Understanding the Dalvik Virtual Machine

Jim Huang (黃敬群)

Developer, 0xlab

jserv@0xlab.org

March 14, 2012 / GTUG Taipei

Rights to copy

http://0xlab.org/

contact@0xlab.org

creative commons
COMMONS DEED

Attribution – ShareAlike 3.0

Corrections, suggestions, contributions and translations are welcome!

Latest update: March 14, 2012

You are free

- to copy, distribute, display, and perform the work
- to make derivative works
- to make commercial use of the work

Under the following conditions

BY: **Attribution**. You must give the original author credit.

Share Alike. If you alter, transform, or build upon this work, you may distribute the resulting work only under a license identical to this one.

- For any reuse or distribution, you must make clear to others the license terms of this work.
- Any of these conditions can be waived if you get permission from the copyright holder.

Your fair use and other rights are in no way affected by the above.

License text: http://creativecommons.org/licenses/by-sa/3.0/legalcode

Myself

was a Kaffe (world-frist open source JVM) Developer

- Threaded Interpreter, JIT, AWT for embedded system, robustness

was a GCJ (Java Frontend for GCC) and GNU Classpath Developer

is an AOSP (Android Open Source Project) contributror

- 45+ patches are merged officially
- bionic libc, ARM optimizations

Goals of This Presentation

- Understand how a virtual machine works
- Analyze the Dalvik VM using existing tools
- VM hacking is really interesting!

Environment Setup

Reference Hardware and Host Configurations

- Android Phone: Nexus S
  - http://www.google.com/phone/detail/nexus-s
  - Install CyanogenMod (**CM9**)

    http://www.cyanogenmod.com/
- Host: Lenovo x200
  - Ubuntu Linux 11.10+ (32-bit)
- AOSP/CM9 source code: 4.0.3
- Follow the instructions in Wiki
  http://wiki.cyanogenmod.com/wiki/Building_from_source

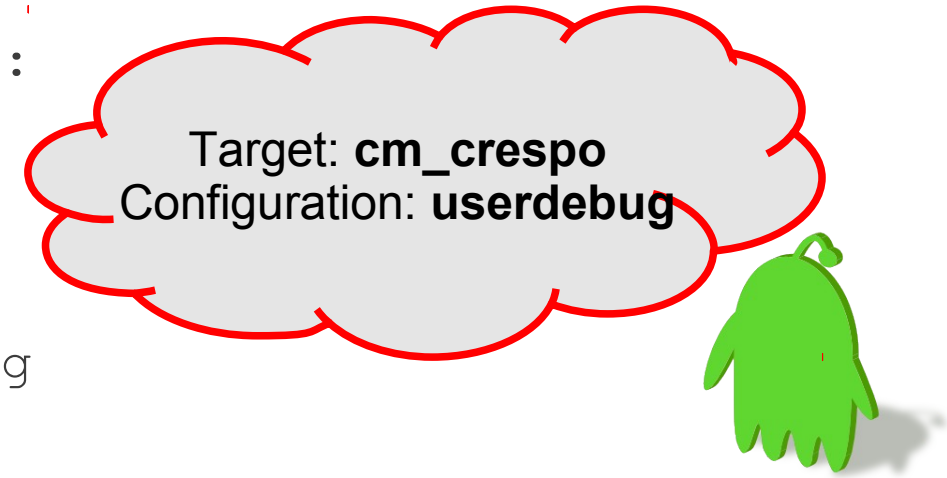
Build CyanogenMod from Source

- cyanogen-ics$ **source build/envsetup.sh**

```
including device/moto/stingray/vendorsetup.sh
including device/moto/wingray/vendorsetup.sh
including device/samsung/maguro/vendorsetup.sh
including device/samsung/toro/vendorsetup.sh
including device/ti/panda/vendorsetup.sh
including vendor/cm/vendorsetup.sh
including sdk/bash_completion/adb.bash
```

- cyanogen-ics$ **lunch**

```
You're building on Linux
Lunch menu... pick a combo:
     1. full-eng
…
     8. full_panda-eng
     9. cm_crespo-userdebug
```



Nexus S Device Configurations

- Which would you like? [full-eng] **9**

```
============================================
PLATFORM_VERSION_CODENAME=REL
PLATFORM_VERSION=4.0.3
TARGET_PRODUCT=cm_crespo
TARGET_BUILD_VARIANT=userdebug
TARGET_BUILD_TYPE=release
TARGET_BUILD_APPS=
TARGET_ARCH=arm
TARGET_ARCH_VARIANT=armv7-a-neon
HOST_ARCH=x86
HOST_OS=linux
HOST_BUILD_TYPE=release
BUILD_ID=MR1
============================================
```

Build Dalvik VM

(ARM Target + x86 Host)

- cyanogen-ics$ **make dalvikvm dalvik**

```
==========================================
PLATFORM_VERSION_CODENAME=REL
PLATFORM_VERSION=4.0.3
TARGET_PRODUCT=cm_crespo
TARGET_BUILD_VARIANT=userdebug
TARGET_BUILD_TYPE=release
...
Install: out/host/linux-x86/lib/libdvm.so
Install: out/target/product/crespo/system/bin/dalvikvm
host C++: dalvikvm <= dalvik/dalvikvm/Main.cpp
host Executable: dalvikvm Install: out/host/linux-x86/bin/dalvikvm
Copy: dalvik (out/host/linux-x86/obj/EXECUTABLES/dalvik_intermediates/dalvik)
Install: out/host/linux-x86/bin/dalvik
```

libdvm.so is the VM engine

"dalvik" is a shell script to launch dvm

Dalvik VM requires core APIs for runtime

```
cyanogen-ics$ out/host/linux-x86/bin/dalvik
E( 6983) No valid entries found in bootclasspath
'/tmp/cyanogen-ics/out/host/linux-x86/framework/core-
hostdex.jar:/tmp/cyanogen-ics/out/host/linux-
x86/framework/bouncycastle-hostdex.jar:/tmp/cyanogen-
ics/out/host/linux-x86/framework/apache-xml-
hostdex.jar'  (dalvikvm)
E( 6983) VM aborting  (dalvikvm)
...
out/host/linux-x86/bin/dalvik: line 28:  6983
Segmentation fault      (core dumped)
ANDROID_PRINTF_LOG=tag ANDROID_LOG_TAGS=""
ANDROID_DATA=/tmp/android-data
ANDROID_ROOT=$ANDROID_BUILD_TOP/out/host/linux-x86
LD_LIBRARY_PATH=$ANDROID_BUILD_TOP/out/host/linux-x86/lib
$ANDROID_BUILD_TOP/out/host/linux-x86/bin/dalvikvm -Xbootclasspath:
$ANDROID_BUILD_TOP/out/host/linux-x86/framework/core-hostdex.jar:
$ANDROID_BUILD_TOP/out/host/linux-x86/framework/bouncycastle-
hostdex.jar: \
$ANDROID_BUILD_TOP/out/host/linux-x86/framework/apache-xml-
hostdex.jar $*
```

Satisfy Dalvik Runtime Dependency

```
cyanogen-ics$ make bouncycastle bouncycastle-hostdex
cyanogen-ics$ make sqlite-jdbc mockwebserver
cyanogen-ics$ make sqlite-jdbc-host
cyanogen-ics$ make mockwebserver-hostdex
cyanogen-ics$ make apache-xml-hostdex
cyanogen-ics$ (cd libcore && make)
cyanogen-ics$ out/host/linux-x86/bin/dalvik
...
I(19820) Unable to open or create cache for
/tmp/cyanogen-ics/out/host/linux-x86/framework/core-
hostdex.jar (/data/dalvik-cache/tmp@cyanogen-
ics@out@host@linux-x86@framework@core-
hostdex.jar@classes.dex)  (dalvikvm)
E(19820) Could not stat dex cache directory
'/data/dalvik-cache': No such file or directory
(dalvikvm)
```

Extra space for "dalvik-cache" is required.

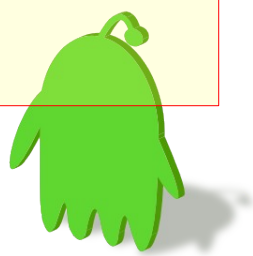
Host-side Dalvik VM

```
cyanogen-ics$ make dexopt
cyanogen-ics$ sudo mkdir -p /data/dalvik-cache
cyanogen-ics$ sudo chmod 777 /data/dalvik-cache
cyanogen-ics$ out/host/linux-x86/bin/dalvik
Dalvik VM requires a class name
```

Finally, host-side dalvik vm is ready.
It just complain no given class.

```
cyanogen-ics$ ls /data/dalvik-cache/
tmp@cyanogen-ics@out@host@linux-x86@framework@apache-xml-hostdex.jar@classes.dex
tmp@cyanogen-ics@out@host@linux-x86@framework@bouncycastle-hostdex.jar@classes.dex
tmp@cyanogen-ics@out@host@linux-x86@framework@core-hostdex.jar@classes.dex
```

Optimized DEX generated by "**dexopt**"



Agenda

(1) How Virtual Machine Works

(2) Dalvik VM

(3) Utilities

How Virtual Machine Works

What is Virtual Machine

- A **virtual machine** (**VM**) is a software implementation of a machine (i.e. a computer) that executes programs like a physical machine.
- Basic parts
  - A set of registers
  - A stack (optional)
  - An execution environment
  - A garbage-collected heap
  - A constant pool
  - A method storage area
  - An instruction set

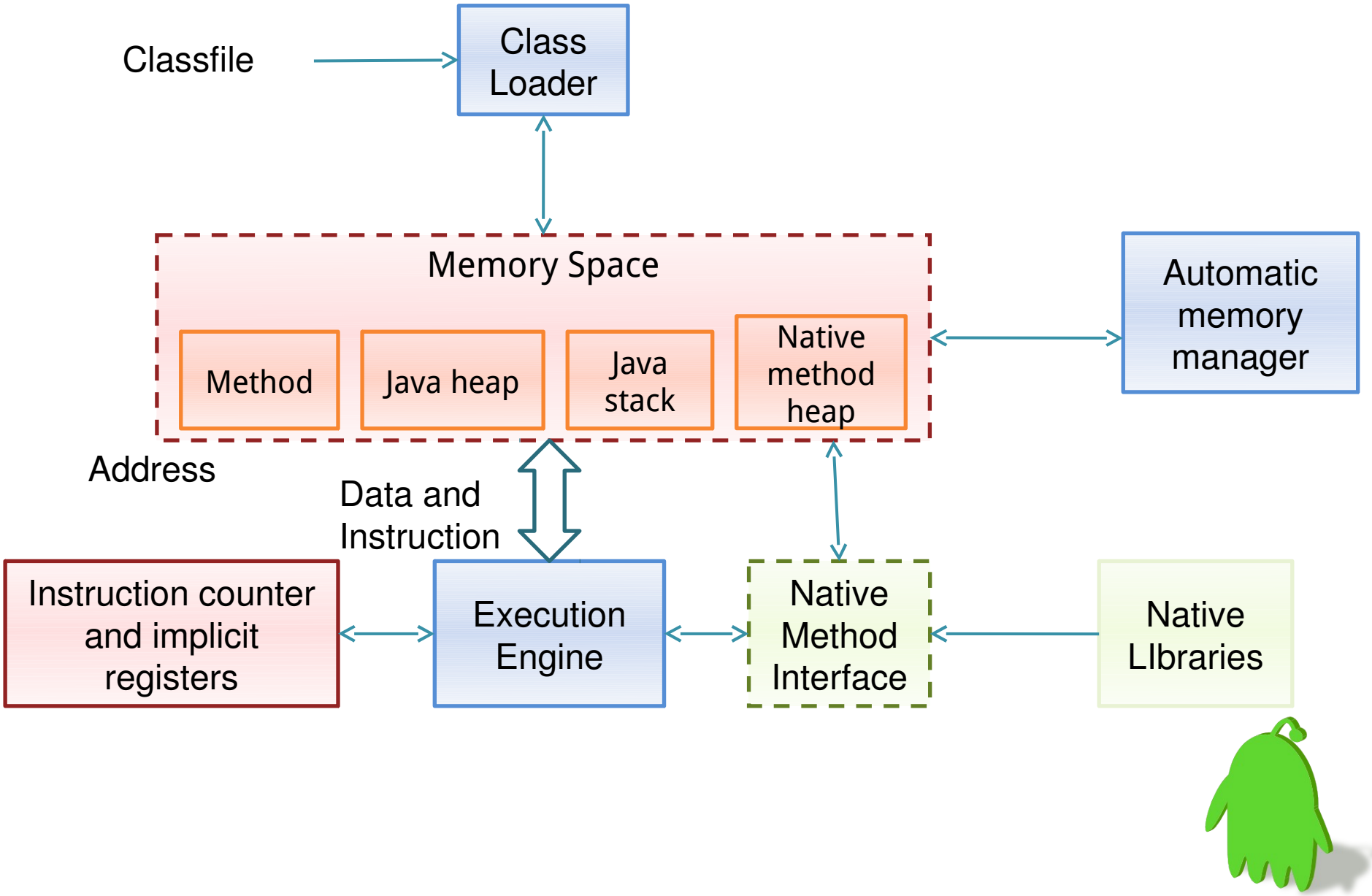
VM Types

- Based on its functionality
  - System Virtual Machine

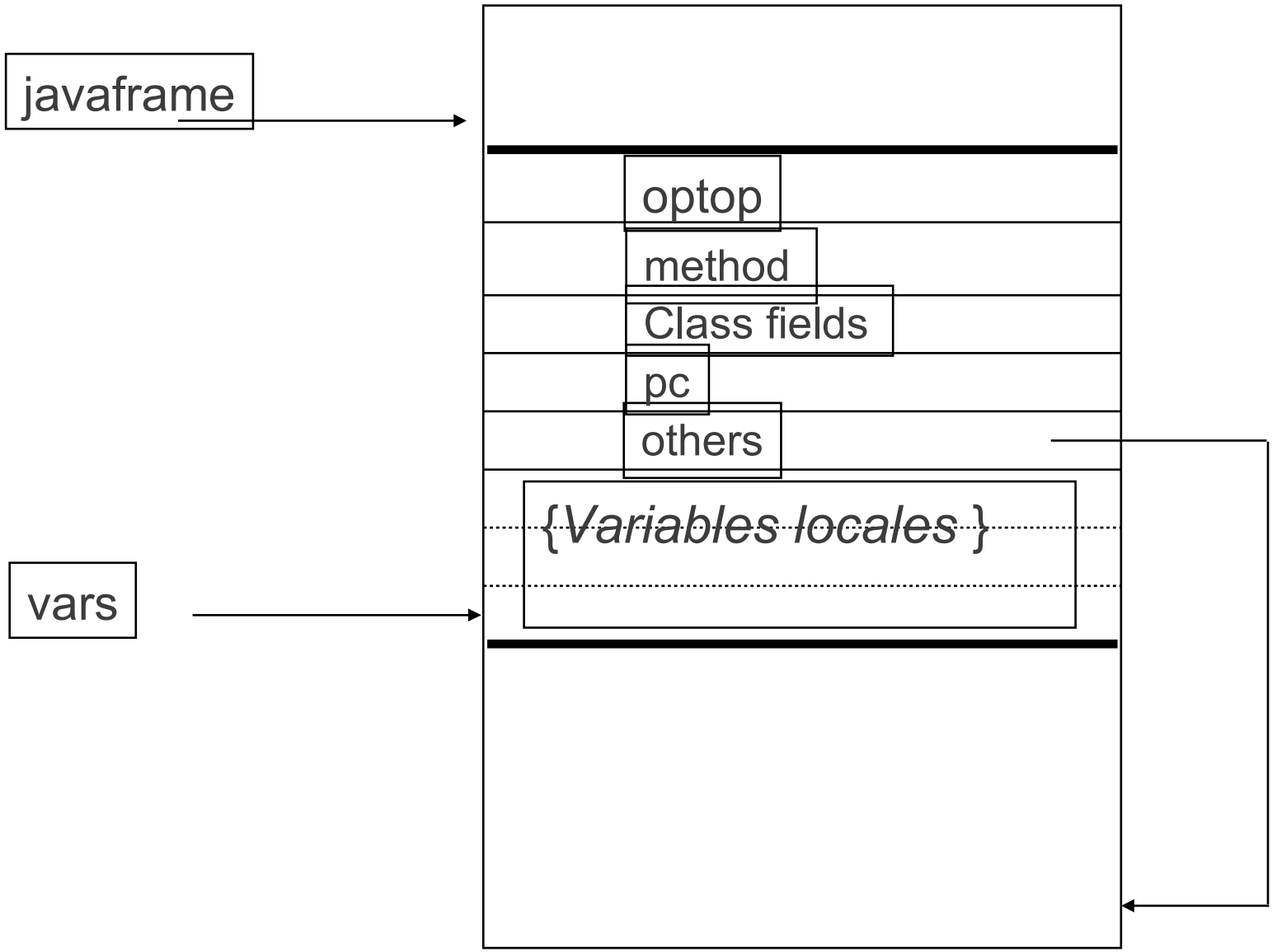
    supports execution of a complete OS
  - Process Virtual Machine

    supports execution of a single process
- Based on its architecture
  - Stack based VM (uses instructions to load in a stack for execution)
  - Register based VM (uses instructions to be encoded in source and destination registers)

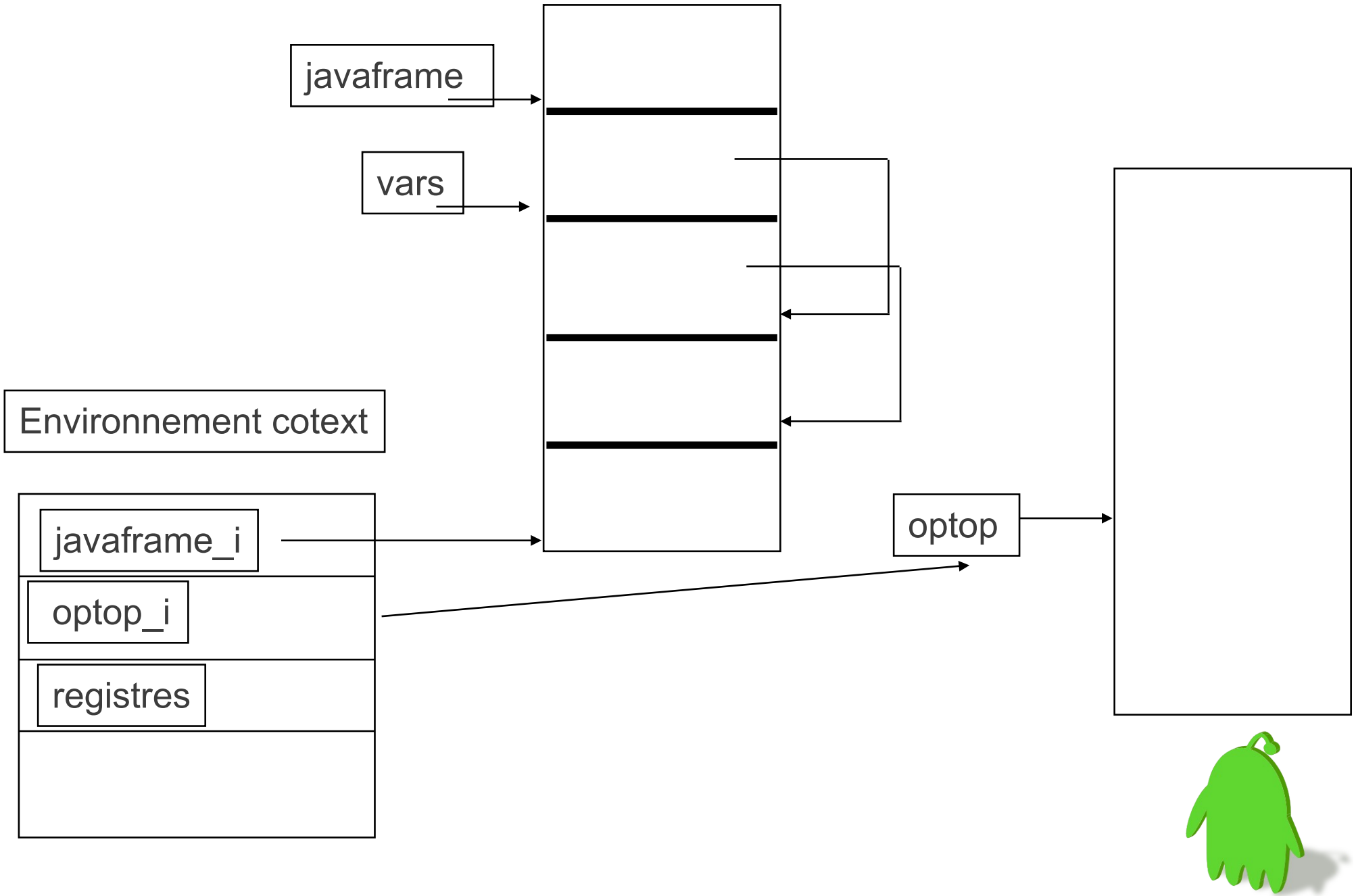
JVM Conceptual Architecture



Segment


javaframe
optop
method
Class fields
pc
others
{Variables locales }
vars

Segment


javaframe
vars
Environnement cotext
javaframe_i
optop_i
registres
optop

Example: JVM

- Example Java source: Foo.java

```
class Foo {
    public static void main(String[] args) {
        System.out.println("Hello, world");
    }
    int calc(int a, int b) {
        int c = 2 * (a + b);
        return c;
    }
}
```

Example: JVM

```
$ javac Foo.java
$ javap -v Foo
Compiled from "Foo.java"
class Foo extends java.lang.Object
...
int calc(int, int);
  Code:
   Stack=3, Locals=4, Args_size=3
   0:   iconst_2
   1:   iload_1
   2:   iload_2
   3:   iadd
   4:   imul
   5:   istore_3
   6:   iload_3
   7:   ireturn
```

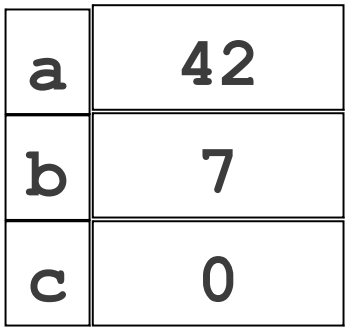
Bytecode execution

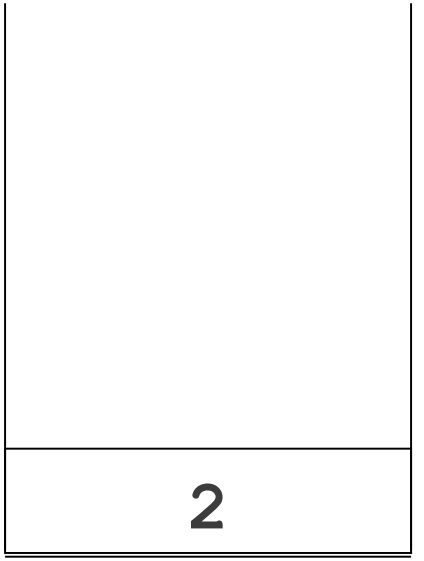
`c := 2 * (a + b)`

- Example bytecode
  - **`iconst 2`**
  - **`iload a`**
  - **`iload b`**
  - **`iadd`**
  - **`imul`**
  - **`istore c`**

- Example bytecode:

```
➡iconst 2
  iload a
  iload b
  iadd
  imul
  istore c
```

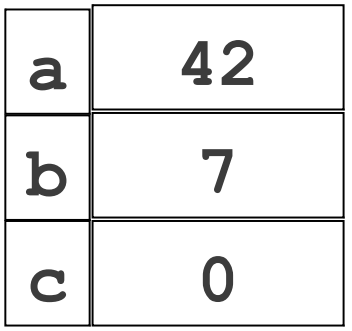


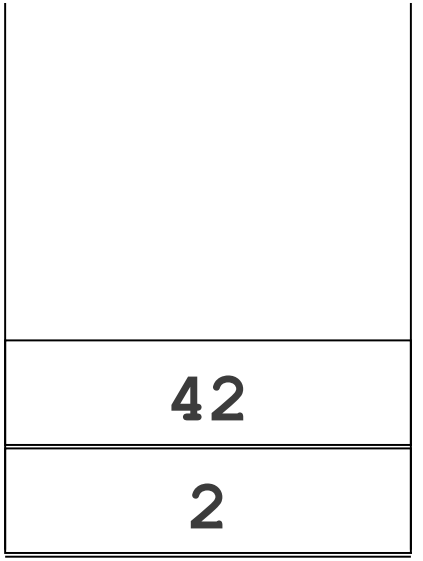


- Computes: `c := 2 * (a + b)`

- Example:

```
   iconst 2
⇨  iload a
   iload b
   iadd
   imul
   istore c
```

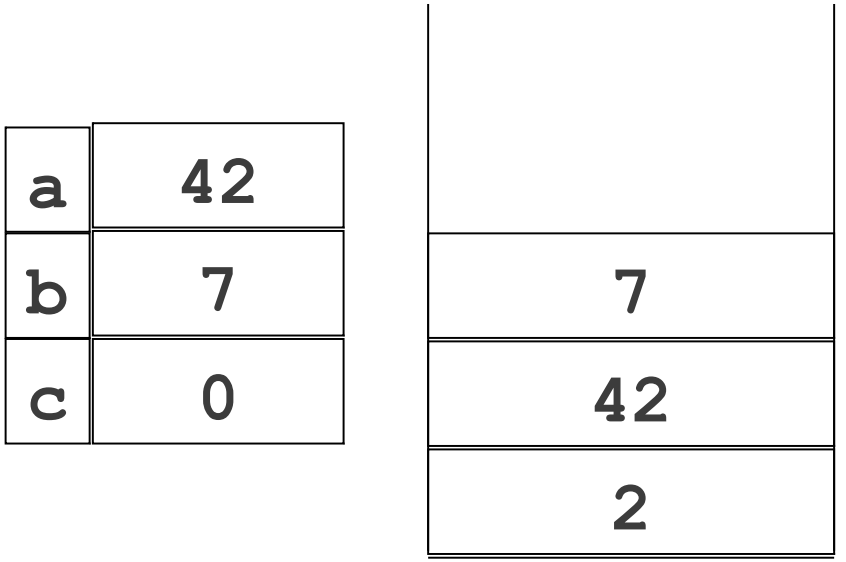




- Computes: `c := 2 * (a + b)`

- Example:

```
iconst 2
iload a
iload b
iadd
imul
istore c
```

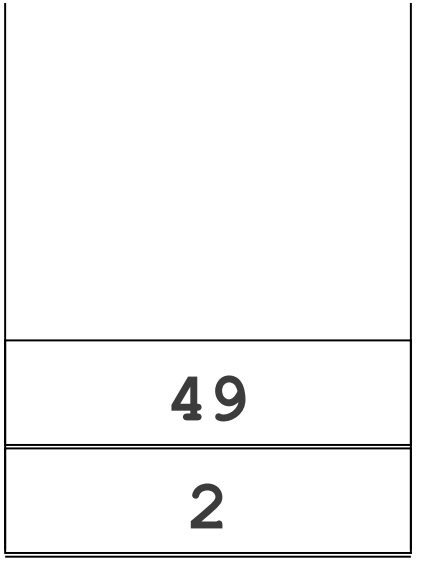


- Computes: `c := 2 * (a + b)`

- Example:

```
iconst 2
iload a
iload b
iadd
imul
istore c
```

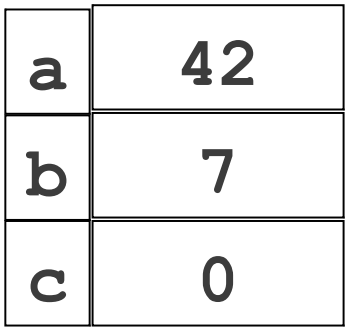
a	42
b	7
c	0

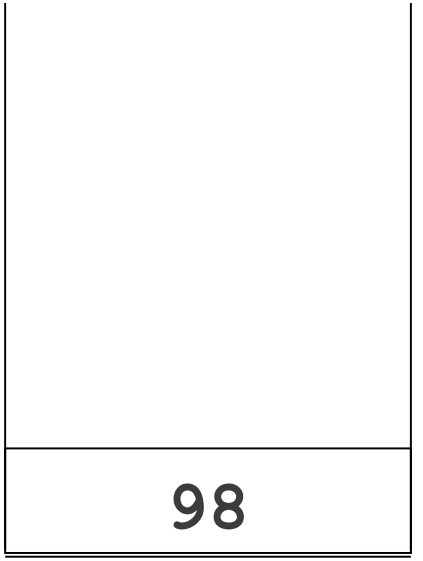


- Computes: `c := 2 * (a + b)`

- Example:

```
    iconst 2
    iload a
    iload b
    iadd
⇨   imul
    istore c
```



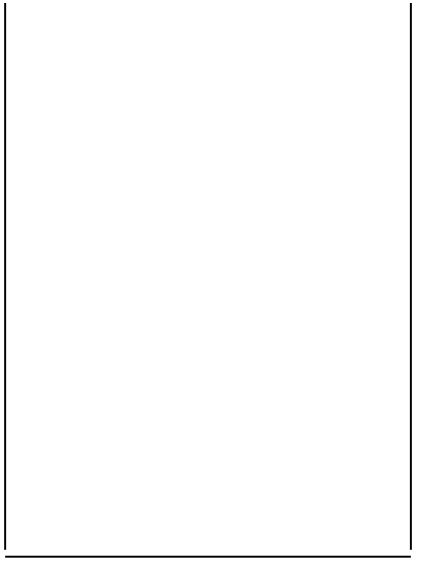


- Computes: `c := 2 * (a + b)`

- Example:

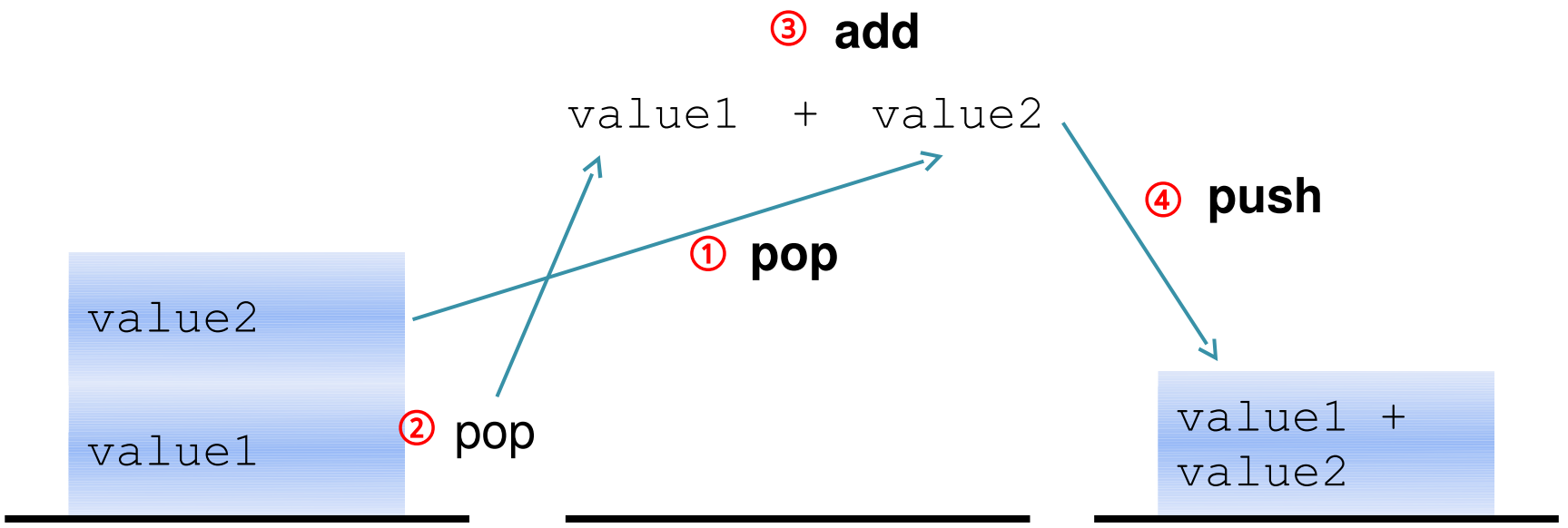
```
    iconst 2
    iload a
    iload b
    iadd
    imul
⇨   istore c
```

a	42
b	7
c	98



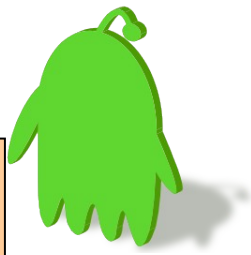
- Computes: `c := 2 * (a + b)`

iadd in specification and implementation




```
case SVM_INSTRUCTION_IADD: {
  /* instruction body */
  jint value1 = stack[stack_size - 2].jint;     ②
  jint value2 = stack[--stack_size].jint;       ①
  stack[stack_size - 1].jint = value1 +
                               value2;          ④ ③

  /* dispatch */
  goto dispatch;
}
```

Taken from SableVM
sablevm/src/libsablevm/instructions_switch.c



Example: Dalvik VM

```
$ dx --dex --output=Foo.dex Foo.class
$ dexdump -d Foo.dex
Processing 'Foo.dex'...
Opened 'Foo.dex', DEX version '035'
...
  Virtual methods   -
    #0              : (in LFoo;)
      name          : 'calc'
      type          : '(II)I'
...
00018c:                 |[00018c] Foo.calc:(II)I
00019c: 9000 0203       |0000: add-int v0, v2, v3
0001a0: da00 0002       |0002: mul-int/lit8 v0, v0, #int 2
0001a4: 0f00            |0004: return v0
```

Java bytecode vs. Dalvik bytecode

(stack vs. register)

```
public int method(int i1, int i2)
{
    int i3 = i1 * i2;
    return i3 * 2;
}
```

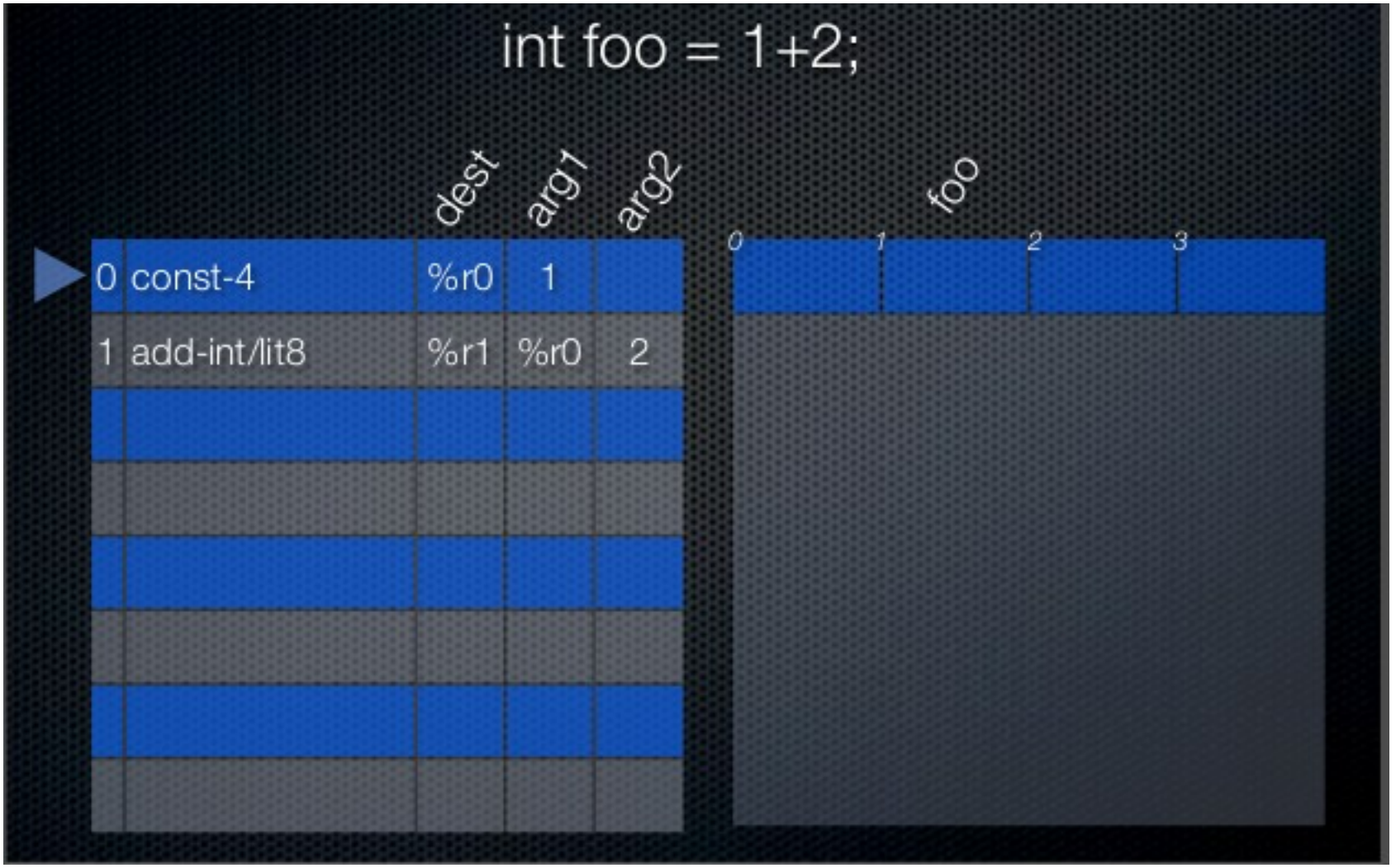
```
.var 0 is "this"
.var 1 is argument #1
.var 2 is argument #2

method public method(II)I
    iload_1
    iload_2
    imul
    istore_3
    iload_3
    iconst_2
    imul
    ireturn
.end method
```

Java

```
this: v1 (Ltest2;)
parameter[0] : v2 (I)
parameter[1] : v3 (I)

.method public method(II)I
    mul-int v0,v2,v3
    mul-int/lit-8 v0,v0,2
    return v0
.end method
```

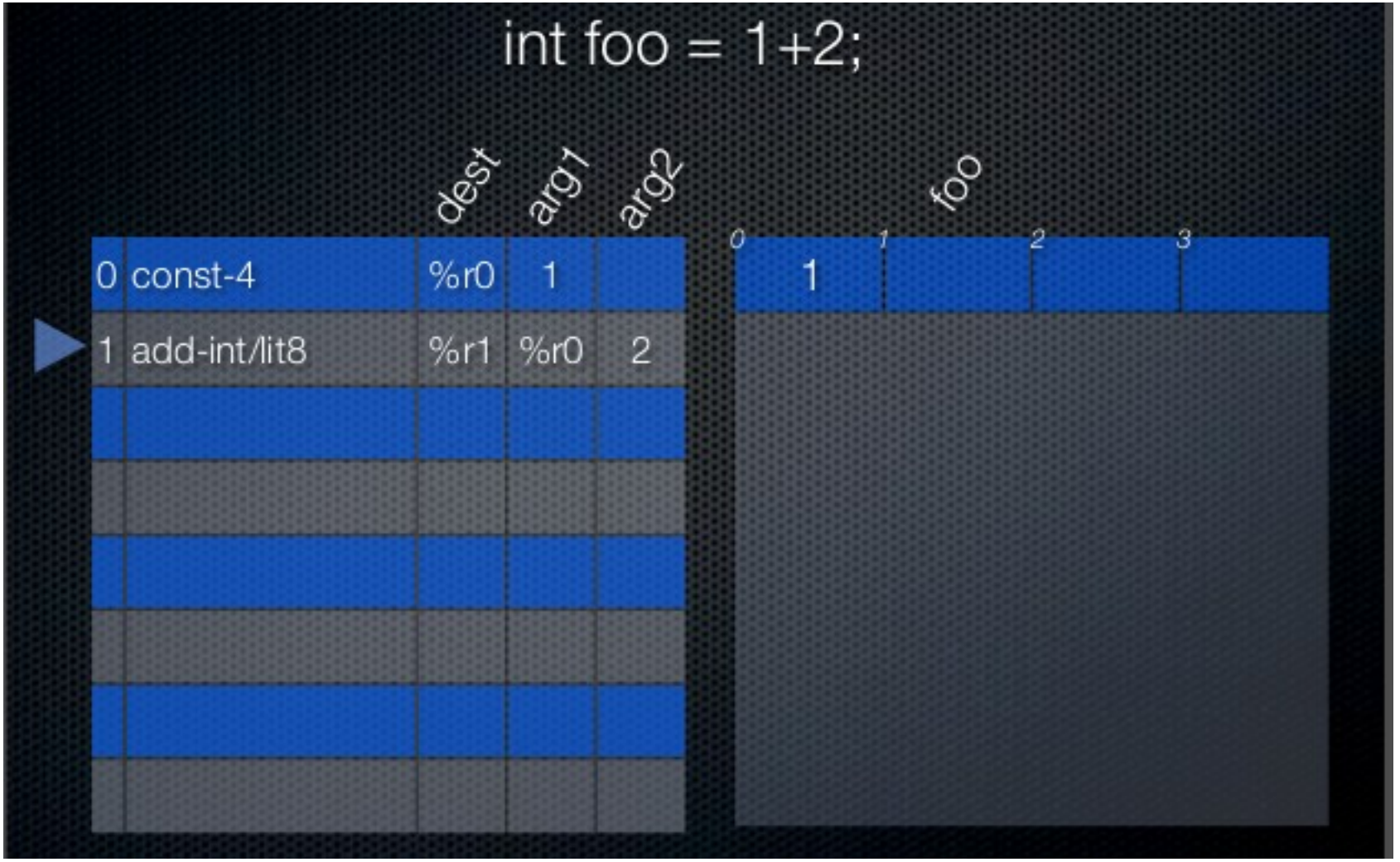
Dalvik

Dalvik is register based



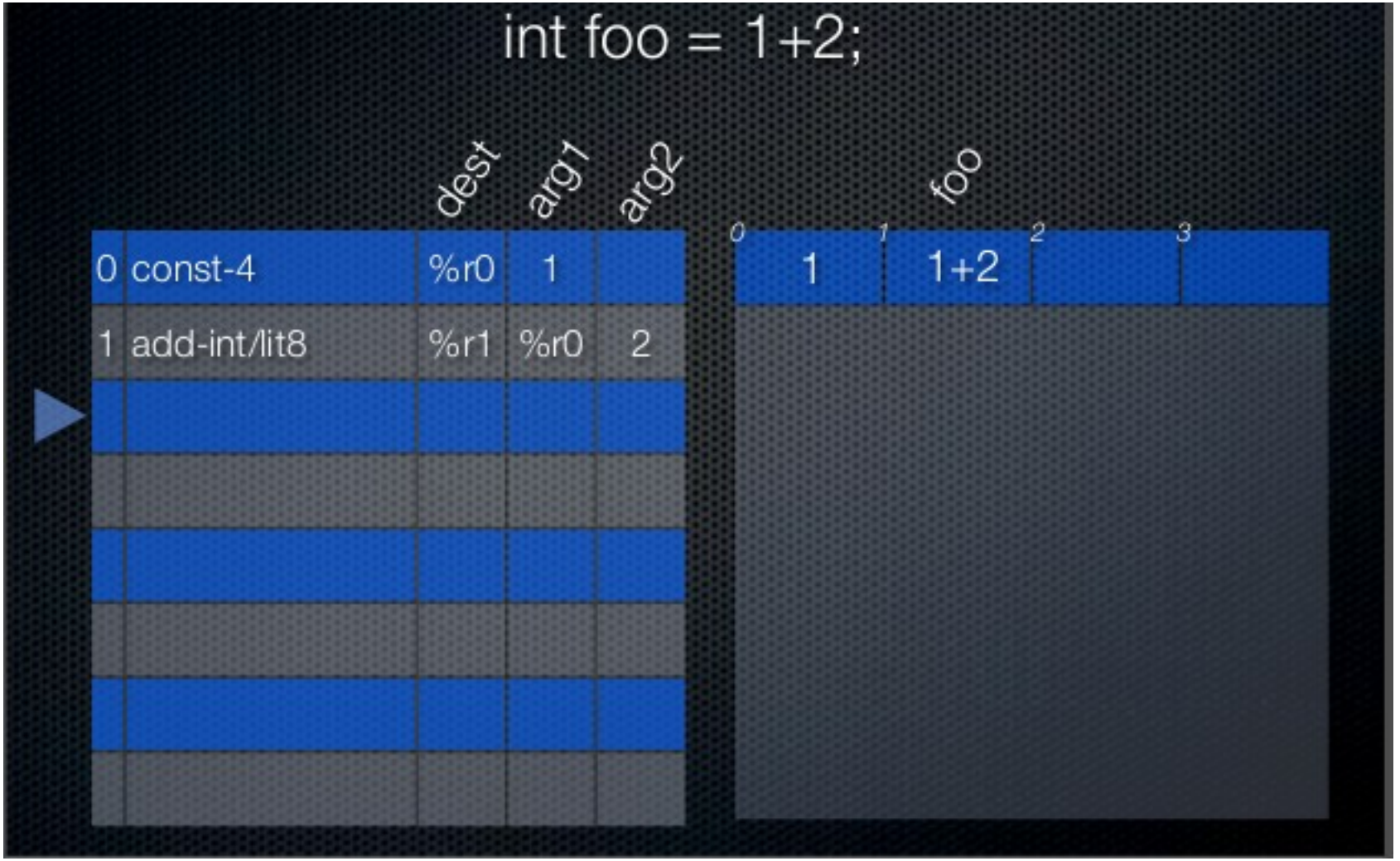
- Dalvik uses 3-operand form, which it what a processoractually uses

Dalvik is register based



- To execute "int foo = 1 + 2", the VM does:
  - const-4 to store 1 into register 0
  - add-int/lit8 to sum the value in register 0 (1) with the literal 2 and store the result intoregister 1 -- namely “foo”

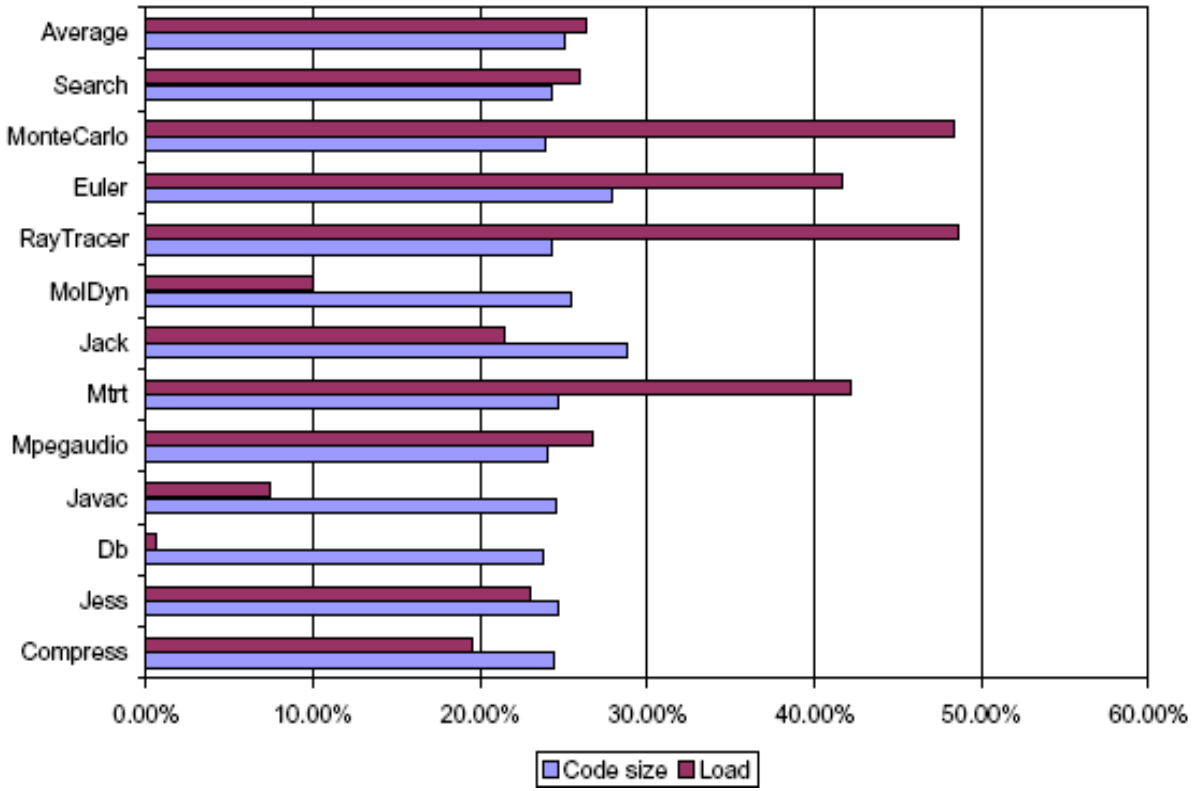
Dalvik is register based



- This is only 2 dispatches, but Dalvik byte code is measured into 2-byte units
- Java byte code was 4-bytes, the Dalvik byte code is actually 6-bytes

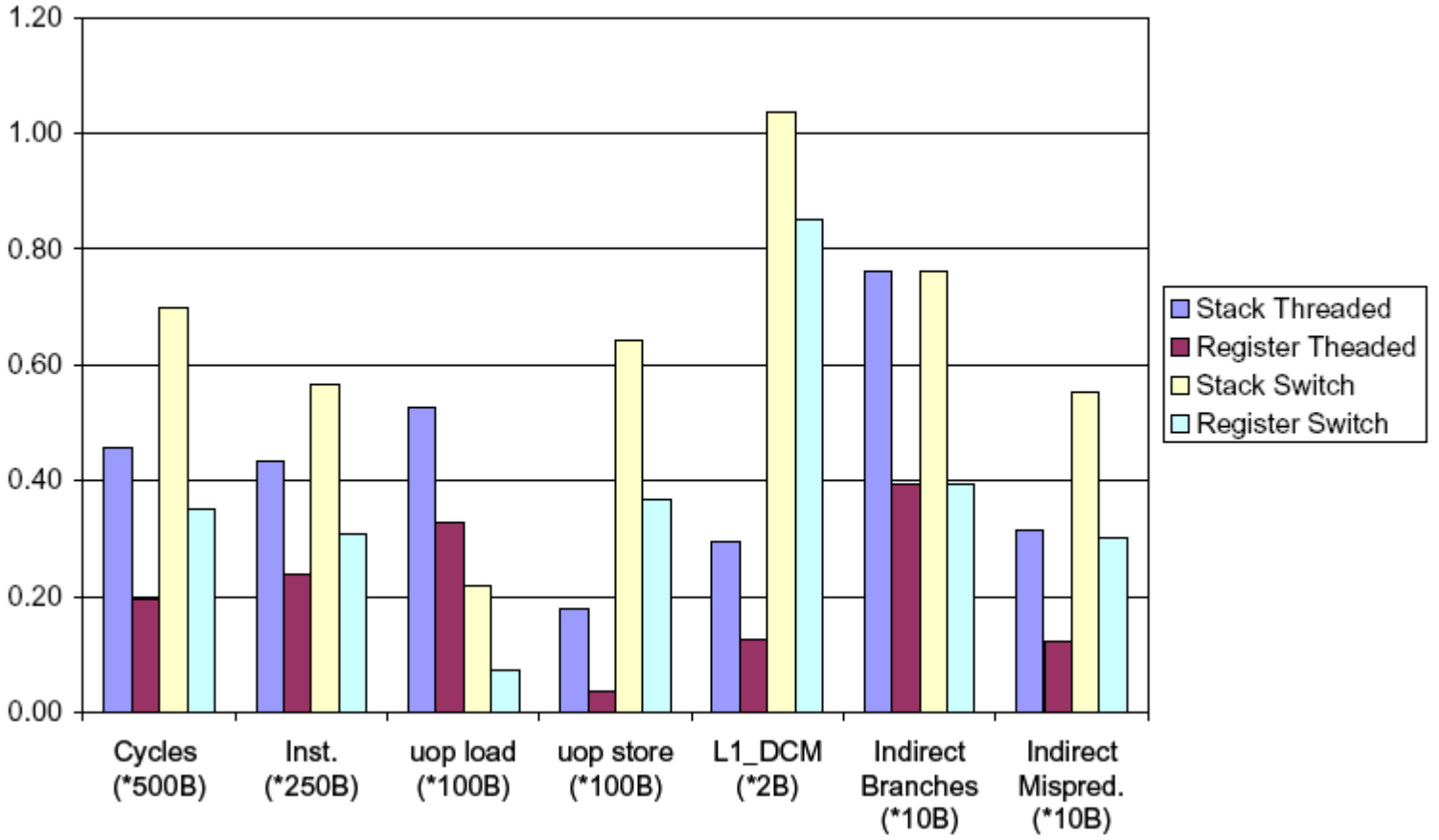
Code Size

- Generally speaking, the code size of register-based VM instructions is larger than that of the corresponding stack VM instructions
- On average, the register code is 25.05% larger than the original stack code



Execution Time

- Register architecture requires an average of 47% fewer executed VM instructions



Source: Virtual Machine Showdown: Stack Versus Registers
Yunhe Shi, David Gregg, Andrew Beatty, M. Anton Ertl

Dalvik-JVM

- Instruction traslation
  - one Dalvik instruction → multiple Java instructions

Dalvik	Java
`add-int` d_0, s_0, s_1	`iload` s'_0
	`iload` s'_1
	`iadd`
	`istore` d'_0

Dalvik VM Execution on ARM Target

```
$ dx --dex --output=foo.jar Foo.class
$ adb push foo.jar /data/local/
$ adb shell dalvikvm \
    -classpath /data/local/foo.jar Foo
Hello, world
```

Instruction Dispatch

```
static void interp(const char* s) {
   for (;;) {
       switch (*(s++)) {
             case 'a': printf (“Hello"); break;
             case 'b': printf (“ "); break;
             case 'c": printf (“world!"); break;
             case 'd': prinf (”\n"); break;
             case 'e': return;
}
int main (int argc, char** argv) {
    interp("abcbde"):
}
```

Computed GOTO (in GCC's way)

```
#define DISPATCH() \
    { goto *op_table[*((s)++) - 'a']; }

static void interp(const char* s) {
  static void* op_table[] =
     { &&op_a, &&op_b, &&op_c, &&op_d, &&op_e };
  DISPATCH();
  op_a: printf((”Hello"); DISPATCH();
  op_b: printf (“ "); DISPATCH();
  op_c: printf (”world!"); DISPATCH();
  op_d: prinf (”\n"); DISPATCH();
  op_e: return;
}
```

Best Dispatch Implementation

- The computed GOTO can be further optimized if we re-write it in assembly.
- The code above uses typically two memory reads. We can lay out all our bytecodes in memory in such a way that each bytecode takes exactly the same amount of memory - this way we can calculate the address directly from the index.
- Added benefit is the cacheline warm-up for frequently used bytecodes.

Class 文件所记录的信息

- 结构信息
  - Class 文件格式版本号
  - 各部分的数量与大小
- 元数据
  - 类 / 继承的超类 / 实现的接口的声明信息
  - 域与方法声明信息
  - 常量池
  - 用户自定义的、 RetentionPolicy为 CLASS 或 RUNTIME 的注解
  - —对应 Java 源代码中“声明”与“常量”对应的信息
- 方法信息
  - 字节码
  - 异常处理器表
  - 操作数栈与局部变量区大小
  - 操作数栈的类型记录（ StackMapTable，Java 6 开始）
  - 调试用符号信息（如 LineNumberTable、LocalVariableTable ）
  - —对应 Java 源代码中“语句”与“表达式”对应的信息

Source: Java Program in Action——Java 程序的编译、加载与执行， 莫枢

Class 文件例子

```
import java.io.Serializable;

public class Foo implements Serializable {
    public void bar() {
        int i = 31;
        if (i > 0) {
            int j = 42;
        }
    }
}
```

结构：
声明与常量

代码：
语句与表达式

输出调试符号信息

编译 Java 源码

```
javac -g Foo.java
```

反编译 Class 文件

```
javap -c -s -l -verbose Foo
```

Class 文件例子

方法
元数据

字节码

```
public Foo();
  Signature: ()V
  LineNumberTable:
   line 2: 0

  LocalVariableTable:
   Start  Length  Slot  Name   Signature
   0      5       0    this       LFoo;

  Code:
   Stack=1, Locals=1, Args_size=1
   0:   aload_0
   1:   invokespecial   #1; //Method java/lang/Object."<init>":()V
   4:   return
```

Class 文件例子

方法
元数据

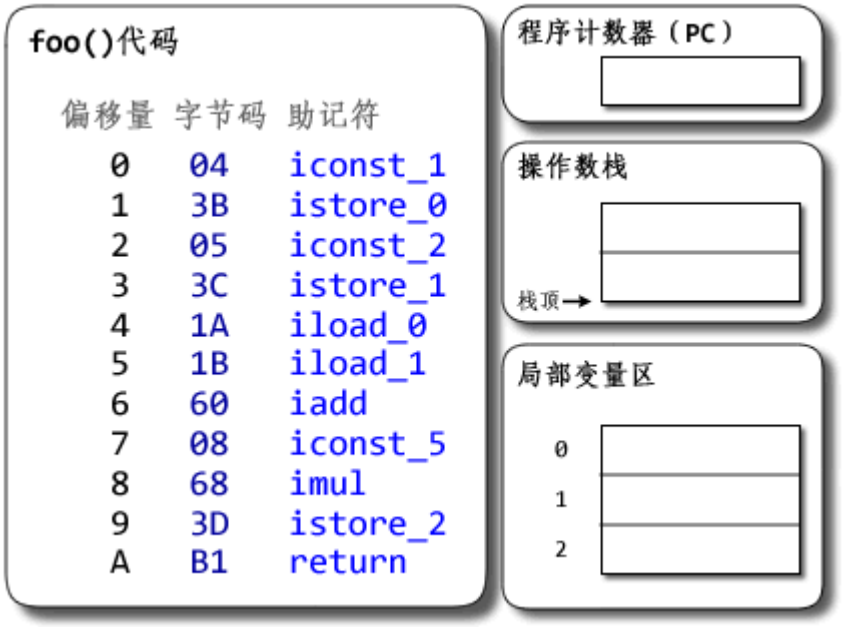
```
public void bar();
  Signature: ()V
  LineNumberTable:
   line 4: 0
   line 5: 3
   line 6: 7
   line 8: 10

  LocalVariableTable:
   Start  Length  Slot  Name
Signature
   10      0      2    j       I
   0      11      0    this
LFoo;
   3      8      1    i       I

  StackMapTable: number_of_entries = 1
   frame_type = 252 /* append */
     offset_delta = 10
     locals = [ int ]
```

Java 6 开始，有分支控制流的方法会带有 StackMapTable，记录每个基本块开头处操作数栈的类型状态

字节码

```
  Code:
   Stack=1, Locals=3, Args_size=1
   0:     bipush    31
   2:     istore_1
   3:     iload_1
   4:     ifle 10
   7:     bipush    42
   9:     istore_2
   10:    return
```

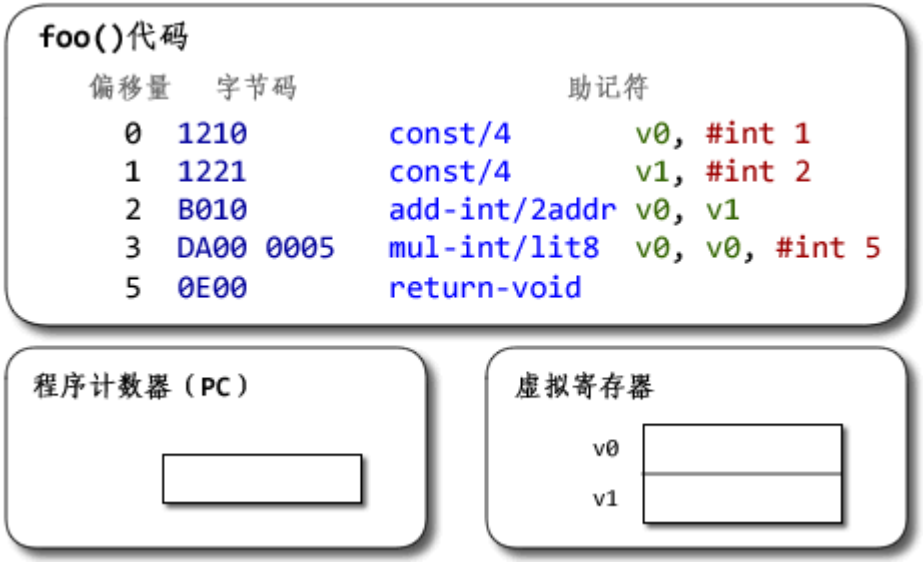
基于栈与基于寄存器的体系结构的区别



概念中的 Java 虚拟机

```
public class Demo {
    public static void foo() {
        int a = 1;
        int b = 2;
        int c = (a + b) * 5;
    }
}
```

概念中的 Dalvik 虚拟机



Source: Java Program in Action——Java 程序的编译、加载与执行， 莫枢

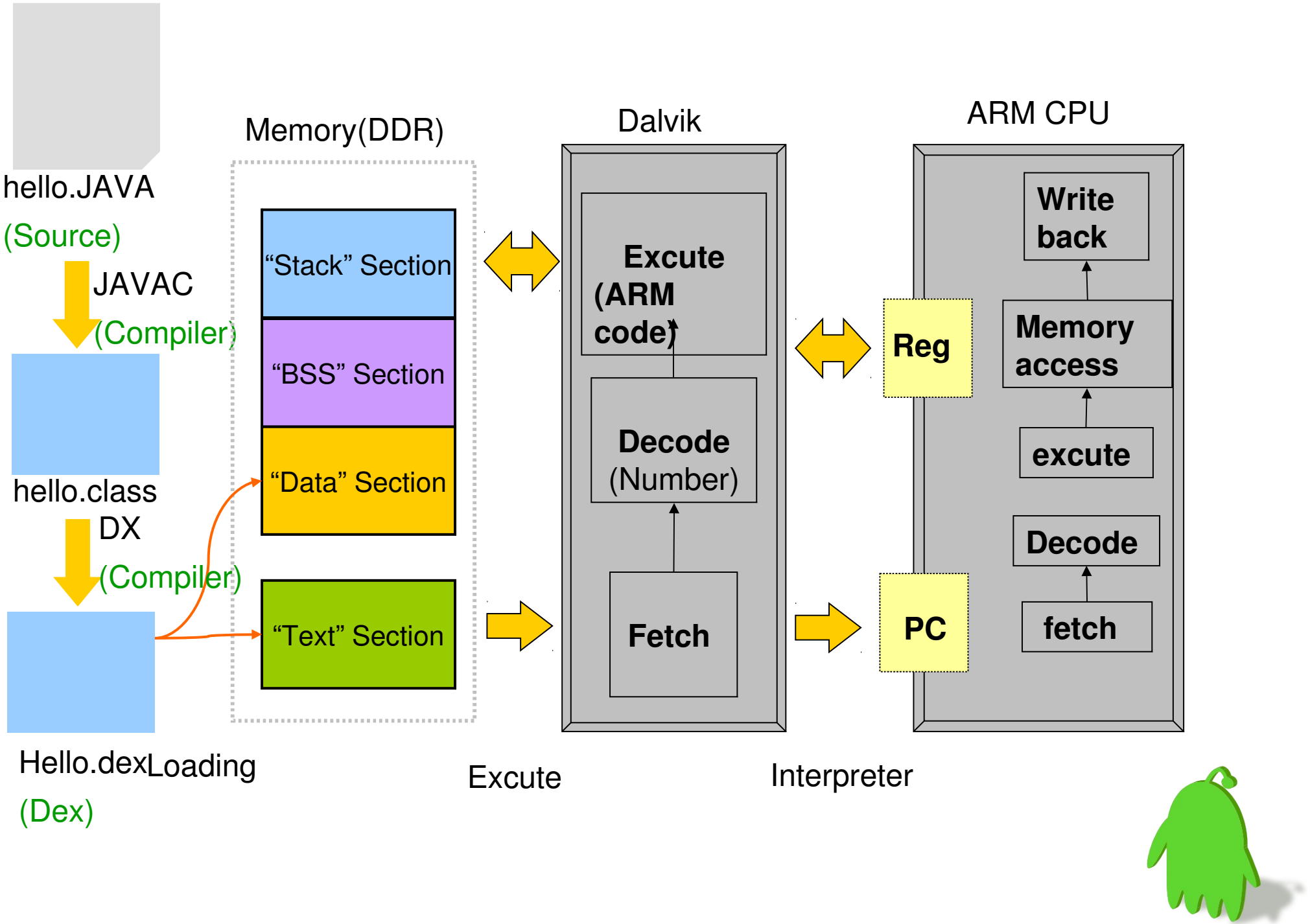
Dalvik VM

Dalvik VM

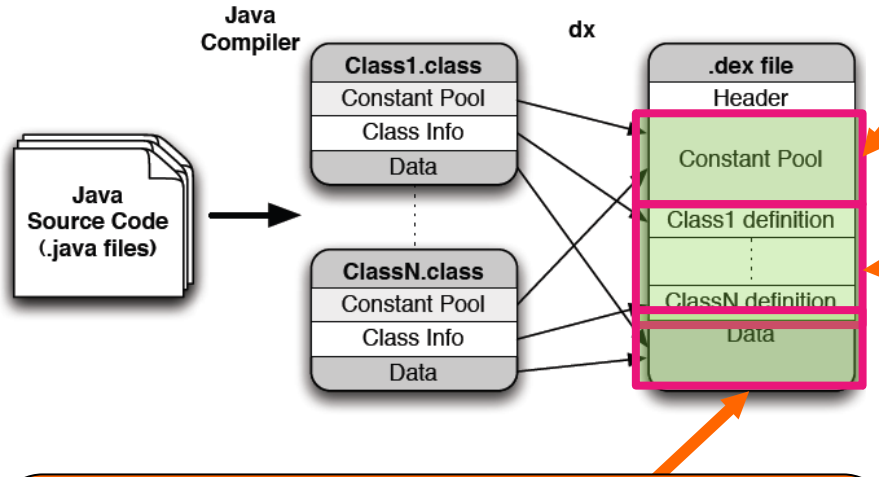
- Dalvik architecture is register based
- Optimized to use less space
- Execute its own Dalvik byte code rather than Java byte code
- Class Library taken from Apache Harmony
  - A compatible, independent implementation of the Java SE 5 JDK under the Apache License v2
  - A community-developed modular runtime (VM and class library) architecture. (deprecated now)

Reasons to Choose Dalvik

- Dalvik (Register based) take average 47 % less executed VM instruction then JVM (Stack based).
- Register code is 25% larger than the corresponding stack code.
- This increased cost of fetching more VM instructions due to larger code size involves only 1.07% extra real machine loads per VM instruction. Which is negligible.
- Some Marketing Reasons too
  - Oracle lawsuit against Google


hello.JAVA
(Source)
JAVAC
(Compiler)
hello.class
DX
(Compiler)
Hello.dex
(Dex)
Loading
Memory(DDR)
“Stack” Section
“BSS” Section
“Data” Section
“Text” Section
Excute
Dalvik
Excute (ARM code)
Decode (Number)
Fetch
Interpreter
ARM CPU
Reg
PC
Write back
Memory access
excute
Decode
fetch

Constant Pool:
References to other classes
Method names
Numerical constants



Class Definition:
Access flags
Class names

Data:
Method code
Info related to methods
Variables

Dalvik Architecture

- Register architecture
- 2^{16} available registers
- Instruction set has 218 opcodes
  - JVM: 200 opcodes
- 30% fewer instructions, but 35% larger code size (bytes) compared to JVM

Constant Pool

- Dalvik
  - Single pool
  - dx eliminates some constants by inlining their values directly into the bytecode
- JVM
  - Multiple

Primitive Types

- Ambiguous primitive types
  - Dalvik

    int/float, long/double use the same opcodes

    does not distinguish : int/float, long/double, 0/null.
  - JVM

    Different: JVM is typed
- Null references
  - Dalvik

    Not specify a null type

    Use zero value

Object Reference

- Comparison of object references
- Dalvik
  - Comparison between two integers
  - Comparison of integer and zero
- JVM
  - if_acmpeq / if_acmpne
  - ifnull / ifnonnull

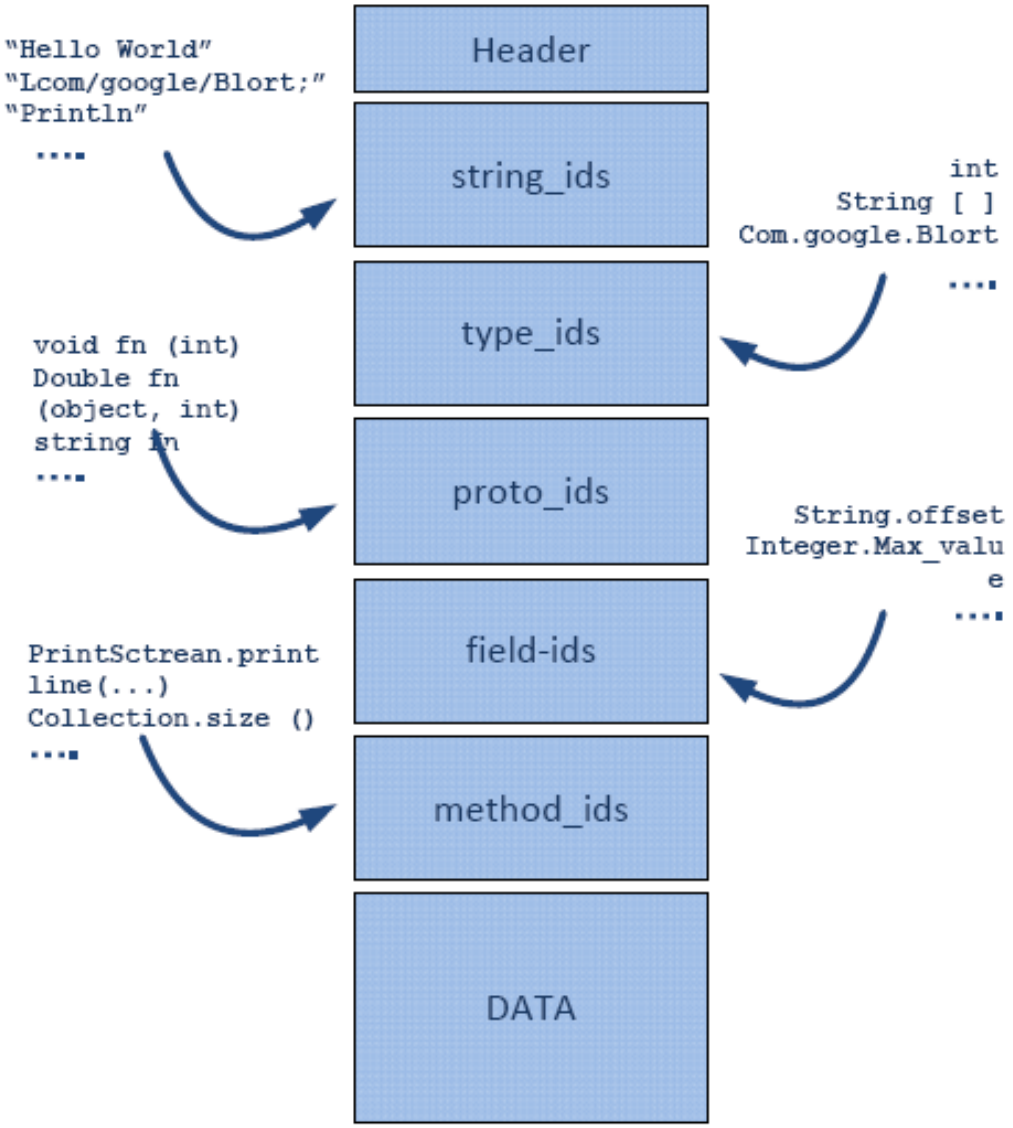
Dalvik

- Storage of primitive types in arrays
- Dalvik
  - Ambiguous opcodes
  - aget for int/float, aget-wide for long/double

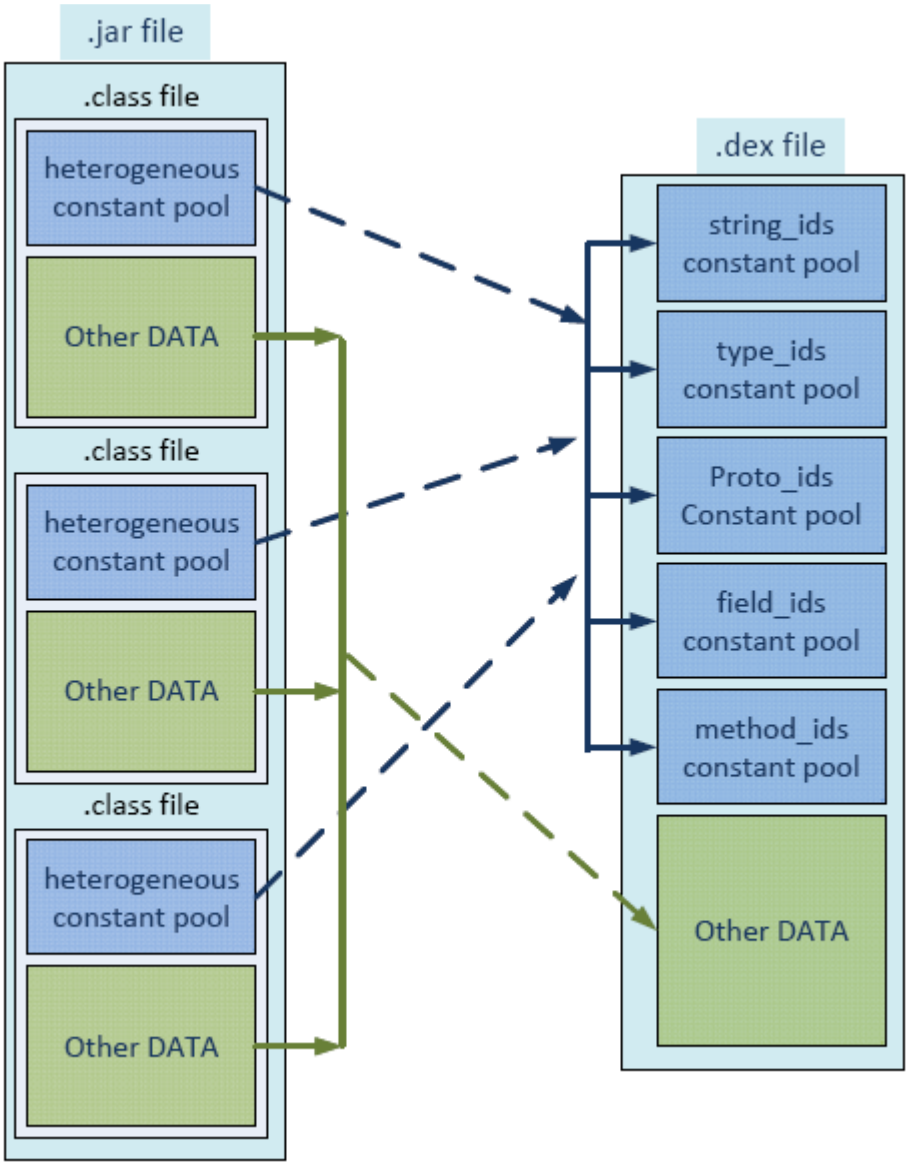
Dalvik

- Dalvik uses annotation to store:
  - signature
  - inner class
  - Interface
  - Throw statement.
- Dalvik is more compact, average of 30% less instructions than JVM.

DEX File Anatomy



Map Java bytecode to Dalvik bytecode



Java bytecode vs. Dalvik bytecode

```
public class Demo {
    private static final char[] DATA = {
        'A','m','b','e','r',
        ' ','u','s','e','s',' ',
        'A','n','d','r','o','i','d'
    };
}
```

Java

```
  0: bipush  18
  2: newarray char
  4: dup
  5: iconst_0
  6: bipush  65
  8: castore
    ...
101: bipush  17
103: bipush  100
105: castore
106: putstatic #2; // DATA
109: return
```

Dalvik

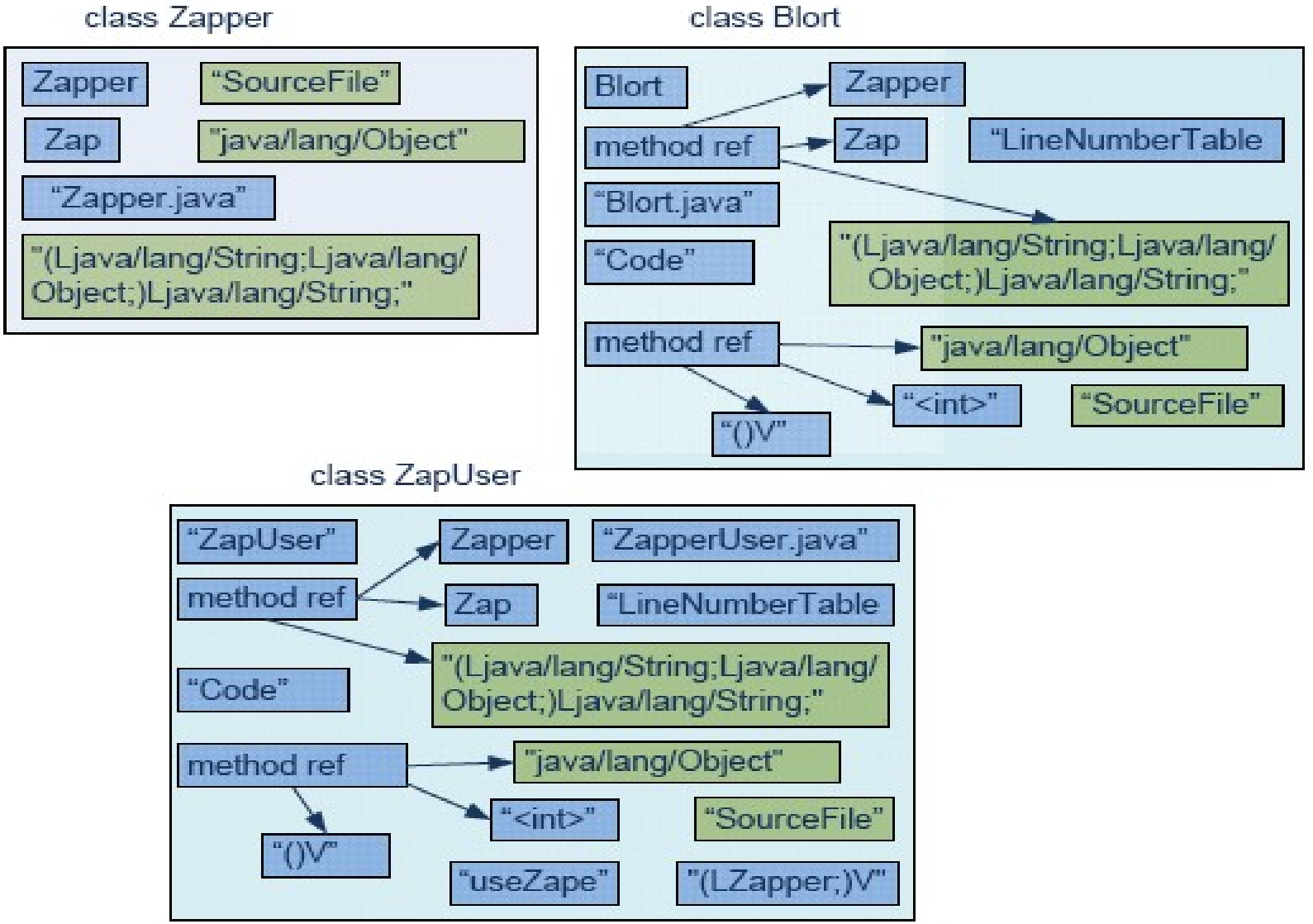
```
|0000: const/16 v0, #int 18
|0002: new-array v0, v0, [C
|0004: fill-array-data v0,
           0000000a
|0007: sput-object v0,
           LDemo;.DATA:[C
|0009: return-void
|000a: array-data (22 units)
```

Shared constant pool

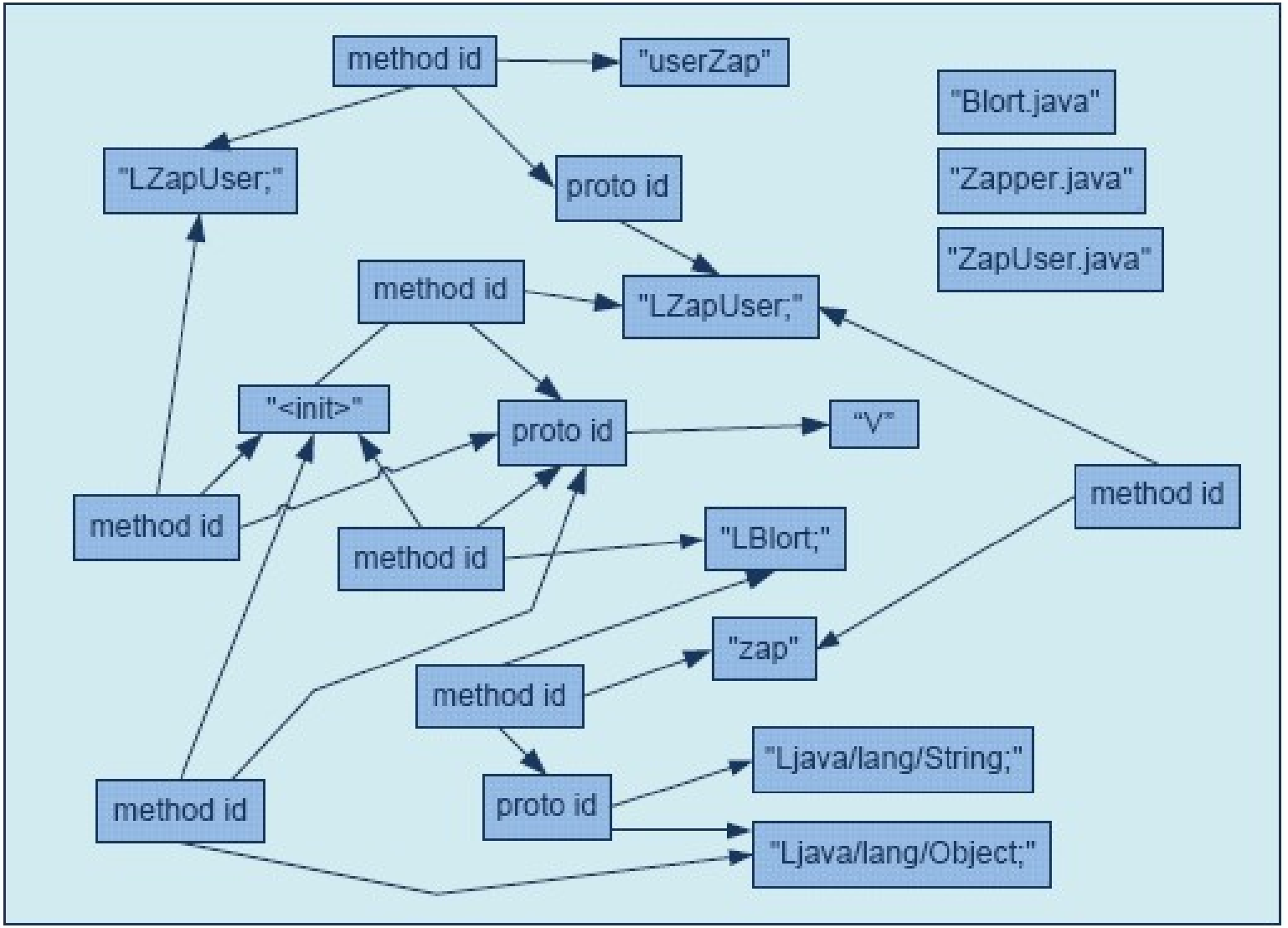
- Zapper.java

```
public interface Zapper {
    public String zap(String s, Object o);
}
public class Blort implements Zapper {
    public String zap(String s, Object o) { … }
}
public class ZapUser {
    public void useZap(Zapper z) { z.zap(...); }
}
```

Shared constant pool



Shared constant pool



Shared constant pool (memory usage)

- minimal repetition
- per-type pools (implicit typing)
- implicit labeling

Contents	Uncompressed jar File		Compressed jar files		Uncompressed dex file	
	In Bytes	In %	In Bytes	In %	In Bytes	In %
Common System Libraries	21445320	100	10662048	50	10311972	48
Web browser Application	470312	100	232065	49	209248	44
Alarm Check Application	119200	100	61658	52	53020	44

```
public static long sumArray(int[] arr) {
    long sum = 0;
    for (int i : arr) {
        sum += i;
    }
    return sum;
}
```

Java class

```
0000: lconst_0
0001: lstore_1
0002: aload_0
0003: astore_3
0004: aload_3
0005: arraylength
0006: istore 04
0008: iconst_0
0009: istore 05
000b: iload 05          // rl ws
000d: iload 04          // rl ws
000f: if_icmpge 0024    // rs rs
0012: aload_3           // rl ws
0013: iload 05          // rl ws
0015: iaload            // rs rs ws
0016: istore 06         // rs wl
0018: lload_1           // rl rl ws ws
0019: iload 06          // rl ws
001b: i2l               // rs ws ws
001c: ladd              // rs rs rs rs ws ws
001d: lstore_1          // rs rs wl wl
001e: iinc 05, #+01     // rl wl
0021: goto 000b
0024: lload_1
0025: lreturn
```

- 25 bytes
- 14 dispatches
- 45 reads
- 16 writes

read stack

write stack

read local

write local

```
public static long sumArray(int[] arr) {
    long sum = 0;
    for (int i : arr) {
        sum += i;
    }
    return sum;
}
```

	bytes	dispatches	reads	writes
.class	25	14	45	16
.dex	18	6	19	6

Dalvik DEX

- 18 bytes
- 6 dispatches
- 19 reads
- 6 writes

```
0000: const-wide/16 v0, #long 0
0002: array-length v2, v8
0003: const/4 v3, #int 0
0004: move v7, v3
0005: move-wide v3, v0
0006: move v0, v7
0007: if-ge v0, v2, 0010          // r r
0009: aget v1, v8, v0             // r r w
000b: int-to-long v5, v1          // r w w
000c: add-long/2addr v3, v5       // r r r r w w
000d: add-int/lit8 v0, v0, #int 1 // r w
000f: goto 0007
0010: return-wide v3
```

Comparison between Dalvik VM and JVM

- Memory Usage Comparison
- Architecture Comparison
- Supported Libraries Comparison

Libraries	Dalvik	Standard Java
java.io	Y	Y
java.net	Y	Y
android.*	Y	N
com.google.*	Y	N
javax.swing.*	N	Y
...	...	...

- Reliability Comparison
- Multiple instance and JIT Comparison
- Concurrent GC

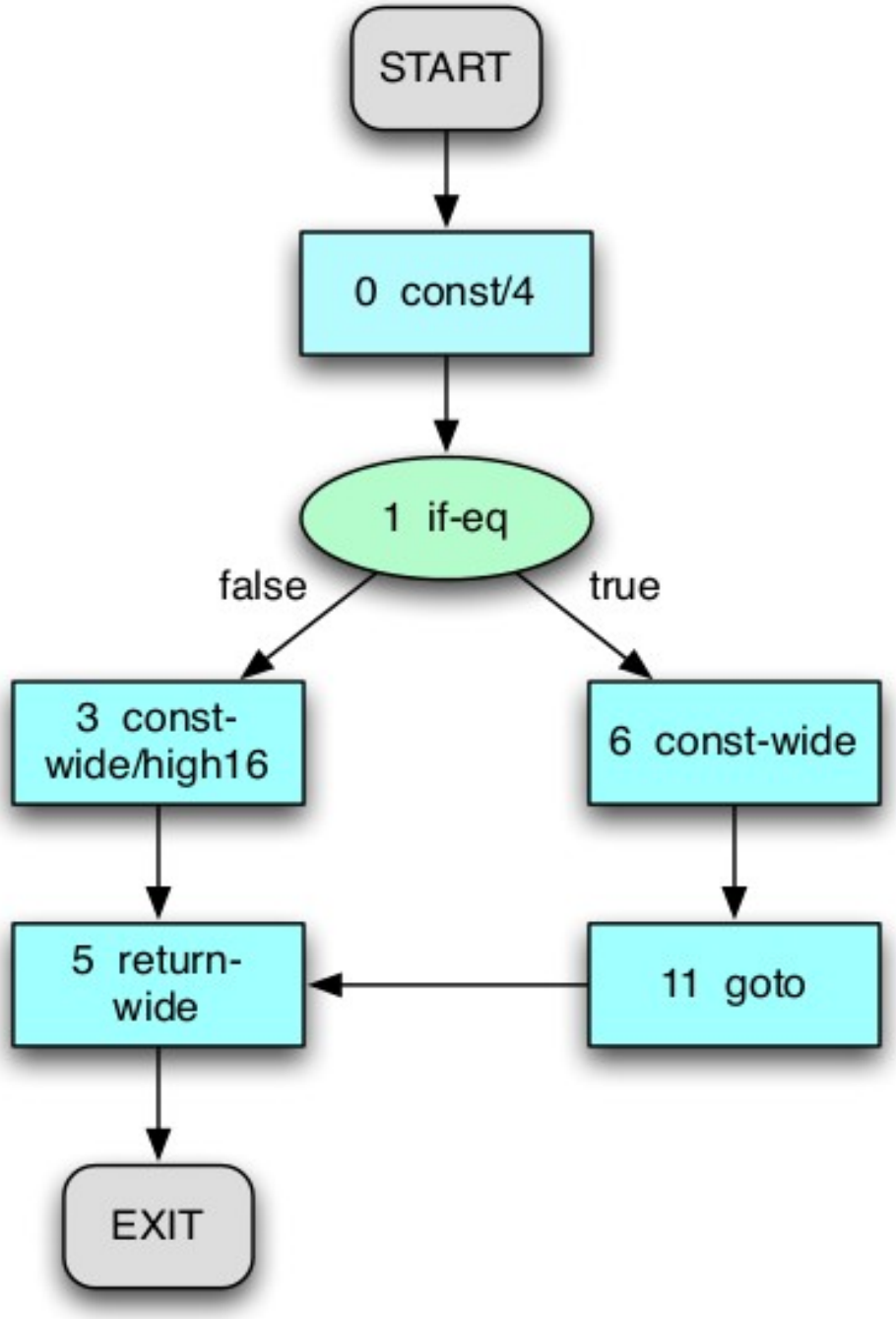
AST to Bytecode

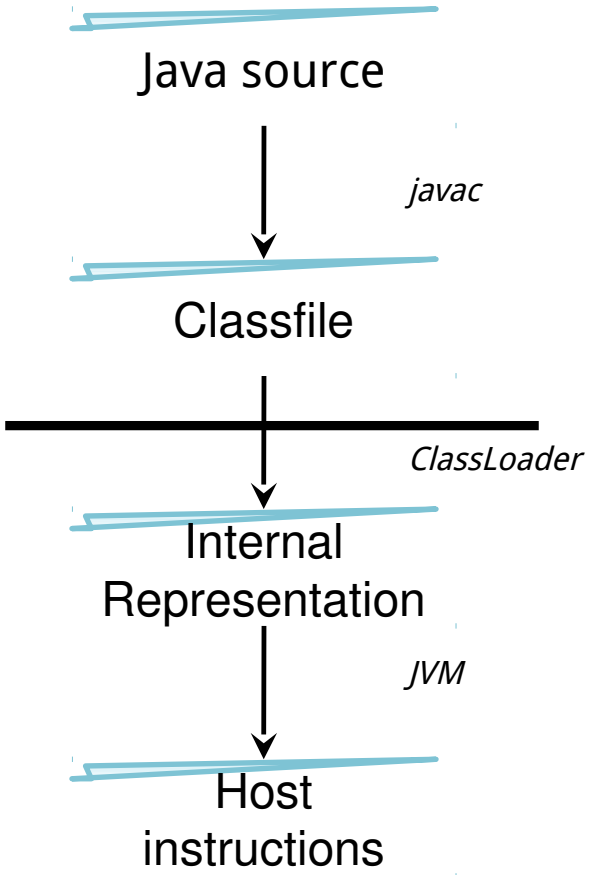
Java source

```
double return_a_double(int a) {
    if (a != 1)
        return 2.5;
    else
        return 1.2;
}
```

DEX Bytecode

```
double return_a_double(int)
0: const/4 v0,1
1: if-eq v3,v0,6
3: const-wide/high16 v0,16388
5: return-wide v0
6: const-wide v0,4608083138725491507
11: goto 5
```






```
public class ZZZZ {
  private int value;
  public void foo() {
    int v = this.value;
  }
}
```

```
…
2a      /* aload_0                          */
b40002  /* getfield #2; //Field value:I     */
3c      /* istore_1                         */
…
```

```
Class ZZZZ

  …

 methods

  …
```

```
Method foo()V
…
dcd00100 /* fast_iaccess_0 #1 */
3c       /* istore_1         */
…
```

```
…
8b4108 ; mov eax,dword ptr ds:[ecx+8]
…
```

Efficient Interpreter in Android

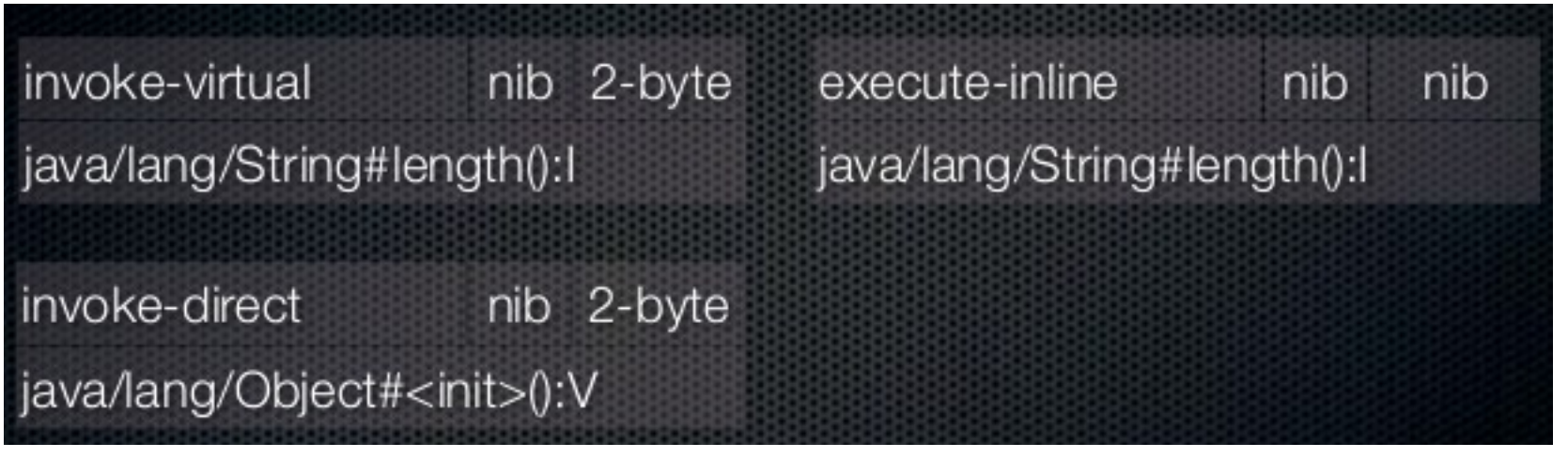
- There are 3 forms of Dalvik
  - dexopt: optimized DEX
  - Zygote
  - libdvm + JIT

Efficient Interpreter: Optimized DEX

- Apply platform-specific optimizations:
  - specific bytecode
  - vtables for methods
  - offsets for attributes
  - method inlining
- Example:

Common operations like String.length have their own special instruction execute-inline

- VM has special code just for those common operations
- Things like calling the Object's constructor - optimized to nothing because the method is empty



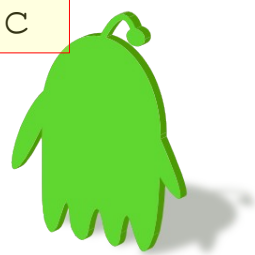
ODEX Example

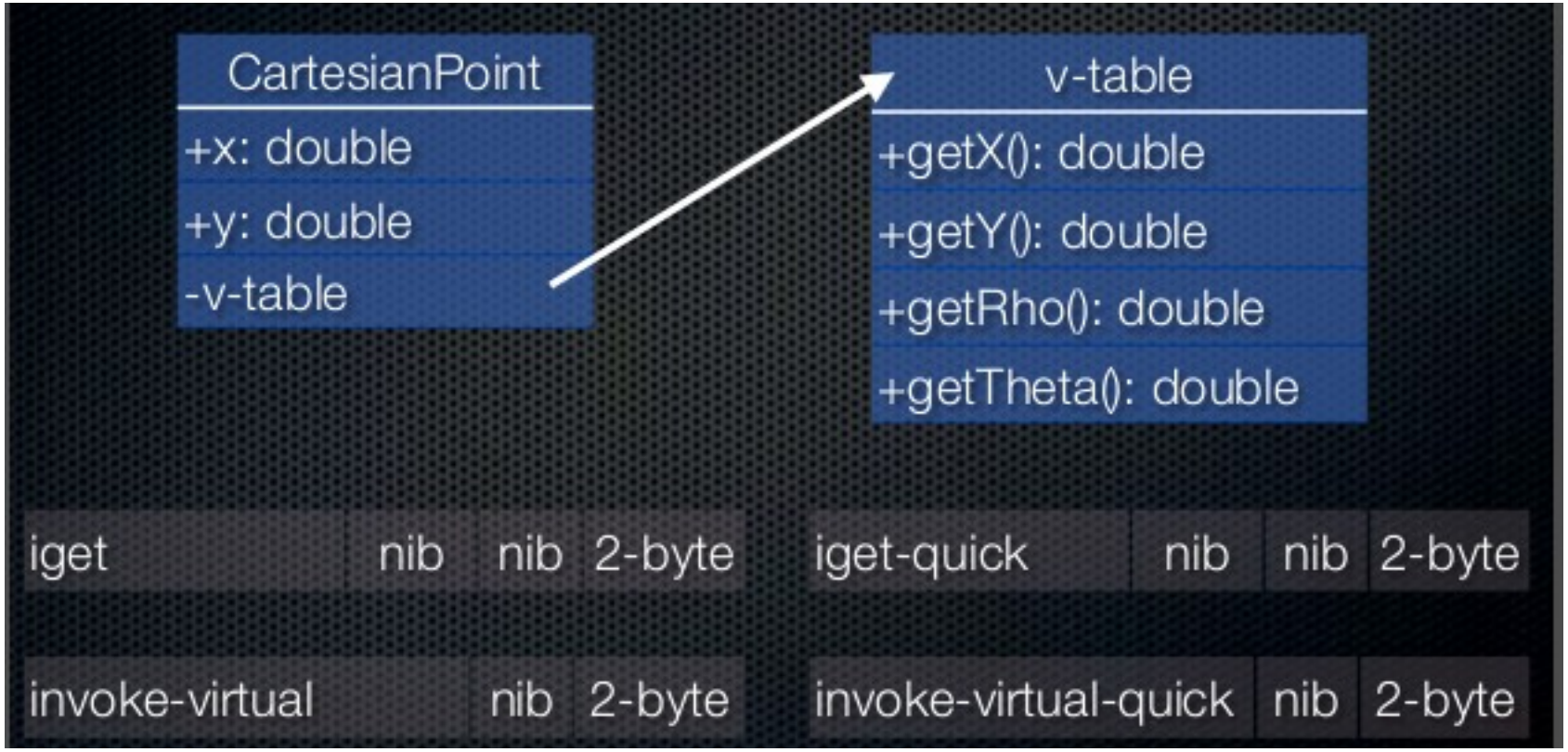
```
$ dexdump -d Foo.dex
...
|[00016c] Foo.main:([Ljava/lang/String;)V
|0000: sget-object v0, Ljava/lang/System;.out:Ljava/io/PrintStream;
|0002: const-string v1, "Hello, world"
|0004: invoke-virtual {v0, v1},
                Ljava/io/PrintStream;.println:(Ljava/lang/String;)V
|0007: return-void
```

Optimized DEX generated by "**dexopt**"

Where is "**println**"?

```
$ dexdump -d \
  /data/dalvik-cache/tmp@cyanogen-ics@tests@Foo.dex
...
|[00016c] Foo.main:([Ljava/lang/String;)V
|0000: sget-object v0, Ljava/lang/System;.out:Ljava/io/PrintStream;
|0002: const-string v1, "Hello, world"
|0004: +invoke-virtual-quick {v0, v1}, [002c] // vtable #002c
|0007: return-void
```





- Virtual (non-private, non-constructor, non-static methods)

  invoke-virtual <symbolic method name> → **invoke-virtual-quick** <vtable index>

  Before:

```
invoke-virtual {v0, v1},
        Ljava/io/PrintStream;.println:(Ljava/lang/String;)V
```

  After:

```
+invoke-virtual-quick {v0, v1}, [002c] // vtable #002c
```

- Can change invoke-virtual to invoke-virtual-quick
  - because we know the layout of the v-table

DEX Optimizations

- Before being executed by Dalvik, DEX files are optimized.
  - Normally it happens before the first execution of code from the DEX file
  - Combined with the bytecode verification
  - In case of DEX files from APKs, when the application is launched for the first time.
- Process
  - The dexopt process (which is actually a backdoor of Dalvik) loads the DEX, replaces certain instructions with their optimized counterparts
  - Then writes the resulting optimized DEX (ODEX) file into the /data/dalvik-cache directory
  - It is assumed that the optimized DEX file will be executed on the same VM that optimized it. ODEX files are NOT portable across VMs.

dexopt: Instruction Rewritten

- Virtual (non-private, non-constructor, non-static methods)

  invoke-virtual <symbolic method name> → **invoke-virtual-quick** <vtable index>

  Before:

  invoke-virtual {v1,v2},java/lang/StringBuilder/append;append(Ljava/lang/String;)Ljava/lang/StringBuilder;

  After:

  invoke-virtual-quick {v1,v2},vtable #0x3b

- Frequently used methods

  invoke-virtual/direct/static <symbolic method name> → execute-inline <method index>

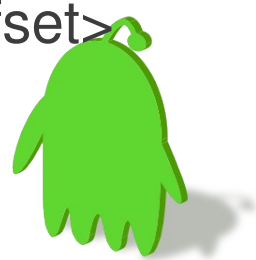
  – Before:

  invoke-virtual {v2},java/lang/String/length

  – After:

  execute-inline {v2},inline #0x4

- instance fields: iget/iput <field name> → iget/iput-quick <memory offset>
  – Before: iget-object v3,v5,android/app/Activity.mComponent
  – After: iget-object-quick v3,v5,[obj+0x28]



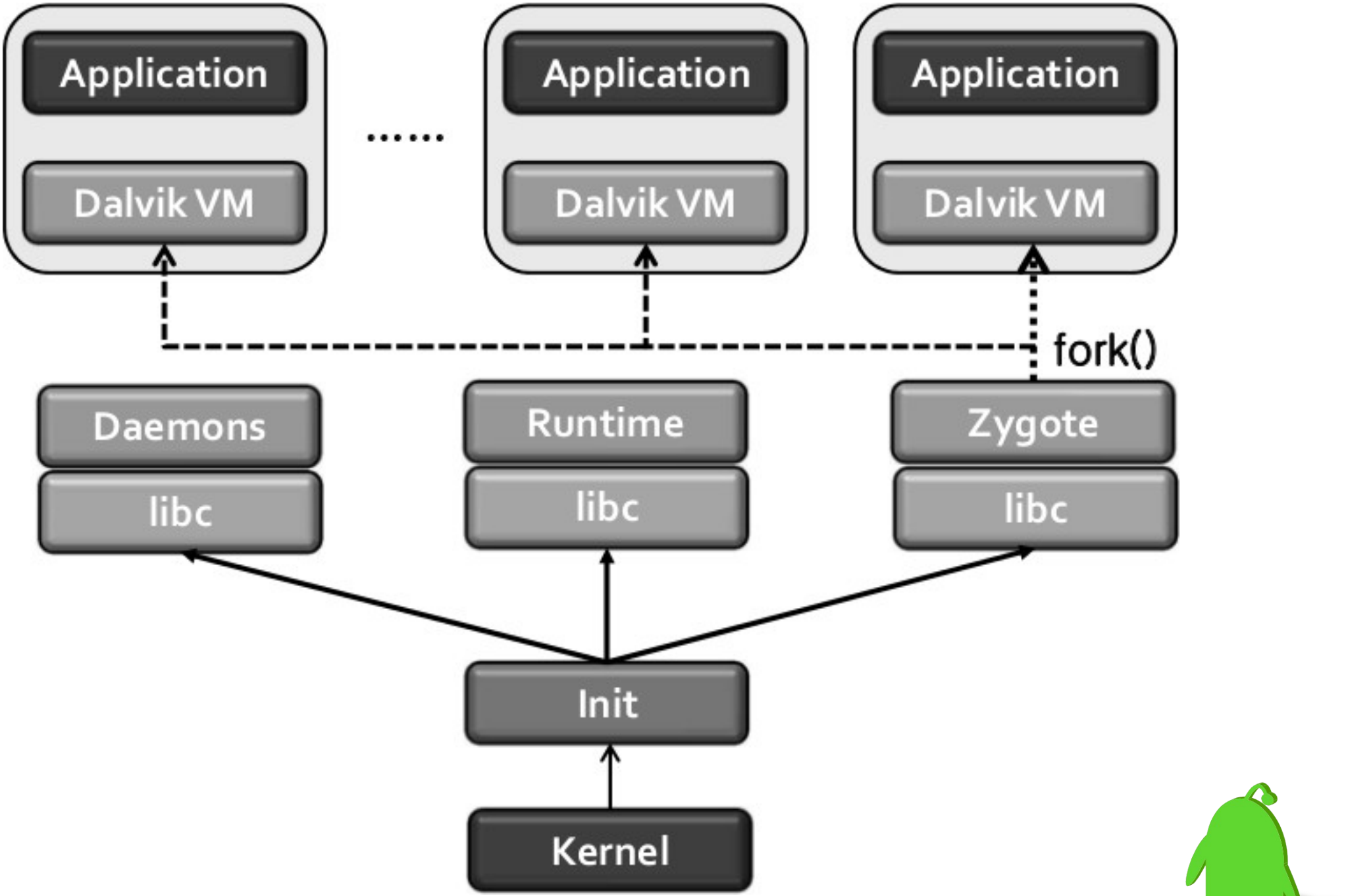
Meaning of DEX Optimizations

- Sets byte ordering and structure alignment
- Aligns the member variables to 32-bits / 64-bits
- boundary (the structures in the DEX/ODEX file itself are 32-bit aligned)
- Significant optimizations because of the elimination of symbolic field/method lookup at runtime.
- Aid of Just-In-Time compiler

Efficient Interpreter: Zygote

is a VM process that starts at system boot time.

- Boot-loader load kernel and start init process.
- Starts Zygote process
- Initializes a Dalvik VM which preloads and pre-initializes core library classes.
- Keep in an idle state by system and wait for socket requests.
- Once an application execution request occur, Zygote forks itself and create new process with pre-loaded Dalvik VM.

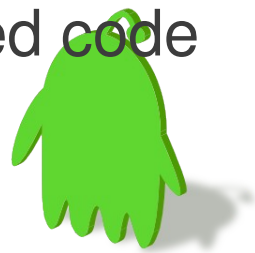

Application
Application
Application
.....
Dalvik VM
Dalvik VM
Dalvik VM
fork()
Daemons
Runtime
Zygote
libc
libc
libc
Init
Kernel

Efficient Interpreter: Just-In-Time Compilation

- **Just-in-time compilation** (**JIT**), also known as **dynamic translation**, is a technique for improving the runtime performance of a computer program.
- A hybrid approach, with translation occurring continuously, as with interpreters, but with caching of translated code to minimize performance degradation

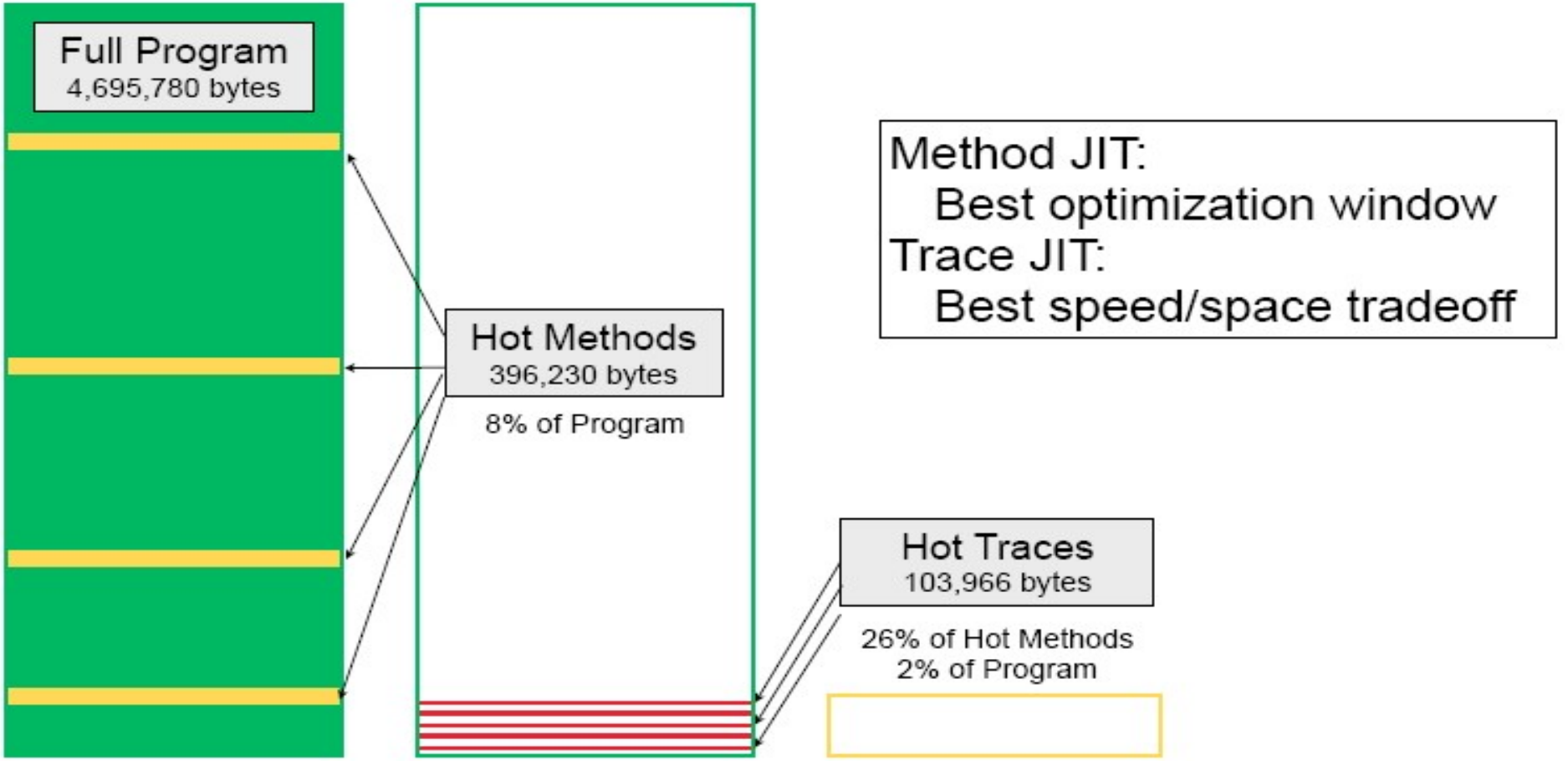
JIT Types

- When to compile
  - install time, launch time, method invoke time, instruction fetch time
- What to compile
  - whole program, shared library, page, method, trace, single instruction
- Android needs a combination that meet the needs of a mobile
  - Minimal additional memory usage
  - Coexist with Dalvik's container-based security model
  - Quick delivery of performance boost
  - Smooth transition between interpretation & compiled code



Android system_server example

Source: Google I/O 2010 - A JIT Compiler for Android's Dalvik VM

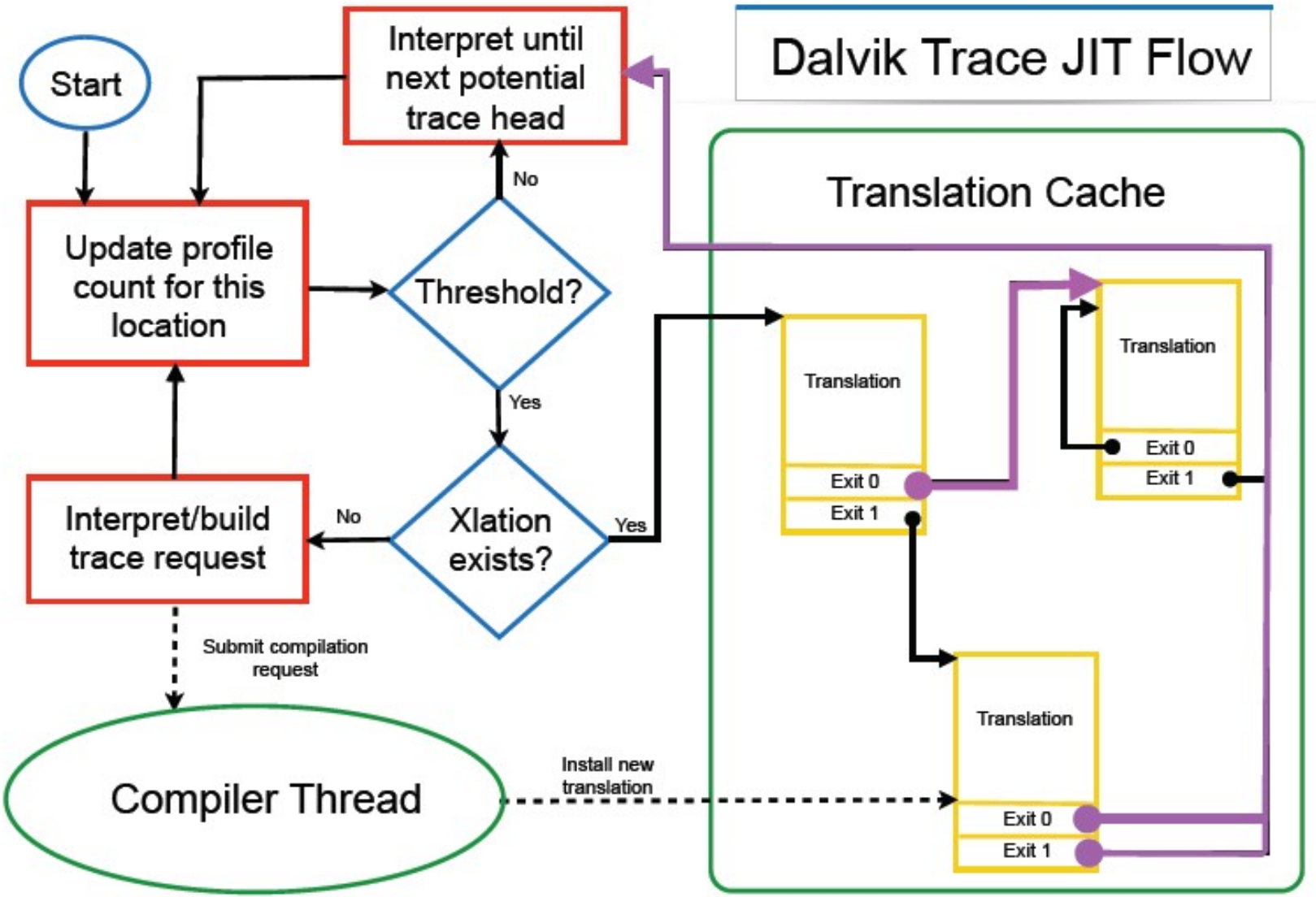


- Compiled Code takes up memory - want the benefits of JIT with small memory footprint
- Small amount compilation provides a big benefit
- In test program, 4.5MB of byte code - 8% of methods: 390K was hot; 25% of code in methods was hot - so 2% in the end
- 90% of time in 10% of the code may be generous

Trace JIT

- Trace : String of Instructions
- Minimizing memory usage critical for mobile devices
- Important to deliver performance boost quickly
  - User might give up on new app if we wait too long to JIT
- Leave open the possibility of supplementing with method based JIT
  - The two styles can co-exist
  - A mobile device looks more like a server when it's plugged in
  - Best of both worlds
    - Trace JIT when running on battery
    - Method JIT in background while charging

Dalvik Trace JIT Flow



Dalvik JIT Overview

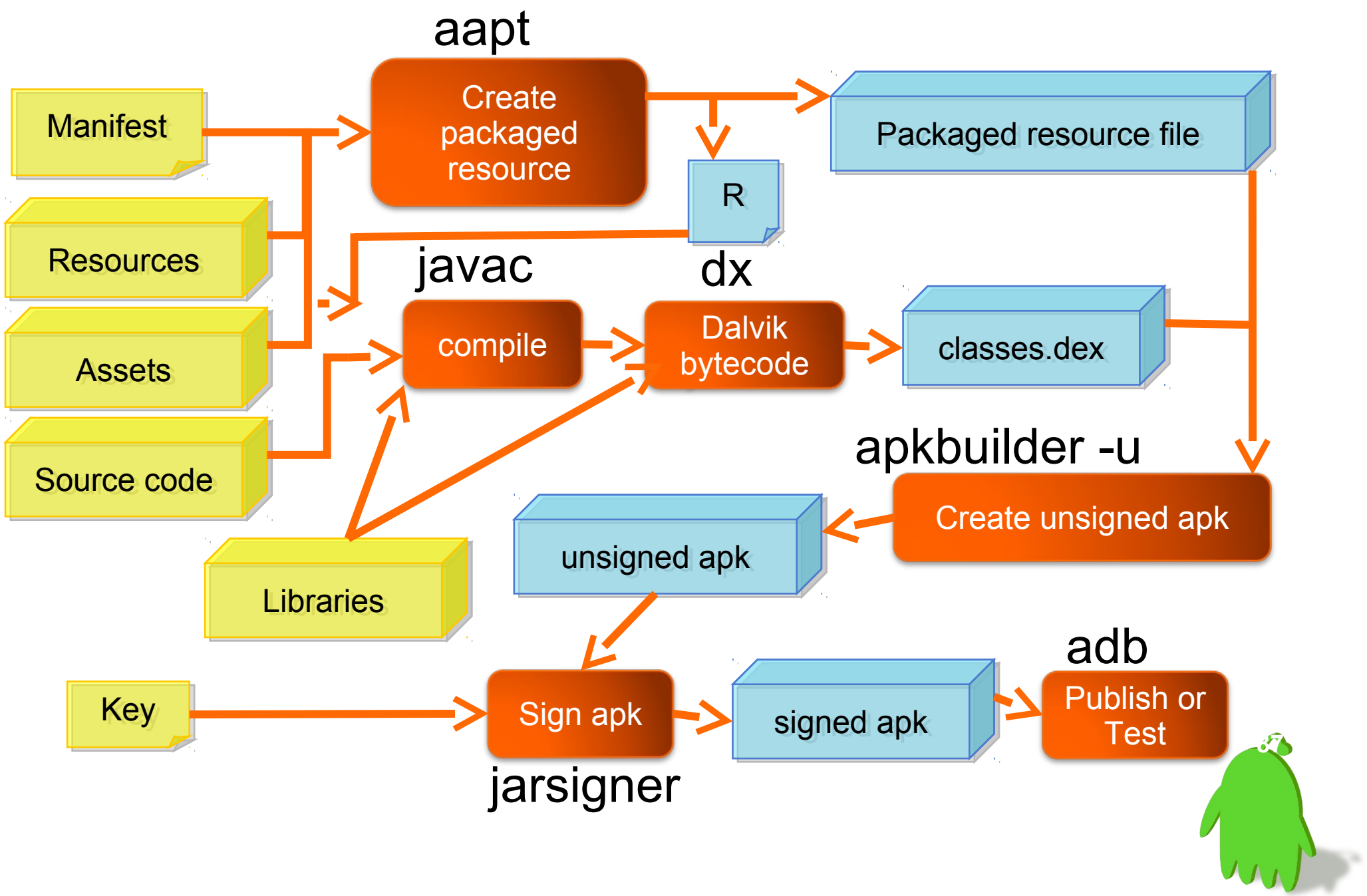
- Tight integration with interpreter
  - Useful to think of the JIT as an extension of the interpreter
- Interpreter profiles and triggers trace selection mode when a potential trace head goes hot
- Trace request is built during interpretation
- Trace requests handed off to compiler thread, which compiles and optimizes into native code
- Compiled traces chained together in translation cache

Dalvik JIT Features

- Per-process translation caches (sharing only within security sandboxes)
- Simple traces - generally 1 to 2 basic blocks long
- Local optimizations
  - Register promotion
  - Load/store elimination
  - Redundant null-check elimination
- Loop optimizations
  - Simple loop detection
  - Invariant code motion
  - Induction variable optimization

Utilities

Android Application Development Flow



APK content

```
$ unzip Angry+Birds.apk
Archive:  Angry+Birds.apk
...
  inflating: AndroidManifest.xml
 extracting: resources.arsc
 extracting: res/drawable-hdpi/icon.png
 extracting: res/drawable-ldpi/icon.png
 extracting: res/drawable-mdpi/icon.png
  inflating: classes.dex
  inflating: lib/armeabi/libangrybirds.so
  inflating: lib/armeabi-v7a/libangrybirds.so
  inflating: META-INF/MANIFEST.MF
  inflating: META-INF/CERT.SF
  inflating: META-INF/CERT.RSA
```

Dalvik DEX

JNI

manifest + signature

APK content

```
$ unzip Angry+Birds.apk
Archive:  Angry+Birds.apk
...
  inflating: AndroidManifest.xml
```

```
Name: classes.dex

SHA1-Digest: I9Vne//i/5Wyzs5HhBVu9dIoHDY=

Name: lib/armeabi/libangrybirds.so
SHA1-Digest: pSdb9FYauyfjDUxM8L6JDmQk4qQ=
```

```
  inflating: classes.dex
  inflating: lib/armeabi/libangrybirds.so
  inflating: lib/armeabi-v7a/libangrybirds.so
  inflating: META-INF/MANIFEST.MF
  inflating: META-INF/CERT.SF
  inflating: META-INF/CERT.RSA
```

AndroidManifest

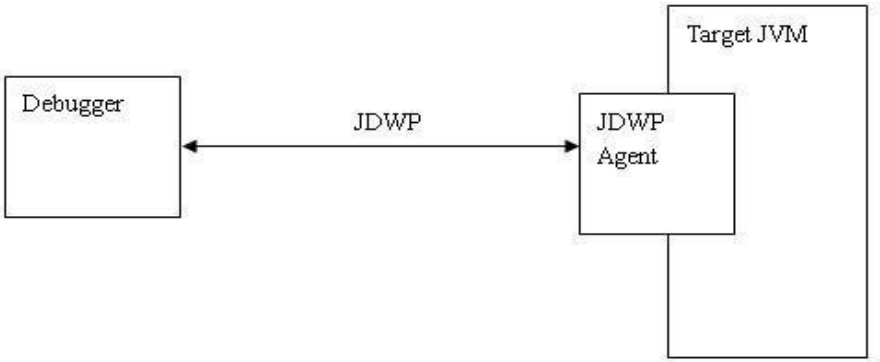

```
$ unzip Angry+Birds.
Archive:  Angry+Bird
...
...
  inflating: AndroidManifest.xml
 extracting: resources.arsc
```

```
$ file AndroidManifest.xml
AndroidManifest.xml: DBase 3 data file (2328 records)

$ apktool d ../AngryBirds/Angry+Birds.apk
I: Baksmaling...
I: Loading resource table...
...
I: Decoding file-resources...
I: Decoding values*/* XMLs...
I: Done.
I: Copying assets and libs...
$ file Angry+Birds/AndroidManifest.xml
Angry+Birds/AndroidManifest.xml: XML  document text
```

so

Use JDB to Trace Android Application

```
#!/bin/bash
adb wait-for-device
adb shell am start \
    -e debug true \
    -a android.intent.action.MAIN \
    -c android.intent.category.LAUNCHER \
    -n org.jfedor.frozenbubble/.FrozenBubble &
debug_port=$(adb jdwp | tail -1);
adb forward tcp:29882 jdwp:$debug_port &
jdb -J-Duser.home=. -connect \
com.sun.jdi.SocketAttach:hostname=localhost,port=29882 &
```



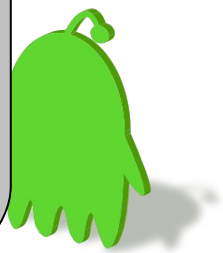
In APK manifest, `debuggable="true"`

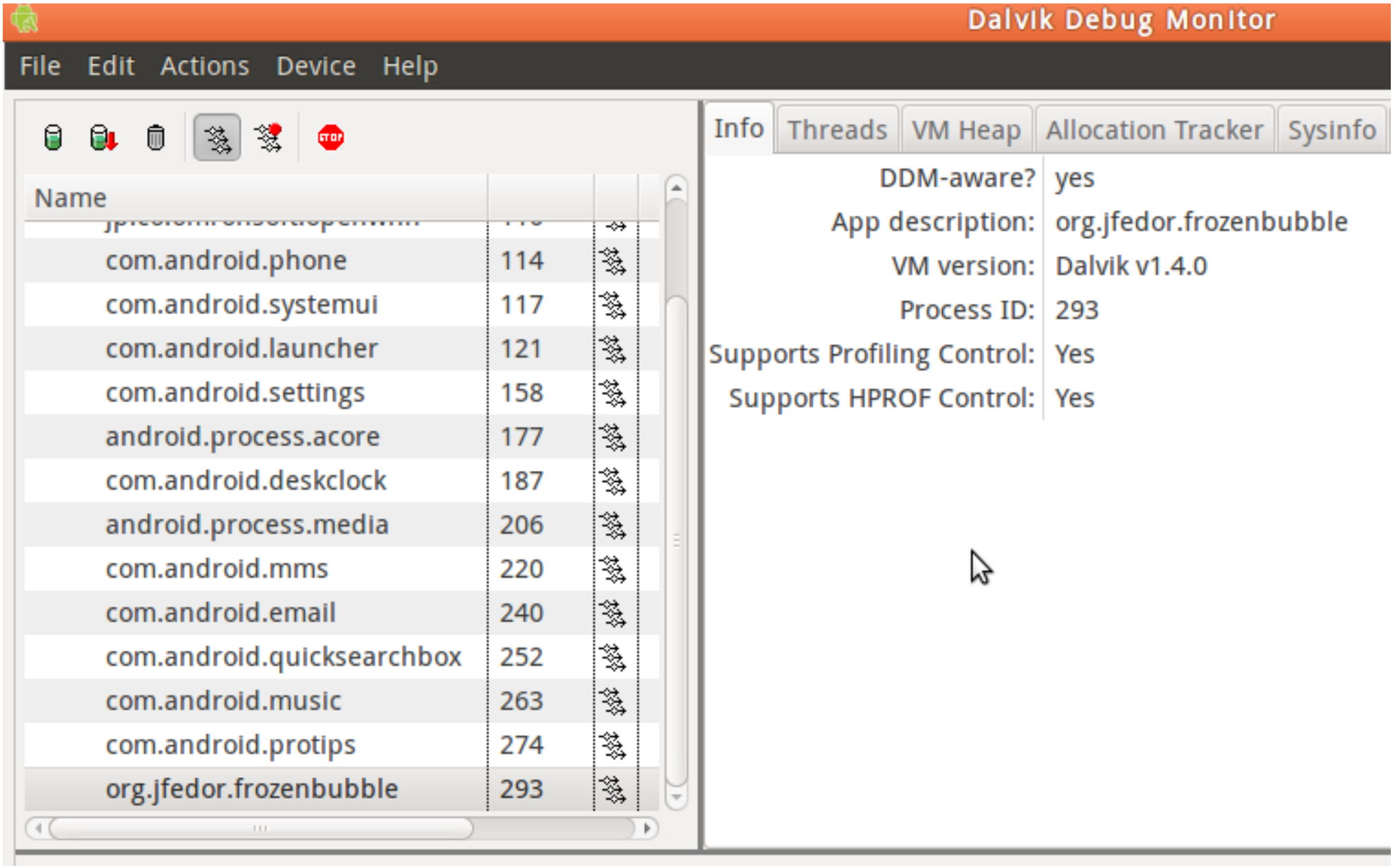
JDWP: Java Debug Wire Protocol

JDB usage

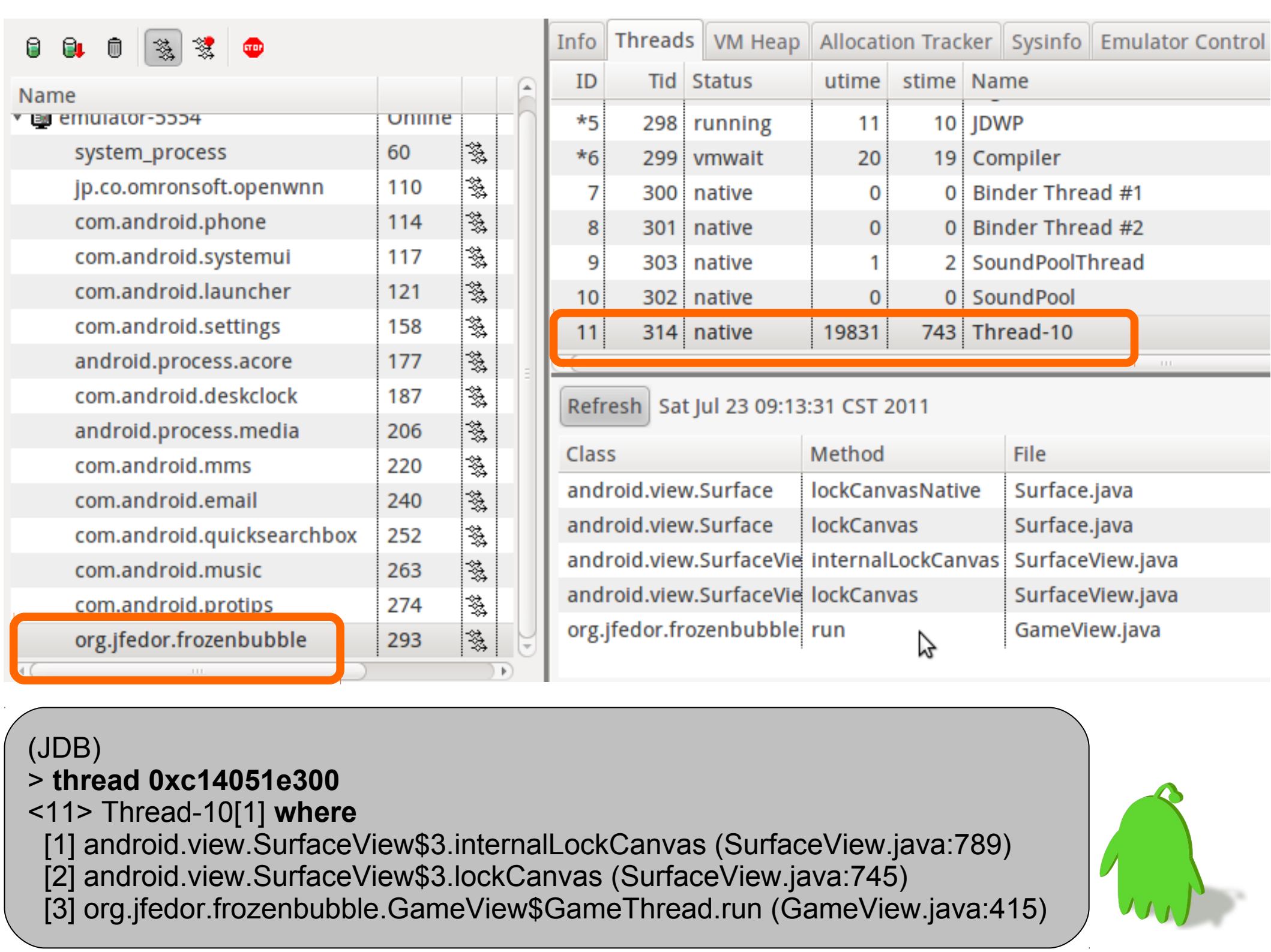
```
> threads
Group system:
  (java.lang.Thread)0xc14050e388  <6> Compiler          cond. Waiting
  (java.lang.Thread)0xc14050e218  <4> Signal Catcher    cond. waiting
  (java.lang.Thread)0xc14050e170  <3> GC                cond. waiting
  (java.lang.Thread)0xc14050e0b8  <2> HeapWorker        cond. waiting
Group main:
  (java.lang.Thread)0xc14001f1a8  <1> main              running
  (org.jfedor.frozenbubble.GameView$GameThread)0xc14051e300
                                  <11> Thread-10        running
  (java.lang.Thread)0xc14050f670  <10> SoundPool        running
  (java.lang.Thread)0xc14050f568  <9> SoundPoolThread   running
  (java.lang.Thread)0xc140511db8  <8> Binder Thread #2  running
  (java.lang.Thread)0xc140510118  <7> Binder Thread #1  running
```

```
> suspend 0xc14051e300
> thread 0xc14051e300
<11> Thread-10[1] where
  [1] android.view.SurfaceView$3.internalLockCanvas (SurfaceView.java:789)
  [2] android.view.SurfaceView$3.lockCanvas (SurfaceView.java:745)
  [3] org.jfedor.frozenbubble.GameView$GameThread.run (GameView.java:415)
```



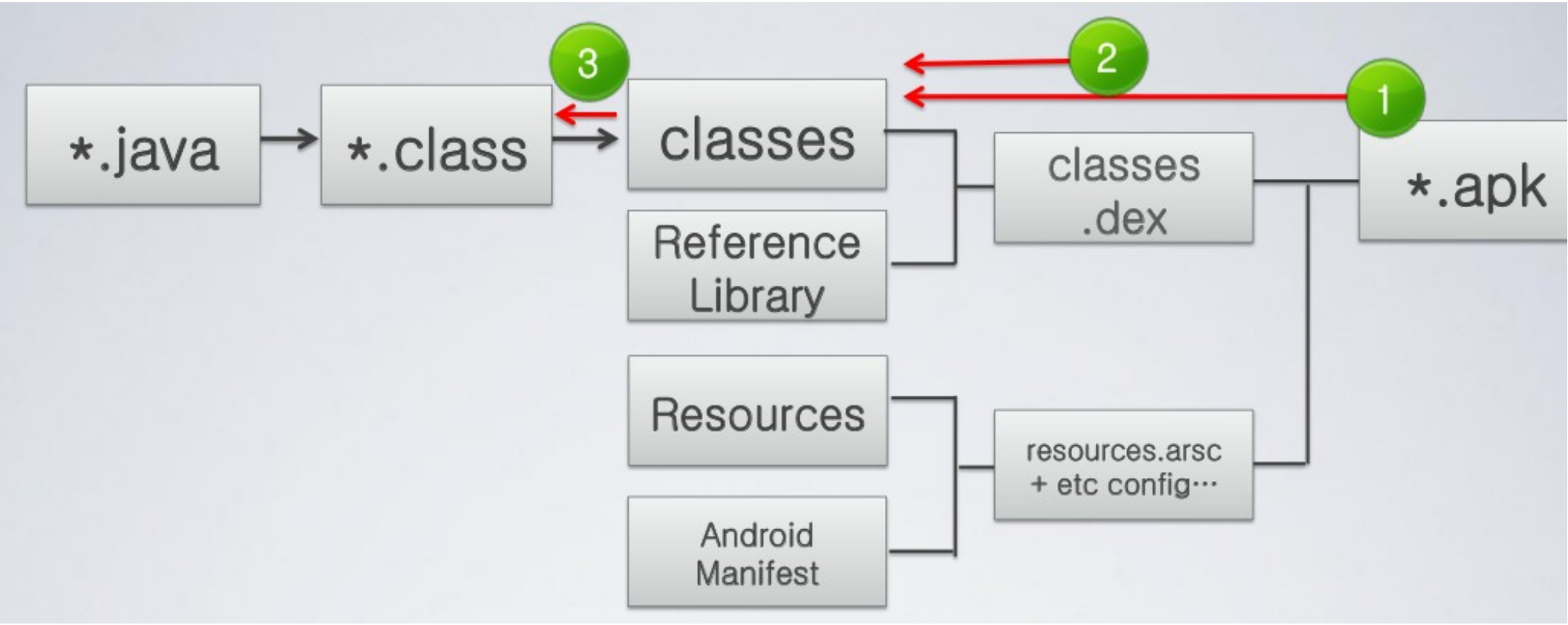


DDMS = Dalvik Debug Monitor Server




```
(JDB)
> thread 0xc14051e300
<11> Thread-10[1] where
  [1] android.view.SurfaceView$3.internalLockCanvas (SurfaceView.java:789)
  [2] android.view.SurfaceView$3.lockCanvas (SurfaceView.java:745)
  [3] org.jfedor.frozenbubble.GameView$GameThread.run (GameView.java:415)
```

- apktool: http://code.google.com/p/android-apktool/
- dex2jar: http://code.google.com/p/dex2jar/
- Jad / jd-gui: http://java.decompiler.free.fr/



smali : assembler/disassembler for Android's dex format

- http://code.google.com/p/smali/
- smali: The assembler
- baksmali: The disassembler
- Fully integrated in apktool

```
$ apktool d ../AngryBirds/Angry+Birds.apk
I: Baksmaling...
I: Loading resource table...
...
I: Decoding file-resources...
I: Decoding values*/* XMLs...
I: Done.
I: Copying assets and libs...
```

Disassembly

```
$ mkdir workspace smali-src
$ cd workspace
$ unzip ../FrozenBubble-orig.apk
Archive:  ../FrozenBubble-orig.apk
  inflating: META-INF/MANIFEST.MF
  inflating: META-INF/CERT.SF
  inflating: META-INF/CERT.RSA
  inflating: AndroidManifest.xml
...
extracting: resources.arsc
$ bin/baksmali -o smali-src workspace/classes.dex
```

Output

org.jfedor.frozenbubble/.FrozenBubble

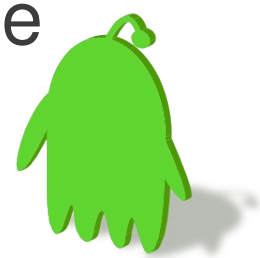
```
smali-src$ find
./org/jfedor/frozenbubble/FrozenBubble.smali
./org/jfedor/frozenbubble/R$id.smali
./org/jfedor/frozenbubble/GameView.smali
./org/jfedor/frozenbubble/SoundManager.smali
./org/jfedor/frozenbubble/LaunchBubbleSprite.smali
./org/jfedor/frozenbubble/Compressor.smali
./org/jfedor/frozenbubble/R$attr.smali
./org/jfedor/frozenbubble/BubbleFont.smali
./org/jfedor/frozenbubble/PenguinSprite.smali
./org/jfedor/frozenbubble/GameView$GameThread.smali
./org/jfedor/frozenbubble/LevelManager.smali
./org/jfedor/frozenbubble/BubbleSprite.smali
./org/jfedor/frozenbubble/R$string.smali
...
```

Generated from resources

Dexmaker: bytecode generator

http://code.google.com/p/dexmaker/

- A Java-language API for doing compile time or runtime code generation targeting the Dalvik VM. Unlike cglib or ASM, this library creates Dalvik .dex files instead of Java .class files.
- It has a small, close-to-the-metal API. This API mirrors the Dalvik bytecode specification giving you tight control over the bytecode emitted.
- Code is generated instruction-by-instruction; you bring your own abstract syntax tree if you need one. And since it uses Dalvik's dx tool as a backend, you get efficient register allocation and regular/wide instruction selection for free.



Reference

- Dalvik VM Internals, Dan Bornstein (2008)
  http://sites.google.com/site/io/dalvik-vm-internals
- Analysis of Dalvik Virtual Machine and Class Path Library, Institute of Management SciencesPeshawar, Pakistan (2009)
  http://serg.imsciences.edu.pk
- Reconstructing Dalvik applications, Marc Schonefeld (2009)
- A Study of Android Application Security, William Enck, Damien Octeau, Patrick McDaniel, and Swarat Chaudhuri (2011)
- dalvik の GC をのぞいてみた , @akachochin (2011)
- 《 Android 惡意代碼分析教程》 , Claud Xiao (2012)
  http://code.google.com/p/amatutor/
- XXX

Reference

- ded: Decompiling Android Applications
  http://siis.cse.psu.edu/ded/
- TBD

http://0xlab.org