



Chapter 3: JNI Types and Data Structures

This chapter discusses how the JNI maps Java types to native C types.

This chapter covers the following topics:

- [Primitive Types](#)
- [Reference Types](#)
- [Field and Method IDs](#)
- [The Value Type](#)
- [Type Signatures](#)
- [Modified UTF-8 Strings](#)

Primitive Types

The following table describes Java primitive types and their machine-dependent native equivalents.

| Primitive Types and Native Equivalents | | |
|--|-------------|------------------|
| Java Type | Native Type | Description |
| boolean | jboolean | unsigned 8 bits |
| byte | jbyte | signed 8 bits |
| char | jchar | unsigned 16 bits |
| short | jshort | signed 16 bits |
| int | jint | signed 32 bits |
| long | jlong | signed 64 bits |
| float | jfloat | 32 bits |
| double | jdouble | 64 bits |
| void | void | not applicable |

The following definition is provided for convenience.

```
#define JNI_FALSE 0
#define JNI_TRUE  1
```

The jsize integer type is used to describe cardinal indices and sizes:

```
typedef jint jsize;
```

Reference Types

The JNI includes a number of reference types that correspond to different kinds of Java objects. JNI reference types are organized in the following hierarchy:

- jobject
 - jclass (java.lang.Class objects)
 - jstring (java.lang.String objects)
 - jarray (arrays)
 - jobjectArray (object arrays)
 - jbooleanArray (boolean arrays)
 - jbyteArray (byte arrays)
 - jcharArray (char arrays)
 - jshortArray (short arrays)
 - jintArray (int arrays)
 - jlongArray (long arrays)
 - jfloatArray (float arrays)
 - jdoubleArray (double arrays)
 - jthrowable (java.lang.Throwable objects)

In C, all other JNI reference types are defined to be the same as jobject. For example:

```
typedef jobject jclass;
```

In C++, JNI introduces a set of dummy classes to enforce the subtyping relationship. For example:

```
class _jobject {};
class _jclass : public _jobject {};
// ...
```

```
typedef _jobject *jobject;
typedef _jclass *jclass;
```

Field and Method IDs

Method and field IDs are regular C pointer types:

```
struct _jfieldID;           /* opaque structure */
typedef struct _jfieldID *jfieldID;  /* field IDs */

struct _jmethodID;         /* opaque structure */
typedef struct _jmethodID *jmethodID; /* method IDs */
```

The Value Type

The jvalue union type is used as the element type in argument arrays. It is declared as follows:

```
typedef union jvalue {
    jboolean z;
    jbyte    b;
    jchar    c;
    jshort   s;
    jint     i;
    jlong    j;
    jfloat   f;
    jdouble  d;
    jobject  l;
} jvalue;
```

Type Signatures

The JNI uses the Java VM's representation of type signatures. The following table shows these type signatures.

| Java VM Type Signatures | |
|---------------------------|-----------------------|
| Type Signature | Java Type |
| Z | boolean |
| B | byte |
| C | char |
| S | short |
| I | int |
| J | long |
| F | float |
| D | double |
| L fully-qualified-class ; | fully-qualified-class |
| [type | type[] |
| (arg-types) ret-type | method type |

For example, the Java method:

```
long f (int n, String s, int[] arr);
```

has the following type signature:

```
(ILjava/lang/String;[I)J
```

Modified UTF-8 Strings

The JNI uses modified UTF-8 strings to represent various string types. Modified UTF-8 strings are the same as those used by the Java VM. Modified UTF-8 strings are encoded so that character sequences that contain only non-null ASCII characters can be represented using only one byte per character, but all Unicode characters can be represented.

All characters in the range \u0001 to \u007F are represented by a single byte, as follows:

- 0xxxxxxx

The seven bits of data in the byte give the value of the character represented.

The null character ('\u0000 ') and characters in the range '\u0080 ' to '\u07FF ' are represented by a pair of bytes x and y:

- x: 110xxxxx
- y: 10yyyyyy

The bytes represent the character with the value ((x & 0x1f) << 6) + (y & 0x3f).

Characters in the range '\u0800 ' to '\uFFFF ' are represented by 3 bytes x, y, and z:

- x: 1110xxxx
- y: 10yyyyyy
- z: 10zzzzzz

The character with the value ((x & 0xf) << 12) + ((y & 0x3f) << 6) + (z & 0x3f) is represented by the bytes.

Characters with code points above U+FFFF (so-called *supplementary characters*) are represented by separately encoding the two surrogate code units of their UTF-16 representation. Each of the surrogate code units is represented by three bytes. This means, supplementary characters are represented by six bytes, *u*, *v*, *w*, *x*, *y*, and *z*:

- **u:** 11101101
- **v:** 1010vvvv
- **w:** 10wwwwww
- **x:** 11101101
- **y:** 1011yyyy
- **z:** 10zzzzzz

The character with the value $0x10000 + ((v \& 0x0f) \ll 16) + ((w \& 0x3f) \ll 10) + (y \& 0x0f) \ll 6 + (z \& 0x3f)$ is represented by the six bytes.

The bytes of multibyte characters are stored in the `class` file in big-endian (high byte first) order.

There are two differences between this format and the standard UTF-8 format. First, the null character (`char`)`0` is encoded using the two-byte format rather than the one-byte format. This means that modified UTF-8 strings never have embedded nulls. Second, only the one-byte, two-byte, and three-byte formats of standard UTF-8 are used. The Java VM does not recognize the four-byte format of standard UTF-8; it uses its own two-times-three-byte format instead.

For more information regarding the standard UTF-8 format, see section 3.9 *Unicode Encoding Forms* of *The Unicode Standard, Version 4.0*.

[Contents](#) | [Previous](#) | [Next](#)