Mapping IDL – Java

Bezeichner

werden ohne Änderung auf Java-Bezeichner abgebildet. Bei möglichen Konflikten erhalten solche Bezeichner einen vorangestellten Unterstrich.

Dies sind:

- Java-Schlüsselwörter,
  z.B. `new`, `while`, `synchronized`, ...
- Bezeichner, die auf `-Holder`, `-Helper`, `-Operations`, `-POA`, `-POATie` oder `-Package` enden

IDL	Java
`new`	`_new`
`fooHelper`	`_fooHelper`

Module

werden auf ein Java-Package mit gleichem Namen abgebildet.

IDL	Java
`module M1 { ... };`	`package M1;`
`module M2 { module M3 { ... }; };`	`package M2.M3;`

Der Modul `CORBA` wird abgebildet auf `org.omg.CORBA`.

Deklarationen außerhalb eines Moduls erscheinen im globalen (unbenannten) Java-Gültigkeitsraum.

ORBacus:
Generierung eines Packages `defaultpkg`

Über Optionen beim Aufruf von `jidl` lassen sich weitere Einstellungen vornehmen
(`--package <PKG>`, `--auto-package`, ...)

→ siehe `jidl --help`

Basistypen

IDL	Java
`boolean`	`boolean`
`char`	`char` (†)
`wchar`	`char`
`octed`	`byte`
`string`	`java.lang.String` (†)
`wstring`	`java.lang.String`
`short`	`short`
`unsigned short`	`short`
`long`	`int`
`unsigned long`	`int`
`long long`	`long`
`unsigned long long`	`long`
`float`	`float`
`double`	`double`
`fixed`	`java.math.BigDecimal`
`Object`	`org.omg.CORBA.Object`
`any`	`org.omg.CORBA.Any`

Derzeit ist kein Mapping für `long double` definiert.

Achtung:

- Der Wertebereich von Java-char ist größer als der von IDL-char.
  Bei Überschreitung des IDL-Bereichs wird eine Exception `CORBA::DATA_CONVERSION` ausgelöst (†).
- Bounded Strings werden auf `java.lang.String` abgebildet. Das Überschreiten der maximalen Länge bewirkt eine `CORBA::MARSHAL`-Exception.
- Große unsigned-Werte werden in Java durch negative Zahlen repräsentiert.

Interfaces

werden auf Java-Interfaces abgebildet. Die Vererbungsbeziehungen zwischen Interfaces können so direkt in Java ausgedrückt werden.

- *Operations-Interface*:
  endet auf das Suffix `Operations` und enthält alle Operationsdefinitionen des IDL-Interface.

- *Signatur-Interface*:
  heißt wie das IDL-Interface, enthält eventuell Konstanten und wird als Typ für Objektreferenzen verwendet.
  Dieses Interface ist vom dazugehörigen Operations-Interface, eventuell einem Basis-Signatur-Interface (bzw. `org.omg.CORBA.Object`) sowie `org.omg.CORBA.portable.IDLEntity` abgeleitet.

IDL

```
interface A { ... };
```

Java

```
// Operations-Interface
public interface AOperations { /* ops */ }

// Signatur-Interface
public interface A
   extends AOperations,
           org.omg.CORBA.Object,
           org.omg.CORBA.portable.IDLEntity
{ /* const */ }
```


IDL

```
interface B : A { ... };

interface C : B, X { };
```

Java

```
public interface BOperations
   extends AOperations
{ ... }

public interface B
   extends BOperations,
           A,
           org.omg.CORBA.portable.IDLEntity
{ ... }


public interface COperations
   extends BOperations,
           XOperations
{ }

public interface C
   extends COperations,
           B,
           X,
           org.omg.CORBA.portable.IDLEntity
{ }
```

`IDLEntity` ist sogenanntes leeres *Marker-Interface*.

```
package org.omg.CORBA.portable;

interface IDLEntity extends java.io.Serializable { }
```

→ wird benötigt für reverses Java-IDL-Mapping

Es gibt keine spezielle `nil`-Objektreferenz. Statt dessen muss die Java-Konstante `null` verwendet werden.

```
D myD = obj.op();

if (myD == null) {
   // ...
}
```


Die Abbildung für Attribute und Operationen erfolgt analog zum C++-Mapping im Operations-Interface:

IDL-Operationen	→	Java-Operationen
IDL-Attribute	→	Paar von Lese- und Schreib-Operationen gleichen Namens

IDL

```
exception Ex { };

interface I
{
   attribute short a1;
   readonly attribute double a2;

   long func (in string val) raises (Ex);
};
```

Java

```
public interface IOperations
{
   public short a1 ();
   public void a1 (short value);

   public double a2 ();

   public int func (String val) throws Ex;
}
```

Alle nutzerdefinierten Typen und Exceptions innerhalb des Interface `<Type>` erscheinen im Java-Package `<Type>Package`.

IDL

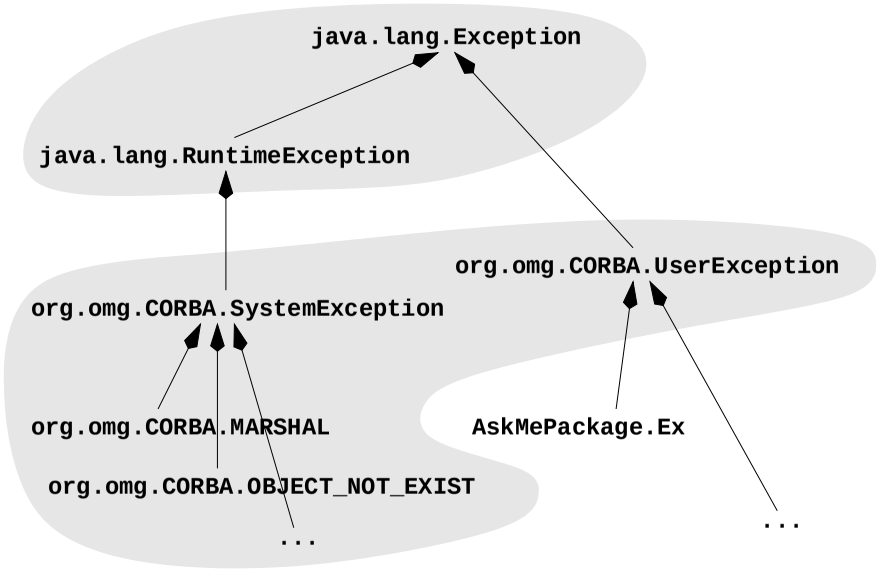
```
interface AskMe
{
   exception Ex { string why; };

   struct Info {
      string name;
      short age;
   };

   Info getInfo (in long id) raises (Ex);
};
```

Java

```
public interface AskMeOperations
{
   public AskMePackage.Info getInfo (int id)
      throws AskMePackage.Ex;
}
```


Exceptions

In CORBA auftretende Exceptions erben von den entsprechenden Java-Exception-Klassen:



`CORBA::SystemExceptions` sind damit auch `java.lang.RuntimeExceptions` und müssen nicht deklariert werden.

Nutzerdefinierte Exceptions werden analog zu `structs` auf `public final`-Klassen abgebildet, die von der Basisklasse `org.omg.CORBA.UserException` abgeleitet sind.

IDL

```
interface AskMe
{
   exception Ex { string why; };

   // ...
};
```

Java

```
package AskMePackage;

final public class Ex
   extends org.omg.CORBA.UserException
{
   public String why;

   public Ex ()
   { }

   public Ex (String _why)
   {
      why = _why;
   }
}
```


Holder-Klassen

Java kennt kein *Call by reference*. Um Werte über `out`- oder `inout`-Parameter an den Aufrufer zurückzugeben, müssen spezielle *Holder*-Klassen verwendet werden. Im Package `org.omg.CORBA` existieren dazu für die Grundtypen u.a. folgende Klassen:

`BooleanHolder`, `CharHolder`, `ByteHolder`, `StringHolder`, `ShortHolder`, `IntHolder`, `LongHolder`, `FloatHolder`, `DoubleHolder`, `ObjectHolder`, `AnyHolder`

Weiterhin wird für jeden nutzerdefinierten IDL-Typ eine entsprechende *Holder*-Klasse generiert.

Alle *Holder*-Klassen besitzen folgenden Grundaufbau:

Java

```
public final class <Type>Holder
   implements org.omg.CORBA.portable.Streamable
{
   public <Type> value;

   public <Type>Holder () { };
   public <Type>Holder (<Type> initial) {
      value = initial;
   }
   ...
}
```

Beispiel:

IDL

```
long func (in short val1,
           inout float val2,
           out StrSeq msg); // sequence <string>
```

generierte Methode in Java

```
public int func (short val1,
                 org.omg.CORBA.FloatHolder val2,
                 StrSeqHolder msg);
```

Benutzung in Java

```
import org.om.CORBA.*;
// ...

FloatHolder f = new FloatHolder();
StrSeqHolder msg = new StrSeqHolder();
int result;

f.value = 3;
// oder oben f = new FloatHolder(3);
result = obj.func (2, f, msg);

System.out.println (f.value);
System.out.println ("Anzahl: " + msg.value.length);
```


Konstanten

Die Repräsentation von IDL-Konstanten in Java ist abhängig von ihrem Gültigkeitsbereich:

Innerhalb eines Interface
werden Konstanten durch Java-Konstanten des entsprechenden Typs, d.h. Elemente der Form

```
public static final <KonstTyp> <KonstName>
```

im Signatur-Interface repräsentiert.

Außerhalb von Interfaces
definierte Konstanten werden auf ein

```
public interface <Konstantenname>
```

mit einem Element

```
public static final <Konstantentyp> value
```

abgebildet.

IDL

```
module M
{
   const long aLong = 123;

   interface I
   {
      const short aShort = 321;
   };
};
```

Java

```
package M;

public interface aLong
{
   public static final int value = (int) 123L;
}

public interface I extends ...
{
   public static final short aShort = (short) 321;
}
```

Benutzung in Java

```
int longValue = M.aLong.value;
short shortValue = M.I.aShort;
```


Aufzählungstypen

werden durch eine Klasse repräsentiert, die pro Konstante zwei statische Member-Daten besitzt (eines als `int`, eines als Typ der gemappten Klasse), Methoden `value` und `from_int` zur Konvertierung sowie einen protected Konstruktor.

IDL

```
enum Farbe { rot, gelb, gruen };
```

Java

```
public final class Farbe ... {
   public static final int _rot = 0;
   public static final Farbe
      rot = new Farbe (_rot);
   public static final int _gelb = 1;
   public static final Farbe
      gelb = new Farbe (_gelb);
   public static final int _gruen = 2;
   public static final Farbe
      gruen = new Farbe (_gruen);

   public int value () { ... }
   public static Farbe from_int (int value) { ... }

   protected Farbe (int init) { ... }
}
```

Die Nummerierung der `int`-Member beginnt bei 0.

Jede Aufzählungskonstante ist damit einmalig und eindeutig. In `switch`-Anweisungen muss allerdings die `int`-Repräsentation benutzt werden.

Beispiel:

Benutzung in Java

```
// Farbe f = new Farbe (0);    // illegal !
Farbe f = ...;

if (f == Farbe.rot)
   ...

// aber:
switch (f.value ()) {
   case Farbe._rot   : // Halt
   case Farbe._gelb  : // Achtung
   case Farbe._gruen : // Los
}
```


Strukturen

werden durch eine Klasse repräsentiert, die für jedes Element der Struktur eine `public`-Membervariable mit gleichem Namen enthält. Zusätzlich zum Default-Konstruktor existiert ein Konstruktor, mit dem alle Struktur-Elemente initialisiert werden können.

IDL

```
struct Offer {
   string description;
   long priceEuro;
   short priceCent;
};
```

Java

```
public final class Offer ... {
   public String description;
   public int priceEuro;
   public short priceCent;

   public Offer () { }
   public Offer (String _description,
                 int _priceEuro,
                 short _priceCent)
   {
      description = _description;
      priceEuro = _priceEuro;
      priceCent = _priceCent;
   }
}
```

Unions

werden durch eine Klasse repräsentiert, die folgende Methoden bereitstellt:

- einen Default-Konstruktor
- `discriminator` zum Zugriff auf den Diskriminator (bei Namenskonflikt `_discriminator`)
- Lese- und Schreibmethoden für jedes Element
- Schreibmethoden für Elemente mit mehr als einer Marke, so dass der Diskriminator explizit gesetzt werden kann
- gegebenenfalls zwei Methoden `_default`, mit der der Diskriminator auf einen von allen Marken verschiedenen Wert automatisch bzw. explizit gesetzt werden kann

Lesende Zugriffe auf nicht gesetzte Elemente oder falsches explizites Setzen des Diskriminators erzeugen eine System-Exception `CORBA::BAD_OPERATION`.

Beispiel:

IDL

```
union Fahrt switch (Farbe) {
   case rot :
   case gelb :  long wartezeit;
   case gruen : float verbrauch;
   // default : ... // illegal !
};
```

Java

```
public final class Fahrt ... {
   public Fahrt () { ... }
   public Farbe discriminator () { ... }

   public int wartezeit () { ... }
   public void wartezeit (int value) { ... }
   public void wartezeit (Farbe discr, int value)
   { ... }

   public float verbrauch () { ... }
   public void verbrauch (float value) { ... }
}
```

Angenommen, „`case rot :`“ käme im Union nicht vor, dann gäbe es keine Methode
`public void wartezeit (Farbe discr, int value)`,
statt dessen aber
`public void _default ()` und
`public void _default (Farbe discr)`.

Sequenzen und Arrays

werden auf Java-Arrays des entsprechenden Grundtyps abgebildet. Es gibt keinen Java-Typ mit dem Namen der Sequenz oder des Arrays!

Die erzeugte *Holder*-Klasse besitzt dementsprechend folgendes Aussehen:

IDL

```
typedef sequence < Fahrt > FahrtSeq;
```

Java

```
public final class FahrtSeqHolder ... {
   public Fahrt[] value;

   public FahrtSeqHolder () { }
   public FahrtSeqHolder (Fahrt[] initial) {
      value = initial;
   }
   ...
}
```

Bei beschränkten Sequenzen und Arrays wird bei der Parameterübergabe die Einhaltung der festgelegten Grenzen überprüft. Bei Überschreitung wird die Exception `CORBA::MARSHAL` geworfen.

Damit auf Arraygrenzen zugegriffen werden kann, sollten sie durch Konstanten definiert werden:

IDL

```
interface Rechner
{
   const short xMax = 100;
   const short yMax = 200;

   typedef long Feld[xMax][yMax];

   void rechne (in Feld f1, out Feld f2);
};
```

Java

```
public interface RechnerOperations
{
   public void rechne
      (int[][] f1, RechnerPackage.FeldHolder f2);
}
```

Benutzung in Java

```
Rechner rechner = ...
int[][] feld = new int[Rechner.xMax][Rechner.yMax];
RechnerPackage.FeldHolder fh =
   new RechnerPackage.FeldHolder ();

// feld belegen ... und Operation aufrufen
rechner.rechne (feld, fh);
```

Typedefs

Java besitzt kein `typedef`-Konstrukt.

Alle so definierten Typen werden daher auf ihre Ausgangstypen abgebildet.

IDL

```
typedef short ID;
typedef Fahrt Trip;     // Fahrt war ein union

interface Holiday {
   Trip makeATrip (out ID number);
};
```

Java

```
public interface HolidayOperations
{
   public Fahrt makeATrip (
      omg.org.CORBA.ShortHolder number);
}
```

Holder-Klassen werden nur für `typedefs` von Sequenzen und Arrays erzeugt.

Helper-Klassen

Für jeden nutzerdefinierten IDL-Typ (inclusive `typedefs`) wird eine *Helper*-Klasse generiert, die diverse statische Hilfsfunktionen bereitstellt. Diese besitzt folgenden Grundaufbau:

Java

```
public class <Type>Helper {

   // Einfuegen in eine any-Variable
   public static void
      insert (org.omg.CORBA.Any a, <Type> t)
      { ... }

   // Extrahieren aus einer any-Variablen
   public static <Type>
      extract (org.omg.CORBA.Any a)
      { ... }

   // Typecode
   public static org.omg.CORBA.TypeCode type ()
   { ... }

   // Repository-ID
   public static String id () { ... }

   ...

   // falls <Type> ein Interface ist
   public static <Type>
      narrow (org.omg.CORBA.Object obj) { ... }
}
```

Für Interfaces enthält die jeweilige *Helper*-Klasse die Methode `narrow`. Unterstützt die übergebene Objektreferenz das Interface nicht, wird die System-Exception `CORBA::BAD_PARAM` geworfen (anders als in C++).

Beispiel in Java

```
// Benutzung des Namensdienstes:

import org.omg.CosNaming.*;
import org.omg.CosNaming.NamingContextPackage.*;
// ...

NamingContext nc = ...
NameComponent[] name = ...
org.omg.CORBA.Object obj;
MyInterface m;

try {
   obj = nc.resolve (name);
}
catch (NotFound nfe) {
   // so ein Pech!
}

try {
   m = MyInterfaceHelper.narrow (obj);
}
catch (org.omg.CORBA.BAD_PARAM bp) {
   // falsches Interface
}
```


Ein ausführlicheres Beispiel: die Implementation der Methode `list` aus dem Namensdienst

Dazu sei z.B. folgende Klasse definiert:

```
public class MyBinding
{
   public String id;
   public String kind;
   public org.omg.CORBA.Object obj;
   boolean is_context;
}
```

Der Namensdienst-Server verwaltet dann pro Namenskontext ein Feld

```
   MyBinding[] mb;
```

das alle Bindings enthält.

```
import org.omg.CosNaming.*;

public void list (int how_many,
                  BindingListHolder bl,
                  BindingIteratorHolder bi)
{
   int num = mb.length < how_many
             ? mb.length : how_many;
   Binding[] b = new Binding[num];

   for (int i=0; i<num; i++) {
      b[i] = new Binding ();
      b[i].binding_name = new NameComponent[1];
      b[i].binding_name[0] = new NameComponent ();
      b[i].binding_name[0].id   = mb[i].id;
      b[i].binding_name[0].kind = mb[i].kind;
      b[i].binding_type = mb[i].is_context
         ? BindingType.ncontext
         : BindingType.nobject;
   }
   if (how_many < mb.length) {
      Binding[] rest =
         new Binding[mb.length-how_many];
      for (int i=0; i<mb.length-how_many; i++) {
         // ... rest[i] fuellen wie oben
         // aus den Elementen mb[i+how_many];
      }
      bi.value = new BindingIteratorImpl (rest);
   }
   else
      bi.value = null;
   bl.value = b;
}
```


Object

→ muss immer als `org.omg.CORBA.Object` referenziert werden, da Java sonst `java.lang.Object` annimmt!

Auszug aus dem Interface in Java:

```
package org.omg.CORBA;

public interface Object
{
   boolean _is_a (String repositoryId);
   boolean _is_equivalent (org.omg.CORBA.Object o);
   boolean _non_existent();
   org.omg.CORBA.Object _duplicate();
   void _release();

   // ...
}
```

→ diese letzten beiden Methoden existieren aus historischen Gründen, sie werden in Java eigentlich nicht benötigt.

Ihre aktuelle Implementation durch ORBacus ist z.B.

```
   public org.omg.CORBA.Object _duplicate()
   { return this; }

   void _release()
   { }
```

ORB

Wie alle Pseudo-Objekte wird auch der ORB durch eine abstrakte Klasse beschrieben. In diesem Fall durch

```
package org.omg.CORBA;
abstract public class ORB;
```

Er kann nicht per `new ORB(...)` erzeugt werden.

Zur Initialisierung gibt es drei Möglichkeiten:

Für (standalone) Java-Applikationen:

```
public static ORB init (
   String[] args,
   java.util.Properties props);
```

Für Java-Applets:

```
public static ORB init (
   java.applet.Applet app,
   java.util.Properties props);
```

Singleton-Initialisierung:

```
public static ORB init ();
```

Diese letzte Variante liefert den sogenannten *singleton* ORB, mit dessen Hilfe sich z.B. Anys und TypeCodes erzeugen lassen. Der Aufruf regulärer Requests über einen solchen ORB (etwa `string_to_object()`) führt zu einer System-Exception `CORBA::NO_IMPLEMENT`.

Zur Initialisierung des korrekten ORB gilt folgende Suchreihenfolge:

1. Applikations- bzw. Applet-Parameter
2. Property-Parameter
3. System-Properties
4. Default-Verhalten

Für ORBacus und Java 2 muss angegeben werden:

```
java -Dorg.omg.CORBA.ORBClass=com.ooc.CORBA.ORB \
     -Dorg.omg.CORBA.ORBSingletonClass=com.ooc.ORBSingleton \
     MyApp
```

bzw. für ein Applet:

```
<APPLET CODE="MyApp.class" ARCHIVE="OB.jar">
   <PARAM  NAME="org.omg.CORBA.ORBClass"
           VALUE="com.ooc.CORBA.ORB">
   <PARAM  NAME="org.omg.CORBA.ORBSingletonClass"
           VALUE="com.ooc.CORBA.ORBSingleton">
</APPLET>
```

Properties werden folgendermaßen verwendet:

```
java.util.Properties props = new java.util.Properties();
props.put ("org.omg.CORBA.ORBClass",
           "com.ooc.CORBA.ORB");
props.put ("org.omg.CORBA.ORBSingletonClass",
           "com.ooc.CORBA.ORBSingleton");
ORB orb = ORB.init (/* args oder applet */, props);
```

Bemerkungen zu Java/CORBA:

- Java 1.1.* enthält keinen eingebauten ORB
  - Hier müssen die entsprechenden Klassen (für ORBacus: `OB.jar`) separat zur Verfügung gestellt werden.
  - Die Angabe über Properties, welcher ORB verwendet werden soll, kann entfallen.
- Ab Java 1.2 ist ein ORB vorhanden
  - Ohne Festlegung über Properties wird Suns Builtin-ORB verwendet (eingeschränkte Funktionalität).
  - Bug in Java 1.2: `ORB.init()` liefert *immer* den Singleton-ORB zurück, auch wenn zuvor eine volle Initialisierung erfolgt ist.
  - Bug im Java 1.2 Plugin (Applets): `ORB.init()` liefert nicht den in den Properties definierten ORB, sondern den eingebauten.
    (Das führt zu Fehlern im ORBacus-ORB.)
- Applets können nur als CORBA-Klienten arbeiten und nur auf Objekte zugreifen, die auf dem WWW-Server existieren.

Auszug aus der ORB-Klasse

```
package org.omg.CORBA;

public abstract class ORB
{
   public abstract String[]
      list_initial_services();

   public abstract org.omg.CORBA.Object
      resolve_initial_references (String id)
      throws org.omg.CORBA.ORBPackage.InvalidName;

   public abstract String
      object_to_string (org.omg.CORBA.Object obj);

   public abstract org.omg.CORBA.Object
      string_to_object (String str);

   public abstract Any create_any();

   // ... und weitere wie z.B. init()
}
```

Objektimplementation in Java

1. Implementation durch Vererbung:

Für jedes Interface *I* wird vom IDL-Compiler eine abstrakte Basisklasse `<I>POA` erzeugt, für die vom Programmierer eine Implementation bereitzustellen ist:

generierter Java-Code

```
abstract public class <I>POA
   extends org.omg.PortableServer.DynamicImplementation
   implements <I>Operations
{
   ...
}
```

Implementierung

```
public class <I>Impl extends <I>POA
{
   ...
}
```

Problem dabei:
Bereits programmierte Klassen für Basisinterfaces können nicht wiederverwendet werden.

```
// IDL
interface A
{ void a_op(); };

interface B : A
{ void b_op(); };
```


Die Klasse für Interface B muss vollständig neu programmiert werden:

```
// Java
public class BImpl extends BPOA
{
   public void a_op()
   { ... }

   public void b_op()
   { ... }
}
```

Vergleich dazu in C++:

```
// C++
class AImpl : virtual public POA_A
{
   void a_op()
   { ... }
};

class BImpl : virtual public POA_B,
              virtual public AImpl
{
   void b_op()
   { ... }
};
```

2. Implementation durch Delegierung

Delegierung bedeutet, dass ein Objekt dem eigentlichen CORBA-Servant als *Delegate* zugeordnet wird.

Der Programmier erstellt dazu eine Klasse, die das Java-Interface `<I>Operations` implementiert.

```
public class ADelegate implements AOperations
{
   public void a_op()
   { ... }
}
```

Jetzt kann auf `ADelegate` bei der Implementierung von `BDelegate` zurückgegriffen werden:

```
public class BDelegate extends ADelegate
                       implements BOperations
{
   public void b_op()
   { ... }
}
```

Der IDL-Compiler erzeugt als Servant weiterhin eine Klasse `<I>POATie`, der der *Delegate* zugeordnet werden muss.

Bei ORBacus muss der IDL-Compiler `jidl` mit der Option `--tie` aufgerufen werden, damit `<I>POATie` generiert wird.

Die Klasse `<I>POATie` besitzt folgenden Grundaufbau:

```
public class <I>POATie extends <I>POA
{
   private <I>Operations _delegate;

   // Konstruktor
   public <I>POATie (<I>Operations delegate)
   {
      _delegate = delegate;
   }

   // Delegate abfragen und setzen
   public <I>Operations _delegate()
   {
      return _delegate;
   }

   public void _delegate (<I>Operations delegate)
   {
      _delegate = delegate;
   }

   // und fuer jede Methode <op> aus <I>Operations:
   <opType> <op> (<opParameters>)
   {
      [return] _delegate.<op> (<opParameters>);
   }

   // ... weitere
}
```

Erzeugung eines Servants

1. bei Vererbung

Hier muss einfach per `new` eine Instanz der Implementationsklasse erzeugt werden:

```
BImpl b = new BImpl();
```

2. bei Delegierung

Hier wird eine Instanz von `<I>POATie` erzeugt und im Konstruktor das Delegate-Objekt übergeben:

```
BDelegate delegate = new BDelegate();
BPOATie b = new BPOATie (delegate);
```

Implizite Aktivierung des Servants

ähnlich wie in C++:

```
B bref = b._this (orb);
```

Ein Servant kann nur einem ORB zugeordnet werden. Dieser muss beim ersten Aufruf von `_this()` übergeben werden. Für folgende Aufrufe zur Ermittlung der CORBA-Objektreferenz kann `_this()` ohne Parameter verwendet werden.

vollständiges Beispiel: mal wieder der Counter

Server-Klasse und `main()`:

```
import org.omg.CORBA.*;
import java.util.Properties;

public class Server {
   public static ORB orb;

   public static void main (String[] args)
      throws org.omg.CORBA.UserException
   {
      Properties props = System.getProperties();
      props.put ("org.omg.CORBA.ORBClass",
                 "com.ooc.CORBA.ORB");
      props.put ("org.omg.CORBA.ORBSingletonClass",
                 "com.ooc.CORBA.ORBSingleton");

      orb = ORB.init(args, props);
      org.omg.PortableServer.POA rootPOA =
         org.omg.PortableServer.POAHelper.narrow(
         orb.resolve_initial_references("RootPOA"));
      org.omg.PortableServer.POAManager manager =
         rootPOA.the_POAManager();

      CounterFactoryImpl cfimp =
         new CounterFactoryImpl ();
      Count.CounterFactory cfref = cfimp._this (orb);

      // ... Referenz bekannt machen
      manager.activate();
      orb.run();
   }
}
```

Zur Erinnerung – die IDL-Definitionen:

```
module Count
{
   interface Counter
   {
      readonly attribute long sum;

      void reset (in long value);
      long increment();
      void destroy();
   };


   interface CounterFactory
   {
      Counter createCounter ();
   };
};
```

Die Implementation der Factory:

```
public class CounterFactoryImpl
   extends Count.CounterFactoryPOA
{
   public Count.Counter createCounter ()
   {
      CounterImpl c = new CounterImpl();
      return c._this (Server.orb);
   }
}
```


Die Implementation des Counters:

```
import org.omg.PortableServer.POA;

public class CounterImpl
   extends Count.CounterPOA
{
   private int s;

   public int sum ()
   {
      return s;
   }

   public void reset (int value)
   {
      s = value;
   }

   public int increment ()
   {
      return ++s;
   }

   public void destroy ()
   {
      POA poa = _default_POA();
      try {
         byte[] id = poa.servant_to_id (this);
         poa.deactivate_object (id);
      }
      catch (org.omg.CORBA.UserException ex) {}
   }
}
```