

# NewJ Library for C++ Developer's Guide



**PureNative** Software

[www.pure-native.com](http://www.pure-native.com)

Copyright © 2002-2004 PureNative Software Corporation. All rights reserved.

PureNative, the PureNative logo, NewJ, the NewJ logo, NewJNI, NewJNIInterop, and Pie are trademarks of PureNative Software Corporation. Java is a registered trademark of Sun Microsystems, Inc. All other trademarks and registered trademarks are property of their respective owners.

The software described herein is licensed. Unauthorized reproduction or distribution of this software, or any portion of it, may result in severe civil and criminal penalties, and will be prosecuted to the maximum extent possible under law. Consult accompanying license agreement for additional permissions and restrictions.

*NewJ Library for C++* is based on the Java 2 Platform, Standard Edition, version 1.2.2 API Specification from Sun Microsystems, Inc. *NewJNI* and *NewJNIInterop* are compatible with the JNI Specification from Sun Microsystems, Inc., and support any JNI-compatible version of the Java platform, including version 1.2, 1.3, or later. *NewJ Library for C++* is not a product of nor endorsed by Sun Microsystems, Inc.

Patent Pending.

Developer release 2: May 2004.

# **{4} Getting Started: Your First NewJ Application**

## **Your First Steps**

This chapter takes you step-by-step through the process of creating a NewJ application. These steps introduce you to the NewJ AppWizards, the Pie Run-time Library, and the Core J2 Library. This chapter also explains how to adapt common Java language source code to C++ using the NewJ Library. After covering this chapter, you will be ready to create your very own NewJ applications.

## **Your First Application**

The application that we will be creating is a simple `grep` program, which may be run from the command line. (For those who are unfamiliar with `grep`, it is a console program which searches a file for occurrences of a given pattern.) Our implementation will be very basic; it will only search one file, one line at a time, for an exact string match, without supporting regular expressions. (If you wish to enrich `JGrep` later to include these features, you are welcome to do so.) This application will make use of and demonstrate various classes in the Pie Run-time Library and Core J2 Library. It will acquaint you with their various features and show you how to use them in your own applications.

## **Step 1: Create a New Project with the NewJ Console AppWizard**

First, we need to create a new project for our console application. To do this, follow these steps in order:

1. From within the Visual C++ 6.0 IDE, select the menu item `File > New...`
2. Select the `Projects` tab
3. Select `"NewJ Console Application"`
4. Enter `JGrep` as the name of our application, which is also the name of our main application class
5. Choose `OK`.
6. Make sure `JGrep` appears as the name of your main application class.
7. Choose `Finish`.
8. When the `New Project Information` window appears, choose `OK`.

Our NewJ console application is now ready for editing.

## Step 2: The main method—Where It All Begins

Second, we need to handle the command-line arguments that we receive. These arguments are passed to the main method of our main application class. This method is also the starting point of execution for our application.

Our simple grep program will only take two arguments, a file name and a search pattern. (If you wish to support more sophisticated command-line arguments, this is left as an exercise for the reader.) The main method will make sure that exactly two arguments have been provided. If not, a usage message will be displayed and the program will terminate. If so, the arguments will be passed on to the doGrep method, which takes these two arguments.

Here is the Java language version of the main method:

```
public static void main(String[] args)
{
    if (args.length != 2)
    {
        System.out.println("usage: JGrep filename searchpattern");
        System.exit(0);
    }

    doGrep(args[0], args[1]);
}
```

Here is the NewJ/C++ version of the main method definition. Please change the main method definition in “JGrep.cpp” to the following.

```
void JGrep::main(::java::lang::StringAR args)
{
    if (args->length_ != 2)
    {
        System::out_->println(PI_T("usage: JGrep filename searchpattern"));
        System::exit(0);
    }

    doGrep(args[0], args[1]);
}
```

Let's consider some of the differences between the two versions. First, the C++ version separates the method declaration from the method definition, so the class name must be specified before the method name along with a double colon to separate the names. Second, the argument type is “::java::lang::StringAR” instead of “String[]”. Note that the Java language implicitly imports the java.lang package/namespace but C++ does not (or any other namespace for that matter). To use the “::java::lang” namespace in C++, put the following state after all of your #include statements but before your method definition:

```
using namespace ::java::lang;
```

If you do this, you can reduce “`::java::lang::StringAR`” to “`StringAR`”. What does the “AR” stand for? you may be wondering. In order to be maintain safety and fidelity, NewJ provides its own implementation of references and arrays. Its references are akin to C++ “smart pointer” objects, and its arrays are akin to “smart pointer” array objects. (For all the particulars about NewJ arrays and references, see the chapter on the Pie Run-time Library Reference.) Thus, the “A” stands for array, and the “R” stands for reference, so together it is an array reference. This is the same as “`String[]`” in the Java language.

The first statement in the Java language version of the main method is:

```
if (args.length != 2)
```

In this statement, `args` is an instance variable which is a reference to a `String` instance. In the Java language, both instance variables and static variables are dereferenced by using a period (`.`). In C++, instance variables are dereferenced with the indirection operator, a dash followed by the greater than symbol (`->`), and static variables/fields are dereferenced using the namespace resolution operator, a double semicolon (`::`). Additionally, an underscore must usually be appended to a field name. This is necessary to distinguish between methods and fields that have the same exact name, which C++ does not support although the Java language does. According to these rules, in the above statement, the period is changed to a dash followed by a semicolon, and an underscore suffix is added to the field named `length`, producing:

```
if (args->length_ != 2)
```

The next Java language statement is:

```
System.out.println("usage: JGrep filename searchpattern");
```

Because `System` is a class name and `out` is a static variable/field, the first period must be changed to a double semicolon. The succeeding period is used to dereference the `out` instance variable/field so an underscore is added as a suffix to `out` and the period is changed to a dash followed by a greater than symbol, producing:

```
System::out->println(PI_T("usage: JGrep filename searchpattern"));
```

Everything looks like you expected except for the `PI_T()` around the string literal. What is `PI_T()` for? In the Java language, all string literals are implicitly `String` objects. In order to support this behavior in C++, the `PI_T()` macro (simulated keyword) is necessary. It causes the string literal to be wide characters, which is often implemented as 16-bit Unicode. From this wide string literal, it creates a `String` object and internalizes it to guarantee that it is unique just like the Java language mandates. (For more information on internalization, see the standard specification for `java.lang.String.intern()`.) Make sure to always put your string literals inside `PI_T()` whenever you are assigning `String` objects to such literals or passing them to a method that takes `String` objects.

## Step 2: Adding a Method to Our Main Class

Now we need to add the `doGrep` method to our main class `JGrep`. In the Java language, we declare and define the method all within the class declaration like so:

```
private static void doGrep(String fileName, String searchPattern)
{
}
```

However, in C++ we declare the method in the class declaration file and then define the method in the class definition file. For `JGrep`, the class declaration file is “`JGrep.h`” and the class definition (implementation) file is “`JGrep.cpp`”.

Add the following method declaration to the class `JGrep` in “`JGrep.h`”:

```
private: static void doGrep(::java::lang::StringR fileName, ::java::lang::StringR
searchPattern);
```

Notice that in C++ a colon is required after the access specifier keyword, which is `private` in this case, and a semicolon is required at the end of the statement.

Now let's add the following method definition in “`JGrep.cpp`”. This may be added at the end of file.

```
void JGrep::doGrep(::java::lang::StringR fileName, ::java::lang::StringR
searchPattern)
{
}
```

Of course, `doGrep()` will not actually do anything yet, but our application may be fully compiled and run at this point. First, set the active configuration to Release. Select the menu item `Build > Set Active Configuration...` When the dialog appears, select “`JGrep - Win32 Release`” and choose OK. To build the application for release, press F7 or select the menu item `Build > Build JGrep.exe`. After it builds successfully, run the application by pressing F5 or selecting the menu item `Build > Start Debug > Go`. Our console application should output the following message:

```
usage: JGrep filename searchpattern
```

So far, so good! If you get different results, please go back and recheck your steps.

## Step 3: Using External Classes and Namespaces

As mentioned earlier, the Java language implicitly imports the `java.lang` package/namespace. This means this statement is never needed:

```
import java.lang.*;
```

However, the foregoing not true of other external classes and packages. These must be explicitly imported. In the our application, we will be making use of the following classes: `java.io.RandomAccessFile`, `java.io.FileNotFoundException`, `java.io.IOException`. And we will need to explicitly import all of these classes. We will import class names specifically instead of entire packages. Whenever practical it is recommended that you import class names instead of packages because it will be easier to see which classes are actually being used and there is less potential for conflicts between class names.

Here are our import statements in the Java language. Note that these are placed at top of the `JGrep.java` source file.

```
import java.io.RandomAccessFile;
import java.io.FileNotFoundException;
import java.io.IOException;
```

In C++, besides using the class names we also need to include the class declarations themselves. After all class declarations have been included, the using statements, if desired, may be added. Here is our C++ implementation. Note that the following statements should be inserted after any existing `#include` statements and before any existing using statements.

```
#include <java/io/RandomAccessFile.h>
#include <java/io/FileNotFoundException.h>
#include <java/io/IOException.h>

using namespace ::java::lang; // see earlier discussion on this
using ::java::io::RandomAccessFile; // the class type
using ::java::io::RandomAccessFileR; // the reference type
using ::java::io::FileNotFoundException; // the class type
using ::java::io::FileNotFoundExceptionR; // the reference type
using ::java::io::IOException; // the class type
using ::java::io::IOExceptionR; // the reference type
```

In the above source code, separate using statements are required for the class type and the reference type, and even for the array reference types if they are being used. You may want to simplify this by using the `pi_importclass` simulated keyword (macro) instead. It takes the following form:

```
pi_importclass(className);
```

where *className* is the name of the class you wish to use. This macro covers the class type itself and all reference types for the class, including array reference types. Here is the C++ implementation using `pi_importclass`:

```
pi_importclass(::java::io::RandomAccessFile);
pi_importclass(::java::io::FileNotFoundException);
pi_importclass(::java::io::IOException);
```

Much simpler, although some C++ developers may prefer the former version for the sake of familiarity with the using statement.

#### Step 4: Reading from the Input File

We're now ready to begin working with files. First, we need to open the file for reading. We will use the standard `java.io.RandomAccessFile` class to accomplish this. We will repeatedly call its `readLine()` method to read each line of text in the file until the end of file is reached, which it indicates by returning null.

Here is the implementation of `doGrep()` in the Java language:

```
private static void doGrep(String fileName, String searchPattern)
{
    try
    {
        RandomAccessFile file = new RandomAccessFile(fileName, "r");

        for (;;) // loop forever
        {
            String line = file.readLine();

            if (line == null)
            {
                break;
            }

            file.close();
        }
    } catch (FileNotFoundException exception)
    {
        System.out.println("Unable to open file '" + fileName + "'");
    }
    catch (IOException exception)
    {
        System.out.println("Unexpected read error occurred");
    }
}
```

And here is the implementation of `doGrep()` in C++:

```
void JGrep::doGrep(::java::lang::StringR fileName, ::java::lang::StringR searchPattern)
{
    try
    {
```

```

        RandomAccessFileR file = pi_new(RandomAccessFile,
RandomAccessFile(fileName, PI_T("r")));

        for (;) // loop forever
            {
                StringR line = file->readLine();

                if (line == pi_null)
                    {
                        break;
                    }
            }

        file->close();
    }
catch (FileNotFoundExceptionR& exception)
    {
        System::out_->println(PI_T("Unable to open file ") + fileName +
PI_C("\n"));
    }
catch (IOException& exception)
    {
        System::out_->println(PI_T("Unexpected read error occurred"));
    }
}

```

Did you notice some of the differences between the two versions? There are the changes we have discussed earlier, such as, appending the letter “R” to reference types, changing class dereferences from a period to two semicolons, appending an underscore to field names, and change field dereferences from a period to a dash followed by a semicolon. But there are also a few other changes that have not been previously covered. Let's consider them one at a time.

First, there is new object instantiation. In Java, to instantiate a new object you use the form:

```
new ClassName(parameter1, parameter2, ...)
```

This form is 100% compatible with C++ but it may not always produce the same results as Java and may not be 100% safe. Why is this? For two reasons. One, the Java language guarantees that new object fields are automatically initialized to 0 or null, but ANSI C++ leaves the value of new object fields undefined, presumably for the sake of performance. Second, the Java language will take care of garbage collecting newly created objects, whether they are assigned to a variable or not. On the other hand, C++ does not provide this feature and thus will leak unassigned new objects. To prevent both of these problems, the Pie Library provides the `pi_new` simulated keyword (macro). It takes the form:

```
pi_new(ClassName, ClassName(parameter1, parameter2, ...))
```

Substituting `pi_new()` for `new`, the Java language statement:

```
RandomAccessFile file = new RandomAccessFile(fileName, "r");
```

becomes:

```
RandomAccessFileR file = pi_new(RandomAccessFile,  
RandomAccessFile(fileName, PI_T("r")));
```

The second change involves the `null` keyword. The `null` keyword in Java is just intended for use with object and interface references, whereas the common definition of `null` is much broader. To rectify this difference, the Pie Library provides its own definition of `null` called `pi_null`. Make sure to always replace `null` with `pi_null` in NewJ application logic.

The catch statement in NewJ C++ applications takes a slightly different form from that in the Java language. Here is our catch statement in the Java language:

```
catch (FileNotFoundException exception)
```

And here is the NewJ C++ version:

```
catch (FileNotFoundExceptionR& exception)
```

In C++, exceptions should always be caught by reference; this saves an extra copy from being performed. But why is the “R” appended to the end of the class name in this statement? Due to the automatic object management features of NewJ, only references to objects and interface are ever thrown *not* the actual objects or interfaces. This is a very important point. This contract means that you can only throw references to objects or interfaces within NewJ application logic. (Note that throwing the result of `pi_new` takes care of this for you. Another good reason to use `pi_new`!) And this contract also means that you can only catch references to objects or interfaces with NewJ application logic.

The last difference may have been so subtle it escaped your notice. Did you see it at the end of this statement:

```
System::out_->println(PI_T("Unable to open file ") + fileName + PI_C('\\'));
```

Yes, the last thing that is appended is a single character, an apostrophe. In order to guarantee that character literals are properly treated as Unicode, you must put each character literal within `PI_C()`. It is recommended that you use `PI_C()` instead of `PI_T()` whenever you need to append a single character because `PI_C()` is much more performant.

## Step 5: Line Counting, Line Parsing, and Outputting Results

The next step is to add the rest of the application logic, the line counting, the line parsing, and outputting results. Let's refine `doGrep()` to have all of these features. Here's the Java language version:

```

private static void doGrep(String fileName, String searchPattern)
{
    try
        {
            RandomAccessFile file = new RandomAccessFile(fileName, "r");

            int lineNumber = 1;

            for (; ; ) // loop forever
                {
                    String line = file.readLine();

                    if (line == null)
                        {
                            break;
                        }

                    if (line.indexOf(searchPattern) != -1)
                        {
                            System.out.println(lineNumber + ": " + line);
                        }

                    lineNumber ++;
                }

            file.close();
        }
    catch (FileNotFoundException exception)
        {
            System.out.println("Unable to open file '" + fileName + "'");
        }
    catch (IOException exception)
        {
            System.out.println("Unexpected read error occurred");
        }
}

```

And here's the NewJ C++ version:

```

void JGrep::doGrep(::java::lang::StringR fileName, ::java::lang::StringR searchPattern)
{
    try
        {
            RandomAccessFileR file = pi_new(RandomAccessFile,
RandomAccessFile(fileName, PI_T("r")));

```

```

        pi_int lineNumber = 1;

        for (;) // loop forever
            {
                StringR line = file->readLine();

                if (line == pi_null)
                    {
                        break;
                    }

                if (line->indexOf(searchPattern) != -1)
                    {
                        System::out_->println(lineNumber + PI_T(": ") +
line);
                    }

                lineNumber ++;
            }

        file->close();
    }
    catch (FileNotFoundExceptionR& exception)
        {
            System::out_->println(PI_T("Unable to open file ") + fileName +
PI_C("\n"));
        }
    catch (IOException& exception)
        {
            System::out_->println(PI_T("Unexpected read error occurred"));
        }
    }
}

```

In this last refinement, only one new difference was introduced. Did you catch it? Let's take a look at it. Here's the statement in the Java language:

```
int lineNumber = 1;
```

And the statement in C++ using NewJ:

```
pi_int lineNumber = 1;
```

Why was `int` changed to `pi_int` even though `int` is a standard C++ type? The reason is that according to ANSI C++ the size of `int` is implementation defined, which means `int` can be different for different compilers and different platforms (and it often is). To guarantee type size consistency like the Java language does, the Pie Library defines its own set of compatible types that are prefixed with “`pi_`”. Make sure to always append “`pi_`” to Java language primitive types when they are used in NewJ applications. (The only exception to this rule is

the boolean type. You may prefer to use its natural equivalent in C++, the `bool` type, instead. In fact, the NewJ Library does this itself.)

## Step 6: Build, Specify Some Program Arguments, and Run

We're now ready to build our application for release. Do this do. (If you need help doing this, see the instructions at the end of Step 2.) After the application builds successfully, we need to specify some program arguments before we run. Select the menu item `Project > Settings...` When the Project Settings dialog appears, under "Settings For", select "Win32 Release". Click on the "Debug" tab. For "Program arguments", enter the file name you wish to search in followed by a space and the text you wish to find. Choose OK. Now we're ready to run. Press `Ctrl+F5` or the menu item `Build > Execute JGrep.exe`. The program will output all of the occurrences it found, if any, in the file you specified.

If you run "JGrep.exe" on "JGrep.cpp" with "JGrep" as the search pattern, you should get something like the following. Your line numbers may vary depending on exactly where you inserted the sections we refined.

```
1: // JGrep.cpp
5: #include "JGrep.h"
26: PIE_DEFINE_MAIN_CLASS(JGrep, "JGrep");
37: JGrep::JGrep()
42: JGrep::~JGrep()
50: void JGrep::main(::java::lang::StringAR args)
54:     System::out_>println(PI_T("usage: JGrep filename searchpattern"));
61: void JGrep::doGrep(::java::lang::StringR fileName, ::java::lang::StringR searchPattern)
```

The complete source code for the Java language and C++ versions of JGrep are included along with the NewJ Library. They are located in the directories `docs/guide/chapter7/java` and `docs/guide/chapter7/cpp`, respectively.

## The Next Step: Your Own Applications

Congratulations! You have successfully completed your first NewJ application. In the process, you have learned how to use the NewJ AppWizards, the Pie Run-time Library, and the Core J2 Library. You also saw how to adapt common Java language source code to C++ using the NewJ Library. You are now ready to begin creating your very own NewJ applications.