MicroEJ
SNI-GT-1.2

Safe Native Interface for GreenThread Context
Profile Specification

ESR0012

Reference: ESR-SPE-0012-SNI-GT
Version: 1.2
Rev: I

Contents

Tables

Illustrations

1 PREFACE TO SNI-GT 1.2 PROFILE, ESR0012

This document defines the *SNI-GT 1.2* profile, targeting Java 2 Platforms that are "green threads" based platforms.

1.1 Who Should Use this Specification

This specification targets the following audiences:

- Individuals who want to build an implementation that complies to the SNI-GT profile specification;
- Application developers who want to design a software application using SNI-GT in the context of a "green threads" Java virtual machine.

1.2 How This Specification is Organized

This specification is organized as follow:

- **Introduction** is a short chapter explaining what SNI-GT is, why it has been designed, and its main assets.
- **Specification** describes the concepts required to understand how to write an application using SNI-GT.
- **SNI-GT API Documentation** lists the SNI-GT APIs as javadoc.

1.3 Comments

Your comments about SNI-GT are welcome. Please send them by electronic mail to the following address: `contact@microej.com`, with SNI-GT in your subject line.

1.4 Related literature

[B-ON]	Beyond: ESR001, `http://www.microej.com`

1.5 Document Conventions

In this document, references to methods of a Java class are written as `ClassName.methodName(args)`. This applies to both static and instance methods. Where the method is static this will be made clear in the accompanying text.

1.6 Definition

1.6.1 native

Java allows to write parts of the application in languages other than in the Java syntax. Such parts are said to be "native" parts. Therefore the Java programmer uses the keyword **`native`** to refer to such non-Java implementations.

1.6.2 static

Java is an Object Oriented language. An application is made of objects that communicate using message sends: an S object sends a message to an R object called the receiver. The message is implemented either in Java by a Java method[1], or in another language if the code is implemented using a Java native method.

Some methods are global to the application and do not refer to a specific object. They are said to be static: they do not rely on a receiver. Therefore the Java programmer uses the keyword **`static`** to refer to such global methods.

1.7 Implementation Notes

The SNI-GT specification does not include any implementation details. SNI-GT implementors are free to use whatever techniques they deem appropriate to implement the specification, with (or without) collaboration of any Java virtual machine provider. SNI-GT experts have taken great care not to mention any special Java virtual machines, nor any of their special features, in order to encourage fair competing implementations.

2 INTRODUCTION

2.1 Architecture

2.1.1 Purpose

The *Safe Native Interface for GreenThread Context*, named SNI-GT, is intended for implementing native Java methods in C language.

SNI-GT allows to:

- call a C function from a Java method.
- access an Immortal array in a C function (see `[B-ON]` specification to learn about immortal objects).

SNI-GT does not allow to:

- access or create a Java object in a C function.
- access Java static variables in a C function.
- call Java methods from a C function.

2.1.2 Green Threads Context

Green threads are threads that are internally managed by the Java virtual machine instead of being natively managed by the underlying Real-Time Operating System (RTOS), if any provided. A green threads Java virtual machine defines a multi-threaded environment without relying on any native RTOS capabilities.

Therefore, the whole Java world runs in one single RTOS task, within which the Java virtual machine re-creates a layer of (green) threads. One immediate advantage is that the Java-world CPU consumption is fully controlled by the RTOS task it is running in, allowing embedded engineers to

1 The term “method” is used in Java whereas “function” is used in C.

easily arbitrate between the different parts of their application. In particular in an open-to-third-parties framework, the maximum CPU time given to the Java world is fully under control at no risk, whatever the number and/or the activities of the Java green threads.

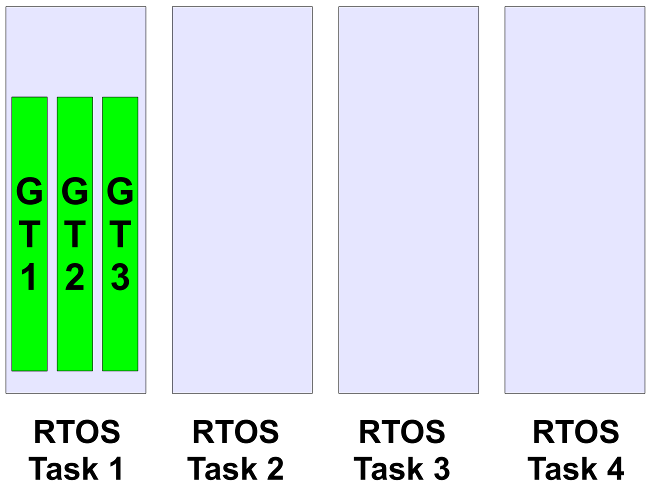

Illustration 2-1: A green threads architecture example

Illustration 2-1 shows 4 RTOS tasks, with the first one embedding 3 green threads. When the Task 1 is scheduled by the RTOS, the Java virtual machine executes. Therefore the Java virtual machine schedules the green threads.

2.2 First Example

This first example shows how to declare and implement a Java `native` method using SNI-GT. First the method has to be declared `native` in Java: this states that the method is written in another language. Then, the implementation of the method is written in C language.

```
package examples;
public class Hello{

   public static void main(String[] args){
      printHelloNbTimes(args.length);
   }

   public static native void printHelloNbTimes(int times);

}
```

The C source file declares a function `Java_examples_Hello_printHelloNbTimes`. It prints the message `Hello world!` several times by invoking the `printf()` function.

```
#include <sni.h>
#include <stdio.h>
void Java_examples_Hello_printHelloNbTimes(jint times){
    while (--times >= 0){
        printf("Hello world!\n");
    }
}
```

2.3 Java And Native Separation

SNI-GT defines how to cross the barrier between Java world and native world:

- Call a C function from Java.
- Pass parameters to the C function.
- Return a value from the C world to the Java world.
- Manipulate (read & write) shared memory both in Java and C : the immortal space.

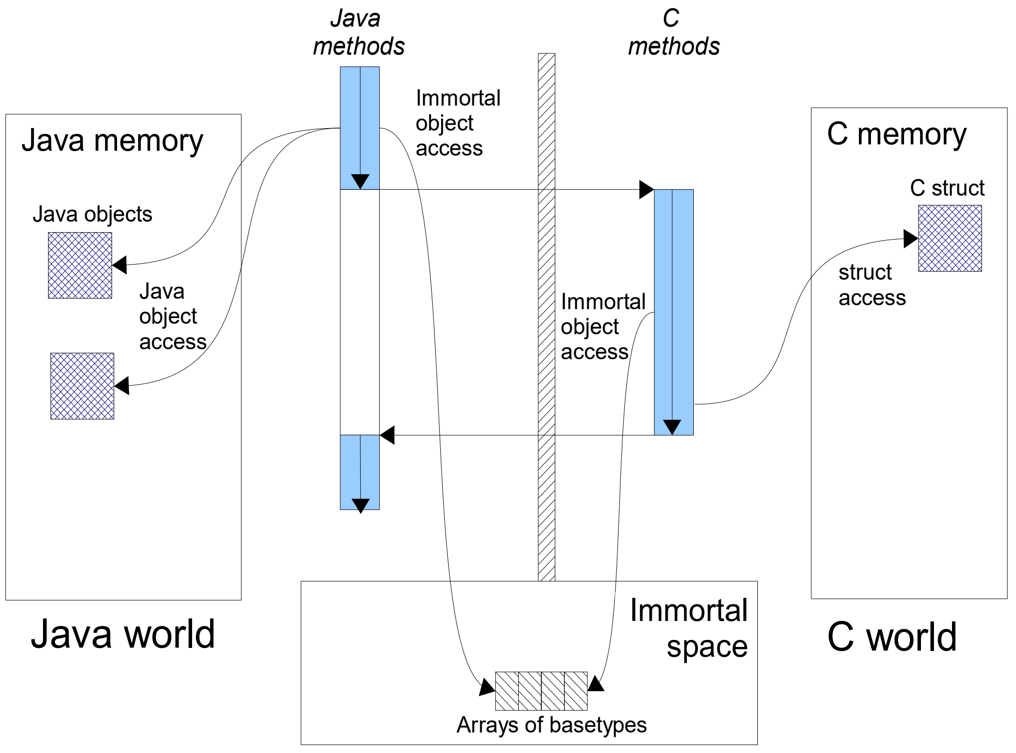

Illustration 2-2: Java to C calling sequence

Illustration 2-2 shows both Java and C code accesses to shared objects in the immortal space, while also accessing their respective memory.

2.4 Starting the “Java world”

2.4.1 Start-up

SNI-GT defines the Java virtual machine start-up process: the Java is under the control of the C world that initiated its launch as one RTOS task.

2.4.2 Without RTOS

When no RTOS is in use, the `main` function is viewed as the single RTOS task. It uses the whole CPU budget, except the time used by interrupts. The system is viewed throughout this specification has having just one RTOS task, which runs the Java world.

3 JAVA WORLD TO C WORLD

3.1 C Function Call From Java world

The SNI-GT specification allows the invocation of methods from Java to C: these methods must be `static native` methods, and the parameters must be base types or immortal array of base types (cf `[B-ON]`). These native methods are used in Java as standard Java methods.

Example:

```
package example;
public class Foo{

   public void bar(){
      int times = 3;
      print(times);
   }

   public static native void print(int times);
}
```

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

void Java_example_Foo_print(jint times){
    while (--times >= 0){
        printf("Hello world!\n");
    }
}
```

3.2 Java Types And C Types

3.2.1 Base Types

Types may have different representations depending on the language. The file `sni.h` defines the C types that represent exactly the Java types.

Java Type	Specification	C type
void	No returned type	`void`
boolean	unsigned 8 bits	`jboolean`
byte	signed 8 bits	`jbyte`
char	unsigned 16 bits	`jchar`
short	signed 16 bits	`jshort`
int	signed 32 bits	`jint`
long	signed 64 bits	`jlong`
float	IEEE 754 single precision 32 bits	`jfloat`
double	IEEE 754 double precision 64 bits	`jdouble`

Table 3-1: Java types to C types

3.2.2 Java Array

The Java arrays (of base types) are represented in C functions as C arrays: the array is a pointer on the first element of the array, all the elements in line within the memory.

Note that in C, strings are represented with C `char`[2] array with a `'\0'` as last character. In Java, strings are `jchar` array, not terminated by `'\0'`.

SNI-GT allows to get a Java array length in a C function.

```
int32_t SNI_getArrayLength(void* array);
```

3.3 Naming Convention

SNI-GT uses a naming convention to name-match the Java native method with its C counterpart function.

The C function name is the concatenation of the following components:

- the prefix “`Java_`”.
- the package name of the class, each sub packages is separated with “_”.
- the separator “_”.
- the class name.
- the separator “_”.
- the method name.

If the method is overloaded by another method, native or not (the two methods have the same name with different arguments), the function name must be followed by the arguments descriptor, obtained with the following components (except if the method has no arguments):

- the separator “__” (two underscores)
- the name of each argument type, without separator, preceded by “`_3`“ if it is an array.

Table 3-2 gives the descriptors of the Java types for arguments.

Java type	SNI-GT name
boolean	Z
byte	B
char	C
short	S
int	I
long	J
float	F
double	D

Table 3-2: SNI-GT Java types descriptors in arguments

The character underscore (“_”) is used as a separator in the name. If this character is used within the Java name (either in package, class name or method name), it is replaced with “`_1`”. Because

2 `sizeof(char)` is `1` whereas `sizeof(jchar)` is `2`

the Java names cannot start with a number, the characters "`_1`" cannot be confused with separator character.

Examples of Java native methods and their counterpart C functions:

```
package example.sni.impl;

class Hello {

  public static native void nativ01(int i);
  public static native void nativ02(boolean b, int[] i);
  public static native void nativ_03();
  public static native void nativ04();
  public static native void nativ04(long l, double d);
  public static native void nativ04(int[] ia, int ib, char[] ca);
}
```

```
void Java_example_sni_impl_Hello_nativ01( jint i);
void Java_example_sni_impl_Hello_nativ02( jboolean b, jint* i);
void Java_example_sni_impl_Hello_nativ_103();
void Java_example_sni_impl_Hello_nativ04();
void Java_example_sni_impl_Hello_nativ04__JD( jlong l, jdouble d);
void Java_example_sni_impl_Hello_nativ04___3II_3C( jint* ia, jint ib,
jchar* ca);
```

3.4 Parameters Constraints

There are strong constraints on arguments given by Java methods to native functions:

- Only base types, array of base types are allowed in the parameters. No other objects can be passed: the native functions cannot access Java objects field nor methods.
- When base type arrays are passed in parameters,
    1. they must have only one dimension. No multi dimension array are allowed (`int[][]` is forbidden for example).
    2. they must be immortal arrays (see [B-ON 1.2]). Use the method `Immortals.setImmortal()` to transform an array into an immortal array.
- Only base types are allowed as return type

This constraints are checked at link-time to ensure that they are respected, except for the immortal arrays constraint (at link-time, compiler cannot figure out if an array reference is immortal or not). If an array used in an argument is not immortal, a `java.lang.IllegalArgumentException` is thrown at runtime when the native method is called.

3.5 Mixing Java and C execution sequence

3.5.1 Java natives: calling C from Java

When a Java native method executes, it executes its C counterpart function. This is done using the CPU budget of the RTOS task that has embedded the Java world.

While the C function executes, no other Java methods executes: the Java world "waits" for the C function to finish. Enough stack memory must be given to the C function in order for it to execute.

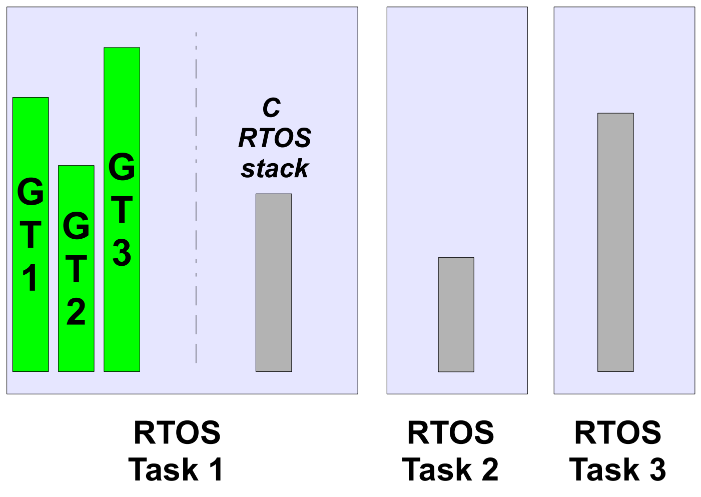

Illustration 3-1: Green threads and native stacks

Illustration 3-1 shows that green threads share the same native stack: the stack of the RTOS task that is running the Java virtual machine.

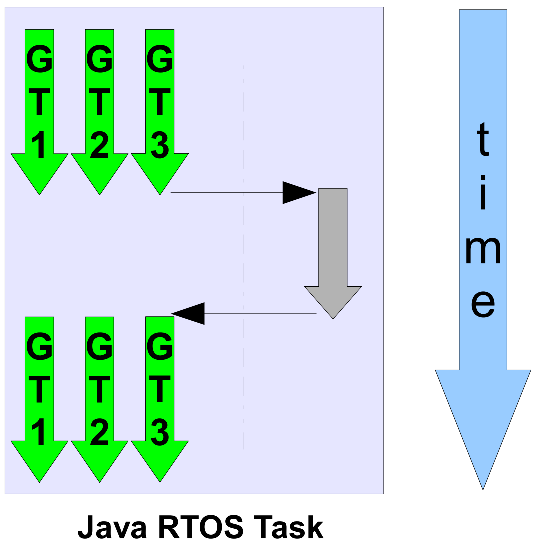

Illustration 3-2: A green threads Java virtual machine RTOS task activity

Illustration 3-2 shows a green threads Java virtual machine RTOS task. Green thread GT3 has called a native method that executes in C. All Java activities is suspended until the C execution has finished.

3.5.2 Synchronization between Java threads and C RTOS tasks

SNI-GT defines C functions that provide controls upon the green threads activities:

- `int32_t SNI_suspendCurrentJavaThread(int64_t timeout)`: suspends the execution of the Java thread that has initiated the current C call. This function does not block the C execution. The suspension is effective only at the end of the native method call (when the C call returns). The green thread is suspended until either a RTOS task calls `SNI_resumeJavaThread` or if the specified amount of milliseconds has elapsed.

- `int32_t SNI_getCurrentJavaThreadID(void)`: permits to retrieve the ID of the current Java thread within the C function (assuming it is a "native Java to C call"). This ID must be given to the `SNI_resumeJavaThread` function in order to resume the green thread execution.
- `int32_t SNI_resumeJavaThread(int32_t id)`: resumes the green thread with given ID. If the thread is not suspended, the resume stays pending.

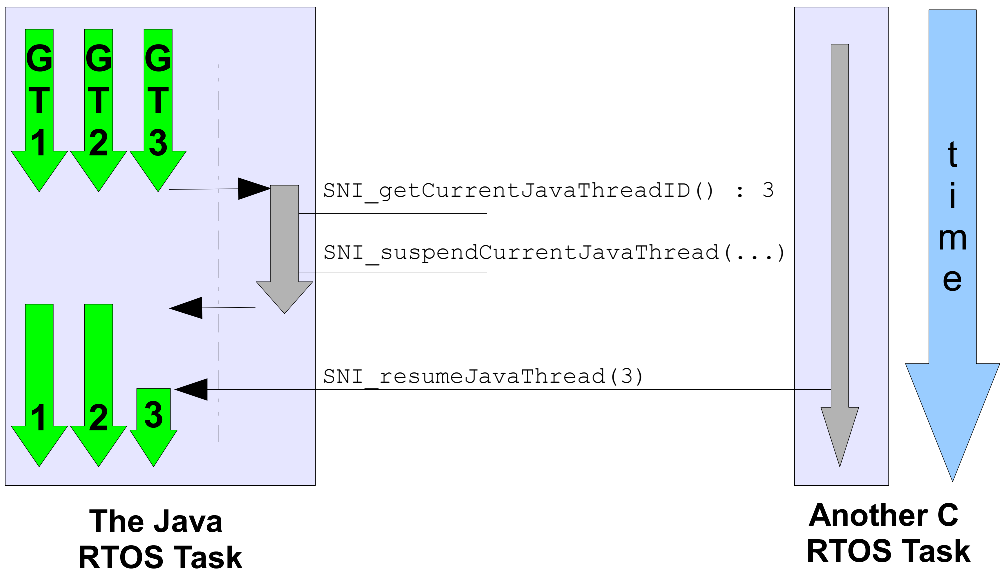

Illustration 3-3: Green threads and RTOS task synchronization

Illustration 3-3 shows a green thread (GT3) which has called a native method that executes in C. The C code suspends it, after having provisioning its ID (e.g. 3). Another RTOS task may later resume the Java green thread.

4 JAVA VIRTUAL MACHINE STARTUP

A green threads Java virtual machine needs first to be initialized, and then started. It is the programmer responsibility to create an RTOS task and to start the Java virtual machine within this task.

SNI-GT defines C functions to create a Java world, to start it and to free it:

- `void* SNI_createVM(void)`: creates and initializes the Java virtual machine context.
- `int32_t SNI_startVM(void*,int32_t,char**)`: starts the Java virtual machine. This function returns when the Java application ends.
- `int32_t SNI_getExitCode(void* vm)`: gets the Java application exit code, after `SNI_startVM` has successfully returned. This is the value passed by the application to `System.exit()` method.

- `void SNI_destroyVM(void* vm)`: does nothing if the Java virtual machine is still running. This function must be called in the RTOS task that created the Java virtual machine.

Illustration 4-1 shows a typical example of Java virtual machine startup code.

```
void javaWorldTask() {
   int32_t err;
   int32_t exitCode;
   void* myVM;

   myVM = SNI_createVM();
   if (myVM == NULL) {
      printf("Failed to create the Java world\n");
   }
   else {
      err = SNI_startVM(myVM, 0, NULL);
      if(err < 0) {
          printf("VM ends with error (%d)\n", err);
      }
      else {
          exitCode = SNI_getExitCode(myVM);
          printf("Java exit code = %d\n", exitCode);
      }
      SNI_destroyVM(myVM);
   }
}
```

Illustration 4-1: Example of Java virtual machine startup code in C

5 SNI-GT APIS

5.1 C Header File

The file `sni.h` contains all the types and functions definitions to interact with the Java world in the C world.

```
/*
 * C header file
 * Copyright 2008-2014 IS2T. All rights reserved.
 * Modification and distribution is permitted under certain conditions.
 * IS2T PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
 */

/*
 * Header file for Safe Native Interface (SNI), version 1.2
 */
#ifndef SNI_H
#define SNI_H

#include <stdint.h>

#ifdef __cplusplus
      extern "C" {
#endif

typedef int8_t          jbyte;    /* 8 bits  */
typedef uint8_t         jboolean; /* 8 bits  */
typedef uint16_t        jchar;    /* 16 bits */
typedef int16_t         jshort;   /* 16 bits */
typedef int32_t         jint;     /* 32 bits */
typedef float           jfloat;   /* 32 bits */
typedef double          jdouble;  /* 64 bits */
typedef int64_t         jlong;    /* 64 bits */

//boolean values
#define JTRUE     (1)
#define JFALSE    (0)
#define JNULL     (0)

#define SNI_OK               (0)   //function succeeded
#define SNI_ERROR            (-1)//an error was detected
#define SNI_INTERRUPTED (1) //see SNI_suspendCurrentJavaThread


/*
 * Returns the length of a Java array
 */
int32_t SNI_getArrayLength(void* array);


/*
 * Creates and initializes a virtual machine.
 * This function MUST be called once before a call to
 * <code>SNI_startVM()</code>.
 *
 * Returns null if an error occurred, otherwise returns a
 * virtual machine instance.
 */
void* SNI_createVM(void);

/*
 * Starts the specified virtual machine and calls the
 * <code>main</code> method of the Java application with
 * the given String arguments.
 * This function returns when the Java application ends.
 *
 * The Java application ends when there are no more Java
 * threads to run or when the Java method
 * <code>System.exit(int)</code> is called.
 *
 * Returns 0 when the virtual machine ends normally or
 * a negative value when an error occurred .
 */
int32_t SNI_startVM(void* vm, int32_t argc, char** argv);

/*
 * Call this method after virtual machine execution
 * to get the Java application exit code.
 *
 * Returns the value given to the <code>System.exit(exitCode)</code>
 * or 0 if the Java application ended without calling
 * <code>System.exit(exitCode)</code>.
 */
int32_t SNI_getExitCode(void* vm);

/*
 * Releases all the virtual machine resources. This method
 * must be called after the end of the execution of
 * the virtual machine.
 */
void SNI_destroyVM(void* vm);

/*
 * Returns the ID of the current Java thread.
 * This function must be called within the virtual machine task.
 *
 * Returns <code>SNI_ERROR</code> if this function is not called
 * within the virtual machine task.
 */
int32_t SNI_getCurrentJavaThreadID(void);

/*
 * Causes the current Java thread to pause its Java execution after the
 * end of the current native method. This function is not blocking.
 * The current Java thread will resume its execution after the reception
 * of an external event or after <code>timeout</code> milliseconds.
 *
 * If a resume has been done on this thread before calling this
 * function, the thread is not paused.
 *
 * The result of calling this method several times during the same
 * native execution is unpredictable.
 *
 * Parameter <code>timeout</code> is the duration in milliseconds of the
 * pause. If <code>timeout</code> is zero, then time is not taken into
 * consideration and the thread simply waits until resumed.
 *
 * Returns <code>SNI_OK</code> if the request is accepted (i.e. the
 * thread will suspend its execution at the end of the current native).
 * Returns <code>SNI_ERROR</code> if the method is called outside of the
 * VM Task.
 * Returns <code>SNI_INTERRUPTED</code> if a resume is pending; the
 * current java thread will not suspend its execution.
 */
int32_t SNI_suspendCurrentJavaThread(int64_t timeout);

/*
 * Resume the given Java thread if it is suspended.
 * If the Java thread is not paused, this resume stays pending.
 * Next call of SNI_suspendCurrentJavaThread() will return immediately.
 *
 * Parameter <code>javaThreadID</code> is the ID of the Java thread to
 * resume.
 *
 * Returns <code>SNI_ERROR</code> if the given Java thread ID is
 * invalid, otherwise returns <code>SNI_OK</code>.
 */
int32_t SNI_resumeJavaThread(int32_t javaThreadID);

#ifdef __cplusplus
      }
#endif

#endif /* SNI_H */
```

5.2 Java API

ej.sni

```
java.lang.Object
  └ ej.sni.PoolOfReusableResources
      └ ej.sni.PoolOfLimitedReusableResources
          └ ej.sni.PoolOfImmortalByteBuffers
```

```
public class PoolOfImmortalByteBuffers
extends PoolOfLimitedReusableResources
```

A pool of reusable immortal byte buffers.

Constructor Summary	*Page*
PoolOfImmortalByteBuffers (int maxNbBuffers, int allocationSize) Allocate a new pool of Immortals byte buffer resources	Error: Reference source not found

Method Summary	
protected Object	**newResource ()** Allocate a new Immortal byte buffer resource

Methods inherited from class ej.sni.PoolOfLimitedReusableResources
getAllocationSize

Methods inherited from class ej.sni.PoolOfReusableResources
release , reserve

Constructor Detail

```
public PoolOfImmortalByteBuffers(int maxNbBuffers,
                                 int allocationSize)
```

Allocate a new pool of Immortals byte buffer resources

Parameters:

`allocationSize` - size of allocated resources (in bytes). 0 (i.e. unlimited) is not allowed.

Method Detail

`protected Object **newResource**()`

Allocate a new Immortal byte buffer resource

Overrides:

newResource in class PoolOfReusableResources

ej.sni

```
java.lang.Object
  └ ej.sni.PoolOfReusableResources
      └ ej.sni.PoolOfLimitedReusableResources
```

Direct Known Subclasses:

PoolOfImmortalByteBuffers

```
abstract public class PoolOfLimitedReusableResources
extends PoolOfReusableResources
```

A pool of resources where resources are allocated at creation time (no lazy allocation).

Constructor Summary	*Page*
PoolOfLimitedReusableResources `(int maxNbResources, int allocationSize)` Allocate a new pool of resources	Error: Reference source not found

Method Summary	
`int`	**getAllocationSize** `()` Return size of allocated resources (in bytes).

Methods inherited from class ej.sni.PoolOfReusableResources
newResource , release , reserve

Constructor Detail

```
public PoolOfLimitedReusableResources(int maxNbResources,
                                      int allocationSize)
```

Allocate a new pool of resources

Parameters:

`maxNbResources` - 0 (i.e. unlimited) is not allowed.
`allocationSize` - size of allocated resources (in bytes). 0 (i.e. unlimited) is not allowed.

Method Detail

`public int` **`getAllocationSize`**`()`

Return size of allocated resources (in bytes).

Returns:
size of allocated resources (in bytes).

ej.sni

```
java.lang.Object
  └─
    ej.sni.PoolOfReusableResources
```

Direct Known Subclasses:

PoolOfLimitedReusableResources

```
abstract public class PoolOfReusableResources
extends Object
```

A pool of reusable resources. A buffer is reserved using reserve() and released using release(Object) . Pool may have a maximum number of resources.

Constructor Summary	*Page*
PoolOfReusableResources (int maxNbResources) Allocate a new pool of resources	Error: Reference source not found

Method Summary	
protected abstract Object	**newResource ()** Allocate a new resource
void	**release (Object buffer)**
Object	**reserve ()** Reserve a buffer.

Constructor Detail

```
public PoolOfReusableResources(int maxNbResources)
```

Allocate a new pool of resources

Parameters:

`maxNbResources` - a strictly positive integer giving the maximum number of allocated resources, or 0 if an unlimited number of resources is allowed

Method Detail

```
public synchronized Object reserve()
```

Reserve a buffer. In case all resources are in use and maxNbresources `is not reached, a new buffer is allocated. Otherwise this function blocks until a buffer is available`

Returns:

an array

```
public synchronized void release(Object buffer)
```

```
protected abstract Object newResource()
```

Allocate a new resource

ej.sni

```
java.lang.Object
  └─
    ej.sni.SNI
```

```
public class SNI
extends Object
```

Constructor Summary	*Page*
SNI ()	Error: Reference source not found

Method Summary	
static void	**toCString (String javaString, byte[] cString)** Transforms a Java String into a C String. The platform default encoding is used to transform Java characters into C characters. The created C String is a NULL terminated String (ends with '\0').
static String	**toJavaString (byte[] cString)** Transforms a C String into a Java String, using platform default encoding.

Constructor Detail

```
public SNI()
```

Method Detail

```
public static void toCString(String javaString,
                             byte[] cString)
```

Transforms a Java String into a C String.
The platform default encoding is used to transform Java characters into C characters.
The created C String is a NULL terminated String (ends with '\0'). The `cString` array length must be at least `javaString.length()+1`.

Parameters:

`javaString` - the Java String
`cString` - byte array which contains the C String.

Throws:

`IllegalArgumentException` - if javaString or cString is null

`ArrayIndexOutOfBoundsException` - if cString is too small to contain the string

`public static String` **`toJavaString`**`(byte[] cString)`

Transforms a C String into a Java String, using platform default encoding. The C String must be NULL terminated.

Parameters:

`cString` - byte array which contains the C String

Returns:

a new Java String.

Throws:

`IllegalArgumentException` - if cString is null or its length is 1,
if cString is not NULL terminated.