Dalvik Executable format

This document describes the layout and contents of `.dex` files, which are used to hold a set of class definitions and their associated adjunct data.

Guide to types

Name	Description
byte	8-bit signed int
ubyte	8-bit unsigned int
short	16-bit signed int, little-endian
ushort	16-bit unsigned int, little-endian
int	32-bit signed int, little-endian
uint	32-bit unsigned int, little-endian
long	64-bit signed int, little-endian
ulong	64-bit unsigned int, little-endian
sleb128	signed LEB128, variable-length (see below)
uleb128	unsigned LEB128, variable-length (see below)
uleb128p1	unsigned LEB128 plus 1, variable-length (see below)

LEB128

LEB128 ("**L**ittle-**E**ndian **B**ase **128**") is a variable-length encoding for arbitrary signed or unsigned integer quantities. The format was borrowed from the DWARF3 (http://dwarfstd.org/Dwarf3Std.php) specification. In a `.dex` file, LEB128 is only ever used to encode 32-bit quantities.

Each LEB128 encoded value consists of one to five bytes, which together represent a single 32-bit value. Each byte has its most significant bit set except for the final byte in the sequence, which has its most significant bit clear. The remaining seven bits of each byte are payload, with the least significant seven bits of the quantity in the first byte, the next seven in the second byte and so on. In the case of a signed LEB128 (`sleb128`), the most significant payload bit of the final byte in the sequence is sign-extended to produce the final value. In the unsigned case (`uleb128`), any bits not explicitly represented are interpreted as `0`.

Bitwise diagram of a two-byte LEB128 value															
First byte								**Second byte**							
`1`	bit$_{6}$	bit$_{5}$	bit$_{4}$	bit$_{3}$	bit$_{2}$	bit$_{1}$	bit$_{0}$	`0`	bit$_{13}$	bit$_{12}$	bit$_{11}$	bit$_{10}$	bit$_{9}$	bit$_{8}$	bit$_{7}$

The variant `uleb128p1` is used to represent a signed value, where the representation is of the value *plus one* encoded as a `uleb128`. This makes the encoding of `-1` (alternatively thought of as the unsigned value `0xffffffff`) — but no other negative number — a single byte, and is useful in exactly those cases where the represented number must either be non-negative or `-1` (or `0xffffffff`), and where no other negative values are allowed (or where large unsigned values are unlikely to be needed).

Here are some examples of the formats:

Encoded Sequence	As sleb128	As uleb128	As uleb128p1
00	0	0	-1
01	1	1	0
7f	-1	127	126
80 7f	-128	16256	16255

File layout

Name	Format	Description
header	header_item	the header
string_ids	string_id_item[]	string identifiers list. These are identifiers for all the strings used by this file, either for internal naming (e.g., type descriptors) or as constant objects referred to by code. This list must be sorted by string contents, using UTF-16 code point values (not in a locale-sensitive manner), and it must not contain any duplicate entries.
type_ids	type_id_item[]	type identifiers list. These are identifiers for all types (classes, arrays, or primitive types) referred to by this file, whether defined in the file or not. This list must be sorted by **`string_id`** index, and it must not contain any duplicate entries.
proto_ids	proto_id_item[]	method prototype identifiers list. These are identifiers for all prototypes referred to by this file. This list must be sorted in return-type (by **`type_id`** index) major order, and then by argument list (lexicographic ordering, individual arguments ordered by **`type_id`** index). The list must not contain any duplicate entries.
field_ids	field_id_item[]	field identifiers list. These are identifiers for all fields referred to by this file, whether defined in the file or not. This list must be sorted, where the defining type (by **`type_id`** index) is the major order, field name (by **`string_id`** index) is the intermediate order, and type (by **`type_id`** index) is the minor order. The list must not contain any duplicate entries.
method_ids	method_id_item[]	method identifiers list. These are identifiers for all methods referred to by this file, whether defined in

		the file or not. This list must be sorted, where the defining type (by **`type_id`** index) is the major order, method name (by **`string_id`** index) is the intermediate order, and method prototype (by **`proto_id`** index) is the minor order. The list must not contain any duplicate entries.
class_defs	class_def_item[]	class definitions list. The classes must be ordered such that a given class's superclass and implemented interfaces appear in the list earlier than the referring class. Furthermore, it is invalid for a definition for the same-named class to appear more than once in the list.
call_site_ids	call_site_id_item[]	call site identifiers list. These are identifiers for all call sites referred to by this file, whether defined in the file or not. This list must be sorted in ascending order of **`call_site_off`**.
method_handles	method_handle_item[]	method handles list. A list of all method handles referred to by this file, whether defined in the file or not. This list is not sorted and may contain duplicates which will logically correspond to different method handle instances.
data	ubyte[]	data area, containing all the support data for the tables listed above. Different items have different alignment requirements, and padding bytes are inserted before each item if necessary to achieve proper alignment.
link_data	ubyte[]	data used in statically linked files. The format of the data in this section is left unspecified by this document. This section is empty in unlinked files, and runtime implementations may use it as they see fit.

Bitfield, string and constant definitions

DEX_FILE_MAGIC

embedded in header_item

The constant array/string `DEX_FILE_MAGIC` is the list of bytes that must appear at the beginning of a `.dex` file in order for it to be recognized as such. The value intentionally contains a newline (`"\n"` or `0x0a`) and a null byte (`"\0"` or `0x00`) in order to help in the detection of certain forms of corruption. The value also encodes a format version number as three decimal digits, which is expected to increase monotonically over time as the format evolves.

```
ubyte[8] DEX_FILE_MAGIC = { 0x64 0x65 0x78 0x0a 0x30 0x33 0x39 0x00
                        = "dex\n039\0"
```

Note: Support for version **`039`** of the format was added in the Android 9.0 release, which introduced two new bytecodes, **`const-method-handle`** and **`const-method-type`**. (These are each described in the Summary of bytecode set (https://source.android.com/devices/tech/dalvik/dalvik-bytecode#instructions) table.) In Android 10, version **`039`** extends the DEX file format to include hidden API information that's only applicable to DEX files on the boot class path.

Note: Support for version **`038`** of the format was added in the Android 8.0 release. Version **`038`** added new bytecodes (**`invoke-polymorphic`** and **`invoke-custom`**) and data for method handles.

Note: Support for version **`037`** of the format was added in the Android 7.0 release. Prior to version **`037`** most versions of Android have used version **`035`** of the format. The only difference between versions **`035`** and **`037`** is the addition of default methods and the adjustment of the **`invoke`**.

Note: At least a couple earlier versions of the format have been used in widely available public software releases. For example, version **009** was used for the M3 releases of the Android platform (November–December 2007), and version **013** was used for the M5 releases of the Android platform (February–March 2008). In several respects, these earlier versions of the format differ significantly from the version described in this document.

ENDIAN_CONSTANT and REVERSE_ENDIAN_CONSTANT

embedded in header_item

The constant `ENDIAN_CONSTANT` is used to indicate the endianness of the file in which it is found. Although the standard `.dex` format is little-endian, implementations may choose to perform byte-swapping. Should an implementation come across a header whose `endian_tag` is `REVERSE_ENDIAN_CONSTANT` instead of `ENDIAN_CONSTANT`, it would know that the file has been byte-swapped from the expected form.

```
uint ENDIAN_CONSTANT = 0x12345678;
uint REVERSE_ENDIAN_CONSTANT = 0x78563412;
```

NO_INDEX

embedded in class_def_item and debug_info_item

The constant `NO_INDEX` is used to indicate that an index value is absent.

Note: This value isn't defined to be **0**, because that is in fact typically a valid index.

The chosen value for `NO_INDEX` is representable as a single byte in the `uleb128p1` encoding.

```
uint NO_INDEX = 0xffffffff;    // == -1 if treated as a signed int
```

access_flags definitions

embedded in class_def_item, encoded_field, encoded_method, and InnerClass

Bitfields of these flags are used to indicate the accessibility and overall properties of classes and class members.

Name	Value	For Classes (and InnerClass annotations)	For Fields	For Methods
ACC_PUBLIC	0x1	`public`: visible everywhere	`public`: visible everywhere	`public`: visible everywhere
ACC_PRIVATE	0x2	* `private`: only visible to defining class	`private`: only visible to defining class	`private`: only visible to defining class
ACC_PROTECTED	0x4	* `protected`: visible to package and subclasses	`protected`: visible to package and subclasses	`protected`: visible to package and subclasses
ACC_STATIC	0x8	* `static`: is not constructed with an outer `this` reference	`static`: global to defining class	`static`: does not take a `this` argument
ACC_FINAL	0x10	`final`: not subclassable	`final`: immutable after construction	`final`: not overridable
ACC_SYNCHRONIZED	0x20			`synchronized`: associated lock automatically acquired

				around call to this method. ★ **Note:** This is only valid to set when **`ACC_NATIVE`** is also set.
ACC_VOLATILE	0x40		**`volatile`**: special access rules to help with thread safety	
ACC_BRIDGE	0x40			bridge method, added automatically by compiler as a type-safe bridge
ACC_TRANSIENT	0x80		**`transient`**: not to be saved by default serialization	
ACC_VARARGS	0x80			last argument should be treated as a "rest" argument by compiler
ACC_NATIVE	0x100			**`native`**: implemented in native code
ACC_INTERFACE	0x200	**`interface`**: multiply-implementable abstract class		
ACC_ABSTRACT	0x400	**`abstract`**: not directly instantiable		**`abstract`**: unimplemented by this class
ACC_STRICT	0x800			**`strictfp`**: strict rules for floating-point arithmetic

ACC_SYNTHETIC	0x1000	not directly defined in source code	not directly defined in source code	not directly defined in source code
ACC_ANNOTATION	0x2000	declared as an annotation class		
ACC_ENUM	0x4000	declared as an enumerated type	declared as an enumerated value	
(unused)	0x8000			
ACC_CONSTRUCTOR	0x10000			constructor method (class or instance initializer)
ACC_DECLARED_SYNCHRONIZED	0x20000			declared **`synchronized`**. ★ **Note:** This has no effect on execution (other than in reflection of this flag, per se).

* Only allowed on for `InnerClass` annotations, and must not ever be on in a `class_def_item`.

MUTF-8 (Modified UTF-8) Encoding

As a concession to easier legacy support, the `.dex` format encodes its string data in a de facto standard modified UTF-8 form, hereafter referred to as MUTF-8. This form is identical to standard UTF-8, except:

- Only the one-, two-, and three-byte encodings are used.
- Code points in the range `U+10000` … `U+10ffff` are encoded as a surrogate

pair, each of which is represented as a three-byte encoded value.

- The code point `U+0000` is encoded in two-byte form.
- A plain null byte (value `0`) indicates the end of a string, as is the standard C language interpretation.

The first two items above can be summarized as: MUTF-8 is an encoding format for UTF-16, instead of being a more direct encoding format for Unicode characters.

The final two items above make it simultaneously possible to include the code point `U+0000` in a string *and* still manipulate it as a C-style null-terminated string.

However, the special encoding of `U+0000` means that, unlike normal UTF-8, the result of calling the standard C function `strcmp()` on a pair of MUTF-8 strings does not always indicate the properly signed result of comparison of *unequal* strings. When ordering (not just equality) is a concern, the most straightforward way to compare MUTF-8 strings is to decode them character by character, and compare the decoded values. (However, more clever implementations are also possible.)

Please refer to The Unicode Standard (http://unicode.org) for further information about character encoding. MUTF-8 is actually closer to the (relatively less well-known) encoding CESU-8 (http://www.unicode.org/reports/tr26/) than to UTF-8 per se.

encoded_value encoding

embedded in annotation_element and encoded_array_item

An `encoded_value` is an encoded piece of (nearly) arbitrary hierarchically structured data. The encoding is meant to be both compact and straightforward to parse.

Name	Format	Description

Name	Format	Description
(value_arg << 5) | value_type	ubyte	byte indicating the type of the immediately subsequent **`value`** along with an optional clarifying argument in the high-order three bits. See below for the various **`value`** definitions. In most cases, **`value_arg`** encodes the length of the immediately-subsequent **`value`** in bytes, as **`(size - 1)`**, e.g., **`0`** means that the value requires one byte, and **`7`** means it requires eight bytes; however, there are exceptions as noted below.
value	ubyte[]	bytes representing the value, variable in length and interpreted differently for different **`value_type`** bytes, though always little-endian. See the various value definitions below for details.

Value formats

Type Name	value_type	value_arg Format	value Format	Description
VALUE_BYTE	0x00	*(none; must be* **`0`***)*	ubyte[1]	signed one-byte inte value
VALUE_SHORT	0x02	size - 1 (0…1)	ubyte[size]	signed two-byte inte value, sign-extended
VALUE_CHAR	0x03	size - 1 (0…1)	ubyte[size]	unsigned two-byte integer value, zero-extended
VALUE_INT	0x04	size - 1 (0…3)	ubyte[size]	signed four-byte inte value, sign-extended
VALUE_LONG	0x06	size - 1 (0…7)	ubyte[size]	signed eight-byte int value, sign-extended
VALUE_FLOAT	0x10	size - 1 (0…3)	ubyte[size]	four-byte bit pattern, zero-extended *to the right*, and interpreted an IEEE754 32-bit floating point value

VALUE_DOUBLE	0x11	size - 1 (0…7)	ubyte[size]	eight-byte bit pattern zero-extended *to the right*, and interpretec an IEEE754 64-bit floating point value
VALUE_METHOD_TYPE	0x15	size - 1 (0…3)	ubyte[size]	unsigned (zero-extended) four-byte integer value, interpr as an index into the `proto_ids` section representing a meth type value
VALUE_METHOD_HANDLE	0x16	size - 1 (0…3)	ubyte[size]	unsigned (zero-extended) four-byte integer value, interpr as an index into the `method_handles` section and represer a method handle val
VALUE_STRING	0x17	size - 1 (0…3)	ubyte[size]	unsigned (zero-extended) four-byte integer value, interpr as an index into the `string_ids` sectio and representing a s value
VALUE_TYPE	0x18	size - 1 (0…3)	ubyte[size]	unsigned (zero-extended) four-byte integer value, interpr as an index into the `type_ids` section a representing a reflec type/class value
VALUE_FIELD	0x19	size - 1 (0…3)	ubyte[size]	unsigned (zero-extended) four-byte integer value, interpr

				as an index into the **`field_ids`** section representing a reflec field value
VALUE_METHOD	0x1a	size - 1 (0…3)	ubyte[size]	unsigned (zero-extended) four-byte integer value, interpr as an index into the **`method_ids`** sectio and representing a reflective method va
VALUE_ENUM	0x1b	size - 1 (0…3)	ubyte[size]	unsigned (zero-extended) four-byte integer value, interpr as an index into the **`field_ids`** section representing the valu an enumerated type constant
VALUE_ARRAY	0x1c	*(none; must be `0`)*	encoded_array	an array of values, in format specified by "**`encoded_array`** format" below. The s of the **`value`** is impl in the encoding.
VALUE_ANNOTATION	0x1d	*(none; must be `0`)*	encoded_annotation	a sub-annotation, in format specified by "**`encoded_annotat`** format" below. The s of the **`value`** is impl in the encoding.
VALUE_NULL	0x1e	*(none; must be `0`)*	*(none)*	**`null`** reference valu
VALUE_BOOLEAN	0x1f	boolean (0…1)	*(none)*	one-bit value; **`0`** for **`false`** and **`1`** for **`tru`** The bit is represente

the **`value_arg`**.

encoded_array format

Name	Format	Description
size	uleb128	number of elements in the array
values	encoded_value[size]	a series of **`size`** **`encoded_value`** byte sequences in the format specified by this section, concatenated sequentially.

encoded_annotation format

Name	Format	Description
type_idx	uleb128	type of the annotation. This must be a class (not array or primitive) type.
size	uleb128	number of name-value mappings in this annotation
elements	annotation_element[size]	elements of the annotation, represented directly in-line (not as offsets). Elements must be sorted in increasing order by **`string_id`** index.

annotation_element format

Name	Format	Description
name_idx	uleb128	element name, represented as an index into the **`string_ids`** section. The string must conform to the syntax for *MemberName*, defined above.
value	encoded_value	element value

String syntax

There are several kinds of item in a `.dex` file which ultimately refer to a string. The following BNF-style definitions indicate the acceptable syntax for these strings.

SimpleName

A *SimpleName* is the basis for the syntax of the names of other things. The `.dex` format allows a fair amount of latitude here (much more than most common source languages). In brief, a simple name consists of any low-ASCII alphabetic character or digit, a few specific low-ASCII symbols, and most non-ASCII code points that are not control, space, or special characters. Starting from version `040` the format additionally allows space characters (Unicode `Zs` category). Note that surrogate code points (in the range `U+d800` … `U+dfff`) are not considered valid name characters, per se, but Unicode supplemental characters *are* valid (which are represented by the final alternative of the rule for *SimpleNameChar*), and they should be represented in a file as pairs of surrogate code points in the MUTF-8 encoding.

SimpleName →	
SimpleNameChar (*SimpleNameChar*)*	
SimpleNameChar →	
`'A'` … `'Z'`	
| `'a'` … `'z'`	
| `'0'` … `'9'`	
| `' '`	since DEX version 040
| `'$'`	

<table>
<tr><td>| '-'</td><td></td></tr>
<tr><td>| '_'</td><td></td></tr>
<tr><td>| U+00a0</td><td>since DEX version 040</td></tr>
<tr><td>| U+00a1 … U+1fff</td><td></td></tr>
<tr><td>| U+2000 … U+200a</td><td>since DEX version 040</td></tr>
<tr><td>| U+2010 … U+2027</td><td></td></tr>
<tr><td>| U+202f</td><td>since DEX version 040</td></tr>
<tr><td>| U+2030 … U+d7ff</td><td></td></tr>
<tr><td>| U+e000 … U+ffef</td><td></td></tr>
<tr><td>| U+10000 … U+10ffff</td><td></td></tr>
</table>

MemberName

used by field_id_item and method_id_item

A *MemberName* is the name of a member of a class, members being fields, methods, and inner classes.

<table>
<tr><td><i>MemberName</i> →</td></tr>
<tr><td> <i>SimpleName</i></td></tr>
<tr><td>| '&lt;' <i>SimpleName</i> '&gt;'</td></tr>
</table>

FullClassName

A *FullClassName* is a fully-qualified class name, including an optional package specifier followed by a required name.

<table>
<tr><td><i>FullClassName</i> →</td></tr>
</table>

OptionalPackagePrefix SimpleName

OptionalPackagePrefix →

(*SimpleName* `'/'`)*

TypeDescriptor

used by type_id_item

A *TypeDescriptor* is the representation of any type, including primitives, classes, arrays, and `void`. See below for the meaning of the various versions.

TypeDescriptor →

`'V'`

| *FieldTypeDescriptor*

FieldTypeDescriptor →

NonArrayFieldTypeDescriptor

| (`'['` * 1…255) *NonArrayFieldTypeDescriptor*

NonArrayFieldTypeDescriptor→

`'Z'`

| `'B'`

| `'S'`

| `'C'`

| `'I'`

| `'J'`

| `'F'`

| `'D'`

| `'L'` *FullClassName* `';'`

ShortyDescriptor

used by proto_id_item

A *ShortyDescriptor* is the short form representation of a method prototype, including return and parameter types, except that there is no distinction between various reference (class or array) types. Instead, all reference types are represented by a single `'L'` character.

ShortyDescriptor →

ShortyReturnType (*ShortyFieldType*)*

ShortyReturnType →

`'V'`

| *ShortyFieldType*

ShortyFieldType →

`'Z'`

| `'B'`

| `'S'`

| `'C'`

| `'I'`

| `'J'`

| `'F'`

| `'D'`

| `'L'`

TypeDescriptor Semantics

This is the meaning of each of the variants of *TypeDescriptor*.

Syntax	Meaning
V	`void`; only valid for return types
Z	`boolean`
B	`byte`
S	`short`
C	`char`
I	`int`
J	`long`
F	`float`
D	`double`
L*fully/qualified/Name*;	the class *`fully.qualified.Name`*
[*descriptor*	array of *`descriptor`*, usable recursively for arrays-of-arrays, though it is invalid to have more than 255 dimensions.

Items and related structures

This section includes definitions for each of the top-level items that may appear in a `.dex` file.

header_item

header_item

appears in the header section

alignment: 4 bytes

Name	Format	Description
magic	ubyte[8] = DEX_FILE_MAGIC	magic value. See discussion above under "**`DEX_FILE_MAGIC`**" for more details.
checksum	uint	adler32 checksum of the rest of the file (everything but **`magic`** and this field); used to detect file corruption
signature	ubyte[20]	SHA-1 signature (hash) of the rest of the file (everything but **`magic`**, **`checksum`**, and this field); used to uniquely identify files
file_size	uint	size of the entire file (including the header), in bytes
header_size	uint = 0x70	size of the header (this entire section), in bytes. This allows for at least a limited amount of backwards/forwards compatibility without invalidating the format.
endian_tag	uint = ENDIAN_CONSTANT	endianness tag. See discussion above under "**`ENDIAN_CONSTANT`** and **`REVERSE_ENDIAN_CONSTANT`**" for more details.
link_size	uint	size of the link section, or **`0`** if this file isn't statically linked
link_off	uint	offset from the start of the file to the link section, or **`0`** if **`link_size == 0`**. The offset, if non-zero, should be to an offset into the **`link_data`** section. The format of the data pointed at is left unspecified by this document; this header field (and the previous) are left as hooks for use by runtime implementations.
map_off	uint	offset from the start of the file to the map item. The offset, which must be non-zero, should be to an

		offset, which must be non-zero, should be to an offset into the `data` section, and the data should be in the format specified by "`map_list`" below.
string_ids_size	uint	count of strings in the string identifiers list
string_ids_off	uint	offset from the start of the file to the string identifiers list, or `0` if `string_ids_size == 0` (admittedly a strange edge case). The offset, if non-zero, should be to the start of the `string_ids` section.
type_ids_size	uint	count of elements in the type identifiers list, at most 65535
type_ids_off	uint	offset from the start of the file to the type identifiers list, or `0` if `type_ids_size == 0` (admittedly a strange edge case). The offset, if non-zero, should be to the start of the `type_ids` section.
proto_ids_size	uint	count of elements in the prototype identifiers list, at most 65535
proto_ids_off	uint	offset from the start of the file to the prototype identifiers list, or `0` if `proto_ids_size == 0` (admittedly a strange edge case). The offset, if non-zero, should be to the start of the `proto_ids` section.
field_ids_size	uint	count of elements in the field identifiers list
field_ids_off	uint	offset from the start of the file to the field identifiers list, or `0` if `field_ids_size == 0`. The offset, if non-zero, should be to the start of the `field_ids` section.
method_ids_size	uint	count of elements in the method identifiers list
method_ids_off	uint	offset from the start of the file to the method identifiers list, or `0` if `method_ids_size == 0`. The offset, if non-zero, should be to the start of the `method_ids` section.
class_defs_size	uint	count of elements in the class definitions list

class_defs_off	uint	offset from the start of the file to the class definitions list, or `0` if `class_defs_size == 0` (admittedly a strange edge case). The offset, if non-zero, should be to the start of the `class_defs` section.
data_size	uint	Size of `data` section in bytes. Must be an even multiple of sizeof(uint).
data_off	uint	offset from the start of the file to the start of the `data` section.

map_list

appears in the data section

referenced from header_item

alignment: 4 bytes

This is a list of the entire contents of a file, in order. It contains some redundancy with respect to the `header_item` but is intended to be an easy form to use to iterate over an entire file. A given type must appear at most once in a map, but there is no restriction on what order types may appear in, other than the restrictions implied by the rest of the format (e.g., a `header` section must appear first, followed by a `string_ids` section, etc.). Additionally, the map entries must be ordered by initial offset and must not overlap.

Name	Format	Description
size	uint	size of the list, in entries
list	map_item[size]	elements of the list

map_item format

Name	Format	Description
type	ushort	type of the items; see table below
unused	ushort	*(unused)*
size	uint	count of the number of items to be found at the indicated offset
offset	uint	offset from the start of the file to the items in question

Type Codes

Item Type	Constant	Value	Item Size In Bytes
header_item	TYPE_HEADER_ITEM	0x0000	0x70
string_id_item	TYPE_STRING_ID_ITEM	0x0001	0x04
type_id_item	TYPE_TYPE_ID_ITEM	0x0002	0x04
proto_id_item	TYPE_PROTO_ID_ITEM	0x0003	0x0c
field_id_item	TYPE_FIELD_ID_ITEM	0x0004	0x08
method_id_item	TYPE_METHOD_ID_ITEM	0x0005	0x08
class_def_item	TYPE_CLASS_DEF_ITEM	0x0006	0x20
call_site_id_item	TYPE_CALL_SITE_ID_ITEM	0x0007	0x04
method_handle_item	TYPE_METHOD_HANDLE_ITEM	0x0008	0x08
map_list	TYPE_MAP_LIST	0x1000	4 + (item.size * 12)
type_list	TYPE_TYPE_LIST	0x1001	4 + (item.size * 2)
annotation_set_ref_list	TYPE_ANNOTATION_SET_REF_LIST	0x1002	4 + (item.size *

			4)
annotation_set_item	TYPE_ANNOTATION_SET_ITEM	0x1003	4 + (item.size * 4)
class_data_item	TYPE_CLASS_DATA_ITEM	0x2000	*implicit; must parse*
code_item	TYPE_CODE_ITEM	0x2001	*implicit; must parse*
string_data_item	TYPE_STRING_DATA_ITEM	0x2002	*implicit; must parse*
debug_info_item	TYPE_DEBUG_INFO_ITEM	0x2003	*implicit; must parse*
annotation_item	TYPE_ANNOTATION_ITEM	0x2004	*implicit; must parse*
encoded_array_item	TYPE_ENCODED_ARRAY_ITEM	0x2005	*implicit; must parse*
annotations_directory_item	TYPE_ANNOTATIONS_DIRECTORY_ITEM	0x2006	*implicit; must parse*
hiddenapi_class_data_item	TYPE_HIDDENAPI_CLASS_DATA_ITEM	0xF000	*implicit; must parse*

string_id_item

appears in the string_ids section

alignment: 4 bytes

Name	Format	Description
string_data_off	uint	offset from the start of the file to the string data for this item. The offset should be to a location in the `data` section, and the data should be in the format specified by "`string_data_item`" below.

There is no alignment requirement for the offset.

string_data_item

appears in the data section

alignment: none (byte-aligned)

Name	Format	Description
utf16_size	uleb128	size of this string, in UTF-16 code units (which is the "string length" in many systems). That is, this is the decoded length of the string. (The encoded length is implied by the position of the `0` byte.)
data	ubyte[]	a series of MUTF-8 code units (a.k.a. octets, a.k.a. bytes) followed by a byte of value `0`. See "MUTF-8 (Modified UTF-8) Encoding" above for details and discussion about the data format. ★ **Note:** It is acceptable to have a string which includes (the encoded form of) UTF-16 surrogate code units (that is, **`U+d800`** … **`U+dfff`**) either in isolation or out-of-order with respect to the usual encoding of Unicode into UTF-16. It is up to higher-level uses of strings to reject such invalid encodings, if appropriate.

type_id_item

appears in the type_ids section

alignment: 4 bytes

Name	Format	Description
descriptor_idx	uint	index into the **`string_ids`** list for the descriptor string of this type. The string must conform to the syntax for *TypeDescriptor*, defined above.

proto_id_item

appears in the proto_ids section

alignment: 4 bytes

Name	Format	Description
shorty_idx	uint	index into the `string_ids` list for the short-form descriptor string of this prototype. The string must conform to the syntax for *ShortyDescriptor*, defined above, and must correspond to the return type and parameters of this item.
return_type_idx	uint	index into the `type_ids` list for the return type of this prototype
parameters_off	uint	offset from the start of the file to the list of parameter types for this prototype, or `0` if this prototype has no parameters. This offset, if non-zero, should be in the `data` section, and the data there should be in the format specified by `"type_list"` below. Additionally, there should be no reference to the type `void` in the list.

field_id_item

appears in the field_ids section

alignment: 4 bytes

Name	Format	Description
class_idx	ushort	index into the `type_ids` list for the definer of this field. This must be a class type, and not an array or primitive type.
type_idx	ushort	index into the `type_ids` list for the type of this field
name_idx	uint	index into the `string_ids` list for the name of this field. The string

name_idx	uint	index into the `string_ids` list for the name of this field. The string must conform to the syntax for *MemberName*, defined above.

method_id_item

appears in the method_ids section

alignment: 4 bytes

Name	Format	Description
class_idx	ushort	index into the `type_ids` list for the definer of this method. This must be a class or array type, and not a primitive type.
proto_idx	ushort	index into the `proto_ids` list for the prototype of this method
name_idx	uint	index into the `string_ids` list for the name of this method. The string must conform to the syntax for *MemberName*, defined above.

class_def_item

appears in the class_defs section

alignment: 4 bytes

Name	Format	Description
class_idx	uint	index into the `type_ids` list for this class. This must be a class type, and not an array or primitive type.
access_flags	uint	access flags for the class (`public`, `final`, etc.). See "`access_flags` Definitions" for details.
superclass_idx	uint	index into the `type_ids` list for the superclass, or the constant value `NO_INDEX` if this class has no superclass (i.e., it is a root class such as `Object`). If present, this must be a class type, and not an array or primitive type.

interfaces_off	uint	offset from the start of the file to the list of interfaces, or `0` if there are none. This offset should be in the `data` section, and the data there should be in the format specified by "`type_list`" below. Each of the elements of the list must be a class type (not an array or primitive type), and there must not be any duplicates.
source_file_idx	uint	index into the `string_ids` list for the name of the file containing the original source for (at least most of) this class, or the special value `NO_INDEX` to represent a lack of this information. The `debug_info_item` of any given method may override this source file, but the expectation is that most classes will only come from one source file.
annotations_off	uint	offset from the start of the file to the annotations structure for this class, or `0` if there are no annotations on this class. This offset, if non-zero, should be in the `data` section, and the data there should be in the format specified by "`annotations_directory_item`" below, with all items referring to this class as the definer.
class_data_off	uint	offset from the start of the file to the associated class data for this item, or `0` if there is no class data for this class. (This may be the case, for example, if this class is a marker interface.) The offset, if non-zero, should be in the `data` section, and the data there should be in the format specified by "`class_data_item`" below, with all items referring to this class as the definer.
static_values_off	uint	offset from the start of the file to the list of initial values for `static` fields, or `0` if there are none (and all `static` fields are to be initialized with `0` or `null`). This offset should be in the `data` section, and the data there should be in the format specified by "`encoded_array_item`" below. The size of the array must be no larger than the number of `static` fields declared by this class, and the elements correspond to the `static` fields in the same order as declared in the corresponding `field_list`. The type of each array element must match the declared type of its corresponding field. If there are fewer elements in the array than there are `static` fields, then the leftover fields are initialized with a type-appropriate `0` or `null`.

call_site_id_item

appears in the call_site_ids section

alignment: 4 bytes

Name	Format	Description
call_site_off	uint	offset from the start of the file to call site definition. The offset should be in the data section, and the data there should be in the format specified by "call_site_item" below.

call_site_item

appears in the data section

alignment: none (byte aligned)

The call_site_item is an encoded_array_item whose elements correspond to the arguments provided to a bootstrap linker method. The first three arguments are:

1. A method handle representing the bootstrap linker method (VALUE_METHOD_HANDLE).
2. A method name that the bootstrap linker should resolve (VALUE_STRING).
3. A method type corresponding to the type of the method name to be resolved (VALUE_METHOD_TYPE).

Any additional arguments are constant values passed to the bootstrap linker method. These arguments are passed in order and without any type conversions.

The method handle representing the bootstrap linker method must have return type `java.lang.invoke.CallSite`. The first three parameter types are:

1. `java.lang.invoke.Lookup`
2. `java.lang.String`
3. `java.lang.invoke.MethodType`

The parameter types of any additional arguments are determined from their constant values.

method_handle_item

appears in the method_handles section

alignment: 4 bytes

Name	Format	Description
method_handle_type	ushort	type of the method handle; see table below
unused	ushort	*(unused)*
field_or_method_id	ushort	Field or method id depending on whether the method handle type is an accessor or a method invoker
unused	ushort	*(unused)*

Method Handle Type Codes

Constant	Value	Description
METHOD_HANDLE_TYPE_STATIC_PUT	0x00	Method handle is a static field setter (accessor)
METHOD_HANDLE_TYPE_STATIC_GET	0x01	Method handle is a static field getter (accessor)
METHOD_HANDLE_TYPE_INSTANCE_PUT	0x02	Method handle is an instance field setter (accessor)

METHOD_HANDLE_TYPE_INSTANCE_GET	0x03	Method handle is an instance field getter (accessor)
METHOD_HANDLE_TYPE_INVOKE_STATIC	0x04	Method handle is a static method invoker
METHOD_HANDLE_TYPE_INVOKE_INSTANCE	0x05	Method handle is an instance method invoker
METHOD_HANDLE_TYPE_INVOKE_CONSTRUCTOR	0x06	Method handle is a constructor method invoker
METHOD_HANDLE_TYPE_INVOKE_DIRECT	0x07	Method handle is a direct method invoker
METHOD_HANDLE_TYPE_INVOKE_INTERFACE	0x08	Method handle is an interface method invoker

class_data_item

referenced from class_def_item

appears in the data section

alignment: none (byte-aligned)

Name	Format	Description
static_fields_size	uleb128	the number of static fields defined in this item
instance_fields_size	uleb128	the number of instance fields defined in this item
direct_methods_size	uleb128	the number of direct methods defined in this item
virtual_methods_size	uleb128	the number of virtual methods defined in this item

static_fields	encoded_field[static_fields_size]	the defined static fields, represented as a sequence of encoded elements. The fields must be sorted by **`field_idx`** in increasing order.
instance_fields	encoded_field[instance_fields_size]	the defined instance fields, represented as a sequence of encoded elements. The fields must be sorted by **`field_idx`** in increasing order.
direct_methods	encoded_method[direct_methods_size]	the defined direct (any of **`static`**, **`private`**, or constructor) methods, represented as a sequence of encoded elements. The methods must be sorted by **`method_idx`** in increasing order.
virtual_methods	encoded_method[virtual_methods_size]	the defined virtual (none of **`static`**, **`private`**, or constructor) methods, represented as a sequence of encoded elements. This list should *not* include inherited methods unless overridden by the class that this item represents. The methods must be sorted by **`method_idx`** in increasing order. The **`method_idx`** of a virtual method must *not* be the same as any direct method.

Note: All elements' **`field_id`**s and **`method_id`**s must refer to the same defining class.

encoded_field format

Name	Format	Description
field_idx_diff	uleb128	index into the **`field_ids`** list for the identity of this field (includes the name and descriptor), represented as a difference from the index of previous element in the list. The index of the first element in a list is represented directly.
access_flags	uleb128	access flags for the field (**`public`**, **`final`**, etc.). See "**`access_flags`** Definitions" for details.

encoded_method format

Name	Format	Description
method_idx_diff	uleb128	index into the **`method_ids`** list for the identity of this method (includes the name and descriptor), represented as a difference from the index of previous element in the list. The index of the first element in a list is represented directly.
access_flags	uleb128	access flags for the method (**`public`**, **`final`**, etc.). See "**`access_flags`** Definitions" for details.
code_off	uleb128	offset from the start of the file to the code structure for this method, or **`0`** if this method is either **`abstract`** or **`native`**. The offset should be to a location in the **`data`** section. The format of the data is specified by "**`code_item`**" below.

type_list

referenced from class_def_item and proto_id_item

referenced from class_def_item and proto_id_item

appears in the data section

alignment: 4 bytes

Name	Format	Description
size	uint	size of the list, in entries
list	type_item[size]	elements of the list

type_item format

Name	Format	Description
type_idx	ushort	index into the `type_ids` list

code_item

referenced from encoded_method

appears in the data section

alignment: 4 bytes

Name	Format	Description
registers_size	ushort	the number of registers used by this code
ins_size	ushort	the number of words of incoming arguments to the method that this code is for
outs_size	ushort	the number of words of outgoing argument space required by this code for method invocation

tries_size	ushort	the number of `try_item`s for this instance. If non-zero, then these appear as the `tries` array just after the `insns` in this instance.
debug_info_off	uint	offset from the start of the file to the debug info (line numbers + local variable info) sequence for this code, or `0` if there simply is no information. The offset, if non-zero, should be to a location in the `data` section. The format of the data is specified by "`debug_info_item`" below.
insns_size	uint	size of the instructions list, in 16-bit code units
insns	ushort[insns_size]	actual array of bytecode. The format of code in an `insns` array is specified by the companion document Dalvik bytecode (/devices/tech/dalvik/dalvik-bytecode.html). Note that though this is defined as an array of `ushort`, there are some internal structures that prefer four-byte alignment. Also, if this happens to be in an endian-swapped file, then the swapping is *only* done on individual `ushort`s and not on the larger internal structures.
padding	ushort *(optional)* = 0	two bytes of padding to make `tries` four-byte aligned. This element is only present if `tries_size` is non-zero and `insns_size` is odd.
tries	try_item[tries_size] *(optional)*	array indicating where in the code exceptions are caught and how to handle them. Elements of the array must be non-overlapping in range and in order from low to high address. This element is only present if `tries_size` is non-zero.
handlers	encoded_catch_handler_list *(optional)*	bytes representing a list of lists of catch types and associated handler addresses. Each `try_item` has a byte-wise offset into this structure. This element is only present if `tries_size` is non-zero.

tries_size is non-zero.

try_item format

Name	Format	Description
start_addr	uint	start address of the block of code covered by this entry. The address is a count of 16-bit code units to the start of the first covered instruction.
insn_count	ushort	number of 16-bit code units covered by this entry. The last code unit covered (inclusive) is `start_addr + insn_count - 1`.
handler_off	ushort	offset in bytes from the start of the associated `encoded_catch_hander_list` to the `encoded_catch_handler` for this entry. This must be an offset to the start of an `encoded_catch_handler`.

encoded_catch_handler_list format

Name	Format	Description
size	uleb128	size of this list, in entries
list	encoded_catch_handler[handlers_size]	actual list of handler lists, represented directly (not as offsets), and concatenated sequentially

encoded_catch_handler format

Name	Format	Description
size	sleb128	number of catch types in this list. If non-positive, then this is the negative of the

		number of catch types, and the catches are followed by a catch-all handler. For example: A `size` of `0` means that there is a catch-all but no explicitly typed catches. A `size` of `2` means that there are two explicitly typed catches and no catch-all. And a `size` of `-1` means that there is one typed catch along with a catch-all.
handlers	encoded_type_addr_pair[abs(size)]	stream of `abs(size)` encoded items, one for each caught type, in the order that the types should be tested.
catch_all_addr	uleb128 *(optional)*	bytecode address of the catch-all handler. This element is only present if `size` is non-positive.

encoded_type_addr_pair format

Name	Format	Description
type_idx	uleb128	index into the `type_ids` list for the type of the exception to catch
addr	uleb128	bytecode address of the associated exception handler

debug_info_item

referenced from code_item

appears in the data section

alignment: none (byte-aligned)

Each `debug_info_item` defines a DWARF3-inspired byte-coded state machine that, when interpreted, emits the positions table and (potentially) the local variable information for a `code_item`. The sequence begins with a variable-length header (the length of which depends on the number of method parameters), is followed by the state machine bytecodes, and ends with an `DBG_END_SEQUENCE` byte.

The state machine consists of five registers. The `address` register represents the instruction offset in the associated `insns_item` in 16-bit code units. The `address` register starts at `0` at the beginning of each `debug_info` sequence and must only monotonically increase. The `line` register represents what source line number should be associated with the next positions table entry emitted by the state machine. It is initialized in the sequence header, and may change in positive or negative directions but must never be less than `1`. The `source_file` register represents the source file that the line number entries refer to. It is initialized to the value of `source_file_idx` in `class_def_item`. The other two variables, `prologue_end` and `epilogue_begin`, are boolean flags (initialized to `false`) that indicate whether the next position emitted should be considered a method prologue or epilogue. The state machine must also track the name and type of the last local variable live in each register for the `DBG_RESTART_LOCAL` code.

The header is as follows:

Name	Format	Description
line_start	uleb128	the initial value for the state machine's `line` register. Does not represent an actual positions entry.
parameters_size	uleb128	the number of parameter names that are encoded. There should be one per method parameter, excluding an instance method's `this`, if any.
parameter_names	uleb128p1[parameters_size]	string index of the method parameter name. An encoded value of `NO_INDEX` indicates that no name is available for the associated parameter. The type descriptor and signature are implied from the method descriptor and signature.

The byte code values are as follows:

Name	Value	Format	Arguments	Description
DBG_END_SEQUENCE	0x00		*(none)*	terminates a c a `code_item`
DBG_ADVANCE_PC	0x01	uleb128 addr_diff	`addr_diff`: amount to add to address register	advances the without emitti
DBG_ADVANCE_LINE	0x02	sleb128 line_diff	`line_diff`: amount to change line register by	advances the emitting a pos
DBG_START_LOCAL	0x03	uleb128 register_num uleb128p1 name_idx uleb128p1 type_idx	`register_num`: register that will contain local `name_idx`: string index of the name `type_idx`: type index of the type	introduces a l current addre `type_idx` ma indicate that t
DBG_START_LOCAL_EXTENDED	0x04	uleb128 register_num uleb128p1 name_idx uleb128p1 type_idx uleb128p1 sig_idx	`register_num`: register that will contain local `name_idx`: string index of the name `type_idx`: type index of the type `sig_idx`: string index of the type signature	introduces a l signature at tl of `name_idx` `sig_idx` may indicate that t (If `sig_idx` is data could be efficiently usi `DBG_START_` ★ **Note:** See the "`dalvik.ann`

				below for cav signatures.
DBG_END_LOCAL	0x05	uleb128 register_num	**register_num**: register that contained local	marks a curre as out of scop address
DBG_RESTART_LOCAL	0x06	uleb128 register_num	**register_num**: register to restart	re-introduces current addre are the same was live in the
DBG_SET_PROLOGUE_END	0x07		*(none)*	sets the **prol** machine regis next position should be con method prolo place for a me **prologue_e** any special (**>**
DBG_SET_EPILOGUE_BEGIN	0x08		*(none)*	sets the **epil** machine regis next position should be con of a method e place to susp method exit). register is cle **0x0a**) opcod
DBG_SET_FILE	0x09	uleb128p1 name_idx	**name_idx**: string index of source file name; **NO_INDEX** if unknown	indicates that number entrie source file na default name
Special Opcodes	0x0a… 0xff		*(none)*	advances the registers, emi clears **prolo**

epilogue_b
description.

Special opcodes

Opcodes with values between `0x0a` and `0xff` (inclusive) move both the `line` and `address` registers by a small amount and then emit a new position table entry. The formula for the increments are as follows:

```
DBG_FIRST_SPECIAL = 0x0a  // the smallest special opcode
DBG_LINE_BASE   = -4      // the smallest line number increment
DBG_LINE_RANGE  = 15      // the number of line increments represen

adjusted_opcode = opcode - DBG_FIRST_SPECIAL

line += DBG_LINE_BASE + (adjusted_opcode % DBG_LINE_RANGE)
address += (adjusted_opcode / DBG_LINE_RANGE)
```

annotations_directory_item

referenced from class_def_item

appears in the data section

alignment: 4 bytes

Name	Format	Description
class_annotations_off	uint	offset from the start of file to the annotations made directly on the cla or `0` if the class has no direct annotations. The offset, if non-zero, shou

		offset, if non-zero, shou be to a location in the **data** section. The form of the data is specified "**annotation_set_it** below.
fields_size	uint	count of fields annotate by this item
annotated_methods_size	uint	count of methods annotated by this item
annotated_parameters_size	uint	count of method parameter lists annotat by this item
field_annotations	field_annotation[fields_size] *(optional)*	list of associated field annotations. The eleme of the list must be sorte in increasing order, by **field_idx**.
method_annotations	method_annotation[methods_size] *(optional)*	list of associated meth annotations. The eleme of the list must be sorte in increasing order, by **method_idx**.
parameter_annotations	parameter_annotation[parameters_size] *(optional)*	list of associated meth parameter annotations. The elements of the list must be sorted in increasing order, by **method_idx**.

Note: All elements' **field_id**s and **method_id**s must refer to the same defining class.

field_annotation format

Name	Format	Description
field_idx	uint	index into the `field_ids` list for the identity of the field being annotated
annotations_off	uint	offset from the start of the file to the list of annotations for the field. The offset should be to a location in the `data` section. The format of the data is specified by "`annotation_set_item`" below.

method_annotation format

Name	Format	Description
method_idx	uint	index into the `method_ids` list for the identity of the method being annotated
annotations_off	uint	offset from the start of the file to the list of annotations for the method. The offset should be to a location in the `data` section. The format of the data is specified by "`annotation_set_item`" below.

parameter_annotation format

Name	Format	Description
method_idx	uint	index into the `method_ids` list for the identity of the method whose parameters are being annotated
annotations_off	uint	offset from the start of the file to the list of annotations for the method parameters. The offset should be to a location in the `data` section. The format of the data is specified by "`annotation_set_ref_list`" below.

annotation_set_ref_list

referenced from parameter_annotations_item

appears in the data section

alignment: 4 bytes

Name	Format	Description
size	uint	size of the list, in entries
list	annotation_set_ref_item[size]	elements of the list

annotation_set_ref_item format

Name	Format	Description
annotations_off	uint	offset from the start of the file to the referenced annotation set or `0` if there are no annotations for this element. The offset, if non-zero, should be to a location in the `data` section. The format of the data is specified by "`annotation_set_item`" below.

annotation_set_item

referenced from annotations_directory_item, field_annotations_item, method_annotations_item, and annotation_set_ref_item

appears in the data section

alignment: 4 bytes

Name	Format	Description
size	uint	size of the set, in entries

entries	annotation_off_item[size]	elements of the set. The elements must be sorted in increasing order, by `type_idx`.

annotation_off_item format

Name	Format	Description
annotation_off	uint	offset from the start of the file to an annotation. The offset should be to a location in the `data` section, and the format of the data at that location is specified by "`annotation_item`" below.

annotation_item

referenced from annotation_set_item

appears in the data section

alignment: none (byte-aligned)

Name	Format	Description
visibility	ubyte	intended visibility of this annotation (see below)
annotation	encoded_annotation	encoded annotation contents, in the format described by "`encoded_annotation` format" under "`encoded_value` encoding" above.

Visibility values

These are the options for the `visibility` field in an `annotation_item`:

Name	Value	Description
VISIBILITY_BUILD	0x00	intended only to be visible at build time (e.g., during

VISIBILITY_BUILD	0x00	intended only to be visible at build time (e.g., during compilation of other code)
VISIBILITY_RUNTIME	0x01	intended to visible at runtime
VISIBILITY_SYSTEM	0x02	intended to visible at runtime, but only to the underlying system (and not to regular user code)

encoded_array_item

referenced from class_def_item

appears in the data section

alignment: none (byte-aligned)

Name	Format	Description
value	encoded_array	bytes representing the encoded array value, in the format specified by "`encoded_array` Format" under "`encoded_value` Encoding" above.

hiddenapi_class_data_item

This section contains data on restricted interfaces used by each class.

Note: The hidden API feature was introduced in Android 10.0 and is only applicable to the DEX files of classes in the boot class path. The list of flags described below may be extended in the future releases of Android. For more information, see restrictions on non-SDK interfaces (https://developer.android.com/distribute/best-practices/develop/restrictions-non-sdk-interfaces)

.

Name	Format	Description
size	uint	total size of the section
offsets	uint[]	array of offsets indexed by **`class_idx`**. A zero array entry at index **`class_idx`** means that either there is no data for this **`class_idx`**, or all hidden API flags are zero. Otherwise the array entry is non-zero and contains an offset from the beginning of the section to an array of hidden API flags for this **`class_idx`**.
flags	uleb128[]	concatenated arrays of hidden API flags for each class. Possible flag values are described in the table below. Flags are encoded in the same order as fields and methods are encoded in class data.

Restriction flag types:

Name	Value	Description
whitelist	0	Interfaces that can be freely used and are supported as part of the officially documented Android framework Package Index (https://developer.android.com/reference/packages.html).
greylist	1	Non-SDK interfaces that can be used regardless of the application's target API level (https://developer.android.com/distribute/best-practices/develop/target-sdk) .
blacklist	2	Non-SDK interfaces that cannot be used regardless of the application's target API level (https://developer.android.com/distribute/best-practices/develop/target-sdk) . Accessing one of these interfaces causes a runtime error (https://developer.android.com/distribute/best-practices/develop/restrictions-non-sdk-interfaces#results-of-keeping-non-sdk) .
greylist-max-o3		Non-SDK interfaces that can be used for Android 8.x and below unless they are restricted.

greylist-max-p4	Non-SDK interfaces that can be used for Android 9.x unless they are restricted.
greylist-max-q5	Non-SDK interfaces that can be used for Android 10.x unless they are restricted.

System annotations

System annotations are used to represent various pieces of reflective information about classes (and methods and fields). This information is generally only accessed indirectly by client (non-system) code.

System annotations are represented in `.dex` files as annotations with visibility set to `VISIBILITY_SYSTEM`.

dalvik.annotation.AnnotationDefault

appears on methods in annotation interfaces

An `AnnotationDefault` annotation is attached to each annotation interface which wishes to indicate default bindings.

Name	Format	Description
value	Annotation	the default bindings for this annotation, represented as an annotation of this type. The annotation need not include all names defined by the annotation; missing names simply do not have defaults.

dalvik.annotation.EnclosingClass

appears on classes

An `EnclosingClass` annotation is attached to each class which is either defined as a member of another class, per se, or is anonymous but not defined within a method body (e.g., a synthetic inner class). Every class that has this annotation must also have an `InnerClass` annotation. Additionally, a class must not have both an `EnclosingClass` and an `EnclosingMethod` annotation.

Name	Format	Description
value	Class	the class which most closely lexically scopes this class

dalvik.annotation.EnclosingMethod

appears on classes

An `EnclosingMethod` annotation is attached to each class which is defined inside a method body. Every class that has this annotation must also have an `InnerClass` annotation. Additionally, a class must not have both an `EnclosingClass` and an `EnclosingMethod` annotation.

Name	Format	Description
value	Method	the method which most closely lexically scopes this class

dalvik.annotation.InnerClass

appears on classes

An `InnerClass` annotation is attached to each class which is defined in the lexical scope of another class's definition. Any class which has this annotation must also have *either* an `EnclosingClass` annotation *or* an `EnclosingMethod` annotation.

Name	FormatDescription	
name	String	the originally declared simple name of this class (not including

name	String	the originally declared simple name of this class (not including any package prefix). If this class is anonymous, then the name is **`null`**.
accessFlags	int	the originally declared access flags of the class (which may differ from the effective flags because of a mismatch between the execution models of the source language and target virtual machine)

dalvik.annotation.MemberClasses

appears on classes

A `MemberClasses` annotation is attached to each class which declares member classes. (A member class is a direct inner class that has a name.)

Name	Format	Description
value	Class[]	array of the member classes

dalvik.annotation.MethodParameters

appears on methods

Note: This annotation was added after Android 7.1. Its presence on earlier Android releases will be ignored.

A `MethodParameters` annotation is optional and can be used to provide parameter metadata such as parameter names and modifiers.

The annotation can be omitted from a method or constructor safely when the parameter metadata is not required at runtime. `java.lang.reflect.Parameter.isNamePresent()` can be used to check whether metadata is present for a parameter, and the associated reflection methods such as `java.lang.reflect.Parameter.getName()` will fall back to default behavior at runtime if the information is not present.

When including parameter metadata, compilers must include information for generated classes such as enums, since the parameter metadata includes whether or not a parameter is synthetic or mandated.

A `MethodParameters` annotation describes only individual method parameters. Therefore, compilers may omit the annotation entirely for constructors and methods that have no parameters, for the sake of code-size and runtime efficiency.

The arrays documented below must be the same size as for the `method_id_item` dex structure associated with the method, otherwise a `java.lang.reflect.MalformedParametersException` will be thrown at runtime.

That is: `method_id_item.proto_idx` -> `proto_id_item.parameters_off` -> `type_list.size` must be the same as `names().length` and `accessFlags().length`.

Because `MethodParameters` describes all formal method parameters, even those not explicitly or implicitly declared in source code, the size of the arrays may differ from the Signature or other metadata information that is based only on explicit parameters declared in source code. `MethodParameters` will also not include any information about type annotation receiver parameters that do not exist in the actual method signature.

Name	Format	Description
names	String[]	The names of formal parameters for the associated method. The array must not be null but must be empty if there are no formal parameters. A value in the array must be null if the formal parameter with that index has no name. If parameter name strings are empty or contain '.', ';', '[' or '/' then a `java.lang.reflect.MalformedParametersException` will be thrown at runtime.
accessFlags	int[]	The access flags of the formal parameters for the associated method. The array must not be null but must be empty if there are no

formal parameters.
The value is a bit mask with the following values:

- 0x0010 : final, the parameter was declared final
- 0x1000 : synthetic, the parameter was introduced by the compiler
- 0x8000 : mandated, the parameter is synthetic but also implied by the language specification

If any bits are set outside of this set then a `java.lang.reflect.MalformedParametersException` will be thrown at runtime.

dalvik.annotation.Signature

appears on classes, fields, and methods

A `Signature` annotation is attached to each class, field, or method which is defined in terms of a more complicated type than is representable by a `type_id_item`. The `.dex` format does not define the format for signatures; it is merely meant to be able to represent whatever signatures a source language requires for successful implementation of that language's semantics. As such, signatures are not generally parsed (or verified) by virtual machine implementations. The signatures simply get handed off to higher-level APIs and tools (such as debuggers). Any use of a signature, therefore, should be written so as not to make any assumptions about only receiving valid signatures, explicitly guarding itself against the possibility of coming across a syntactically invalid signature.

Because signature strings tend to have a lot of duplicated content, a `Signature` annotation is defined as an *array* of strings, where duplicated elements naturally refer to the same underlying data, and the signature is taken to be the concatenation of all the strings in the array. There are no rules about how to pull apart a signature into separate strings; that is entirely up to the tools that generate `.dex` files.

Name	Format	Description
value	String[]	the signature of this class or member, as an array of strings that is to be concatenated together

dalvik.annotation.Throws

appears on methods

A `Throws` annotation is attached to each method which is declared to throw one or more exception types.

Name	Format	Description
value	Class[]	the array of exception types thrown

Last updated 2020-01-06.