Reconstructing DALVIK Applications

Marc Schönefeld

CANSECWEST 2009, MAR18

Motivation

- As a reverse engineer I have the tendency to look in the code that is running on my mobile device
- Coming from a JVM background I wanted to know what Dalvik is really about
- The final motivation to write a converter from Dalvik bytecode to JVM bytecode came while I was working on my tax declaration, which is deferred since then :)

What is dalvik

- Dalvik is the runtime that runs userspace Android applications
  - invented by Dan Bornstein
  - named after a village in Iceland
  - register-based
  - runs Dalvik bytecode instructions (not java bytecode)

Dalvik vs. JVM

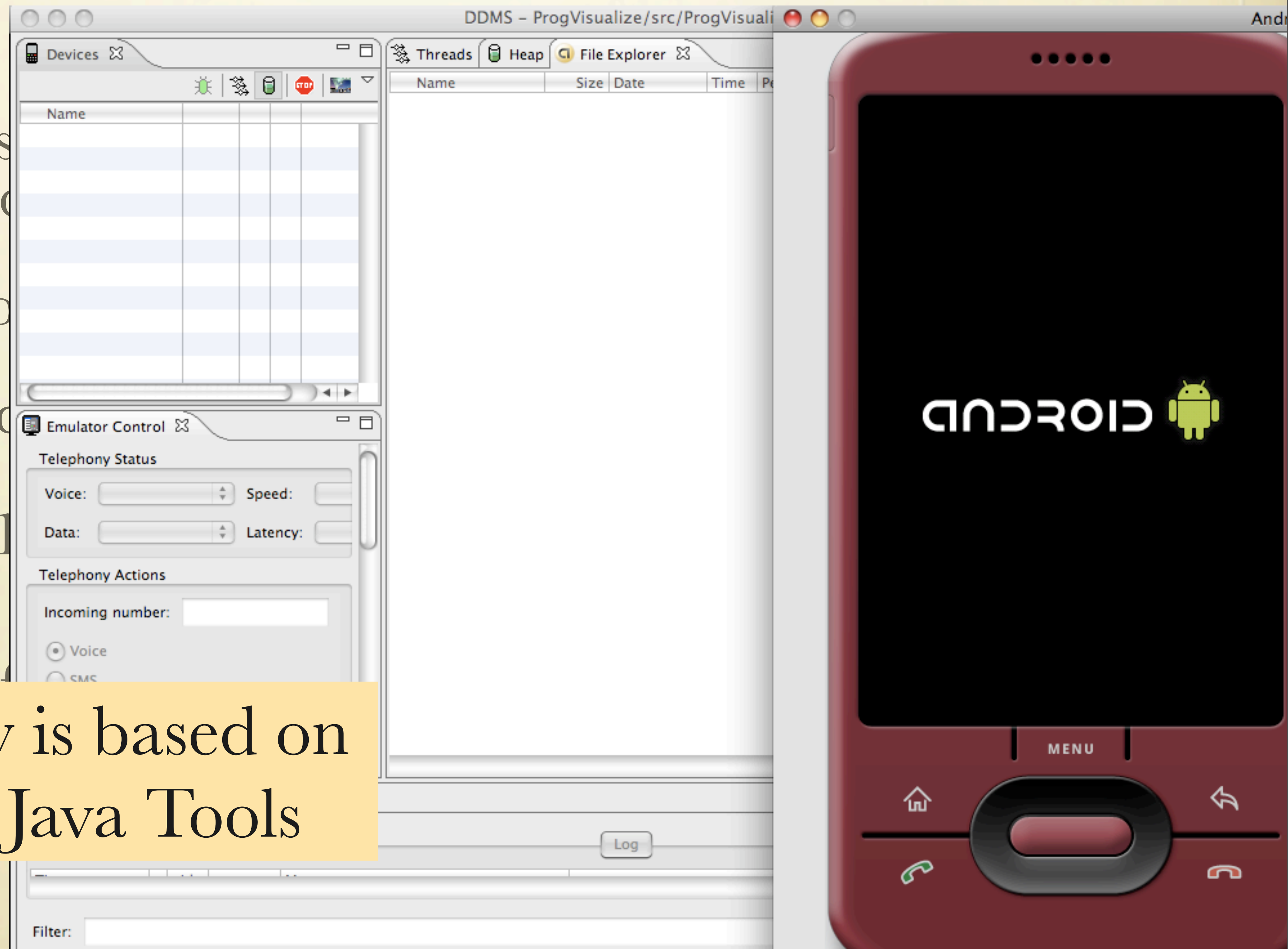
Criteria	Dalvik	JVM
Architecture	Register-based	Stack-based
OS-Support	Android	All
RE-Tools	a few (dexdump,ddx)	many (jad, bcel, findbugs, ...)
Executables	DEX	JAR
Constant-Pool	per Application	per Class

Dalvik development process

- Dalvik apps are developed using java developer tools on a standard desktop system (like eclipse),
- compiled to java classes (javac)
- transformed to DX with the dx tool (classes.dex)
- classes.dex plus meta data and resources go into a dalvik application 'apk' container
- this is transferred to the device or an emulator (adb, or download from android market)

Dalvik development process

- Dalvik apps a standard
- compiled to
- transformed
- classes.dex application



Dalvik Dev is based on standard Java Tools

Dalvik vs. Java highlevel

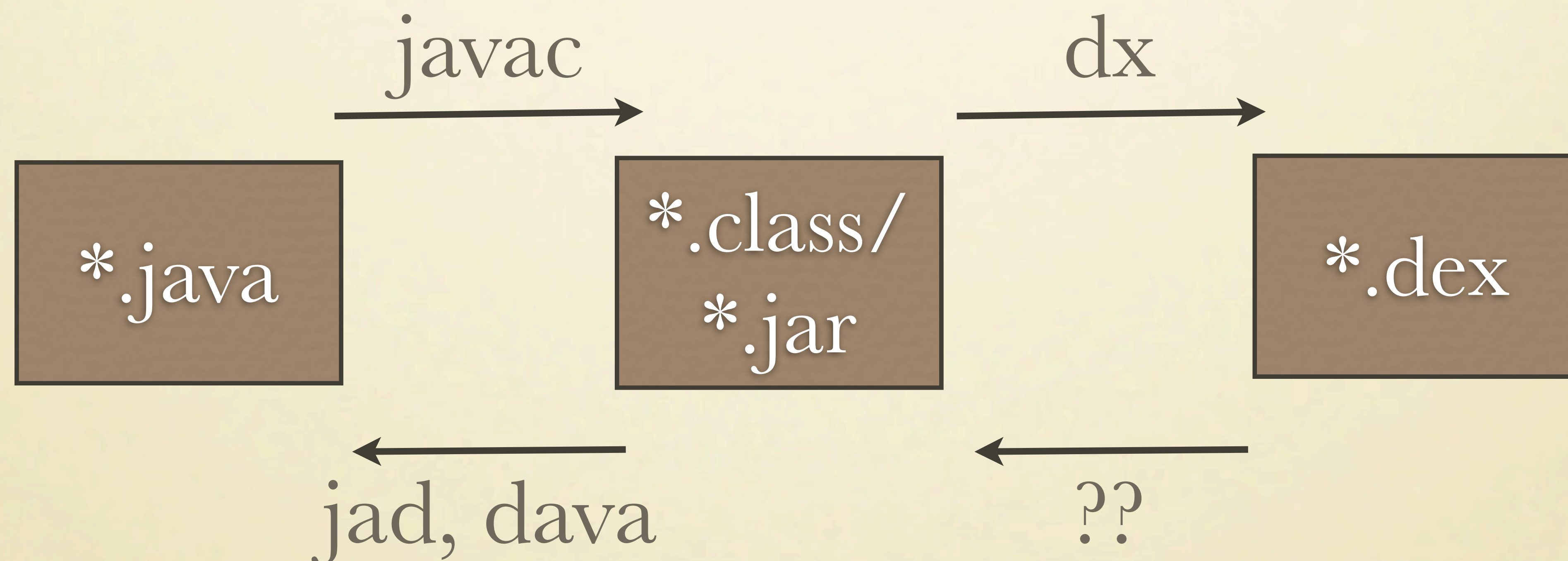
	Dalvik	Standard Java
java.io	Y	Y
java.net	Y	Y
android.*	Y	N
com.google.*	Y	N
javax.swing.*	N	Y
...	...	...

Dalvik development from a RE perspective

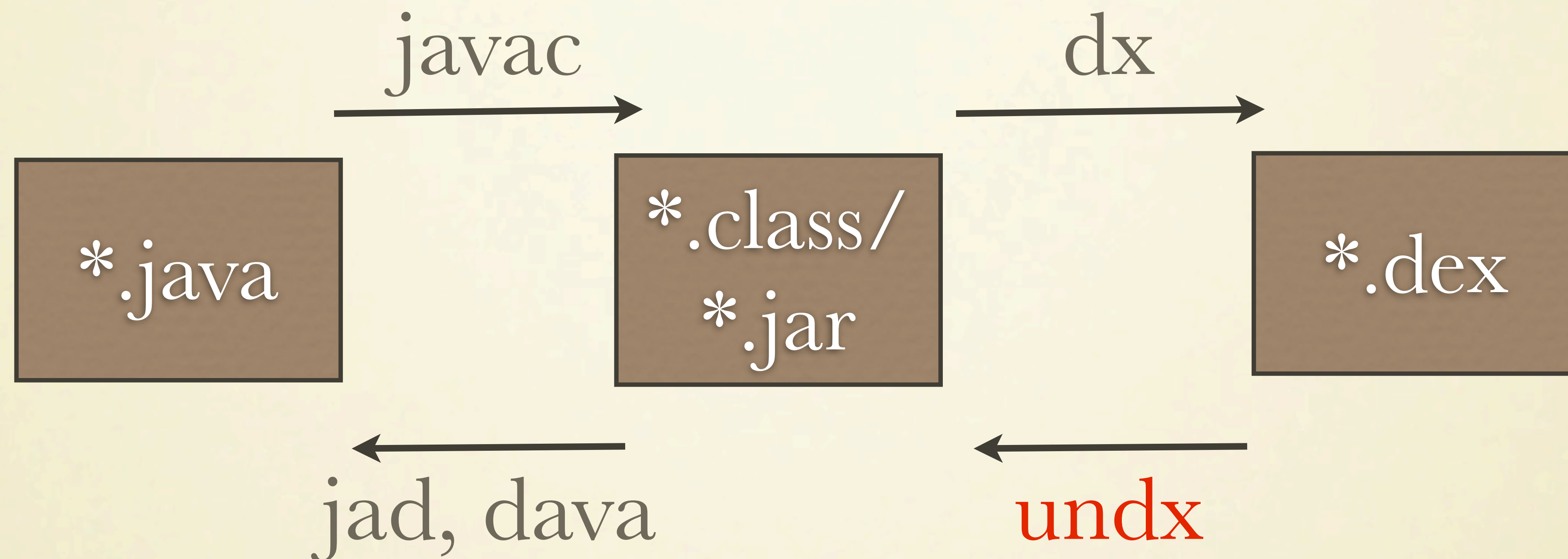
- Dalvik applications are available as apk files, no source included, so you buy/download a cat in the bag. How can you find out, whether
  - the application contains malicious code, ad/spyware, or phones home to the vendor ?
  - has unpatched security holes (dex generated from vulnerable java code) ?
  - contains copied code, which may violate GPL or other license agreements ?

A Dalvik Reconstruction Tool

- To find out about these open questions we need to close the gap in the current dalvik development process



Close the gap



- to decompile/reconstruct Dalvik code, only a transformer from dex to jar is necessary
- sounds simple, but wait for the details

Tool Design choices

- How to to parse dex files?
  - write a DEX parser
  - or utilize something existing
- How to translate to class files?
  - how to write the java classes

Parsing DEX files

- The dexdump tool of the android sdk can perform a complete dump of dex files, it is used by undx

	using Dexdump	**parsing dex files directly**
Speed	Time advantage, don't have to write everything from	Direct access to binary structures (arrays, jump
Control	(-) Dexdump has a number of nasty bugs	make a bug , fix a bug
Available info	Dexdump omits info	it's all in the dex

- The decision was to use as much of useable information from dexdump, for the rest we parse the dex file directly

Parsing DEX files

- This is useful dexdump output, good to parse

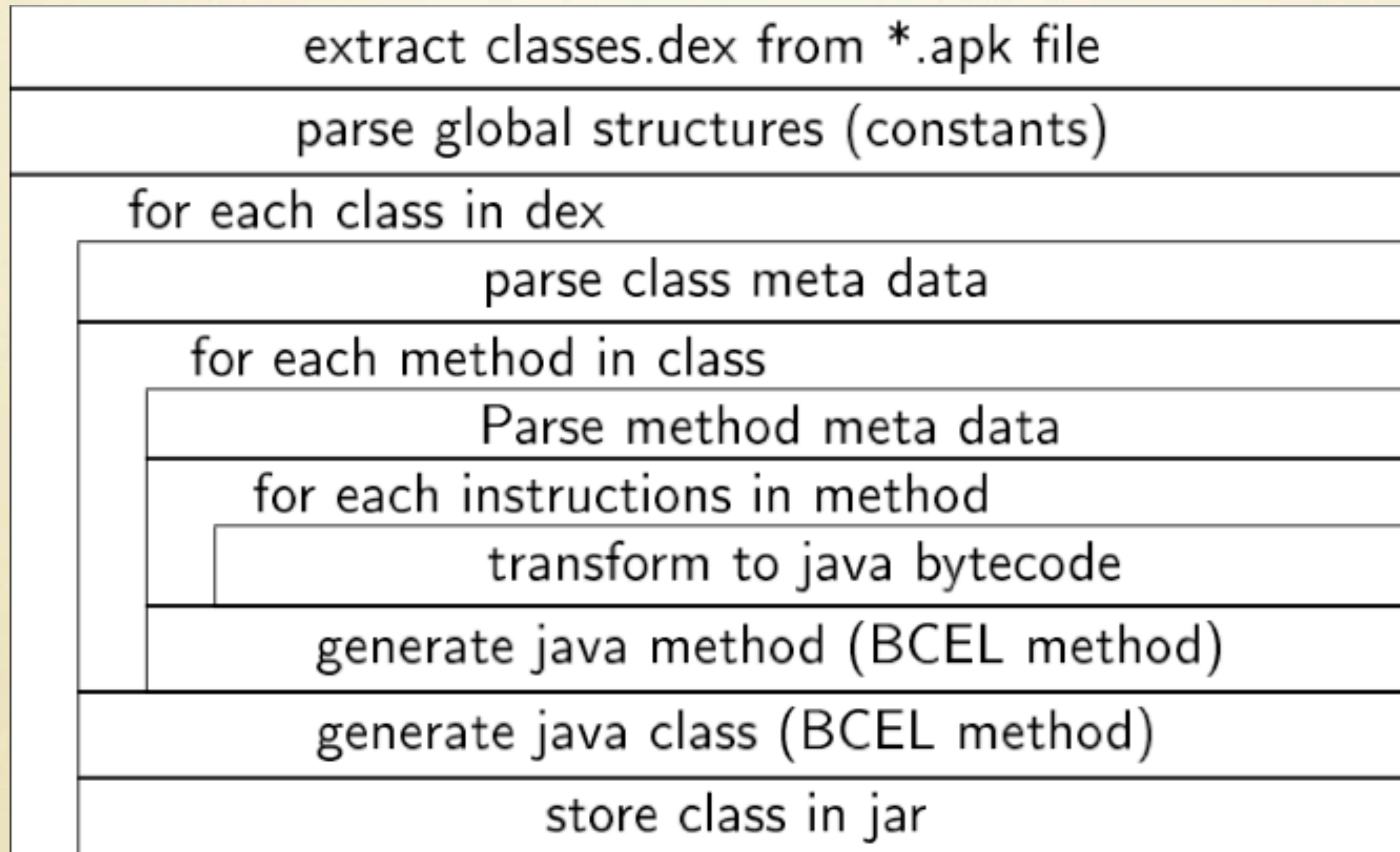
```
    #1              : (in LUnDxTest;)
      name          : 'main'
      type          : '([Ljava/lang/String;)V'
      access        : 0x0009 (PUBLIC STATIC)
      code          -
      registers     : 3
      ins           : 1
      outs          : 2
      insns size    : 20 16-bit code units
00024c:                      |[00024c] UnDxTest.main:([Ljava/lang/String;)V
00025c: 2200 0200            |0000: new-instance v0, LObj; // class@0002
000260: 7010 0000 0000       |0002: invoke-direct {v0}, LObj;.<init>:()V // method@0000
000266: 1251                 |0005: const/4 v1, #int 5 // #5
000268: 6e20 0200 1000       |0006: invoke-virtual {v0, v1}, LObj;.doit:(I)V // method@0002
00026e: 1301 ff00            |0009: const/16 v1, #int 255 // #ff
000272: 6e20 0200 1000       |000b: invoke-virtual {v0, v1}, LObj;.doit:(I)V // method@0002
000278: 1301 f000            |000e: const/16 v1, #int 240 // #f0
00027c: 6e20 0200 1000       |0010: invoke-virtual {v0, v1}, LObj;.doit:(I)V // method@0002
000282: 0e00                 |0013: return-void
```

Parsing DEX files

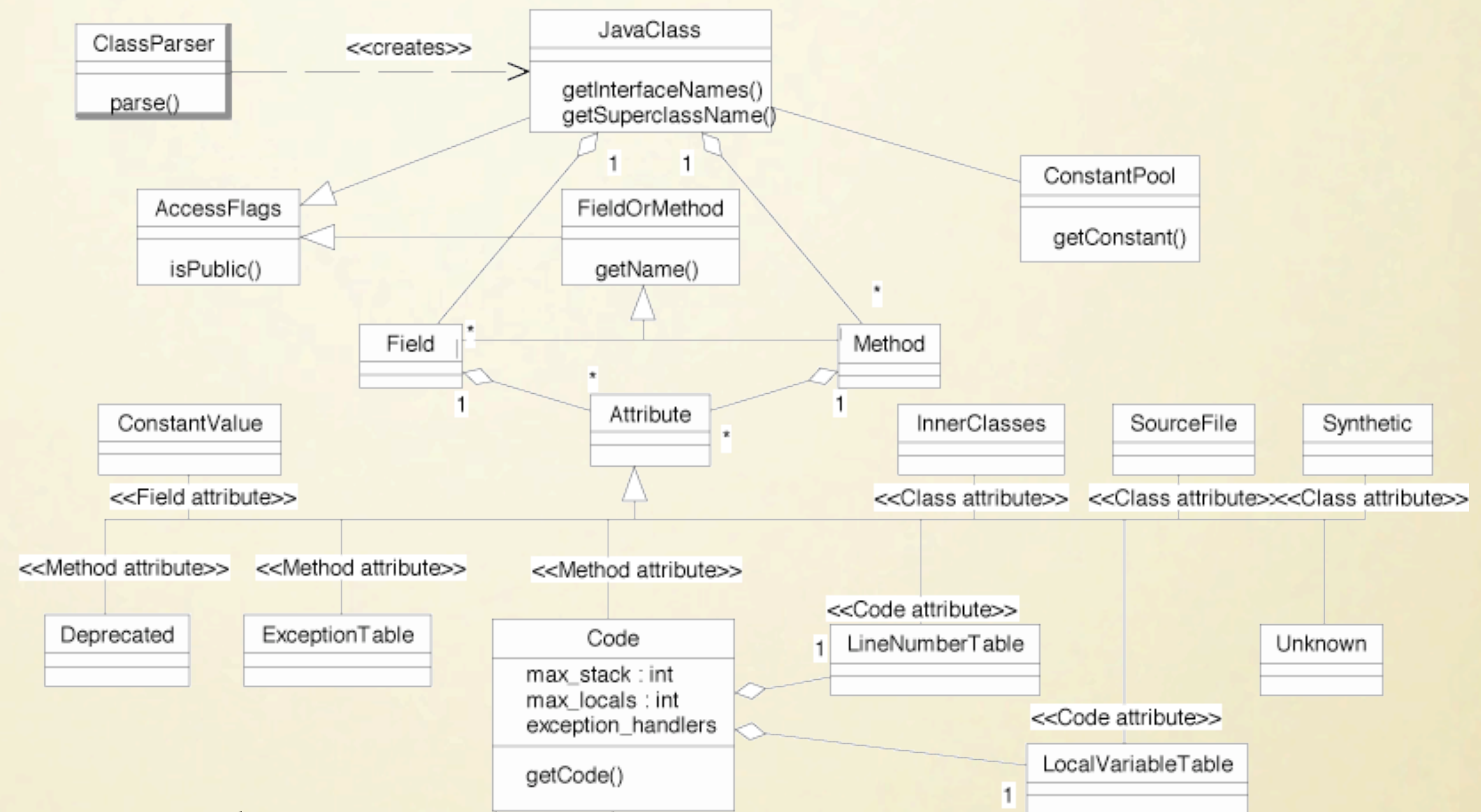
- This is incomplete dexdump output, where is the rest of array-data (only first four data units shown) ?

```
name          : '<clinit>'
type          : '()V'
access        : 0x10008 (STATIC CONSTRUCTOR)
code          -
registers     : 1
ins           : 0
outs          : 0
insns size    : 34 16-bit code units
000310:                                        |[000310] MD5.<clinit>:()V
000320: 1200                                   |0000: const/4 v0, #int 0 // #0
000322: 6900 0200                              |0001: sput-object v0, LMD5;.md5:LMD5; // field@0002
000326: 1300 1000                              |0003: const/16 v0, #int 16 // #10
00032a: 2300 0f00                              |0005: new-array v0, v0, [C // class@000f
00032e: 2600 0700 0000                         |0007: fill-array-data v0, 0000000e // +00000007
000334: 6900 0000                              |000a: sput-object v0, LMD5;.hexChars:[C // field@0000
000338: 0e00                                   |000c: return-void
00033a: 0000                                   |000d: nop // spacer
00033c: 0003 0200 1000 0000 3000 3100 3200 ... |000e: array-data (20 units)
catches       : (none)
positions     :
0x0000 line=7
0x0003 line=8
```

Structure

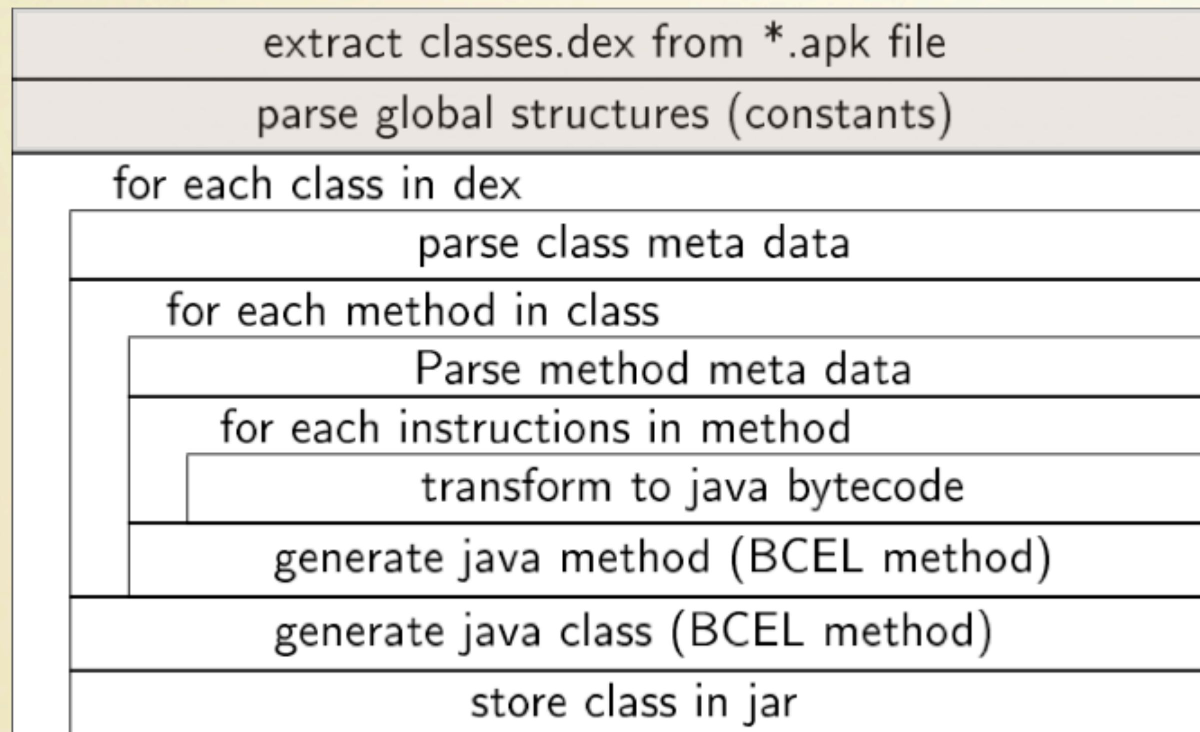


Generating java Code



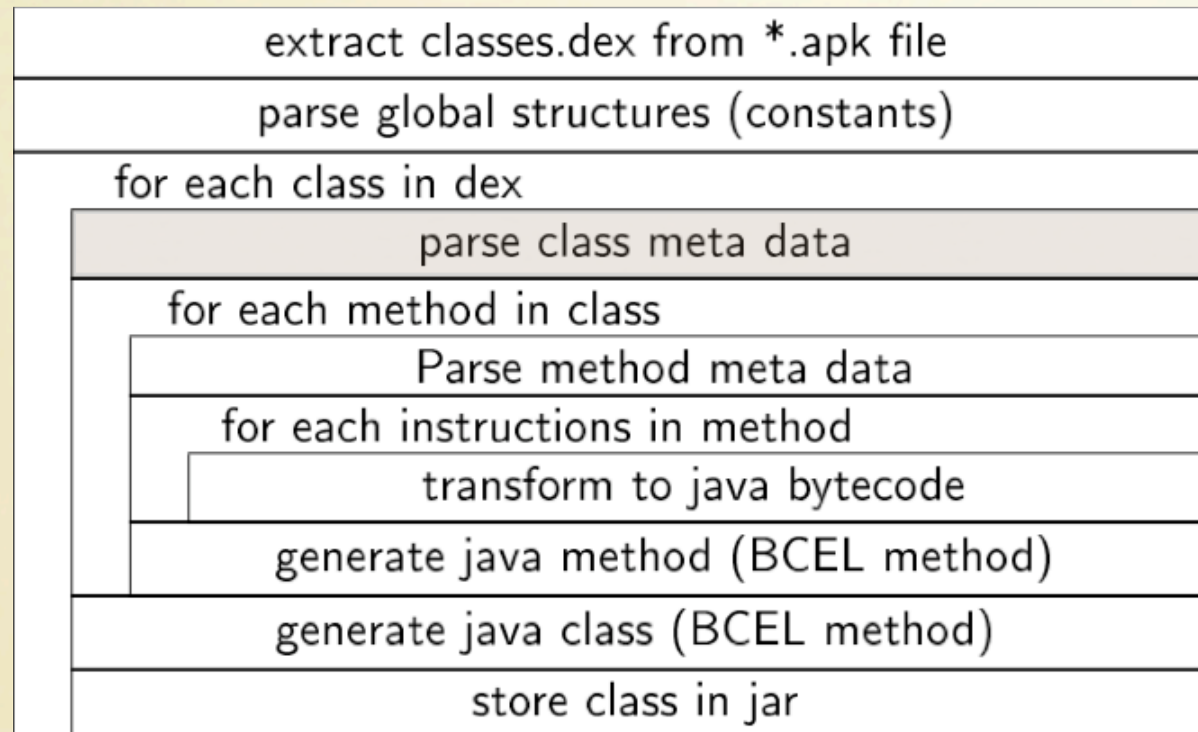
- http://jakarta.apache.org/bcel/
- We chose the BCEL library from Apache as it has a very broad functionality (compared to alternatives like are ASM and javassist)

Structure



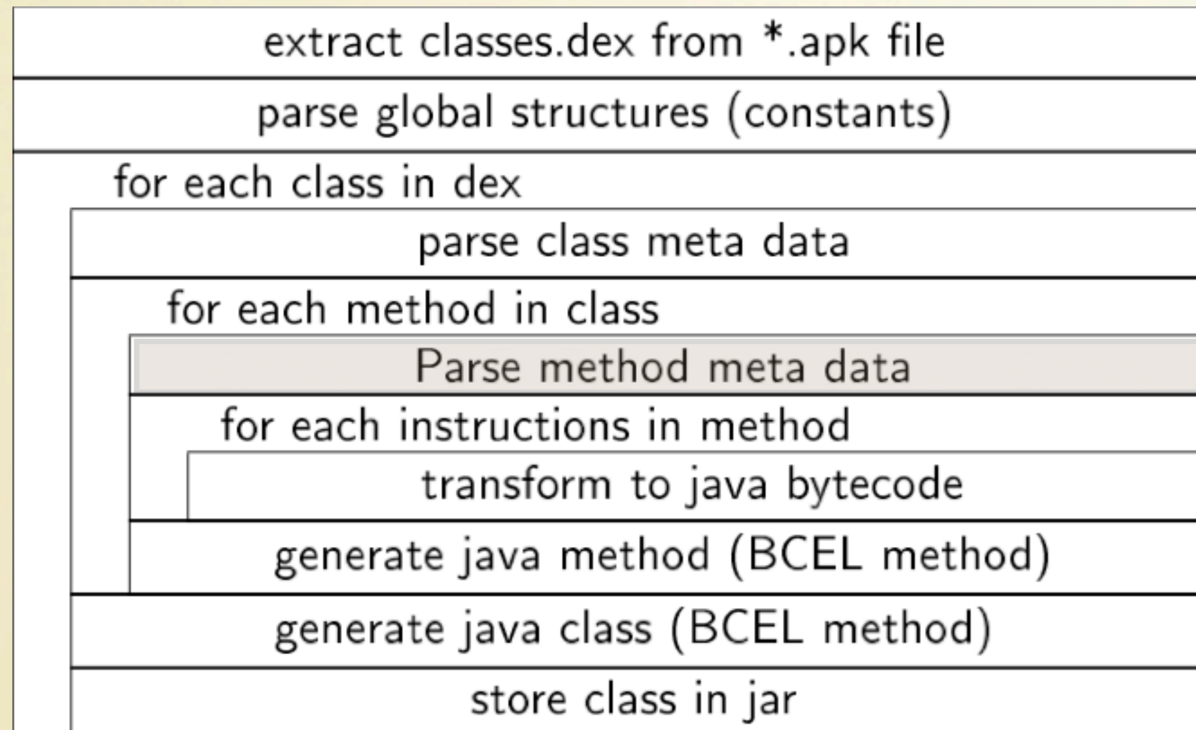
- In this step the global meta information is extracted and transformed into relevant BCEL constant structures
- Retrieve the string table which later becomes the Java constant pool

Class meta data



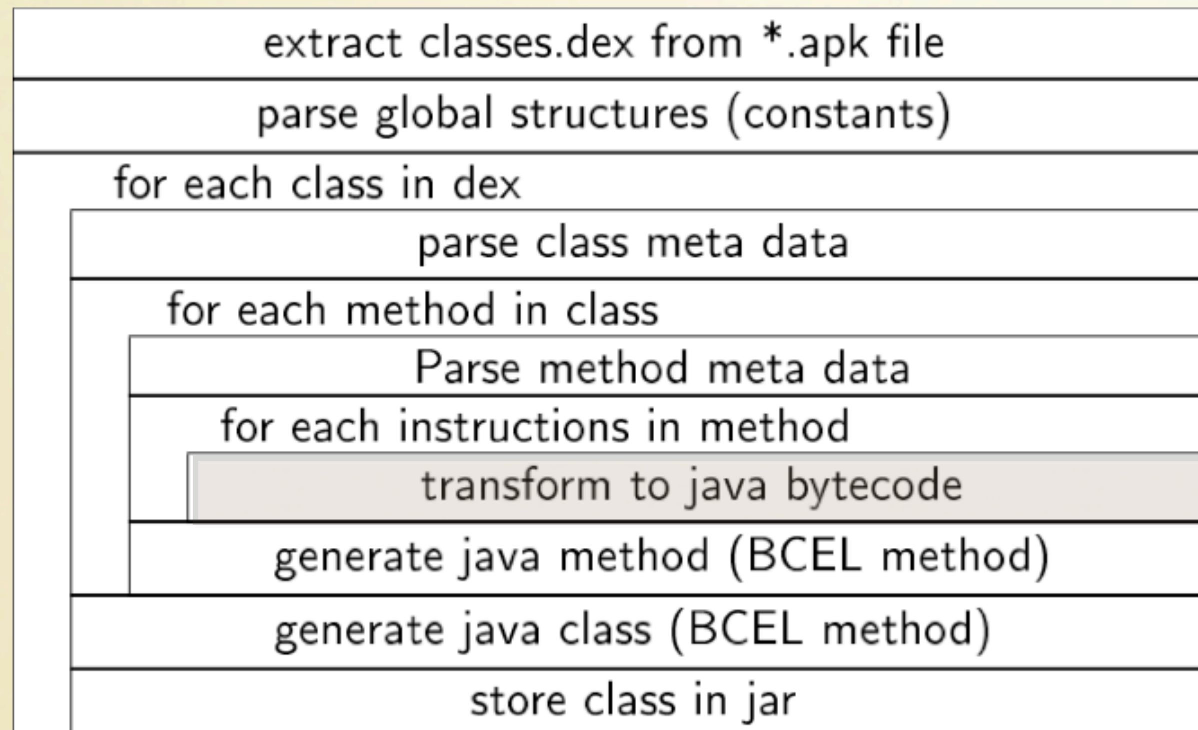
- In this step the class meta information is extracted and transformed into BCEL JavaClass structures
- Static and instance field information (incl. setting the appropriate visibility flags)

Method meta data



- method meta information is transformed into BCEL Method structures
- Extracting signatures,
- setting up local variable tables,
- mapping Dalvik registers to JVM registers

INSTRUCTIONS



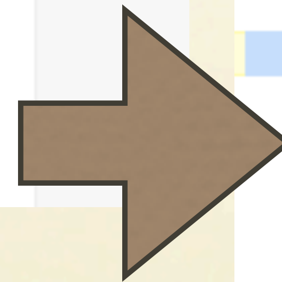
- First create BCEL InstructionList
- we create a NOP proxy for every Dalvik instruction, to have jump targets for forward jumps
- For every Dalvik instruction add an equivalent JVM bytecode to the InstructionList

CREATING JVM BYTECODE

```
IC)

s
         |[0003c0] MD5.getInstance:()LMD5;
         |0000: sget-object v0, LMD5;.md5:LMD5; // field@0002
         |0002: if-nez v0, 000b // +0009
         |0004: new-instance v0, LMD5; // class@0002
         |0006: invoke-direct {v0}, LMD5;.<init>:()V // method@0001
         |0009: sput-object v0, LMD5;.md5:LMD5; // field@0002
         |000b: sget-object v0, LMD5;.md5:LMD5; // field@0002
         |000d: return-object v0
```

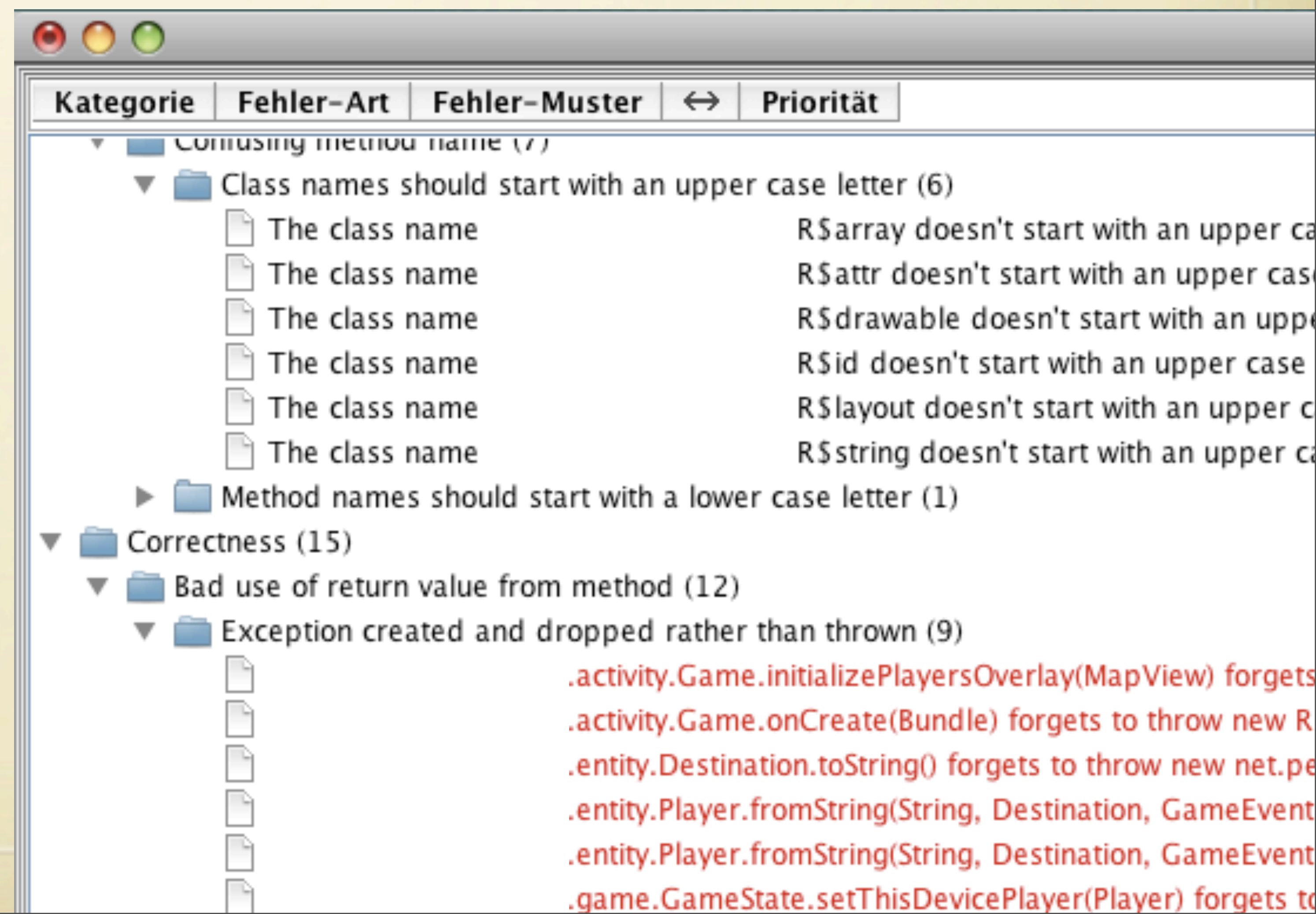
- Dalvik regs to jvm regs
- choose opcode sequence
  (obey stack balance!)
- type inference
  (by following data flow)



```
 public static MD5 getInstance();
Code:
 0:   getstatic       #14; //Field md5:LMD5;
 3:   astore_0
 4:   aload_0
 5:   ifnonnull       20
 8:   new     #4; //class MD5
 11:  astore_0
 12:  aload_0
 13:  invokespecial   #73; //Method "<init>":()V
 16:  aload_0
 17:  putstatic       #14; //Field md5:LMD5;
 20:  getstatic       #14; //Field md5:LMD5;
 23:  astore_0
 24:  aload_0
 25:  areturn
```

What to do with the JVM bytecode

- **Analyze it (findbugs)**
  - Check the codebase before running the APK
- Decompile it
- Compare it
- re-use it



What to do with the JVM bytecode

- Analyze it (findbugs)
- **Decompile it**
- Compare it
- re-use it

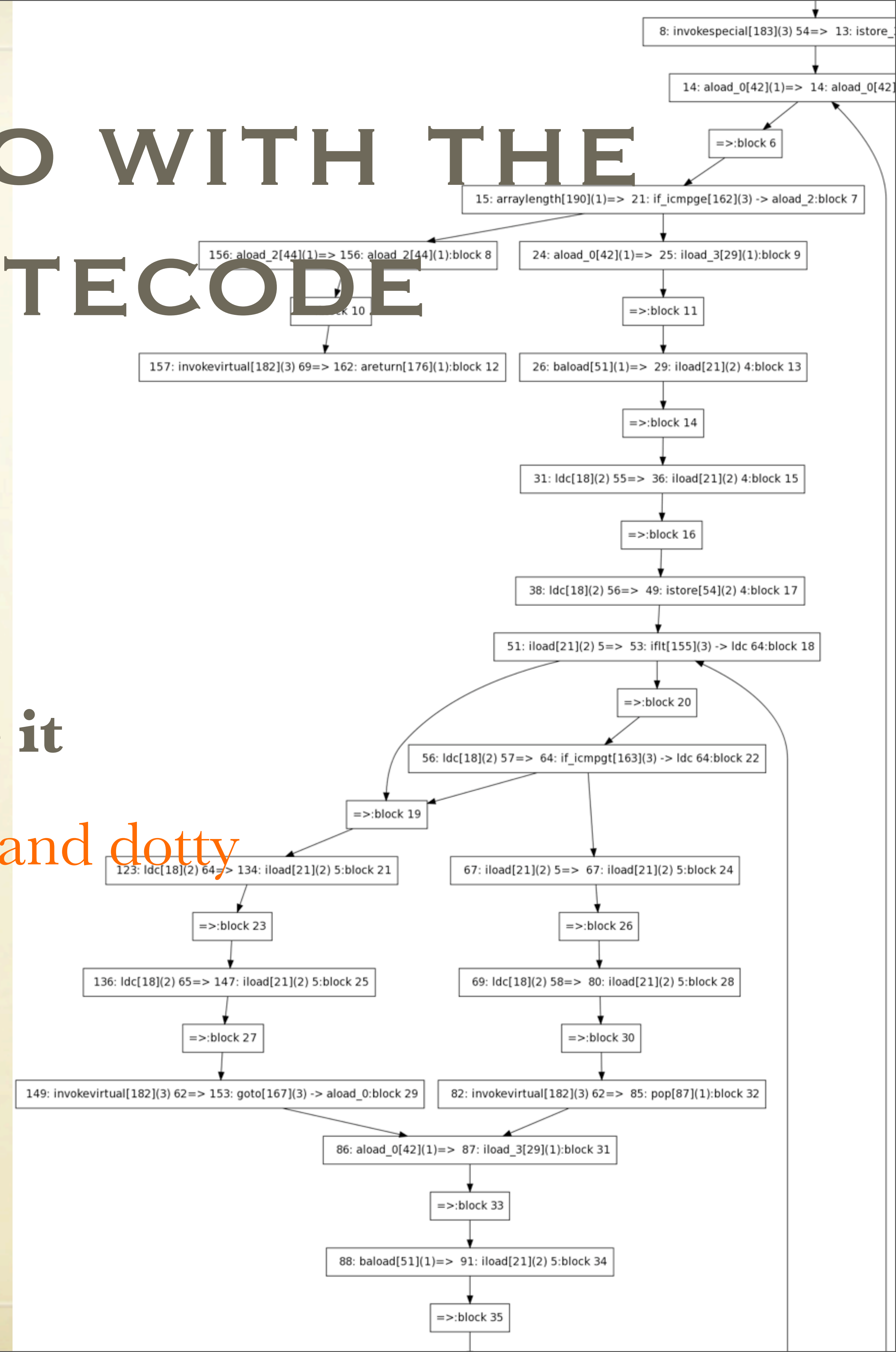
```
public class WebDialog extends Dialog
{

    public WebDialog(Context arg0)
    {
        super(arg0);
        Object obj = JVM INSTR new #14  <Class WebView>;
        ((WebView) (obj)).WebView(arg0);
        webView = ((WebView) (obj));
        obj = webView;
        obj = ((WebView) (obj)).getSettings();
        boolean flag = true;
        ((WebSettings) (obj)).setJavaScriptEnabled(flag);
        obj = webView;
        setContentView(((android.view.View) (obj)));
        obj = "Welcome";
        setTitle(((CharSequence) (obj)));
    }

    public void loadUrl(String arg0)
    {
        WebView webview = webView;
        webview.loadUrl(arg0);
```

What to do with the JVM bytecode

- Analyze it (findbugs)
- Decompile it
- **Graph and compare it**
  - 50 LOC with BCEL and dotty
-



TRIVIA

- Every tool needs a name, so I asked Dan Bornstein for a suggestion, he came up with **undx** , as it reverses the effects of dx
- Some facts:
  - 4000 lines of code (I am a horrible coder, no comments, but code repetitions, organically grown)
  - written in Java, only dependency is BCEL
  - command-line tool

Availability

- will be released at http://undx.sourceforge.net/
- give me two weeks from now, to remove some of the existing bugs (QA),
  - I am currently struggling with a bug (mine or BCEL) that causes problems for larger codebases
- will be published under GPL

Q & A

- marc.schoenefeld@gmx.org