E3 JDK-Installation

Das Zusatzkapitel gibt einen Überblick über die Verzeichnisse des JDK und die Bedeutung der darin installierten Dateien; diese findet man nach der J2SE-Installation auf dem Rechner vor. Ferner wird der `classpath`-Mechanismus erläutert und auch auf die Bedeutung optionaler Pakete eingegangen. Dabei halten wir uns eng an die relevante *Sun-Dokumentation*, die sich findet unter:

`java.sun.com/j2se/1.5.0/docs/tooldocs/` *und*

`java.sun.com/j2se/1.5.0/docs/guide/extensions/`

Es existieren verschiedene Fassungen für die Verzeichnisinstallation auf den Betriebssystemen Windows, Linux und Solaris. Wir präsentieren die Verhältnisse für Windows; die Filesstruktur der anderen Betriebssyteme ist ähnlich. Der `classpath`-Mechanismus wird für Windows und Linux /Solaris dargestellt.

E3.1 Verzeichnisse und Dateien für die Java-Entwicklung

Die wichtigsten Verzeichnisse und Dateien für die Java2-*Softwareentwicklung* lauten:

`jdk1.5.0`

Das Wurzelverzeichnis der JDK-Installation enthält Copyright-, Lizenz- und README-Dateien. Insbesondere jedoch findet sich dort `src.zip`, der archivierte Quellcode des gesamten J2SE-Paketklassen.

`jdk1.5.0\bin`

Die Entwicklungstools des JDK (`java`, `javac`, `jar`, `javadoc`, ...). Die Umgebungsvariable `path` sollte auf dieses Verzeichnis weisen, damit die Tools von der Kommandozeile aufgerufen werden können.

`jdk1.5.0\lib`

Weitere Dateien, die von den Entwicklungstools benötigt werden. Diese befinden sich in den Archiven `tools.jar` und `dt.jar`.

`jdk1.5.0\jre`

Das Wurzelverzeichnis der JVM-Laufzeitumgebung, die von den JDK-Entwicklungstools verwendet wird. Die Laufzeitumgebung ist eine konkrete Implementierung der Java2-Spezifikation. Dieses Verzeichnis nennt die Systemeigenschaft (system property) `java.home`.

`jdk1.5.0\jre\bin`

EXEs und DLLs, die von Tools und der Javalaufzeitumgebung benötigt werden. Die exe-Files sind identisch mit denen in `jdk1.5.0\bin`

`jdk1.5.0\jre\bin\client`

DLLs, die von der Java2 HotSpot Client JVM verwendet werden.

`jdk1.5.0\jre\bin\server`

DLLs, die von der Java2 HotSpot Server JVM verwendet werden.

`jdk1.5.0\jre\lib`

Codebibliotheken und Resourcedateien, auf die die JVM zugreift. Dazu gehören:

- **`rt.jar`** : Die sogenannten Bootstrap-Klassen, d.h. die `.class`-Dateien, in denen die Standard-Pakete und -APIs der J2SE implementiert sind.
- `charsets.jar` : Klassen zur Zeichenkonvertierung

`jdk1.5.0\jre\lib\ext`

Für Java-Extensions (optionale Pakete) vorgesehenes Default-Installationsverzeichnis. Hier können Klassen (in .jar-Files gebündelt) abgelegt werden, für die der Java-Extension-Mechanismus zum Tragen kommen soll (s.u.). Enthält u.a.

- `localedata.jar` : Lokalisierungsdaten für Klassen der Pakete `java.text` und `java.util.`

`jdk1.5.0\jre\lib\security`

Dateien für die Java-Sicherheitsverwaltung. Werden vom Security Manager der JVM ausgewertet, um die Rechte von Java-Anwendungen auf der aktuellen Plattform zu kontrollieren. U.a. die Files `java.policy` und `java.security`.

`jdk1.5.0\jre\bin\applet`

Verzeichnis für `.jar`-Files mit Klassen, die von Applets benötigt werden. Dies verkürzt die Startzeit grösserer Applets, indem deren `.class`-Files vom lokalen Filessystem (statt über das Netzwerk) geladen werden. Es greifen jedoch dieselben Sicherheitsmechanismen, als wären diese Klassen aus dem Netz geladen worden.

`jdk1.5.0\jre\lib\fonts`

TrueType-Fonts, die vom Betriebssytem verwendet werden.

Zusätzliche Dateien und Verzeichnisse:

Weitere Verzeichnisse und Dateien enthalten Demoprogramme, den J2SE-Quellcode und C-Headerfiles:

`jdk1.5.0\src.zip`

Archivierter J2SE-Quellcode aller Standard-Pakete. Somit lässt sich studieren, wie von Sun selbst die Standard-J2SE Klassen implementiert wurden.

`jdk1.5.0\demo`

Demoprogramme (inklusive Quellcode) für typische J2SE-Bereiche.

`jdk1.5.0\demo\applets`

Zahlreiche Applets

`jdk1.5.0\demo\jfc`

Demoprogramme, die die Grafikprogrammierung mit Java Foundation Classes (JFC), d.h. Java2D und Swing darstellen.

jdk1.5.0\demo\jpda

Demoprogramme zur Verwendung des Java-Debug-Frameworks.

jdk1.5.0\demo\plugin

Demoprogramme zur Verwendung der Java Plug-In Technologie.

jdk1.5.0\include

C-Headerfiles, die benötigt werden, um unter Java mittels JNI auf plattformspezifischen (native) Code zugreifen zu können.

Anmerkung: Selbstgeschrieben Klassen bzw. Pakete sollten (abgesehen von Nutzern des Extension-Mechanismus) außerhalb der JDK-Verzeichnisstruktur installiert werden. Auf diese Weise sind sie vom Deinstallieren und Neuinstallieren eines J2SDK nicht betroffen.

E3.2 Der classpath-Mechanismus

Durch die Variable `classpath` wird festgelegt, unter welchen Verzeichnispfaden die Java Laufzeitumgebung nach Paketen, Klassen (Bytecode bzw. Quellcode) und anderen Ressourcen sucht. Dabei handelt es sich Klassen, die nicht (mitgelieferter) Teil der J2SE-Plattform (bootstrap-Klassen) von Sun oder Java-Extensions (s.u.) sind, sondern Klassen anderen Ursprungs bzw. selbstprogrammierte und –kompilierte Klassen.

Die Entwicklungstools `java`, `javac`, ... suchen benötigte `.class`-Files zuerst im aktuellen Arbeitsverzeichnis, sodann im Verzeichnis der Bootstrap-Klassen, dann im Verzeichnis der Java-Extension-Klassen und erst zuletzt im den Verzeichnissen des `classpath`. Somit muss der `classpath` nur gesetzt werden, wenn es sich um Klassen handelt, die sich nicht im aktuellen Arbeitsverzeichnis oder im Extension-Verzeichnis befinden.

Der classpath kann auf zweierlei Weise gesetzt werden:

- **Variante1:** Verwendung der `–classpath`-Option der JDK-Tools (`java`, `javac`, ...). Dies erlaubt ein individuelles Einstellen für jede kompilierte und ausgeführte Anwendung, ohne dadurch andere Anwendungen zu beeinflussen. Das Tool java hat auch die Option –cp als Abkürzung für –classpath.
- **Variante2:** Dauerhaftes Setzen der CLASSPATH-Umgebungsvariablen für das verwendete Betriebssystem. Dadurch erspart man sich die ausdrückliche Angabe bei jedem Kompilations- und Ausführungsvorgang.

Anmerkung: IDEs (wie Eclipse) ermöglichen, den Klassenpfad komfortabel einzustellen. Dennoch ist auch in diesem Fall der `classpath`-Mechanismus wirksam.

Anmerkung: Durch Ablegen von `.class`-Dateien als .jar-Files im Extension-Verzeichnis können diese schneller gefunden und somit auch die zugehörige Anwendung beschleunigt werden.

Setzen unter Windows:

Variante 1: `C:> sdkTool –classpath classpath1; classpath2 …`

Variante 2: `C:> set CLASSPATH=classpath1;classpath2 …`

Es dürfen keine Leezeichen um das Gleichheitszeichen (=) erscheinen.

Setzen unter Linux / Solaris:

Variante 1: % *sdkTool* **–classpath** *classpath1*:*classpath2* …

Variante 2: % **setenv CLASSPATH** *classpath1*:*classpath2* …

Dabei steht *sdkTool* für eines der Kommandozeilen-Entwicklungstools (`java`, `javac`, `javadoc`, ...).

Es bedeuten *classpath1* und *classpath2* Verzeichnispfade zu .jar-, .zip- oder .class-Dateien. Klassen können in Verzeichnissen (Ordnern) oder Archiv-Dateien gespeichert werden. Jede Pfad-Angabe sollte mit einem Datei- oder Verzeichnisnamen enden, abhängig davon, worauf der Pfad zeigt:

- Im Falle eines **.jar- oder .zip-Files**, die .class-Dateien enthalten, endet der `classpath` mit dem konkreten Namen des .jar- oder .zip-Files.
- Im Falle von **.class-Files** eines **namenlosen Standardpakets** endet der `classpath` mit dem Verzeichnis, das die .class-Files enthält.
- Im Falle von **.class-Files** eines **benannten Pakets** endet der `classpath` mit dem Verzeichnis, in dem das Wurzelpaket (bei geschachtelten Paketen) liegt.

Mehrere aufeinanderfolgende `classpath`-Einträge werden unter Windows durch *Semikolon*, unter Linux/Solaris durch *Doppelpunkt* voneinander getrennt.

Während der Ausführung eines Entwicklungstools übersteuern die mittels –`classpath`-Option gesetzen Pfade die mittels CLASSPATH-Umgebungsvariable gesetzen Pfade.

Standardmäßig wird bequemerweise im *aktuellen Arbeitsverzeichnis und dessen Unterverzeichnissen* nach Klassen gesucht. Durch das Setzen des `classpath` wird dieser Default übersteuert. Soll auch *weiterhin* das aktuelle Arbeitsverzeichnis im Klassenpfad enthalten sein, so muss ein *Punkt* (.) in die `classpath`-Liste aufgenommen werden!

Anmerkung: `classpath`-Einträge, die weder Verzeichnisse noch Archive (.zip- oder .jar-Files) sind, werden ignoriert.

Setzen der CLASSPATH-Umgebungsvariablen beim Systemstart:

Durch Einträge in *Skriptdateien* oder mittels *Systemtools* kann die CLASSPATH-Umgebungsvariable beim Starten des Betriebssystems automatisch gesetzt werden. Das Vorgehen ist betriebssystemabhängig.

Windows 95, 98 und ME: Eintrag in der `autoexec.bat`.

NT, Win2000, XP: Das System-Tool der Systemsteuerung erlaubt die Definition von Umgebungsvariablen.

Linux / Solaris:

Shells csh, tcsh: `setenv`-Kommando in der .cshrc-Datei.

Shells sh, ksh: `export`-Kommando in der .profile-Datei.

Reihenfolge der Pfadeinträge:

Die Reihenfolge, in der die diversen `classpath`-Suchpfade angeführt werden, spielt eine nicht unwichtige Rolle. Nicht-Bootstrap- und Nicht-Extension-Klassen werden in den verschiedenen Verzeichnisspfaden gesucht in *genau der Reihenfolge*, in der diese Verzeichnisse bei der `classpath`-Definition angegeben wurden. Somit kann man den *Suchvorgang schneller und zuverlässiger* machen, indem man eine sinnvolle Reihenfolge der `classpath`-Einträge wählt.

Der `classpath`-Inhalt kann über die system property: `java.class.path` in Java-Programmen abgefragt werden.

Zusammenhang zwischen classpath und Paketnamen:

Java-Klassen können in Paketen organisatorisch zusammengefasst werden, denen Verzeichnisse des Filesystems entsprechen. Pakete können *geschachtelt* werden, so dass ein Paket weitere Pakete enthält, was durch entsprechende *Unterverzeichnisse* abgebildet wird. Bei erforderlicher Angabe eines Paketnamens ist *stets der komplette, voll qualifizierte Paketname* anzugeben (inclusive aller übergeordneter Pakete), wie z.B. in `java.awt.event.MouseEvent`.

Es soll beispielsweise von der Java Laufzeitumgebung die Klasse `Rechnung.class` im Paket `betrieb.faktura` gefunden werden. Der Pfad zum relevanten Verzeichnis sei:

Windows: `C:\java\MyClasses\betrieb\faktura\Rechnung.class`

Linux/ Solaris: `java/MyClasses/betrieb/faktura/Rechnung.class`

Dann lauten der im `classpath` anzugebende *Pfad*, durch den die Klasse `Rechnung` z.B beim Aufruf von `java` gefunden wird und der voll qualifizierte *Paketname* der auszuführenden Klasse `Rechnung` wie folgt:

Windows:

```
java –classpath C:\java\MyClasses  betrieb.faktura.Rechnung
```

Linux/ Solaris:

```
java –classpath java/MyClasses  betrieb.faktura.Rechnung
```

Dadurch werden auch alle anderen von `Rechnung.class` benötigten Klassen gefunden, die sich im Paket betrieb.faktura befinden.

Wichtig ist die *komplette Paketangabe* `betrieb.faktura.Rechnung`. Eine verkürzte Angabe in der Form `faktura.Rechnung` *reicht nicht aus* – auch dann nicht, wenn der `classpath` bis hinunter auf `C:\java\MyClasses\betrieb` bzw. `java/MyClasses/betrieb` gesetzt wird. In diesem Fall würde die Klasse `Rechnung.class` *nicht gefunden* werden.

Der vollständig qualifizierte Paketname ist Teil des Inhalts einer Klasse und kann nur durch Veränderung des Klassencodings und Neukompilation verändert werden.

E3.3 Auffinden von Klassendateien

E3.3.1 Finden von .class-Dateien durch java (Java Launcher)

Durch das Tool `java` (den Java Launcher) wird die JVM gestartet. Dabei werden zur Programmausführung benötigte `.class`-Files in folgender Reihenfolge gesucht und geladen:

- **Bootstrap-Klassen**: Diese Klassen stellen die *Implementation der Standard-J2SE-Pakete (Java core platform)* und der mit ihr ausgelieferten Bibliotheken dar. Physisch installiert sind diese primär in der Datei `jre/lib/rt.jar`. Der Pfad zu den bootstrap Klassen wird in der system property: `sun.boot.class. path.system` gespeichert - und sollte in Java-Programmen tunlichst nicht umdefiniert werden. Zu den Bootstrap-Klassen gehören z.B. jene der Paket `java.lang` und `java.io` etc..
- **Extension-Klassen** (neuere Bezeichnung: Optional Packages): Dabei handelt es sich um Klassen, die den Funktionsumfang der Java-Plattform quasi *erweitern* und den *Java Extension Mechanismus* verwenden. Sie müssen in `.jar`-Files (oder `zip`-Files) zusammengefasst sein - und sind einfach im Verzeichnis `jdk1.5.0/jre/lib/ext` zu installieren. Jedes .jar-und .zip-Files in diesem Verzeichnis wird als Java-Extension betrachtet und durch den Java-Extension-Mechanismus geladen, als ob es sich um Bootstrap-Klassen handelte. Nicht archivierte `.class`-Files werden *nicht* berücksichtigt. Es können auch andere Extension-Verzeichnisse für installierte optionale Pakete festgelegt werden durch die system property: `java.ext.dirs`. Die darin genannten Verzeichnisse werden nach Extension-Klassen installierter optionaler Pakete durchsucht. Sind mehrere `.jar`-Files mit gleichnamigen Klassen vorhanden, so ist nicht definiert, welche Klasse wirklich geladen wird. Jedes `.jar`- oder `.zip`-File kann als Extension-Behälter dienen.
- **User-Klassen** (benutzerdefinierte Klassen): Alle auf Basisi der J2SE-Plattform (selbst-) entwickelten Klassen, die *nicht* Teil der Standard-Bootstrap-Klassen sind und *nicht* den Extension-Mechanismus nutzen. Auch von Drittanbietern oder OpenSource-Organisationen angebotenen Bibliotheken gehören dazu - und werden meist in Form von `.jar`-Files zur Verfügung gestellt.

Nur für User-Klassen ist der `classpath` (s.o.) vorzugeben, damit sie von der Java Laufzeitumgebung gefunden werden. Bootstrap-Klassen und Extension-Klassen werden automatisch und somit zuverlässig von der Laufzeitumgebung gefunden, ohne dass sie in den `classpath` einzutragen wären.

Anmerkung: Die von den Java-Entwicklungstools (`java`, `javac`, `javadoc`) benötigten Klassen sind in dem *separaten* Archiv `/lib/tools.jar` zusammengefasst. Um sie für Eigenentwicklungen zu verwenden, müssen sie in den `classpath` aufgenommen werden. Dagegen werden sie bei Aufruf der Entwicklungstools automatisch gefunden.

Dem Java launcher `java` kann als Option `–classpath` bzw. `–cp` eine Liste von Klassenpfaden mitgegeben werden. Alternativ kann durch die **Option `–jar`** der Ort eines `.jar`-Files angegeben werden, das die zur Ausführung der Anwendung benötigten User-Klassen enthält. Die Option `–jar` übersteuert alle anderen Klassenpfad-Vorgaben. Indiesem Fall *müssen wirklich alle* benötigten User Klassen in dem angegebenen Archiv-File *enthalten* sein.

E3.3.2 Finden von Klassen-Dateien durch `javac` und `javadoc`

Falls eine benötigte Klasse in Form des Quellcodes (`.java`-File) *und auch* des Bytecodes (`.class`-File) vorliegt, verwendet `javadoc` *stets* das Quellcodefile. Dagegen verwendet der Compiler `javac` stets das `.class`-File, es sei denn, dieses ist *älteren Datums* als das zugehörige `.java`-File. Alle *veralteten* `.class`-Files werden somit durch `javac` *neukompiliert*.

Der `classpath` wird von `javac` und `javadoc` standardmäßig sowohl nach Quellcode- als auch Bytecodefiles abgesucht. Jedoch existiert für beide Tools die Option `–sourcepath`, mittels der sich für *Quellcodefiles* eine vom `classpath` abweichender Suchpfad definieren lässt. Die Such nach `.class`-Bytecodefiles wird dadurch nicht beeinflusst.

E3.3.3 Sicherheitsmechanismen

In Java-Programmen referenzierte Klassen und Interfaces werden durch den *Java-Classloader* geladen; es ist auch möglich, eigene Classloader zu entwickeln.

Mit einem Classloader ist eine sogenannte *security policy* (Sicherheitsrichtlinie) verbunden. Diese legt fest, welche Rechte der betreffende Code auf der Ausführungsplattform besitzt (z.B. Zugriff auf das lokale Dateisystem etc.). Diese Sicherheitsrichtlinien beziehen sich jedoch *nur* auf Extension- und User-Klassen. Dagegen werden die J2SE-Bootstrap-Klassen (*core plattform classes* aus `rt.jar`) stets als *vertrauenswürdig* ("trusted") betrachtet und unterliegen nicht den Vorgaben der Sicherheitsrichtlinie. Wurde *keine* Sicherheitsrichtlinie implementiert (durch entsprechende Einträge in das Files `java.policy`), dann gelten *alle* geladenen Klassen als vertrauenswürdig.

Die J2SDK-Installation von Sun enthält ein vorgeferigtes Standard-Sicherheitsrichtlinien-File `java.policy`, das alle Extension-Klassen (*installierte* optionale Pakete) als ebenso vertrauenswürdig deklariert wie Bootstrap-Klassen und nur User-Klassen grundsätzlichen Sicherheitsrestriktionen unterwirft. Für Extension-Klassen werden durch den `java.policy`-Eintrag alle Sicherheitsrestriktionen explizit aufgeheoben. Nur diese "großzügigen" Rechte ermöglichen es installierten optionalen Paketen, auch native Code zu enthalten. Natürlich sollte man sich somit vor Installation und Verwendung optionaler Pakete genau überlegen, ob diese den erforderlichen Sicherheitsansprüchen der Installationsplattform auch wirlich genügen.

Hier ein (gekürzter) Auszug:

```
// Standard extensions get all permissions by default
grant codeBase "file:${{java.ext.dirs}}/*" {
```

```
   permission java.security.AllPermission;
};
// Default permissions granted to all domains
grant {
// Allows any thread to stop itself using the
// java.lang.Thread.stop() method that takes no argument.
// […] It is strongly recommended that you either remove this
// permission from this policy file or further restrict it to
// code sources that you specify, because Thread.stop() is
// potentially unsafe.
// See "http://java.sun.com/notes" for more information.
permission java.lang.RuntimePermission "stopThread";
// allows anyone to listen on un-privileged ports
permission java.net.SocketPermission "localhost:1024-", "listen";
// "standard" properies that can be read by anyone
permission java.util.PropertyPermission "java.version", "read";
permission java.util.PropertyPermission "java.vendor", "read";
permission java.util.PropertyPermission "java.vendor.url","read";
permission java.util.PropertyPermission "java.class.version","read";
permission java.util.PropertyPermission "os.name", "read";
permission java.util.PropertyPermission "os.version", "read";
permission java.util.PropertyPermission "os.arch", "read";
permission java.util.PropertyPermission "file.separator", "read";
permission java.util.PropertyPermission "path.separator", "read";
permission java.util.PropertyPermission "line.separator", "read";
// [...]
};
```

Natürlich kann die Default-Einstellung für Extension-Klassen (installierte optionale Pakete) durch Anpassen bzw. Hinzufügen von: `grant codeBase` Einträgen verändert werden. Zugriffsrechte optionaler Pakete sind *individuell* eingestellbar, indem deren Namen in `"file:${{java.ext.dirs}}/*"` statt des Platzhalters `*` genannt und Sicherheitsattribute aufgeführt werden.

Nicht-installierte optionale Pakete jedoch, die aus dem Netz geladen werden, erhalten nur dann Zugriff auf kritische Systemressourcen (z.B. Filesystemzugriff), wenn sie signiert sind, oder von einer als vertrauenswürdig deklarierten Quelle stammen.

E3.3.4 Extension-Klassen (Optional Packages)

Extension-Klassen werden in `.jar`- oder `.zip`-Files zusammengefasst. Sie enthalten typischerweise Java-Coding, können aber plattformspezifischen (native) Code (DLLs, Libs, Executables, ...) sowie Daten, Bilder oder weitere vom optionalen Paket ver-

wendete Ressourcen umfassen. Im professionellen Bereich stellen optionale Pakete die Implementierung einer grundlegenden API dar. Sun selbst stellt solche Pakete zur Verfügung – Beispiel sind `JavaServlet` und `Java3D`. Die meisten dieser "offiziellen" optionalen Pakete liegen im Paket-Namensraum `javax.*`, wobei das "x" für die ältere Bezeichnung Extension steht.

Man unterscheidet *zwei Arten* von optionalen Paketen:

- Installierte optionale Pakete (**installed optional packages**) sind im dafür vorgesehenen Verzeichnis der JDK-Verzeichnisstruktur abgelegte `.jar`- oder –`.zip`-Files. Diese können verwendet werden, als wären sie Teil der Standard-Bootstrap-Klassen, ohne sie in den `classpath` eintragen zu müssen. Den Klassen und `.jar`-Files optionaler Pakete haftet nichts Besonderes an. Einzig und allein der Ort ihrer Installation in `jre/lib/ext` macht aus einem `.jar`-File ein optionales Paket.
- Geladene optionale Pakete (**downloaded optional packages**) werden durch Einträge in der *Manifest-Datei* eines anderen `.jar`-Files, typischerweise eines Applets oder einer Anwendung *referenziert*. Bei dem Eintrag handelt es sich um das `Class-Path` Attribut der Manifest-Datei. Die Inhalte geladener optionaler Pakete können von den Klassen des darauf referenzierenden `.jar`-Files verwendet werden. Auf diese Weise kann das Coding von Anwendungen und Applets auf mehrere `.jar`-Files verteilt werden. Für geladene optionale Pakete gilt eine wichtige Einschränkung: Sie dürfen aus Sicherheitsgründen keinen plattformspezifischen (native) Code enthalten, genauer: Sie können *nicht* durch Java-Netzwerk-Code zur Laufzeit in die *laufende* JVM geladen werden, auch wenn der Code als vertrauenswürdig eingestuft wird. Optionale Paket mit native Code müssen somit fest *installiert* werden. Optionale Pakete, die nicht als `.jar`- sondern als `.exe`- oder `.bin`-File zusammengefasst wurden, werden aus Sicherheitsgründen *nicht* übers Netz geladen. Geladene optionale Pakete sollten durch ein Zertifikat signiert sein.

Wenn die JVM nach benötigten `.class`-Files sucht, so sucht sie zuerst innerhalb der Bootstrap-Klassen, danach in den Verzeichnissen der *installierten* optionalen Pakete und schlieslich im Bereich der *geladenen* optionalen Pakete. Nur wenn an all diesen Orten die gesuchte Klasse nicht zu finden ist, wird der `classpath` danach abgesucht.

Es werden lokal fest installierte optionale Paketinhalte bevorzugt verwendet, falls vorhanden. Falls nicht, so werden optionale Pakete übers Netz geladen.

Manifest-Datei:

Jedes `.jar`-File kann eine *Manifest-Datei* enthalten, die den Inhalt des `.jar`-Files auflistet und weitere spezifische Informationen enthält. Die Manifest-Datei des `.jar`-Files eines *installierten optionalen Pakets* sollte *Versions- und Herstellerinformationen* enthalten; diese wird eventuell von Applets benötigt, die auf das Paket zugreifen. Hier ein Beispiel eines solchen Manifest-Files [SUN05b]:

```
Manifest-version: 1.0
```

```
Extension-Name: javax.extension
Specification-Version: 1.0
Specification Vendor: Sun Microsystems, Inc.
Implementation-Vendor: Sun Microsystems, Inc.
Implementation-Vendor-Id: com.sun
```

Die Manifest-Informationen können vom *Java-Plug-In-Mechanismus* gelesen und genutzt werden, während er ein Applet im *Browser* ausführt; Browser wie der Internet Explorer oder der Netscape Navigator unterstützen optionale Pakte mittels Java Plug-Ins.

Die Manifest-Einträge des Applet-.jar-Files legen feste, welche optionalen Pakete welchen Ursprungs und welchen Ursprungs zusätzlich benötigt werden. Es kann festgestellt werden, ob die erforderlichen optionalen Pakete auf dem Rechner bereits vorhanden sind, korrekte Version tragen (nicht veraltet sind) und korrekten Hersteller-Ursprungs sind. Falls dies nicht der Fall ist, kann der Applet-Verwender durch das Java-Plug-In aufgefordert werden, die erforderlichen zusätzlichen Paket über das Netz laden zu lassen.

Installierte Optionale Pakete werden in den Speicher geladen, wenn sie verwendet werden und stehen dann allen Java-Anwendungen weiterhin (ohne erneutes Laden) zur Verfügung, die in der gleichen momentan aktiven JVM laufen.

Geladene optionale Pakete werden durch `Class-Path` Attributeinträge im Manifest-File des darauf zugreifenden .jar-Files referenziert. Ein `Class-Path` Eintrag könnte z.B. lauten:

```
Class-Path:  servlet.jar  infobus.jar  res/images/
```

Mehrere zu ladende .jar-Files können aufgezählt werden. Alle darin enthaltenen Klassen können nur von den Klassen *des* .jar-Files benutzt werden, in dessen Manifest-File dieser Eintrag aufgenommen wurde.

Der Ort, an dem die .jar-Files zu ladender optionaler Pakete liegen, spielt (im Gegensatz zu installierten optionalen Paketen!) keine Rolle. Einzig und allein durch seinen Eintrag im `Class-Path` Attribut der Manifest-Datei eines anderen .jar-Files, wird ein .jar-File zu einem geladenen optionalen Paket, nicht aufgrund der Ablage in einem speziellen Verzeichnis.

Die in dem `Class-Path` Attribut enthaltenen Einträge nennen .jar-Files oder Verzeichnisse nur *relativ* zum referenzierenden optionalen Paket, d.h. realtiv zum Netz- und Verzeichnisort (*code base*), von dem das referenzierende optionale Paket geladen wurde. Dies geschieht aus *Sicherheitsgründen*: Auf diese Weise ist garantiert, dass geladene optionale Pakete aus der gleichen Quelle stammen, wie das darauf referenzierende .jar-File.

Alle `Class-Path` Einträge, die nicht mit einem `'/'` enden, werden als .jar-File angesehen. Falls ein Eintrag eine ungültige Struktur hatt oder sich auf nicht vorhandene Ressourcen beziehen, wird er ignoriert.

Nur Applets und Anwendungen, die in .jar-Files zusammengefasst sind, können geladene optionale Pakete verwenden, da nur solche Applets und Anwendungen

über ein Manifest-File verfügen, aus dessen Einträgen geladene optionale Pakete referenziert werden.

Geladene optionale Pakete werden *erst* aus dem Netz geladen, wenn eine benötigte Klasse nicht schon im Bereich der Bootstrap-Klassen und installierten optionalen Pakete zu finden war, um unnötigen Netzverkehr zu vermeiden.

Der Extension-Mechanismus führt *keine Installation* geladener optionaler Pakete in der JDK-Verzeichnisstruktur durch. Geladene optionale Pakete werden somit auch nachdem sie aus dem Netz geladen wurden *nicht* zu installierten optionalen Paketen.

Für *weitere Informationen* zum Extension-Mechanismus verweisen wir auf die Sun-Dokumentation der J2SE 5.0, dort insbesondere auf:

`java.sun.com/j2se/1.5.0/docs/guide/extensions/spec.html` *und*

`java.sun.com/j2se/1.5.0/docs/guide/extensions/versioning.html`

Hier finden sich u.a. weitere Informationen zur Installation optionaler Pakete, dem *sealing*-Mechanismus (erzwingt die Verwendung von Paketklassen aus demselben `.jar`-File), Sicherheitsaspekten, Aufruf von Java-Installationsprogrammen für geladene optionale Pakete und J2SE-Klassen, die den Extension-Mechanismus unterstützen, wie z.B. `java.lang.ClassLoader` und `java.net.URLClassLoader`.

Abfrage von Umgebungsvariablen:

In der J2SE 5.0 existieren in `java.lang.System` zwei Methoden zur Abfrage der Umgebungsvariablen des Betriebssystems.

Die Methode: `public static String` **`getenv`**`( String name )`

liefert den Wert der Umgebungsvariablen `name` (ungleich `null`, sonst `NullPointerException`) als `String` zurück. Falls die Umgebungsvariable `name` auf dem System nicht definiert ist, wird `null` zurückgeliefert. Auf Unix muss `name` casesensitiv sein, auf Windows typischerweise nicht.

Die Methode: `public static Map<String, String>` **`getenv`**`()`

liefert eine nicht-veränderliche `String`-`Map` mit den aktuellen Umgebungsvariablen des Betriebssystems. Falls das System keine Umgebungsvariablen unterstützt, wird eine leere `Map` zurückgeliefert. Bei Abfrage eines Umgebungswertes, der nicht vom Typ `String` ist, wird eine `ClassCastException` geworfen.

Falls ein *Security Manger* existiert, wird in beiden Fällen aus Sicherheitsgründen dessen Methode `checkPermission()` gerufen und je nach Konfiguration eventuell eine `SecurityException` geworfen.