DYNAMIC LINK LIBRARY

- A dynamic-link library (DLL) is **a module that contains functions and data that can be used by another module (application or DLL)**.

- A DLL can define two kinds of functions: exported and internal.The exported functions are intended to be called by other modules, as well as from within the DLL where they are defined.

- Significance:
  DLL files are stored in a variety of program functions (sub-procedures) implementation process, when the program needs to call the function to load the DLL, and then get the address of the function, and finally make the call.
  The advantage of using DLL files is that the program does not need to load all the code at the beginning of the run, but only when a function is needed by the program to remove it from the DLL. In addition, using DLL files can also reduce the volume of your program.

ADVANTAGES OF DLL

(1)Save more memory and reduce page switching

(2)programs written in different programming languages can call the same DLL function just according to the function calling convention

(3)Suitable for large-scale software development, make the development process independent, the coupling degree is small, facilitates the development and the test between different developer and the development organization.

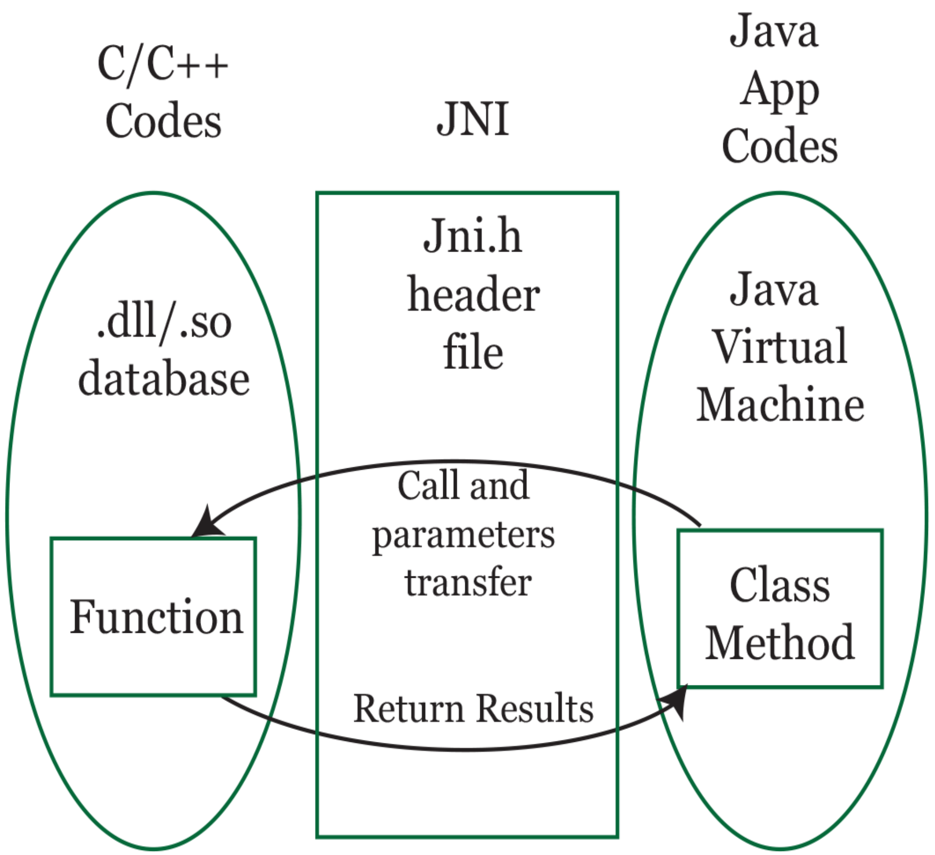
(4)Save disk space: When an application uses dynamic linking, multiple applications can share a single copy of the DLL on disk. In contrast, when an application uses a static-link library, each application links the library code to the executable image as a separate copy.

DLL IMPLEMENTATION WITH JNI

JAVA NATIVE INTERFACE

The Java Native Interface (JNI) enables runtime linking to dynamic native libraries.

- The need to handle some hardware
- Performance improvement for a very demanding process
- An existing library that we want to reuse instead of rewriting it in Java



NATIVE METHOD

- When making a native executable program, we can choose to use static or shared libs:
- Static libs – all library binaries will be included as part of our executable during the linking process.Thus, we won't need the libs anymore, but it'll increase the size of our executable file.
- Shared libs – the final executable only has references to the libs, not the code itself. It requires that the environment in which we run our executable has access to all the files of the libs used by our program.
- The latter is what makes sense for JNI as we can't mix bytecode and natively compiled code into the same binary file.
- Therefore, our shared lib will keep the native code separately within its *.so/.dll/.dylib* file (depending on which Operating System we're using) instead of being part of our classes.

PROBLEM STATEMENT

- Write a program to create a Dynamic Link Library for any mathematical operations and write an application program to test it. (Java Native Interface/Use VB/VC++)

Key components needed

Java Code – our classes. They will include at least one *native* method.

Native Code – the actual logic of our native methods, usually coded in C or C++.

JNI header file – this header file for C/C++ (*include/jni.h* into the JDK directory) includes all definitions of JNI elements that we may use into our native programs.

C/C++ Compiler – we can choose between GCC, Clang, Visual Studio, or any other we like as far as it's able to generate a native shared library for our platform.

- **Steps to create Dll with JNI**
  - Write a Java Class that uses C Code- **TestJNI.java**
  - **Compile TestJNI.java**
  - Create the C/C++ Header file **- TestJNI.h**
  - C Implementation **- TestJNI.c**
  - Run the Java Program

1. Write a Java Class that uses C Codes - TestJNI.java

```
public class TestJNI {
       static {

           System.loadLibrary(“libcal"); // Load native library at runtime
                 // cal.dll (Windows) or libcal.so (Unix)
          }

         // Declare a native method add()
         private native int   add(int   n1,  int   n2);

        // Test Driver
        public static void main(String[] args) {
           // invoke the native method
              System.out.println(“Addition is=”+new TestJNI().add(10,20);
                 }
     }
```

it will be implemented in a separate native shared library.

“native” keyword – any method marked as native must be **implemented in a native, shared lib**.

System.loadLibrary(String libname) – a static method that **loads a shared library** from the file system into memory and makes its exported functions available for our Java code.

2. Compile Java code:

```
$ javac TestJNI.java
```

3. Create the C/C++ Header file **TestJNI.h**

```
$ javah -jni TestJNI
```

4. C Implementation - TestJNI.c

```
#include <jni.h>
#include <stdio.h>
#include “TestJNI.h"
// Implementation of native method add() of TestJNI class

JNIEXPORT jint JNICALL Java_TestJNI_add(JNIEnv *env , jobject
thisObj , jint n1 , jint n2)
  {
          jint res;
          res=n1+n2;
          return res;
}
```

JNIEXPORT- marks the **function into the shared lib as exportable** so it will be included in the function table, and thus JNI can find it

JNICALL – combined with *JNIEXPORT*, it ensures that our **methods are available for the JNI framework**

JNIEnv – a structure containing methods that we can use our native code to access Java elements
jobject: reference to "this" Java object.

JavaVM – a structure that lets us manipulate a running JVM (or even start a new one) adding threads to it, destroying it, etc…

JNI defines the following JNI types in the native system that correspond to Java types:

Java Primitives: **jint**, jbyte, jshort, jlong, jfloat, jdouble, jchar, jboolean for Java Primitive of int, byte, short, long, float, double, char and boolean, respectively.

The native programs:

- Receive the arguments in JNI type (passed over by the Java program).
- For reference JNI type, convert or copy the arguments to local native types
- Perform its operations, in local native type.
- Create the returned object in JNI type, and copy the result into the returned object.
- Return.

5.Compile c-program:

```
$gcc -I /usr/local/jdk1.8.0_91/include   /usr/local/jdk1.8.0_91/include/linux  -o  libcal.so–shared TestJNI.c
```

6. Run java program

```
$java -Djava.library.path=.TestJNI
Addition is=30
```