How the JVM Executes Java

this slide intentionally left blank

Classes and Class Files

- Use `javap` to examine class file

HelloWorld0.java

```
public class HelloWorld0 {
  public static void main ( String [] args ) {
    System.out.println("Hello, it's Java
time");
```

```
$ javap HelloWorld0
Compiled from "HelloWorld0.java"
public class HelloWorld0 {
  public HelloWorld0();
  public static void main(java.lang.String[]);
}
```

Classes and Class Files

- Use `javap` to examine class file

HelloWorld0.java

```
public class HelloWorld0 {
  public static void main ( String [] args ) {
    System.out.println("Hello, it's Java
time");
```

```
$ javap -c HelloWorld0
Compiled from "HelloWorld0.java"
public class HelloWorld0 {
 public HelloWorld0();
  Code:
   0: aload_0
   1: invokespecial #1    // Method java/lang/Object."<init>":()V
   4: return

 public static void main(java.lang.String[]);
  Code:
   0: getstatic     #2    // …
   3: ldc           #3    // String Hello, it's Java time
   5: invokevirtual #4    // …
   8: return
}
```

Classes and Class Files

- Use `javap` to examine class file

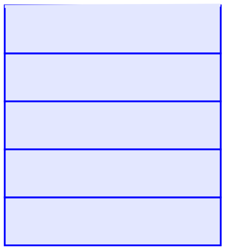
HelloWorld0.java

```
public class HelloWorld0 {
  public static void main ( String [] args ) {
    System.out.println("Hello, it's Java
time");
```

```
$ javap -verbose HelloWorld0
Classfile /Users/george/work/java/JVM/HelloWorld0.class
  Last modified 27-May-2014; size 437 bytes
  MD5 checksum d587c1c612c2809f3a6288317fd631fa
  Compiled from "HelloWorld0.java"
public class HelloWorld0
  SourceFile: "HelloWorld0.java"
  minor version: 0
  major version: 52
  flags: ACC_PUBLIC, ACC_SUPER
Constant pool:
   #1 = Methodref    #6.#15   //  java/lang/Object."<init>":()V
   #2 = Fieldref     #16.#17  //  java/lang/System.out:Ljava/io...
   #3 = String       #18      //  Hello, it's Java time
   #4 = Methodref    #19.#20  //  …
   …
```

Introducing Bytecode

- Executable code for the Java Virtual Machine
- Compact – 1 byte for opcode
  - most opcodes do not take arguments
  - JVM is stack based machine
- Opcodes are largely type related
  - `fload` float
  - `imul` int
  - `dstore` double
- Also control flow
  - `invokevirtual`
  - `goto`
  - `areturn`



Introducing Bytecode

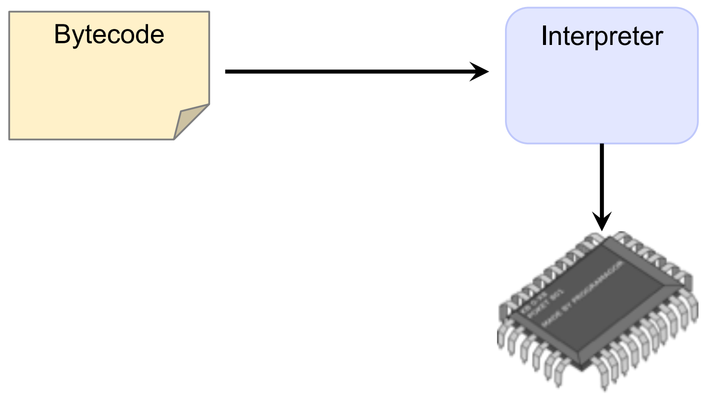
- A simple example:

```
public class IncExample {
  public int inc ( int i ) {
    int result;
    result = i + 1;
    return result;
  }
}
```

```
$ javap -verbose IncExample
…
 public int inc(int);
    descriptor: (I)I
    flags: ACC_PUBLIC
    Code:
      stack=2, locals=3, args_size=2
         0: iload_1
         1: iconst_1
         2: iadd
         3: istore_2
         4: iload_2
         5: ireturn
      LineNumberTable:
        line 4: 0
        line 5: 4
…
```

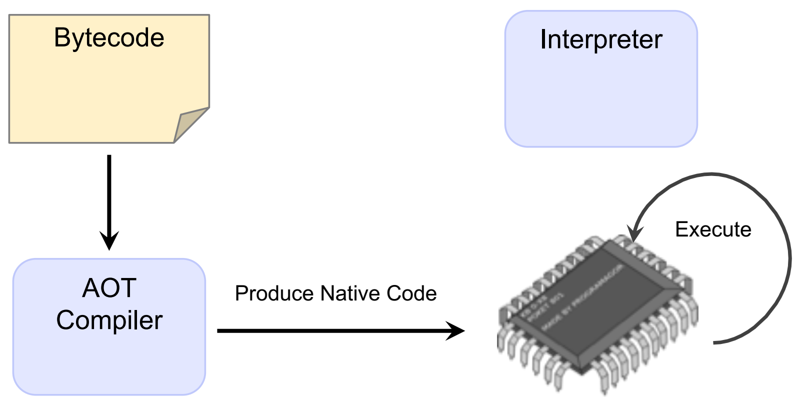
Executing Bytecode

- The simple view



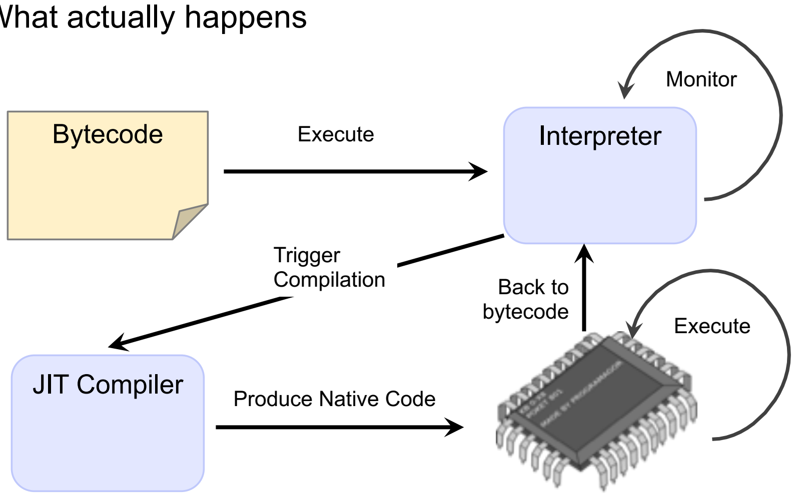
Executing Bytecode

- The slightly more complicated view



Executing Bytecode

- What actually happens



Dynamic Compilation Options

- Client mode
  - `java –client`
  - less aggressive optimisation
  - no speculative optimisation
  - smaller memory footprint
  - faster startup
- Server mode
  - `java –server`
  - full optimisation
  - suited to long running applications
- Tiered Compilation
  - start by using client compilations
  - switch to server mode when available

Execution Example

- Simple example
  - loop necessary to see dynamic compilation

```
public class HelloWorld {

  public static void main(String[] args) {
    for (int i = 0; i < 100_000; i++) {
      hello();
    }
  }

  private static void hello() {
    System.err.println("Hello, world!");
  }

}
```

Execution Example

- Run in default mode

```
$ time java HelloWorld 2>/dev/null
real    0m0.336s
user    0m0.467s
sys     0m0.120s
```

- Interpreted mode

```
$ time java -Xint HelloWorld 2> /dev/null
real    0m1.969s
user    0m1.828s
sys     0m0.152s
```

- Compiled mode

```
$ time java -Xcomp HelloWorld 2> /dev/null
real    0m2.365s
user    0m2.198s
sys     0m0.241s
```

Dynamic Compiler Diagnostics

- Command line flags provide further information

```
$ java -XX:-TieredCompilation -XX:+PrintCompilation HelloWorld
                                                      2> /dev/null
  72    1          java.lang.String::hashCode (55 bytes)
  79    2          java.lang.String::indexOf (70 bytes)
 104    3          sun.nio.cs.UTF_8$Encoder::encodeArrayLoop
                                                         (489 bytes)
 120    4          java.nio.Buffer::position (5 bytes)
 122    5          java.nio.ByteBuffer::arrayOffset (35 bytes)
 127    6          java.nio.Buffer::position (43 bytes)
 127    7     n    java.lang.System::arraycopy (native) (static)
 ...
 247   47          java.lang.String::indexOf (7 bytes)
 247   48          HelloWorld::hello (9 bytes)
 249   49    !     java.io.PrintStream::println (24 bytes)
 256   50          java.io.PrintStream::print (13 bytes)
 259   51    !     java.io.PrintStream::write (83 bytes)
 262   52    !     java.io.PrintStream::newLine (73 bytes)
 264   53          java.io.BufferedWriter::newLine (9 bytes)
 265   54 %        HelloWorld::main @ 2 (18 bytes)
```

Dynamic Compiler Diagnostics

- Decoding the output

Time offset (ms) | Compiler Task No | Name of method | Size of Compiled code

```
249   49   !    java.io.PrintStream::println (24 bytes)
```

Method flags:

`!`	method has exception handler(s)
`s`	method declared `synchronized`
`n`	native method (no compilation, generate wrapper)
`%`	on-stack replacement used

Dynamic Compiler Diagnostics

- `-Xbatch` forces compiler into application thread
  - application paused while compiler runs
  - b flag shown for compilation
  - overall time increases

```
$ java –Xbatch -XX:-TieredCompilation -XX:+PrintCompilation
                                             HelloWorld 2> /dev/null

 75     1     b     java.lang.String::hashCode (55 bytes)
 84     2     b     java.lang.String::indexOf (70 bytes)
112     3     b     sun.nio.cs.UTF_8$Encoder::encodeArrayLoop
                                                          (489 bytes)
141     4     b     java.nio.Buffer::position (43 bytes)
143     5      n    java.lang.System::arraycopy (native)   (static)
147     6     b     java.nio.Buffer::position (5 bytes)
161     7     b     java.nio.charset.CoderResult::isUnderflow (13 bytes)
161     8     b     java.io.BufferedWriter::ensureOpen (18 bytes)
162     9     b     java.io.PrintStream::ensureOpen (18 bytes)
162    10    !b     java.io.BufferedWriter::write (117 bytes)
…
```

Dynamic Compiler Diagnostics

- Summary statistics can be obtained

```
$ java -Xbatch -XX:-TieredCompilation -XX:+PrintCompilation
                                  -XX:+CITime HelloWorld 2> /dev/null
...
Accumulated compiler times (for compiled methods only)
------------------------------------------------
  Total compilation time    :  0.195 s
    Standard compilation    :  0.194 s, Average : 0.004
    On stack replacement    :  0.001 s, Average : 0.001

  Total compiled methods    :     52 methods
    Standard compilation    :     51 methods
    On stack replacement    :      1 methods
  Total compiled bytecodes  :  11838 bytes
    Standard compilation    :  11820 bytes
    On stack replacement    :     18 bytes
  Average compilation speed:  60754 bytes/s

  nmethod code size         :  57376 bytes
  nmethod total size        : 111752 bytes
```

On Stack Replacement?

- Normal compilation triggered by call count
  - method has been called a specific number of times
  - `-XX:CompileThreshold`
  - defaults to 10000
- JVM can detect method that is in long loop
  - e.g. `main` in example
- Compiler can replace method while it is executing
  - On Stack Replacement

```
422    54 %  b         HelloWorld::main @ 2 (18 bytes)
```

`LogCompilation`

- For more detail than `PrintCompilation` provides
- Generates a detailed (& large) XML file
  - Often 100s of MB
- Full detail of compilation events
- Hard to handle without tooling (e.g. JITWatch)

Viewing Assembly Code

- Possible to see assembly code output from compiler
  - requires additional library for display

```
public class TinyExample {

  public static void main ( String [] args ) {
    for ( int i=0; i < 1000000; i++ ) {
      tiny();
    }
  }

  private static int tiny() {
    return 1 + 1;
  }

}
```

Assembly Primer

- AT&T format
  - Usually <operator> <src>, <dst>

```
mov %rdx, %rax      ; move %rdx into accumulator
add %rcx, %rax      ; add %rcx to accumulator
XOR %eax, %eax      ; Zero a register
mov $0, %eax        ; Zero a register
XCHG %eax, %eax     ; No-op, often used for memory fences
```

Assembly Syntax

- Common prefixes

```
b  – byte (8 bits)
s  – short (16 bits)
w  – word (16 bits)
l  – long (32 bits)
q  – quad (64 bits)
t  – ten (80 bits, floating point)
```

```
ax  – accumulator
bp  – frame pointer
sp  – stack pointer
```

```
e  – 32-bit
r  – 64-bit
```

Viewing Assembly Code

```
$ java -XX:+UnlockDiagnosticVMOptions -XX:+PrintAssembly TinyExample
…
Code:
[Entry Point]
[Verified Entry Point]
[Constants]
  # {method} {0x0000000123b9f300} 'tiny' '()I' in 'TinyExample'
  #           [sp+0x40]  (sp of caller)
  0x000000010f211560: mov    %eax,-0x14000(%rsp)
  0x000000010f211567: push   %rbp
  0x000000010f211568: sub    $0x30,%rsp         ;*iconst_2
                                                ; -
TinyExample::tiny@0 (line 10)

  0x000000010f21156c: mov    $0x2,%eax
  0x000000010f211571: add    $0x30,%rsp
  0x000000010f211575: pop    %rbp
  0x000000010f211576: test   %eax,-0x202d47c(%rip)        #
0x000000010d1e4100
                                                ;   {poll_return}
  0x000000010f21157c: retq
…
```

Basic Optimisation

- Method inlining
  - most commonly applied optimisation
  - removes call overhead
  - can be tuned with command line options

```
$ java -Xbatch -XX:-TieredCompilation -XX:+PrintCompilation
       -XX:+UnlockDiagnosticVMOptions -XX:+PrintInlining
       HelloWorld 2> /dev/null

…
191 14   b  java.io.BufferedOutputStream::flushBuffer (29 bytes)
              @ 20   java.io.FileOutputStream::write (12 bytes)
                                                     inline (hot)
               \-> TypeProfile (4466/4466 counts) =
                                            java/io/FileOutputStream
                  @ 8   java.io.FileOutputStream::writeBytes (0 bytes)
                                                    native method
…
```

Monomorphic Dispatch

- How many different types are seen at a call site?
  - often, it's only 1
  - JVM optimizes for this case
  - aggressive optimisation (so only in server mode)
  - can be backed out
- Hotspot optimizes vtbl lookup
  - subclasses have the same vtbl structure as their parent
  - hotspot collapses the child vtbl into the parent
  - class word in the object header is checked
  - if changed then this optimisation is backed out
- Classloading can invalidate monomorphic dispatch

Further JVM Optimisations

- Loop Unrolling

```
private static final String[] options =
                          { "Yes", "No", "Cancel" };

public void meth1 () {
  for ( String opt: options ) {
    process(opt);
  }
}
```

```
private static final String[] options =
                          { "Yes", "No", "Cancel" };

public void meth1 () {
  process(options[0]);
  process(options[1]);
  process(options[2]);
}
```

Further JVM Optimisations

- Lock Coarsening
  - effective but depends on scope of lock

```
public void doStuff () {
  for ( String opt: options ) {
    process(opt);
  }
}

public synchronized String process ( String opt ) {
  …
}
```

```
public void doStuff () {
  synchronized ( this ) {
    for ( String opt: options ) {
      process(opt);
    }
  }
}
```

Further JVM Optimisations

- Lock Elision
  - also depends on local scope of lock

```
public void doStuff () {
  List l = new ArrayList();
  synchronized (l) {
    for (option : options) {
     l.add(process(option));
    }
  }
}
```

List l is local and does not "escape" this thread

```
public void doStuff () {
  List l = new ArrayList();
  for (option : options) {
    l.add(process(option));
  }
}
```

Lock is not required

Further JVM Optimisations

- Dead Code Elimination
  - remove code that does nothing

```
public class TinyExample {

  public static void main ( String [] args ) {
    for ( int i=0; i < 1000000; i++ ) {
      tiny();
    }
  }

  private static void tiny() {
    int i = 100;
    i += 1;
  }
}
```

Code has no effect, may be removed during compilation

Further JVM Optimisations

- Escape Analysis
  - allows reduction of heap usage

```
private static class Foo {
  public final String a;
  public final String b;

  Foo(String a, String b) {
    this.a = a;
    this.b = b;
  }
}
```

```
public void one() {
  Foo f = new Foo("Hello", "JVM");
  two(f);
}

public void two(Foo f) {
  System.out.print(f.a);
  System.out.print(", ");
  three(f);
}

public void three(Foo f) {
  System.out.print(f.b);
  System.out.println('!');
}
```

Further JVM Optimisations

- Escape Analysis
  - allows reduction of heap usage

```
private static class Foo {
  public final String a;
  public final String b;

  Foo(String a, String b) {
    this.a = a;
    this.b = b;
  }
}
```

```
public void oneTwoThree() {
  System.out.print("Hello");
  System.out.print(", ");
  System.out.print("JavaOne");
  System.out.println('!');
}
```