ANDROID

# Dalvik VM Internals

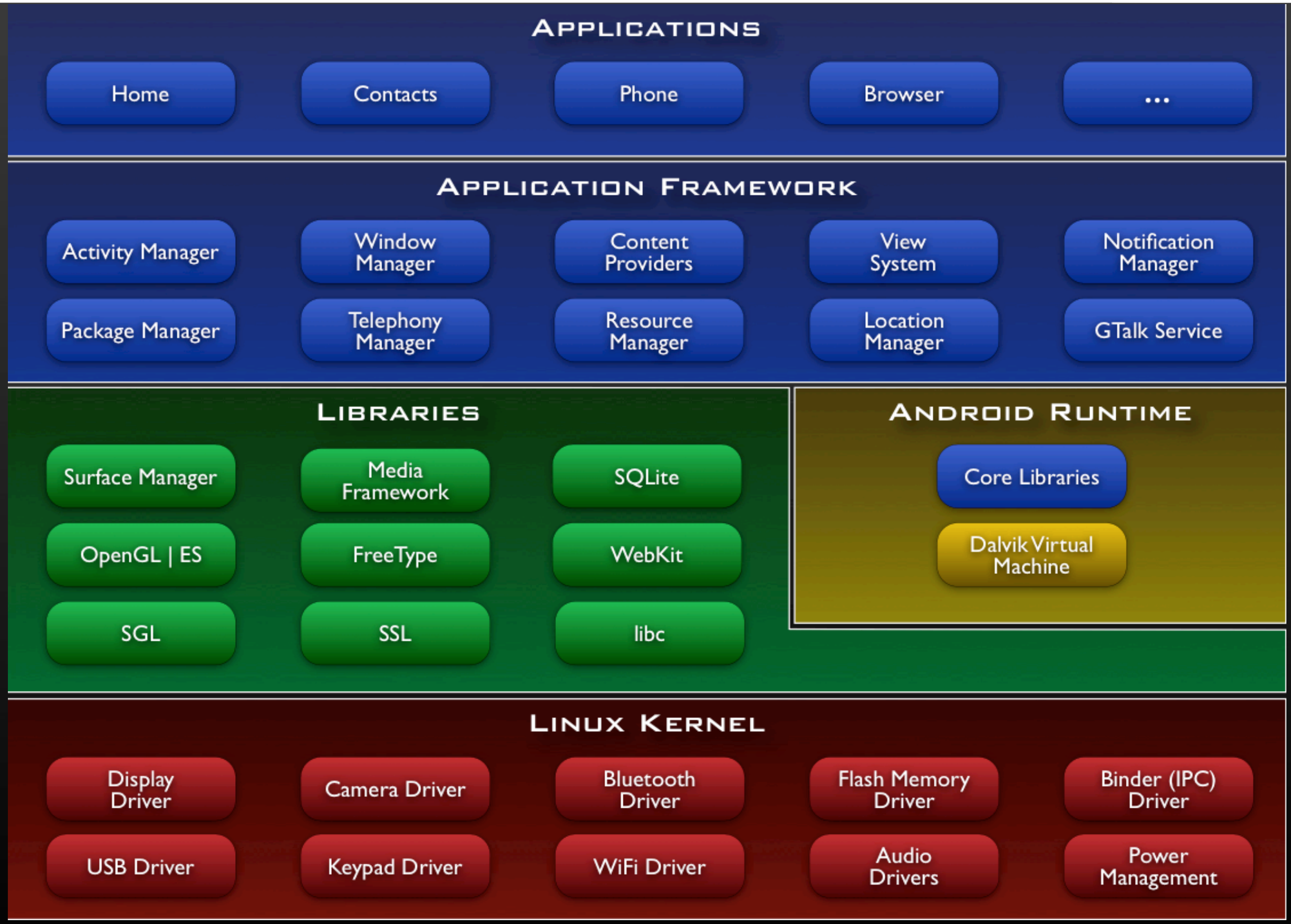
Dan Bornstein
Google

- Intro
- Memory
- CPU
- Advice
- Conclusion

# Dalvík, Iceland

ANDROID

# The Big Picture




android

# The Big Picture


android

# What is the Dalvik VM?

It is a virtual machine to…

- run on a slow CPU
- with relatively little RAM
- on an OS without swap space

android

# What is the Dalvik VM?

It is a virtual machine to…

- run on a slow CPU
- with relatively little RAM
- on an OS without swap space
- while powered by a battery

# Memory Efficiency

- Intro
- Memory
- CPU
- Advice
- Conclusion

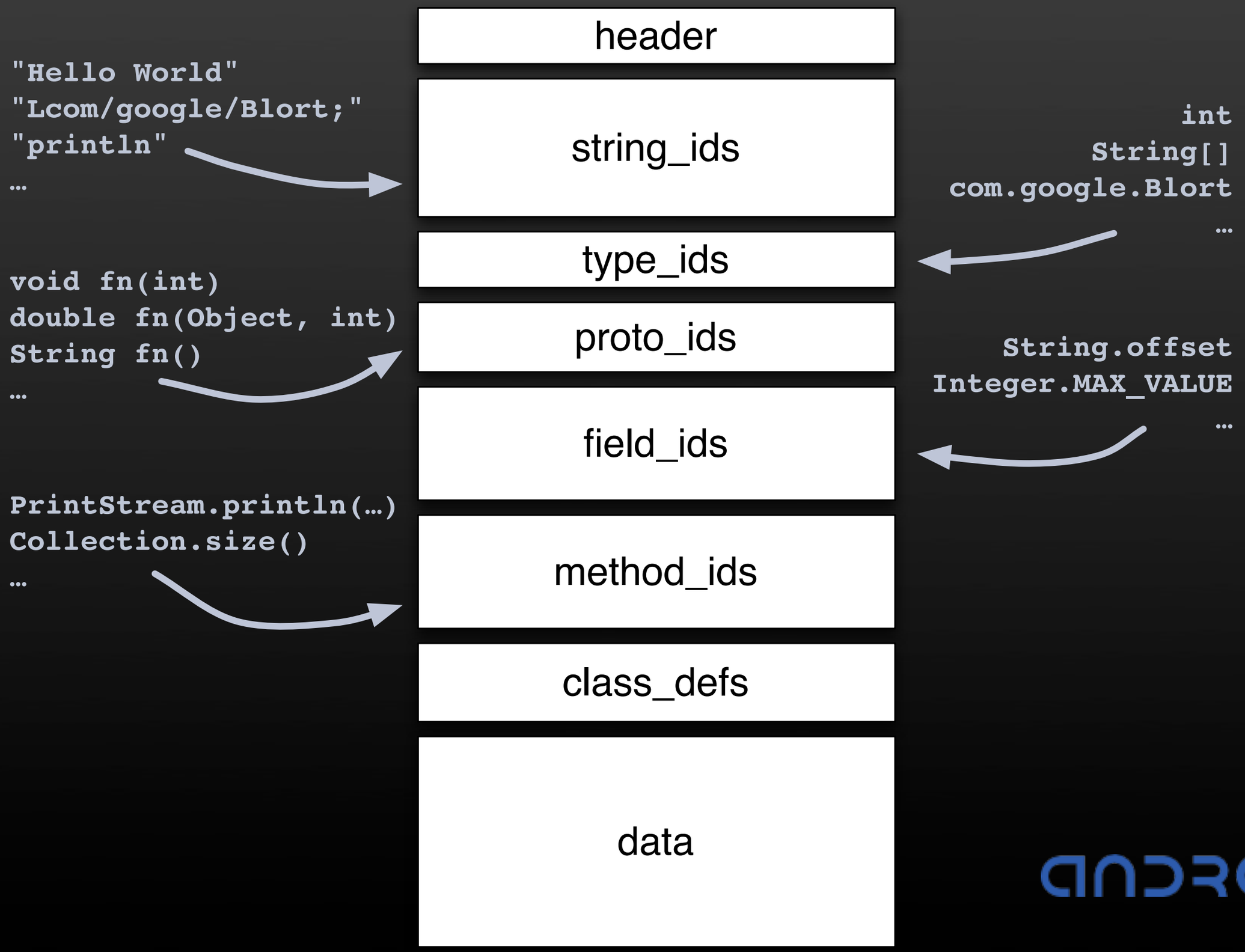
# Problem: Memory Efficiency

- total system RAM: 64 MB
  - available RAM after low-level startup: 40 MB
  - available RAM after high-level services have started: 20 MB
- multiple independent mutually-suspicious processes
  - separate address spaces, separate memory

# Problem: Memory Efficiency

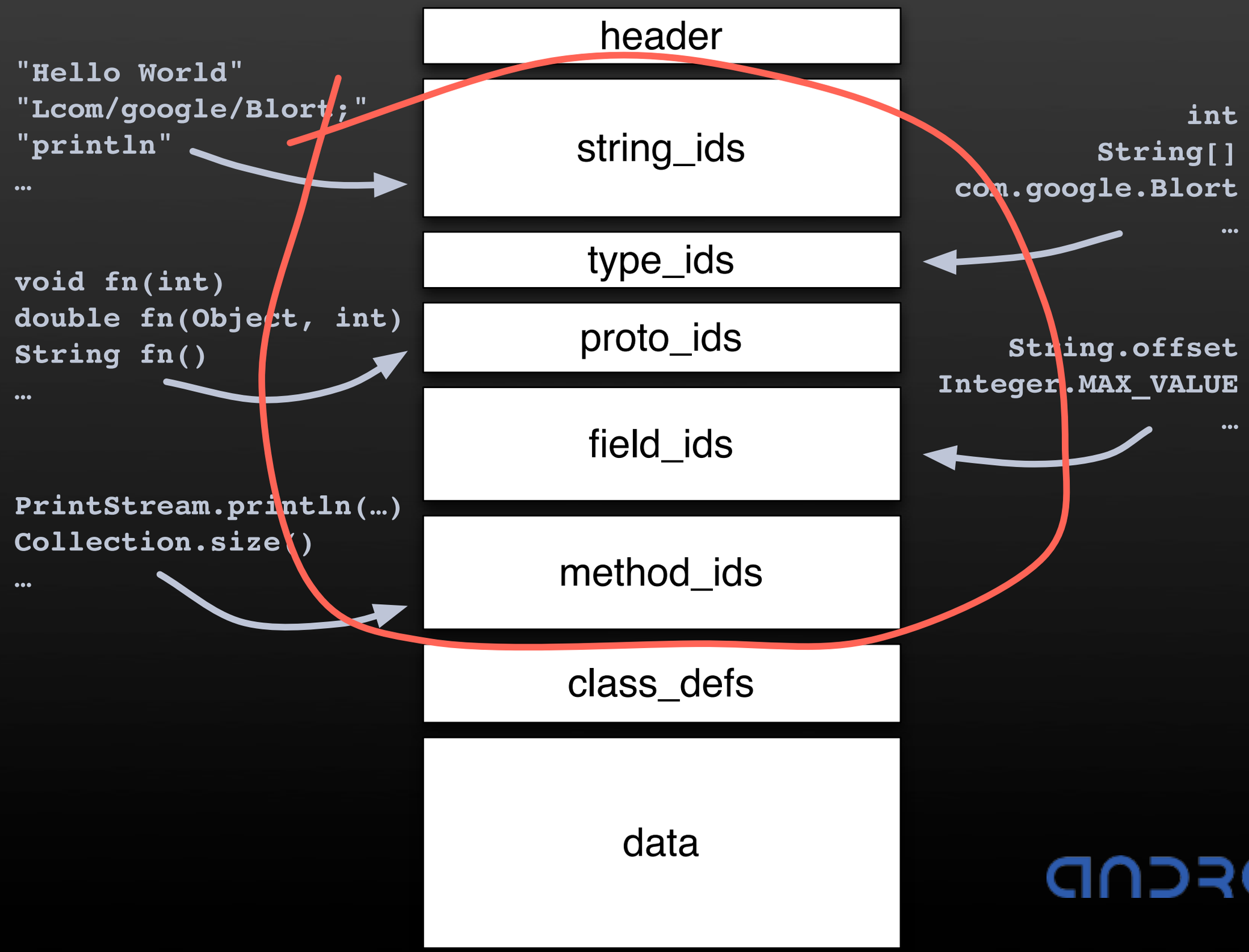
- total system RAM: 64 MB
  - available RAM after low-level startup: 40 MB
  - available RAM after high-level services have started: 20 MB
- multiple independent mutually-suspicious processes
  - separate address spaces, separate memory
- large system library: 10 MB

# Dex File Anatomy

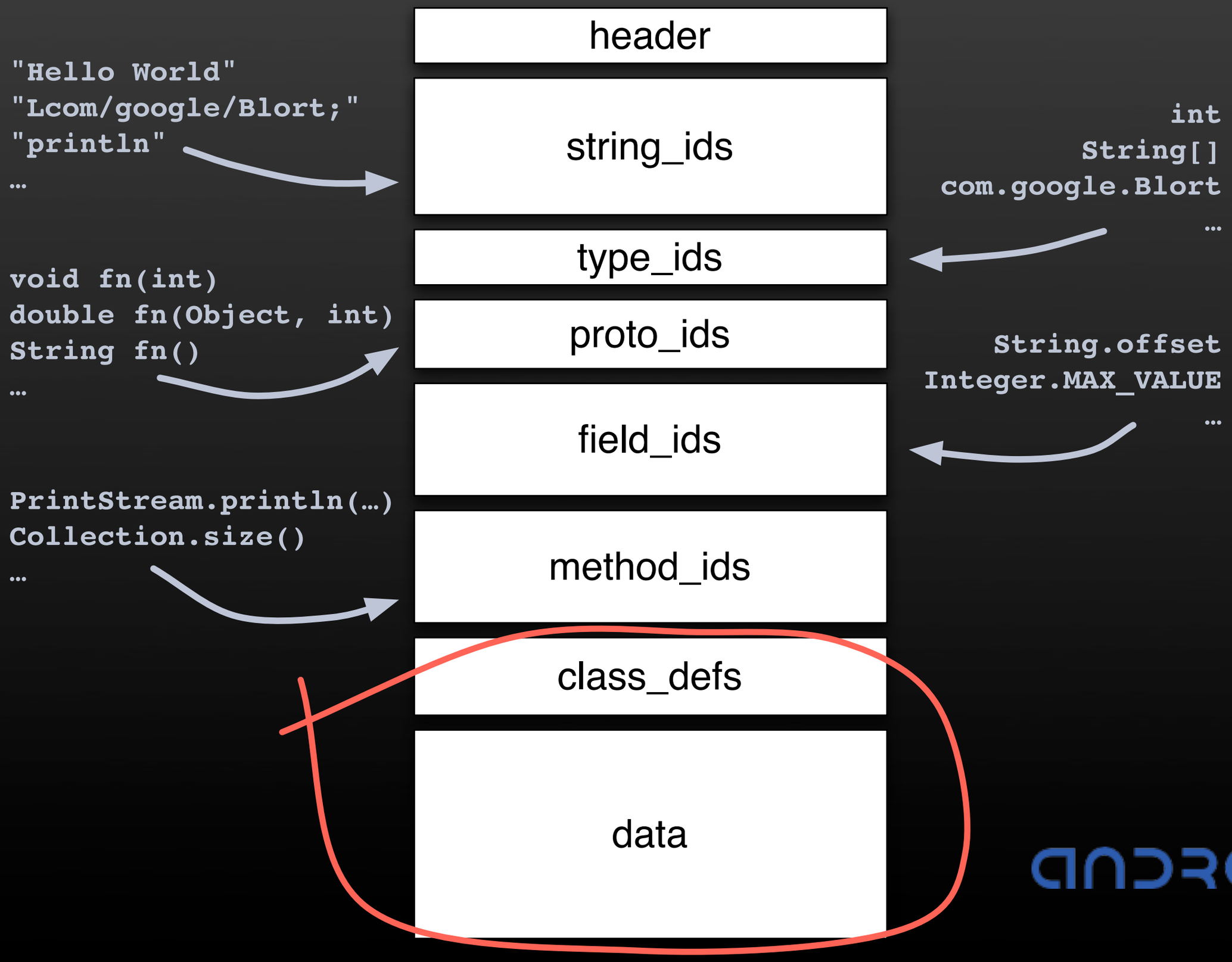



ANDROID

# Dex File Anatomy

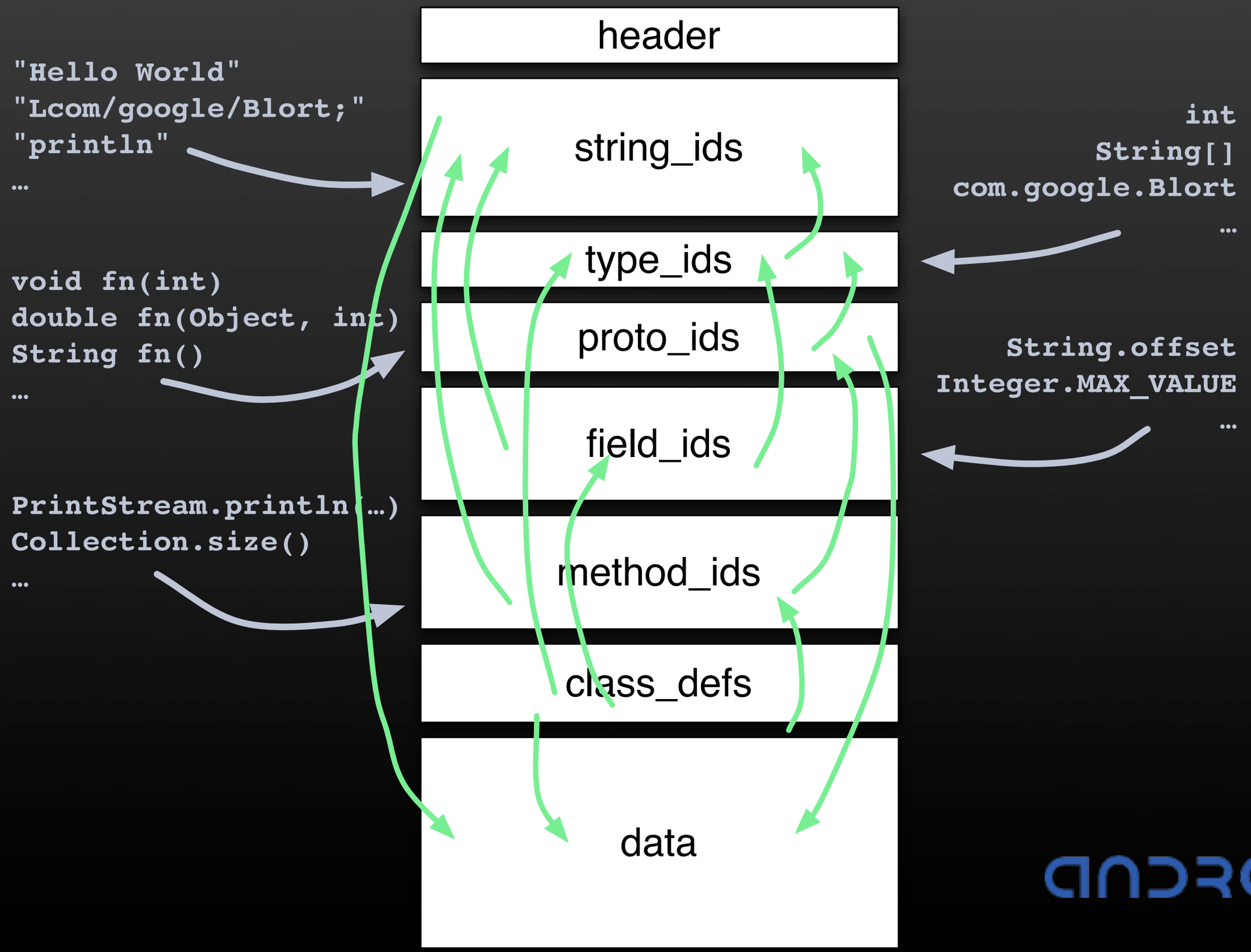


# Dex File Anatomy



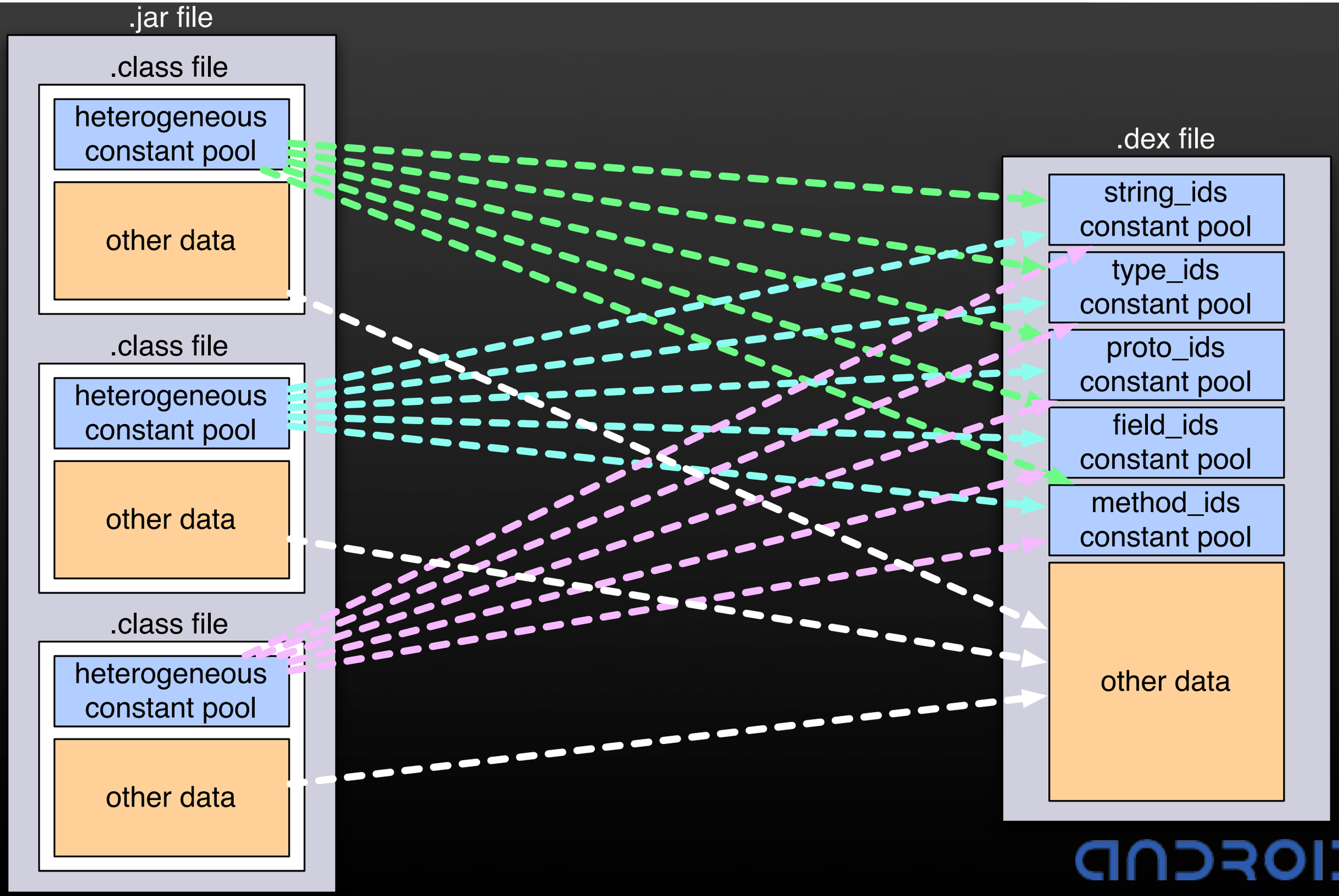

ANDROID

# Dex File Anatomy




android

# Dex File Anatomy




ANDROID

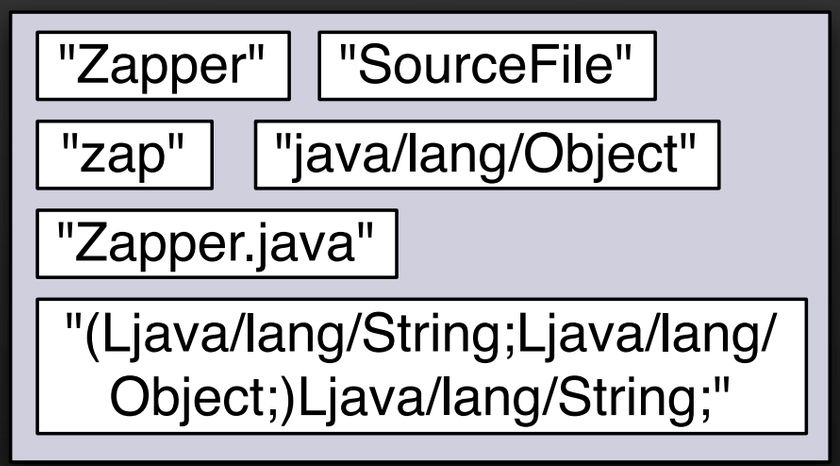
# Shared Constant Pool

```
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(...);
    }
}
```

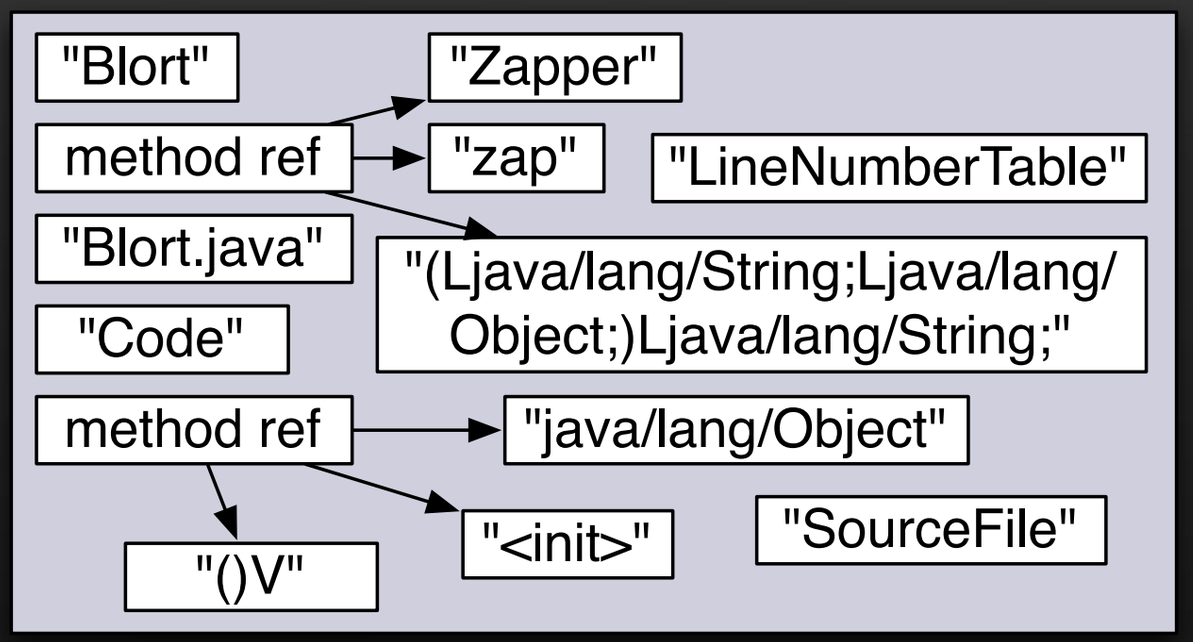
# Shared Constant Pool

## Original .class files

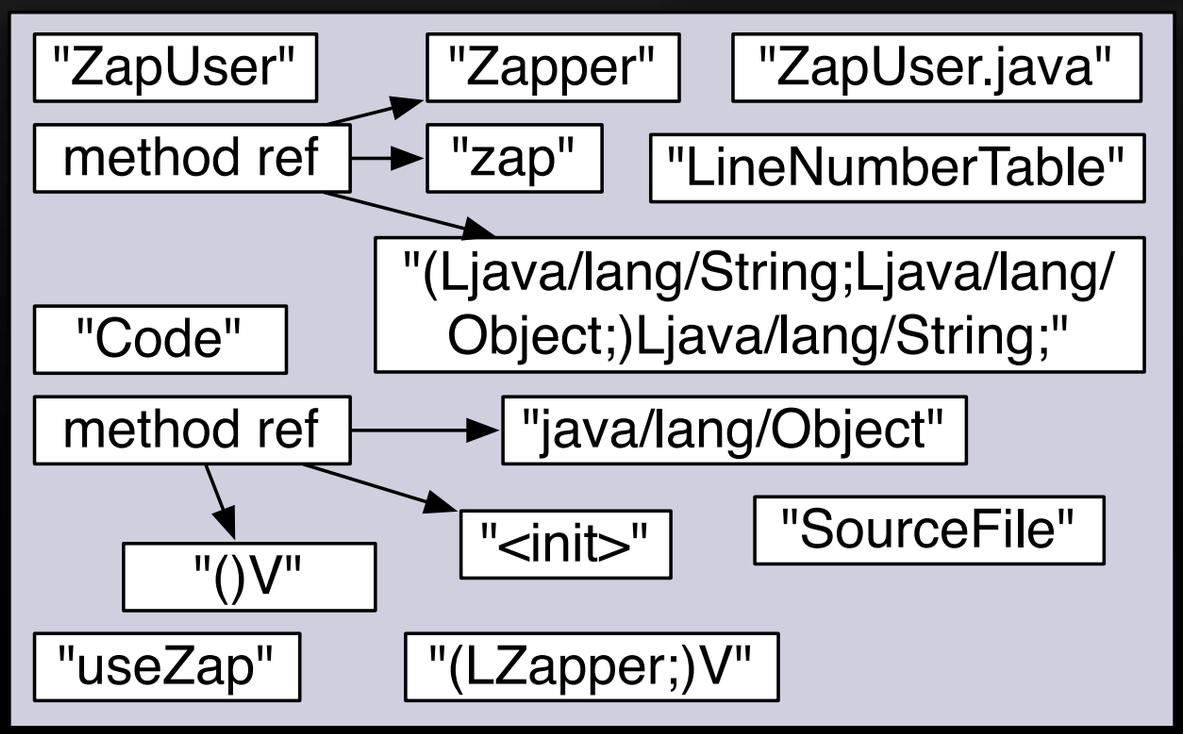
class Zapper




class Blort



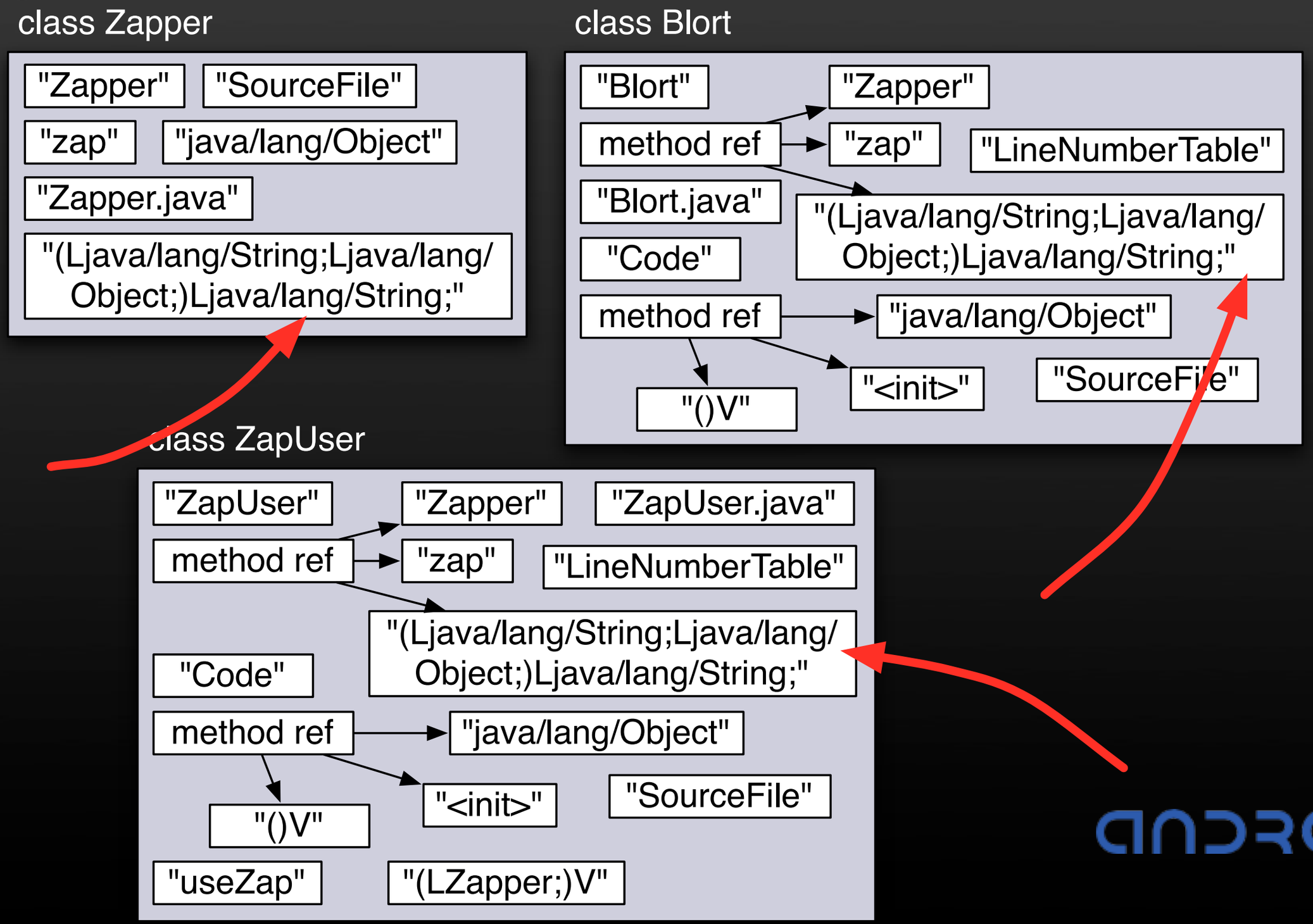

class ZapUser




android

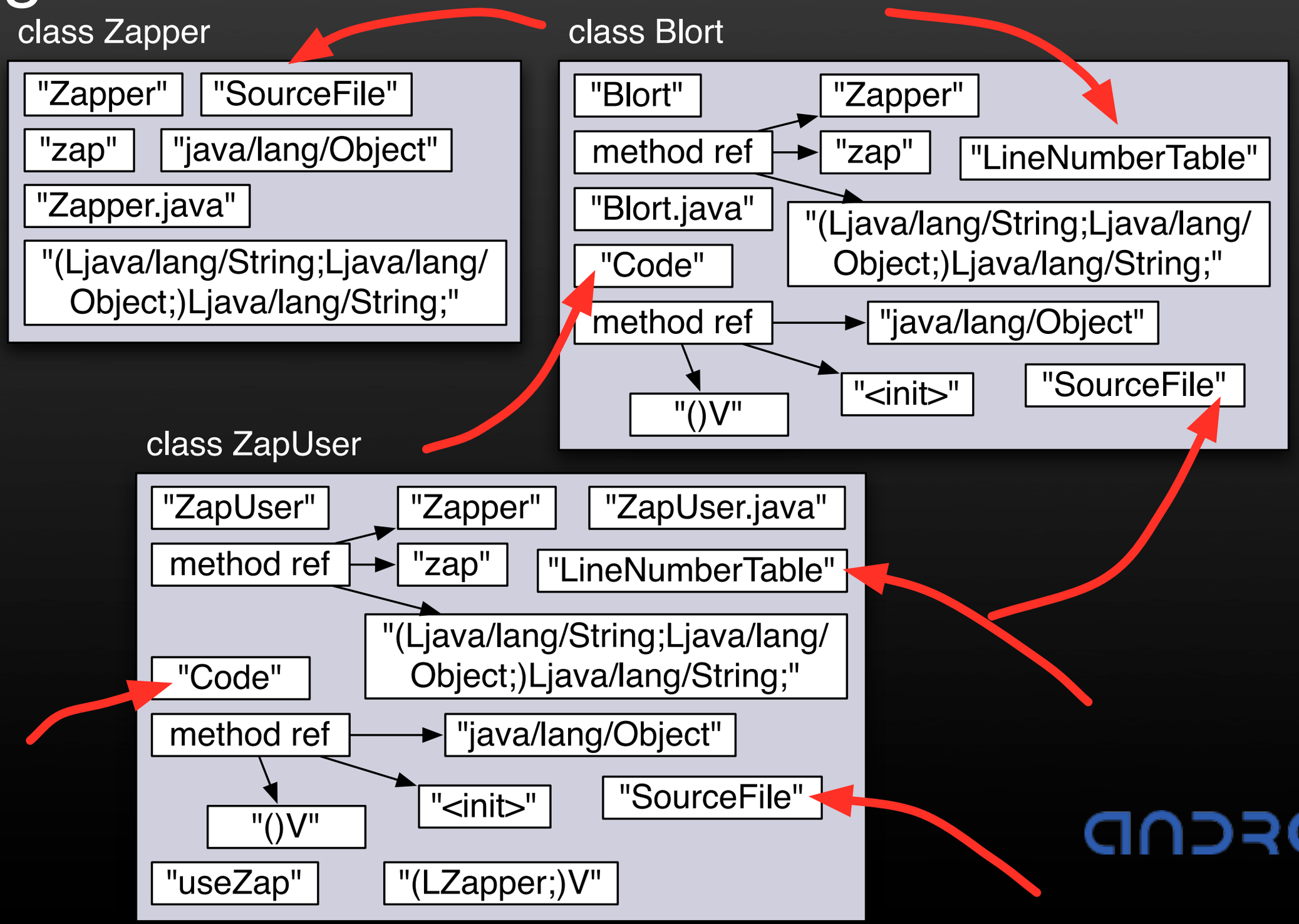
# Shared Constant Pool

## Original .class files



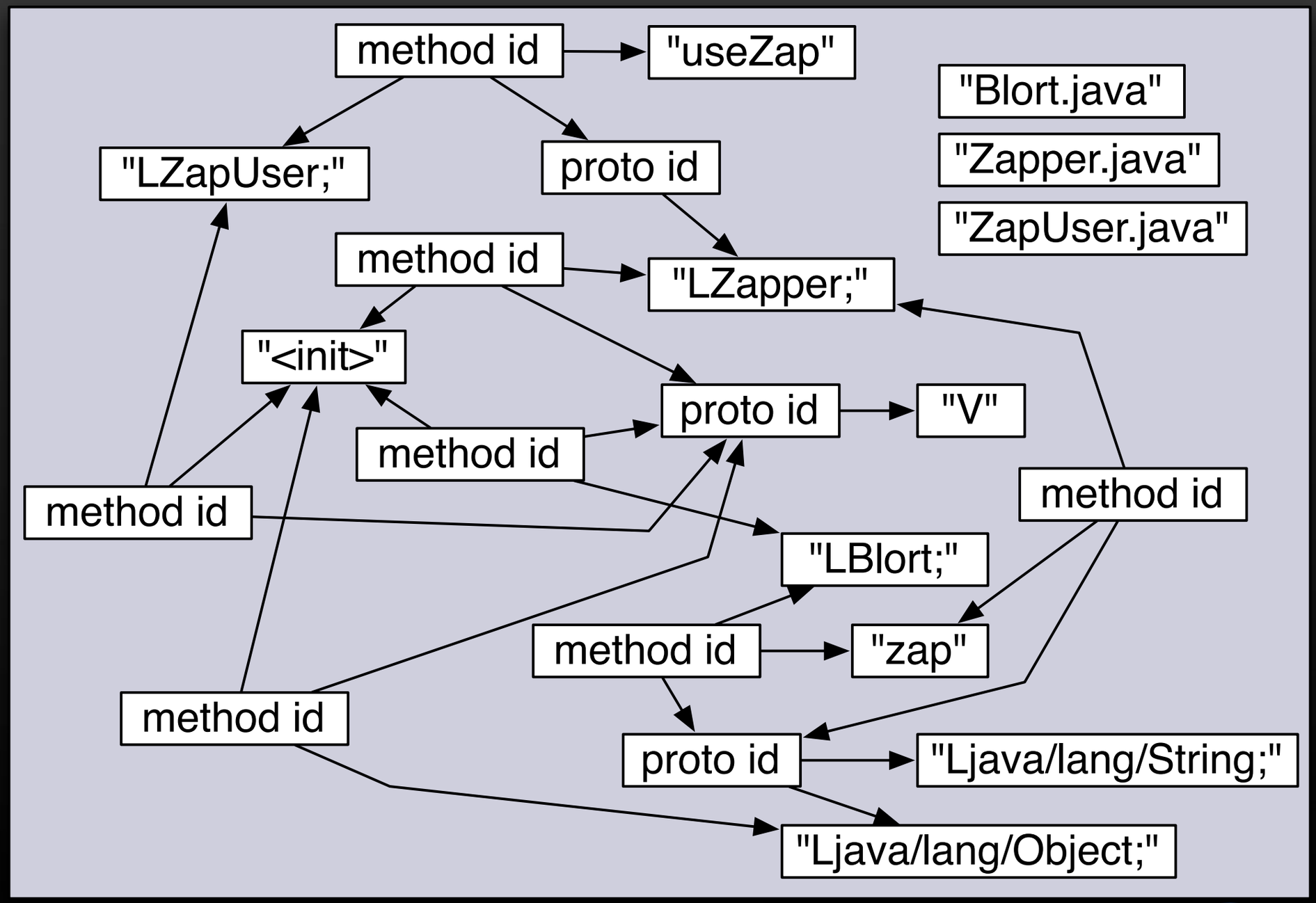
# Shared Constant Pool

## Original .class files



# Shared Constant Pool

.dex file



# Shared Constant Pool

Memory is saved via…

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

# Size Comparison

common system libraries

(U) 21445320 — 100%
(J) 10662048 — 50%
(D) 10311972 — 48%

web browser app

(U) 470312 — 100%
(J) 232065 — 49%
(D) 209248 — 44%

alarm clock app

(U) 119200 — 100%
(J) 61658 — 52%
(D) 53020 — 44%

(U) uncompressed jar file
(J) compressed jar file
(D) uncompressed dex file

ANDROID

# 4 Kinds Of Memory

- clean vs. dirty
  - clean: **`mmap()`**ed and unwritten
  - dirty: **`malloc()`**ed
- shared vs. private
  - shared: used by many processes
  - private: used by only one process

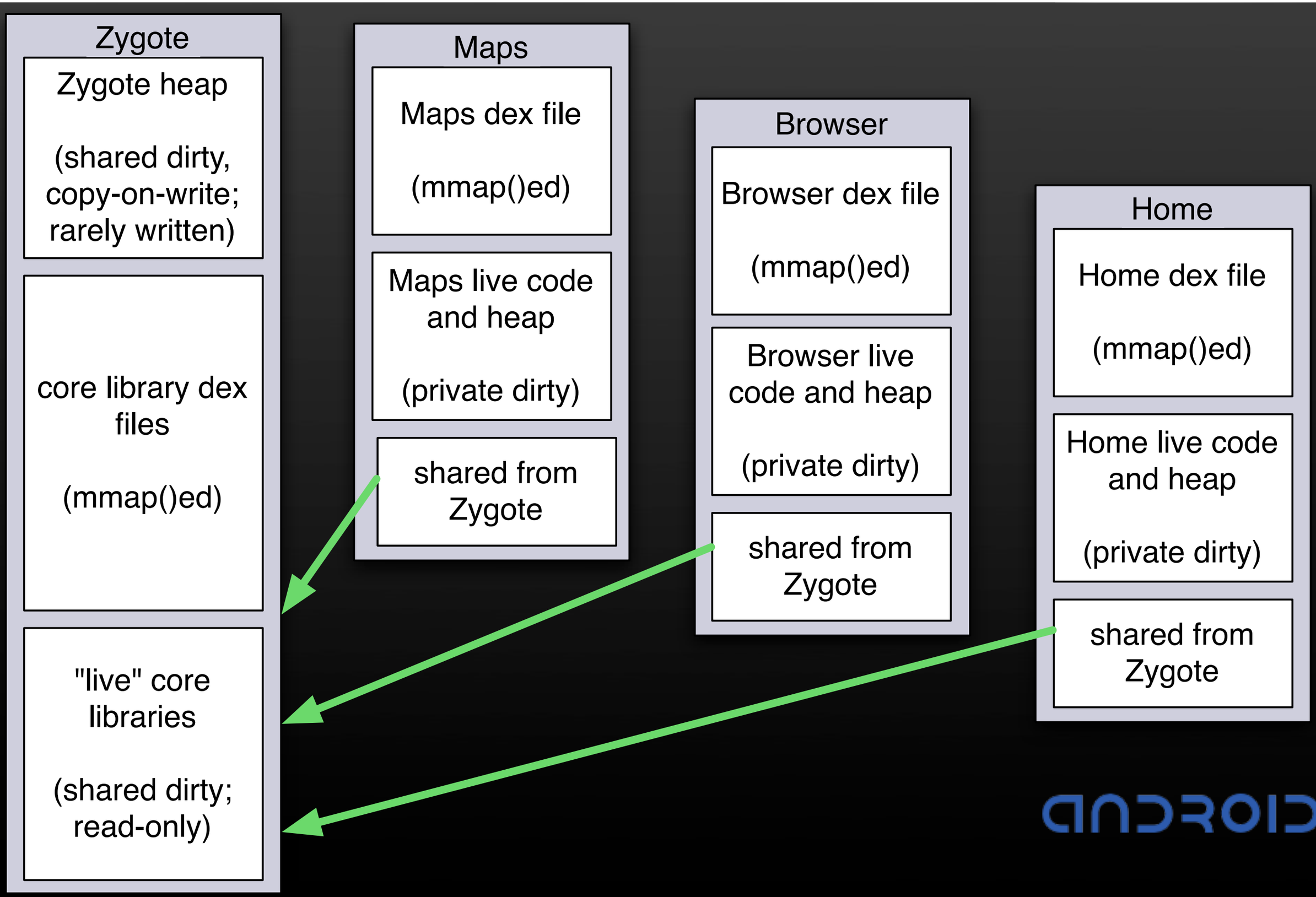
# 4 Kinds Of Memory

- clean (shared or private)
  - common dex files (libraries)
  - application-specific dex files
- shared dirty
  - *???*
- private dirty
  - application “live” dex structures
  - application heap

# Enter The Zygote

- nascent VM process
- starts at boot time
- preloads and preinitializes classes
- **`fork()`**s on command

# Enter The Zygote



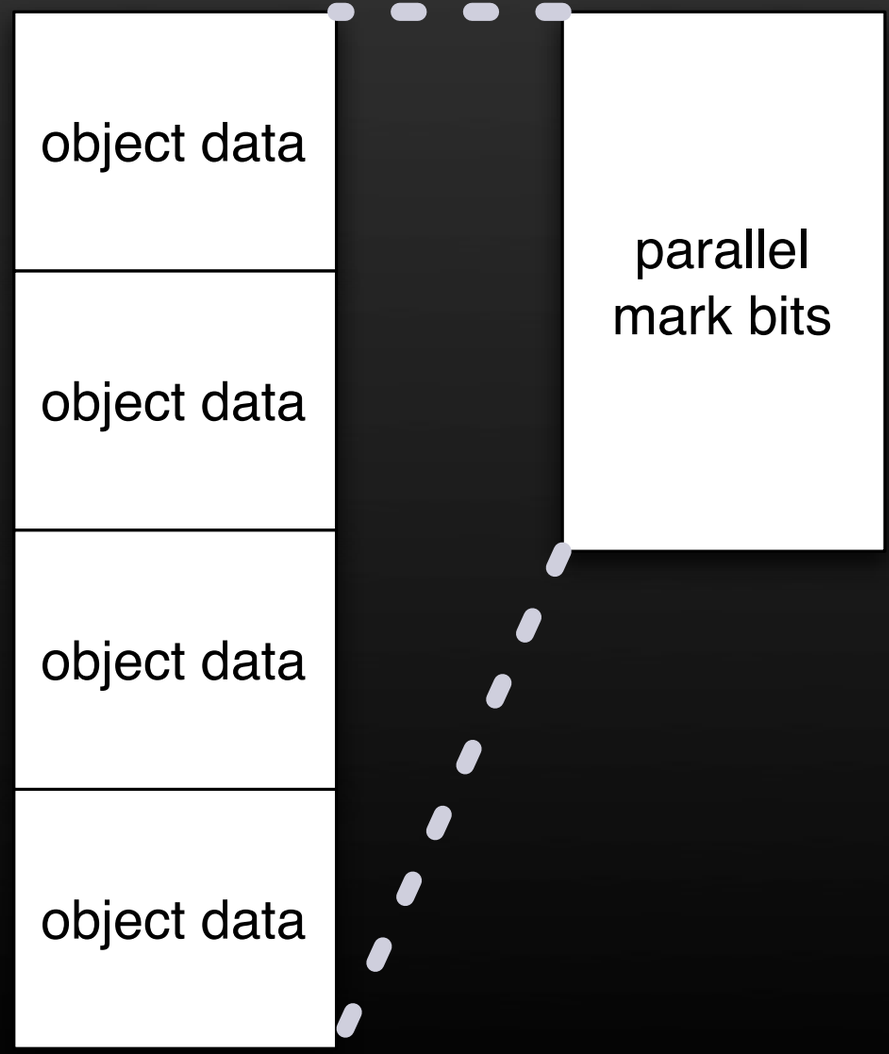
# 4 Kinds Of Memory

- clean (shared or private)
  - common dex files (libraries)
  - application-specific dex files
- shared dirty
  - library “live” dex structures
  - shared copy-on-write heap (mostly not written)
- private dirty
  - application “live” dex structures
  - application heap

# GC And Sharing

## embedded mark bits


## separated mark bits



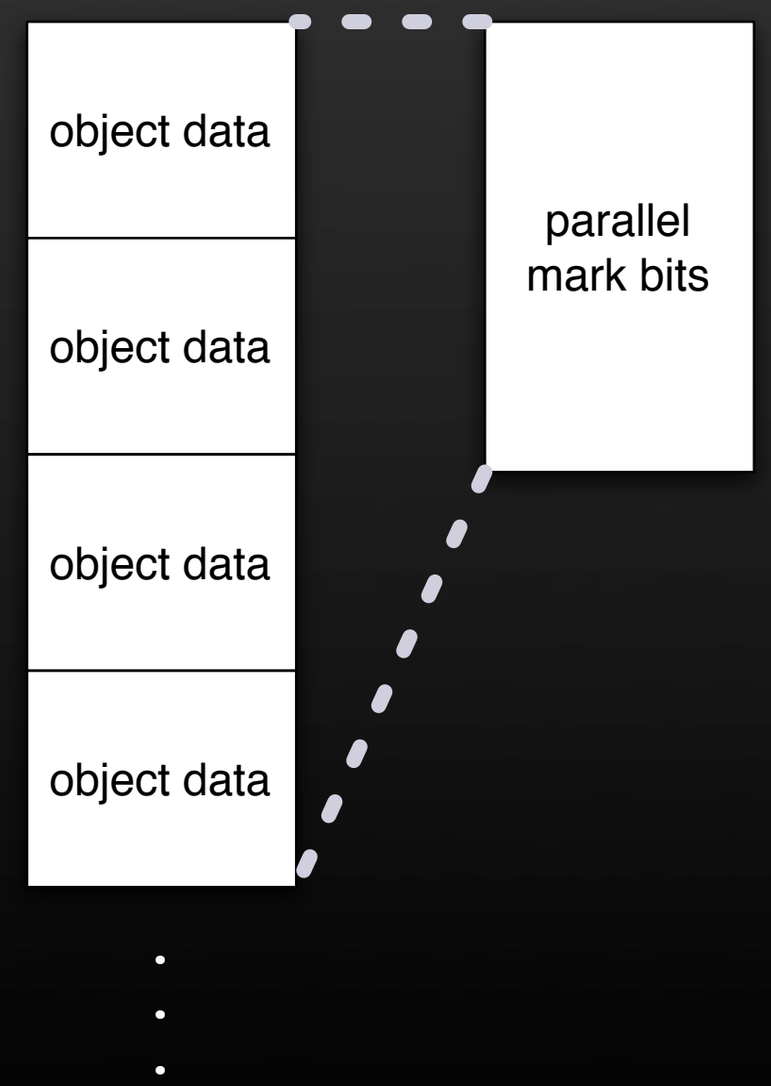
# GC And Sharing

- separate process, separate heaps, separate GCs
- GCs must be independent
- GC should respect the sharing!

# GC And Sharing

Mark bits kept separate from other heap memory.

- avoids un-sharing pages
- better small cache behavior
- doesn’t waste memory




ANDROID

# CPU Efficiency

- Intro
- Memory
- CPU
- Advice
- Conclusion

# Problem: CPU Efficiency

- CPU speed: 250-500MHz
- bus speed: 100MHz
- data cache: 16-32K
- available RAM for apps: 20 MB

# No JIT

- usually doesn’t matter
- lots of native code
  - system provides libs for graphics, media
  - JNI available
- hardware support common (graphics, audio)

# Install-Time Work

- verification
  - dex structures aren’t “lying”
    - valid indices
    - valid offsets
- code can’t misbehave

# Install-Time Work

- optimization
  - byte-swapping and padding (unnecessary on ARM)
  - static linking
  - “inlining” special native methods
  - pruning empty methods
  - adding auxiliary data

# Register Machine

## Why?

- avoid instruction dispatch
- avoid unnecessary memory access
- consume instruction stream efficiently
  - higher semantic density per instruction

# Register Machine

## The stats

- 30% fewer instructions
- 35% fewer code units
- 35% *more* bytes in the instruction stream
  - but we get to consume two at a time

# Example #1: Source

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

# Example #1: .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
```

read stack

write stack

read local

write local

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

# Example #1: .dex

```
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
```

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

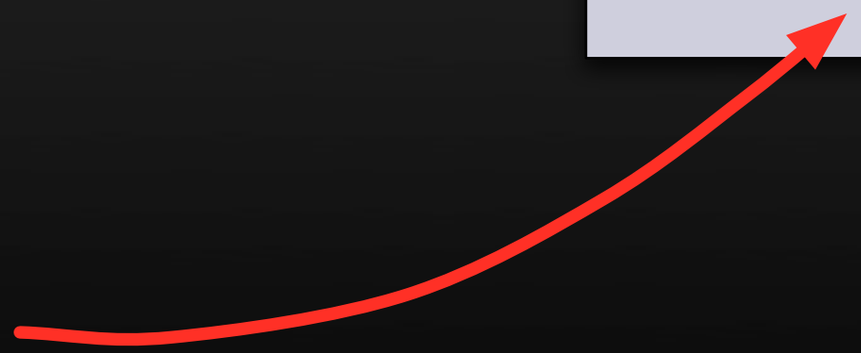
# Example #2: Source

```
private static final int[] S33KR1T_1NFORM4T10N = {
    0x4920616d, 0x20726174,
    0x68657220, 0x666f6e64,
    0x206f6620, 0x6d756666,
    0x696e732e
};
```

# Example #2: .class

```
0000: bipush #+07
0002: newarray int
0004: dup
0005: iconst_0
0006: ldc #+4920616d
0008: iastore
0009: dup
000a: iconst_1
000b: ldc #+20726174
000d: iastore
000e: dup
000f: iconst_2
0010: ldc #+68657220
0012: iastore
0013: dup
0014: iconst_3
0015: ldc #+666f6e64
0017: iastore
0018: dup
0019: iconst_4
001a: ldc #+206f6620
001c: iastore
001d: dup
001e: iconst_5
001f: ldc #+6d756666
0021: iastore
0022: dup
0023: bipush #+06
0025: ldc #+696e732e
0027: iastore
0028: putstatic Example2.S33KR1T_1NF0RM4T10N:[I
002b: return
```

```
...
dup
bipush #+NN
ldc #VVVVVVVV
iastore
...
```



# Example #2: Hack!

```
private static final int[] S33KR1T_1NF0RM4T10N;

static {
    String s =
        "\u4920\u616d\u2072\u6174\u6865" +
        "\u7270\u666f\u6e64\u206f\u6620" +
        "\u6d75\u6666\u696e\u732e";
    S33KR1T_1NF0RM4T10N = new int[7];

    for (int i = 0, j = 0; i < 7; i++, j += 2) {
        S33KR1T_1NF0RM4T10N[i] =
            (s.charAt(j) << 16) | s.charAt(j+1);
    }
}
```

# Example #2: .dex

```
0000: const/4 v0, #int 7 // #7
0001: new-array v0, v0, int[]
0003: fill-array-data v0, 000a
0006: sput-object v0,
        Example2.S33KR1T_1NF0RM4T10N:int[]
0008: return-void
0009: nop // spacer
000a: array-data // for fill-array-data @ 0003
        0: 1226858861 // #4920616d
        1: 544366964  // #20726174
        2: 1751478816 // #68657220
        3: 1718578788 // #666f6e64
        4: 544171552  // #206f6620
        5: 1836410470 // #6d756666
        6: 1768846126 // #696e732e
0026:
```

# Example #2: .dex

```
0000: const/4 v0, #int 7 // #7
0001: new-array v0, v0, int[]
0003: fill-array-data v0, 000a
0006: sput-object v0,
        Example2.S33KR1T_1NF0RM4T10N:int[]
0008: return-void
0009: nop // spacer
000a: array-data // for fill-array-data @ 0003
        0:  1315272293 // #4e657665
        1:  1914726255 // #7220676f
        2:  1852727584 // #6e6e6120
        3:  1734964837 // #67697665
        4:  544829301  // #20796f75
        5:  544567355  // #2075703b
        6:  544105846  // #206e6576
        7:  1701978215 // #65722067
        8:  1869508193 // #6f6e6e61
        9:  543974772  // #206c6574
        10: 544829301  // #20796f75
        11: 543453047  // #20646f77
        12: 1848520238 // #6e2e2e2e
003e:
```

# Interpreters 101

## The portable way

```
static void interp(const char* s) {
    for (;;) {
        switch (*(s++)) {
            case 'a': printf("Hell");    break;
            case 'b': printf("o");       break;
            case 'c': printf(" w");       break;
            case 'd': printf("rld!\n"); break;
            case 'e': return;
        }
    }
}

int main(int argc, char** argv) {
    interp("abcbde");
}
```

# Interpreters 101

## The gcc 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("Hell"); DISPATCH();
    op_b: printf("o"); DISPATCH();
    op_c: printf(" w"); DISPATCH();
    op_d: printf("rld!\n"); DISPATCH();
    op_e: return;
}
```

# Interpreters 101

## ARM assembly

Two memory reads

```
op_table:
     .word op_a
     .word op_b
     ...

#define DISPATCH() ldrb r0, [rPC], #1 \
                   ldr  pc, [rOP_TABLE, r0, lsl #2]
op_a: ...
      DISPATCH()
op_b: ...
      DISPATCH()
...
```

# Interpreters 101

## ARM assembly (cleverer)

One memory read

```
#define DISPATCH() ldrb r0, [rPC], #1 \
                   add  pc, rFIRST_OP, r0, lsl #6
.align 64
op_a: // address gets stored in rFIRST_OP
... up to 16 instructions ...
op_b:
... up to 16 instructions ...
op_c:
... up to 16 instructions ...
...
```

# Optimizing Your Code

- Intro
- Memory
- CPU
- Advice
- Conclusion

# Time Scale

- human interaction scale
  - 10-30 interactions / sec
- human perception scale
  - 25-30 image frames / sec
  - continuous audio, synched within 100 msec
- computer scale
  - run as much and as fast as possible

# Get Plenty Of Rest

A well-behaved app…

- spends most of its time sleeping
- reacts quickly and decisively to user and network input

# Loop Wisely

```
(1) for (int i = initializer; i >= 0; i--)

(2) int limit = calculate limit;
    for (int i = 0; i < limit; i++)

(3) Type[] array = get array;
    for (Type obj : array)

(4) for (int i = 0; i < array.length; i++)

(5) for (int i = 0; i < this.var; i++)

(6) for (int i = 0; i < obj.size(); i++)

(7) Iterable<Type> list = get list;
    for (Type obj : list)
```

ANDROID

# Loop Wisely

```
(1) for (int i = initializer; i >= 0; i--)

(2) int limit = calculate limit;
    for (int i = 0; i < limit; i++)

(3) Type[] array = get array;
    for (Type obj : array)

(4) for (int i = 0; i < array.length; i++)

(5) for (int i = 0; i < this.var; i++)

(6) for (int i = 0; i < obj.size(); i++)

(7) Iterable<Type> list = get list;
    for (Type obj : list)
```

# Loop Wisely

```
(1) for (int i = initializer; i >= 0; i--)

(2) int limit = calculate limit;
    for (int i = 0; i < limit; i++)

(3) Type[] array = get array;
    for (Type obj : array)

(4) for (int i = 0; i < array.length; i++)

(5) for (int i = 0; i < this.var; i++)

(6) for (int i = 0; i < obj.size(); i++)

(7) Iterable<Type> list = get list;
    for (Type obj : list)
```

**Danger! Danger!**

**Danger! Danger!**

# Avoid Allocation

- short-lived objects need to be GCed
- long-lived objects take precious memory

# That’s all!

- Intro
- Memory
- CPU
- Advice
- Conclusion

# Questions?