Exceptions, Pakete und Zugriffsmodifikatoren

Aufgabe 1: Exceptions

```
import java.awt.AWTException;

public class ExceptionTest {
     public static void main(String[] args) {
          System.out.println("Anfang main");
          ExceptionTest ex = new ExceptionTest();
          ex.run();
          System.out.println("Ende main");
     }

     void run() {
          System.out.println("Anfang run");
          try {
               System.out.println("Anfang try-Block");
               int index = -1;          // Test: -1 / 0 / 1 / 2

               final String[] args = { "OK", "NULL", null };
               String value = args[index];
               doRiskyStuff(value);
               System.out.println("Ende try-Block");
          }
          catch (IndexOutOfBoundsException ex) {
               System.out.println(
                         "catch IndexOutOfBoundsException");
          }
          catch (AWTException ex) {
               System.out.println("catch AWTException");
          }
          finally {
               System.out.println("finally");
          }
          System.out.println("Ende run");
     }

     void doRiskyStuff(String value) throws AWTException
     {
          System.out.println("Anfang doRiskyStuff");
          if (value.toLowerCase().equals("null")) {
               System.out.println("werfe Exception");
               throw new AWTException("simuliere AWT Fehler");
          }
          System.out.println("Ende doRiskyStuff");
     }
}
```

Analysieren Sie den Ablauf des vorstehenden Programms. Was wird jeweils ausgegeben?

a) Ausgabe für **index = -1** (in run):

```
Anfang main
Anfang run
Anfang try
catch
   IndexOutOfBoundsException
finally
Ende run
Ende main
```

b) Ausgabe für **index = 0**:

```
Anfang main
Anfang run
Anfang try-Block
Anfang doRiskyStuff
Ende doRiskyStuff
Ende try-Block
finally
Ende run
Ende main
```

c) Ausgabe für **index = 1**:

```
Anfang main
Anfang run
Anfang try-Block
Anfang doRiskyStuff
werfe Exception
catch AWTException
finally
Ende run
Ende main
```

d) Ausgabe für **index = 2**:

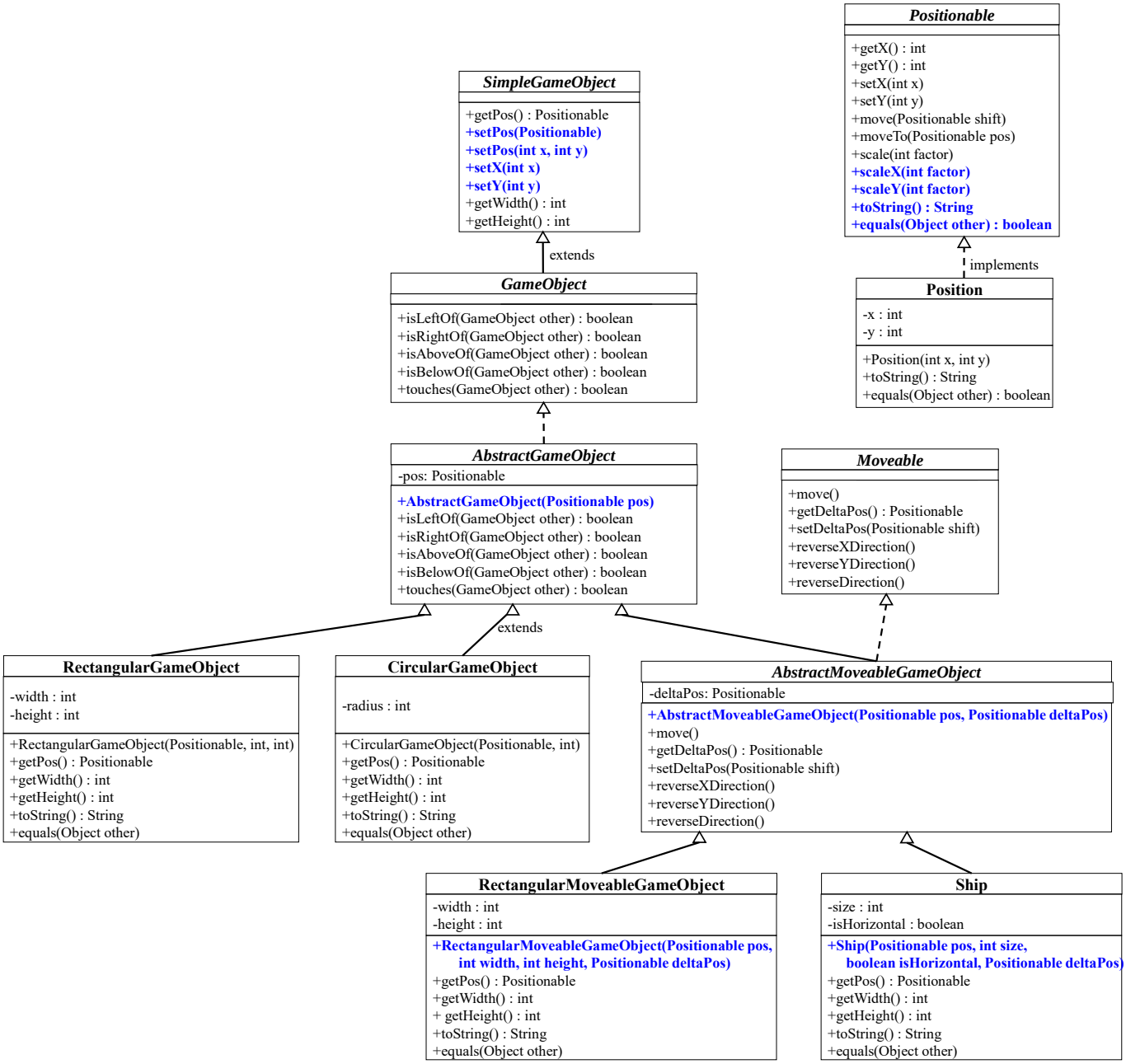
```
Anfang main
Anfang run
Anfang try-Block
Anfang doRiskyStuff
finally
Exception in thread "main"
java.lang.NullPointerException
   at ExceptionTest.doRiskyStuff(
           ExceptionTest.java:40)
   at ExceptionTest.run(
           ExceptionTest.java:20)
   at ExceptionTest.main(
           ExceptionTest.java:8)
```

Aufgabe 2: Spielobjekte

Betrachten Sie das Ergebnis und die Klassenhierarchie der in Übung 4 angelegten Interfaces und Klassen für Spielobjekte. Dieses nehmen wir in den nachfolgenden Übungen als Grundlage für die weiteren Implementierungen.

Falls Sie noch nicht alles implementiert haben, sollten Sie es vervollständigen.

Frage: Könnte/Sollte man die Klassen **RectangularGameObject** bzw. **CircularGameObject** ggf. anders in der Klassenhierarchie aufhängen?



Antwort:

- Beide unter **AbstractMoveableGameobject** (und RectangularGameObject ist Spezialfall von RectangularMoveableGameObject, ebenso evtl. Ship).

- Wir können auch alle GameObject-Objekte potentiell beweglich machen. Dazu erbt *GameObject* von Interface *Moveable*, und AbstractMoveableGameObject ist dann die eigentliche AbstractGameObject-Klasse.
  - Vorteil: Alle Spielobjekte können sich bei Bedarf bewegen. (Ist in AbstractMoveableGameObject bereits generisch implementiert.)
  - Einziger (Mini-)Nachteil: Zusätzliches Argument im Konstruktor: deltaPos ist jeweils anzugeben.
  - Lösung dafür: 2. Konstruktor, der deltaPos = 0 setzt (kann später mit setDeltaPos() gesetzt werden.)

Aufgabe 3: Statische Methoden und Singleton-Pattern

Um zu erreichen, dass für eine bestimmte Klasse maximal ein Objekt existieren kann, lässt sich das Entwurfsmuster **Singleton** anwenden. Dies wollen wir am Beispiel eines Loggers zur Ausgabe von Fehlermeldungen realisieren.

a) Implementieren Sie dazu – im Paket **games.basic.logging** – die Klasse ***Logger*** als Singleton-Pattern wie folgt:

- Merken Sie sich die einzige Objektinstanz der Loggerklasse in einen – von außen nicht sichtbaren Attribut **instance**.
- Stellen Sie eine **statische** Methode **Logger getInstance()** bereit, die ein (bzw. das einzige) Objekt der Klasse Logger zurückliefert. Sollte das Objekt noch nicht existieren, erzeugen Sie es hier („lazy object creation“).
- Machen Sie den Konstruktor **private**, so dass von außerhalb der Klasse keine Objekte erzeugbar sind.

b) Anschließend ergänzen Sie die Klasse um weitere Attribute und Methoden. In unseren Fall benötigen wir lediglich eine Methode

- **void log(String message),** welches die Nachricht auf dem Bildschirm bzw. (später) in eine Log-Datei ausgibt.
- **Optional**: Ergänzen Sie dies um eine Methode **void log(Exception ex)**, welches eine Exception, d.h. deren Meldung (→ Methode ex.**getMessage**()) sowie den Stacktrace (→ Methode ex.**printStackTrace**()) auf dem Bildschirm ausgibt.

c) **Zusatzaufgabe:** Falls Sie noch Lust haben, können Sie die **log**-Methode dahingehend erweitern (oder eine zusätzliche **log**-Methode erstellen), dass die Nachricht in eine Datei geschrieben wird.

Hinweise:

- Um in eine Textdatei zu schreiben, können Sie die Datei mittels
  BufferedWriter os = new BufferedWriter(new FileWriter(dateiName));
  öffnen.
- Die entsprechenden Klassen sind im Paket **java.io** definiert.
- Das Schreiben können Sie dann mit os.**write**("String"); erledigen.
- Vergessen Sie nicht, die Datei am Ende zu schließen.

d) **Zusatzaufgabe:** Geben Sie bei den log-Ausgaben jeweils das aktuelle Datum und Uhrzeit mit aus.

Hinweis:

- Die aktuelle Zeit lässt ermitteln sich mit

```
Calendar.getInstance(TimeZone.getTimeZone("Europe/Berlin"))
```

- Weitere Informationen zum Auslesen der Zeit finden Sie u.a. in der Java-API http://docs.oracle.com/javase/7/docs/api/index.html?java/util/Calendar.html .

games.basic.logging.Logger.java:

```
package games.basic.logging;

import java.text.DateFormat;
import java.text.SimpleDateFormat;
import java.util.Calendar;
import java.util.Date;
import java.util.Locale;
import java.util.TimeZone;

public class Logger {

	private static Logger instance = null;

	private Logger() {
	}

	public static synchronized Logger getInstance() {
		if (instance == null) {
			instance = new Logger();
		}

		return instance;
	}

	public void log(String message) {
		Calendar gmtCal = Calendar.getInstance(TimeZone.getTimeZone("GMT"));
//		Date date = new Date();
		Date date = gmtCal.getTime();
		String timeStamp;
```

```
                timeStamp = date.toString();
                System.out.println( timeStamp + ": " + message );

//              timeStamp = (new SimpleDateFormat()).format( date );
//              System.out.println( timeStamp + ": " + message );

                System.out.printf( "%tF %tT %s %n",  gmtCal, gmtCal, message);
                // 2010-05-04 09:52:17 message          (NICHT GMT!)
        }
}
```

games.basic.logging.LogTest.java:

```
package games.basic.logging;

public class LogTest {

      public static void main(String[] args) {
            Logger logger = Logger.getInstance();

            logger.log("Message 1");

            int sum = 0;
            for (int i = 0; i < Integer.MAX_VALUE; i++) {
                  sum += i;
            }

            logger.log("Message 2: sum = " + sum);

      }
}
```

Ausgabe:

```
Thu Apr 28 02:01:28 CEST 2011: Message 1
2011-04-28 00:01:28 Message 1
Thu Apr 28 02:01:31 CEST 2011: Message 2: sum = 1073741825
2011-04-28 00:01:31 Message 2: sum = 1073741825
```

Zusatz-Aufgabe 4 (für Java-Experten): Java-Rätsel: Elvis lebt

Was gibt das folgende Java-Programm aus?

```
import java.util.Calendar;
import java.util.TimeZone;

public class Elvis {

	public static final Elvis INSTANZ = new Elvis();
	private final int alter;
	private static final int AKTUELLES_JAHR =
	    Calendar.getInstance(TimeZone.getTimeZone("Europe/Berlin"))
	            .get(Calendar.YEAR);

	private Elvis() {
		alter = AKTUELLES_JAHR - 1935;
	}

	public int getAlter() {
		return alter;
	}

	public static void main(String[] args) {
		System.out.println("Elvis wäre jetzt " +
		                   INSTANZ.getAlter() + " Jahre alt.");
	}
}
```

Ausgabe:

```
Elvis wäre jetzt -1935 Jahre alt.
```

Grund:

- Falsche Reihenfolge der Initialisierung (bzw. zyklische Initialisierungs-Abhängigkeiten)
  1. INSTANZ = new Elvis() → Konstruktor-Aufruf
  2. Konstruktor Elvis():
     - alter = AKTUELLES_JAHR – 1930;
     - Aber: AKTUELLES_JAHR ist noch nicht gesetzt (hat z.Z. noch Java-Default-Wert 0) → alter = 0 – 1930 = -1930
  3. Setze AKTUELLES_JAHR = Calender.getInstance(…)…
     - Zu spät: Jetzt ist alter bereits gesetzt, aktueller Wert von AKTUELLES_JAHR wird nicht verwendet.
- Lösung:
  1. Reihenfolge vertauschen: AKTUELLES_JAHR = … vor INSTANZ = new Elvis();
  2. Oder: Statt Konstante AKTUELLES_JAHR eine Funktion getAktuellesJahr() verwenden → wird dann im Konstruktor aufgerufen

Zusatz-Aufgabe 5: Schiffe versenken

Erweitern Sie Ihr Schiffe versenken Programm um einen „Automatik“-Modus. Ihr Programm soll also selbständig Spielzüge tätigen, d.h. selbst versuchen, die Schiffe zu versenken.

a) In der ersten Version können Sie einfach ein Spielfeld raten, d.h. zufällig wählen.

   Zum Testen können Sie entweder das Spielfeld (d.h. die Lage des oder der Schiffe) vom Nutzer wählen lassen, oder sich jeweils das Spielfeld auf dem Bildschirm anzeigen lassen, um den Spielverlauf zu verfolgen.

b) Fallen Ihnen noch bessere Algorithmen ein, um mit möglichst wenigen Spielzügen alle Schiffe zu versenken?

Zusatz-Aufgabe 6: Java Native Interface

Zum Aufrufen von Methoden, die nicht in Java implementiert sind, dient das Java Native Interface (JNI). Solche Methoden werden in Java als **native** deklariert. Den Aufruf solcher Methoden können Sie hier einmal bei Interesse ausprobieren:

a) Erstellen Sie dazu eine Klasse **StrLen** im Paket **de.hsnr.java.jni** mit

- der Methode **public static native int strlen(String s),** die keine Implementierung in Java, d.h. keinen Rumpf besitzt, sowie
- einer statischen Klasseninitialisierung zum Laden der zugehörigen Implementierung:

```
static {
            System.loadLibrary( "JNI_Strlen" );
}
```

b) Eine dazu passenden Implementierung der Methode **strlen** finden Sie in Moodle (https://moodle.hsnr.de/course/view.php?id=4399#section-3) unter Kurs Java 1 → Übung 5 → JNI_Strlen_XXX.dll. [1]
(Früher unter: http://lionel.kr.hs-niederrhein.de/~nitsche/java-info/uebung05/JNI_Strlen.dll).

Laden Sie sich (für Windows-Systeme) diese DLL-Library herunter und speichern Sie sich auf Ihrem Rechner. Die DLL muss sich in ihrem (Library)-Pfad (LD_LIBRARY_PATH bzw. PATH) befinden, damit Sie ein Java-Programm mit der Klasse StrLen ausführen können.
In Eclipse können Sie den Pfad auch direkt setzen: Properties → Java Build Path → Libraries → Add Class Folder (und dann denn Ordner einfügen, und dort noch einmal bei Native Library Location angeben.)

c) Rufen Sie die Methode **de.hsnr.java.jni.StrLen.strlen** in einem Haupt- bzw. Testprogramm auf.

Hinweis: Wenn Sie mehr erfahren wollen, oder diese oder andere native Methoden selbst implementieren wollen, finden Sie dazu bei Interesse weitere Hinweise im Openbook **„Java ist auch eine Insel“**, **8. Auflage**[2], Kapitel 27.2 – 27.3 (siehe z.B. http://www.iks.hs-merseburg.de/~uschroet/Literatur/Java_Lit/JAVA_Insel/javainsel_27_002.htm).

[1] Dort finden Sie die Dateien JNI_StrLen_32bit.dll sowie JNI_StrLen_64bit.dll.
[2] Leider ist das Kapitel über Java Native Interfaces (JNI) in der aktuellen 10. Auflage nicht online verfügbar. Das Buch wurde in 2 Teile (Java-Einstieg : „Java ist eine Insel“ und Fortgeschrittene Themen „Java 7 – Mehr als eine

Ausgabe:

```
StrLenTest
s[0] = Hallo
de.hsnr.java.jni.StrLen: try loading Library 'strlen'.
de.hsnr.java.jni.StrLen: loaded Library 'strlen'.
   -> strlen = 5
s[1] = s2
   -> strlen = 2
s[2] = |s3| + ? = 13
   -> strlen = 13
Hallo Java-Freunde:!
Hallo Java-Freunde:!
Hallo Java-Freunde:!
```

Setzen des Library-Pfades:

1. LD_LIBRARY_PATH (bzw. PATH) Variable setzen

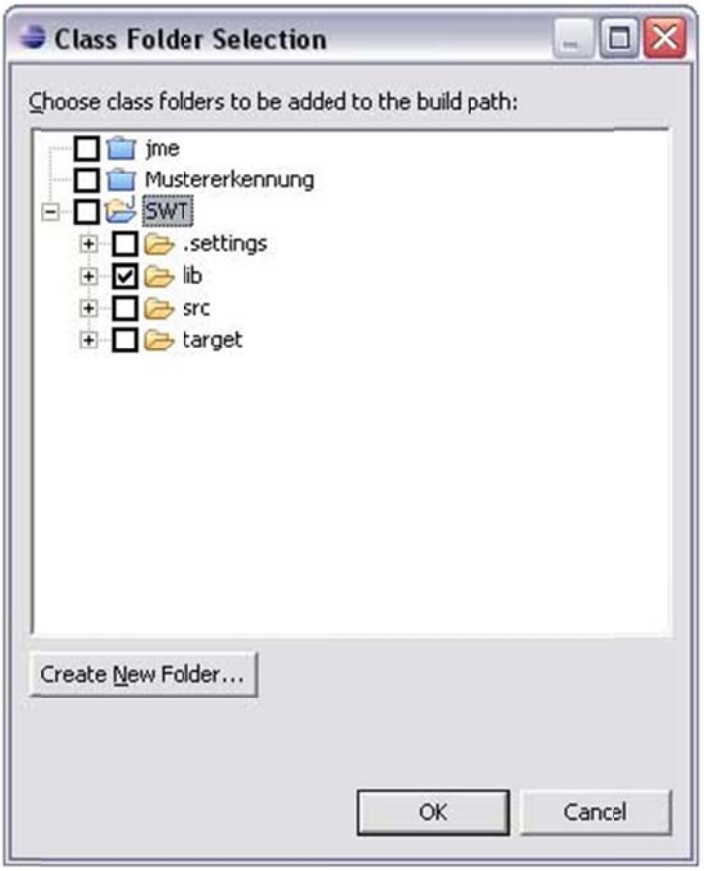
2. Beim Aufruf von Java:

   java -Djava.library.path=Laufwerk:\Pfad -jar Laufwerk:\Pfad\start.jar

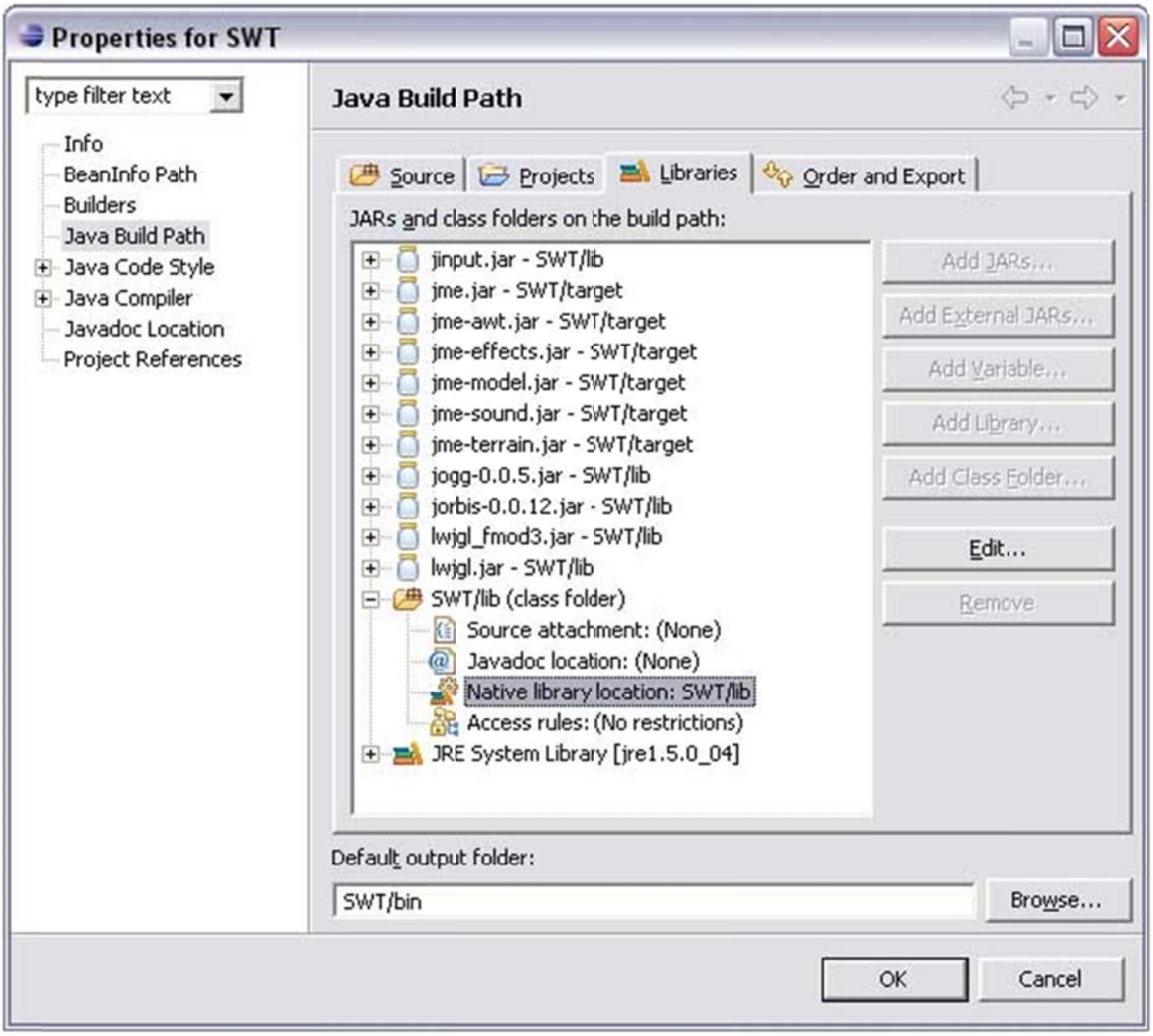
3. Direkt in Eclipse:

   - Weiter gehts mit den `.dll` Files, den *Native Libraries*. Dazu klickt man auf *Add Class Folder …* und wählt den `libs` Ordner aus.

Insel“) aufgeteilt, von denen nur der erste Teil online verfügbar frei ist. Links auf das Original-Buch der 8. Auflage wie
http://openbook.galileocomputing.de/javainsel8/javainsel_27_002.htm
werden leider vom Verlag auf die aktuelle Auflage weitergeleitet und verschwinden deshalb im „Nirwana“.



- Jetzt hat man es schon fast geschafft. Wie im Bild unten zu sehen ist muss man dem Compiler noch beibringen dass der eben ausgewählte Ordner noch die *Native Libraries* beinhaltet. Hat man das durch Doppelklick auf die "Native Library" Eintrag getan kann man mit dem Button *Workspace Folder* den `libs` Ordner nochmals auswählen und hat jetzt auch die *Native Libraries* verknüpft. Alles in allem sieht das in den Projekt-Properties dann so aus:


Properties for SWT
type filter text
Info
BeanInfo Path
Builders
Java Build Path
Java Code Style
Java Compiler
Javadoc Location
Project References
Java Build Path
Source
Projects
Libraries
Order and Export
JARs and class folders on the build path:
jinput.jar - SWT/lib
jme.jar - SWT/target
jme-awt.jar - SWT/target
jme-effects.jar - SWT/target
jme-model.jar - SWT/target
jme-sound.jar - SWT/target
jme-terrain.jar - SWT/target
jogg-0.0.5.jar - SWT/lib
jorbis-0.0.12.jar - SWT/lib
lwjgl_fmod3.jar - SWT/lib
lwjgl.jar - SWT/lib
SWT/lib (class folder)
Source attachment: (None)
Javadoc location: (None)
Native library location: SWT/lib
Access rules: (No restrictions)
JRE System Library [jre1.5.0_04]
Add JARs...
Add External JARs...
Add Variable...
Add Library...
Add Class Folder...
Edit...
Remove
Default output folder:
SWT/bin
Browse...
OK
Cancel

3