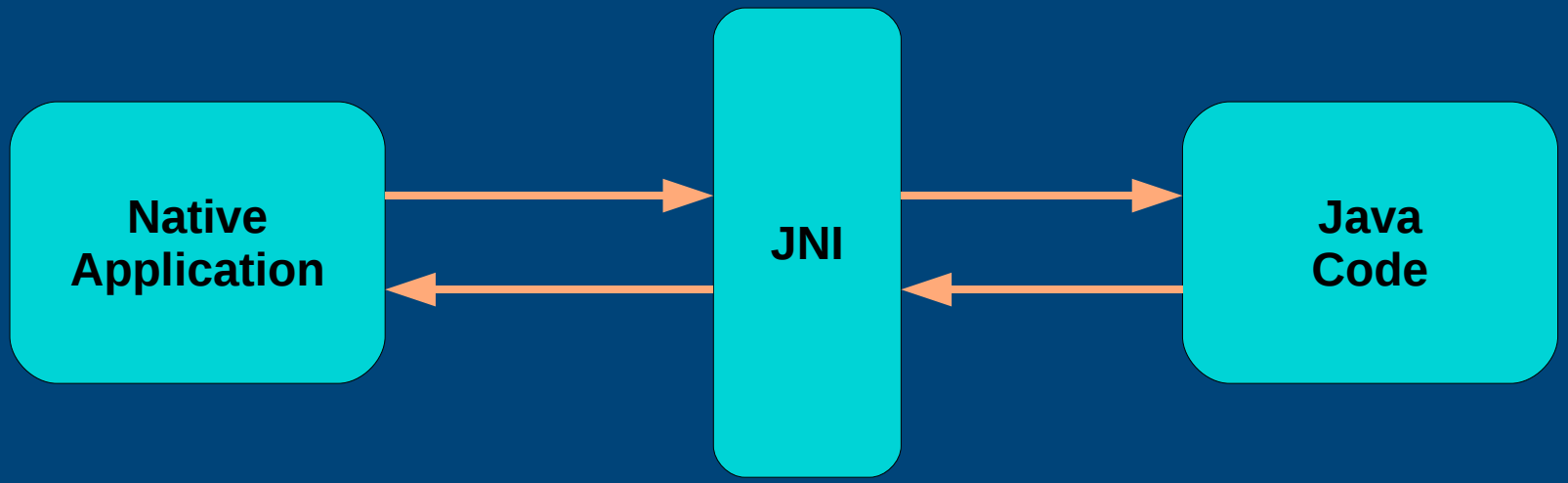
Finding Bugs in Java Native Interface Programs

Goh Kondo, Tamiya Onodera

Tokyo Research Laboratory
IBM Research

Presented in class by
Raghu V. Rao
CISC879-012 Software Testing and Maintenance

Java Native Interface (JNI)



- A framework that allows Java code running in a Java Virtual Machine to call and be called by native applications

Motivation

- JNI is tedious and error-prone
  - For a simple expression that is a few terms in Java, JNI takes several lines in native code
  - A whole dedicated chapter in *The Java Native Interface: Programmer's Guide and Specification*
- Some JNI functions must be called in a specific order
- **Mistakes not caught by the compiler!**

Problems Addressed

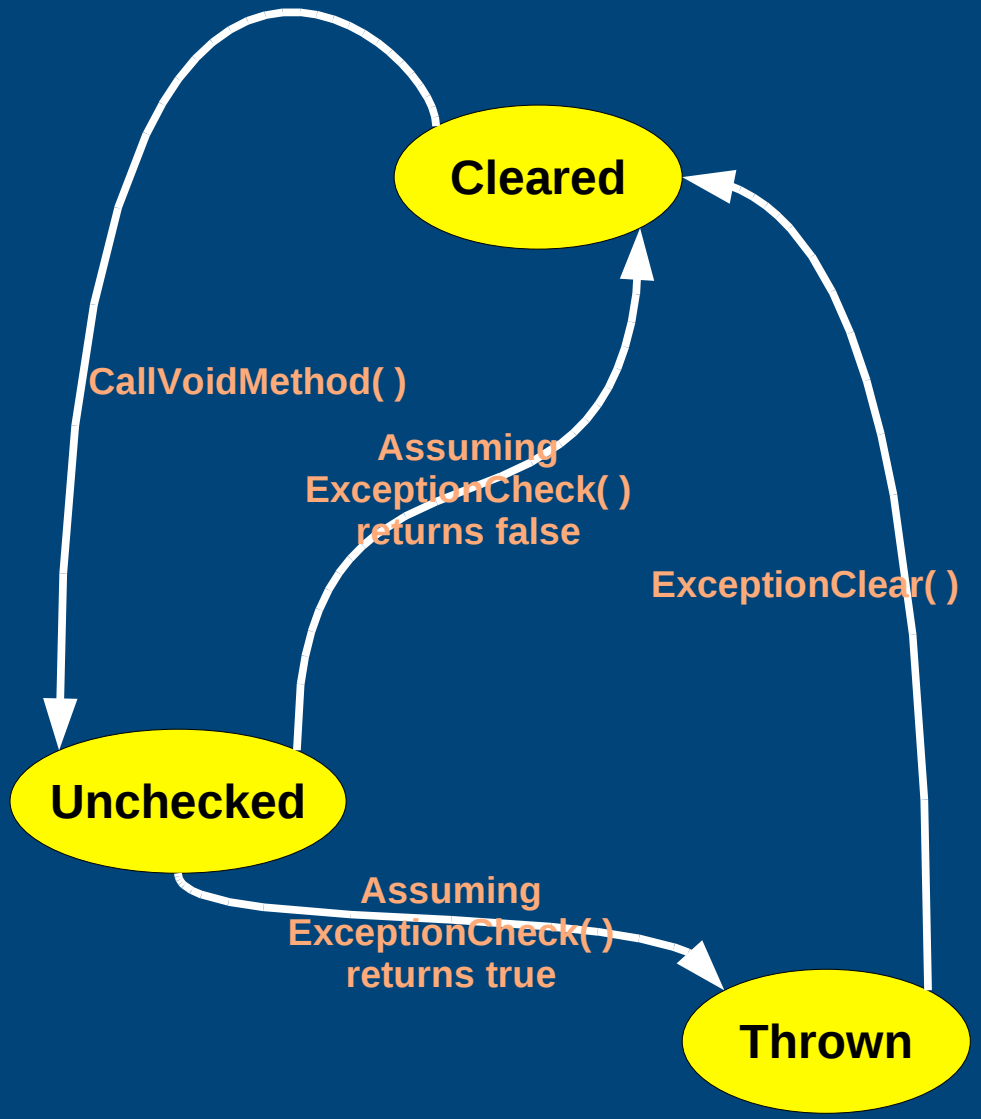
- Of the 15 problems pointed out in The Java Native Interface book, the authors address:
  - Error Checking
  - Retaining Virtual Machine Resources
  - Using Invalid Local References
- In addition: JNI function calls in critical regions

Problem 1: Error Checking

```
jclass cls = env->GetObjectClass(obj);
jmethodID mid =
env->GetMethodID(cls, "foo", "()V");
env->CallVoidMethod(obj, mid);
if (env->ExceptionCheck()) {
  /* error handling */
  env->ExceptionClear(); /* or return */
}
mid = env->GetMethodID(cls, "bar", "()V");
env->CallVoidMethod(obj, mid);
```

- Highlighted segment checks if an exception was thrown
- Programmers often forget to include this check

Addressing Problem 1



- Typestate analysis
- Precondition: must start in state “Clear”
- Reports an error if JNI functions are invoked in states “Unchecked” or “Thrown”

Addressing Problem 1: Example

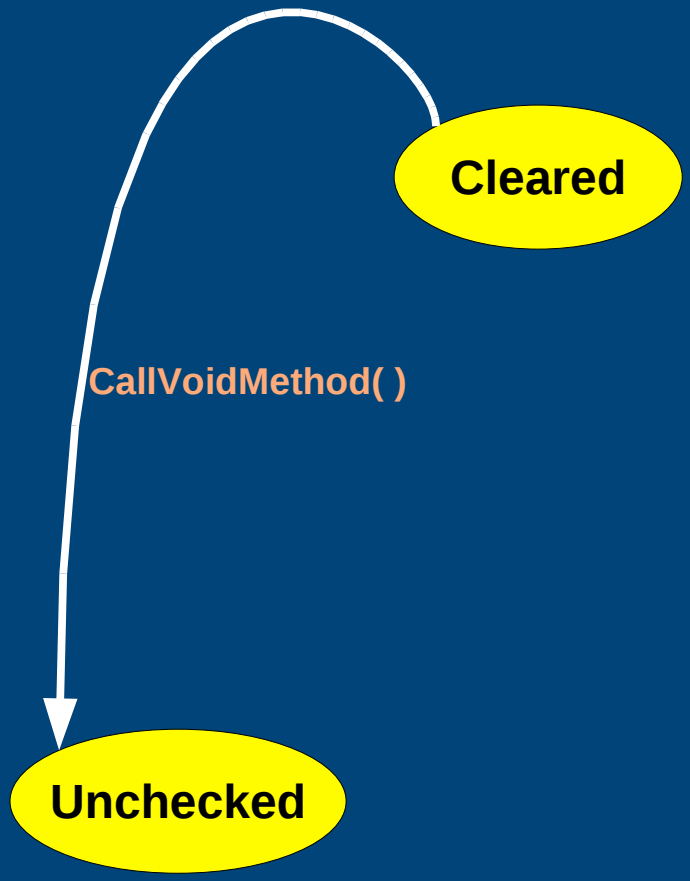
Cleared

Unchecked

Thrown

```
jclass cls = env->GetObjectClass(obj);
jmethodID mid =
env->GetMethodID(cls, "foo", "()V");
env->CallVoidMethod(obj, mid);
if (env->ExceptionCheck()) {
  /* error handling */
  env->ExceptionClear(); /* or return */
}
mid = env->GetMethodID(cls, "bar", "()V");
env->CallVoidMethod(obj, mid);
```

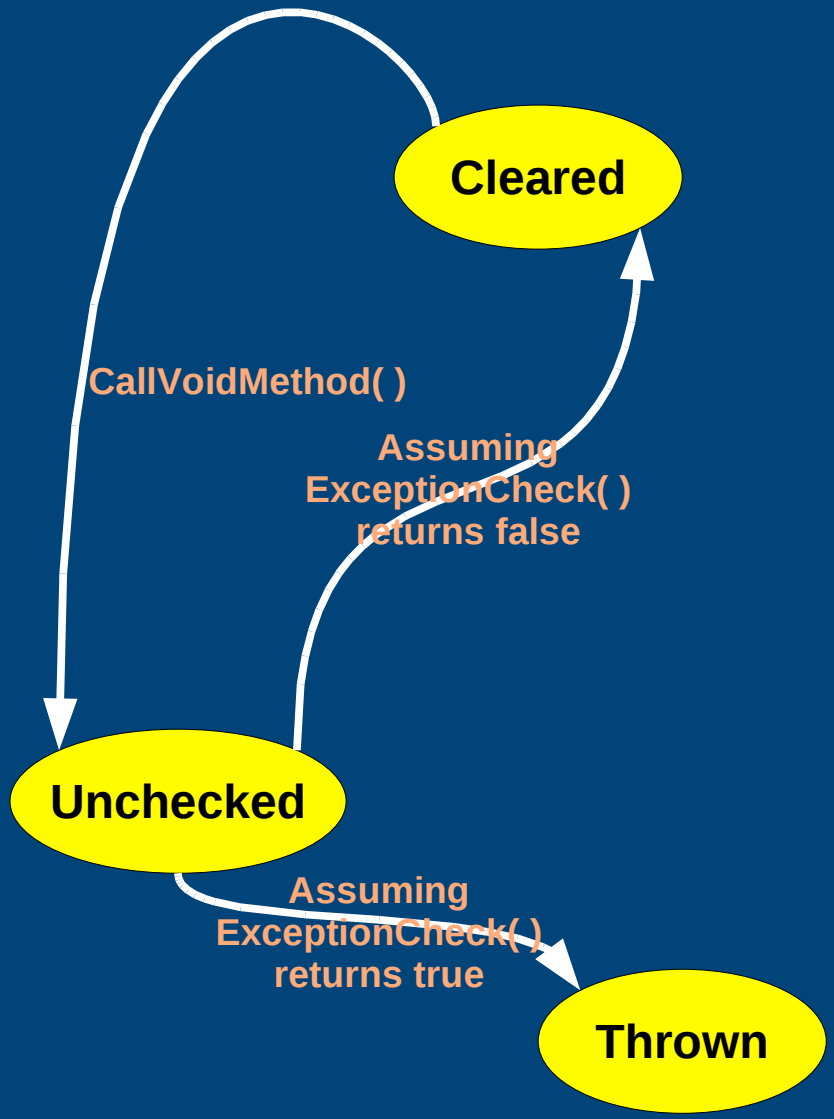
Addressing Problem 1: Example




```
jclass cls = env->GetObjectClass(obj);
jmethodID mid =
env->GetMethodID(cls, "foo", "()V");
env->CallVoidMethod(obj, mid);
if (env->ExceptionCheck()) {
  /* error handling */
  env->ExceptionClear(); /* or return */
}
mid = env->GetMethodID(cls, "bar", "()V");
env->CallVoidMethod(obj, mid);
```

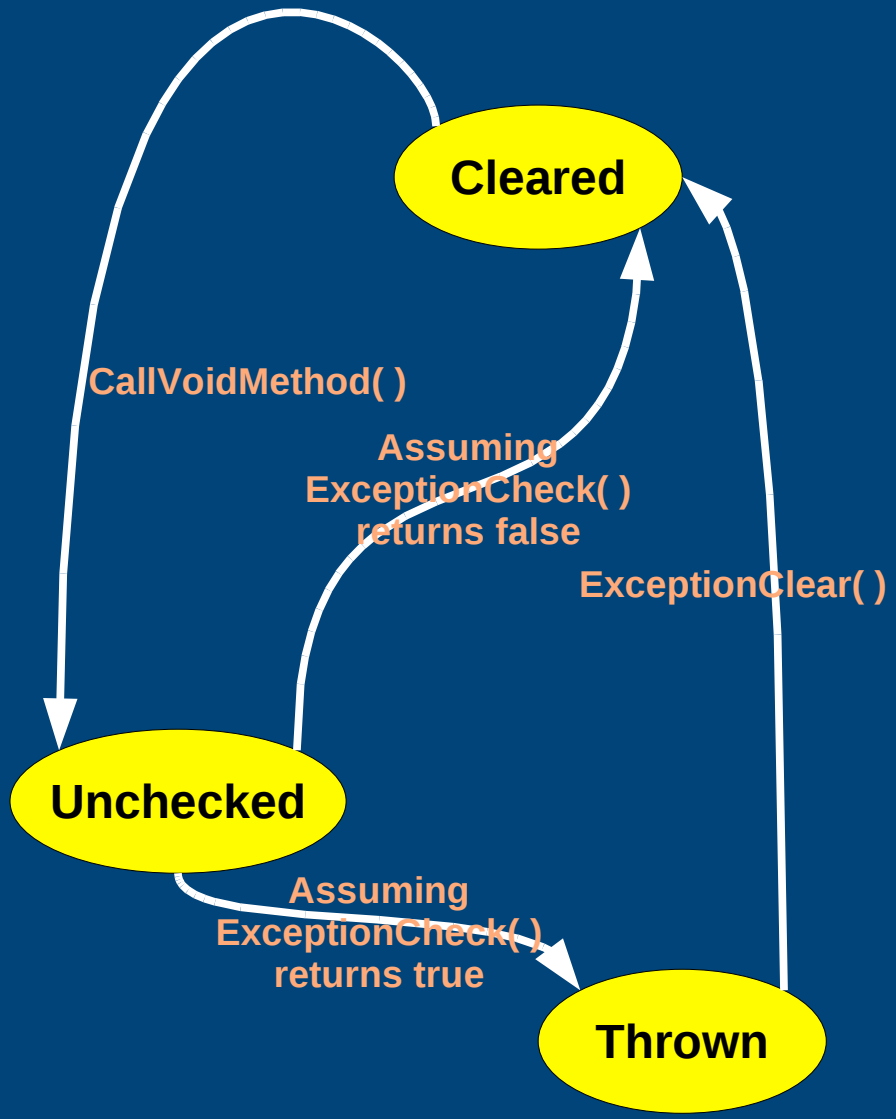
Thrown

Addressing Problem 1: Example




```
jclass cls = env->GetObjectClass(obj);
jmethodID mid =
env->GetMethodID(cls, "foo", "()V");
env->CallVoidMethod(obj, mid);
if (env->ExceptionCheck()) {
  /* error handling */
  env->ExceptionClear(); /* or return */
}
mid = env->GetMethodID(cls, "bar", "()V");
env->CallVoidMethod(obj, mid);
```

Addressing Problem 1: Example




```
jclass cls = env->GetObjectClass(obj);
jmethodID mid =
env->GetMethodID(cls, "foo", "()V");
env->CallVoidMethod(obj, mid);
if (env->ExceptionCheck()) {
  /* error handling */
  env->ExceptionClear(); /* or return */
}
mid = env->GetMethodID(cls, "bar", "()V");
env->CallVoidMethod(obj, mid);
```

Problem 2: Retaining Virtual Memory Resources

GetStringChars(JNIEnv *env, jstring str, jboolean *isCopy)

GetStringUTFChars(JNIEnv *env, jstring str, jboolean *isCopy)

Get<Type>ArrayElements(JNIEnv *env, <ArrayType> array, jboolean *isCopy)

ReleaseStringChars

ReleaseStringUTFChars

Release<Type>ArrayElements

- These functions allow native code to allocate Java VM resources dynamically
- Allocated resources are freed by these methods; *programmers often forget to call them*

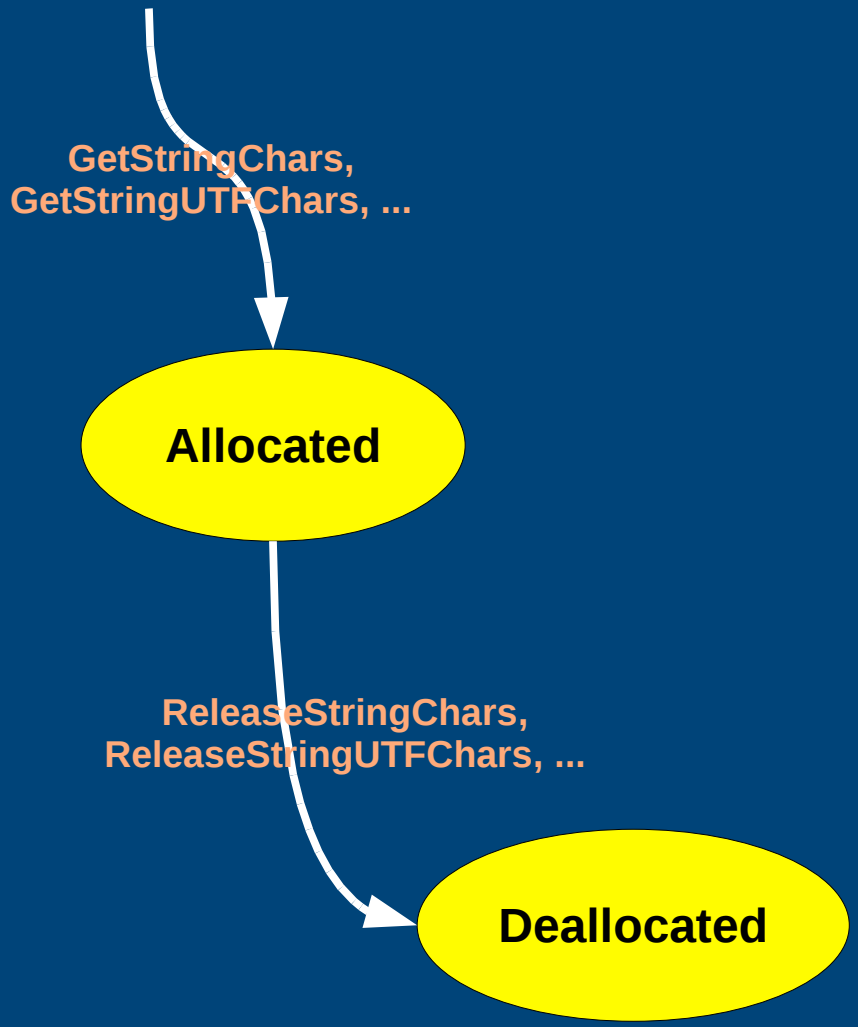
Problem 2: Retaining Virtual Memory Resources (contd.)

```
GetStringChars(JNIEnv *env, jstring str,
    jboolean *isCopy)

GetStringUTFChars(JNIEnv *env,
    jstring str, jboolean *isCopy)

Get<Type>ArrayElements(JNIEnv
    *env, <ArrayType> array, jboolean
    *isCopy)
```

```
jboolean isCopy;
const char *cstr =
  (*env)->GetStringChars(env, jstr,
    &isCopy);
...
if (isCopy) { /* INCORRECT! */
  (*env)->ReleaseStringChars(env, jstr,
    cstr); }
```

- Another problem is with the parameter isCopy
  - *isCopy ← JNI_TRUE, if a copy was made for the return value
  - *isCopy ← JNI_FALSE, otherwise
- Incorrect assumption: no need to release returned resource *if *isCopy == JNI_FALSE or if NULL was passed to isCopy*

Addressing Problem 2

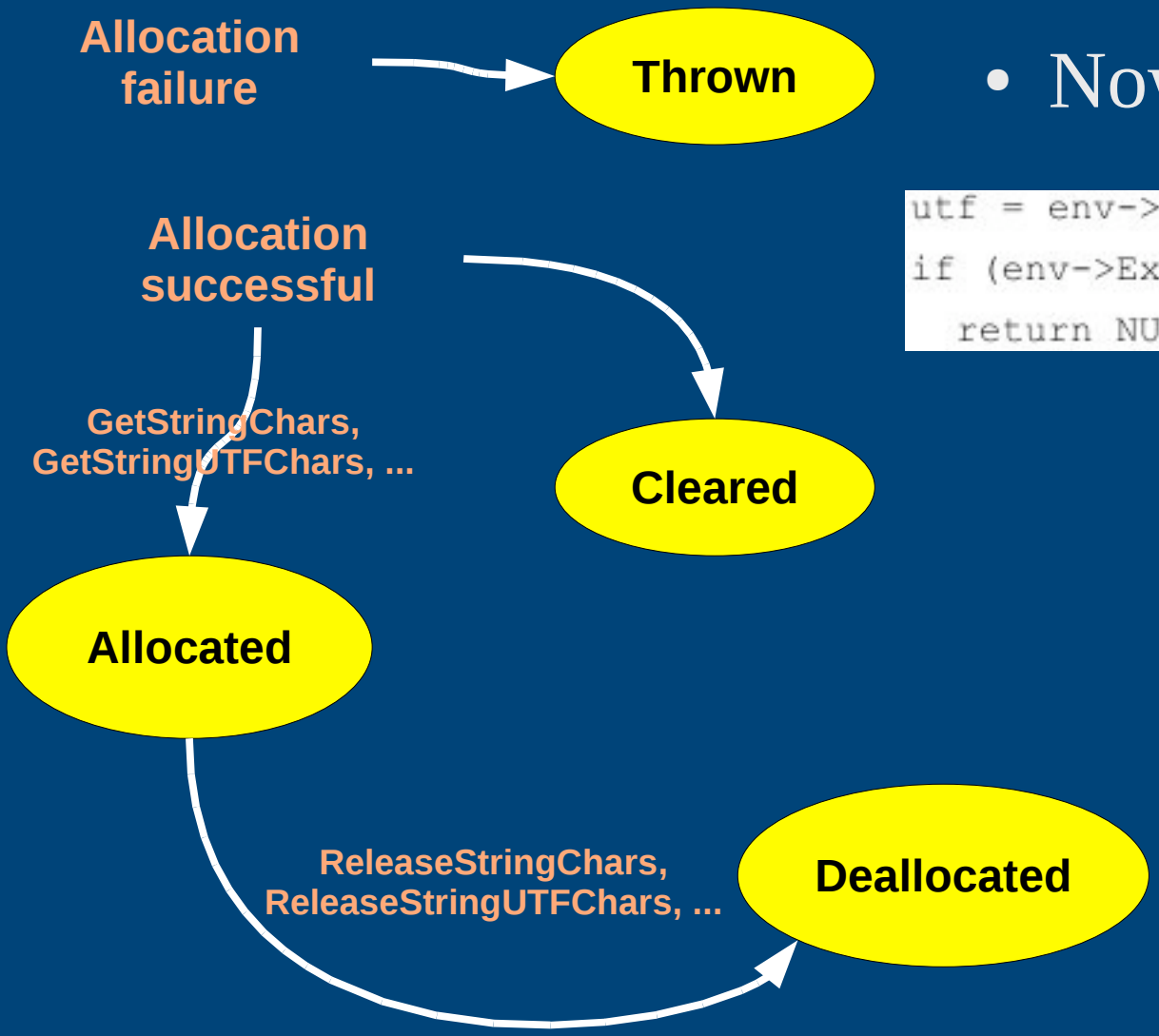


- Typestate analysis

```
utf = env->GetStringUTFChars(str, NULL);
if (env->ExceptionOccurred())
  return NULL;
```

- Does not handle all cases – what if an allocation fails? We'll have a false positive!

Addressing Problem 2: What happens when allocation fails?



- Now, no false positive.

```
utf = env->GetStringUTFChars(str, NULL);
if (env->ExceptionOccurred())
  return NULL;
```

Problem 3: Invalid Local References

```
static jclass fooCls;
JNIEXPORT void JNICALL
Java_Foo_initialize(JNIEnv *env, jclass cls)
{ /* fooCls should be converted into global */
  fooCls = (*env)->FindClass(env, "Foo");
}
JNIEXPORT void JNICALL
Java_Foo_bar(JNIEnv *env, jobject object)
{
/*fooCls is no longer valid */
jmethodID mid =
(*env)->GetMethodID(env, fooCls, "foo", "()V");
  …
}
```

fooCls only has a local reference

Bad use of fooCls

Problem 3: Invalid Local References

```
static jclass fooCls;
JNIEXPORT void JNICALL
Java_Foo_initialize(JNIEnv *env, jclass cls)
{ /* fooCls should be converted into global */
  fooCls = (*env)->FindClass(env, "Foo");
}
JNIEXPORT void JNICALL
Java_Foo_bar(JNIEnv *env, jobject object)
{
/*fooCls is no longer valid */
jmethodID mid =
(*env)->GetMethodID(env, fooCls, "foo", "()V");
  ...
}
```

fooCls only has a local reference

We need a call to *NewGlobalRef* to convert the local ref. to a global ref.

Bad use of fooCls

Addressing Problem 3

- No typestate analysis
- Instead, syntax check
  - Find an assignment whose left hand side is a global variable and right hand side takes value returned by a function call other than NewGlobalRef (or NewWeakGlobalRef)

Problem 4: Calling a JNI Function in a Critical Region

```
CallInCriticalRegion(JNIEnv *env,
     jobject obj, jstring)
{
  jboolean isCopy;
  const jchar *cstr =
    env->GetStringCritical(jstr,&isCopy);

  /* no JNI functions should be here! */
  env->CallVoidMethod(obj, mid);

  env->ReleaseStringCritical(jstr, cstr);
}
```

- Critical Regions
  - Start with *GetStringCritical* or *GetPrimitiveArrayCritical*
  - End with *ReleaseStringCritical* or *ReleasePrimitiveArrayCritical*
  - Native code must not call other JNI functions in between
- The call to CallVoidMethod here is in a critical region, a problem.
- Safe to overlap, but it could make programmers make more mistakes than usual

Problem 4: Calling a JNI Function in a Critical Region (contd.)

- Overlapped Critical Regions are OK
  - But a lot of care should be taken!

```
jchar *s1, *s2;
s1 = (*env)->GetStringCritical(env, jstr1);
if (s1 == NULL) {
    ... /* error handling */
}
s2 = (*env)->GetStringCritical(env, jstr2);
if (s2 == NULL) {
    (*env)->ReleaseStringCritical(env, jstr1, s1);
    ... /* error handling */
}
...     /* use s1 and s2 */
(*env)->ReleaseStringCritical(env, jstr1, s1);
(*env)->ReleaseStringCritical(env, jstr2, s2);
```

Problem 4: Calling a JNI Function in a Critical Region (contd.)

- Overlapped Critical Regions are OK
  - But a lot of care should be taken!

```
jchar *s1, *s2;
s1 = (*env)->GetStringCritical(env, jstr1);
if (s1 == NULL) {
    ... /* error handling */
}
s2 = (*env)->GetStringCritical(env, jstr2);
if (s2 == NULL) {
    (*env)->ReleaseStringCritical(env, jstr1, s1);
    ... /* error handling */
}
...     /* use s1 and s2 */
(*env)->ReleaseStringCritical(env, jstr1, s1);
(*env)->ReleaseStringCritical(env, jstr2, s2);
```

Problem 4: Calling a JNI Function in a Critical Region (contd.)

- Overlapped Critical Regions are OK
  - But a lot of care should be taken!

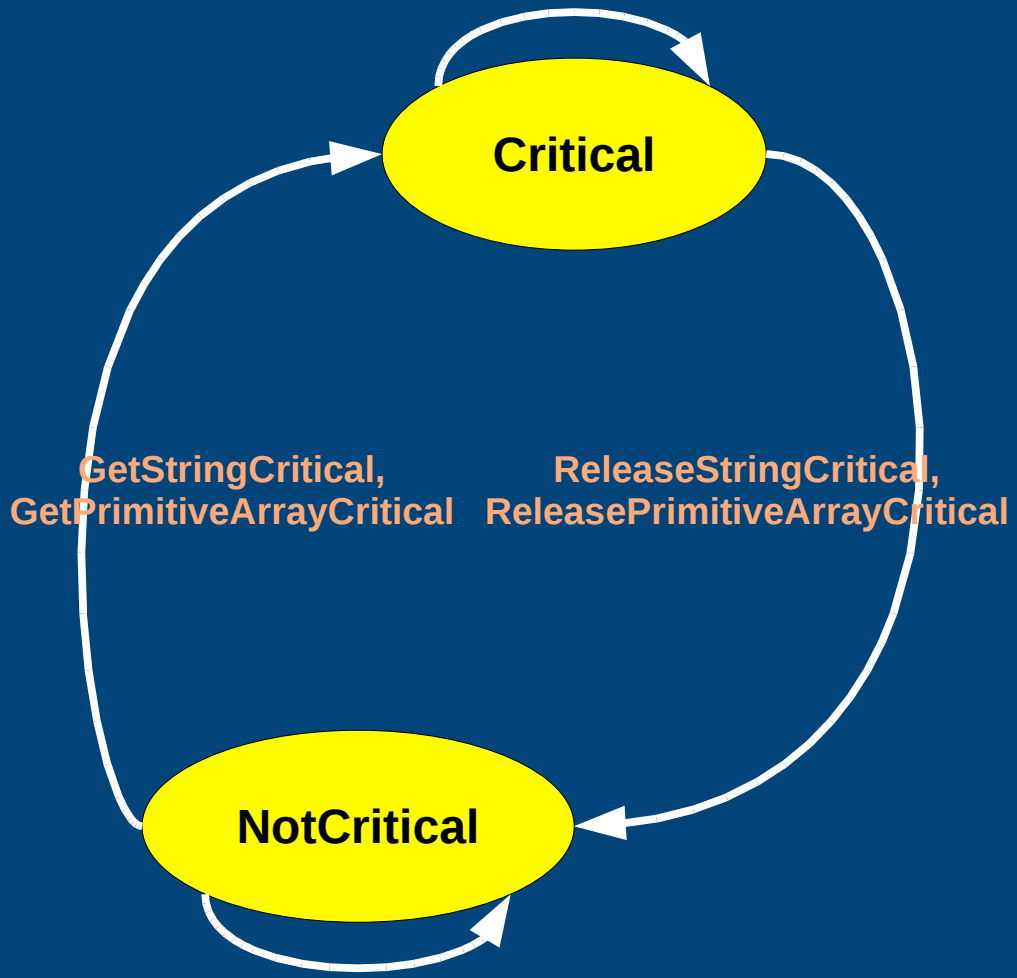
```
jchar *s1, *s2;
s1 = (*env)->GetStringCritical(env, jstr1);
if (s1 == NULL) {
    ... /* error handling */
}
s2 = (*env)->GetStringCritical(env, jstr2);
if (s2 == NULL) {
    (*env)->ReleaseStringCritical(env, jstr1, s1);
    ... /* error handling */
}
...     /* use s1 and s2 */
(*env)->ReleaseStringCritical(env, jstr1, s1);
(*env)->ReleaseStringCritical(env, jstr2, s2);
```

Problem 4: Calling a JNI Function in a Critical Region (contd.)

- Overlapped Critical Regions are OK
  - But a lot of care should be taken!

```
jchar *s1, *s2;
s1 = (*env)->GetStringCritical(env, jstr1);
if (s1 == NULL) {
    ... /* error handling */
}
s2 = (*env)->GetStringCritical(env, jstr2);
if (s2 == NULL) {
    (*env)->ReleaseStringCritical(env, jstr1, s1);
    ... /* error handling */
}
...     /* use s1 and s2 */
(*env)->ReleaseStringCritical(env, jstr1, s1);
(*env)->ReleaseStringCritical(env, jstr2, s2);
```

Addressing Problem 4



- Typestate analysis
- Not complete though

Addressing Problem 4: Why approach is not complete

Critical

```
jchar *s1, *s2;
s1 = (*env)->GetStringCritical(env, jstr1);
if (s1 == NULL) {
    ... /* error handling */
}
s2 = (*env)->GetStringCritical(env, jstr2);
if (s2 == NULL) {
    (*env)->ReleaseStringCritical(env, jstr1, s1);
    ... /* error handling */
}
...     /* use s1 and s2 */
(*env)->ReleaseStringCritical(env, jstr1, s1);
(*env)->ReleaseStringCritical(env, jstr2, s2);
```

NotCritical

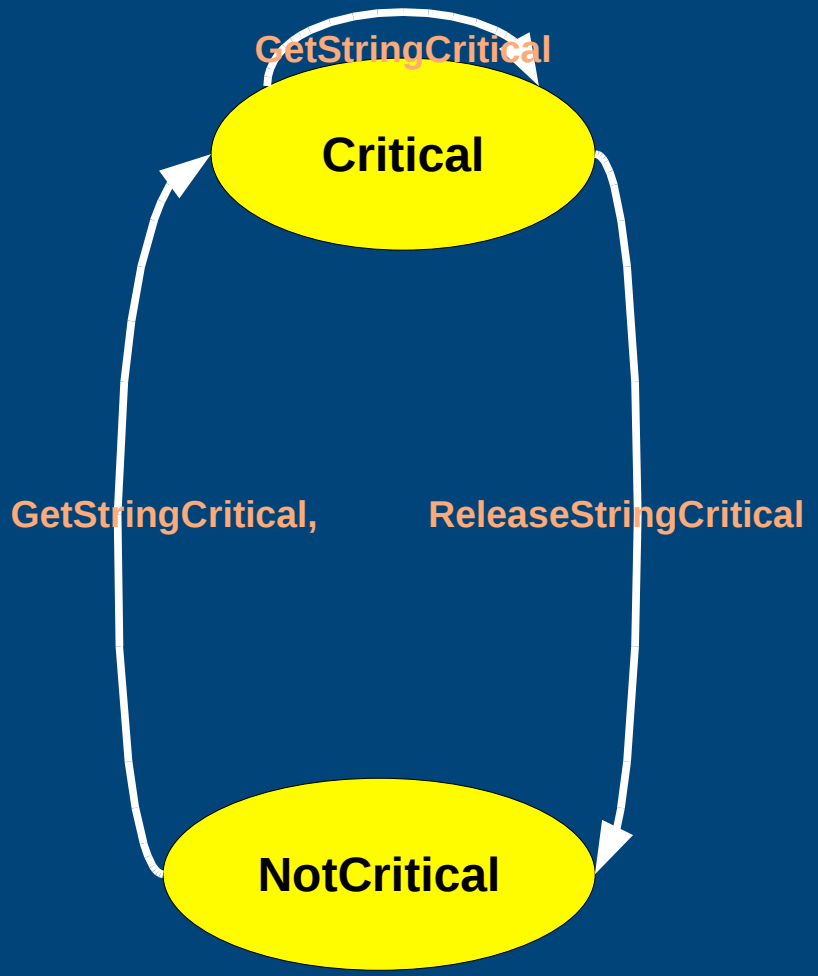
Addressing Problem 4: Why approach is not complete


```
jchar *s1, *s2;
s1 = (*env)->GetStringCritical(env, jstr1);
if (s1 == NULL) {
    ... /* error handling */
}
s2 = (*env)->GetStringCritical(env, jstr2);
if (s2 == NULL) {
    (*env)->ReleaseStringCritical(env, jstr1, s1);
    ... /* error handling */
}
...     /* use s1 and s2 */
(*env)->ReleaseStringCritical(env, jstr1, s1);
(*env)->ReleaseStringCritical(env, jstr2, s2);
```

Addressing Problem 4: Why approach is not complete


```
jchar *s1, *s2;
s1 = (*env)->GetStringCritical(env, jstr1);
if (s1 == NULL) {
    ... /* error handling */
}
s2 = (*env)->GetStringCritical(env, jstr2);
if (s2 == NULL) {
    (*env)->ReleaseStringCritical(env, jstr1, s1);
    ... /* error handling */
}
...     /* use s1 and s2 */
(*env)->ReleaseStringCritical(env, jstr1, s1);
(*env)->ReleaseStringCritical(env, jstr2, s2);
```

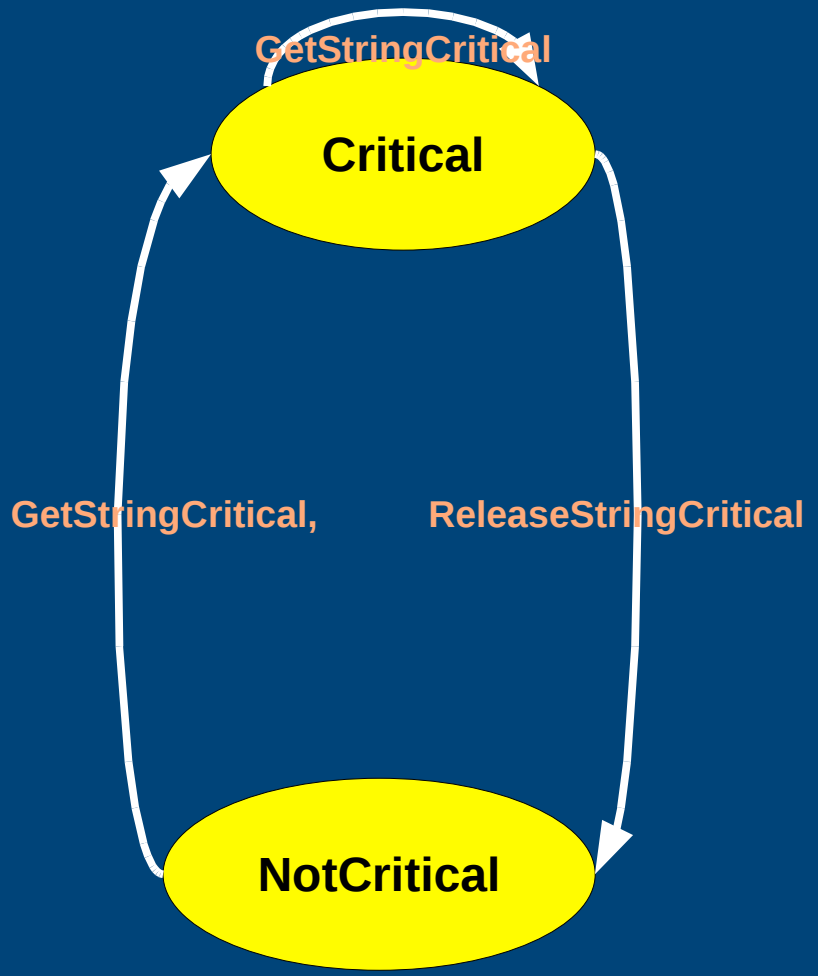
Addressing Problem 4: Why approach is not complete




```
jchar *s1, *s2;
s1 = (*env)->GetStringCritical(env, jstr1);
if (s1 == NULL) {
    ... /* error handling */
}
s2 = (*env)->GetStringCritical(env, jstr2);
if (s2 == NULL) {
    (*env)->ReleaseStringCritical(env, jstr1, s1);
    ... /* error handling */
}
...     /* use s1 and s2 */
(*env)->ReleaseStringCritical(env, jstr1, s1);
(*env)->ReleaseStringCritical(env, jstr2, s2);
```

Both ReleaseStringCritical calls correspond to the first GetStringCritical call (i.e. s1). One of these will be invoked, as the case may be.

Addressing Problem 4: Why approach is not complete




```
jchar *s1, *s2;
s1 = (*env)->GetStringCritical(env, jstr1);
if (s1 == NULL) {
    ... /* error handling */
}
s2 = (*env)->GetStringCritical(env, jstr2);
if (s2 == NULL) {
    (*env)->ReleaseStringCritical(env, jstr1, s1);
    ... /* error handling */
}
...     /* use s1 and s2 */
(*env)->ReleaseStringCritical(env, jstr1, s1);
(*env)->ReleaseStringCritical(env, jstr2, s2);
```

That we are back in “NotCritical”, is it OK *not* to have this call? Regardless of many GetStringCriticalCalls, just *one* call to ReleaseStringCritical will transition to “NotCritical”. *Approach is therefore incomplete.*

Implementation

- Typestate analysis implemented on BEAM (Brand, 2000)
  - BEAM has extensibility for typestate checking by a user-defined typestate configuration
- Syntax checker also implemented on BEAM
  - BEAM allows developers to implement their own checkers to analyze BEAM's internal representations (flowgraph, AST)
  - Selected to use flowgraph
    - Allows the same representation for both C and C++
    - Brings more efficiency

Experimental Results

- Executed the tool on 4 open-source projects
  - grep'ed for files containing “JNIEnv”

Table 2: Benchmark programs

	revision/ CVS date	description	# of files	# of lines
Harmony [1]	r566512	Java class library	159	53159
Java Gnome [9]	411	Java interface to Gnome	20	3670
Gnu Classpath [7]	2008-01-22	Java class library	69	27364
Mozilla Firefox [5]	2008-01-22	web browser	22	22427

Experimental Results (contd.)

- Linux 2.6 on an IBM IntelliStation M Pro (Intel Core 2 2.66 GHz, 3 GB RAM)
- Most frequent: error checking

	Harmony	Java Gnome	Gnu Classpath	Mozilla Firefox	Total	
Non JNI	84	3	12	15	114	
Error Checking	22	2	22(10)	9(3)	55	86
Virtual Machine Resource	18	0	7	0	25	
Using Invalid Local References	4	0	0	0	4	
JNI Function Call In a Critical Region	2 (1)	N/A	N/A	N/A	2	
Time (mm:ss)	31:59	0:12	1:59	2:02		

Experimental Results (contd.)

- BEAM with the JNI analysis generated 43% [86/(114+86)] of all bug reports
- Reported bugs increased by 76% (86/114)
- Executed BEAM without JNI analysis features; it ran at the same speed: *JNI analysis not an overhead!*

	Harmony	Java Gnome	Gnu Classpath	Mozilla Firefox	Total	
Non JNI	84	3	12	15	114	
Error Checking	22	2	22(10)	9(3)	55	86
Virtual Machine Resource	18	0	7	0	25	
Using Invalid Local References	4	0	0	0	4	
JNI Function Call In a Critical Region	2 (1)	N/A	N/A	N/A	2	
Time (mm:ss)	31:59	0:12	1:59	2:02		

Summary

- Static analysis techniques for JNI programs to detect:
  - Mistakes of error checking
  - Memory leaks
  - Invalid uses of local references
  - JNI function calls in critical regions
- Methods used
  - Typestate analysis
  - Syntax checking
- Implemented on BEAM
- Found 86 bugs; no overhead; increased bug reports by 76%

Future Work

- Address the following problems
  - Terminating Unicode Strings
  - Violating Access Control Rules
  - Disregarding Internationalization
  - Excessive Local Reference Creation