac as the first-line static analysis tool

- Did you declare a `serialVersionUID` in your exception type?
  (All exceptions are serializable; thanks RMI!)
- Did you override `equals` and `hashCode` correctly?
  (Nuanced and accurate check.)
- Is that fallthrough in a switch intentional?
  (Possible security issue; c.f. “`goto fail; goto fail;`”)
- Is it really acceptable to have a new use of that deprecated item?
- “Advice on removing javac lint warnings”
  https://blogs.oracle.com/darcy/entry/warnings_removal_advice

Joseph D. Darcy's Oracle Weblog

Advice on removing javac lint warnings

Deprecation warnings

- Several hundred deprecation warnings in the JDK
- Generated by use of deprecated types, methods, constructors, fields
  - Mandated by JLS §9.6.4.6 `@Deprecated`
- Can resolve warnings by:

Deprecation warnings

- Several hundred deprecation warnings in the JDK
- Generated by use of deprecated types, methods, constructors, fields
  - Mandated by JLS §9.6.4.6 `@Deprecated`
- Can resolve warnings by:
  - Propagating `@Deprecated` to the use-sites too

Deprecation warnings

- Several hundred deprecation warnings in the JDK
- Generated by use of deprecated types, methods, constructors, fields
  - Mandated by JLS §9.6.4.6 `@Deprecated`
- Can resolve warnings by:
  - Propagating `@Deprecated` to the use-sites too
  - Removing use of the deprecated element(s)

Deprecation warnings

- Several hundred deprecation warnings in the JDK
- Generated by use of deprecated types, methods, constructors, fields
  - Mandated by JLS §9.6.4.6 `@Deprecated`
- Can resolve warnings by:
  - Propagating `@Deprecated` to the use-sites too
  - Removing use of the deprecated element(s)
  - `@SuppressWarnings("deprecation")`

A wrinkle in Java SE 5.0 through 8

```
import p.DepLib;

public class Client2 {
  public static void
    main(String… args) {
      DepLib dl = new DepLib();
      dl.foo();
    }
}
```

```
// DepLib.java
package p;
@Deprecated
public class DepLib {
  public DepLib() {}
  public void foo() {return;}
}
```

A wrinkle in Java SE 5.0 through 8

```
import p.DepLib;

public class Client2 {
  public static void
    main(String… args) {
      DepLib dl = new DepLib();
      dl.foo();
    }
}
```

```
Note: Client2.java uses or
  overrides a deprecated API.
```

```
// DepLib.java
package p;
@Deprecated
public class DepLib {
  public DepLib() {}
  public void foo() {return;}
}
```

A wrinkle in Java SE 5.0 through 8

```
import p.DepLib;


public class Client2 {
  public static void
    main(String… args) {
      DepLib dl = new DepLib();
      dl.foo();
    }
}
```

```
// DepLib.java
package p;
@Deprecated
public class DepLib {
  public DepLib() {}
  public void foo() {return;}
}
```

A wrinkle in Java SE 5.0 through 8

```
import p.DepLib;
@SuppressWarnings("deprecation")
public class Client2 {
  public static void
    main(String… args) {
      DepLib dl = new DepLib();
      dl.foo();
    }
}
```

```
// DepLib.java
package p;
@Deprecated
public class DepLib {
  public DepLib() {}
  public void foo() {return;}
}
```

A wrinkle in Java SE 5.0 through 8

```
import p.DepLib;
@SuppressWarnings("deprecation")
public class Client2 {
  public static void
    main(String… args) {
      DepLib dl = new DepLib();
      dl.foo();
    }
}
```

```
Note: Client2.java uses or
  overrides a deprecated API.
```

```
// DepLib.java
package p;
@Deprecated
public class DepLib {
  public DepLib() {}
  public void foo() {return;}
}
```

Deprecation warning in detail

With javac -Xlint:deprecation …

```
Client2.java:1: warning: [deprecation] DepLib in p has
  been deprecated
  import p.DepLib;
          ^
  1 warning
```

- Warning on `import` mandated by the JLS
- Use of the type in this location *cannot* be annotated for suppression
- Warning is unhelpful; if all other uses of the deprecated type can be suppressed, shouldn’t have to resort to using its fully qualified name everywhere to be warning-free.

Specification update: JLS §9.6.3.6 `@Deprecated`

A Java compiler must produce a deprecation warning when a type, method, field, or constructor whose declaration is annotated with the annotation `@Deprecated` is used (i.e. overridden, invoked, or referenced by name), unless:

- The use is within an entity that is itself annotated with the annotation `@Deprecated`; or
- The use is within an entity that is annotated to suppress the warning with the annotation `@SuppressWarnings("deprecation");` or
- The use and declaration are both within the same outermost class.

Specification update: JLS §9.6.3.6 @Deprecated

A Java compiler must produce a deprecation warning when a type, method, field, or constructor whose declaration is annotated with the annotation `@Deprecated` is used (i.e. overridden, invoked, or referenced by name), unless:

- The use is within an entity that is itself annotated with the annotation `@Deprecated`; or
- The use is within an entity that is annotated to suppress the warning with the annotation `@SuppressWarnings("deprecation")`; or
- The use and declaration are both within the same outermost class; or
- The use is within an import statement that imports the type or member whose declaration is annotated with `@Deprecated.`

Result: now tractable to clear a code base of deprecation warnings.

Result: now tractable to clear a code base of deprecation warnings.

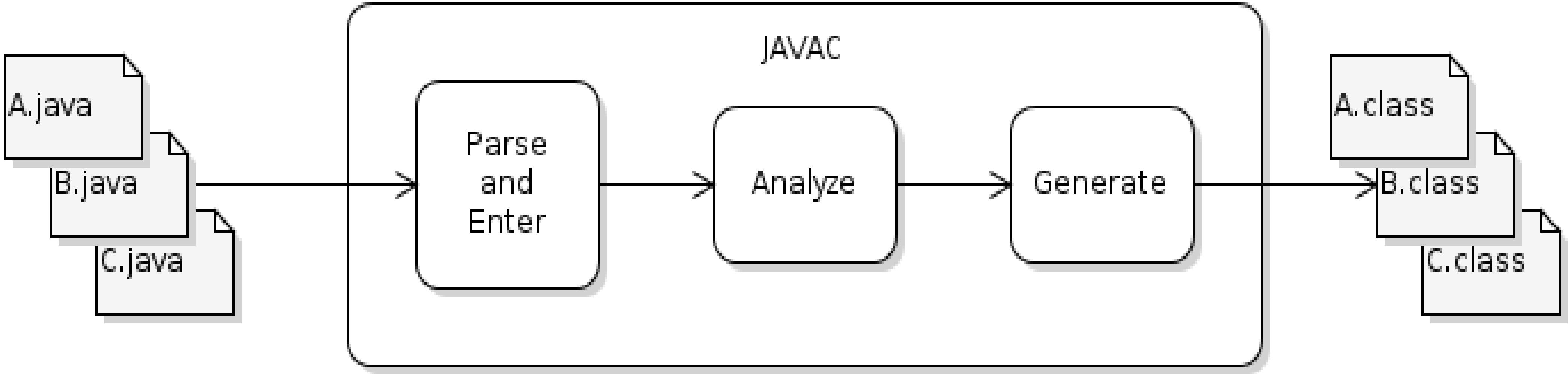
(If you're interested in deprecation, see also *Saving the Future from the Past: Innovations in Deprecation* [CON6856], Wednesday, Oct 28, 3:00 p.m by Dr. Deprecator, @DrDeprecator.)

Tiered Attribution

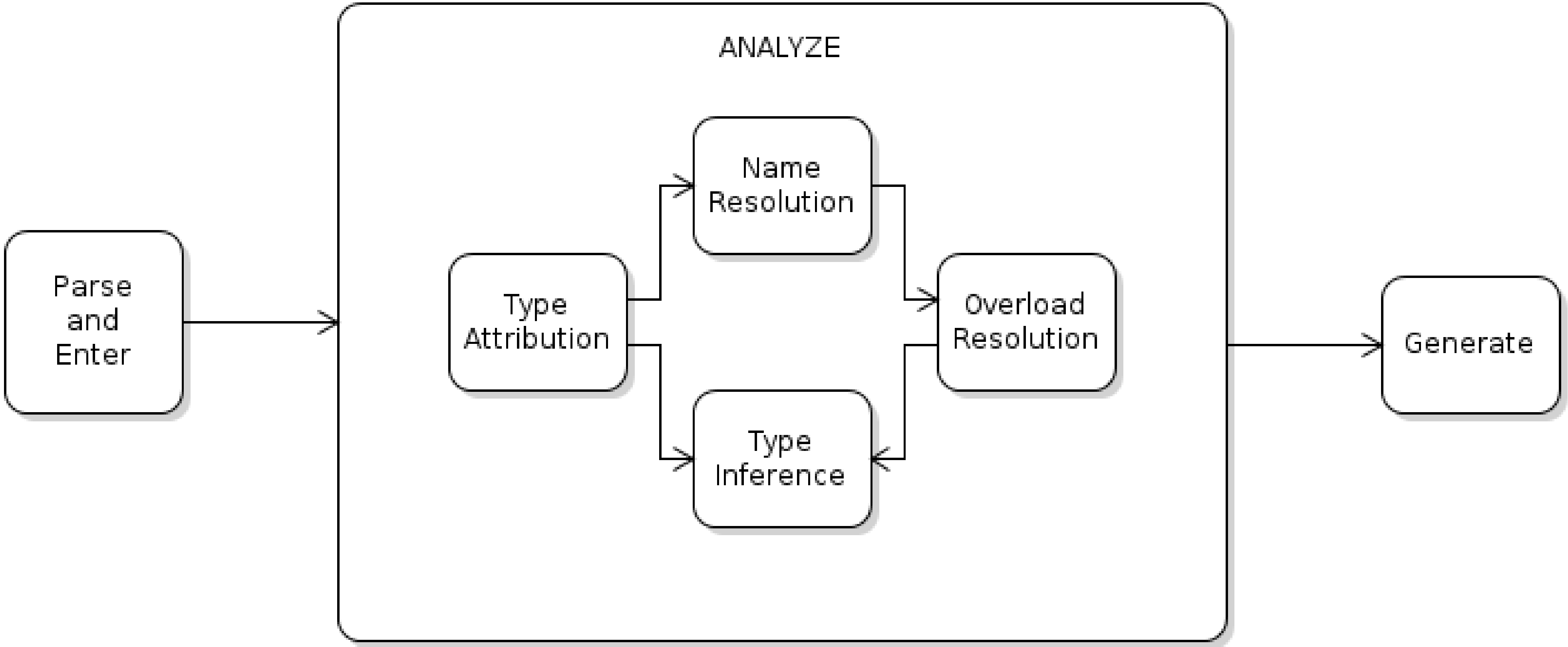
Or why and how we re-engineered the compiler from inside out

JEP 215: Tiered Attribution for javac

Type Attribution in the compiler pipeline



Type Attribution in the compiler pipeline



Type Attribution in action

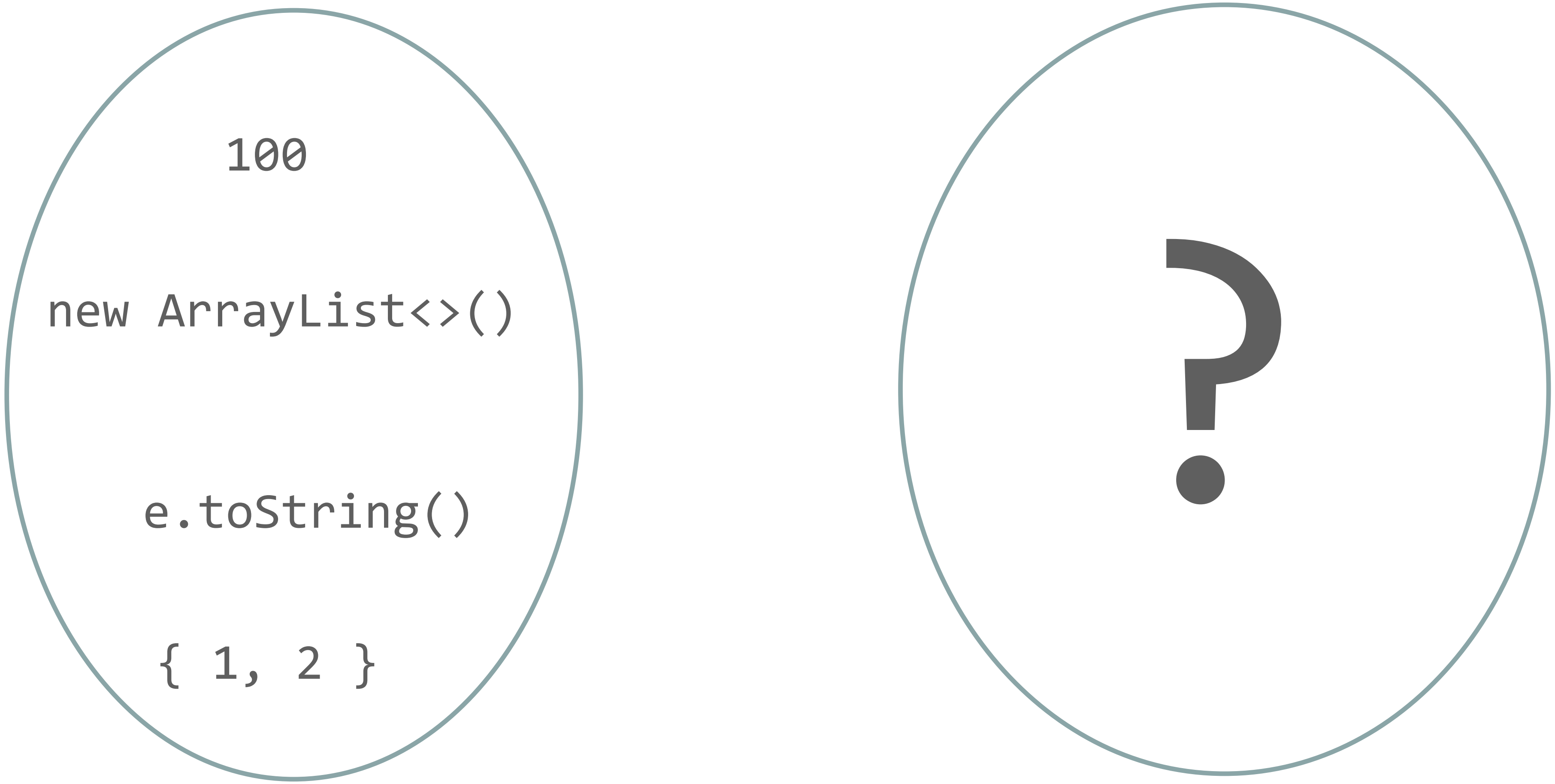
The main responsibility of type attribution is to assign a type to each element of a Java program.

```
void foo(int i, int j, boolean b, boolean v) {
    // Type attribution determines that the expression 'i + j'
    // has type int. The same type is assigned to variable k
    int k = i + j;
    // In this case type attribution determines that the expression 'b && v'
    // has type boolean
    System.out.println(b && v);
}
```

Java SE 8 introduced the concept of poly expressions

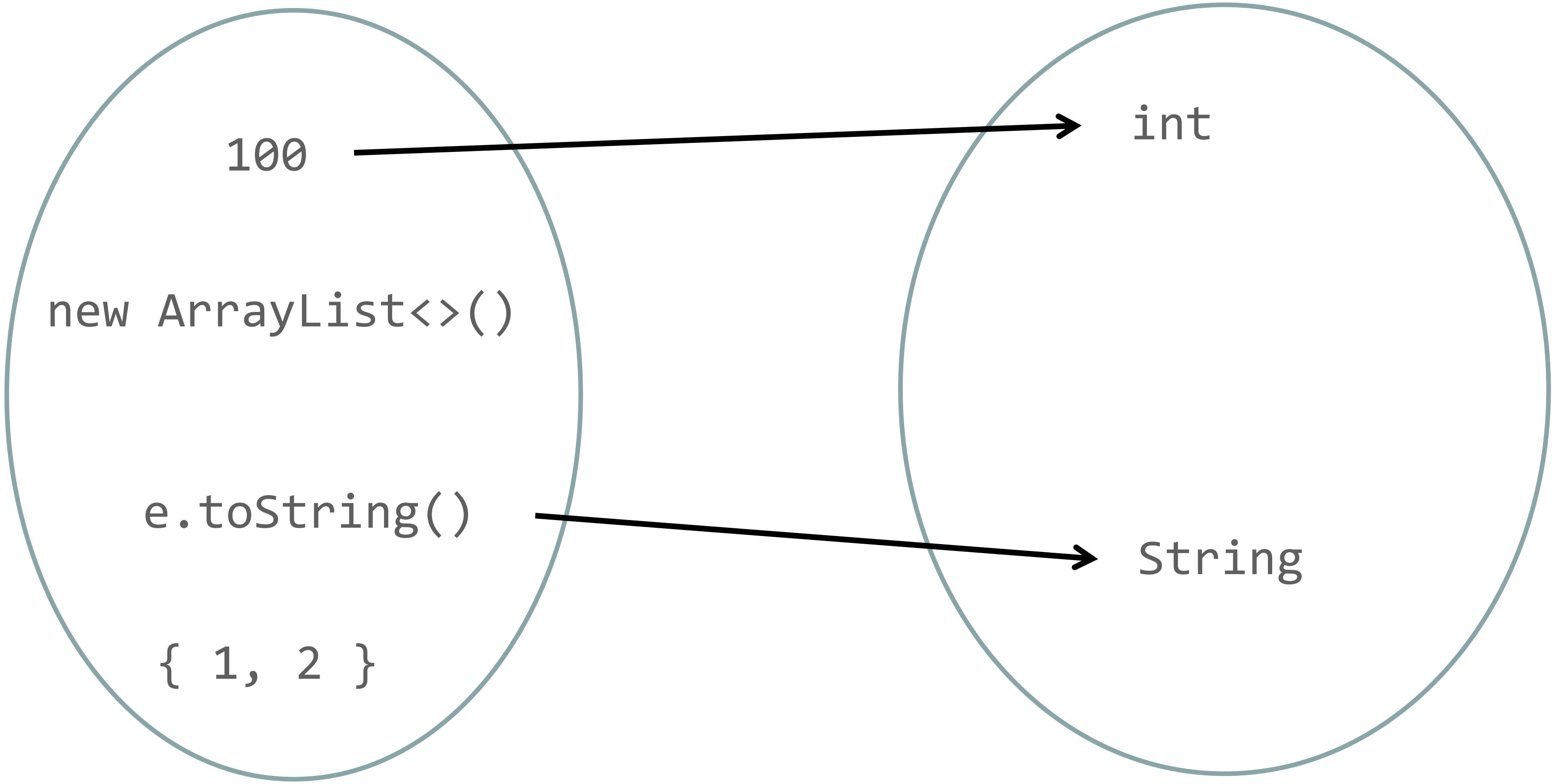
- Expressions for which its type can be influenced by the target type
- They can have different types in different contexts

Not all expressions are created equal!



```
100

new ArrayList<>()

e.toString()

{ 1, 2 }
```

Standalone expressions



100

new ArrayList<>()

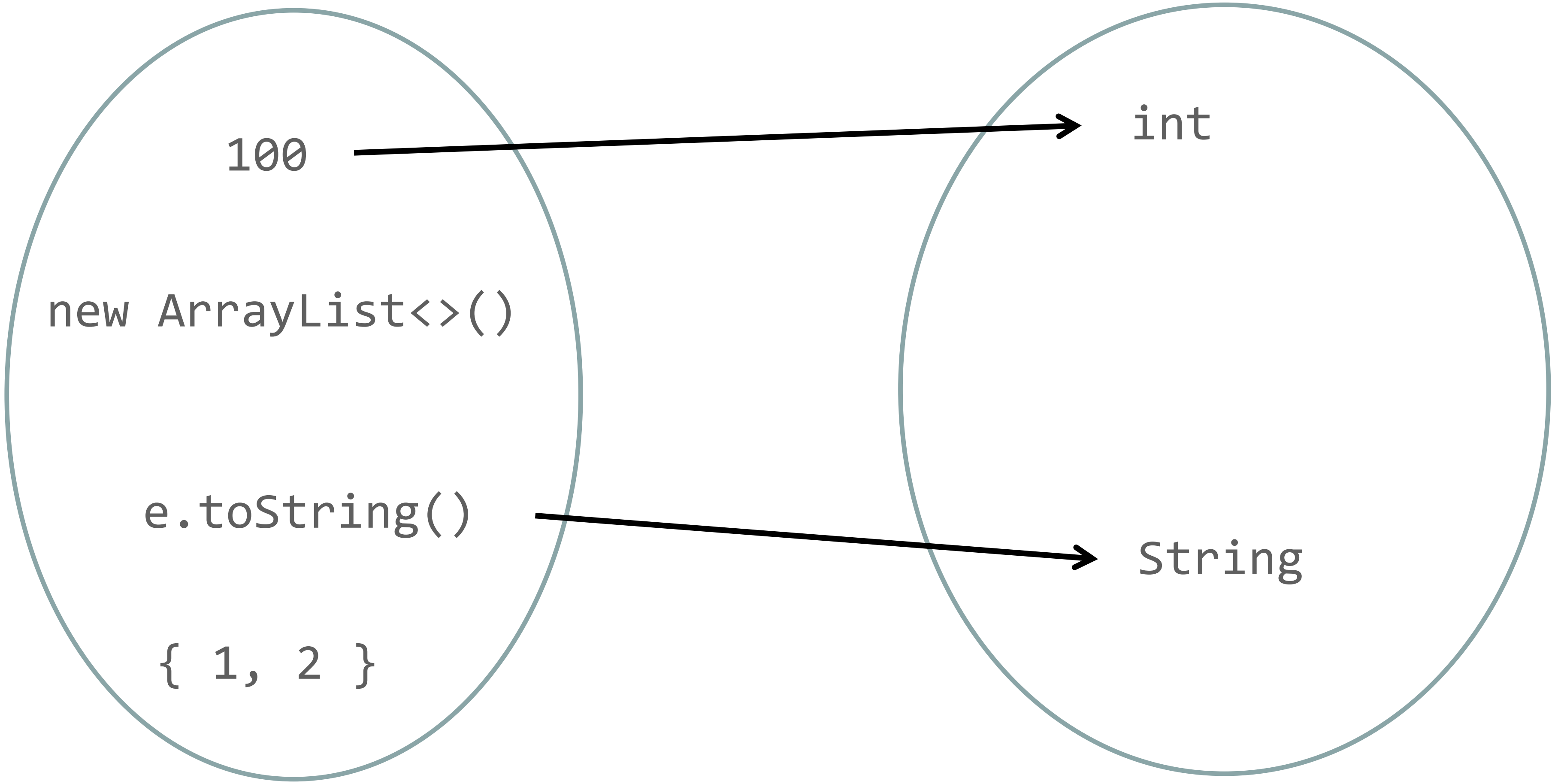
e.toString()

{ 1, 2 }

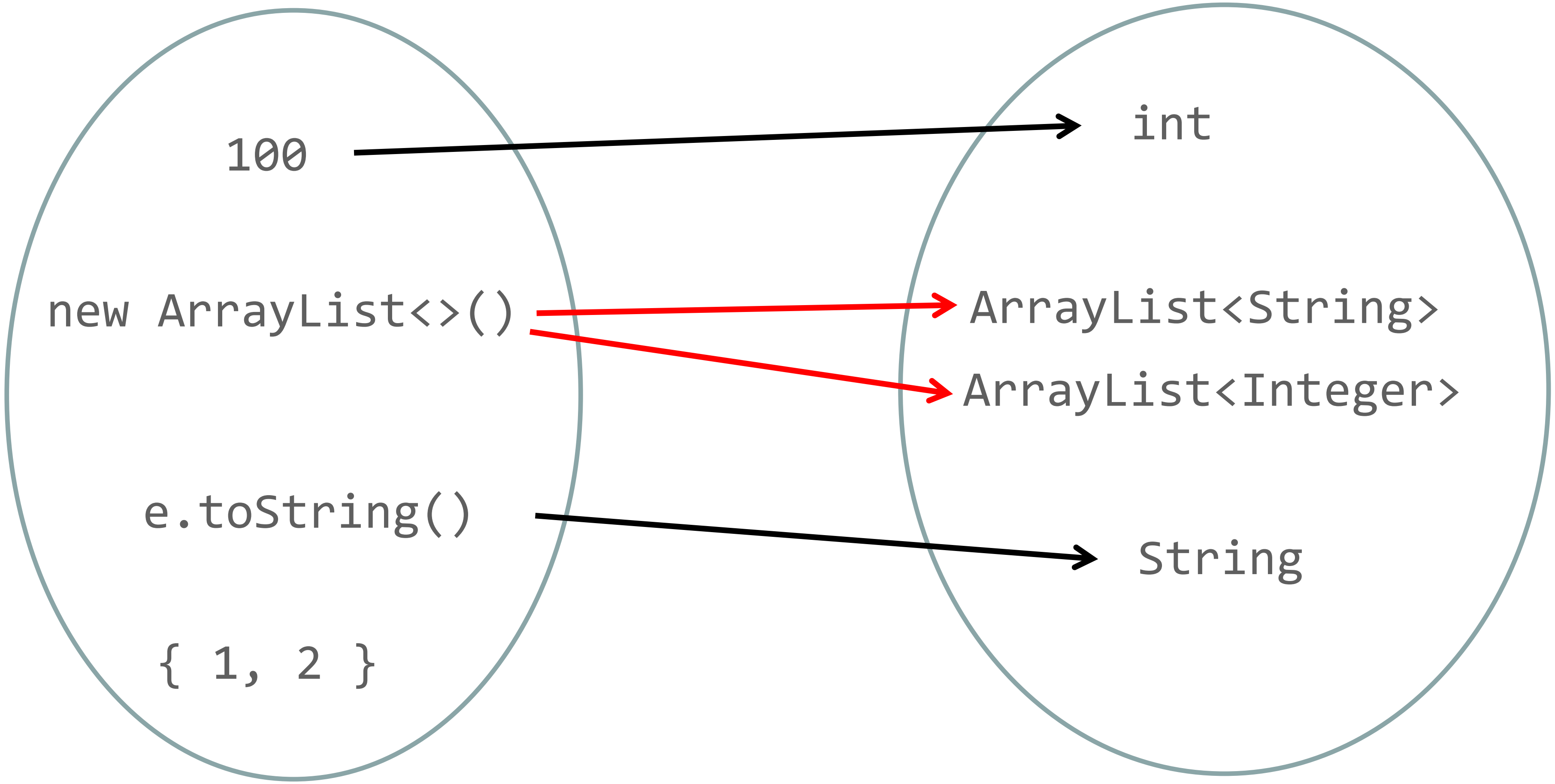
int

String

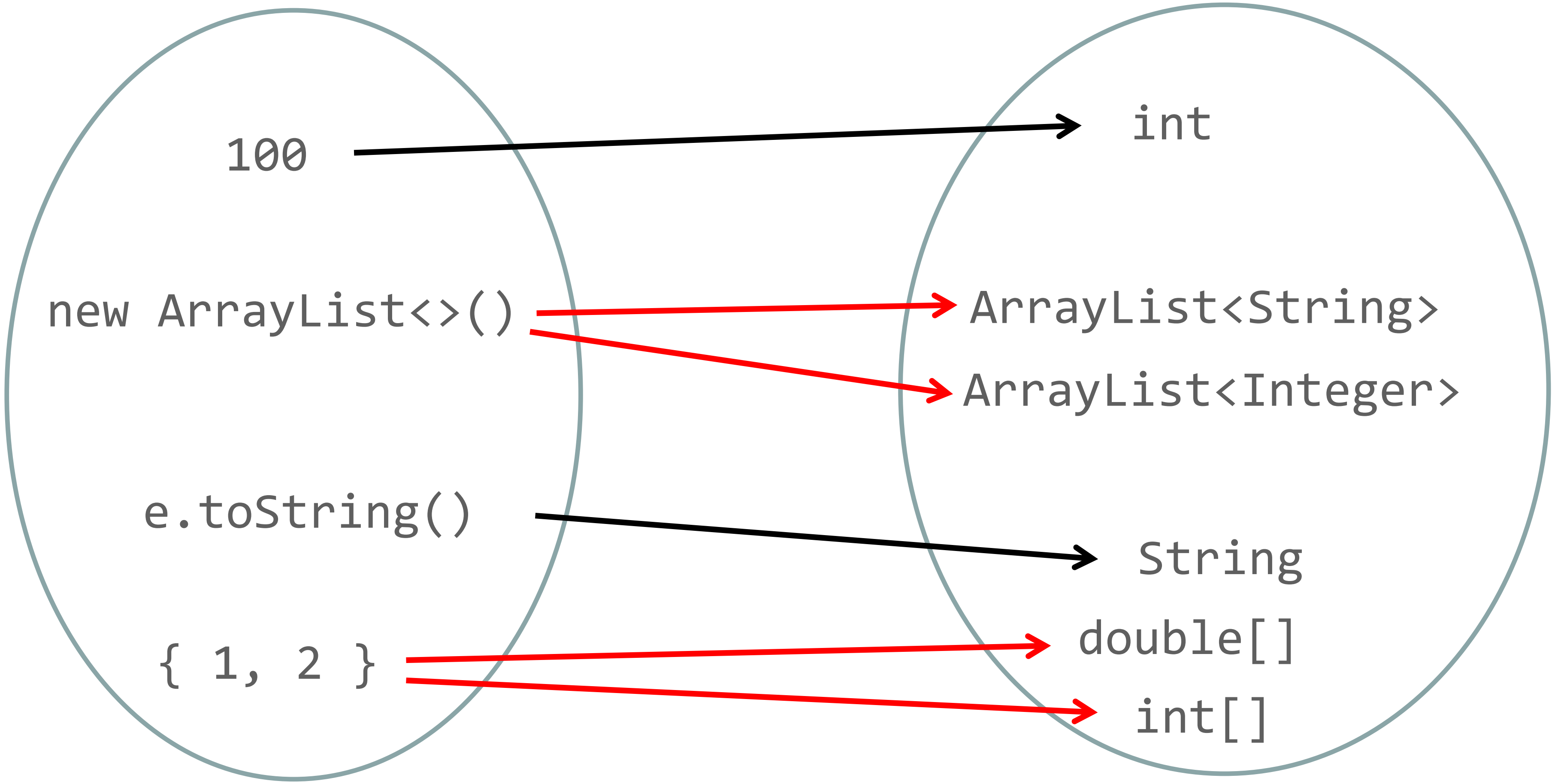
Poly expressions



Poly expressions



Poly expressions



Poly expressions

- Lambda expressions
- Method references
- Generic method calls
- Diamond instance creation expressions
- Conditional poly expressions
- Parenthesized poly expressions

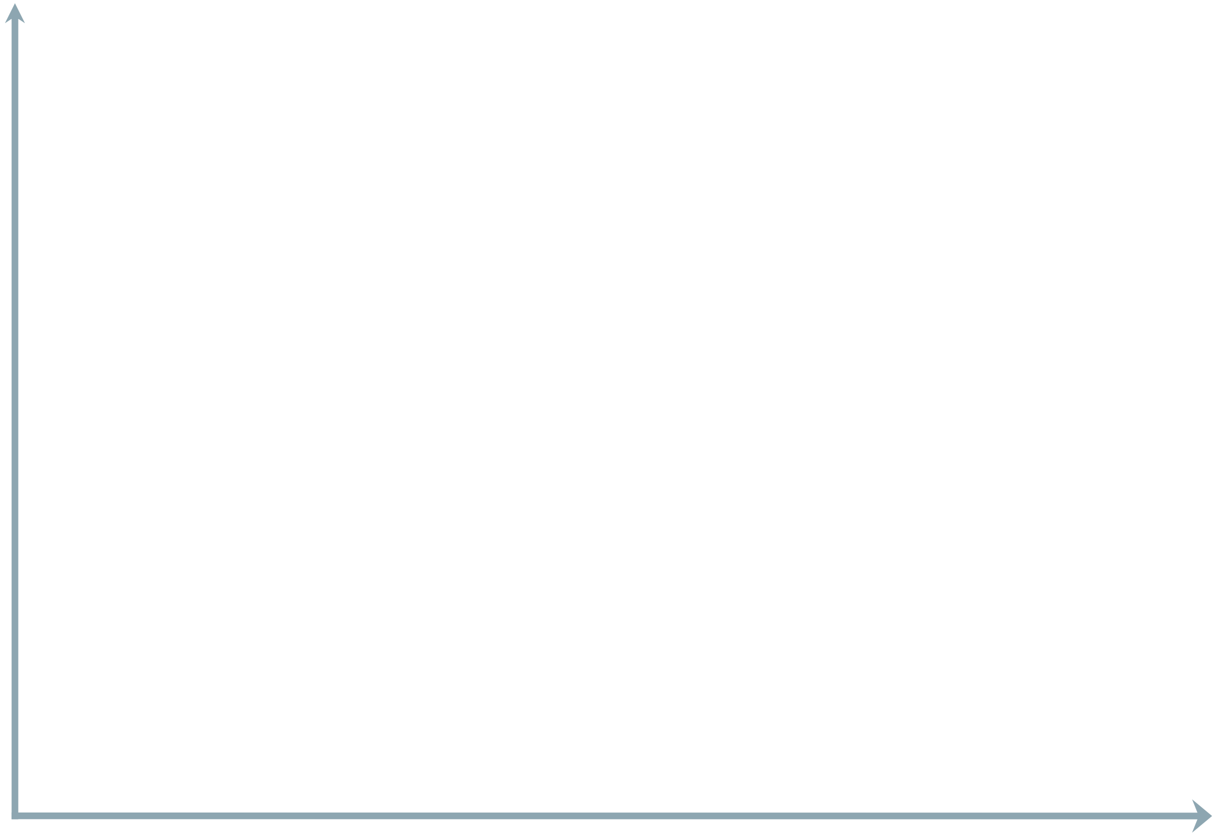
Poly expressions

- **Lambda expressions**
- Method references
- Generic method calls
- **Diamond instance creation expressions**
- Conditional poly expressions
- Parenthesized poly expressions

Type Inference

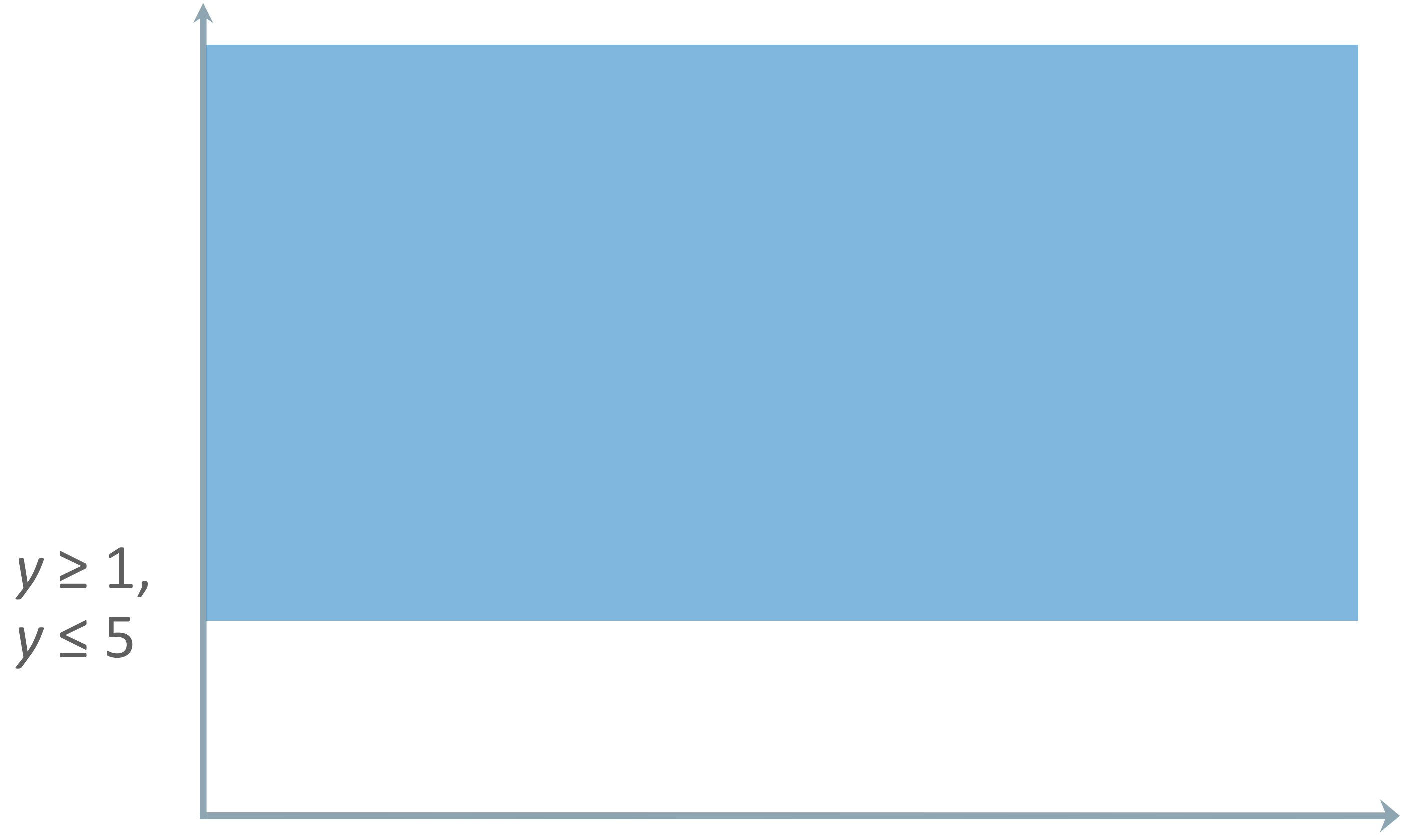
It’s a compiler's ability to look at each method invocation and corresponding declaration to determine the type argument(s) that make the invocation applicable

Type inference is like linear programming



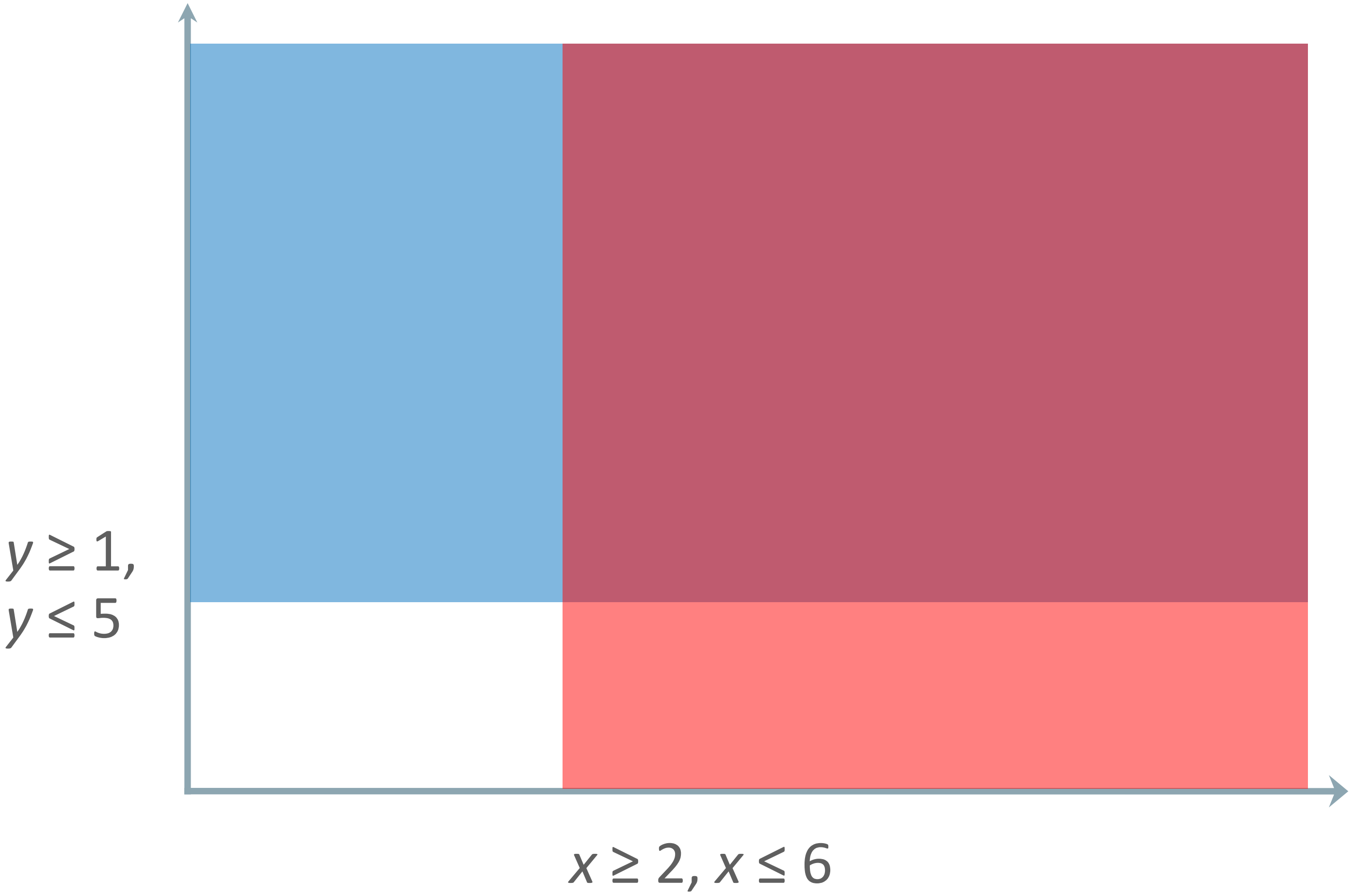
Linear programming:
subject to constraints,
maximize:
2x + y

Type inference is like linear programming



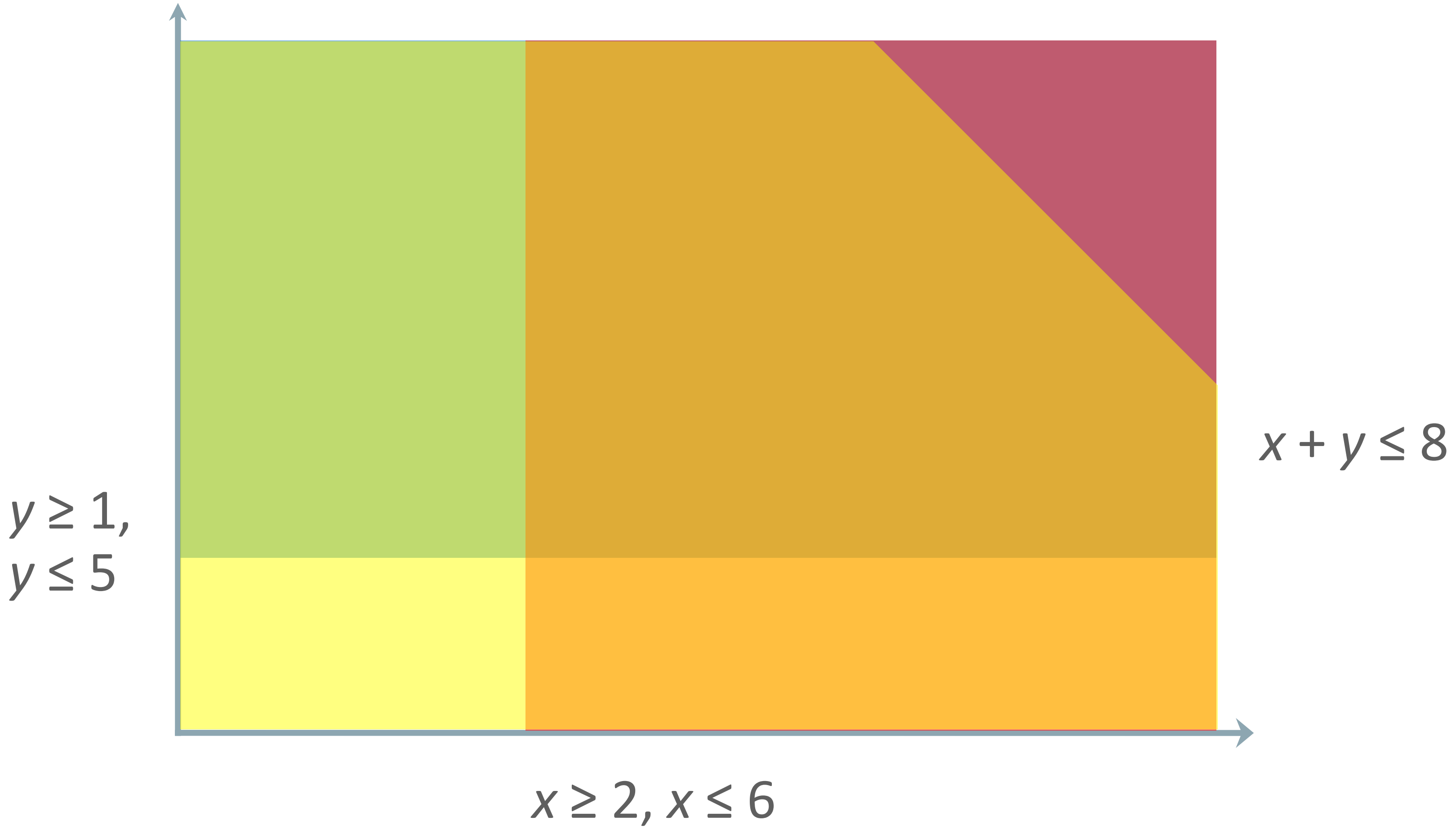
Linear programming: subject to constraints, maximize:
$2x + y$

Type inference is like linear programming



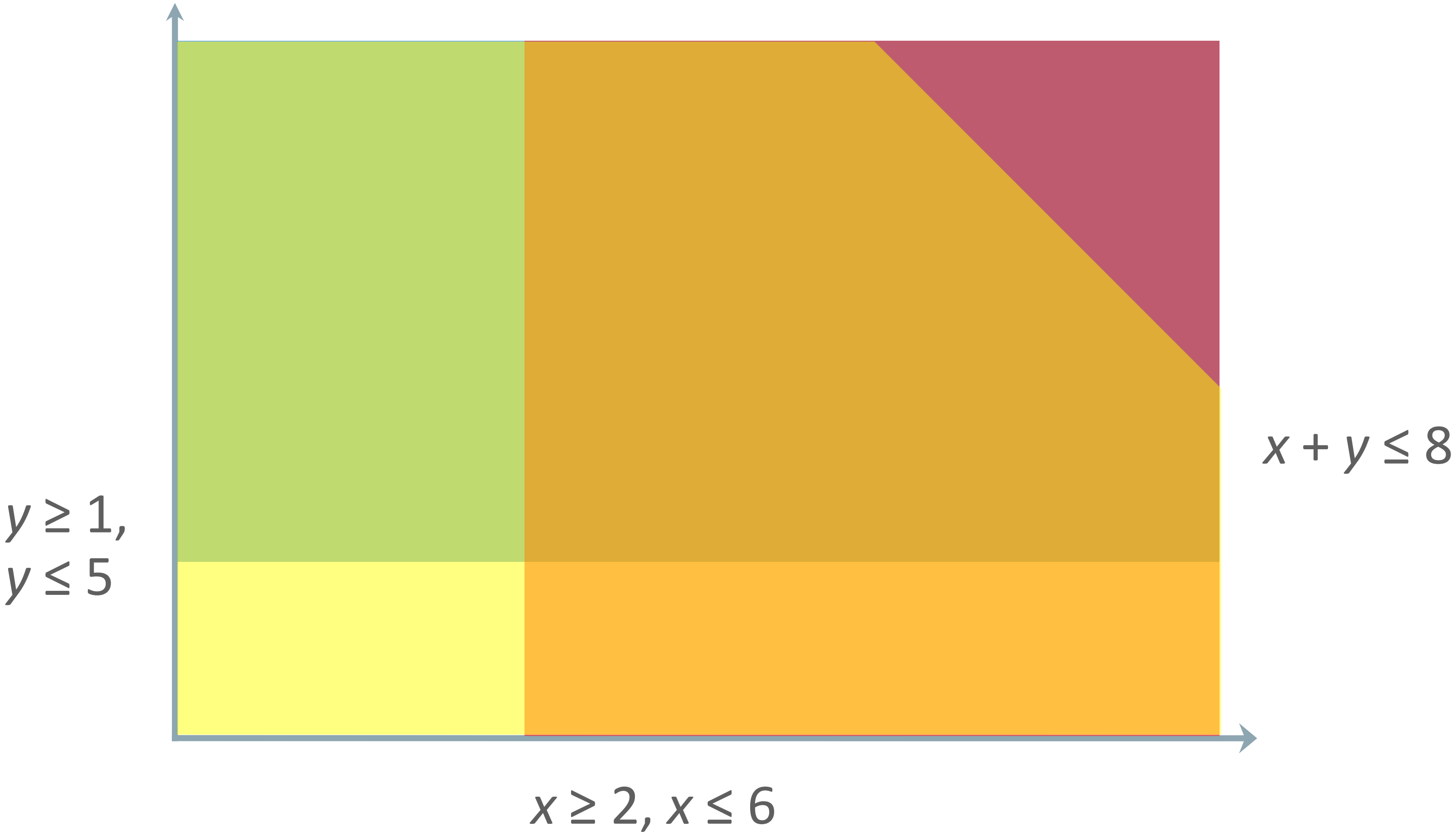
Linear programming: subject to constraints, maximize:
2x + y

Type inference is like linear programming



Linear programming:
subject to constraints,
maximize:
2x + y

Type inference is like linear programming



Linear programming: subject to constraints, maximize:
2x + y

Simplex method: One of the top-ten algorithms of the 20th century, often very fast solutions, but exponential in the worst case.

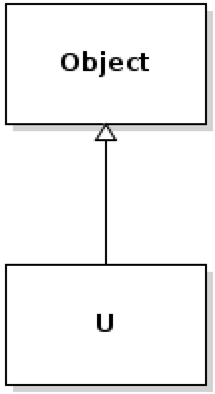
Attribution + Inference in action

```
class C<U> {
    U foo;
    C() {}
    C(C<U> o) { foo = o.foo; }
    C(U foo) { this.foo = foo;}
}
```

```
public static void main(String argv[]) {
    // There are no type arguments in the right-hand side.
    // How does javac figure out what the types are?
    C<String> c1 = new C<>();
}
```

Attribution + Inference in action

From U's declaration:

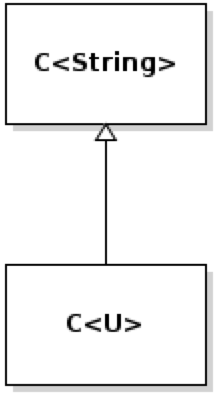


U <: Object

```
class C<U> {
    U foo;
    C() {}
    C(C<U> o) { foo = o.foo; }
    C(U foo) { this.foo = foo;}
}

public static void main(
    String args[]) {
    C<String> c1 = new C<>();
}
```

Attribution + Inference in action

From the target type:



C<String>

C<U>

U = String

```
class C<U> {
    U foo;
    C() {}
    C(C<U> o) { foo = o.foo; }
    C(U foo) { this.foo = foo;}
}

public static void main(
    String args[]) {
    C<String> c1 = new C<>();
}
```

Attribution + Inference in action

```
class C<U> {
    U foo;
    C() {}
    C(C<U> o) { foo = o.foo; }
    C(U foo) { this.foo = foo;}
}
```

```
public static void main(String argv[]) {
    // Summing up, for this invocation javac has discovered that:
    //   U <: Object and
    //   U = String,
    // With this information the inference engine can figure out
    // that the instantiation of U has only one solution:
    //    U := String
    C<String> c1 = new C<>();
}
```

But once we complicate it...

```
class C<U> {
    U foo;
    C() {}
    C(C<U> o) { foo = o.foo; }
    C(U foo) { this.foo = foo;}
}
```

```
// compilation time: 0m0.353s
public static void main(String argv[]) {
    C<String> c2 = new C<>(
                       new C<>());
}
```

But once we complicate it...

```
class C<U> {
    U foo;
    C() {}
    C(C<U> o) { foo = o.foo; }
    C(U foo) { this.foo = foo;}
}
```

```
// compilation time: 0m0.364s
public static void main(String argv[]) {
    C<String> c3 = new C<>(
                       new C<>(
                           new C<>()));
}
```

But once we complicate it...

```
class C<U> {
    U foo;
    C() {}
    C(C<U> o) { foo = o.foo; }
    C(U foo) { this.foo = foo;}
}
```

```
// compilation time: 0m0.436s
public static void main(String argv[]) {
    C<String> c4 = new C<>(
                       new C<>(
                           new C<>(
                               new C<>())));
}
```

But once we complicate it...

```
class C<U> {
    U foo;
    C() {}
    C(C<U> o) { foo = o.foo; }
    C(U foo) { this.foo = foo;}
}
```

```
// compilation time: 0m0.510s
public static void main(String argv[]) {
    C<String> c5 = new C<>(
                       new C<>(
                           new C<>(
                               new C<>(
                                   new C<>()))));
}
```

But once we complicate it...

```
class C<U> {
    U foo;
    C() {}
    C(C<U> o) { foo = o.foo; }
    C(U foo) { this.foo = foo;}
}
```

```
// compilation time: 0m0.645s
public static void main(String argv[]) {
    C<String> c6 = new C<>(
                       new C<>(
                           new C<>(
                               new C<>(
                                   new C<>(
                                       new C<>())))));
}
```

But once we complicate it...

```
class C<U> {
    U foo;
    C() {}
    C(C<U> o) { foo = o.foo; }
    C(U foo) { this.foo = foo;}
}
```

```
// compilation time: 0m0.899s
public static void main(String argv[]) {
    C<String> c7 = new C<>(
                       new C<>(
                           new C<>(
                               new C<>(
                                   new C<>(
                                       new C<>(
                                           new C<>())))))));
}
```

But once we complicate it...

```
class C<U> {
    U foo;
    C() {}
    C(C<U> o) { foo = o.foo; }
    C(U foo) { this.foo = foo;}
}
```

```
// compilation time: 0m1.448s
public static void main(String argv[]) {
    C<String> c8 = new C<>(
                       new C<>(
                           new C<>(
                               new C<>(
                                   new C<>(
                                       new C<>(
                                           new C<>(
                                               new C<>()))))))));
}
```

But once we complicate it…

```
class C<U> {
    U foo;
    C() {}
    C(C<U> o) { foo = o.foo; }
    C(U foo) { this.foo = foo;}
}
```

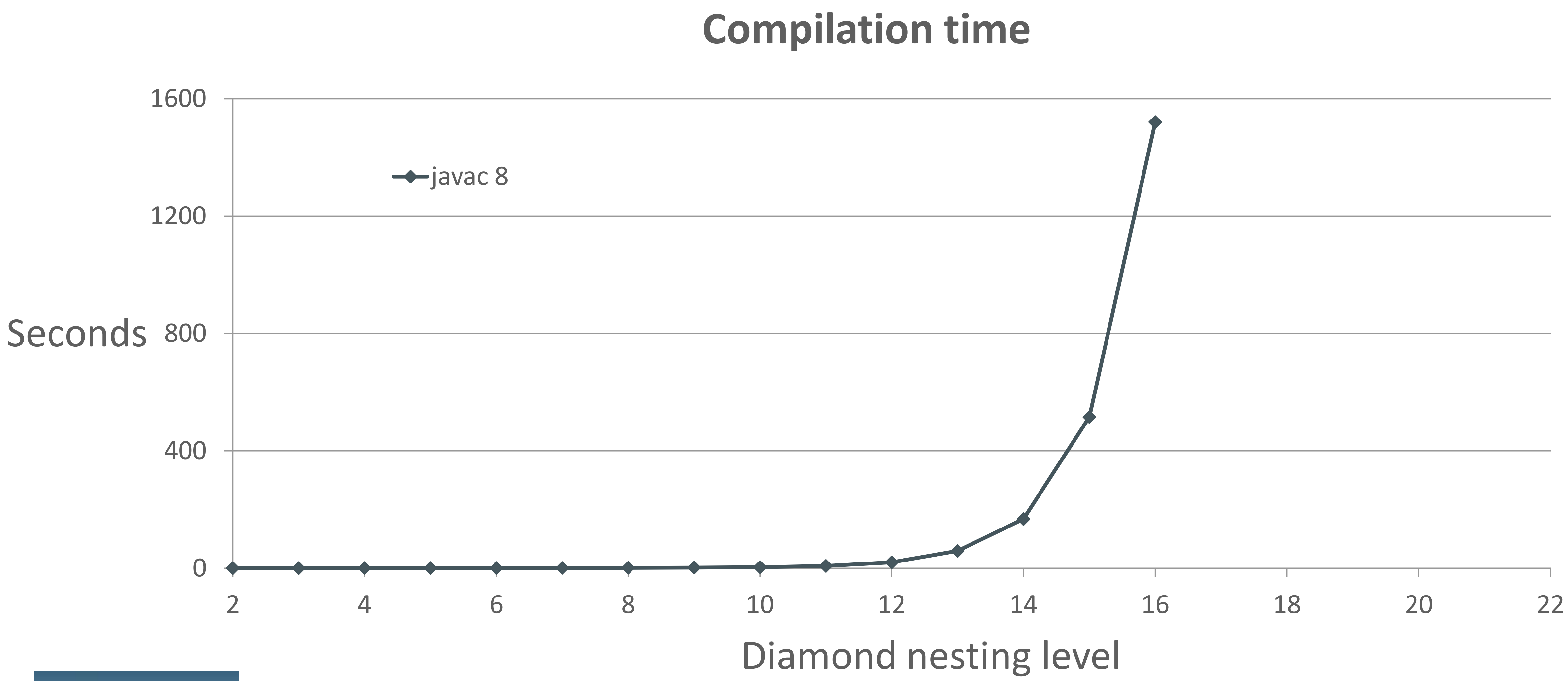
```
// compilation time: 25m20.590s
public static void main(String argv[]) {
    C<String> c16 = new C<>(
                       new C<>(
                           new C<>(
                               new C<>(
                                             ...
                                                     new C<>(
                                                         new C<>(
                                                             new C<>())))))))))))))));
}
```

Also for lambdas...

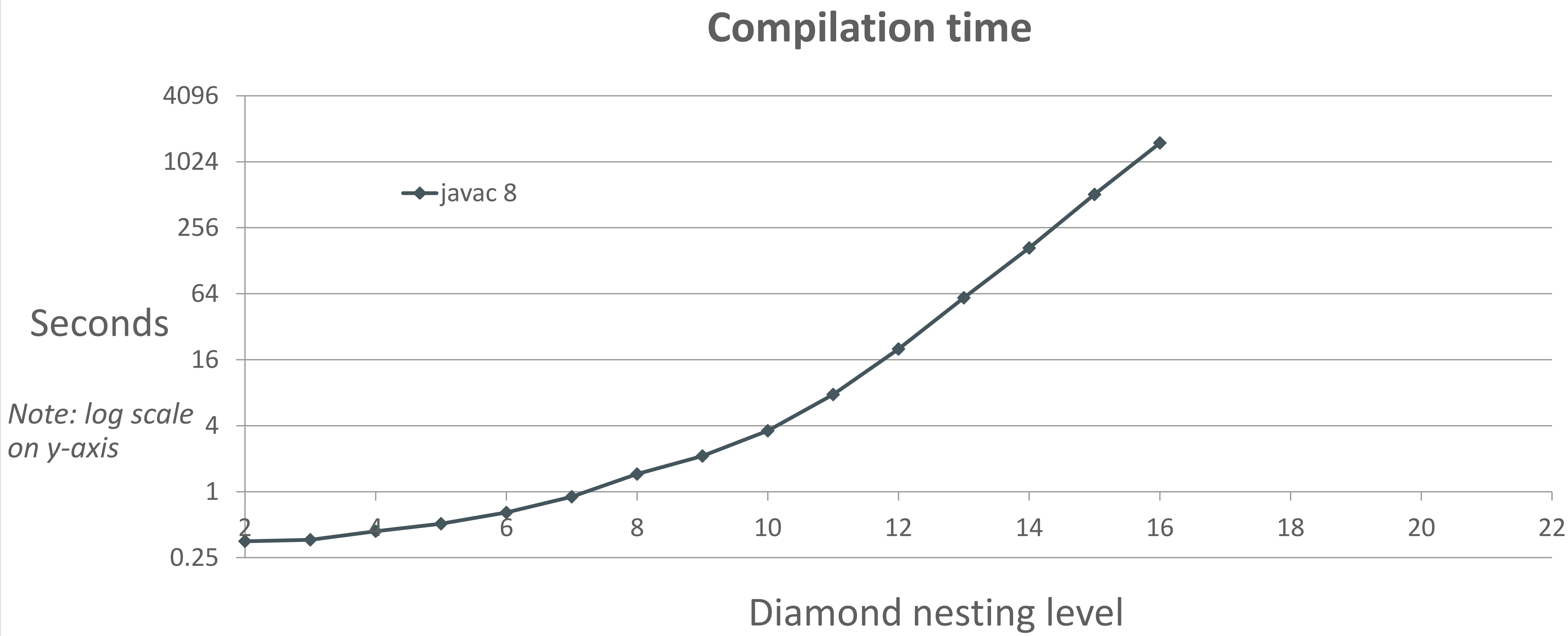
```
// compilation time: 0m49.278s
void foo() {
        m(null, () ->
            m(null, () ->
                m(null, () ->
                    m(null, () ->
                        m(null, () ->
                            m(null, () ->
                                m(null, () ->
                                    m(null, () ->
                                        m(null, (Callable<String>)null)))))))));
}
```

```
class Klass {
    static class A0 { }
    static class A1 { }
    static class A2 { }
    <Z extends A0> Z m(A0 t,
        Callable<Z> ct) { return null; }
    <Z extends A1> Z m(A1 t,
        Callable<Z> ct) { return null; }
    <Z extends A2> Z m(A2 t,
        Callable<Z> ct) { return null; }
    <Z> Z m(Object o,
        Callable<Z> co) { return null; }
}
```

Compiler performance, linear scale



Compiler performance, log scale



Note: log scale on y-axis

Exponential explosion

- Poly expressions are type checked several times against multiple target types. The same applies for nested expressions, like arguments
- Type inference got more complex and slower
- We can say that the Java 8 compiler is trying to solve a problem as complex as finding cheap tickets during Thanksgiving days :)

Exponential explosion, mitigating factors

- Occurs in limited settings: typically generated code from tools, IDEs, etc. Mainly for deeply nested calls which are generally not a recommended coding pattern.
- Very hard to find in user-written code
- No noticeable slowdown has been detected while compiling big projects like JDK
- There are alternatives to improve the performance if needed, such as adding explicit type arguments

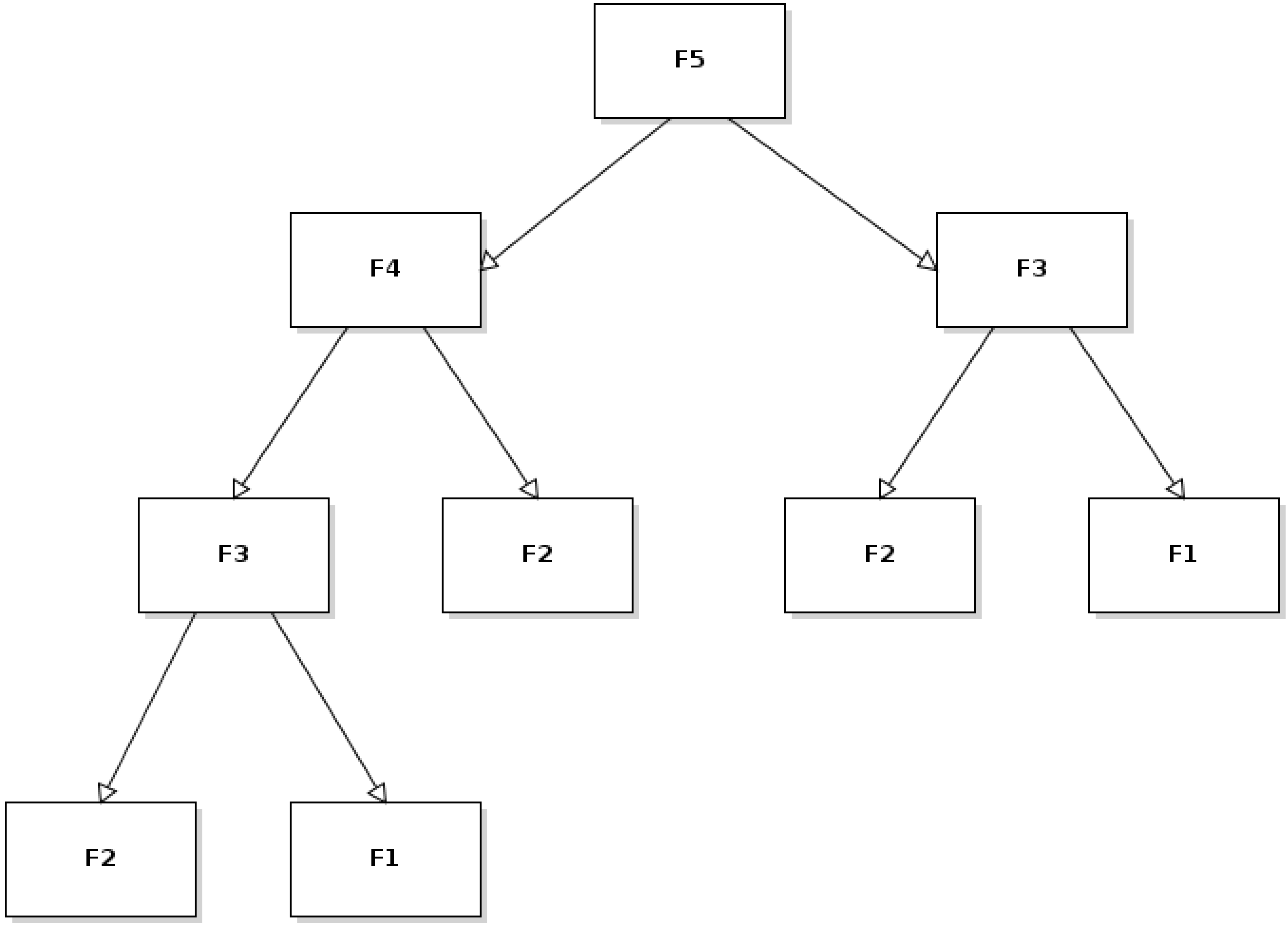
Tiered attribution goals

- Re-engineer the compiler in order to add performance robustness and avoid exponential slowdowns
- Improve compiler performance by reducing the number of passes needed to attribute a given expression
- Produce same binary results as the previous implementation

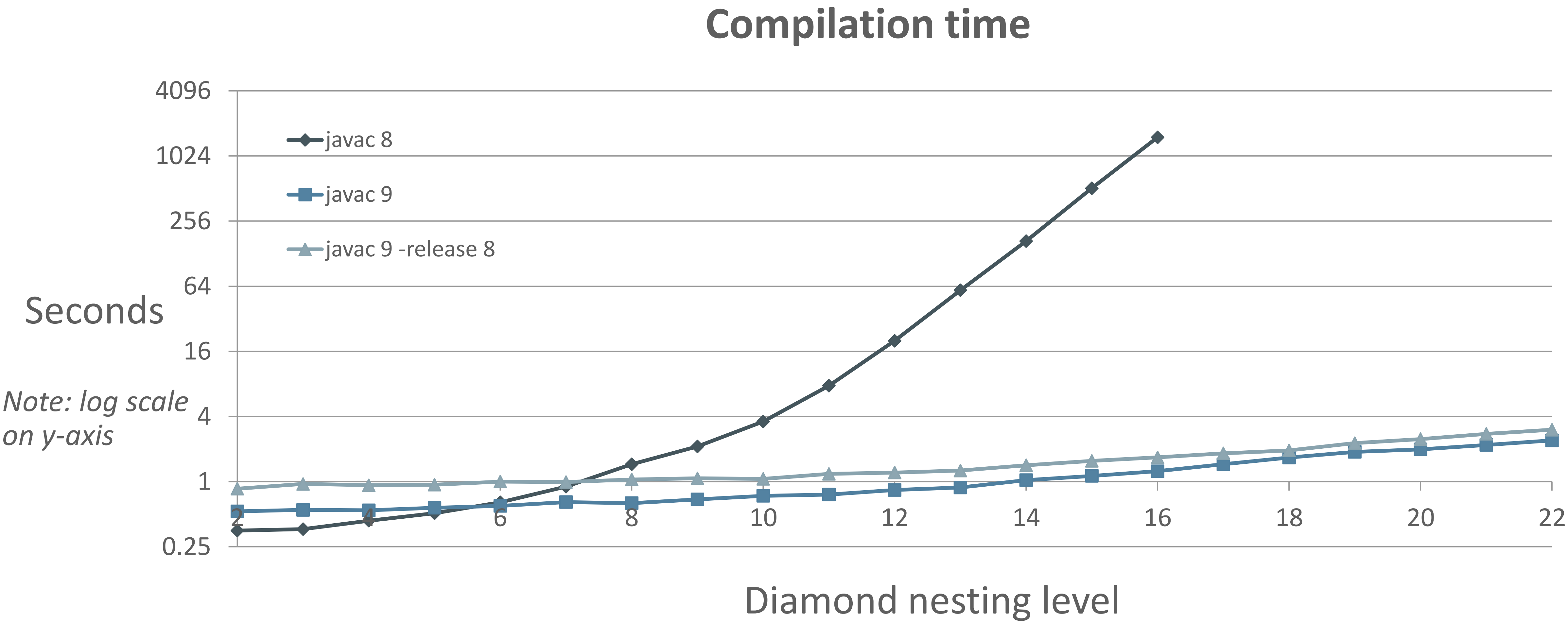
Main ideas behind Tiered Attribution

- Gather structural information about an expression
- Use that information during overload resolution to discriminate overloads
- Attribute every expression only once

Computing Fibonacci Numbers

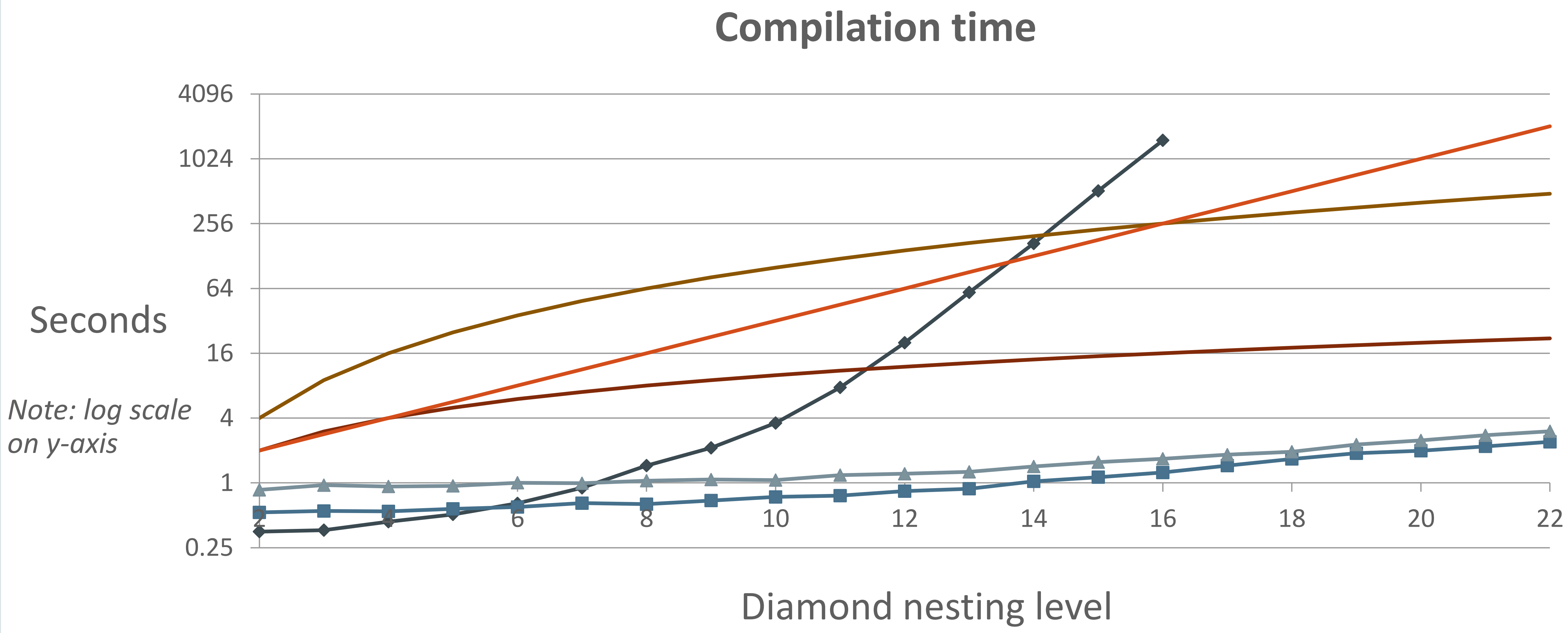


Compiler performance with tiered attribution

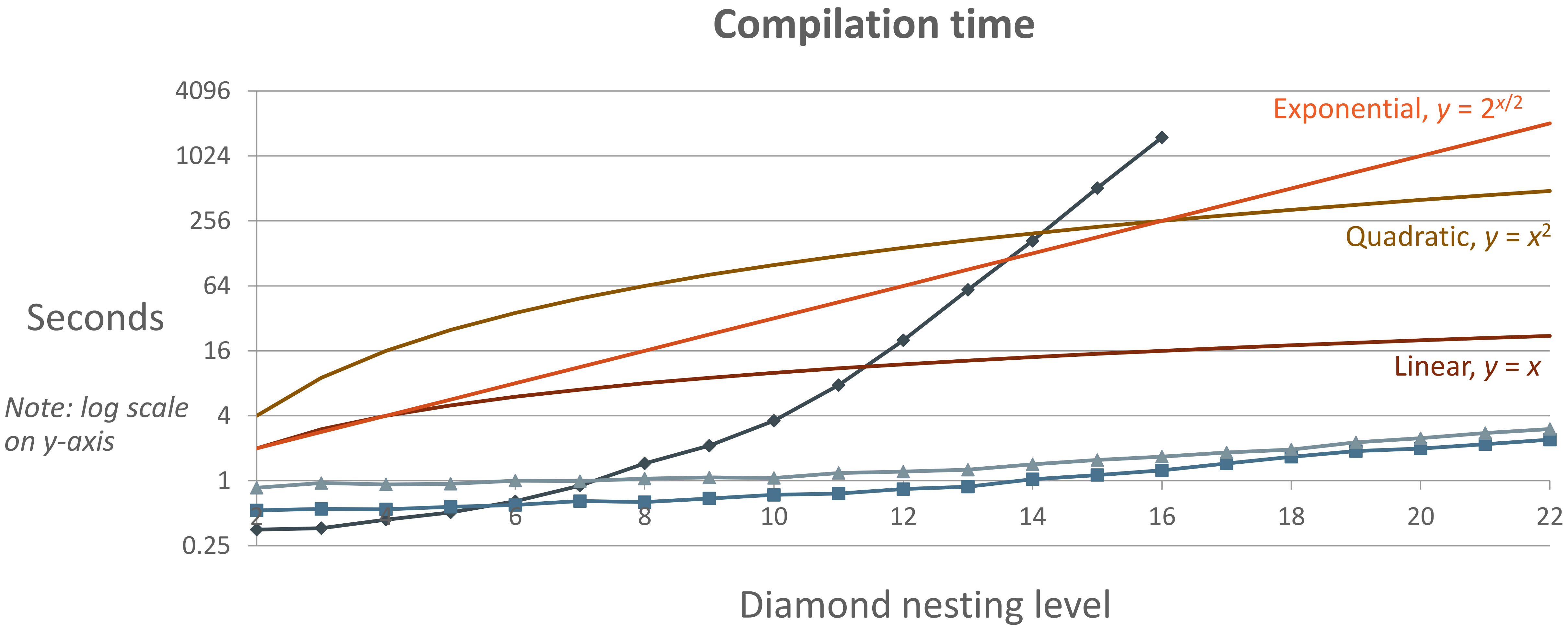


Note: log scale on y-axis

Compiler performance with tiered attribution, cont.



Compiler performance with tiered attribution, cont.



Note: log scale on y-axis

Tiered Attribution results

- Performance improvement, worst case dramatically improved
- Compatibility with existing attribution approach
- Opened the door to more optimizations and improvements

Summary

JDK 9 Language and Tooling features

- Fundamental development changes coming with modularity
- Smaller improvements coming in other areas:
  - Finish long-anticipated polishing of Project Coin, Project Lambda, and other language enhancements
  - Increased developer convenience; more informative warnings
  - More robust compiler performance
- Can follow developments in OpenJDK
- EA builds of JDK 9 available *today* for your evaluation

Acknowledgements

Engineers who worked on development, specification, QE, or conformance

- Leonid Arbuzov
- Srikanth Adayapalam
- Oleg Barbashov
- Dmitry Bessonov
- Alex Buckley
- Maurizio Cimadamore
- Andrei Eremeev
- Robert Field
- Joel Franck
- Jon Gibbons
- Brian Goetz
- Sonali Goel
- Jan Lahoda
- Andreas Lundblad
- Eric McCorkle
- Andrey Nazarov
- Ella Nekipelova
- Matherey Nunez
- Bhavesh Patel
- Sergei Pikalev
- Alexander Posledov
- Georgiy Rakov
- Victor Rudometov
- Steve Sides
- Dan Smith
- Kumar Srinivasan
- Elena Votchennikova

Q & A

Slides: https://blogs.oracle.com/darcy/resource/JavaOne/J1_2015-jdk9-langtools.pdf

JDK 9 EA builds: https://jdk9.java.net/

JDK 9 EA builds with Jigsaw: https://jdk9.java.net/jigsaw/

JavaOne™

ORACLE®