Java Native Interface (JNI)

- Das JNI erlaubt den Aufruf extern implementierter Funktionen. Damit ist ein Zugriff auf alle Ressourcen einer Maschine möglich. Der Aufruf erfolgt über die Java Virtual Machine (JVM).

- Diese Erweiterungen sind selbstverständlich nicht mehr plattformabhängig.

- Wo bringen externe Module Vorteile?
  - Zugriff auf systemspezifische Funktionen möglich.
  - Bereits bestehende Programme in einer anderen Sprache können schrittweise in Java übertragen werden.
  - Firmenspezifisches Know-how kann in externen Modulen geschützt werden.

- Auch umgekehrt ist ein Zugriff aus einer anderen Programmiersprache auf ein Java-Modul möglich. Dies erfolgt über die Java Invocation-API.
  Java selbst nutzt diese nativen Funktionen. Vor allem Methoden, die systemspezifische Implementierungen benötigen, z.B.:

```
/*
 * @(#)FileInputStream.java      1.58 02/02/06
 *
 * Copyright 2002 Sun Microsystems, Inc. All rights reserved.

 */

package java.io;

import java.nio.channels.FileChannel;
import sun.nio.ch.FileChannelImpl;
……………
…………..

/**
     * Opens the specified file for reading.
     * @param name the name of the file
     */
    private native void open(String name) throws FileNotFoundException;

    /**
     * Reads a byte of data from this input stream. This method blocks
     * if no input is yet available.
     *
     * @return     the next byte of data, or <code>-1</code> if the end of the
     *             file is reached.
     * @exception  IOException  if an I/O error occurs.
     */
    public native int read() throws IOException;
```

- Für den Aufruf einer extern definierten Funktion muss diese bei Windows in einer DLL vorliegen.

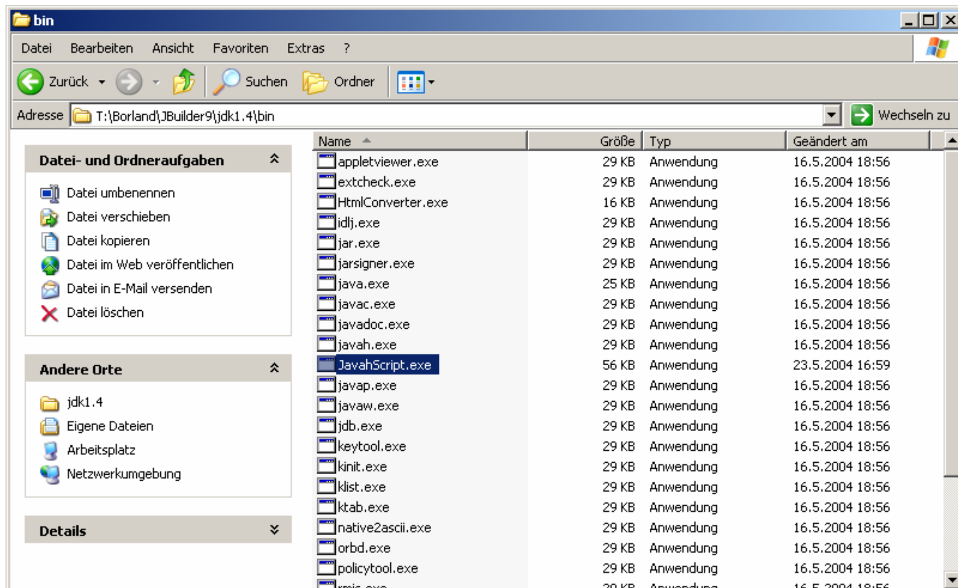
- Das Arbeiten mit JNI erscheint anfangs etwas kompliziert. Wenn man aber das Vorgehen kennt, ist es keine problematische Sache.

- Der Aufruf von Funktionen mit Parameter und Werterückgaben werden nächstes Mal behandelt.

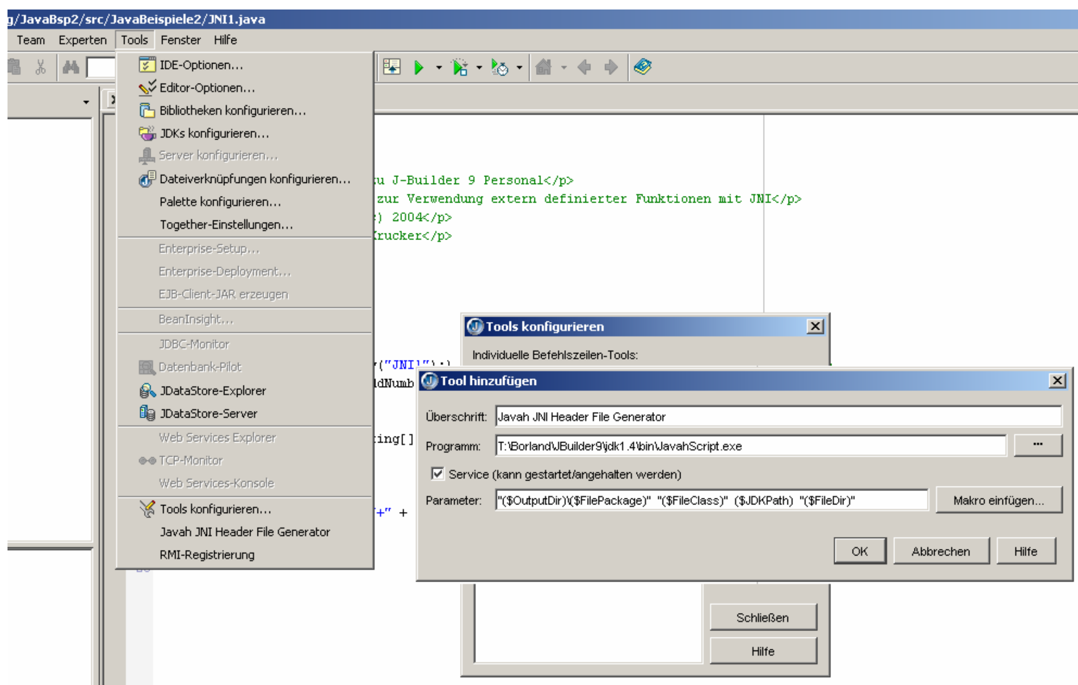
JNI mit Borland C++ Builder V5.0

Das Erstellen der DLL mit Borland C++ ist an sich problemlos, wenn einige Dinge beachtet werden:

1. Das Tool JavahScript.exe für die Headerfile-Erzeugung im JDK1.4\bin- Verzeichnis einbringen:



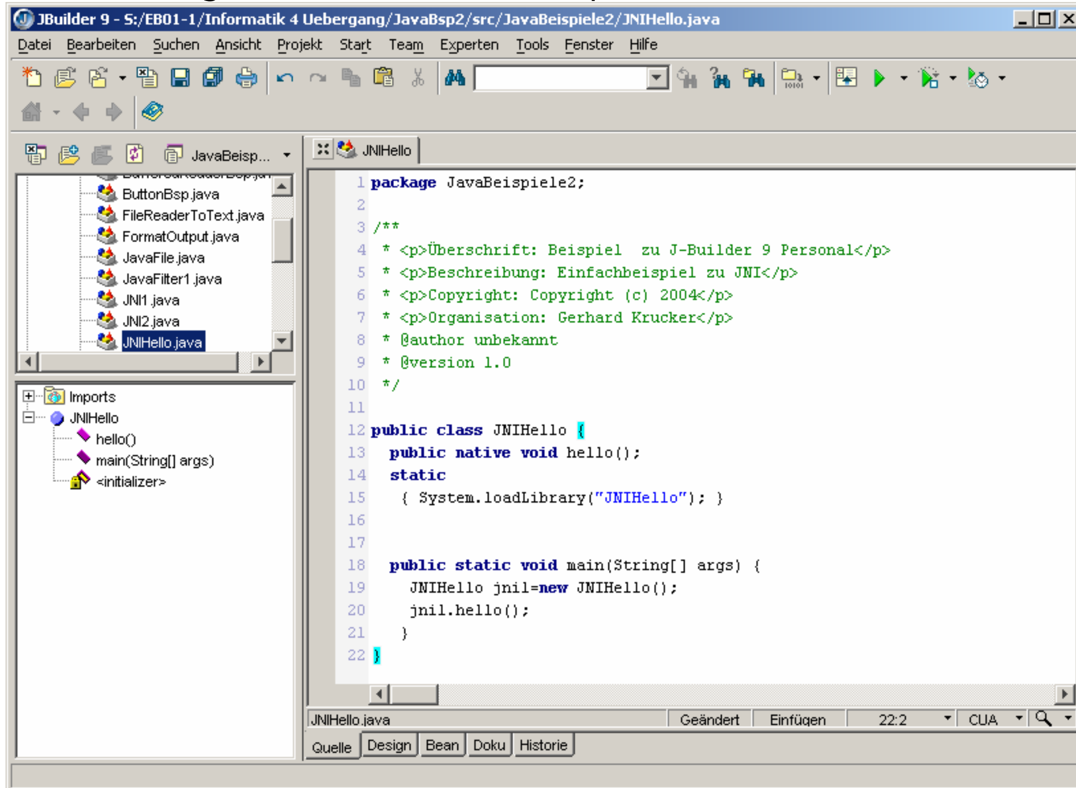
2. Nun wird das Tool im Tools-Menu installiert:



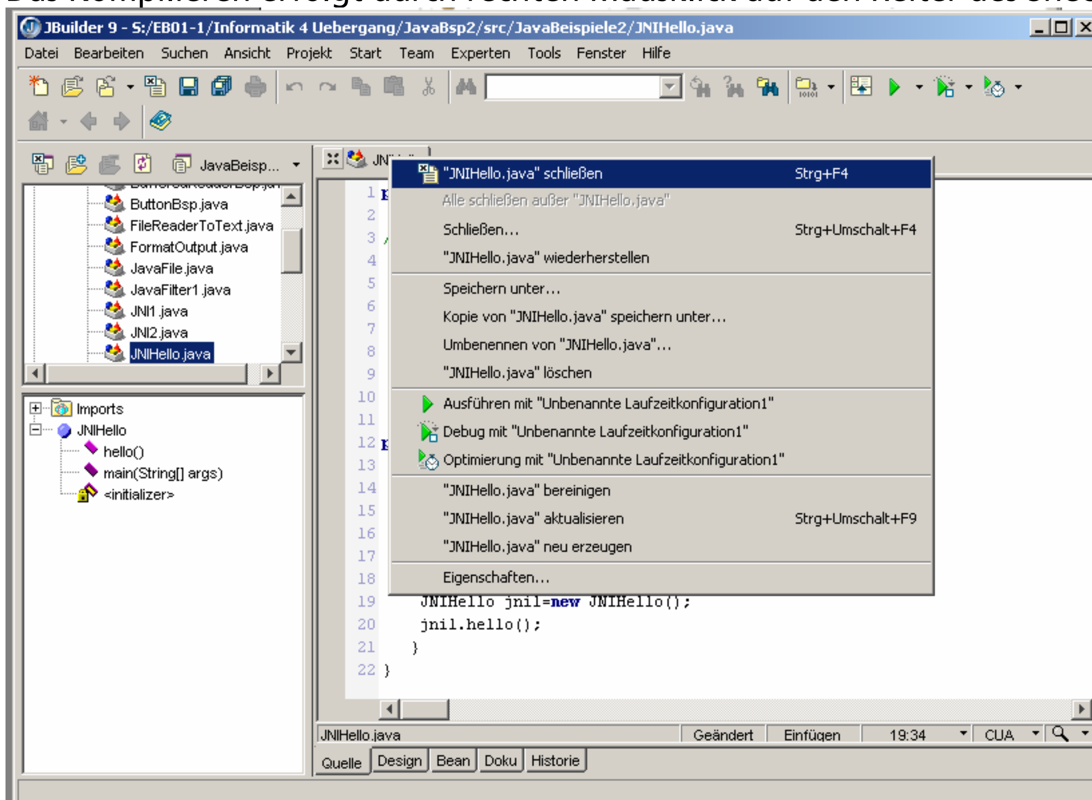
Der Parametereintrag muss genauso lauten: (Mit allen Leerschlägen)
`"($OutputDir)\($FilePackage)"  "($FileClass)"  ($JDKPath)  "($FileDir)"`

Jetzt wird der Eintrag als Tool im Menü sichtbar.

3. Das Java-Programm codieren und kompilieren. Das `.class`–File muss erzeugt werden.

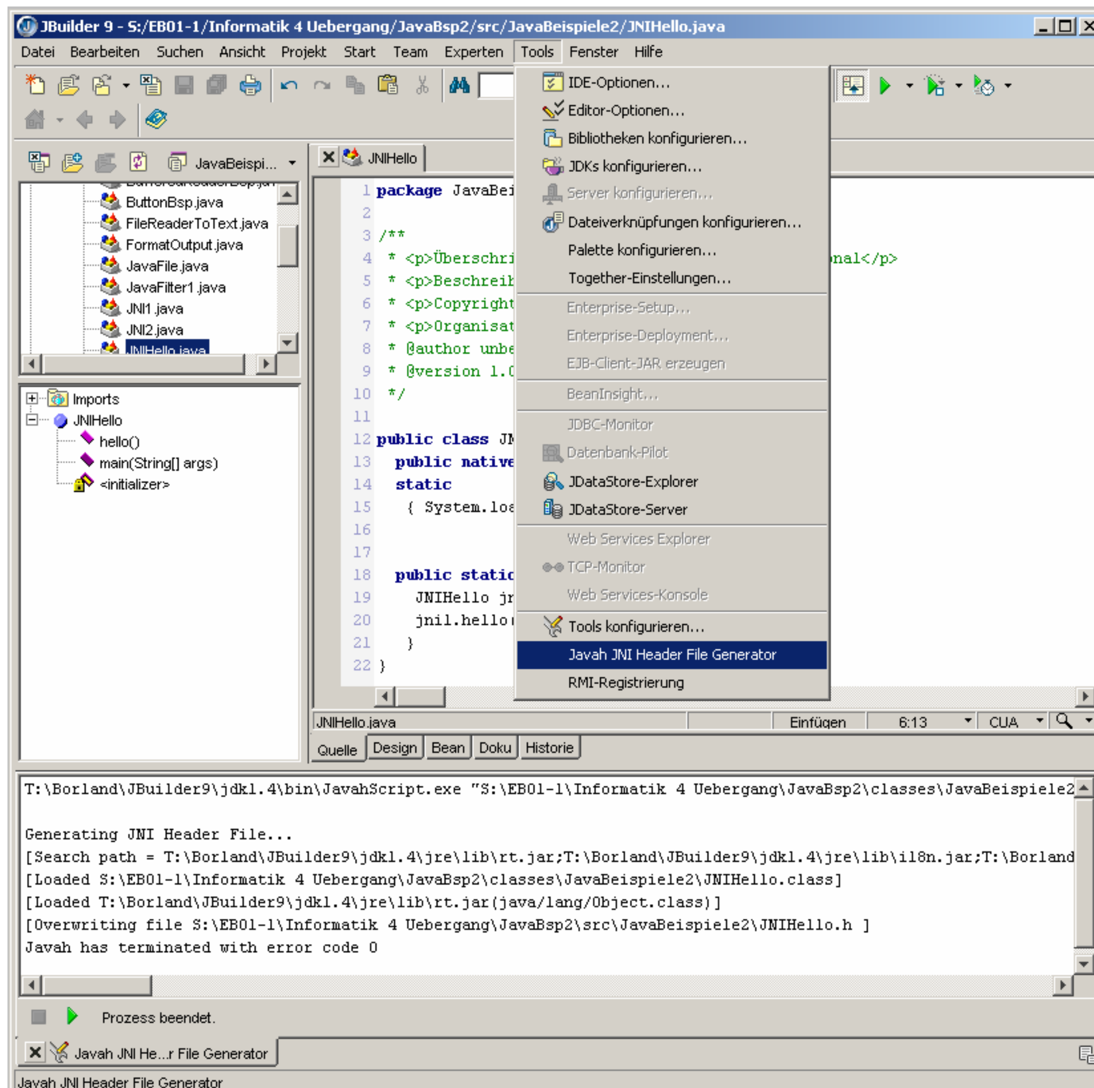


In der Klasse muss die zu verwendende externe Funktion als Methoden-Prototyp definiert werden. In einem `static`-Block wird die DLL mit `System.loadLibrary(..)` geladen. Hier wird der DLL-Name ohne Extension angegeben.
Das Kompilieren erfolgt durch rechten Mausklick auf den Reiter des Sheets:



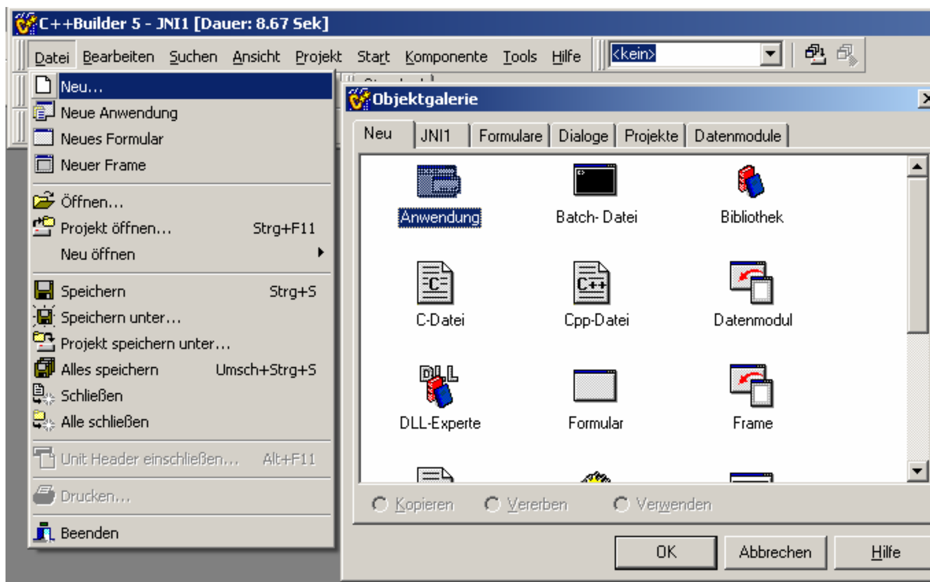
Nun ist das `.class`-File erzeugt worden.

4. Nun wird das Headerfile für das C-Modul erzeugt. Dazu im Tools-Menü das Tool anklicken. Im Message-Fenster wird der Erfolg angezeigt.

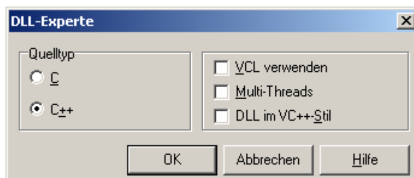


Die Arbeiten in der Java-Umgebung sind vorläufig abgeschlossen.

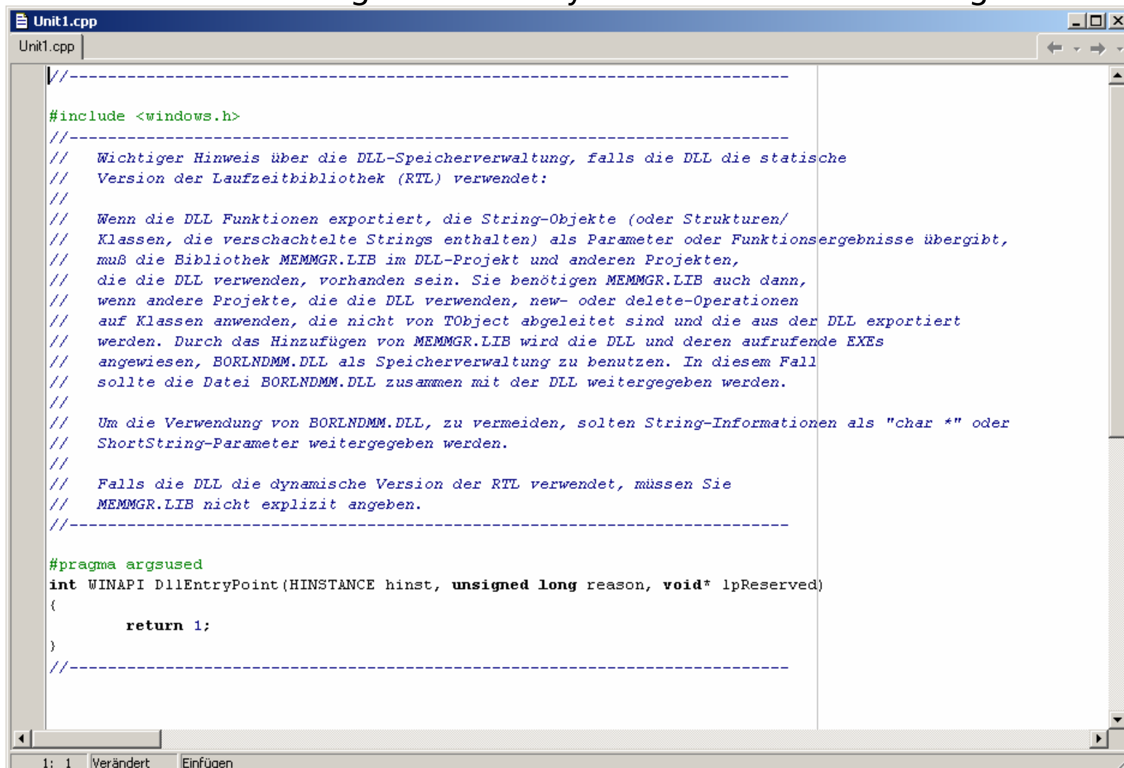
5. Borland C++ Builder aufstarten. Die DLL wird als Projekt mit DLL Experten erstellt:



Vorgaben für DLL wählen:

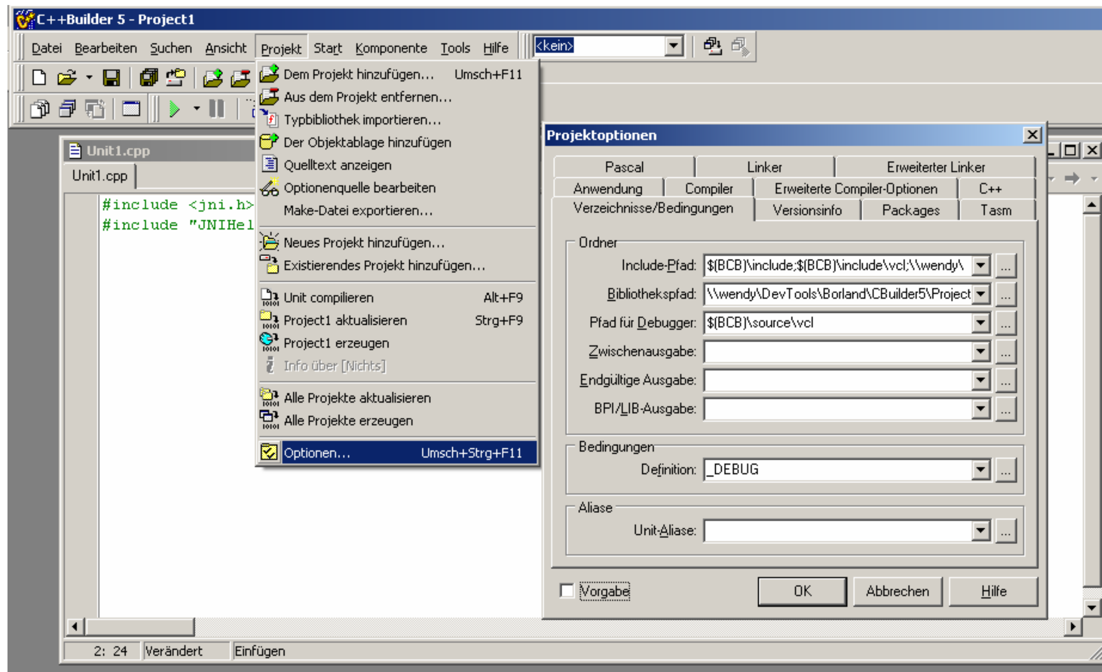


6. Es wird eine Unit erzeugt mit DLLEntryPoint und weiteren Einträgen:

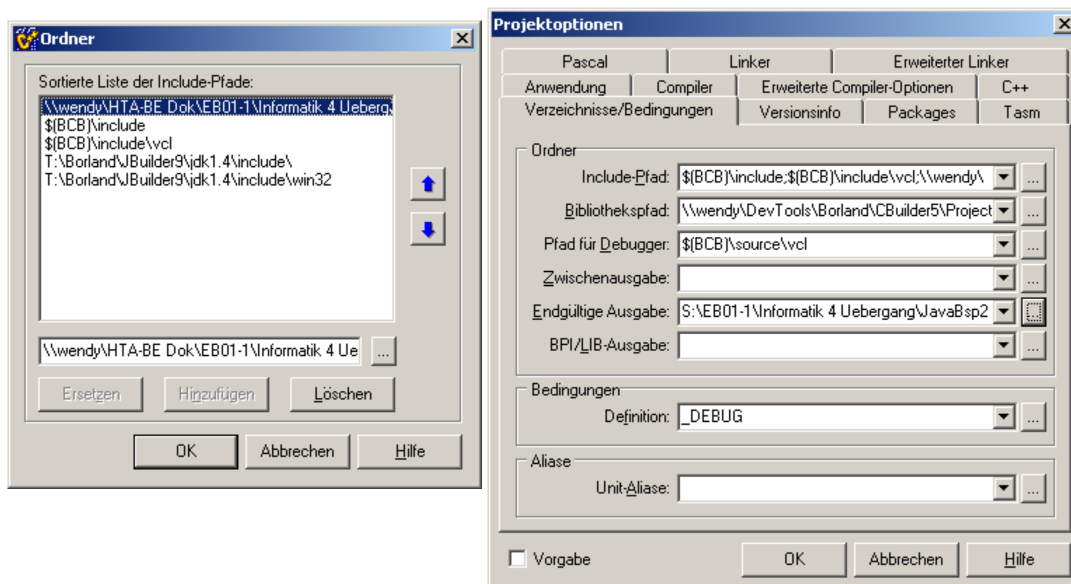


Der gesamte Inhalt wird gelöscht!

7. Die Include-Directory- Liste muss für `<JNI.h>` u.a. erweitert werden. Dies erfolgt über die Projektoptionen:

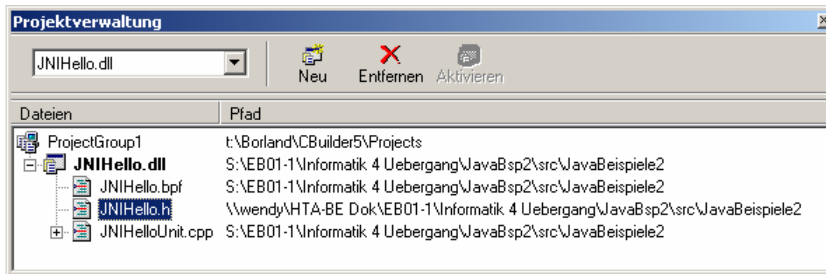


Der Include-Pfad muss mit den Einträgen zum Ziel `...jdk1.4\include` und `....jdk1.4\include\win32` ergänzt werden. Der Ausgabepfad wird so definiert, dass die DLL in das aktuelle Projektverzeichnis geschrieben wird. (Das Projektverzeichnis ist das Verzeichnis, das vor dem JPX-File des Packages steht)). Damit wird die DLL ohne weitere Vorkehrungen beim Programmlauf gefunden.

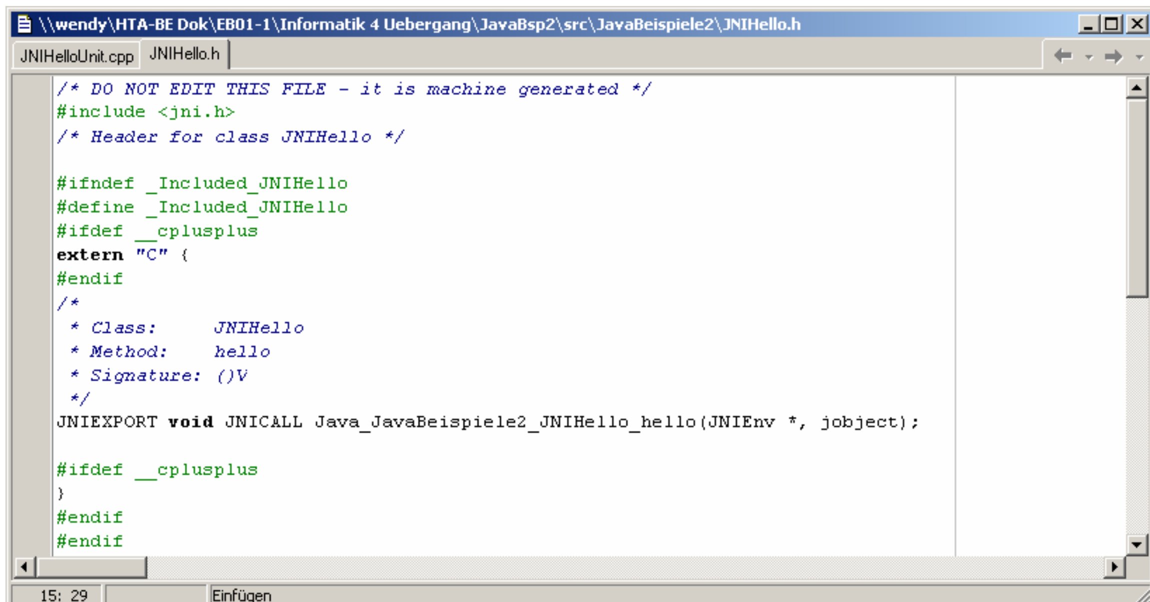


8. Das Projekt nun unter einem zweckmässigen Namen abspeichern. Der DLL-Name muss natürlich dem Parameter bei `System.loadLibrary(..)` entsprechen.

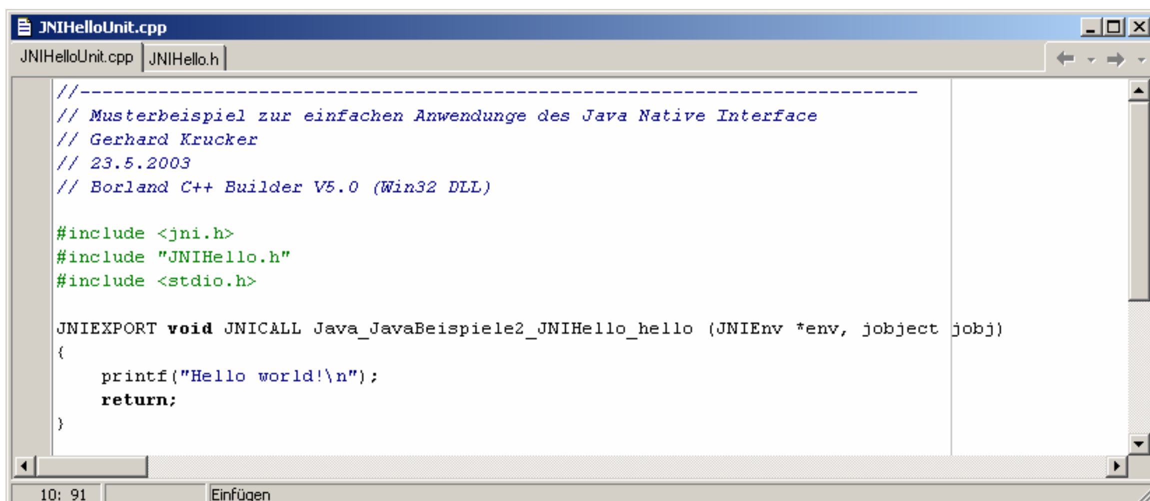
9. Die Projektverwaltung öffnen und das JNI-Headerfile zum Projekt zufügen. Hier: `JNIHello.h`. (Rechte Maustaste auf den DLL-Namen klicken – Hinzufügen..)



10. Das JNI-Headerfile öffnen und den Funktionsprototypen mit dem Paketnamen ergänzen. Hier `JavaBeispiele2` zwischen zwei Unterstreichungszeichen. Diese Ergänzung ist nicht dokumentiert, aber zumindest bei der Benutzung von C++ Builder 5.0 notwendig.

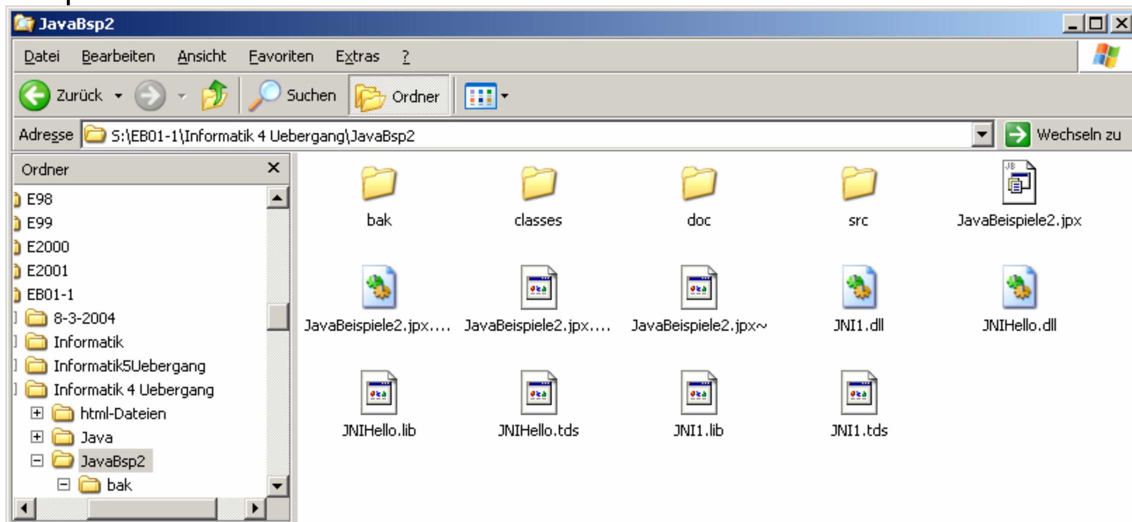


11. Den Code in das .CPP-File eintippen. Der Funktionskopf wird zweckmässigerweise aus dem JNI-Headerfile kopiert:



12. Nun das Projekt kompilieren (Erzeugen) und kontrollieren, dass die DLL wirklich in das Verzeichnis mit dem JPX-File geschrieben worden ist.

Beispiel:



13. Nun wieder zur JBuilder-IDE wechseln. Das Java File starten und prüfen, ob der Programmlauf korrekt erfolgt: