Making Java more dynamic: runtime code generation for the JVM

scienta

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

@interface Secured {
  String user();
}

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

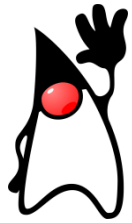
discovers at runtime

depends on

does not know about

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

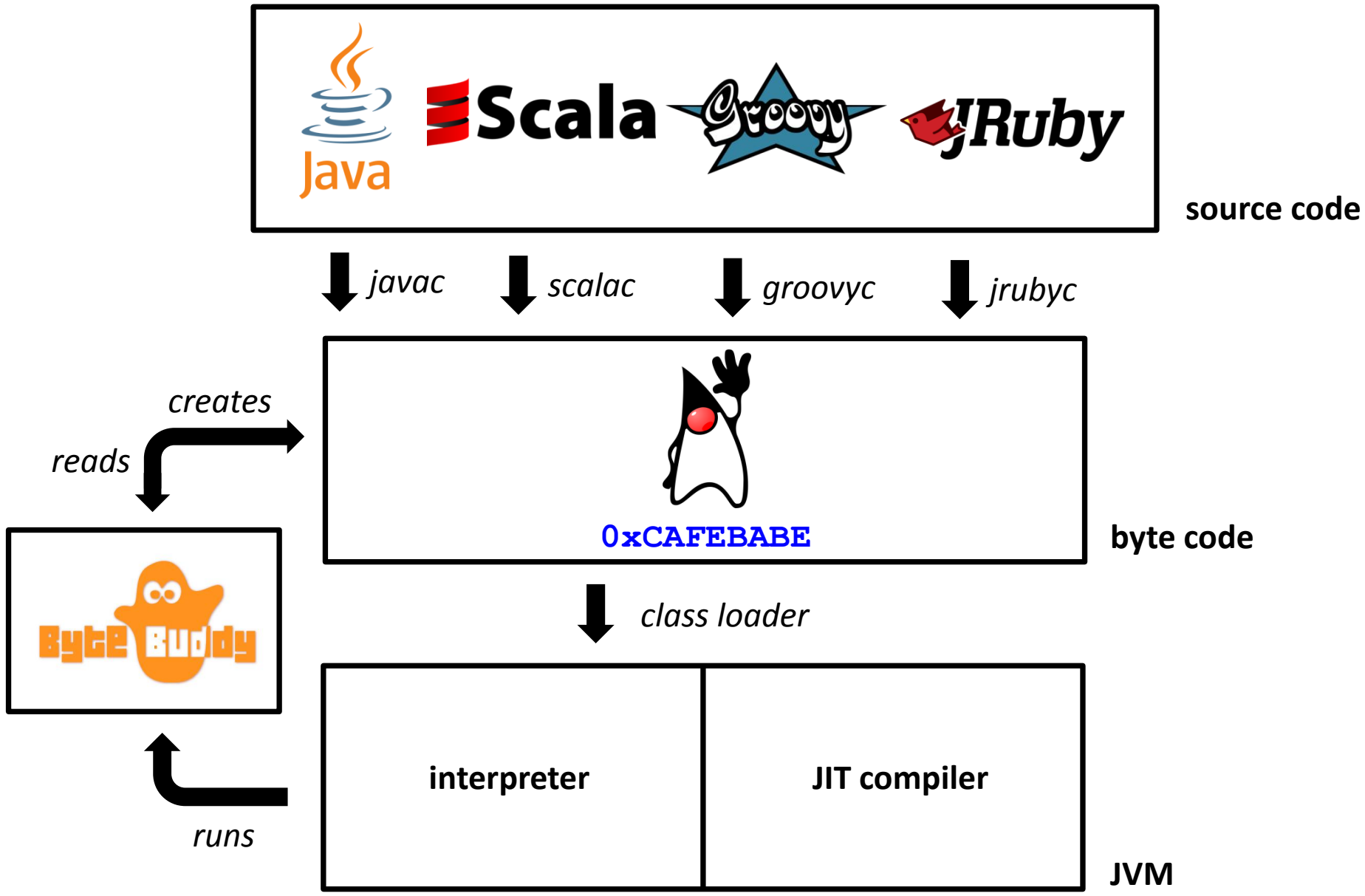
```
class SecuredService extends Service {
  @Override(user = "ADMIN")
  void deleteEverything() {
    if(!"ADMIN".equals(UserHolder.user)) {
      throw new IllegalStateException("Wrong user");
    }
    super.deleteEverything();
  }
}
```



redefine class
(build time, agent)

create subclass
(Liskov substitution)

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


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;
}
```

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.

Do-it-yourself as an alternative?

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

At best, this makes testing an issue.

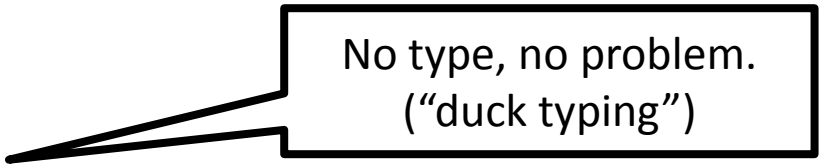
Maybe still the easiest approach for simple cross-cutting concerns.

In general, declarative programming often results in readable and modular code.

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.


spring


HIBERNATE


mockito


Guice


eclipse link


play


Clover

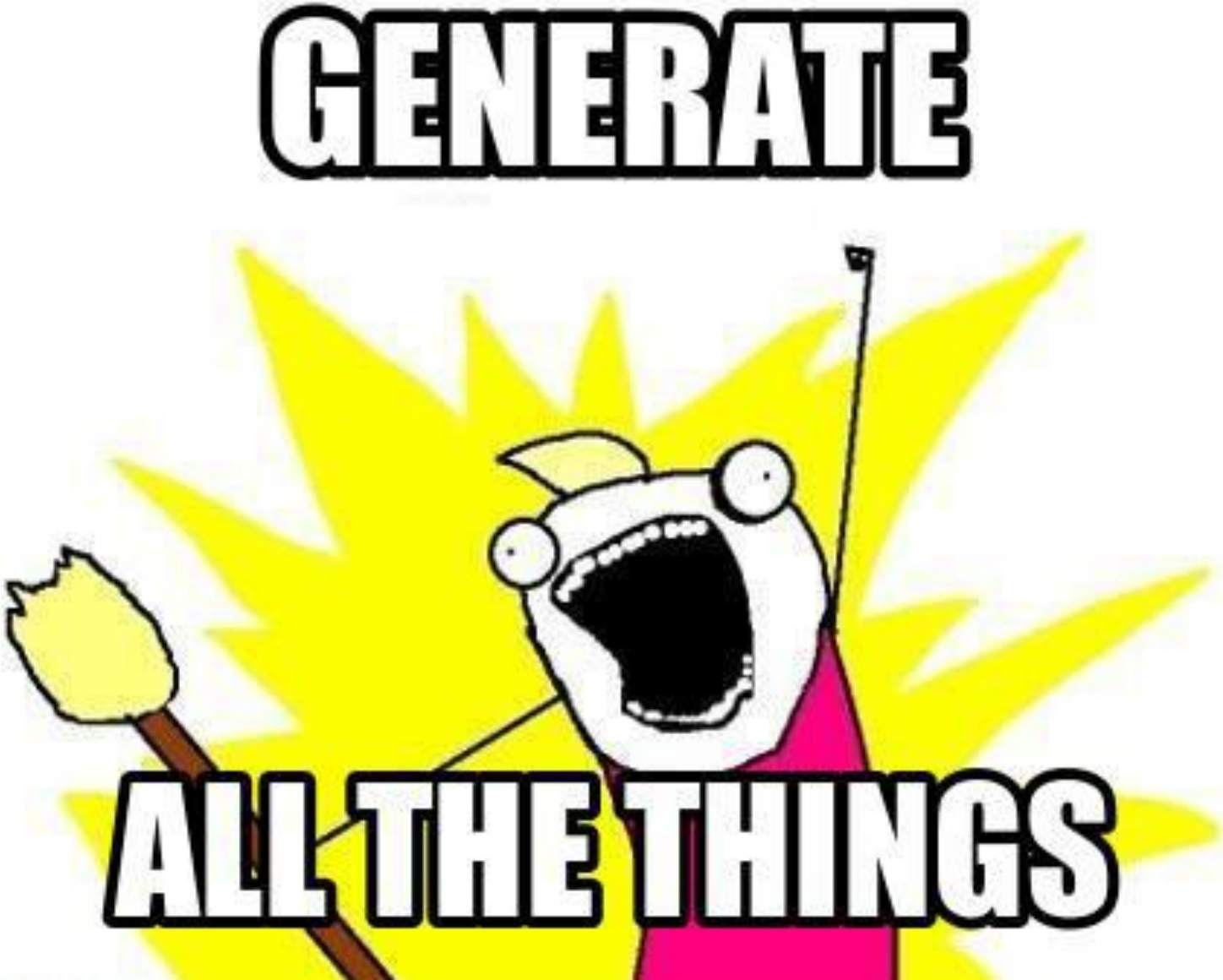

OpenEJB


APACHE WICKET


APACHE
SHIRO


OpenJDK


GRAILS

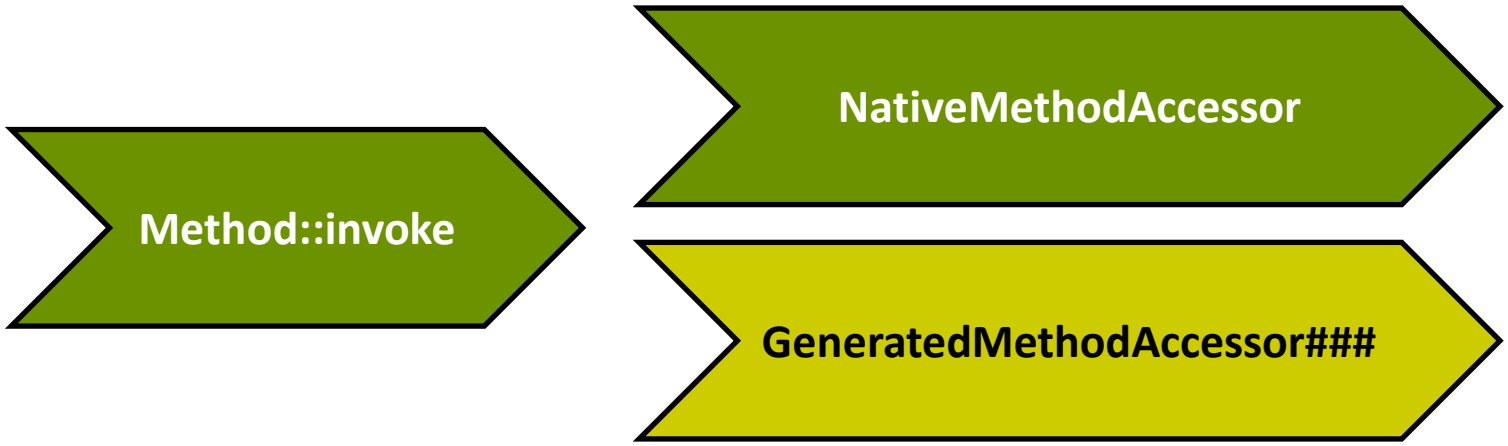

GENERATE
ALL THE THINGS

The performance myth.

There is no point in “byte code optimization”.

```
int compute() {
  return i * ConstantHolder.value;
}
```

It’s not true that “reflection is slower than generated code”.




```
-Dsun.reflect.inflationThreshold=#
```

The JIT compiler knows its job pretty well. NEVER “optimize” byte code.
Never use JNI for something you could also express as byte code.
However, avoid reflective member lookup.

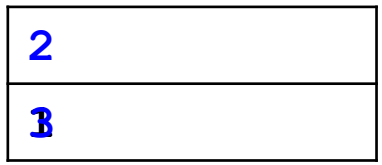
Java source code

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

Java byte code

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

operand stack



ASM

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));
```

ASM

- Byte code-level API gives full freedom
- Requires knowledge of byte code
  (stack metaphor, JVM type system)
- Requires a lot of manual work
  (stack sizes / stack map frames)
- Byte code-level APIs are not type safe
  (jeopardy of verifier errors, visitor call order)
- Byte code itself is little expressive
- Low overhead (visitor APIs)
- ASM is currently more popular than BCEL
  (used by the OpenJDK, considered as public API)
- Versioning issues for ASM (especially v3 to v4)

Byte Buddy: basic example

```
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: invocation delegation

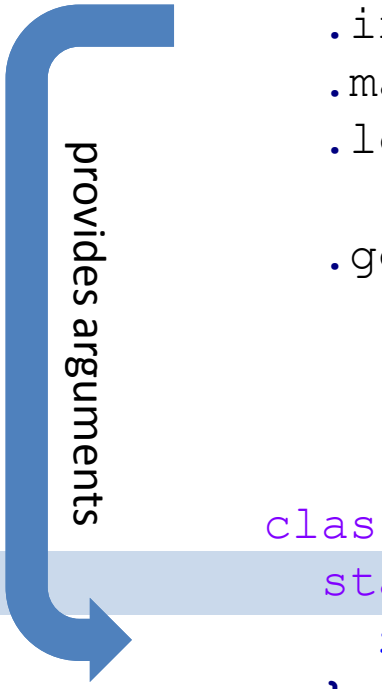
```
Class<?> dynamicType = new ByteBuddy()
  .subclass(Object.class)
  .method(named("toString"))
  .intercept(to(MyInterceptor.class))
  .make()
  .load(getClass().getClassLoader(),
        ClassLoadingStrategy.Default.WRAPPER)
  .getLoaded();
```

identifies best match

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

Byte Buddy: invocation 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: runtime HotSwap

```
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!"));
```


```
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);
}
```

Reality check: Reinvent Java?

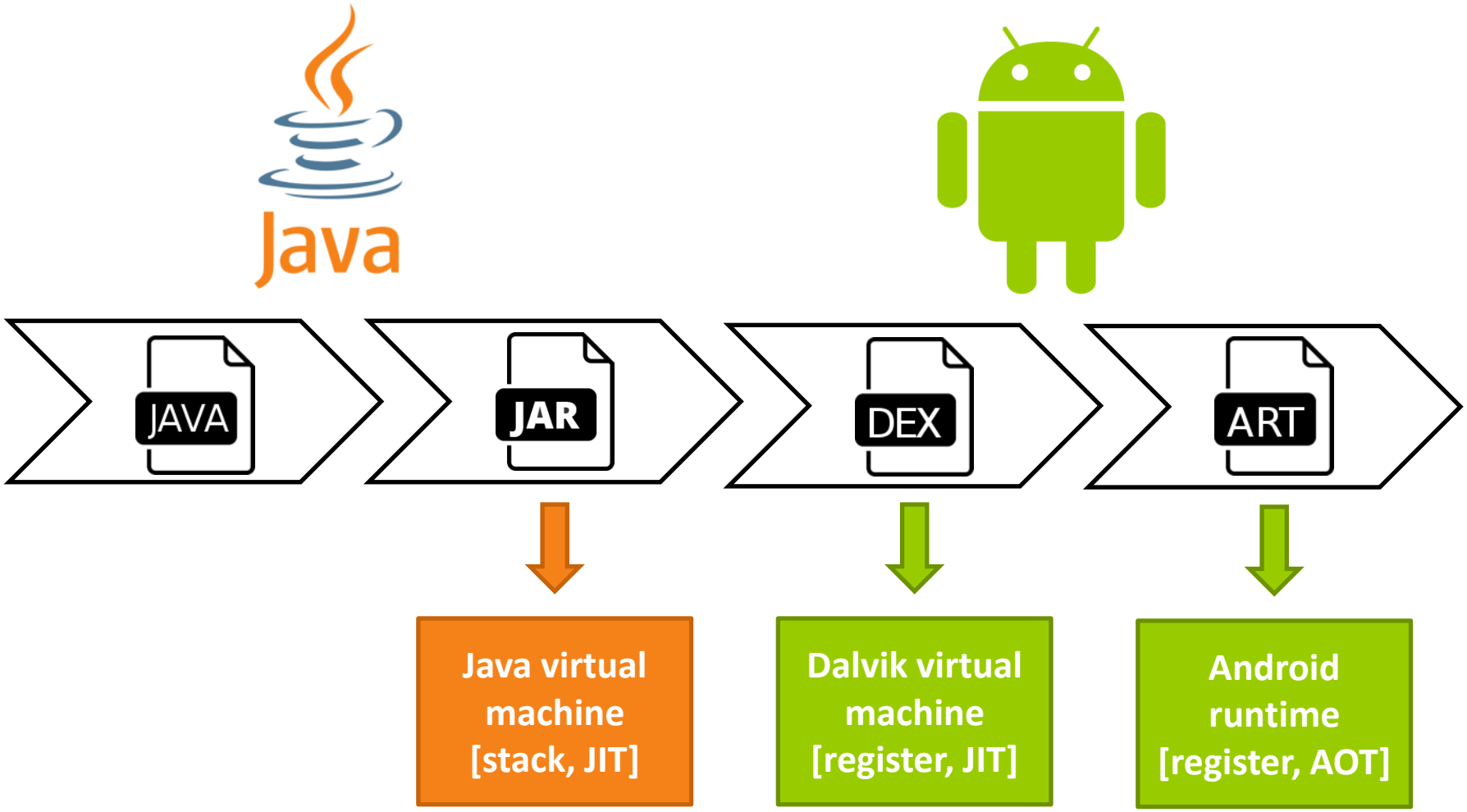
Many applications are built around a central infrastructure. A lot of code does not solve domain problems but bridged between domain and infrastructure.

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.



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

	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 *1'850.567*	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://documents4j.com
https://github.com/documents4j/documents4j

---

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