E1 Java SDK und Eclipse

Um in einer Programmiersprache Anwendungen zu erstellen und auszuführen, werden verschiedene Werkzeuge (Compiler, Laufzeitumgebung, Debugger etc.) benötigt. Für Java wird von SUN (aber auch von anderen Herstellern) eine solche "Werkzeugsammlung" als Java SDK (Software Development Kit) kostenlos angeboten.

Grundsätzlich ist die Entwicklung und Ausführung von Java Programmen allein mit einem solchen SDK möglich. In der Praxis werden jedoch integrierte Entwicklungsumgebungen eingesetzt, die auf den SDKs aufbauen und deren lose Sammlung von Werkzeugen unter einer einheitlichen, bedienerfreundlichen Oberfläche vereinen.

Unter der Vielzahl der Java IDEs (Integrated Development Environment) stellen wir das Eclipse SDK vor, das als Bestandteil des Eclipse Toolkit als Open Source Lizenz frei verfügbar ist. Nach der Installation von Eclipse unter Microsoft-Windows und Linux werden das Anlegen von Projekten und die elementaren Arbeitsschritte der Programmerstellung (Edieren von Quelltext, Übersetzen, Ausführen, Debuggen) beschrieben.

Im Rahmen einer Programmiereinführung können hier nur die einfachsten Funktionen des Eclipse SDKs dargestellt werden; über die weiteren Features dieses IDEs sowie die vielfältigen Einsatzmöglichkeiten von Eclipse sollten sich interessierte Leserinnen anhand der angegebenen Referenzen informieren.

E1.1 Installation der Tools

Im folgenden werden die Voraussetzungen und benötigten Anforderungen für die Installation und den Einsatz von Eclipse 3.0 für die im studentischen Umfeld am häufigsten eingesetzten Betriebssysteme Microsoft Windows und Linux beschrieben. Falls Eclipse auf anderen Plattformen betrieben werden soll, informieren Sie sich auf der angegebenen Download Seite von Eclipse.

E1.1.1 Voraussetzungen

Eclipse unterliegt der CPL (Common Public License), einer Open Source Lizenz, die jeder Nutzerin das Recht gibt, das Produkt kostenlos zu nutzen, es zu vervielfältigen und zu verändern.

Die folgenden Angaben beziehen sich auf das Eclipse SDK 3.1. Sollten andere Versionen zum Einsatz kommen, müssen die Anforderungen - insbesondere hinsichtlich Java SDK - überprüft und gegebenenfalls angepasst werden:

- Prozessor mit Intel x86 Architektur, mind. 660 MHz
- Betriebssystem Microsoft Windows (XP, 98, ME, NT, 2000, Server 2003) oder Linux (Red Hat Enterprise Linux WS3, SuSE Linux 8.2)

- 256 MB Hauptspeicher, besser mehr
- mind. 300 MB Plattenplatz
- Sun Java 2 SDK, Standard Edition, Version 1.5
  alternativ: IBM 32-bit SDK, Java 2 Technology Edition, Version 1.5

Um Eclipse als Anwendung starten zu können, wird bereits ein installiertes Java *Runtime* Environment benötigt. Auch wenn für die Java-Entwicklung unter Eclipse kein Java *Development* Environment zwingend notwendig ist, wird die Installation eines SDK (Software *Development* Kit) empfohlen.

Detailinformationen über eine bereits installierte Java Umgebung erhält man, indem in der Konsole (Windows: *Start* | *Ausführen* | `cmd`) der Befehl **`java -version`** eingegeben wird:

```
                              - Konsole -
C:\Home\Benutzerin\Dateien> java -version
java version "1.5.0_02"
Java(TM) 2 Runtime Environment, Standard Edition, (build 1.5.0...)
Java HotSpot(TM) Client VM (build 1.5.0_02.b09, mixed mode,  ...)

C:\Home\Benutzerin\Dateien>
```

Wird der Befehl nicht gefunden, sollten Sie unter Windows im Explorer nach der Datei `java.exe` suchen und diese dann ausführen. Unter Linux geben Sie zum Suchen den Befehl **`find / -name javac 2>/dev/null`** ein. Gefundene Dateien werden mit vollständigem Pfad in der Konsole angezeigt (hier ist es besser, nach dem Programm `javac` zu suchen, da die Suche nach `java` zu viele unbrauchbare Ergebnisse liefert; der Versionstest ist dann natürlich mit dem Programm `java` des angezeigten Verzeichnisses durchzuführen).

E1.1.2 Installation eines Java SDK

Falls ein geeignetes Java SDK installiert ist, überspringen Sie diesen Abschnitt.

Wir empfehlen die Installation des SUN J2SE SDK, das man unter der folgenden Adresse findet:

`java.sun.com/j2se`

Dort ist die gewünschte J2SE auszuwählen, wobei zu beachten ist, dass die Java Version 1.5 in der Edition *J2SE 5.0* enthalten ist. Klicken Sie auf die entsprechende J2SE, dann auf *Downloads* und schließlich auf *Download J2SE SDK*. Nach Bestätigen einer Lizenzvereinbarung kann die Software für das gewünschte Betriebssystem ausgewählt werden. Anschließend geben Sie für die Datei *j2sdk*windows*.exe* bzw. *j2sdk*linux*.rpm.bin* ein Verzeichnis zum Sichern an. Während des Downloads, der auch bei schnellen Verbindungen einige Minuten in Anspruch nehmen kann, sollten

die betriebssystemspezifischen Installationshinweise (*Installation Instructions*) gelesen werden.

Nach erfolgtem Download kann die eigentliche Installation, wie in den Hinweisen beschrieben, durchgeführt werden (betriebssystemspezifisches Ausführen der oben erwähnten Installationsdatei).

Nach der Installation und einer eventuell notwendigen Erweiterung der Umgebungsvariablen PATH um das *bin*-Verzeichnis des SDK sollte der oben beschriebene Versionstest mit **`java -version`** erfolgreich durchführbar sein. Falls mehrere SDKs installiert sind, stellen Sie sicher, dass das *bin*-Verzeichnis des gewünschten SDKs vor allen anderen *bin*-Verzeichnissen in der PATH-Variablen aufgeführt ist.

Unter der oben angegebenen Adresse stehen weitere Informationen über Java sowohl Online Ansicht als auch zum Download bereit, etwa die API-Spezifikation, Code Beispiele und Tutorials, aber auch der Java Quellcode selbst.

E1.1.3 Erstes Java Programm im Java SDK

Grundsätzlich ist es möglich, Java Programme allein im Java SDK zu entwickeln und auszuführen. Um dies zu verifizieren, erstellen wir mit einem beliebigen Editor (Notepad, Emacs, vi) eine Textdatei namens `First.java` mit folgendem Inhalt:

Editor: First.java

```
public class First {
   public static void main( String[] args ){
      System.out.println( "Erstes Programm" );
   }
}
```

Diese Quelldatei wird in einem Konsolenfenster mit `javac` übersetzt :

- Konsole -

```
C:\Home\Benutzerin\Dateien> javac First.java

C:\Home\Benutzerin\Dateien>
```

Anschließend sollte sich im gleichen Verzeichnis eine Datei namens `First.class` befinden. Im Fall einer Fehlermeldung ist sicher zu stellen, dass sich die Datei `First.java` im Ausführungsverzeichnis der Konsole befindet (im dargestellten Beispiel in `c:\Home\Benutzerin\Dateien`). Möglicherweise wurde beim Dateinamen Großkleinschreibung missachtet oder aufgrund eines Tippfehlers im Quelltext ein Syntaxfehler erzeugt. Nach Korrektur des Fehlers kann der Übersetzungsvorgang mit dem gleichen Kommando nochmals angestoßen werden.

Natürlich können Sie auch im Quelltext einen Syntaxfehler provozieren, etwa das Semikolon am Ende der dritten Zeile vergessen, und beobachten, wie der Compiler darauf reagiert.

Um schließlich die erzeugte Datei `First.class` innerhalb der virtuellen Java Maschine auszuführen, verwendet man das Kommando `java` folgendermaßen :

```
                              - Konsole -
C:\Home\Benutzerin\Dateien> java First
Erstes Programm


C:\Home\Benutzerin\Dateien>
```

Das Programm wird ausgeführt und der Text `Erstes Programm` in einer eigenen Zeile im Konsolenfenster ausgegeben (unter Microsoft Windows erscheint zusätzlich eine Leerzeile).

Zusätzliche Informationen zu den JDK-Tools `java` und `javac` finden sich in E1.5.

E1.1.4 Installation des Eclipse SDK

Umfangreiche Informationen zum Eclipse Projekt befindet sich auf der Seite

`www.eclipse.org`

Von dort gelangt man via *downloads* zur Download-Seite der letzten freigegebenen SDK-Version (z.B. 3.1). Wählen Sie das SDK für Ihr Betriebssystem aus (bei Linux wird noch nach den verwendeten GUI Toolkits *motif* oder *GTK* unterschieden) und klicken Sie anschließend auf einen Standort in Ihrer Nähe. Der Download startet automatisch und fordert die Benutzerin zur Eingabe eines lokalen Download-Verzeichnisses auf.

Die Installation selbst besteht einfach darin, die komprimierte Datei aus dem Downloadverzeichnis in ein Zielverzeichnis zu entpacken, das Sie frei wählen können. Unter Windows etwa `c:\programme`, unter Linux das Verzeichnis `/opt`. Da im Komprimat bereits ein Verzeichnis `eclipse` enthalten ist, befindet sich die Eclipse-Installation anschließend unter `c:\programme\eclipse` bzw. unter `/opt/eclipse`.

Auf die übliche Art und Weise lassen sich zum Schnellstart von Eclipse eine Verknüpfung auf dem Desktop anlegen (Windows: *Drag-and-Drop* des Eclipse-Icon mit der rechten Maus auf den Desktop, dann *Verknüpfung hier erstellen* wählen; Linux: *Drag-and-Drop* des Eclipse-Icons auf den Desktop, dann *Link here* wählen).

E1.2 Starten von Eclipse

Eclipse wird durch Doppelklick auf das Eclipse-Icon oder durch Eingabe des Kommandos `eclipse` in der Konsole gestartet. Nach dem Eclipse *Splashscreen* erscheint der folgende Dialog, in dem der Workspace, das Verzeichnis für die einzelnen Projektdateien, konfiguriert werden kann. Bestätigen Sie das Fenster mit *OK*.

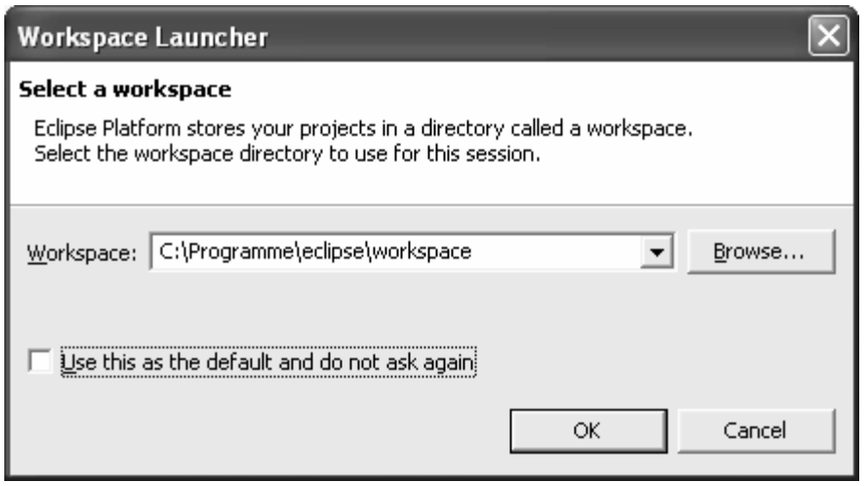

Abb. E1.1: Auswahl des Eclipse Workspace

Die verschiedenen Startoptionen von Eclipse können später unter *Help* | *Help Contents* | *Workbench User Guide* | *Tasks* | *Running Eclipse* eingesehen werden.

Nach kurzer Zeit erscheint die Eclipse Anwendung, beim erstmaligen Start mit einem speziellen *Welcome Screen*, der direkten Zugriff auf eine Überblicksdokumentation, verschiedene Tutorials, Beispiele sowie auf die News dieser Eclipse Version ermöglicht. Um zur eigentlichen Anwendung zu gelangen, schließen Sie diesen Screen durch Klicken auf das X-Symbol im Tab-Reiter. Der *Welcome Screen* kann jederzeit über *Help* | *Welcome* wieder angezeigt werden.

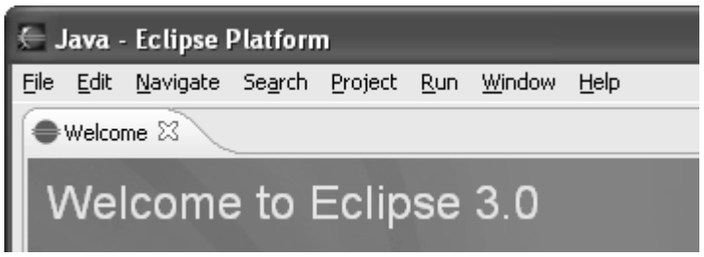

Abb. E1.2: Eclipse Welcome Screen

E1.3 Erstellen einer Java Applikation

E1.3.1 Erstellen eines Java Projektes

Innerhalb von Eclipse sind Java Applikationen in Projekten organisiert. Um ein Projekt zu erstellen, klicken Sie auf den kleinen Pfeil neben dem *New* Icon in der Toolbar und wählen *Project* aus. Dies kann auch über das Menü *File* | *New* | *Project* erreicht werden.

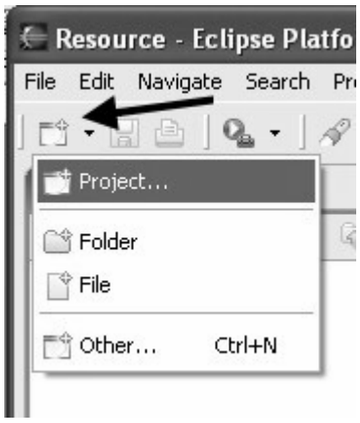

Abb. E1.3: Neues Eclipse Projekt auswählen

Dadurch wird der *Project Assistant* aktiviert, in dem die Voreinstellung *Java Project* mit *Next* zu bestätigen ist. Im folgenden Dialog ist ein eindeutiger Projektname anzugeben, etwa `ErstesProjekt`, die anderen Einstellungen können mit *Finish* bestätigt werden.

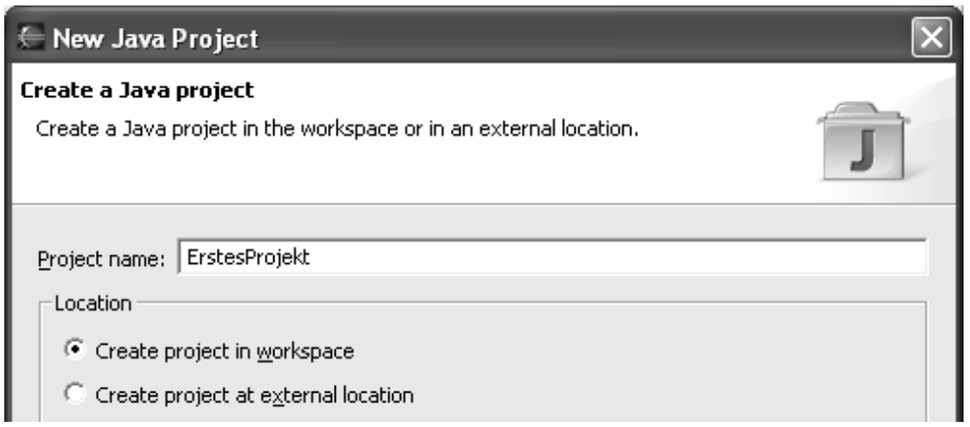

Abb. E1.4: Neues Eclipse Projekt erstellen

Da es sich um ein Java Projekt handelt, schlägt Eclipse in einem Dialogfenster vor, automatisch in die *Java Perspective* zu wechseln. Bestätigen Sie das Fenster mit *Yes*, zukünftige Nachfragen nach einem automatischen Perspektivenwechsel können durch Auswahl von *Remeber my decision* vorweg genommen werden.

E1.3.2 Erstellen einer Java Klasse

Um in der Java Perspektive eine neue Klasse zu erstellen, klicken Sie auf das *C-Icon* in der Toolbar. Man kann auch das Menü *File* | *New* | *Class* verwenden oder mit dem kleinen Pfeil neben dem *C-Icon* eine Liste aktivieren und *Class* auswählen.

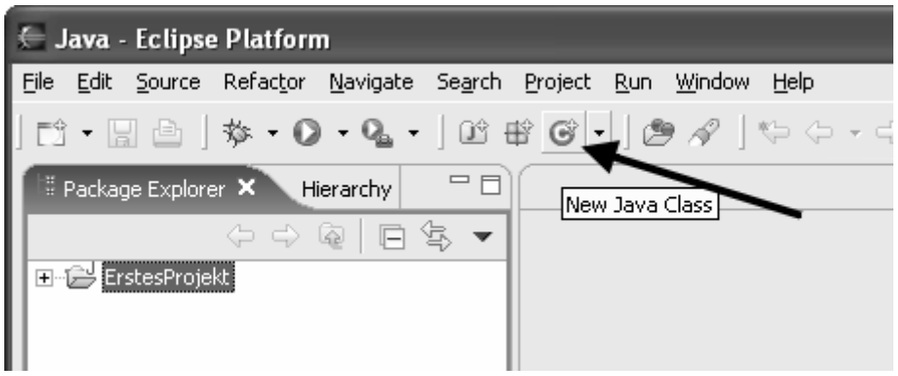

Abb. E1.5: Eclipse Java Perspektive: C-Icon

Im folgenden Dialog geben Sie den Namen der zu erstellenden Klasse an (z.B. `Test`) und stellen sicher, dass `ErstesProjekt` als *Source Folder* eingetragen und die Option zum Erstellen eines Methodenrumpfes für die Methode *main()* aktiviert ist. Die Warnung wegen der Verwendung des *default packages* kann vorerst ignoriert werden.

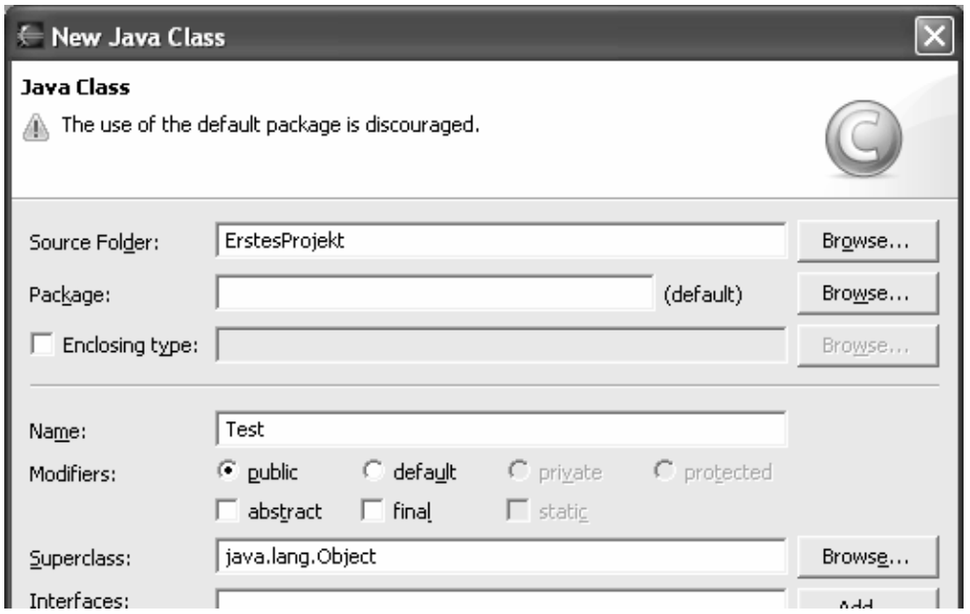

Abb. E1.6: Eclipse: Erstellen einer Java Klasse

Anschließend wird innerhalb des Projektes *ErstesProjekt* die Datei *Test.java* erzeugt, die bereits den Quelltext der Klasse und den Rumpf der *main()*-Methode enthält.

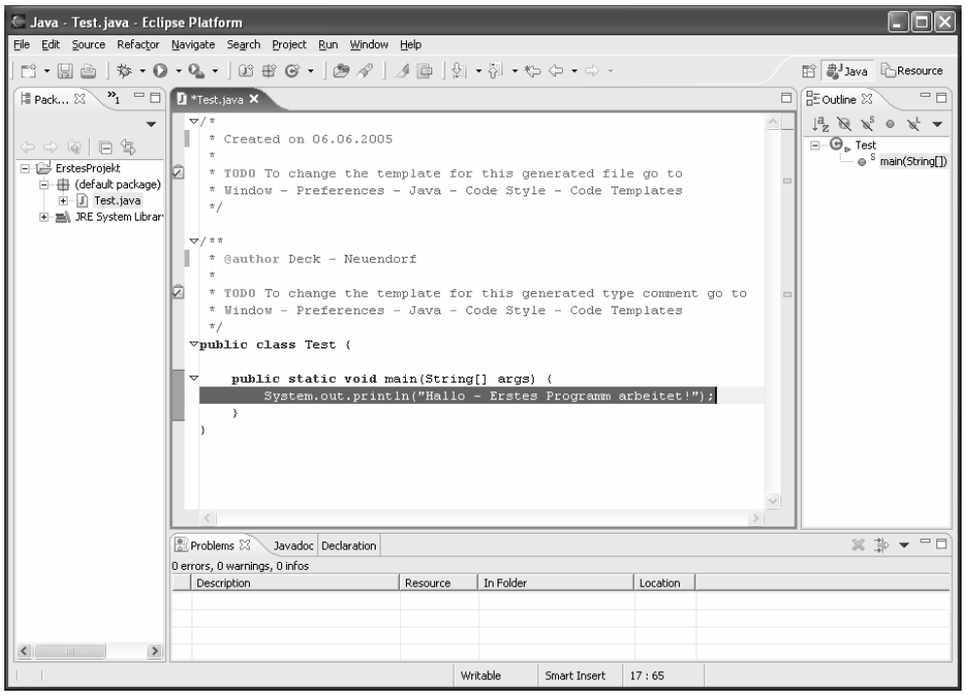

Abb. E1.7: Eclipse: Java Perspective

Die *main()*-Methode erweitern wir, indem wir im Java-Editor eine Zeile einfügen:

```
public static void main( String[] args ) {
   System.out.println( "Hallo - Erstes Programm arbeitet!");
}
```

E1.3.3 Übersetzen des Quelltextes

Beim Speichern der Java-Datei (Klick auf das Disketten-Symbol), wird diese automatisch übersetzt. Eventuelle Syntaxfehler werden von Eclipse frühzeitig erkannt, wobei ein Assistent über Korrekturvorschläge informiert. Hier lohnt es sich, die umfangreiche Dokumentation über Eclipse zu Rate zu ziehen.

E1.3.4 Starten der Applikation

Um die Anwendung erstmals zu starten, aktivieren wir das Drop-Down Menü durch Klicken des kleinen Pfeiles rechts neben dem *Run Icon*, wählen *Run As* und anschließend *Java Application*.

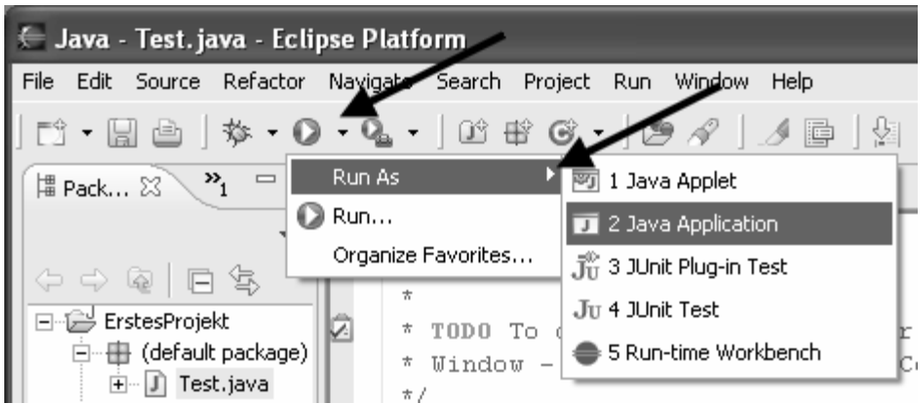

Abb. E1.8: Eclipse: Erzeugen einer Run Konfiguration

Das Testprogramm wird gestartet, worauf sich im unteren Teil der Eclipse Anwendung ein *Console* Fenster öffnet, das den Ausgabetext des Programms enthält. Falls der Quelltext zwischenzeitlich geändert wurde, muss noch ein Dialog bestätigt werden.

Soll anschließend die Anwendung erneut ausgeführt werden, kann man entweder direkt auf das Run Icon klicken oder "Test" aus der Liste der erzeugten Run-Konfigurationen wählen.

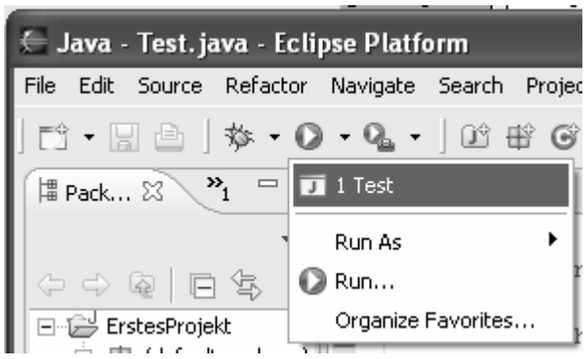

Abb. E1.9: Eclipse: Starten einer Anwendung

E1.3.6 Einbinden von `.java`-Quellcode-Files in bestehende Eclipse-Projekte

Einem bestehenden Eclipse-Projekt können bereits existierende `.java` –Quellcode-Dateien (z.B. die von uns bereitgestellten Lösungen der Übungsaufgaben) als physische *Kopie* direkt hinzugefügt werden. Dazu dient die *Import*-Funktion, die über das *Kontextmenü* (rechte Maustaste) des geöffneten Eclipse-Projekts im *Package-Explorer* aufgerufen wird:

Import... → *File System*

Im folgenden Dialogfenster wird mittels *Browse*-Button des Eingabefeldes *From Directory* bestimmt, aus welchem Verzeichnis die `.java`–Datei importiert wird. Die Dateien des angewählten Verzeichnisses werden aufgelistet und die Quellcodedatei kann zum Import selektiert werden. Über den *Browse*-Button des Eingabefeldes *Into Folder* wird festgelegt, *in* welches Verzeichnis die Quellcodedatei kopiert wird. Es ist der Workspace-Ordner des aktuellen Projekts zu wählen. Drücken des *Finish*-Buttons schließt die Operation ab: Die `.java`–Datei wird ins aktuelle Arbeitsverzeichnis kopiert und dem Eclipse-Projekt hinzugefügt.

E1.3.6 Einbinden von `IO.java` in Eclipse-Projekte

Natürlich kann auch `IO.java` so wie jedes andere Quellcode-File wie beschrieben einem Eclipse-Projekt mittels *Import*-Funktion (Kontextmenü) als physische *Kopie* hinzugefügt werden.

Die Datei `IO.java` allen Projekten als Kopie hinzuzufügen hat den Nachteil, dass Änderungen an `IO` für jede einzelne Kopie durchgeführt werden müssten. Vorteilhafter ist es, die Funktionalität von `IO` nur einmal als zentrale Ressource vorzuhalten und aus allen Projekten nur zu *referenzieren* statt `IO.java` stets zu kopieren.

Zu diesem Zweck muß `IO.class` in gepackter Form als ZIP- oder JAR-File (`IO.zip` oder `IO.jar`) vorliegen und sollte an zentraler Stelle (z.B. im *Workspace*-Ordner) abgelegt sein.

Die gepackte Datei kann auf zwei verschiedene Weisen durch Java-Eclipse-Projekte referenziert werden:

Projektspezifisches Referenzieren als externe Datei: Für das geöffnete Projekt ist *Project → Properties → Java Build Path* aufzurufen. Im Dialogfenster ist der Tab-Reiter *Libraries* zu selektieren und der Button *Add External JARs ...* zu betätigen. Der sich öffnende *JAR Selection*-Dialog ermöglicht die Selektion von `IO.jar` bzw. `IO.zip`. Betätigen des *OK*-Buttons schließt den Vorgang ab. Das selektierte File wird den *Libraries* hinzugefügt, die zur Kompilation des Projekts zur Verfügung stehen - und somit im Projekt verendet werden können, ohne dass eine Kopie der refernzierten Files angelegt wird. Der Bezug auf `IO.jar` bzw. `IO.zip` erscheint auch in der *Package Explorer* Sicht des aktuellen Projekts.

Die Referenz auf `IO.jar` bzw. `IO.zip` kann *entfernt* werden, indem erneut *Project → Properties → Java Build Path* aufgerufen und der Tab-Reiter *Libraries* ausgewählt wird. `IO.jar` bzw `IO.zip` ist zu selektieren und kann durch den *Remove*-Button aus dem *Build Path* entfernt werden. Betätigen des *OK*-Buttons schließt den Vorgang ab. Der Bezug auf `IO.jar` bzw. `IO.zip` verschwindet dadurch auch aus der *Package Explorer* Sicht des aktuellen Projekts.

Projektübergreifender Eintrag in *Classpath*-Variablen: Man kann auch mit den *Classpath*-Variablen des *Java Build Path* arbeiten. Dazu wird mittels der Menüfolge *Window → Preferences → Java → Build Path → Classpath Variables* im Dialogfenster durch Betätigen des Buttons *New ...* der Pfad zur Datei `IO.jar` bzw. `IO.zip` hinzugefügt. Der neuen *Classpath*-Variablen ist ein frei wählbaer Name zu geben. Betätigen des *OK*-Buttons schließt den Vorgang ab.

Nun kann ein geöffnetes Java-Eclipse-Projekt durch Aufruf von *Project → Properties → Java Build Path*, Selektion des Tabs *Libraries* und Betätiges des Buttons *Add Variable ...* auf den vorgenommenen Classpath-Eintrag zugreifen. Auch bei dieser Vorgehensweise erscheint der Bezug auf `IO.jar` bzw. `IO.zip` für das aktuelle Projekt in der *Package-Explorer* Sicht.

E1.4 Debuggen einer Java Applikation

E1.4.1 Start des Debug-Modus

Zum Debuggen einer Java Applikation setzen wir einen so genannten *Breakpoint*, indem man die entsprechende Code-Zeile durch Doppelklick an der linken Laufleiste des Editors auswählt. Der Breakpoint wird durch einen blauen Punkt markiert.

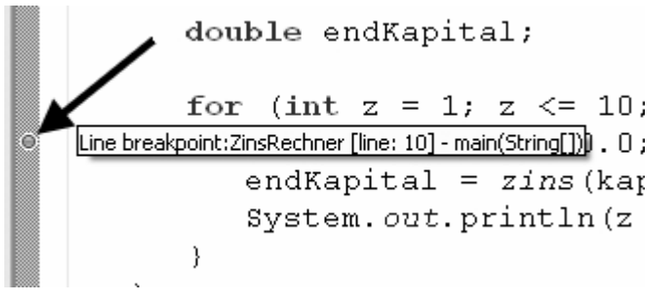

Abb. E1.10: Eclipse: Auswahl eines Breakpoints

Um die Anwendung im Debug-Modus zu starten, aktivieren wir das Drop-Down Menü durch Klicken des kleinen Pfeiles rechts neben dem *Debug Icon*, wählen *Debug As* und anschließend *Java Application* (vgl. Abb. E1.11).

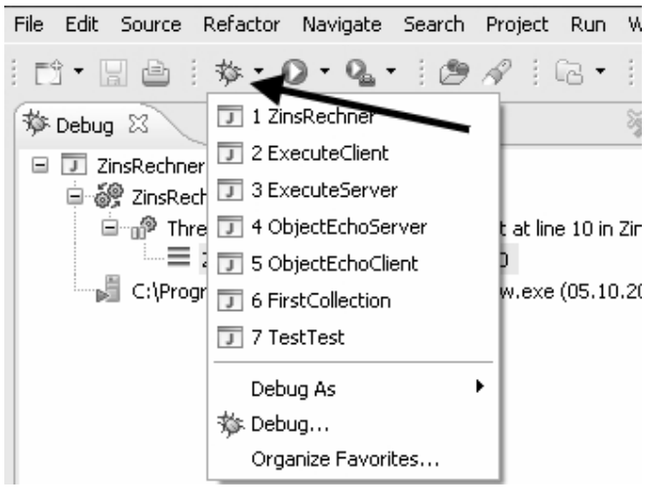

Abb. E1.11: Eclipse: Programmstart im Debug-Modus über Debug Icon

Alternativ lässt sich der Debug-Modus auch durch Klick der rechten Maustaste in den Editor und Auswahl von *Debug As →Java Application* aktivieren. Später kann dieser Modus über Klicken des *Debug Icon* direkt ausgewählt werden.

Anschließend sollte das *Confirm Perspective Switch* Popup bestätigt werden, damit Eclipse aus der *Java Perspective* in die *Debug Perspective* wechselt. Zukünftige Nachfragen kann man sich ersparen, wenn man zuvor die Checkbox "Remember my decision" bestätigt. In welcher Perspective man sich gerade befindet, lässt sich am einfachsten am aktivierten Icon in der *Perspective Bar* erkennen (vgl. Abb. E1.12). Dort kann man durch Klick auf das entsprechende Icon die Perspektive wechseln.

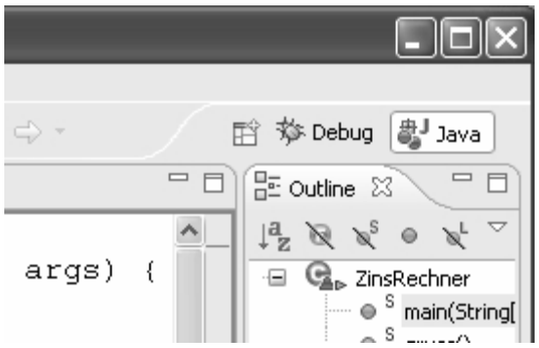

Abb. E1.12: Eclipse: Debug und Java Icon in der Perspective Bar

Grundsätzlich sollten folgende Punkte beachtet werden:

- Bei einer mit *Run* gestarteten Anwendung findet **kein Debugging** statt, selbst wenn Breakpoints gesetzt sind. Der Start muss im Debug-Modus erfolgen
- Es ist empfohlen, die jeweils passende Perspektive auszuwählen:
  Debuggen in der *Debug Perspective*, "normales" Run in der *Java Perspective*!
  Die aktive Perspektive sagt nichts über Debugging oder normales Run aus.

E1.4.2 Navigieren im Debug-Modus

Eine im Debug-Modus gestartete Anwendung führt die Anweisungen aus, die vor dem Erreichen des Breakpoints durchlaufen werden. Am Breakpoint wird die Ausführung angehalten, die aktuellen Werte der Variablen werden in der *Variables View* angezeigt (siehe Abb. E1.13). Durch Klicken auf die Variablen lassen sich auch Inhalte von komplexen Strukturen (z.B. Arrays, Objekte) visualisieren.

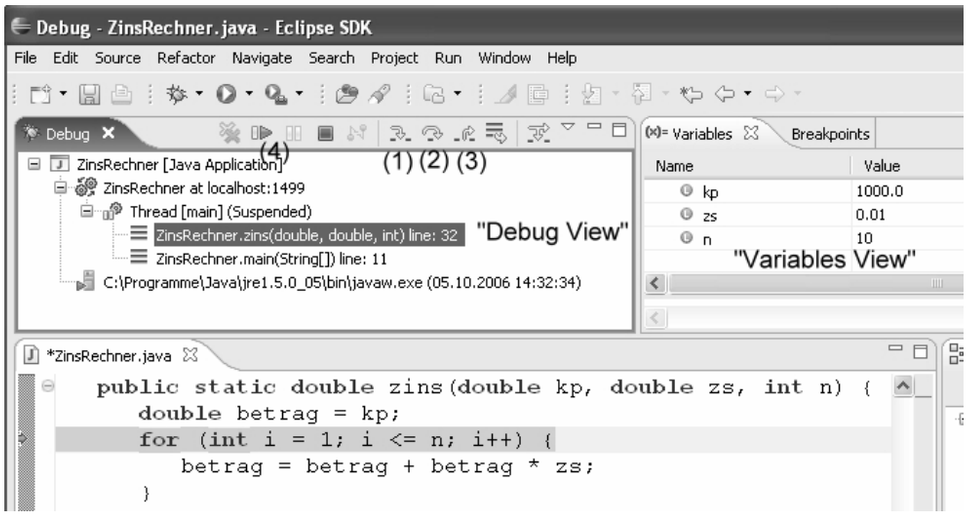

Abb. E1.13: Eclipse: Debug View und Variables View

Mit den folgenden Buttons kann man schrittweise durch den Code navigieren (vgl. Abb. E1.13, (1) bis (4)):

- *Step Into* (1) : schrittweise Ausführung; bei Methodenaufruf **in** Methode
- *Step Over* (2): schrittweise Ausführung; bei Methodenaufruf **als ein Schritt**
- *Step Return* (3): verlässt aktuelle Methode
- *Resume* (4): Ausführung zum nächsten Breakpoint bzw. Programmende

Die Möglichkeiten zum Debuggen in Eclipse sind sehr vielfältig. Als Einstieg zur Vertiefung empfehlen wir die Dokumentation "Debugging your programs" im "Java Development User Guide" (*Help → Help Contents → Java Development User Guide → Getting Started → Basic Tutorials)*.

E1.5 Die JDK-Tools `javac` und `java`

Der **Java-Compiler** `java` verfügt über zahlreiche interessante Aufrufoptionen, die unter [SUN05d] dokumentiert sind. Neben der Angabe des abzusuchenden classpath kann z.B. festgelegt werden, ob und welche Debug-Informationen kompiliert werden, welche JDK-Sourcecodeversion akzeptiert wird, ob Informationen über den Kompilationsvorgang erzeugt werden und welche Warnungen vom Compiler ausgegeben werden.

Hervorzuheben ist die **Option `-d`**: Diese legt fest, in welches *Zielverzeichnis* die erzeugten Class-Files abgelegt werden. Das Verzeichnis muss bereits existieren:

```
javac -d Demos First.java
```

Für die Klassen eines Pakets werden jedoch Unterverzeichnisses erzeugt, die Paketnamen und -struktur entsprechen. Mittels dieser Option können Java-Quellcode-Files und Class-Files in verschiedenen Verzeichnissen getrennt voneinander verwaltet

werden, was insbesondere in grossen Projekten sinnvoll ist. Wird die Option nicht verwendet, liegen die Class-Files im selben Verzeichnis wie die zugehörigen Java-Quellfiles.

Statt alle `javac`-Kommandozeilenargumente (Optionen und zu kompilierende Quellcode-Files) einzeln auf der Kommandozeile aufzuzählen, können diese (durch Leerzeichen oder Zeilensprünge getrennt) in einem *Argument-File* aufgelistet werden. Dies erlaubt, beliebig lange `javac`-Kommandos anzulegen und wiederzuverwenden. Die enthaltenen Java-File-Pfade beziehen sich auf das aktuelle Arbeitsverzeichnis. Der Filename (inklusive Pfadangabe) des Argumentfiles wird durch Vorstellen von @ direkt an javac übergeben:

```
javac @pfad\FirstArgFile
```

Es können beliebig viele Argumentfiles verwendet werden, z.B. das File optionen für javac-Optionen und das File quellen für Quellcode-Filenamen:

```
javac @pfad1\optionen @pfad2\quellen
```

Wenn `javac` während der Übersetzung eines Projekts weitere Quell- oder Class-Files benötigt (z.B. verwendete Paket- oder Oberklassen, implementierte Interfaces), dann sucht er diese im aktuellen Arbeitsverzeichnis oder in allen Verzeichnissen, die durch die classpath-Umgebungsvariable definiert sind. Findet `javac` dabei nur ein gesuchte Class-File, so verwendet er dieses; findet `javac` nur ein gesuchtes Quellfile, so kompilierte er dieses verwendet die kompilierten Class-Files; findet `javac` sowohl Class- als auch zugehöriges Quellcode-Files, so prüft er, ob die Quellcode-Files veraltet (älter als das Quellcode-File) sind und rekompiliert diese vor der Verwendung. Durch die Option `-verbose` kann die Compilationstätigkeit von `javac` kontrolliert werden.

Der **Java Application Launcher** `java` veranlasst die Abarbeitung einer Java-Anwendung. Dazu wird eine JVM gestartet, das angegebene Class-File geladen und dessen `main()`-Methode aufgerufen. Es können zahlreiche technische Optionen für `java` spezifiziert werden, wie in [SUN05d] erläutert. Diese regeln z.B., ob die HotSpot Client VM oder Server VM eingesetzt wird, ob die Ausführung eine 32-bit oder 64-bit Umgebung nutzt, ob und wie JIT-Kompilation eingesetzt wird, wieviel initialer und maximaler Speicher der Anwendung zur Verfügung steht und wie die Garbage Collection vorgeht.

Hervorzuheben ist die *Option* **`-jar`**: Diese veranlasse `java`, eine in einem JAR-File verpackte Java-Anwendung auszuführen. Statt einem Class-File wird `java` dazu der Name des auszuführenden JAR-Files übergeben, z.B. `java –jar First.jar`

Das zugehörige JAR-Manifest-File muss dafür eine Zeile der Form:

`Main-Class:` *`classname`*

Enthalten. Es bezeichnet *`classname`* die Klasse, mit deren `main()`-Methode die Ausführung startet.

E1.6 Das JDK-Tool `javadoc`

Mit dem Tool `javadoc` (im *`bin`*-Verzeichnis der JDK-Installation) kann sehr einfach professionell formatierte HTML-Dokumentation generiert werden. Das Format der erzeugten Seiten entspricht exakt dem Format der JDK-Dokumentation [SUN05a]. Somit dient `javadoc` dazu, Java-Klassen und ihre Elemente (Attribute und Methoden) zu dokumentieren.

Dazu muß der jeweilige Java-Quellcode mit `javadoc`-spezifischen Kommentaren versehen werden: Alle Kommentarinhalte zwischen dem öffnenden Kommentarmarker **`/**`** und dem schließenden Marker **`*/`** werden durch `javadoc` extrahiert und automatisch in die generierte HTML-Dokumentation eingefügt.

Der Aufruf des Dokumentations-Tools erfolgt durch Angabe des betreffenden Paketnamens und der Quellcodefiles (einzelnes `.java`-File oder Aufzählung).

```
javadoc [options]  packagename sourcefiles
```

Wird kein Paketname übergeben, so sucht `javadoc` die betreffenden `.java`-Files im aktuellen Arbeitsverzeichnis.

Einige *Optionen*:

- Nur Kommentare *öffentlicher* Klassenelemente extrahieren: `-public`
- Auch *private* Klassenelemente in Dokumention aufnehmen: `-private`
- Explizier *Dateipfad* (path) zu Quellcode-Dateien angeben: `-d path`

Weitere Aufrufoptionen finden sich in der JDK-Tools-Dokumentation [SUN05d].

Soll sich ein `javadoc`-Kommentar auf eine Klasse insgesamt beziehen, muss er direkt vor deren Quellcode plaziert werden. Kommentare einzelner Methoden müssen vor den Methodenköpfen stehen, Attributkommentare vor dem Attribut.

Besonders interessant sind die durch @ eingeleiteten sogenannten **`javadoc-Tags`**. Diese unterrichten `javadoc` über die semantische Bedeutung nachfolgender Kommentare. Dadurch werden die betreffenden Inhalte in der HTML-Dokumentation speziell angeordnet und formatiert. Auch die mit `javadoc`-Tags versehenen Kommentarinhalte müssen innerhalb **`/**`** und **`*/`** vom Java-Quellcode abgesetzt werden. Hier nur einige wenige der zahlreichen `javadoc`-Tags:

`@version`	Versionsangabe
`@author`	Nennung des Autors
`@param`	Spezifikationen eines Methodenparameters
`@return`	Spezifikation des Rückgabewerts einer Methode
`@exception`	Beschreibung der Ausnahmebehandlung
`@see`	Querverweis auf anderes Paket, Klasse bzw. Methode

Auch die explizite Verwendung von HTML-Tags innerhalb von `javadoc`-Kommentaren ist zur individuellen Formatierung möglich.

Durch Aufruf von `javadoc` werden zahlreiche `.html`-Dateien erzeugt, ferner eine Datei `stylesheet.css` sowie eine `package-list`. Die Datei `index.html` er-

öffnet den Einstieg in die erzeugte Dokumentation, die in jedem Browser betrachtet werden kann.

Eine javadoc-konforme Quellcode-Kommentierung verdeutlicht folgende Klasse:

```
/**
   Eine Klasse zur Währungsumrechnung
   @author  Deck, Neuendorf
   @version 1.0.0
*/
public class Waehrung {
   /**
      Der Umrechnungskurs als Attribut der Klasse
   */
   private double kurs;

   /**
      Der Konstruktor setzt den Umrechnungskurs fest
      @param   k  Der Umrechnungskurs
   */
   public Waehrung( double k ) {
      kurs = k;
   }

   /**
      Berechnung des Fremdwährungsbetrags
      @param   euro   Der umzurechnende Euro-Betrag
      @return  fremd  Der Betrag in Fremdwährung
   */
   public double inFremd( double euro ) {
      double fremd = euro * kurs;
      return fremd;
   }

   /**
      Ausgabe des Kurswerts auf der Konsole
      @see  java.lang.System#println( String n );
   */
   public void ausgabe(){
      System.out.println( "Kurswert = " + kurs );
   }
}
```

Dokumentation wird für diese Klasse generiert durch den Aufruf:

```
javadoc –private Waehrung.java
```

Weitere Möglichkeiten von javadoc finden sich in [SUN05d].

E1.6.1 javadoc-Aufruf unter Eclipse 3.1

Der Java-Quellcode eines Eclipse-Projekts ist mit javadoc-Kommentaren zu versehen. Folgende Maßnahmen sind zum javadoc–Aufruf in Eclipse erforderlich:

1. Einen Ordner für die von javadoc generierten Dateien im Dateisystem anlegen.

2. Für das geöffnete Java-Eclipse-Projekt wird festgelegt, wo die von `javadoc` generierten Dateien abzuspeichern sind. Dazu dient die Menüfolge:

Project → Properties → Javadoc Location

Im folgenden Dialogfenster wird mittels *Browse*-Button der vorgesehene Ordner im Dateisystem selektiert.

3. Das Tool `javadoc` wird aufgerufen. Zwei Möglichkeiten:

a) Menü: *Properties → Generate Javadoc*

b) Kontextmenü des Projekts im Package-Explorer: *Export ... → Javadoc*

In beiden Fällen wird der Benutzer durch eine Folge von Dialogen geführt in denen alle relevanten `javadoc`-Optionen selektiert werden können.

E1.7 Das JDK-Tool `jar`

Java-Archive dienen der einfachen da kompakten Code-Auslieferung kompletter Java-Applikationen und -Projekte. Bei Java-Anwendungen handelt es sich physisch meist um eine große Anzahl von `.class`-Dateien. Um diese gepackt weitergeben zu können, wurden Java-Archive eingeführt.

Ein Java-Archiv stellt eine `.jar`-Datei dar, in der verschiedene Dateien (typischerweise zahlreiche `.class`-Files und weitere Ressourcen wie Bilder und Audiodateien aber z.B. auch `javadoc`-Dokumentation) komprimiert gespeichert und in einer einzigen Archiv-Datei zusammengefasst zur Verfügung gestellt werden. Somit beschleunigen Java-Archive insbesondere die Übertragung von Java-Anwendungen im Netz, da nur wenige große `.jar`-Files statt vieler kleiner einzelner `.class`-Files versandt und geladen werden müssen. Dies spielt insbesondere beim Einsatz von Applets eine wichtige Rolle.

Das JAR-Format ist eine Variante des ZIP-Formats. Der Inhalt eines `.jar`-Files kann mit herkömmlichen unzip-Programmen (wie Winzip) betrachtet und extrahiert werden.

Neben `.class`-Files und Ressourcen enthält jedes `.jar`-File ein sogenanntes *Manifest* in der Datei `Manifest.mf` mit Meta-Informationen über die im Archiv zusammengefassten Dateien. Zu den Meta-Informationen zählen z.B. Prüfsummen der komprimierten Dateien und eventuell Angaben zur digitalen Signatur. Ein `.jar`-File kann digital signiert werden; in diesem Fall enthält es auch noch `.sf`-Signaturdateien.

Die *Manifest-Datei* enthält zumindest die Manifest-Versionsnummer. Soll eine im `.jar`-File archivierte Java-Anwendung durch den Interpreter `java` direkt ausführbar sein, so muß angegeben werden, welches `.class`-File die `main()`-Methode der Anwendung enthält. Dazu dient der Manifest-Eintrag `Main-Class`. Diesem wird der Name der betreffenden `.class`-Datei (aber *ohne* die Extension .class) zugewiesen. Das Manifest einer ausführbaren `.jar`-Datei `Bank.jar` mit `main()`in `Transaction.class` enthielte also zumindest die Einträge:

```
Manifest-Version: 1.0
Main-Class: Transaction
```

Die in `Bank.jar` archivierte Anwendung kann durch Aufruf von `java` mit der Option `–jar` direkt gestartet werden:

```
java –jar Bank.jar
```

Die **Aufrufsyntax von des Tools `jar`** (*bin*-Verzeichnis) ist fast identisch mit dem `tar`-Kommando unter Unix:

```
jar Funktion[options]  Zieldatei  Dateiliste
```

`Zieldatei` ist das JAR-Archiv, in dem alle unter `Dateiliste` angegebenen Dateien archiviert werden.

Folgende `Funktions`-Alternativen stehen zur Verfügung:

- `c` neues `.jar`-Archiv erzeugen (create)
- `u` bestehendes `.jar`-Archiv aktualisieren (update)
- `t` Inhalt eines Archivs auf Standardausgabe auflisten
- `x` alle oder aufgelistete Dateien des Archivs extrahieren

Die Arbeitsweise der Funktionen kann durch `options` beeinflusst werden:

- `f` Archivname des `.jar`-Files unter `Zieldatei` vorgeben
- `v` Fortschrittsmeldungen beim Packen oder Entpacken einer `.jar`-Datei ausgeben (`verbose`)
- `0` Dateien werden unkomprimiert in `.jar`-Datei abgelegt (null)
- `M` kein Manifest erzeugen
- `m` Manifest aus externer Datei importieren

Soll z.B. ein neues JAR-Archiv namens `Bank.jar` aus *allen* `.class`-Dateien im aktuellen Verzeichnis erstellt werden, so lautet der `verbose`-Aufruf von `jar`:

```
jar cvf  Bank.jar  *.class
```

Soll dieses bestehende Archiv dann mit einer neuen Version von Giro.class versehen werden (update), so lautet der `verbose`-Aufruf von `jar`:

```
jar uvf  Bank.jar  Giro.class
```

Mit `jar tf Bank.jar` wird der Inhalt von `Bank.jar` ausgegeben.

Mit `jar xf Bank.jar` wird die `.jar`-Datei wieder entpackt.

Besonders Applets profitieren von Java-Archiven: Alle im Applet benötigten Klassen werden einzeln vom Webserver bezogen, wobei für jede geladene Datei eine gesonderte http-Verbindung aufgebaut wird. Muß jedoch nur eine einzelne und zudem komprimierte `.jar`-Datei geladen werden, so ist dafür nur eine http-Zyklus nötig und die Übertragungszeit wird deutlich reduziert.

Applet-Code kann deshalb in `.jar`-Files gepackt und diese in HTML-Seiten eingebunden werden. Im `applet`-Tag ist dazu der Parameter `archive` vorgesehen:

```
<applet
   code=Transaction.class  archive=Bank.jar
   width=600   height=400                  >
</applet>
```

Alle im Applet `Transaction` benötigten `.class`-Files werden in der Datei `Bank.jar` zusammengefasst und aus dieser geladen.

Außer mit dem JDK-Tool `jar` können `.jar`-Dateien auch aus Java-Programmen heraus gelesen und geschrieben werden. Dazu dienen die Klassen des J2SE-Pakets `java.util.jar` [SUN05c]. Zum Schreiben und Lesen von `.zip`-Dateien dient das J2SE-Paket `java.util.zip` [SUN05c].

Zahlreiche Details und technische Anwendungsmöglichkeiten des JAR-Formats und seiner Manifest-Datei gehen aus der JAR File Spezifikation [SUN05a] und der Tool Dokumentation zu `jar` [SUN05d] hervor.

E1.7.1 `jar`-Aufruf unter Eclipse 3.1

Für Java-Projekte unter Eclipse können `.jar`-Files komfortabel erstellt werden. Für das geöffnete selektierte Projekt wird über das Kontextmenü (rechte Maustaste) im *Package Explorer* eine Folge von Dialogen aufgerufen:

Export ... → *JAR file*

Der Benutzer wird durch eine Dialogfolge geführt, in der alle `jar`-Aufrufoptionen und Vorgaben bezüglich aufzunehmender Dateien, Ablageort und *Manifest*-Einträge getroffen werden können. Das erzeugte `.jar`-File wird in das aktuelle Eclipse-Projekt aufgenommen und erscheint in dessen *Package Explorer* Sicht.

Wurde für die `.jar`-Datei im *Manifest*-File die `Main-Class` eingetragen, so kann die enthaltene Java-Anwendung in Eclipse direkt ausgeführt werden. Dazu wird im *Package Explorer* im Kontextmenü des `.jar`-Files dessen Ausführung angestossen:

Run → *Run...*

Im folgenden Dialogfenster können Aufrufparamerter angegeben und die Anwendung durch den *Run*-Button gestartet werden.

E2.8 Ein-/ Ausgabe mittels der Klasse IO.java

Die wesentlichen öffentlichen *Methoden* der Klasse `IO` sind hier mit ihren Schnittstellen und Wirkungsweise aufgeführt. `IO.java` ist von der Webseite des Buches downloadbar. Von den Methoden `write()` und `writeln()` existieren überschriebene Fassungen, so dass alle relevanten Datentypen als Parameter zur Ausgabe an diese übergeben werden können. Da die genannten Methoden `static` deklariert sind, sind sie direkt auf der Klasse `IO` aufrufbar, ohne dass eine vorherige Objektinstanziierung erfolgen müsste.

Konsolenausgaben:

```
// Die Methode advance() gibt n Leerzeilen auf der Konsole aus
```

```
public static void advance( int n )
// Die Methoden write() geben ihren Parameter auf der Konsole aus
public static void write( String s )
public static void write( int i )
public static void write( float f )
public static void write( double d )
// Die Methoden writeln() geben ihren Parameter auf der Konsole
// aus und führen einen Zeilenvorschub durch.
public static void writeln( )  // nur Zeilenvorschub
public static void writeln( String s )
public static void writeln( int i )
public static void writeln( float f )
public static void writeln( double d )
```

Konsoleneingaben:

```
// Die Methode promptAndRead……() geben den String s auf der
// Konsole aus und erwarten vom Benutzer eine Eingabe, deren
// Datentyp ihrem Rückgabewert entspricht. Der eingegebene Wert
// wird an an den Aufrufer zurückgeliefert.
public static String promptAndReadString( String s )
public static char promptAndReadChar( String s )
public static int promptAndReadInt( String s )
public static float promptAndReadFloat( String s )
public static double promptAndReadDouble( String s )
public static boolean promptAndReadBoolean( String s )
```

Runden von Kommazahlen:

```
// Die Methoden round() liefern den auf n Nachkommastellen
// gerundeten Wert von x zurück, ohne den Parameter x dabei
// zu verändern.
public static double round( double x, int n )
public static float  round( float  x, int n )
```