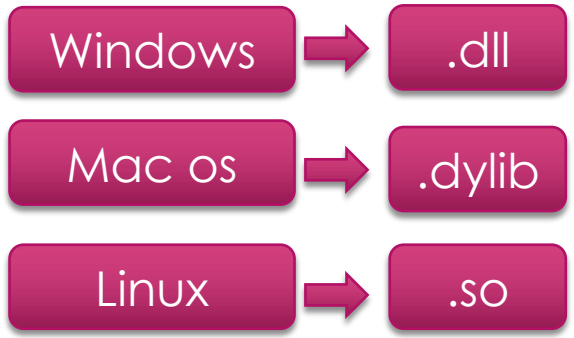
Java Native Interface (JNI)

What is JNI?

- Java native interface (**JNI**) allows calling native libraries
- Native libraries
  - Typically **C** or **C++**
- Many useful scientific libraries are written in C or C++
  - Example: GNU Scientific Library (www.gnu.org/software/gsl/)
  - GSL reference manual: https://www.gnu.org/software/gsl/doc/latex/gsl-ref.pdf
- JNI provides the bridge to call GSL functions
- Obviously, we need to have some knowledge of C or C++
- **General procedure to create the dynamic libraries compatible with java**
  - Step 1: add "**native**" methods to your class and then compile it into class file
  - Step 2: use "javah -jni" or "javac –h" to create the header file for the compiled class file
  - Step 3: create a C or C++ project and write the source code by importing the header file
  - Step 4: compile the C or C++ code and create the shared library (***.dll**, ***.dylib**, ***.so**)



Java Native Interface

- **General procedure:**
  - Step 4: compile the C or C++ code and create the shared library (***.dll**, ***.dylib**, ***.so**)

Dynamic libraries

- After creating the native library with JNI headers
  - Step 1: add the dynamic library to the java path: -Djava.library.path = /path/to/dynamic/library
  - Step 2: load the dynamic library before calling the native method
    - Typically done via static initializers

static initializer

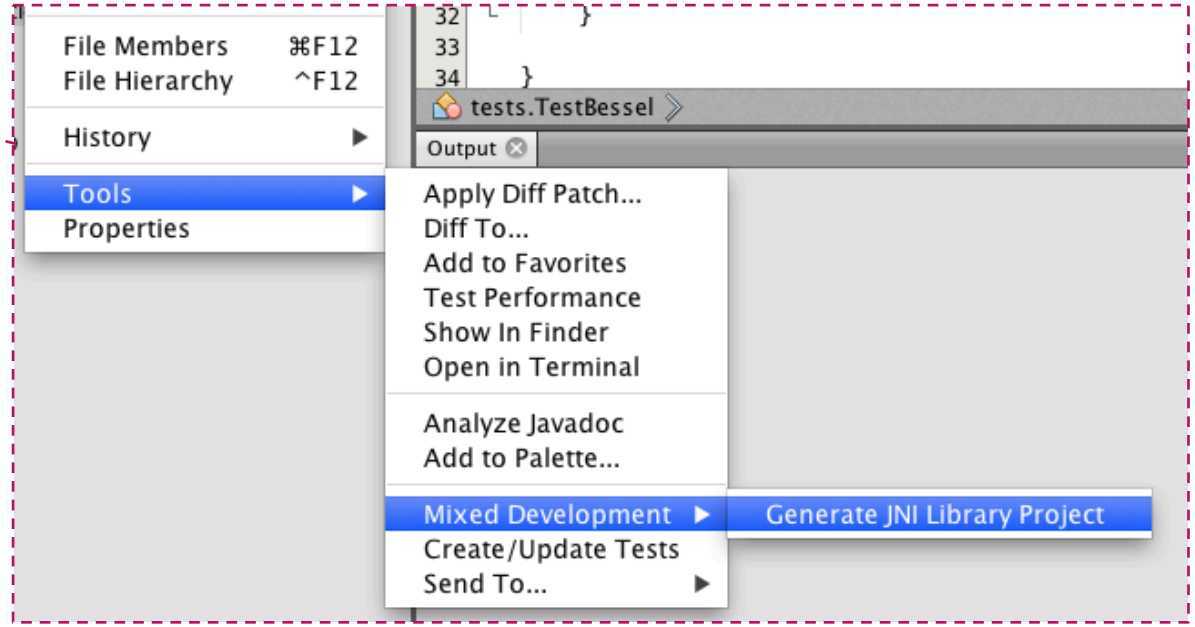
```
static {
      System.loadLibrary("library name");
}
```

- How to properly call native libraries
  - Naming on Mac os: **lib******.dylib
  - When using System.LoadLibrary("library name") **drop** "lib" and ".dylib"
  - If you do not drop "lib" from the beginning, it will NOT be loaded by java
  - You **HAVE TO** drop "lib" from the beginning of the name of the library
- How to do mixed development?
  - The **best** IDE for this in my opinion is "NetBeans" IDE

Almost all of the process is automated in NetBeans IDE

Java Native Interface

- How to do mixed development?
  - The **best** IDE for this in my opinion is "NetBeans" IDE → Almost all of the process is automated in NetBeans IDE
- We need to install C/C++ plugin for NetBeans
- After adding native methods to your java file
  - Right click on the class file (.java file)
  - Tools → Mixed development
  - Generate JNI Library Project
  - This creates a new C/C++ **project**
- If adding more native methods
  - Need to update header file
  - Right click on the class name inside the ".java" file and select:
    - Mixed development → Generate JNI header
    - This updates the previously created header file



Don't forget to point the **linker** to the installed GNU scientific library on your computer

Java Native Interface

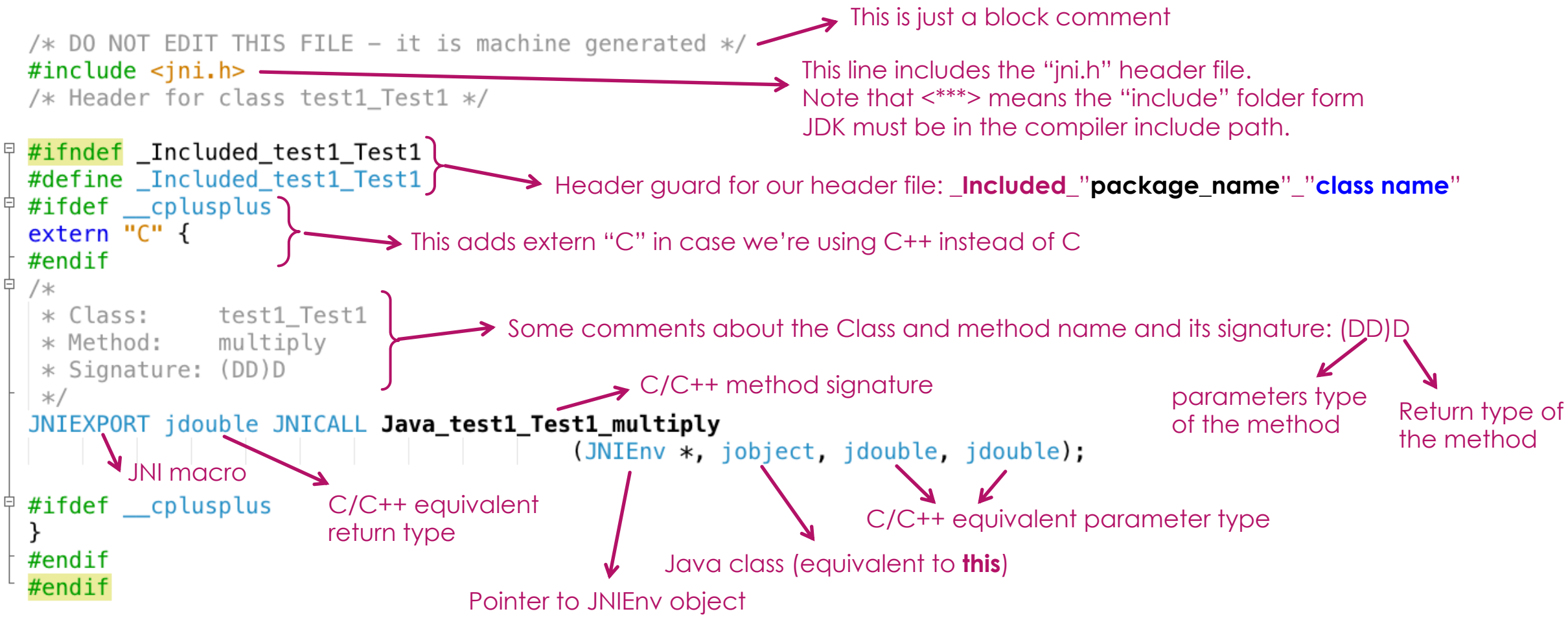
- What is the **downside** of using JNI?
  - It ties us to a specific platform → we lose platform independence
  - C/C++ libraries are platform dependent
- General format of header functions for native methods

Java_"package name"_"class name"_"method name"

- "package name" can be *multiple* packages separated by underscore _
  - Example: java.lang.Math.sin → **Java**_java_lang_Math_sin
- General types of parameters in JNI method calls
  - JNIEnv → a pointer to **JNIEnv object**: mainly used to call java classes and their instance variables
  - Calling java class: jobject → equivalent to **this** keyword
  - Primitives: jint, jdouble, jboolean, jchar, jfloat → equivalent C/C++ primitive types → **typedef**
  - Classes: jstring, jclass → jclass is used for reflection

Java Native Interface

- Let's have a closer look to the header file:



JNI: How to Check the Platform


```
public class OSChecker {

	private static String OS = System.getProperty("os.name").toLowerCase();

	public static void main(String[] args) {
		System.out.println(OS);
		if (isWindows()) {
			System.out.println("This is Windows");
		} else if (isMac()) {
			System.out.println("This is Mac");
		} else if (isUnix()) {
			System.out.println("This is Unix or Linux");
		} else if (isSolaris()) {
			System.out.println("This is Solaris");
		} else {
			System.out.println("Your OS is not support!!");
		}
	}

	public static boolean isWindows() {
		return (OS.indexOf("win") >= 0);
	}

	public static boolean isMac() {
		return (OS.indexOf("mac") >= 0);
	}

	public static boolean isUnix() {
		return (OS.indexOf("nix") >= 0 || OS.indexOf("nux") >= 0);
	}

	public static boolean isSolaris() {
		return (OS.indexOf("sunos") >= 0);
	}
}
```

Using **System Property**: "os.name"

In order to retain platform independence in our java code we **MUST** provide a dynamic library for all the platforms that we want to support.

Check of os name contains "win"

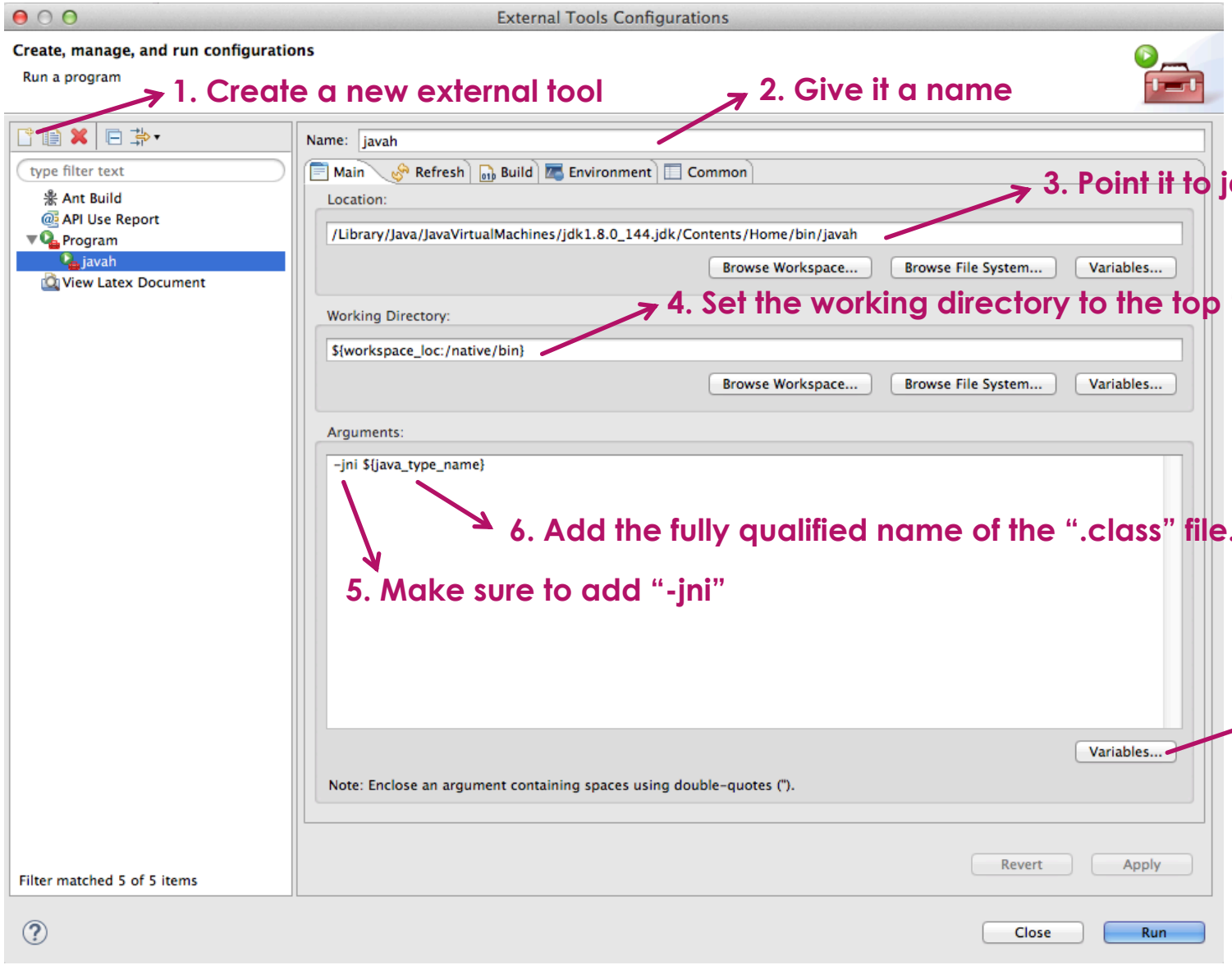
Check of os name contains "mac"

Check the OS and load the corresponding dynamic library using **System.loadLibrary(***)**

JNI in Eclipse IDE

- Unlike NetBeans IDE, we need to manually configure Eclipse IDE for mixed development
- **Step 1**: install C/C++ CDT plugin for Eclipse IDE through the Eclipse marketplace
- **Step 2**: Create java project and add "**native**" methods to the java classes
- **Step 3**: Configure the "external tools" for running "javah –jni" tool
  - Set the working directory to the top-level "bin" folder of the project
- **Step 4**: Run "javah –jni" on the current class that contains "native" methods
- **Step 5**: Create a C++ project and copy the header file to the "src" folder
- **Step 6**: Create the ".cpp" file and add the implementation of the methods
- **Step 7**: Add "include" folders of JDK to the compiler include paths
- **Step 8**: Set the target artifact to "**dynamic** library"
- **Step 9**: Compile the C++ project and generate the dynamic library
- **Step 10**: Point the –Djava.library.path to the folder containing the dynamic library
- **Step 11**: Use a static initializer to load the dynamic library in java

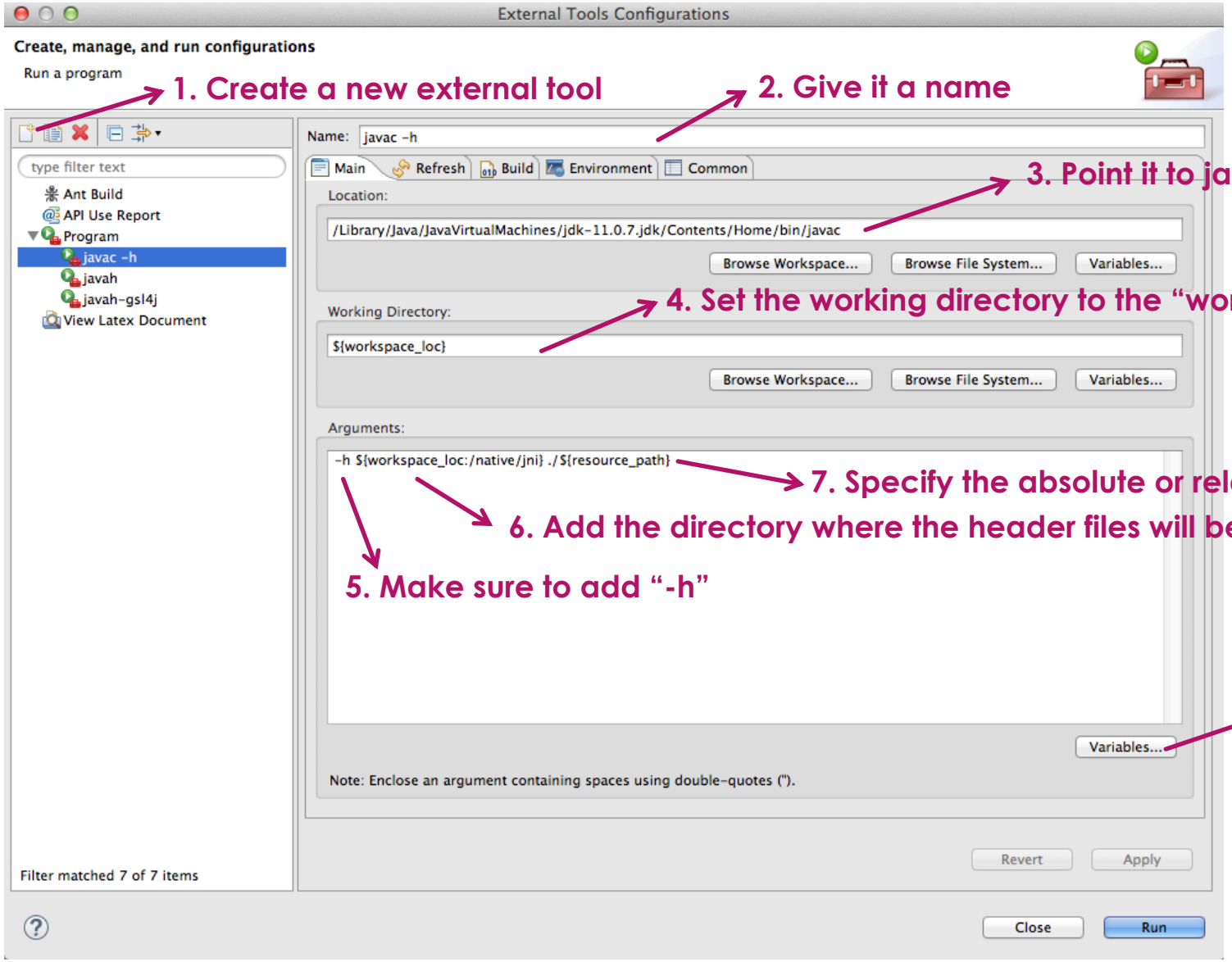
JNI in Eclipse IDE: External Tool



JDK <= 8

- "**javah**" acts on binary files "**.class**"
- "**-d /path/to/folder**" can be set to generate the header files in a given folder

JNI in Eclipse IDE: External Tool



JDK >= 9

- "**javah**" was removed from JDK 9
- Need to point to "**javac**" as the external tool

- ${resource_path}
- ${resource_loc}

JNI: Overloading & Overriding

- Can we overload a **native** method?
  - **Yes, we can** → JNI uses fully qualified name for C++ methods → **unique signature in C++**
  - The overloading method can be native or regular method.
- Can we override a **native** method?
  - **Yes, we can** → JNI uses fully qualified name for C++ methods → unique signature in C++
  - The overriding method can be native or regular method.
- Can we have static **native** methods?
  - **Yes, we can** → jobject is replaced with jclass in C++
- Should we load the native library in the superclass or the subclass?
  - JVM loads the entire superclass hierarchy when loading a subclass into the memory
  - Static initializer → is executed the first time a class is loaded into the memory
  - Accessing a static field or method → loads the class into the memory
  - Instantiating a class for the first time → loads the class into the memory

JNI: Signature of Overloading

- Can we overload a **native** method?
  - **Yes, we can** → JNI uses fully qualified name for C++ methods → **unique signature in C++**
  - The overloading method can be native or regular method.
- What is the JNI method signature of overloading a method?

Two underscores

Java_"package name"_"class name"_"method name"__"method parameter signature"

- **Overloading native with different parameters**

```
package test.overloading;

public class OverloadNative {

	static {
		System.loadLibrary("test_native");
	}

	public static native double func(double x) ;
	public static native double func(double x, double y) ;

}
```

```
/*
 * Class:     test_overloading_OverloadNative
 * Method:    func
 * Signature: (D)D
 */
JNIEXPORT jdouble JNICALL Java_test_overloading_OverloadNative_func__D
  (JNIEnv *, jclass, jdouble);

/*
 * Class:     test_overloading_OverloadNative
 * Method:    func
 * Signature: (DD)D
 */
JNIEXPORT jdouble JNICALL Java_test_overloading_OverloadNative_func__DD
  (JNIEnv *, jclass, jdouble, jdouble);
```

NOTE: / replaced by _ , ; replaced by _2 , [replaced by _3 , _ replaced by _1

JNI: Signature of Overloading

- Can we overload a **native** method?
  - **Yes, we can** → JNI uses fully qualified name for C++ methods → **unique signature in C++**
  - The overloading method can be native or regular method.
- What is the JNI method signature of overloading a method?

Two underscores

Java_"package name"_"class name"_"method name"__"method parameter signature"

- **Overloading native with non-native method** → No need to add method params signature

```
package test.overloading;

public class OverloadNative {

	static {
		System.loadLibrary("test_native");
	}

	public static native double func(double x) ;
	public static double func(double x, double y) {
		return x+y ;
	}
}
```

```
/*
 * Class:     test_overloading_OverloadNative
 * Method:    func
 * Signature: (D)D
 */
JNIEXPORT jdouble JNICALL Java_test_overloading_OverloadNative_func
  (JNIEnv *, jclass, jdouble);
```

Non-native method requires an implementation body.

JNI: Signature of Overloading

- Can we overload a **native** method?
  - **Yes, we can** → JNI uses fully qualified name for C++ methods → **unique signature in C++**
  - The overloading method can be native or regular method.
- What is the JNI method signature of overloading a method?

Two underscores

Java_"package name"_"class name"_"method name"__"method parameter signature"

- **Overloading native with different return types**

```
package test.overloading;

public class OverloadNative2 {

	static {
		System.loadLibrary("test_native");
	}

	public static native double func(double x) ;
	public static native double[] func(double[] x) ;

}
```

```
/*
 * Class:     test_overloading_OverloadNative2
 * Method:    func
 * Signature: (D)D
 */
JNIEXPORT jdouble JNICALL Java_test_overloading_OverloadNative2_func__D
  (JNIEnv *, jclass, jdouble);

/*
 * Class:     test_overloading_OverloadNative2
 * Method:    func
 * Signature: ([D)[D
 */
JNIEXPORT jdoubleArray JNICALL Java_test_overloading_OverloadNative2_func___3D
  (JNIEnv *, jclass, jdoubleArray);
```

NOTE: / replaced by _ , ; replaced by _2 , [replaced by _3

JNI: Execution of Static Initializers

- Order of execution when loading a class into the memory → entire inheritance hierarchy is loaded
  1) Static initializer of the **super**class: static {***}
  2) Static initializer of the **sub**class: static {***}
  3) Instance initializer of the **super** class: {***} → always copied to the beginning of the constructor
  4) Constructor of the **super**class (super(***)) → first line of the constructor of the subclass
  5) Instance initializer of the **sub**class: {***} → always copied to the beginning of the constructor
  6) Constructor of the **sub**class
- **Example**: Animal → Mammal → Cat

```
public class Cat extends Mammal { … }
```

```
public class Mammal extends Animal { … }
```

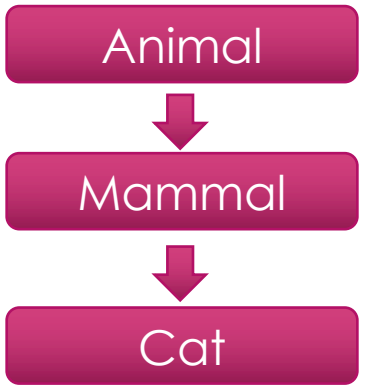
```
public class Animal { … }
```

```
public static void main(String[] args) {
      // instantiate subclass --> Cat
      Animal cat = new Cat() ;
}
```

Result of execution

```
Inside Animal static initializer
Inside Mammal static initializer
Inside Cat static initializer
Inside Animal instance initializer
Inside Animal constructor
Inside Mammal instance initializer
Inside Mammal constructor
Inside Cat instance initializer
Inside Cat constructor
```

Inheritance hierarchy



JNI: Primitive Types

- An overview of primitive types mapping from java to C/C++
- Signature of different types

Java Type	Native Type	Description
boolean	jboolean	8 bits, unsigned
byte	jbyte	8 bits, signed
char	jchar	16 bits, unsigned
double	jdouble	64 bits
float	jfloat	32 bits
int	jint	32 bits, signed
long	jlong	64 bits, signed
short	jshort	16 bits, signed
void	void	N/A

Used for types in C/C++

Java Type	Signature
boolean	Z
byte	B
char	C
double	D
float	F
int	I
long	J
short	S
void	V (nothing)

Used for parameter signature (e.g. overloading)

Java Type	Signature
object	**L***fully-qualified-class*;
type[]	**[***type*
method signature	(*arg-types*) *ret-type*

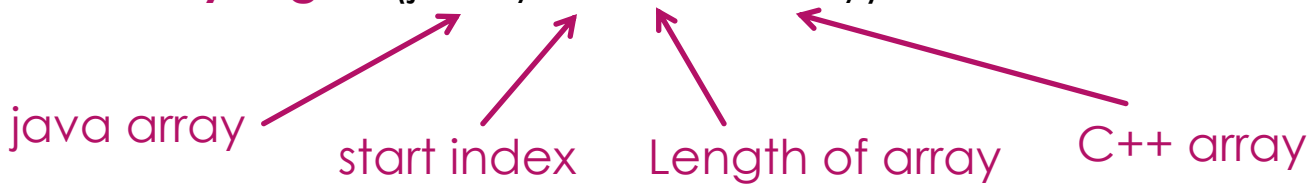
Used for parameter signature (e.g. overloading)

```
120 typedef union jvalue {
121     jboolean z;
122     jbyte    b;
123     jchar    c;
124     jshort   s;
125     jint     i;
126     jlong    j;
127     jfloat   f;
128     jdouble  d;
129     jobject  l;
130 } jvalue;
```

jni.h

JNI: Working with Primitive Arrays

- **Primitive**: double → jdouble , float → jfloat , int → jint (signed integer)
- Primitive **arrays**: double[] → jdoubleArray, float[] → jfloatArray, int[] → jintArray **→ j<type>Array**
- All these types are subtypes of **jobject**
- We need to use "JNIEnv *env" to create these arrays and fill them up with the values from java
- **Step 1**: create a new jdoubleArray:
  - **jdoubleArray** jarray = env -> **NewDoubleArray**(len) ; → we pass in the length of array
- **Step 2**: create the C++ array and fill it with the values you have
  - jdouble carray[] = {1.1, 2.2, 3.3, 4.4} ;
- **Step 3**: use "JNIEnv *env" to copy the values from C++ array to java array:
  - env -> **SetDoubleArrayRegion**(jarray, 0, len, carray)



- Use **env -> GetDoubleArrayRegion(jarray, 0, len, carray)** to copy from java array to C++ array

JNI: Working with Object Arrays

- **Array of objects → jobjectArray**
- Example: double[][] x : 2D array is an array of 1D arrays → it translates into jobjectArray
- We need to use "JNIEnv *env" to create these arrays and fill them up with the values from java
- **Step 1**: create a new jobjectArray:
  - **jobjectArray** j2darray = env -> **NewObjectArray**(len1) ; → we pass in the length of array

Number of rows in double[][]

- **Step 2**: create the C++ array and fill it with the values you have
  - jdouble carray[] = {1.1, 2.2, 3.3, 4.4} ;
- **Step 3**: use "JNIEnv *env" to copy the values from C++ array to java array:
  - env -> **SetDoubleArrayRegion**(jarray, 0, len2, carray) ;

Number of columns in double[][]

- **Step 4**: cast the jdoubleArray to jobject
  - jobject jarray_obj = (jobject) jarray ;
- **Step 5**: set each element of the jobjectArray:
  - env -> **SetObjectArrayElement**(j2darray, k, jarray_obj) ;

JNI: Working with Strings

- **String** in java is an object with **UTF-16** encoding (signature: Ljava/lang/String;)
  - char data type is two bytes in java → **char** is mapped to **jchar** (unsigned short)
- **String** in C++ → char data type one byte (unsigned) in C/C++
  - C-style string literal: **char*** → we can use sizeof() operator
  - C++ string object: **std::string** → header: <string> ➔ string.c_str()
- Java only works with heap which is dynamic memory allocation
  - String objects and String literals are on the heap (string pool, Stirng.intern())
- In order to create or access java Strings we need to **ask JVM**
- Important method in JNIEnv:
  - Creating new java strings: **jstring str = env -> NewStringUTF("Hello") ;**
  - Accessing an existing java string: **const char* str = env -> GetStringUTFChars(jstring, isCopy)**
  - Releasing String: **env -> ReleaseStringUTF(jstring, char*)**
- Must use **env → release(…)** for any method of env that returns a pointer on **native heap**

2-bytes unsigned int

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

This is a pointer

const char *

jboolean *

Garbage collector doesn't clean up native heap

JNI: Accessing Fields in C++

- The process is similar to the **Reflection API** in java
- Let's look at an example of Reflection API to access a field
  - Consider a class "**Person**" that has a field "**name**"

```
public class Person {

      String name ;

      public Person(String name) {
            this.name = name ;
      }
}
```

a **field** (default access, package-level access)

Constructor (public access)

  - We can use reflection API to access any field (even private) of an instance of this class

```
// create an instance
Person person = new Person("Bob") ;

// use reflection API
Class<?> clazz = person.getClass() ;
Field f1 = clazz.getDeclaredField("name") ;
System.out.println(f1);
System.out.println(f1.get(person));
```

Create an instance of the class

Stored in "**metaspace**"

Get a **Class** object that has **meta-data** of the object

Get a **Field** identifier

Invoke **field.get(obj)** on the instance object

Reflection API throws lots of Exceptions!

JNI: Accessing Fields in C++

- In **C++**, we take the following steps for **instance** fields:
  - **Step 1**: get the jclass type of the object
    - env -> **FindClass**("package/class")
  - **Step 2**: get the jfieldID type of the object field
    - env -> **GetFieldID**(jclass, name, signature)
  - **Step 3**: ask JVM to give you the field:
    - Field of primitive type

✧ env -> **Get<Type>Field**(jobject, jfieldID)

- Field of object type

✧ env -> **GetObjectField**(jobject, jfieldID)

- **Step 4**: do proper casting

  for object fields
  - jobject → jstring

```
public class Person {

      String name ;

      public Person(String name) {
            this.name = name ;
      }
}
```

Signature of String class

```
// step 1: jclass of Person
jclass person_class = env -> FindClass("test7/Person") ;
// step 2: field ID of Person
jfieldID fid = env -> GetFieldID(person_class, "name", "Ljava/lang/String;") ;
// step 3: get jobject of the field
jobject name_obj = env -> GetObjectField(Person, fid) ;
// step 4: cast jobject to jstring
jstring name = (jstring) name_obj ;
// print jstring to the console
const char* chararray_name = env -> GetStringUTFChars(name, 0) ;
printf("%s\n", chararray_name) ;
env -> ReleaseStringUTFChars(name, chararray_name) ;
```

JNI: Accessing Static Fields in C++

- In **C++**, we take the following steps for **static** fields:
  - **Step 1**: get the jclass type of the object
    - env -> **FindClass**("package/class")
  - **Step 2**: get the jfieldID type of the object field
    - env -> **GetStaticFieldID**(jclass, name, signature)
  - **Step 3**: ask JVM to give you the field:
    - Field of primitive type

✧ env -> **GetStatic<Type>Field**(jclass, jfieldID)

- Field of object type

✧ env -> **GetStaticObjectField**(jclass, jfieldID)

- **Step 4**: do proper casting

  for object fields
  - jobject → jstring

```
public class Person {

      static String name = "Bob" ;
      static int age = 21 ;
}
```

java Reflection API

```
Field f = Person.class.getDeclaredField("name") ;
System.out.println(f);
Object nameObj = f.get(Person.class) ;
System.out.println(nameObj);
```

java native interface (JNI)

```
jclass person_class = env -> FindClass("test8/Person") ;
jfieldID name_fid = env -> GetStaticFieldID(person_class, "name",
"Ljava/lang/String;") ;
jfieldID age_fid = env -> GetStaticFieldID(person_class, "age", "I") ;
jint age = env -> GetStaticIntField(person_class, age_fid) ;
printf("age = %d \n", age) ;
jobject name_obj = env -> GetStaticObjectField(person_class, name_fid)
;
jstring name = (jstring) name_obj ;
const char* name_array = env -> GetStringUTFChars(name, 0) ;
printf("name = %s\n", name_array) ;
env -> ReleaseStringUTFChars(name, name_array) ;
```

JNI: Setting Object Fields in C++

- In **C++**, we take the following steps for **instance** fields:
  - **Step 1**: get the jclass type of the object
    - env -> **FindClass**("package/class")
  - **Step 2**: get the jfieldID type of the object field
    - env -> **Get[Static]FieldID**(jclass, name, signature)
  - **Step 3**: ask JVM to set the field:
    - Field of primitive type (Int, Double, …)

✧ env -> **Set<Type>Field**(jobject, jfieldID, value)

- Field of object type

✧ env -> **SetObjectField**(jobject, jfieldID, jobj)

- **Step 4**: for static fields

✧ env -> **SetStatic<Type>Field**(jclass, jfieldID)

✧ env -> **SetStaticObjectField**(jclass, jfieldID)

```
public class Person {

      String name ;
      int age ;

      public Person(String name) {
            this.name = name ;
      }
}
```

```
jclass person_class = env -> FindClass("test9/Person") ;
jfieldID name_id = env -> GetFieldID(person_class, "name",
"Ljava/lang/String;") ;
env -> SetObjectField(person, name_id, name) ;
jfieldID age_id = env -> GetFieldID(person_class, "age", "I") ;
env -> SetIntField(person, age_id, 31) ;
```

Signature of String class

jobject (an instance object)

Non-static field

JNI: Caching Field IDs in C++

- In **C++**, we take the following steps for instance & static fields:
  - **Step 1**: get the jclass type of the object
    - env -> **FindClass**("package/class")
  - **Step 2**: get the jfieldID type of the object field
    - env -> **Get[Static]FieldID**(jclass, name, signature)
- We see that jclass and jfieldID do not depend on the object instance
  - jclass & jfieldID → meta information about the class itself
  - We need to only evaluate them once even for different instances of a class
- Approach 1: **Method caching**
  - Use "**static**" keyword in the C++ function
- Approach 2: **Global caching**
  - Define global variables in .cpp file
  - Define a static "**initIDs()**" in java class and put it inside a static initializer

In the entire life of the program, **static** line in **func()** gets executed only **once**.

```
void func(int x) {
     static int y = 2.0*x ;
     cout << y << endl ;
}

int main() {
     double x = 3.0 ;
     func(x) ;
     func(x+1.0) ;
     func(x+2.0) ;
}
```

JNI: Calling Methods in C++

- The process is similar to the **Reflection API** in java
- Let's look at an example of Reflection API to access a method (**call/invoke** a method)
  - Consider a class "**Person**" that has a field "name" with a method "**getName()**"

```
public class Person {
      String name ;
      public Person(String name) {
            this.name = name ;
      }
      public String getName() {
            return name ;
}
```

a field (default access, package-level access)

Constructor (public access)

a **method** (public access)

  - We can use reflection API to access any method (even private) of an instance of this class

```
// create an instance
Person person = new Person("Bob") ;
// use Reflection API
Class<?> clazz = person.getClass() ;
// method: getName()
Method m1 = clazz.getDeclaredMethod("getName") ;
Object obj = m1.invoke(person) ;
String name = (String) obj ;
System.out.println(name);
```

Create an instance of the class

Stored in "**metaspace**"

Get a **Class** object that has **meta-data** of the object

Get a **Method** identifier

Invoke **method.invoke(obj, args)** on the instance object

JNI: Calling Methods in C++

- In **C++**, we take the following steps for **instance** methods:
  - **Step 1**: get the jclass type of the object
    - env -> **FindClass**("package/class")
  - **Step 2**: get the jmethodID type of the object method
    - env -> **GetMethodID**(jclass, name, signature)

Method signature

  - **Step 3**: ask JVM to invoke the method:
    - Method of primitive return type

args

✧ env -> **Call<Type>Method**(jobject, jmethodID, …)

    - Method of object return type

args

✧ env -> **CallObjectMethod**(jobject, jmethodID, …)

  - **Step 4**: do proper casting

    for the returned object
    - jobject → jstring

```
public class Person {
      String name ;
      public Person(String name) {
            this.name = name ;
      }
      public String getName() {
            return name ;
      }
      public void printName() {
            System.out.println(name ) ;
      }
}
```

JNI call to printName()

```
jclass person_class = env -> GetObjectClass(person) ;
jmethodID print_name_id = env -> GetMethodID(person_class,
"printName", "()V") ;
env -> CallVoidMethod(person, print_name_id) ;

jmethodID print_info_id = env -> GetMethodID(person_class,
"printInfo", "(I)Ljava/lang/String;") ;
jobject info_obj = env -> CallObjectMethod(person,
print_info_id, 23) ;
jstring info = (jstring) info_obj ;
const char* info_array = env -> GetStringUTFChars(info, 0) ;
printf("info = %s", info_array) ;
env -> ReleaseStringUTFChars(info, info_array) ;
```

JNI: Calling Static Methods in C++

- In **C++**, we take the following steps for **instance** methods:
  - **Step 1**: get the jclass type of the object
    - env -> **FindClass**("package/class")
  - **Step 2**: get the jmethodID type of the object method
    - env -> **GetStaticMethodID**(jclass, name, signature)

Method signature

  - **Step 3**: ask JVM to invoke the method:
    - Method of primitive return type

args

✧ env -> **CallStatic<Type>Method**(jclass, jmethodID, …)

    - Method of object return type

args

✧ env -> **CallStaticObjectMethod**(jclass, jmethodID, …)

  - **Step 4**: do proper casting

    for the returned object
    - jobject → jstring

```
package test13;

public class Person {

      String name ;
      int age ;

      public static void setInfo(Person person,
String name, int age) {
            person.name = name ;
            person.age = age ;
      }

}
```

JNI call to setInfo()

```
/*
 * Class:     test13_TestCallStaticMethods
 * Method:    callSetInfo
 * Signature: (Ltest13/Person;Ljava/lang/String;I)V
 */
JNIEXPORT void JNICALL
Java_test13_TestCallStaticMethods_callSetInfo
  (JNIEnv *jvm, jclass TestCallStaticMethods, jobject person,
jstring name, jint age) {
      jclass person_class = jvm -> FindClass("test13/Person")
;
      jmethodID set_info_id = jvm ->
GetStaticMethodID(person_class, "setInfo",
"(Ltest13/Person;Ljava/lang/String;I)V") ;
      jvm -> CallStaticVoidMethod(person_class, set_info_id,
person, name, age) ;
}
```

JNI: Calling Interface Methods

- This is similar to calling a method of an object ➔ **Automatically handles Lambda expressions**
  - **Step 1**: get the jclass type of the interface
    - env -> **FindClass**("package/interface")
  - **Step 2**: get the jmethodID type of the interface method
    - env -> **GetMethodID**(jclass, name, signature)
  - **Step 3**: ask JVM to invoke the method:
    - Method of primitive return type

✧ env -> **Call<Type>Method**(jobject, jmethodID, **args**...)

    - Method of object return type

✧ env -> **CallObjectMethod**(jobject, jmethodID, **args**...)

  - **Step 4**: do proper casting

    for the returned object
    - jobject → jstring

Interfaces can have **Static** Methods

Interfaces can have **default** Methods

Interfaces can have **abstract** Methods

```
public interface MathFunction {

      double value(double x) ;

      default double valueScaled(double x) {
            return value(x)*5.0 ;
      }

      static MathFunction functionScaled(MathFunction func) {
            return x -> func.value(x)*5.0 ;
      }
}
```

JNI: Instantiating Objects in C++

- This is similar to calling a method of an object → constructor is a special method: "**<init>**"
  - **Step 1**: get the jclass type of the class
    - env -> **FindClass**("package/class")
  - **Step 2**: get the jmethodID of the constructor
    - env -> **GetMethodID**(jclass, "**<init>**", "(…)**V**")
  - **Step 3**: ask JVM to create new object:
    - Constructor always returns an **object**

✧ env -> **NewObject**(jclass, jmethodID, …)

  - **Step 4**: Usually, there is no need to type cast the jobject

args

Constructor always returns **void**

Constructor is **never** static

```
public class Person {

	String name ;
	int age ;

	public Person(String name, int age) {
		this.name = name ;
		this.age = age ;
	}

	@Override
	public String toString() {
		return "Person [name=" + name + ", age=" + age + "]";
	}
}
```

Format Specifiers in C++

Format Specifier	Type
%c	Character
%d	Signed Integer
%e , %E	Scientific notation of floats
%f	Float values
%g , %G	Similar to %e or %E
%hi	Signed integer (short)
%hu	Unsigned integer (short)
%i	Integer
%l , %ld , %li	Long
%lf	Double
%Lf	Long double
%lu	Unsigned int or long
%lli , %lld	Long long
%llu	Unsigned long long

Format Specifier	Type
%o	Octal representation
%p	Pointer
%s	String
%u	Unsigned int
%x , %X	Hexadecimal representation
%n	Prints nothing
%%	Prints % character

JNI: GNU Scientific Library (GSL)

- Installing GNU scientific library (gsl)
  - www.gnu.org/software/gsl/
- It's better to choose the default installation location
  - **/usr/local/include** → default installation for all of the gsl header files
  - **/usr/local/lib** → default location for the compiled static libraries: **libgsl.a** , **libgslblas.a**
  - Need to point the compiler to the header files ("include" folders)
  - Need to point the linker to the dynamic libraries of GSL

- **How to install gsl to its default location?**
  - Download gsl library
  - Open a terminal at the gsl folder
  - Execute: "./configure && make && make install" (requires sudo in Linux)
- **Example 1**: Using JNI to call **Airy** functions from gsl
  - Airy functions are in the gsl_sf → *special functions*

JNI: GNU Scientific Library (GSL)

Introduction

The GNU Scientific Library (GSL) is a numerical library for C and C++ programmers. It is free software under the GNU General Public License.

The library provides a wide range of mathematical routines such as random number generators, special functions and least-squares fitting. There are over 1000 functions in total with an extensive test suite.

The complete range of subject areas covered by the library includes,

Complex Numbers	Roots of Polynomials
Special Functions	Vectors and Matrices
Permutations	Sorting
BLAS Support	Linear Algebra
Eigensystems	Fast Fourier Transforms
Quadrature	Random Numbers
Quasi-Random Sequences	Random Distributions
Statistics	Histograms
N-Tuples	Monte Carlo Integration
Simulated Annealing	Differential Equations
Interpolation	Numerical Differentiation
Chebyshev Approximation	Series Acceleration
Discrete Hankel Transforms	Root-Finding
Minimization	Least-Squares Fitting
Physical Constants	IEEE Floating-Point
Discrete Wavelet Transforms	Basis splines
Running Statistics	Sparse Matrices and Linear Algebra

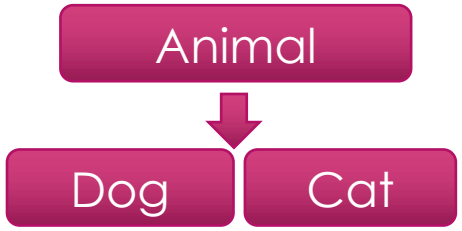
List of procedures and routines that are currently available in GSL

Unlike the licenses of proprietary numerical libraries the license of GSL does not restrict scientific cooperation. It allows you to share your programs freely with others.

JNI: Calling Non-Virtual Methods

- By default, all methods in Java are **virtual** methods
  - This means any method can potentially be **overridden** by a subclass
  - **jvm -> Call<Static><type>Method()** → **type**: a primitive or Object (**return** type)
- Calling/Accessing virtual methods has some runtime overhead
  - JVM has to traverse the inheritance tree for the particular method
- If we know a method has not been overridden
  - We can use **jvm -> CallNonvirtual<type>Method(**jobject, jclass, jmethodID, …**)**
  - This hints the JMV that it does not need to traverse the inheritance tree
  - "jclass" indicates to JVM which class is to be used for the particular method
  - Another use case: we want to always invoke a method in the **superclass**

Inheritance hierarchy




```
static jclass animal_class = jvm -> FindClass("test17/Animal") ;
static jclass dog_class = jvm -> FindClass("test17/Dog") ;
static jclass cat_class = jvm -> FindClass("test17/Cat") ;
static jmethodID animal_speak_id = jvm -> GetMethodID(animal_class, "speak", "()V") ;
static jmethodID dog_speak_id = jvm -> GetMethodID(dog_class, "speak", "()V") ;
static jmethodID cat_speak_id = jvm -> GetMethodID(cat_class, "speak", "()V") ;
jvm -> CallVoidMethod(animal, animal_speak_id) ;
jvm -> CallNonvirtualVoidMethod(animal, animal_class, animal_speak_id) ;
jvm -> CallNonvirtualVoidMethod(animal, animal_class, dog_speak_id) ;
```

This is virtual call

This is non-virtual call

JNI: Working with Generics

- **Generic Methods**
  - We **cannot** use primitives with Generics
  - Parameterized argument type
    - <T>, bounded: <T extends Number>
  - Parameterized return type
  - Wild cards
  - in C++, we lose all of type information → **jobject**
- **Generic Classes** (Template classes)
  - We **cannot** use primitives with Generic classes
  - In java, the type is known to the compiler
  - In C++, the type is general **jobject**
  - "jobject" in C++ is just a pointer
    - Points to the memory address of the object

Parameterized type

```
public static native <T> void printInfoNative(T arg) ;

public static <T> void printInfo(T arg) {
      System.out.println(arg);
}
```

Parameterized class

```
public class Group<T> {

      T[] members ;

      public Group(T[] people) {
            this.members = people ;
      }

      public void printMembers() {
            for (T member : members)
                  System.out.println(member);
      }

}
```

JNI: Working with Generics

- **Generic Methods**
  - We **cannot** use primitives with Generics
  - Parameterized argument type
    - Unbounded: <T>
    - bounded: <T extends Number>
  - Parameterized return type
  - Wild cards
  - in C++, we lose all of type information
    - The type becomes **jobject**
- Use **jvm -> IsInstanceOf(**jobject, jclass**)**
  - Test for the actual type of the jobject
  - This is similar to **instanceof** operator

Parameterized type

```
public static native <T> void printInfoNative(T arg) ;

public static <T> void printInfo(T arg) {
      System.out.println(arg);
}
```

Bounded parameterized type

```
public static native <T extends Number> void printInfoNative(T arg) ;

public static <T extends Number> void printInfo(T arg) {
      System.out.println(arg);
}
```

Testing for the actual type of the jobject

```
// do some tests
jclass Double_class = jvm -> FindClass("java/lang/Double") ;
jclass Integer_class = jvm -> FindClass("java/lang/Integer") ;
jboolean is_int = jvm -> IsInstanceOf(number, Integer_class) ;
if(is_int)
     printf("the type is Integer\n") ;
else
     printf("the type is Double\n") ;
```

JNI: Java Reflection API in C++

- We can pass **Method** and **Field** modifiers from Java Reflection API to C++
- JNI has specific methods to work with these objects
  - **FromReflectedMethod()**
  - **FromReflectedField()**
  - **ToReflectedMethod()**
  - **ToReflectedField()**
  - **GetSuperclass()**
  - **IsAssignableFrom()**

Java Reflection

```
// create an instance
Person person = new Person("Bob") ;

// use reflection API
Class<?> clazz = person.getClass() ;
Field f1 = clazz.getDeclaredField("name") ;
System.out.println(f1);
System.out.println(f1.get(person));
```

JNI Reflection related functions

```
jmethodID FromReflectedMethod(jobject method) {
    return functions->FromReflectedMethod(this,method);
}

jfieldID FromReflectedField(jobject field) {
    return functions->FromReflectedField(this,field);
}

jobject ToReflectedMethod(jclass cls, jmethodID methodID, jboolean isStatic) {
    return functions->ToReflectedMethod(this, cls, methodID, isStatic);
}

jobject ToReflectedField(jclass cls, jfieldID fieldID, jboolean isStatic) {
    return functions->ToReflectedField(this, cls, fieldID, isStatic);
}

jclass GetSuperclass(jclass sub) {
    return functions->GetSuperclass(this, sub);
}

jboolean IsAssignableFrom(jclass sub, jclass sup) {
    return functions->IsAssignableFrom(this, sub, sup);
}
```

JNI: Working with Java Constants

- Compile-time Constants
  - public static final **<type> <name>** = **<value>**
- JNI (Javah) translates these to C++ constants
  - Preprocessor definitions (directives)
    - **#undef <name>**
    - **#define <name> = <value>**
  - Adds them to the header file
- **Signature**
  - "**package name**"_"**class name**"_**<name>**
- This is useful when dealing with constants in Java
- ONLY works for **java primitive types**
  - int, double, float, long, short, boolean, byte
- boolean is translated into numeric value **0L**

```
public class TestConstants {

     public static final int VALUE = 1 ;
     public static final double PRICE = 2.5 ;
     public static final boolean IS_READY = false ;
     public static final String NAME = "Bob" ;

     public static native void printConstants() ;
}
```

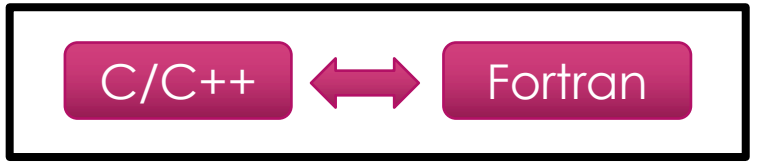
Generated header file

```
#undef test21_TestConstants_VALUE
#define test21_TestConstants_VALUE 1L
#undef test21_TestConstants_PRICE
#define test21_TestConstants_PRICE 2.5
#undef test21_TestConstants_IS_READY
#define test21_TestConstants_IS_READY 0L
/*
 * Class:     test21_TestConstants
 * Method:    printConstants
 * Signature: ()V
 */
JNIEXPORT void JNICALL Java_test21_TestConstants_printConstants
  (JNIEnv *, jclass);
```

JNI: Using Fortran Language

- **C++** and **Fortran** have nice interoperability
- Need to prototype Fortran functions with pass-by-reference or pass-by-pointer in C++
- Install "Photran" Eclipse plugin
- Install gnu fortran "gfortran" compiler
- **Step 1**: Create C++ project
- **Step 2**: add fortran source files
  - *.f , *.f90, …
- **Step 3**: Add "**gfortran**" to the tool chain
  - Fortran files get compiled to **.o** file
- **Step 4**: **extern "C" <function>_**
- **Step 5**: g++ linker takes care of linking
  - Linker doesn't care if the binary comes from C++ or Fortran

Toolchain: g++ & gfortran



g++ --version

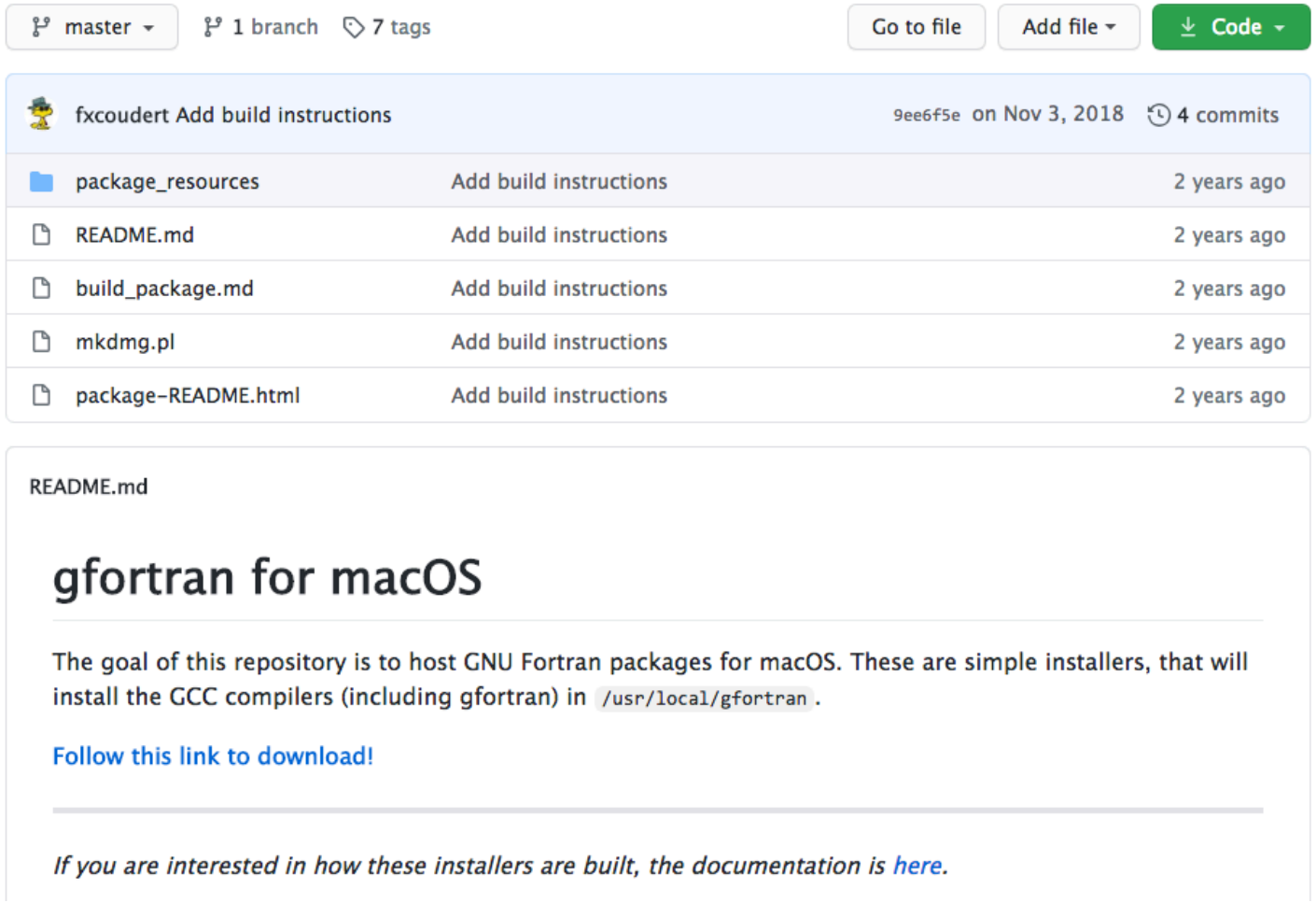
```
Configured with: --
prefix=/Applications/Xcode.app/Contents/Developer/usr --with-gxx-
include-dir=/usr/include/c++/4.2.1
Apple LLVM version 6.0 (clang-600.0.56) (based on LLVM 3.5svn)
Target: x86_64-apple-darwin13.4.0
Thread model: posix
```

gfortran --version

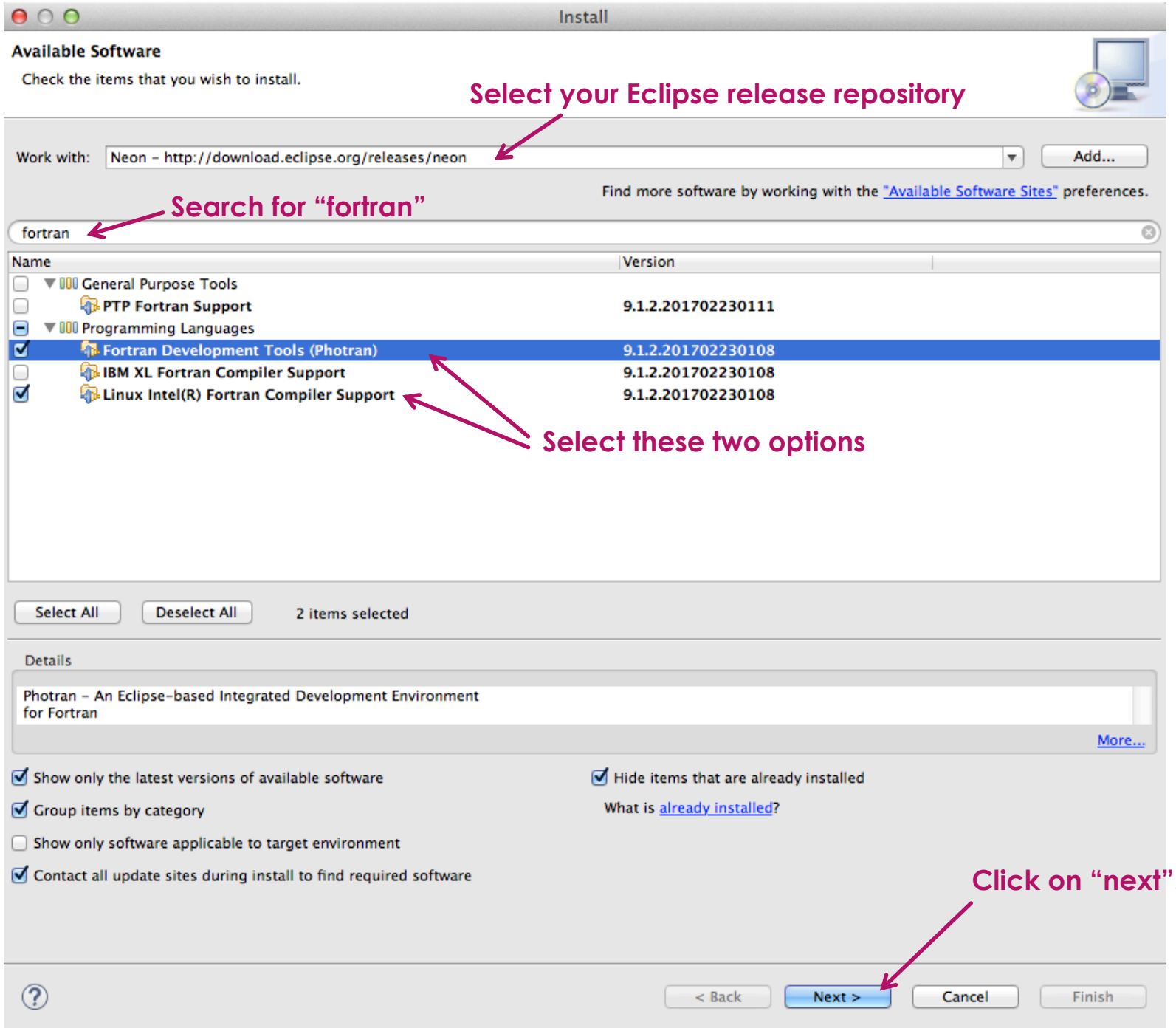
```
GNU Fortran (GCC) 4.9.0
Copyright (C) 2014 Free Software Foundation, Inc.

GNU Fortran comes with NO WARRANTY, to the extent permitted by law.
You may redistribute copies of GNU Fortran
under the terms of the GNU General Public License.
For more information about these matters, see the file named COPYING
```

JNI: Using Fortran Language

- Download and install gfortran from this github repository (macOS)
  - https://github.com/fxcoudert/gfortran-for-macOS



- Download and install **Fortran** plugin for Eclipse IDE



JNI: Working with Inner Classes

- **Inner classes** can also have **native** methods
- Nested inner class → can access all of the fields
  - Associated with the state of the object. Instantiation: **Object.InnerClass** c = **obj**.new InnerClass()

Type of inner class

Instance of a class

- Static nested inner class → cannot access any fields
  - Associated with the class. Can be instantiated independent of any instance of the class.
- **Actual** name of inner class: **$**inner class name
  - Object.InnerClass → **Object$InnerClass** (this is classpath description)
  - Compiler creates separate **.class** files for each inner class
- Each inner class gets its own header file when "javah –jni" or "javac –h" is run
- Expected signature

Java_"package name"_"outer class name"$"inner class name"_"method name"

- Actual signature

Java_"package name"_"outer class name"_00024"inner class name"_"method name"

JNI: Working with Inner Classes

- Overloading signature for inner classes
  - **$ sign** is **not** used in the parameter signature

- In the method name (**$** → **_00024**):
  - **"outer class name"$"inner class name"**
  - **"outer class name"_00024"inner class name"**

- In the parameter signature (**$** → **/** → **_00024**):
  - **"outer class name"$"inner class name"**
  - **"outer class name"/"inner class name"**
    - **"outer class name"_00024"inner class name"**

```
public class Person {

      public String name ;
      public int age ;

      public Person(String name, int age) {
            this.name = name ;
            this.age = age ;
      }
      // nested inner class
      public class Book {
            String title ;

            public Book(String title) {
                  this.title = title ;
            }

            public native void printTitle() ;

      }
      // static inner class
      public static class Bag {
            int size ;

            public Bag(int size) {
                  this.size = size ;
            }

            public native void printSize() ;
            public static native void printSize(Bag bag) ;
      }
}
```

JNI: Static Inner Classes in C++

- Similar to regular classes, we should be able to instantiate inner classes in C++
- Instantiating **static** inner class
  - **Person$Bag** → No need for an instance of "person"
  - This is like an independent class

Static nested inner class

```
// static inner class
public static class Bag {
      int size ;

      public Bag(int size) {
            this.size = size ;
      }

      public native void printSize() ;
      public static native void printSize(Bag bag) ;

      @Override
      public String toString() {
            return "Bag size = " + size ;
      }
}
```

```
/*
 * Class:     test22_TestInnerClass
 * Method:    createBag
 * Signature: ()Ltest22/Person/Bag;
 */
JNIEXPORT jobject JNICALL Java_test22_TestInnerClass_createBag
  (JNIEnv *jvm, jclass TestInnerClass) {
      jclass Bag_class = jvm -> FindClass("test22/Person$Bag") ;
      jmethodID constructor_id = jvm -> GetMethodID(Bag_class, "<init>", "(I)V")
;
      jobject bag_obj = jvm -> NewObject(Bag_class, constructor_id, 32) ;
      return bag_obj ;
}
```

Instantiating Person.Bag object without any reference to an instance of "person"

JNI: Nested Inner Classes in C++

- Similar to regular classes, we should be able to instantiate inner classes in C++
- Instantiating **nested** inner class
  - **Person$Book** → need an instance of the "person" object
  - In the Bytecode the compiler adds a reference to the instance "person" object to constructor

nested inner class

```
// inner class
public class Book {
      String title ;

      public Book(String title) {
            this.title = name+"'s book = " + title ;
      }

      public native void printTitle() ;

      @Override
      public String toString() {
            return title ;
      }

}
```

Bytecode signature

```
public <init>(Ltest22/Person;Ljava/lang/String;)V
```

Can access instance fields of Person

```
/*
 * Class:     test22_TestInnerClass
 * Method:    createBook
 * Signature: (Ltest22/Person;)Ltest22/Person/Book;
 */
JNIEXPORT jobject JNICALL Java_test22_TestInnerClass_createBook
  (JNIEnv *jvm, jclass TestInnerClass, jobject person) {
      jclass Book_class = jvm -> FindClass("test22/Person$Book") ;
      jmethodID constructor_id = jvm -> GetMethodID(Book_class, "<init>",
"(Ltest22/Person;Ljava/lang/String;)V") ;
      jstring book_name = jvm -> NewStringUTF("War and Peace in C++") ;
      jobject book_obj = jvm -> NewObject(Book_class, constructor_id, person,
book_name) ;
      return book_obj ;
}
```

Instantiating Person.Book object requires a reference to an instance of "person"

JNI: Method-Local Inner Classes

- The scope of definition is inside the scope of a **method**
  - We **cannot** define it as static class
- Classpath description of the method-local inner class
  - **OuterClass$1MethodClass** → **$1** is replaced by **_000241** when generating header files
  - No reference to the outer class in the constructor of the nested method-local inner class
- Compiler generates a separate **class** file for each inner class
  - `System.out.println(Student.class);`
- Method-local inner classes can also have **native** methods
- We can easily run "javah –jni" on the classpath description of the inner class
  - This generates a header file for the inner class
- Note that if we run the "javah –jni" tool on the outer class we do not get a header file for the method-local inner class

JNI: Method-Local Inner Classes

- Illegal access modifiers for method-local inner classes
  - public, private, static
- Legal access modifiers: **abstract**, **final**

Method-local inner class → `class test22.TestMethodLocal$1Student`

```
public static void test1() {

     class Student {
          String name ;
          int age ;
          public Student(String name, int age) {
               this.name = name ;
               this.age = age ;
          }
          @Override
          public String toString() {
               return "Student [name=" + name + ", age=" + age + "]";
          }
          public native void printInfo() ;
     }

     Student student1 = new Student("Bob", 21) ;
     System.out.println(student1);
     System.out.println(Student.class);

}
```

Native method

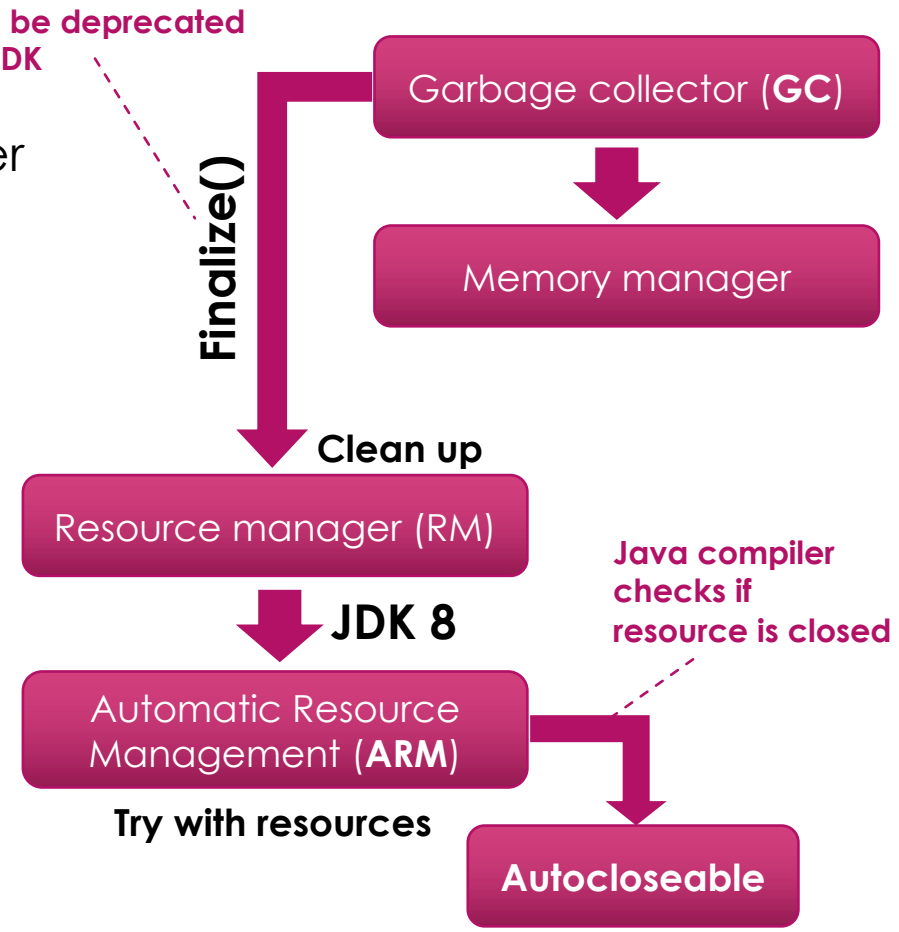
javah –jni **test22.TestMethodLocal$1Student**

```
/*
 * Class:     Student
 * Method:    printInfo
 * Signature: ()V
 */
JNIEXPORT void JNICALL Java_test22_TestMethodLocal_000241Student_printInfo
  (JNIEnv *, jobject);
```

$ → _00024
$1 → _000241

JNI: Managing Memory Leaks

- Allocating on the heap in C++ using **new** keyword requires manual **delete** ------- Stack ~ 2MB
- Sometimes we need to keep a C++ resource (native resource) until the java object is reclaimed by garbage collector
- **General approach:**
  - Private static native method "**initIDs()**" inside static initializer
  - Initialize and prepare the resource when object is created
  - Private native method "**close0()**" or "**cleanup0()**"
  - Public method "**close()**" for manual close and cleanup
  - Override "**finalize()**" method from Object class
    - Put "**close0()**" inside the finalize method
- **Good practice:**
  - Mark the class with "**Autocloseable**" interface
  - This also allows for **try with resources**



JNI: Managing Memory Leaks

- **Example**: writing your own FileReader
- Automatic resource management
  - Implement Autocloseable
- Need to open a file stream in C++
  - Native method: **open0()**
- Need to close a file stream in C++
  - Native method: **close0()**
- As a safe measure, we override finalize()
  - Call **close()** or **close0()** in **finalize()**
- Java compiler issues a warning if
  - **close()** is not explicitly called
- It is a bad practice to rely on **finalize()**
  - Garbage collector may never run

```
public class FileReader implements AutoCloseable {

      static {
            System.loadLibrary("native_c");
            initIDs() ;
      }

      String file ;

      private static native void initIDs() ;
      private native void close0() ;
      private native void open0() ;

      public FileReader(String file) {
            this.file = file ;
            open0() ;
      }

      public native String next() ;
      public native boolean eof() ;

      @Override
      public void close() throws Exception {
            close0() ;
            System.out.println("closing the file reader in 'close()'");
      }
      @Override
      protected void finalize() throws Throwable {
            close0() ;
            System.out.println("closing the file reader in 'finalize()'");
      }
}
```

Resource management interface

Open & close file stream

Read next word

End of file

Resource management through GC

JNI: Working with Standard I/O

- **Working with Standard IO**
- **C++** system standard IO:
  - #include <iostream>
  - **std::cin** → console in
  - **std::cout** → console out

  } Standard I/O in java
- **In java**
  - System.***in***
  - System.***out***

  } Standard I/O in java
- Let's write our own standard I/O
  - Create a class **ConsoleIn** and **ConsoleOut**
  - Use singleton design pattern → only ONE instance of the standard I/O per JVM
  - No need to manually open standard I/O
  - No need to manually close standard I/O

  } We don't need to close **std::cin** or **std::cout**

```
public class ConsoleIn {

      static {
            System.loadLibrary("native_c");
            initIDs();
      }

      private static native void initIDs() ;

      public ConsoleIn() {

      }

      public native String next() ;
      public native int nextInt() ;
      public native double nextFloat() ;
      public native double nextDouble() ;

}
```

Read next word
Read next int
Read next float
Read next double

We need to provide methods for all primitive types

JNI: Using "auto" keyword in C++

- All of the JNI types in C++ start with letter "j" → all types are defined in "jni.h"
- **C++ 11** introduced the keyword "**auto**" → type inference
  - This is used for type inference
- We can take advantage of "**auto**" in JNI function calls
- Don't forget to make use "static" in functions

Java class

```
public class Person {

	String name ;
	int age ;

	public Person(String name, int age) {
		this.name = name ;
		this.age = age ;
	}

	@Override
	public String toString() {
		return "Person [name=" + name + ",
age=" + age + "]";
	}

}
```

Java native method

```
public static native Person createPerson(String name, int age) ;
```

C++ implementation

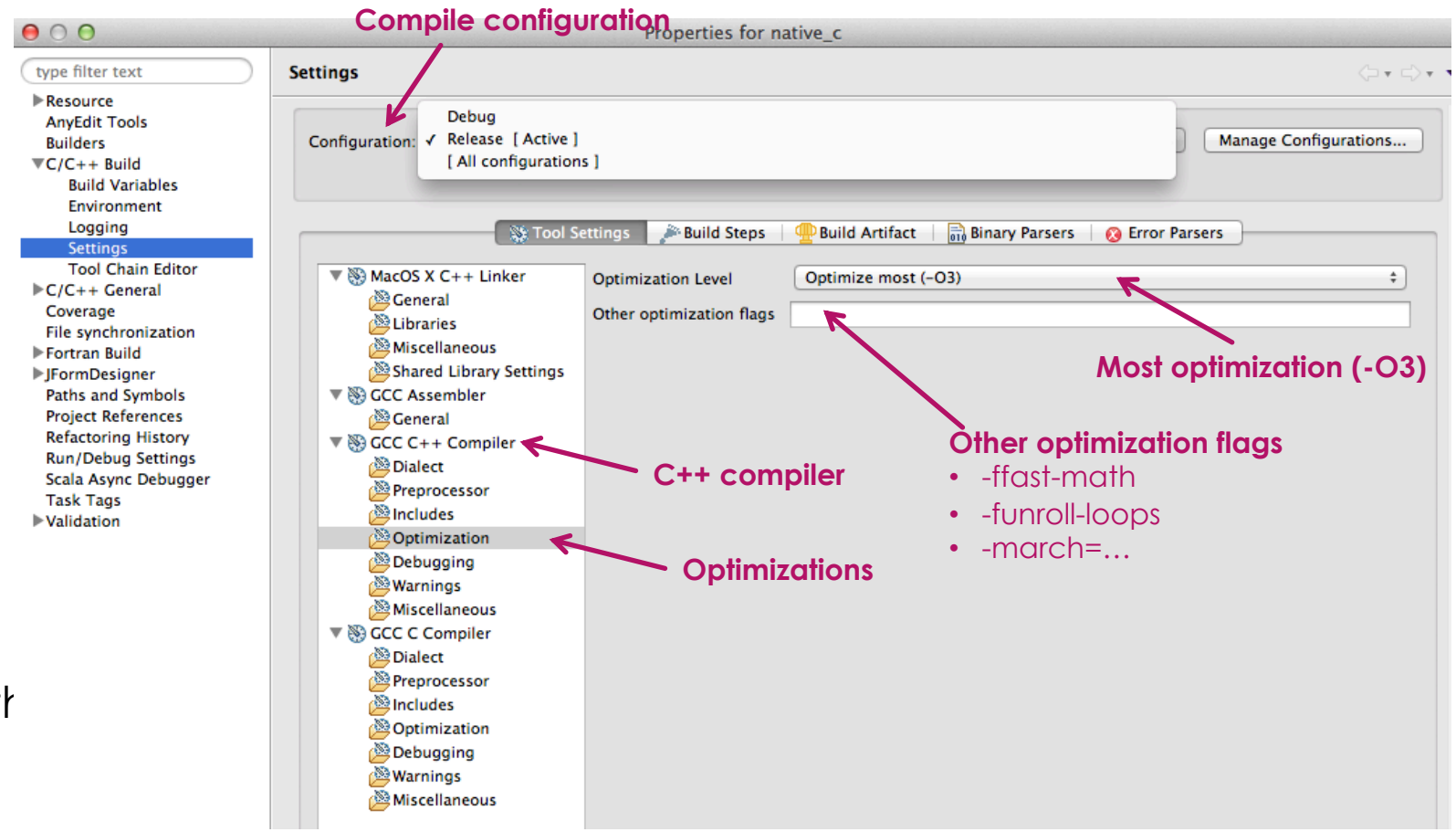
```
/*
 * Class:     test25_TestAuto
 * Method:    createPerson
 * Signature: (Ljava/lang/String;I)Ltest25/Person;
 */
JNIEXPORT jobject JNICALL Java_test25_TestAuto_createPerson
  (JNIEnv *jvm, jclass TestAuto_class, jstring j_name, jint j_age) {
	static auto Person_class = jvm -> FindClass("test25/Person") ;
	static auto constructor_id = jvm -> GetMethodID(Person_class, "<init>",
"(Ljava/lang/String;I)V") ;
	auto person_obj = jvm -> NewObject(Person_class, constructor_id, j_name,
j_age) ;
	return person_obj ;
}
```

JNI: Windows Platform

- **gcc** comes by default with Linux
- On Mac os: install **Xcode** or **command line tools**
- On Windows:
  - MinGW → 32 bit (default)
  - **MinGW-w64 → 64 bit**
  - Make sure you install the right one
- In Eclipse IDE
  - Make sure **MINGW_HOME** environment variable is set to the root directory to the MinGW folder
  - Root of MinGW → where the "**bin**" folder is
- Dynamic libraries:
  - Linux → .so, Mac os → .dylib, windows → .dll
  - **Linux** and **Mac os** → You **MUST** drop "lib" from the beginning of the name
  - **Windows** → you must **NOT** drop "lib" from the beginning of the name

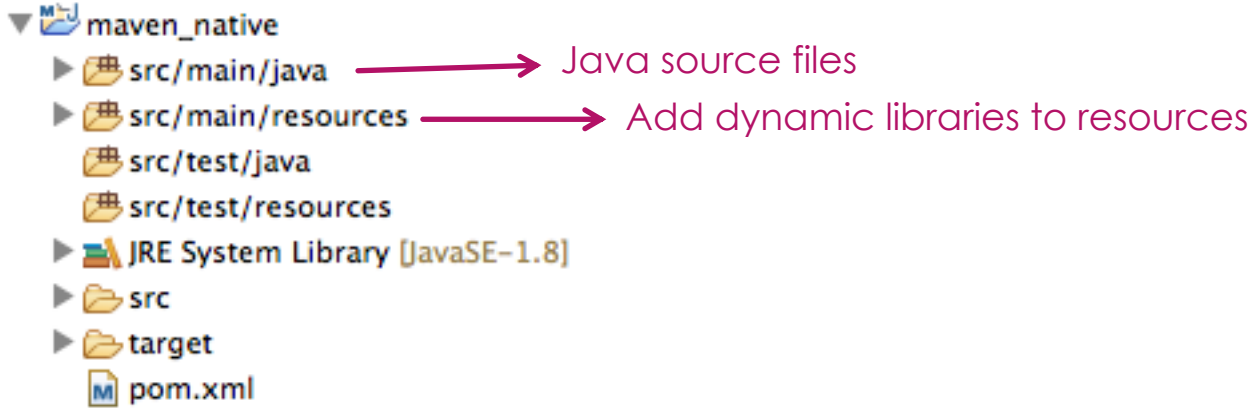
JNI: Debug vs. Release Mode

- C++ projects can be compiled in two configurations
  - **Debug**
  - **Release**
- **Debug mode**
  - Verbose assembly code
  - Typically no optimization
  - Slower than release mode
- **Release**
  - Very concise assembly code
  - Typically with "**-O3**" flag
  - Lots of static optimization by th compiler
  - Best speed and lowest memory

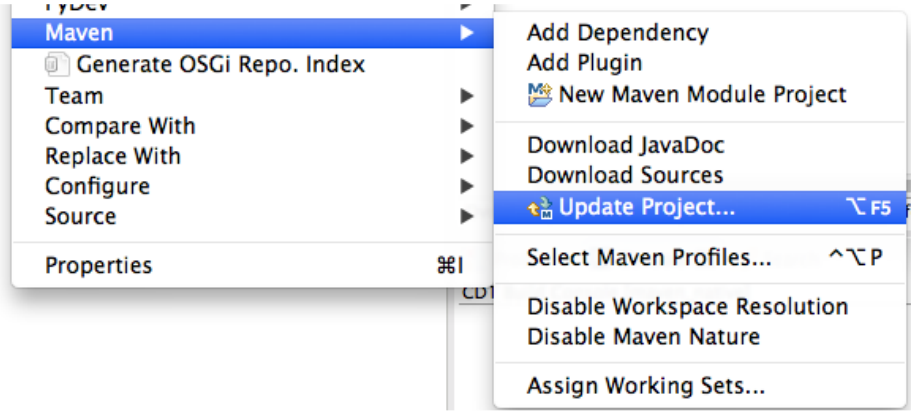


JNI: Working with Maven

- Maven is a build tool for java
- Goals and plugins
  - Compile → compiles the java files
  - Test → compiles java test files (junit)
  - Package → packages into jar file
- Convention over configuration
  - Standard folder structure
  - Source folders
    - src/main/java
    - src/main/resources
    - src/test/java
    - src/test/resources
- **POM**: project object model (XML file)



Any file in the resources folder will be placed at the **root** of the **jar** file



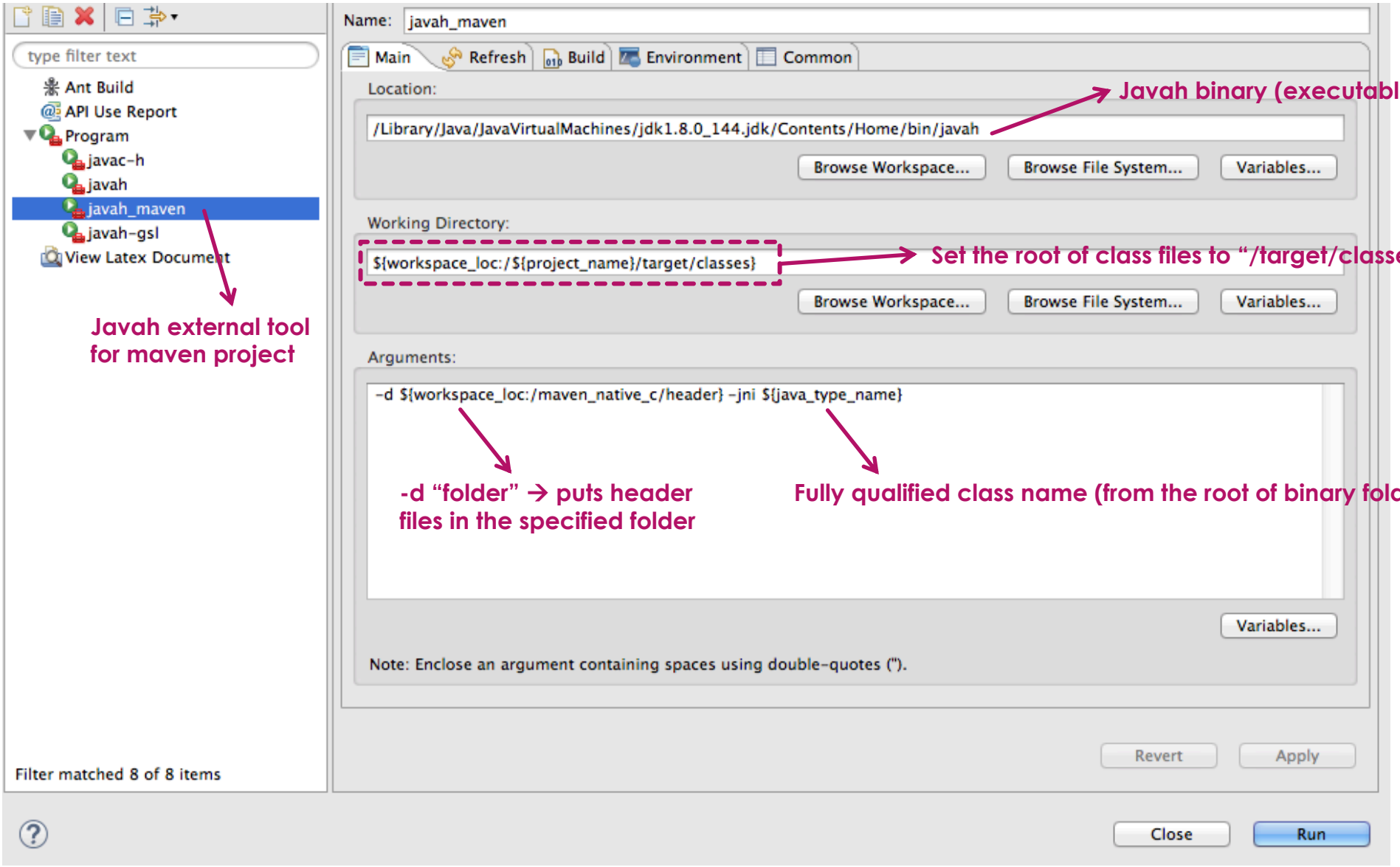
How to set JRE

- maven.compiler.source = 1.8
- maven.compiler.target = 1.8
- Use properties & update maven project

JNI: Working with Maven

- Our goal is to correctly package dynamic library into the maven build
- **Step 1**: create java files (some have native methods)
- **Step 2**: build the maven project with "maven build" for the compile phase
  - This creates "**target**" folder and puts all the class files inside "**classes**" folder
  - The binary (class) files are in "**project_name/target/classes**" path → **Working directory for javah**
  - This is different from regular Eclipse IDE java projects (*default*: project_name/bin)
- **Step 3**: setup the javah external tool to point to the "classes" folder
  - This is the main difference from javah when using a regular java project in Eclipse IDE
- **Step 4**: download "**Native-Utils**" java file from github and add it to your project
  - https://github.com/adamheinrich/native-utils
- **Step 5**: add dynamic library to the root of "src/main/resources"
- **Step 6**: use Native-Utils to load the dynamic library from the root of jar file
  - **NativeUtils.loadFromJar**("/lib****.dylib") → **You must specify full name of dynamic library**

JNI: Working with Maven



JNI: Distributing C++ Project

- **gcc** and **g++** support "**make**" command with specific **build targets**
- Why distribute source file of the C++ project?
  - Others can use it to build binaries for their own platform that is not provided by you
- Eclipse CDT has two modes of build operation:
  - Internal builder → does not create "make" files
  - GNU builder → creates "make" targets
- All platforms can support "make" when installing gcc
  - On windows, use "**.bat**" file in the root directory of "MinGW" or "**MinGW-w64**"
- Make sure you set your project builder to "GNU builder" so that it can create "make" targets
- Usage: **make "target name"** → "makefile"
- Common make targets:
  - Nothing: make → will compile all the .cpp files and create .o files → just compilation
  - "all": make all → will execute all of the build targets in the "makefile"

JNI: Platform Independent Jar

- There are two main options:
  - **Option 1**: Make **separate** jar files
    - Bundle each jar with appropriate dynamic library
  - **Option 2**: Make **only one** jar file
    - Bundle the jar with all of the dynamic libraries (for all the supported platforms)
    - Check the platform (CPU + OS) at runtime
    - Load the appropriate dynamic library at runtime
- How to check the platform at runtime?
  - **System.getProperties()**
  - System.getProperty("os.name")
  - System.getProperty("os.arch")
  - System.getProperty("os.version")

Properties is a HashTable

```
public class TestSystemProperties {

      public static void main(String[] args) {
            Properties properties = System.getProperties() ;
            properties.forEach((k,v) -> System.out.println(k + " : " + v));
      }

}
```

Internal iterator

JNI: Platform Independent Jar

- How to check the platform at runtime?
  - **System.getProperties()**
  - System.getProperty("os.name")
  - System.getProperty("os.arch")

From JDK 8 (Oracle JDK) → "System" class

```
/**
 * System properties. The following properties are guaranteed to be defined:
 * <dl>
 * <dt>java.version         <dd>Java version number
 * <dt>java.vendor          <dd>Java vendor specific string
 * <dt>java.vendor.url      <dd>Java vendor URL
 * <dt>java.home            <dd>Java installation directory
 * <dt>java.class.version   <dd>Java class version number
 * <dt>java.class.path      <dd>Java classpath
 * <dt>os.name              <dd>Operating System Name
 * <dt>os.arch              <dd>Operating System Architecture
 * <dt>os.version           <dd>Operating System Version
 * <dt>file.separator       <dd>File separator ("/" on Unix)
 * <dt>path.separator       <dd>Path separator (":" on Unix)
 * <dt>line.separator       <dd>Line separator ("\n" on Unix)
 * <dt>user.name            <dd>User account name
 * <dt>user.home            <dd>User home directory
 * <dt>user.dir             <dd>User's current working directory
 * </dl>
 */
```

PATH environment variable

JNI: Anonymous Classes

- The scope of definition is inside the scope of a normal variable
  - We define them when instantiating a class or even an interface
- Classpath description of the anonymous class
  - **OuterClass$X** → **$** is replaced by **_00024**, X is 1, 2, 3, … → **$X** is replaced by **_00024X**
  - **OuterClass$1** → **$1** is replaced by **_000241** when generating header files
- Compiler generates a **separate class** file for each anonymous class
  - `System.out.println(Person.class);`
- Anonymous classes can also have **native** methods
  - Similar to Method-local inner classes
- We can easily run "javah –jni" on the classpath description of the anonymous class
  - This generates a header file for the inner class
- Note that if we run the "javah –jni" tool on the outer class we do not get a header file for the anonymous class

JNI: Anonymous Classes

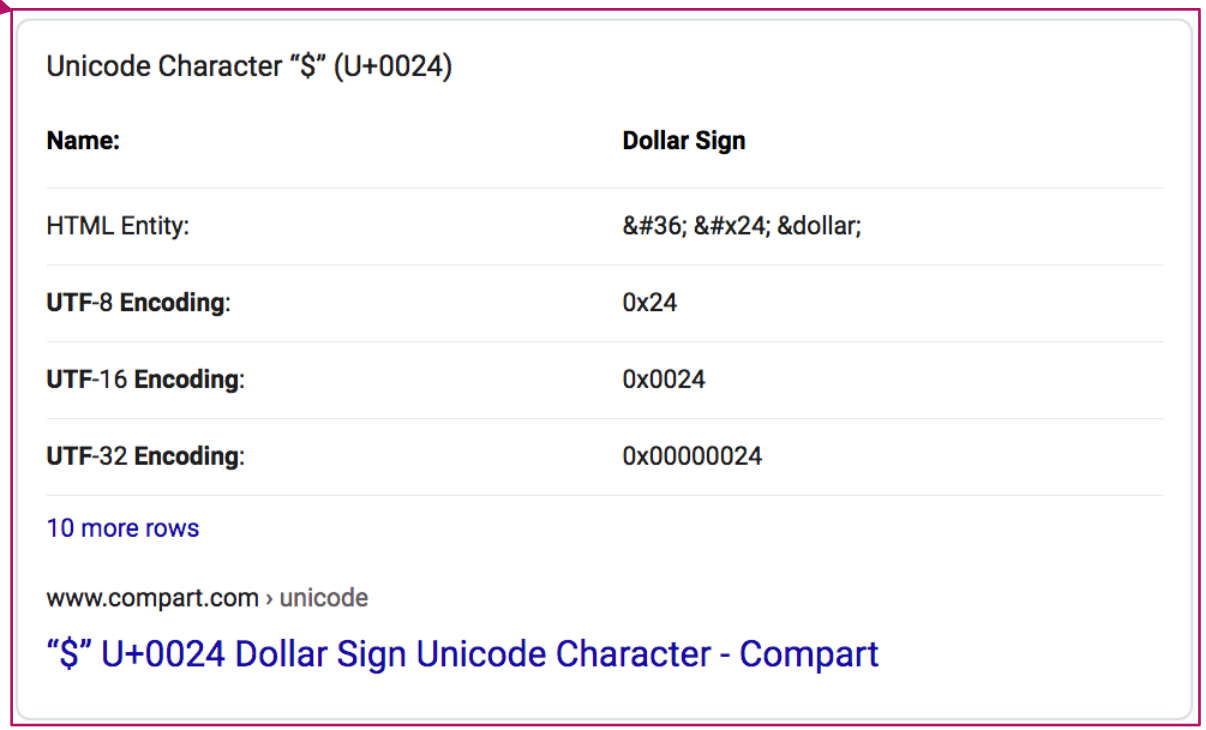
- What happens when manually running "javah –jni" on an **anonymous class**
  - **package.Outerclass$X** → X is a numeric character
  - Javah strips away the first part before $ sign
  - The header file will have the name **X.h**
  - Header files can't start with a numeric character
  - Javah removes X from the name of the header file
  - The header file will end up having the name "**.h**" → This is like a hidden file in Mac os
  - Simply rename this file to something more useful
- What happens when manually running "javah –jni" on a **method-local inner class** (MLIclass)
  - **package.Outerclass$1MLIclass** → this is the generated class file
  - Javah strips away the first part before $ sign
  - The header file will have the name **1MLIclass.h** → file cannot start with a numeric character
  - Javah removes 1 from the name of the header file → "**MLIclass.h**" is generated

Note: in general the name of header file does not really matter.

Only the name of the C/C++ function matters: **Java_package_class_method**

JNI: Anonymous Classes

- **Why $ sign is replaced by 00024?**
- Java uses UTF-16 encoding for strings
- Let's see what the Hexadecimal representation of the $ sign is:
  - We can just google it:
- UTF-8 Hex representation of $ sign:
  - 0x24 → remove x : 024 ➔ 1 byte
- **UTF-16** Hex representation of $ sign:
  - **0x0024 → remove x : 00024** ➔ 2 bytes
- UTF-32 Hex representation of $ sign:
  - 0x00000024 → remove x : 0…24 ➔ 4 bytes
- HTML representation of $ sign:
  - &#x24



Unicode Character "$" (U+0024)

Name:	**Dollar Sign**
HTML Entity:	$ $ $
UTF-8 Encoding:	0x24
UTF-16 Encoding:	0x0024
UTF-32 Encoding:	0x00000024

10 more rows

www.compart.com › unicode

"$" U+0024 Dollar Sign Unicode Character - Compart

JNI: Regular vs. Critical Access

- We need to copy a primitive array over into a C-array
  - jvm → GetDoubleArrayRegion(jarray, start, end, carray)
  - jvm → GetDoubleArrayElements(jarray, isCopy) : returns **jdouble*** → **Primitive pointer**
- This will have significant overhead if the array is very large (len > 10_000_000)
- JVM allows us to directly manipulate java heap for primitive arrays using "**critical**" access
  - jvm → GetPrimitiveArrayCritical(jarray, isCopy) : returns **void*** → **Void pointer** → **Requires manual type casting**
- We need to make some promises to the JVM
  - JVM will temporarily disable garbage collection on the array
  - We should not call any arbitrary JNI functions after that
- When JNI function returns a **RAW pointer** to something (jdouble*, const char*, …)
  - We need to call "**Release…()**" function
  - jvm → RelaseDoubleArrayElements(jarray, carray, mode)
  - jvm → RelasePrimitiveArrayCritical(jarray, carray, mode)

→ **mode** = { **0**: copy back and delete; **JNI_COMMMIT**; **JNI_ABORT** }

JNI: Critical Access of Strings

- Similar copying issue exists for Java Strings
  - jvm → GetStringUTFChars(jstring, isCopy) : returns **const char*** → **Primitive pointer (C-style string)**
- This will have significant overhead if we are processing many java strings in C++
- JVM allows us to directly manipulate java heap for Strings using "**critical**" access
  - jvm → GetStringCritical(jstring, isCopy) : returns **const jchar*** → **Primitive pointer (NOT a C-style string)**
- We need to make some promises to the JVM
  - JVM will temporarily disable garbage collection on the Strings
  - We should not call any arbitrary JNI functions after that
- When JNI function returns a **RAW pointer** to something (jdouble*, const char*, …)
  - We need to call "**Release…()**" function
  - jvm → ReleaseStringCritical(jstring, const jchar*)
- **Main difficulty**
  - How to convert const jchar* [UTF-16, two bytes] to const char* [UTF-8/ASCII, one byte]

JNI: Creating an "Unsafe" Class

- What is an "**Unsafe**" class?
  - Allows us accessing and manipulating **off-heap** memory directly
- What are the methods of an "Unsafe" class?
  - allocateMemory() → Bytes of memory to allocate → Returns a memory address (64 bit)
  - freeMemory()
  - putByte(), putShort(), putInt(), putLong()
  - putFloat(), putDouble()
  - getByte(), getShort(), getInt(), getLong()
  - getFloat(), getDouble()
- Helps us directly work with raw pointers
- Having a contiguous block of memory increases locality and potentially reduces cache misses
- We cannot use it to allocate something on the stack
  - When JNI call returns, all the stuff on C++ stack is destroyed

char* array = new char[bytes]

Working with "long" integers

JNI: "sun.misc.Unsafe"

- What is the "**sum.misc.Unsafe**" class?
  - It's an unsafe class built into JDK
- Provides low-level memory access
- We need to use Reflections to get access to it
- It's a Singleton
- The instance is called "**theUnsafe**"

```
Class<?> clazz = Class.forName("sun.misc.Unsafe") ;
Field field = clazz.getDeclaredField("theUnsafe") ;
field.setAccessible(true) ;
Unsafe unsafe = (Unsafe) field.get(null) ;
```

Working with Objects

- S getUnsafe() : Unsafe
- N getInt(Object, long) : int
- N putInt(Object, long, int) : void
- N getObject(Object, long) : Object
- N putObject(Object, long, Object) : void
- N getBoolean(Object, long) : boolean
- N putBoolean(Object, long, boolean) : void
- N getByte(Object, long) : byte
- N putByte(Object, long, byte) : void
- N getShort(Object, long) : short
- N putShort(Object, long, short) : void
- N getChar(Object, long) : char
- N putChar(Object, long, char) : void
- N getLong(Object, long) : long
- N putLong(Object, long, long) : void
- N getFloat(Object, long) : float
- N putFloat(Object, long, float) : void
- N getDouble(Object, long) : double
- N putDouble(Object, long, double) : void

Working with primitives

- N getByte(long) : byte
- N putByte(long, byte) : void
- N getShort(long) : short
- N putShort(long, short) : void
- N getChar(long) : char
- N putChar(long, char) : void
- N getInt(long) : int
- N putInt(long, int) : void
- N getLong(long) : long
- N putLong(long, long) : void
- N getFloat(long) : float
- N putFloat(long, float) : void
- N getDouble(long) : double
- N putDouble(long, double) : void
- N getAddress(long) : long
- N putAddress(long, long) : void
- N allocateMemory(long) : long
- N reallocateMemory(long, long) : long
- N setMemory(Object, long, long, byte) : void
- setMemory(long, long, byte) : void
- N copyMemory(Object, long, Object, long, long) : v
- copyMemory(long, long, long) : void
- N freeMemory(long) : void

JNI: Registering Native Methods

- JNI provides a method for manually registering native java methods
  - Jvm -> RegisterNatives(jclass, JNINativeMehtod*, jint)
- **JNI type**: JNINativeMethod **nm**
  - Name: **nm**.*name* = "…" [a string, char*]
  - Signature: **nm**.*signature* = "…" [a string, char*]
  - Function Pointer: **nm**.*fnPtr* = … [C++ function pointer, void*]
- This is called "eager linking"
- JVM uses lazy linking
- When a dynamic library is loaded by JVM, it won't link
  - Until a native method is actually called
- **Why "RegisterNatives" is useful?**
  - **No need to use JNI naming convention**: Java_"package name"_"class name"_"method name"
  - This is very useful when trying to reuse already existing C/C++ source codes

Name of java native method

Signature of java native method

Only the name of the C/C++ function matters: **Java_package_class_method**

JNI functions MUST be in the **global** scope

Exception: anonymous namespace

Function Pointers don't need to be in the **global** scope

JNI: Registering Native Methods

- Extremely useful feature: we can use functions inside C++ namespaces!!
- **No need to have JNI calling convention for the function, BUT…**
  - The first parameter must be "**JNIEnv*** "
  - For instance method: second parameter must be "**jobject** "
  - For static method: second parameter must be "**jclass** "
- Example:

Now we can create stand-alone static C++ libraries for JNI and reuse them

```
package test30;

public class Person {

      static {
            System.loadLibrary("native_c");
            registerNatives() ;
      }

      private static native void registerNatives() ;

      // some native methods
      public static native int doubleAge(int age) ;

}
```

JNI knows what parameters to pass to the function

Some function from some math library

```
jint doubleIt(JNIEnv* jvm, jclass Person_class, jint x) {
      return 2*x ;
}

/*
 * Class:     test30_Person
 * Method:    registerNatives
 * Signature: ()V
 */
JNIEXPORT void JNICALL Java_test30_Person_registerNatives
  (JNIEnv *jvm, jclass Person_class) {

      JNINativeMethod doubleAge_method ;
      doubleAge_method.name = (char*) "doubleAge" ;
      doubleAge_method.signature = (char*) "(I)I" ;
      doubleAge_method.fnPtr = (void*) doubleIt ;

      jvm -> RegisterNatives(Person_class, &doubleAge_method, 1) ;
}
```

Must add these two parameters

Function pointer

JNI: Load Handler in C++

- When JVM loads the dynamic library it searches for a particular function
  - **JNI_OnLoad**(JavaVM* vm, void* reserved)
- If JVM finds the "JNI_OnLoad" symbol in the dynamic library, it runs it
- The full signature is like this:

```
extern "C" JNIEXPORT jint JNICALL JNI_OnLoad(JavaVM* vm, void* reserved)
```

- The return type is the version of JDK defined as preprocessor symbols in "jni.h" header file
  - JNI_VERSION_1_4
  - JNI_VERSION_1_6
  - JNI_VERSION_1_8
- We can acquire an instance of JNIEnv* using the JavaVM*

```
JNIEnv* jvm ;
vm -> GetEnv((void**) &jvm, JNI_VERSION_1_8) ;
```

- Note that the second argument specifies which version of JNI we want to work with.

JNI: Solving Refactoring Problem

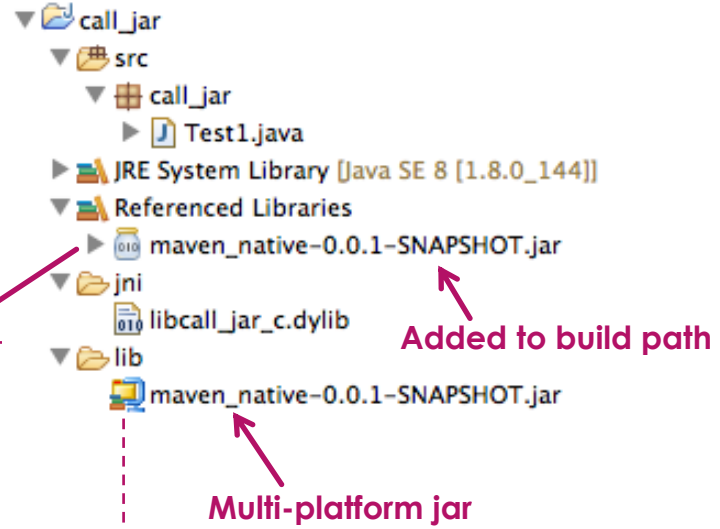
- The conventional JNI binding is tied to the package name and Class name
- This is necessary so that JVM can search for specific native methods in the dynamic library
- Moving a class to another package breaks the JNI/C++ binding
  - Clearly, this causes a big problem if we need to refactor java code
- It may also change the signature of the method parameters
  - Only related to cases when overloading native methods
- Using "**RegisterNatives**" we can solve the java refactoring problem
- **Steps**:
  - Create a java class called "**NativeLibraryLoader**"
    - Add private native "**registerNatives(String classPath)**" to this class
    - Call "**registerNatives(String classPath)**" method inside static initializer
  - Create regular C/C++ functions that take JNIEnv* and jclass/jobject
  - Eagerly link native methods manually

JNI: Using Jar Files in C++

- We can call/use classes inside Jar files when developing JNI applications
- We need to put the jar file on the classpath at runtime
  - **Step 1**: ask JVM to find the class on the classpath: jvm -> FindClass("classpath")
  - **Step 2**: if need an instance of the class, ask JVM: jvm -> NewObject()
  - **Step 3**: ask JVM to access any methods inside the object
- Example:
  - Call static native method "multiply(x, y)" in a jar file from C++

```
public static native double callJar(double x, double y) ;
```

```
JNIEXPORT jdouble JNICALL Java_call_1jar_Test1_callJar
  (JNIEnv *jvm, jclass Test1_class, jdouble x, jdouble y) {
      // step 1: get jclass
      jclass TestMavenJNI_class = jvm -> FindClass("maven_native/TestMavenJNI") ;
      // step 2: get method_id
      jmethodID multiply_id = jvm -> GetStaticMethodID(TestMavenJNI_class, "multiply", "(DD)D") ;
      // step 3: call it
      jdouble result = jvm -> CallStaticDoubleMethod(TestMavenJNI_class, multiply_id, x, y) ;
      // return result
      return result ;
}
```



JNI: Working with Javap

- "**javap**" is a tool that comes with JDK
- It allows disassembling class files (binary files) into human-readable format
- We typically use verbose mode "-v" with javap: "javap –v"
- Similar to javah, it acts on class files (compiled binary) and not java files (source text)
- It is useful for finding the signature of a native method
  - Signature is called "descriptor" in the output of javap
- Example: "**test33/Person**" class

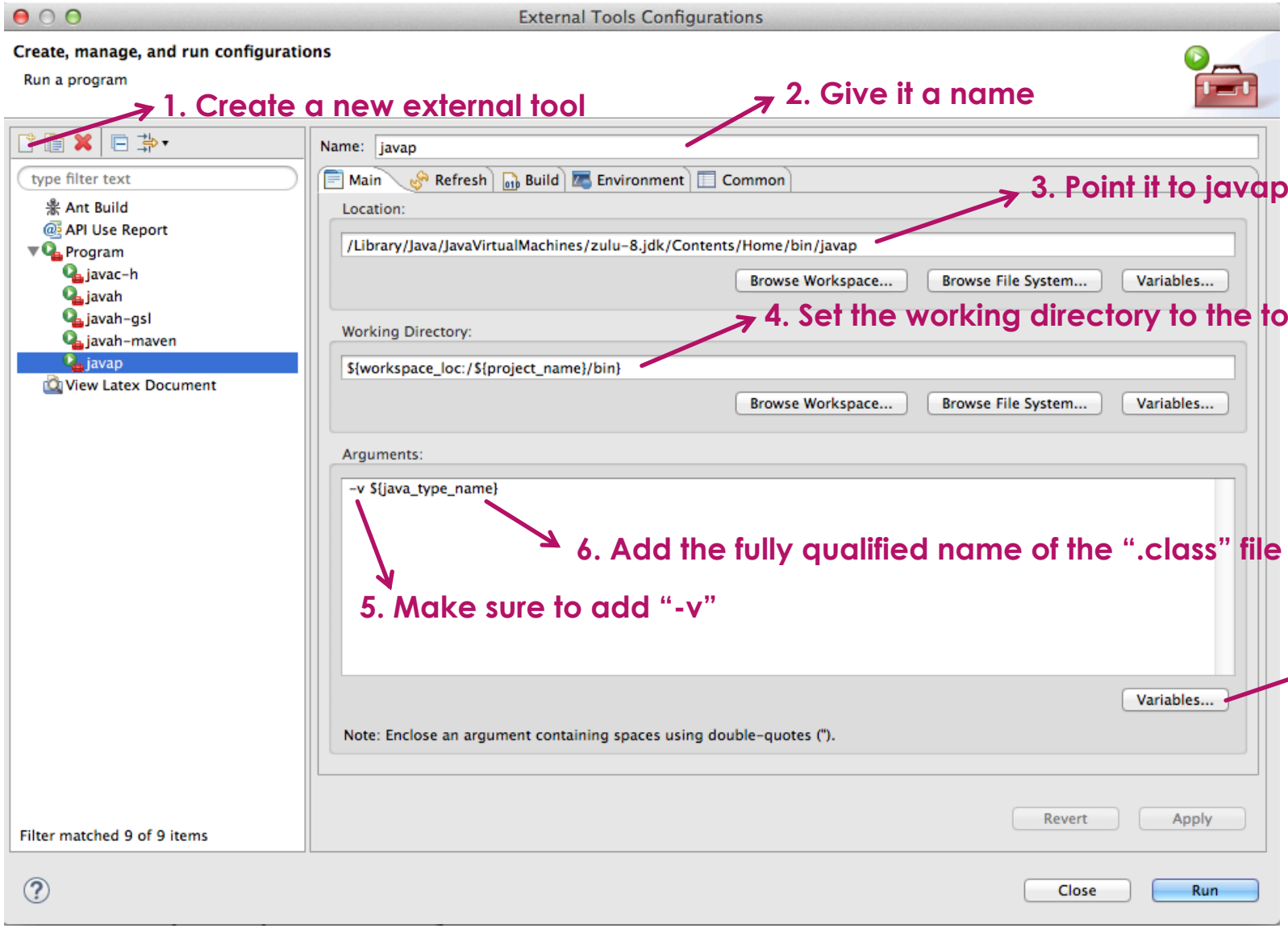
```
package test33;

public class Person {

      String name;
      int age ;

      public Person(String name, int age) {
            this.name = name ;
            this.age = age ;
      }

      public native void printInfo() ;

}
```

Constructor descriptor

Method descriptor

```
public test33.Person(java.lang.String, int);
  descriptor: (Ljava/lang/String;I)V
  flags: ACC_PUBLIC
  Code:
    stack=2, locals=3, args_size=3
…
LocalVariableTable:
        Start  Length  Slot  Name   Signature
            0      15     0  this   Ltest33/Person;
            0      15     1  name   Ljava/lang/String;
            0      15     2   age   I

 public native void printInfo();
  descriptor: ()V
  flags: ACC_PUBLIC, ACC_NATIVE
```

JNI in Eclipse IDE: javap



JDK <= 8 or JDK >=9

- "**javap**" acts on binary files "**.class**"
- "**-v**" can be set to generate a verbose result

JNI: Working with Javap

- "**javap**" syntax: **javap [option] [classname]**
  - -help or --help or -? : prints a help message for the javap command
  - -version : prints Version information of java
  - -v or -verbose : Prints additional information like stack size, number of locals and arguments for methods
  - -l : prints line number and local variable tables
  - -public : prints only public classes and members
  - -protected : prints protected/public classes and members
  - -package : prints package/protected/public classes and members (default)
  - -c : prints Disassembled code
  - -s : prints internal type signatures → **What we typically need for JNI development in C++**
  - -sysinfo : prints system info (path, size, date, MD5 hash) of class being processed
  - -constants : prints final constants of class

See here for some examples: https://www.geeksforgeeks.org/javap-tool-in-java-with-examples/

JNI: C++ Classes/Structs in Java

- Working with **C++ classes** and **structs** in java
  - JNI does not provide an easy way to map java objects to C++ objects and vice versa
- How can we directly work with C++ types (classes)?
  - We interact with C++ through method/function calls [JNI: FFI (foreign function interact)]
  - We need to create C++ objects on the heap (never on the stack)
- Send the address (pointer) of the C++ object to java as "jlong" type [64-bit address]
  - This approach is similar to our approach for developing "Unsafe" class
  - C++ Pointers are 64-bit == "jlong" in C++ or "long" in java
- Use static methods to handle **allocation** and **freeing** of C++ objects
  - private static native jlong allocate(…) → MUST return an **address**
- Create "methods" in the java class that mimic the "member functions" in C++ class
  - private native → convention is to add "0" to the end
- Make constructor private → Use public static methods for allocating and freeing

JNI: C++ Classes/Structs in Java

- Example: **Person.h** and **Person.cpp** → **Person.java**
- We don't really need .cpp file → only the header file is enough
- **Step 1**: create Person.java and add the native methods
  - The only information we need in java is the 64-bit address

C++ class

```
#ifndef PERSON_H_
#define PERSON_H_

#include <iostream>

class Person {
private:
      std::string name ;
      int age ;
public:
      Person(std::string name, int age);
      virtual ~Person() noexcept ;
      // methods
      std::string& getName() ;
      int getAge() ;
      void sayHi() ;
};

#endif /* PERSON_H_ */
```

Native interfacing

Equivalent Java class

```
public class Person {

      static {
            System.loadLibrary("native_c");
      }

      /*-------------------------------------------------*/

      // pointer (64-bit address)
      private long pointer ;

      // memory management [static methods]
      private static native long allocate0(String name, int age) ;
      private static native void free0(long pointer) ;

      // native C++ interfacing
      private native String getName0(long pointer) ;
      private native int getAge0(long pointer) ;
      private native void sayHi0(long pointer) ;
```

JNI: C++ Classes/Structs in Java

- Example: **Person.h** and **Person.cpp** → **Person.java**
- We don't really need .cpp file → only the header file is enough
- **Step 2**: add public java methods for interacting with the rest of java code
  - The only information we need in java is the 64-bit address → Make the constructor **private**

C++ class

```
#ifndef PERSON_H_
#define PERSON_H_

#include <iostream>

class Person {
private:
      std::string name ;
      int age ;
public:
      Person(std::string name, int age);
      virtual ~Person() noexcept ;
      // methods
      std::string& getName() ;
      int getAge() ;
      void sayHi() ;
};

#endif /* PERSON_H_ */
```

Java interfacing

Equivalent Java class

```
public class Person {
      // java interfacing
      private Person(String name, int age) {
            this.pointer = allocate0(name, age) ;
      }
      public static Person allocate(String name, int age) {
            return new Person(name, age) ;
      }
      public static void free(Person person) {
            Person.free0(person.pointer);
      }
      public String getName() {
            return getName0(pointer) ;
      }
      public int getAge() {
            return getAge0(pointer) ;
      }
      public void sayHi() {
            sayHi0(pointer);
      }
```

JNI: Java Exceptions in C++

- Java puts a strong emphasis on exceptions and **exception handling**
- Certain JNI functions use the Java exception mechanism to report error conditions
- JNI functions report error conditions by returning an error code and throwing a Java exception
  - The error code is usually a special return value (such as NULL) that is outside of the range of normal return values
- There are **two cases** where the programmer needs to check for exceptions **without** being able to first check an error code:
  - The JNI functions that invoke a Java method return the result of the Java method.
    - Must call **ExceptionOccurred()** to check for possible exceptions that occurred during the execution of the Java method.
  - Some of the JNI array access functions do not return an error code, but may throw an ArrayIndexOutOfBoundsException or ArrayStoreException.
- In all other cases, a non-error return value guarantees that no exceptions have been thrown.

JNI: Java Exceptions in C++

- There are **two ways** to handle an exception in native code:
  - The native method can choose to return immediately, causing the exception to be thrown in the Java code that initiated the native method call.
  - The native code can **clear** the exception by calling **ExceptionClear()**, and then execute its own exception-handling code.
- After an exception has been raised, the native code must first clear the exception before making other JNI calls. When there is a pending exception, the JNI functions that are **safe to call** are:

Checking/handling exceptions

```
ExceptionOccurred(): jthrowable
ExceptionDescribe(): void
ExceptionClear(): void
ExceptionCheck(): jboolean
```

Releasing memory resources

```
ReleaseStringChars()
ReleaseStringUTFChars()
ReleaseStringCritical()
Release<Type>ArrayElements()
ReleasePrimitiveArrayCritical()
```

Related to Garbage collection

```
DeleteLocalRef()
DeleteGlobalRef()
DeleteWeakGlobalRef()
```

- Check for exceptions with: **ExceptionCheck()**
- Handle the exception with: **ExceptionClear()**
- Print the stack trace with: **ExceptionDescribe()**
- Get Exception Object with: **ExceptionOccurred()**

jvm → Throw() jvm → ThrowNew()

JNI: Java Exceptions in C++

- Example: assume **Person.java** throws checked exception in the constructor

```
public Person(String name, int age) throws IOException {

     if (age<=0) {
          throw new IOException("age cannot be zero or
negative.") ;
     }

     this.name = name ;
     this.age = age ;
}
```

A native method that calls the constructor

```
public static native Person createPerson(String name, int age) ;
```

Person bob = **new** Person(…) → Forced to handle exception in java

Person bob = **createPerson**(…) → Native method is not forced to handle checked exceptions in java

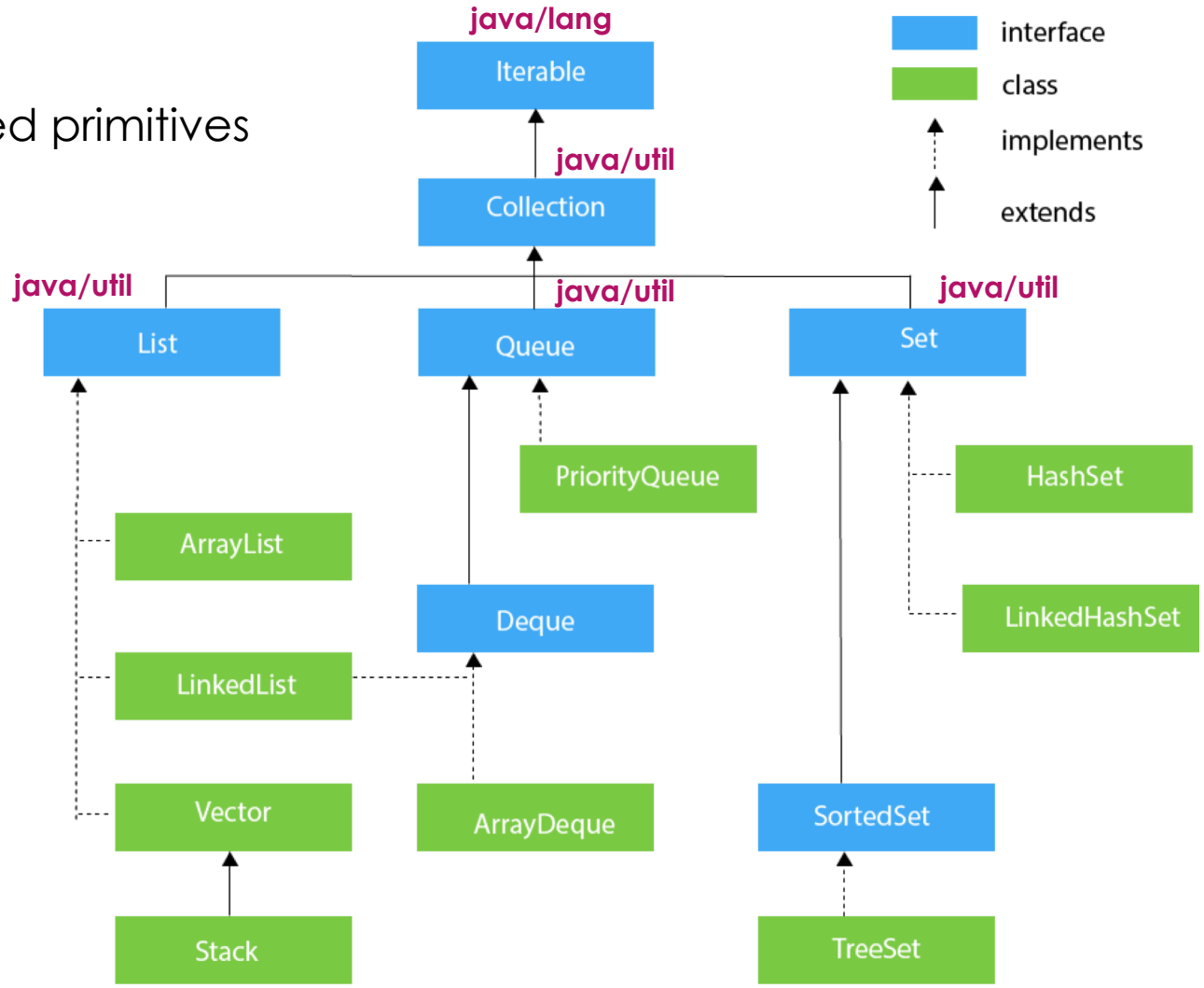
- Handling the java Exception in C++

This code is equivalent to "try … catch …" block in java

```
jobject Person_obj = jvm -> NewObject(Person_class, constructor_id, name, age) ;
if(jvm -> ExceptionCheck()) { // check for exception
     jvm -> ExceptionDescribe() ; // print stack trace
     std::cout << "Java Exception occurred in C++!!" << std::endl ;
     jvm -> ExceptionClear() ; // handle the exception
     return jvm -> NewObject(Person_class, constructor_id, jvm -> NewStringUTF("Bob"), 21) ;
} else {
     std::cout << "Successfully created person object in C++!!" << std::endl ;
     return Person_obj ;
}
```

JNI: Working with Java Collections

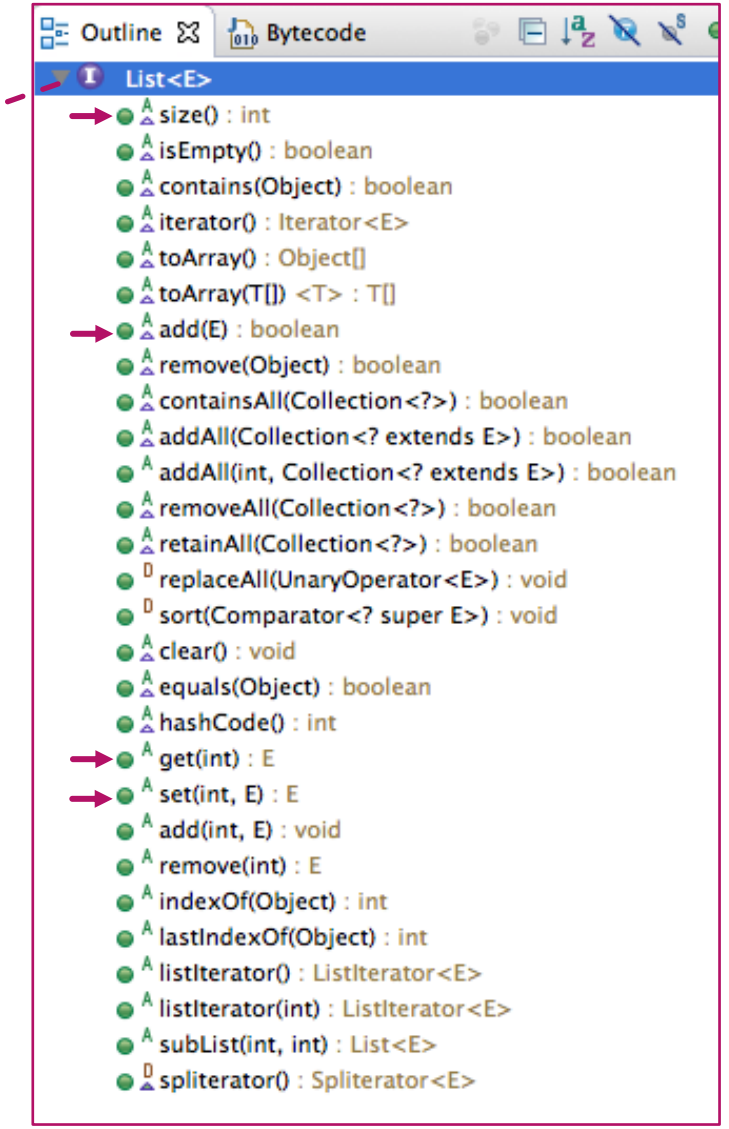
- We **can not** use primitives with Generics in Java
- **General approach**: Need to call the methods
  - No auto boxing or auto unboxing for boxed primitives
- **List<E>**
  - **E : jobject** in C++
  - "**size**()", "**get**(index)", "**add**(obj)", …
- **Set<E>**
  - **E : jobject** in C++
  - "**size**()", …
- **Map<K,V>**
  - **K : jobject** in C++
  - **V : jobject** in C++
  - "**put**(K, V)", "**get**(K)"



JNI: Working with Java List<E>

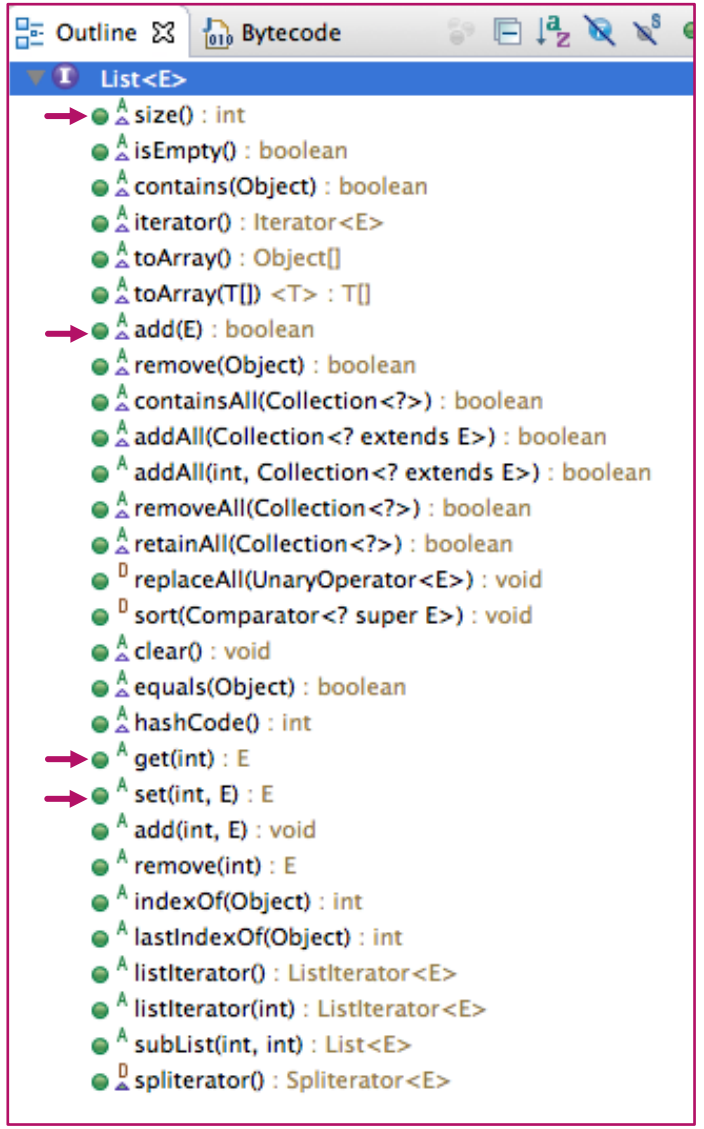
- **General approach**: Need to call the methods
  - No auto boxing or auto unboxing for boxed primitives in C++
- **List<E>**
  - **E : jobject** in C++, **E : java/lang/Object** in java
- Type erasure: generics are purely compile-time concepts
- Signature (used for jclass description → classpath description):
  - **public abstract interface java/util/List implements java/util/Collection**
- Important methods and their signatures [JDK8] for jmethodID
  - "**size**()" → "**()I**"
  - "**get**(index)" → "**(I)Ljava/lang/Object;**"
  - "**add**(obj)" → "**(Ljava/lang/Object;)Z**"
  - "**set**(index, obj)" → "**(ILjava/lang/Object;)Ljava/lang/Object;**"
- List<E> interface does not have any static method

Unbounded generics



JNI: Working with Java List<E>

- Methods of List<E> and their signatures
  - "**isEmpty**()" → "**()Z**"
  - "**contains**(obj)" → "**(Ljava/lang/Object;)Z**"
  - "**iterator**()" → "**()L???;**"
  - "**toArray**()" → "**()[Ljava/lang/Object;**"
  - "**toArray**(T[])" → "**([Ljava/lang/Object;)[Ljava/lang/Object;**"
  - "**remove**(obj)" → "**(Ljava/lang/Object;)Z**"
  - "**containsAll**(col)" → "**(Ljava/util/Collection;)Z**"
  - "**addAll**(col)" → "**(Ljava/util/Collection;)Z**"
  - "**removeAll**(index, col)" → "**(Ljava/util/Collection;)Z**"
  - "**retainAll**(col)" → "**(Ljava/util/Collection;)Z**"
  - "**replaceAll**(uop)" → "**(Ljava/util/function/UnaryOperator;)V**"
  - "**sort**(cmp)" → "**(L???;)V**"



JNI: Working with Java List<E>

- Methods of List<E> and their signatures
  - "**clear**()" → "**()V**"
  - "**equals**(obj)" → "**(Ljava/lang/Object;)Z**"
  - "**hashcode**()" → "**()I**"
  - "**add**(index, obj)" → "**(ILjava/lang/Object;)V**"
  - "**remove**(index)" → "**(I)Ljava/lang/Object;**"
  - "**indexOf**(obj)" → "**(Ljava/lang/Object;)I**"
  - "**lastIndexOf**(obj)" → "**(Ljava/lang/Object;)I**"
  - "**listIterator**()" → "**()L???;**"
  - "**listIterator**(index)" → "**(I)L???;**"
  - "**subList**(index,index)" → "**(II)Ljava/lang/Object;**"
  - "**Spliterator**()" → "**()L???;**"