Seminar

Softwareentwicklung

(Inside Java and .NET)

Dynamisches Laden und Binden in Java

Autor

Reinhard Stumptner

Inhalt

1. Der Lebenszyklus eines Java Typs

1.1 Laden, binden und initialisieren eines Typs

Die Java Virtual Machine macht Typen für laufende Programme durch laden, binden und initialisieren verfügbar. Laden ist der Prozess binäre Daten eines Typs in die virtuelle Maschine zu bringen.

Das Binden besteht aus drei Schritten:

- Verifikation (strukturelle Konsistenz, Bytecode)
- Vorbereitung (z.B. Methodentabelle aufbauen)
- Constant Pool auflösen (symbolische durch direkte Referenzen ersetzen)

Wurden diese Schritte erfolgreich abgeschlossen, werden den Klassenvariablen Initialwerte zugewiesen.

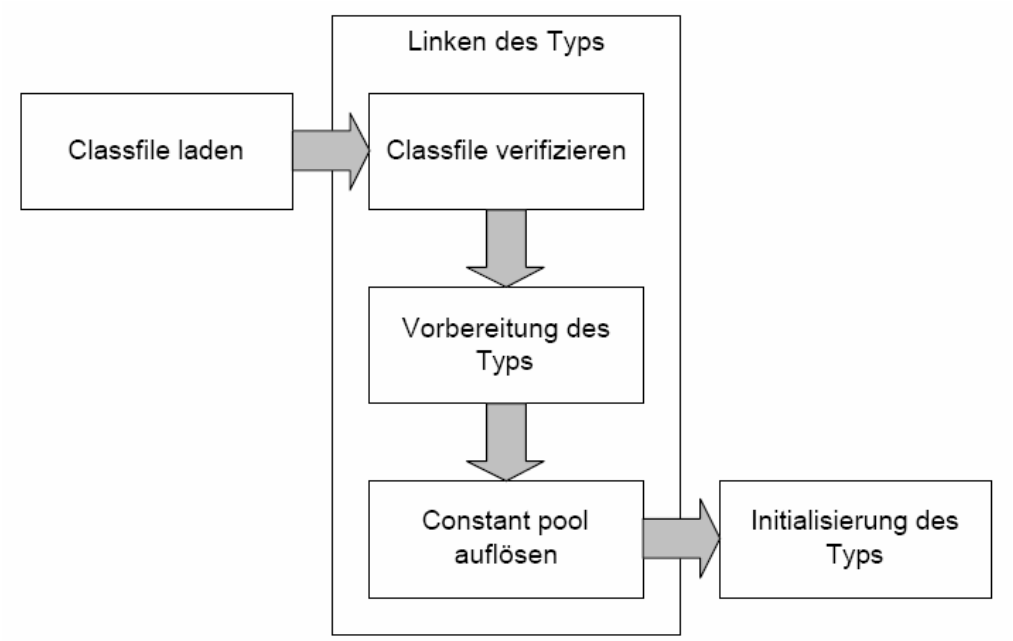


Abbildung 1: Das Anfangsstadium eines Typs

Jede Klasse bzw. jedes Interface muss vor der ersten aktiven Verwendung initialisiert werden. Was bedeutet „aktive Verwendung“? Man kann hierbei sechs Fälle unterscheiden:

- Erzeugen einer Instanz einer Klasse mit new(), implizite Erzeugung, Reflection, Cloning oder Deserialisierung
- Aufruf einer statischen Methode einer Klasse
- Zuweisung eines statischen Feldes einer Klasse (mit Ausnahme von Feldern mit der Eigenschaft „final“)
- Einbindung von Reflection
- Initialisierung einer Subklasse einer Klasse
- Die Klasse, die beim Programmstart die main() Methode enthält

Jede andere Verwendung eines Typs wird als „passiv“ bezeichnet und löst keine Initialisierung aus.

Wie man der obigen Liste entnehmen kann, bewirkt die Initialisierung einer Klasse, die Initialisierung der Superklasse, d.h. aller Superklassen. Diese Vorschrift trifft nicht auf Interfaces zu, diese werden erst bei der Verwendung eines ihrer nicht-konstanten Felder initialisiert. Bei der ersten aktiven Verwendung eines Typs muss dieser also Initialisiert werden, bevor dies geschehen kann, muss er erst vom Linker gebunden und davor vom Lader geladen werden.

1.1.1 Classfile laden

Der Ladeprozess besteht aus drei Hauptaktivitäten, nämlich erzeugen eines binären Datenstroms, der den Typ repräsentiert, umwandeln(parsen) dieser Daten in interne Strukturen in der Method Area und erzeugen einer Instanz von `java.lang.Class`, die den gewünschten Typ repräsentiert. Bei der Method Area handelt es sich um einen Speicherbereich für ausführbare Programmteile, enthält klassenweise Konstanten-Pool, Felder und Methoden-Daten und Code für Methoden und Konstruktoren. Dieser Bereich wird beim Starten der JVM erzeugt, ist von fester oder variabler Größe und nicht notwendigerweise zusammenhängend. Die `Class` – Instanz dient als Interface zwischen Programm und internen Datenstrukturen. Das Parsen in der Method Area und das Anlegen einer Instanz eines `Class` Objekts am Heap nennt man erzeugen eines Typs. Geladen wird entweder mit dem Bootstrap Class Loader, ein Teil der Java VM, oder mit user-defined `Class` Loaders. Der Bootstrap Class Loader ist dafür verantwortlich, dass die 'Kernel Classes' der Java API korrekt geladen werden. Er hat noch einige weitere Aufgaben wie z.B. das Laden von Array- Klassen. Der Lader wartet normalerweise nicht, bis eine Klasse aktiv verwendet wird, sondern er versucht, die binäre Repräsentation von Typen zu cachen, um sie früh oder zusammenhängende Gruppen laden zu können.

Es gibt, wie erwähnt, grundsätzlich zwei verschiedene Arten von Class Loadern:

- Der Bootstrap Class Loader ist ein Teil der Java Virtual Machine und ist dafür verantwortlich, dass die 'Kernel Classes' der Java API korrekt geladen werden. Er hat noch einige weitere Aufgaben wie z.B. das Laden von Array-Klassen.
- Benutzerdefinierte Class Loader können explizit vom Programmentwickler durch das Class Loader Interface implementiert werden. In der Regel handelt es sich bei diesen selbst definierten Class Loader um Subklassen der abstrakten Klasse `java.lang.ClassLoader`. Class Loader werden in Anwendungen eingesetzt, um die Eigenschaften des dynamischen Ladens der JVM noch zu erweitern (z.B. können Klassen über ein Netzwerk herunter geladen, „on-the-fly“ generiert oder aus einer verschlüsselten Datei extrahiert werden).

1.1.2 Classfile verifizieren

Nachdem ein Typ geladen wurde, muss sichergestellt werden, dass er der Semantik von Java entspricht. Wie beim Laden hat der Entwickler auch bei der Verifikation gewisse Flexibilität, wie und wann Typen zu verifizieren sind.
Ein Teil der Verifikationsphase, die eigentlich Teil des Bindens ist, passiert bereits beim Laden, nämlich wenn der binäre Datenstrom auf korrektes Format (erwartete Länge, Komponenten am richtigen Platz, …) geprüft wird. Weiters muss sichergestellt werden, ob jede Klasse (außer Object) eine Superklasse besitzt, die geladen worden musste.
Ein weiterer Teil der Verifikationsphase ist die Prüfung der symbolischen Referenzen. Das dynamische Binden beinhaltet das Auffinden von einem Typ symbolisch referenzierter Klassen, Interfaces, Felder und Methoden. Diese Referenzen sind im Constant Pool gespeichert und müssen nun durch direkte ersetzt werden. Für jedes dieser symbolisch referenzierten Entities muss sichergestellt werden, dass es existiert, und wenn dem so ist, dass die benötigten Zugriffsrechte gewährt sind. Dieser Prozess ist Teil der dritten Phase des Bindens, gehört aber logisch auch zur Verifikation. Die Auflösung der Referenzen kann bis zur ersten Verwendung verzögert werden und sogar nach der Initialisierung stattfinden.
Während der „offiziellen“ Verifikationsphase wird alles geprüft, was noch nicht geprüft wurde oder später nicht mehr geprüft werden wird. Folgende Überprüfungen werden mit hoher Wahrscheinlichkeit in dieser Phase durchgeführt:

- Von final Klassen wurde nicht geerbt
- Final Methoden wurden nicht überschrieben
- Keine inkompatiblen Methodendeklarationen (zwei Methoden unterscheiden sich nur in deren return Typen)
- Einträge im Constant Pool sind untereinander konsistent
- Wohlgeformtheit spezieller Einträge im Constant Pool (Klassen-, Methodennamen, …)
- Integritätsprüfung des Bytecodes

Die Verifikation des Bytecodes ist wohl die komplizierteste Angelegenheit. Alle Java VM müssen die Integrität des Codes für jede Methode, die ausgeführt wird, garantieren. Wenn ein Jump beispielsweise an eine Stelle nach dem Ende einer Methode führt, darf das nicht zum Absturz führen, sondern muss vorher erkannt und gemeldet werden. Diese Überprüfung sollte in der Verifikationsphase durchgeführt werden, da dies dem laufenden Programm einen bedeutenden Speedup gibt.

1.1.3 Vorbereiten des Typs

Wurde eine Klasse erfolgreich geladen und verifiziert, kann sie nun vorbereitet werden. Hierbei wird für die Klassenvariablen Speicher reserviert und, abhängig vom Datentyp, meist mit „0“ initialisiert. Die Initialisierung mit deren tatsächlichen Anfangswerten erfolgt erst in der Initialisierungsphase, da bei der Vorbereitung kein Java Code ausgeführt wird. Zur Steigerung der Performanz laufender Javaprogramme werden zusätzliche Datenstrukturen im Speicher angelegt. Ein Beispiel dafür ist die Methodentabelle, die für jede Methode einer Klasse Zeiger auf den Methodenbeginn im Speicher enthält, einschließlich derer in Superklassen.

1.1.4 Constant Pool auflösen

Das Constant Pool ist die Laufzeit-Darstellung pro Klasse bzw. pro Interface. Es ist eine Tabelle `constant_pool` im Java `class` File und enthält verschiedenste Arten von Konstanten, numerische Literale die zur Übersetzungszeit bekannt sind, Methoden- und Feld-Referenzen auflösbar erst zur Laufzeit. Diese Tabelle entspricht in etwa den Symboltabellen bei konventionellen Programmiersprachen.

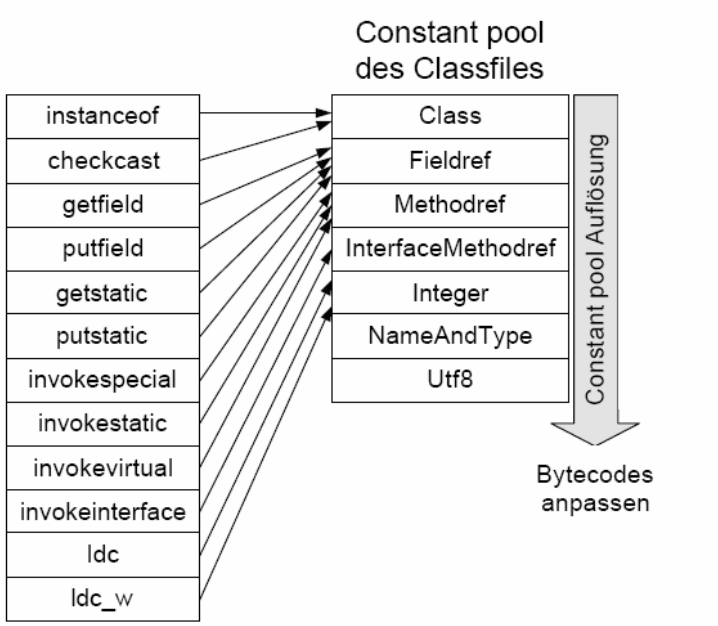


Abbildung 2: Constant Pool Auflösung

1.1.5 Initialisieren des Typs

Nun, in der Letzten Phase, muss die Klasse bzw. das Interface noch mit den Werten initialisiert werden, die der Programmierer als Startwerte für die Klassenvariablen vorgesehen hat. Man unterscheidet zwei Arten von Initializern, Class Variable Initializer und Static Initializer. Ein Beispiel für einen Class Variable Initializer wäre etwa:

```
class ClassInit {
     static int i=3*5*Math.random();
}
```

Ein Static Initializer sieht so aus:

```
class StaticInit {
     static int i;
     static {
          i=13*Math.random();
     }
}
```

1.1.5.1 Die <clinit> Methode

Beide Initializer werden in einer speziellen Methode gespeichert, der class (interface) initialization method „<clinit>“. Diese Art von Methoden kann nur von der Java VM aufgerufen werden und besteht aus zwei Schritten:

- Initialisierung der direkten Superklasse (wenn diese nicht bereits initialisiert ist)
- Ausführen von <clinit>, falls vorhanden

Wird eine Klassenvariable mit einem konstanten Wert initialisiert, so kann das vom Compiler erledigt werden, es wird kein Code für <clinit> erzeugt.
Bsp.:

```
class InitKonst {
     static int i=10;
     static final int j=1;
}
```

Bevor eine Klasse initialisiert wird, wird immer die direkte Superklasse initialisiert, d.h. für Object erfolgt die Initialisierung immer als erstes, gefolgt von den Klassen abwärts in der Vererbungshierarchie. Die Initialisierung eines Interfaces erfordert nicht die Initialisierung der Superinterfaces. Laufen mehrere Threads, kann nur einer eine Klasse initialisieren, dazu später noch mehr.

1.5.1.2 Aktive vs. Passive Verwendung

Das folgende kleine Beispiel soll den Unterschied zwischen aktiver und passiver Verwendung illustrieren:

```
class A {
     static int x=10*Math.random();
     static {System.out.println(“init A”);}
}

class B extends A {
     static int y=20*Math.random();
     static {System.out.println(“init B”);}
}

class C {
     static { System.out.println(“init C”);}
     public static void main(String[] args) {
          int z=B.x;     // aktive Verwendung von A,
                         // passive Verwendung von B
          System.out.println(“finished”);
     }
}
```

Ausgabe:

```
init C
init A
finished
```

Vor dem Ausführen von main() der Klasse C wird dieselbe geladen, bei der Instruktion “int z=B.x;” wird auf ein Feld einer fremden Klasse verwiesen, diese muss nun geladen werden. B erbt das Feld “x” von der Klasse A, wird

von B nur das Feld x verwendet, ist es nicht nötig, A und B zu laden, es genügt lediglich A zu laden.

2 Der Lebenszyklus eines Objekts

Nachdem eine Klasse erfolgreich geladen, gebunden und initialisiert wurde, kann sie verwendet werden, das heißt, es kann auf statische Felder zugegriffen werden, statische Methoden können ausgeführt werden und es ist möglich, Instanzen dieser Klassen zu erzeugen. Das Erzeugen einer Instanz stellt den Beginn des Lebenszyklus eines Objekts dar, Garbage Collection (bzw. `finalize()`) dessen Ende.

2.1 Instanzierung einer Klasse

Klassen können explizit oder implizit instanziert werden. Explizite Instanzierung kann auf vier Arten geschehen:

- `new()`
  ```
  A  obj = new A();
  ```
- `newInstance()`
  ```
  Class  myClass=Class.forName(ClassName);
  A  obj = (A) myClass.newInstance(ClassName);
  ```
- `clone()`
  ```
  A  obj1 = new A();
  A  obj2 = obj1.clone();
  ```
- Deserialisieren eines Objekts mit `getObject()`, enthalten in `java.io.ObjectInputStream`

Zusätzlich gibt es nun zahlreiche Situationen, in denen Objekte implizit instanziert werden.
Die String Objekte, die der main Methode (`void main(String() args)`) übergeben werden, möchte ich hierbei als Beispiel nennen. Weiters wird beim Laden einer Klasse ein `Class` Objekt erzeugt, beim Auflösen des Constant Pool beispielsweise werden String Objekte erzeugt.
Wenn die Java VM eine neue Instanz einer Klasse erzeugt, wird zuerst für ihre Instanzvariablen Speicher am Heap reserviert. Die Reservierung von Speicher für Instanzvariablen einer Klasse erfolgt für die Klasse selbst und für alle ihre Superklassen. Beim Allokieren von Speicher für die Variablen am Heap, werden den Variablen die default Initialwerte zugewiesen. Nach Abschluss dieses Prozesses werden alle Instanzvariablen mit den richtigen Startwerten versehen. Hierbei unterscheidet man drei Techniken, je nachdem, wie ein Objekt erzeugt wurde. Bei einem `clone()` werden die Werte der Variablen des geklonten Objekts kopiert, wurde via `readObject()` deserialisiert, wird der Wert vom InputStream gelesen. Erfolgte die Erzeugung des Objekts jedoch mit `new()`, wird dessen „instance initialization method“ (`<init>`) ausgeführt.
Der Java Compiler erzeugt für jede Klasse zumindest eine `<init>()`Methode, nämlich die des default Konstruktors, die den no-arg Konstruktor der Superklasse aufruft. Beginnt der Konstruktor mit dem expliziten Aufruf eines anderen Konstruktors (`this()`) derselben Klasse, besteht `<init>` aus dem Aufruf dieses Konstruktors gefolgt von den restlichen Anweisungen. Beinhaltet der Konstruktor keinen solchen Aufruf von `this()`, wird `<init>()` der Superklasse ausgeführt (außer bei Object),

danach alle Initializer der Instanzvariablen und wieder alle restlichen Instruktionen. Tritt beim Ausführen des Konstruktors ein Fehler auf, kann dieser vom Rufer nicht abgefangen werden, sondern der Rufer muss ebenfalls mit einem Fehler enden.

Beispiel:

```
public class Konstruktoren {
      public String name;
      public int min=0;
      public int max=10; //„=0“, „=10“: Initializer der
Instanzvariablen

      public Konstruktoren() {   //default Konstruktor
          // Aufruf des default Konsruktors von Object
          // Ausführen von „min=0“, „max=10“ =
          //     Initializer der Instanzvariablen
          name="Konstruktor";
      }

      public Konstruktoren(String name) {
          // Aufruf des default Konsruktors von Object
          // Ausführen von „min=0“, „max=10“ =
          //     Initializer der Instanzvariablen

          this.name=name;
      }

      public Konstruktoren(String name, int min) {
          this(name); // kein Aufruf des Konsruktors von Object
          this.min=min;
      }

      public Konstruktoren(String name, int min, int max) {
          this(name,min);
          this.max=max;
      }
}

public class SubKonst extends Konstruktoren{
      public double base=0.5;

      public SubKonst() {   //default Konstruktor
          // Aufruf des default Konsruktors von Konstruktoren
          // Ausführen von „base=0.5“ =
          //     Initializer der Instanzvariablen
      }



      public SubKonst (String name, int min, int max, double base) {
          super(name,min,max);
          // Ausführen von „base=0.5“ =
          //     Initializer der Instanzvariablen
          this.base=base;
      }
}
```

2.2 Garbage Collection und Finalisierung von Objekten

Programme können Speicher für Objekte am Heap reservieren, aber nicht explizit freigeben. Diese Aufgabe übernimmt der Garbage Collector. Besitzt eine Klasse eine `finalize()` Methode, wird diese vom Garbage Collector, vor Freigeben des Speichers des Objekts dieser Klasse, ausgeführt. Exceptions, die von einem solchen Finalizer geworfen werden, werden ignoriert.

2.3 Freigeben eines Typs

Die JVM lädt, bindet und initialisiert Klassen, so dass sie von Programmen verwendet werden können, natürlich müssen sie auch wieder freigegeben werden, wenn sie nicht mehr benötigt werden, d.h. wenn sie über keine Referenz eines Objekts der Applikation erreichbar sind. Wäre dem nicht so, würde die Method Area kontinuierlich wachsen. Typen, die vom BOOTSTRAP Loader geladen wurden, sind immer erreichbar, es können also nur solche von der VM freigegeben werden, die von einem benutzerdefinierten Class Loader geladen wurden, freigegeben werden. Eine `Class` Instanz ist erreichbar, wenn entweder ein Programm diese explizit referenziert oder die Repräsentation eines Objekts am Heap in der Method Area verweist auf diese Instanz. Jedes Objekt am Heap besitzt eine Art Zeiger auf seine Typ-Information in der Method Area. Von den Typinformationen ausgehend kann die JVM die `Class` Instanzen der betroffenen Klasse, sowie aller Superklassen und Superinterfaces auffinden.

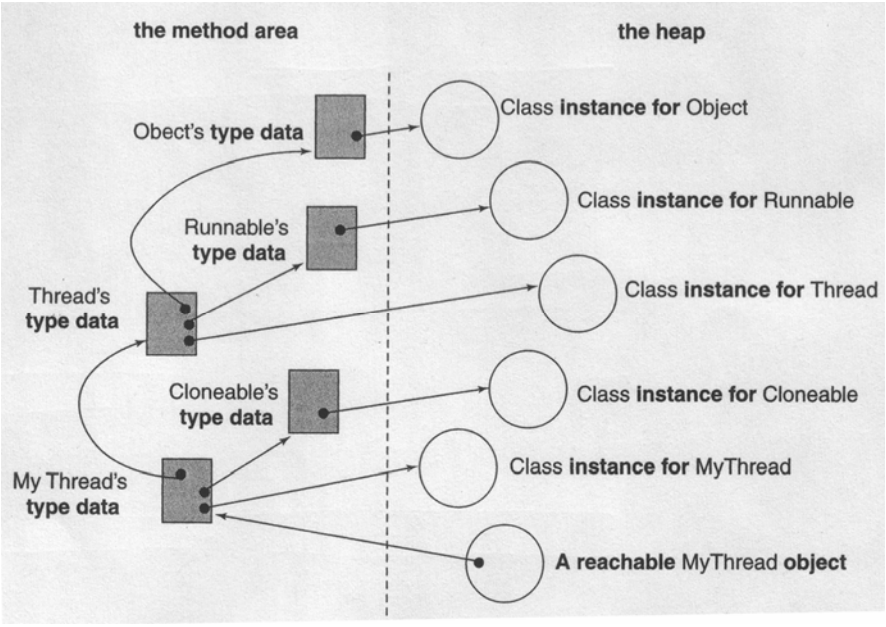


Abbildung 3: Erreichbarkeit von Objekten/Typen

Ist nun eine Referenz zur Instanz von `MyThread` bekannt, kann der Garbage Collector `Cloneable`, `Thread`, `Runnable` und `Object` erreichen.

3 Dynamisches Binden („The Linking Model“)

3.1 Auflösen symbolischer Referenzen

Wie bereits erwähnt, können in Java benutzerdefinierte Lader entworfen werden, diese ermöglichen das dynamische Laden und Binden von Klassen und Interfaces, die zur Compilezeit unbekannt sind. Beim Kompilieren eines Programms werden `.class` - Files erzeugt, die über symbolische Referenzen miteinander verbunden sind. Diese symbolischen Referenzen werden im Konstantenpool (Constant Pool) der Klasse gespeichert.
Bevor ein laufendes Programm eine symbolische Referenz benutzen kann, muss diese aufgelöst werden, d.h. die symbolische Referenz wird durch eine direkte ersetzt, diesen Prozess nennt man „constant pool resolution“. Instruktionen der JVM benutzen symbolische Referenzen durch die Verwendung des Index dieser Referenz im Konstantenpool. Die Instruktion, die eine solche Referenz benutzt, gehört natürlich zu einer Klasse, die gerade ausgeführt wird. Jeder Eintrag im Konstantenpool wird nur einmal aufgelöst, verschiedene Instruktionen auch verschiedener Methoden können sich auf denselben Eintrag (die nun direkte Referenz) beziehen.
Der Prozess des Bindens von Klassen beinhaltet nicht nur das Auflösen der Referenzen, die benutzte Klasse muss weiters auf Korrektheit und vorhandene Zugriffsberechtigung überprüft werden. Stößt die JVM beispielsweise auf eine `getstatic` – Instruktion, die sich auf ein Feld einer anderen Klasse bezieht, werden folgende Überprüfungen ausgeführt:

- Die andere Klasse existiert
- Es bestehen die benötigten Zugriffsrechte auf die Klasse
- Das bezeichnete Feld existiert
- Das Feld hat den erwarteten Datentyp (die symbolische Referenz beinhaltet den Typ des Feldes)
- Es bestehen die benötigten Zugriffsrechte auf das Feld
- Das Feld ist statisch

Man unterscheidet grundsätzlich frühe und späte Resolution (Auflösung). Ist ein Programm, bevor es ausgeführt wird, komplett gebunden, spricht man von früher Resolution, geschieht das Binden im Gegensatz erst in letzter Minute, wird dies als späte Resolution bezeichnet. Hierbei ist natürlich auch ein Mittelweg denkbar. Der Programmierer hat, auch bei früher Auflösung, den Eindruck einer Auflösung zur Laufzeit, da Fehler, die dabei auftreten, erst gemeldet werden, wenn die Referenz benutzt wird. Benutzt das Programm eine referenzierte, fehlerhafte Klasse nicht, wird auch kein Fehler gemeldet, was durchaus Sinn macht.

3.2 Resolution und dynamische Erweiterung

Die Architektur von Java erlaubt es nicht nur, Typen zur Laufzeit zu binden, Applikationen können sogar zur Laufzeit entscheiden, welche Typen gebunden werden sollen, hierbei spricht man von dynamischer Erweiterbarkeit. Es gibt im Prinzip zwei Mechanismen, die das Einbinden von im Source Code nicht spezifizierten Typen erlauben. Die einfachere Möglichkeit bietet zwei Varianten einer `forName()` - Methode aus `java.lang.Class`:

- `public static Class forName (String className) throws ClassNotFoundException;`

- `public static Class forName (String className, boolean initialize, ClassLoader loader) throws ClassNotFoundException;`

Die 3-Parameter Methode wurde in der Version 1.2 hinzugefügt.
Ist `initialize true`, wird die Klasse geladen, gebunden und initialisiert, bevor `forName()` endet, anderenfalls wird die Klasse nicht explizit initialisiert. Mit `loader` wird der Class Loader spezifiziert, der den Typ laden soll, ist der Parameter `null`, wird die Klasse vom Bootstrap Class Loader geladen. Die 1-Parameter `forName()` - Methode benutzt immer den Lader, der die rufende Klasse geladen hat. Das Ergebnis ist entweder die Instanz der Klasse oder, im Fehlerfall, eine `ClassNotFoundException`.
Einen anderen Weg des dynamischen Erweiterns bietet `loadClass()` aus `java.lang.ClassLoader`:

- `protected Class loadClass (String name) throws ClassNotFoundException;`
- `protected Class loadClass (String name, boolean resolve) throws ClassNotFoundException;`

Durch den `resolve`- Parameter kann man bestimmen, ob die Klasse geladen und auch gleich gebunden werden soll. Im Normalfall sollte man das Timing beim Binden der JVM überlassen und die 1-Parameter Version von `loadClass()` verwenden.
Der Unterschied zwischen den beiden Ansätzen ist nun, dass `forName()` die Klasse lädt und bindet, während es bei `loadClass()` offen ist, ob auch gebunden werden soll.
Benutzerdefinierte Lader (`java.lang.ClassLoader`) werden vor allem dann verwendet, wenn das Laden von Typen auf speziellen Wegen, wie herunterladen über ein Netzwerk, extrahierten aus einer Datenbank oder aus verschlüsselten Dateien, usw., passiert. Ein weiterer Aspekt bei der Verwendung von benutzerdefinierten Ladern ist die Sicherheit. Typen könnten z.B. in „protection Domains“ geladen werden, darauf wird hier nicht näher eingegangen.
Wenn bei der Auflösung des Konstantenpool festgestellt wird, dass ein Typ geladen werden muss, dann wird hierfür derselbe Class Loader benutzt, der den referenzierenden Typen geladen hat.

3.3 Lader und das „Parent-Delegation Model“

Wenn ein Lader dem Parent-Delegation Model folgend einen Typ zu laden versucht, gibt er diesen Auftrag immer an seine Superklasse weiter. Am Ende dieser Aufrufkette steht für gewöhnlich der Bootstrap Class Loader. Das Ergebnis dieses Ladeprozesses ist immer eine Instanz der Klasse `Class` die die geladene Klasse repräsentiert. Der Class Loader der den Ladevorgang anstößt, das heißt einen anderen Class Loader mit dem Ladevorgang beauftragt, heißt initialisierender Class Loader, den der die Klasse wirklich lädt nennt man definierenden Class Loader.
Ein Class Loader L kann also eine Klasse C erzeugen, indem er das selbst erledigt oder die Anfrage an einen anderen Class Loader weitergibt. Zur Laufzeit wird eine Klasse durch ihren Namen und den definierenden Class

Loader bestimmt. Klasse und definierender Class Loader gehören zu genau einem so genannten „Runtime Package“.

Die JVM benutzt jeweils eine von 3 Möglichkeiten, eine Klasse C, die durch ihren Namen N gekennzeichnet wird, zu erzeugen:

1. Wenn die Klasse D, die die Erzeugung der Klasse C ausgelöst hat, vom Bootstrap Class Loader definiert wurde, wird auch C vom Bootstrap Class Loader initialisiert.
2. Wenn die Klasse D, von einem benutzerdefinierten Class Loader definiert wurde, wird auch die Klasse C von einem benutzerdefinierten Class Loader initialisiert.
3. Array-Klassen werden ausschließlich durch die JVM erzeugt (nie durch einen Class Loader).

Im Folgenden wird beschrieben, wie eine Klasse C, die durch den Namen N gekennzeichnet und mit dem Bootstrap Class Loaders geladen wird. Zunächst wird von der JVM überprüft, ob der Bootstrap Class Loader der initialisierende Class Loader der Klasse, die durch N beschrieben wird, ist. Ist dies der Fall, heißt das, dass die Klasse C schon erzeugt wurde und somit keine weitere Bearbeitung nötig ist. Ist der Bootstrap Class Loader nicht der initialisierende Class Loader, so wird eine der folgenden Aktionen ausgeführt:

- Die JVM sucht nach einer Repräsentation von C. Geschieht dies erfolgreich, leitet der Bootstrap Class Loader eine Klasse, die durch N gekennzeichnet wird, von der Repräsentation ab.
- Der Bootstrap Class Loader leitet den noch nicht bearbeiteten Ladevorgang von C zu einem benutzerdefinierten Class Loader L weiter, indem der Klassenname N als Argument an die Methode `loadClass()` von L übergeben wird. Die JVM hält dabei fest, dass der Bootstrap Class Loader initialisierender Class Loader von C ist.

Ähnlich funktioniert der Ladevorgang bei benutzerdefinierten Class Loadern. Es wird die Methode `loadClass()` des Class Loader L mit dem Argument N, dem Klassennamen, aufgerufen. Der Rückgabewert der Methode ist die erzeugte Klasse C. Die JVM merkt sich nun, dass der L der initialisierende Class Loader von C ist. Wenn `loadClass()` aufgerufen wird, muss L eine der beiden folgenden Aktionen ausführen, damit C erfolgreich geladen werden kann:

- L kann ein Byte-Array erstellen, das C als `ClassFile`-Struktur repräsentiert. Danach muss die Methode `defineClass()` der Klasse `ClassLoader` aufgerufen werden. Dadurch versucht die JVM eine Klasse, deren Name N ist, mit dem Class Loader L von dem Byte-Array abzuleiten.
- Der Class Loader L leitet den noch nicht behandelten Ladevorgang von C zu einem anderen Class Loader weiter und gibt die Klasse C zurück.

3.4 Constant Pool Resolution

Wenn eine Klasse geladen wird, wird versucht, die binäre Form für den Code einer Klasse oder eines Interfaces zu finden. Wie erwähnt, ist das Ergebnis ein Class Object das die Klasse oder das Interface repräsentiert. Beim Binden werden diese binären Daten in der Laufzeitumgebung der virtuellen Maschine

kombiniert und können dann ausgeführt werden. Schließlich muss die Klasse noch initialisiert werden, dabei werden die statischen Initialisierer und die Initialisierer der Klasse ausgeführt. Die meisten dieser Prozesse sind Operationen auf einen Konstantenpool, eine Laufzeitdatenstruktur für jede Klasse, die mit den Symboltabellen herkömmlicher Programmiersprachen vergleichbar sind. JVM Instruktionen holen sich ihre Operanden vom Konstantenpool. Klassen, Methoden und Felder, egal ob sie von JVM Instruktionen oder von Einträgen im Konstantpool referenziert werden, „benutzen“ den Konstantenpool. Diese Referenzen sind anfangs immer symbolisch, da die Adressen, an denen sich referenzierte Typen zur Laufzeit befinden, zur Kompilezeit noch nicht bekannt sind. Zur Laufzeit müssen alle symbolischen Referenzen durch direkte ersetzt werden. Das dynamische Auffinden dieser konkreten Werte nennt man Constant Pool Resolution. Die Auflösung des Konstantenpools kann das Laden, Binden und Initialisieren ein oder mehrerer Typen beinhalten. Es gibt einige verschiedene Arten von Einträgen im Konstantenpool, bei denen sich der Auflöseprozess im Detail unterscheidet. Instruktionen, die einen Eintrag in den Konstantenpool referenzieren, sind für dessen Auflösung verantwortlich. Einträge, die von anderen Einträgen im Konstantenpool referenziert werden, werden aufgelöst, wenn der referenzierende Eintrag aufgelöst wird. Nachdem jeder Eintrag im Konstantenpool von beliebig vielen Instruktionen referenziert werden kann, kann es vorkommen, dass der Eintrag bereits aufgelöst wurde, das Ergebnis ist immer das Entity, dass von der ersten Auflösung erzeugt wurde. Die Auflösung des Konstantenpools wird normalerweise von einer Instruktion initiiert die einen Eintrag im Konstantenpool referenziert und ausgeführt wird. Beim Binden gibt es einige zusätzliche Einschränkungen, die überprüft werden müssen. Z.B. wird bei der `getfield` Operation nicht nur der Eintrag im Konstantenpool benötigt, zusätzlich muss etwa überprüft werden, ob das Feld nicht statisch ist, anderenfalls wird eine Exception geworfen. Diese Exceptions sind Subclasses von `VirtualMachineError`.

3.4.1 Resolution von Klassen und Interfaces

Ein Konstantenpooleintrag der als `CONSTANT_Class` gekennzeichnet ist, repräsentiert eine Klasse oder ein Interface. Ein `CONSTANT_Class` Eintrag muss immer vor einer `CONSTANT_Methodref` aufgelöst werden. Der Auflöseprozess hängt immer davon ab, um welchen Typ vom `CONSTANT_Class` es sich handelt. Array Klassen werden anders behandelt als Non-Array Klassen oder Interfaces. Das `name_index` Item eines `CONSTANT_Class` Eintrags referenziert `CONSTANT_Utf8` und repräsentiert den eindeutigen Namen der Klasse oder des Interfaces. Hierbei gibt es einige Unterscheidungen:

- Ist das erste Zeichen des Namens kein „[“ handelt es sich um eine Referenz auf eine Non-Array Klasse oder ein Interface.
  - Wenn die aktuelle Klasse nicht von einem Class Loader geladen wurde, wird der Eintrag „gewöhnlich“ aufgelöst.
  - Wenn die aktuelle Klasse von einem Class Loader geladen wurde, wird benutzerdefinierter Code zur Auflösung ausgeführt.
- Wenn das erste Zeichen ein „[“ ist, handelt es sich um eine Array Klasse, die speziell aufgelöst werden muss.

Die aktuelle Klasse wurde nicht von einem Class Loader geladen:
Auflösung einer Non-Array Klasse C

1. C und ihre Superklassen werden geladen
    a. Wurde C noch nicht geladen, sucht die JVM nach `C.class` und versucht sie zu laden. Natürlich gibt es keine Garantie, dass `C.class` auch die erwartete Klasse C enthält oder `C.class` überhaupt ein valides `class` File ist. Weiters ist es möglich, dass C zwar schon geladen aber noch nicht initialisiert wurde. In dieser Phase müssen folgende Fehler aufgespürt werden:
        - `NoClassDefFoundError`:
            - Es wurde keine Datei mit entsprechenden Namen gefunden
            - Keine wohlgeformte `class` Datei
            - Datei enthält nicht die erwartete Klasse
        - `ClassFormatError`:
            - Die Klasse spezifiziert keine Superklasse und ist nicht die Klasse `Object`
    b. Wurde die Superklasse der Klasse, die geladen wird, noch nicht geladen, wird rekursiv bei Schritt 1 fortgesetzt.

2. Wenn der Ladeprozess für C und die Superklassen erfolgreich war, kann C gebunden und initialisiert werden.

3. C wird gebunden, verifiziert und vorbereitet:
    a. Zuerst wird die binäre Datenstruktur von C auf strukturelle Korrektheit geprüft. Der Verifikationsprozess selbst kann das Laden von Klassen oder Interfaces auslösen (aber nicht Initialisierung).
        - Ist die binäre Struktur nicht korrekt, wird ein `VerifyError` geworfen.
    b. War die Verifikation erfolgreich, wird die Klasse vorbereitet. Bei der Vorbereitung werden die statischen Felder der Klasse erzeugt und mit den standard Defaultwerten initialisiert. Hierbei werden nicht die statischen Initialisierer ausgeführt, das heißt es muss kein Java Code ausgeführt werden.
        - Enthält eine nicht abstrakte Klasse eine abstrakte Methode, wird ein `AbstractMethodError` geworfen.

4. Die Klasse wird initialisiert:
    - `ExceptionInInitializerError`
        - Wirft der Initializer eine Exception, wird diese durch eine `ExceptionInInitializerError` Exception ersetzt und weitergegeben.

5. Schließlich werden Zugriffsrechte überprüft:
    - `IllegalAccessError`

- Exception wird geworfen wenn eine Klasse A, die ursprünglich `public` deklariert wurde, auf `private` geändert wird, nachdem eine andere Klasse B, die A referenziert, kompiliert wurde.

Wenn kein Fehler entdeckt wurde, war die Constant Pool Resolution erfolgreich. Wenn bei den Schritten 1 bis 4 eine Exception geworfen wird, muss die Klasse als nicht benutzbar markiert und verworfen werden.
Wird bei Schritt 5 eine Exception geworfen, ist die Klasse trotzdem gültig und benutzbar, in beiden Fällen endet die Auflösung jedoch fehlerhaft.

Die aktuelle Klasse wurde von einem Class Loader geladen:
Auflösung einer Non-Array Klasse C

Wenn eine Klasse mit einem Class Loader geladen wird und eine andere Klasse referenziert, dann wird für diese derselbe Class Loader verwendet, indem die `loadClass()` Methode mit dem Pfad der zu ladenden Klasse aufgerufen wird. Das Ergebnis dieser Methode ist die aufgelöste Klasse. Jeder Class Loader wird von der abstrakten Klasse Class Loader abgeleitet. Diese Subclasses ergänzen dabei die Art und Weise, wie eine Klasse dynamisch geladen wird. Class Loader können Klassen aus verschiedensten Quellen, nicht nur aus Dateien, erzeugen. Eine Klasse kann z.B. über ein Netzwerk runter geladen oder aus einer verschlüsselten Datei erzeugt werden. Der Aufruf der `loadClass()` Methode bewirkt, dass die Klasse geladen und optional gebunden und initialisiert wird:

Gewünschte Klasse wurde bereits geladen:

- Wenn ein Class Loader eine Klasse laden, aber nicht binden soll, die bereits geladen wurde, dann wird die geladene Klasse einfach zurückgegeben.
- Soll der Class Loader eine Klasse laden, binden und initialisieren, die bereits geladen, aber noch nicht gebunden wurde, dann lädt er die Klasse keine weiters mal, sondern bindet sie nur.
- Ist die Klasse bereits geladen, gebunden und initialisiert, wird sie einfach zurückgegeben.

Gewünschte Klasse wurde noch NICHT geladen:

- Der Class Loader kann ein Array von Bytes anlegen, die das `class` File repräsentieren. Darauf wird die Methode `defineClass` angewandt, um diese Bytes zu einer Klasse oder zu einem Interface zu konvertieren. `defineClass` kann einen rekursiven Aufruf von `loadClass` bewirken, um Superklassen der neu definierten Klasse zu laden. Diese Information ist in einem `super_class` Eintrag gespeichert. Die Superklasse wird dabei nie sofort gebunden und initialisiert.
- Der Class Loader kann auch die statische Methode `findSystemClass` mit dem eindeutigen Namen der Klasse oder des Interfaces aufrufen. Die resultierende Klasse wird nicht als von einem Class Loader geladen gekennzeichnet.

Nachdem eine Klasse oder ein Interface mit den zugehörigen Superklassen erfolgreich geladen wurde, kann sie, oder auch nicht, gebunden und initialisiert werden. Dies ist über den zweiten Parameter von `loadClass` einstellbar. Wurde der Ladevorgang von einem Eintrag im Constant Pool angestoßen, wird die Klasse immer gebunden. Der Linking - Prozess wird durch Aufruf der `resolveClass` - Methode eingeleitet. Der Bindeprozess an sich funktioniert genauso wie beim Binden ohne Class Loader. Nachdem alle Superklassen geladen wurden, wird der Bytecode der zu ladenden Klasse verifiziert. Ist dies erfolgreich, kann die Klasse vorbereitet und initialisiert werden. Schließlich werden noch Zugriffsrechte auf die Klasse überprüft.

3.4.2 Resolution von Array Klassen

Wie bereits beschrieben, werden Array Klassen im Constant Pool durch eine „[“ im `name_index` gekennzeichnet. Die Dimensionen der Arrays können durch die Anzahl solcher „[“ bestimmt werden. Im field descriptor steht der Basistyp der Array Klasse. Um eine Array Klasse aufzulösen, müssen folgende Schritte durchgeführt werden:

1. Anzahl der Dimensionen und Basistyp aus dem field descriptior auslesen.
2. Basistyp:
    a. Repräsentiert der field descriptor einen primitiven Datentyp (erstes Zeichen ist kein „L“), ist dieser Typ der Basistyp.
    b. Handelt es sich um einen Referenztypen, muss dieser aufgelöst werden, bevor er verwendet werden kann.
3. Wurde bereits eine Array Klasse mit gleichem Basistypen und gleicher Anzahl von Dimensionen geladen, wird diese verwendet.

3.4.3 Resolution von Feldern und Methoden

Ein `CONSTANT_Fieldref`- Eintrag repräsentiert eine Klassen- oder Instanzvariable oder eine Konstante eines Interfaces. `CONSTANT_Methodref` kennzeichnet eine statische oder eine nicht-statische Methode einer Klasse. Referenzen zu Interface- Methoden werden in einem `CONSTANT_Interfacemethodref`- Eintrag vermerkt. Damit Referenzen auf Felder oder Methoden aufgelöst werden können, muss vorher die zugehörige Klasse erfolgreich aufgelöst worden sein, d.h. jede Exception, die bei der Auflösung der Klasse geworfen wird, muss auch bei den entsprechenden `CONSTANT_Fieldref` und `CONSTANT_Methodref` geworfen werden.
Symbolische Referenzen zu Klassenvariablen werden durch einen Zeiger auf dessen Wert, enthalten in den Typinformationen der Klasse in der Method Area, aufgelöst. Mit statischen Methoden funktioniert es sehr ähnlich. Auch hier gibt es in der Method Area die benötigten Informationen, um die Methode aufzurufen.
Direkte Referenzen zu Instanzvariablen eines Objekts entsprechen einem Offset. Variablen sind im Image des Objekts zu finden und Methoden in der Methodentabelle. Die Methodentabelle enthält auch jene Methoden, die von Superklassen geerbt wurden.

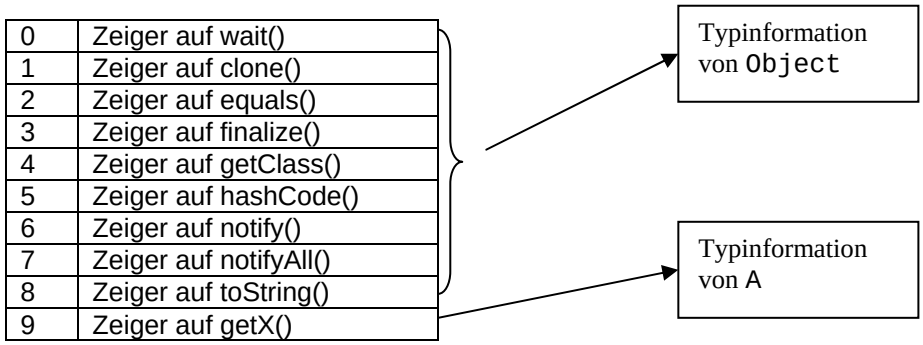
Beispiel:

```
class A {
     int x;
     String s=“Hello“;
     public void getX(){return x;}
}
```

Layout der A- Instanz:

0	Zeiger in Method Area
1	x
2	s

Methodentabelle von A:



_quick Anweisungen:
Operanden, die auf Einträge im Konstantenpool verweisen, warden durch `_quick` aufgelöst. Eine `ldc` (load double Constant) Anweisung beispielsweise wird durch `ldc_quick` ersetzt, genauso bei `invokevirtual`.

Exceptions beim Auflösen einer Feldreferenz:

- Feld wurde nicht gefunden: `NoSuchFieldError`
- Benötigte Zugriffsrechte existieren nicht: `IllegalAccessError`

Exceptions beim Auflösen einer Methodenreferenz:

- Entsprechend: `NoSuchMethodError`
- und entsprechend: `IllegalAccessError`

3.4.4 Resolution von Interface Methoden

Eine `CONSTANT_InterfaceMethodref` wird in ein Maschinenabhängiges Format gebracht und nicht weiter behandelt, es wird auch keine Exception geworfen.

3.4.5 Resolution von Strings

`CONSTANT_String` kennzeichnet im Konstantenpool eine Instanz eines Strings (`java.lang.String`). Die Unicode Zeichen finden sich in `CONSTANT_Utf8`. Gleiche String Literale verweisen bei Java auf dieselbe Instanz der Klasse `String`. Das Abbilden einer Zeichenkette auf die entsprechende String- Referenz kann auch die Methode `intern()` übernehmen:

```
(“a“+“b“+“c“+“d“).intern()==“abcd“    // =true
```

Die JVM interpretiert jeden `CONSTANT_String` als eine Folge von Unicode-Zeichen und löst ihn folgendermaßen auf:

- Wenn die gleiche Zeichenfolge früher bereits aufgelöst wurde, wird das Ergebnis dieser Auflösung geliefert.
- Wurde `intern()` früher auf das gleiche Wort angewandt, ist auch hier das Ergebnis das gleiche.
- Ansonsten wird eine neue String- Instanz erzeugt.

Bei der String- Auflösung tritt normalerweise keine Exception auf.

3.4.6 Resolution von anderen Elementen im Constant Pool

`CONSTANT_Integer`, `CONSTANT_Long`, `CONSTANT_Float` und `CONSTANT_Double` haben Werte, die im Konstantenpool direkt dargestellt werden können. Auch hier tritt kein Fehlerfall ein. `CONSTANT_NameAndType` und `CONSTANT_Utf8` werden nicht direkt aufgelöst, sie können nur von anderen Einträgen im Konstantenpool referenziert werden.

3.4.7 Beispiel: Salutation- Applikation

```
class Salutation {
      private static final String hello=“Hello, world!“;
      private static final String greeting=“Greetings, planet!“;
      private static final String salutation=“Salutations, orb!“;
      private static int choice=(int)(Math.random()*2.99);

      public static void main(String[] args) {
            String s =hello;
            if (choice==1) s=greeting;
            else if (choice==2) s=salutation;
            System.out.println(s);
      }
}
```

Beim Initialisieren wird sichergestellt, dass alle Superklassen von Salutation initialisiert wurden.

Der Konstantenpool wird vom Compiler erzeugt:

1	CONSTANT_String_info	30
2	CONSTANT_String_info	31
3	CONSTANT_String_info	39
4	CONSTANT_Class_info	37
5	CONSTANT_Class_info	44
6	CONSTANT_Class_info	45
7	CONSTANT_Class_info	46
8	CONSTANT_Class_info	47
9	CONSTANT_Methodref_info	7, 16
10	CONSTANT_Fieldref_info	4, 17
11	CONSTANT_Fieldref_info	8, 18
12	CONSTANT_Methodref_info	5, 19
13	CONSTANT_Methodref_info	6, 20
14	CONSTANT_Double_info	2.99
16	CONSTANT_NameAndType_info	26, 22
17	CONSTANT_NameAndType_info	41, 32
18	CONSTANT_NameAndType_info	49, 34
19	CONSTANT_NameAndType_info	50, 23
20	CONSTANT_NameAndType_info	51, 21
21	CONSTANT_Utf8_info	"()D"
22	CONSTANT_Utf8_info	"()V"
23	CONSTANT_Utf8_info	"(Ljava/lang/String;)V"
24	CONSTANT_Utf8_info	"([Ljava/lang/String;)V"
25	CONSTANT_Utf8_info	"<clinit>"
26	CONSTANT_Utf8_info	"<init>"
27	CONSTANT_Utf8_info	"Code"
28	CONSTANT_Utf8_info	"ConstantValue"
29	CONSTANT_Utf8_info	"Exceptions"
30	CONSTANT_Utf8_info	"Greetings, planet!"

Abbildung 4.1: Der Konstantenpool von Salutation

31	CONSTANT_Utf8_info	"Hello, world!"
32	CONSTANT_Utf8_info	"I"
33	CONSTANT_Utf8_info	"LineNumberTable"
34	CONSTANT_Utf8_info	"Ljava/io/PrintStream;"
35	CONSTANT_Utf8_info	"Ljava/lang/String;"
36	CONSTANT_Utf8_info	"LocalVariables"
37	CONSTANT_Utf8_info	"Salutation"
38	CONSTANT_Utf8_info	"Salutation.java"
39	CONSTANT_Utf8_info	"Salutations, orb!"
40	CONSTANT_Utf8_info	"SourceFile"
41	CONSTANT_Utf8_info	"choice"
42	CONSTANT_Utf8_info	"greeting"
43	CONSTANT_Utf8_info	"hello"
44	CONSTANT_Utf8_info	"java/io/PrintStream"
45	CONSTANT_Utf8_info	"java/lang/Math"
46	CONSTANT_Utf8_info	"java/lang/Object"
47	CONSTANT_Utf8_info	"java/lang/System"
48	CONSTANT_Utf8_info	"main"
49	CONSTANT_Utf8_info	"out"
50	CONSTANT_Utf8_info	"println"
51	CONSTANT_Utf8_info	"random"
52	CONSTANT_Utf8_info	"salutation"

Abbildung 4.2: Der Konstantenpool von Salutation

Verifikation:

- Bytecode ist syntaktisch korrekt
- Salutation entspricht der Java Semantik
- Salutation wird die JVM nicht zum Absturz bringen

`<clinit>()`:

```
 0 invokestatic #13 <Method double random()>
 3 ldc2_w #14 <Double 2.99>
 6 dmul     // Multiplikation
 7 d2i      // Konvertierung: double → int
 8 putstatic #10 <Field int choice>
11 return
```

Hier wird das `choice`- Feld mit dem Anfangswert versehen. `invokestatic #13` enthält eine symbolischr Referenz auf `random()` der Klasse `java.lang.Math`. Das bewirkt, dass `java.lang.Math` geladen und gebunden wird. Das Ergebnis von `random()` wird auf den Stack gelegt.

`invokestatic` wird durch `invokestatic_quick` ersetzt. `ldc2_w #14` wird durch den Wert `2.99` aufgelöst und durch `ldc2_w_quick` ersetzt. Auch `2.99` wird nun auf den Stack gelegt. Danach werden beide Werte entnommen, multipliziert und das Ergebnis wird wieder auf den Stack gelegt. wandelt den `double`- Wert zu `int` um. `putstatic #10` referenziert einen `CONSTANT_Fieldref_info` Eintrag, die `choice`- Variable, im Konstantenpool. Die Einträge werden angepasst und `putstatic` wird durch `putstatic_quick` ersetzt.

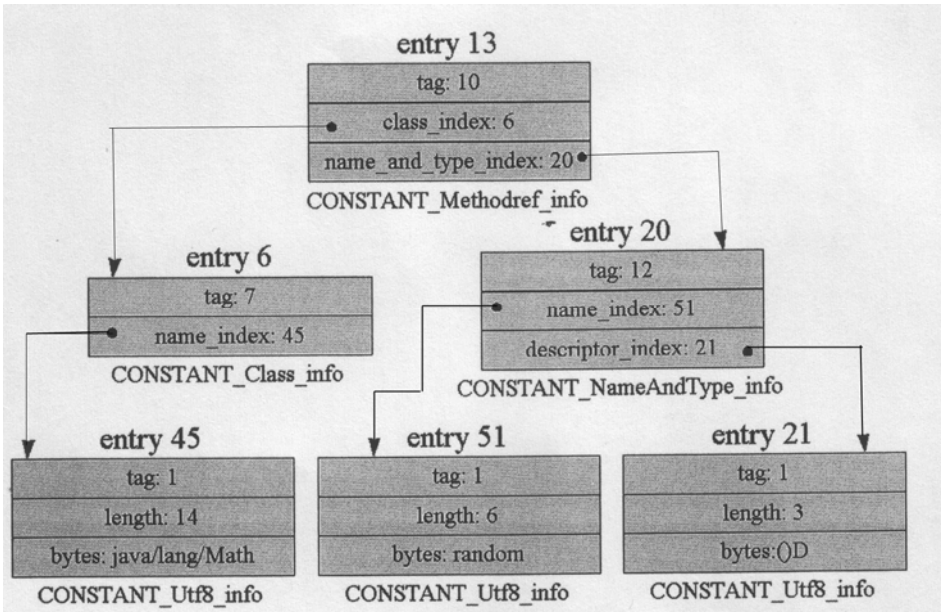


Abbildung 5: Symbolische Referenzen von Salutation zu Math.random()

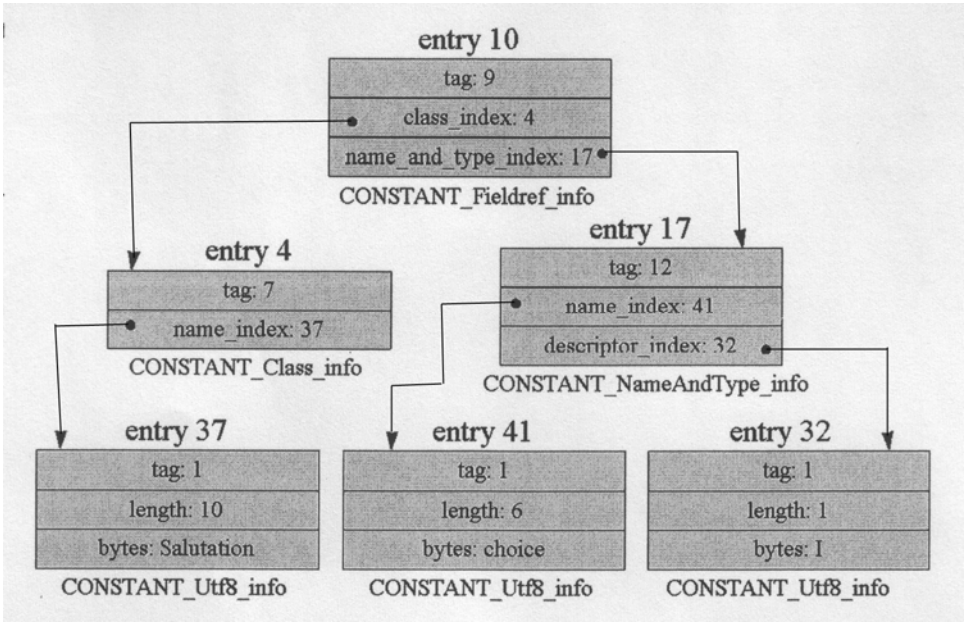


Abbildung 6: Symbolische Referenz zum `choice`- Feld

```
main():
      0 ldc #2 <String "Hello, world!">
      2 astore_1       // speichert Referenz in 1. lokalen Variable
                       // s = hello;
      3 getstatic #10 <Field int choice>
      6 iconst_1       // push 1
      7 if_icompne 16  // if (choice == 1) (choice sei hier =1)
     10 ldc #1 <String "Greetings, planet!">
     12 astore_1       // s = greeting;
     13 goto 26
     16 getstatic #10 <Field int choice>
     19 iconst_2       // push 2
     20 if_icompne 26  // if (choice == 2)
     23 ldc #3 <String "Salutations, orb!">
     25 astore_1       // s = salutation;
     26 getstatic #11 <Field java.io.Printstream out>
     29 aload_1        // push s für System.out.println(s);
     30 invokevirtual #12 <Method void println(java.lang.String)
     33 return
```

Auflösen der symbolischen Referenzen:

- `ldc #2 <String "Hello, world!">`
  - symbolische Referenz zu einem `CONSTANT_String_info` Eintrag
  - String Objekt mit dem Wert "Hello, world!" wird erzeugt und im Konstantenpool referenziert
  - `ldc` → `ldc_quick`
  - Referenz des Strings wird auf den Stack gelegt
- `astore_1`
  - String entnehmen und in lokaler Variable(s) speichern
- `getstatic #10 <Field int choice>`
  - Eintrag Nr. 10 wurde bereits bei `<clinit>()` aufgelöst
  - `getstatic` → `getstatic_quick`
- `iconst_1`
- `if_icompne 16`
- `ldc #1 <String "Greetings, planet!">`
- `astore_1`
- `goto 26`
- `getstatic #10 <Field int choice>`
  - bereits aufgelöst
  - `getstatic` → `getstatic_quick`
- `iconst_2`
- `if_icompne 26`
- `ldc #3 <String "Salutations, orb!">`
  - String Objekt erzeugen
  - Referenz bei CP Eintrag 3 vermerken
  - `ldc` → `ldc_quick`
- `astore_1`
- `getstatic #11 <Field java.io.Printstream out>`
  - `java.lang.System` muss geladen und gebunden werden
  - `CONSTANT_Fieldref_info`, 11. Eintrag im Pool
  - Prüfung auf Vorhandensein eines statischen Feldes `out`
  - direkte Referenz zum Feld wird installiert
- `aload_1`
- `invokevirtual #12 <Method void println(java.lang.String)>`
  - `CONSTANT_Methodref_info`
  - `java.io.PrintStream` wird geladen und gebunden

- Prüfung: Methode ist `public`, gibt `void` zurück und hat einen `String` als Eingangsparameter
- Beim Ausführen der Methode werden wertere Typen geladen werden müssen

Während der Ausführung von Salutation wurde `10 ldc #1 <String "Greetings, planet!">` nie erreicht, die symbolische Referenz daher nicht aufgelöst (d.h. kein String- Objekt erzeugt).

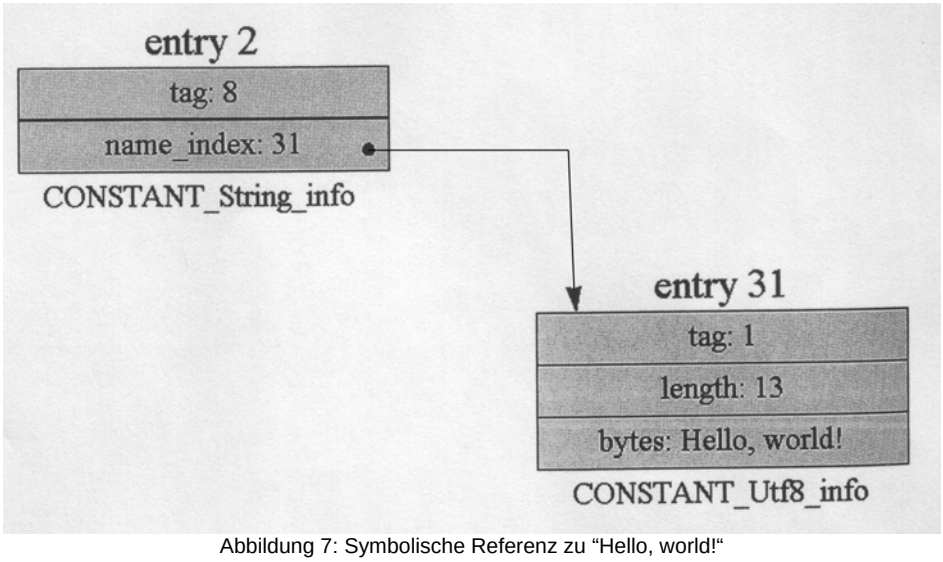


Abbildung 7: Symbolische Referenz zu “Hello, world!“

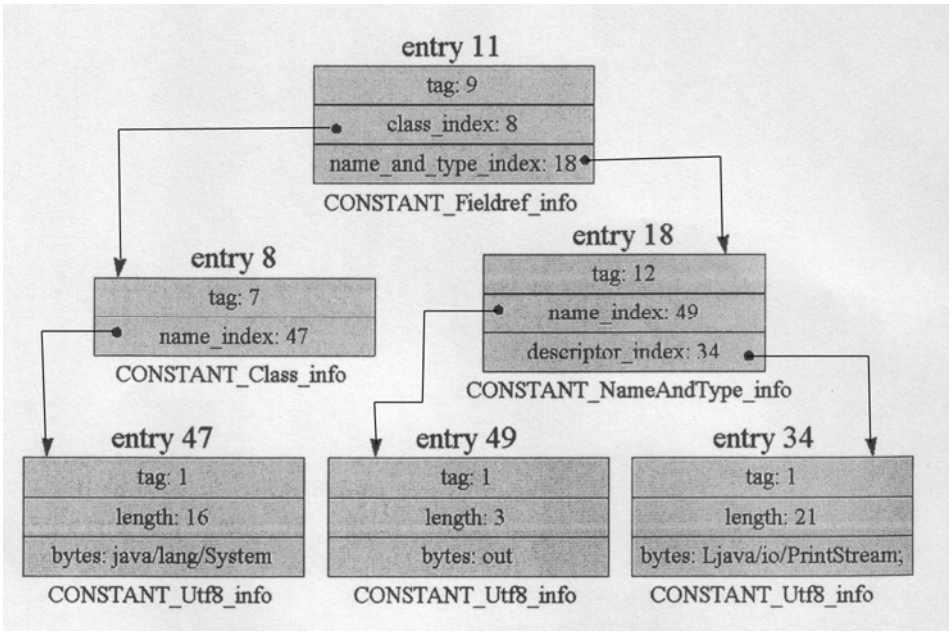


Abbildung 8: Symbolische Referenz zu `System.out`

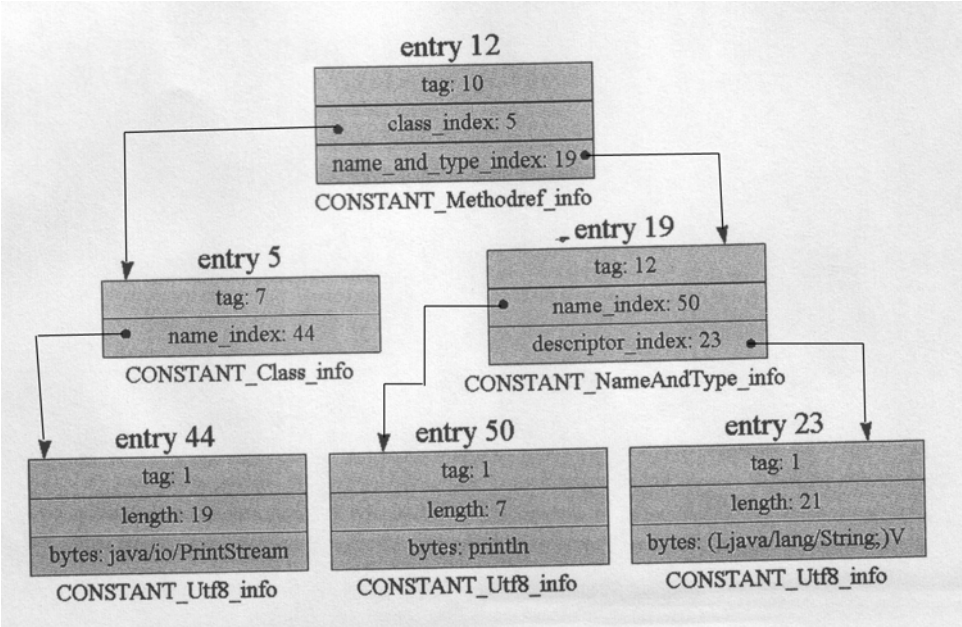


Abbildung 9: Symbolische Referenz zu `PrintStream.println()`

4 Benutzerdefinierte Lader

4.1 „Gewöhnlicher“ Lader

```
class MyLoader extends ClassLoader {
    public Class findClass(String name) throws ClassNotFoundException {
        Class   newClass=searchLoadedType(name); //class may be loaded already
        if (newClass==null){
            byte[] classData=null;
            try {
                classData = loadClassData(name);
            }
            catch (Exception e) {
                System.err.println(e);
                e.printStackTrace();
            }
            newClass=defineClass(name, classData, 0, classData.length);
        }
        String   superClass=(newClass.getSuperclass()).getName();
        if (superClass!=null && superClass!="java.lang.Object") findClass(superClass);
        return newClass;
    }
    //read data for .class file
    private byte[] loadClassData(String name) throws Exception {
        File source=new File("<<PATH>>\\"+name+".class");
        BufferedInputStream in=new BufferedInputStream(new FileInputStream(source));
        byte[] b=new byte[in.available()];
        in.read(b, 0, in.available());
        return b;
    }
    public Class searchLoadedType(String name){
        return findLoadedClass(name);
    }
}
```

4.2 Dekodierender Lader

Die Bytes der .class Dateien wurden zuvor verschlüsselt (file[i] XOR key), bis auf decryptClassData() sieht der Lader aus, wie zuvor.

```
Public byte[] decryptClassData(String name, byte key){
 byte[] b=null;
 try {
     File source=new File(“<<PATH>>\\”+name+”.class”);
```

```
    BufferedInputStream in=new BufferedInputStream(new FileInputStream(source));
    b=new byte[in.available()];
    in.read(b, 0, in.available());
    for (int i=0; i<b.length; i++) {
        b[i]=(byte)(b[i] ^ key);
    }
    in.close();
}
catch (Exception e) {
    System.out.println(e);
}
return b;
   }
```

4.3 Kompilierender Lader

Dieser Lader kompiliert eine Java Datei und liefert anschließend die Class Instanz der kompilierten Klasse.

```
Public class CompilingClassLoader extends ClassLoader {

  private byte[] getBytes(String filename) throws IOException {
    File file = new File(filename);
    long len = file.length();
    byte raw[] = new byte[ (int) len];
    FileInputStream fin = new FileInputStream(file);
    int r = fin.read(raw);
    if (r != len) {
      throw new IOException("Can't read all lines, " + r + " != " + len);
    }
    fin.close();
    return raw;
  }


  private  oolean compile(String javaFile) throws IOException {
    System.out.println("CCL: Compiling " + javaFile + "...");
    Process p = Runtime.getRuntime().exec("javac " + javaFile);
    try {
      p.waitFor();
    }
    catch (InterruptedException ie) {
      System.out.println(ie);
    }
    int ret = p.exitValue();
    return ret == 0;
  }

  public Class loadClass(String name,  oolean resolve) throws ClassNotFoundException {
    Class c = null;
    c = findLoadedClass(name);
    String fileStub = name.replace('.', '/');
    String javaFilename = fileStub + ".java";
    String classFilename = fileStub + ".class";
    File javaFile = new File("<<PATH>>\\"+javaFilename);
    File classFile = new File("<<PATH>>\\"+classFilename);
    if (javaFile.exists() && (!classFile.exists() ||
        javaFile.lastModified() > classFile.lastModified())) {
      try {
        if (!compile("<<PATH>>\\"+javaFilename) || !classFile.exists()) {
          throw new ClassNotFoundException("Compile failed: " + javaFilename);
        }
      }
      catch (IOException ie) {
        throw new ClassNotFoundException(ie.toString());
      }
    }
    try {
      byte raw[] = getBytes(classFilename);
      c = defineClass(name, raw, 0, raw.length);
    }
    catch (Exception e) {
        System.out.println(e);
    }
    if (c == null) {
      c = findSystemClass(name);
    }
```

```
    if (resolve && c != null) {
      resolveClass(c);
    }
    if (c == null) {
      throw new ClassNotFoundException(name);
    }
    return c;
  }
}
```

5 Literatur

Primär

- Bill Venners : Inside the Java 2 Virtual Machine, McGraw Hill, 1999, Kap.7+8
- Frank Yellin, Tim Lindholm: The Java Virtual Machine Specification, Addison-Wesley 1997, Kap.5

Sekundär

- http://www.javaworld.com/javaworld/javaqa/2003-06/01-qa-0606-load.html
- http://www.javageeks.com/Papers/ClassForName/ClassForName.pdf
- http://www.javaworld.com/javaworld/javaqa/2003-03/01-qa-0314-forname.html
- nibbler.tk.informatik.tu-darmstadt.de/ LectureNotes/ws0203/GdI1/GdI-V7.pdf
- www.informatik.uni-bremen.de/agbs/lehre/ws0203/pi1/ - 33k
- www.inf.uni-konstanz.de/cgip/lehre/ info2_03/vorlesung/reflection.pdf
- www.ifi.unizh.ch/groups/richter/ Classes/SAS_SS01
- www.ssw.uni-linz.ac.at/Teaching/ Lectures/PSW2/2003/i18n.pdf
- www.gm.fh-koeln.de/~ehses/ap3/folien/folien10.pdf
- www.fh-wedel.de/~si/projekte/ss98/Ausarbeitung/ DistributedProgramming/07dynamclass.html
- www4.informatik.uni-erlangen.de/Lehre/ WS00/V_OODS1/Tutorial/6-A6.pdf
- www.panix.com/~mito/articles/articles/ classloader/j-classloader-ltr.pdf