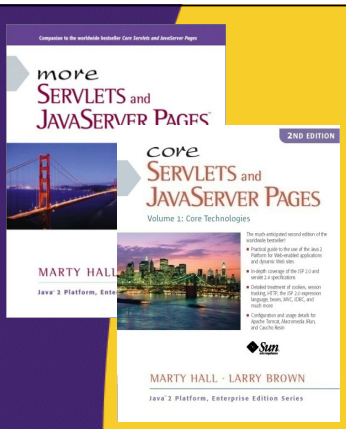
© 2012 Marty Hall

Invoking Native Applications from Java

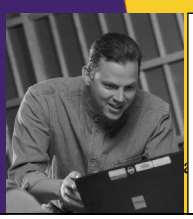
Originals of Slides and Source Code for Examples:
http://courses.coreservlets.com/Course-Materials/java.html

Customized Java EE Training: http://courses.coreservlets.com/
Java, JSF 2, PrimeFaces, Servlets, JSP, Ajax, jQuery, Spring, Hibernate, RESTful Web Services, Hadoop, Android.
Developed and taught by well-known author and developer. At public venues or onsite at *your* location.

© 2012 Marty Hall



For live Java EE training, please see training courses at http://courses.coreservlets.com/.
JSF 2, PrimeFaces, Servlets, JSP, Ajax (with jQuery), GWT, Android development, Java 6 and 7 programming, SOAP-based and RESTful Web Services, Spring, Hibernate/JPA, XML, Hadoop, and customized combinations of topics.



Taught by the author of *Core Servlets and JSP*, *More Servlets and JSP*, and this tutorial. Available at public venues, or customized versions can be held on-site at your organization. Contact hall@coreservlets.com for details.

Agenda

- Integration options
- Invoking native programs
- Calling native functions

Linking to Programs in Other Languages

- **Invoke the program at the OS level**
  - Use ProcessBuilder to invoke a random program, pass in arguments via the standard input, and read results via the standard output
    - Pros: easy to set up, can call arbitrary programs
    - Cons: limited argument passing, slow: big startup overhead
- **Use sockets**
  - Use regular sockets to exchange data
    - Pros: fast if on same machine, can split in future
    - Cons: work to set up on both ends, need to parse data
- **Use native methods**
  - Use JNI to link C and Java code
    - Pros: fast: suitable for fine-grained interactions
    - Cons: lots of work to set up, requires C, C++, or assembly

Invoking Native Programs

1. **Create a ProcessBuilder**
   - ProcessBuilder builder =
     new ProcessBuilder("program", "argument");
   - Note that environment variables such as PATH are *not* automatically set, so you should use full path to program
2. **Start the process**
   - builder.start();

Options

- Wait for process to terminate
  - Process p = builder.start();
  - int returnCode = p.waitFor();
- Examine return code later
  - int returnCode = p.exitCode();

Example: Starting Internet Explorer

- **Full path to Internet Explorer:**
  - C:\Program Files\Internet Explorer\iexplore.exe
    - Must use \\ to get \ in Java strings
    - The .exe extension can be omitted on Windows
- **Internet Explorer accepts command line arguments**
  - The initial URL to be displayed
    - Overrides homepage

Example: Code

```
public class InvokeIE {
  public static void main(String[] args) {
    String url = "http://www.google.com/";
    if (args.length > 0) {
      url = args[0];
    }
    try {
      ProcessBuilder builder =
        new ProcessBuilder(
          "C:\\Program Files\\Internet Explorer\\iexplore",
          url);
      builder.start();
    } catch(Exception e) {
      System.out.println(e);
    }
  }
}
```

Example: Results

```
DOS> java InvokeIE http://www.jhuapl.edu/
```

Reading Results from Native Programs

1. **Create a ProcessBuilder**
   - ProcessBuilder builder =
     new ProcessBuilder("program", "argument");
2. **Start the process (referencing Process)**
   - Process p = builder.start();
3. **Attach a Reader (to input, not output!)**
   - BufferedReader reader =
     new BufferedReader (new InputStreamReader
     (p.getInputStream()));
4. **Read results**
   - Call reader.readLine() until result is null
5. **Close the stream**
   - reader.close();

Example: Invoking the Unix "ls" Command

```
import java.io.*;

public class InvokeLS {
  public static void main(String[] args) {
    String flags = "-al";
    if (args.length > 0) {
      flags = args[0];
    }
    try {
      ProcessBuilder builder =
        new ProcessBuilder("/usr/bin/ls", flags);
      Process process = builder.start();
```

Example: Invoking the Unix "ls" Command (Continued)

```
      BufferedReader reader =
        new BufferedReader
          (new InputStreamReader
            (process.getInputStream()));
      String line;
      while((line = reader.readLine()) != null) {
        System.out.printf("Output: %s%n", line);
      }
      reader.close();
      int status = process.exitValue();
      if (status != 0) {
        System.out.printf("Error: process exited with %d.%n",
                          status);
      }
    } catch(Exception e) {
      System.out.println(e);
    }
  }
}
```

Example: Invoking the Unix "ls" Command (Indented Results)

```
Unix> java InvokeLS
Output: total 12
Output: drwxr-xr-x   2 hall     instruct
         512 Nov 26 10:00 .
Output: drwxr-xr-x   6 hall     instruct
        2048 Nov 26 09:38 ..
Output: -rw-r--r--   1 hall     instruct
        1257 Nov 26 10:00 InvokeLS.class
Output: -rw-r--r--   1 hall     instruct
         846 Nov 26 10:00 InvokeLS.java
```

Calling Native Methods

- **You can call C functions from Java**
  - C++ functions must be declared "extern C"
  - You cannot directly call FORTRAN, but C can easily (?) act as intermediary
    - See http://www.csharp.com/javacfort.html
- **You can call Java functions from C**
- **Much more work**
  - Very tedious and low-level programming on both C and Java sides
- **Much more powerful**
  - Can pass real data types (not just strings)
  - Doesn't start a new OS process for each call
- **More details**
  - General: http://java.sun.com/docs/books/tutorial/native1.1/
  - MATLAB: http://www.mathworks.com/access/helpdesk/help/techdoc/matlab_external/f44062.html

Using Native Methods

1. **Create Java class with native method**
   - Method stub with declaration **`native`**
   - Load shared library via System.loadLibrary
2. **Compile the Java code**
   - Use javac normally
3. **Create a header file for the Java class**
   - Use "javah -jni *ClassName*"
4. **Write a C program with designated function**
   - Must include *ClassName*.h and jni.h
5. **Compile C program into shared library**
   - Include path must incorporate *javahome*/include and *javahome*/include/*operatingsystem*
6. **Run the Java program**
   - Use java normally

Creating Java Class

- **Must use native declaration**
- **Must load shared library before invoking native method**

```
public class HelloWorld {
  static {
    System.loadLibrary("hello");
  }

  public native void displayHelloWorld();

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

Creating a Header File

> javac HelloWorld.java
> javah -jni HelloWorld

- **Result: HelloWorld.h**

```
/* DO NOT EDIT THIS FILE - it is machine generated */
#include <jni.h>
/* Header for class HelloWorld */

#ifndef _Included_HelloWorld
#define _Included_HelloWorld
#ifdef __cplusplus
extern "C" {
#endif
...
JNIEXPORT void JNICALL Java_HelloWorld_displayHelloWorld
  (JNIEnv *, jobject);
...
```

Creating C Program

```
#include <jni.h>
#include "HelloWorld.h"
#include <stdio.h>

JNIEXPORT void JNICALL
Java_HelloWorld_displayHelloWorld
                          (JNIEnv *env, jobject obj)
{
    printf("Hello world!\n");
    return;
}
```

Compiling C Program Into Shared Library

- **Must include the .h files for JNI**
  - General
  - OS-specific

```
Solaris> gcc
       -I/usr/java1.5/include
       -I/usr/java1.5/include/solaris
       HelloWorldImp.c
       -o libhello.so
```

Invoking Java Program

```
Solaris> java HelloWorld
Hello world!
```

Mapping Java Types to C Types (Primitives)

Java Type	Native Type	Size in Bits
boolean	jboolean	8, unsigned
byte	jbyte	8
char	jchar	16, unsigned
short	jshort	16
int	jint	32
long	jlong	64
float	jfloat	32
double	jdouble	64
void	void	

Mapping Java Objects to C

- All calls are call by reference
- All Objects are jobject in C
- A few predefined jobject subtypes
  - jstring
  - jintArray, jshortArray, jlongArray
  - jfloatArray, jdoubleArray
  - jcharArray
  - jbyteArray
  - jbooleanArray
  - jobjectArray

Calling Java Methods from C

- Call the function GetObjectClass
- Call GetMethodID
- Call CallVoidMethod

```
JNIEXPORT void JNICALL
Java_Callbacks_nativeMethod(JNIEnv *env, jobject obj,
                            jint depth) {
  jclass cls = (*env)->GetObjectClass(env, obj);
  jmethodID mid = (*env)->GetMethodID(env, cls,
                                      "callback", "(I)V");
  if (mid == 0) { return; }
  printf("In C, depth = %d, about to enter Java\n",
                                                depth);
  (*env)->CallVoidMethod(env, obj, mid, depth);
  printf("In C, depth = %d, back from Java\n", depth);
```

Summary

- **Invoking operating-system programs is straightforward**
  - Use ProcessBuilder.start() to start program, optionally with command-line arguments
  - You can read standard output
    - Attach BufferedReader to input stream
- **You can use sockets to communicate**
  - See earlier lectures
  - Very fast if both programs are on same machine
- **JNI provides tightest integration and highest-performance result**
  - Very low-level and tedious. Hard to maintain.

© 2012 Marty Hall

Questions?

JSF 2, PrimeFaces, Java 7, Ajax, jQuery, Hadoop, RESTful Web Services, Android, Spring, Hibernate, Servlets, JSP, GWT, and other Java EE training.

Customized Java EE Training: http://courses.coreservlets.com/
Java, JSF 2, PrimeFaces, Servlets, JSP, Ajax, jQuery, Spring, Hibernate, RESTful Web Services, Hadoop, Android.
Developed and taught by well-known author and developer. At public venues or onsite at *your* location.