The case for Interface Injection

Tobias Ivarsson

tobias@thobe.org

work: Neo4j @ Neo Technology
web: http://thobe.org/
twitter: @thobe


"WHO ARE YOU? HOW DID YOU GET IN MY HOUSE?"

How long have you been a Festis, Bob?

- Got into Jython 2007 to scratch an itch
  - Python (2.5) had (something similar to) try-with-resource: the **`with`**-statement, Jython was behind and didn't...

    (ask me how the with-statement is better than try-with-resource)
- Did some Jython compiler work, got interested in JVM internals
- Was here in 2008 (a.k.a. the first summit). Hang-around at dinners...
- Hired by Neo Technology to work on the Neo4j graph database
- A database written in Java can make good use of most features of the JVM...
- ... which completes the circle, and brings me here!

Interface Injection

What is Interface Injection?

- The ability to inject an interface into a class that doesn't already implement that interface
  - *You don't implement this interface? - Now you do!*
- The interface must be injectable, and must support the target class

```
package org.foo;
interface Named {
    String getName();
}

package com.bar;
class Person { // doesn't implement Named
    String getName() { ... }
}
```

What is Interface Injection?

- For classes that already implement the methods, injection is easy
- For classes that don't, the interface can supply them!

```
package org.foo;
interface Named {
    String getName();
}

package com.bar;
class Person {
    String name() { ... }
    injected Named {
      String getName(){ return this.name(); }
    }
}
```

What interface injection is not

- The ability to inject *any* interface into *any* class
  - Only injectable interfaces
  - This means no cost for non-injectable interfaces!

Injection opportunities

- Interface cast: `injected = ((InjectableInterface)object);`
  (The **`checkcast`** *`[0xC0]`* instruction)
- instanceof: `object instanceof InjectableInterface`
  (The **`instanceof`** *`[0xC1]`* instruction)
- invokeinterface: `((InjectableInterface)object).someMethod();`
  (The **`invokeinterface`** *`[0xB9]`* instruction)
- Reflective cast: `InjectableInterface.class.cast(object);`
- Reflective instance check:
  `InjectableInterface.class.isInstance(object);`
- Reflective subtype check:

```
InjectableInterface.class.isAssignableFrom(
    TargetClass.class );
```

Interface inheritance and injectable interfaces

- Injectable interface that doesn't extend any other interface
  - Can be attempted for injection into any class

Interface inheritance and injectable interfaces

- Injectable interface that doesn't extend any other interface
  - Can be attempted for injection into any class
- Injectable interface that extends other injectable interface(s)
  - The parent interface(s) will need to be injected before injection of this interface is attempted

Interface inheritance and injectable interfaces

- Injectable interface that doesn't extend any other interface
  - Can be attempted for injection into any class
- Injectable interface that extends other injectable interface(s)
  - The parent interface(s) will need to be injected before injection of this interface is attempted
- Injectable interface that extends non-injectable interface(s)
  - Injection will only be possible into classes that implements the extended interfaces

Interface inheritance and injectable interfaces

- Injectable interface that doesn't extend any other interface
  - Can be attempted for injection into any class
- Injectable interface that extends other injectable interface(s)
  - The parent interface(s) will need to be injected before injection of this interface is attempted
- Injectable interface that extends non-injectable interface(s)
  - Injection will only be possible into classes that implements the extended interfaces
- Combine by least common denominator

Interface inheritance and injectable interfaces

- Normal interface extending an injectable interface
  - Use case:
    private injectable interface, public interface extending it
  - Use the injectable interface interface internally in your runtime
  - Let your users implement the normal public interface
- Casting to the normal interface will not trigger injection, even though it extends an injectable interface
- Shields your users from the surprising behavior of:

```
obj.getName() != ((Named)obj).getName()
```

Rules for overrides

- If the class implements the interface directly, normal rules apply (possibly with defender methods added to the mix).
- If the class implements a parent interface of the injectable interface, the methods defined by that interface are "final".
- If the class has methods that match the name+signature of methods in the interface, and the injector can access them, they are added as default implementations, but allowed to be overridden by the injector - even if they are final.
- If the interface is injected in the parent class, the injected methods are used as defaults, but may be overridden by this injector.
- Methods injected into parent take precedence over methods from the target.

Rules for overrides

- If the class implements the interface directly, normal rules apply (possibly with defender methods added to the mix).
- If the class implements a parent interface of the injectable interface, the methods defined by that interface are "final".
- If the class has methods that match the name+signature of methods in the interface, and the injector can access them, they are added as default implementations, but allowed to be overridden by the injector - even if they are final.
- If the interface is injected in the parent class, the injected methods are used as defaults, but may be overridden by this injector.
- Methods injected into parent take precedence over methods from the target.

Injecting an interface

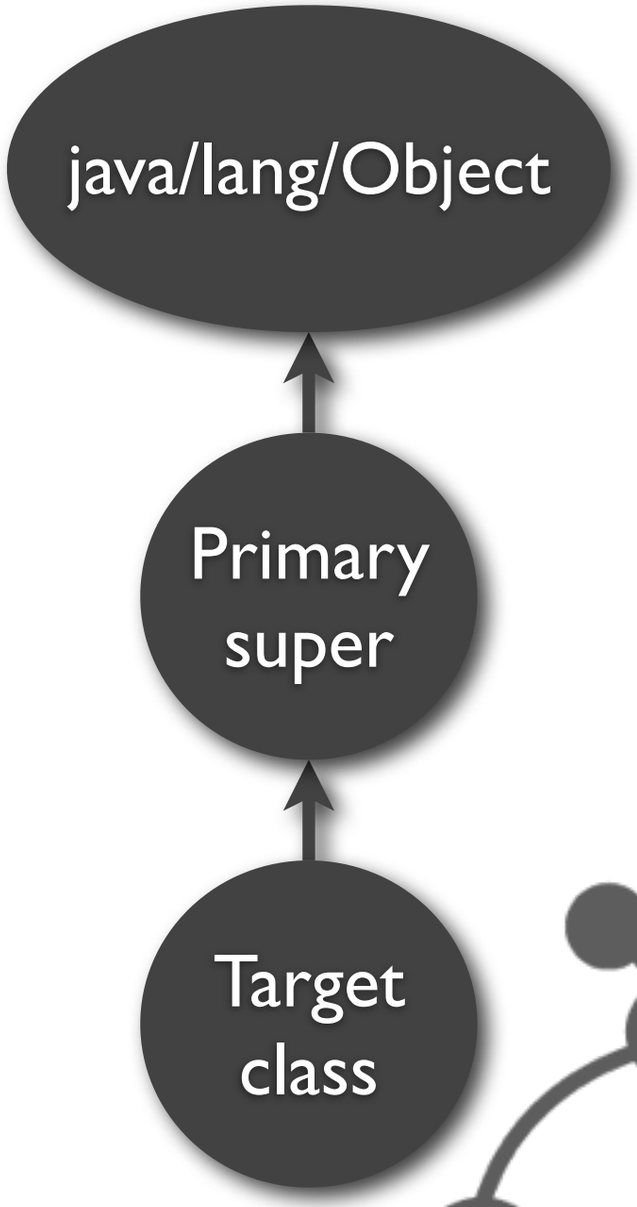
```
((MyInterface)obj).method()
```

Injecting an interface

3. Get injection record for primary super of
   of primary super (java/lang/Object)

2. Get injection record for primary super of target

((MyInterface)obj).method()
1. inject into obj.getClass() “Target class”
   Get injection record for Target class

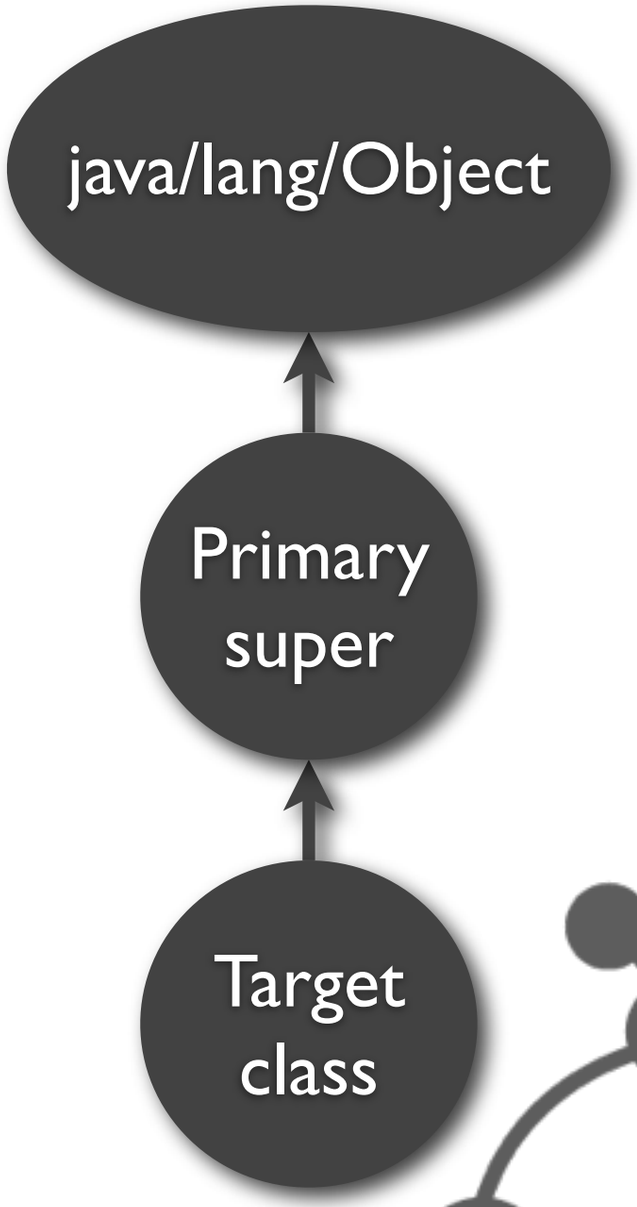


Injecting an interface

3. Get injection record for primary super of of primary super (java/lang/Object)
4. inject(MyInterface.class, Object.class, null)
5. Returns injection record (null)

2. Get injection record for primary super of target

((MyInterface)obj).method()
1. inject into obj.getClass() "Target class"
   Get injection record for Target class

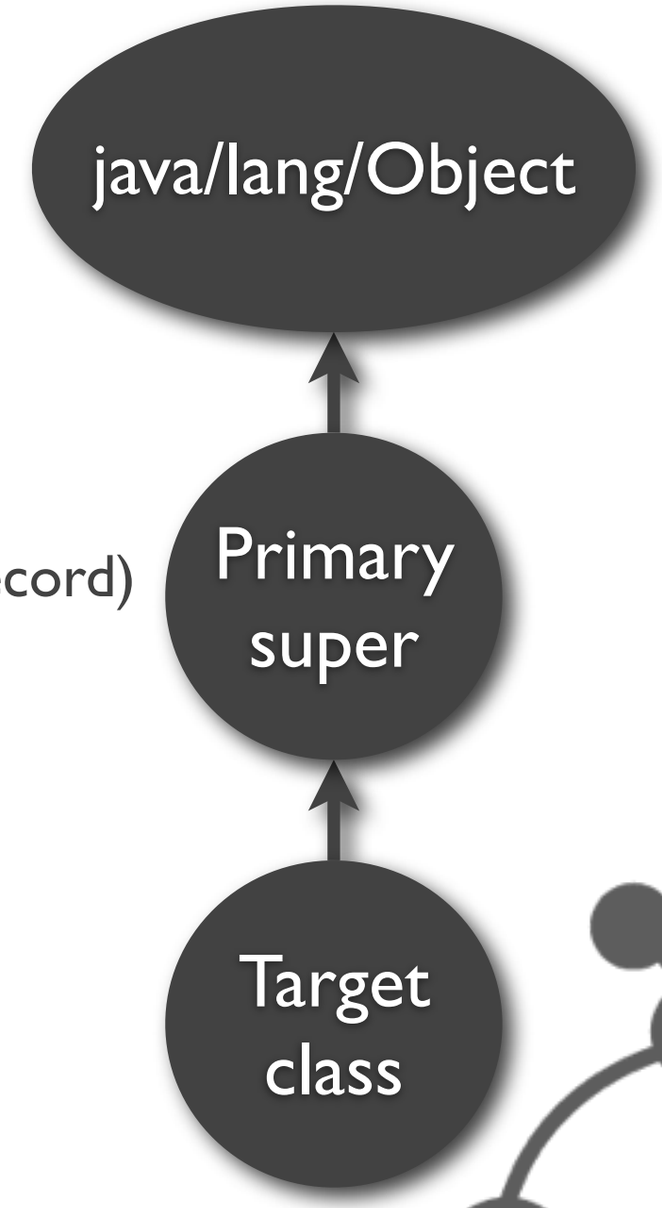


Injecting an interface

3. Get injection record for primary super of of primary super (java/lang/Object)
4. inject(MyInterface.class, Object.class, null)
5. Returns injection record (null)

2. Get injection record for primary super of target
6. inject(MyInterface.class, PrimarySuper.class, super_record)
7. Returns injection record

((MyInterface)obj).method()
1. inject into obj.getClass() "Target class"
   Get injection record for Target class

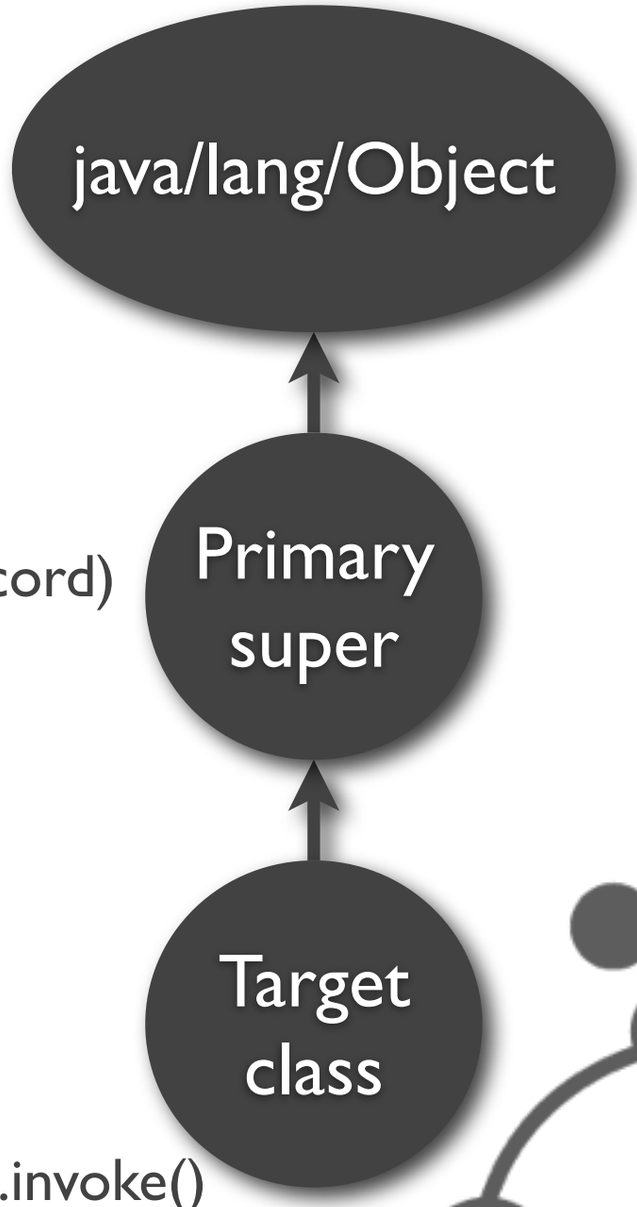


Injecting an interface

3. Get injection record for primary super of of primary super (java/lang/Object)
4. inject(MyInterface.class, Object.class, null)
5. Returns injection record (null)

2. Get injection record for primary super of target
6. inject(MyInterface.class, PrimarySuper.class, super_record)
7. Returns injection record

((MyInterface)obj).method()
1. inject into obj.getClass() "Target class"
   Get injection record for Target class
8. inject(MyInterface.class, Target.class, super_record)
9. (Returned injection record).getMethodHandle(offset).invoke()



The API

```
public abstract class InterfaceInjector {
    protected InjectionRecord inject(
      InjectionRecord.Builder injectionRecordBuilder );
}
public final class InjectionRecord.Builder {
    Class<?> getInterface()
    Class<?> getTragetClass()
    boolean isCompletelyDefined()
    Iterable<Method> undefinedMethods()
    Iterable<Method> overridableMethods()
    boolean isInjectable(Method ifaceMethod)
    boolean isDefined(Method ifaceMethod)
    void inject(Method ifaceMethod, MethodHandle handle)
    boolean injectIfUndefined(
      Method ifaceMethod, MethodHandle handle)
    boolean injectIfOverridable(
      Method ifaceMethod, MethodHandle handle)
    MethodHandle getInjectedMethod(Method ifaceMethod)
    InjectionRecord build()
}
```

The API

```
public abstract class InterfaceInjector {
    protected InjectionRecord inject(
      InjectionRecord.Builder injectionRecordBuilder );
}
public final class Injection
    Class<?> getInterface()
    Class<?> getTragetClass()
    boolean isCompletelyDefined()
    Iterable<Method> undefinedMethods()
    Iterable<Method> overridableMethods()
    boolean isInjectable(Method ifaceMethod)
    boolean isDefined(Method ifaceMethod)
    void inject(Method ifaceMethod, MethodHandle handle)
    boolean injectIfUndefined(
      Method ifaceMethod, MethodHandle handle)
    boolean injectIfOverridable(
      Method ifaceMethod, MethodHandle handle)
    MethodHandle getInjectedMethod(Method ifaceMethod)
    InjectionRecord build()
}
```

The interface we are building an injection for

The API

```
public abstract class InterfaceInjector {
    protected InjectionRecord inject(
      InjectionRecord.Builder injectionRecordBuilder );
}
public final class InjectionRecord.Builder {
    Class<?> getInterface()
    Class<?> getTragetClass()
    boolean isCompletelyDefined
    Iterable<Method> undefinedMethods()
    Iterable<Method> overridableMethods()
    boolean isInjectable(Method ifaceMethod)
    boolean isDefined(Method ifaceMethod)
    void inject(Method ifaceMethod, MethodHandle handle)
    boolean injectIfUndefined(
      Method ifaceMethod, MethodHandle handle)
    boolean injectIfOverridable(
      Method ifaceMethod, MethodHandle handle)
    MethodHandle getInjectedMethod(Method ifaceMethod)
    InjectionRecord build()
}
```

The class to inject the interface in

The API

```
public abstract class InterfaceInjector {
    protected InjectionRecord inject(
      InjectionRecord.Builder injectionRecordBuilder );
}
public final class InjectionRecord.Builder {
    Class<?> getInterface()
    Class<?> getTragetClass()
    boolean isCompletelyDefined()
    Iterable<Method> undefinedMetho
    Iterable<Method> overridableMeth ds()
    boolean isInjectable(Method ifaceMethod)
    boolean isDefined(Method ifaceMethod)
    void inject(Method ifaceMethod, MethodHandle handle)
    boolean injectIfUndefined(
      Method ifaceMethod, MethodHandle handle)
    boolean injectIfOverridable(
      Method ifaceMethod, MethodHandle handle)
    MethodHandle getInjectedMethod(Method ifaceMethod)
    InjectionRecord build()
}
```

Is the injection done?

The API

```
public abstract class InterfaceInjector {
    protected InjectionRecord inject(
      InjectionRecord.Builder injectionRecordBuilder );
}
public final class InjectionRecord.Builder {
    Class<?> getInterface()
    Class<?> getTragetClass()
    boolean isCompletelyDefined()
    Iterable<Method> undefinedMethods()
    Iterable<Method> overridableMethods()
    boolean isInjectable(Method ifaceMethod)
    boolean isDefined(Method ifaceMethod)
    void inject(Method ifaceMethod, MethodHandle handle)
    boolean injectIfUndefined(
      Method ifaceMethod, MethodHandle handle)
    boolean injectIfOverridable(
      Method ifaceMethod, MethodHandle handle)
    MethodHandle getInjectedMethod(Method ifaceMethod)
    InjectionRecord build()
}
```

Which methods *must* be defined

The API

```
public abstract class InterfaceInjector {
    protected InjectionRecord inject(
      InjectionRecord.Builder injectionRecordBuilder );
}
public final class InjectionRecord.Builder {
    Class<?> getInterface()
    Class<?> getTragetClass()
    boolean isCompletelyDefined()
    Iterable<Method> undefinedMethods()
    Iterable<Method> overridableMethods()
    boolean isInjectable(Method ifaceMethod
    boolean isDefined(Method ifaceMethod)
    void inject(Method ifaceMethod, MethodHandle handle)
    boolean injectIfUndefined(
      Method ifaceMethod, MethodHandle handle)
    boolean injectIfOverridable(
      Method ifaceMethod, MethodHandle handle)
    MethodHandle getInjectedMethod(Method ifaceMethod)
    InjectionRecord build()
}
```

Which methods *may* be defined

The API

```
public abstract class InterfaceInjector {
    protected InjectionRecord inject(
      InjectionRecord.Builder injectionRecordBuilder );
}
public final class InjectionRecord.Builder {
    Class<?> getInterface()
    Class<?> getTragetClass()
    boolean isCompletelyDefined()
    Iterable<Method> undefinedMethods()
    Iterable<Method> overridableMethods()
    boolean isInjectable(Method ifaceMethod)
    boolean isDefined(Method ifaceMethod)
    void inject(Method ifaceMethod, MethodHandle handle)
    boolean injectIfUndefined(
      Method ifaceMethod, MethodHandle handle)
    boolean injectIfOverridable(
      Method ifaceMethod, MethodHandle handle)
    MethodHandle getInjectedMethod(Method ifaceMethod)
    InjectionRecord build()
}
```

Can we inject an implementation of this method?

The API

```
public abstract class InterfaceInjector {
    protected InjectionRecord inject(
      InjectionRecord.Builder injectionRecordBuilder );
}
public final class InjectionRecord.Builder {
    Class<?> getInterface()
    Class<?> getTragetClass()
    boolean isCompletelyDefined()
    Iterable<Method> undefinedMethods()
    Iterable<Method> overridableMethods()
    boolean isInjectable(Method ifaceMetho
    boolean isDefined(Method ifaceMethod)
    void inject(Method ifaceMethod, MethodH
    boolean injectIfUndefined(
      Method ifaceMethod, MethodHandle handle)
    boolean injectIfOverridable(
      Method ifaceMethod, MethodHandle handle)
    MethodHandle getInjectedMethod(Method ifaceMethod)
    InjectionRecord build()
}
```

Is this method defined?

The API

```
public abstract class InterfaceInjector {
    protected InjectionRecord inject(
      InjectionRecord.Builder injectionRecordBuilder );
}
public final class InjectionRecord.Builder {
    Class<?> getInterface()
    Class<?> getTragetClass()
    boolean isCompletelyDefined()
    Iterable<Method> undefinedMethods()
    Iterable<Method> overridableMethods()
    boolean isInjectable(Method ifaceMethod)
    boolean isDefined(Method ifaceMethod)
    void inject(Method ifaceMethod, MethodHandle
    boolean injectIfUndefined(
      Method ifaceMethod, MethodHandle handle)
    boolean injectIfOverridable(
      Method ifaceMethod, MethodHandle handle)
    MethodHandle getInjectedMethod(Method ifaceMethod)
    InjectionRecord build()
}
```

Inject an implementation

The API

```
public abstract class InterfaceInjector {
    protected InjectionRecord inject(
      InjectionRecord.Builder injectionRecordBuilder );
}
public final class InjectionRecord.Builder {
    Class<?> getInterface()
    Class<?> getTragetClass()
    boolean isCompletelyDefined()
    Iterable<Method> undefinedMethods()
    Iterable<Method> overridableMethods()
    boolean isInjectable(Method ifaceMethod)
    boolean isDefined(Method ifaceMethod)
    void inject(Method ifaceMethod, MethodHandle handle)
    boolean injectIfUndefined(
      Method ifaceMethod, MethodHandle handle)
    boolean injectIfOverridable(
      Method ifaceMethod, MethodHandle handle)
    MethodHandle getInjectedM                    d)
    InjectionRecord build()
}
```

Build the injection record,
typically end with:
return builder.build();

How to implement an injectable interface

```
volatile interface Named {
    String getName();
    static {
        java.lang.invoke.InterfaceInjector.setInjector(
          new NamedInjector() );
    }
}

class NamedInjector extends java.lang.invoke.InterfaceInjector {
    protected java.lang.invoke.InterfaceImplementation inject(
      InjectionRecord.Builder builder ) {
        if (!builder.isCompletelyDefined()) {
            builder.inject(Named.class.getMethod("getName"),
              builder.getTargetClass().getMethod("name"));
        }
        return builder.build();
    }
    }
```

How to implement an injectable interface

```
volatile interface Named {
    String name();
    static
        invoke.InterfaceInjector.setInjector(
          new Injector() );
    }
}

class NamedInjector implements java.lang.invoke.InterfaceInjector {
    protected java.lang.invoke.InterfaceImplementation inject(
      InjectionRecord.Builder builder ) {
        if (!builder.isCompletelyDefined()) {
            builder.inject(Named.class.getMethod("getName"),
              builder.getTargetClass().getMethod("name"));
        }
        return builder.build();
    }
    }
```

bit not used by classes today, similar semantics to what we need.

How [illegible]ement an injectable interface

it doesn't have to be spelled 'volatile'

bit not used by classes today, similar semantics to what we need.

```
volatile [illegible]face Named {
    Stri[illegible]ame();
    stat[illegible]
        [illegible]nvoke.InterfaceInjector.setInjector(
            ne[illegible]ector() );
    }
}

class NamedInjecto[illegible]ds java.lang.invoke.InterfaceInjector {
    protected java.l[illegible].invoke.InterfaceImplementation inject(
      InjectionRecord.Builder builder ) {
        if (!builder.isCompletelyDefined()) {
            builder.inject(Named.class.getMethod("getName"),
              builder.getTargetClass().getMethod("name"));
        }
        return builder.build();
    }
    }
```

How to implement an injectable interface

```
volatile interface Named {
    String name();
    static
        invoke.InterfaceInjector.setInjector(
            new Injector() );
    }
}

class NamedInjector implements java.lang.invoke.InterfaceInjector {
    protected java.lang.invoke.InterfaceImplementation inject(
      InjectionRecord.Builder builder ) {
        if (!builder.isCompletelyDefined()) {
            builder.inject(Named.class.getMethod("getName"),
              builder.getTargetClass().getMethod("name"));
        }
        return builder.build();
    }
    }
```

just means that the classfile for the interface is eager loaded.

bit not used by classes today, similar semantics to what we need.

How to implement an injectable interface

```
volatile interface Named {
    String getName();
    static {
        java.lang.invoke.InterfaceInjector.setInjector(
          new NamedInjector() );
    }
}

class NamedInjector extends java.lang.invoke.Inter[illegible]or {
    protected java.lang.invoke.InterfaceImplementatio[illegible]
      InjectionRecord.Builder builder ) {
        if (!builder.isCompletelyDefined()) {
            builder.inject(Named.class.getMethod("getName"),
              builder.getTargetClass().getMethod("name"));
        }
        return builder.build();
    }
    }
```

setting the injector is what actually makes this an injectable interface

How to implement an injectable interface

```
volatile interface Named {
    String getName();
    static {
        java.lang.invoke.InterfaceInjector.setInjector(
          new NamedInjector() );
    }
}

class NamedInjector extends java.lang.invoke.InterfaceInjector {
    protected java.lang.invoke.InterfaceImplementation inject(
      InjectionRecord.Builder builder ) {
        if (!builder.isCompletelyDefined()) {
            builder.inject(Named.class.getMethod("getName"),
              builder.getTargetClass().getMethod("name"));
        }
        return builder.build();
    }
    }
```

Potentially surprising behavior ahead...

```
injectable interface Named {
    String getName()
        injected_as { return this.getClass().getName(); }
}
class Person {
    private String name;
    Person( String name ) { this.name = name; }
    public final String getName() { return name; }
}

Person person = new Person( "James" );
printf("Person.getName(): %s%n", person.getName());
// Person.getName(): James
printf("Named.getName(): %s%n", ((Named)person).getName());
// Named.getName(): com.bar.Person

// Java semantics would not expect different output!
```

How about...

- ...allowing overrides for methods defined in java.lang.Object, that haven't been redefined by the class? Let the injected implementation become default!
  - Potentially harmful!

How about...

- ...allowing overrides for methods defined in java.lang.Object, that haven't been redefined by the class? Let the injected implementation become default!
  - Potentially harmful!

```
injectable interface AddAHashCode {
    int hashCode() injected_as { return 4; }
}

Person person = new Person();
printf("0x%x%n", person.hashCode()); // 0x0099CC - or something
printf("0x%x%n", ((AddAHashCode)person).hashCode()); // 0x4
printf("0x%x%n", person.hashCode()); // which one?
```

How about...

- ...allowing overrides for methods defined in java.lang.Object, that haven't been redefined by the class? Let the injected implementation become default!
  - Potentially harmful!

Very useful for the related feature of defender methods...

```
injectable interface AddAHashCode {
    int hashCode() injected_as { return 4; }
}

Person person = new Person();
printf("0x%x%n", person.hashCode()); // 0x0099CC - or something
printf("0x%x%n", ((AddAHashCode)person).hashCode()); // 0x4
printf("0x%x%n", person.hashCode()); // which one?
```

Current status

- In the patch repo: horribly outdated
- In my repo: compiles, crashes on runtime
  - I have had it in better shape, but haven't had time to keep up
- We need more specification of the details!
  - ... and are at a point where we have enough understanding to have those discussions

What changed this weekend?

- Used to be that an injected interface ***had*** to use existing matching methods
  - But what if that method isn't accessible by the injector? Should it really be allowed to expose private methods?!
- 180° change: accessible methods from the class are only provided as convenient defaults, all methods may be specific to the interface
  - Means that you run into surprising behavior
  - Saves you from ever running into UnableToInject
    - Important for many of the use cases!
  - Thanks for all the great discussions that lead me to accept this as a good solution

Use cases for interface injection

Adapters

```
package java.util;
public injectable interface List<T> {
    ...
}

public class Arrays {
    ...
    public static <T> List<T> asList(T... array) {
        return (List) array;
    }
    ...
}
```

Traits

```
interface Identified { long id(); }

trait /*<-means injectable interface*/ NamedByTypeAndId
   extends Identified {
    String getName() default /*<-stolen from defender methods*/{
        return this.getClass().getSimpleName()+"["+this.id()+"]";
    }
}

obj = new Entity(14) with trait NamedByType
obj.getName() // Entity[14]
```

- ...unless you want traits to modify the vtable of the class

```
person.getName(); // James
((NamedByType)person).getName(); // Person[12]
person.getName(); // Person[12]
```

  Injection here would ***change the behavior*** of the class
  - This would have destructive consequences

Dynamic types

- Your dynamic language defines a master interface, containing the basic operations of your language
  - Including methods for looking up method handles for your invokedynamic call sites...
- This master interface would have an injector capable of injecting it into ***any*** class
- Your runtime can treat ***all*** objects as if they were objects from your language
- No need to wrap objects from other languages
  - No more PyString, GroovyString, et.c. only java.lang.String!

How does interface injection relate to other features?

Reflection

- `injectable.isInstance()` and `injectable.isAssignableFrom()` provide injection points
- However, `Class.getInterfaces()` will only return the *declared* implemented interfaces - not the injected ones, since having this change will not be helpful
- Add: `boolean Class.isInjectable()`
- Should these also be added?

```
Class<?>[] Class.getAcceptedInterfaceInjections()
Class<?>[] Class.getRejectedInterfaceInjections()
```

open for discussion, as is the naming of these methods

Invoke dynamic

- Both use method handles extensively
- Otherwise actually quite different
  - Invoke dynamic requires *call sites* to use invokedynamic
  - Interface injection uses regular invokeinterface, the logic is on the *other side*, injected into the target class
    - Can interact with static languages (JavaPL), and *existing code*.

Defender methods *(or is it "Extension methods" now?)*

- Similarities on the surface
- Use similar functionalities under the hood: modify target classes
- Also large differences:
  - Defender methods inject methods for declared implemented interfaces
  - Defender methods are injected when the target class is loaded
    Injectable interfaces, lazily when objects are cast
  - Defender methods for java.lang.Object.* methods is safe!
- Defender methods on injectable interfaces will be a good way to specify defaults, to simplify writing injectors
  - reduce to a default injector in most cases!

Meta Object Protocol

- IMHO a MOP would be best implemented through the use of interface injection
- Interface LinkageProvider to provide linkage information to other languages to link to your language
- The master interface of your language would be injectable, and use a default injector provided by the MOP library
- When injecting the master interface of a language, we use the Linkage from the class that we are injecting into for getting the MethodHandles that define the implementation

My raw outline for a MetaObjectProtocol

```
interface Linkage {
    // ... something ... see Atillas work
}
interface LinkageProvider {
    Linkage linker();
}

@InjectionLinker.Factory( MyLangLinker.class )
private injectable interface MyLangInternal {
    static { setInjector( LinkageInjector.INSTANCE ); }
    // ... the core methods of your language ...
}

public abstract class MyLangObject
   implements MyLangInternal, LinkageProvider {
    public final Linkage linker() {
        return MyLangLinker.INSTANCE;
    }
}
```

My raw outline for a MetaObjectProtocol

```
public final class LinkageInjector extends InterfaceInjector {
    protected final InjectionRecord inject(
      InjectionRecord.Builder builder ) {

        InjectionLinker.Factory factory =
          builder.getInterface().getAnnotation(
            InjectionLinker.Factory.class );

        if (factory == null) return null;

        InjectionLinker linker = factory.value().newInstance();

        if ( LinkageProvider.class.isAssignableFrom(
                 builder.getTargetClass()) ) {
            return linker.linkAs( builder );
        } else {
            return linker.linkAsPojo( builder );
        }
    }
}
```

My raw outline for a MetaObjectProtocol

```
public abstract class InjectionLinker {
    public @interface Factory {
        Class<? extends InjectionLinker> value();
    }

    protected abstract InjectionRecord injectAsPojo(
      InjectionRecord.Builder builder );

    final InjectionRecord injectWithLinkageProvider(
      InjectionRecord.Builder builder ) {
        // iterate through all the methods in the injectable
        // interface, and get LinkedMethod objects for each
        // return Method handles that use LinkedMethod for
        // the actual lookup for that method implementation
    }
}
public abstract class LinkedMethod {
    MethodHandle getImplementation( Linkage linker );
}
```

Interface injection should be limited to only after-the-fact addition of interfaces.

Invokedynamic replaces all use cases for interface injection!

This feature will not work, here is why: ...

If injectable interfaces changed by ... we could...

There is no way we can have interface like things that define different behavior!

Please prove me wrong!

At the breakout session after the break, at 11:20

All opinions are welcome!