ORACLE®

Java 8 for Compiler Writers

Daniel Smith
JSR 335 Specification Author

* Subject to change

New Java SE 8 VM-related Features

- Default Methods
- Lambda Metafactory
- Type Annotations
- Misc.: Repeatable Annotations, Parameter Reflection

Default Methods: Overview

JSR 335

- Java source allows an interface to declare a method as "`default`" and give it a body.
- A default method's body should be invoked if the class hierarchy doesn't provide an implementation.
- Interfaces can also declare `private` and `static` methods, which are never inherited.

Default Methods: Class File Format

- Methods in interfaces don't have to be `abstract` (and thus permit `Code`)
- Methods in interfaces allow additional modifiers
- `invokestatic` and `invokespecial` accept `InterfaceMethodrefs` (the instructions are overloaded)

  * Applies to version 52.0+ class files

Default Methods: Permitted Interface Flags

🟩 `public`	🟩 `bridge`
🟨 `private`	🟩 `varargs`
▨ `protected`	▨ `native`
🟨 `static`	🟩 `abstract`
▨ `final`	🟨 `strict`
▨ `synchronized`	🟩 `synthetic`

Key
Green: previously permitted
Yellow: newly permitted
Grey: not permitted

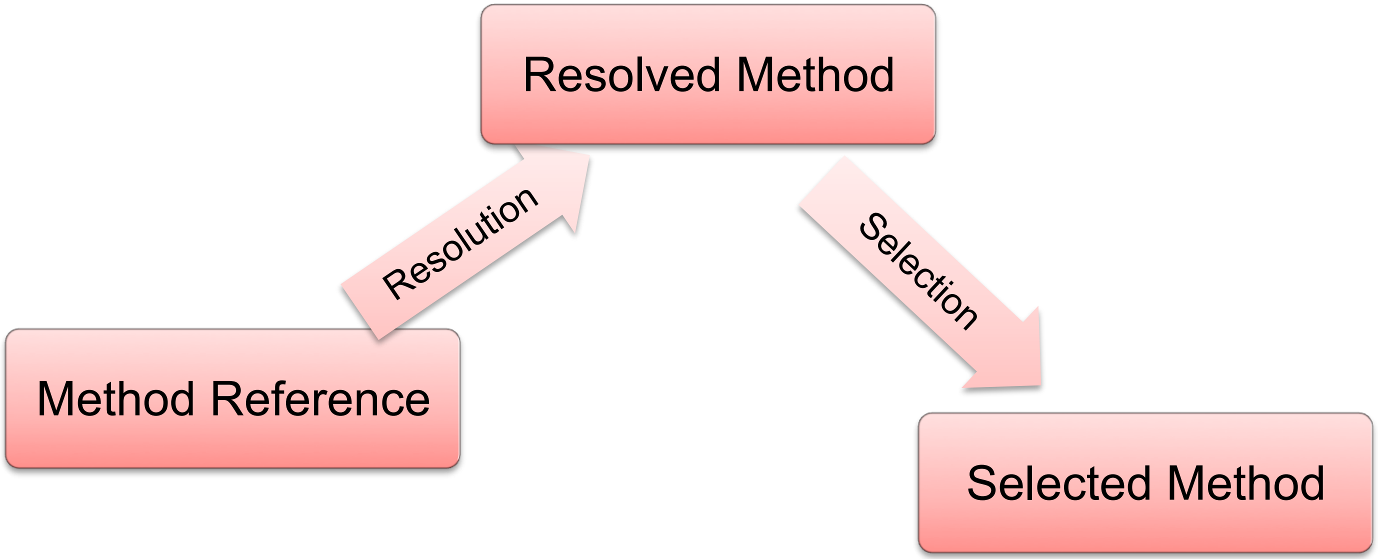
* All methods must be public or private (not package-access)

Default Methods: Permitted Invocation Forms

🟩 `invokevirtual C.m`	⬜ `invokevirtual I.m`
⬜ `invokeinterface C.m`	🟩 `invokeinterface I.m`
🟩 `invokestatic C.m`	🟨 `invokestatic I.m`
🟩 `invokespecial C.m`	🟨 `invokespecial I.m`

Key
Green: previously permitted
Yellow: newly permitted
Grey: not permitted

Default Methods: Semantics of Invocation



Default Methods: Maximally Specific Methods

The *maximally specific superinterface methods* of a **class** for a **name+descriptor** is the set of all methods satisfying:

- Declared in a superinterface
- Matching name and descriptor
- Neither `private` nor `static`
- Not trumped by a satisfactory method in a subinterface

Default Methods: Maximally Specific Example

```
interface I { void m(); }
interface J { default void m() { System.out.println(“J.m”); } }
interface K extends I { default void m() { System.out.println(“K.m”); } }

class C implements I, J {}
class D extends C implements K {}

Result for C: { I.m, J.m }
Result for D: { J.m, K.m }
```

Default Methods: Resolution

Resolving method reference `T.m()V`

- Try T
- Try T’s superclasses
- Try the maximally specific superinterface methods
  - Pick one
- `NoSuchMethodError`

Default Methods: Selection

Selecting an implementation of `U.m()V` from `S`

- Try `S`
- Try `S`'s superclasses
- Try the maximally specific superinterface methods
  - If exactly one is non-`abstract`, select it
- `AbstractMethodError` or `IncompatibleClassChangeError`

Default Methods: Semantics of `invokeinterface`

- Resolve `I.m()V` (result is an interface method or an `Object` method)
- Select an implementation from the receiver's class

(By design, affects behavior of invocations in old class files.)

Default Methods: Semantics of `invokevirtual`

- Resolve `C.m()V` (result is a class method or an interface method)
- Select an implementation from the receiver's class

(By design, affects behavior of invocations in old class files.)

Default Methods: Semantics of `invokestatic`

- For class methods, no change
- For interface methods:
  - Resolve `I.m()V`
  - Select the resolved method

Default Methods: Semantics of `invokespecial`

- Three instructions in one (other references are prohibited):
  - Invoke `<init>` methods
  - Invoke a class's or interface's own methods (probably `private`)
  - Invoke superclass or direct superinterface methods

Default Methods: Semantics of `invokespecial`

Invoking super methods

- For class methods (where current class `D` is a subclass of `C`):
  - Resolve `C.m()V`
  - Select an implementation from the superclass of `D`*
- For interface methods (where current class `D` implements `I`):
  - Resolve `I.m()V`
  - Select an implementation from `I`

* Assuming `ACC_SUPER` is set

Default Methods: Summary

- In version 52.0 class files:
  - Interface methods don't have to be `abstract`, can be `public`/`private` and instance/`static`
  - `invokestatic` and `invokespecial` can reference interface methods
- In all class files:
  - Resolution and selection are updated to new inheritance model

Lambda Metafactory: Overview

- Lambda expressions and method refs in Java source are compiled to:
  - A method
  - Captured values
  - A target functional interface
  - An `invokedynamic` call to a runtime library
- Evaluation produces an object that:
  - Implements the interface via the method
  - Stores the captured values

Lambda Metafactory: Contract

- Inputs
  - A set of interfaces to implement { *Predicate, Serializable* }
  - Types of captured values (*String, int*)
  - A method name "*test*"
  - A set of method descriptors to implement { *(Object)Z* }
  - A generics-instantiated descriptor *(File)Z*
  - A method implementation *SomeClass.lambda$0*
- Output: a factory (*String, int*) → *Predicate* & *Serializable*

Lambda Metafactory: API

```
package java.lang.invoke;

public class LambdaMetafactory {
  public static CallSite metafactory(MethodHandles.Lookup caller,
                                     String invokedName,
                                     MethodType invokedType,
                                     MethodType samType,
                                     MethodHandle implMethod,
                                     MethodType instantiatedMethodType);
...
```

Lambda Metafactory: API

```
  public static CallSite altMetafactory(MethodHandles.Lookup caller,
                                       String invokedName,
                                       MethodType invokedType,
                                       Object... args);
                                    // MethodType samType,
                                    // MethodHandle implMethod,
                                    // MethodType instantiatedMethodType,
                                    // int flags,
                                    // int icount, Class... markerInterfaces,
                                    // int tcount, MethodType... bridges);
}
```

Lambda Metafactory: Relevance

- Strictly speaking, just a library
- But highly optimized (ideally…) for the VM
- Java is committed to it, other compilers can benefit from the free engineering work

Type Annotations: Overview

JSR 308

- Java source allows type uses and type parameter declarations to be annotated
  - `@Target(ElementType.TYPE_USE)`
  - `@Target(ElementType.TYPE_PARAMETER)`
- Annotations can be processed by a tool or compiler plug-in to enforce custom typing rules (e.g., `@NotNull`)

Type Annotations: Class File Attributes

- New attributes:
  - `RuntimeVisibleTypeAnnotations`
  - `RuntimeInvisibleTypeAnnotations`
- Stored on the smallest enclosing class, field, method, or `Code`

Type Annotations: Contents of an Annotation

```
type_annotation {
  target_type; // the type of the targeted program element
  target_info; // identifies the targeted program element
  target_path; // identifies targeted type in a compound type
  type_index;
  element_value_pairs;
}
```

Type Annotations: Accessing

- `javax.lang.model`
- `javax.ide`
- `com.sun.source.tree`

Repeatable Annotations

- Java source supports multiple uses of the same annotation instance if the annotation
- `@Repeatable` to opt in and define the container annotation type
- No VM impact

Parameter Reflection

- New attribute: `MethodParameters`

  Consists of a list of names and access flags
- Compilers should provide an opt-in facility
- Access reflectively with `Method.getParameters()`

  By default, “`arg0`”, “`arg1`”, …

New Java SE 8 VM-related Features

- Default Methods
- Lambda Metafactory
- Type Annotations
- Misc.: Repeatable Annotations, Parameter Reflection

Hardware and Software

ORACLE®

Engineered to Work Together