An Introduction to Working with JVM Bytecode

Charles Forsythe

Why Learn Bytecode: The JVM

Here's Some Annoying Java Code

```
package pkg1;

public class StringHolder {
    private String string;

    public void setString(String s) {
        string = s;
    }

    public String getString() {
        return s;
    }
}
```

What if you could just do this?

```
>> pkg1

StringHolder

string : String
```

Workshop: Build a DSL Compiler

We’re going to build a simple compiler for a this simple language.

What Are We Going To Cover

- The operation of the Java Virtual Machine
- The way that JVM code is packaged (classes)
- Java 1-6

Important Things We Won't Cover

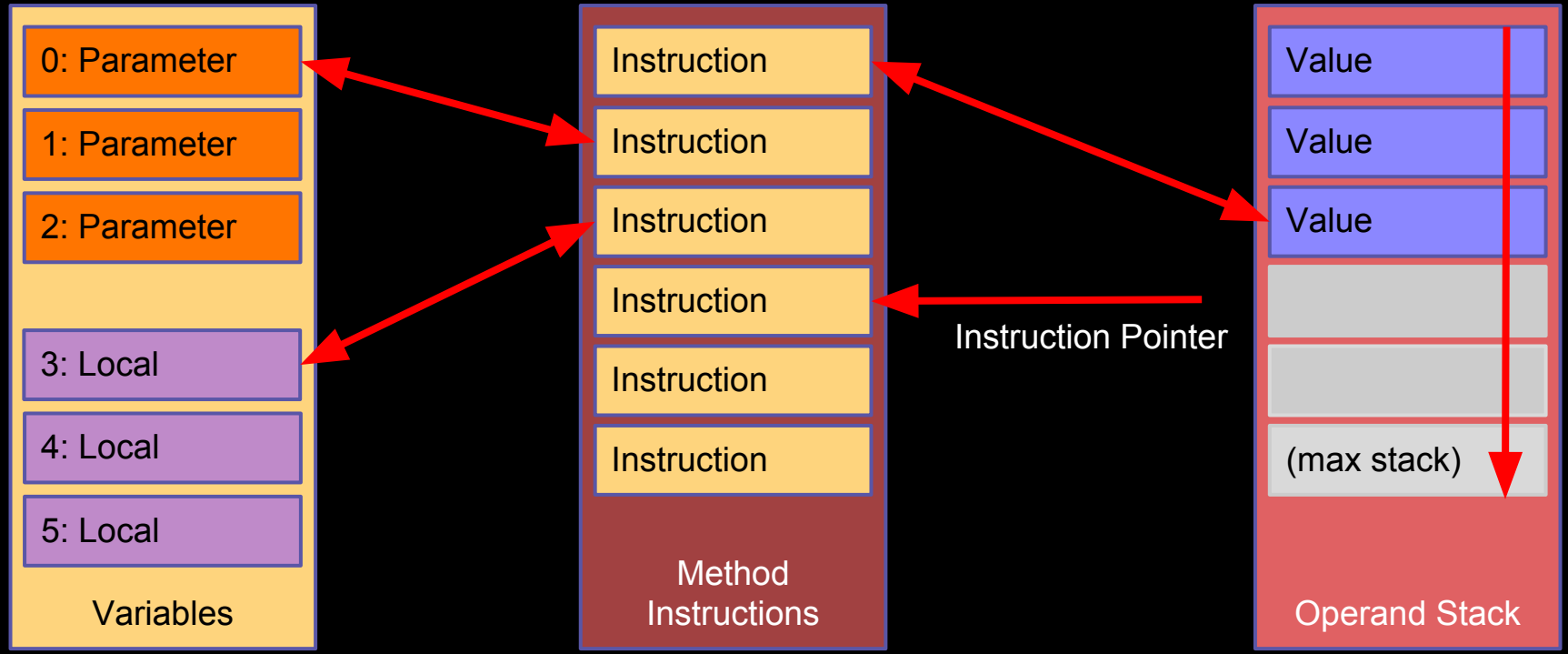
- How exceptions are caught
- New cool JVM 7 (and Java 8) stuff*
- A lot of attributes that are used by tools and reflection
- Tools

* Mostly.

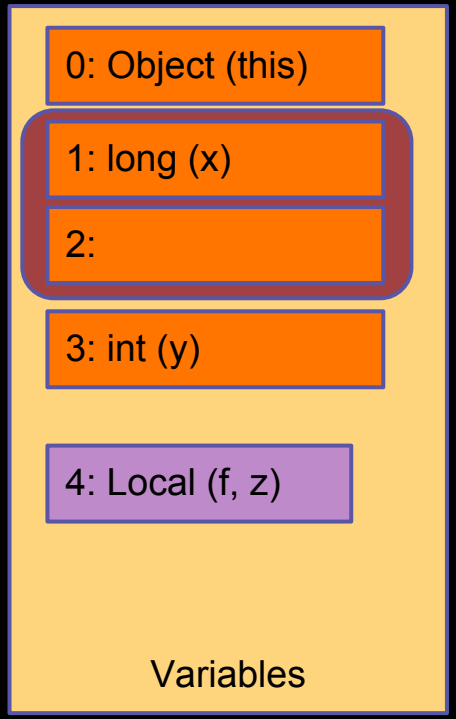
Fundamentals: JVM Basics

Understanding the way bytecode is packaged and executed.

Anatomy of the JVM: Stack-Based



Anatomy of the JVM: Type Safe




```
void method(long x, int y)
{
    {
        float f = y;
    }

    {
        boolean z = x > 0;
    }
}
```

```
// float f = y;
    iload 3  // Load int from slot 3 (y)
    i2f      // Convert top of stack to float
    fstore 4 // Store float in slot 4 (f)
```

Types Handled by JVM Instructions

Integer	byte*, short*, char*, int, boolean *mostly
Float	float
Double	double
Long	long
Object Reference	java.lang.Object Array

The JVM Likes Integers

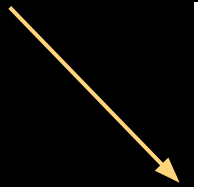
byte, char, and short are integers unless stored in arrays.

boolean is not distinguished from integer.

i2b	Truncate integer to 8-bit signed
i2c	Truncate integer to 16-bit unsigned
i2s	Truncate integer to 16-bit signed

Java Types in the JVM

```
boolean foo(byte b, char c, int i)  {
    return i > (b + c);
}
```



```
 0: iload_3
 1: iload_1
 2: iload_2
 3: iadd
 4: if_icmple 11
 7: iconst_1
 8: goto 12
11: iconst_0
12: ireturn
```

```
int foo(int b, int c, int i) {
    return (i > (b + c)) ? 1 : 0;
}
```

Type Descriptors

void	V (can only be a return type)
boolean	Z
byte	B
short	S
char	C
int	I
long	J
float	F
double	D
Object	Ljava/lang/Object;
Array	[D = double[] [[Ljava/lang/String; = java.lang.String[][]

Example Method Descriptors

```
boolean foo(byte b, char c, int i)  {
    return i > (b + c);
}
```

(BCI)Z

```
int foo(int b, int c, int i) {
    return (i > (b + c)) ? 1 : 0;
}
```

(III)I

Descriptors and Signatures

Example of a type description:

```
List<String>
```

Descriptor:

```
Ljava/util/List;
```

Signature:

```
Ljava/util/List<Ljava/lang/String;>;
```

Descriptors and Signatures

Example of a type description:

```
Map<Integer, String>
```

Descriptor:

```
Ljava/util/Map;
```

Signature:

```
Ljava/util/Map<Ljava/lang/Integer;Ljava/lang/String;>;
```

Descriptors and Signatures

Class declaration with type parameters:

```
public class MyMap<K, V> extends AbstractMap
```

Class superclass:

```
Ljava/util/AbstractMap;
```

Class signature:

```
<K:Ljava/lang/Object;V:Ljava/lang/Object;>Ljava/util/AbstractMap;
```

Type parameter reference and its signature:

```
List<K>
Ljava/util/List<TK;>;
```

Descriptors and Signatures

Class declaration with type parameters:

```
public class Foo<T extends InputStream & Closeable, X extends Serializable>
```

Class superclass:

```
    Ljava/lang/Object;
```

Class signature:

```
<T:Ljava/io/InputStream;:Ljava.io.Closeable;,X::Ljava.io.Serializable;
>Ljava/lang/Object;
```

Erasure types:

```
    T    java/io/InputStream
    X    java/io/Serializable
```

Class File: Packaging the Code

Getting code into the JVM to be executed requires packaging it in the correct binary format.

Class File Format

```
ClassFile {
      u4              magic
      u2              minor_version
      u2              major_version
      u2              constant_pool_count
      constant[]      constant_pool
      u2              access_flags
      u2              this_class
      u2              super_class
      u2              interface_count
      u2[]            interfaces
      u2              fields_count
      field[]         fields
      u2              methods_count
      method[]        methods
      u2              attribute_count
      attribute[]     attributes
}
```

u2	*unsigned, 2-byte int*
u4	*unsigned, 4-byte int*
constant	*Constant Pool Item*
field	*Field Definition*
method	*Method Definition*
attribute	*Attribute Definition*

Magic and Version

- Magic is `0xCAFEBABE`
- Major Version is Java version + 44
  - Java 1.1 = 45
  - Java 8 = 52
- Minor version is 0 (except for Java 1.1 where is was 3)

Class File Format: Attributes

```
ClassFile {
    u4              magic
    u2              minor_version
    u2              major_version
    u2              constant_pool_count
    method[]        methods
    u2              attribute_count
    attribute[]     attributes
}
```

Attribute structure:

```
attribute {
    u2    attribute_name_index;
    u4    attribute_length;
    byte  data[attribute_length];
}
```

Class File Format: Constant Pool

```
ClassFile {
    u4              magic
    u2              minor_version
    u2              major_version
    u2              constant_pool_count
    constant[]      constant_pool
    u2              access_flags
```

The constant pool array is referenced using 1-based indexes.
The constant pool count is 1 larger than the size of the array.

For example:

Constant Pool Size:	20
Constant Pool Indexes:	1-20
Constant Pool Count:	21

```
}
```

Constant Pool: Examples

```
#52   Integer_info = 60000
…
#75   MethodRef_info = boolean MyClass.equals(Object)
```

```
// push int 60000 onto operand stack
ldc #52              // Bytecode: 18 52
…
// push parameter and (myClass) target object
// call target.equals(param)
invokevirtual #75   // Bytecode: 182 0 75
```

LDC-able Constants

JVM 1-4	Integer, Float, Long, Double, String
JVM 5	Class `java.lang.Class`

JVM 7	MethodHandle, MethodType
	`java.lang.invoke.MethodHandle`
	`java.lang.invoke.MethodType`

Non-LDC-able Constants

- Field_info
- Method_info
- InterfaceMethod_info

- UTF8_info

Binary Qualified Names

Class `foo.bar.baz` becomes `foo/bar/baz*`

Each element in the *qualified name,* foo, bar, and baz is an *unqualified name.* Unqualified names are also used for methods and fields.

* "...for historical reasons."
  Java Virtual Machine Specification

Binary Unqualified Name

Forbidden Characters: `.` `;` `[` `/` `<` `>`

Special case methods: `<init>` `<clinit>`

Examples of valid qualified names:

```
perry/the/Platypus$AgentP
2/+/2/=/4
:-)/:-(/8-]
כותרת/नाम/название
```

Class File Format: Methods

```
ClassFile {
    u4              magic
    u2              minor_version
    u2              major_version

    method {
        u2 access_flags
        u2 name_index          // UTF8_info
        u2 descriptor_index    // UTF8_info
        u2 attribute count
        attribute[attribute_count]
    }

    field[]         fields
    u2              methods_count
    method[]        methods
    u2              attribute_count
    attribute[]     attributes
}
```

Uncommon Access Flags for Methods

ACC_SYNTHETIC	0x1000
ACC_BRIDGE	0x0040

```
public class A {
    public Object value() {
        return "Object";
    }
}

public class B extends A {
    public String value() {
        return "String";
    }
}
```

A.class

```
public java.lang.Object value();
    flags: ACC_PUBLIC
```

B.class

```
public java.lang.String value();
    flags: ACC_PUBLIC

public java.lang.Object value();
    flags: ACC_PUBLIC, ACC_BRIDGE, ACC_SYNTHETIC
```

Code Attribute

```
Code_attribute {
    u2   attribute_name_index
    u4   attribute_length
    u2   max_stack
    u2   max_locals
    u4   code_length
    byte code[code_length]
    u2   exception_table_length
    exception_table[exception_table_length] {
        u2 start_offset   (inclusive)
        u2 end_offset     (exclusive)
        u2 handler_offset
        u2 catch_type     (index to Class_info or 0 = any)
    }
    u2 attributes_count;
    attribute[attributes_count];
}
```

Max stack/locals

Bytecode

Exception Table

Attributes

Attributes: JVM Critical

StackMapTable

The StackMapTable describes the data types live on the local variable and operand stack at key bytecode offsets (branch targets).

- Optional in Java 6
- Required* in Java 7 *could be disabled
- Required in Java 8 (for real)

Many consider it the worst thing ever.

Code Attribute: Bytecode

```
Code_attribute {
    u2   attribute_name_index
    u4   attribute_length
    u2   max_stack
    u2   max_locals
    u4   code_length
    byte code[code_length]
    u2   exception_table_length
    exception_table[exception_table_length] {
        u2 start_offset   (inclusive)
        u2 end_offset     (exclusive)
        u2 handler_offset
        u2 catch_type     (index to Class_info or 0 = any)
    }
    u2 attributes_count;
    attribute[attributes_count];
}
```

Instruction Set: Quick Constants

Integer Constant Instructions

One-Byte		Two-Byte	Three-Byte
-1	ICONST_M1	bipush *(-128 - 127)*	sipush *(-32768 - 32767)*
0	ICONST_0		
1	ICONST_1		
2	ICONST_2		
3	ICONST_3		
4	ICONST_4		
5	ICONST_5		

Other One-Byte Constant Instructions

Float		Long		Double		Object Reference	
0.0f	FCONST_0	0L	LCONST_0	1.0d	DCONST_0	null	ACONST_NULL
1.0f	FCONST_1	1L	LCONST_1	2.0d	DCONST_1		
2.0f	FCONST_2						

Instruction Set: Constant Pool

Load Constant (LDC) Variants

`LDC` *(1-255)*	Load constant with index 1-255
`LDC_W` *(1-65536)*	Load constant with index 1-65535
`LDC2_W` *(1-65535)*	Load two-slot constant (long, double) with index 1-65535

Legal Types for LDC and LDC_W

Constant Type	*Type on stack*
`Integer_info`	`int`
`Float_info`	`float`
`String_info`	`java.lang.String`
`Class_info`	`java.lang.Class`
`MethodType_info`	`java.lang.invoke.MethodType`
`MethodHandle_info`	`java.lang.invoke.MethodHandle`

Legal Types for LDC2_W

`Long_info`	`long`
`Double_info`	`double`

Instruction Set: Create Arrays

Create Arrays

`newarray` ***`prim[n]`*** allocate array of size *n* for primitive type *prim*

`anewarray` ***`type[n]`*** allocate array of size n for object type type

```
new String[5]
  bipush 5
  anewarray Ljava/lang/String;
```

`multianewarray type[][][]` Allocate multidimensional array with initialization

```
new String[4][3][2]
  iconst_4
  iconst_3
  iconst_2
  multianewarray [[[Ljava/lang/String; 3
```

```
new boolean[10][20][]
  bipush 10
  bipush 20
  multianewarray [[[B 2
```

Arrays: Access

Access Array Elements

`aaload, aastore`	load/store Object references
`baload, bastore`	load/store byte (signed 8 bit)
`caload, castore`	load/store char (unsigned 16 bit)
`daload, dastore`	load/store double (64 bit float)
`faload, fastore`	load/store float (32 bit float)
`iaload, iastore`	load/store int (signed 32 bit)
`laload, lastore`	load/store long (signed 64 bit)
`saload, sastore`	load/store short (signed 16 bit)

Special Array Operation

`arraylength`	return the dimension of the array as an integer

invokevirtual

```
public class A {
  public String value() { return “A”;}
}

public class B extends A {
//  public String value() {
//    return “B”;
//  }
}

public class C extends B {
  public String extractValue() {
    C c = new C();
    return c.value();
  }
}
```

Constant pool:

```
#4 = Methodref C.value:()String;
```

```
0: new #2 // class C
3: dup
4: invokespecial #3 // C."<init>"
7: astore_1
8: aload_1
9: invokevirtual #4 // C.value()
12: astore_2
13: aload_2
14: areturn
```

invokeinterface

```
public interface X {
     String value();
}

public class Y {
     public String value(X val) {
          return val.value();
     }
}
```

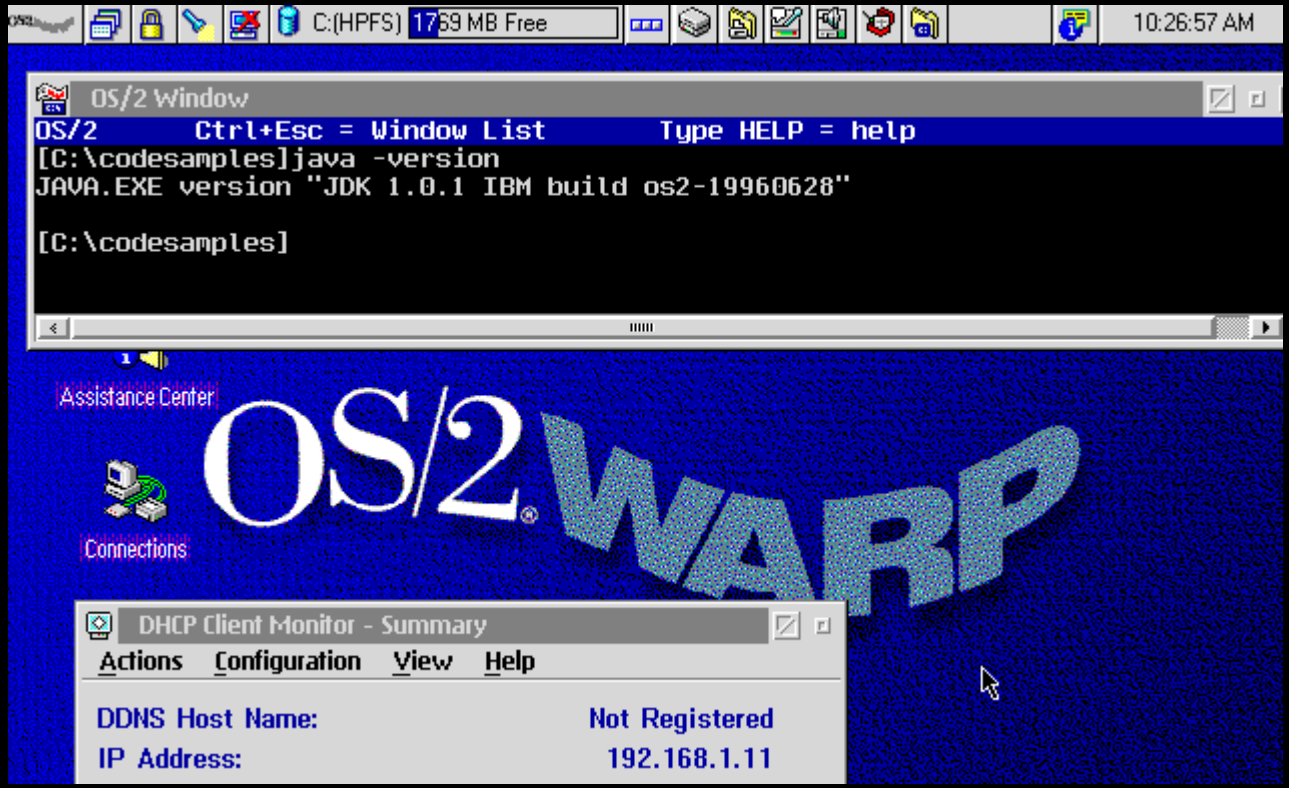
```
Constant pool:
#2 = InterfaceMethodref X.value:()
String;

Y.value(X)
     0: aload_1
     1: invokeinterface #2,  1
     6: areturn
```

invokenonvirtual Java 1.0

```
public class A {
  private String inner() {return “A”;}
  public String value() {
    return inner();
  }
}

public class B extends A {
  private String inner() {return “B”;}
}

public class C extends B {
    public String value() {
        return super.value();
    }
}
```

```
Compiled from C.java
public class C extends B {

Method C()
 0 aload_0
 1 invokenonvirtual #4 // B.<init>()
 4 return


Method java.lang.String value()
 0 aload_0
 1 invokenonvirtual #5 // A.value()
 4 areturn
}
```

Yes, I did do this with Java 1.0



invokespecial Java 1.1+

```
public class A {
  private String inner() {return “A”;}
  public String value() {
    return inner();
  }
}

public class B extends A {
  private String inner() {return “B”;}
}

public class C extends B {
    public String value() {
        return super.value();
    }
}
```

```
public class C extends B {
// Flags: ACC_PUBLIC, ACC_SUPER

public C();
 0: aload_0
 1: invokespecial #1 // B."<init>"
 4: return

public java.lang.String getValue();
  0: aload_0
  1: invokespecial #2 // B.value:()
  4: areturn
}
```

Other invoke opcodes

- invokeinterface
- invokestatic

- invokedynamic

Flow Control: Unconditional

`goto`	branch to bytecode offset of ±32K (16 bit signed offset)
`goto_w`	branch to bytecode offset of ±2G (64 bit signed offset)

Do not use: Deprecated in JVM 7

`jsr`	internal call to bytecode offset of ±32K (16 bit signed offset)
`jsr_w`	internal call to bytecode offset of ±2G (64 bit signed offset)
`ret`	Return from internal call

Flow Control: Conditionals

These flow control branch based on an integer on the stack

`ifeq`	Branch if value is 0
`ifne`	Branch if value is not 0
`ifgt`	Branch if value is greater than 0
`ifge`	Branch if value is 0 or greater than 0
`iflt`	Branch if value is less than 0
`ifle`	Branch if value is 0 or less than 0

These flow control branch based on an Object ref on the stack

`ifnull`	Branch if reference is `null`
`ifnotnull`	Branch if reference is not `null`

Flow Control: Exits

All of these (except “return”) use the value at the top of the stack. Any other stack contents are discarded.

`return`	Exit with no return value (method must have return type V)
`dreturn`	Exit with double return value (method must have return type D)
`freturn`	Exit with float return value (method must have return type F)
`lreturn`	Exit with long return value (method must have return type J)
`ireturn`	Exit with int return value (method must have return type I)
`areturn`	Exit with Object ref return value (must match return type)

`athrow` Exit by throwing Throwable value. If it’s a checked Exception, it must be declared.

Reading and Writing Fields

`getstatic`	Read a static field onto the stack
`putstatic`	Write a value to a static field (must not be **final**)
`getfield`	Read a member field onto the stack
`putfield`	Write a value to a member field (must not be **final**)

Accessing Locals

aload, astore	Read/write an Object reference into a slot
dload, dstore	Read/write a double into a slot (and the next slot)
fload, fstore	Read/write a float into a slot
iload, istore	Read/write an integer into a slot
lload, lstore	Read/write a log into a slot (and the next slot)

All of these have 5 forms. The “two byte” form has an opcode and a one-byte slot index*. It can access any slot from 0-255. There are also single-byte opcodes for the first four slots for all of these types:

_xload_0, _xstore_0
_xload_1, _xstore_1
_xload_2, _xstore_2
_xload_3, _xstore_3

*If the `wide` opcode preceded the _xload opcode, then the slot index becomes 16 bits.

Moving stuff on the stack

dup	Duplicate the 1-slot value on the top of the stack
dup2	Duplicate top two slots on the stack (may be single 2-slot value)
dup_x1	Duplicate top 1-slot value and insert into the third position
dup_x2	Duplicate top 1-slot value and insert into the fourth slot position
dup2_x1	Duplicate top two slots and put as 4th and 5th positions
dup2_x2	Duplicate top two slots and put as 5th and 6th positions
pop	Remove top one-slot value from stack
pop2	Remove two slots from the top of stack (may be single 2-slot)
swap	Swap top two one-slot values (must be 1-slot values)

New

new Create a new Object reference (not for arrays)

This creates an *uninitialized* object instance. The code *must* initialize it by calling an `<init>` method on the instance.

This is a common construction:

```
new #4  // class X
dup     // two copies of unitialized X ref on stack
invokespecial #5 // X.<init>()
```

No Operation

nop Do nothing

Of course there's a nop. It's opcode is 0, meaning that an unitialized byte array automatically contains nop instructions.

Let’s Build the GetSet Compiler

We’re going to build a compiler for a DSL that makes Javabeans (value objects).

The source parser has been written. We will fill in the `ClassBuilder` that generates the bytecode for our output classes.

GetSet: A DSL for value objects

```
# Lines starting with # are comments
>> com.company.package

<< java.util.Date
<< java.util.List

DatedList<T>

name : String
date ! Date
list !! List<T>
```

Package declaration (mandatory, no default package).

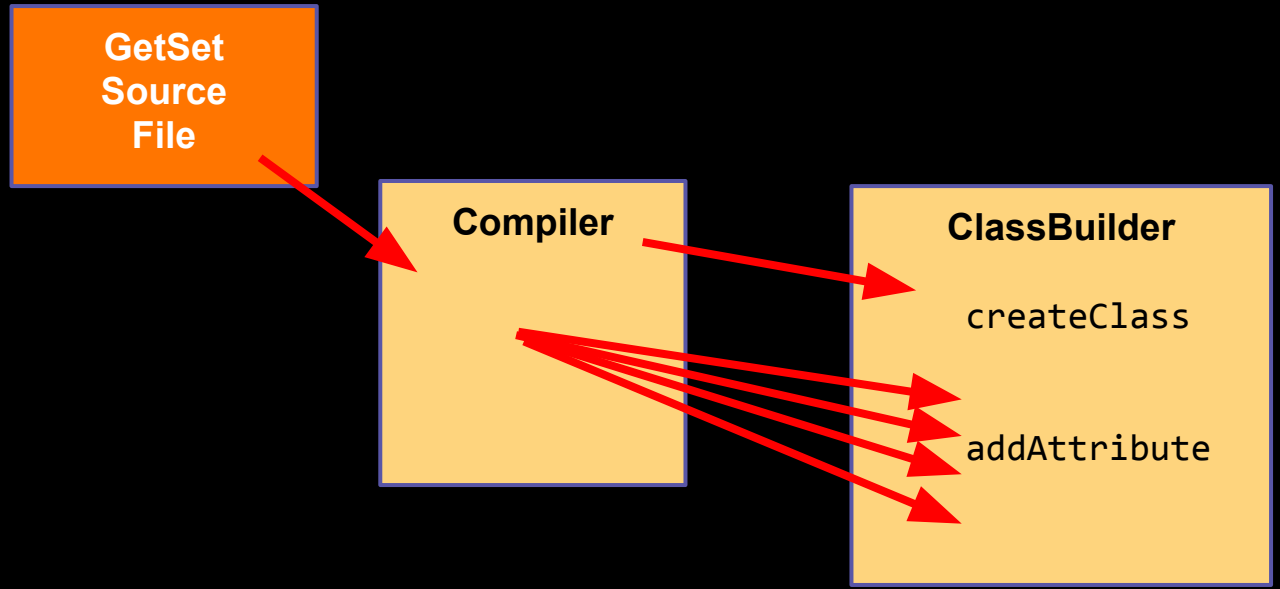
Imports to simplify names. java.lang is auto-imported. Imports are optional.

Class name. Allows declaration of unbounded generic type parameters.

Create getters and setters for type. Ex:
`setName(String)`
`List<T> getList()`

`:`	Get and set any value to this attribute
`!`	Get can return `null` but set may not set `null`
`!!`	Get never returns `null`, set never accepts `null`

GetSet Compiler Overview



GetSet Compiler Overview

The `createClass` method is called once.

```
public ClassNode createClass(String name, String signature)
```

The method takes the internal name of the class and its generic signature (this is **`null`** if there are no generic elements).

It returns an ASM `ClassNode`. This is roughly based on the class file. A `ClassNode` can be easily compiled into a class file byte array:

```
ClassWriter cw = new ClassWriter(0);
cn.accept(cw);
byte[] classfile = cw.toByteArray();
```

You can set flags to have ASM compute the stack map and max stack sizes

GetSet Compiler Overview

The `addAttribute` method is called once.

```
public void addAttribute(String attrName,
                         String attrSignature,
                         List<String> genericParameters,
                         Set<Options> options)
```

This method adds a Javabean attribute. Each call adds the following to the `ClassNode`:

- A member field of the correct type
- A “getter” for that field
- A “setter” for that field

GetSet Compiler Overview

We will need to use a couple more ASM classes that are children of `ClassNode`:

`FieldNode`

We will need to create a field for each attribute and add it to the `ClassNode`.

`MethodNode`

We will need to create a `MethodNode` for each getter, each setter, and the default constructor. These will also need to be added to the parent `ClassNode`.

GetSet Compiler Overview

```
public void addAttribute(String attrName,
                         String attrSignature,
                         List<String> genericParameters,
                         Set<Options> options)
```

The parameters to the method are:

attrName	Attribute name; used to name the field, the getter and the setter
attrSignature	The signature (if generic) or descriptor (if not generic) of the attribute
genericParameters	Generic parameters of the class *(we won't be using this)*
options	Attribute options, e.g. **NOT_NULL** and **NEVER_NULL**

Exercises

GetSet: Exercise 0

Get the “Exercise 0” Unit Test to pass. This test just checks that the Java version is Java 7. We’ll set some other basic attributes while we’re at it:

```
public ClassNode createClass(String name, String signature) {
    cn.version = Opcodes.V1_7;
    cn.access = Opcodes.ACC_PUBLIC + Opcodes.ACC_SUPER;

    cn.name = name.replace('.', '/');
    cn.superName = "java/lang/Object";

    return cn;
}
```

GetSet: Exercise 1

Compile “Nothing.gs” which is a bean with no attributes

```
>> pkg1

Nothing
```

GetSet: Exercise 1

Q: What do we need to add to Exercise 0 to get a properly-formed class?

A: *A default constructor.*

I add this field to ClassBuilder:

```
private final MethodNode constructor =
  new MethodNode(Opcodes.ACC_PUBLIC, "<init>", "()V", null, null);
```

This code to createClass:

```
constructor.instructions.add(new VarInsnNode(Opcodes.ALOAD, 0));
constructor.instructions.add(new MethodInsnNode(Opcodes.INVOKESPECIAL,
    "java/lang/Object", "<init>", "()V", false));
constructor.instructions.add(new InsnNode(Opcodes.RETURN));

constructor.maxStack = 1;
constructor.maxLocals = 1;

cn.methods.add(constructor);
```

GetSet: Exercise 2

Compile “IntHolder.gs” which is a bean with an int attribute

```
>> pkg1

IntHolder

value : int
```

GetSet: Exercise 2

We need to handle the addition of an attribute, “value”. This will result in the addition of the following things to the class, IntHolder:

```
private int value;

public int getValue() {
     return value;
}

public void setValue(int v) {
     value = v;
}
```

GetSet: Exercise 2

The parser will call `addAttribute`, which needs to add a field, a getter, and a setter.

```
public void addAttribute(String attrName, String attrSignature,
        List<String> genericParameters,
        Set<Options> options) {
    FieldNode fn = addField(attrName, attrSignature);
    addGetter(fn);
    addSetter(fn);
}
```

Why pass the FieldNode to `addGetter` and `addSetter`? It carries the info we need:

```
fn.name = “value”
fn.descriptor = “I”
fn.signature = null  (we will use this later)
```

GetSet: Exercise 2

The `addField` method creates a `FieldNode` and adds it to the class.

```
private FieldNode addField(String attrName,
          String attrSignature) {
     FieldNode fn =
               new FieldNode(Opcodes.ACC_PRIVATE,
                         attrName,
                         attrSignature,
                         null, null);
     cn.fields.add(fn);
     return fn;
}
```

Generic signature (**null** for now)

Initial value (for **static final** only)

GetSet: Exercise 2

The addGetter method creates a MethodNode…

```
private void addGetter(FieldNode fn) {
    MethodNode mn = new MethodNode(Opcodes.ACC_PUBLIC,
            makeAccessorName("get", fn.name),
            "()" + fn.desc, null, null);
```

makeAccessorName will take “value” and return “getValue”

Getter descriptor: “()I”

Generic signature (**null** for now)

Array of checked Exceptions

GetSet: Exercise 2

Then, addGetter creates the bytecode for the method:

```
mn.instructions.add(new VarInsnNode(Opcodes.ALOAD, 0));
mn.instructions.add(new FieldInsnNode(Opcodes.GETFIELD, cn.name, fn.name,
fn.desc));
// TODO: handle non-int
mn.instructions.add(new InsnNode(Opcodes.IRETURN));

mn.maxStack = 1;
mn.maxLocals = 1;

// Don't forget this:

cn.methods.add(mn);
```

Generated Code:

```
aload_0
getfield IntHolder.value I
ireturn
```

GetSet: Exercise 2

The `addSetter` method creates a `MethodNode`…

```
private void addSetter(FieldNode fn) {
    MethodNode mn = new MethodNode(Opcodes.ACC_PUBLIC,
            makeAccessorName("set", fn.name),
            String.format("(%s)V", fn.desc), null, null);
```

makeAccessorName will take “value” and return “setValue”

Setter descriptor: “`(I)V`”

Generic signature (**null** for now)

Array of checked Exceptions

GetSet: Exercise 2

Then, `addSetter` creates the bytecode for the method:

```
mn.instructions.add(new VarInsnNode(Opcodes.ALOAD, 0));
// TODO: handle non-int
mn.instructions.add(new VarInsnNode(Opcodes.ILOAD, 1));
mn.instructions.add(new FieldInsnNode(Opcodes.PUTFIELD, cn.name, fn.name,
fn.desc));
mn.instructions.add(new InsnNode(Opcodes.RETURN));

mn.maxStack = 2;
mn.maxLocals = 2;

cn.methods.add(mn);
```

Generated Code:

```
aload_0
iload_1
putfield IntHolder.value I
return
```

GetSet: Exercise 2

Here's the implementation of `makeAccessorName`:

```
private String makeAccessorName(String prefix, String attrName) {
     return String.format("%s%S%s", prefix,
                attrName.substring(0,1),
                attrName.substring(1));
}
```

GetSet: Exercise 3

Compile “PrimitiveHolder.gs” which is a bean with primitive attributes

```
>> pkg1

PrimitiveHolder

value : boolean
value : byte
value : short
value : char
value : int
value : long
value : float
value : double
```

GetSet: Exercise 3 - Testing

We can't use regular Java reflection to find return-value polymorphic functional, so we use new Java 7 reflection.

```
// This will find a getValue method, but just the first one in the
// class (returns byte)
Method get = holderClass.getMethod("getValue");

// This will find a getValue method that returns int
MethodHandle get =
    MethodHandles.lookup().findVirtual(holderClass,
                                       "getValue",
                                       MethodType.methodType(int.class);
```

The first parameter of **methodType** is the return type.

GetSet: Exercise 3

In Exercise 2, we assumed that the attribute was an **int**. Now we have to handle any primitive type. Replace these hardcoded opcodes with calls to methods. These methods should return the proper typed opcode for the primitive type.

```
// TODO: handle non-int
mn.instructions.add(new InsnNode(Opcodes.IRETURN));

// TODO: handle non-int
mn.instructions.add(new VarInsnNode(Opcodes.ILOAD, 1));

// Don't forget, double and long take 2 slots as operands and locals
mn.maxStack = getSlots(fn.desc);
mn.maxLocals = getSlots(fn.desc);
```

GetSet: Exercise 4

Compile “ObjectHolder.gs” which is a bean with basic object attributes

```
>> pkg1

<< java.util.Date

ObjectHolder

date : Date
string : String
```

GetSet: Exercise 4

We have to modify the code in Exercise 4 to handle object references. We created methods to get the return and load opcode. We should have created methods to choose the correct opcodes. These methods can be altered to return an object-reference opcode when the descriptor is not a primitive.

Getter:

```
mn.instructions.add(new InsnNode(getReturnOpcode(fn.desc)));
```

Setter:

```
mn.instructions.add(new VarInsnNode(getLoadOpcode(fn.desc), 1));
```

GetSet: Exercise 5

Compile “GenericObjectHolder.gs” which is a bean with basic object attributes

```
>> pkg1

<< java.util.List
<< java.util.Map

GenericObjectHolder

string : String
list : List<Integer>
map : Map<Integer, String>
```

GetSet: Exercise 5

The ClassBuilder gets the ***signature*** of the attribute. This includes parameterized type (generic) information. The code needs to strip this off and generate “an erasure” to put in the descriptor. When there the signature contains generic information, the signature should be set, otherwise it should be **null**. This can be done on the FieldNode and reused when creating the getters and setters.

```
FieldNode fn =
     new FieldNode(Opcodes.ACC_PRIVATE,
               attrName,
               getErasure(attrSignature),
               attrSignature.contains("<") ? attrSignature : null,
               null);
```

GetSet: Exercise 5

This is my implementation of getErasure.

```
private String getErasure(String signature) {
    if (signature.contains("<")) {
        return signature.substring(0, signature.indexOf('<')) + ";";
    }
    return signature;
}
```

GetSet: Exercise 6

Compile “BasicGenericObject.gs” which is a bean with type parameters

```
>> pkg1

<< java.util.List

BasicGenericHolder<L, T>

thing : T
list : List<L>
```

GetSet: Exercise 6

In the case of the **`java/util/List`**, there will be a clause in angle brackets that we can “erase.” For unresolved generic type T, we will get the pseudo-descriptor “**`TT;`**”. This erases to **`java/lang/Object`**. (In the case where the language supports bounds on the type parameter, the unresolved type will resolve to its boundary type.)

`thing` : *T*
Erasure: `java/lang/Object`

`list` : *List<L>*
Erasure: `java/util/List`

GetSet: Exercise 7

Compile “NotNull.gs” which is a bean with an attribute that starts out uninitialized (`null`), but cannot be set to `null`.

```
>> pkg1

NotNull

string ! String
```

GetSet: Exercise 7

For this, we need to check the Options to see if **NOT_NULL** is set. If it is, we need to add logic to check for null and throw a NullPointerException if the reference is null.

We will need to encode a jump in ASM. To do this, we create a label node and use it in the jump:

```
LabelNode label = new LabelNode();
mn.instructions.add(new JumpInsnNode(Opcodes.IFNONNULL, label));
```

Later, we insert the label at the jump target as if it were an instruction:

```
mn.instructions.add(label);
```

GetSet: Exercise 7

This is the code we'll need to insert:

```
aload_0
aload_1

     dup
     if_notnull label
     new java/lang/NullPointerException
     dup
     invokespecial java/lang/NullPointerException.<init>()V
     athrow
label:
StackMap(locals: <this type>, <attr type> operands: <this type>, <attr type>)

putfield IntHolder.value I
return
```

GetSet: Exercise 7

Here's how we add the Stack Map. First, take the "L" and ";" off the object descriptor* to get the raw "internal" type name.

```
String pName = fn.desc.substring(1, fn.desc.length() - 1);
```

Now create the FrameNode as add it as a pseudo-instruction:

```
mn.instructions.add(new FrameNode(Opcodes.F_FULL,
                    2, new Object[] { cn.name, pName },
                    2, new Object[] { cn.name, pName }));
```

* This won't work for array descriptors, we'll fix that later.

GetSet: Exercise 8

Compile “NeverNull.gs” which is a bean with an attribute that starts out initialized to a default, and cannot be set to `null`, so it is never null.

```
>> pkg1

NeverNull

string !! String
```

GetSet: Exercise 8

We need to initialize “never null” fields. The String class (among many others) has a default constructor, so we can take advantage of that to create an initial value.

This initialization code needs to be added to the constructor. We put a label into the constructor to act like a “bookmark” for adding initialization code later:

```
private final LabelNode initReturn = new LabelNode();
```

This label is inserted before the “return” in the constructor:

```
constructor.instructions.add(initReturn);
constructor.instructions.add(new InsnNode(Opcodes.RETURN));
```

GetSet: Exercise 8

This is the code we need to generate. Note that it *only* works for objects with default constructors. It doesn't work for arrays, or abstract classes. We'll fix the array problem in the next exercise.

```
aload_0
new [attribute object type]
dup
invokespecial [attribute object type].<init>()V
putfield [attribute name].[attribute type]
```

GetSet: Exercise 8

After we add a never-null attribute, we need to insert initialization code before the label:

```
InsnList initVariable = new InsnList();  // create an instruction list
String pName = descToTypeName(fn.desc);  // get “internal name” of type
initVariable.add(new VarInsnNode(Opcodes.ALOAD, 0));
initVariable.add(new TypeInsnNode(Opcodes.NEW, pName));
initVariable.add(new InsnNode(Opcodes.DUP));
initVariable.add(new MethodInsnNode(Opcodes.INVOKESPECIAL, pName,
"<init>", "()V", false));
initVariable.add(new FieldInsnNode(Opcodes.PUTFIELD, cn.name, fn.name,
fn.desc));
constructor.instructions.insertBefore(initReturn, initVariable);
constructor.maxStack = 4; // we need more stack for this
```

GetSet: Exercise 9

Compile “NeverNullArray.gs” which is a bean with attributes that hold arrays. Because the arrays need to be initialized (never null constraint) we have to allocate empty arrays.

```
>> pkg1

NeverNullArray

strings !! String[][][]
ints !! int[][]
```

GetSet: Exercise 9

Here's raw bytecode to allocate the empty arrays we need:

```
new String[0][0][0]

     iconst_0
     iconst_0
     iconst_0
     multianewarray [[[Ljava/lang/String; 3

new int[0][0]

     iconst_0
     iconst_0
     multianewarray [[I 2
```

GetSet: Exercise 9

This is the code to add one `iconst_0` for each dimension and then insert a `multianewarray` instruction:

```
int i = 0;
while(pName.charAt(i) == '[') {
    initVariable.add(new InsnNode(Opcodes.ICONST_0));
    i++;
}
initVariable.add(new MultiANewArrayInsnNode(pName, i));
```

GetSet: Exercise 9

Here's the entire initialization code creator:

```
if (options.contains(Options.NEVER_NULL)) {
    InsnList initVariable = new InsnList();
    String pName = descToTypeName(fn.desc);
    initVariable.add(new VarInsnNode(Opcodes.ALOAD, 0));
    if (pName.startsWith("[")) {
        int i = 0;
        while(pName.charAt(i) == '[') {
            initVariable.add(new InsnNode(Opcodes.ICONST_0));
            i++;
        }
        initVariable.add(new MultiANewArrayInsnNode(pName, i));
    } else {
        initVariable.add(new TypeInsnNode(Opcodes.NEW, pName));
        initVariable.add(new InsnNode(Opcodes.DUP));
        initVariable.add(new MethodInsnNode(Opcodes.INVOKESPECIAL, pName, "<init>", "()V", false));
    }
    initVariable.add(new FieldInsnNode(Opcodes.PUTFIELD, cn.name, fn.name, fn.desc));
    constructor.instructions.insertBefore(initReturn, initVariable);
    constructor.maxStack = 4;
}
```