

Praktikum aus Softwareentwicklung 2, Stunde 14

Lehrziele/Inhalt

1. Java Service
2. Sicherheit

Java Service

Ein *Java Service* ist ein Interface, Objekte die dieses Interface implementieren können zur Laufzeit angefordert werden. Implementierungen eines Java Services (*Java Service Provider*) müssen das Service-Interface implementieren, in einer Jar-Datei liegen und dort als Service Provider registriert werden. Dazu muss man im Verzeichnis *META-INF/services/* eine Text-Datei mit dem Namen des Interfaces anlegen, als Inhalt der Datei muss der voll qualifizierte Name der Klasse angegeben werden. Es können in der Datei auch mehrere Klassen angegeben werden, diese müssen jeweils in einer eigenen Zeile stehen. Für Verwender gibt es seit Java 6 mit der Klasse *java.util.ServiceLoader* eine bequeme Möglichkeit Service Provider zu erzeugen. In Java Versionen vor 6 muss man die Dateien in *services* auslesen (zB mit *Class.getResourceAsStream*) und Service Provider über Reflection (zB mit *Class.forName* und *Class.newInstance*) anlegen.

Services können verwendet werden, um Anwendungen erweiterbar oder Teile davon austauschbar zu machen; oder um Anwendungen klar zu strukturieren. In der Praxis werden Java Services zB: in Java selbst eingesetzt, um XML-Provider und JDBC-Treiber einzubinden; und NetBeans nutzt Java Services als Basis des Plug-in-Mechanismus.

Hinweis! Als Services werden häufig Factories geliefert mit denen man dann die gewünschten Objekte erzeugen kann.

Beispiel

Abbildung 42 zeigt eine Beispiel-Klasse die Services für das Interface *Runnable* lädt und die *run*-Methoden der gefundenen Services ausführt. In Abbildung 43 ist ein Service abgebildet, das das Interface *Runnable* implementiert.

```
public class Executor {
    public static void main(String[] args) {
        ServiceLoader<Runnable> runnableLoader =
            ServiceLoader.load(Runnable.class);
        for (Runnable runnable : runnableLoader) {
            runnable.run();
        }
    }
}
```

Abbildung 42) Beispiel: Java Service, Executor lädt alle Services die Runnable implementieren und führt ihre run-Methode aus

HelloService.java:

```
package at.jku.ssw.psw2.javaservice.provider;

public class HelloService implements Runnable {
    @Override
    public void run() {
        System.out.println("Hello!");
    }
}
```

META-INF/services/java.lang.Runnable:

```
at.jku.ssw.psw2.javaservice.provider.HelloService
```

Abbildung 43) Beispiel Java Service: HelloService implementiert das Interface Runnable und ist im Verzeichnis META-INF/services/ registriert

Damit der Executor das Hello-Service findet muss das Hello-Service in eine Jar-Datei verpackt werden und die Jar-Datei in den Klassenpfad von Executor eingefügt werden. Jar-Dateien können mit dem Kommandozeilenwerkzeug *jar* erstellt werden. In unserem Beispiel mit dem Befehl: "*jar cf HelloService.jar -C bin/ .*" wenn wir davon ausgehen, dass die Klassen und das Verzeichnis META-INF in *bin* liegen.

Der Executor kann mit dem Befehl:

"java -cp .;HelloService.jar at.jku.ssw.psw2.javaservice.Executor" ausgeführt werden. Dabei gehen wir davon aus, dass der Klassenpfad von Executor das lokale Verzeichnis ist und die Datei *HelloService.jar* ebenfalls im lokalen Verzeichnis liegt. Liegen die Klassen oder die Jar-Datei woanders muss der erste Teil (".") bzw. der zweite Teil ("*HelloService.jar*") im Klassen-Pfad ("-cp") angepasst werden.

Sicherheit

Das Thema Sicherheit war vom Start an Bestandteil von Java. Wobei Sicherheit in Java bedeutet, dass der Code keinen Schaden anrichten darf. Auf Sprachebene werden dazu Bereichsprüfungen bei Arrays durchgeführt, Typ-Casts nur erlaubt wenn das gecastete Objekt den Typ des Casts erfüllt und auf Zeigerarithmetik verzichtet. Auf Klassenbibliotheksebene werden Zugriffe auf Ressourcen wie Dateien und das Netz über einen Sicherheitsmanager geschützt. Den Sicherheitsmanager kann man konfigurieren, wobei man Code Rechte zuteilt, das kann abhängig zB vom Speicherort oder der Signatur geschehen.

Das Sicherheitsmanagement verteilt sich auf: die Virtuell Maschine (Arrayzugriffe, Typ-Casts, Bytecode-Prüfung), den Klassenlader (lädt die Klassen), Sicherheitsmanager (prüft ob ein Zugriff erlaubt ist), Verschlüsselung (Codesignierung) und der Java Authentication and Authorization Service (Autorisierung von Benutzern).

Die Konfiguration des Sicherheitsmanagers (*java.lang.SecurityManager*) erfolgt über Policy-Dateien. Java liest standardmäßig zwei Policy-Dateien aus, eine im Verzeichnis der Java-Installation "*jre/lib/security/java.policy*" und eine im Basisverzeichnis des Benutzers "*.java.policy*". Zusätzlich kann man über das System-Property *java.security.policy* eine Policy-Datei angeben. Das funktioniert als Kommandozeilenparameter (zB: *java.-Djava.security.policy=MyPolicy.policy MyApp*) oder im

Programm (zB: `System.setProperty("java.security.policy", "MyPolicy.policy");`). Damit die Policy-Datei ausgewertet wird muss ein Sicherheitsmanager aktiv sein. Ein Sicherheitsmanager kann durch Angabe des System-Properties `java.security.manager` beim Programmstart installiert werden; oder programmatisch über `System.setSecurityManager(new SecurityManager());` gesetzt werden. Policy-Dateien und `SecurityManager` können aus Sicherheitsgründen nur gesetzt werden, wenn bisher kein `SecurityManager` installiert ist oder die aktive Policy das erlaubt (`policy.allowSystemProperty=true`).

Policy-Dateien

Die Policies werden in Text-Dateien abgelegt. Der Inhalt besteht aus *grant*-Einträgen, pro Eintrag kann der betroffene Code-Quelle und beliebig viele Permissions angegeben werden:

```
grant Codesource
{
    Permission_1;
    Permission_2;
}
```

Die *Codesource* besteht aus einer URL der Code-Basis und vertrauenswürdigen Zertifikaten:

```
grant
codebase codebase-URL
certificate-name
{
    ...
}
```

Eine Permission besteht aus einer Permission-Klasse, einem Zielwert und einer Aktion:

```
{
    permission className target action, ...;
    ...
}
```

Beispiel einer Policy-Datei:

```
grant codeBase "file:${java.home}/lib/ext/*" {
    permission java.security.AllPermission;
};
grant {
    permission java.lang.RuntimePermission "stopThread";
    permission java.net.SocketPermission "localhost:1024-", "listen";
    permission java.util.PropertyPermission "java.version", "read";
    permission java.util.PropertyPermission "java.vendor", "read";
};
grant {
    permission javax.crypto.CryptoPermission "DES", 64;
    permission javax.crypto.CryptoPermission "DESede", *;
};
grant codeBase "http://www.ssw.uni-linz.ac.at/classes/" {
    permission java.net.SocketPermission "*:1024-65535", "connect";
    permission java.io.FilePermission "${user.home}${/}-",
        "read,write,execute";
};
```

Regeln und Meta-Zeichen in Policy-Dateien

- Die Code-Basis ist eine URL, daher muss immer "/" und niemals "\" verwendet werden, zB: "file:/C:/bla/"
- Mit $\${...}$ kann auf System-Properties zugegriffen werden. Als Kürzel für $\${file.separator}$ ist $\${/}$ definiert.
- Pfad-Endungen haben folgende Bedeutung:
 - $\{/$: alle class-Dateien in der angegebenen URL
 - $\/*$: alle class- und jar-Dateien in der angegebenen URL
 - $\/-$: alle class- und jar-Dateien in der angegebenen URL und Unterverzeichnissen

Permissions

Damit Code sicherheitskritische Methoden aufrufen darf muss er die entsprechende Permission haben; oder eine Permission, die die gewünschte Permission impliziert. Beispiel: Will ein Programm die Datei `c:\autoexec.bat` lesen, dann muss es die Permission `java.io.FilePermission` (oder mehr zB `java.security.AllPermission`) mit dem Target `"file:/C:/${/}autoexec.bat"` (oder mehr zB: `"file:${/}*"` oder `"<<ALL_FILES>>"`) und der Action `"read"` (oder mehr zB: `"read,write"`) haben.

Die Permissions aus den Policy-Dateien werden zu Laufzeit auf Java-Objekte abgebildet. Alle Permissions erben von `java.security.Permission`, eine beispielhafte Auswahl von Permissions ist in Abbildung 44 gegeben.

<i>Permission</i>	<i>Target</i>	<i>Action</i>
<code>java.io.FilePermission</code>	Dateipfad <<ALL FILES>>	read, write, execute, delete
<code>java.net.SocketPermission</code>	Host:Portrange	accept, connect, listen, resolve
<code>java.util.PropertyPermission</code>	Name des Systemproperties	read, write
<code>java.lang.RuntimePermission</code>	createClassLoader setSecurityManager exitVM stopThread ...	
<code>java.net.NetPermission</code>	setDefaultAuthenticator setCookieHandler setResponseCache ...	
<code>java.awt.AWTPermission</code>	accessClipboard watchMousePointer readDisplayPixels ...	
<code>java.security.SecurityPermission</code>	getPolicy setPolicy ...	
<code>java.security.AllPermission</code>		

Abbildung 44) Beispielhafte Aufzählung von Security Permissions

Signieren von Jar-Dateien

Damit man Code in Jar-Dateien, unabhängig von ihrem Speicherort vertrauen kann ist es möglich Jar-Dateien zu signieren. Signierte Dateien haben die Vorteile vor nachträglichen Veränderungen geschützt zu sein und an ihren Urheber zuordenbar zu sein. Damit kann man zum Beispiel Firmeninternen Jar-Dateien alle Rechte geben und Code von außen einschränken.

Mit dem Kommandozeilenwerkzeug *jarsigner* kann man Jar-Dateien signieren, zB:
jarsinger demo.jar MyKeyStore dabei ist *demo.jar* die zu signierende Jar-Datei und *MyKeyStore* der zu verwendende Keystore.

Bevor man mit *jarsigner* arbeiten kann muss man einen Schlüssel im Keystore haben. Mit dem Kommandozeilenwerkzeug *keytool* kann man einen Schlüssel erstellen, zB: *keytool -genkey*.

Ablauf eine Berechtigungsprüfung

Wird eine Sicherheitskritische Methode (zB *System.exit*) aufgerufen, dann fragt diese den *SecurityManager* ob das erlaubt ist. Der *SecurityManager* untersucht den Methoden-Stack (Aktivierungssätze) und prüft ob jede Methode in der Aufrufkette zumindest eine Permission hat die diese Methode impliziert. Eine Methode hat die Permissions ihrer Klasse, einer Klasse sind Permissions über eine *ProtectionDomain* zugeordnet. Eine *ProtectionDomain* speichert die Herkunft (URL), die Zertifikate und eine Sammlung von Permissions (*PermissionCollection*).

Beispiel: *System.exit*

Die Methode *System.exit* ruft *Runtime.exit* auf:

```
public static void exit(int status) {
    Runtime.getRuntime().exit(status);
}
```

Runtime.exit prüft ob ein Sicherheitsmanager installiert ist, wenn ja prüft sie mit *checkExit* ob *exit* erlaubt ist.

```
public void exit(int status) {

    SecurityManager security = System.getSecurityManager();
    if (security != null) {
        security.checkExit(status);
    }
    Shutdown.exit(status);
}
```

SecurityManager.checkExit erzeugt ein Objekt der Klasse *RuntimePermission* und prüft diese *Permission* über *SecurityManager.checkPermission*.

```
public void checkExit(int status) {
    checkPermission(new RuntimePermission("exitVM."+status));
}
```

SecurityManager.checkPermission läuft über den Stack (*stack walk*) und prüft ob jede Methode am Stack zumindest eine Permission hat die *exit* impliziert.