Cogent Logic Ltd.

High Quality Hands-On Training for Software Developers

Summer 2003 Cryptography Training Events

Cryptography for Java Developers runs in central London on:

- Wednesday 26 June 2013
- Saturday 29 June 2013

Cryptography for C Developers runs in central London on:

- Wednesday 10 July 2013
- Saturday 13 July 2013

To register for these courses, please:

- Follow the link at www.cogentlogic.com/London or
- Call freephone: 08000 438 478

- 1 Introduction to Cryptography
- 2 Cryptographic Service Providers
- 3 Symmetric Key Cryptography
- 4 Symmetric Key Cryptography for Android and iOS
- 5 Asymmetric Key Cryptography
- 6 Digital Signatures
- 7 Authenticated Encryption
- 8 Digital Certificates
- 9 PKI
- 10 Key Stores and Trust Stores
- 11 SSL and TLS (JSSE)
- 12 Accessing LDAP Servers with JNDI
- 13 Certificate Revocation Lists and OCSP
- 14 Privilege Management Infrastructure

- Android Training Courses
  - *Developing Mobile Applications with Android* is for Java programmers wishing to get up to speed on Android development.
  - *Software Development with Java* is for programmers wishing acquire a thorough grounding in Java.
- iOS Training Courses
  - *Developing Mobile Applications with iOS* is for Objective-C programmers wishing to get up to speed on iOS development.
  - *Software Development with Objective-C* is for programmers wishing acquire a thorough grounding in Objective-C.

- # Ruby on Rails Training Courses

  - *Developing Web Applications with Ruby on Rails* is for Ruby programmers wishing to get up to speed on Rails development.

  - *Software Development with Ruby* is for programmers wishing acquire a thorough grounding in Ruby.

Java Native Interface with Eclipse and Android

Jeff Lawson

Contents

Java Native Interface with Eclipse and Android

What Is JNI and Why Use It?

Jeff Lawson

Contents

What Is JNI?

- The *Java Native Interface* is a technology that enables Java developers to call C and C++ software from Java programs
- Asynchronous calls are possible with C/C++ calling back into Java
- JNI has been part of the Java platform since 1997

Why Use JNI?

- JNI is useful for:
  - Calling C/C++ libraries that off functions not available in Java, e.g. telephony, sound, graphics (images, 2D/3D animation, physics)
  - Calling platform-native code, e.g. Windows Registries
  - Re-using an existing corporate/organisation code base in C/C++
  - Improving the performance of compute-bound Java code
  - Incorporating Java solutions (JVM) into C/C++ projects

What is the Process for Using JNI?

- To make use of JNI we simply:
  - Write C/C++ code intended to be called from Java; this is often purpose-built Java-access code, e.g.
  - In Java, declare and call native methods that call into C/C++, e.g.
  - Compile the C/C++ code into a library, e.g.
  - Load the library into the Java program at runtime, e.g.
  - Compile and run the Java program

JNI Documentation

- The official source for JNI documentation is:

  `docs.oracle.com/javase/7/docs/technotes/guides/jni/`

- The book The Java Native Interface *Programmer's Guide and Specification* is available in print and as a free web download that Sun used to provide but Oracle seem not to, so search for it online!

Java Native Interface with Eclipse and Android

Compiling C/C++ Programs

Jeff Lawson

Contents

Compiling C/C++

- C and C++ source code typically comprises:
  - One or more implementation file(s), with file name extensions `.c` and `.cpp`, respectively
  - Associated header file(s) with `.h` file name extension
- The collection of source code files must be:
  - Compiled into object code
  - Linked with zero or more libraries into a final library or executable
- JNI Java code calls into one or more libraries:
  .o files on Unixes (Linux and Mac OS X), .dll files on Windows
- Tool collections, like a compiler plus a linker are known as a *toolchain*

GNU Compiler Collection

- *GNU C Compiler* (GCC) was originally developed by Richard Stallman for use with the GNU Project, a free Unix-like operating system

- GCC has since grown to support C++, Obj-C, Java and more; it is now known as the *GNU Compiler Collection* and is available from:

  `gcc.gnu.org`

- GCC is available for:

  - Mac OS X as an optional Xcode component, *Command Line Tools*
  - Linux with `up2date`, `yum` or `apt-get`, as applicable to the distro
  - Windows with MinGW (not Cygwin because it targets Cygwin)

GCC on Mac OS X

- To install GCC on Mac OS X through Xcode:
  - Xcode menu, Preferences..., Downloads tab, Components tab
  - Select Command Line Tools and click the Install button

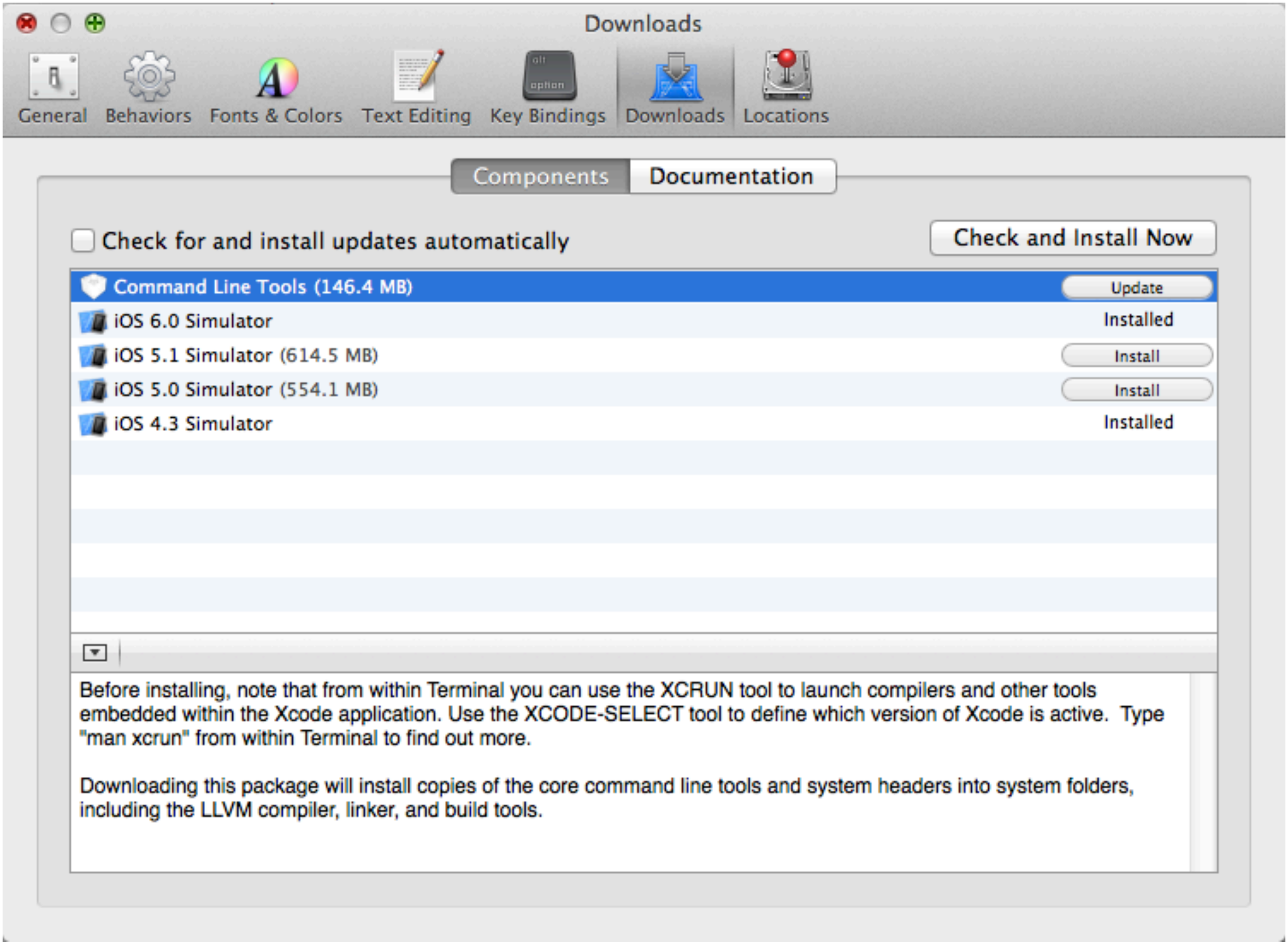
...

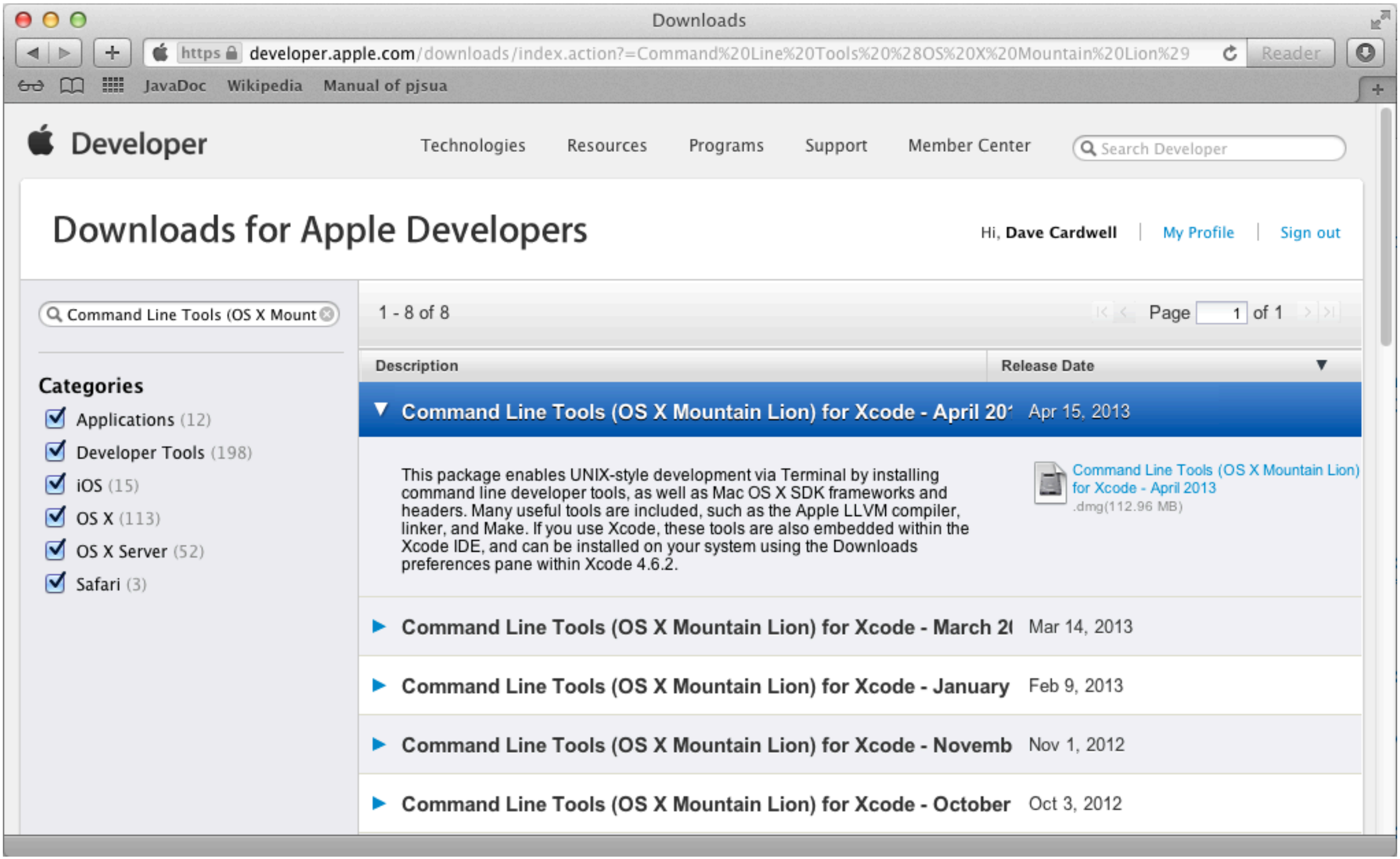
...

- To Install GCC on Mac OS X without Xcode, download the Command Line Tools from:

```
developer.apple.com/downloads/index.action?=
    Command%20Line%20Tools%20%28OS%20X%20Mountain%20Lion%29
```

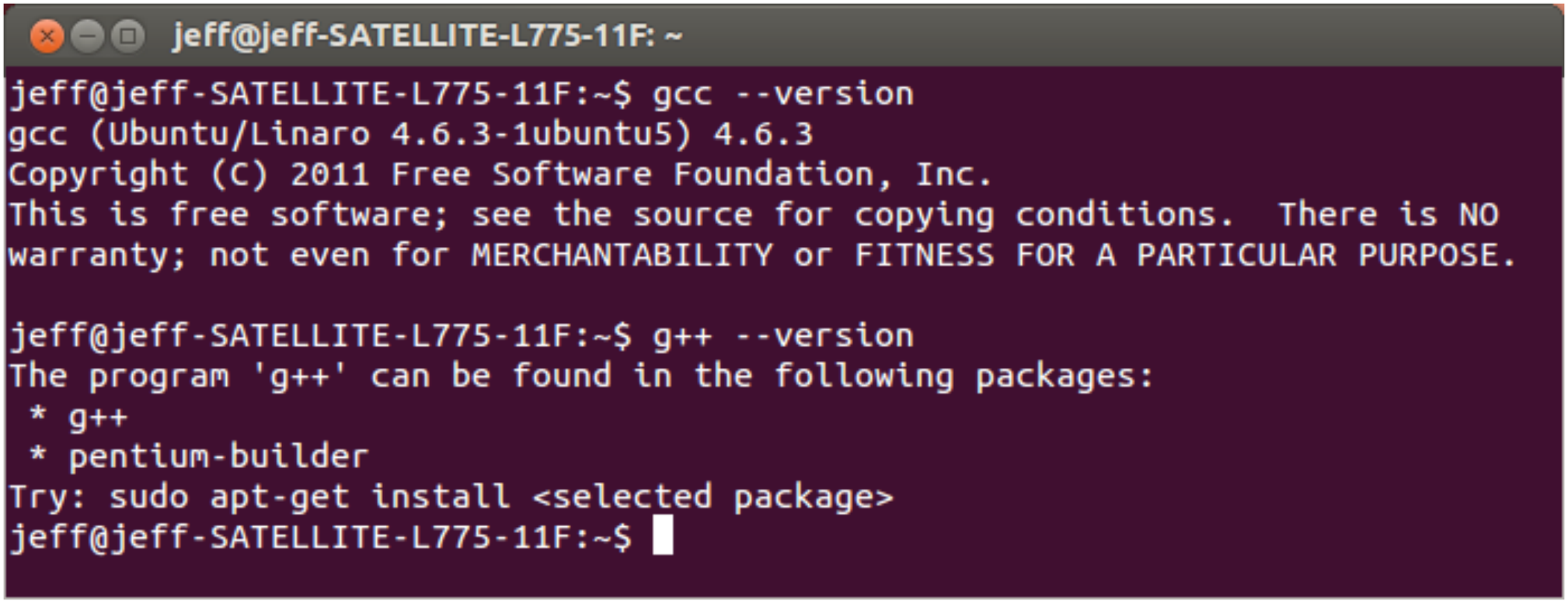
...





GCC on Linux

- First, check for the presence of GCC by entering `gcc --version` then, if GCC is absent, install it
- To install GCC on Red Hat Enterprise, enter `up2date gcc`
- For CentOS / Fedora Core, enter `yum install gcc`
- For Debian / Ubuntu, enter `sudo apt-get install gcc`
- Similarly, for the C++ compiler:
    - Check for the C++ compiler by entering `g++ --version`
    - If necessary, install, e.g. `sudo apt-get install g++`



```
jeff@jeff-SATELLITE-L775-11F: ~
jeff@jeff-SATELLITE-L775-11F:~$ sudo apt-get install g++
[sudo] password for jeff:
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following extra packages will be installed:
  g++-4.6 libstdc++6-4.6-dev
Suggested packages:
  g++-multilib g++-4.6-multilib gcc-4.6-doc libstdc++6-4.6-dbg libstdc++6-4.6-doc
The following NEW packages will be installed
  g++ g++-4.6 libstdc++6-4.6-dev
0 upgraded, 3 newly installed, 0 to remove and 208 not upgraded.
Need to get 8,615 kB of archives.
After this operation, 25.4 MB of additional disk space will be used.
Do you want to continue [Y/n]? Y
Get:1 http://gb.archive.ubuntu.com/ubuntu/ precise/main libstdc++6-4.6-dev amd64 4.6.3-1ubuntu5 [1,66
0 kB]
Get:2 http://gb.archive.ubuntu.com/ubuntu/ precise/main g++-4.6 amd64 4.6.3-1ubuntu5 [6,954 kB]
Get:3 http://gb.archive.ubuntu.com/ubuntu/ precise/main g++ amd64 4:4.6.3-1ubuntu5 [1,442 B]
Fetched 8,615 kB in 10s (828 kB/s)
Selecting previously unselected package libstdc++6-4.6-dev.
(Reading database ... 142534 files and directories currently installed.)
Unpacking libstdc++6-4.6-dev (from .../libstdc++6-4.6-dev_4.6.3-1ubuntu5_amd64.deb) ...
Selecting previously unselected package g++-4.6.
Unpacking g++-4.6 (from .../g++-4.6_4.6.3-1ubuntu5_amd64.deb) ...
Selecting previously unselected package g++.
Unpacking g++ (from .../g++_4%3a4.6.3-1ubuntu5_amd64.deb) ...
Processing triggers for man-db ...
Setting up g++-4.6 (4.6.3-1ubuntu5) ...
Setting up g++ (4:4.6.3-1ubuntu5) ...
update-alternatives: using /usr/bin/g++ to provide /usr/bin/c++ (c++) in auto mode.
Setting up libstdc++6-4.6-dev (4.6.3-1ubuntu5) ...
jeff@jeff-SATELLITE-L775-11F:~$
```

GCC on Windows

- To install GCC on Windows, use Minimalist GNU for Windows (MinGW) from `www.mingw.org`

- Download the installer from:
  `sourceforge.net/projects/mingw/files/Installer/mingw-get-inst/`

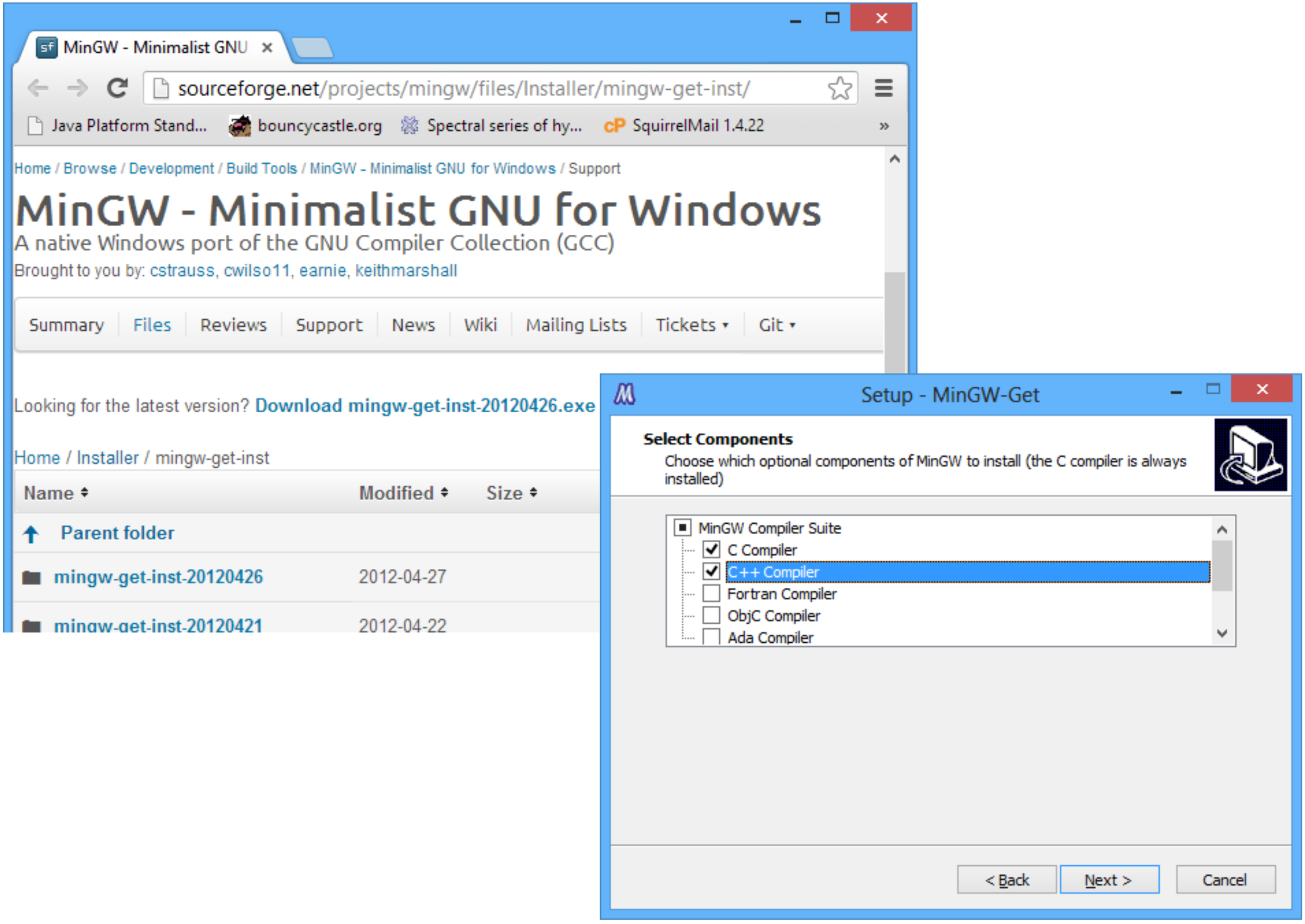
  e.g. `mingw-get-inst-20120426.exe`

- The C++ compiler is not installed by default, so select it (there is no need to install MSYS)

- Several components will downloaded from through a shell script

- Manually add `C:\MinGW\bin` to the PATH environment variable

For 64-bit support, see:
`mingw-w64.sourceforge.net`



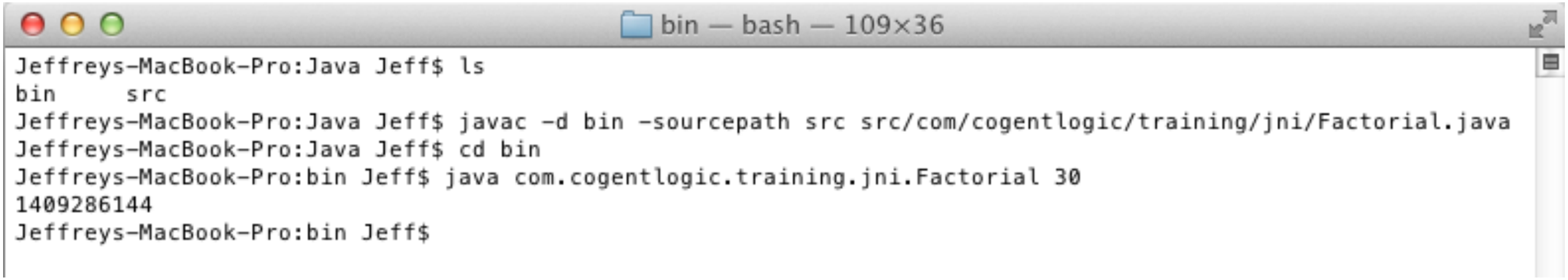
Sample Java Program

- Let's look at compilation and execution of a simple Java program

```
package com.cogentlogic.training.jni;

public class Factorial
{
  public static int evaluate(int n)
  {
     return n == 1 ? 1 : n * evaluate(n - 1);
  }

  public static void main(String[] args)
  {
     if (args.length == 1)
        System.out.println(Factorial.evaluate(
                                   Integer.parseInt(args[0])));
  }
}
```

- We compile and execute the Java program in the usual way:

```
javac -d bin -sourcepath src
      src/com/cogentlogic/training/jni/Factorial.java

java com.cogentlogic.training.jni.Factorial 30
```



Sample C Program

- Here is an equivalent program in C:

```
#include "Factorial.h"
#include <stdio.h>        // printf
#include <stdlib.h>       // atoi

int factorial(int n)
{
  return n == 1 ? 1 : n * factorial(n - 1);
}

int main(int argc, const char **argv)
{
  // Should be two arguments: the command and the parameter
  if (argc != 2)
     return -1;
  int nFactorial = factorial(atoi(argv[1]));
  printf("%d\n", nFactorial);
  return 0;
}
```

- We typically need a C header file:

```
#ifndef FACTORIAL_H_
#define FACTORIAL_H_

int factorial(int n);

#endif /* FACTORIAL_H_ */
```

- To compile the C source code `Factorial.c` in to the object code file `Factorial.o`:

```
gcc -o Factorial.o Factorial.c
```

- To execute the C program on Mac/Linux:

```
./Factorial.o 30
```

- To execute the C program on Windows:

```
.\Factorial.o 30
```

(or `Factorial.o 30`)

Sample C++ Program

- Here an equivalent program in C++:

```
#include "Factorial.h"
#include <stdlib.h>      // atoi
#include <iostream>      // std::cout, etc.

Factorial::Factorial()
{
}

Factorial::~Factorial()
{
}

int Factorial::evaluate(int n)
{
  return n == 1 ? 1 : n * evaluate(n - 1);
}
```

- (C++ code continued):

```
int main(int argc, const char **argv)
{
  // Should be two arguments: the command and the parameter
  if (argc != 2)
     return -1;

  Factorial* factorial = new Factorial();
  int nFactorial = factorial->evaluate(atoi(argv[1]));

  std::cout << nFactorial << std::endl;

  return 0;
}
```

- We typically need a header file:

```
#ifndef FACTORIAL_H_
#define FACTORIAL_H_

class Factorial
{
public:
  Factorial();
  virtual ~Factorial();
  int evaluate(int n);
};

#endif /* FACTORIAL_H_ */
```

- To compile the C++ source code `Factorial.cpp` in to the object code file `Factorial.o`:

```
g++ -o Factorial.o Factorial.cpp
```

- To execute the C++ program:

```
./Factorial.o 30
```

Java Native Interface with Eclipse and Android

Writing C Functions and C++ Methods Callable from Java

Jeff Lawson

Contents

- JNI Code
- Calling C from Java
- Calling C++ from Java

JNI Code

- Java Native Interface documentation can be found at:

  `docs.oracle.com/javase/7/docs/technotes/guides/jni`

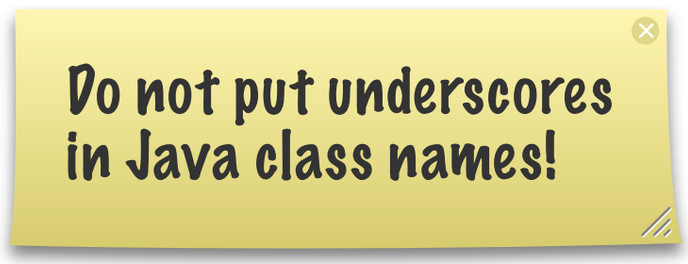
- JNI requires C and C++ functions to be named as follows:

  `Java_<package_name>_<Java_class_name>_<function_name>`

  where `<package_name>` contains underscores in place of periods

- For example, for a Java class called `Test` in the package `test.greet` needing to make a call to a native method it knows as `greeting`, the C/C++ function is named:

  `Java_test_greet_Test_greeting`



- C++ functions must be exposed with C linkage using `extern "C"`

Calling C from Java

- Sample C header file: (*MathsC.h*)

```
#ifndef MATHSC_H_
#define MATHSC_H_

#include <jni.h>

JNIEXPORT jint JNICALL
    Java_com_cogentlogic_training_jni_Maths1_fibonacci(JNIEnv*,
                                                       jobject,
                                                       jint);

#endif  // MATHSC_H_
```

- Sample C implementation file: (*MathsC.c*):

```
#include "MathsC.h"

int fib(int n)
{
  // F0 = 0;  F1 = 1
  // Fn = Fn-1 + Fn-2   for n > 1
  return n <= 1 ? n : fib(n - 1) + fib(n - 2);
}

JNIEXPORT jint JNICALL
  Java_com_cogentlogic_training_jni_Maths1_fibonacci(JNIEnv* penv,
                                                     jobject obj,
                                                     jint n)
{
  return fib(n);
}
```

- This code needs to be compiled then linked as a library

- The Java code loads the library and calls its native method(s), e.g. (*Maths1.java*)

```
package com.cogentlogic.training.jni;

public class Maths1
{
  static
  {
     System.loadLibrary("MathsC");
  }

  public native int fibonacci(int n);

  public static void main(String[] args)
  {
     if (args.length == 1)
     {
        Maths1 maths1 = new Maths1();
        System.out.println("" +
                  maths1.fibonacci(Integer.parseInt(args[0])));
     }
  }
}
```

- To compile `MathC.c` on Mac OS X:

```
gcc -I/System/Library/Frameworks/JavaVM.framework/Headers
    -o MathsC.o -c MathsC.c
```

`-c` means 'compile only'
The `-I` option specifies the location of `jni.h`

- To link `MathC.o` on Mac OS X:

```
gcc -shared -o libMathsC.jnilib MathsC.o
```

The generated library must be named `lib<name>.jnilib` where `<name>` is referenced by the Java code

- To run *Maths1.java* on Mac OS X, compile the class:

```
javac -d bin -sourcepath src
      src/com/cogentlogic/training/jni/Maths1.java
```

then enter:

```
java com.cogentlogic.training.jni.Maths1 30
```

- To compile and link `MathC.c` on Linux (tested on ubuntu 12.04 LTS):

```
gcc -fPIC -I/usr/lib/jvm/java-7-oracle/include
          -I/usr/lib/jvm/java-7-oracle/include/linux
          -o libMathsC.so -shared MathsC.c
```

  x86-64 architectures require Position Independent Code for shared libraries, hence, the use of `-fPIC`

- To run *Maths1.java* on Linux, compile the class:

```
javac -d bin -sourcepath src
      src/com/cogentlogic/training/jni/Maths1.java
```

  then enter:

```
java -Djava.library.path=./ com.cogentlogic.training.jni.Maths1 30
```

- To compile and link `MathC.c` on Windows (32- and 64-bit) using 32-bit GCC and 32-bit Java*:

```
gcc -Wall -D_JNI_IMPLEMENTATION_ -Wl,--kill-at
    -I"C:/Program Files (x86)/Java/jdk1.7.0_21/include"
    -I"C:/Program Files (x86)/Java/jdk1.7.0_21/include/win32"
    -shared -o MathsC.dll MathsC.c
```

  x86-64 architectures require `-m64`

- To run *Maths1.java* on Windows, compile the class:

```
"C:\Program Files (x86)\Java\jdk1.7.0_21\bin\javac" -d bin
   -sourcepath src src/com/cogentlogic/training/jni/Maths1.java
```

  then enter:

```
"C:\Program Files (x86)\Java\jdk1.7.0_21\bin\java"
                          com.cogentlogic.training.jni.Maths1 30
```

* From `java.sun.com`, download Windows 32-bit JDK as the version *Windows x86,* e.g. `jdk-7u21-windows-i586.exe`

Calling C++ from Java

- Sample C header file: (*MathsCPP.h*)

```
#include <jni.h>

class MathsCPP
{
public:
    int fib(int n);
};

#ifdef __cplusplus
extern "C" {
#endif

JNIEXPORT jint JNICALL
  Java_com_cogentlogic_training_jni_Maths2_fibonacci(JNIEnv*,
                                                     jobject, jint);

#ifdef __cplusplus
}
#endif
```

```
#ifndef MATHSCPP_H_
omitted for lack of space
```

- Sample C++ implementation file: (*MathsCPP.cpp*):

```
#include "MathsCPP.h"

int MathsCPP::fib(int n)
                  {return n <= 1 ? n : fib(n - 1) + fib(n - 2);}

#ifdef __cplusplus
extern "C" {
#endif

JNIEXPORT jint JNICALL
 Java_com_cogentlogic_training_jni_Maths2_fibonacci(JNIEnv* penv,
                                                    jobject obj, jint n)
{
  MathsCPP* mathsCPP = new MathsCPP();
  int nFib = mathsCPP->fib(n);
  delete mathsCPP;
  return nFib;
}

#ifdef __cplusplus
}
#endif
```

- The Java code loads the library and calls its native method(s), e.g. (*Maths2.java*)

```
package com.cogentlogic.training.jni;

public class Maths2
{
   static
   {
      System.loadLibrary("MathsCPP");
   }

   public native int fibonacci(int n);

   public static void main(String[] args)
   {
      if (args.length == 1)
      {
         Maths2 maths2 = new Maths2();
         System.out.println("" +
                  maths2.fibonacci(Integer.parseInt(args[0])));
      }
   }
}
```

- To compile `MathCPP.cpp` on Mac OS X:

```
g++ -I/System/Library/Frameworks/JavaVM.framework/Headers
    -o MathsCPP.o -c MathsCPP.cpp
```

`-c` means 'compile only'
The `-I` option specifies the location of `jni.h`

- To link `MathCPP.o` on Mac OS X:

```
g++ -shared -o libMathsCPP.jnilib MathsCPP.o
```

The generated library must be named `lib<name>.jnilib` where `<name>` is referenced by the Java code

- To run *Maths2.java* on Mac OS X, compile the class:

```
javac -d bin -sourcepath src
      src/com/cogentlogic/training/jni/Maths2.java
```

then enter:

```
java com.cogentlogic.training.jni.Maths2 30
```

Java Native Interface with Eclipse and Android

Compiling C/C++ Programs with Eclipse

Jeff Lawson

Contents

Adding C/C++ to Eclipse

- Various Eclipse packages are available from:

  `www.eclipse.org/downloads/`

- You can choose a package that supports both Java and C/C++

- To add C/C++ support to a Java-based version of Eclipse:

  - Select `Install New Software...` from the `Help` menu

  - Select the releases software site,
    e.g. `Juno - http://download.eclipse.org/releases/juno`

  - Select `C/C++ Development Tools` under `Programming Languages`

  - Click the `Next >` button twice, accept the license then click `Finish`

Install

Available Software

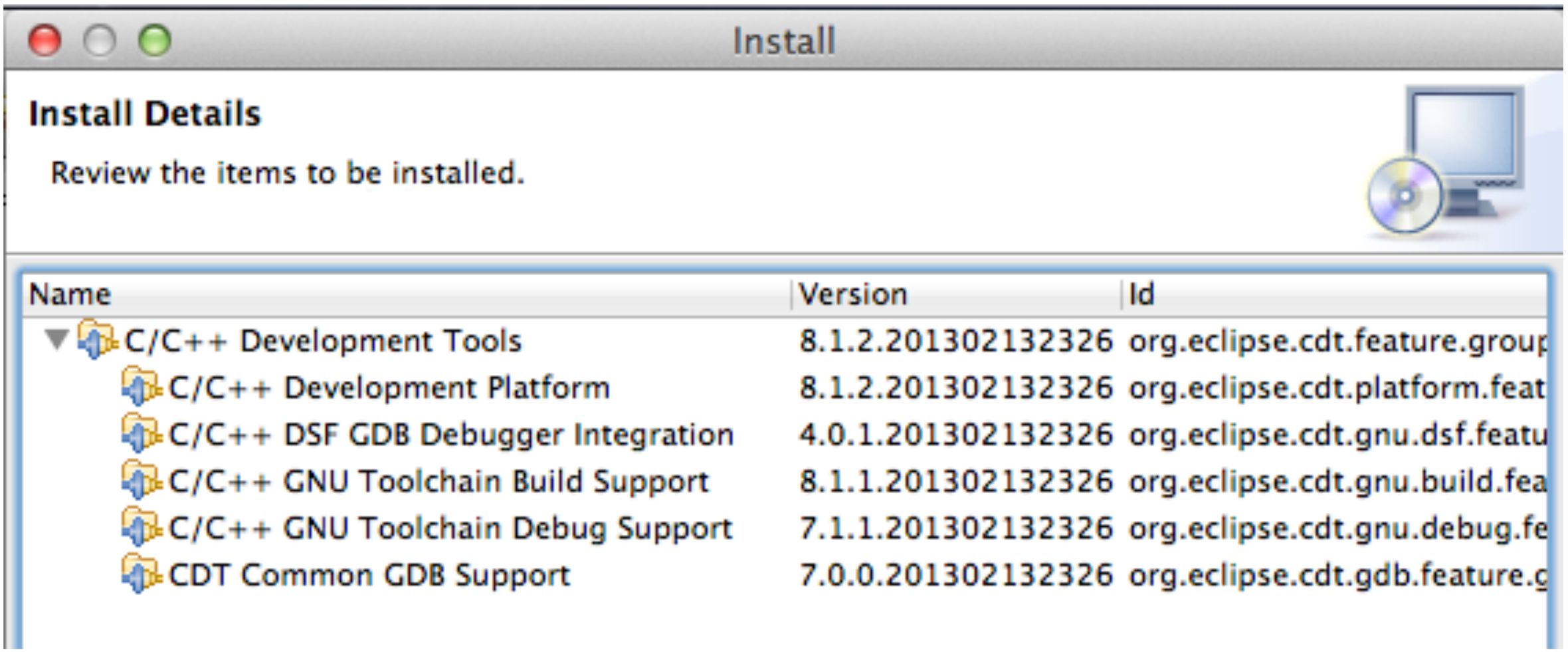
Check the items that you wish to install.

Work with: Juno - http://download.eclipse.org/releases/juno — Add...

Find more software by working with the "Available Software Sites" preferences.

type filter text

Name	Version
Modeling	
Programming Languages	
Autotools support for CDT	3.0.1.201302132326
C/C++ Call Graph Visualization	1.1.0.201302051708
C/C++ Development Tools	8.1.2.201302132326
C/C++ Development Tools SDK	8.1.2.201302132326
C/C++ Library API Documentation Hover Help	1.0.0.201302051708
C/C++ Unit Testing Support	7.0.0.201302132326
CDT Visual C++ Support	1.0.0.201302132326

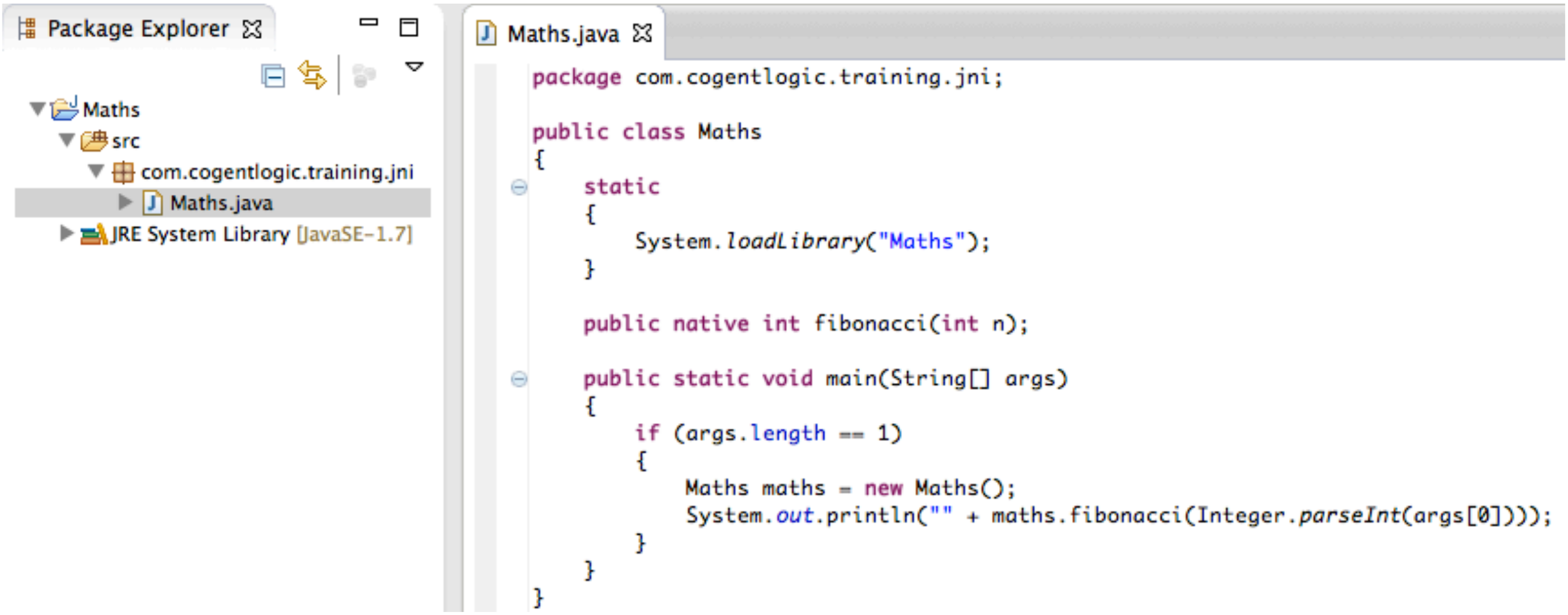


- Finally, from the `Window` menu select `Open Perspective` followed by `Other...`
- Select the `C/C++` perspective

Java and C/C++ with Eclipse

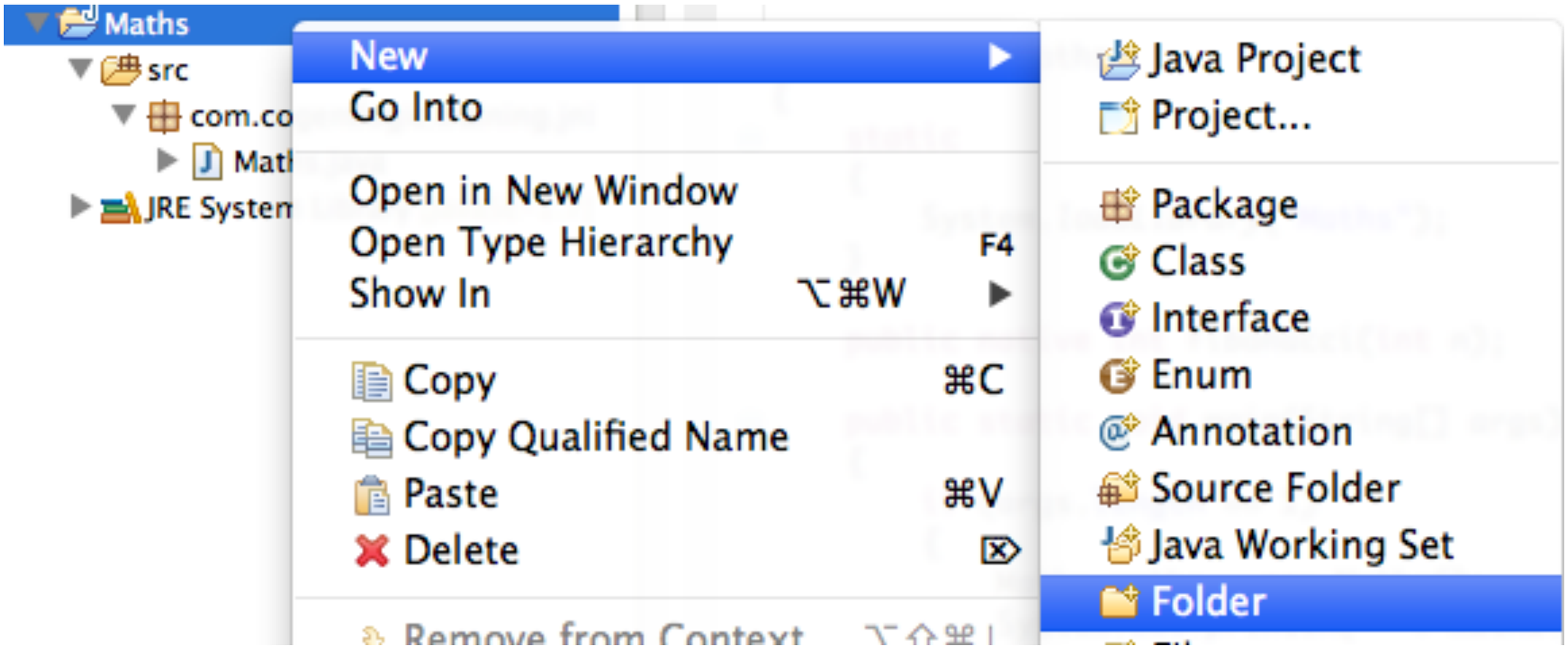
- You can create separate Java and C/C++ projects and import the C/C++ library into the Java project
- You will most likely want to use a combination Java/C/C++ project!
- The general procedure is:
  - Create a Java project with Java source code
  - Add C/C++ source code in its own folder, e.g. `jni`
  - Convert the project to a mixed Java and C/C++ project
  - Configure settings to have Java load the C/C++ library
- The following procedure applies to Mac OS X...

- Here a Java project has been created and a class has been added:

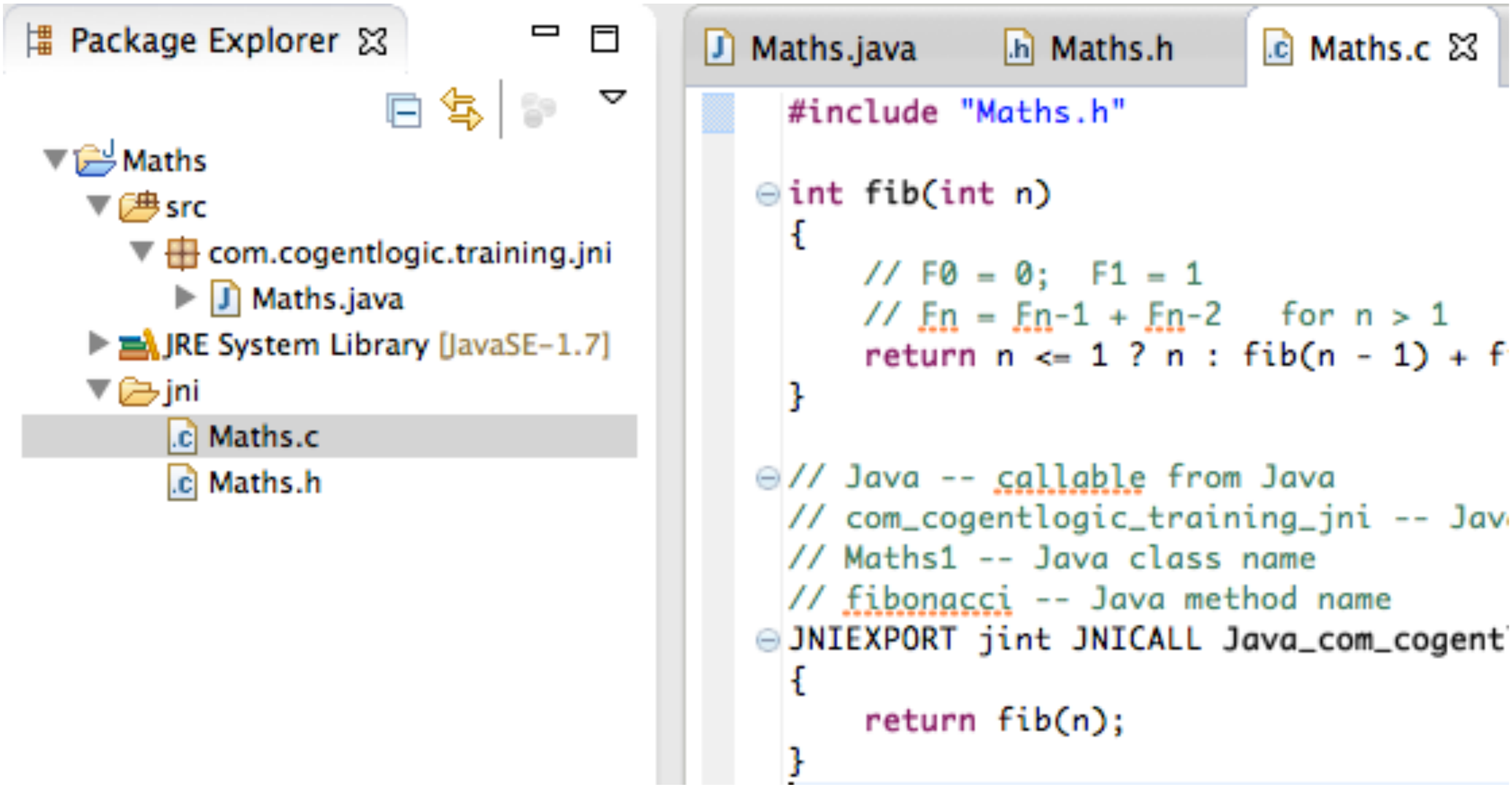


- The `Maths` class is similar to `Maths1` seen earlier
- The library will be called `Maths`, similar to `MathsC` seen earlier

- Create a folder to hold the C/C++ code, here called `jni`:

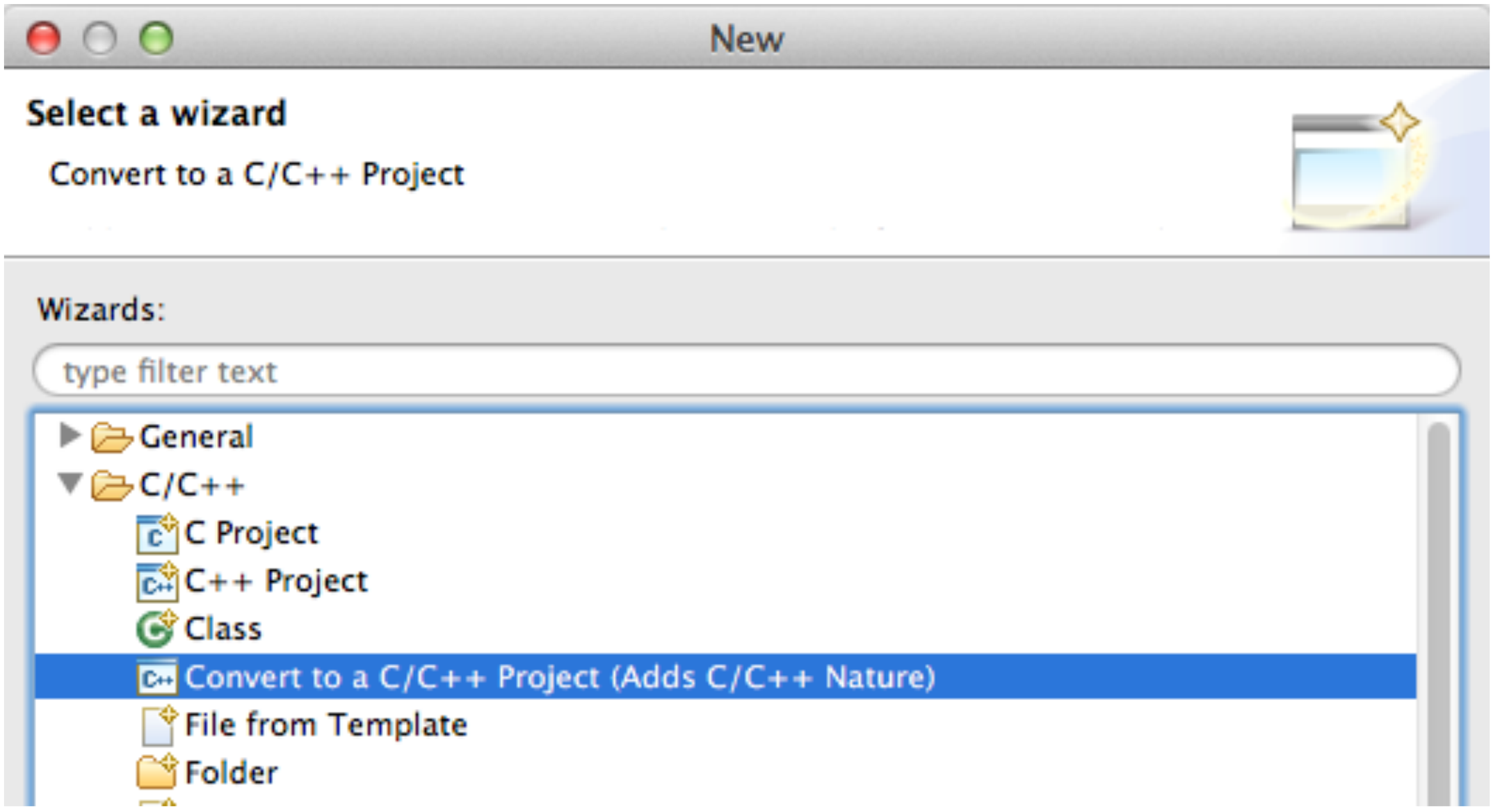


- Add a C header file and a C or C++ implementation file:

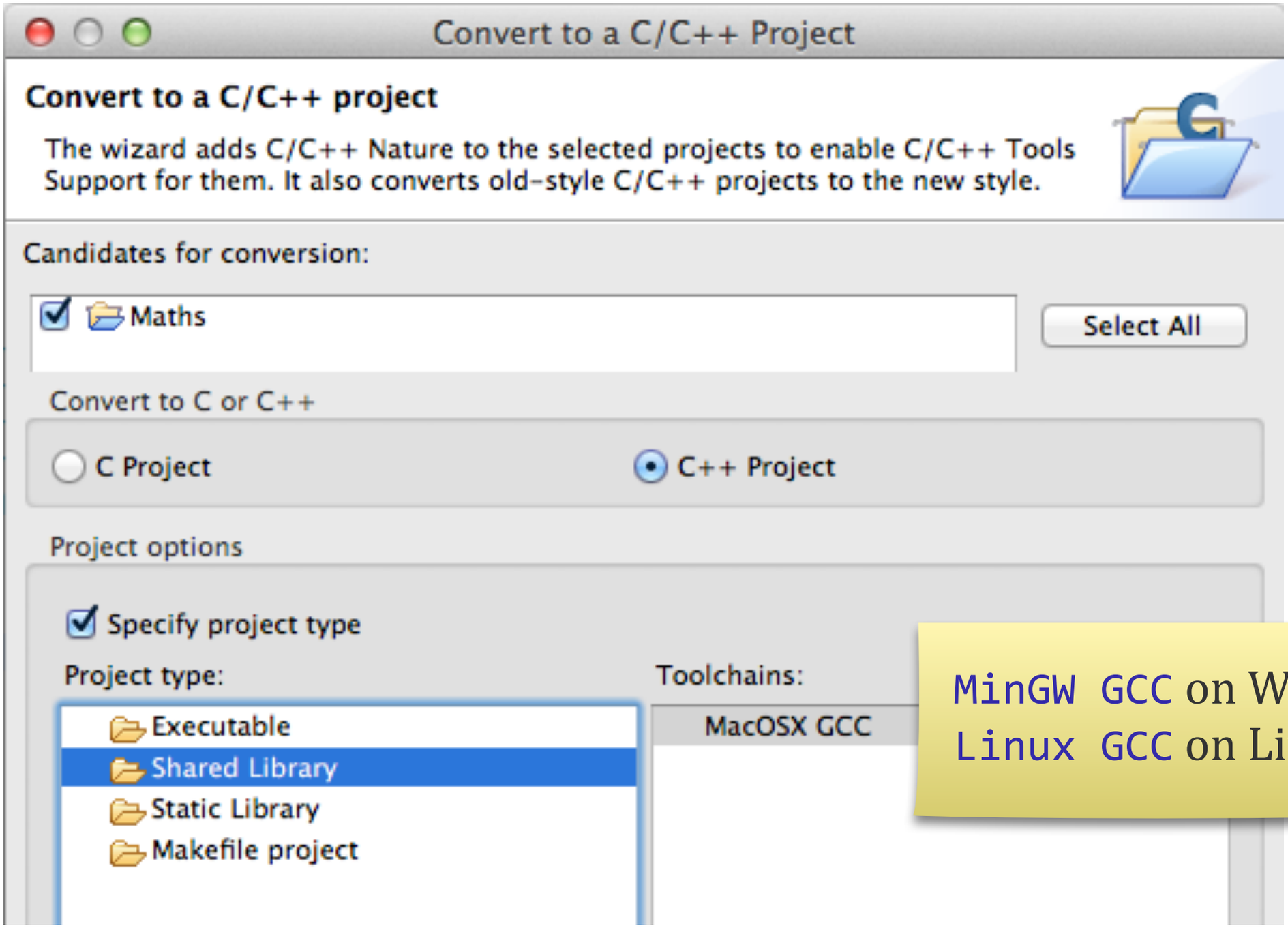


- These files are pretty much the same as `MathsC.h` and `MathsC.c` seen earlier

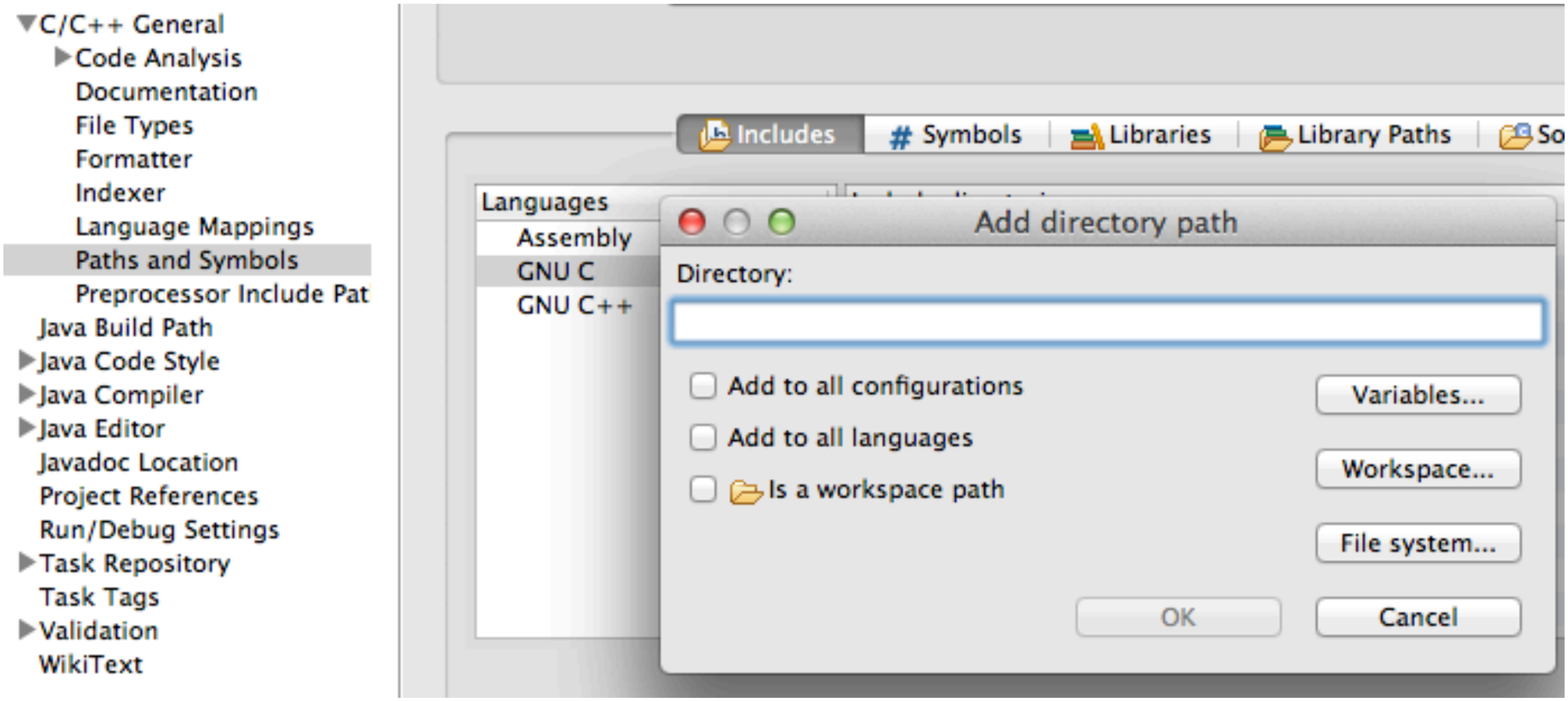
- Convert the project
  - From the `File` menu, select `New` the `Other...`
  - Under `C/C++`, select `Convert to a C/C++ Project`



- Click the `Next >` button
- Select `Shared Library` then click the `Finish` button



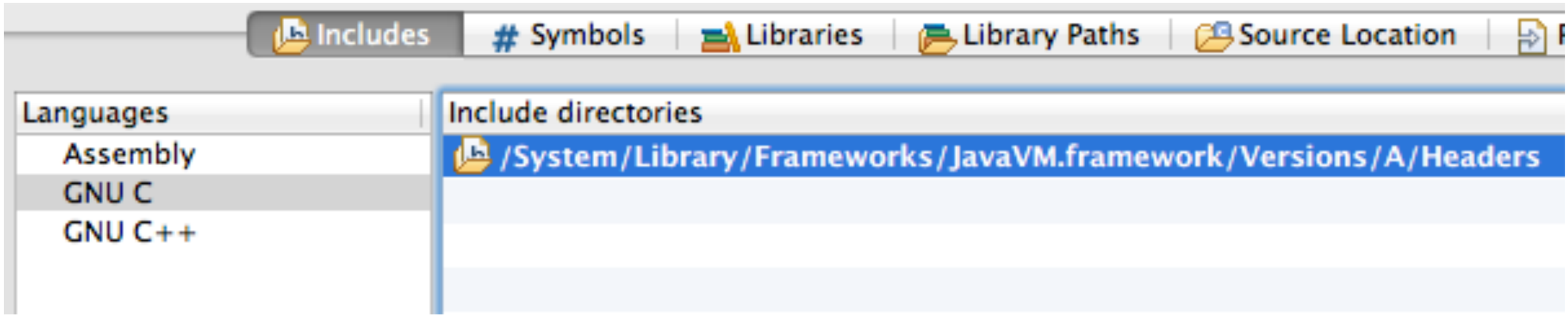
- The project will have C/C++ build errors because we have not specified the JNI header files path
- In the project's `Properties`, `C/C++ General`, select `Paths and Symbols`
- On the `Includes tab`, select GNU C and click the `Add...` button



- Click the `Filesystem...` button and navigate to the JNI header files path, e.g. for Mac OS X, `/System/Library/Frameworks/JavaVM.framework/Headers`

- Click the `OK` button followed by the `Apply` button

- Accept the offer to rebuild the project then click the next `OK` button

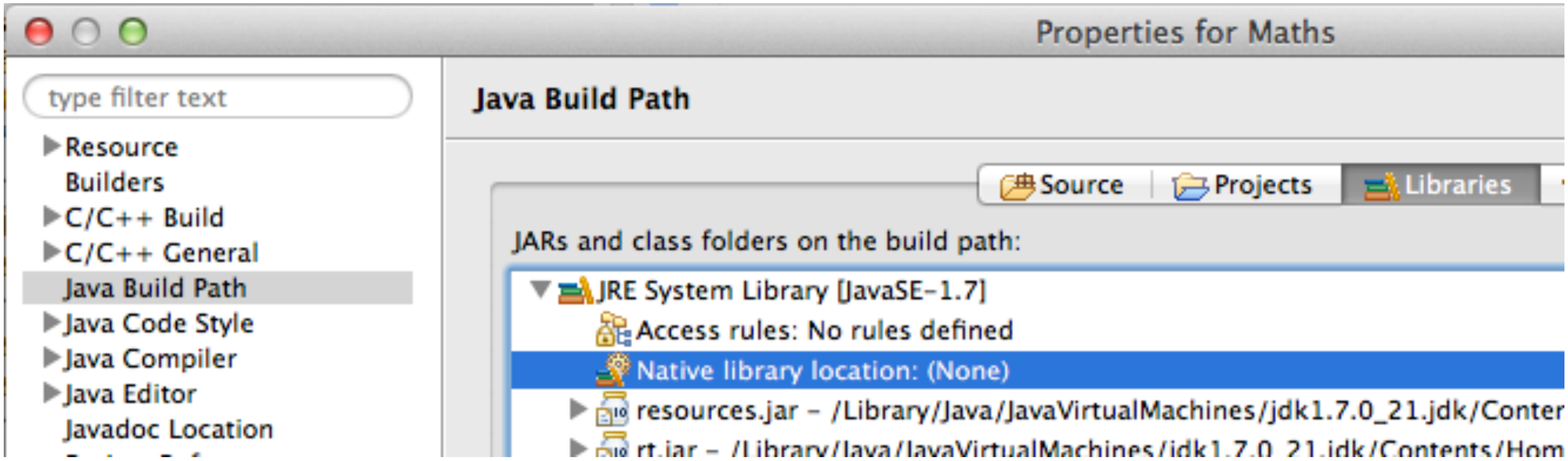


(if the path points to a symbolic link, it will be dereferenced, as here)

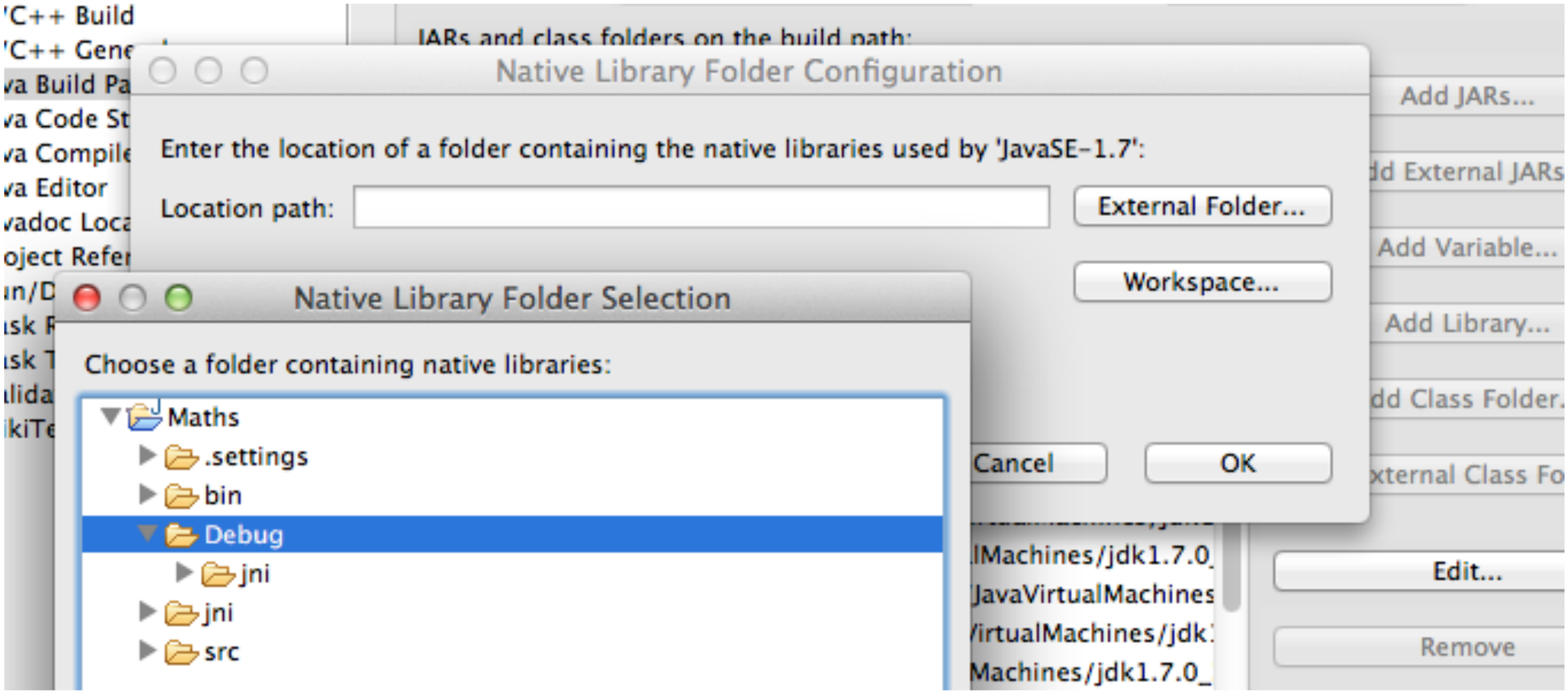
- For Linux, add: `/usr/lib/jvm/java-7-oracle/include` and `/usr/lib/jvm/java-7-oracle/include/linux`

- For 32-bit projects on Windows, add:
  `C:\Program Files (x86)\Java\jdk1.7.0_21\include` and
  `C:\Program Files (x86)\Java\jdk1.7.0_21\include\win32`

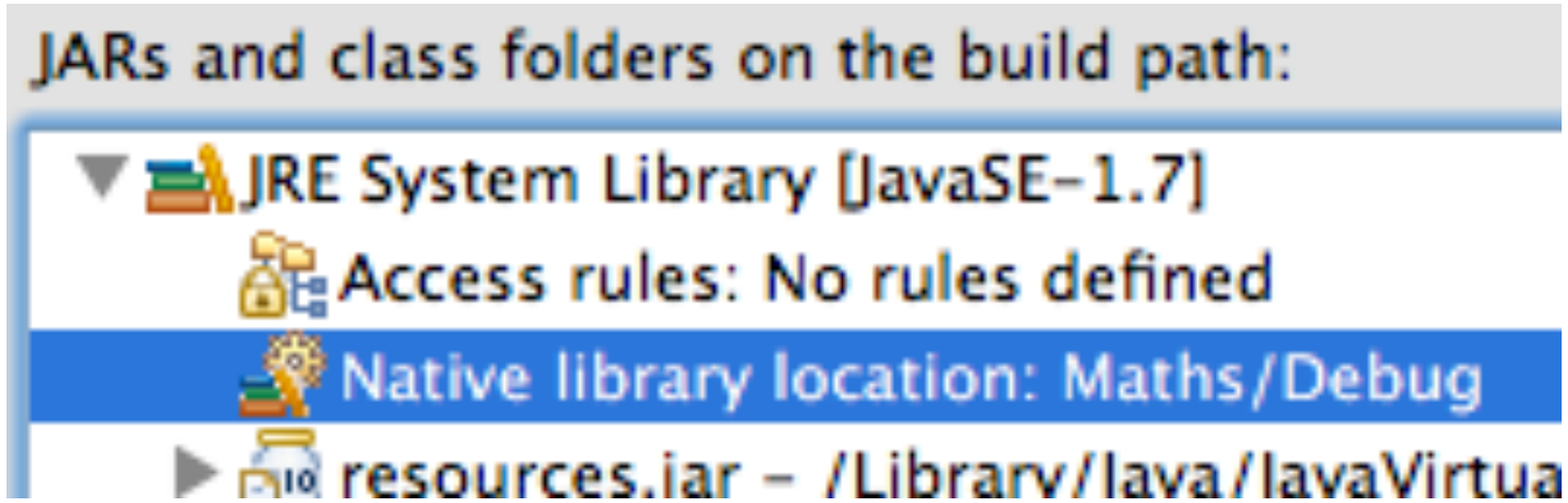
- The C/C++ library will not have been built!
- Simply select `Build Project` from the `Project` menu and the C/C++ library will be generated
- The Java project does not know where to find the library
- In the project's `Properties`, `Java Build Path`, select the `Libraries` tab
- Open up the JRE entry and select `Native library location`



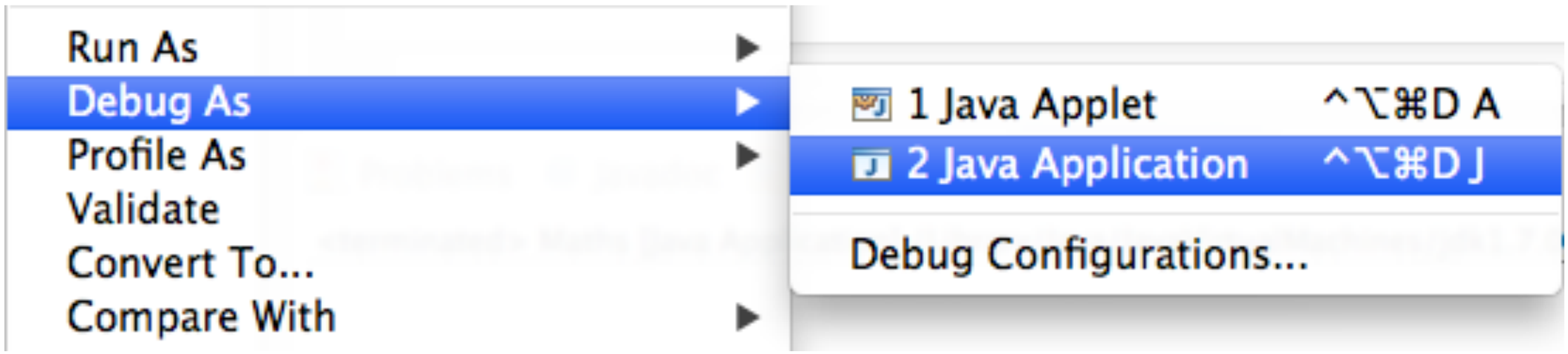
- Click the `Edit...` button followed by the `Workspace...` button
- For a debug build, the C/C++ library is in the `Debug` folder under `bin`



- You have just specified the `java.library.path` value!

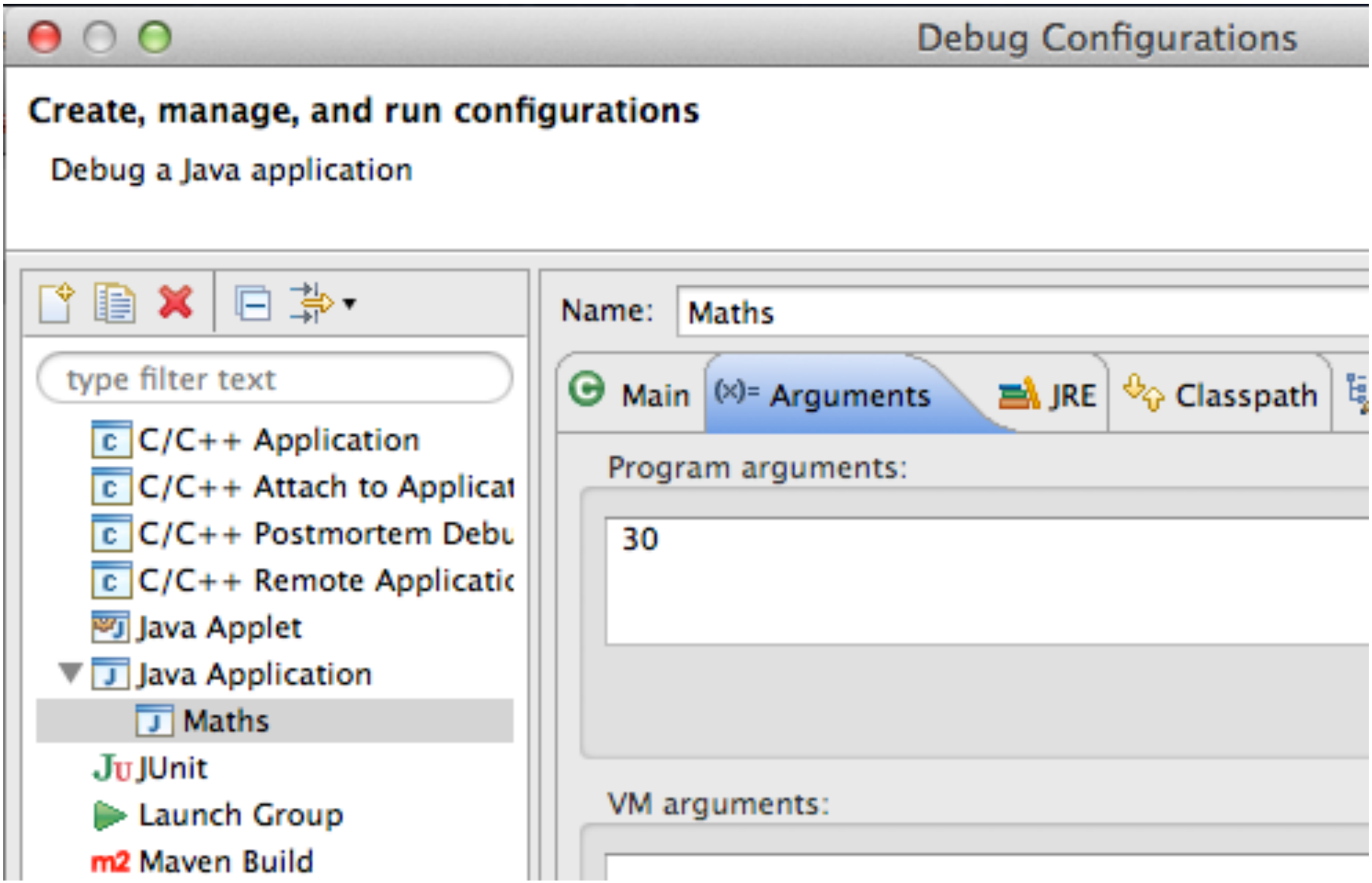


- You can now run or debug your application in the usual way



(do this to create a launch configuration)

- Our software requires a program argument
- Open up the `Debug Configurations` and enter a suitable parameter



- Click the Debug button and trace through the code

Maths.java | Maths.h | Maths.c

```
package com.cogentlogic.training.jni;

public class Maths
{
    static
    {
        System.loadLibrary("Maths");
    }

    public native int fibonacci(int n);

    public static void main(String[] args)
    {
        if (args.length == 1)
        {
            Maths maths = new Maths();
            System.out.println("" + maths.fibonacci(Integer.parseInt(args[0])));
        }
    }
}
```

Console | Tasks | Problems | Executables

```
Maths [Java Application] /Library/Java/JavaVirtualMachines/jdk1.7.0_21.jdk/Contents/Home/bin/java (7 Jun 2013 22:40:32)
832040
```

Linux-Specific Details

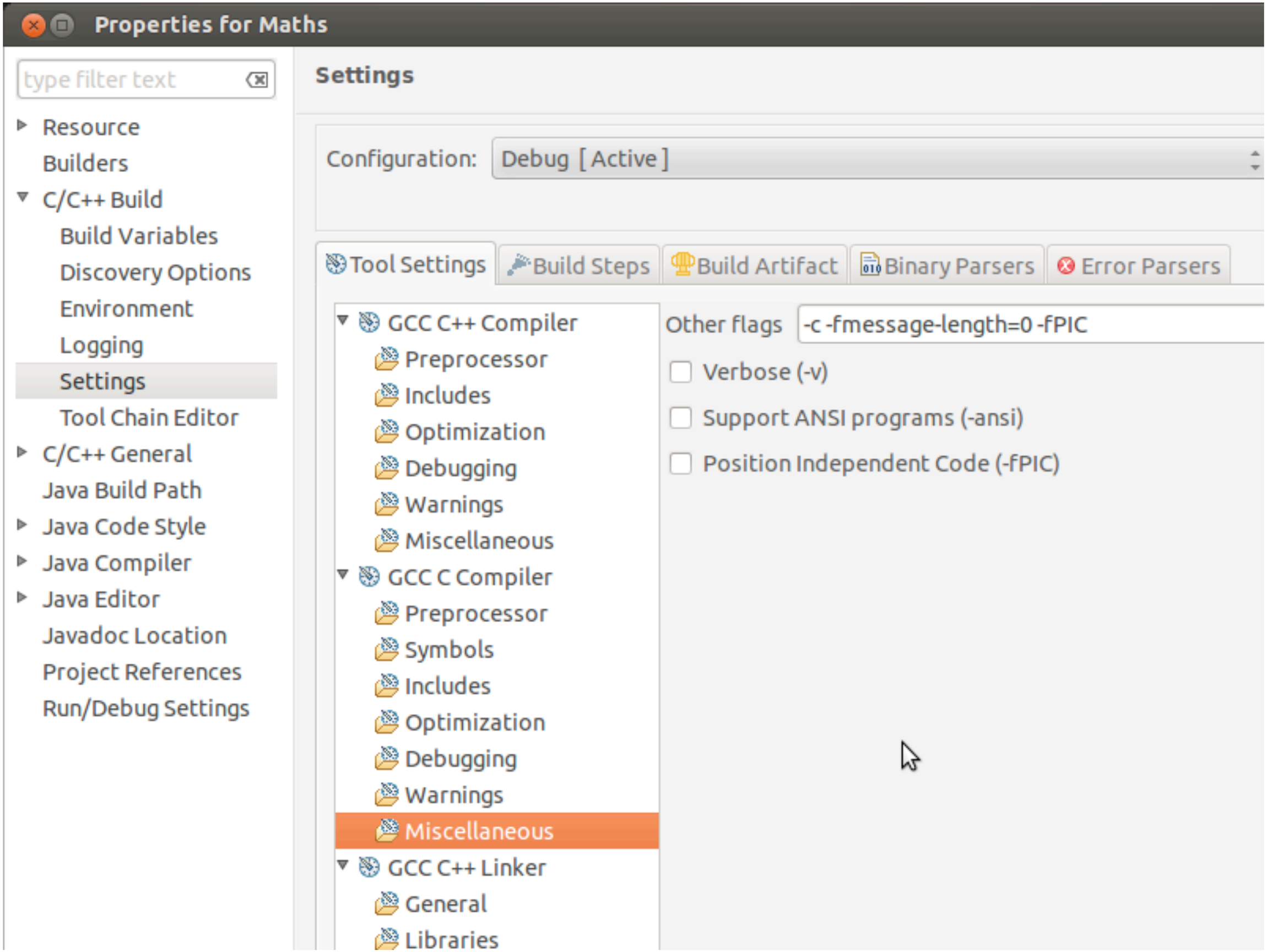
- The preceding procedure works on Linux with a few configuration changes (Unbuntu covered here)

- Firstly, if Eclipse is installed from the Ubuntu Software Centre, you might get an Unsatisfied Link Error upon launching Eclipse.
  To fix this for 64-bit software, at a command line, enter:
  `ln -s /usr/lib/jni/libswt-* ~/.swt/lib/linux/x86_64/`

- As mentioned earlier, specify the JNI header files, as:
  `/usr/lib/jvm/java-7-oracle/include` and
  `/usr/lib/jvm/java-7-oracle/include/linux`

- Remember to specify the `-fPIC` build option for the GCC compiler ...

- You must build the project from the Java perspective (not C/C++)



Windows-Specific Details

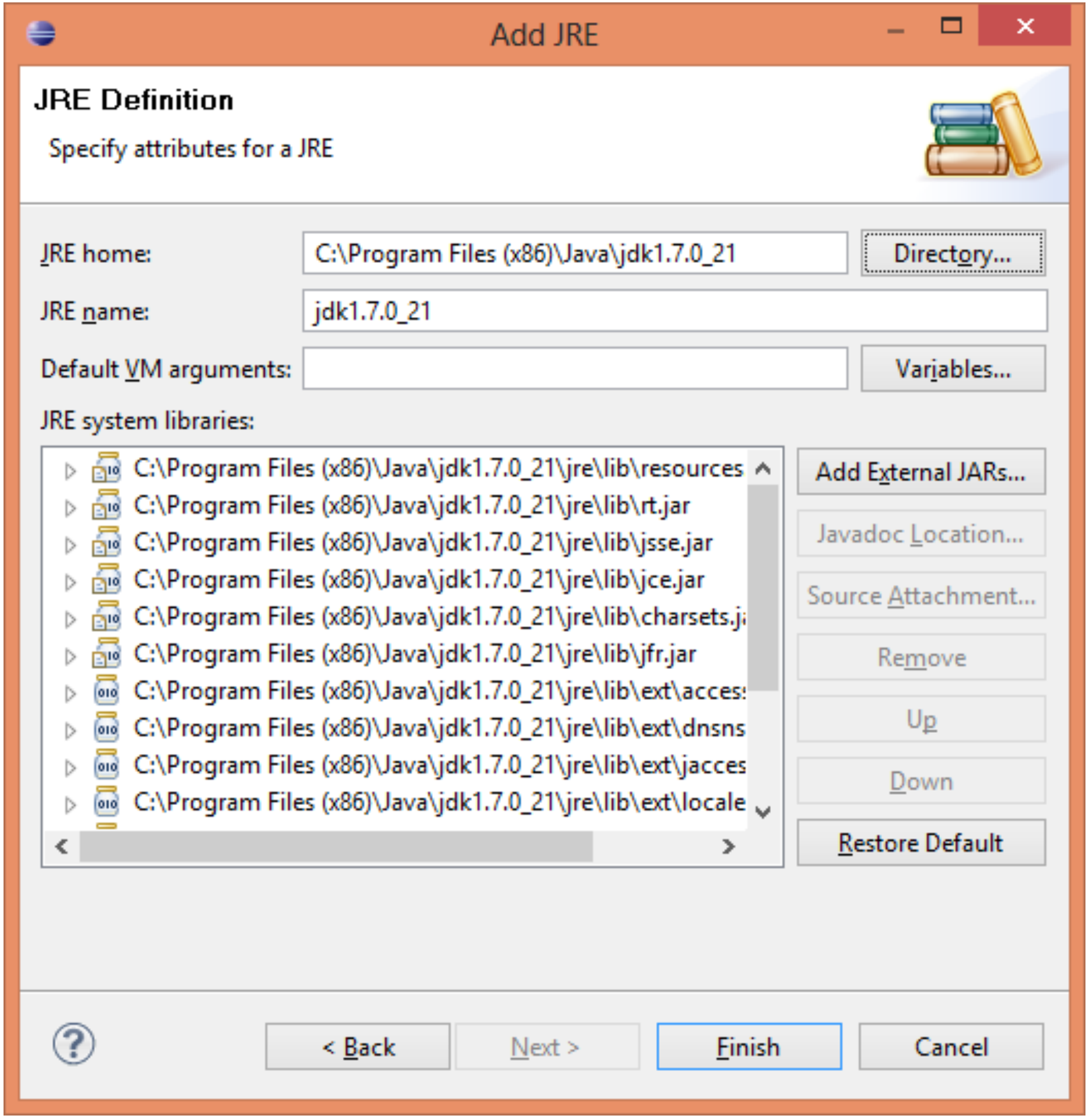
- The preceding procedure works on Windows with a few configuration changes

- Firstly, remember for 64-bit projects on Windows, MinGW-w64 and the `-m64` build flag are required

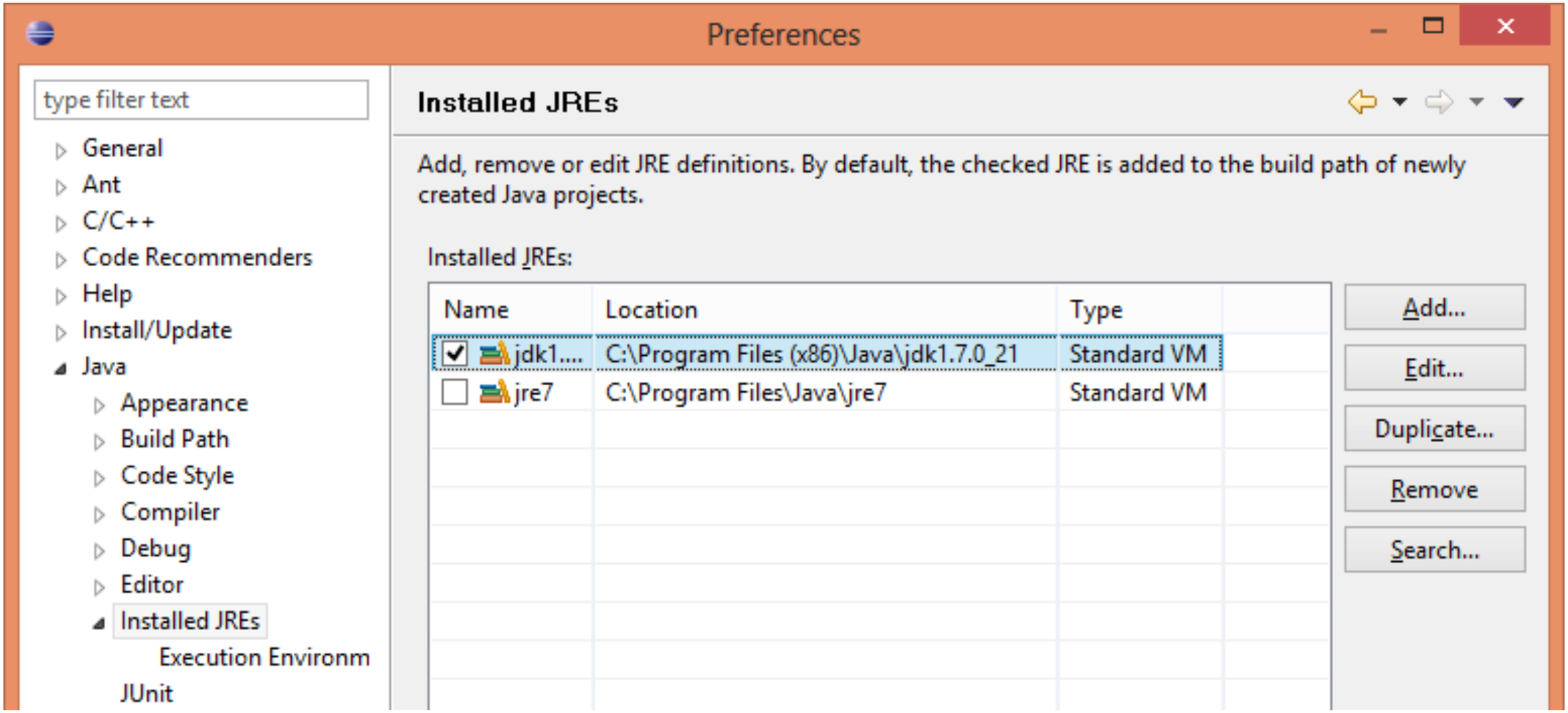
- We will consider 32-bit projects

- The Eclipse workspace must be configured to use a 32-bit JDK (or use a 32-bit version of Eclipse): from the `Windows` menu select `Preferences` then under the `Java` category select `Installed JREs`

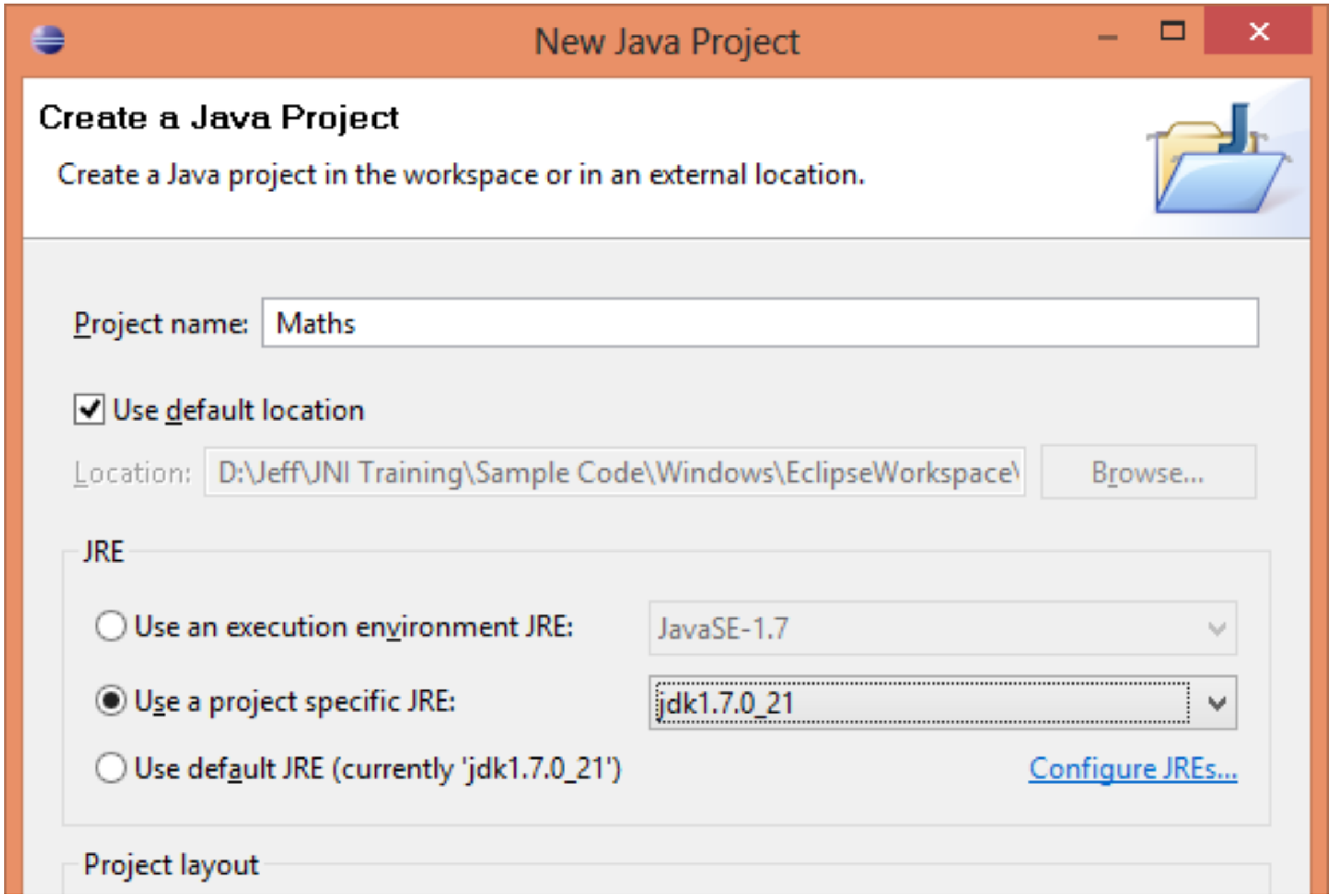
- Click the `Add...` button to add a JRE then click the `Directory...` button to specify a path to a 32-bit JRE,
  e.g. `C:\Program Files (x86)\Java\jdk1.7.0_21`



- Select the 32-bit JDK:



- When creating a Java project, be sure to select the 32-bit JDK:

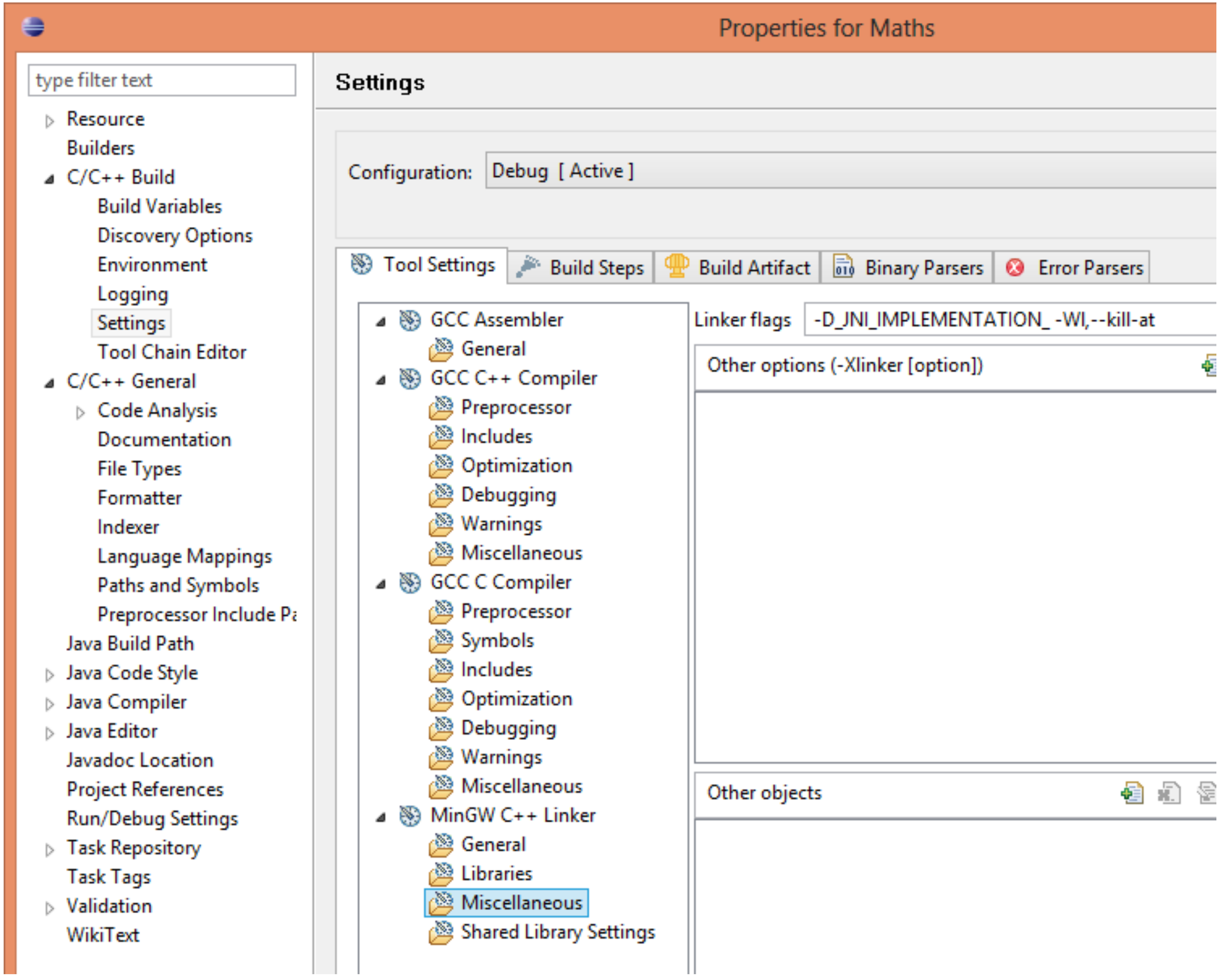


- As mentioned earlier, specify the JNI header files, as:
  `C:\Program Files (x86)\Java\jdk1.7.0_21\include` and
  `C:\Program Files (x86)\Java\jdk1.7.0_21\include\win32`

- The MinGM C++ Linker flags must be set:
  `-D_JNI_IMPLEMENTATION_ -Wl,--kill-at`
  Open the project's `Properties`, `C/C++ Build` category, `Settings` item.
  Select the `Tool Settings` tab followed by the `Miscellaneous` item under `MinGW C++ Linker`.
  Add the flags to the `Linker flags` field.

- Finally, for the Maths sample code, the generated C/C++ library will be called `libMaths.dll`; in the Windows Java source code this must be referenced with the preceding `lib`, i.e.

```
System.loadLibrary("libMaths");
```



Java Native Interface with Eclipse and Android

Mapping Strings and Other Data Types

Jeff Lawson

Contents

JNI Primitive Data Types

- Primitive data types can be passed to and returned from JNI methods without difficulty

- `jni.h` defines machine-independent 'JNI types' that map to C types:

```
typedef unsigned char  jboolean;
typedef unsigned short  jchar;
typedef short  jshort;
typedef float  jfloat;
typedef double  jdouble;
```

- These types correspond, respectively, to the Java data types:
  `boolean`, `char`, `short`, `float`, `double`

- `jni_md.h`, included in `jni.h`, defines machine-dependent mappings:

```
#if defined(__LP64__) && __LP64__ /* for -Wundef */
typedef int  jint;
#else
typedef long  jint;
#endif
typedef long long  jlong;
typedef signed char  jbyte;
```

`__LP64__` is set by the compiler to flag 64-bit builds

- These three data types map to Java data types `int`, `long` and `byte`

- `jni.h` makes use of `jint` for the definition of `jsize`:

```
  typedef jint  jsize;
```

The `javah` Tool

- `javah` can be invoked from the command line to generate C/C++ prototype methods
- `javah` runs against a Java source code file, seeking native methods and generating the corresponding C/C++ method declarations
- For `Maths.java` containing `public native int fibonacci(int n);` use:

```
javah com.cogentlogic.training.jni.Maths
```

  which will generate `com_cogentlogic_training_jni_Maths.h` containing:

```
JNIEXPORT jint JNICALL
   Java_com_cogentlogic_training_jni_Maths_fibonacci
                                (JNIEnv *, jobject, jint);
```

The `JNIEnv` Interface Pointer

- The `JNIEnv` interface pointer is always passed as the first argument to a JNI native method and provides access to general-purpose methods, e.g. `GetJavaVM`

- `JNIEnv` provides support for string and management, array operations, Java instance/static method/field invocation/access, exception handling, reflection

- See `JNIEnv` documentation at:
  `docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/functions.html`

- `JNIEnv` is used for thread-local storage so it is not valid across threads, i.e. it cannot be passed to another thread

- `JNIEnv` is used differently in C and in C++: `env` is a pointer to a C++ object that must be dereferenced in C

- In C++, we can invoke methods straightforwardly:

```
JavaVM* jvm;
env->GetJavaVM(&jvm);
```

- In C, `env` must be passed as the first parameter in each `env` call:

```
JavaVM* jvm;
(*env)->GetJavaVM(env, &jvm);
```

Do not name C and C++ files the same, e.g. `X.c` and `X.cpp`. If you do then one of the files won't be

- (`JavaVM` is a reference to the Java Virtual Machine and can be used across threads. `JavaVM` enables native threads to attach to the JVM.)

JNI Strings

- Java `String` objects appear in JNI as `jstring` objects

- To make use of a `jstring` object (JVM string!) in C we must convert it to a C string, typically with `GetStringUTFChars` (for UTF-8), e.g.

  ```
  const char* pchName = env->GetStringUTFChars(strName, 0);
  ```

  which returns `0` for an out-of-memory condition (the second argument is an optional pointer to a `boolean` that tells us whether the returned string is a copy of the original string)

- When we have finished with such C strings we must release them, e.g.

  ```
  env->ReleaseStringUTFChars(strName, pchName);
  ```

- Strings can be allocated for return to the Java code with `NewString` (for Unicode) or `NewStringUTF` (see sample project *Strings*)

References to Java Objects

- JNI native methods always pass a reference object as a second parameter
- For native instance methods, the object is a `jobject` that references the Java object that invoked the method
- For native static methods, the object is a `jclass` references the Java class that invoked the method
- These objects can be used to call instance/static methods in the calling Java object/class and to access the instance/static fields
- More generally, Java objects can be passed as parameters to JNI methods and their members can be accessed too

- Native types map to Java object types as follows:

jobject	all objects
jclass	java.lang.Class
jstring	java.lang.String
jarray	all arrays
jobjectArray	Object[]
jbooleanArray	boolean[]
jbyteArray	byte[]
jcharArray	char[]
jshortArray	short[]
jintArray	int[]
jlongArray	long[]
jfloatArray	float[]
jdoubleArray	double[]
jthrowable	java.lang.Throwable

Accessing Java Arrays

- Native methods can take arrays as parameters and return arrays
- We need to transform JVM array data to C array data before use
- We can simulate pass-by-value by using a copy of the Java array data
- We can simulate pass-by-reference by using a pointer to the Java array data where possible
- In either case, we can choose to update the Java array data or leave it unchanged!

- Sample Java code to manipulate arrays: (*Arrays*)

```
package com.cogentlogic.training.jni;

public class Arrays
{
  // Forgetting to do this is one of several way to
  // produce the UnsatisfiedLinkError exception!
  static
  {
        System.loadLibrary("Arrays");
  }

  private native float[] traverse(double[] dblA, float[] fB);

  public static void main(String[] args)
  {
     double[] dblA = {1.1, 2.2, 3.3};
     float[] fB = {4.4f, 5.5f, 6.6f, 7.7f};

     Arrays arrays = new Arrays();
     float[] fC = arrays.traverse(dblA, fB);
```

- Sample Java code to manipulate arrays (continued):

```
        System.out.println("dblA:");
        for (int n=0; n<dblA.length; n++)
           System.out.println("     " + dblA[n]);

        System.out.println("fB:");
        for (int n=0; n<fB.length; n++)
           System.out.println("    " + fB[n]);

        System.out.println("fC:");
        for (int n=0; n<fC.length; n++)
           System.out.println("    " + fC[n]);
    }
}
```

- `javah com.cogentlogic.training.jni.Arrays` generates:

```
JNIEXPORT jfloatArray JNICALL
   Java_com_cogentlogic_training_jni_Arrays_traverse
                (JNIEnv *, jobject, jdoubleArray, jfloatArray);
```

- Sample C code to manipulate arrays: (*Arrays*)

```
JNIEXPORT jfloatArray JNICALL
Java_com_cogentlogic_training_jni_Arrays_traverse(JNIEnv* env,
jobject thiz, jdoubleArray dblA, jfloatArray fB)
{
  int n;
  // Get the lengths of the array parameters
  jint nLenA = (*env)->GetArrayLength(env, dblA);
  jint nLenB = (*env)->GetArrayLength(env, fB);

  // Access a copy of the double array
  jdouble dblAA[nLenA];
  (*env)->GetDoubleArrayRegion(env, dblA, 0, nLenA, dblAA);

  // Update the copy of the double array
  for (n=0; n<nLenA; n++)
     dblAA[n] += 100.0;

  // Commit the contents of the copy to the Java array reference
  (*env)->SetDoubleArrayRegion(env, dblA, 0, nLenA, dblAA);
```

- Sample C code to manipulate arrays (continued):

```
// Access the double array directly
jboolean bCopyA;
jdouble* pdblA = (*env)->GetDoubleArrayElements(env, dblA,
                                                &bCopyA);
if (pdblA)
{
   jdouble* pdbl = pdblA;       // retain for deallocation
   for (n=0; n<nLenA; n++)
      *pdbl++ += 10000.0;

   // 0 => copy back and release native array:
   (*env)->ReleaseDoubleArrayElements(env, dblA, pdblA, 0);
}
```

The `&bCopyA` parameter is optional (can be NULL):

`bCopyA == JNI_TRUE` => `pdblA` will point to a copy of the array

`bCopyA == JNI_FALSE` => `pdblA` will point to the original array

- Sample C code to manipulate arrays (continued):

```
jfloatArray fC = (*env)->NewFloatArray(env, nLenB);
if (fC)
{
  jboolean bCopyB;
  jfloat* pfB = (*env)->GetFloatArrayElements(env, fB, &bCopyB);
  if (pfB)
  {
     jboolean bCopyC;
     jfloat* pfC = (*env)->GetFloatArrayElements(env, fC, &bCopyC);
     if (pfC)
     {
        jfloat* pfb = pfB, pfc = pfC + nLenB;
        for (n=0; n<nLenB; n++)
           *--pfc = *pfb++;
        (*env)->ReleaseFloatArrayElements(env, fC, pfC, 0);
     }
     (*env)->ReleaseFloatArrayElements(env, fB, pfB, JNI_ABORT);
  }
}                          // JNI_ABORT => release but do not copy back
return fC;                 // JNI_COMMIT => copy back but do not release
                           // 0 => release and copy back
```

- Remember, the original arrays were:

```
double[] dblA = {1.1, 2.2, 3.3};
float[] fB = {4.4f, 5.5f, 6.6f, 7.7f};
```

- The corresponding output is:

```
dblA:
     10101.1
     10102.2
     10103.3
fB:                    fC:
    4.4                    7.7
    5.5                    6.6
    6.6                    5.5
    7.7                    4.4
```

- If you need to discard an array created with `New<TYPE>Array`, use `DeleteLocalRef`

Java Native Interface with Eclipse and Android

Accessing Java Methods and Fields from C/C++

Jeff Lawson

Contents

- Method and Field Descriptors
- Accessing Java Class Members

Method and Field Descriptors

- To access Java methods and fields from C/C++ we need to identify each member unambiguously by specifying their signatures and data types

- `javah com.cogentlogic.training.jni.Arrays` seen earlier, generated more that the native method prototype:

```
JNIEXPORT jfloatArray JNICALL
    Java_com_cogentlogic_training_jni_Arrays_traverse
                    (JNIEnv *, jobject, jdoubleArray, jfloatArray);
```

  it also produced:

```
Signature: ([D[F)[F
```

- `[D`, `[F`, `[F` are *descriptors,* here indicating arrays of `double` and `float`

- Descriptors enable us to specify Java methods and fields

- The Java class file disassembler, `javap`, produces class member signatures
- Consider this Java class: (*MethodsAndFields*)

```
public class MethodsAndFields
{
  private static int s_n = 123;
  private int m_n = 456;

  private static void setS(int n) {s_n = n; }
  private void setI(int n) {m_n = n;}

  private native static int mafS(int n);
  private native int mafI(int n);

  public static void main(String[] args)
  {
     System.out.println("mafS(2) returns " +
                        MethodsAndFields.mafS(2) +
                        " with s_n = " + MethodsAndFields.s_n);
     MethodsAndFields maf = new MethodsAndFields();
     System.out.println("mafI(3) returns " + maf.mafI(3) +
                        " with m_n = " + maf.m_n);
  }
}
```

- The -p option indicates include private members

- The -s option indicates show signature (that's what we're interested in!)

  ```
  javap -p -s com.cogentlogic.training.jni.MethodsAndFields
  ```

  produces:

```
private static int s_n;                                 Signature: I
private double m_dbl;                                   Signature: D
public com.cogentlogic.training.jni.MethodsAndFields();
  (constructor)                                         Signature: ()V
private static void setS(int);                          Signature: (I)V
private void setI(double);                              Signature: (D)V
private static native int mafS(int);                   Signature: (I)I
private native double mafI(double);                     Signature: (D)D
public static void main(java.lang.String[]);
                                        Signature: ([Ljava/lang/String;)V
```

Accessing Java Class Members

- Sample C code to access a static method and a static field:

(*MethodsAndFields*)

```
JNIEXPORT jint JNICALL
   Java_com_cogentlogic_training_jni_MethodsAndFields_mafS(
                                  JNIEnv* env, jclass clazz, jint n)
{
  jfieldID idFieldS = (*env)->GetStaticFieldID(env, clazz,
                                                  "s_n", "I");
  jint s_n = (*env)->GetStaticIntField(env, clazz, idFieldS);

  jmethodID idMethodS = (*env)->GetStaticMethodID(env, clazz,
                                                   "setS",
"(I)V");
  (*env)->CallStaticVoidMethod(env, clazz, idMethodS, s_n + n);

  return (*env)->GetStaticIntField(env, clazz, idFieldS) + 1;
}
```

- Sample C+ code to access an instance method and field:

(*MethodsAndFields*)

```
#ifdef __cplusplus
extern "C" {
#endif

JNIEXPORT jdouble JNICALL
   Java_com_cogentlogic_training_jni_MethodsAndFields_mafI(
                              JNIEnv* env, jobject thiz, jdouble dbl)
{
  jclass clazz = env->GetObjectClass(thiz);
  jfieldID idFieldI = env->GetFieldID(clazz, "m_dbl", "D");
  jdouble m_dbl = env->GetDoubleField(thiz, idFieldI);

  jmethodID idMethodI = env->GetMethodID(clazz, "setI", "(D)V");
  env->CallVoidMethod(thiz, idMethodI, m_dbl + dbl);

  return env->GetDoubleField(thiz, idFieldI) + 10000.0;
}

#ifdef __cplusplus
}
#endif
```

Java Native Interface with Eclipse and Android

Exception Handling

Jeff Lawson

Contents

- Throwing Exceptions from Native Methods
- Catching Exceptions in Native Methods

Throwing Exceptions from Native Methods

- Native C++ methods might have C++ exceptions to deal with. We assume that this is handled by the C++ developer. Here we are concerned only with Java exceptions!

- In project *Arrays*, we saw code the checked for out-of-memory conditions but did not handle them well e.g.

```
jdouble* pdblA = (*env)->GetDoubleArrayElements(env, dblA, 0);
if (pdblA)
{
  // Memory allocated so proceed...
```

- A better way to handle this condition would to throw an appropriate Java exception, i.e. `java.lang.OutOfMemoryError`
(this not an exception that we normally catch in Java but it handles the `GetDoubleArrayElements` condition well)

- If we need to throw an exception from a native method we do it by calling `ThrowNew`, passing a `jclass` object for the desired Java exception, e.g.

```
jclass clazz = (*env)->FindClass(env,
                                 "java/lang/OutOfMemoryError");
if (clazz)
  (*env)->ThrowNew(env, clazz, chMsg);
```

- It is important to realize that calling `ThrowNew` does *not* generate an exception in the native method: the native method continues to run to completion and should degrade gracefully, freeing resources, etc.

- The Java code should catch the exception in the usual way, of course, e.g. (*ExceptionThrowFromJNI*)

```
try
{
  float[] fC = etfj.mightThrow(dblA, fB);
}
catch (OutOfMemoryError except)
{
  except.printStackTrace();
}
```

Catching Exceptions in Native Methods

- A native method could invoke a Java method that throws an exception
- The native method can catch the exception by making an `ExceptionOccurred` call to check whether an exception was thrown!
- Sample Java code that throws an exception: (*ExceptionCatchInJNI*)

```
void dodgyCode() throws ClassCastException
{
  Object objInteger = new Integer(666);
  System.out.println((String)objInteger);
}
```

- Sample C code that ‘catches’ a Java exception: (*ExceptionCatchInJNI*)

```
jclass clazz = (*env)->GetObjectClass(env, thiz);
jmethodID idDodgy = (*env)->GetMethodID(env, clazz,
                                        "dodgyCode", "()V");
(*env)->CallVoidMethod(env, thiz, idDodgy);
jthrowable except = (*env)->ExceptionOccurred(env);
if (except)
{
  (*env)->ExceptionDescribe(env);
  (*env)->ExceptionClear(env);

  // Handle exception, e.g. scrutinize the jthrowable object
  // to determine the type of exception
  // ...

  (*env)->Throw(env, except);
  (*env)->DeleteLocalRef(env, except);
}
```

Java Native Interface with Eclipse and Android

SWIG

Jeff Lawson

Contents

- What is SWIG?
- Installing SWIG
- SWIG Use

What is SWIG?

- Simplified Wrapper and Interface Generator (SWIG) is a tool that facilitates access to C/C++ libraries from other languages such as Java
- SWIG takes an interface file (.i file extension) as input
- Interface files are just C/C++ header files will SWIG preprocessor directives, specically:
    - SWIG module declaration, e.g. %module Maths
    - Section to be copied into a C wrapper file (not parsed by SWIG)
    - Section parsed by SWIG to generate the remaining output
- For more on SWIG, see `www.swig.org`

Installing SWIG

- On Mac OS X, typically install SWIG using Homebrew:
  - Install Homebrew by running the script shown at:

    `mxcl.github.io/homebrew/`

    e.g. `ruby -e "$(curl -fsSL https://raw.github.com/mxcl/homebrew/go)"`
    - Install SWIG by entering: `brew install swig`
- On Linux (Ubuntu), enter: `sudo apt-get install swig`
- On Windows, download SWIG in a ZIP file from `www.swig.org/download.html`, extract it and add its location to the PATH variable

SWIG Use

- First, create an interface file, e.g.

```
%module Maths

// This added to the C wrapper file (not parsed by SWIG)
%{
// Put includes and other declarations here
extern int fibonacci(int);
%}

// This parsed by SWIG to generate the three output files
extern int fibonacci(int);
```

- Next run the SWIG tool against the interface file, e.g.

```
swig -java -package com.cogentlogic.training
     -outdir src/com/cogentlogic/training
     jni/swiginterface.i
```

- The three files generated from the previous command are:
  - `src/com/cogentlogic/training/Maths.java`
  - `src/com/cogentlogic/training/MathsJNI.java`
  - `jni/swiginterface_wrap.c`
- The module was specified as `Maths`, remember

- `MathsJNI.java` simply declares the native method in a Java class

```
package com.cogentlogic.training;

public class MathsJNI
{
  public final static native int fibonacci(int jarg1);
}
```

- `Maths.java` contains sample code for making use of `MathsJNI.java`:

```
package com.cogentlogic.training;

public class Maths
{
  public static int fibonacci(int arg0)
  {
     return MathsJNI.fibonacci(arg0);
  }
}
```

- `swiginterface_wrap.c` contains a good deal of SWIG code that we can ignore; it also contains the declarations made in the interface file and a skeleton JNI method with a suitable name, e.g.

```
extern int fibonacci(int);

SWIGEXPORT jint JNICALL
    Java_com_cogentlogic_training_MathsJNI_fibonacci(
                    JNIEnv *jenv, jclass jcls, jint jarg1)
```

- The skeleton native method contains: (*SWIG*)

```
jint jresult = 0 ;
int arg1 ;
int result;

(void)jenv;
(void)jcls;
arg1 = (int)jarg1;
result = (int)fibonacci(arg1);
jresult = (jint)result;
return jresult;
```

- We see typical extraneous code that is symptomatic of a tool that is capable of handling complex scenarios!

- All we need to do is implement `fibonacci` in C, typically in another source code file e.g. in `Fib.c` (with a header file `Fib.h`)

```
int fibonacci(int n)
{ return n <= 1 ? n : fibonacci(n - 1) + fibonacci(n - 2); }
```

- To use the SWIG output:
  - Create a Java project with C/C++ conversion, specifying header path(s) and referencing the Debug folder in the Java Build Path
  - Add a jni folder and add a SWIG interface file to the jni folder
  - Add suitable declarations to the interface file
  - Run the SWIG tool (this could be automated within the project)
  - Provide implementations for the JNI methods declared in the SWIG wrapper file
  - Make use of the native methods in your Java code
  - Build the project and check the library name so it can be loaded ...

Package Explorer

- Arrays
- ExceptionCatchInJNI
- ExceptionThrowFromJNI
- Maths
- MethodsAndFields
- Strings
- SWIG
  - src
    - com.cogentlogic.training
      - Maths.java
      - MathsJNI.java
      - Swig.java
  - JRE System Library [JavaSE-1.7]
  - Debug
    - jni
      - libSWIG.dylib
      - makefile
      - objects.mk
      - sources.mk
  - jni
    - Fib.c
    - Fib.h
    - swiginterface_wrap.c
    - swiginterface.i

Swig.java | Maths.java | swiginterface.i | Ma

```
package com.cogentlogic.training;

public class Swig
{
    static
    {
        System.loadLibrary("SWIG");
    }

    public static void main(String[] args)
    {
        System.out.println(Maths.fibonacci(30));
    }
}
```

Java Native Interface with Eclipse and Android

Using Standard C/C++ Libraries and Open Source Libraries

Jeff Lawson

Contents

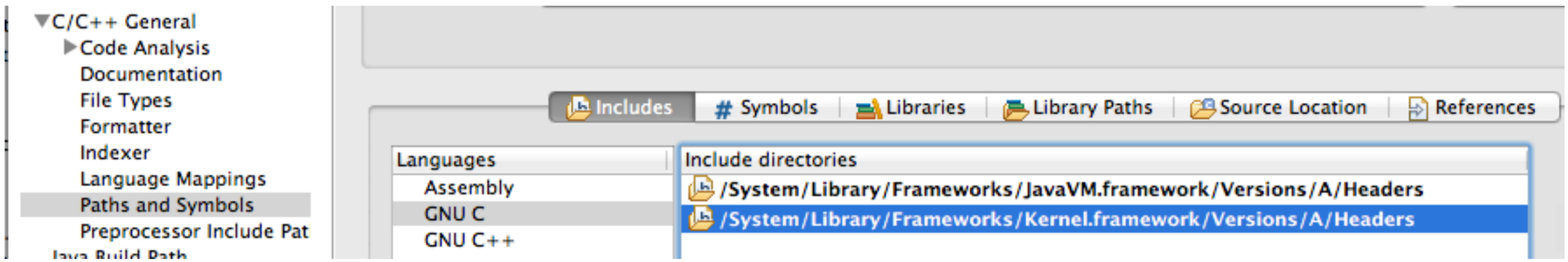
- Standard C/C++ Libraries
- Open Source Libraries
- OpenSSL

Standard C/C++ Libraries

- C/C++ compilers usually support standard libraries for i/o, strings, maths, etc., e.g. C++ Standard Template Library

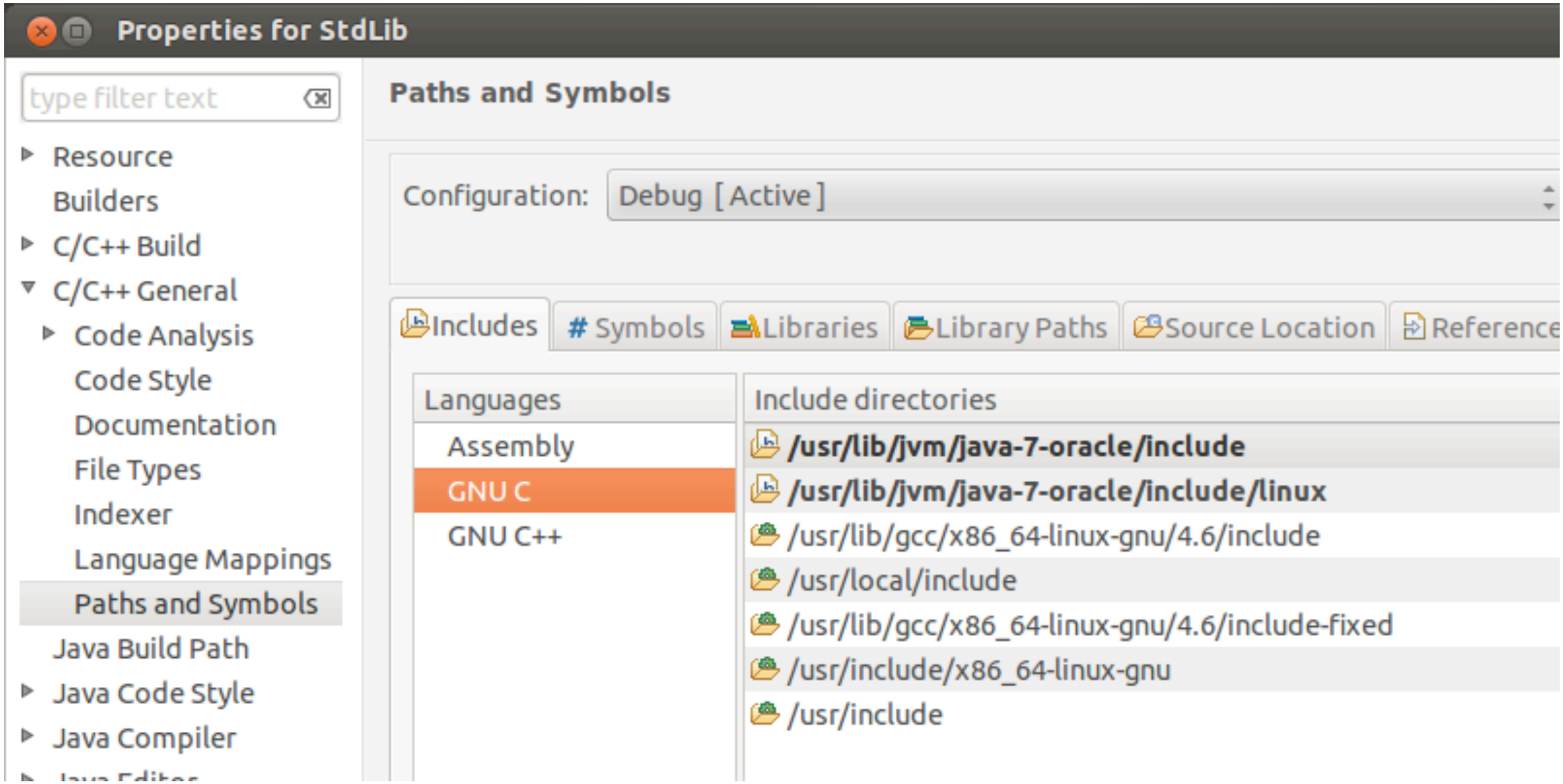
- Resolving standard libraries is system-dependent, e.g. to support `string.h` on Mac OS X, add to Paths and Symbols:

  `/System/Library/Frameworks/Kernel.framework/Headers`

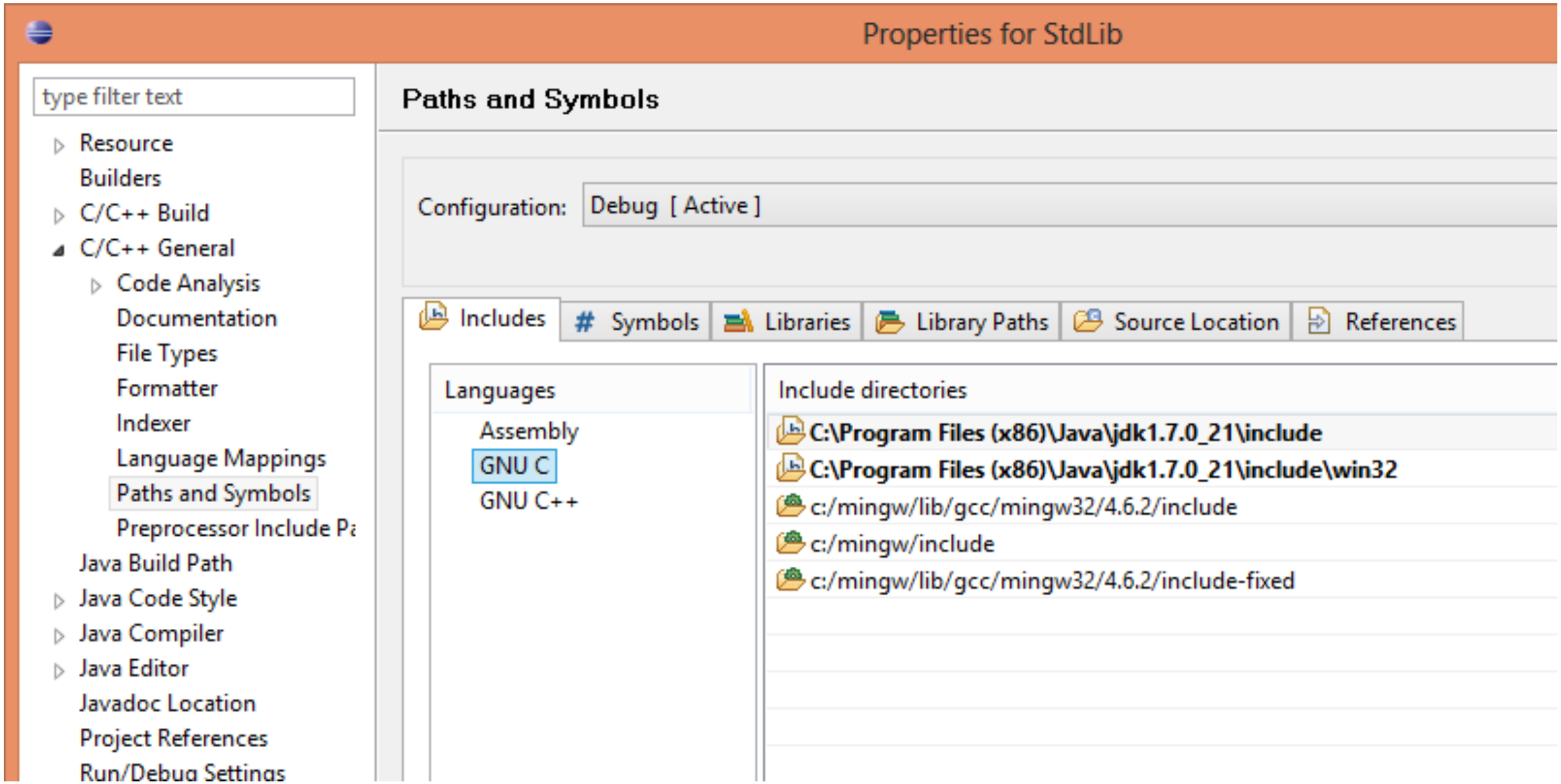


- See the sample code (*StdLib*)

- The Linux installation automatically picks up the header paths ...



- The Windows installation automatically picks up the header paths ...



Open Source Libraries

- Perhaps the most significant benefit of using the Java Native Interface is that it enables us to make use of a whole host of open source and business C/C++ libraries

- Boost provides a collection of libraries that are typically used as source code, rather library files; see:

  `http://www.boost.org`

- OpenSSL, available from `www.openssl.org`, provides a great C library for use across all platforms

- Libraries can be linked statically, using `.a` files, or dynamically, using `.o` files

OpenSSL

- To use OpenSSL, download the compressed file from:

  `www.openssl.org/source` e.g. `openssl-1.0.1e.tar.gz`

- Extract this file then build it from a command prompt be entering:
  - `./config shared` (just `./config` for static libraries)
  - `sudo make`
  - `sudo make install`

Ensure that there are no spaces in the path!

- On Ubuntu, OpenSSL will be installed in `/usr/local/ssl/` as:
  - `lib` folder containing two library files, e.g. `libcrypto.so.1.0.0` and `libssl.so.1.0.0`
  - `include` folder containing header files

- For `System.loadLibrary` to identify the library files, rename `libcrypto.so.1.0.0` and `libssl.so.1.0.0` by removing the trailing `.1.0.0`

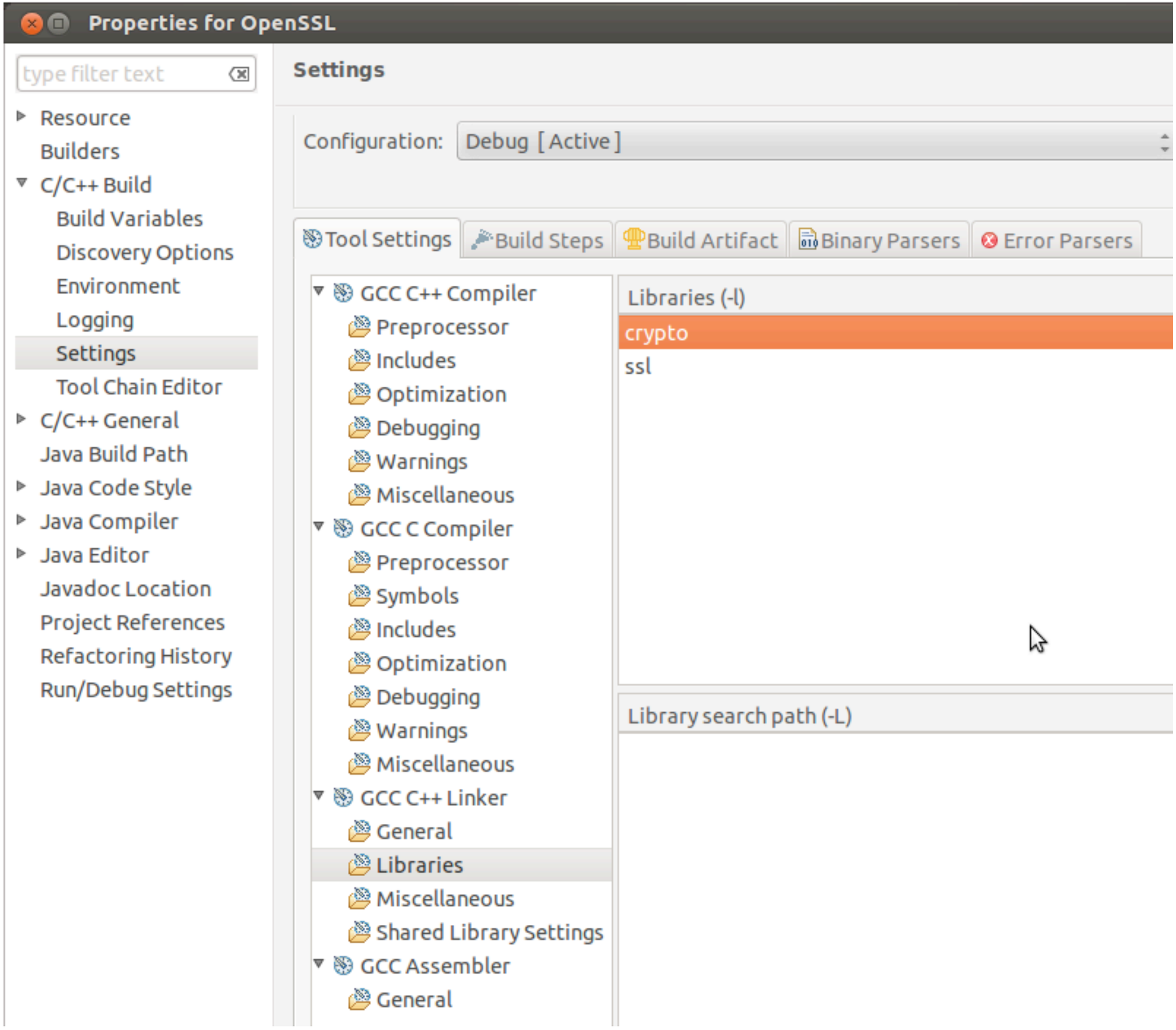
- To use OpenSSL in a JNI project, create a project in the usual way, with simple C/C++ and Java place-holder source code, and build it to produce the a library file in the `Debug` folder, then:

  - To the `Debug` folder, copy the renamed `libcrypto.so` and `libssl.so` files

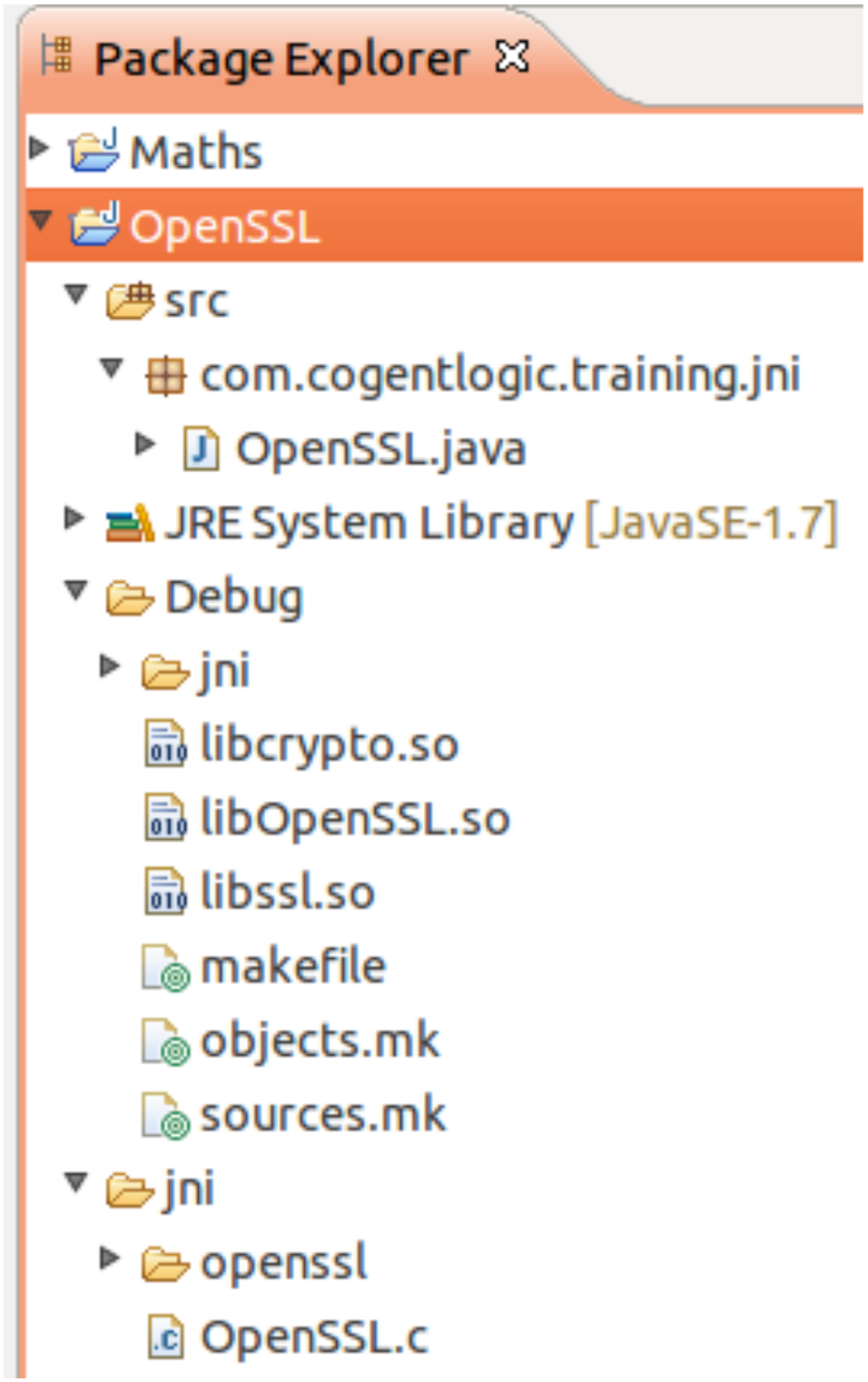
  - To the `jni` folder, copy the OpenSSL `include` folder, probably renaming it to `openssl`, for instance

  - Tell the linker to make use of both `crypto` and `ssl` libraries by adding them to `Libraries (-l)` under `C/C++ Build`, `Settings`, `GCC C++ Linker`, `Libraries`

...



- The project is constructed like this:



Do *not* add the path to the `openssl` include folder to the `Paths and Symbols` library settings.

Doing so would make them available using:

```
#include <rand.h>
```

but this could conflict with the system's version of OpenSSL (remember that system paths are already included in the library settings).

Instead, we can use:

```
#include "openssl/rand.h"
```

and there will be no confusion.

- To make use of the two OpenSSL libraries plus our own library:

```
static
{
  System.loadLibrary("crypto");
  System.loadLibrary("ssl");
  System.loadLibrary("OpenSSL");
}
```

- In the C/C++ source code, include the desired OpenSSL header files and make the desired library calls

- Build and debug/run as usual

- Sample Java code making use of OpenSSL through JNI: (*OpenSSL*)

```
public native byte[] encryptSymmetric(byte[] bytesPlain,
                                      byte[] bytesKey, byte[] bytesIV);
public native byte[] decryptSymmetric(byte[] bytesCipher,
                                      byte[] bytesKey, byte[] bytesIV);

public static void main(String[] args)
{
  byte[] bytesKey = new byte[32];
  byte[] bytesIV = new byte[16];
  String strPlainMessage =
                "The quick brown fox jumped over the lazy dogs!";
  OpenSSL openSSL = new OpenSSL();
  byte[] bytesCypher = openSSL.encryptSymmetric(
                strPlainMessage.getBytes(), bytesKey, bytesIV);
  System.out.println(Arrays.toString(bytesCypher));
  byte[] bytesRecovered =
      openSSL.decryptSymmetric(bytesCypher, bytesKey, bytesIV);
  String strRecovered = new String(bytesRecovered);
  System.out.println(strRecovered);
}
```

- In C, we have two native methods:

```
#include <stdlib.h>      // for malloc and free
#include <string.h>      // for strcpy
#include <jni.h>
#include "openssl/rand.h"
#include "openssl/evp.h"

JNIEXPORT jbyteArray JNICALL
    Java_com_cogentlogic_training_jni_OpenSSL_encryptSymmetric(
                JNIEnv* env, jobject thiz, jbyteArray bytesPlain,
                        jbyteArray bytesKey, jbyteArray bytesIV)
{ //... }

JNIEXPORT jbyteArray JNICALL
    Java_com_cogentlogic_training_jni_OpenSSL_decryptSymmetric(
                JNIEnv* env, jobject thiz, jbyteArray bytesCipher,
                        jbyteArray bytesKey, jbyteArray bytesIV)
{ //... }
```

- Extract from the C implementation in the sample project *OpenSSL:*

```
// Encrypt
int nCipherLen1;
int nCipherLen2;

EVP_CIPHER_CTX ctx;
EVP_CIPHER_CTX_init(&ctx);

EVP_EncryptInit(&ctx, EVP_aes_256_cbc(),
                      (unsigned char*)pbyteKey,
                      (unsigned char*)pbyteIV);

EVP_EncryptUpdate(&ctx, bytesCipherBuffer,
                        &nCipherLen1,
                        (unsigned char*)pbytePlain,
                        nLenPlain);

EVP_EncryptFinal(&ctx, bytesCipherBuffer + nCipherLen1,
                       &nCipherLen2);

EVP_CIPHER_CTX_cleanup(&ctx);

int nCipherLen = nCipherLen1 + nCipherLen2;
```

Java Native Interface with Eclipse and Android

JNI with the Android NDK

Jeff Lawson

Contents

- Android NDK
- Android NDK Installation
- Making Use of Native Code
- JNI Code
- Sample Android App with Native Code

Android NDK

- The Android Native Development Kit is a toolset that enables Android apps to use native code, i.e. compiled C and/or C++ (machine code)

- The NDK supports four instruction sets:
    - ARM v5TE and ARM v7-A
    - MIPS
    - x86

- Any one, two, three or all four instruction sets can be included in a single application package (.apk file)

- Android 1.5 (and later) apps can make Java Native Interface (JNI) calls into C and C++ code; Android 2.3 apps can make use of native activities

Android NDK Installation

- The Android NDK can be downloaded as a ZIP file from:

  `developer.android.com/sdk/ndk`

- Extract the ZIP file to a convenient location, referred to hereafter as `<NDK>`

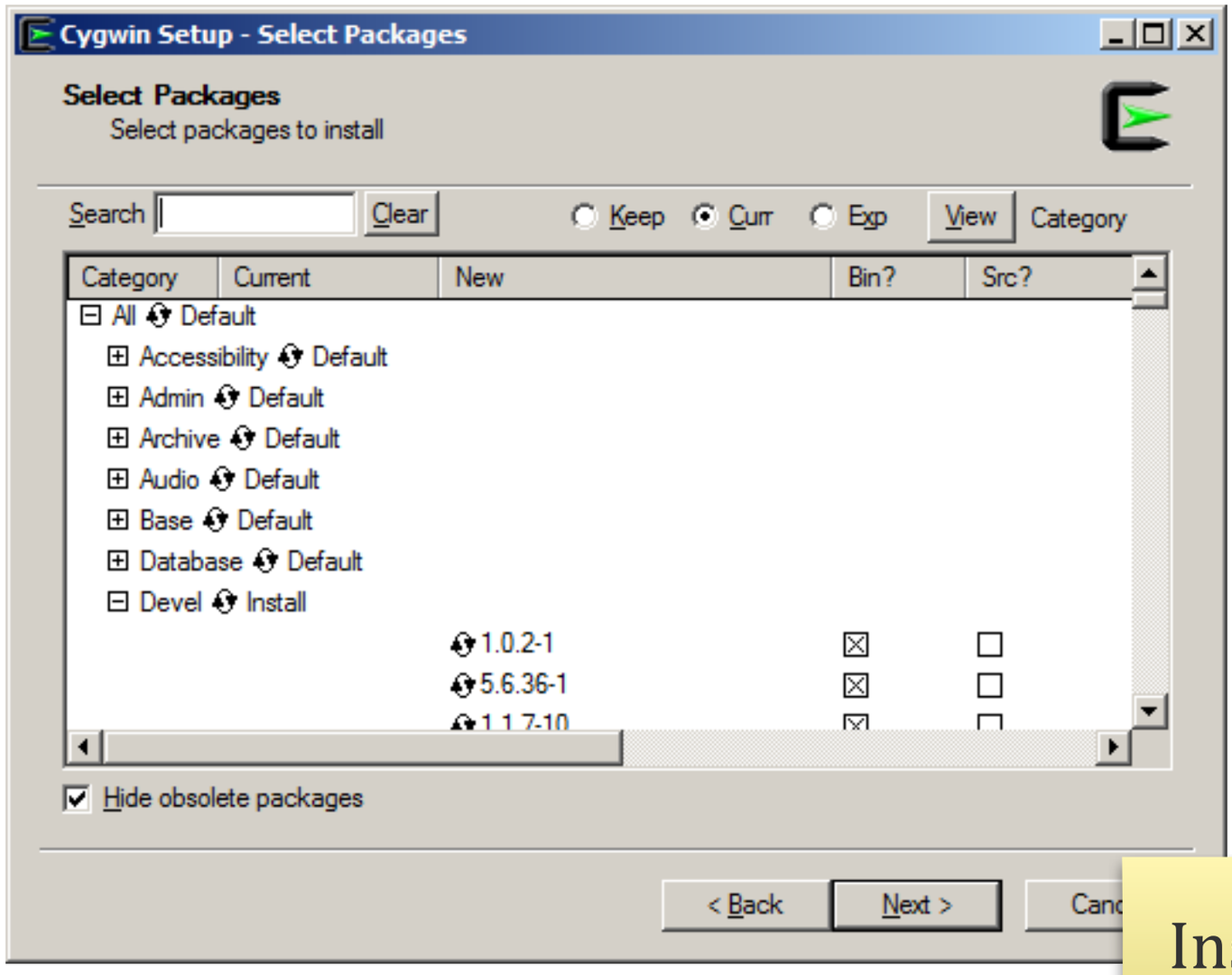
- The documentation is found in `<NDK>/docs`

- Check out `OVERVIEW.html`

- Platform-specific installation notes can be found in `INSTALL.html`, e.g. Windows requires the use of Cygwin 1.7 (Linux tools on Windows) with development tools in Devel ...

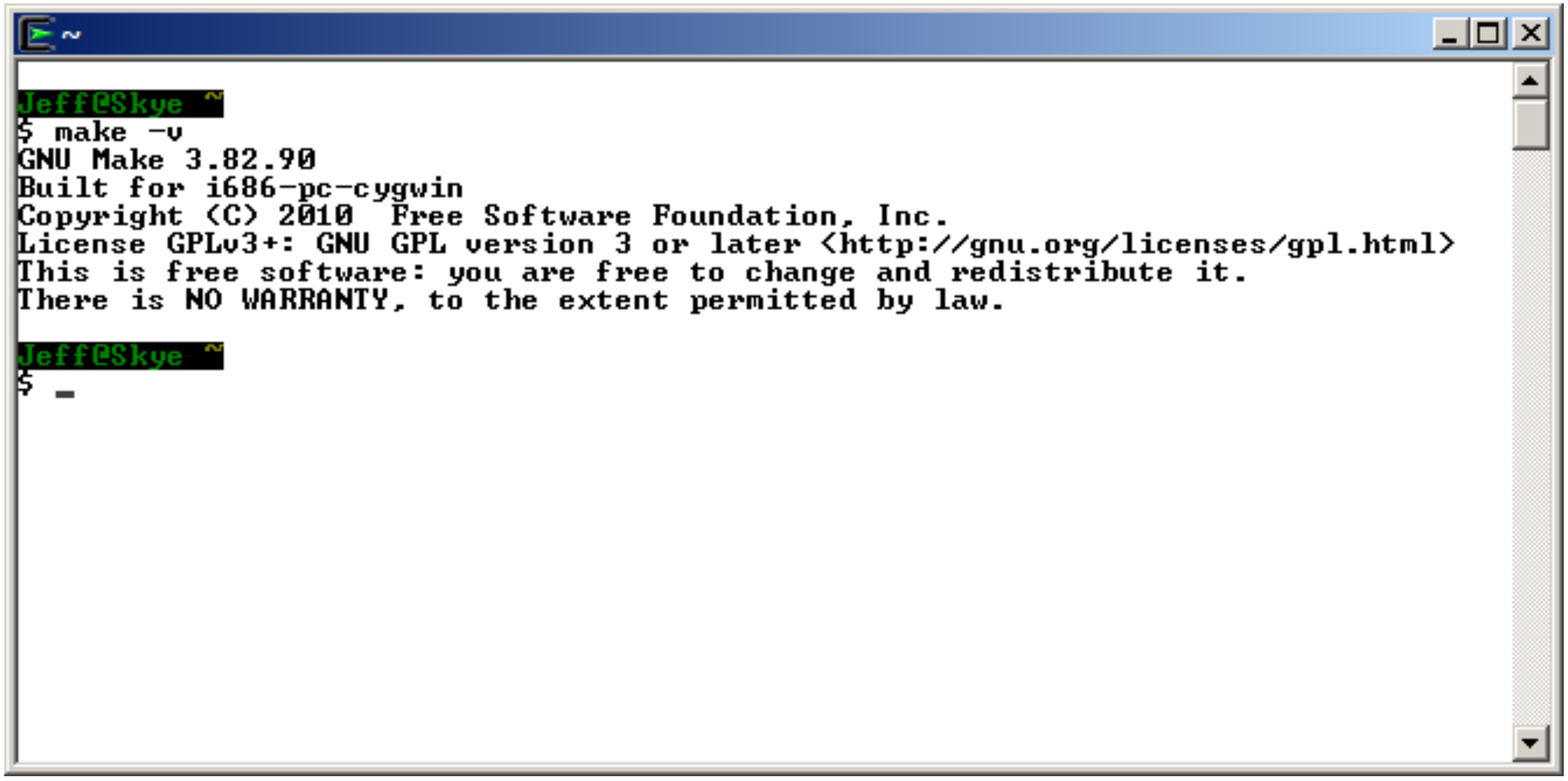
- Download and install Cygwin from `cygwin.com`

- Click *Default* along side the Devel package to change it to *Install* (for Devel, the default is that it isn't installed!)



Installation takes a *long* time!

- To test the Cygwin installation, open a bash shell and enter `make -v`



Making Use of Native Code

- Native code is typically held in a subdirectory of an Android project, called `jni`
- To describe the C/C++ code, a build script called `Android.mk` is created in `<project>/jni/` — see `ANDROID-MK.html`
- To target more than one system, a configuration file called `Application.mk` — see `APPLICATION-MK.html`
- Build native code by opening a command prompt at the `jni` directory and entering `<ndk>/ndk-build` (use `ndk-build clean` to remove defunct object code from previous builds)
- Then, build the Android app in the usual way and the native code will be included in the application package, `.apk`, file

- C/C++ code can make use of `<math.h>` plus other libraries and can access native Android APIs — see `STABLE-APIS.html`

- Declare the main C/C++ source code files, grouped as shared libraries, in `Android.mk` but *do not* declare header files for C code (C++ will need headers files for class declarations)!

- If `android:debuggable` is `true` in `AndroidManifest.xml`, then debuggable object files will be generated

- *Debug* requires Android 2.2 or higher

Sample Android App with Native Code

- Sample C code: (*MyLogic —Maths.c*)

```
#include <jni.h>

int fib(int n)
{
  return n <= 1 ? n : fib(n - 1) + fib(n - 2);
}

JNIEXPORT jint JNICALL
    Java_com_cogentlogic_training_jni_MainActivity_fibonacci(
                                    JNIEnv* penv, jobject obj, jint n)
{
  return fib(n);
}
```

- Sample C++ code: (*MyLogic —Greet.cpp*):

```
##include <jni.h>
#include "Greet.h"

static const char s_chGreeting[] = "Hello from native C++, ";

const char* Greet::getGreeting()
{
  return s_chGreeting;
}

#ifdef __cplusplus
extern "C" {
#endif

JNIEXPORT jstring JNICALL
    Java_com_cogentlogic_training_jni_MainActivity_greeting(
                        JNIEnv* env, jobject obj, jstring strName)
{
  // ...
}

#ifdef __cplusplus
}
#endif
```

- Java_com_cogentlogic_training_jni_MainActivity_greeting implementation:

```
Greet greet;
char chBuffer[100];
char* pDest = chBuffer;
const char* pSrc = greet.getGreeting();
while ((*pDest++ = *pSrc++))
  ;
const char* pchName = env->GetStringUTFChars(strName, 0);
if (!pchName)
  return 0;             // out of memory
--pDest;
pSrc = pchName;
while ((*pDest++ = *pSrc++))
  ;
*pDest = 0;
*--pDest = '!';
env->ReleaseStringUTFChars(strName, pchName);
return env->NewStringUTF(chBuffer);
```

- Calling C and C++ functions from Java requires a library to be created:
  - Sample `Android.mk`:

```
LLOCAL_PATH := $(call my-dir)
include $(CLEAR_VARS)
LOCAL_MODULE    := MyLogic
LOCAL_SRC_FILES := Greet.cpp Maths.c
include $(BUILD_SHARED_LIBRARY)
```

  - Sample `Application.mk`:

```
APP_PLATFORM := android-7
APP_ABI := all
```

- To compile C/C++ libraries o Mac OS X and Linux, open a command prompt at the jni folder and enter ndk-build:

```
jeffmacbookpro:jni Jeff$ ndk-build
Compile++ thumb  : MyLogic <= Greet.cpp
Compile thumb  : MyLogic <= Maths.c
SharedLibrary  : libMyLogic.so
Install        : libMyLogic.so => libs/armeabi-v7a/libMyLogic.so
Compile++ thumb  : MyLogic <= Greet.cpp
Compile thumb  : MyLogic <= Maths.c
SharedLibrary  : libMyLogic.so
Install        : libMyLogic.so => libs/armeabi/libMyLogic.so
Compile++ x86    : MyLogic <= Greet.cpp
Compile x86    : MyLogic <= Maths.c
SharedLibrary  : libMyLogic.so
Install        : libMyLogic.so => libs/x86/libMyLogic.so
Compile++ mips   : MyLogic <= Greet.cpp
Compile mips   : MyLogic <= Maths.c
SharedLibrary  : libMyLogic.so
Install        : libMyLogic.so => libs/mips/libMyLogic.so
```

- Ensure that there are no spaces in the path to the lib folder
  Refresh your Android project to see the newly generated files

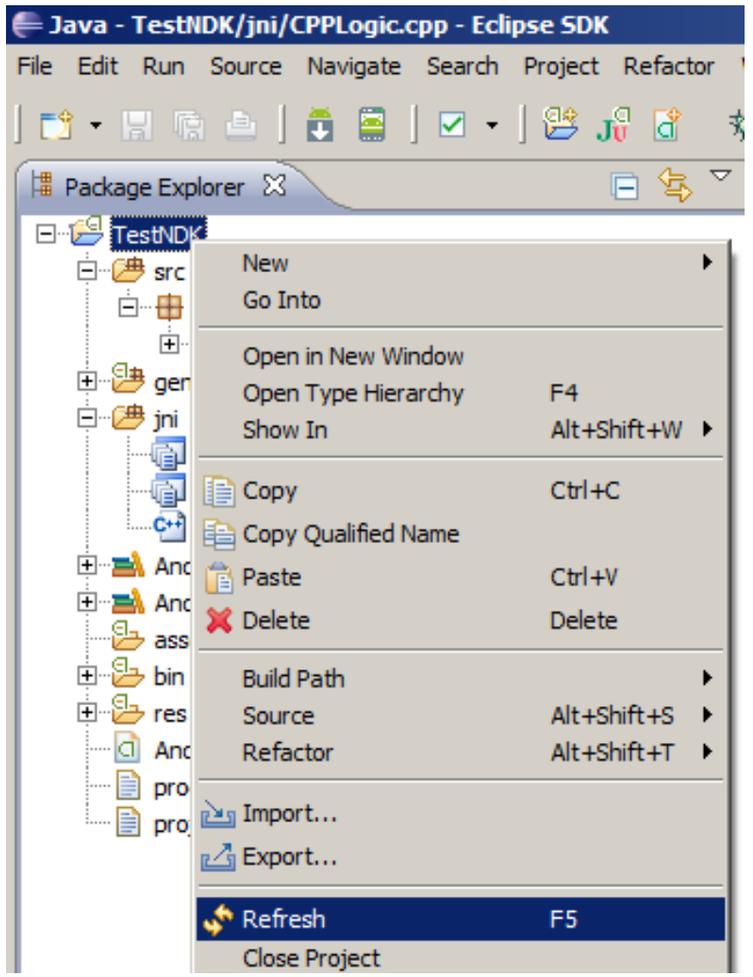
- To compile C/C++ libraries from Cygwin, for a project called `MyLogic` held on disk at `D:\workspace\`, first switch to the `jni` subdirectory with:

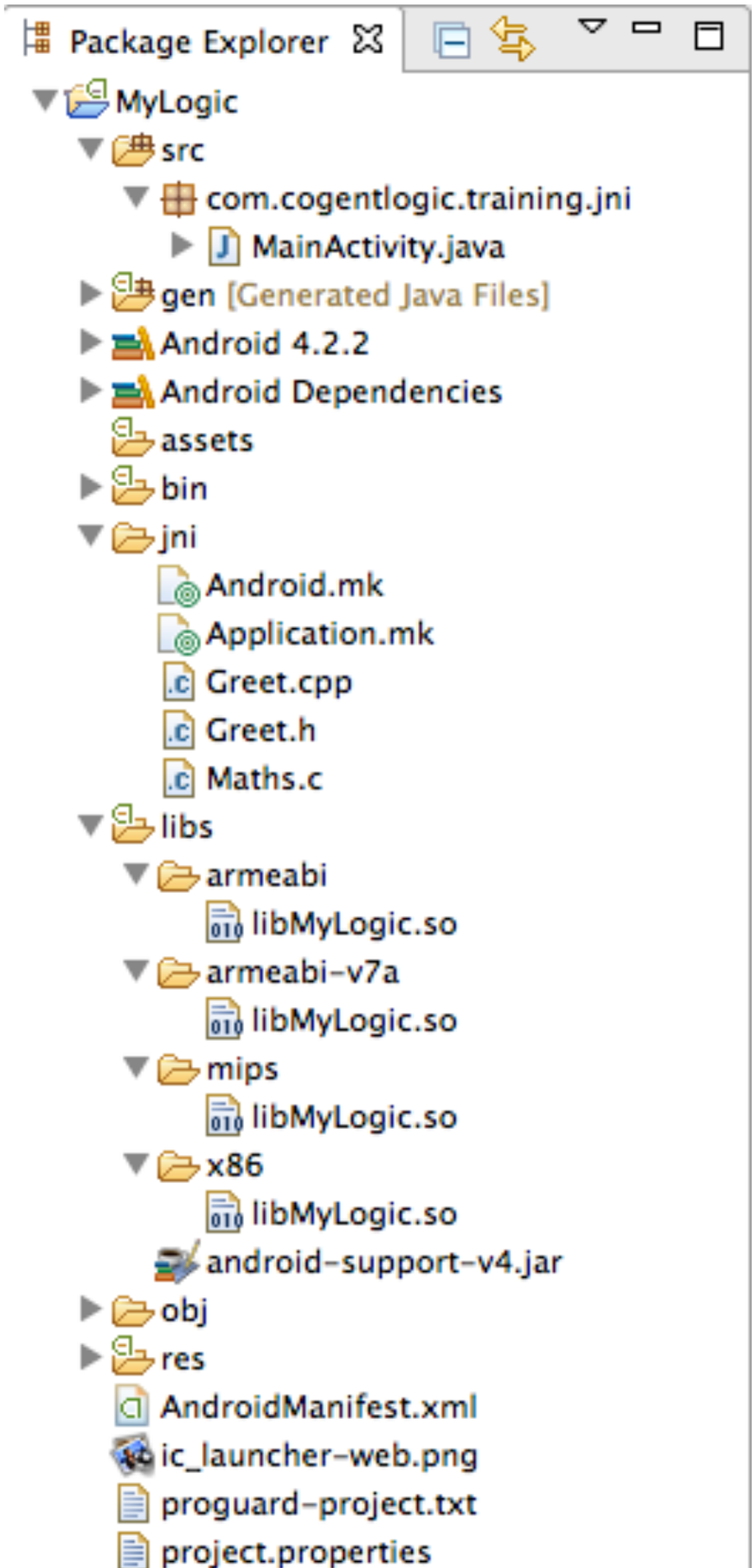
```
cd /cygdrive/d/workspace/MyLogic/jni
```

- For an NDK installation in `C:\android-ndk-r7c\`, invoke the compiler with:

```
/cygdrive/c/android-ndk-r7c/ndk-build
```

- To view the results of the compilation in Eclipse, refresh Package Explorer:





`APP_ABI := all` in `Application.mk`

results in the four object targets of

`armeabi`, `armeabi-v7a`, `mips` and `x86`

`LOCAL_MODULE := MyLogic` in `Android.mk`

results in the libraries `libMyLogic.so`

- Sample Java code: (*MyLogic — MainActivity.java*):

```
public class MainActivity extends Activity
{
  static {  System.loadLibrary("MyLogic");  }

  public native String greeting();
  public native int fibonacci(int n);

  @Override
  public void onCreate(Bundle savedInstanceState)
  {
     super.onCreate(savedInstanceState);

     // Call C++ and C
     TextView  tv = new TextView(this);
     tv.setText(greeting("Android "+ fibonacci(30)));
     setContentView(tv);
  }
}
```

Java Native Interface with Eclipse and Android

Using Native APIs

Jeff Lawson

Contents

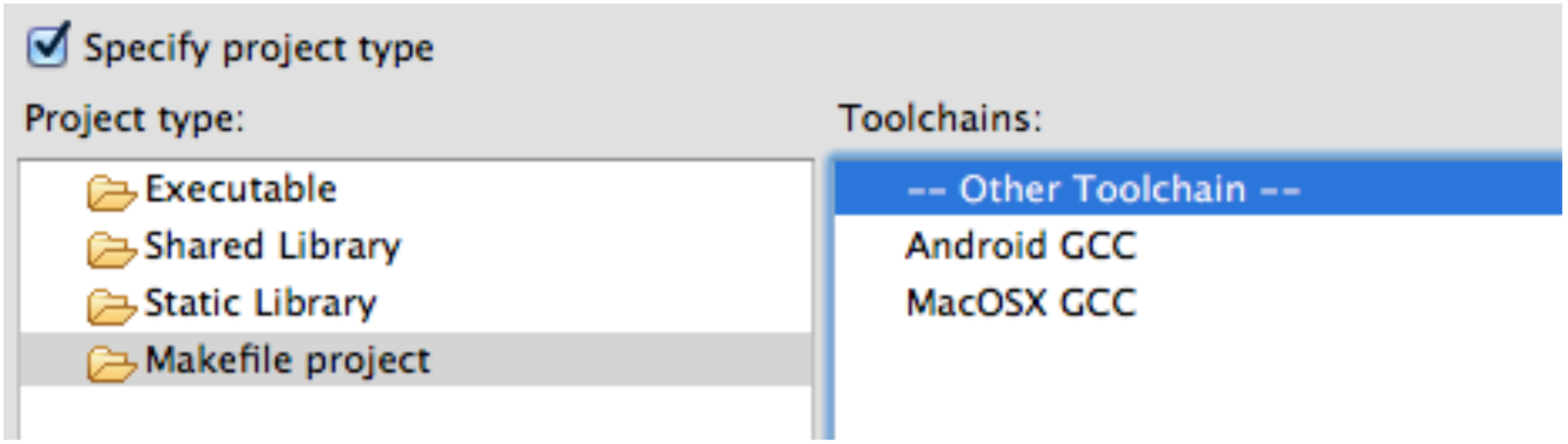
- Native APIs
- Android Logging API

Native APIs

- JNI provides a way for us to make use of the native platform that Java is supposed to dispense with!
- Android explicitly expose useful system components for access through JNI
- For details, see: `<NDK>/docs/STABLE-APIS.html`
- Note: the libraries that implement the system APIs are pre-installed on the host system, of course!
- We will look at native logging that makes use of the `/system/lib/liblog.so` library
- `Android.mk` references the logging library with `LOCAL_LDLIBS := -`

Android Logging API

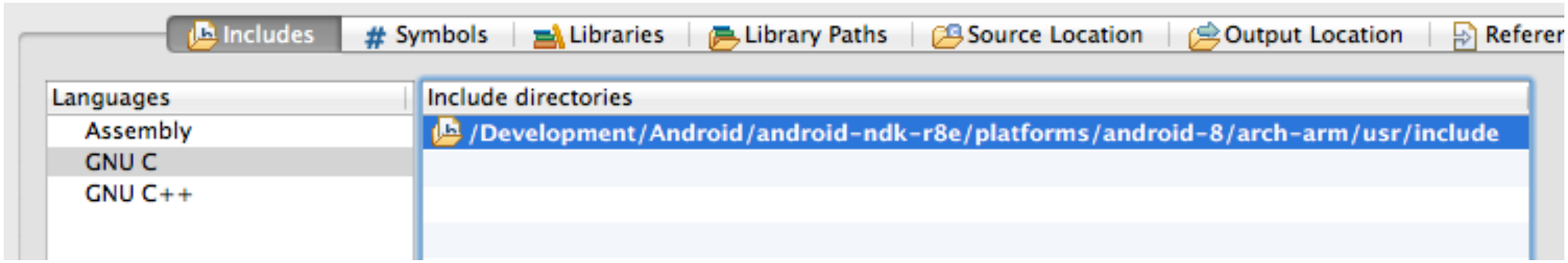
- To make use of the Android logging API, follow these steps:
  - Create an Android project and convert it for C/C++ use:
    - Select the `Specific project type` of `Makefile project` and select `-- Other Toochain --`



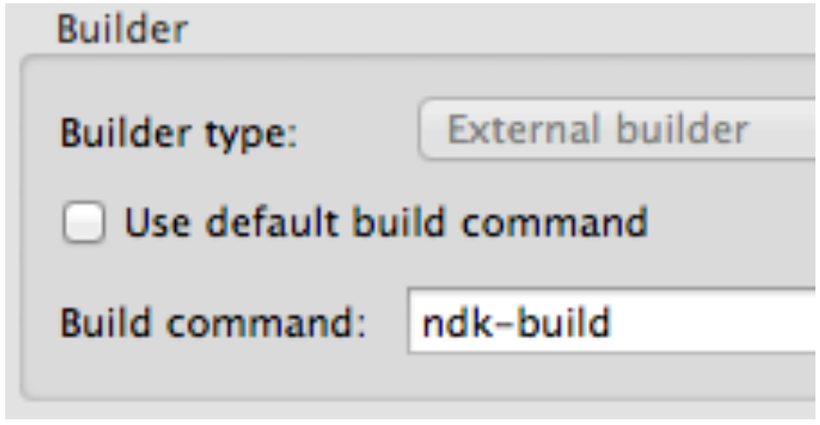
- Reference the Android header files:

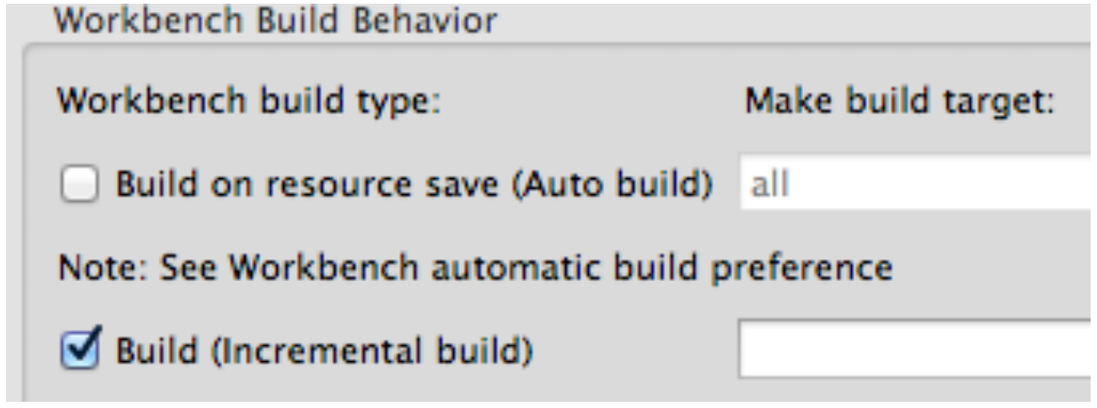
  `<NDK>/platforms/android-8/arch-arm/usr/include`

  in the usual place under `Paths and Symbols`



- In the project's `Properties`, select the `C/C++ Build` category, `Builder Settings`, clear the `Use default build command` checkbox and enter `ndk-build`; then, on the `Behaviour` tab, along-side `Build (Incremental build)`, clear the `all` entry:





- Also, under `C/C++ Build`, select the Environment category then add a `PATH` environment variable that references the NDK (so `ndk-build` can be found):

```
usr/bin:/bin:/usr/sbin:/sbin:/Development/
                                Android/android-ndk-r8e
```

Environment variables to set

Variable	Value	Origin
CWD	/Jeff/JNI_Training/SampleCode/MacOSX/AndroidWorkspace/Logr/	BUILD SYSTEM
PATH	/usr/bin:/bin:/usr/sbin:/sbin:/Development/Android/android-ndk-r8e	USER: CONFIG
PWD	/Jeff/JNI_Training/SampleCode/MacOSX/AndroidWorkspace/Logr/	BUILD SYSTEM

- Add a `jni` folder to the project and create the following files there:
  - `Android.mk`
  - `Application.mk`
  - `Logging.c`

- To `Android.mk`, add:

```
LOCAL_PATH := $(call my-dir)
include $(CLEAR_VARS)
LOCAL_MODULE    := Logging
LOCAL_SRC_FILES := Logging.c
LOCAL_LDLIBS := -llog
include $(BUILD_SHARED_LIBRARY)
```

- To `Application.mk`, add:

```
APP_PLATFORM := android-7
APP_ABI := armeabi
```

- To Logging.c, add:

```
#include <jni.h>
#include <android/log.h>

JNIEXPORT void JNICALL
   Java_com_cogentlogic_training_jni_MainActivity_log(
                  JNIEnv* env, jobject thiz, jstring strMsg)
{
  __android_log_write(ANDROID_LOG_ERROR, "LJC",
                        "on_call_tsx_state");
  const char* pchMsg = (*env)->GetStringUTFChars(env,
                                                  strMsg, 0);
  if (pchMsg)
  {
     int nThing = 123;
     __android_log_print(ANDROID_LOG_INFO,
                           "LJC", "%s : %d\n", pchMsg, nThing);
  (*env)->ReleaseStringUTFChars(env, strMsg, pchMsg);
}
```

- In the Android activity, load the library, declare the native method and invoke the native method:

```
static
{
  System.loadLibrary("Logging");
}

private native void log(String strMsg);

protected void onCreate(Bundle savedInstanceState)
{
  super.onCreate(savedInstanceState);
  setContentView(R.layout.activity_main);

  log("LCJ");
}
```

- The output will appear in ADT's LogCat, e.g.

```
06-12 15:27:54.298: E/LJC(6178): on_call_tsx_state
06-12 15:27:54.298: I/LJC(6178): LCJ : 123
```

Java Native Interface with Eclipse and Android

Debugging Native Code in Eclipse

Jeff Lawson

Contents

- C/C++ Debugging for Android

C/C++ Debugging for Android

- Android C/C++ debugging has been supported since API Level 9, hence, specify `android:minSdkVersion="9"` in the manifest

- The `android:debuggable` attribute is required in the manifest's `application` element (ignore the warning--that's just for Java!)

- To create debug support files in your project:
  - Run the project in Debug mode
  - Open a command prompt at the project's path
  - Enter: `ndk-gdb` then exit with `quit`

- These files will be added to `obj/local/armeabi` :

  `app_process` `gdb.setup` `libc.so`

- We need to edit `gdb.setup` but we don't want our changes to be overwritten when `ndk-gdb` next runs!
- Copy `gdb.setup` and name the copy `gdb2.setup` to
- Open `gdb2.setup` and delete the line comprising: `target remote :5039`
- We now need to configure several project settings...
- We need a C/C++ debug configuration:
  - From the `Run` menu, select `Debug Configurations...`
  - Select the `C/C++ Application` category and click the `New` button
  - Specify a name for the configuration and, on the `Main` tab, browse for and set the path to `obj/local/armeabi/app_process`

...

Debug Configurations

Create, manage, and run configurations

type filter text

- Android Application
  - Logr
  - MyLogic
- Android JUnit Test
- Android Native Applicati
- C/C++ Application
  - Logr Debug
- C/C++ Attach to Applic
- C/C++ Postmortem Deb
- C/C++ Remote Applicat
- C/C++ Unit
- Java Applet
- Java Application
- JUnit
- Launch Group
- Remote Java Application

Filter matched 16 of 17 items

Name: Logr Debug

Main | Arguments | Environment | Debugger | Source | Common

C/C++ Application:

/Jeff/JNI_Training/SampleCode/MacOSX/AndroidWorkspace/Logr/obj/local/armeabi/app_process

Variables... Search Project... Browse...

Project:

Logr Browse...

Build (if required) before launching

Build configuration: Default

Select configuration using 'C/C++ Application'

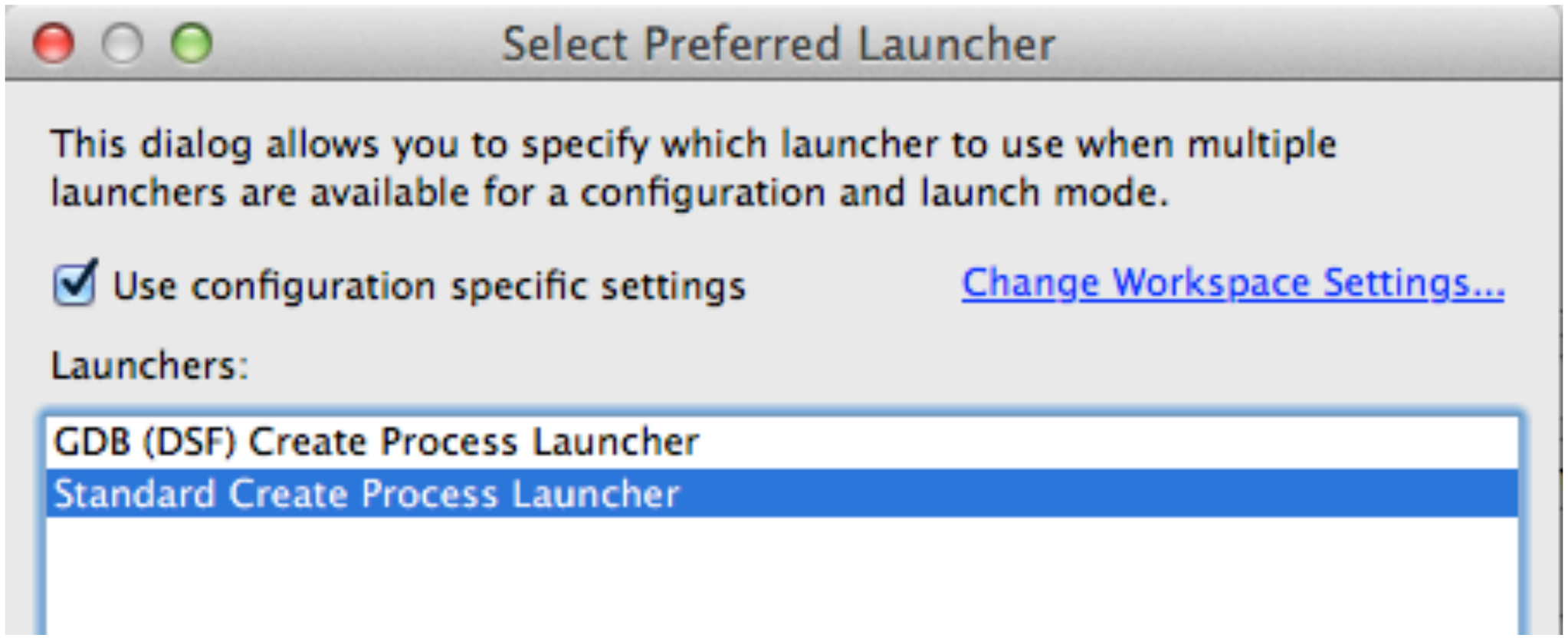
Enable auto build Disable auto build

Use workspace settings Configure Workspace Settings...

Using GDB (DSF) Create Process Launcher - Select other... Apply Revert

Close Debug

- Click blue `Select other...` link at the bottom of the `Main` tab and select the `Standard Create Process Launcher` option:



- On the `Debugger` tab, select the `gdbserver` debugger

- Set the path to GDB debugger, e.g. `/Development/Android/android-ndk-r8e/toolchains/arm-linux-androideabi-4.7/prebuilt/darwin-x86_64/bin/arm-linux-androideabi-gdb`

- Set the path to GDB command line,
  e.g. `...obj/local/armeabi/gdb2.setup`

- On the `Connection` sub-tab, specify:

  - Type: `TCP`

  - Port number: `5039`

- Click the `Apply` button and close the dialog

- Find the `ndk-gdb` file in the `<NDK>` and copy it with the name:
  `ndk-gdb-eclipse`

- Open `ndk-gdb-eclipse` and use `#` to comment-out the line:
  `$GDBCLIENT -x `native_path $GDBSETUP``

- Finally, debug! :
    - Set a breakpoint in your Java code
    - Set a breakpoint in your C/C++ code
    - Start the Android app in Debug mode
    - When the app stops at the Java breakpoint:
        - Open a command prompt at the project's folder
        - Enter the command: `ndk-gdb-eclipse`
        - From the `Run` menu, select `Debug Configurations...`
        - Select the named `C/C++ Application` configuration
        - Click the `Debug` button
    - Click the `Continue` button in the Java debugger
    - The debugger will stop at your C/C++ breakpoint