Runtime Code Generation

Bytebuddy Library

About Presenter

- Myself Tirumalesh I work for CGI.
- I am working on Java, I have 3+ years of experience in java.
- Pre CGI,worked for 2 startups ignite intelligence and echidna.
- I have experience in e-commerce and utility domains.
- I am passionate about learning new Technologies.

Why do we need Runtime Code Generation?

- Sometimes we need to intercept the method calls on a object so that we can perform different operations/ behaviour on the same method.

- Operations that can be done on the method levels are security check, logging when method is called and mocking the behaviors, etc.

- Popular frameworks like:
  - Spring security
  - Hibernate
  - Mockito (Junit framework)

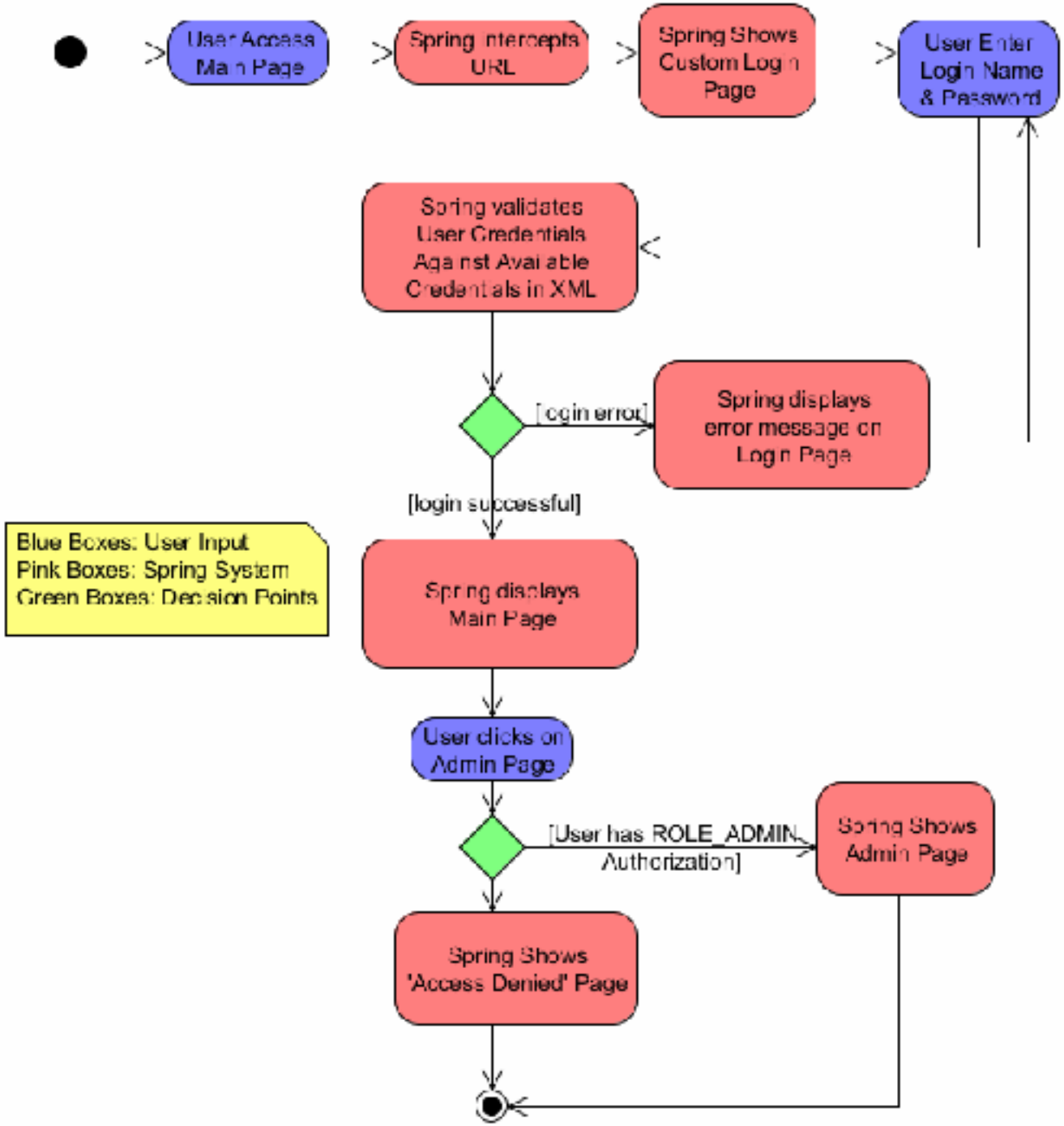
  Uses Runtime code generation.

Overview of Spring Security

- Provides Enterprise-Level **Authentication** and **Authorization** services.

- Authentication is based on implementation of GrantedAuthority interface.
  Usually “ROLE_USER”,” ROLE_ADMIN”,etc..

- Authorization is based on Access Control List.


User Access Main Page
Spring intercepts URL
Spring Shows Custom Login Page
User Enter Login Name & Password
Spring validates User Credentials Against Available Credentials in XML
[login error]
Spring displays error message on Login Page
[login successful]
Blue Boxes: User Input
Pink Boxes: Spring System
Green Boxes: Decision Points
Spring displays Main Page
User clicks on Admin Page
[User has ROLE_ADMIN Authorization]
Spring Shows Admin Page
Spring Shows 'Access Denied' Page

What’s the challenge?

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

Annotation

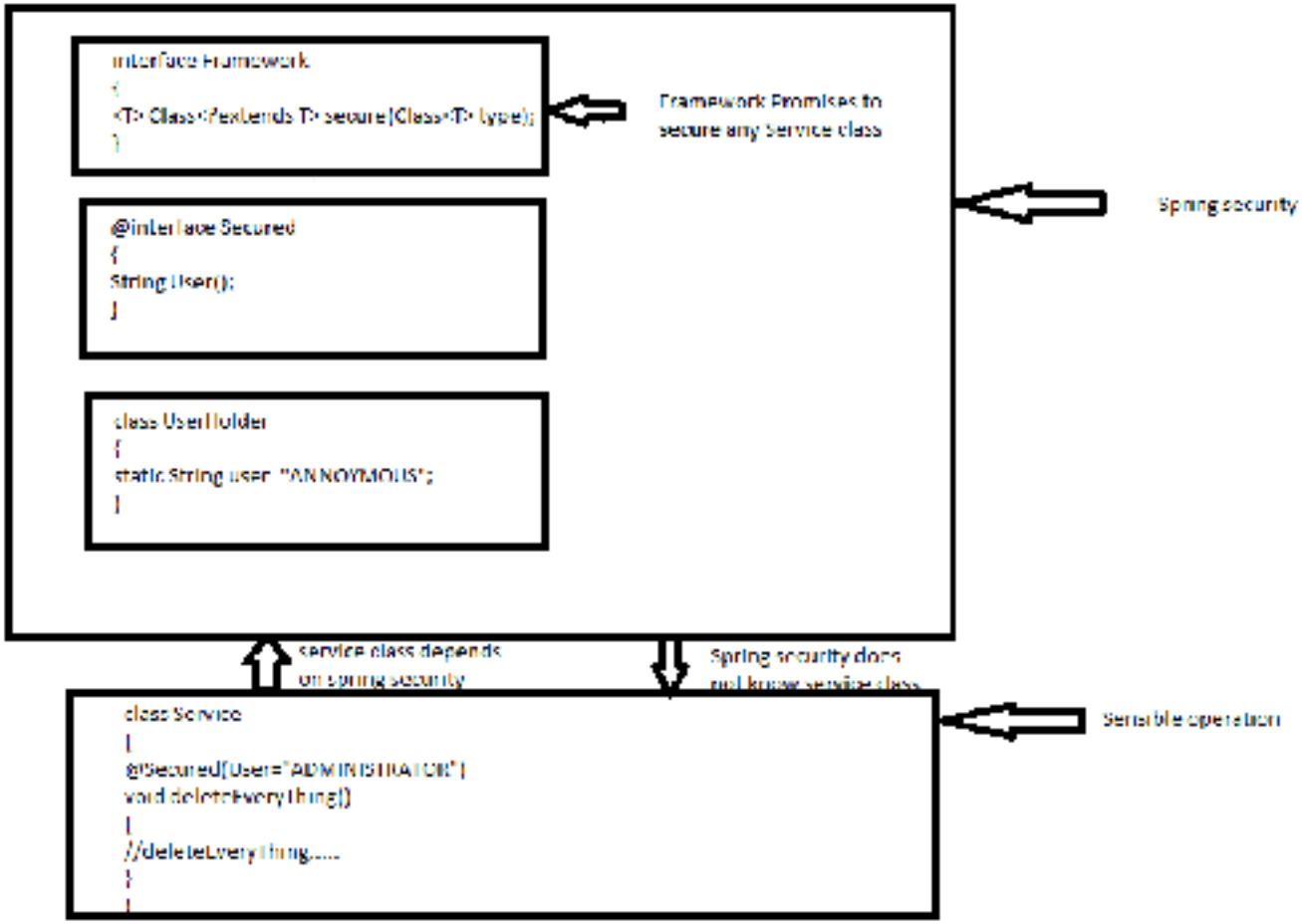
Spring security

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

Holds the type of user who logged in

```
class Service
{
@Secured(User="ADMINISTRATOR")
void deleteEverything()
{
//deleteEverything.....
}
}
```

Sensible operation


interface Framework
{
<T> Class<?extends T> secure(Class<T> type);
}
Framework Promises to secure any Service class
@interface Secured
{
String User();
}
class UserHolder
{
static String user "ANNONYMOUS";
}
Spring security
Service class depends on spring security
Spring security does not know service class
class Service
{
@Secured(User="ADMINISTRATOR")
void deleteEveryThing()
{
//deleteEveryThing.....
}
}
Sensible operation

What are the issues in the previous code?

- Secured Annotation will not do anything and they are not cared by the JVM at runtime.
- At runtime both jar files(Spring security and Service class) will ends on the same class path.
- So How service class can be linked to the spring security so that all methods with the @secured can be executed only by the authorized users.

Runtime code generation for the rescue

- Since spring security uses internally runtime code generation, it will extend the service class and overrides the methods.
- If the user is valid i.e. if the user is admin then spring security will allow the execution of the method otherwise it will throw the exception.
- When the user check criteria passes then it will call the super method.

Spring security uses RTC library

```
class SecuredService extends Service
{
@Override
void deleteEveryThing()
{
if("Admin".equals(UserHolder.user))
{
throw new IllegalStateException("Invalid user");
}
super.deleteEveryThing();
}
}
```

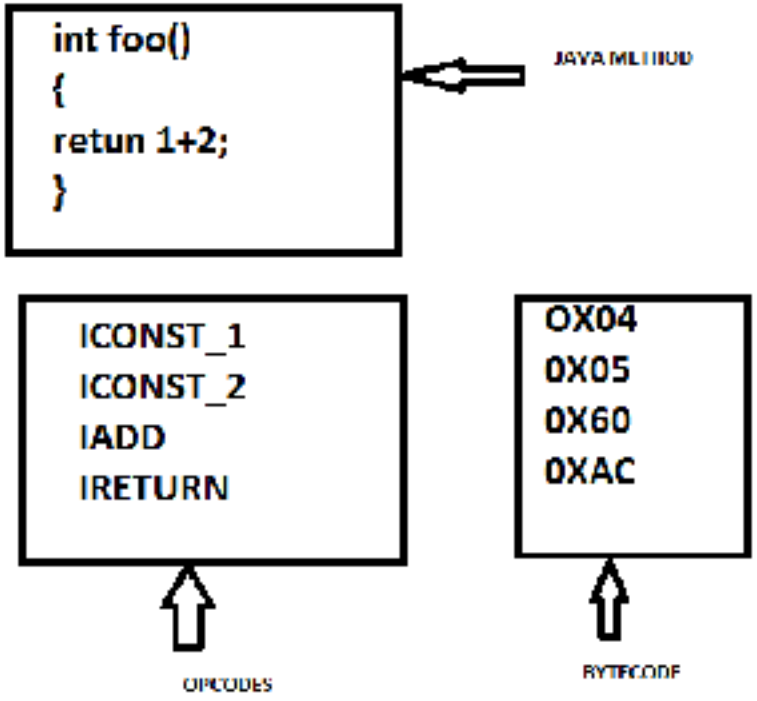
SecuredService class rebases Service class using run time code generation

```
class Service
{
@Secured(User="Admin")
void deleteEveryThing()
{
//deleteEveryThing.....
}
}
```

Libraries for Run time code generation

- Java Proxies
- Cglib
- Javassist
- **ByteBuddy**

How to do bytecode generation?



ASM

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

Drawbacks of ASM

- Requires knowledge of byte code.

  (stack metaphor, JVM type system)

- Requires a lot of manual work.

  (stack sizes/stack frames)

- Bytecode level apis are not safe.

  (jeopardy of verifier errors/ visitor call order)

- ASM codes are error prone.

Why not reflection API?

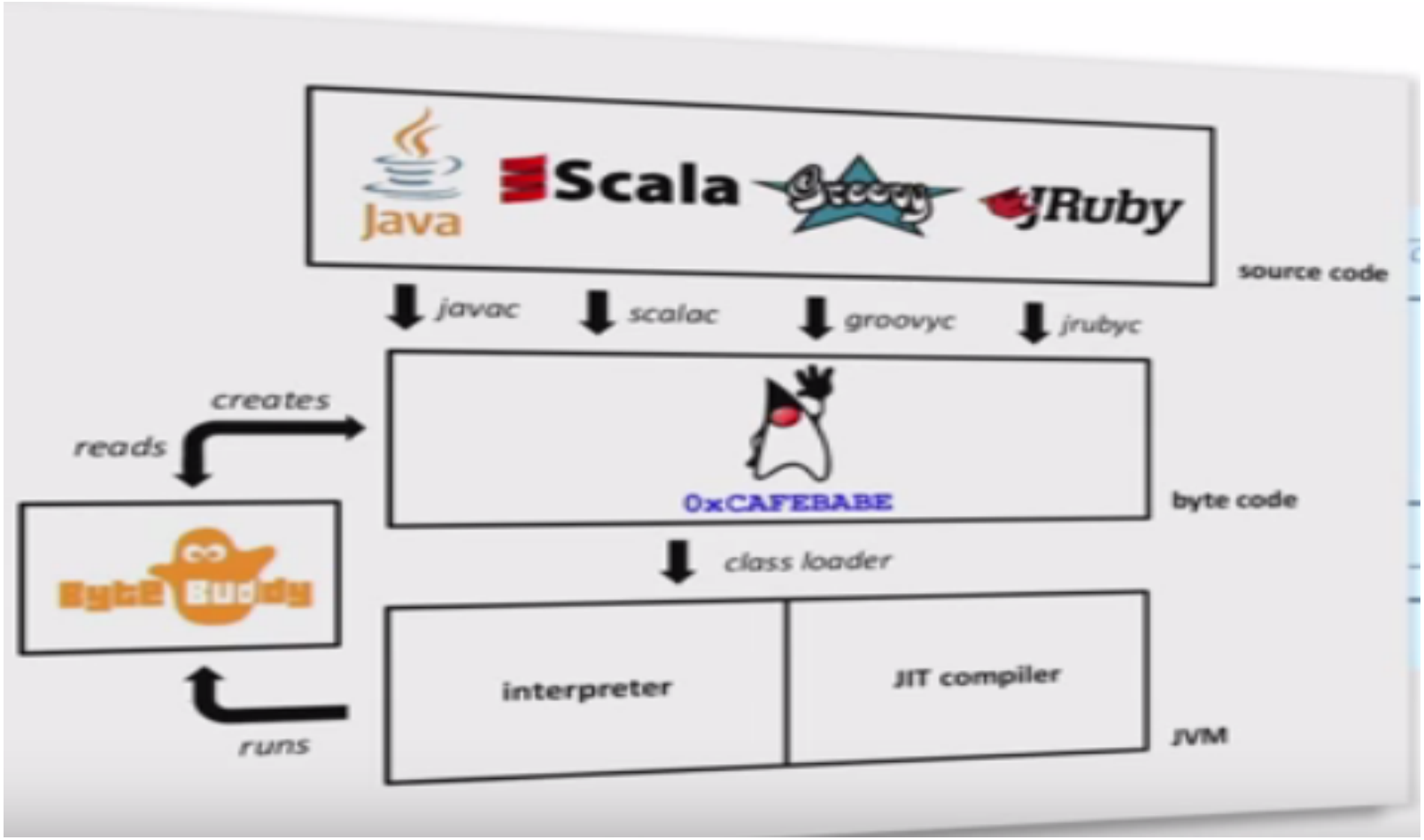
```
class Class
{
Method getDeclaredMethod(String name,Class<?>….parameterTypes)
throws NoSuchMethodException,SecurityException;

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

Drawback of reflection api!

- Reflection can also do the run time code generation by calling the required methods,annotations,parameter types,arguments.
  - **Drawbacks of reflection api:-**
- Reflection is Untype safe.
- Reflection forces to throw checked type exceptions.
- Reflection doesn’t have feature for **sub classing** and **redefining the class**.
- Lot of security checks are involved when we call “getDeclared()” method which gives less performance.

- Byte Buddy is a code generation and manipulation library for creating and modifying Java classes during the runtime of a Java application and without the help of a compiler.
- It is developed by a software consultant Ralph WinterHalter.
- Performance of ByteBuddy is much better when compared to other libraries.
- Frameworks like Spring security,Hibernate and Mockito are switched from cglib to ByteBuddy.


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


spring
HIBERNATE
mockito
Guice
eclipselink
play
Clover
OpenEJB
APACHE WICKET
APACHE SHIRO
OpenJDK
GRAILS

Performance: library comparison

	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

DEMO 1….
Method intercept

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

DEMO 2…
Method Delegation…

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

Class MyInterceptor
{
static String intercept()
{
return “Hello World”;
}
}
```

DEMO 3…
@Origin if we want to know the method origin

```
Class<?> dynamicType = new ByteBuddy()
.subclass(Object.class)
.intercept(to(MyInterceptor.class))
.make()
.load(getClass().getClassLoader(),ClassLoadingStrategy.Default.WRAPP
ER)
.getLoaded();

Class MyInterceptor
{
static String intercept(@Origin Method m)
{
return “Hello World From”+m.getName();
}
}
```

Different annotations of bytebuddy

@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.

DEMO 4…
redefining a class

```
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!”));
```

ByteBuddyAgent

```
class SecurityAgent {
public static void premain(String arg, Instrumentation inst)
{
new
AgentBuilder.Default() .type(ElementMatchers.any()) .transf
orm((builder, type) ->
builder .method(ElementMatchers.isAnnotatedBy(Secured.cl
ass) .intercept(MethodDelegation.to(SecurityInterceptor.cla
ss) .andThen(SuperMethodCall.INSTANCE)))) .installOn(inst);
 }
 }
```

Mockito usage of ByteBuddy

- Mockito is a junit framework which is used for unit testing.
- Mocking is the act of removing external dependencies from a unit test in order to create a controlled environment around it. Typically, we mock all other classes that interact with the class that we want to test. Common targets for mocking are:
- Database connections,
- Web services,
- Classes that are slow ,
- Classes with side effects, and
- Classes with non-deterministic behavior.

Reinvent Java!!!!

- Just to log a method or just to have security on a method why we need to use whole **heavy integration framework** such as spring.
- These frameworks are requires you to code against framework rather than code against plain java.
- Indeed these framework helps a lot but for just to log a method or to have security on a method why we need to have the entire framework/jar in our application this makes application to be slower in the production.

WHY NOT TO USE AGENTS FOR THESE INSTEAD OF USING WHOLE HEAVY FRAEWORK!!!!!

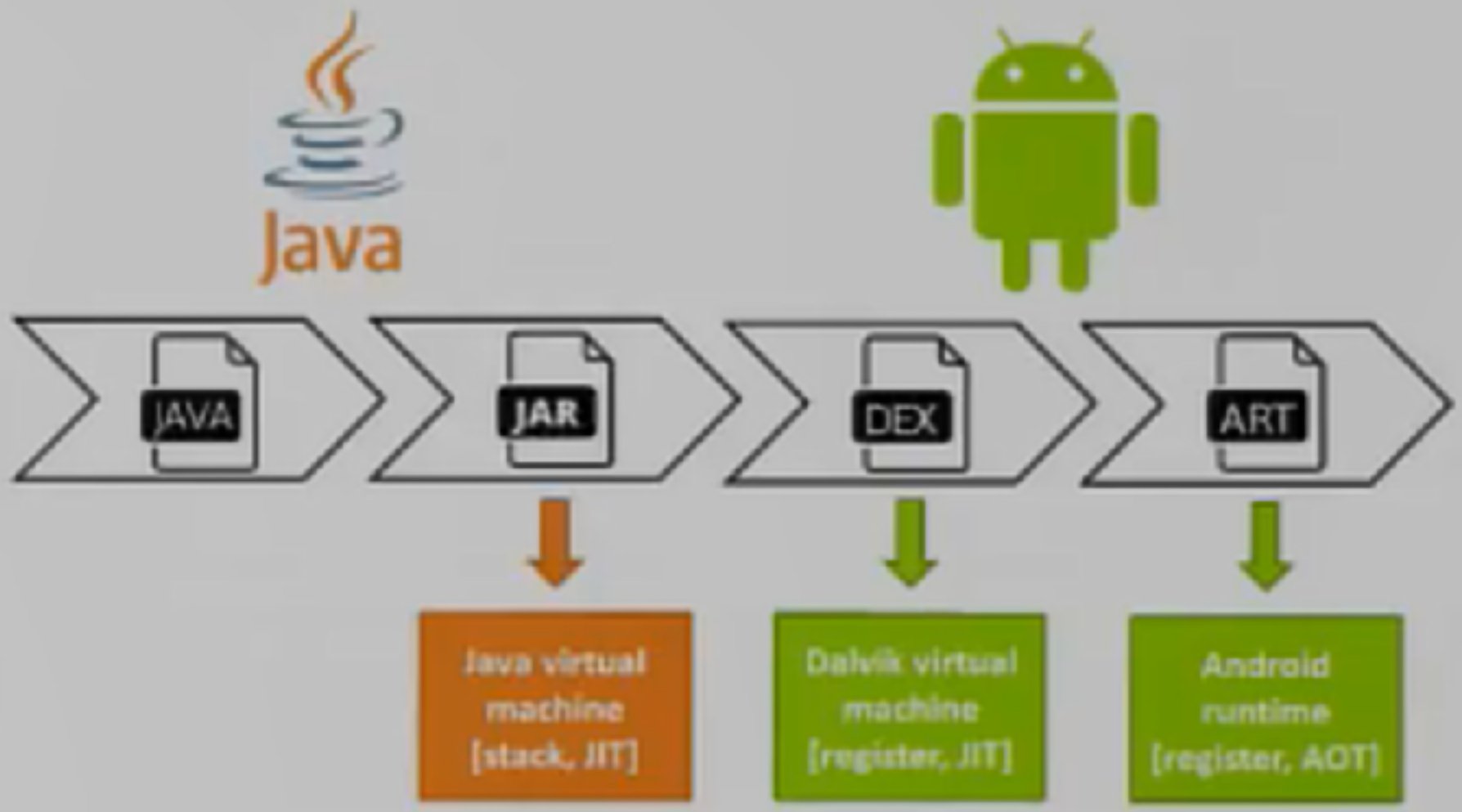
->It is not so hard to use agents.

->We can write specific logging library.

->We can write specific security library.

- We don't to need other frameworks to get integrate into our application and subclass our classes for less/least functionalities.

Android makes things more complicated.



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

Limitations of Byte Buddy

- Using this library we can’t add new methods.
- Using this library we can’t add new fields.
- Android Run Time(ART) is not compatible with byte buddy.
- Dalvik byte code is not compatible with byte buddy.

References

- http://bytebuddy.net/#/
- https://www.youtube.com/watch?v=jo1v8csBorw
- https://zeroturnaround.com/rebellabs/how-to-make-java-more-dynamic-with-runtime-code-generation/

Thank You…