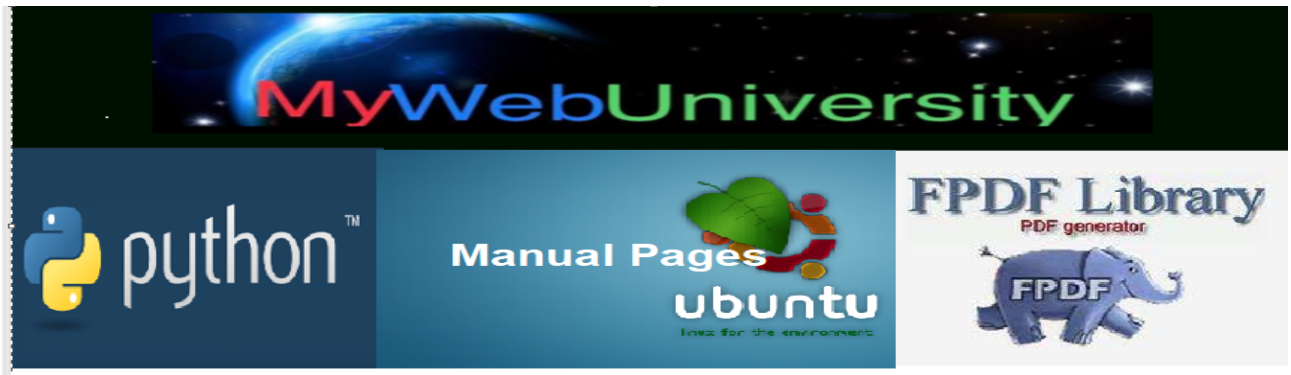

Full credit is given to the above companies including the OS that this PDF file was generated!

Linux Ubuntu 26.04 Manual Pages on command 'jnativescan.1'

$ man jnativescan.1

JNATIVESCAN(1) JDK Commands JNATIVESCAN(1)

NAME

jnativescan - static analysis tool that scans one or more jar files for uses of native func?
tionalities, such as restricted method calls or native method declarations.

SYNOPSIS

jnativescan [options]

options

See Options for the jnativescan Command

DESCRIPTION

The jnative tool is a static analysis tool provided by the JDK that scans a JAR file for uses of native functionalities, such as restricted method calls or native method declara?
tions.

jnativescan accepts a runtime class path and module path configuration, as well as a set of root modules, and a target release. It scans the jars on the class and module paths, and reports uses of native functionalities either in a tree like structure, which also identi?
fies that calling classes and methods, or as a list of module names when the --print-na?
tive-access flag is specified.

OPTIONS FOR THE JNATIVESCAN COMMAND

The following options are available:

--class-path path

Used to specify a list of paths pointing to jar files to be scanned.

All jar files specified through this list will be scanned. If a jar file contains a Class-Path attribute in its manifest, jar files listed there will be scanned as well. Jar

files listed in the Class-Path manifest attribute that can not be found are ignored. All the jar files found are treated as if they belonged to the unnamed module.

--module-path path

Used to specify a list of paths pointing to jar files or directories containing jar files, that the tool can use to find modules that need to be scanned. The list of jar files that will be scanned depends on the --add-modules option.

For both the --class-path and --module-path options, path should be a search path that consists of one or more jar files, separated by the system-specific path separa? tor. For example:

? Linux and macOS:

--class-path /some/foo.jar:/another/different/bar.jar

Note:

On Windows, use a semicolon (;) as the separator instead of a colon (:).

? Windows:

--class-path C:\some\foo.jar;C:\another\different\bar.jar

--add-modules module[,module...]

Used to specify a comma-separated list of module names that indicate the root modules to scan. All the root modules will be scanned, as well as any modules that they de? pend on. This includes dependencies on service implementations specified through the uses directive in a module's module-info file. All modules found on the module path that provide an implementation of such a service will be scanned as well.

--release version

Used to specify the Java SE release that specifies the set of restricted methods to scan for. For multi-release jar files, this option also indicates the version of class file that should be loaded from the jar. This option should be set to the ver? sion of the runtime under which the application is eventually intended to be run. If this flag is omitted, the version of jnativescan is used as release version, which is the same as the version of the JDK that the tool belongs to.

--print-native-access

Print a comma-separated list of module names that use native functionalities, instead of the default tree structure.

--help or -h

Prints out a full help message.

--version

Prints out the abbreviated version string of the tool.

EXAMPLE OF jnativescan USE

jnativescan accepts a runtime configuration in the form of a class path, module path, set of root modules, and a target release version. For the class path, the tool will scan all jar files, including those found recursively through the Class-Path manifest attribute. For the module path, the tool scans all root modules specified through --add-modules, and any (transitive) dependence of the root modules, including any modules that contain service implementations that are used by a scanned module.

By default, the tool prints out which jars, classes, and methods use native functionalities, in a tree-like structure. The following is an example output:

```
$ jnativescan --class-path app.jar
app.jar (ALL-UNNAMED):
  foo.Main:
    foo.Main::main(String[])void references restricted methods:
      java.lang.foreign.MemorySegment::reinterpret(long)MemorySegment
    foo.Main::nativeMethod()void is a native method declaration
```

app.jar (ALL-UNNAMED) is the path to the jar file, with the module name in parentheses behind it. Since in this case the jar file appears on the class path, ALL-UNNAMED is printed to indicate the unnamed module. The second line of the output, foo.Main, indicates that methods using native functionalities were found in the foo.Main class. The next line:

```
foo.Main::main(String[])void references restricted methods:
```

Indicates that the main(String[]) method in the foo.Main class references a restricted method, which is listed on the following line as:

```
java.lang.foreign.MemorySegment::reinterpret(long)MemorySegment
```

Lastly, the text:

```
foo.Main::nativeMethod()void is a native method declaration
```

Indicates that the foo.Main class contains a declaration of a native method named nativeMethod.

If we add --print-native-access to the example command line, we instead get a list of the names of modules that contain accesses to native functionalities:

```
$ jnativescan --class-path app.jar --print-native-access
ALL-UNNAMED
```

In this case the output consists of just ALL-UNNAMED, which indicates a jar file on the class path, that is, in the unnamed module, contains an access to native functionalities.