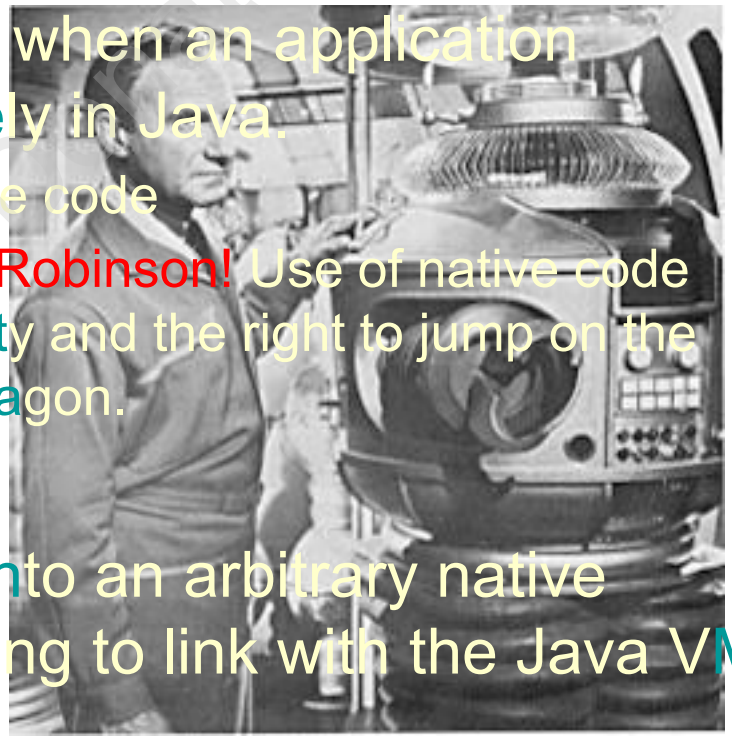
Native Code

"…there is no point in sacrificing portability for a meaningless speed improvement; don't go native until you determine that you have no other choice."

Rule #3: don't optimize until you really know what needs to be optimized.

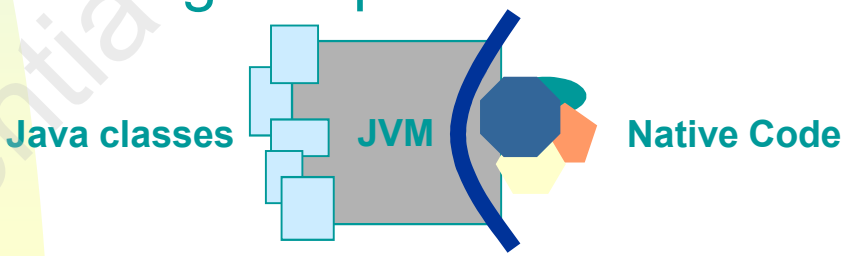
Native Code

- Two aspects:
  - Java Native Interface *(JNI)*
    - handle those situations when an application cannot be written entirely in Java.
      - Interfacing Java to native code
      - Warning! Warning! Will Robinson! Use of native code leads to loss of portability and the right to jump on the 100% pure Java bandwagon.
  - Java Invocation API
    - can load the Java VM into an arbitrary native application without having to link with the Java VM source code
  - see if you can use a JIT before resorting to native code...



Native Code

- Java Native Interface *(JNI)*
  - may:
    - need to access platform-dependent features
    - need to interface with legacy code
    - have time/resource constraints that Java can't handle
  - important to have a standard interface:
    - ensures that same native library can work with all JVMs on a given platform



    - Microsoft: "Standard? Ho Ho Ho! You need RNI"

Native Code

- 'native' Java keyword
  - says "get this from outside the Java environment"

```
// provides access to UNIX system calls
class UNIXSysCalls
  {
  public native static int chmod (String path, int mode);
  … (others)
  }
```

  - native code used in the normal fashion:

```
result = UNIXSysCalls.chmod ("/home/bob/SecretStuff", 0600);
```

  - implementation in C++:

```
#include <jni.h>
extern "C" jint Java_UNIXSysCalls_chmod
  (JNIEnv *env, jclass c, jstring path, jint mode)
  {
  char *cpath = env -> GetStringUTFChars (path);
  jint result = chmod (cpath, mode);
  env -> ReleaseStringUTFChars (path, cpath);
  return (result);
  }
```

Native Code

- some salient points:
  - name mangling:
    - the prefix *Java_*
    - a (mangled fully-qualified) class name
    - an underscore ("_") separator
    - a mangled method name
  - JNIEnv is always the first argument
    - points to lookup table of JNI functions
      - `GetStringUTFChars, etc…`
    - table layout is COM-compliant: *"This means that, as soon as cross-platform support for COM is available, the JNI can become a COM interface to the Java VM."*
  - second argument:
    - for a nonstatic native method: a reference to the object
    - for a static native method: a reference to its Java class

Native Code

- simple development process:
  - write Java class with native method definitions
  - compile to produce a class file
  - create C/C++ signatures
    - javah -jni filename
  - write C/C++ code corresponding to the generated signatures in the header file
  - create a shared library from the C/C++ code
- use:
  - same as any other Java application
  - need to have shared library in CLASSPATH

Native Code

- some security ramifications:
  - applets can't use native methods
    - 'cos they can't access DLLs

  - a wayward bit of native code can bring the whole shooting match crashing down
    - *"native methods are a significant security risk for Java programs. The C runtime system has no protection against array bounds errors, indirection through bad pointers, and so on. It is particularly important that programmers of native methods handle all error conditions to preserve the integrity of the Java system."*
  - JNI allows native methods to raise arbitrary Java exceptions…native code may also handle outstanding Java exceptions

Native Code

- Java Invocation API
  - JDK 1.1's JVM is a DLL or shared library
    - not on all platforms, 'though
  - allows a C/C++ application to create a JVM within itself
    - remember TCL?

Native Code

```
#include <jni.h>

JavaVM *jvm;              // denotes a Java VM
JNIEnv *env;              // pointer to native method interface
JDK1_1InitArgs vm_args; // JDK 1.1 VM initialization arguments

// Get the default initialization arguments and set the class path
JNI_GetDefaultJavaVMInitArgs (&vm_args);
vm_args.classpath = ...;

// load and initialize a Java VM, return a JNI interface pointer in env
JNI_CreateJavaVM (&jvm, &env, &vm_args);

// invoke the Main.test method using the JNI
jclass cls = env -> FindClass ("Main");
jmethodID mid = env -> GetStaticMethodID (cls, "test", "(I)V");
env -> CallStaticVoidMethod (cls, mid, 100);

jvm -> DestroyJavaVM ();
```

mangled signature

Native Code

- signatures
  - to call an arbitrary Java method, you need to learn the rules for ‘mangling’ the names of method signatures
  - representation of method’s name & parameters
    - *“there’s no rational reason why programmers are forced to use this mangling scheme…using the mangled signatures lets you partake in the mystique of programming close to the virtual machine.”*

Native Code

- method parameters and return type
  - long f (float i, String [] s) ==
    "(F[Ljava/lang/String;)J"

  - Z, V, B, C, D, S, F, I, J, L*classname*;
  - [indicates array
- so, "(I)V" ==
  - void test (int x)