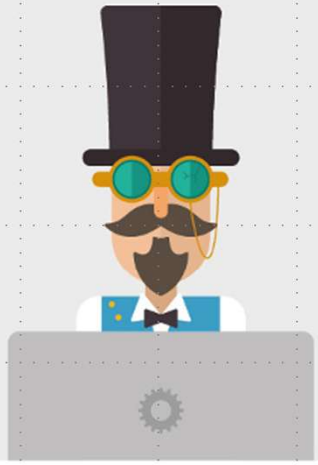
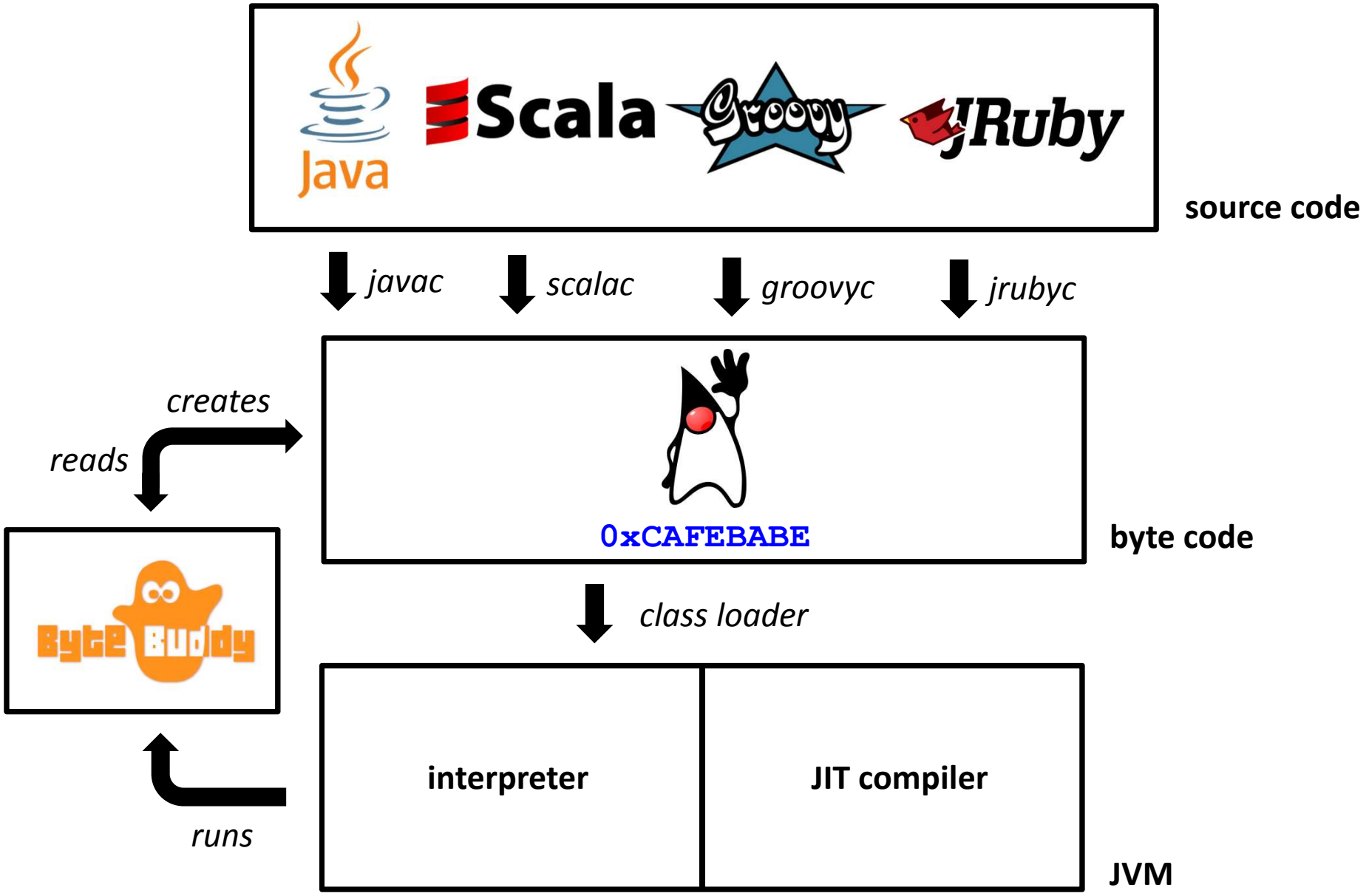
Understanding Java byte code and the class file format



geecon


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

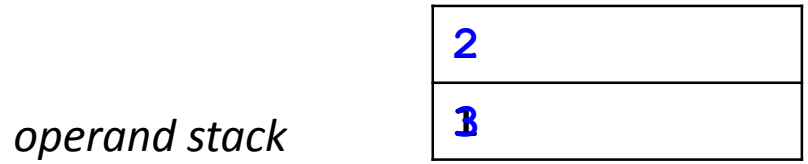
```
0xB1 RETURN
```

source code

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

byte code

```
0x4  ICONST_1
0x5  ICONST_2
0x6  IADD
0xAC IRETURN
```



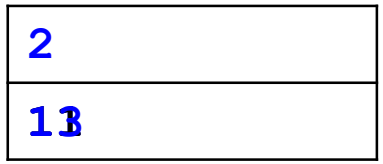
source code

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

byte code

```
0x10 0x0B   BIPUSH 11
0x05        ICONST_2
0x60        IADD
0xAC        IRETURN
```

operand stack



source code

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

byte code

```
0x03 ILOAD_1
0x04 ICONST_1
0x60 IADD
0xAC IRETURN
```

operand stack

1
i + 1

local variable array

1 : i
0 : this

source code

```
int foo(int i) {
  int i2 = i + 1;
  return i2;
}
```

byte code

```
0x00 ILOAD_1
0x01 ICONST_1
0x02 IADD
0x03 ISTORE_2
0x04 ILOAD_2
0x05 IRETURN
```

operand stack

`1`
`i + 1`

local variable array

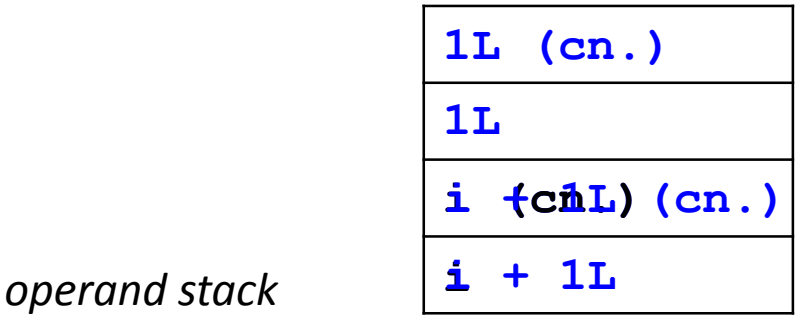
`2 : i + 1`
`1 : i`
`0 : this`

source code

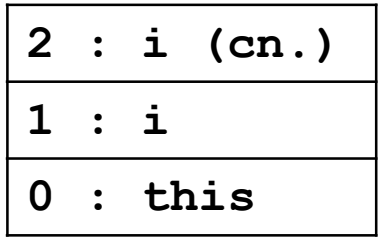
```
long foo(long i) {
  return i + 1L;
}
```

byte code

```
0: LLOAD_1
1: LCONST_1
2: LADD
3: LRETURN
```



operand stack



local variable array

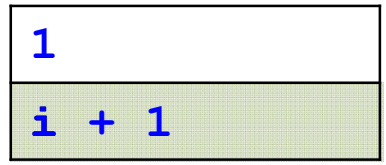
source code

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

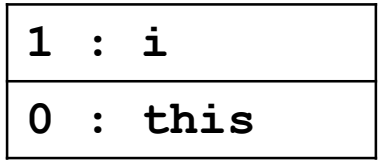
byte code

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

operand stack



local variable array

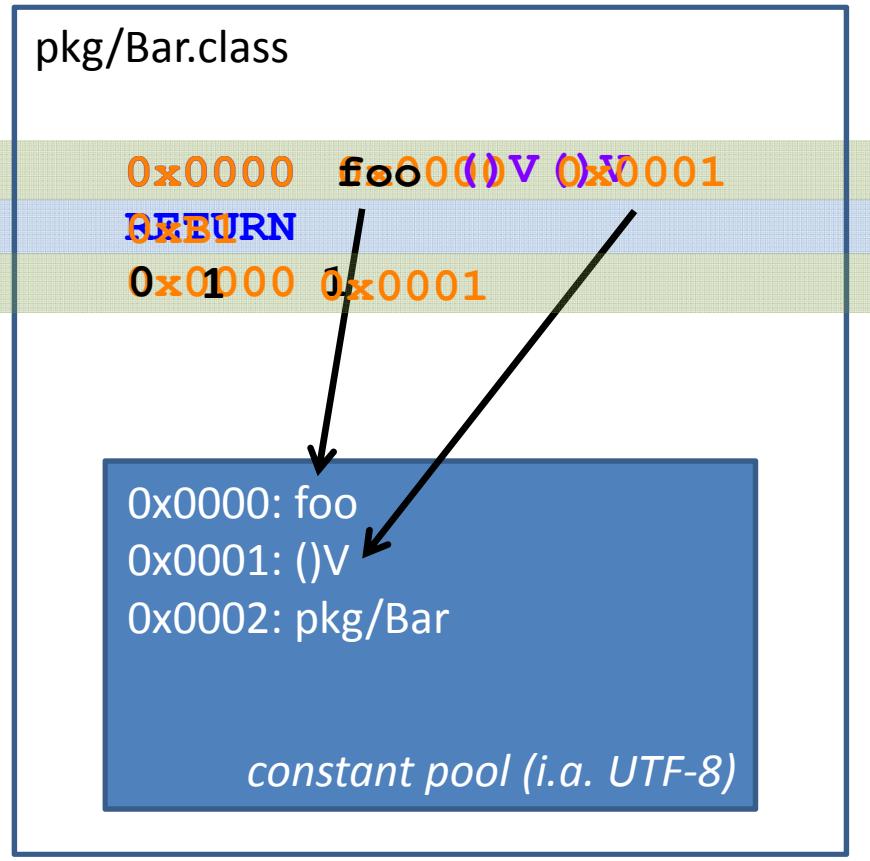


Java type	JVM type (non-array)	JVM descriptor	stack slots
boolean	I	Z	1
byte		B	1
short		S	1
char		C	1
int		I	1
long	L	J	2
float	F	F	1
double	D	D	2
void	-	V	0
java.lang.Object	A	Ljava/lang/Object;	1

source code

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

byte code



operand stack ////////////////

local variable array `0 : this`

source code

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

byte code

```
0x0009 foo()V0x0001
RETURN
0x0000 0x0000
```

constant pool

Modifier	Value
static	0x0008
public	0x0001

operand stack ////////////////

local variable array ////////////////

source code

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

byte code

```
0x00 ALOAD_0
0x01 ILOAD_1
0x02 ICONST_1
0x03 IADD
0x04 INVOKEVIRTUAL pkg/Bar.foo(I)I
0x07 IRETURN
```

constant pool

operand stack

1
i + 1
this.foo(i + 1)

local variable array

1 : i
0 : this

source code

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

byte code

```
0x0A  ILOAD_0
0x0B  ICONST_1
0x0C  IADD
0xB8  0x0002  0x0001  0x0000  INVOKESTATIC pkg/Bar foo (I)I
0xAC  IRETURN
```

operand stack

1
foo(i + 1)

local variable array

0 : i

source code

```
package pkg;
class Bar {
  @Override
  public String toString() {
    return super.toString();
  }
}
```

byte code

```
0x0A ALOAD_0
0xB7 INVOKESPECIAL java/lang/Object
                   toString
                   ()Ljava/lang/String;
0xAC ARETURN
```

0x0004
0x0005
0x0006

constant pool

operand stack

```
this.toString()
```

local variable array

```
0 : this
```

INVOKESTATIC *pkg/Bar foo ()V*
Invokes a static method.

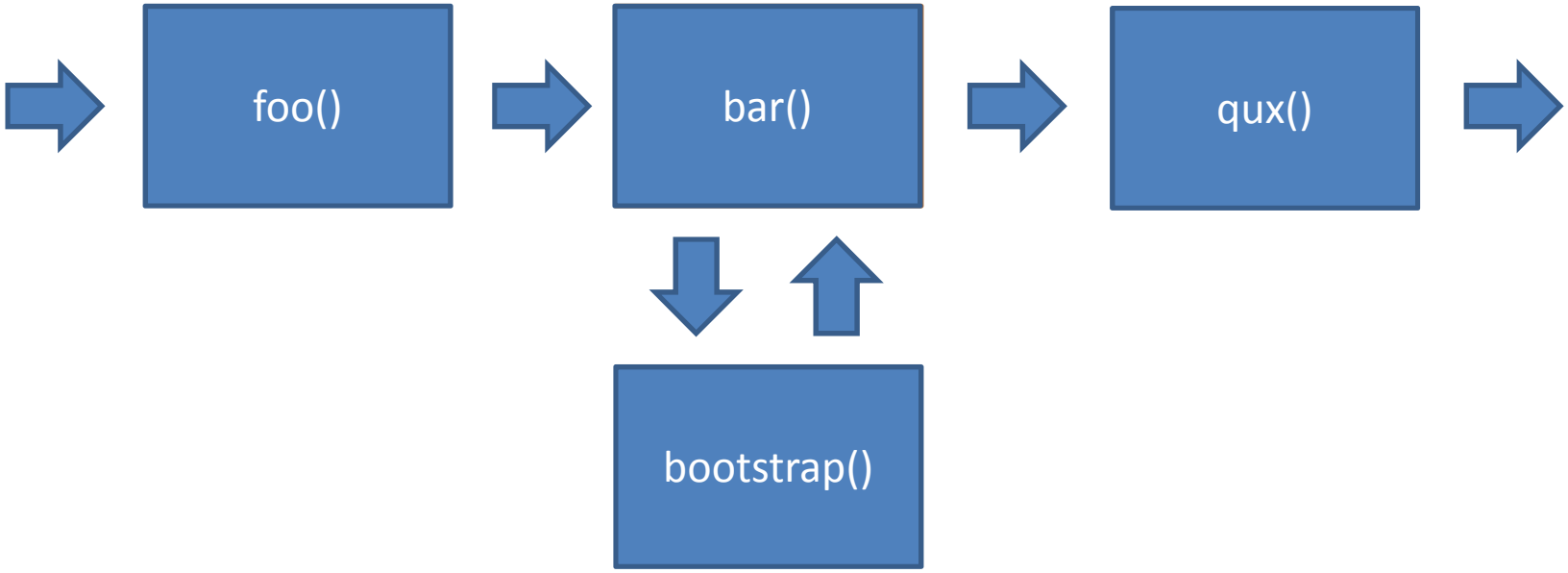
INVOKEVIRTUAL *pkg/Bar foo ()V*
Invokes the most-specific version of an inherited method on a non-interface class.

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.)

```
void foo() {
  bar(); // invokedynamic
}
```



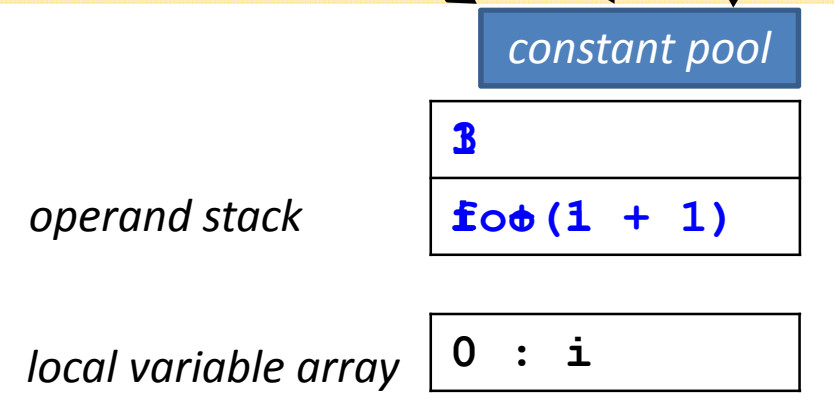
In theory, this could be achieved by using reflection. However, using invokedynamic offers a more native approach. This is especially important when dealing with primitive types (double boxing).
Used to implement lambda expressions, more important for dynamic languages.

source code

```
package pkg;
class Bar {
  static int foo(int i) {
    if (i > 3) {
      return 0;
    } else {
      return foo(i + 1);
    }
  }
}
```

byte code

```
0000: ILOAD_0
0001: ICONST_3
0002: IF_ICMPLE 0008
0005: ICONST_0
0006: IRETURN
0008: ILOAD_0
0009: ICONST_1
000A: IADD
000B: INVOKESTATIC pkg/Bar foo (I)I
000E: IRETURN
```



```
java.lang.VerifyError: Inconsistent stackmap frames at branch
                       target XXX in method pkg.Bar.foo(I)I
                       at offset YYY
```

The verifier rejects malformed class files and byte code:

- compliance to class file format
- type safety
- access violations

type inferencing
(until Java 5, failover for Java 6)

Follow any possible path of jump instructions: Assert the consistency of the contents of the *operand stack* and *local variable* array for any possible path.

Inference might require several iterations over a method‘s byte code.

Legacy: -XX:-UseSplitVerifier

type checking
(from Java 6)

Require the otherwise infered information about the contents of the *operand stack* and the *local variable array* to be embedded in the code as *stack map frames* at any target of a jump instruction.

Checking requires exactly one iteration over a method‘s byte code.

source code

```
int foo() {
  try {
    return 1;
  } catch (Exception e) {
    return 0;
  }
}
```

byte code

```
0000: ICONST_1
0001: IRETURN
0002: ICONST_0
0003: IRETURN
```

exception table:

<from> <to> <catch> java/lang/Exception

constant pool

source code

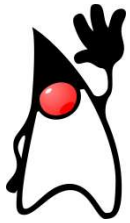
byte code

discreetly overlooked:
attributes
(i.a. annotations)

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

```
0xCAFEBABE
0x0052
0x0004
0000: UTF-8[foo]
0001: UTF-8[()V]
0002: UTF-8[pkg/Bar]
0003: UTF-8[java/lang/Object]
0x0000
0x0002
0x0003
0x0000
0x0000
0x0000 0x0000 ()V 0x0001
RETURN
0x0000 0x0001
```

```
class SecuredService extends Service {
  @Override
  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...
  }
}
```

```
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!"));
```

http://rafael.codes
@rafaelcodes

---

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

---

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