



# K Native Interface (KNI)

---

Specification, Version 1.0

A Lightweight Native Interface for the  
Java™ 2 Platform, Micro Edition

Sun Microsystems, Inc.  
4150 Network Circle  
Santa Clara, California 95054  
U.S.A. 650-960-1300

October, 2002

Copyright © 2002 Sun Microsystems, Inc., 4150 Network Circle, Santa Clara, California 95054, U.S.A. All rights reserved.

Sun Microsystems, Inc. has intellectual property rights relating to technology embodied in the product that is described in this document. In particular, and without limitation, these intellectual property rights may include one or more of the U.S. patents listed at <http://www.sun.com/patents> and one or more additional patents or pending patent applications in the U.S. and in other countries.

This document and the product to which it pertains are distributed under licenses restricting their use, copying, distribution, and decompilation. No part of the product or of this document may be reproduced in any form by any means without prior written authorization of Sun and its licensors, if any.

Third-party software, including font technology, is copyrighted and licensed from Sun suppliers.

Sun, Sun Microsystems, the Sun logo, Java, J2ME, K Virtual Machine (KVM), and Java Developer Connection are trademarks or registered trademarks of Sun Microsystems, Inc. in the U.S. and other countries.

The Adobe® logo is a registered trademark of Adobe Systems, Incorporated.

Federal Acquisitions: Commercial Software - Government Users Subject to Standard License Terms and Conditions.

DOCUMENTATION IS PROVIDED "AS IS" AND ALL EXPRESS OR IMPLIED CONDITIONS, REPRESENTATIONS AND WARRANTIES, INCLUDING ANY IMPLIED WARRANTY OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NON-INFRINGEMENT, ARE DISCLAIMED, EXCEPT TO THE EXTENT THAT SUCH DISCLAIMERS ARE HELD TO BE LEGALLY INVALID.

---

Copyright © 2002 Sun Microsystems, Inc., 4150 Network Circle, Santa Clara, California 95054, Etats-Unis. Tous droits réservés.

Sun Microsystems, Inc. a les droits de propriété intellectuels relatants à la technologie incorporée dans le produit qui est décrit dans ce document. En particulier, et sans la limitation, ces droits de propriété intellectuels peuvent inclure un ou plus des brevets américains énumérés à <http://www.sun.com/patents> et un ou les brevets plus supplémentaires ou les applications de brevet en attente dans les Etats - Unis et dans les autres pays.

Ce produit ou document est protégé par un copyright et distribué avec des licences qui en restreignent l'utilisation, la copie, la distribution, et la décompilation. Aucune partie de ce produit ou document ne peut être reproduite sous aucune forme, par quelque moyen que ce soit, sans l'autorisation préalable et écrite de Sun et de ses bailleurs de licence, s'il y en a.

Le logiciel détenu par des tiers, et qui comprend la technologie relative aux polices de caractères, est protégé par un copyright et licencié par des fournisseurs de Sun.

Sun, Sun Microsystems, le logo Sun, Java, J2ME, et Java Developer Connection sont des marques de fabrique ou des marques déposées de Sun Microsystems, Inc. aux Etats-Unis et dans d'autres pays.

Le logo Adobe® est une marque déposée de Adobe Systems, Incorporated.

LA DOCUMENTATION EST FOURNIE "EN L'ÉTAT" ET TOUTES AUTRES CONDITIONS, DECLARATIONS ET GARANTIES EXPRESSES OU TACITES SONT FORMELLEMENT EXCLUES, DANS LA MESURE AUTORISEE PAR LA LOI APPLICABLE, Y COMPRIS NOTAMMENT TOUTE GARANTIE IMPLICITE RELATIVE A LA QUALITE MARCHANDE, A L'APTITUDE A UNE UTILISATION PARTICULIERE OU A L'ABSENCE DE CONTREFAÇON.



Please  
Recycle



Adobe PostScript

# Contents

---

## **Preface   vii**

## **1. Background   1**

1.1    Some History   1

1.2    Why KNI?   2

## **2. KNI Goals   15**

## **3. KNI Scope   19**

3.1    Version Information   20

3.2    Class and Interface Operations   20

3.3    Exceptions   20

3.4    Object Operations   20

3.5    Instance Field Access   21

3.6    Static Field Access   21

3.7    String Operations   21

3.8    Array Operations   21

3.9    Parameter (Operand Stack) Access   22

3.10   Handle Operations   22

## **4. KNI Data Types   23**

4.1    Primitive and Reference Types   23

4.1.1   Primitive Types   23

4.1.2   Reference Types   24

4.1.3	Return Types	25
4.2	Field IDs	25
4.3	String Formats	26
4.3.1	UTF-8 Strings	26
4.3.2	Class Descriptors	27
4.3.3	Field Descriptors	28
4.4	Constants	28
<b>5.</b>	<b>KNI Functions</b>	<b>31</b>
5.1	Version Information	32
5.1.1	KNI_GetVersion	32
5.2	Class and Interface Operations	33
5.2.1	KNI_FindClass	33
5.2.2	KNI_GetSuperClass	34
5.2.3	KNI_IsAssignableFrom	35
5.3	Exceptions	36
5.3.1	KNI_ThrowNew	36
5.3.2	KNI_FatalError	37
5.4	Object Operations	37
5.4.1	KNI_GetObjectClass	37
5.4.2	KNI_IsInstanceOf	38
5.5	Instance Field Access	38
5.5.1	KNI_GetFieldID	38
5.5.2	KNI_Get<Type>Field	39
5.5.3	KNI_Set<Type>Field	40
5.5.4	KNI_GetObjectField	41
5.5.5	KNI_SetObjectField	41
5.6	Static Field Access	42
5.6.1	KNI_GetStaticFieldID	42
5.6.2	KNI_GetStatic<Type>Field	43
5.6.3	KNI_SetStatic<Type>Field	44
5.6.4	KNI_GetStaticObjectField	45

5.6.5	KNI_SetStaticObjectField	45
5.7	String Operations	46
5.7.1	KNI_GetStringLength	46
5.7.2	KNI_GetStringRegion	47
5.7.3	KNI_NewString	47
5.7.4	KNI_NewStringUTF	48
5.8	Array Operations	49
5.8.1	KNI_GetArrayLength	49
5.8.2	KNI_Get<Type>ArrayElement	49
5.8.3	KNI_Set<Type>ArrayElement	51
5.8.4	KNI_GetObjectArrayElement	52
5.8.5	KNI_SetObjectArrayElement	52
5.8.6	KNI_GetRawArrayRegion	53
5.8.7	KNI_SetRawArrayRegion	53
5.9	Parameter (Operand Stack) Access	54
5.9.1	KNI_GetParameterAs<Type>	54
5.9.2	KNI_GetParameterAsObject	55
5.9.3	KNI_GetThisPointer	56
5.9.4	KNI_GetClassPointer	57
5.9.5	KNI_ReturnVoid	57
5.9.6	KNI_Return<Type>	58
5.10	Handle Operations	59
5.10.1	KNI_StartHandles	59
5.10.2	KNI_DeclareHandle	60
5.10.3	KNI_IsNullHandle	60
5.10.4	KNI_IsSameObject	61
5.10.5	KNI_ReleaseHandle	61
5.10.6	KNI_EndHandles	62
5.10.7	KNI_EndHandlesAndReturnObject	62
<b>6.</b>	<b>KNI Programming Overview</b>	<b>63</b>
6.1	The 'kni.h' Include File	63

6.2	Sample KNI Application	63
6.2.1	Java Code	63
6.2.2	The Corresponding Native Code	64
6.2.3	Compiling and Running the Sample Application in the KVM	65
<b>7.</b>	<b>Examples</b>	<b>67</b>
7.1	Parameter Passing	67
7.2	Returning Values from Native Functions	69
7.2.1	Returning Primitive Values	69
7.2.2	Returning Object References	69
7.2.3	Returning Null Object References	70
7.3	Accessing Fields	70
7.3.1	General Procedure for Accessing Fields	70
7.3.2	Accessing Instance Fields	71
7.3.3	Accessing Static Fields	72
7.4	Accessing Arrays	74
7.5	Accessing Strings	75

# Preface

---

This document, *K Native Interface Specification*, defines a new native function interface, called the *K Native Interface* (KNI), for the Java™ 2 Platform, Micro Edition (J2ME™). The K Native Interface is designed to improve the portability and compatibility of the native functions that are written for the K Virtual Machine (KVM), the CLDC HotSpot™ Implementation JVM, or other Java virtual machine implementations that support the J2ME Connected, Limited Device Configuration (CLDC) standard. KNI is designed as a logical subset of the Java Native Interface (JNI). However, KNI is significantly more lightweight and more efficient in small devices than a full JNI implementation.

---

## Who Should Use This Specification

The audience for this document includes:

1. Device manufacturers and other companies and individuals who want to port the K Virtual Machine (KVM) or another J2ME CLDC compliant virtual machine onto a new platform, and who want to link their own native functions or other native libraries into the virtual machine.
2. Third party (“clone”) virtual machine vendors and other companies and individuals who want to write their own Java virtual machine implementation, and who want to ensure that the native code linked into the virtual machine is as portable across different J2ME virtual machines as possible.

Note that KNI is an *implementation-level interface*. A virtual machine conforming to the CLDC Specification or another J2ME configuration specification is not required to support the K Native Interface, but may do so at the implementation level.

---

## Related Documents

*The Java™ Language Specification (Java Series), Second Edition* by James Gosling, Bill Joy, Guy Steele and Gilad Bracha. Addison-Wesley, 2000, ISBN 0-201-31008-2

*The Java™ Virtual Machine Specification (Java Series), Second Edition* by Tim Lindholm and Frank Yellin (Addison-Wesley, 1999)

*Programming Wireless Devices with the Java™ 2 Platform, Micro Edition (Java Series)* by Roger Riggs, Antero Taivalsaari, and Mark VandenBrink. Addison-Wesley, 2001, ISBN 0-201-74627-1.

*The Java™ Native Interface: Programmer's Guide and Specification (Java Series)* by Sheng Liang. Addison-Wesley, 1999, ISBN 0-201-32577-2.

*Connected, Limited Device Configuration Specification*, version 1.0, Java Community Process, Sun Microsystems, Inc.

<http://java.sun.com/aboutJava/communityprocess/final/jsr030/>

*Java 2 Platform Micro Edition (J2ME™) Technology for Creating Mobile Devices*, A White Paper, Sun Microsystems, Inc.

<http://java.sun.com/products/cldc/wp/KVMwp.pdf>

---

## Version History

*September 7, 2001*: The first 0.1 version based on an earlier draft prepared by Efren Serra.

*September 30, 2001*: Version 0.2 with significant changes to the goals and high-level structure of the document.

*October 8, 2001*: Version 0.3 with updated programming overview and significant changes to the data type and function chapters.

*October 10, 2001*: Version 0.4 prepared after detailed review meeting.

*November 6, 2001*: Version 0.9 with minor revisions to function descriptions. Removed `KNI_GetStringChars`. Changed `KNI_Throw` to `KNI_ThrowNew`. Added a small sample program to illustrate string access.

*January 3, 2002*: Version 0.99 -- Small polishing, revised examples.

*January 15, 2002*: Version 1.0: Two additional functions, `KNI_NewString` and `KNI_IsSameObject`, added. Final polishing.



*January 18, 2002:* Version 1.0: Replaced the KNI\_SetResultAs\* functions with more portable KNI\_Return\* functions. Some changes to the handle operations as well.

*February 11, 2002:* Version 1.0: Minor clarifications to the description of the KNI\_ThrowNew function. Updated the parameter types of KNI\_Get/SetRawArrayRegion functions.

*October 18, 2002:* Version 1.0: Reformatted the KDWP Specification version 1.0 for Section 508 Accessibility. No changes in technical content of the specification.



# Background

---

## 1.1 Some History

The Java™ 2 Platform, Micro Edition (J2ME™) has recently become very popular in the consumer device space. The roots of the J2ME platform can be traced back to the *Spotless* research project that was started at Sun Labs in January 1998. The Spotless project designed a new, compact, highly portable Java™ virtual machine that was aimed specifically for small, resource-constrained devices such as cellular phones, pagers, personal organizers, home appliances, and the likes. The product version of the Spotless virtual machine became known as the *K Virtual Machine (KVM)*.

The research goal of the Spotless project was to build a Java execution environment that would have a total memory footprint of only one tenth of the size of the full Java execution environment in 1998. Consequently, the designers of the Spotless system left out several features that were considered overly expensive or unnecessary for embedded systems use.

One of the features that was intentionally left out of the Spotless system was support for the Java Native Interface (JNI). The Java Native Interface is an API that is intended to provide binary compatibility for the native functions that are linked into a Java virtual machine, so that the virtual machine can perform operating system specific operations such as file system calls, graphics operations, and so on. By using the Java Native Interface, multiple Java virtual machine implementations can utilize the same native function implementations, and dynamically load the necessary native libraries in binary form. Unfortunately, JNI is rather expensive in terms of memory footprint and performance overhead. Furthermore, some of the JNI APIs pose potential security threats in the absence of the full Java security model.

The official standardization efforts for the Java 2 Platform, Micro Edition were started in October 1999. The Connected, Limited Device Configuration (CLDC) standardization effort (Java Specification Request JSR-30) decided to ratify the decision to leave out the Java Native Interface support. The way native functions

are linked into a virtual machine conforming to the CLDC Specification is strictly an implementation issue. Consequently, the product version of the Spotless system, the K Virtual Machine, does not support any of the JNI APIs.

---

## 1.2 Why KNI?

Currently, the K Virtual Machine has a very low-level VM-specific interface for native function support. When implementing new native functions for the K Virtual Machine, the programmer uses exactly the same API calls as the KVM would use internally for the equivalent operations. For example, to push and pop parameters to and from the execution stack in native functions, the native function programmer would use exactly the same 'pushStack' and 'popStack' operations that the KVM uses internally for various operations. Similarly, to read or change data stored in the fields of objects, the native function programmer would use exactly the same low-level constant pool resolution operations that the KVM uses internally.

While this approach is highly efficient and well-suited for experienced programmers who understand how the K Virtual Machine works, this approach has some drawbacks. For instance, since this approach exposes the internal data structures and operations of the KVM to the native function programmer, any changes to those internal structures and operations in future versions of the KVM could render the existing native function implementations invalid. Due to the dependency on KVM-specific data structures and operations, this also means that native functions written for the KVM would not work with any other Java virtual machine without considerable modifications.

When used by less experienced programmers, the current native function interface of the KVM can also be rather error-prone because of garbage collection issues. Starting from version 1.0.2, the KVM has a compacting garbage collector. This means that any time the programmer (directly or indirectly) calls functions that allocate memory from the Java heap, therefore potentially causing the VM to collect garbage, the existing objects in the Java heap may move to a new location in memory. As a result, any native pointers that the native function programmer holds to objects in the Java heap may become invalid. Unless the native function programmer is very careful, this can lead to spurious errors that are extremely difficult to trace without special tools.

Even though the majority of the programmers who need to add new native functions to the KVM are usually experienced embedded systems developers (e.g., software engineers working for major device manufacturers), it seems unnecessary to require all the native function programmers to be intimately familiar with the internal data structures and garbage collection details of the KVM. Also, as the number of library standardization efforts for the Java 2 Micro Edition grows, more and more programmers will need to add new native functionality to the KVM or any new virtual machine conforming to the CLDC Specification.

To facilitate the integration of native functionality across a wide variety of CLDC target devices, there is a need for a native function interface that provides high performance and low memory overhead without the pitfalls of low-level interfaces discussed above. For this purpose, a new interface called the *K Native Interface* (KNI) has been developed.



## KNI Goals

---

A Java virtual machine commonly needs access to various native functions in order to interact with the outside world. For instance, all the low-level graphics functions, file access functions, networking functions, or other similar routines that depend on the underlying operating system services typically need to be written in native code.

The way these native functions are made available to the Java virtual machine can vary considerably from one virtual machine implementation to another. In order to minimize the work that is needed when porting the native functions, the *Java Native Interface (JNI)* standard was created.

In a traditional Java virtual machine implementation, the Java Native Interface serves two purposes:

1. JNI serves as a common interface for virtual machine implementers so that the same native functions will work unmodified with different virtual machines.
2. JNI provides Java-level APIs that make it possible for a Java programmer to dynamically load libraries and access native functions in those libraries.

Unfortunately, because of its general nature, JNI is rather expensive and introduces a significant memory and performance overhead to the way the JVM calls native functions. Also, the ability to dynamically load and call arbitrary native functions from Java programs could pose significant security problems in the absence of the full Java 2 security model.

**High-level goal of the KNI.** The *high-level goal* of the K Native Interface Specification is to define a *logical subset of the Java Native Interface* that is *appropriate for low-power, memory-constrained devices*. KNI follows the function naming conventions and other aspects of the JNI as far as this is possible and reasonable within the strict memory limits of CLDC target devices and in the absence of the full Java 2 security model. Since KNI is intended to be significantly more lightweight than JNI, some aspects of the interface, such as the parameter passing conventions, have been completely redesigned and are significantly different from JNI.

More specifically, the goals of the KNI include the following:

- **Source-level portability of native code.** KNI is intended to make it possible to share source code of native functions so that the same native function source code can be used in multiple virtual machines without modifications (of course, native code that is operating system specific will have to be ported from one system to another.)
- **Isolation of virtual machine implementation details from the native functions.** KNI is intended to isolate the native function programmer from virtual machine implementation details. When writing new native functions, the programmer is not required to know anything about garbage collection details, object layout, class data structures, include file dependencies, or any other VM-specific implementation details.
- **Memory efficiency and minimal performance overhead.** KNI is intended to be significantly more efficient and compact than a full JNI implementation. Unlike in JNI, no temporary data structures or “marshalling” of parameter passing is required at the implementation level.

As mentioned above, KNI is a “*logical subset*” of JNI that follows the JNI conventions as much as it makes sense given the strict memory constraints, but with some considerable design and implementation differences. The list below summarizes the most significant design differences between the JNI and KNI:

1. **KNI is an implementation-level API.** KNI is an implementation-level API that is targeted primarily at VM implementers and device manufacturers who need to add new native functions to an existing J2ME virtual machine. Unlike JNI, the presence of the KNI is completely invisible to the Java programmer.
2. **No binary compatibility of native code.** KNI is intended to provide *source-level* compatibility of native code. Unlike JNI, KNI does not guarantee that binary libraries containing native code could be linked into different virtual machines without recompilation.
3. **No support for dynamically loaded native libraries.** KNI does not provide any mechanisms for dynamically loading native libraries into the virtual machine. Hence, any new functionality added via the native function mechanism must be done by modifying the VM *native function tables* and rebuilding from source. This is compliant with the security requirements imposed by the CLDC Specification.
4. **No Java-level access to arbitrary native functions.** Unlike JNI, KNI provides no Java-level APIs to invoke other native functions than those that have been pre-built into the Java virtual machine implementation. This is compliant with the security requirements imposed by the CLDC Specification.
5. **No class creation, object instantiation or Java method calling from native functions.** To greatly simplify the implementation of the native interface and to keep the implementation small and less error-prone, KNI does not provide any mechanisms for creating new Java classes, instantiating objects (other than strings), or calling Java methods from native code. When creating new objects that need to be manipulated in native code, the necessary objects must be created at the Java level and passed to native code as parameters.



6. **Parameter passing conventions are different from JNI.** To reduce implementation overhead, the parameter passing conventions of KNI are significantly different from JNI. KNI uses a “register-based” approach in which arguments can be read directly from the stack frame in an implementation-independent fashion. Unlike in the old KVM native interface, no explicit pushing or popping of parameters is required or allowed.



## JNI Scope

JNI is designed to be a logical subset of the Java Native Interface (JNI). The scope of JNI compared to JNI is presented in [TABLE 1](#) below. For a summary of the JNI Scope, refer to the *Java Native Interface Programmer's Guide and Specification*, pages 175–179.

**TABLE 1** JNI vs. JNI Scope

JNI Function Categories	Supported by JNI (YES/NO)
Version Information	YES
Class and Interface Operations	YES
Exceptions	YES
Global and Local References	NO
Object Operations	YES
Instance Field Access	YES
Static Field Access	YES
Instance Method Calls	NO
Static Method Calls	NO
String Operations	YES
Array Operations	YES
Native Method Registration	NO
Monitor Operations	NO
JavaVM Interface	NO
Reflection Support	NO

In addition, JNI introduces a set of new operations for method parameter (operand stack) access, and for manipulating *handles* to objects. These operations are JNI-specific and not available in the JNI.

Each of the supported function categories is introduced in more detail below. Note that the naming of these functions is different from JNI. Each function supported by the KNI uses a prefix “KNI\_” in front of the function name.

---

## 3.1 Version Information

`KNI_GetVersion` function returns the version of the KNI interface.

---

## 3.2 Class and Interface Operations

- `KNI_FindClass` returns a reference to a class or interface type of a given name.
- `KNI_GetSuperClass` function returns the superclass of a given class or interface.
- `KNI_IsAssignableFrom` function checks if an instance of one class or interface can be assigned to an instance of another class or interface. This function is useful for runtime type checking.

---

## 3.3 Exceptions

- `KNI_ThrowNew` function raises an exception in the current thread.
- `KNI_FatalError` function prints a descriptive message and terminates the current virtual machine instance.

---

## 3.4 Object Operations

- `KNI_GetObjectClass` function returns the class of a given instance.
- `KNI_IsInstanceOf` function checks whether an object is an instance of a given class or interface.

---

## 3.5 Instance Field Access

- `KNI_GetFieldID` function performs a symbolic lookup on a given class and returns the field ID of a named instance field.
- `KNI_Get<Type>Field` and `KNI_Set<Type>Field` functions access the instance fields of primitive or reference types.

---

## 3.6 Static Field Access

- `KNI_GetStaticFieldID` function performs a symbolic lookup on a given class and returns the field ID of a named static field.
- `KNI_GetStatic<Type>Field` and `KNI_SetStatic<Type>Field` functions access the static fields of primitive or reference types.

---

## 3.7 String Operations

- `KNI_GetStringLength` function returns the number of Unicode characters in a string represented by a `java.lang.String` object.
- `KNI_GetStringRegion` function provides access to the content of a `java.lang.String` object.
- `KNI_NewString` function creates a `java.lang.String` object from the given Unicode string.
- `KNI_NewStringUTF` function creates a `java.lang.String` object from the given UTF-8 string.

---

## 3.8 Array Operations

- `KNI_GetArrayLength` function returns the number of elements in an array.
- `KNI_Get<Type>ArrayElement` and `KNI_Set<Type>ArrayElement` functions allow a native method to access arrays elements of primitive or reference types.
- `KNI_GetRawArrayRegion` and `KNI_SetRawArrayRegion` functions copy multiple elements in or out of arrays of primitive types.

---

## 3.9 Parameter (Operand Stack) Access

- `KNI_GetParameterAs<Type>` functions allow a native method to access the method parameters (local variables) in the operand stack.
- `KNI_GetThisPointer` function allows a non-static native method to access the 'this' pointer in the current stack frame.
- `KNI_GetClassPointer` function allows a static native method to access the class pointer.
- `KNI_Return<Type>` functions allow a native method to return a value of a primitive type.

---

## 3.10 Handle Operations

- `KNI_StartHandles` function allocates space for a specified number of handles that can be used for holding object pointers inside a native function.
- `KNI_DeclareHandle` function defines a local handle that can hold an object pointer inside a native function.
- `KNI_IsNullHandle` function checks if a handle is null, i.e., not pointing to any object at the time.
- `KNI_IsSameObject` function checks if two handles refer to the same object.
- `KNI_ReleaseHandle` function sets the value of a handle to null.
- `KNI_EndHandles` function undeclares and deallocates the handles that were allocated earlier by the matching `KNI_StartHandles` call.
- `KNI_EndHandlesAndReturnObject` function undeclares and deallocates the handles that were allocated earlier by the matching `KNI_StartHandles` call, and returns a value of a reference type.

## KNI Data Types

---

This chapter specifies the native data types supported by KNI. The data types are defined in the `kni.h` header file.

---

### 4.1 Primitive and Reference Types

KNI defines a set of C/C++ types that correspond to the primitive and reference types in the Java programming language. These primitive and reference types are used for referring to data structures inside KNI functions.

In addition, KNI defines special *return types* that must be used for specifying the return type of each native function that uses the KNI API.

To ensure maximum portability of native code, you shall not use any VM-specific data types in your native code that uses the KNI API.

#### 4.1.1 Primitive Types

[TABLE 2](#) describes the primitive types in the Java programming language and the corresponding types in the KNI. Just like their counterparts in the Java programming language, all the primitive types in the KNI have well-defined sizes.

**TABLE 2** The KNI primitive types

Java language type	KNI primitive type	Description
<code>boolean</code>	<code>jboolean</code>	unsigned 8 bits
<code>byte</code>	<code>jbyte</code>	signed 8 bits
<code>char</code>	<code>jchar</code>	unsigned 16 bits
<code>short</code>	<code>jshort</code>	signed 16 bits
<code>int</code>	<code>jint</code>	signed 32 bits

**TABLE 2** The KNI primitive types

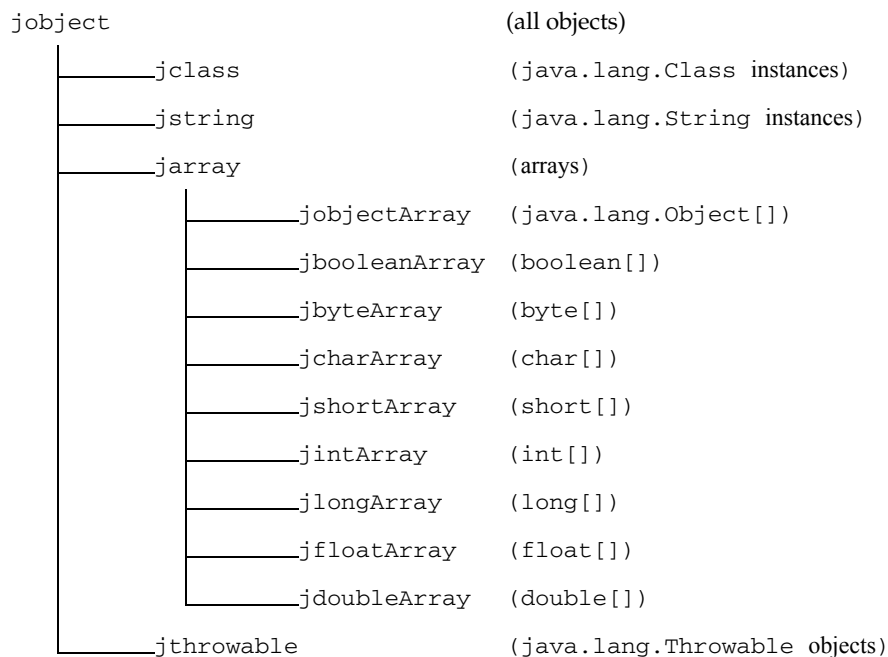
Java language type	KNI primitive type	Description
long	jlong	signed 64 bits
float	jfloat	32 bits
double	jdouble	64 bits

The `jsize` integer type is used to describe cardinal indices and sizes; this is accomplished via the C/C++ typedef mechanism:

```
typedef jint jsize;
```

## 4.1.2 Reference Types

KNI includes a number of reference types that correspond to the different kinds of reference types in the Java programming language. The KNI reference types are organized in the type hierarchy shown below.



**FIGURE 1** KNI reference types

When used in conjunction with *The C Programming Language*, all other KNI reference types are defined to be the same as type `jobject` by the use of the C typedef mechanism. For example:



```
typedef struct _jobject* jobject;
typedef jobject jclass;
```

Note that `jobject` is intended to be an *opaque* type. That is, the KNI programmer may not rely on the internal structure of this type, as its definition may vary from one virtual machine implementation to another.

### 4.1.3 Return Types

KNI defines special *return types* that must be used when defining the return type for each native function that uses the KNI. The following return types are supported.

**TABLE 3** Native function return types

Java language type	KNI type	Corresponding KNI return type
<code>void</code>	<n/a>	<code>KNI_RETURNTYPE_VOID</code>
<code>boolean</code>	<code>jboolean</code>	<code>KNI_RETURNTYPE_BOOLEAN</code>
<code>byte</code>	<code>jbyte</code>	<code>KNI_RETURNTYPE_BYTE</code>
<code>char</code>	<code>jchar</code>	<code>KNI_RETURNTYPE_CHAR</code>
<code>short</code>	<code>jshort</code>	<code>KNI_RETURNTYPE_SHORT</code>
<code>int</code>	<code>jint</code>	<code>KNI_RETURNTYPE_INT</code>
<code>long</code>	<code>jlong</code>	<code>KNI_RETURNTYPE_LONG</code>
<code>float</code>	<code>jfloat</code>	<code>KNI_RETURNTYPE_FLOAT</code>
<code>double</code>	<code>jdouble</code>	<code>KNI_RETURNTYPE_DOUBLE</code>
<object ref>	<code>jobject</code>	<code>KNI_RETURNTYPE_OBJECT</code>

The presence of the return types allows the virtual machine implementing the K Native Interface to use different implementation techniques for returning values back from native functions. For example, in the KVM implementation of the KNI, the actual C data type of the KNI return types is ‘void’, whereas other virtual machines may need to map the return types to applicable VM-specific data types.

## 4.2 Field IDs

Field IDs are regular C pointer types:

```
struct _jfieldID; /* opaque C structure */
typedef struct _jfieldID* jfieldID; /* field ID */
```

Note that `fieldID` is intended to be an *opaque* type. That is, the KNI programmer may not rely on the internal structure of this type, as its definition may vary from one virtual machine implementation to another.

---

## 4.3 String Formats

The KNI uses UTF-8 strings in the `const char*` format for reading class names and field names that are provided as parameters to certain KNI functions. These UTF-8 strings are converted to Unicode strings inside the KNI implementation as necessary. Some KNI operations read and return data in buffers that are assumed to be large enough for Unicode strings (each Unicode character is 16 bits wide.)

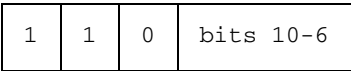
### 4.3.1 UTF-8 Strings

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 characters of up to 16 bits can be represented. All characters in the range `'\u0001'` to `'\u007f'` are represented by a single byte, as follows:

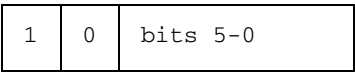


The seven bits of data in the byte give the value of the character that is represented. The null character (`'\u0000'`) and characters in the range `'\u0080'` to `'\u07ff'` are represented by a pair of bytes, `x` and `y`, as follows:

`x`:



`y`:



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

Characters in the range `'\u0800'` to `'\uffff'` are represented by three bytes, `x`, `y`, and `z`, as follows:

x:

1	1	1	0	bits 15-12
---	---	---	---	------------

y:

1	0	bits 11-6
---	---	-----------

z:

1		bits 5-0
---	--	----------

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

There are two differences between this format and the standard UTF-8 format. First, the null byte (byte) 0 is encoded using the two-byte format rather than the one-byte format. This means that KNI UTF-8 strings never have embedded nulls. Second, only the one-byte, two-byte, and three-byte formats are used. KNI does not recognize the longer UTF-8 formats.

## 4.3.2 Class Descriptors

A class descriptor represents the name of a class or an interface. It can be derived from a fully qualified class or interface name as defined in The Java Language Specification by substituting the "." character with the "/" character. For instance, the class descriptor for `java.lang.String` is:

`"java/lang/String"`

Array classes are formed using the "[" character followed by the field descriptor of the element type. The class descriptor for `"int[]"` is:

`"[I"`

and the class descriptor for `"double[][][]"` is:

`"[[[D"`

### 4.3.3 Field Descriptors

Table [TABLE 4](#) shows the field descriptors for the primitive types exported by the KNI.

**TABLE 4** The field descriptors for the KNI primitive types

Field Descriptor	The Java language type
Z	boolean
B	byte
C	char
S	short
I	int
J	long
F	float
D	double

The field descriptors of reference types begin with the “L” character, followed by the class descriptor, and terminated by the “;” character. Field descriptors of array types are formed following the same rule as class descriptors of array classes. [TABLE 5](#) contains examples of field descriptors for reference types and their Java programming language counterparts.

**TABLE 5** Examples of field descriptors for reference types

Field Descriptor	The Java language type
“Ljava/lang/String;”	String
“[I”	int[]
“[Ljava/lang/Object;”	java.lang.Object[]

---

## 4.4 Constants

File ‘kni.h’ defines certain constants and macros that are commonly used in native functions. These include the following:

- KNIEXPORT
- KNI\_FALSE
- KNI\_TRUE
- KNI\_OK
- KNI\_ERROR
- KNI\_VERSION

KNIEXPORT is a macro used to specify the calling and linkage conventions of both KNI functions and native method implementations. The programmer is recommended to place the KNIEXPORT macro before the function return type. For example:

```
KNIEXPORT void Java_mypackage_Cls_f();
```

is the prototype for a C function that implements method `f` of class `Cls` in package `mypackage`.

KNI\_FALSE and KNI\_TRUE are constants defined for the `jboolean` type:

```
#define KNI_FALSE 0
#define KNI_TRUE 1
```

KNI\_OK represents the successful return value of a few KNI functions, and KNI\_ERR is sometimes used to represent error conditions.

```
#define KNI_OK 0
#define KNI_ERR -1
```

KNI\_VERSION represents the KNI version number.

```
#define KNI_VERSION 0x00010000 /* KNI version 1.0 */
```



## KNI Functions

---

This chapter specifies the KNI functions. For each KNI function, we provide the following information:

- function prototype,
- a detailed description, including parameters, return values, and potential exceptions.

Note that since KNI is an implementation-level interface, the interface places some important restrictions and expectations on the programmer.

We use the adverb *must* to describe restrictions and conventions that the KNI programmer is expected to adhere to. For instance, when some KNI function *must* receive a non-null object as a parameter, it is the programmer's responsibility not to pass a null value to that function. Programmer-level restrictions like this allow the KNI implementation to be more efficient, as the KNI implementation does not check for null pointers or other similar conditions. In general, KNI is a "shoot yourself in the foot" interface that performs only a minimum number of checks at runtime to ensure the validity of the parameters.

---

## 5.1 Version Information

### 5.1.1 KNI\_GetVersion

**TABLE 6** KNI\_GetVersion

Function	Description
Prototype	<code>jint KNI_GetVersion();</code>
Description	Returns the version number of the KNI interface.
Parameters	None.
Return Values	Returns the version number of the KNI interface. Hexadecimal number "0x00010000" identifies KNI version 1.0.
Exceptions	None.



---

## 5.2 Class and Interface Operations

### 5.2.1 KNI\_FindClass

**TABLE 7** KNI\_FindClass

Function	Description
Prototype	<code>void KNI_FindClass(const char* name, jclass classHandle);</code>
Description	<p>Initializes a handle with a reference to the named class or interface. The function assumes that the class has already been loaded to the system and properly initialized. If this is not the case, or if no class is found, the handle will be initialized to NULL.</p> <p>The name argument is a class descriptor (See <a href="#">Section 4.3.2 “Class Descriptors”</a>). For example, the descriptor for the <code>java.lang.String</code> class is:</p> <p><code>“java/lang/String”</code></p> <p>The descriptor of the array class <code>java.lang.Object[]</code> is:</p> <p><code>“[Ljava/lang/Object;”</code>.</p>
Parameters	<p><code>name</code>: the descriptor of the class or interface to be returned. The name is represented as a UTF-8 string.</p> <p><code>classHandle</code>: a handle in which the class pointer will be returned.</p>
Return Values	Returns nothing directly, but returns the class pointer in the ‘classHandle’ handle. If the class cannot be found, the returned handle will be set to NULL.
Exceptions	None.

## 5.2.2 KNI\_GetSuperClass

**TABLE 8** KNI\_GetSuperClass

Function	Description
Prototype	<code>void KNI_GetSuperClass(jclass classHandle, jclass superclassHandle);</code>
Description	Initializes the <code>superclassHandle</code> handle to contain a pointer to the superclass of the given class (represented by <code>classHandle</code> ). If <code>classHandle</code> represents the class <code>java.lang.Object</code> , then the function sets the <code>superclassHandle</code> handle to <code>NULL</code> .
Parameters	<code>classHandle</code> : a handle initialized with a reference to the class object whose superclass is to be determined. <code>superclassHandle</code> : a handle which, upon return from this function, will contain a reference to the superclass object.
Return Values	Returns nothing directly, but parameter <code>superclassHandle</code> is used as a result value.
Exceptions	None.

## 5.2.3 KNI\_IsAssignableFrom

**TABLE 9** KNI\_IsAssignableFrom

Function	Description
Prototype	<code>jboolean KNI_IsAssignableFrom(jclass classHandle1, jclass classHandle2);</code>
Description	Determines whether an object of class or interface <code>classHandle1</code> can be safely cast to a class or interface <code>classHandle2</code> .
Parameters	<code>classHandle1</code> : handle initialized with a reference to the first class or interface argument <code>classHandle2</code> : handle initialized with a reference to the second class or interface argument
Return Values	Returns <code>KNI_TRUE</code> if any of the following is true:  The first and second argument refer to the same class or interface The first argument refers to a subclass of the second argument The first argument refers to a class that has the second argument as one of its interfaces The first and second arguments both refer to array of classes with element types <code>X</code> and <code>Y</code> , and <code>KNI_IsAssignableFrom(X, Y)</code> is <code>KNI_TRUE</code> ; otherwise it returns <code>KNI_FALSE</code>
Exceptions	None.

---

## 5.3 Exceptions

### 5.3.1 KNI\_ThrowNew

**TABLE 10** KNI\_ThrowNew

Function	Description
Prototype	<code>jint KNI_ThrowNew(const char* name, const char* message);</code>
Description	Instantiates an exception object with the message specified by <code>message</code> , and causes that exception to be thrown. A thrown exception will be pending in the current thread, but does not immediately disrupt native code execution. The actual exception will be thrown when the program control returns from the native function back to the virtual machine. Note that <code>KNI_ThrowNew</code> does not run the constructor on the new exception object.
Parameters	<code>name</code> : the name of a <code>java.lang.Throwable</code> class as a UTF-8 string. <code>message</code> : the message used to construct the <code>java.lang.Throwable</code> object; represented as a UTF-8 string.
Return Values	<code>KNI_OK</code> on success; otherwise <code>KNI_ERR</code> .
Exceptions	None.

---

**Note** – Since KNI does not allow Java code to be run from native methods, `KNI_ThrowNew` does not run the constructor of the exception object when it creates a new exception object. If you use `KNI_ThrowNew` to throw exceptions that have other fields than those defined in class `java.lang.Throwable`, the additional fields will not be initialized.

---

## 5.3.2 KNI\_FatalError

**TABLE 11** KNI\_FatalError

Function	Description
Prototype	<code>void KNI_FatalError(const char* message);</code>
Description	Prints an error message to the system's standard error output, and terminates the execution of the virtual machine immediately.
Parameters	message: the error message as a UTF-8 string.
Return Values	This function does not return.
Exceptions	None.

---

## 5.4 Object Operations

### 5.4.1 KNI\_GetObjectClass

**TABLE 12** KNI\_GetObjectClass

Function	Description
Prototype	<code>void KNI_GetObjectClass(jobject objectHandle, jclass classHandle);</code>
Description	Sets the handle <code>classHandle</code> to point to the class of the object represented by <code>objectHandle</code> .
Parameters	<code>objectHandle</code> : a handle pointing to an object whose class is being sought. <code>classHandle</code> : a handle which, upon return of this function, will contain a reference to the class of this object.
Return Values	Returns nothing directly, but parameter <code>classHandle</code> is used as a return value.
Exceptions	None.

## 5.4.2 KNI\_IsInstanceOf

**TABLE 13** KNI\_IsInstanceOf

Function	Description
Prototype	<code>jboolean KNI_IsInstanceOf(jobject objectHandle, jclass classHandle);</code>
Description	Tests whether an object represented by <code>objectHandle</code> is an instance of a class or interface represented by <code>classHandle</code> .
Parameters	<code>objectHandle</code> : a handle pointing to an object. <code>clazz</code> : a handle pointing to a class or interface.
Return Values	Returns <code>KNI_TRUE</code> if the given object is an instance of given class, <code>KNI_FALSE</code> otherwise.
Exceptions	None.

---

## 5.5 Instance Field Access

### 5.5.1 KNI\_GetFieldID

**TABLE 14** KNI\_GetFieldID

Function	Description
Prototype	<code>jfieldID KNI_GetFieldID(jclass classHandle, const char* name, const char* signature);</code>
Description	Returns the field ID of an instance field of the class represented by <code>classHandle</code> . The field is specified by its name and signature. The <code>Get&lt;Type&gt;Field</code> and <code>Set&lt;Type&gt;Field</code> families of accessor functions use field IDs to retrieve the value of instance fields. The field must be accessible from the class referred to by <code>classHandle</code> .
Parameters	<code>classHandle</code> : a handle pointing to a class whose field ID will be retrieved. <code>name</code> : the field name as a UTF-8 string. <code>signature</code> : the field descriptor as a UTF-8 string.
Return Values	Returns a field ID or <code>NULL</code> if the lookup fails for any reason.
Exceptions	None.

## 5.5.2 KNI\_Get<Type>Field

**TABLE 15** KNI\_Get<Type>Field

Function	Description
Prototype	<code>&lt;NativeType&gt; KNI_Get&lt;Type&gt;Field(jobject objectHandle, jfieldID fieldID);</code>
Description	Returns the value of an instance field of a primitive type. The field to access is denoted by <code>fieldID</code> .
Parameters	<code>objectHandle</code> : a handle pointing to an object whose field is to be accessed. <code>fieldID</code> : the field ID of the given instance field.
Return Values	Returns the value of a primitive instance field.
Exceptions	None.

This family of functions consists of eight members that are listed in the table below. Note that those functions that manipulate `jfloat` or `jdouble` data types are assumed to be available only if the underlying virtual machine or J2ME configuration supports floating point data types.

**TABLE 16** KNI\_Get<Type>Field Functions

KNI_Get<Type>Field	<NativeType>
<code>KNI_GetBooleanField</code>	<code>jboolean</code>
<code>KNI_GetByteField</code>	<code>jbyte</code>
<code>KNI_GetCharField</code>	<code>jchar</code>
<code>KNI_GetShortField</code>	<code>jshort</code>
<code>KNI_GetIntField</code>	<code>jint</code>
<code>KNI_GetLongField</code>	<code>jlong</code>
<code>KNI_GetFloatField</code>	<code>jfloat</code>
<code>KNI_GetDoubleField</code>	<code>jdouble</code>

### 5.5.3 KNI\_Set<Type>Field

**TABLE 17** KNI\_Set<Type>Field

Function	Description
Prototype	<code>void KNI_Set&lt;Type&gt;Field(jobject objectHandle, jfieldID fieldID, &lt;NativeType&gt; value);</code>
Description	Sets the value of an instance field of a primitive type. The field to modify is denoted by <code>fieldID</code> .
Parameters	<code>objectHandle</code> : a handle pointing to an object whose field is to be modified. <code>fieldID</code> : the field ID of the given instance field. <code>value</code> : the value to set to the instance field.
Return Values	<code>void</code> .
Exceptions	None.

This family of functions consists of nine members that are listed in the table below. Note that those functions that manipulate `jfloat` or `jdouble` data types are assumed to be available only if the underlying virtual machine or J2ME configuration supports floating point data types.

**TABLE 18** KNI\_Set<Type>Field Functions

KNI_Set<Type>Field	<NativeType>
<code>KNI_SetBooleanField</code>	<code>jboolean</code>
<code>KNI_SetByteField</code>	<code>jbyte</code>
<code>KNI_SetCharField</code>	<code>jchar</code>
<code>KNI_SetShortField</code>	<code>jshort</code>
<code>KNI_SetIntField</code>	<code>jint</code>
<code>KNI_SetLongField</code>	<code>jlong</code>
<code>KNI_SetFloatField</code>	<code>jfloat</code>
<code>KNI_SetDoubleField</code>	<code>jdouble</code>



## 5.5.4 KNI\_GetObjectField

**TABLE 19** KNI\_GetObjectField

Function	Description
Prototype	<code>void KNI_GetObjectField(jobject objectHandle, jfieldID fieldID, jobject toHandle);</code>
Description	Returns the value of a field of a reference type, and assigns it to the handle <code>toHandle</code> . The field to access is denoted by <code>fieldID</code> .
Parameters	<code>objectHandle</code> : a handle pointing to an object whose whose field is to be accessed. <code>fieldID</code> : the field ID of the given instance field. <code>toHandle</code> : a handle to which the return value will be assigned.
Return Values	Returns nothing directly, but handle <code>toHandle</code> will contain the return value.
Exceptions	None.

## 5.5.5 KNI\_SetObjectField

**TABLE 20** KNI\_SetObjectField

Function	Description
Prototype	<code>void KNI_SetObjectField(jobject objectHandle, jfieldID fieldID, jobject fromHandle);</code>
Description	Sets the value of an instance field of a reference type. The field to modify is denoted by <code>fieldID</code> .
Parameters	<code>objectHandle</code> : a handle pointing to an object whose field is to be modified. <code>fieldID</code> : the field ID of the given instance field. <code>fromHandle</code> : a handle pointing to an object that will be assigned to the field.
Return Values	<code>void</code> .
Exceptions	None.

---

## 5.6 Static Field Access

### 5.6.1 KNI\_GetStaticFieldID

**TABLE 21** KNI\_GetStaticField

Function	Description
Prototype	<code>jfieldID KNI_GetStaticFieldID(jclass classHandle, const char* name, const char* signature);</code>
Description	Returns the field ID of a static field of the class represented by <code>classHandle</code> . The field is specified by its name and signature. The <code>GetStatic&lt;Type&gt;Field</code> and <code>SetStatic&lt;Type&gt;Field</code> families of accessor functions use field IDs to retrieve the value of static fields. The field must be accessible from the class referred to by <code>classHandle</code> .
Parameters	<code>classHandle</code> : a handle pointing to a class whose field ID will be retrieved. <code>name</code> : the field name as a UTF-8 string <code>signature</code> : the field descriptor as a UTF-8 string.
Return Values	Returns a field ID or NULL if the lookup fails for any reason.
Exceptions	None.

## 5.6.2 KNI\_GetStatic<Type>Field

**TABLE 22** KNI\_GetStatic<Type>Field

Function	Description
Prototype	<NativeType> KNI_GetStatic<Type>Field(jclass classHandle, jfield fieldID);
Description	Returns the value of a static field of a primitive type. The field to access is denoted by fieldID.
Parameters	classHandle: a handle pointing to a class or interface whose static field is to be accessed. fieldID: a field ID of the given class.
Return Values	Returns the value of a static field of a primitive type.
Exceptions	None.

This family of functions consists of eight members that are listed in the table below. Note that those functions that manipulate jfloat or jdouble data types are assumed to be available only if the underlying virtual machine or J2ME configuration supports floating point data types:

**TABLE 23** KNI\_GetStatic<Type>Field Functions

KNI_GetStatic<Type>Field	<NativeType>
KNI_GetStaticBooleanField	jboolean
KNI_GetStaticByteField	jbyte
KNI_GetStaticCharField	jchar
KNI_GetStaticShortField	jshort
KNI_GetStaticIntField	jint
KNI_GetStaticLongField	jlong
KNI_GetStaticFloatField	jfloat
KNI_GetStaticDoubleField	jdouble

## 5.6.3 KNI\_SetStatic<Type>Field

**TABLE 24** KNI\_SetStatic<Type>Field

Function	Description
Prototype	<code>void KNI_SetStatic&lt;Type&gt;Field(jclass classHandle, jfieldID fieldID, &lt;NativeType&gt; value);</code>
Description	Sets the value of a static field of a primitive type. The field to modify is denoted by <code>fieldID</code> .
Parameters	<code>classHandle</code> : a handle pointing to the class or interface whose static field is to be modified. <code>fieldID</code> : a field ID of the given class. <code>value</code> : the value to set to the static field.
Return Values	<code>void</code> .
Exceptions	None.

This family of functions consists of eight members that are listed in the table below. Note that those functions that manipulate `jfloat` or `jdouble` data types are assumed to be available only if the underlying virtual machine or J2ME configuration supports floating point data types:

**TABLE 25** KNI\_SetStatic<Type>Field Functions

KNI_SetStatic<Type>Field	<NativeType>
<code>KNI_SetStaticBooleanField</code>	<code>jboolean</code>
<code>KNI_SetStaticByteField</code>	<code>jbyte</code>
<code>KNI_SetStaticCharField</code>	<code>jchar</code>
<code>KNI_SetStaticShortField</code>	<code>jshort</code>
<code>KNI_SetStaticIntField</code>	<code>jint</code>
<code>KNI_SetStaticLongField</code>	<code>jlong</code>
<code>KNI_SetStaticFloatField</code>	<code>jfloat</code>
<code>KNI_SetStaticDoubleField</code>	<code>jdouble</code>

## 5.6.4 KNI\_GetStaticObjectField

**TABLE 26** KNI\_GetStaticObjectField

Function	Description
Prototype	<code>void KNI_GetStaticObjectField(jclass classHandle, jfield fieldID, jobject toHandle);</code>
Description	Returns the value of a static field of a reference type, and assigns it to the handle <code>toHandle</code> . The field to access is denoted by <code>fieldID</code> .
Parameters	<code>classHandle</code> : a handle pointing to a class or interface whose static field is to be accessed. <code>fieldID</code> : a field ID of the given class. <code>toHandle</code> : a handle to which the return value will be assigned.
Return Values	Nothing directly, but handle <code>toHandle</code> will contain the return value.
Exceptions	None.

## 5.6.5 KNI\_SetStaticObjectField

**TABLE 27** KNI\_SetStaticObjectField

Function	Description
Prototype	<code>void KNI_SetStaticObjectField(jclass classHandle, jfield fieldID, jobject fromHandle);</code>
Description	Sets the value of a static field of a reference type. The field to modify is denoted by <code>fieldID</code> .
Parameters	<code>classHandle</code> : a handle pointing to the class or interface whose static field is to be modified. <code>fieldID</code> : a field ID of the given class. <code>fromHandle</code> : a handle pointing to an object that will be assigned to the field.
Return Values	<code>void</code> .
Exceptions	None.

---

## 5.7 String Operations

### 5.7.1 KNI\_GetStringLength

**TABLE 28** KNI\_GetStringLength

Function	Description
Prototype	<code>jsize KNI_GetStringLength(jstring stringHandle);</code>
Description	Returns the number of Unicode characters in a <code>java.lang.String</code> object. If the given string handle is <code>NULL</code> , the function returns -1.
Parameters	<code>stringHandle</code> : a handle pointing to a <code>java.lang.String</code> string object whose length is to be determined.
Return Values	Returns the length of the string, or -1 if the given string handle is <code>NULL</code> .
Exceptions	None.

## 5.7.2 KNI\_GetStringRegion

**TABLE 29** KNI\_GetStringRegion

Function	Description
Prototype	<code>void KNI_GetStringRegion(jstring stringHandle, jsize offset, jsize n, jchar* jcharbuf);</code>
Description	Copies a number of Unicode characters from a <code>java.lang.String</code> object (denoted by <code>stringHandle</code> ), beginning at <code>offset</code> , to the given buffer <code>jcharbuf</code> . No range checking is performed. It is the responsibility of the native function programmer to allocate the return buffer, and to make sure that it is large enough.
Parameters	<code>stringHandle</code> : a handle pointing to a <code>java.lang.String</code> object whose contents are to be copied. <code>offset</code> : the offset within the string object at which to start copying (offset 0: first character in the Java string.) <code>n</code> : the number of 16-bit Unicode characters to copy. <code>jcharbuf</code> : a pointer to a buffer to hold the Unicode characters.
Return Values	Data is returned in user-allocated buffer <code>jcharbuf</code> . Each character in the returned buffer is 16 bits wide.
Exceptions	None.

## 5.7.3 KNI\_NewString

**TABLE 30** KNI\_NewString

Function	Description
Prototype	<code>void KNI_NewString(const jchar* uchars, jsize length, jstring stringHandle);</code>
Description	Creates a <code>java.lang.String</code> object from the given Unicode sequence. The resulting object is returned to the caller via the <code>stringHandle</code> handle.
Parameters	<code>uchars</code> : a Unicode sequence that will make up the contents of the new string object. <code>stringHandle</code> : a handle to hold the reference to the new <code>java.lang.String</code> object..
Return Values	Returns nothing directly, but handle <code>stringHandle</code> will contain a reference to the new string object.
Exceptions	<code>OutOfMemoryError</code> if the virtual machine runs out of memory.

## 5.7.4 KNI\_NewStringUTF

**TABLE 31** KNI\_NewStringUTF

Function	Description
Prototype	<code>void KNI_NewStringUTF(const char* utf8chars, jstring stringHandle);</code>
Description	Creates a <code>java.lang.String</code> object from the given UTF-8 string. The resulting object is returned to the caller via the <code>stringHandle</code> handle.
Parameters	<code>utf8chars</code> : a UTF-8 string that will make up the contents of the new string object. <code>length</code> : length of the Unicode string. <code>stringHandle</code> : a handle to hold the reference to the new <code>java.lang.String</code> object..
Return Values	Returns nothing directly, but handle <code>stringHandle</code> will contain a reference to the new string object.
Exceptions	<code>OutOfMemoryError</code> if the virtual machine runs out of memory.



---

## 5.8 Array Operations

### 5.8.1 KNI\_GetArrayLength

**TABLE 32** KNI\_GetArrayLength

Function	Description
Prototype	<code>jsize KNI_GetArrayLength(jarray arrayHandle);</code>
Description	Returns the number of elements in a given Java array object. The <code>array</code> argument may denote an array of any element type, including primitive types or reference types. If the given array handle is NULL, the function returns -1.
Parameters	<code>arrayHandle</code> : a handle pointing to an array object whose length is to be determined.
Return Values	Returns the number of elements in the array, or -1 if the given array handle is NULL.
Exceptions	None.

### 5.8.2 KNI\_Get<Type>ArrayElement

**TABLE 33** KNI\_Get<Type>ArrayElement

Function	Description
Prototype	<code>&lt;NativeType&gt; KNI_Get&lt;Type&gt;ArrayElement(&lt;ArrayType&gt; arrayHandle, jint index);</code>
Description	Returns an element of an array of <Type> type. The given array reference must not be NULL. No array type checking or index range checking is performed.
Parameters	<code>arrayHandle</code> : a handle pointing to an array object. <code>index</code> : the index of the element in the array. Index 0 denotes the first element in the array.
Return Values	Returns the element at position <code>index</code> of an array of <Type> type.
Exceptions	None.

This family of functions consists of eight members that are listed in the table below. Note that those functions that manipulate `jfloat` or `jdouble` data types are

assumed to be available only if the underlying virtual machine or J2ME configuration supports floating point data types.

**TABLE 34** KNI\_Get<Type>ArrayElement Functions

<b>KNI_Get&lt;Type&gt;ArrayElement</b>	<b>&lt;ArrayType&gt;</b>	<b>&lt;NativeType&gt;</b>
KNI_GetBooleanArrayElement	jbooleanArray	jboolean
KNI_GetByteArrayElement	jbyteArray	jbyte
KNI_GetCharArrayElement	jcharArray	jchar
KNI_GetShortArrayElement	jshortArray	jshort
KNI_GetIntArrayElement	jintArray	jint
KNI_GetLongArrayElement	jlongArray	jlong
KNI_GetFloatArrayElement	jfloatArray	jfloat
KNI_GetDoubleArrayElement	jdoubleArray	jdouble

## 5.8.3 KNI\_Set<Type>ArrayElement

**TABLE 35** KNI\_Set<Type>ArrayElement

Function	Description
Prototype	<code>void KNI_Set&lt;Type&gt;ArrayElement (&lt;ArrayType&gt; arrayHandle, jint index, &lt;NativeType&gt; value);</code>
Description	Sets an element of an array of <Type> type. The given array handle must not be NULL. No array type checking or index range checking is performed.
Parameters	arrayHandle: a handle pointing to an array object. index: the index of the element in the array. Index 0 denotes the first element in the array. value: the value that will be stored in the array.
Return Values	void.
Exceptions	None.

This family of functions consists of eight members that are listed in the table below. Note that those functions that manipulate `jfloat` or `jdouble` data types are assumed to be available only if the underlying virtual machine or J2ME configuration supports floating point data types.

**TABLE 36** KNI\_Set<Type>ArrayElement Functions

KNI_Set<Type>ArrayElement	<ArrayType>	<NativeType>
KNI_SetBooleanArrayElement	<code>jbooleanArray</code>	<code>jboolean</code>
KNI_SetByteArrayElement	<code>jbyteArray</code>	<code>jbyte</code>
KNI_SetCharArrayElement	<code>jcharArray</code>	<code>jchar</code>
KNI_SetShortArrayElement	<code>jshortArray</code>	<code>jshort</code>
KNI_SetIntArrayElement	<code>jintArray</code>	<code>jint</code>
KNI_SetLongArrayElement	<code>jlongArray</code>	<code>jlong</code>
KNI_SetFloatArrayElement	<code>jfloatArray</code>	<code>jfloat</code>
KNI_SetDoubleArrayElement	<code>jdoubleArray</code>	<code>jdouble</code>

## 5.8.4 KNI\_GetObjectArrayElement

**TABLE 37** KNI\_GetObjectArrayElement

Function	Description
Prototype	<code>void KNI_GetObjectArrayElement(jobjectArray arrayHandle, jint index, jobject toHandle);</code>
Description	Returns an element of an array of a reference type. The given array handle must not be NULL. No array type checking or index range checking is performed.
Parameters	arrayHandle: a handle pointing to an array object. index: the index of the element in the array. Index 0 denotes the first element in the array. toHandle: a handle to which the return value will be assigned.
Return Values	Returns nothing directly, but handle toHandle will contain the return value.
Exceptions	None.

## 5.8.5 KNI\_SetObjectArrayElement

**TABLE 38** KNI\_SetObjectArrayElement

Function	Description
Prototype	<code>void KNI_SetObjectArrayElement(jobjectArray arrayHandle, jint index, jobject fromHandle);</code>
Description	Sets an element of an array of a reference type. The given array handle must not be NULL. No array type checking or index range checking is performed.
Parameters	arrayHandle: a handle pointing to an array object. index: the index of the element in the array. Index 0 denotes the first element in the array. fromHandle: a handle from which value will be read.
Return Values	void.
Exceptions	None.

## 5.8.6 KNI\_GetRawArrayRegion

**TABLE 39** KNI\_GetRawArrayRegion

Function	Description
Prototype	<code>void KNI_GetRawArrayRegion(jarray arrayHandle, jsize offset, jsize n, jbyte* dstBuffer);</code>
Description	Gets a region of <i>n</i> bytes of an array of a primitive type. The given array handle must not be NULL. No array type checking or range checking is performed.
Parameters	arrayHandle: a handle initialized with the array reference. offset: a byte offset within the array. n: the number of bytes of raw data to get. dstBuffer: the destination of the data as bytes.
Return Values	void.
Exceptions	None.

## 5.8.7 KNI\_SetRawArrayRegion

**TABLE 40** KNI\_SetRawArrayRegion

Function	Description
Prototype	<code>void KNI_SetRawArrayRegion(jarray arrayHandle, jsize offset, jsize n, const jbyte* srcBuffer);</code>
Description	Sets a region of <i>n</i> bytes of an array of a primitive type. The given array handle must not be NULL. No array type checking or range checking is performed.
Parameters	arrayHandle: a handle initialized with the array reference. offset: a byte offset within the array. n: the number of bytes of raw data to change. srcBuffer: the source of the data as bytes.
Return Values	void.
Exceptions	None.

---

## 5.9 Parameter (Operand Stack) Access

KNI introduces a new set of functions for reading the parameters (local variables) that a Java method has passed on to a native method, as well as a set of functions for returning values back from native methods to Java methods. These functions are typically used in conjunction of the handle operations introduced in [Section 5.10 “Handle Operations.”](#)

### 5.9.1 KNI\_GetParameterAs<Type>

**TABLE 41** KNI\_GetParameterAs<Type>

Function	Description
Prototype	<code>&lt;ReturnType&gt; KNI_GetParameterAs&lt;Type&gt;(jint index);</code>
Description	Returns the value of a parameter of a primitive type <Type> at physical location specified by <code>index</code> . Parameter indices are mapped from left to right. Index value 1 refers to the leftmost parameter that has been passed on to the native method. Remember that parameters of type <code>long</code> or <code>double</code> take up two entries on the operand stack. No type checking or index range checking is performed.
Parameters	<code>index</code> : the index of a parameter of a primitive type to the native method. Index value 1 refers to the leftmost parameter in the Java method.
Return Values	Returns the argument of a primitive type at position <code>index</code> in the list of arguments to the native method.
Exceptions	None.

---

**Note** – It is important to note that parameters of type `long` or `double` take up two entries on the operand stack. For example, when calling the following function

```
native void foo(int a, long b, int c);
```

the index of parameter ‘a’ would be 1, index of parameter ‘b’ would be 2, and index of parameter ‘c’ would be 4.

---

This family of functions consists of eight members that are listed in the table below. Note that those functions that manipulate `jfloat` or `jdouble` data types are assumed to be available only if the underlying virtual machine or J2ME configuration supports floating point data types.

**TABLE 42** Support for accessing a native method actual parameters

<b>KNI_GetParameterAs&lt;Type&gt;</b>	<b>&lt;Type&gt;</b>	<b>&lt;ReturnType&gt;</b>
KNI_GetParameterAsBoolean	Boolean	jboolean
KNI_GetParameterAsByte	Byte	jbyte
KNI_GetParameterAsChar	Char	jchar
KNI_GetParameterAsShort	Short	jshort
KNI_GetParameterAsInt	Int	jint
KNI_GetParameterAsLong	Long	jlong
KNI_GetParameterAsFloat	Float	jfloat
KNI_GetParameterAsDouble	Double	jdouble

## 5.9.2 KNI\_GetParameterAsObject

**TABLE 43** KNI\_GetParameterAsObject

<b>Function</b>	<b>Description</b>
Prototype	<code>void KNI_GetParameterAsObject(jint index, jobject toHandle);</code>
Description	Reads the value (object reference) of the parameter specified by <code>index</code> , and stores it in the handle <code>toHandle</code> . Parameter indices are mapped from left to right. Index value 1 refers to the leftmost parameter that has been passed on to the native method. No type checking or index range checking is performed.
Parameters	<code>index</code> : the index of a parameter of a reference type to the native method. Index value 1 refers to the leftmost parameter in the Java method. <code>toHandle</code> : a handle to to which the return value will be assigned.
Return Values	Returns nothing directly, but handle <code>toHandle</code> will contain the return value.
Exceptions	None.

### 5.9.3 KNI\_GetThisPointer

**TABLE 44** KNI\_GetThisPointer

Function	Description
Prototype	<code>void KNI_GetThisPointer(jobject toHandle);</code>
Description	Reads the value of the 'this' pointer in the current stack frame of an instance (non-static) native method, and stores the value in the handle <code>toHandle</code> .
Parameters	<code>toHandle</code> : a handle to contain the 'this' pointer.
Return Values	Returns nothing directly, but handle <code>toHandle</code> will contain the return value. The return value is unspecified if this method is called inside a static native method.
Exceptions	None.

---

**Note** – The function `KNI_GetThisPointer` should only be used only inside instance (non-static) native methods. The return value of this function is unspecified if the function is called from a static native method.

---



## 5.9.4 KNI\_GetClassPointer

**TABLE 45** KNI\_GetClassPointer

Function	Description
Prototype	<code>void KNI_GetClassPointer(jclass toHandle);</code>
Description	Reads the value of the class pointer in the current stack frame of a static native method, and stores the value in the handle <code>toHandle</code> .
Parameters	<code>toHandle</code> : a handle to contain the class pointer.
Return Values	Returns nothing directly, but handle <code>toHandle</code> will contain the return value.
Exceptions	None.

## 5.9.5 KNI\_ReturnVoid

**TABLE 46** KNI\_ReturnVoid

Function	Description
Prototype	<code>void KNI_ReturnVoid();</code>
Description	Returns a void value from a native function. Calling this function terminates the execution of the native function immediately.
Parameters	None.
Return Values	None
Exceptions	None.

**Note – IMPORTANT:** The `KNI_ReturnVoid` function **MUST BE CALLED** at the end of a native function when a native function does not want to return a value back to the calling Java function. Otherwise the operand stack of the Java virtual machine may become corrupted.

## 5.9.6 KNI\_Return<Type>

**TABLE 47** KNI\_Return<Type>

Function	Description
Prototype	<code>void KNI_Return&lt;Type&gt;(&lt;NativeType&gt; value);</code>
Description	Returns a primitive value from a native function. Calling any of the KNI_Return<Type> functions will terminate the execution of the native function immediately.
Parameters	value: the result value of the native method.
Return Values	Returns a primitive value back to the function that called the native function containing the KNI_Return<Type> call.
Exceptions	None.

**Note** – The semantics of the KNI\_Return<Type> functions are similar to the standard C/C++ return statement. The execution of the function is terminated immediately, and the control is returned back to the caller.

This family of functions consists of eight members that are listed in the table below. Note that those functions that manipulate jfloat or jdouble data types are assumed to be available only if the underlying virtual machine or J2ME configuration supports floating point data types.

**TABLE 48** Support for setting a native result value

KNI_Return<Type>	<Type>	<NativeType>
KNI_ReturnBoolean	Boolean	jboolean
KNI_ReturnByte	Byte	jbyte
KNI_ReturnChar	Char	jchar
KNI_ReturnShort	Short	jshort
KNI_ReturnInt	Int	jint
KNI_ReturnLong	Long	jlong
KNI_ReturnFloat	Float	jfloat
KNI_ReturnDouble	Double	jdouble

---

## 5.10 Handle Operations

To avoid garbage collection problems, all the object references used by KNI are *handles* to objects rather than direct object references. Many KNI functions (such as `KNI_GetParameterAsObject` and `KNI_GetObjectField`) use object handles as input or output parameters. It is the programmer's responsibility to allocate, declare and unallocate the handles that are needed inside a native function.

Seven handle functions are provided, as listed below.

---

**Note** – It is important to notice that the implementation of KNI handle operations may vary significantly from one virtual machine to another. The native function programmer should not assume anything about the internal structure or behavior of the handles. In order to write portable code, the programmer must only use the public KNI handle operations defined below.

---

### 5.10.1 KNI\_StartHandles

**TABLE 49** KNI\_StartHandles

Function	Description
Prototype	<code>void KNI_StartHandles(n)</code>
Description	Allocates enough space for <code>n</code> handles for the current native method.
Parameters	<code>n</code> : a positive integer specifying the number of handles to allocate.
Return Values	Nothing directly, but allocates space for a number of handles in an implementation-dependent fashion.
Exceptions	None.

---

**Note** – Every call to `KNI_StartHandles` must be matched with a corresponding `KNI_EndHandles` or `KNI_EndHandlesAndReturnObject` call. Nested `KNI_StartHandles` and `KNI_EndHandles/KNI_EndHandlesAndReturnObject` calls are not allowed inside the same C/C++ code block (`{ ... }`).

---

## 5.10.2 KNI\_DeclareHandle

**TABLE 50** KNI\_DeclareHandle

Function	Description
Prototype	<code>void KNI_DeclareHandle(handle)</code>
Description	Declares a handle that can be used as an input, output, or in/out parameter to an appropriate KNI function. The initial value of the handle is unspecified. If the programmer has not allocated enough space for the handle in the preceding call to <code>KNI_StartHandles</code> , the behavior of this function is unspecified.
Parameters	<code>handle</code> : the name by which the handle will be identified between the calls to <code>KNI_StartHandles</code> and <code>KNI_EndHandles</code> / <code>KNI_EndHandlesAndReturnObject</code> .
Return Values	Nothing directly, but declares a handle in an implementation-dependent fashion.
Exceptions	None.

## 5.10.3 KNI\_IsNullHandle

**TABLE 51** KNI\_IsNullHandle

Function	Description
Prototype	<code>jboolean KNI_IsNullHandle(handle)</code>
Description	Checks if the given handle is NULL.
Parameters	<code>handle</code> : the handle whose contents will be examined.
Return Values	<code>KNI_TRUE</code> if the handle is NULL, <code>KNI_FALSE</code> otherwise.
Exceptions	None.

## 5.10.4 KNI\_IsSameObject

**TABLE 52** KNI\_IsSameObject

Function	Description
Prototype	jboolean KNI_IsSameObject(handle1, handle2)
Description	Checks if the given two handles refer to the same object. No parameter type checking is performed.
Parameters	handle1: a handle pointing to an object. handle2: a handle pointing to an object.
Return Values	KNI_TRUE if the handles refer to the same object, KNI_FALSE otherwise.
Exceptions	None.

## 5.10.5 KNI\_ReleaseHandle

**TABLE 53** KNI\_ReleaseHandle

Function	Description
Prototype	void KNI_ReleaseHandle(handle)
Description	Sets the value of the given handle to NULL.
Parameters	handle: the handle whose value will be set to NULL.
Return Values	None.
Exceptions	None.

## 5.10.6 KNI\_EndHandles

**TABLE 54** KNI\_EndHandles

Function	Description
Prototype	<code>void KNI_EndHandles()</code>
Description	Undeclares and unallocates the handles defined after the preceding <code>KNI_StartHandles</code> call.
Parameters	None.
Return Values	None.
Exceptions	None.

---

**Note** – Every call to `KNI_StartHandles` must be matched with a corresponding `KNI_EndHandles` or `KNI_EndHandlesAndReturnObject` call. Nested `KNI_StartHandles` and `KNI_EndHandles/KNI_EndHandlesAndReturnObject` calls are not allowed inside the same C/C++ code block (`{ ... }`).

---

## 5.10.7 KNI\_EndHandlesAndReturnObject

**TABLE 55** KNI\_EndHandlesAndReturnObject

Function	Description
Prototype	<code>void KNI_EndHandlesAndReturnObject(jobject objectHandle);</code>
Description	Undeclares and unallocates the handles defined after the preceding <code>KNI_StartHandles</code> call. At the same time, this function will terminate the execution of the native function immediately and return an object reference back to the caller.
Parameters	<code>objectHandle</code> : the handle from which the result value will be read.
Return Values	Returns an object value back to the function that called the native function containing the <code>KNI_EndHandlesAndReturnObject</code> call.
Exceptions	None.

## KNI Programming Overview

---

---

### 6.1 The 'kni.h' Include File

As mentioned earlier in [Chapter 2](#), one of the key goals of the KNI is to isolate the native function programmer from the implementation details of the virtual machine. Consequently, the KNI programmer should never include any VM-specific files into the files that implement native functions.

The file 'kni.h' is a C header file that provides the declarations and definitions of all the data types exported by KNI as well as the prototypes of all the functions exported by KNI. When implementing native methods using KNI, this header file must always be included

---

### 6.2 Sample KNI Application

This section shows a simple example that illustrates the use of KNI. We write a small Java class that calls a native (C) input/output function to print a message to standard output. We highlight the necessary steps that are required when writing new native functions. More comprehensive examples illustrating the use of KNI are provided in [Chapter 7](#).

#### 6.2.1 Java Code

Below is a small Java application that defines a class named HelloWorld contained in a package called mypackage.

```
package mypackage;

public class HelloWorld {
```

```

    public native void sayHello();
    public static void main(String[] args) {
        new HelloWorld().sayHello();
    }
}

```

The `HelloWorld` class definition contains two method declarations: a native method called `sayHello` and a Java method called `main`. When the application is run, the `main` method creates an instance of the `HelloWorld` class and invokes the native method `sayHello` for this instance.

In this example, the native method `sayHello` is implemented in a separate C programming language source file illustrated in the next subsection.

## 6.2.2 The Corresponding Native Code

The function that implements the native method `sayHello` must follow the function prototype definition specified in the header file that would be generated on invocation of `javah` on the `mypackage.HelloWorld` file. In this case, since the `HelloWorld` application is contained in a package called `mypackage`, the name of the native function must be as follows:

```
void Java_mypackage_HelloWorld_sayHello()
```

The `mypackage.HelloWorld.sayHello` method is implemented in a C source file `'Java_mypackage_HelloWorld.c'` as follows:

```

#include <kni.h>
#include <stdio.h>

KNIEXPORT KNI_RETURNTYPE_VOID Java_mypackage_HelloWorld_sayHello()
{
    char* message = "hello, world!";
    fprintf(stdout, "%s\n", message);
    KNI_ReturnVoid();
}

```

In this case, the implementation of the native function is very simple. It uses the standard C input/output function `fprintf` to display the message "hello, world!".

The C source file includes three header files:

- `kni.h` – This C header file provides the declarations and definitions of all the data types exported by KNI as well as the prototypes of all the functions exported by KNI. When implementing native methods, this C header file must always be included.
- `stdio.h` – The code snippet above includes `stdio.h` because it uses the standard C input/output function `fprintf`.



## 6.2.3 Compiling and Running the Sample Application in the KVM

---

**Note** – The information provided in this subsection is KVM-specific and not part of the KNI Specification. Implementation details for other Java virtual machines supporting the K Native Interface may be entirely different.

---

Below is a summary of the steps to run the sample KNI application in the KVM:

1. Create the Java class source file `HelloWorld.java` shown above. In the simplest case, the Java class source file should reside in `'${KVM_ROOT}/api/src/mypackage/HelloWorld.java'` so that the `JavaCodeCompact` tool can find, compile and romize the class automatically as part of the build process.
2. Create a C source file `Java_mypackage_HelloWorld.c` that implements the native method defined above. In this example, the C source file will reside in `'${KVM_ROOT}/kvm/VmUnix/src/Java_mypackage_HelloWorld.c'`.
3. Now that the native method has been implemented, we need to rebuild the KVM runtime interpreter and update its native function table (e.g., `${KVM_ROOT}/tools/jcc/nativeFunctionTableUnix.c`); this latter step is performed by `JavaCodeCompact`. Rebuild the KVM system by compiling it with the following compilation option:

```
gnumake USE_KNI=true
```

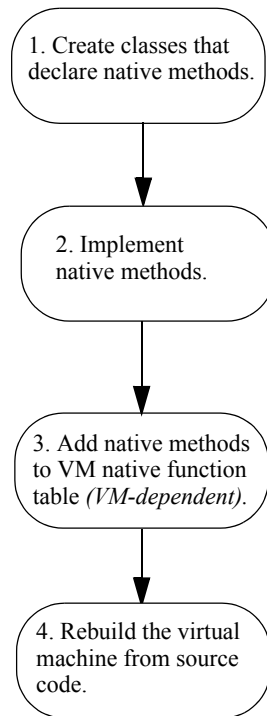
4. Run the Java class bytecode file `HelloWorld.class` with the new KVM interpreter. On a Solaris system and from the top of the current release of the CLDC directory hierarchy, we would invoke the KVM interpreter as follows:

```
${KVM_ROOT}/kvm/VmUnix/build/kvm -classpath  
${KVM_ROOT}/api/classes mypackage.HelloWorld
```

You should see the following displayed on the standard output:

```
hello, world!
```

The high-level process for implementing native methods is illustrated below in [FIGURE 2](#).



**FIGURE 2** Steps involved in implementing native methods

## Examples

---

This chapter contains simple examples that illustrate the use of the K Native Interface.

---

### 7.1 Parameter Passing

Consider the following Java class that illustrates parameter passing from a Java program to a set of native methods:

```
package mypackage;

public class ParameterPassing {
    private native void passOne(int i1);
    private native void passTwo(int i1, int i2);
    private native void passThree(int i1, int i2, int i3);

    public static void main(String[] args) {
        ParameterPassing p = new ParameterPassing();
        p.passOne(2);
        p.passTwo(2, 4);
        p.passThree(2, 4, 8);
    }
}
```

Below is the corresponding C programming language implementation of the native methods declared above:

```
#include <kni.h>
#include <stdio.h>

KNIEXPORT KNI_RETURNTYPE_VOID
Java_mypackage_ParameterPassing_passOne() {
    jint i1 = KNI_GetParameterAsInt(1);
    fprintf(stdout, "Parameter(s) passed: %ld\n", i1);
    KNI_ReturnVoid();
}
```

```

KNIEXPORT KNI_RETURNTYPE_VOID
Java_mypackage_ParameterPassing_passTwo() {
    jint i1 = KNI_GetParameterAsInt(1);
    jint i2 = KNI_GetParameterAsInt(2);
    fprintf(stdout, "Parameter(s) passed: %ld, %ld\n", i1, i2);
    KNI_ReturnVoid();
}

KNIEXPORT KNI_RETURNTYPE_VOID
Java_mypackage_ParameterPassing_passThree() {
    jint i1 = KNI_GetParameterAsInt(1);
    jint i2 = KNI_GetParameterAsInt(2);
    jint i3 = KNI_GetParameterAsInt(3);
    fprintf(stdout, "Parameter(s) passed: %ld, %ld, %ld\n",
            i1, i2, i3);
    KNI_ReturnVoid();
}

```

In general, methods parameters of a primitive type can be accessed in native code by calling the appropriate `KNI_GetParameterAs<Type>(jint index)` function with index starting at one (1). Parameters are mapped from left to right, i.e., index value 1 always refers to the leftmost parameter that has been passed on from the Java method to the native method. It is important to remember that long and double parameters occupy two entries in the operand stack.

**Accessing the ‘this’ pointer in instance methods.** In instance native methods (non-static native methods), the reference to the current instance (the value of the ‘this’ pointer) can be obtained by calling function

`KNI_GetThisPointer(thisHandle)`, where `thisHandle` is a handle declared in the context of the current native method. Handle `thisHandle` will contain the value of the ‘this’ pointer after calling this function.

---

**Note –** The ‘this’ pointer is not available in static native methods. The return value of the `KNI_GetThisPointer` function is unspecified for static native methods.

---

**Accessing the class pointer in static methods.** In static methods, the class pointer can be obtained by calling function `KNI_GetClassPointer(classHandle)`, where `classHandle` is a handle declared in the context of the current native method. Handle `classHandle` will contain the value of the class pointer after calling this function.

---

**Note –** If you need to access the class pointer in an instance (non-static) method, it is recommended that you do this by calling `KNI_GetThisPointer` first to obtain a handle to an object pointer, and then calling `KNI_GetObjectClass`.

---

---

## 7.2 Returning Values from Native Functions

Every KNI function MUST call one of the `KNI_ReturnVoid`, `KNI_Return<Type>` or `KNI_EndHandlesAndReturnObject` functions.

Even when a native function does not want to return any value back to the calling Java function, function `KNI_ReturnVoid` must be called, or otherwise the operand stack of the Java virtual machine may become corrupted.

### 7.2.1 Returning Primitive Values

Below is a small native code fragment that illustrates how to return a primitive value from a native function.

```
#include <kni.h>
KNIEXPORT KNI_RETURNTYPE_INT
Java_mypackage_MyClass_myNativeFunction1() {
    // Return integer 123 to the calling Java method
    KNI_ReturnInt(123);
}
```

### 7.2.2 Returning Object References

Because all the object references in KNI are handles rather than direct object references, returning object references from KNI function is slightly more complicated than returning primitive values. Below is a small native code fragment that illustrates how to return an object reference from a native function. In this example, we will simply return the 'this' pointer that is implicitly passed to every instance (non-static) method in Java.

```
#include <kni.h>
KNIEXPORT KNI_RETURNTYPE_OBJECT
Java_mypackage_MyClass_myNativeFunction2() {

    KNI_StartHandles(1);
    KNI_DeclareHandle(objectHandle);

    // Read the 'this' pointer
    KNI_GetThisPointer(objectHandle);

    // Return the 'this' pointer to the calling Java method
    KNI_EndHandlesAndReturnObject(objectHandle);
}
```

## 7.2.3 Returning Null Object References

If a native function wants to return a NULL pointer back to the calling Java function, the object handle must be set to NULL explicitly by calling `KNI_ReleaseHandle`. Below is a small example.

```
#include <kni.h>
KNIEXPORT KNI_RETURNTYPE_OBJECT
Java_mypackage_MyClass_myNativeFunction3() {

    KNI_StartHandles(1);
    KNI_DeclareHandle(objectHandle);

    // Set the handle explicitly to NULL
    KNI_ReleaseHandle(objectHandle);

    // Return the null reference to the calling Java method
    KNI_EndHandlesAndReturnObject(objectHandle);
}
```

---

## 7.3 Accessing Fields

The Java programming language supports two kinds of fields. Each instance of a class has its own copy of the *instance fields* of the class, whereas all instances of a class share the *static fields* of the class.

### 7.3.1 General Procedure for Accessing Fields

Field access is a two-step process. For instance fields, you first call `KNI_GetFieldID` to obtain the *field ID* for the given class reference, field name, and field descriptor (refer to [Section 4.3.3 “Field Descriptors”](#) for an overview of field descriptors):

```
fid = KNI_GetFieldID(classHandle, "count", "I");
```

Once you have obtained the *field ID*, you can pass the *object reference* and the *field ID* to the appropriate instance field access function:

```
jint count = KNI_GetIntField(objectHandle, fid);
```

For static fields, the procedure is similar, except that a separate set of functions is used:

1. Call `KNI_GetStaticFieldID` for static fields instead of `KNI_GetFieldID` for instance fields. `KNI_GetStaticFieldID` and `KNI_GetFieldID` have the same return type `jfieldID`.

2. Once a static field ID has been obtained, one can pass the *class reference*, instead of the *object reference*, to the appropriate static field access function.

```
fid = KNI_GetStaticFieldID(classHandle, "staticCount", "I");
jint staticCount = KNI_GetStaticIntField(classHandle, fid);
```

Remember that when you access a field of a *reference* type (a field that contains an object instead of a primitive value), the object reference will be returned as a handle. Below is a small example:

```
KNI_GetObjectField(objectHandle, fid, toHandle);
```

This function would read the instance field represented by `fid`, and assign the value of that field to `toHandle`.

## 7.3.2 Accessing Instance Fields

Let us take a look at an example program that illustrates how to access instance fields from a native method implementation.

```
package mypackage;

public class InstanceFieldAccess {
    private int value;

    private native void accessFieldNatively();

    public static void main(String[] args) {
        InstanceFieldAccess p = new InstanceFieldAccess();
        p.value = 100;
        p.accessFieldNatively();
        System.out.println("In Java:");
        System.out.println("  Value = " + p.value);
    }
}
```

The `InstanceFieldAccess` class defines an instance field value. The `main` method creates an object of this class, sets the instance field, and then calls the native method `InstanceFieldAccess.accessFieldNatively`. As we will see shortly, the native method prints to the standard output the value of the instance field.

Below is the implementation of the native method `mypackage.InstanceFieldAccess.accessFieldNatively`.

```
#include <kni.h>
#include <stdio.h>

KNIEXPORT KNI_RETURNTYPE_VOID
Java_mypackage_InstanceFieldAccess_accessFieldNatively() {
```

```

/* Declare handles */
KNI_StartHandles(2);
KNI_DeclareHandle(objectHandle);
KNI_DeclareHandle(classHandle);

/* Get 'this' pointer */
KNI_GetThisPointer(objectHandle);

/* Get instance's class */
KNI_GetObjectClass(objectHandle, classHandle);

/* Get field id and value */
jfieldID fid = KNI_GetFieldID(classHandle, "value", "I");
jint value = KNI_GetIntField(objectHandle, fid);

/* Print field value */
fprintf(stdout, "In C:\n Value = %ld\n", value);

KNI_EndHandles();
KNI_ReturnVoid();
}

```

Interpreting the `InstanceFieldAccess` class with the KVM runtime interpreter produces the following output:

In C:

Value = 100

In Java:

Value = 100

### 7.3.3 Accessing Static Fields

Accessing static fields is similar to accessing instance fields. Let us take a look at a minor variation of the `InstanceFieldAccess` example:

```

package mypackage;

public class StaticFieldAccess {
    private static int value;

    private native void accessFieldNatively();

    public static void main(String[] args) {
        StaticFieldAccess p = new StaticFieldAccess();
        value = 100;
        p.accessFieldNatively();
        System.out.println("In Java: ");
        System.out.println(" Value = " + value);
    }
}

```



```
}
```

The `StaticFieldAccess` class defines a static integer field value. The `StaticFieldAccess.main` method creates an object, initializes the static field, and then calls the native method `StaticFieldAccess.accessFieldNatively`. The native method prints to the standard output the value of the static field and then sets the field to a new value. To verify that the field has indeed changed, the program prints the static field value again after the native method returns.

Below is the implementation of the native method `StaticFieldAccess.accessFieldNatively`.

```
#include <kni.h>
#include <stdio.h>

KNIEXPORT KNI_RETURNTYPE_VOID
Java_mypackage_StaticFieldAccess_accessFieldNatively() {

    /* Declare handle */
    KNI_StartHandles(1);
    KNI_DeclareHandle(classHandle);

    /* Get class pointer */
    KNI_GetClassPointer(classHandle);

    /* Get "I" "value" field id and its value */
    jfieldID fid = KNI_GetStaticFieldID(classHandle, "value", "I");
    jint value = KNI_GetStaticIntField(classHandle, fid);

    /* Print "I" "value" field */
    fprintf(stdout, "In C:\n Value = %ld\n", value);

    /* Change "I" "value" field */
    KNI_SetStaticIntField(classHandle, fid, 200);

    KNI_EndHandles();
    KNI_ReturnVoid();
}
```

Interpreting the `StaticFieldAccess` class with the KVM runtime interpreter produces the following output:

In C:

Value = 100

In Java:

Value = 200

---

## 7.4 Accessing Arrays

The KNI provides access to arrays of a *primitive or reference type*. For example, in the following code segment written in the *The Java Programming Language*<sup>1</sup>:

```
int[] iarr;  
float[] farr;  
int[][] aiarr
```

`iarr` and `farr` are primitive arrays while `aiarr` is an array of a reference type.

Accessing arrays of a primitive type in a native method requires use of the family of `KNI_Get/Set<Type>ArrayElement` where `<Type>` is any one of the primitive types.

Let us look at a simple example. The main method of the following Java class calls a native method `sumArrayNatively` that adds up the contents of an `int` array.

```
package mypackage;  
  
public class SumIntArray {  
    private native int sumArrayNatively(int[] arr);  
    public static void main(String[] args) {  
        SumIntArray p = new SumIntArray();  
        int arr[] = new int[10];  
        for (int i = 0; i < 10; i++) {  
            arr[i] = i;  
        }  
        int sum = p.sumArrayNatively(arr);  
        System.out.println("sum: " + sum);  
    }  
}
```

The corresponding native C programming language code is shown below.

```
#include <kni.h>  
#include <stdio.h>  
  
KNIEXPORT KNI_RETURNTYPE_INT  
Java_mypackage_SumIntArray_sumArrayNatively() {  
    jint i, sum = 0;  
  
    /* Declare handle */  
    KNI_StartHandles(1);  
    KNI_DeclareHandle(arrayHandle);
```

1. *The Java™ Programming Language (Java Series), Second Edition* by Ken Arnold and James Gosling (Addison-Wesley, 1998)

```

/* Read parameter #1 to arrayHandle */
KNI_GetParameterAsObject(1, arrayHandle);

/* Sum int array components */
for (i = 0; i < 10; i++) {
    sum += KNI_GetIntArrayElement(arrayHandle, i);
}

/* Set result sum */
KNI_EndHandles();
KNI_ReturnInt(sum);
}

```

Remember that in KNI, arrays are represented by the `jarray` reference type and its “subtypes” such as `jintArray`. Just as a `jstring` is not a C string type, neither is `jarray` a C array type. One cannot implement the `Java_IntArray_sumArray` native method by indirecting through a `jarray` reference. Instead, one must use the proper `KNI_Get<Type>ArrayElement` or `KNI_Set<Type>ArrayElement` functions to access the array elements.

---

## 7.5 Accessing Strings

Below is a small sample program that illustrates string access using KNI.

```

package mypackage;

public class StringAccess {

    private native void accessStringNatively();

    public static void main(String[] args) {
        StringAccess p = new StringAccess();
        p.accessStringNatively("Parameter");
    }
}

```

The corresponding native code is below.

```

#include <kni.h>
#include <stdio.h>

KNIEXPORT KNI_RETURNTYPE_VOID
Java_mypackage_StringAccess_accessStringNatively() {

    /* Allocate static buffer for the Unicode string */
    jchar buffer[256];
    jsize size;
    int i;

```

```

/* Declare handle */
KNI_StartHandles(1);
KNI_DeclareHandle(stringHandle);

/* Read parameter #1 to stringHandle */
KNI_GetParameterAsObject(1, stringHandle);

/* Get the length of the string */
size = KNI_GetStringLength(stringHandle);

/* Copy the Java string to our own buffer (as Unicode) */
KNI_GetStringRegion(stringHandle, 0, size, buffer);

/* Print the Unicode characters as 8-bit chars */
for (int i = 0; i < length; i++) {
    fprintf(stdout, "%c", (char)buffer[i]);
}

KNI_EndHandles();
KNI_ReturnVoid();
}

```

---

**Note – IMPORTANT:** Remember that function `KNI_GetStringRegion` returns a Unicode string. This means that each returned character is 16 bits wide. It is important to take this into account when allocating buffer space for the returned string region.

---