Brian Goetz (Oracle) & Dan Heidinga (IBM)

30 July 2013

JVM bridge methods: a road not taken

Important Disclaimers

- THE INFORMATION CONTAINED IN THIS PRESENTATION IS PROVIDED FOR INFORMATIONAL PURPOSES ONLY.
- WHILST EFFORTS WERE MADE TO VERIFY THE COMPLETENESS AND ACCURACY OF THE INFORMATION CONTAINED IN THIS PRESENTATION, IT IS PROVIDED “AS IS”, WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED.
- ALL PERFORMANCE DATA INCLUDED IN THIS PRESENTATION HAVE BEEN GATHERED IN A CONTROLLED ENVIRONMENT. YOUR OWN TEST RESULTS MAY VARY BASED ON HARDWARE, SOFTWARE OR INFRASTRUCTURE DIFFERENCES.
- ALL DATA INCLUDED IN THIS PRESENTATION ARE MEANT TO BE USED ONLY AS A GUIDE.
- IN ADDITION, THE INFORMATION CONTAINED IN THIS PRESENTATION IS BASED ON IBM’S CURRENT PRODUCT PLANS AND STRATEGY, WHICH ARE SUBJECT TO CHANGE BY IBM, WITHOUT NOTICE.
- IBM AND ITS AFFILIATED COMPANIES SHALL NOT BE RESPONSIBLE FOR ANY DAMAGES ARISING OUT OF THE USE OF, OR OTHERWISE RELATED TO, THIS PRESENTATION OR ANY OTHER DOCUMENTATION.
- NOTHING CONTAINED IN THIS PRESENTATION IS INTENDED TO, OR SHALL HAVE THE EFFECT OF:
- - CREATING ANY WARRANT OR REPRESENTATION FROM IBM, ITS AFFILIATED COMPANIES OR ITS OR THEIR SUPPLIERS AND/OR LICENSORS

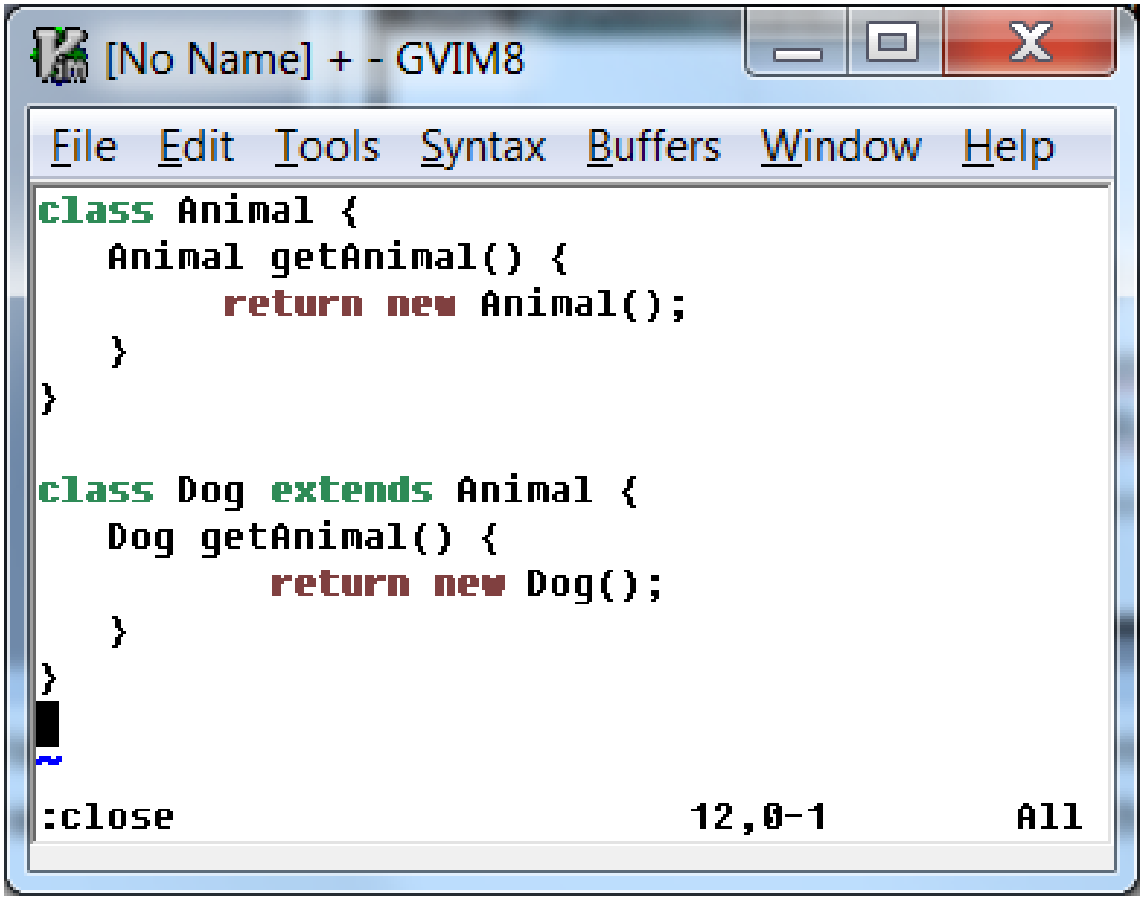
Outline

- Background on bridge methods
  - What they are and where they come from
- Why bridge methods matter for JSR 335's default methods
  - Possible implementation strategies
- The chosen solution and future directions

"today's problems courtesy of yesterday's solutions."

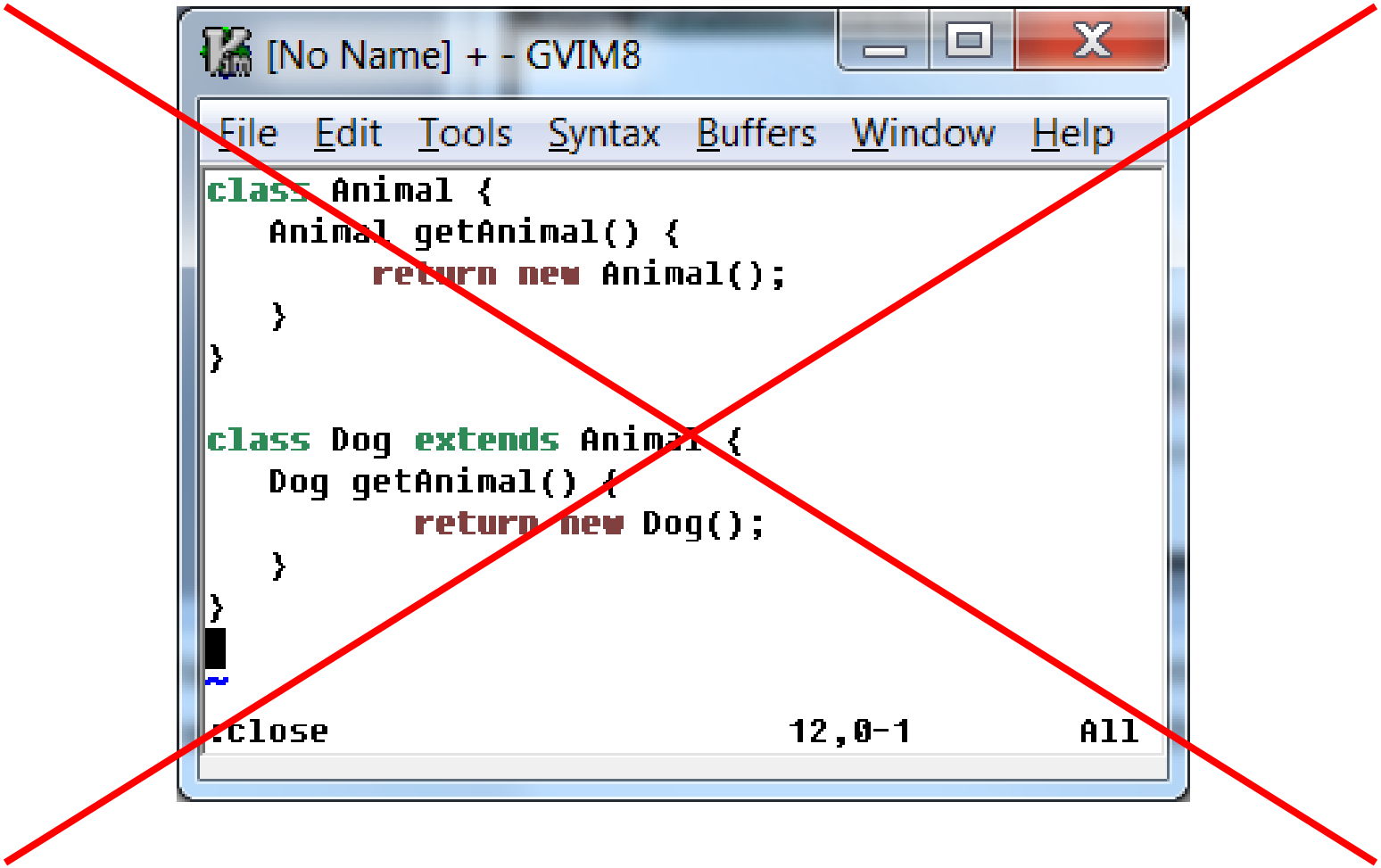
Source: If applicable, describe source origin

Covariant returns



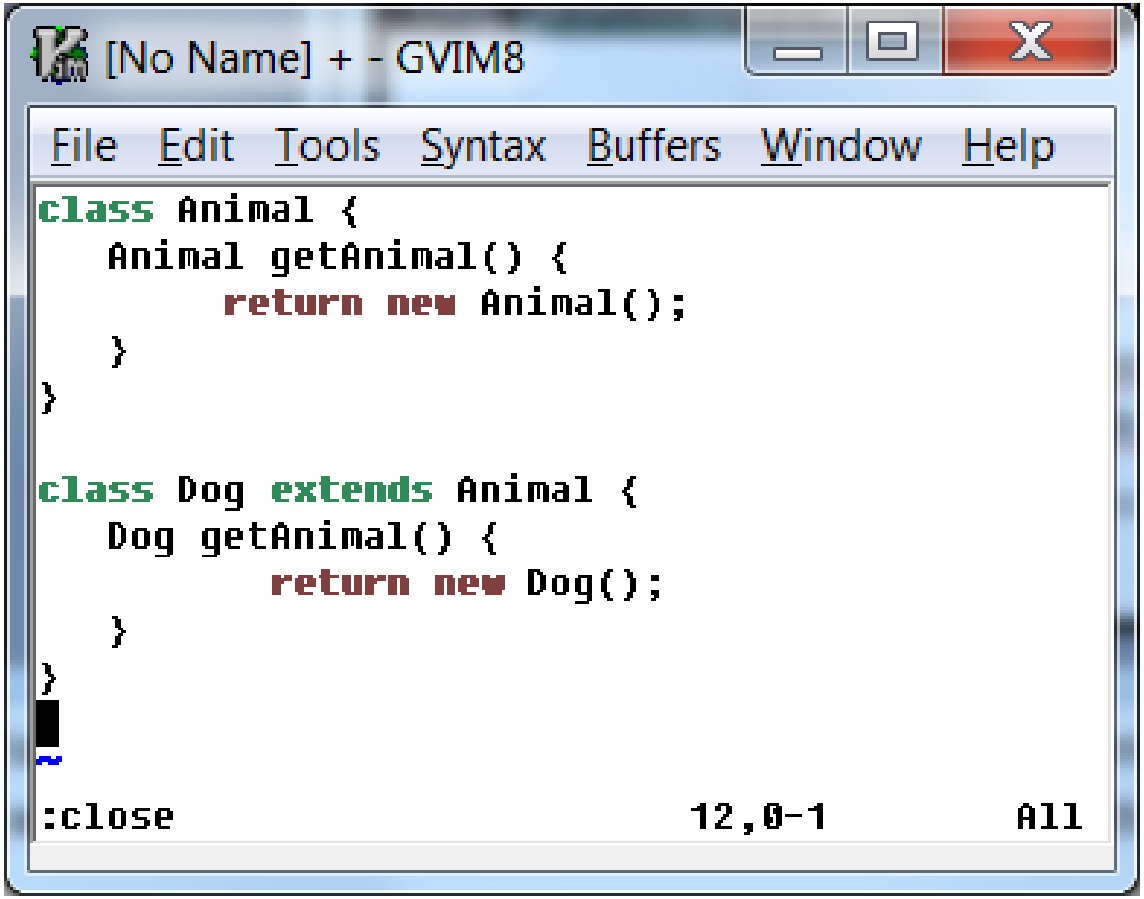
Covariant returns

Java 1



Covariant returns – new in Java 5

Java 5

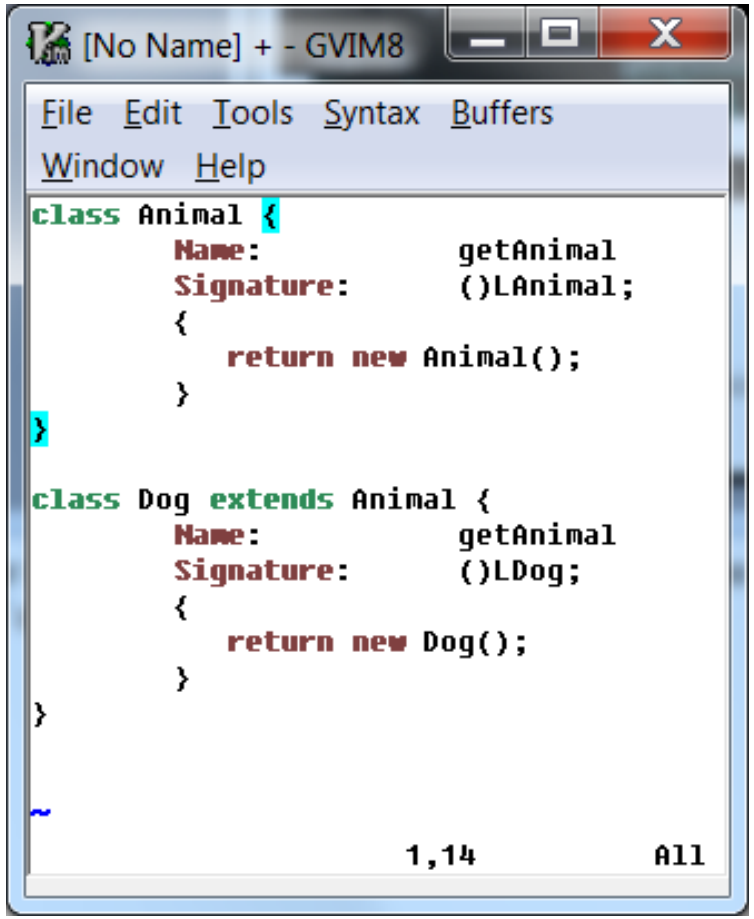


- Since Java 5, Dog's getAnimal() overrides Animal's despite having different return types.

```
Animal anAnimal = new Dog();
anAnimal.getAnimal() instanceof Dog == true
```

Covariant method overriding – JVM rules

Java 5



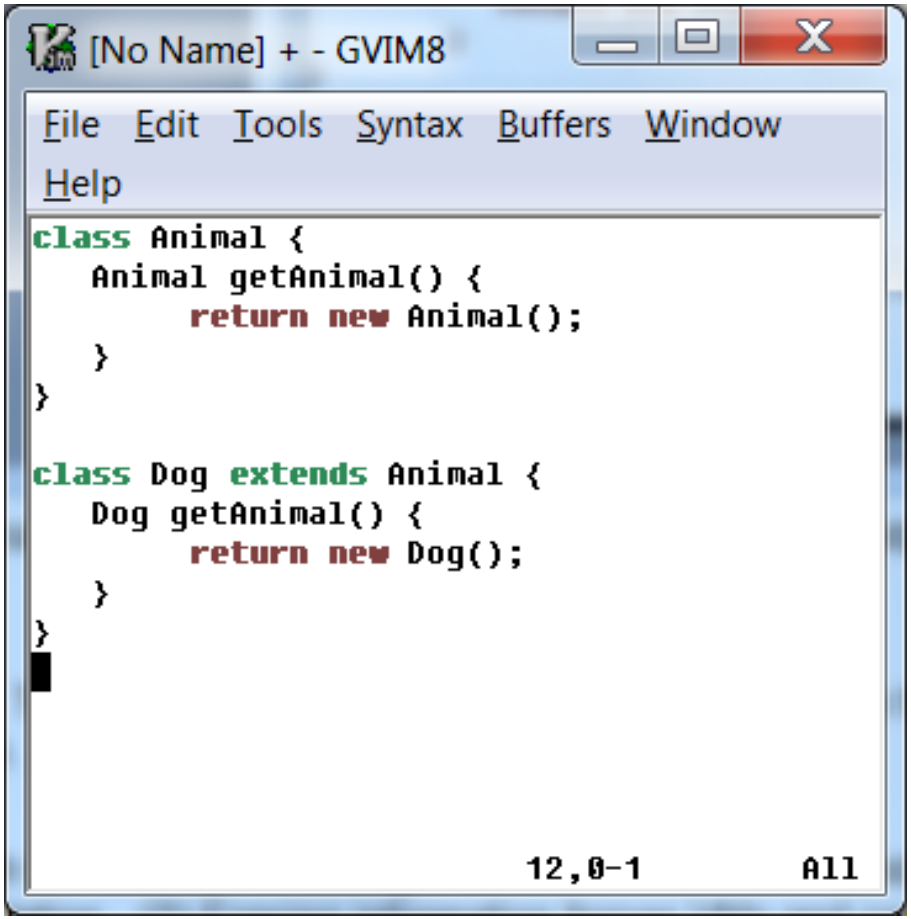
5.4.5 Method overriding

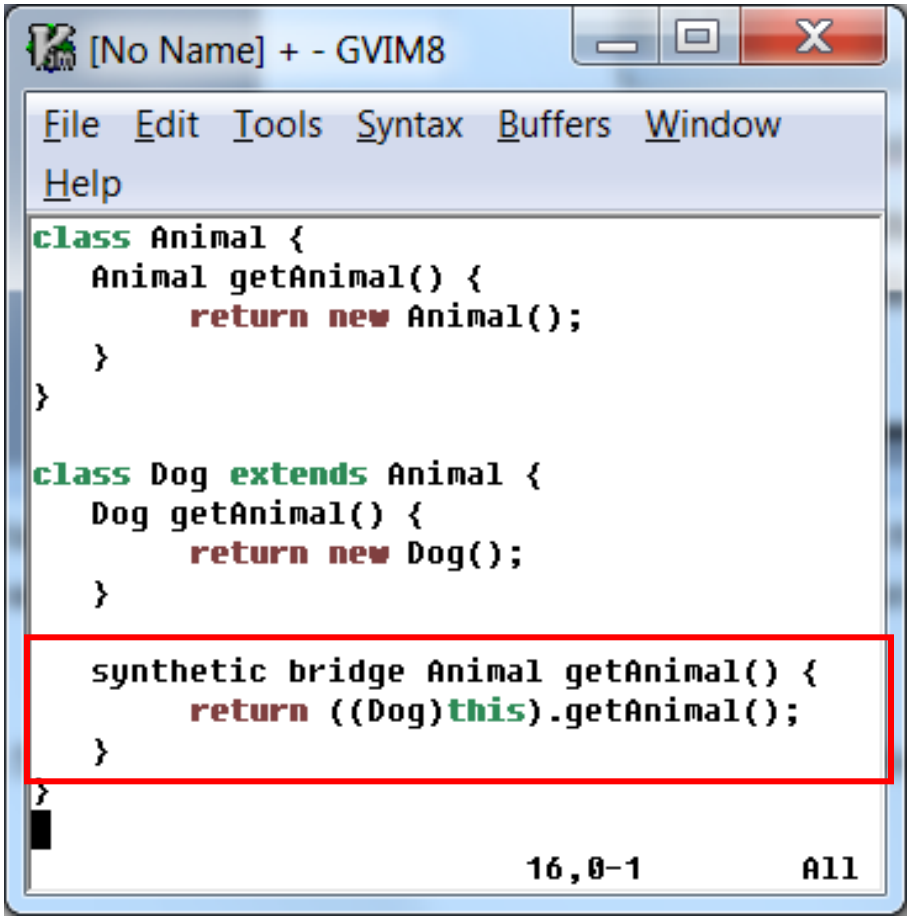
An instance method *m1* declared in class C overrides another instance method *m2* declared in class A iff all of the following are true:

- C is a subclass of A.
- *m2* has the same name and descriptor as *m1*.
- Either:
  - *m2* is marked ACC_PUBLIC; or is marked ACC_PROTECTED; or is marked neither ACC_PUBLIC nor ACC_PROTECTED nor ACC_PRIVATE and belongs to the same runtime package as C, or
  - *m1* overrides a method *m3*, *m3* distinct from *m1*, *m3* distinct from *m2*, such that *m3* overrides *m2*.

Covariant return: bridge method

Java 5





- Javac generates a method to *bridge* between the two signatures of getAnimal()

Bridge methods: working definition

Java 5

"Synthetic methods that bridge between Java the language's and the JVM's view of types; are generated at classfile creation"

- JVM specification 3rd edition says:

The ACC_BRIDGE flag is used to indicate a bridge method generated by the compiler.

ACC_BRIDGE	0x0040	A bridge method, generated by the compiler.

All interface methods must have their ACC_ABSTRACT and ACC_PUBLIC flags set; they may have their ACC_VARARGS, ACC_BRIDGE and ACC_SYNTHETIC flags set and must not have any of the other flags in Table 4.5 set (JLS §9.4).

Bridge methods in practice

Java 5

CC image from: http://www.flickr.com/photos/laundry/4974136834/sizes/m/in/photolist-8zxKzU-dV26mC-dsaPMe-8tmBrk-8tmBqB-8xkpGa-8tmBrg-dsaw12-ek5uHE-9L1bes-adNU8v-advB8p-ae5mL4-aeNueX-aefiYc-adyLPs/

- Bridge method origins:
  - differing definitions of method override
  - the desire to leave all linkage decisions to runtime.

Leaky abstraction

Java 5

- The Big Hammer of Java 5 was "we'll fake out the VM by erasing stuff“

Costs:

- Bridge methods break the 1-to1 correspondence between source code and classfile
- Bridge methods leak into stack traces
- java.lang.reflect.Method.isBridge()
- JVMTI can see and modify them
- Users generating classfiles can mark anything as a bridge -> can’t be trusted

```
Exception in thread "main" java.lang.Error
        at Dog.getAnimal(Dog.java:2)
        at Dog.getAnimal(Dog.java:1)
        at Dog.main(Dog.java:6)
```

Separate compilation – Contravariant underride

Java 5

```
abstract class AbstractCallback {
   /* Added later */
   abstract Object callback();
}
```

```
class Test {
   public static void main(String[] args) {
            AbstractCallback a = new C();
           /* Added later */
           Object o = a.callback();
   }
}
```

```
class C extends AbstractCallback {
   String callback() { return "C"; }
}
```

Separate compilation – AbstractMethodError

Java 5

```
abstract class AbstractCallback {
   /* Added later */
   abstract Object callback();
}
```

```
class Test {
   public static void main(String[] args) {
          AbstractCallback a = new C();
          /* Added later */
          Object o = a.callback();
   }
}
```

```
Exception in thread "main" java.lang.AbstractMethodError: AbstractCallback.callback()Ljava/lang/Object;
      at Test.main(Test.java:5)
```

```
class C extends AbstractCallback {
   String callback() { return "C"; }
}
```

Separate compilation problems – 3 classes

Java 5

```
class Foo {
 public Object m() {
       return this;
 }
}
```

```
class Bar extends Foo {
}
```

```
class Zoo extends Bar {
 public String m() {
       return super.m().toString();
 }
 public static void main(String[] args) {
       Foo f = new Zoo();
       Object o = f.m();
 }
}
```

Separate compilation problems - bytecode

Java 5

```
class Foo {
  public java.lang.Object m();
  0: aload_0
  1: areturn
}

class Zoo extends Bar {
  public java.lang.String m();
  0: aload_0
  1: invokespecial Bar.m:()Ljava/lang/Object;
  4: invokevirtual Object.toString:()Ljava/lang/String;
  7: areturn

  public synthetic bridge java.lang.Object m();
  0: aload_0
  1: invokevirtual m:()Ljava/lang/String;
  4: areturn

  public static void main(java.lang.String[]);
  9: invokevirtual Foo.m:()Ljava/lang/Object;
}
```

```
class Bar extends Foo {
}
```

Separate compilation problems - callstack

Java 5

```
class Foo {
  public java.lang.Object m();
  0: aload_0
  1: areturn
}

class Zoo extends Bar {
  public java.lang.String m();
  0: aload_0
  1: invokespecial Bar.m:()Ljava/lang/Object;
  4: invokevirtual Object.toString:()Ljava/lang/String;
  7: areturn

  public synthetic bridge java.lang.Object m();
  0: aload_0
  1: invokevirtual m:()Ljava/lang/String;
  4: areturn

  public static void main(java.lang.String[]);
  9: invokevirtual Foo.m:()Ljava/lang/Object;
}
```

```
class Bar extends Foo {
}
```

Foo.m()Object

Zoo.m()String

Zoo.m()Object

Separate compilation problems – recompile Bar

Java 5

```
class Foo {
 public Object m() {
        return this;
 }
}
```

```
class Bar extends Foo {
 /* Separate compilation: added later */
 public String m() {
        return super.m().toString();
 }
}
```

```
class Zoo extends Bar {
 public String m() {
        return super.m().toString();
 }
 public static void main(String[] args) {
        Foo f = new Zoo();
        Object o = f.m();
 }
}
```

Separate compilation problems - bytecode

Java 5

```
class Foo {
  public java.lang.Object m();
  0: aload_0
  1: areturn
}

class Zoo extends Bar {
  public java.lang.String m();
  0: aload_0
  1: invokespecial Bar.m:()Ljava/lang/Object;
  4: invokevirtual Object.toString:()Ljava/lang/String;
  7: areturn

  public synthetic bridge java.lang.Object m();
  0: aload_0
  1: invokevirtual m:()Ljava/lang/String;
  4: areturn

  public static void main(java.lang.String[]);
  9: invokevirtual Foo.m:()Ljava/lang/Object;
}
```

```
class Bar extends Foo {
 public java.lang.String m();
 0: aload_0
 1: invokespecial Foo.m:()Ljava/lang/Object;
 4: invokevirtual Object.toString:()Ljava/lang/String;
 7: areturn

 public synthetic bridge java.lang.Object m();
 0: aload_0
 1: invokevirtual m:()Ljava/lang/String;
 4: areturn
}
```

Separate compilation problems - bytecode

Java 5

```
class Foo {
  public java.lang.Object m();
  0: aload_0
  1: areturn
}

class Zoo extends Bar {
  public java.lang.String m();
  0: aload_0
  1: invokespecial Bar.m:()Ljava/lang/Object;
  4: invokevirtual Object.toString:()Ljava/lang/String;
  7: areturn

  public synthetic bridge java.lang.Object m();
  0: aload_0
  1: invokevirtual m:()Ljava/lang/String;
  4: areturn

  public static void main(java.lang.String[]);
  9: invokevirtual Foo.m:()Ljava/lang/Object;
}
```

```
class Bar extends Foo {
 public java.lang.String m();
 0: aload_0
 1: invokespecial Foo.m:()Ljava/lang/Object;
 4: invokevirtual Object.toString:()Ljava/lang/String;
 7: areturn

 public synthetic bridge java.lang.Object m();
 0: aload_0
 1: invokevirtual m:()Ljava/lang/String;
 4: areturn
}
```

Separate compilation problems

Java 5

```
Exception in thread "main" java.lang.StackOverflowError
        at Bar.m(Bar.java:1)
        at Zoo.m(Zoo.java:3)
        at Bar.m(Bar.java:1)
```

```
        at Bar.m(Bar.java:1)
        at Zoo.m(Zoo.java:3)
        at Bar.m(Bar.java:1)
        at Zoo.m(Zoo.java:3)
        at Bar.m(Bar.java:1)
        at Zoo.m(Zoo.java:3)
        at Bar.m(Bar.java:1)
        at Zoo.m(Zoo.java:3)
        at Bar.m(Bar.java:1)
        at Zoo.m(Zoo.java:3)
        at Bar.m(Bar.java:1)
        at Zoo.m(Zoo.java:3)
        at Bar.m(Bar.java:1)   <----------
        at Zoo.m(Zoo.java:3)
        at Zoo.m(Zoo.java:1)   <----------
        at Zoo.main(Zoo.java:8)
```

Generic Substitution – new in Java 5

Java 5

```
abstract class C<T> {
   abstract T id(T x);
}

class D extends C<String> {
   String id(String x) { return x; }
}
```

Generic Substitution – classfile view

Java 5

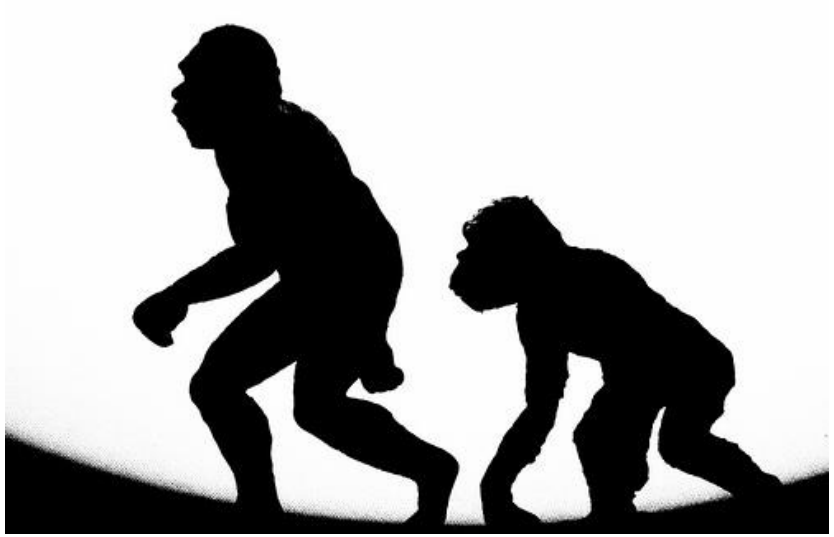
```
abstract class C<T> {
   Name:              id
   Signature:         (Ljava/lang/Object;) Ljava/lang/Object;
}
class D extends C<String> {
   Name:              id
   Signature:         (Ljava/lang/String;) Ljava/lang/String;
   {
            return x;
   }
   Name:              id
   Signature:         (Ljava/lang/Object;) Ljava/lang/Object;
   {
            aload 1
            checkcast String
            return (String)id((String)x);
   }
}
```

Why all this talk about Java 5 features?

Java 8

- One of the goals of JSR 335 (Lambda) is to enable interface evolution
  - Little point in adding cool new features like Lambda if you can't use them in old code
  - Binary compatibility prevents the JDK from modifying Collections interfaces

- Solution: allow interface methods to have a default implementation.
  - Allows interfaces to be upgraded as implementers will get the default behaviour if they haven't implemented the method



CC image from:http://www.flickr.com/photos/pagedooley/3191664147/sizes/m/in/photolist-5S37yc-5Z2Kre-8Cfqf7-eyTcty-e5Vi55/

Description of default methods

Java 8

- Allow interface contracts to evolve without breaking existing implementers
- Avoids the “garbage class” to hold static helper methods.
- Declaration-site and virtual.
- Avoid the brittleness of static extension methods

```
default void forEach(Consumer<? super T> action) {
        Iterator<T> iter = iterator();
        while (iter.hasNext()) {
                action.accept(iter.next());
        }
}
```

Description of default methods

Java 8

- Allow interface contracts to evolve without breaking existing implementers
- Avoids the “garbage class” to hold static helper methods.
- Declaration-site and virtual.
- Avoid the brittleness of static extension methods

```
public void forEach(java.util.Consumer<? super T>);
  Code:
     0: aload_0
     1: invokeinterface #1,  1            // InterfaceMethod iterator:()Ljava/util/Iterator;
     6: astore_2
     7: aload_2
     8: invokeinterface #2,  1            // InterfaceMethod java/util/Iterator.hasNext:()Z
    13: ifeq          31
    16: aload_1
    17: aload_2
    18: invokeinterface #3,  1            // InterfaceMethod java/util/Iterator.next:()Ljava/lang/Object;
    23: invokeinterface #4,  2            // InterfaceMethod java/util/Consumer.accept:(Ljava/lang/Object;)V
    28: goto          7
    31: return
```

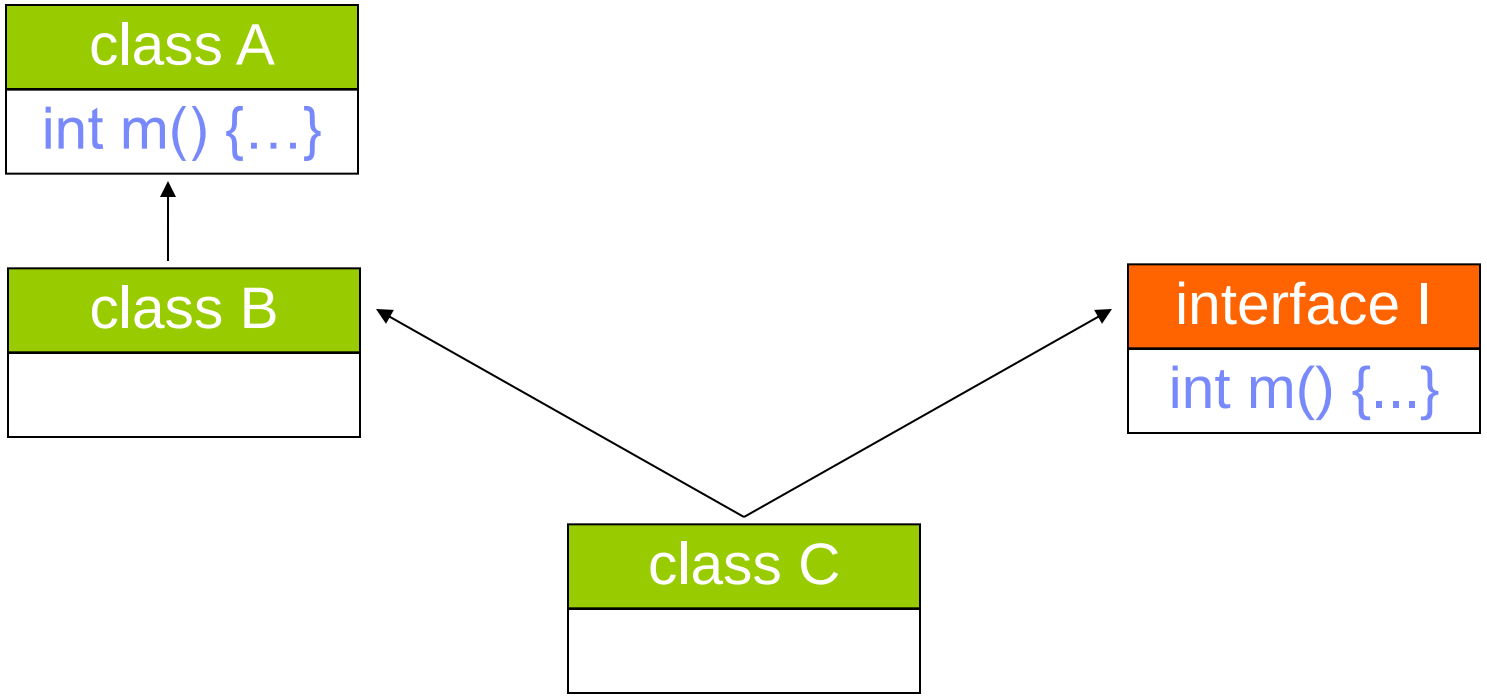
Default method inheritance

Java 8

- There are three rules for how this affects inheritance semantics:
  - If there is a declaration (concrete or abstract) from a superCLASS, ignore all superINTERFACES. That is, class-interface conflicts always resolved towards classes.
  - If there are two or more versions of a method inherited from superinterfaces, prefer a subtype over a supertype.
  - If there is not exactly one version from a superinterface, make the class provide an implementation (as if the method were abstract.)

superCLASS overrides superINTERFACE

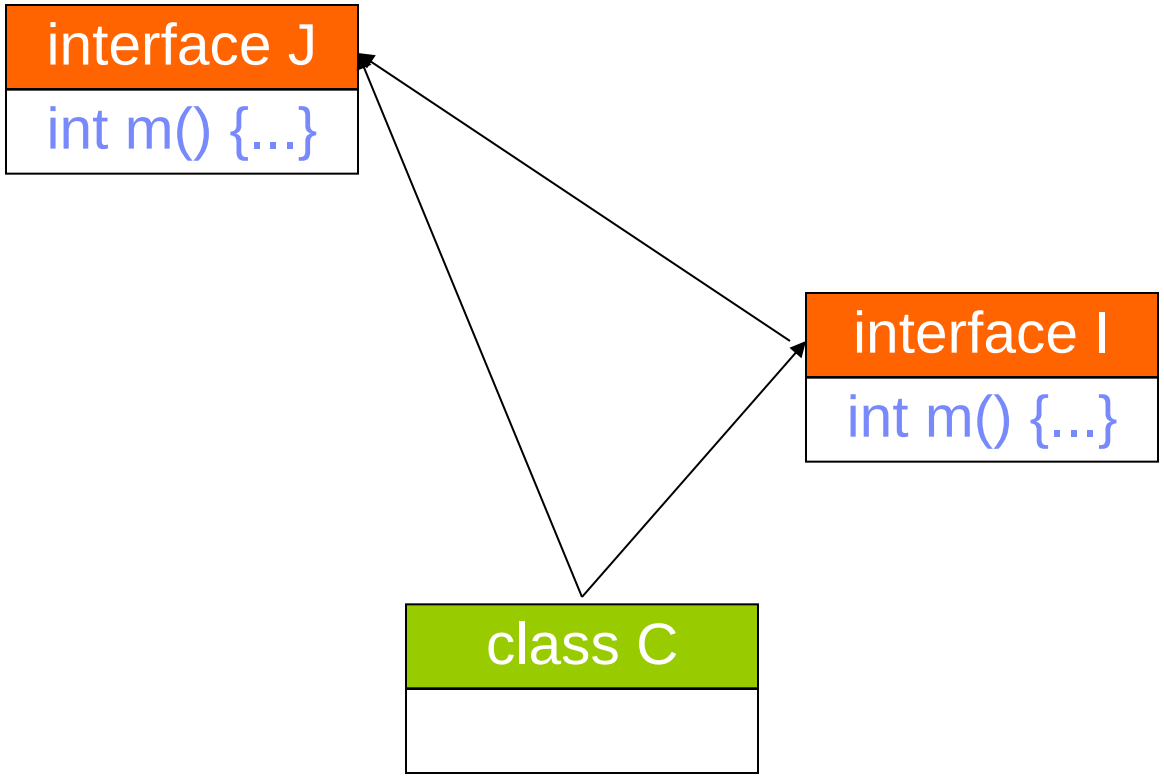
Java 8



- If there is a declaration (concrete or abstract) from a superCLASS, ignore all superINTERFACES. That is, class-interface conflicts always resolved towards classes.

Prefer most specific interface

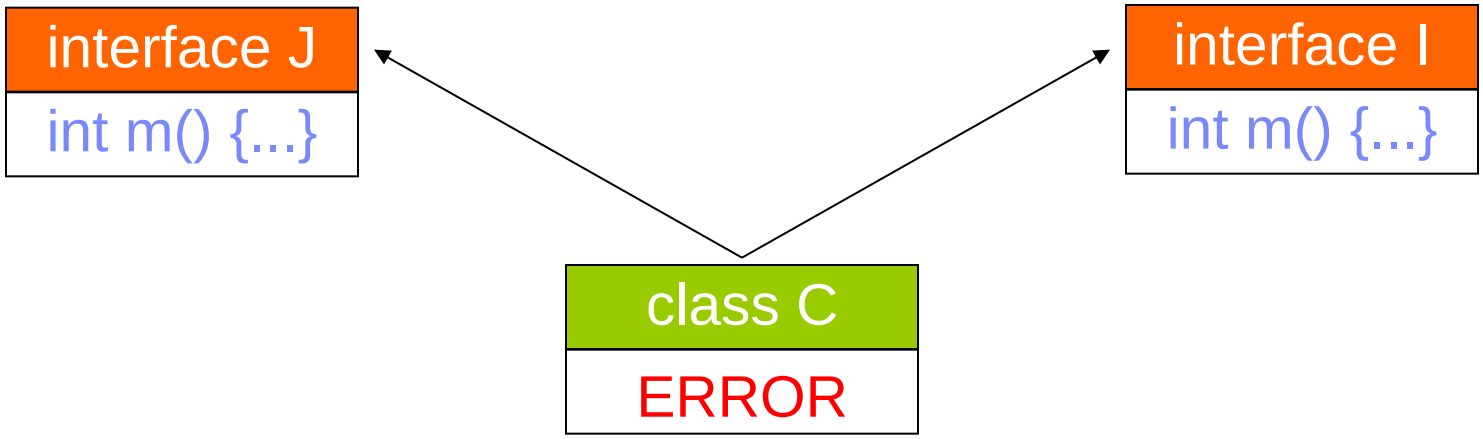
Java 8



- If there are two or more versions of a method inherited from superinterfaces, prefer a subtype over a supertype.

superCLASS overrides superINTERFACE

Java 8



- If there is not exactly one version from a superinterface, make the class provide an implementation (as if the method were abstract.)

Default methods and bridges

Java 8

CC image from: http://en.wikipedia.org/wiki/File:GoldenGateBridge-001.jpg

Default methods and bridges

```
interface Collection<T> {
   default Collection<T> unmodifiable() { ... }
}

interface List<T> extends Collection<T> {
   default List<T> unmodifiable() { ... }
}
```

```
List myList = ...
Collection c = myList;
c.unmodifiable();
```

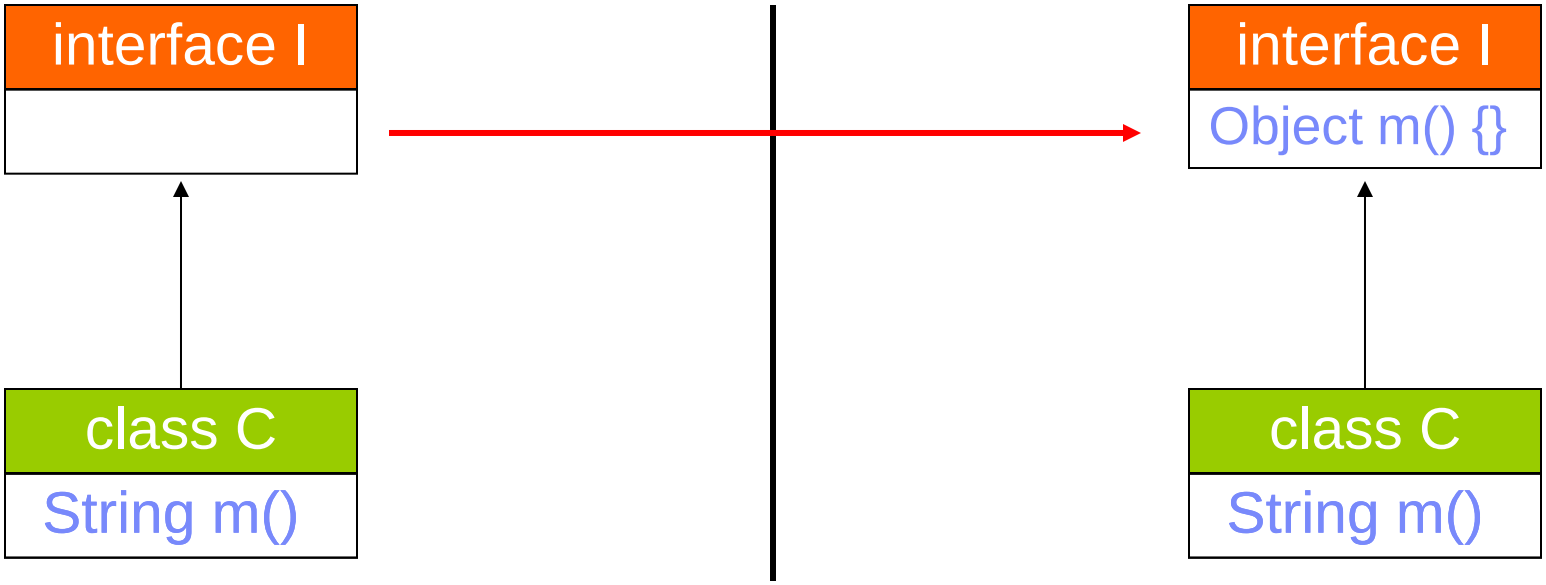
Default methods and bridges

```
interface Collection<T> {
   default Collection<T> unmodifiable() {... }
}

interface List<T> extends Collection<T> {
   default List<T> unmodifiable() { ... }

   default synthetic bridge Collection unmodifiable() { ... }
}
```

```
List myList = ...
Collection c = myList;
c.unmodifiable();
```

Add a new default method

Java 8

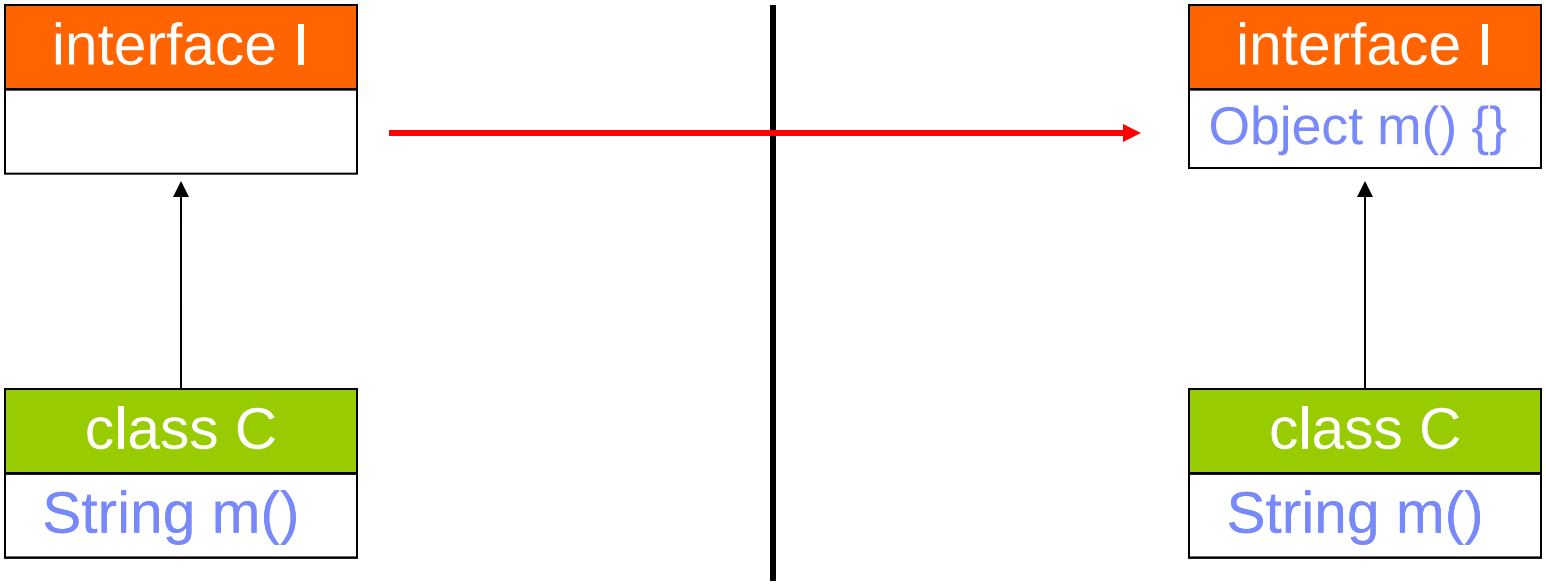


invokevirtual m()Object ⟶ I.m()

Wait a minute! Shouldn't that be C.m()?

Road 1: use invokedynamic

Java 8



invokedynamic m()Object → C.m() Always

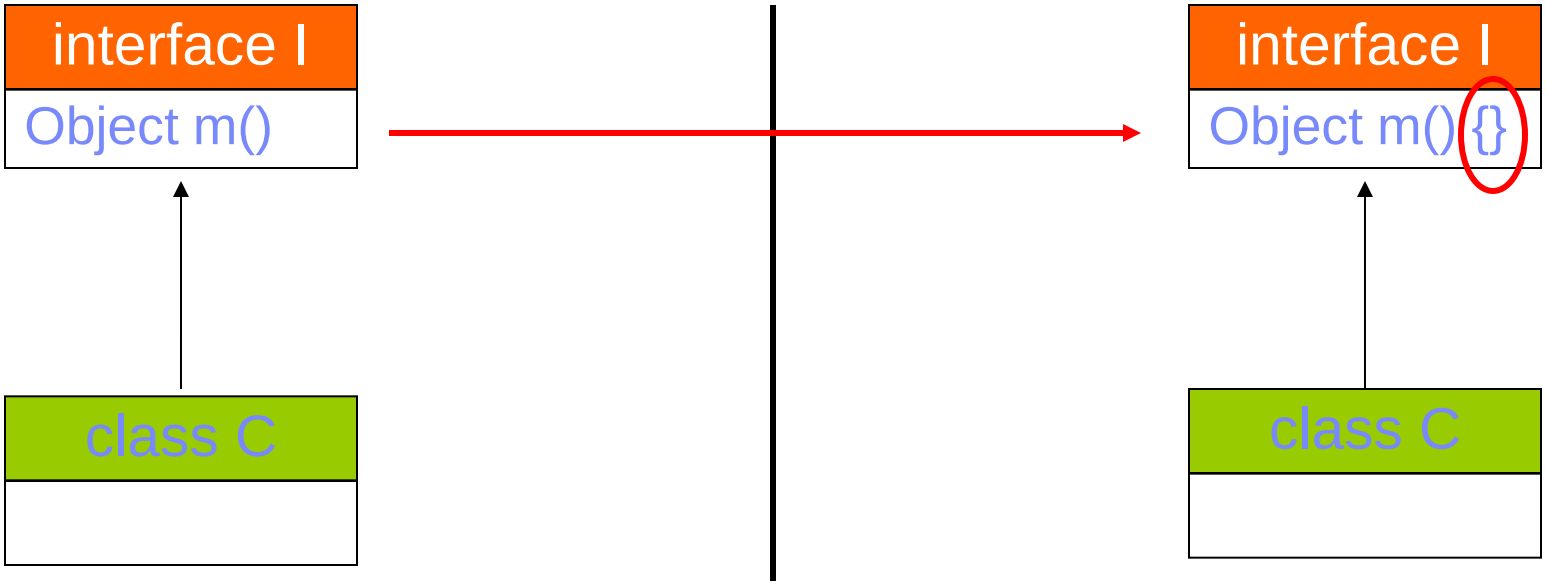
bootstrap

bridge m()Object to m()String

Default bootstrap method

Road 1: Add a default to an existing interface method

Java 8




```
I i = new C();
Object o = i.m();
```

invokeinterface m()Object

Road 1: “use invokedynamic” closed!

Java 8

CC image: http://www.flickr.com/photos/ell-r-brown/4379829973/sizes/m/in/photostream/

Road 2: Make it the VMs problem

Java 8

- The issues list looks like:
  - Bridge methods for covariant return & generic erasure
  - Binary compatibility prevents using invokedynamic
  - Separate compilation requires existing invoke bytecodes to work with default methods

- If the VM generates the bridge methods, then:
  - No bridge problems
  - No need for invokedynamic
  - Existing invoke bytecodes just work

Road 2: Make it the VMs problem

Java 8

- But….
  - The VM doesn't know anything about generics
  - The VM needs to guess about whether methods actually bridge
    - (Remember, it's just a "bit" to the VM)

- Questions:
  - Do VM generated bridge methods have bytecodes?
  - Can you instrument them with JVMTI?
  - Does reflection see them?
  - Do they appear in stacktraces?

- How do you specify when the VM needs to generate a bridge when there isn't a spec for bridge methods today?

Road 2: “Make the VM do it” closed!

Java 8

CC image: http://www.flickr.com/photos/loop_oh/2951460503/sizes/m/in/photostream/

Road 3: Make it Javac's problem

Java 8

Q) Where are existing bridges generated?

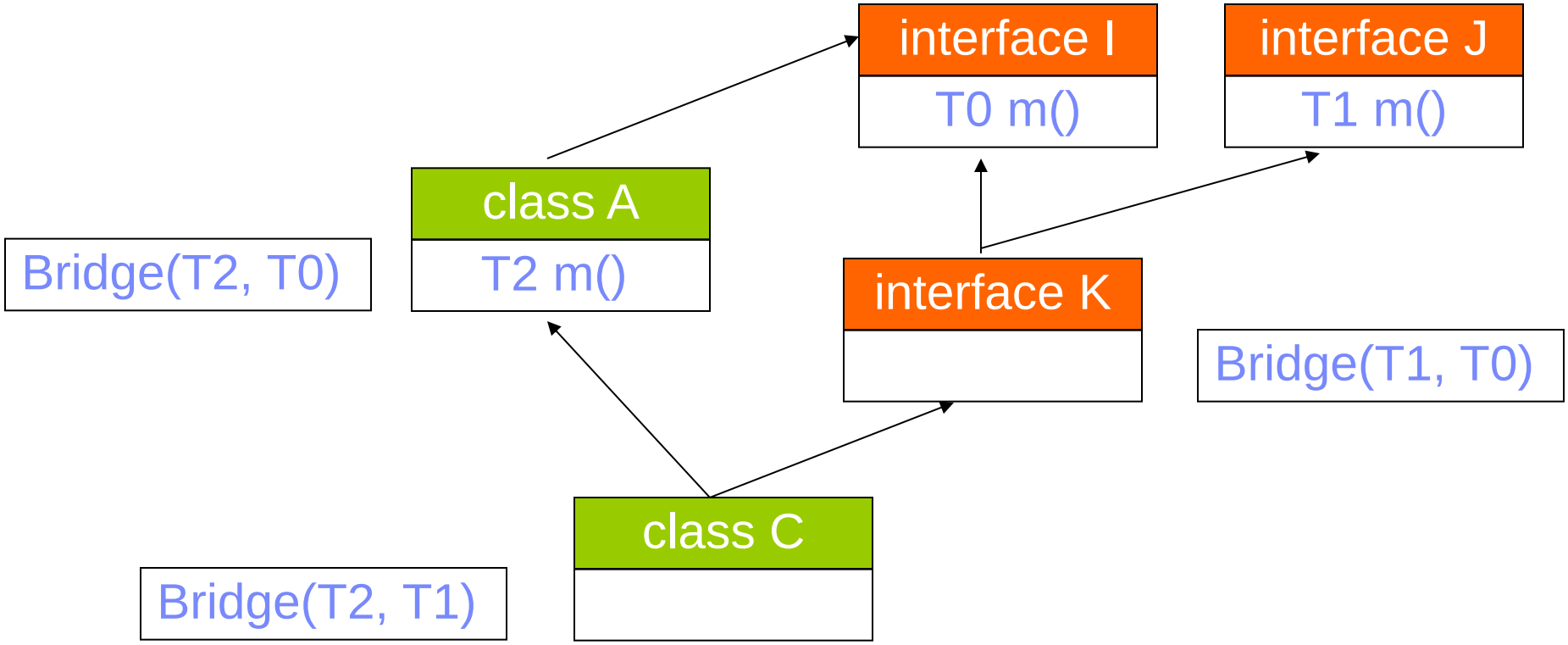
A) In classfiles by javac

This is exactly as brittle as bridges in classes. (Maintains the status quo)

Disappointing to make the existing problem worse

Javac generated bridges

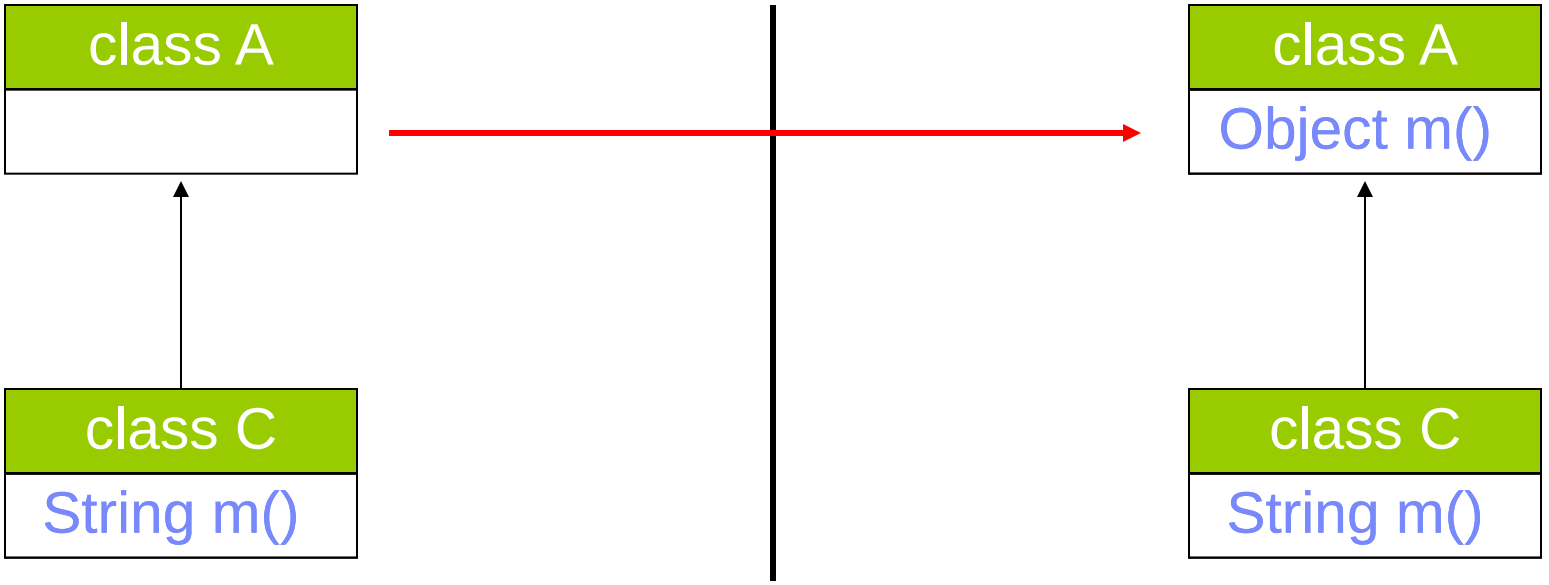
Java 8



- Javac generates bridge method at the highest possible point in the hierarchy
- Javac continues to generate bridges into classes where the bridged method resides

Examples of separate compilation

Java 8

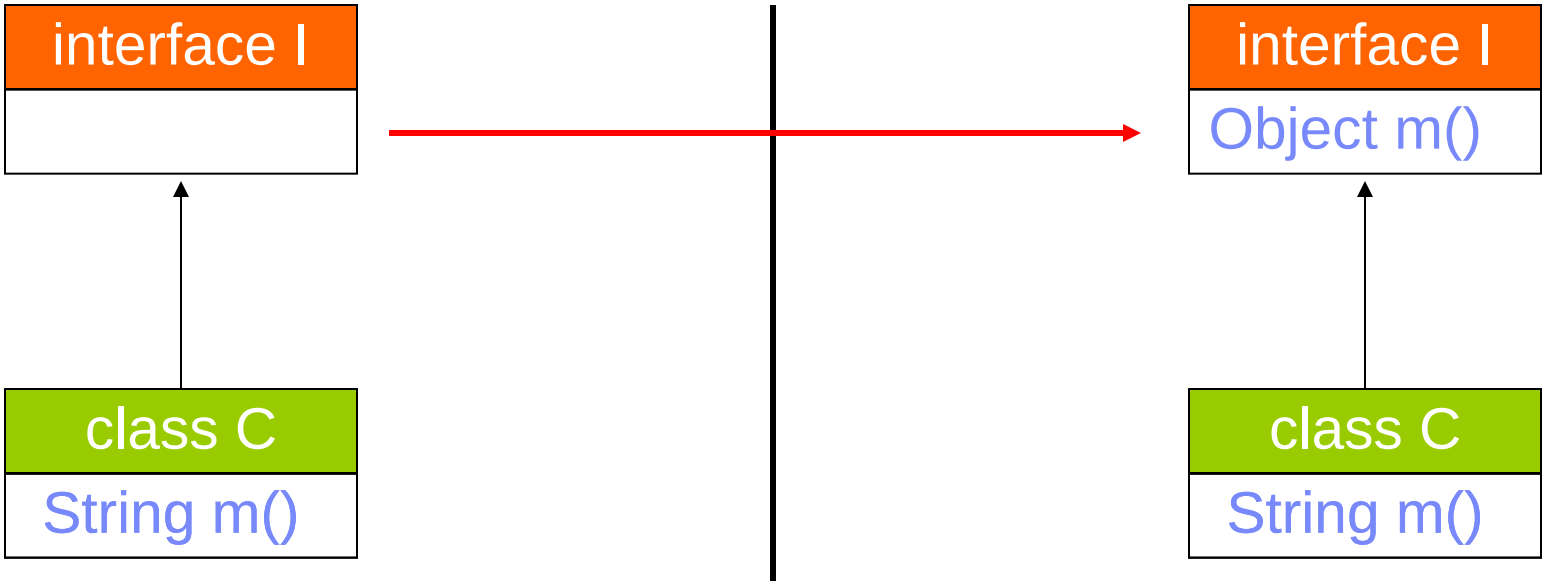


invokevirtual m()Object → A.m() — Separate Compilation

invokevirtual m()Object → C.m() — Consistent Compilation

Examples of separate compilation

Java 8

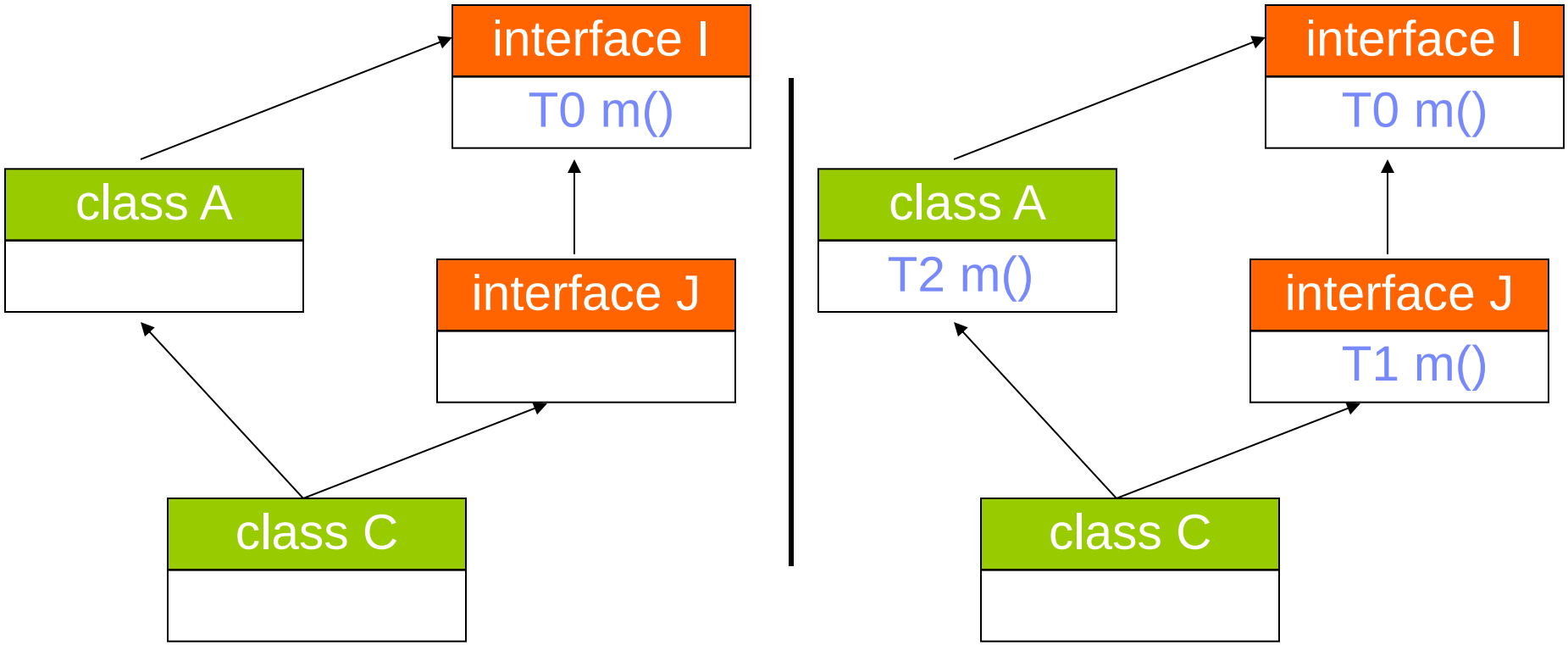


invokevirtual m()Object → I.m() Separate Compilation

invokevirtual m()Object → C.m() Consistent Compilation

Examples of separate compilation - 2

Java 8



invokevirtual C m()T0 → A.m()T2 (inherited bridge)
invokevirtual C m()T1 → J.m()T1 (inconsistent)
invokevirtual C m()T2 → A.m()T2

Future directions

- Features in Java are "forever“.
- But it may be possible to move all bridge methods into the VM in the future, or help the VM to do it better.
- The promising direction seems to be reifying information about the override relationship in the classfile, so the VM can conclude "Oh, these methods are the same" and ignore bridges if it has a better implementation.
- Or, erase the erasure.

Legal Notice

IBM and the IBM logo are trademarks or registered trademarks of IBM Corporation, in the United States, other countries or both.

Java and all Java-based marks, among others, are trademarks or registered trademarks of Oracle in the United States, other countries or both.

Other company, product and service names may be trademarks or service marks of others.

THE INFORMATION DISCUSSED IN THIS PRESENTATION IS PROVIDED FOR INFORMATIONAL PURPOSES ONLY. WHILE EFFORTS WERE MADE TO VERIFY THE COMPLETENESS AND ACCURACY OF THE INFORMATION, IT IS PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, AND IBM SHALL NOT BE RESPONSIBLE FOR ANY DAMAGES ARISING OUT OF THE USE OF, OR OTHERWISE RELATED TO, SUCH INFORMATION. ANY INFORMATION CONCERNING IBM'S PRODUCT PLANS OR STRATEGY IS SUBJECT TO CHANGE BY IBM WITHOUT NOTICE.