Making Java more dynamic: runtime code generation for the JVM

The talk was way too high-level. I wish the speaker would have explained more technical details!

The talk was way too high-level. I wish the speaker would have explained more technical details!

This was way too low-level. I did not understand what this talk was about.

The talk was way too high-level. I wish the speaker would have explained more technical details!

This was way too low-level. I did not understand what this talk was about.

Perfect level. Great talks!

```
@interface Secured {
  String user();
}

class UserHolder {
  static String user = "ANONYMOUS";
}
```

```
@interface Secured {
  String user();
}

class UserHolder {
  static String user = "ANONYMOUS";
}
```

```
class Service {
  @Secured(user = "ADMIN")
  void deleteEverything() {
    // delete everything...
  }
}
```

```
interface Framework {
  <T> Class<? extends T> secure(Class<T> type);
}

@interface Secured {
  String user();
}

class UserHolder {
  static String user = "ANONYMOUS";
}
```

```
class Service {
  @Secured(user = "ADMIN")
  void deleteEverything() {
    // delete everything...
  }
}
```

```
interface Framework {
  <T> Class<? extends T> secure(Class<T> type);
}

@interface Secured {
  String user();
}

class UserHolder {
  static String user = "ANONYMOUS";
}
```

```
class Service {
  @Secured(user = "ADMIN")
  void deleteEverything() {
    // delete everything...
  }
}
```

```java
interface Framework {
  <T> Class<? extends T> secure(Class<T> type);
}

@interface Secured {
  String user();
}

class UserHolder {
  static String user = "ANONYMOUS";
}
```


```java
class Service {
  @Secured(user = "ADMIN")
  void deleteEverything() {
    // delete everything...
  }
}
```

```
interface Framework {
  <T> Class<? extends T> secure(Class<T> type);
}

@interface Secured {
  String user();
}

class UserHolder {
  static String user = "ANONYMOUS";
}
```

depends on

does not know about

```
class Service {
  @Secured(user = "ADMIN")
  void deleteEverything() {
    // delete everything...
  }
}
```

```
interface Framework {
  <T> Class<? extends T> secure(Class<T> type);
}

@interface Secured {
  String user();
}

class UserHolder {
  static String user = "ANONYMOUS";
}
```

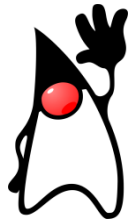
discovers at runtime

depends on

does not know about

```
class Service {
  @Secured(user = "ADMIN")
  void deleteEverything() {
    // delete everything...
  }
}
```

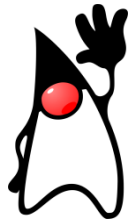
```
class Service {
  @Secured(user = "ADMIN")
  void deleteEverything() {
    if(!"ADMIN".equals(UserHolder.user)) {
      throw new IllegalStateException("Wrong user");
    }
    // delete everything...
  }
}
```



redefine class
(build time, agent)

```
class Service {
  @Secured(user = "ADMIN")
  void deleteEverything() {
    // delete everything...
  }
}
```

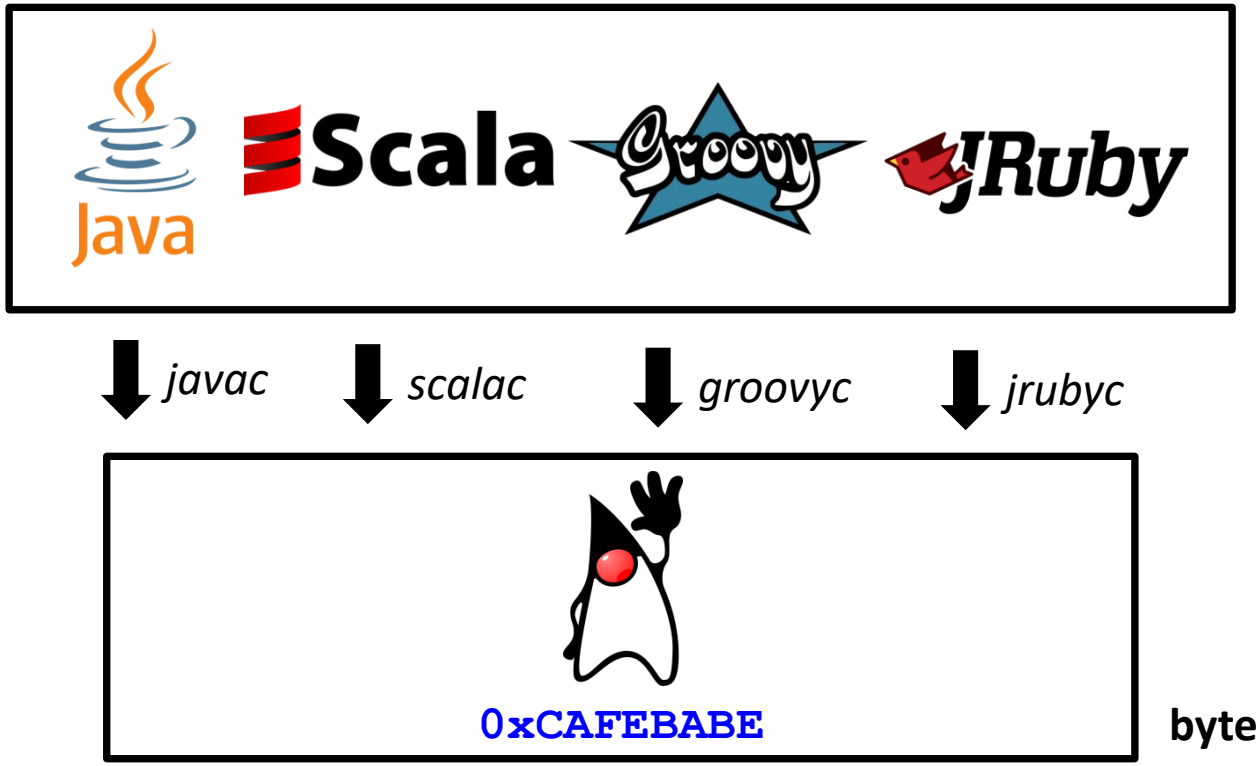
```
class SecuredService extends Service {
  @Override
  void deleteEverything() {
    if(!"ADMIN".equals(UserHolder.user)) {
      throw new IllegalStateException("Wrong user");
    }
    super.deleteEverything();
  }
}
```

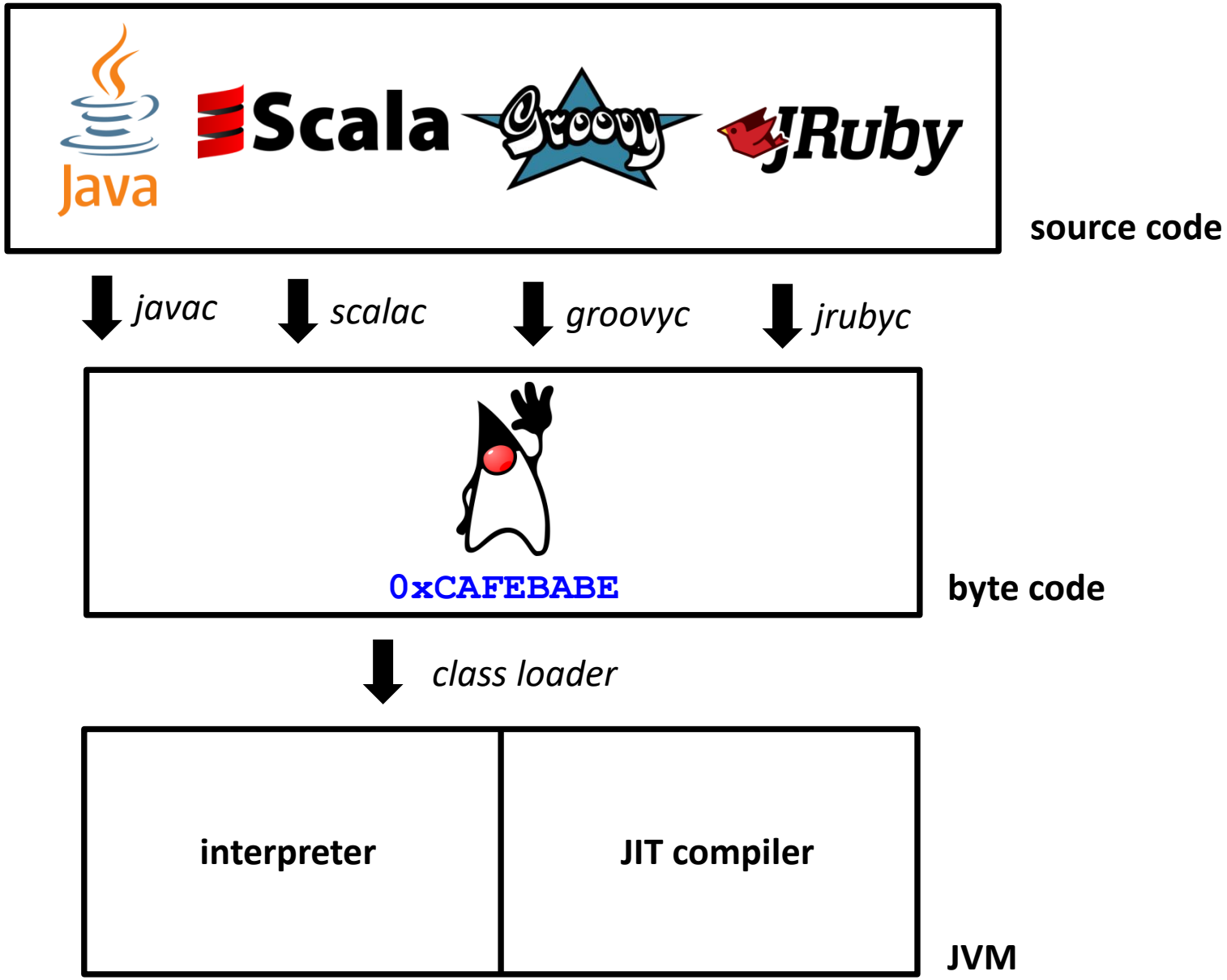


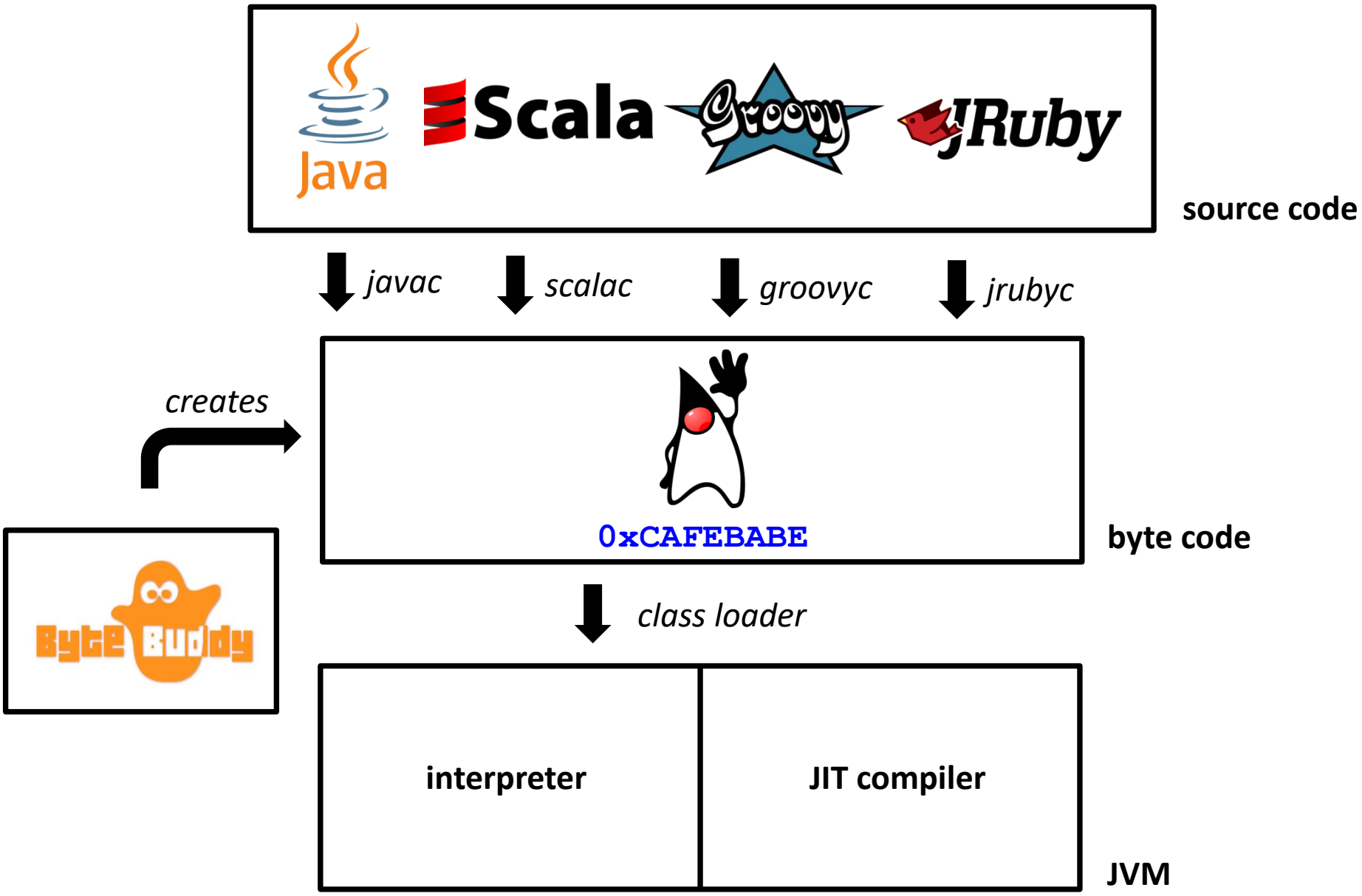
create subclass
(Liskov substitution)

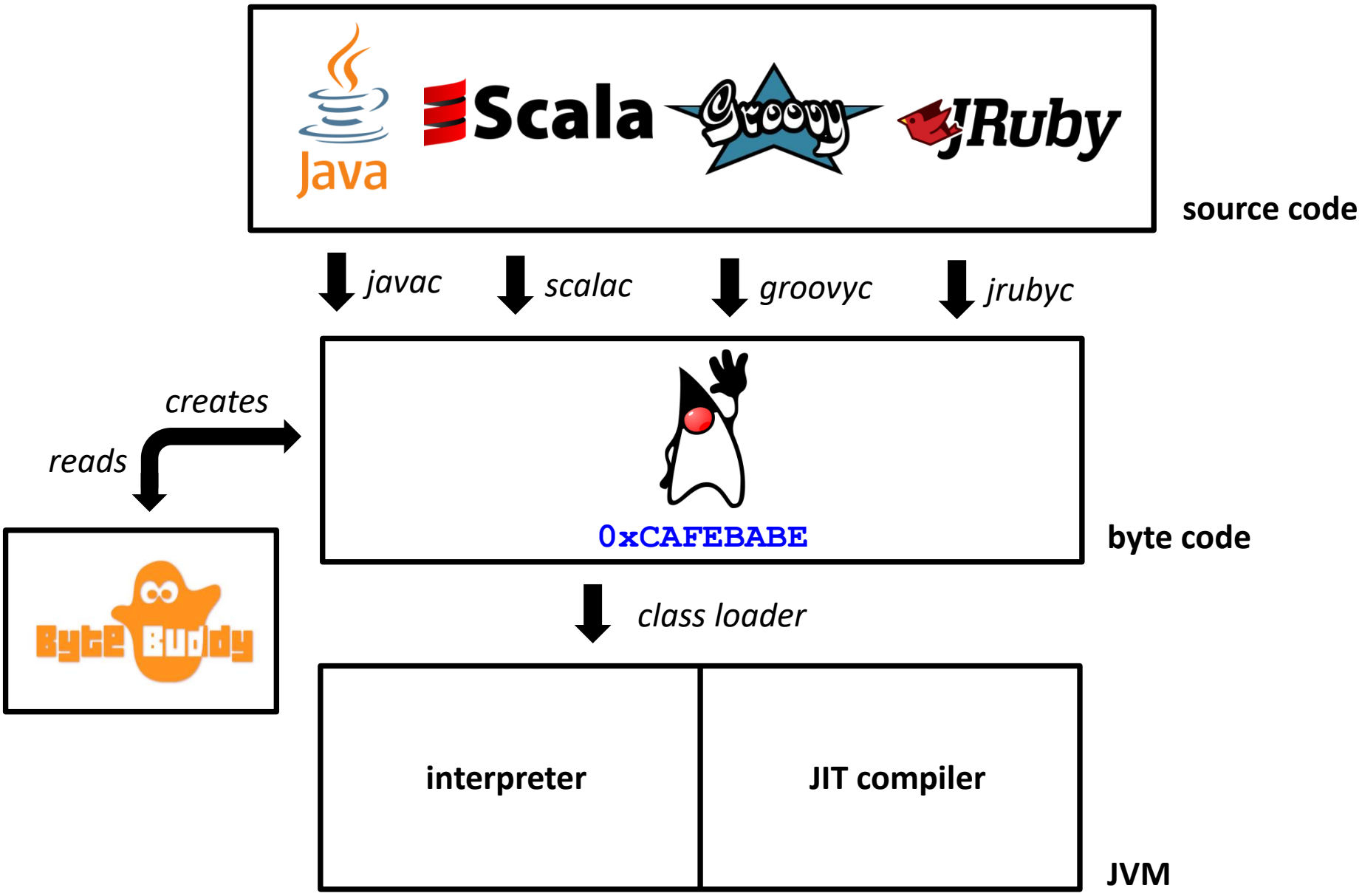
```
class Service {
  @Secured(user = "ADMIN")
  void deleteEverything() {
    // delete everything...
  }
}
```

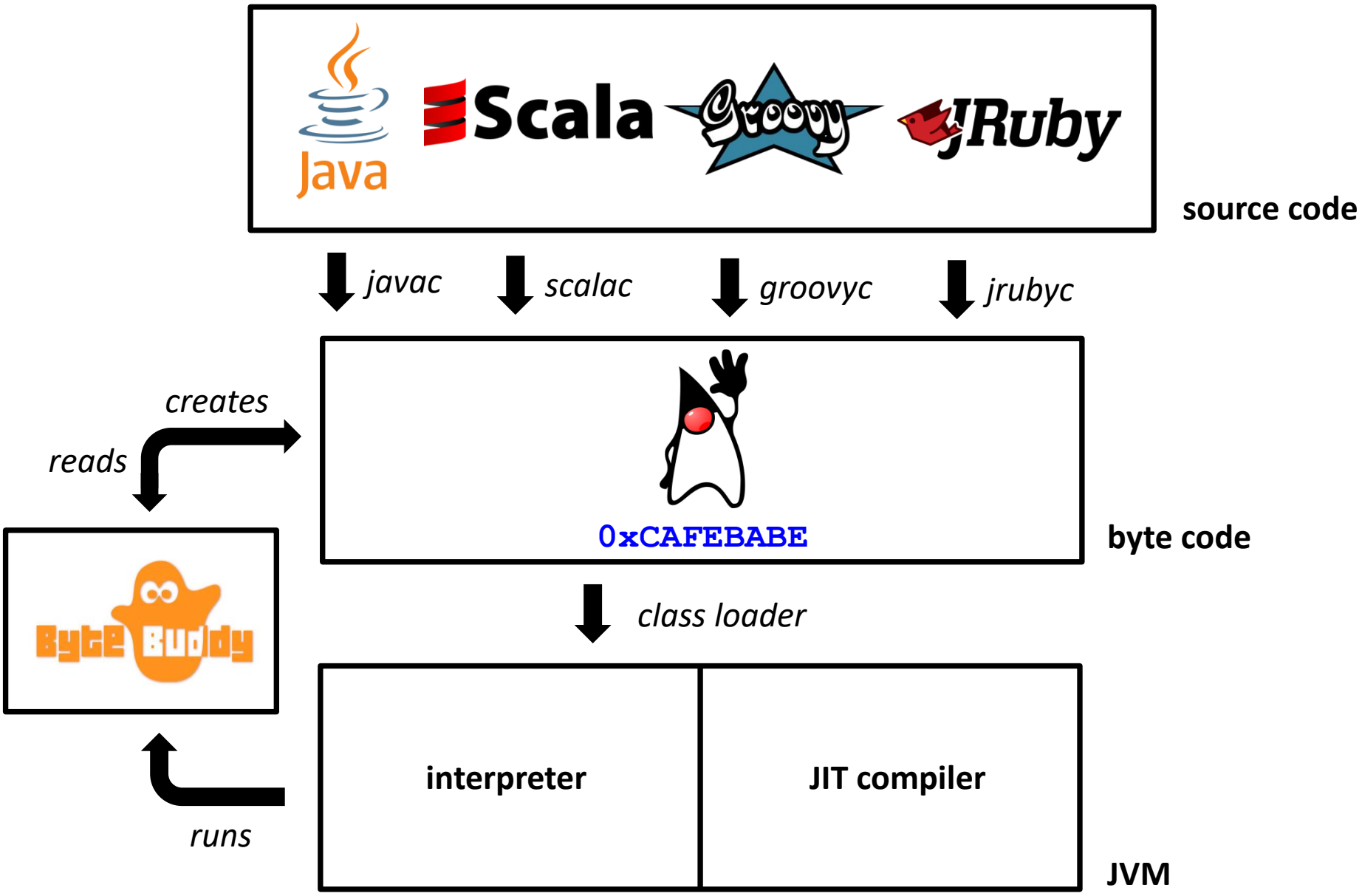
source code


Java
Scala
Groovy
JRuby
source code
javac
scalac
groovyc
jrubyc
0xCAFEBABE
byte code


Java
Scala
Groovy
JRuby
source code
javac
scalac
groovyc
jrubyc
0xCAFEBABE
byte code
class loader
interpreter
JIT compiler
JVM


Java
Scala
Groovy
JRuby
source code
javac
scalac
groovyc
jrubyc
creates
0xCAFEBABE
byte code
Byte Buddy
class loader
interpreter
JIT compiler
JVM


Java
Scala
Groovy
JRuby
source code
javac
scalac
groovyc
jrubyc
creates
reads
0xCAFEBABE
byte code
Byte Buddy
class loader
interpreter
JIT compiler
JVM


Java
Scala
Groovy
JRuby
source code
javac
scalac
groovyc
jrubyc
creates
reads
0xCAFEBABE
byte code
Byte Buddy
class loader
interpreter
JIT compiler
JVM
runs

Isn't reflection meant for this?

```
class Class {
  Method getDeclaredMethod(String name,
                           Class<?>... parameterTypes)
      throws NoSuchMethodException,
             SecurityException;
}

class Method {
  Object invoke(Object obj,
                Object... args)
      throws IllegalAccessException,
             IllegalArgumentException,
             InvocationTargetException;
}
```

Isn't reflection meant for this?

```
class Class {
  Method getDeclaredMethod(String name,
                           Class<?>... parameterTypes)
      throws NoSuchMethodException,
             SecurityException;
}

class Method {
  Object invoke(Object obj,
                Object... args)
      throws IllegalAccessException,
             IllegalArgumentException,
             InvocationTargetException;
}
```

Reflection implies neither type-safety nor a notion of fail-fast.

Note: there are no performance gains when using code generation over reflection!

Thus, runtime code generation only makes sense for *user type enhancement*: While the framework code is less type safe, this type-unsafety does not spoil the user's code.


spring


HIBERNATE


mockito


Guice


eclipse link


play


Clover


OpenEJB


APACHE WICKET


APACHE
SHIRO


OpenJDK


GRAILS

Java source code

```
int foo() {
  return 1 + 2;
}
```

Java byte code

Java source code

```
int foo() {
  return 1 + 2;
}
```

Java byte code

```
ICONST_1
ICONST_2
IADD
```

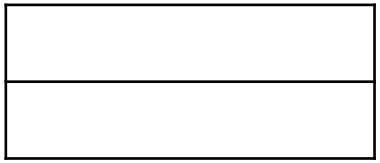
Java source code

```
int foo() {
  return 1 + 2;
}
```

Java byte code

```
ICONST_1
ICONST_2
IADD
```

operand stack



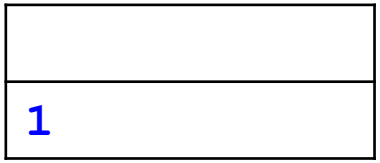
Java source code

```
int foo() {
  return 1 + 2;
}
```

Java byte code

```
ICONST_1
ICONST_2
IADD
```

operand stack



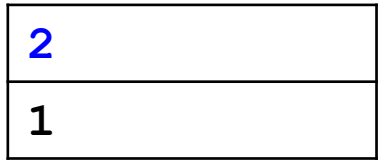
Java source code

```
int foo() {
  return 1 + 2;
}
```

Java byte code

```
ICONST_1
ICONST_2
IADD
```

operand stack



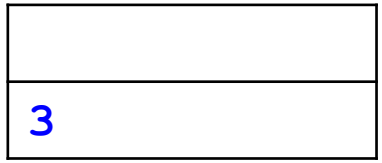
Java source code

```
int foo() {
  return 1 + 2;
}
```

Java byte code

```
ICONST_1
ICONST_2
IADD
```

operand stack



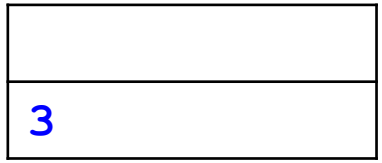
Java source code

```
int foo() {
  return 1 + 2;
}
```

Java byte code

```
ICONST_1
ICONST_2
IADD
IRETURN
```

operand stack



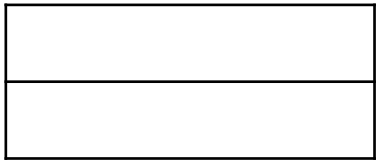
Java source code

```
int foo() {
  return 1 + 2;
}
```

Java byte code

```
ICONST_1
ICONST_2
IADD
IRETURN
```

operand stack



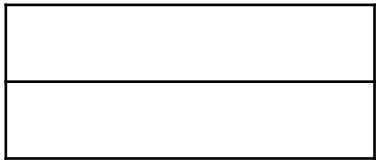
Java source code

```
int foo() {
  return 1 + 2;
}
```

Java byte code

```
ICONST_1    0x04
ICONST_2    0x05
IADD        0x60
IRETURN     0xAC
```

operand stack



```
MethodVisitor methodVisitor = ...
methodVisitor.visitInsn(Opcodes.ICONST_1);
methodVisitor.visitInsn(Opcodes.ICONST_2);
methodVisitor.visitInsn(Opcodes.IADD);
methodVisitor.visitInsn(Opcodes.IRETURN);
```

visitor API

```
MethodVisitor methodVisitor = ...
methodVisitor.visitInsn(Opcodes.ICONST_1);
methodVisitor.visitInsn(Opcodes.ICONST_2);
methodVisitor.visitInsn(Opcodes.IADD);
methodVisitor.visitInsn(Opcodes.IRETURN);
```

tree API

```
MethodNode methodNode = ...
InsnList insnList = methodNode.instructions;
insnList.add(new InsnNode(Opcodes.ICONST_1));
insnList.add(new InsnNode(Opcodes.ICONST_2));
insnList.add(new InsnNode(Opcodes.IADD));
insnList.add(new InsnNode(Opcodes.IRETURN));
```

```
int foo() {
  return 1 + 2;
}
```

```
"int foo() {" +
"  return 1 + 2;" +
"}"
```

```
"int foo() {" +
"  return 1 + 2;" +
"}"
```

- Strings are not typed (“SQL quandary”)

```
"int foo() {" +
"  return 1 + 2;" +
"}"
```

- Strings are not typed (“SQL quandary”)
- Specifically: Security problems!

```
"int foo() {" +
"  return 1 + 2;" +
"}"
```

- Strings are not typed (“SQL quandary”)
- Specifically: Security problems!
- Makes debugging difficult
  (unlinked source code, exception stack traces)

```
"int foo() {" +
"  return 1 + 2;" +
"}"
```

- Strings are not typed (“SQL quandary”)
- Specifically: Security problems!
- Makes debugging difficult
  (unlinked source code, exception stack traces)
- Bound to Java as a language

```
"int foo() {" +
"  return 1 + 2;" +
"}"
```

- Strings are not typed (“SQL quandary”)
- Specifically: Security problems!
- Makes debugging difficult
  (unlinked source code, exception stack traces)
- Bound to Java as a language
- The Javassist compiler lags behind *javac*

```
"int foo() {" +
"  return 1 + 2;" +
"}"
```

- Strings are not typed (“SQL quandary”)
- Specifically: Security problems!
- Makes debugging difficult
  (unlinked source code, exception stack traces)
- Bound to Java as a language
- The Javassist compiler lags behind *javac*
- Requires special Java source code instructions for realizing cross-cutting concerns

```
class SecuredService extends Service {
  @Override
  void deleteEverything() {
    if(!"ADMIN".equals(UserHolder.user)) {
      throw new IllegalStateException("Wrong user");
    }
    super.deleteEverything();
  }
}
```

```
class SecuredService extends Service {
  @Override
  void deleteEverything() {
    if(!"ADMIN".equals(UserHolder.user)) {
      throw new IllegalStateException("Wrong user");
    }
    super.deleteEverything();
  }
}
```

generic delegation

```
interface MethodInterceptor {
  Object intercept(Object object,
                   Method method,
                   Object[] arguments,
                   MethodProxy proxy)
    throws Throwable
}
```

```
class SecuredService extends Service {
  @Override
  void deleteEverything() {
    if(!"ADMIN".equals(UserHolder.user)) {
      throw new IllegalStateException("Wrong user");
    }
    super.deleteEverything();
  }
}
```

generic delegation

```
interface MethodInterceptor {
  Object intercept(Object object,
                   Method method,
                   Object[] arguments,
                   MethodProxy proxy)
    throws Throwable
}
```

```
class SecuredService extends Service {
  @Override
  void deleteEverything() {
    if(!"ADMIN".equals(UserHolder.user)) {
      throw new IllegalStateException("Wrong user");
    }
    super.deleteEverything();
  }
}
```

generic delegation

```
interface MethodInterceptor {
  Object intercept(Object object,
                   Method method,
                   Object[] arguments,
                   MethodProxy proxy)
    throws Throwable
}
```


```
class SecuredService extends Service {
  @Override
  void deleteEverything() {
    methodInterceptor.intercept(this,
      Service.class.getDeclaredMethod("deleteEverything"),
      new Object[0],
      new $MethodProxy());
  }

  class $MethodProxy implements MethodProxy {
    // inner class semantics, can call super
  }
}


interface MethodInterceptor {
  Object intercept(Object object,
                   Method method,
                   Object[] arguments,
                   MethodProxy proxy)
    throws Throwable
}
```

```
class SecuredService extends Service {
  @Override
  void deleteEverything() {
    methodInterceptor.intercept(this,
      Service.class.getDeclaredMethod("deleteEverything"),
      new Object[0],
      new $MethodProxy());
  }

  class $MethodProxy implements MethodProxy {
    // inner class semantics, can call super
  }
}


interface MethodInterceptor {
  Object intercept(Object object,
                   Method method,
                   Object[] arguments,
                   MethodProxy proxy)
    throws Throwable
}
```

generic delegation

```
class SecuredService extends Service {
  @Override
  void deleteEverything() {
    methodInterceptor.intercept(this,
      Service.class.getDeclaredMethod("deleteEverything"),
      new Object[0],
      new $MethodProxy());
  }

  class $MethodProxy implements MethodProxy {
    // inner class semantics, can call super
  }
}


interface MethodInterceptor {
  Object intercept(Object object,
                   Method method,
                   Object[] arguments,
                   MethodProxy proxy)
    throws Throwable
}
```

generic delegation

```
class SecuredService extends Service {
  @Override
  void deleteEverything() {
    methodInterceptor.intercept(this,
      Service.class.getDeclaredMethod("deleteEverything"),
      new Object[0],
      new $MethodProxy());
  }

  class $MethodProxy implements MethodProxy {
    // inner class semantics, can call super
  }
}


interface MethodInterceptor {
  Object intercept(Object object,
                   Method method,
                   Object[] arguments,
                   MethodProxy proxy)
    throws Throwable
}
```

generic delegation

```
Class<?> dynamicType = new ByteBuddy()
  .subclass(Object.class)
  .method(named("toString"))
  .intercept(value("Hello World!"))
  .make()
  .load(getClass().getClassLoader())
  .getLoaded();
```

```
assertThat(dynamicType.newInstance().toString(),
           is("Hello World!"));
```

```
Class<?> dynamicType = new ByteBuddy()
  .subclass(Object.class)
  .method(named("toString"))
  .intercept(value("Hello World!"))
  .make()
  .load(getClass().getClassLoader())
  .getLoaded();


assertThat(dynamicType.newInstance().toString(),
           is("Hello World!"));
```

```
Class<?> dynamicType = new ByteBuddy()
  .subclass(Object.class)
  .method(named("toString"))
  .intercept(value("Hello World!"))
  .make()
  .load(getClass().getClassLoader())
  .getLoaded();


assertThat(dynamicType.newInstance().toString(),
           is("Hello World!"));
```

```
Class<?> dynamicType = new ByteBuddy()
  .subclass(Object.class)
  .method(named("toString"))
  .intercept(value("Hello World!"))
  .make()
  .load(getClass().getClassLoader())
  .getLoaded();


assertThat(dynamicType.newInstance().toString(),
           is("Hello World!"));
```

```
Class<?> dynamicType = new ByteBuddy()
  .subclass(Object.class)
  .method(named("toString"))
  .intercept(value("Hello World!"))
  .make()
  .load(getClass().getClassLoader())
  .getLoaded();


assertThat(dynamicType.newInstance().toString(),
           is("Hello World!"));
```

```
Class<?> dynamicType = new ByteBuddy()
  .subclass(Object.class)
  .method(named("toString"))
  .intercept(value("Hello World!"))
  .make()
  .load(getClass().getClassLoader())
  .getLoaded();


assertThat(dynamicType.newInstance().toString(),
           is("Hello World!"));
```

```
Class<?> dynamicType = new ByteBuddy()
  .subclass(Object.class)
  .method(named("toString"))
  .intercept(value("Hello World!"))
  .make()
  .load(getClass().getClassLoader())
  .getLoaded();


assertThat(dynamicType.newInstance().toString(),
           is("Hello World!"));
```

```
Class<?> dynamicType = new ByteBuddy()
  .subclass(Object.class)
  .method(named("toString"))
  .intercept(value("Hello World!"))
  .make()
  .load(getClass().getClassLoader())
  .getLoaded();
```

```
assertThat(dynamicType.newInstance().toString(),
           is("Hello World!"));
```

```
Class<?> dynamicType = new ByteBuddy()
  .subclass(Object.class)
  .method(named("toString"))
  .intercept(to(MyInterceptor.class))
  .make()
  .load(getClass().getClassLoader())
  .getLoaded();
```

```
class MyInterceptor {
  static String intercept() {
    return "Hello World";
  }
}
```

```
Class<?> dynamicType = new ByteBuddy()
  .subclass(Object.class)
  .method(named("toString"))
  .intercept(to(MyInterceptor.class))
  .make()
  .load(getClass().getClassLoader())
  .getLoaded();
```

```
class MyInterceptor {
  static String intercept() {
    return "Hello World";
  }
}
```

```
Class<?> dynamicType = new ByteBuddy()
  .subclass(Object.class)
  .method(named("toString"))
  .intercept(to(MyInterceptor.class))
  .make()
  .load(getClass().getClassLoader())
  .getLoaded();
```

identifies best match

```
class MyInterceptor {
  static String intercept() {
    return "Hello World";
  }
}
```

```
Class<?> dynamicType = new ByteBuddy()
  .subclass(Object.class)
  .method(named("toString"))
  .intercept(to(MyInterceptor.class))
  .make()
  .load(getClass().getClassLoader())
  .getLoaded();
```

provides arguments

```
class MyInterceptor {
  static String intercept(@Origin Method m) {
    return "Hello World from " + m.getName();
  }
}
```

```
Class<?> dynamicType = new ByteBuddy()
  .subclass(Object.class)
  .method(named("toString"))
  .intercept(to(MyInterceptor.class))
  .make()
  .load(getClass().getClassLoader())
  .getLoaded();
```

provides arguments

```
class MyInterceptor {
  static String intercept(@Origin Method m) {
    return "Hello World from " + m.getName();
  }
}
```

```
Class<?> dynamicType = new ByteBuddy()
  .subclass(Object.class)
  .method(named("toString"))
  .intercept(to(MyInterceptor.class))
  .make()
  .load(getClass().getClassLoader())
  .getLoaded();
```

provides arguments

```
class MyInterceptor {
  static String intercept(@Origin Method m) {
    return "Hello World from " + m.getName();
  }
}
```

Annotations that are not visible to a class loader are ignored at runtime.

Thus, Byte Buddy's classes can be used without Byte Buddy on the class path.

`@Origin Method|Class<?>|String`	Provides caller information
`@SuperCall Runnable|Callable<?>`	Allows super method call
`@DefaultCall Runnable|Callable<?>`	Allows default method call
`@AllArguments T[]`	Provides boxed method arguments
`@Argument(index) T`	Provides argument at the given index
`@This T`	Provides caller instance
`@Super T`	Provides super method proxy

```
class Foo {
  String bar() { return "bar"; }
}

Foo foo = new Foo();

new ByteBuddy()
  .redefine(Foo.class)
  .method(named("bar"))
  .intercept(value("Hello World!"))
  .make()
  .load(Foo.class.getClassLoader(),
        ClassReloadingStrategy.installedAgent());

assertThat(foo.bar(), is("Hello World!"));
```

```
class Foo {
  String bar() { return "bar"; }
}

Foo foo = new Foo();

new ByteBuddy()
  .redefine(Foo.class)
  .method(named("bar"))
  .intercept(value("Hello World!"))
  .make()
  .load(Foo.class.getClassLoader(),
        ClassReloadingStrategy.installedAgent());

assertThat(foo.bar(), is("Hello World!"));
```

```
class Foo {
  String bar() { return "bar"; }
}

Foo foo = new Foo();

new ByteBuddy()
  .redefine(Foo.class)
  .method(named("bar"))
  .intercept(value("Hello World!"))
  .make()
  .load(Foo.class.getClassLoader(),
        ClassReloadingStrategy.installedAgent());

assertThat(foo.bar(), is("Hello World!"));
```

```
class Foo {
  String bar() { return "bar"; }
}

Foo foo = new Foo();

new ByteBuddy()
   .redefine(Foo.class)
   .method(named("bar"))
   .intercept(value("Hello World!"))
   .make()
   .load(Foo.class.getClassLoader(),
         ClassReloadingStrategy.installedAgent());

assertThat(foo.bar(), is("Hello World!"));
```

The instrumentation API does not allow introduction of new methods.

This might change with JEP-159: Enhanced Class Redefiniton.

Byte Buddy: Java agents

```
class Foo {
  String bar() { return "bar"; }
}

assertThat(new Foo().bar(), is("Hello World!"));
```

Byte Buddy: Java agents

```
class Foo {
  String bar() { return "bar"; }
}

assertThat(new Foo().bar(), is("Hello World!"));
```


```
public static void premain(String arguments,
    Instrumentation instrumentation) {
  new AgentBuilder.Default()
    .type(named("Foo"))
    .transform( (builder, type, classLoader) ->
      builder.method(named("bar"))
             .intercept(value("Hello World!"));
    )
    .installOn(instrumentation);
}
```

Byte Buddy: Java agents

```
class Foo {
  String bar() { return "bar"; }
}

assertThat(new Foo().bar(), is("Hello World!"));
```


```
public static void premain(String arguments,
    Instrumentation instrumentation) {
  new AgentBuilder.Default()
    .type(named("Foo"))
    .transform( (builder, type, classLoader) ->
      builder.method(named("bar"))
             .intercept(value("Hello World!"));
    )
    .installOn(instrumentation);
}
```

Byte Buddy: Java agents

```
class Foo {
  String bar() { return "bar"; }
}

assertThat(new Foo().bar(), is("Hello World!"));
```


```
public static void premain(String arguments,
    Instrumentation instrumentation) {
  new AgentBuilder.Default()
    .type(named("Foo"))
    .transform( (builder, type, classLoader) ->
      builder.method(named("bar"))
             .intercept(value("Hello World!"));
    )
    .installOn(instrumentation);
}
```

Byte Buddy: Java agents

```
class Foo {
  String bar() { return "bar"; }
}

assertThat(new Foo().bar(), is("Hello World!"));
```


```
public static void premain(String arguments,
    Instrumentation instrumentation) {
  new AgentBuilder.Default()
    .type(named("Foo"))
    .transform( (builder, type, classLoader) ->
      builder.method(named("bar"))
             .intercept(value("Hello World!"));
    )
    .installOn(instrumentation);
}
```

Byte Buddy: adding code to a method using advice

```
class TimerAdvice {
  @OnMethodEnter
  static long enter() {
    return System.nanoTime();
  }
  @OnMethodExit
  static void exit(@Enter long start, @Origin String method) {
    long duration = System.nanoTime() - start;
    System.out.println(method + " took " + duration);
  }
}
```

Byte Buddy: adding code to a method using advice

```
class TimerAdvice {
  @OnMethodEnter
  static long enter() {
    return System.nanoTime();
  }
  @OnMethodExit
  static void exit(@Enter long start, @Origin String method) {
    long duration = System.nanoTime() - start;
    System.out.println(method + " took " + duration);
  }
}
```

```
class Foo {
  String bar() {
    return "bar";
  }
}
```

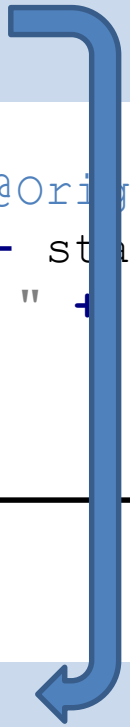
Byte Buddy: adding code to a method using advice

```
class TimerAdvice {
  @OnMethodEnter
  static long enter() {
    return System.nanoTime();
  }
  @OnMethodExit
  static void exit(@Enter long start, @Origin String method) {
    long duration = System.nanoTime() - start;
    System.out.println(method + " took " + duration);
  }
}
```

```
class Foo {
  String bar() {
    return "bar";
  }
}
```

Byte Buddy: adding code to a method using advice

```
class TimerAdvice {
  @OnMethodEnter
  static long enter() {
    return System.nanoTime();
  }
  @OnMethodExit
  static void exit(@Enter long start, @Origin String method) {
    long duration = System.nanoTime() - start;
    System.out.println(method + " took " + duration);
  }
}
```



```
class Foo {
  String bar() {
    long $start = System.nanoTime();
    return "bar";
  }
}
```

Byte Buddy: adding code to a method using advice

```
class TimerAdvice {
  @OnMethodEnter
  static long enter() {
    return System.nanoTime();
  }
  @OnMethodExit
  static void exit(@Enter long start, @Origin String method) {
    long duration = System.nanoTime() - start;
    System.out.println(method + " took " + duration);
  }
}
```

```
class Foo {
  String bar() {
    long $start = System.nanoTime();
    return "bar";
  }
}
```

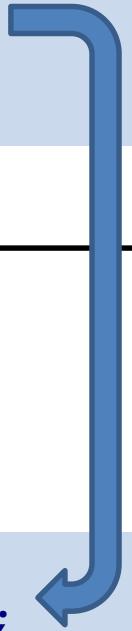
Byte Buddy: adding code to a method using advice

```
class TimerAdvice {
  @OnMethodEnter
  static long enter() {
    return System.nanoTime();
  }
  @OnMethodExit
  static void exit(@Enter long start, @Origin String method) {
    long duration = System.nanoTime() - start;
    System.out.println(method + " took " + duration);
  }
}
```

```
class Foo {
  String bar() {
    long $start = System.nanoTime();
    String $result = "bar";
    return $result;
  }
}
```

Byte Buddy: adding code to a method using advice

```
class TimerAdvice {
  @OnMethodEnter
  static long enter() {
    return System.nanoTime();
  }
  @OnMethodExit
  static void exit(@Enter long start, @Origin String method) {
    long duration = System.nanoTime() - start;
    System.out.println(method + " took " + duration);
  }
}
```



```
class Foo {
  String bar() {
    long $start = System.nanoTime();
    String $result = "bar";
    long $duration = System.nanoTime() - $start;
    System.out.println("Foo.bar()" + " took " + $duration);
    return $result;
  }
}
```

Byte Buddy: Java runtime agents

```
class Foo {
  String bar() { return "bar"; }
}

System.out.println(new Foo().bar());
```


```
public static void agentmain(String arguments,
    Instrumentation instrumentation) {
  new AgentBuilder.Default()
    .with(RedefinitionStrategy.RETRANSFORMATION)
    .disableClassFormatChanges()
    .type(named("Foo"))
    .transform( (builder, type, classLoader) ->
      builder.visit(Advice.to(TimerAdvice.class)
                          .on(named("bar"))));
    )
    .installOn(instrumentation);
}
```

Byte Buddy: Java runtime agents

```
class Foo {
  String bar() { return "bar"; }
}

System.out.println(new Foo().bar());
```


```
public static void agentmain(String arguments,
    Instrumentation instrumentation) {
  new AgentBuilder.Default()
    .with(RedefinitionStrategy.RETRANSFORMATION)
    .disableClassFormatChanges()
    .type(named("Foo"))
    .transform( (builder, type, classLoader) ->
      builder.visit(Advice.to(TimerAdvice.class)
                          .on(named("bar"))));
    )
    .installOn(instrumentation);
}
```

Byte Buddy: Java runtime agents

```
class Foo {
  String bar() { return "bar"; }
}

System.out.println(new Foo().bar());
// bar
// Foo.bar() took 1630
```


```
public static void agentmain(String arguments,
    Instrumentation instrumentation) {
  new AgentBuilder.Default()
    .with(RedefinitionStrategy.RETRANSFORMATION)
    .disableClassFormatChanges()
    .type(named("Foo"))
    .transform( (builder, type, classLoader) ->
      builder.visit(Advice.to(TimerAdvice.class)
                          .on(named("bar"))));
    )
    .installOn(instrumentation);
}
```

Byte Buddy: Java runtime agents

```
class Foo {
  String bar() { return "bar"; }
}

System.out.println(new Foo().bar());
// bar
// Foo.bar() took 1630
```


```
public static void agentmain(String arguments,
    Instrumentation instrumentation) {
  new AgentBuilder.Default()
    .with(RedefinitionStrategy.RETRANSFORMATION)
    .disableClassFormatChanges()
    .type(named("Foo"))
    .transform( (builder, type, classLoader) ->
      builder.visit(Advice.to(TimerAdvice.class)
                          .on(named("bar"))));
    )
    .installOn(instrumentation);
}
```

Byte Buddy: Java runtime agents

```
class Foo {
  String bar() { return "bar"; }
}

System.out.println(new Foo().bar());
// bar
// Foo.bar() took 1630
```


```
public static void agentmain(String arguments,
    Instrumentation instrumentation) {
  new AgentBuilder.Default()
    .with(RedefinitionStrategy.RETRANSFORMATION)
    .disableClassFormatChanges()
    .type(named("Foo"))
    .transform( (builder, type, classLoader) ->
      builder.visit(Advice.to(TimerAdvice.class)
                          .on(named("bar"))));
    )
    .installOn(instrumentation);
}
```

Build-time instrumentation: API

```
public class SimplePlugin implements Plugin {

  @Override
  public boolean matches(TypeDescription target) {
    return target.getName().equals("Foo");
  }

  @Override
  public DynamicType.Builder<?> apply(
      DynamicType.Builder<?> builder,
      TypeDescription typeDescription) {
  return builder.method(named("bar"))
                .intercept(value("Hello World!"));
  }
}
```

Build-time instrumentation: API

```
public class SimplePlugin implements Plugin {

  @Override
  public boolean matches(TypeDescription target) {
    return target.getName().equals("Foo");
  }

  @Override
  public DynamicType.Builder<?> apply(
      DynamicType.Builder<?> builder,
      TypeDescription typeDescription) {
  return builder.method(named("bar"))
                .intercept(value("Hello World!"));
  }
}
```

Build-time instrumentation: API

```
public class SimplePlugin implements Plugin {

  @Override
  public boolean matches(TypeDescription target) {
    return target.getName().equals("Foo");
  }

  @Override
  public DynamicType.Builder<?> apply(
      DynamicType.Builder<?> builder,
      TypeDescription typeDescription) {
  return builder.method(named("bar"))
                .intercept(value("Hello World!"));
  }
}
```

Build-time instrumentation: API

```
public class SimplePlugin implements Plugin {

  @Override
  public boolean matches(TypeDescription target) {
    return target.getName().equals("Foo");
  }

  @Override
  public DynamicType.Builder<?> apply(
      DynamicType.Builder<?> builder,
      TypeDescription typeDescription) {
  return builder.method(named("bar"))
                .intercept(value("Hello World!"));
  }
}
```

Build-time instrumentation: API

```
public class SimplePlugin implements Plugin {

  @Override
  public boolean matches(TypeDescription target) {
    return target.getName().equals("Foo");
  }

  @Override
  public DynamicType.Builder<?> apply(
      DynamicType.Builder<?> builder,
      TypeDescription typeDescription) {
  return builder.method(named("bar"))
                .intercept(value("Hello World!"));
  }
}
```

Build plugins can be applied as Java agent.

Adapters allow build to agent transformation. (Agent APIs are more specific.)

Build-time instrumentation with Maven.

```
<plugin>
  <groupId>net.bytebuddy</groupId>
  <artifactId>byte-buddy-maven-plugin</artifactId>
  <executions>
    <execution>
      <goals>
        <goal>transform</goal>
      </goals>
    </execution>
  </executions>
  <configuration>
    <transformations>
      <transformation>
        <plugin>pkg.SimplePlugin</plugin>
      </transformation>
    </transformations>
  </configuration>
</plugin>
```

Build-time instrumentation with Maven.

```
<plugin>
  <groupId>net.bytebuddy</groupId>
  <artifactId>byte-buddy-maven-plugin</artifactId>
  <executions>
    <execution>
      <goals>
        <goal>transform</goal>
      </goals>
    </execution>
  </executions>
  <configuration>
    <transformations>
      <transformation>
        <plugin>pkg.SimplePlugin</plugin>
      </transformation>
    </transformations>
  </configuration>
</plugin>
```

Build-time instrumentation with Maven.

```
<plugin>
  <groupId>net.bytebuddy</groupId>
  <artifactId>byte-buddy-maven-plugin</artifactId>
  <executions>
    <execution>
      <goals>
        <goal>transform</goal>
      </goals>
    </execution>
  </executions>
  <configuration>
    <transformations>
      <transformation>
        <plugin>pkg.SimplePlugin</plugin>
      </transformation>
    </transformations>
  </configuration>
</plugin>
```

Build-time instrumentation with Maven.

```
<plugin>
  <groupId>net.bytebuddy</groupId>
  <artifactId>byte-buddy-maven-plugin</artifactId>
  <executions>
    <execution>
      <goals>
        <goal>transform</goal>
      </goals>
    </execution>
  </executions>
  <configuration>
    <transformations>
      <transformation>
        <plugin>pkg.SimplePlugin</plugin>
        <artifactId>foo</artifactId>
      </transformation>
    </transformations>
  </configuration>
</plugin>
```

Build-time instrumentation with Gradle.

```
buildscript {
  repositories { jCenter() }
  dependencies {
    classpath "net.bytebuddy:byte-buddy-gradle-plugin:+"
  }
}
apply plugin: "net.bytebuddy.byte-buddy"

configurations {
  simplePlugin
}
dependencies {
  simplePlugin "bar:foo:1.0"
}

byteBuddy {
  transformation {
    plugin = "pkg.SimplePlugin"
    classPath = configurations.simplePlugin
  }
}
```

Build-time instrumentation with Gradle.

```
buildscript {
  repositories { jCenter() }
  dependencies {
    classpath "net.bytebuddy:byte-buddy-gradle-plugin:+"
  }
}
apply plugin: "net.bytebuddy.byte-buddy"

configurations {
  simplePlugin
}
dependencies {
  simplePlugin "bar:foo:1.0"
}

byteBuddy {
  transformation {
    plugin = "pkg.SimplePlugin"
    classPath = configurations.simplePlugin
  }
}
```

Build-time instrumentation with Gradle.

```
buildscript {
  repositories { jCenter() }
  dependencies {
    classpath "net.bytebuddy:byte-buddy-gradle-plugin:+"
  }
}
apply plugin: "net.bytebuddy.byte-buddy"

configurations {
  simplePlugin
}
dependencies {
  simplePlugin "bar:foo:1.0"
}

byteBuddy {
  transformation {
    plugin = "pkg.SimplePlugin"
    classPath = configurations.simplePlugin
  }
}
```

Build-time instrumentation with Gradle.

```
buildscript {
  repositories { jCenter() }
  dependencies {
    classpath "net.bytebuddy:byte-buddy-gradle-plugin:+"
  }
}
apply plugin: "net.bytebuddy.byte-buddy"

configurations {
  simplePlugin
}
dependencies {
  simplePlugin "bar:foo:1.0"
}

byteBuddy {
  transformation {
    plugin = "pkg.SimplePlugin"
    classPath = configurations.simplePlugin
  }
}
```

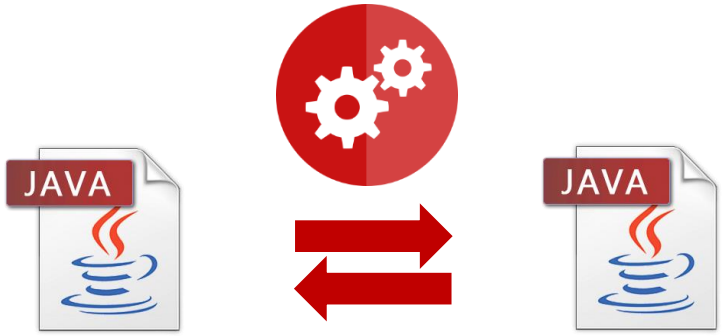
Reality check: Reinvent Java?

Many applications are built around a central infrastructure. A lot of code does not solve domain problems but bridges between domain and infrastructure.

Reality check: Reinvent Java?

Many applications are built around a central infrastructure. A lot of code does not solve domain problems but bridges between domain and infrastructure.

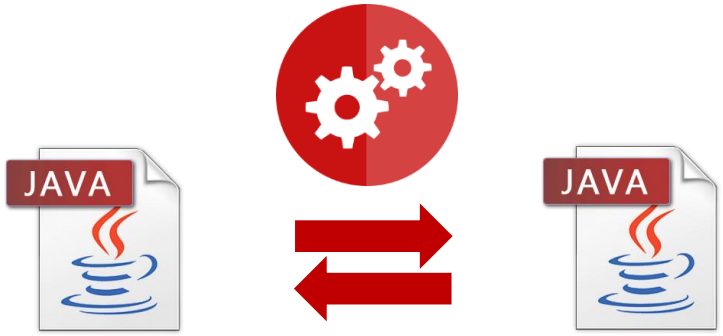
Java agents allow to add a decentralized infrastructure at runtime. In the source code, the infrastructure is only declared.



Reality check: Reinvent Java?

Many applications are built around a central infrastructure. A lot of code does not solve domain problems but bridges between domain and infrastructure.

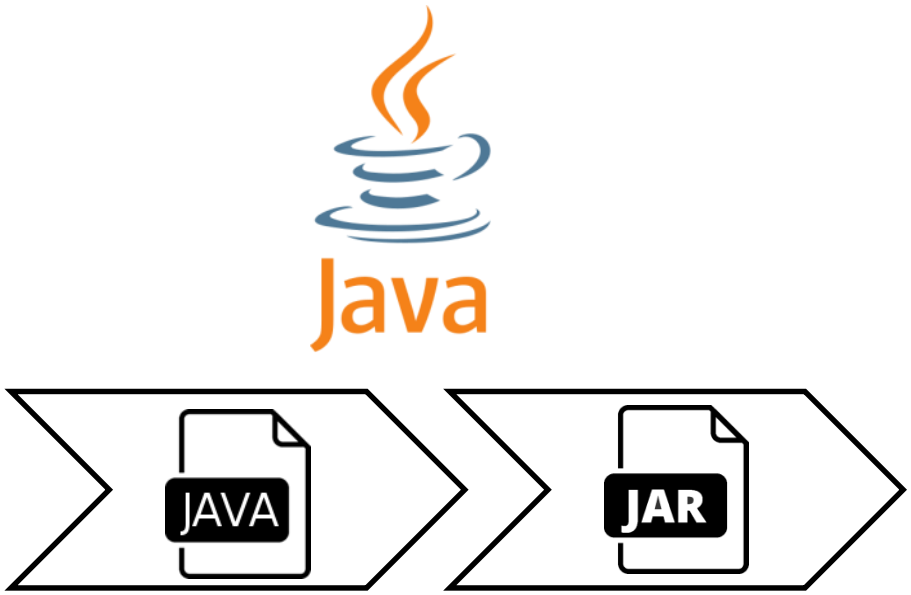
Java agents allow to add a decentralized infrastructure at runtime. In the source code, the infrastructure is only declared.



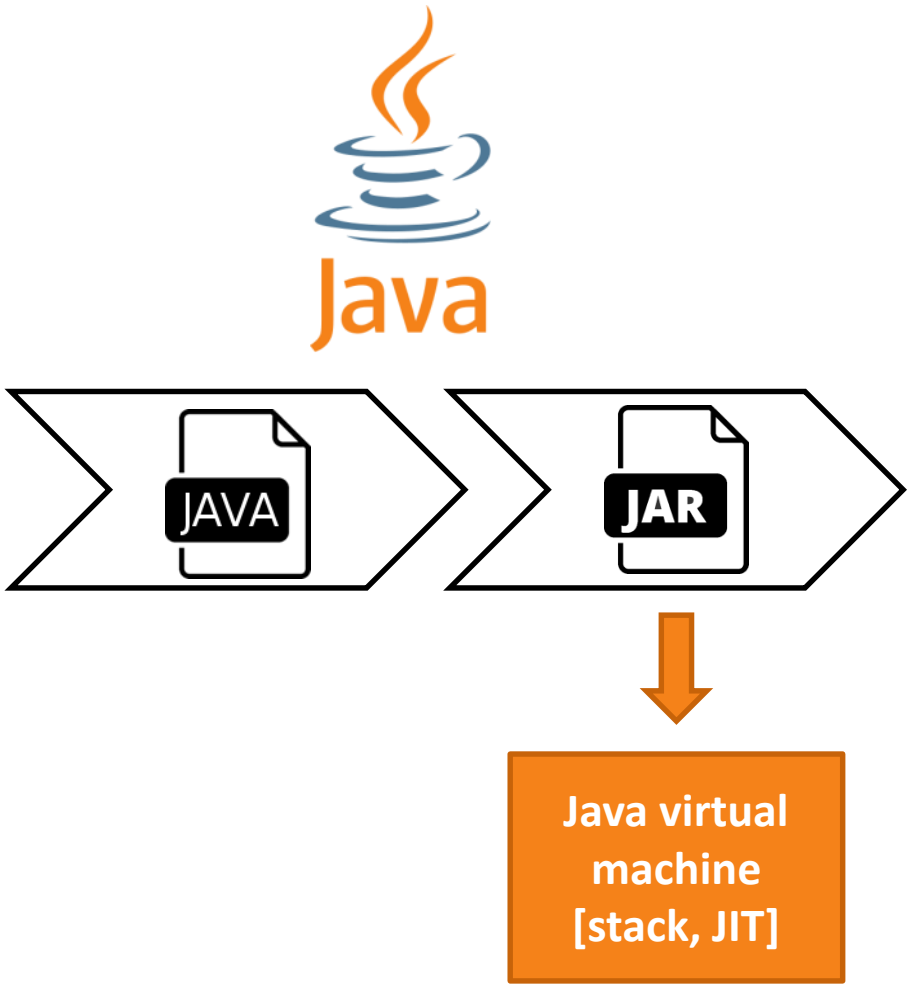
“Plain old Java applications” (POJAs)

Working with POJOs reduces complexity. Reducing infrastructure code as a goal

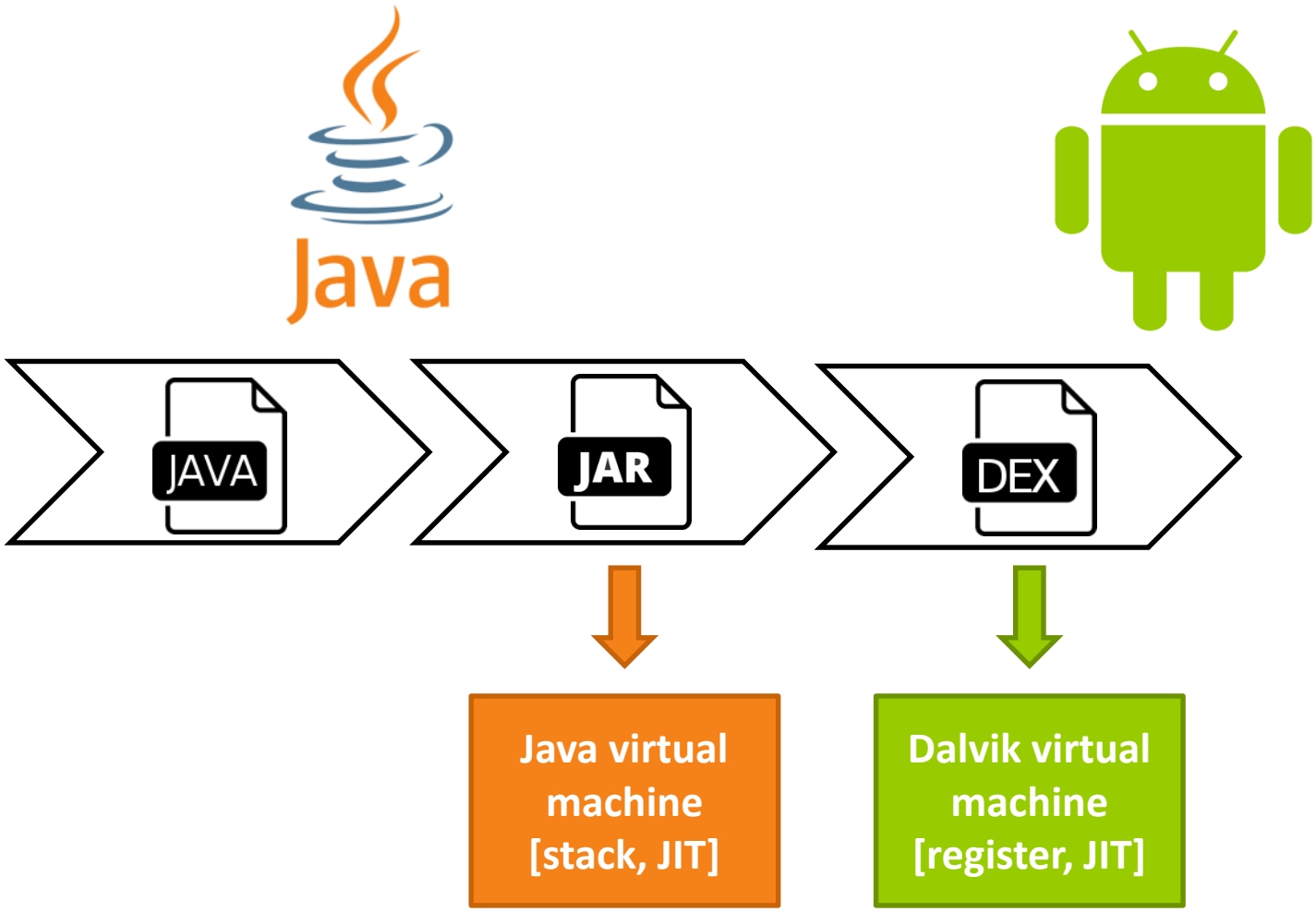
Android makes things more complicated.



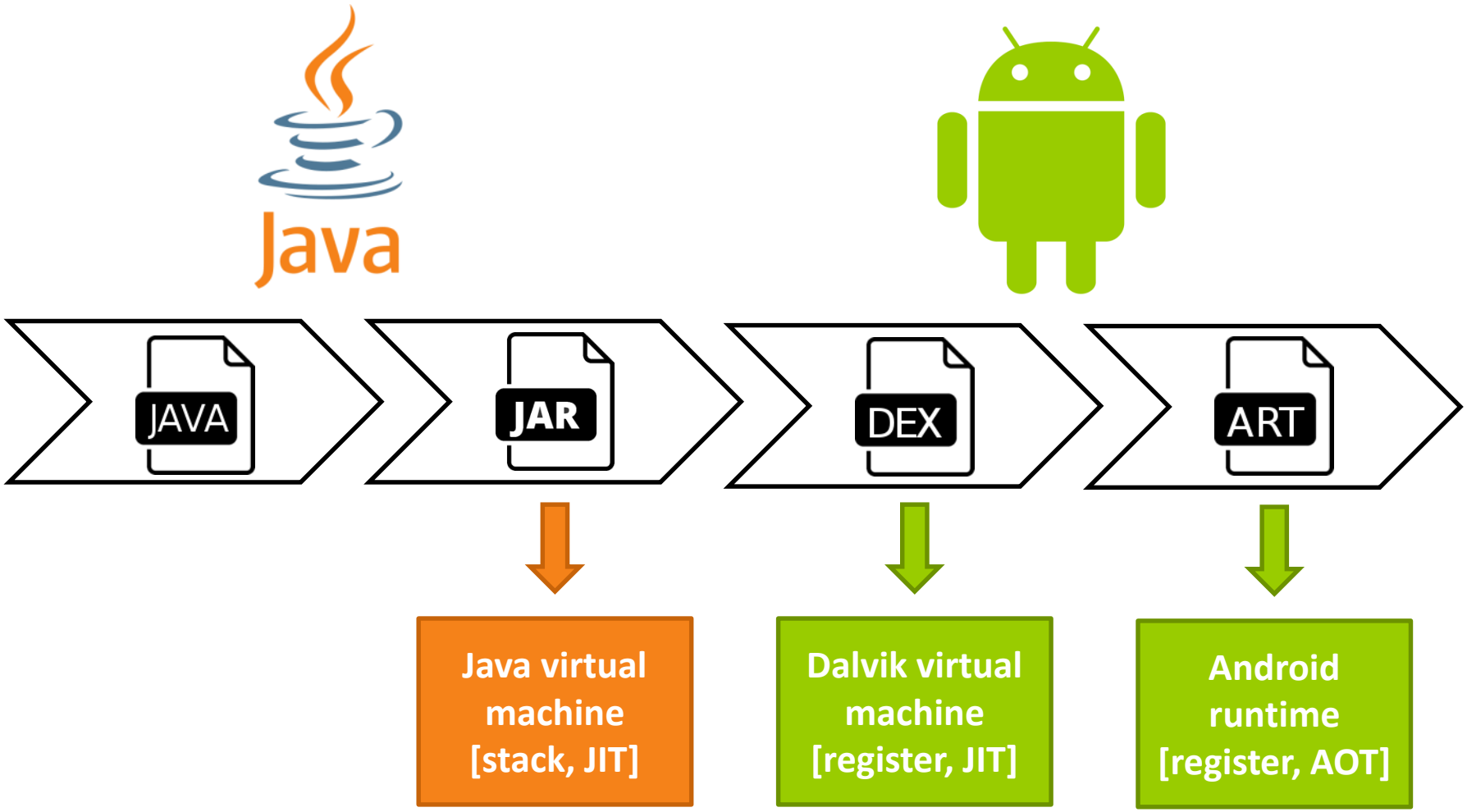
Android makes things more complicated.



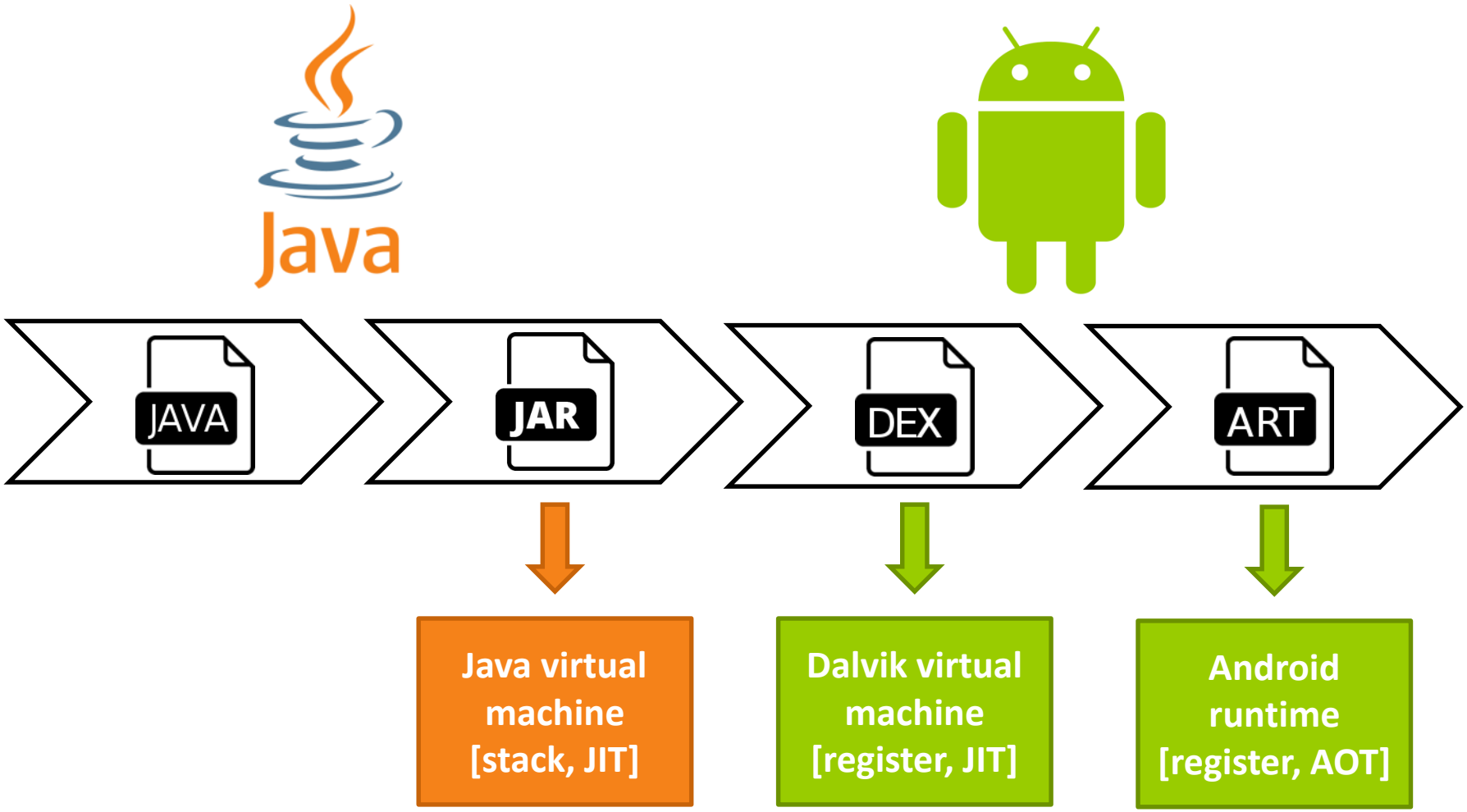
Android makes things more complicated.



Android makes things more complicated.



Android makes things more complicated.



Solution: Embed the Android SDK's dex compiler (Apache 2.0 license).
Unfortunately, only subclass instrumentation possible.

Performance: library comparison

	baseline	Byte Buddy	cglib	Javassist	Java proxy
(1)	0	142	515	193	70
(2a)	0	1‘126	960	1’070	1’060
(2b)	0.002	0.002	0.003	0.011	0.008
(3a)	0	882 *5’408*	1‘632	683	n/a
(3b)	0.004	0.004 *0.004*	0.021	0.025	n/a

Results are measured in nanoseconds (rounded).
All benchmarks run with JMH, source code: https://github.com/raphw/byte-buddy
(1) Extending the Object class without any methods but with a default constructor
(2a) Implementing an interface with 18 methods, method stubs
(2b) Executing a method of this interface
(3a) Extending a class with 18 methods, super method invocation
(3b) Executing a method of this class

Performance: library comparison

	baseline	Byte Buddy	cglib	Javassist	Java proxy
(1)	0	142	515	193	70
(2a)	0	1‘126	960	1’070	1’060
(2b)	0.002	0.002	0.003	0.011	0.008
(3a)	0	882 *5’408*	1‘632	683	n/a
(3b)	0.004	0.004 *0.004*	0.021	0.025	n/a

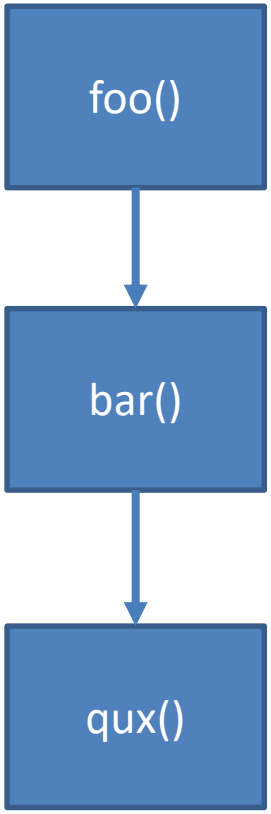
Results are measured in nanoseconds (rounded).
All benchmarks run with JMH, source code: https://github.com/raphw/byte-buddy
(1) Extending the Object class without any methods but with a default constructor
(2a) Implementing an interface with 18 methods, method stubs
(2b) Executing a method of this interface
(3a) Extending a class with 18 methods, super method invocation
(3b) Executing a method of this class

Performance: library comparison

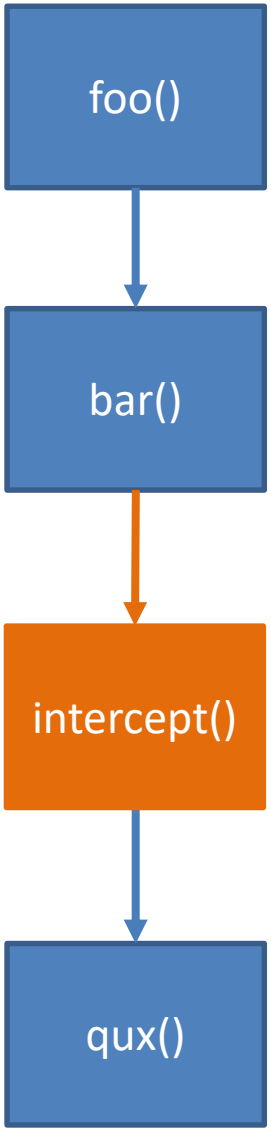
	baseline	Byte Buddy	cglib	Javassist	Java proxy
(1)	0	142	515	193	70
(2a)	0	1‘126	960	1’070	1’060
(2b)	0.002	0.002	0.003	0.011	0.008
(3a)	0	882 *5’408*	1‘632	683	n/a
(3b)	0.004	0.004 *0.004*	0.021	0.025	n/a

Results are measured in nanoseconds (rounded).
All benchmarks run with JMH, source code: https://github.com/raphw/byte-buddy
(1) Extending the Object class without any methods but with a default constructor
(2a) Implementing an interface with 18 methods, method stubs
(2b) Executing a method of this interface
(3a) Extending a class with 18 methods, super method invocation
(3b) Executing a method of this class

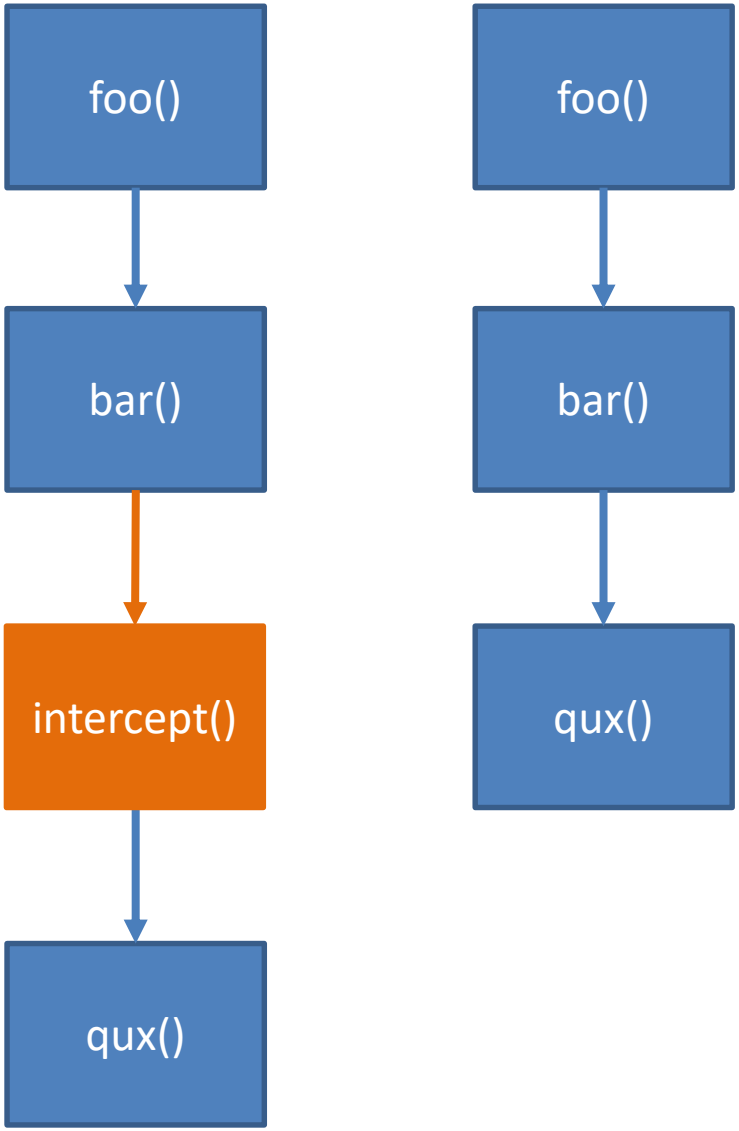
Performance: delegation vs. advice



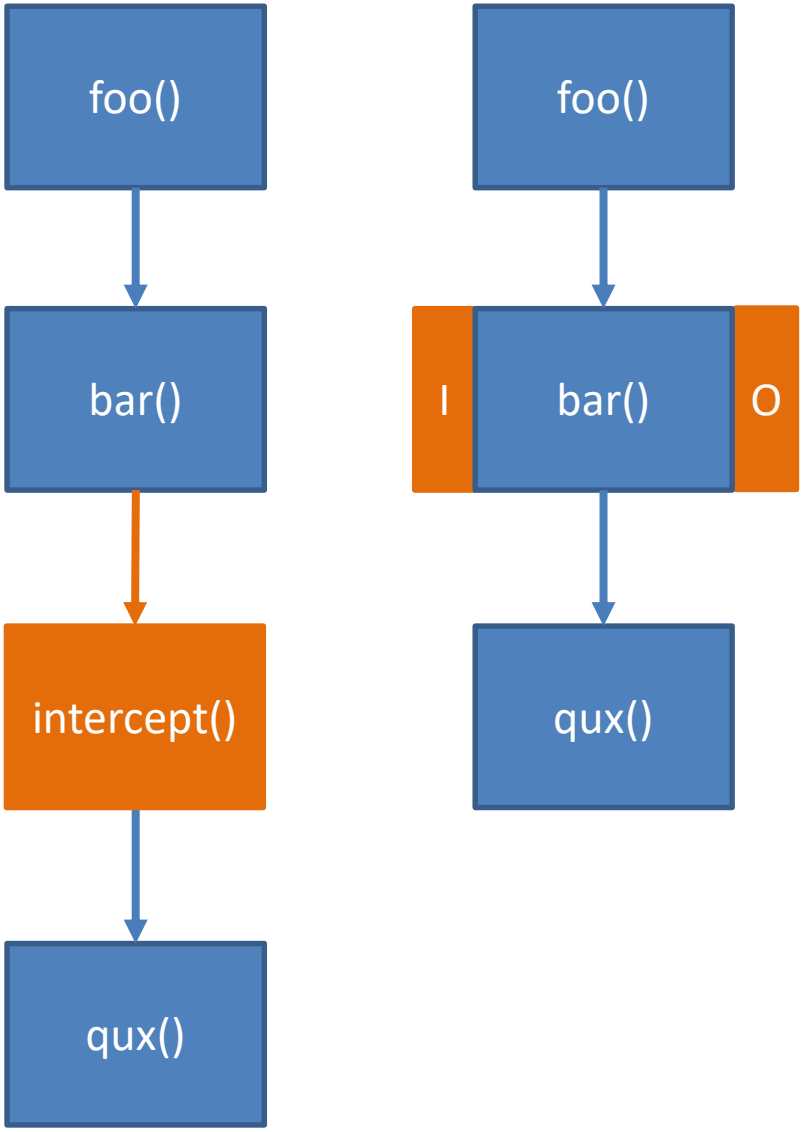
Performance: delegation vs. advice



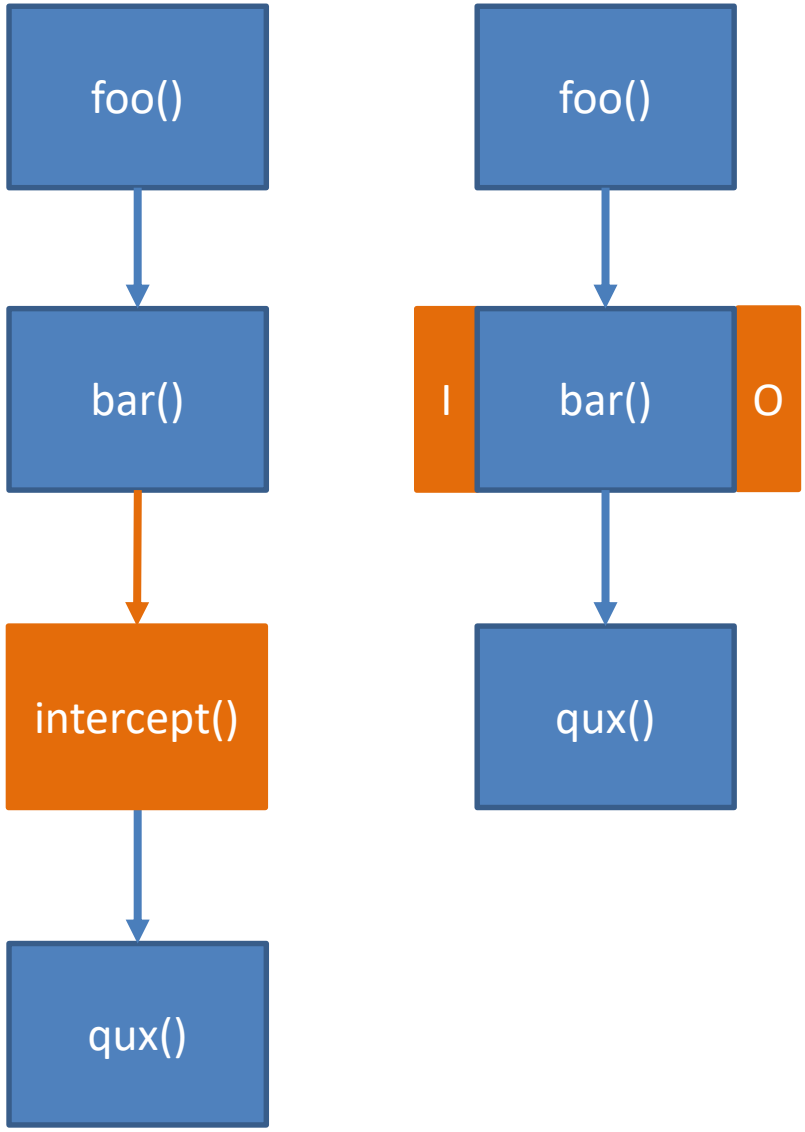
Performance: delegation vs. advice



Performance: delegation vs. advice



Performance: delegation vs. advice



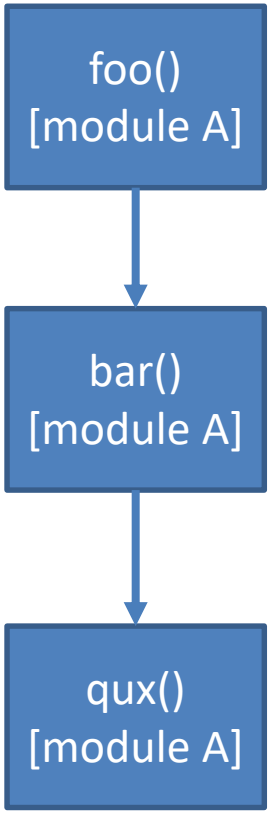
-XX:MaxInlineSize=35 (auto-reduced)
-XX:FreqInlineSize=325
-XX:InlineSmallCode=2000

-XX:MaxInlineLevel=9

-XX:InlineSynchronizedMethods=true

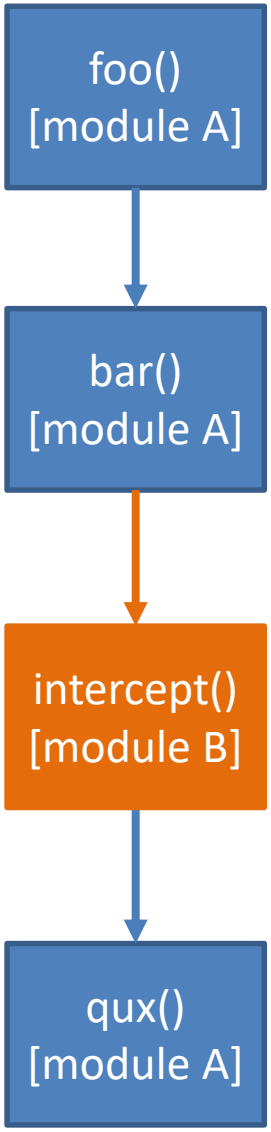
-XX:ReservedCodeCacheSize=48M
-XX:+PrintCodeCache
-XX:+PrintCodeCacheOnCompilation

Java 9: instrumentation meets modules



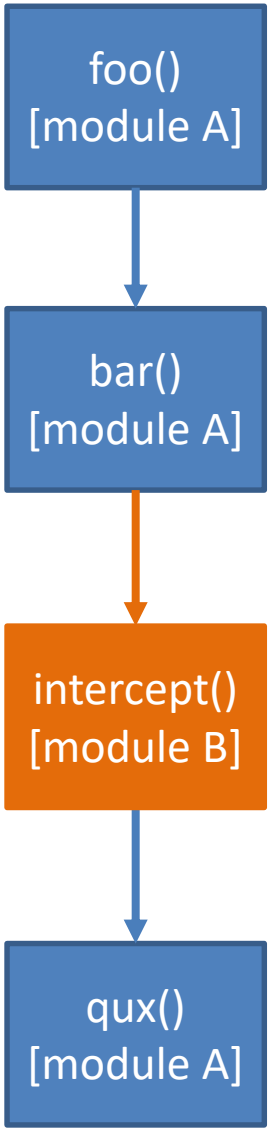

```
instrumentation.addReads(moduleA, moduleB);
```

Java 9: instrumentation meets modules




```
instrumentation.addReads(moduleA, moduleB);
```

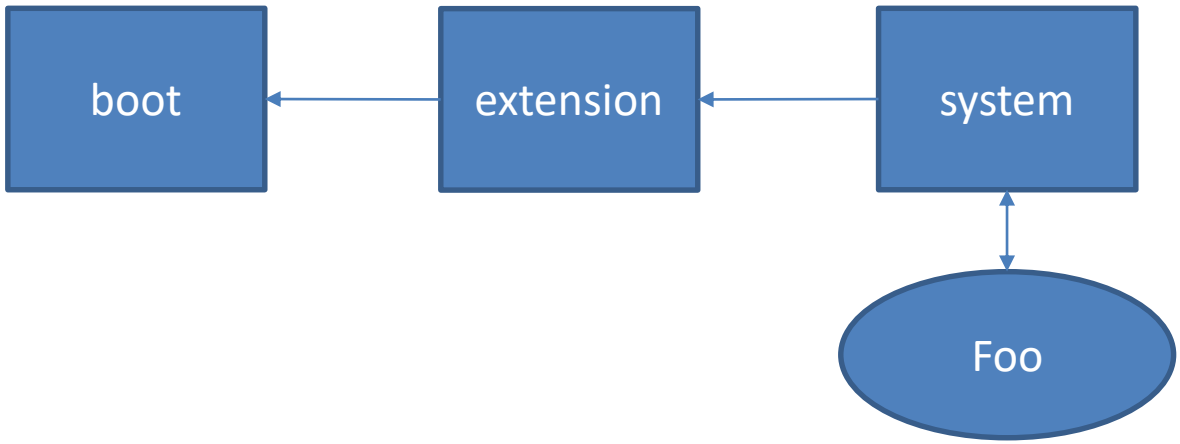
Java 9: instrumentation meets modules



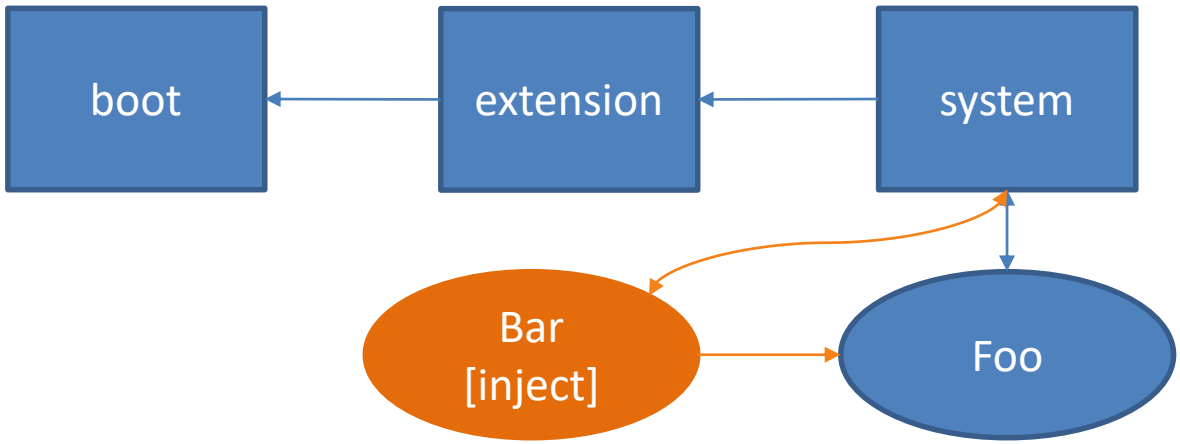

```
Module moduleA = findModule("A"),
       moduleB = findModule("B");

moduleA.addReads(moduleB);


instrumentation.addReads(moduleA, moduleB);
```

Loading and leaking runtime classes

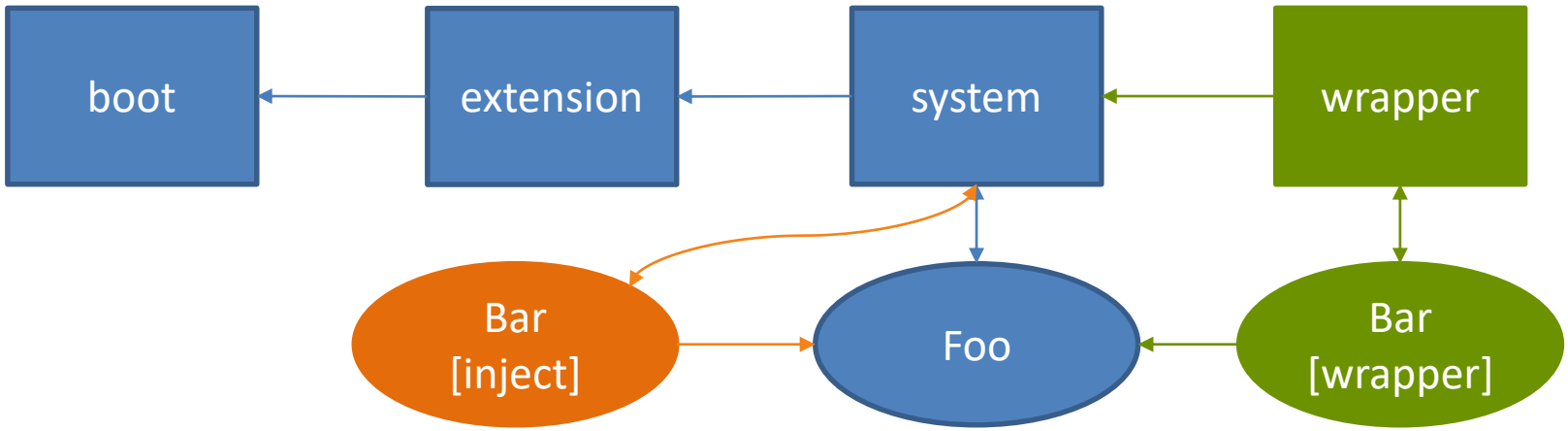
Loading and leaking runtime classes



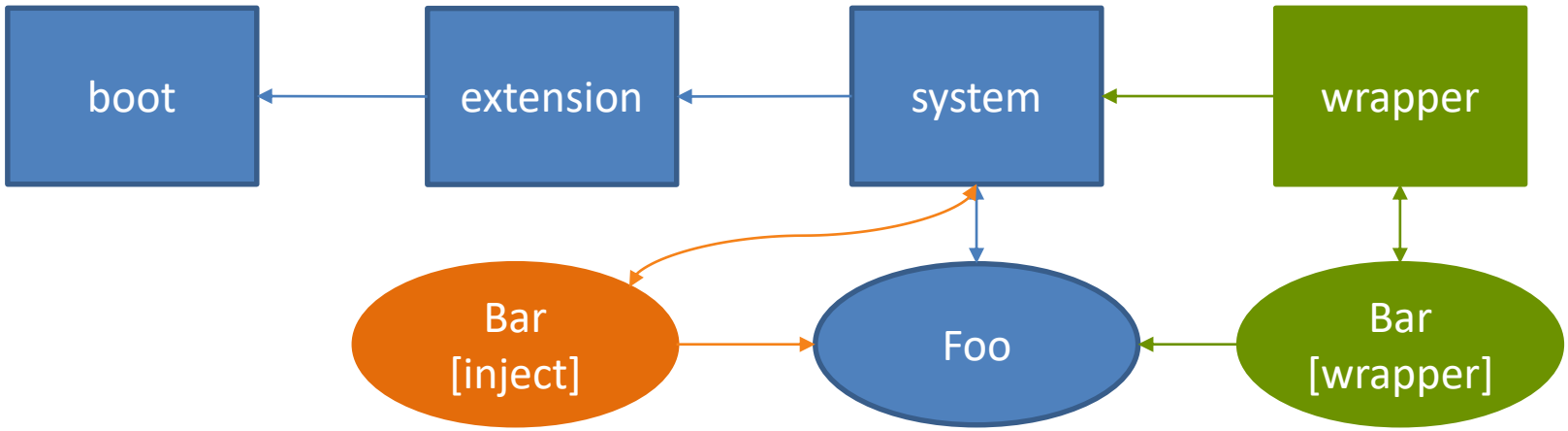
Loading and leaking runtime classes



Loading and leaking runtime classes

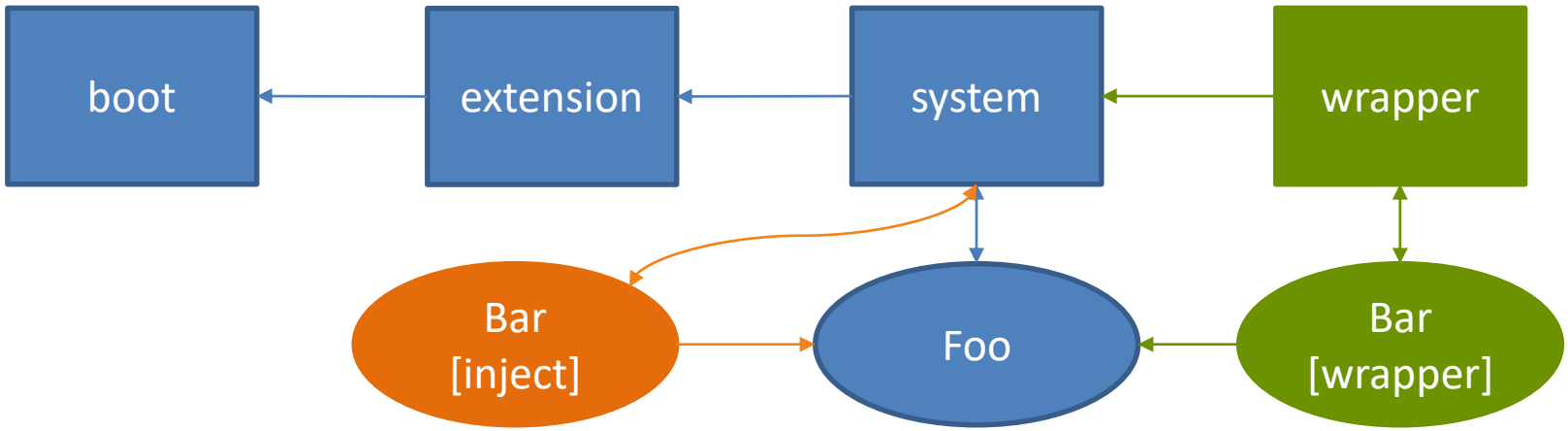


Loading and leaking runtime classes



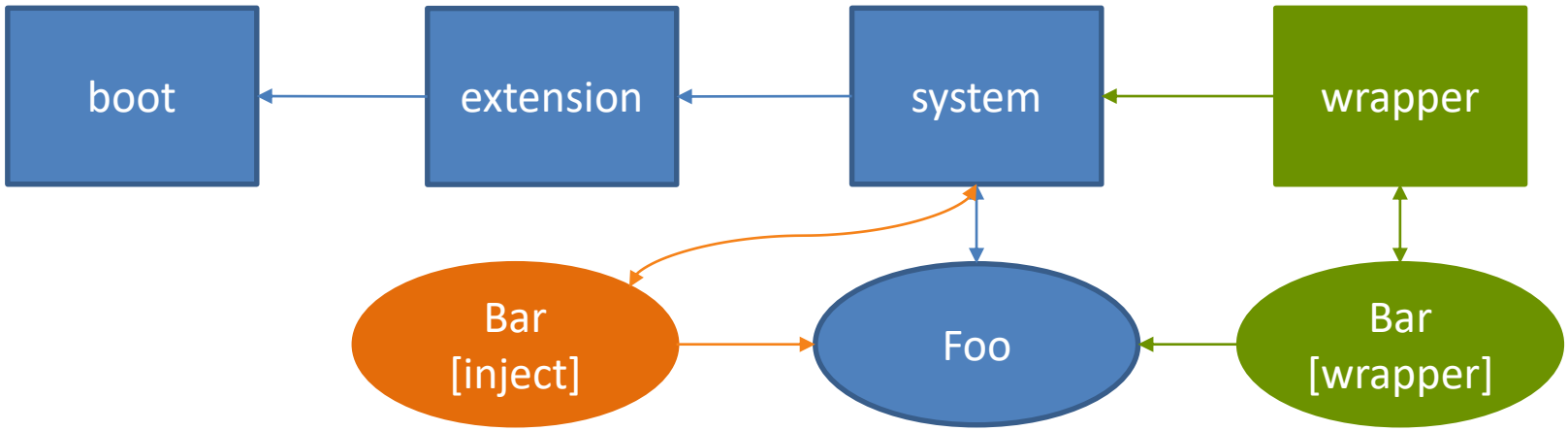

```
public class Foo {
  /* package-private */ String bar() { return "foo"; }
}

public class Bar extends Foo {
  @Override
  /* package-private */ String bar() { return "bar"; }
}
```

Loading and leaking runtime classes




```
public class Foo {
  /* package-private */ String bar() { return "foo"; }
}

public class Bar extends Foo {
  @Override
  /* package-private */ String bar() { return "bar"; }
}
```

Loading and leaking runtime classes




```
public class Foo {
  /* package-private */ String bar() { return "foo"; }
}

public class Bar extends Foo {
  @Override
  /* package-private */ String bar() { return "bar"; }
}
```

Caching runtime classes

Caching runtime classes


```
WeakHashMap<Class<?>, Class<?>> proxies;
```

Caching runtime classes


```
WeakHashMap<Class<?>, Class<?>> proxies;
```

Caching runtime classes


```
WeakHashMap<Class<?>, Class<?>> proxies;
```

`Map<Class<?>, Class<?>>`

Value references key strongly (parent class). Requires active life-cycle management.
When injecting into the source class loader, this is the only meaningful solution.

Caching runtime classes


```
WeakHashMap<Class<?>, Class<?>> proxies;
```

`Map<Class<?>, Class<?>>`

Value references key strongly (parent class). Requires active life-cycle management. When injecting into the source class loader, this is the only meaningful solution.

`WeakHashMap<Class<?>, WeakReference<Class<?>>>`

Proxy class is collected when there is no instance available.

Caching runtime classes


```
WeakHashMap<Class<?>, Class<?>> proxies;
```

`Map<Class<?>, Class<?>>`

Value references key strongly (parent class). Requires active life-cycle management. When injecting into the source class loader, this is the only meaningful solution.

`WeakHashMap<Class<?>, WeakReference<Class<?>>>`

Proxy class is collected when there is no instance available.

`WeakHashMap<Class<?>, SoftReference<Class<?>>>`

Proxy class is collected when the heap is filled and there is no instance available.

Caching runtime classes


```
WeakHashMap<Class<?>, Class<?>> proxies;
```

`Map<Class<?>, Class<?>>`
Value references key strongly (parent class). Requires active life-cycle management.
When injecting into the source class loader, this is the only meaningful solution.

`WeakHashMap<Class<?>, WeakReference<Class<?>>>`
Proxy class is collected when there is no instance available.

`WeakHashMap<Class<?>, SoftReference<Class<?>>>`
Proxy class is collected when the heap is filled and there is no instance available.

The only proper solution would be ephemerons which are not available in the JVM.
Discussed in January 2016 / OpenJDK mailing list (core-libs-dev).

Dealing with bridge methods

```
class Foo<T> {
  void bar(T arg) { /* do something */ }
}

class Bar<S extends Number> extends Foo<S> {
  @Override
  void bar(S arg) { /* do something */ }
}
```

Dealing with bridge methods

```
class Foo {
  void bar(Object arg) { /* do something */ }
}

class Bar extends Foo {
  @Override
  void bar(Number arg) { /* do something */ }
  /* bridge */
  void bar(Object arg) { this.bar((Number) arg); }
}
```

Dealing with bridge methods

```
class Foo {
  void bar(Object arg) { /* do something */ }
}

class Bar extends Foo {
  @Override
  void bar(Number arg) { /* do something */ }
  /* bridge */
  void bar(Object arg) { this.bar((Number) arg); }
}
```

Dealing with bridge methods

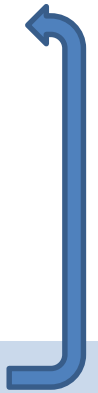
```
class Foo {
  void bar(Object arg) { /* do something */ }
}

class Bar extends Foo {
  @Override
  void bar(Number arg) { /* do something */ }
  /* bridge */
  void bar(Object arg) { this.bar((Number) arg); }
}


class Bar$Proxy extends Bar {
  @Override void bar(Number arg) { super.bar(arg); }
  @Override void bar(Object arg) { super.bar(arg); }
}
```

Dealing with bridge methods

```
class Foo {
  void bar(Object arg) { /* do something */ }
}

class Bar extends Foo {
  @Override
  void bar(Number arg) { /* do something */ }
  /* bridge */
  void bar(Object arg) { this.bar((Number) arg); }
}


class Bar$Proxy extends Bar {
  @Override void bar(Number arg) { super.bar(arg); }
  @Override void bar(Object arg) { super.bar(arg); }
}
```

Dealing with bridge methods

```
class Foo {
  void bar(Object arg) { /* do something */ }
}

class Bar extends Foo {
  @Override
  void bar(Number arg) { /* do something */ }
  /* bridge */
  void bar(Object arg) { this.bar((Number) arg); }
}


class Bar$Proxy extends Bar {
  @Override void bar(Number arg) { super.bar(arg); }
  @Override void bar(Object arg) { super.bar(arg); }
}
```



Dealing with bridge methods

```
class Foo {
  void bar(Object arg) { /* do something */ }
}

class Bar extends Foo {
  @Override
  void bar(Number arg) { /* do something */ }
  /* bridge */
  void bar(Object arg) { this.bar((Number) arg); }
}


class Bar$Proxy extends Bar {
  @Override void bar(Number arg) { super.bar(arg); }
  @Override void bar(Object arg) { super.bar(arg); }
}
```

Dealing with bridge methods: visibility bridges

```
class Foo {
  public void bar() { /* do something */ }
}

public class Bar extends Foo { }
```

Dealing with bridge methods: visibility bridges

```
class Foo {
  public void bar() { /* do something */ }
}

public class Bar extends Foo {
  /* bridge */
  public void bar() { super.bar(); }
}
```

Dealing with bridge methods: visibility bridges

```
class Foo {
  public void bar() { /* do something */ }
}

public class Bar extends Foo {
  /* bridge */
  public void bar() { super.bar(); }
}
```

Dealing with bridge methods: visibility bridges

```
class Foo {
  public void bar() { /* do something */ }
}

public class Bar extends Foo {
  /* bridge */
  public void bar() { super.bar(); }
}
```

```
class Qux {
  public final void bar() { /* do something */ }
}

public class Baz extends Qux { }
```

Dealing with bridge methods: visibility bridges

```
class Foo {
  public void bar() { /* do something */ }
}

public class Bar extends Foo {
  /* bridge */
  public void bar() { super.bar(); }
}
```

```
class Qux {
  public final void bar() { /* do something */ }
}

public class Baz extends Qux { }

new Baz().bar(); // works
Baz.class.getMethod("bar").invoke(new Baz()); // access error
```

Generic runtime types and dealing with meta data

```
@interface Foo {}

interface Bar<T> {
  @Foo
  void foo(@Foo this, T arg);
}
```

Generic runtime types and dealing with meta data

```
@interface Foo {}

interface Bar<T> {
  @Foo
  void foo(@Foo this, T arg);
}


new ByteBuddy()
  .subclass(parameterizedType(Bar.class,
                              String.class).build())
  .method(named("foo"))
    .intercept(value("bar"))
    .attribute(ForInstrumentedMethod.INCLUDING_RECEIVER)
  .make();
```

Generic runtime types and dealing with meta data

```
@interface Foo {}

interface Bar<T> {
  @Foo
  void foo(@Foo this, T arg);
}


new ByteBuddy()
  .subclass(parameterizedType(Bar.class,
                              String.class).build())
  .method(named("foo"))
    .intercept(value("bar"))
    .attribute(ForInstrumentedMethod.INCLUDING_RECEIVER)
  .make();
```

Generic runtime types and dealing with meta data

```
@interface Foo {}

interface Bar<T> {
  @Foo
  void foo(@Foo this, T arg);
}


new ByteBuddy()
  .subclass(parameterizedType(Bar.class,
                              String.class).build())
  .method(named("foo"))
    .intercept(value("bar"))
    .attribute(ForInstrumentedMethod.INCLUDING_RECEIVER)
  .make();
```

Generic runtime types and dealing with meta data

```
@interface Foo {}

interface Bar<T> {
  @Foo
  void foo(@Foo this, T arg);
}


new ByteBuddy()
  .subclass(parameterizedType(Bar.class,
                              String.class).build())
  .method(named("foo"))
    .intercept(value("bar"))
    .attribute(ForInstrumentedMethod.INCLUDING_RECEIVER)
  .make();
```

Byte Buddy handles meta data and generic types transparently.

Not only important for “build-time” instrumentation but also for meta data APIs.

Super method proxies: cglib

```
interface Foo {
  void bar(int arg);
  String qux(String arg);
}
```

Super method proxies: cglib

```
interface Foo {
  void bar(int arg);
  String qux(String arg);
}

class $MethodProxy implements MethodProxy {
  @Override
  public Object invokeSuper(Object self,
                            Method method,
                            Object[] arguments) {
    if (method.equals($bar)) {
      ((Foo) self).bar((Integer) arguments[0]);
      return null;
    } else if (method.equals($qux)) {
      return ((Foo) self).qux((String) arguments[0]);
    } else {
      throw new IllegalStateException(); }
  }
}
```

Super method proxies: cglib

```
interface Foo {
  void bar(int arg);
  String qux(String arg);
}

class $MethodProxy implements MethodProxy {
  @Override
  public Object invokeSuper(Object self,
                            Method method,
                            Object[] arguments) {
    if (method.equals($bar)) {
      ((Foo) self).bar((Integer) arguments[0]);
      return null;
    } else if (method.equals($qux)) {
      return ((Foo) self).qux((String) arguments[0]);
    } else {
      throw new IllegalStateException(); }
  }
}
```

Super method proxies: cglib

```
interface Foo {
  void bar(int arg);
  String qux(String arg);
}

class $MethodProxy implements MethodProxy {
  @Override
  public Object invokeSuper(Object self,
                            Method method,
                            Object[] arguments) {
    if (method.equals($bar)) {
      ((Foo) self).bar((Integer) arguments[0]);
      return null;
    } else if (method.equals($qux)) {
      return ((Foo) self).qux((String) arguments[0]);
    } else {
      throw new IllegalStateException(); }
  }
}
```

Super method proxies: cglib

```
interface Foo {
  void bar(int arg);
  String qux(String arg);
}

class $MethodProxy implements MethodProxy {
  @Override
  public Object invokeSuper(Object self,
                            Method method,
                            Object[] arguments) {
    if (method.equals($bar)) {
      ((Foo) self).bar((Integer) arguments[0]);
      return null;
    } else if (method.equals($qux)) {
      return ((Foo) self).qux((String) arguments[0]);
    } else {
      throw new IllegalStateException(); }
  }
}
```

Super method proxies: cglib

```
interface Foo {
  void bar(int arg);
  String qux(String arg);
}
```

Super method proxies: cglib

```
interface Foo {
  void bar(int arg);
  String qux(String arg);
}

class $BarMethodProxy implements Runnable {
  final Foo self; final int arg; // constructor omitted
  @Override
  public void run() {
    self.bar(arg);
  }
}
```

Super method proxies: cglib

```
interface Foo {
  void bar(int arg);
  String qux(String arg);
}

class $BarMethodProxy implements Runnable {
  final Foo self; final int arg; // constructor omitted
  @Override
  public void run() {
    self.bar(arg);
  }
}
```

Super method proxies: cglib

```
interface Foo {
  void bar(int arg);
  String qux(String arg);
}

class $BarMethodProxy implements Runnable {
  final Foo self; final int arg; // constructor omitted
  @Override
  public void run() {
    self.bar(arg);
  }
}
```

Super method proxies: cglib

```
interface Foo {
  void bar(int arg);
  String qux(String arg);
}

class $BarMethodProxy implements Runnable {
  final Foo self; final int arg; // constructor omitted
  @Override
  public void run() {
    self.bar(arg);
  }
}

class $QuxMethodProxy implements Callable<String> {
  final Foo self; final String arg; // constructor omitted
  @Override
  public String call() {
    return self.bar(arg);
  }
}
```

Problems caused by “optional” types

```
class SomeFramework {

  void doFrameworkStuff() {
    if (isFancyPluginAvailable()) {
      init(new OptionalPluginType());
    }
    /* do work */
  }

  private void init(OptionalPluginType plugin) {
    /* do work */
  }
}
```

Problems caused by “optional” types

```
class SomeFramework {

  void doFrameworkStuff() {
    if (isFancyPluginAvailable()) {
      init(new OptionalPluginType());
    }
    /* do work */
  }

  private void init(OptionalPluginType plugin) {
    /* do work */
  }
}
```

Problems caused by “optional” types

```
class SomeFramework {

  void doFrameworkStuff() {
    if (isFancyPluginAvailable()) {
      init(new OptionalPluginType());
    }
    /* do work */
  }

  private void init(OptionalPluginType plugin) {
    /* do work */
  }
}
```

Problems caused by “optional” types

```
class SomeFramework {

  void doFrameworkStuff() {
    if (isFancyPluginAvailable()) {
      init(new OptionalPluginType());
    }
    /* do work */
  }

  private void init(OptionalPluginType plugin) {
    /* do work */
  }
}


// causes NoClassDefFoundError
SomeFramework.class.getMethods();
```

Not requiring loaded types: an reflection API alternative

When a class file transformer is applied, a class is not yet loaded and it is therefore not possible to access loaded types. To provide an API for matchers and transformers, Byte Buddy offers an abstraction to the Java reflection API:

```
interface TypeDescription
interface MethodDescription
interface FieldDescription
interface AnnotationDescription
```

Any reference of such a description is resolved lazily. This also allows working with optional types. The Byte Buddy agent builder offers an automatic fallback as using loaded types is more performant.

http://rafael.codes
@rafaelcodes

---

http://documents4j.com
https://github.com/documents4j/documents4j

---

http://bytebuddy.net
https://github.com/raphw/byte-buddy