Dalvik and ART

Jonathan Levin

http://NewAndroidBook.com/

http://www.technologeeks.com

Wait.. Isn’t Android all ART now?

- Well.. Yes, and no.. The actual runtime **is** ART, but..
  - Your applications still compile into Dalvik (DEX) code
  - Final compilation to ART occurs on the device, during install
  - Even ART binaries have Dalvik embedded in them
  - Some methods may be left as DEX, to be interpreted
  - Dalvik is *much* easier to debug than ART.

What we **won't** be discussing

- Dalvik VM runtime architecture*
  - Mostly replaced by ART, prominent features removed
  - No talk about JIT (ART does AOT)
  - No JNI
- Dalvik specific debug settings
  - Not really relevant anymore, either

*** - We discuss these aspects later on, in the context of ART – but that's part II**

What we **will** be discussing

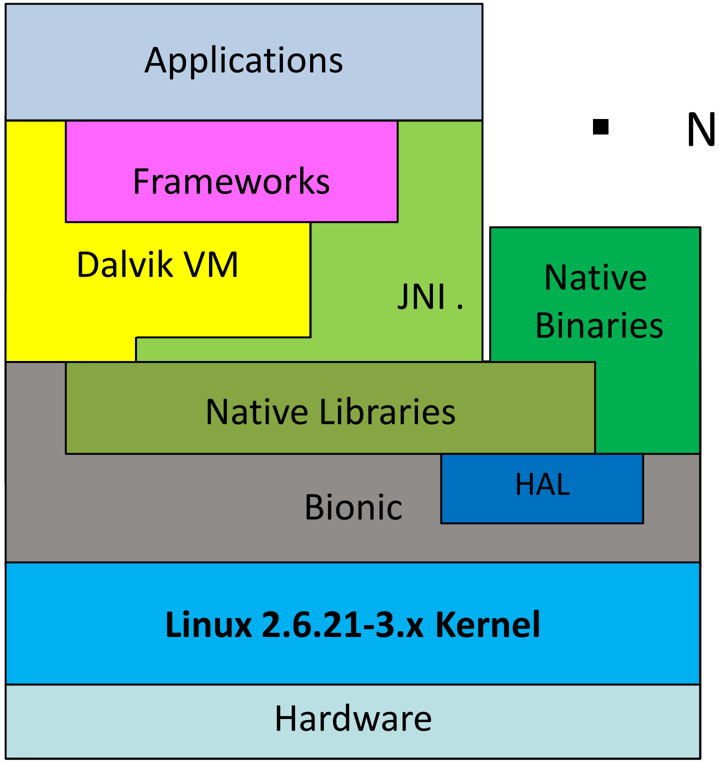
- DEX file structure
- DEX code generation
- DEX verification and optimization
- DEX decompilation and reverse engineering

Interlude (Necessary Plug)

- Me: Jonathan Levin, CTO of http://Technologeeks.com
  - Training and consulting on internals/debugging, networking
  - Follow us on Twitter (@Technologeeks), Etc. Etc. Etc
- My Book: "Android Internals: A Confectioner's Cookbook"
  - http://www.NewAndroidBook.com/ for tools, articles, and more
  - Unofficial sequel to Karim Yaghmour's "Embedded Android"
    - More on the **how** and **why** Android frameworks and services work
  - (presently) only in-depth book on the subject
- Just in case anyone's into iOS (w/41% probability?)
  - http://www.newosxbook.com/
  - 2nd Edition (covers iOS 8, OS X 10.10) due March '15

Part I - Dalvík

Dalvík and the Android Architecture



The Dalvík Virtual Machine* is:

- Customized, optimized JVM
  - Based on Apache “Harmony” JVM
- Not fully J2SE or J2ME compatible
  - Java compiles into DEX code
    - 16-bit opcodes
    - Register, rather than stack-based

*** - Android L replaces Dalvik by the Android RunTime – but does not get rid of it fully (more later)**

A Brief History of Dalvík

- Dalvík was introduced along with Android
  - Created by Dan Bornstein
  - Named after an Icelandic town
- 2.2 (Froyo) brought Just-in-Time compilation
- 4.4 (KitKat) previews ART
- 5.0 (Lollipop) ART supersedes.

Dalvik, Iceland (photo by the author)

Dalvik VM vs. Java

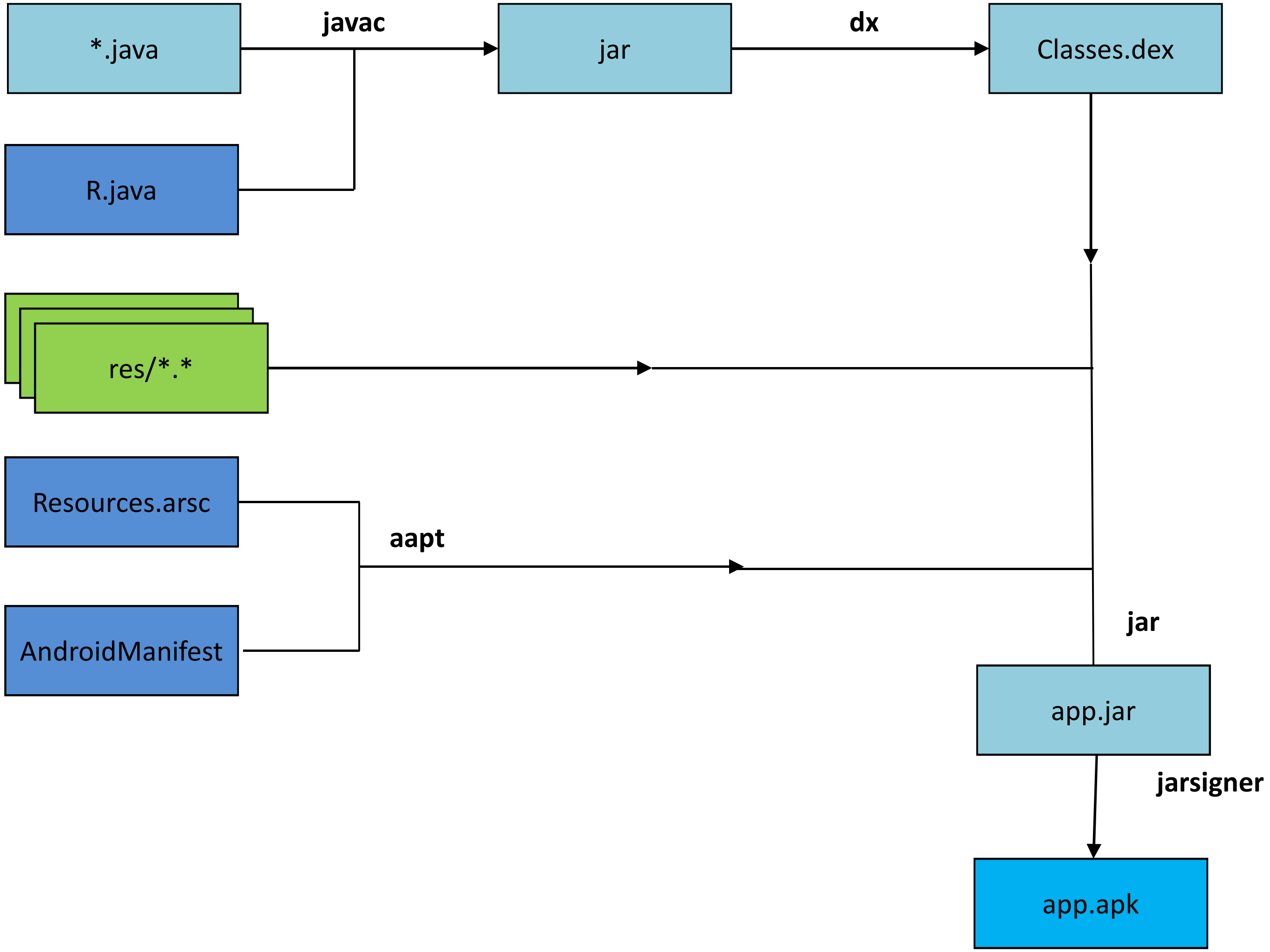
- Dalvík is a virtual machine implementation
  - Based on Apache Harmony
  - Borrows heavily from Java*

- Brings significant improvements over Java, in particular J2ME:
  - Virtual Machine architecture is optimized for memory sharing
    - Reference counts/bitmaps stored separately from objects
    - Dalvik VM startup is optimized through Zygote

- Java .class files are further compiled into DEX.

*** - So heavily, in fact, that Oracle still carries Sun's grudge against Google**

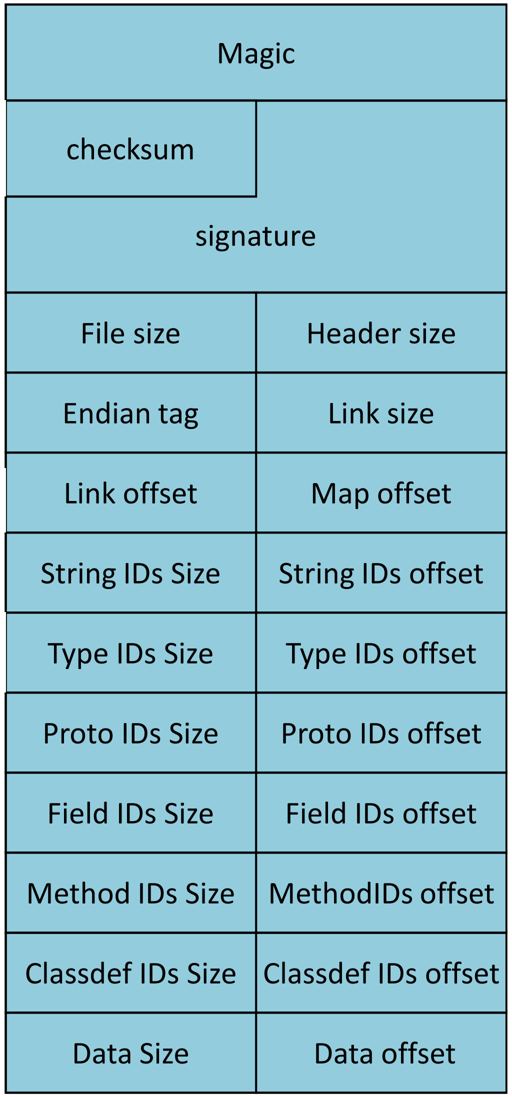
Reminder: Creating an APK



The DEX file format

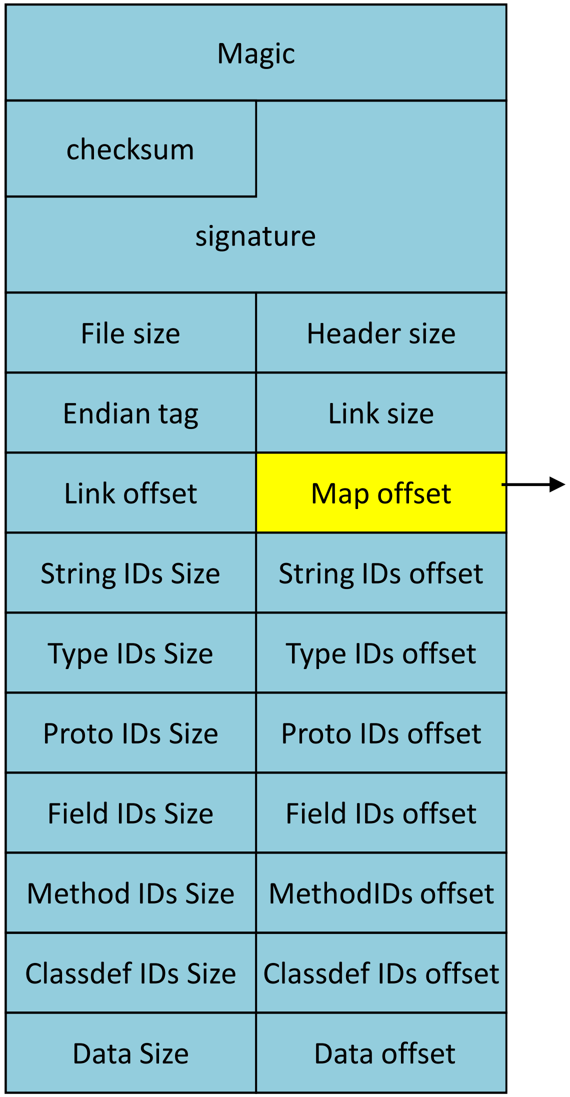
- The “dx” utility converts multiple .class files to classes.dex
  - Script wrapper over `java -Xmx1024M -jar ${SDK_ROOT}.../lib/dx.jar`
  - Java byte code is converted to DEX bytecode
    - DEX instructions are 16-bit multiples, as opposed to Java’s 8-bit
  - Constant, String, Type and Method pools can be merged
    - Significant savings for strings, types, and methods in multiple classes
- Overall memory footprint diminished by about 50%
- DEX file format fully specified in [Android Documentation]()

The DEX file format



	Magic		DEX Magic header ("dex\n“ and version (“035 “)
Adler32 of header (from offset +12)	checksum		
	signature		SHA-1 hash of file (20 bytes)
Total file size	File size	Header size	Header size (0x70)
0x12345678, in little or big endian form	Endian tag	Link size	Unused (0x0)
Unused (0x0)	Link offset	Map offset	Location of file map
Number of String entries	String IDs Size	String IDs offset	
Number of Type definition entries	Type IDs Size	Type IDs offset	
Number of prototype (signature) entries)	Proto IDs Size	Proto IDs offset	
Number of field ID entries	Field IDs Size	Field IDs offset	
Number of method ID entries	Method IDs Size	MethodIDs offset	
Number of Class Definition entries	Classdef IDs Size	Classdef IDs offset	
Data (map + rest of file)	Data Size	Data offset	

The DEX file format



Magic	
checksum	
signature	
File size	Header size
Endian tag	Link size
Link offset	Map offset
String IDs Size	String IDs offset
Type IDs Size	Type IDs offset
Proto IDs Size	Proto IDs offset
Field IDs Size	Field IDs offset
Method IDs Size	MethodIDs offset
Classdef IDs Size	Classdef IDs offset
Data Size	Data offset

Type	Implies	Size	Offset
0x0	DEX Header	1 (implies Header Size)	0x0
0x1	String ID Pool	Same as String IDs size	Same as String IDs offset
0x2	Type ID Pool	Same as Type IDs size	Same as String IDs offset
0x3	Prototype ID Pool	Same as Proto IDs size	Same as ProtoIDs offset
0x4	Field ID Pool	Same as Field IDs size	Same as Field IDs offset
0x5	Method ID Pool	Same as Method IDs size	Same as Method IDs offset
0x6	Class Defs	Same as ClassDef IDs size	Same as ClassDef IDs offset
0x1000	Map List	1	Same as Map offset
0x1001	Type List	List of type indexes (from Type ID Pool)	
0x1002 0x1003	Annotation set Annotation Ref	Used by Class, method and field annotations	
0x2000	Class Data Item	For each class def, class/instance methods and fields	
0x2001	Code	DexCodeItems – contains the actual byte code	
0x2002	String Data	Pointers to actual string data	
0x2003	Debug Information	Debug_info_items containing line no and variable data)	
0x2004	Annotation	Field and Method annotations	
0x2005	Encoded Array	Used by static values	
0x2006	Annotations Directory	Annotations referenced from individual classdefs	

Looking up classes, methods, etc.

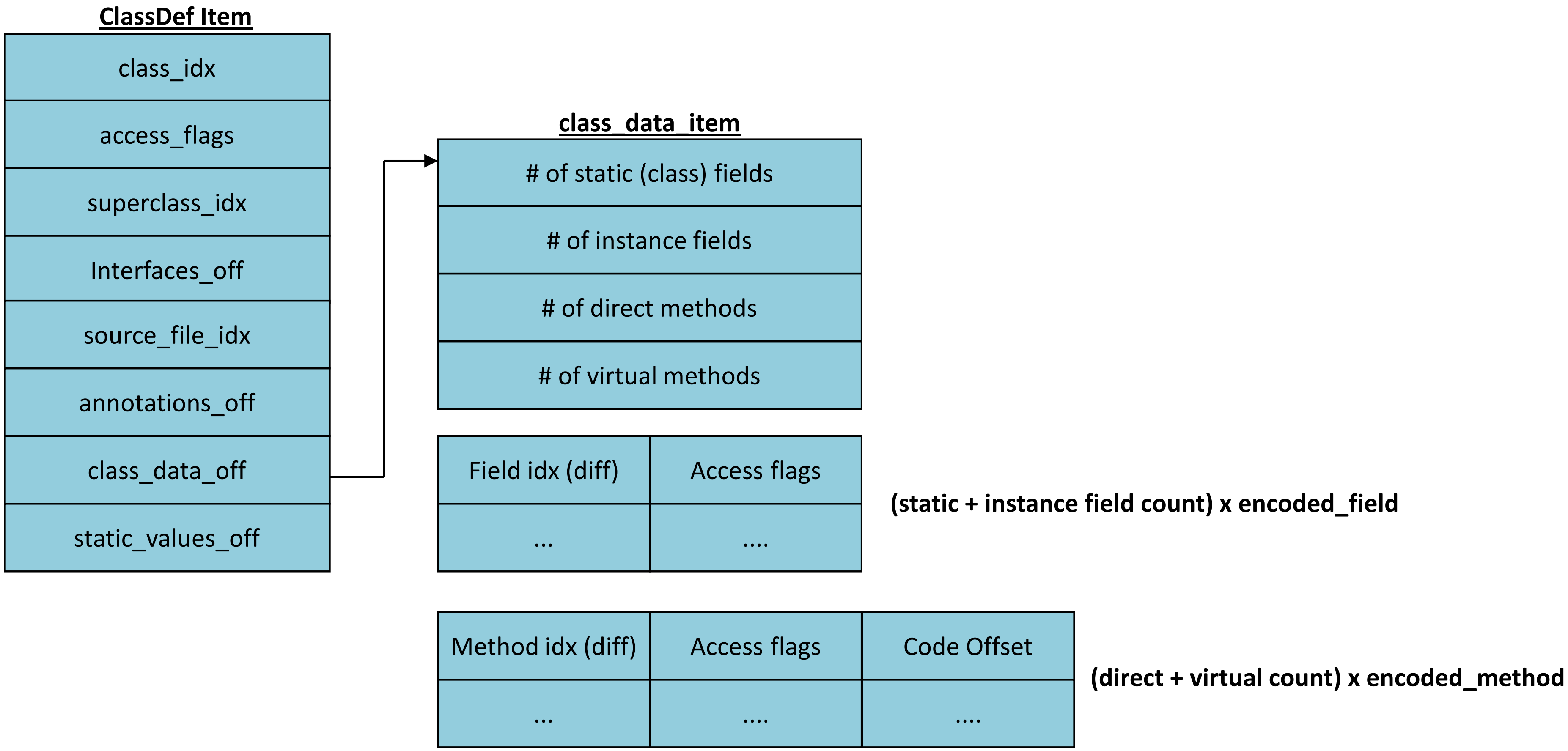
- Internally, DEX instructions refer to Indexes (in pools)
- To find a method:
  - DexHeader’s Method IDs offset points to an array of MethodIDs
  - Each method ID points to a class index, prototype index and method name
- To find a field:
  - DexHeader’s Field Ids offset points to an array of FieldIDs
  - Each Field ID points to a class index, type index, and the field name
- To get a class:
  - DexHeader’s Class Defs Ids offset points to an array of ClassDefs
  - Each ClassDef points to superclass, interface, and class_data_item
  - Class_data_item shows # of static/instance fields, direct/virtual methods
  - Class_data_item is followed by DexField[], DexMethod[] arrays
    - DexField, DexMethod point to respective indexes, as well as class specific access flags

Finding a class’s method code

class_idx	**Index of the class’ type id, from Type ID pool**
access_flags	**ACC_PUBLIC, _PRIVATE, _PROTECTED, _STATIC, _FINAL, etc. Etc..**
superclass_idx	**Index of the superclass’ type id, from Type ID pool**
Interfaces_off	**Offset of type_list containing this class’ implemented interface, if any**
source_file_idx	**Index of the source file name, in String pool**
annotations_off	**Offset of an annotations_directory_item for this class**
class_data_off	**Offset of this class’s class_data_item**
static_values_off	**Offset to initial values of any fields defined as static (i.e. Class)**

access_flags and **static_values_off** particulary useful for fuzzing/patching classes

Finding a class’s method code (II)



ClassDef Item

class_idx
access_flags
superclass_idx
Interfaces_off
source_file_idx
annotations_off
class_data_off
static_values_off

class_data_item

of static (class) fields
of instance fields
of direct methods
of virtual methods

Field idx (diff)	Access flags
…	….

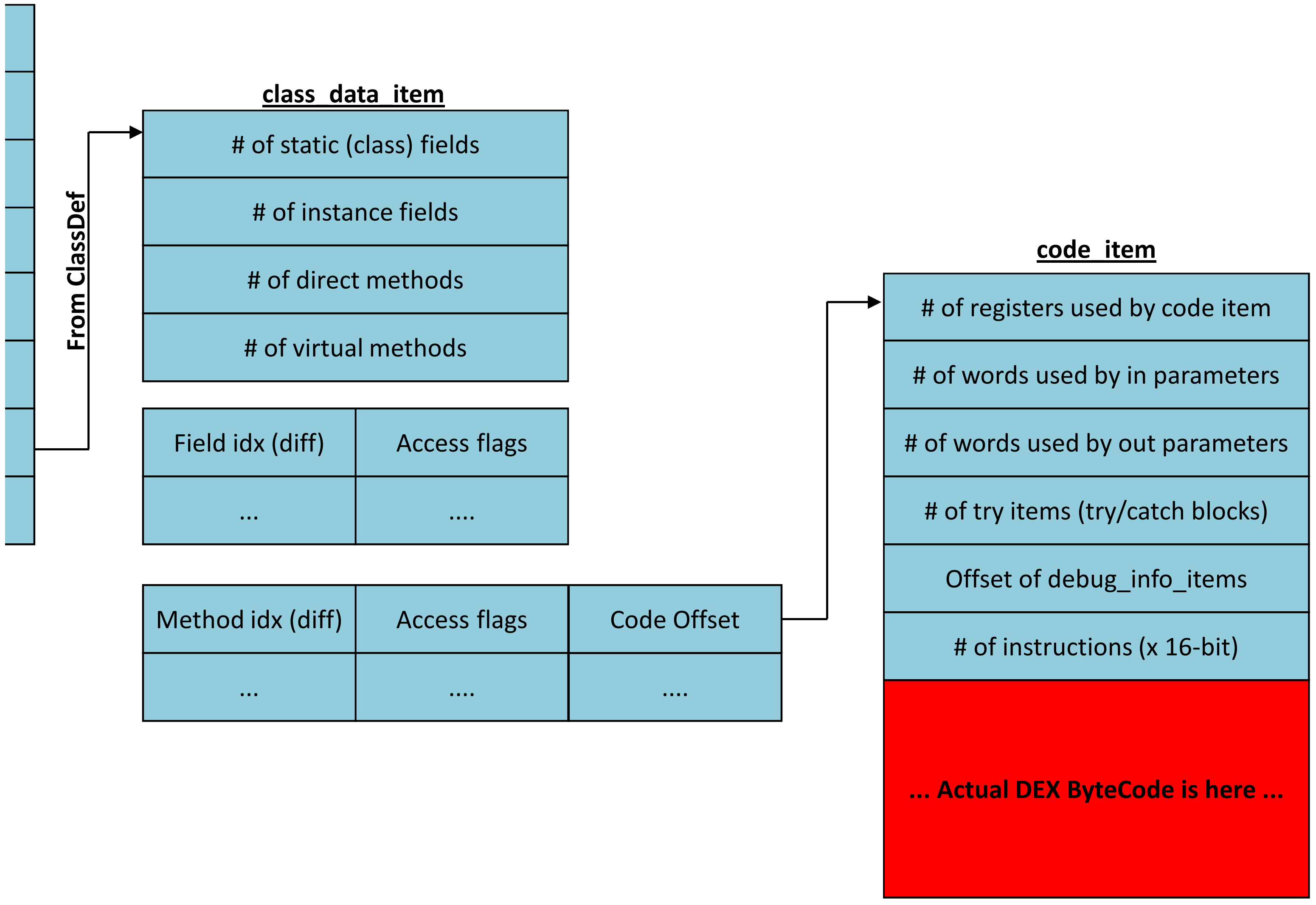
(static + instance field count) x encoded_field

Method idx (diff)	Access flags	Code Offset
…	….	….

(direct + virtual count) x encoded_method

Class_data_item fields are all ULEB128 encoded (*sigh*)

Finding a class's method code (III)



From ClassDef

class_data_item

- # of static (class) fields
- # of instance fields
- # of direct methods
- # of virtual methods

Field idx (diff)	Access flags
...	....

Method idx (diff)	Access flags	Code Offset
...	....	....

code_item

- # of registers used by code item
- # of words used by in parameters
- # of words used by out parameters
- # of try items (try/catch blocks)
- Offset of debug_info_items
- # of instructions (x 16-bit)

... Actual DEX ByteCode is here ...

The DEX Bytecode

- The Android Documentation is good, but lacking
  - Bytecode instruction set
  - Instruction formats
- No documentation on optimized code
  - ODEX codes (used in 0xE3-0xFF) are simply marked as “unused”
- Not yet updated to reflect ART DEX changes (still undocumented)
  - DEX opcode 0x73 claimed by return-void-barrier
  - ODEX codes 0xF2-0xFA are moved to 0xE3-0xEB. 0xEC-0xFF now unused

The DEX Bytecode

- VM Architecture allows for up to 64k registers
  - In practice, less than 16 are actively used
- Bytecode is method, field, type and string aware
  - Operands in specific instructions are IDs from corresponding pools
- Bytecode is also primitive type-aware
  - Instructions support casting, as well as specific primitive types
- DEX bytecode is strikingly similar to Java bytecode
  - Allows for easy de/re-compilation back and forth to/from java

DEX vs. Java

- Java VM is stack based, DEX is register based
  - Operations in JVM use stack and r0-r3; Dalvik uses v0-v65535
  - Stack based operations have direct register-base parallels
  - Not using the stack (= RAM, via L1/L2 caches) makes DEX somewhat faster.
- Java Bytecode is actually more compact than DEX
  - Java instructions take 1-5 bytes, DEX take 2-10 bytes (in 2-byte multiples)
- DEX bytecode is more suited to ARM architectures
  - Straightforward mapping from DEX registers to ARM registers
- DEX supports bytecode optimizations, whereas Java doesn’t
  - APK’s classes.dex are optimized before install, on device (more later)

DEX vs. Java Bytecode

Class, Method and Field operators

DEX Opcode	Java Bytecode	Purpose
60-66:sget-* 52-58:iget-*	b2:getstatic b4:getfield	Read a static or instance variable
67-6d:sput 59-5f:iput	b3:putstatic b5:putfield	Write a static or instance variable
6e: invoke-virtual 6f: invoke-super 70: invoke-direct 71: invoke-static 72: invoke-interface	b6: Invokevirtual ba: invokedynamic b7: invokespecial b8: Invokestatic b9: Invokeinterface	Call a method
20: instance-of	c1: instanceof	Return true if obj is of class
1f: check-cast	c0: checkcast	Check if a type cast can be performed
bb:new	22: new-instance	New (unconstructed) instance of object

DEX vs. Java Bytecode

Flow Control instructions

DEX Opcode	Java Bytecode	Purpose
32..37: if-* 38..3d: if-*z	a0-a6: if_icmp* 99-9e: if*	Branch on logical
2b: packed-switch	ab: lookupswitch	Switch statement,
2c: sparse-switch	aa: tableswitch	Switch statement
28: goto 29: goto/16 30: goto/32	a7: goto c8: goto_w	Jump to offset in code
27: throw	bf:athrow	Throw exception

DEX vs. Java Bytecode

Data Instructions

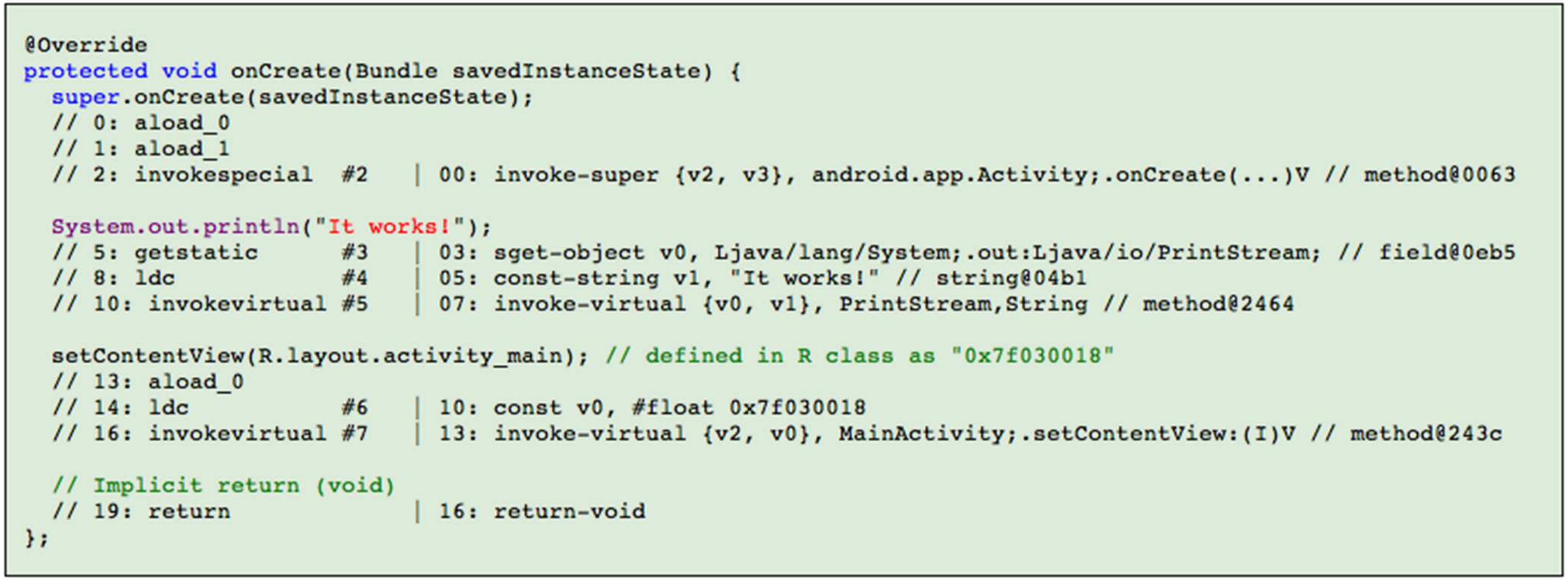
DEX Opcode	Java Bytecode	Purpose
12-1c: const*	12:ldc 13: ldc_w 14: ldc2_w	Define Constant
21: array-length	be: arraylength	Get length of an array
23: new-array	bd: anewarray	Instantiate an array
24-25: filled-new-array[/range] 26: fill-array-data	N/A	Populate an array

Arithmetic instructions are, likewise, highly similar

DEX vs. Java Bytecode

- Example: A “Hello World” activity:

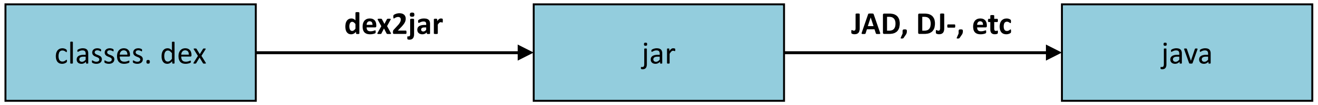
Listing d-dec: Demonstrating Java source, class and DEX bytecode



```
@Override
protected void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);
  // 0: aload_0
  // 1: aload_1
  // 2: invokespecial  #2   | 00: invoke-super {v2, v3}, android.app.Activity;.onCreate(...)V // method@0063

  System.out.println("It works!");
  // 5: getstatic      #3   | 03: sget-object v0, Ljava/lang/System;.out:Ljava/io/PrintStream; // field@0eb5
  // 8: ldc            #4   | 05: const-string v1, "It works!" // string@04b1
  // 10: invokevirtual #5   | 07: invoke-virtual {v0, v1}, PrintStream,String // method@2464

  setContentView(R.layout.activity_main); // defined in R class as "0x7f030018"
  // 13: aload_0
  // 14: ldc           #6   | 10: const v0, #float 0x7f030018
  // 16: invokevirtual #7   | 13: invoke-virtual {v2, v0}, MainActivity;.setContentView:(I)V // method@243c

  // Implicit return (void)
  // 19: return             | 16: return-void
};
```

DEX to Java

- It comes as no surprise that .dex and .class are isomorphic
- DEX debug items map DEX offsets to Java line numbers
- Dex2jar tool can easily “decompile” from .dex back to a .jar
- Standard Practice:



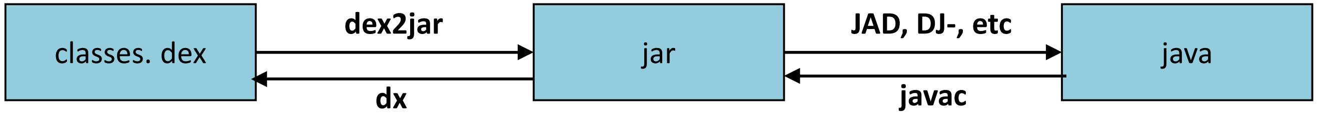
DEX file from a given .apk

Single JAR file with
multiple .class files

Source code:
- : no comments
+ : nicely indented

- Extremely useful for reverse engineering
  - Even more so useful for malice and mischief

DEX to Java



classes. dex

dex2jar

dx

jar

JAD, DJ-, etc

javac

java

DEX file from a given .apk

Single JAR file with
multiple .class files

Source code:
- : no comments
+ : nicely indented

- Flow from DEX to JAVA is **<u>bidirectional</u>**, meaning that an attacker can:
  - Decompile your code back to Java
  - Remove annoyances like ads, registration
  - Uncover sensitive data (app logic, or poorly guarded secrets)
  - Replace certain classes with others, potentially malicious ones
  - Recompile back to JAR, then DEX
  - Put cloned/trojaned version of your app on Play or another market

- ASEC/OBB “solutions” for this fail miserably when target device is rooted.

DEX Obfuscation

- Quite a few DEX "obfuscators" exist, with different approaches:
  - Functionally similar to binutils' `strip`, either java (ProGuard) or sDEX
    - Rename methods, field and class names
    - Break down string operations so as to "chop" hard-coded strings, or encrypt
    - Can use dynamic class loading (DexLoader classes) to impede static analysis
  - Can add dead code and dummy loops (at minor impact to performance)
  - Can also use goto into other instructions (e.g. DexLABS)
- In practice, quite limited, due to:
  - Reliance on Android Framework APIs (which remain unobfuscated)
  - JDWP and application debuggability at the Java level
  - If Dalvik can execute it, so can a proper analysis tool (e.g. IDA, dextra)
  - Popular enough obfuscators (e.g. DexGuard) have de-obfuscators...
- ... Which is why JNI is so popular

DEX Optimization (dexopt)

- Pre-5.0, installd runs `dexopt` on APK, during installation
  - Extracts the APK’s classes.dex
  - Performs runtime verification and optimization
  - Plops optimized DEX file in /data/dalvik-cache

```
root@android:/data/dalvik-cache # ls -s
total 28547
24 system@app@ApplicationsProvider.apk@classes.dex
1359 system@app@Browser.apk@classes.dex
958 system@app@Contacts.apk@classes.dex
625 system@app@ContactsProvider.apk@classes.dex
99 system@app@DeskClock.apk@classes.dex
795 system@app@DownloadProvider.apk@classes.dex
13 system@app@DrmProvider.apk@classes.dex
...
root@android# file system\@app\@LatinIME.apk\@classes.dex
system@app@LatinIME.apk@classes.dex: Dalvik dex file (optimized for host) version 036
```

- **ART still optimizes DEX**, but uses `dex2oat` instead (q.v. Part II)
  - ODEX files are actually now OAT files (ELF shared objects)
  - Actual DEX payload buried deep inside

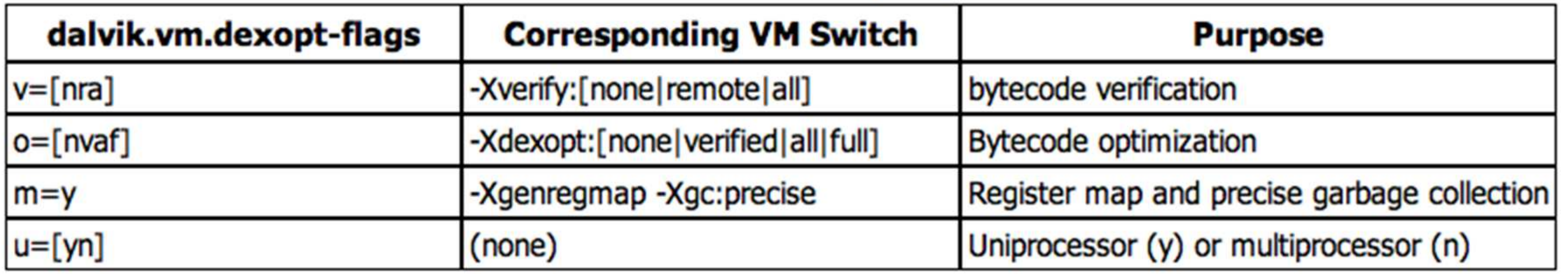
DEX Optimization (dexopt)

- `dexopt` is user-friendly ... But only for the right user (`installd`)

```
shell@hammerhead:/ $ dexopt
Usage:

Short version: Don't use this.

Slightly longer version: This system-internal tool is used to
produce optimized dex files. See the source code for details.
```

- The program runs a Dalvik VM with special switches

Table d-dexopt: Dexopt flags



dalvik.vm.dexopt-flags	Corresponding VM Switch	Purpose
v=[nra]	-Xverify:[none|remote|all]	bytecode verification
o=[nvaf]	-Xdexopt:[none|verified|all|full]	Bytecode optimization
m=y	-Xgenregmap -Xgc:precise	Register map and precise garbage collection
u=[yn]	(none)	Uniprocessor (y) or multiprocessor (n)

DEX Optimization (dexopt)

- What happens during optimization?
  - Bytecode verification: Deducing code paths, register mapss, and precise GC
  - Wrapping with ODEX header (for optimized data/dependency tables)
  - Opcodes replaced by quick opcode variants*

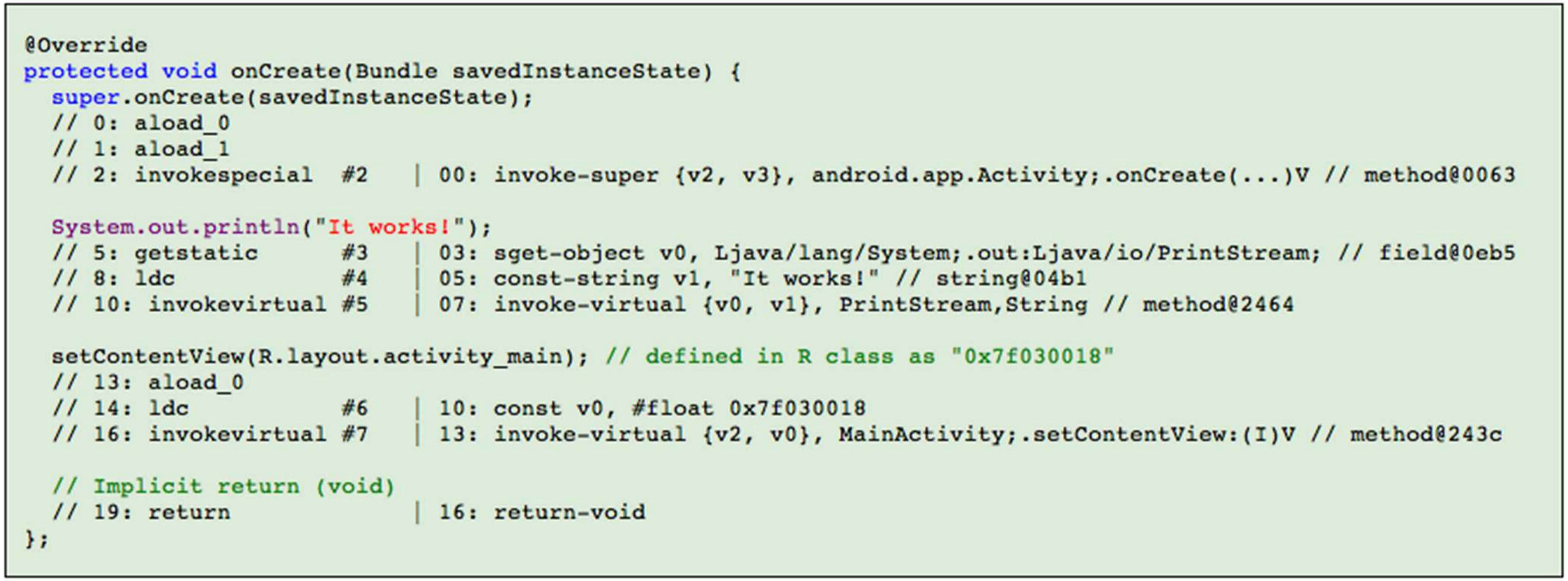
art/compiler/dex/dex_to_dex_compiler.cc

DEX Opcode	ODEX Opcode	Optimization
0e: return-void	73: return-void-barrier	Barrier (in constructors)
52:iget	e3: iget-quick	Use byte offset for field, eliminating costly lookup, and merge primitive datatypes into a 32-bit (wide) instruction, reducing overall code size.
53: iget-wide	e4: iget-wide-quick	
54: iget-object	e5:iget-object-quick	
59: iput	e6: iput-quick	
5a: iput-wide	e7: iput-wide-quick	
5b: iput-object	e8: iput-object-quick	
6e: invoke-virtual	e9/ea: invoke-virtual-quick[/range]	Vtable, eliminating lookup

*** - Pre-ART optimization also added execute-inline, as well as –volatile variants for iget/iput – but those have been removed**

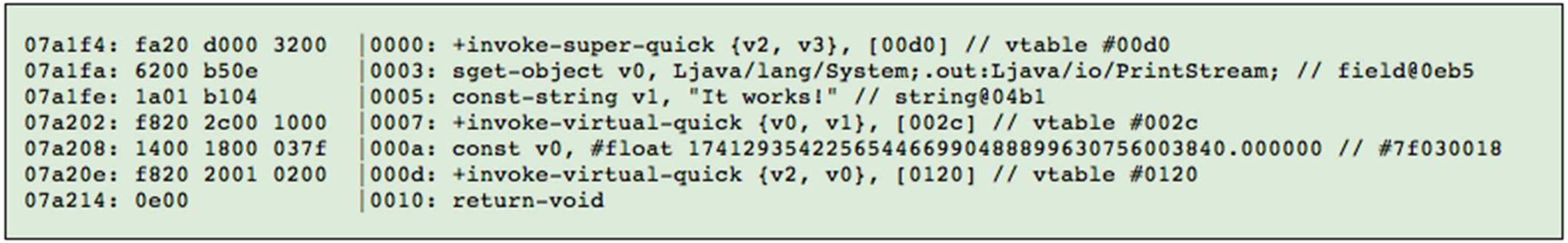
DEX Optimization (dexopt)

Listing d-dec: Demonstrating Java source, class and DEX bytecode



```
@Override
protected void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);
  // 0: aload_0
  // 1: aload_1
  // 2: invokespecial  #2   | 00: invoke-super {v2, v3}, android.app.Activity;.onCreate(...)V // method@0063

  System.out.println("It works!");
  // 5: getstatic      #3   | 03: sget-object v0, Ljava/lang/System;.out:Ljava/io/PrintStream; // field@0eb5
  // 8: ldc            #4   | 05: const-string v1, "It works!" // string@04b1
  // 10: invokevirtual #5   | 07: invoke-virtual {v0, v1}, PrintStream,String // method@2464

  setContentView(R.layout.activity_main); // defined in R class as "0x7f030018"
  // 13: aload_0
  // 14: ldc           #6   | 10: const v0, #float 0x7f030018
  // 16: invokevirtual #7   | 13: invoke-virtual {v2, v0}, MainActivity;.setContentView:(I)V // method@243c

  // Implicit return (void)
  // 19: return             | 16: return-void
};
```

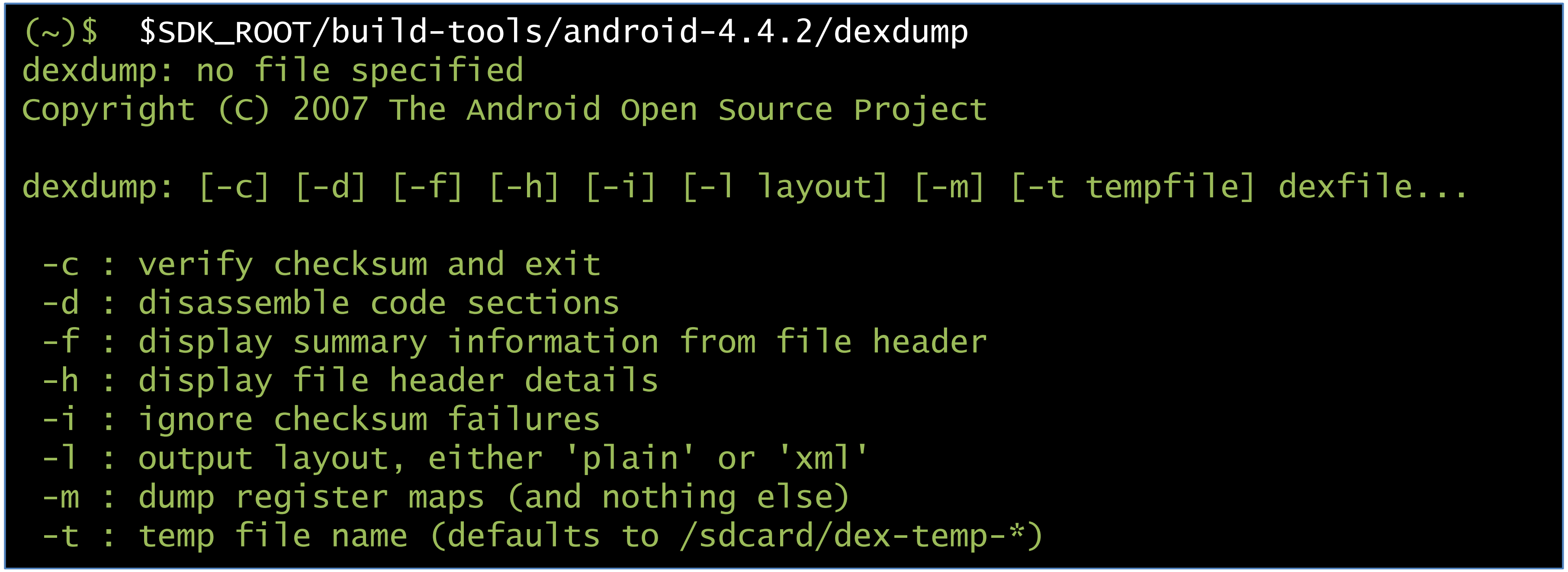
Listing d-optdump: Optimized DEX version of sample App's OnCreate()



```
07a1f4: fa20 d000 3200   |0000: +invoke-super-quick {v2, v3}, [00d0] // vtable #00d0
07a1fa: 6200 b50e        |0003: sget-object v0, Ljava/lang/System;.out:Ljava/io/PrintStream; // field@0eb5
07a1fe: 1a01 b104        |0005: const-string v1, "It works!" // string@04b1
07a202: f820 2c00 1000   |0007: +invoke-virtual-quick {v0, v1}, [002c] // vtable #002c
07a208: 1400 1800 037f   |000a: const v0, #float 174129354225654466990488899630756003840.000000 // #7f030018
07a20e: f820 2001 0200   |000d: +invoke-virtual-quick {v2, v0}, [0120] // vtable #0120
07a214: 0e00             |0010: return-void
```

Example: Reversing DEX

- You can use the AOSP-supplied DEXDUMP to disassemble DEX

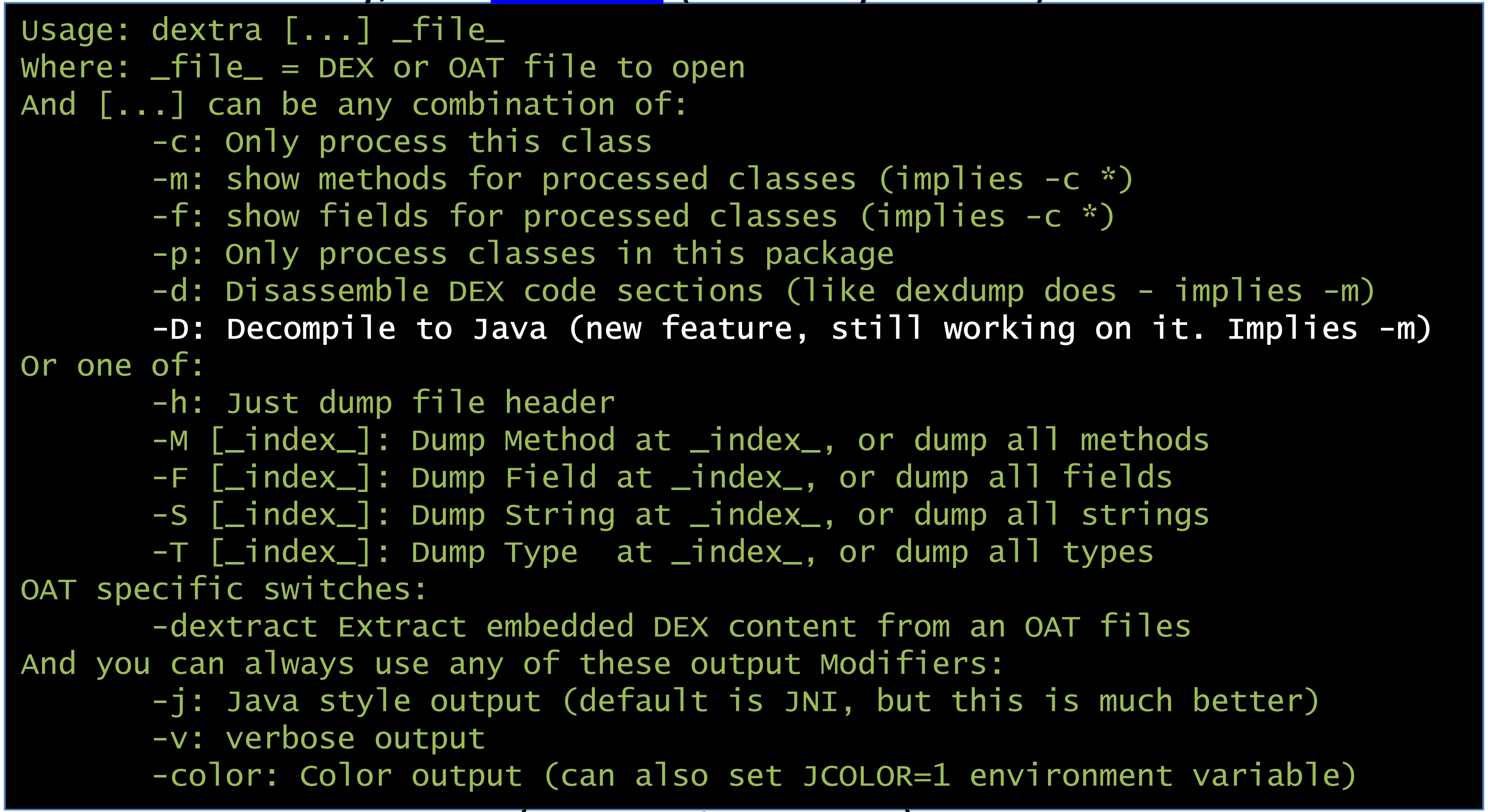


```
(~)$  $SDK_ROOT/build-tools/android-4.4.2/dexdump
dexdump: no file specified
Copyright (C) 2007 The Android Open Source Project

dexdump: [-c] [-d] [-f] [-h] [-i] [-l layout] [-m] [-t tempfile] dexfile...

 -c : verify checksum and exit
 -d : disassemble code sections
 -f : display summary information from file header
 -h : display file header details
 -i : ignore checksum failures
 -l : output layout, either 'plain' or 'xml'
 -m : dump register maps (and nothing else)
 -t : temp file name (defaults to /sdcard/dex-temp-*)
```

(Interactive Demo)

Example: Reversing DEX

- Alternatively, use DEXTRA (formerly Dexter)

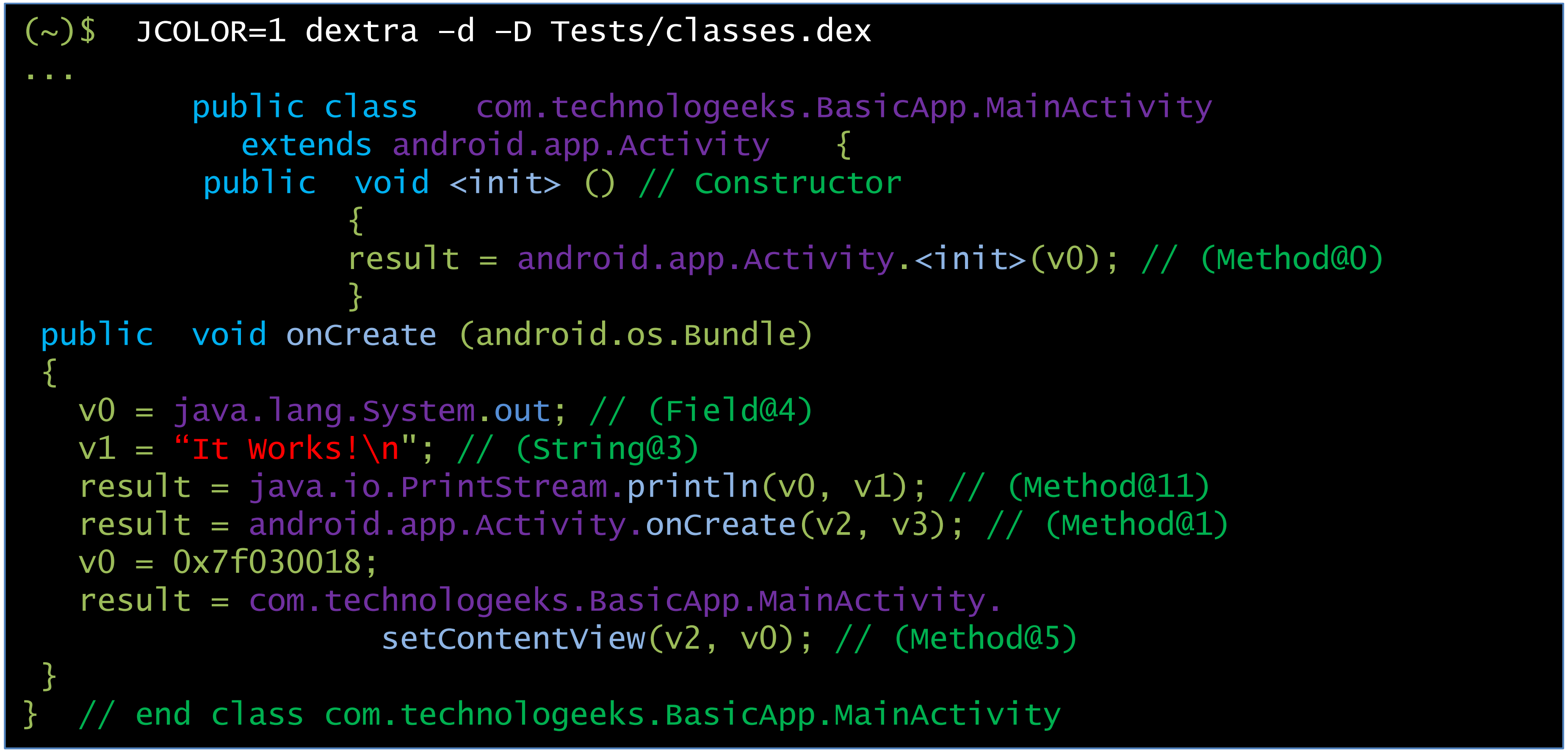


```
Usage: dextra [...] _file_
Where: _file_ = DEX or OAT file to open
And [...] can be any combination of:
       -c: Only process this class
       -m: show methods for processed classes (implies -c *)
       -f: show fields for processed classes (implies -c *)
       -p: Only process classes in this package
       -d: Disassemble DEX code sections (like dexdump does - implies -m)
       -D: Decompile to Java (new feature, still working on it. Implies -m)
Or one of:
       -h: Just dump file header
       -M [_index_]: Dump Method at _index_, or dump all methods
       -F [_index_]: Dump Field at _index_, or dump all fields
       -S [_index_]: Dump String at _index_, or dump all strings
       -T [_index_]: Dump Type  at _index_, or dump all types
OAT specific switches:
       -dextract Extract embedded DEX content from an OAT files
And you can always use any of these output Modifiers:
       -j: Java style output (default is JNI, but this is much better)
       -v: verbose output
       -color: Color output (can also set JCOLOR=1 environment variable)
```

(Interactive Demo)

Example: Reversing DEX

- You can use the AOSP-supplied DEXDUMP to disassemble DEX



```
(~)$  JCOLOR=1 dextra -d -D Tests/classes.dex
...
         public class   com.technologeeks.BasicApp.MainActivity
           extends android.app.Activity    {
          public  void <init> () // Constructor
                 {
                 result = android.app.Activity.<init>(v0); // (Method@0)
                 }
 public  void onCreate (android.os.Bundle)
 {
   v0 = java.lang.System.out; // (Field@4)
   v1 = “It works!\n"; // (String@3)
   result = java.io.PrintStream.println(v0, v1); // (Method@11)
   result = android.app.Activity.onCreate(v2, v3); // (Method@1)
   v0 = 0x7f030018;
   result = com.technologeeks.BasicApp.MainActivity.
                   setContentView(v2, v0); // (Method@5)
 }
}  // end class com.technologeeks.BasicApp.MainActivity
```

(Interactive Demo)

So why is Dalvik deprecated?

- JIT is slow, consuming both cycles and battery power
- Garbage collection (esp. GC_FOR_ALLOC) causes hangs/jitter
- Dalvik VM is 32-bit, and can't benefit from 64-bit architecture
  - And everybody* wants 64-bit, now that iOS has it...
- KitKat was the first to introduce ART, And Lollipop adopts it
  - For more details on ART Internals, stick around for Part II..

*** - Well, maybe everybody except Qualcomm... Or .. On second thought, maybe they do, too?**

Take Away

- DEX is a Dangerous Executable format...
  - Risks to app developers are significant, with no clear solutions
  - (And we haven’t even mentioned fun with DEX fuzzing)

- DEX isn’t DEAD yet – even with ART:
  - Still buried deep inside those OAT files
  - FAR easier to reverse engineer embedded DEX, than do so for OAT

Parts we didn’t discuss here are in the book

Stick around for Part II – after the break!