Some of the Examples Where native methods are used in java are :

**java.lang.System class**

public static native long currentTimeMillis();

public static native long nanoTime();

**java.lang.Runtime**

public native void gc();

public native int availableProcessors();

public native long freeMemory();

public native long totalMemory();

public native long maxMemory();

**java.io.FileInputStream**

private native void open(String name) throws FileNotFoundException;

public native int read() throws IOException;

public native long skip(long n) throws IOException;

public native int available() throws IOException;

private native void close0() throws IOException; // called by "close()" method

**java.io.FileOutputStream**

private native void open(String name) throws FileNotFoundException;

private native void openAppend(String name) throws FileNotFoundException;

public native void write(int b) throws IOException;

private native void writeBytes(byte b[], int off, int len) throws IOException;

private native void close0() throws IOException; // called by "close()" method

**java.lang.Thread**

public static native Thread currentThread();

public static native void sleep(long millis) throws InterruptedException;

private native void start0(); // it is called by "start()" method

public final native boolean isAlive();

private native void setPriority0(int newPriority); // it is called by "setPriority"

**java.lang.Object**

public final native Class getClass();

public native int hashCode();

protected native Object clone() throws CloneNotSupportedException;

public final native void notify();

public final native void notifyAll();

public final native void wait() throws InterruptedException;

**Class**

private static native Class forName0(String name, boolean initialize,

ClassLoader loader)throws ClassNotFoundException;

private native String getName0();

..... most of the methods

**Benefits and Trade-Offs**

The presense of native methods offers many benefits, the biggest being the extension of Java power. However, there is always a downside to all good things, and native methods definitely have their downsides. Depending on what the goals of your application are, the downsides may not be that terrible. Foremost is the fact that, by definition, the use of native methods defeats several of Java's main goals: **platform neutrality**, **security** etc.

But still , Java is such a nice language to develop , that you don’t need to use native methods unless you have to.

**Platform Neutrality**

Most likely, native methods are implemented in C or C++. Although those languages have standards, these standards leave a lot of room for implementation-defined attributes. For example, the file systems of UNIX and Win32 have some differences. For each platform you choose to support, you will have to implement several flavors of the native method.

The Java language and runtime provide a number of features that make applications more robust and safe. Java's memory management, synchronization features, and lack of address manipulation help prevent common programming mistakes from slipping through the development and testing phases of your product. However, once you drop out of Java into a native method, you are, once again, at the mercy of the language and system in which you are implementing the native method.

**Security Concerns**

A Java virtual machine is much more capable of detecting an "evil" Java program than an application in other languages. Once you drop into a native method, the Java virtual machine can no longer verify, catch, or prevent the program from violating the security of the environment in which the Java virtual machine is running.

**How Does This Magic Work?**

**Using Dynamic Linking**

Java implementation interfaces to native methods , by using the dynamic linking capabilities of the underlying operation system. The Java virtual machine is a complete program, which is already compiled for its respective platform. The nature of Java enables it easily to absorb a Java class and execute its behavior. However, for a compiled native method, things are not so simple. Somehow, the Java virtual machine must be taught how to call this native method. This is done by relying on the implementation of native methods to reside in a dynamic link library, which the operating system magically loads and links into the process that is running the Java virtual machine. On the Solaris platform, such a library is often called *shared objects*, or *shared libraries*, or simply *dot-so's (.so's)*. On Win32 platforms, they are called *dynamic link libraries* *(DLLs)*.

Before a native method is invoked, the Java virtual machine must be told to find, load, and link the necessary DLLs, which contain the native method implementations. This is conveniently achieved by using the static method java.lang.System.loadLibrary("mypro"). The string "mypro" is mapped to a DLL named mypro.so on Solaris and mypro.dll on Win32.

**Defining the Calling Convention**

In, essence, Sun defines the method its Java virtual machine will use to call external functions. In order to dynamically link and call the implementation of a native method successfully, the Java virtual machine must know several details. It must know the name of the function within the DLL (the implementation of the native method) to locate the symbol and its entry point. It also must know how to call that function (its return type, number of parameters, and types of parameters).

The Sun JDK provides a tool, named javah, to help you create your native method implementation functions. The developer of native methods runs javah, passing it the name of a class. javah produces header file (.h) . The .h file will contain the prototypes of the functions Java will call, and thus expect to find in the DLL. Thus, the developer needs to fill in only the details of the functions in the c file and build the DLL appropriately.

**How the Virtual Machine Makes It Work**

When a class is first used by Java, its class descriptor is loaded into memory. The *class descriptor* can be thought of as a directory for all services provided by the class-there is only one class descriptor loaded, regardless of how many instances of that class exist. Among its entries is a list of *method descriptors,* which contain information specific to methods, including where the code is, what parameters they take, and method modifiers.

If a method descriptor has its native modifier set, the block will include a pointer to the function that implements that native method. This function resides in some DLL but will be loaded into the Java processes address space by the operating system. At the time the class descriptor with native methods is loaded, the associated DLL does not have to be loaded, and thus the function pointer will not be set. Sometime prior to a native method being called, the associated DLL should be loaded. This is done via a call to java.system.loadLibrary(). When this call is made, Java will find and load the DLL but will still not resolve symbols; the resolution phase is delayed until the point of use. At the time of a call to a native method, Java will first check to see whether the native method implementation function has already been resolved-that is, its pointer is not null. If it has been previously resolved, the call is performed; otherwise, the resolution of the symbols is attempted. The resolution is performed by making an operating system call to see whether the symbol exists in the caller's address space. If the symbols are correctly resolved, the call is performed as if the Java virtual machine was making a standard C call to its own internal functions. If the resolution fails, the exception java.lang.UnsatisfiedLinkError will be thrown at the point of the native method call.