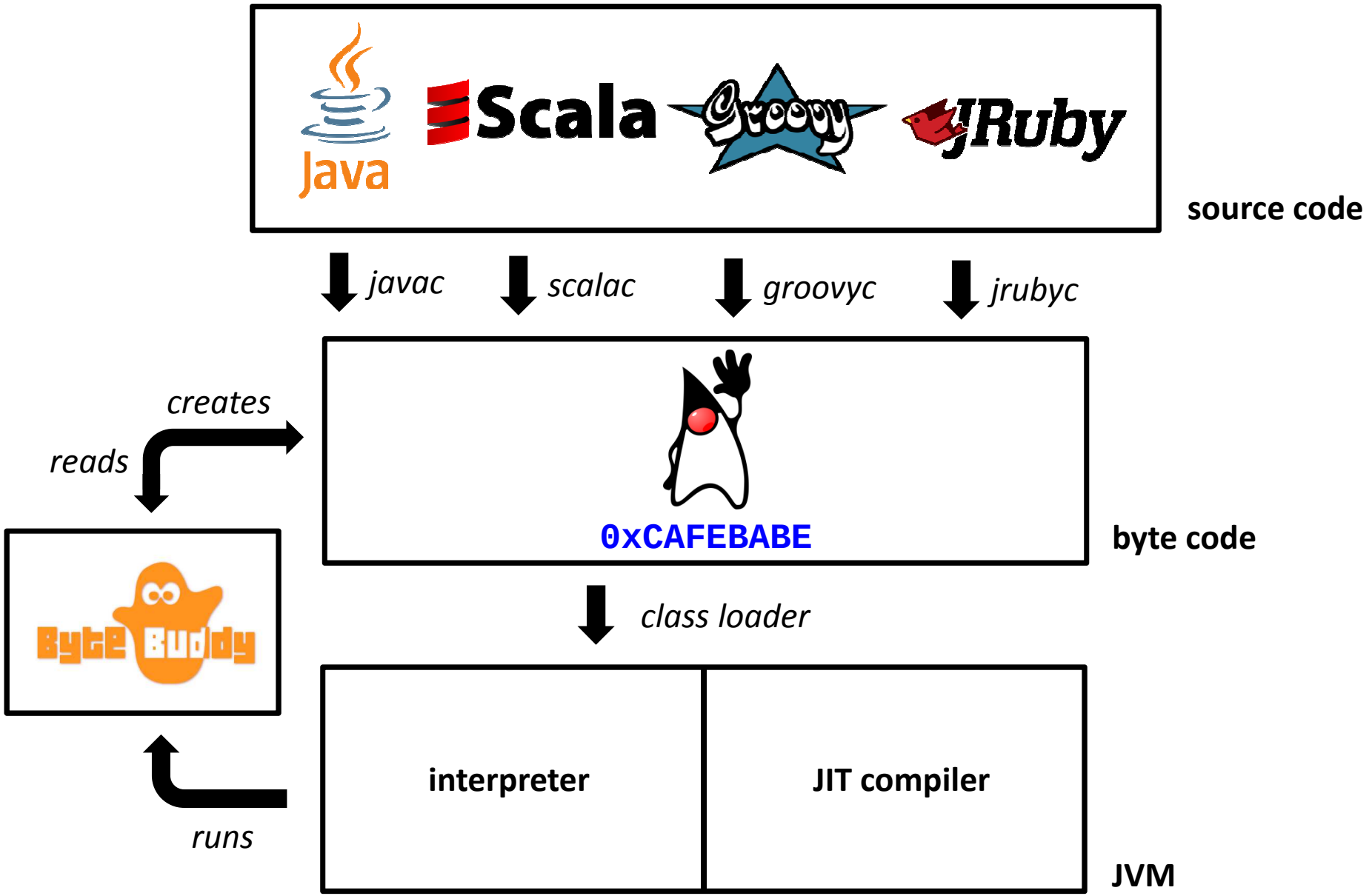
Java byte code
in practice


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

source code

```
void foo() {
  return;
}
```

byte code

```
0xB1 RETURN
```

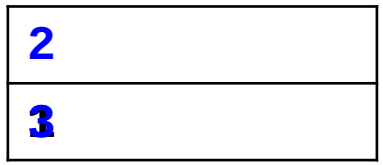
source code

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

byte code

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

operand stack



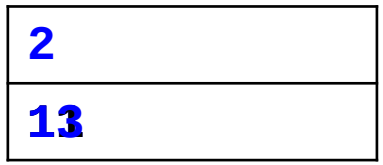
source code

```
int foo() {
  return 11 + 2;
}
```

byte code

```
0x10SH 0x0B
0x05ST_2
0x60
0xACURN
```

operand stack



source code

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

byte code

```
0x1B  ILOAD_1
0x1C  ICONST_1
0x1D  IADD
0x1E  IRETURN
```

operand stack

1
i + 1

local variable array

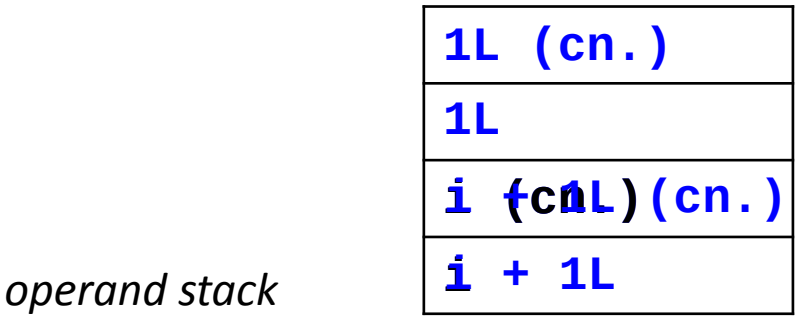
1 : i
0 : this

source code

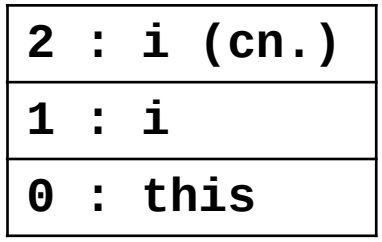
```
long foo(long i) {
  return i + 1L;
}
```

byte code

```
0x1E  LLOAD_1
0x88  I2L
0x0A  LCONST_1
0x61  LADD
0xAD  LRETURN
```



operand stack



local variable array

source code

```
short foo(short i) {
    return (short) (i + 1);
}
```

byte code

```
0x1B  ILOAD_1
0x04  ICONST_1
0x60  IADD
0x93  I2S
0xAC  IRETURN
```

operand stack

1
i + 1

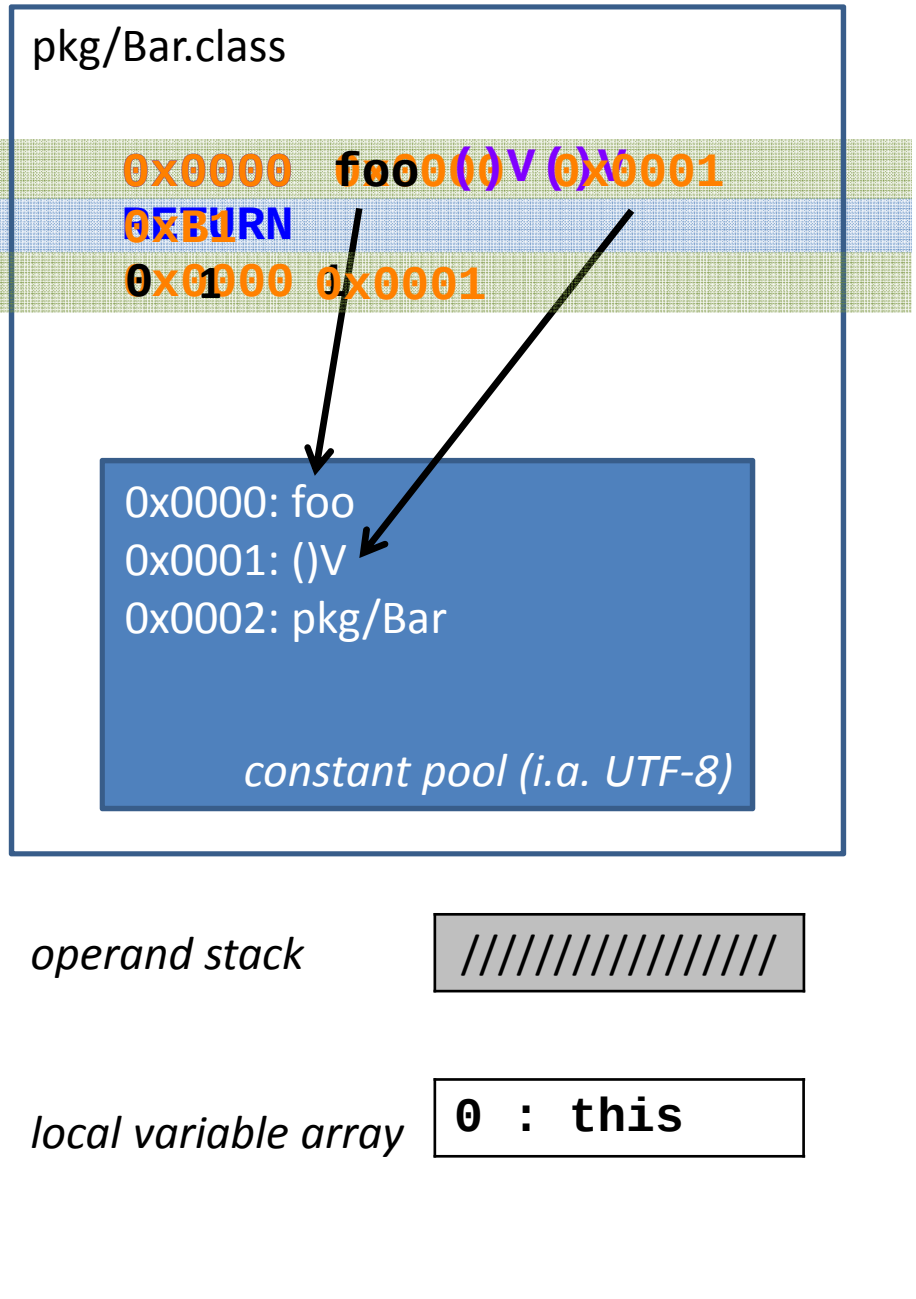
local variable array

1 : i
0 : this

source code

```
package pkg;
class Bar {
  void foo() {
    return;
  }
}
```

byte code

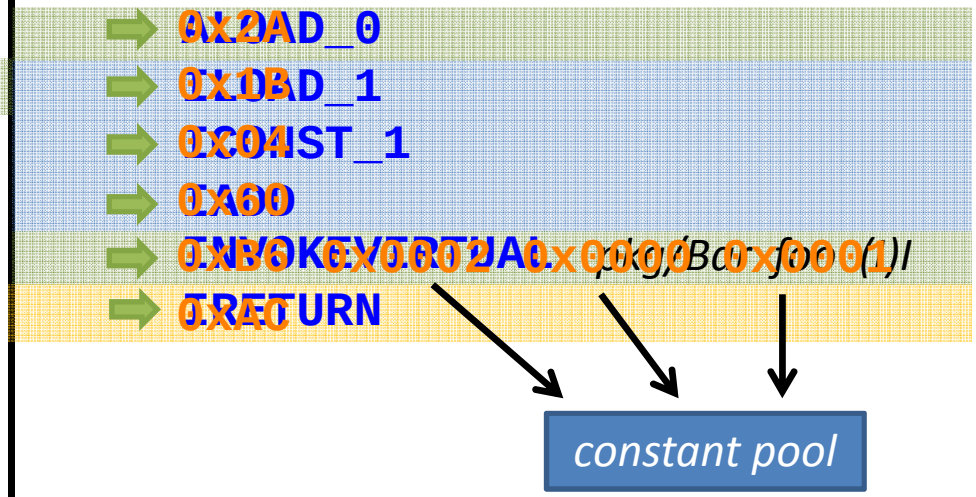


Java type	JVM type (non-array)	JVM descriptor	stack slots
boolean	I	Z	1
byte		B	1
short		S	1
char		C	1
int		I	1
long	L	J	2
float	F	F	1
double	D	D	2
void	-	V	0
java.lang.Object	A	Ljava/lang/Object;	1

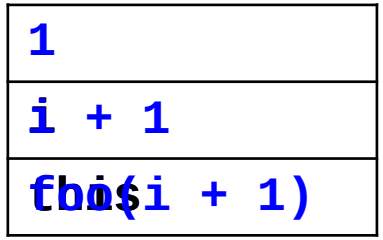
source code

```
package pkg;
class Bar {
  int foo(int i) {
    return foo(i + 1);
  }
}
```

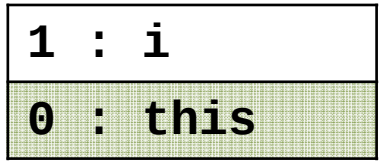
byte code



operand stack



local variable array



INVOKEVIRTUAL *pkg/Bar foo ()V*

Invokes the most-specific version of an inherited method on a non-interface class.

INVOKESTATIC *pkg/Bar foo ()V*

Invokes a static method.

INVOKESPECIAL *pkg/Bar foo ()V*

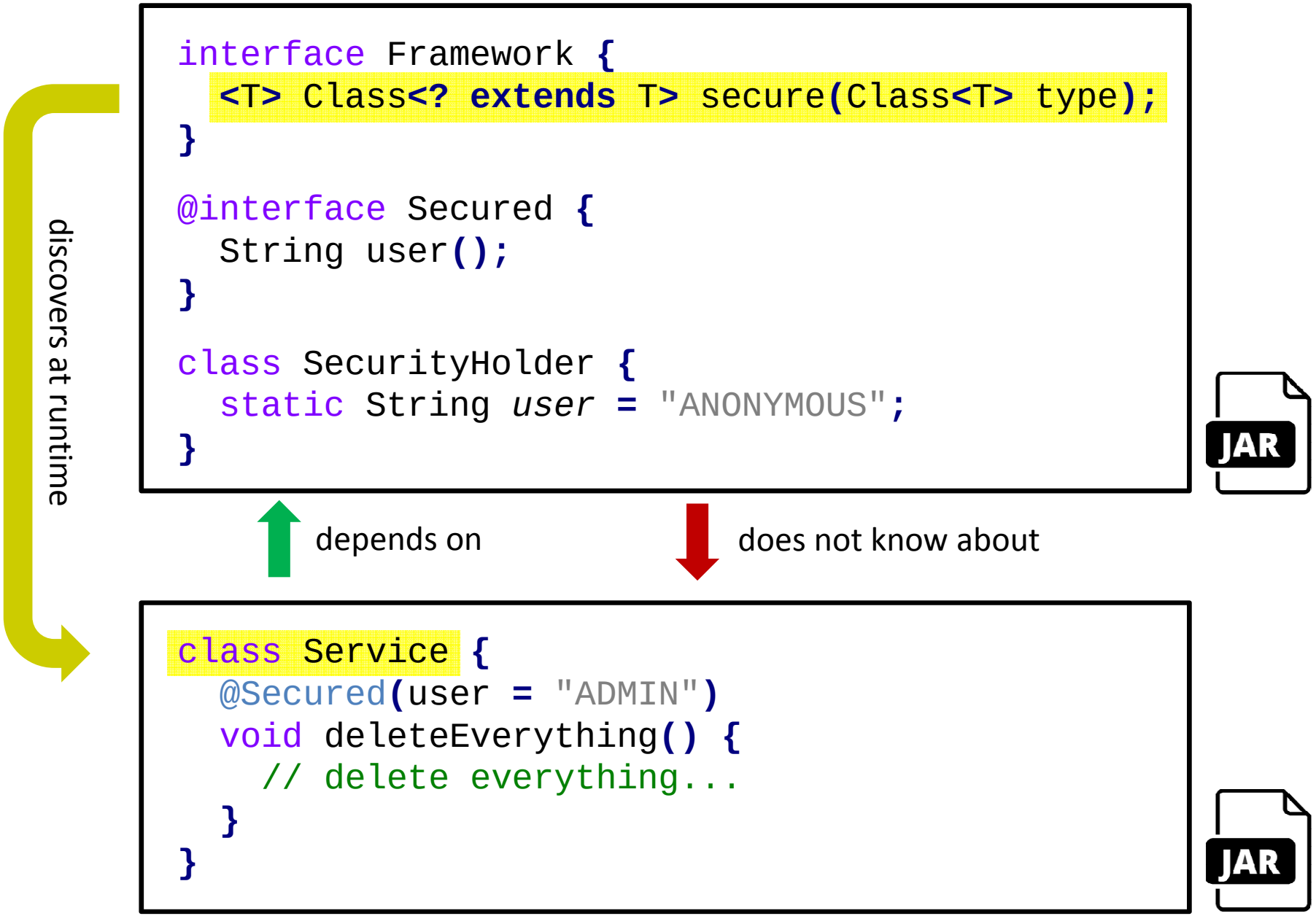
Invokes a super class's version of an inherited method.
Invokes a "constructor method".
Invokes a private method.
Invokes an interface default method (Java 8).

INVOKEINTERFACE *pkg/Bar foo ()V*

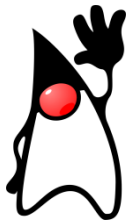
Invokes an interface method.
(Similar to INVOKEVIRTUAL but without virtual method table index optimization.)

INVOKEDYNAMIC *foo ()V bootstrap*

Queries the given *bootstrap method* for locating a method implementation at runtime.
(`MethodHandle`: Combines a specific method and an `INVOKE*` instruction.)


interface Framework {
<T> Class<? extends T> secure(Class<T> type);
}
@interface Secured {
String user();
}
class SecurityHolder {
static String user = "ANONYMOUS";
}
JAR
depends on
does not know about
discovers at runtime
class Service {
@Secured(user = "ADMIN")
void deleteEverything() {
// delete everything...
}
}
JAR

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



redefine class
(build time, agent)

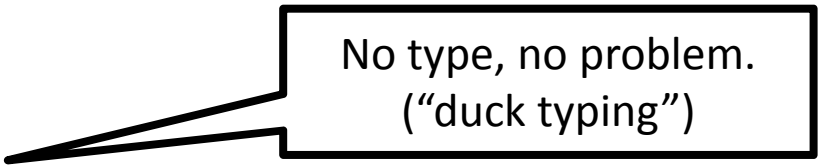
create subclass
(Liskov substitution)

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

The “black magic” prejudice.

```
var service = {
  /* @Secured(user = "ADMIN") */
  deleteEverything: function () {
    // delete everything ...
  }
}

function run(service) {
  service.deleteEverything();
}
```



In dynamic languages (also those running on the JVM) this concept is applied a lot!
For framework implementors, type-safety is conceptually impossible.
But with type information available, we are at least able to **fail fast** when generating code at runtime in case that types do not match.

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.

Byte Buddy: subclass creation

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

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

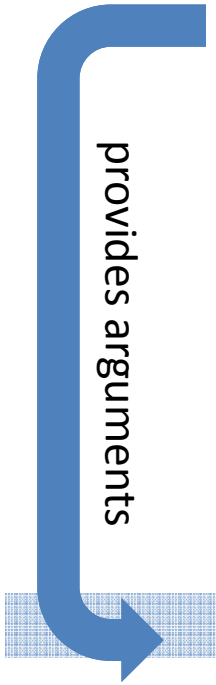
Byte Buddy: method delegation

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


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

Byte Buddy: method delegation (2)

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




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

Annotations that are not on the class path are ignored at runtime.

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

Byte Buddy: dependency injection

`@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

Byte Buddy: using Hot Swap

```
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.

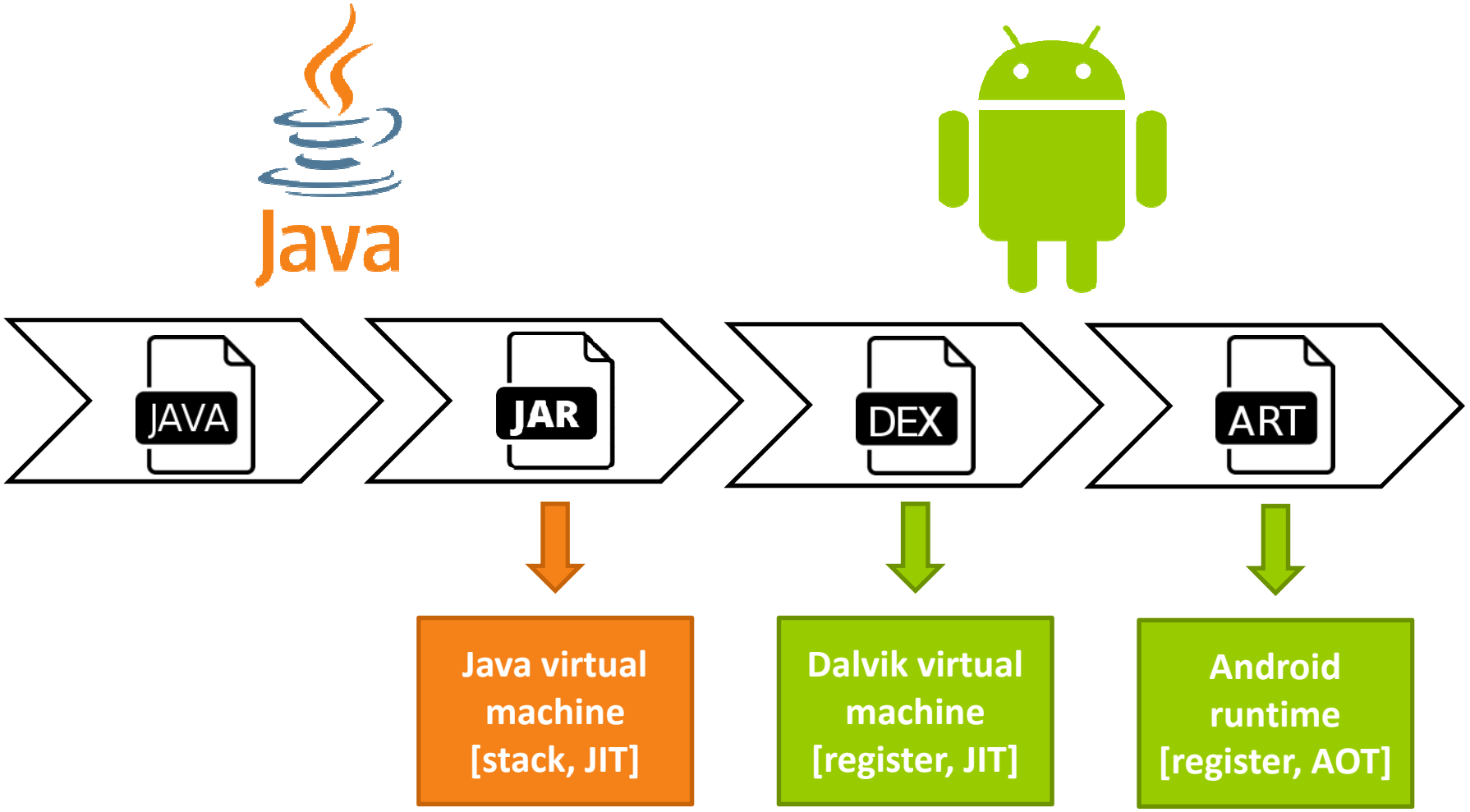
Byte Buddy: Java agents

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

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

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

Android makes things more complicated.



Solution: Embed the Android SDK's dex compiler (Apache 2.0 license).

Unfortunately, no recent version central repository deployments of Android SDK.

	Byte Buddy	cglib	Javassist	Java proxy
(1)	60.995	234.488	145.412	68.706
(2a)	153.800	804.000	706.878	973.650
(2b)	0.001	0.002	0.009	0.005
(3a)	172.126 *2290.246*	1’480.525	625.778	n/a
(3b)	0.002 *0.003*	0.019	0.027	n/a

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

http://rafael.codes
@rafaelcodes

---

http://www.bouvet.no
@bouvet

bouvet

---

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