

# F Überblick über die 5. Übung

---

- Einführung in CORBA
- Verwendung von CORBA-Objekten
  - ◆ Interface Definition Language (IDL)

## Organisation

---

- 22. November 2007
  - ◆ CORBA Einführung, IDL, Mapping von IDL nach Java, Java-Client
- 29. November 2007
  - ◆ serverseitiges Mapping, POA, Namensdienst

## F.1 Einführung in CORBA

- Inhalt: Programmierung von CORBA-Anwendungen
- Implementierungssprache: Java wird verwendet (aber: CORBA ist Programmiersprachenunabhängig!)
- Allgemeine Konzepte von CORBA: siehe Vorlesung
- Vor allem: Betrachtung von praktische Problemen
- Als Ergänzung: Behandlung von speziellen CORBA-Features

## 1 CORBA

- Common Object Request Broker Architecture and Specification (CORBA) – Zentrale Spezifikation ("The Core Spec")
- Weitere separate Spezifikationen, die auf CORBA aufbauen
  - ◆ CORBAservices
  - ◆ Domain Interfaces
  - ◆ CORBA Component Model
  - ◆ (Unified Modelling Language & Meta Object Facility)
- Viele "Task Forces" und "Special Interest Groups" innerhalb der OMG
  - ◆ Ergänzung von neuer Konzepte und Erweiterungen
  - ◆ Revision von existierenden Standards
- ➔ Alle OMG-Spezifikationen werden ständig weiterentwickelt!

## 2 CORBA-Versionen

- CORBA 1.x (October 1991)
  - ◆ CORBA-Objektmodell und Architektur
  - ◆ Schnittstellenbeschreibungssprache (Interface Definition Language, IDL)
  - ◆ Sprachabbildungen für C, C++ und Smalltalk
- CORBA 2.0 (Juli 1996)
  - ◆ Interoperabilität durch IIOP als Protokoll, das alle ORB-Implementierungen unterstützen müssen
- CORBA 2.1 (August 1997)
  - ◆ IDL-Erweiterungen
  - ◆ Neue Sprachabbildungen (Cobol, Ada)
- CORBA 2.2 (Februar 1998)
  - ◆ *Portable Object Adaptor* (POA) ersetzt *Basic Object Adaptor* (BOA)
  - ◆ Neue Sprachabbildung (Java)

## 2 CORBA Versionen (2)

- CORBA 2.3/2.3.1 (Juni/Oktober 1999)
  - ◆ Überarbeitete Sprachabbildungen zur Anpassung an den POA
  - ◆ Valuetypes, object-by-value-Parameter
  - ◆ Separate Dokumente für die Sprachabbildungen
- CORBA 2.4/2.4.1 (Oktober/November 2000)
  - ◆ CORBA Messaging, Minimum CORBA, Real-time CORBA
- CORBA 3.0/3.0.3 (Juni/November 2002)
  - ◆ Fault Tolerant CORBA
  - ◆ CORBA-Komponentenmodell: Unterstützung von Enterprise JavaBeans; "CORBAcomponent software marketplace": Verteilung von Komponenten

### 3 Informationen zu CORBA

- Wer wirklich alle Details zu CORBA wissen will, muss letztendlich die Spezifikationen lesen!
- Spezifikationen sind öffentlich verfügbar
  - ◆ OMG Webseite: <http://www.omg.org/>
  - ◆ Webseiten der OOVs-Übung
- Viele Bücher von sehr unterschiedlicher Qualität
- Vorsicht vor CORBA-Produktdokumentation
  - ◆ Oft werden proprietäre Erweiterungen beschrieben.

### 4 CORBA-Produkte vs. CORBA-Standard

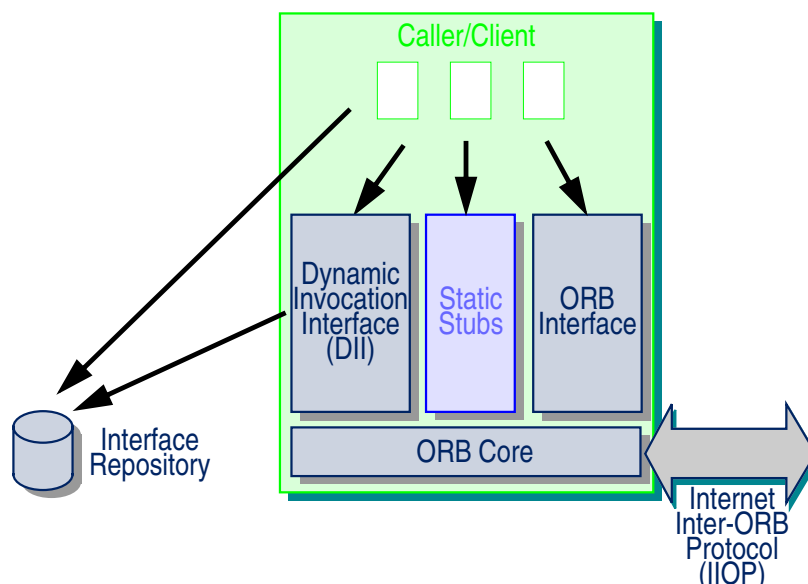
- Keine etabliertes "CORBA"-Zertifikat
  - ◆ Jeder kann behaupten, zu CORBA Version x.y kompatibel zu sein.
  - ◆ Open Group beginnt mit Produkt-Zertifizierung (seit CORBA 2.1).
- CORBA-Produkte führ(t)en proprietäre Erweiterungen ein
  - ◆ Gut, solange man sich nicht darauf verlässt
  - ◆ In der CORBA-Frühzeit ging es oft nicht anders, z.B. BOA.
  - ◆ Heute kann man (fast) alles auf einem standardisierten Weg erreichen.
- Einige Features in Produkten entsprechen nicht 100%ig den Spezifikationen
  - ◆ Z.B. Sprachabbildungen
  - ◆ Spezifikationen ändern sich, Produkte erst etwas später.

## F.2 Verwendung von CORBA-Objekten

### 1 CORBA-Objekte aus Sicht des Klienten

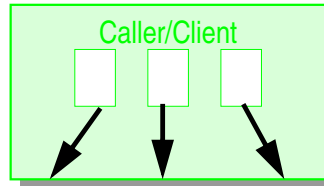
- CORBA-Objekte haben (genau) eine Schnittstelle (Interface)
  - ◆ Beschreibung der Schnittstelle in der CORBA Interface Definition Language
  - ◆ IDL-Schnittstellen sind ein "Vertrag" zwischen dem CORBA-Objekt und dessen Aufrufern.
- Aufrufer eines CORBA-Objekts haben nur eine (opaque) Objektreferenz
  - ◆ Objekt-Methoden werden über die Referenz aufgerufen.
  - ◆ Das Objekt kann lokal oder entfernt sein.
  - ◆ Objektschnittstelle kann abgefragt werden (vgl. Java Reflection).
  - ◆ Aufrufe können zur Laufzeit erzeugt werden (Dynamic Invocation).
- Object Request Broker (ORB) übermittelt Aufrufe und Antworten
  - ◆ Nur der ORB kann Objektreferenzen interpretieren.
- Aufrufer/Klient ist nur eine *Rolle* bei einem Aufruf (z.B. Callbacks).

### 2 Architektur des Aufrufers/Klienten



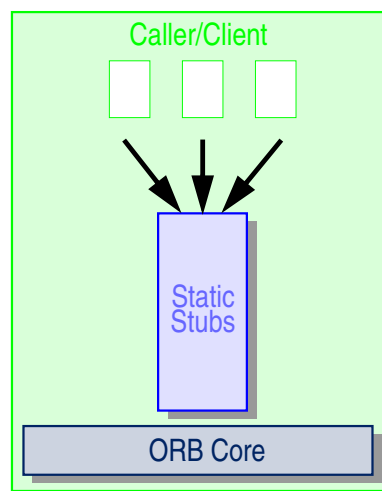
### 3 Der Klient

- Ruft Operationen an CORBA-Objekten auf.
- Muss selbst kein CORBA-Objekt sein.



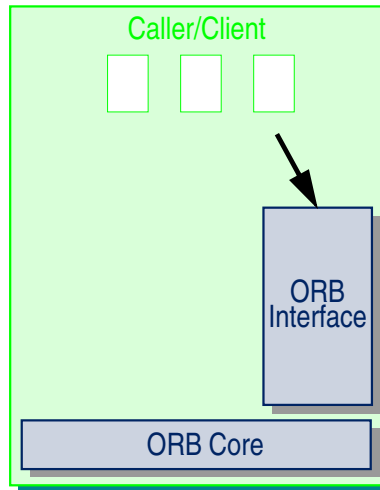
### 4 Statische Stubs

- Können automatisch aus der IDL-Schnittstelle erzeugt werden.
- Marshalling der Aufrufparameter
- Demarshalling der Rückgabewerte/Exceptions des Aufrufes



## 5 ORB-Schnittstelle

- Export von ersten Objektreferenzen (ORB, POA, Services, ...)
- Verarbeitung von Objektreferenzen (Umwandlung in Strings und umgekehrt)

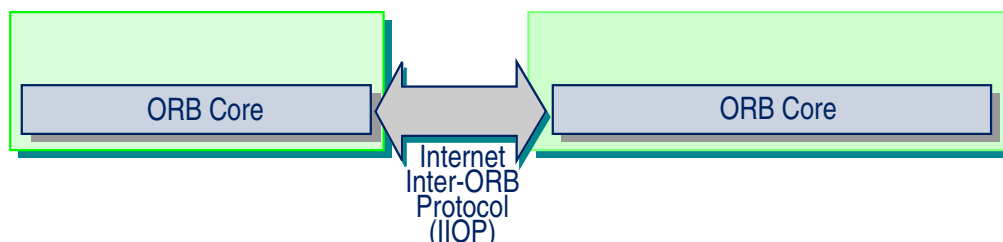


## 6 Der Kern des ORB (ORB Core)

- Übertragung von Aufrufen mit Hilfe von Informationen in den Objektreferenzen

## 7 General Inter-ORB Protocol (GIOP)

- Standard-Transportprotokoll zwischen ORBs
- Grundlage für die Interoperabilität
- GIOP über TCP-Verbindungen: Internet Inter-ORB Protocol (IIOP)
- Jeder CORBA 2.x ORB muss IIOP implementieren



## 8 Zusammenfassung Aufrufer/Klient

- Müssen selbst keine CORBA-Objekte sein.
- Können Operationen an CORBA-Objekten aufrufen.
- Opake Objektreferenzen
- ORB überträgt die Aufruf-Daten.

## F.3 Interface Definition Language (IDL)

- Grundlegendes
- Bezeichner (Identifiers)
- Primitive Datentypen
- Zusammengesetzte Datentypen
- Schnittstellen von CORBA-Objekten
- Designfragen



# 1 Grundlegendes

- IDL dient der Beschreibung von Datentypen und Schnittstellen.
- Unabhängig von der/den Implementierungs-Programmiersprache(n)
- Syntax ist stark an C++ angelehnt
  - ◆ Beschreibung von Daten und Schnittstellen (Typen, Attribute, Methoden, ...)
  - ◆ Keine steuernden Anweisungen (if, while, for, ...)
- Präprozessor wie in C++
  - ◆ **#include** um andere IDL-Dateien einzubinden
  - ◆ **#define** für Makros
- Kommentare wie in C++ und Java:

```
// Das ist ein einzeliger Kommentar
/*
 * Das ist ein mehrzeiliger
 * Kommentar
 */
```

## 2 Bezeichner (Identifiers)

- Bestimmte reservierte Wörter
  - ◆ **module**, **interface**, **struct**, **void**, **long**, ...
- Alle anderen Kombinationen von kleinen und großen Buchstaben, Zahlen und Unterstrichen sind erlaubt.
  - ◆ Erstes Zeichen muss Buchstabe sein!
- "\_" als Escape-Zeichen für reservierte Wörter
  - ◆ z.B. **"\_module"**, um einen Bezeichner **"module"** zu erzeugen
  - ◆ Vereinfacht Erweiterung der Menge der reservierten Wörter.

## 2 Bezeichner (Identifiers)

- Sobald ein Bezeichner benutzt ist, sind alle anderen Varianten mit anderer Gross-/Kleinschreibung verboten!

- Beispiel:

```
module Beispiel1 { ... };
module BEISPIEL1 { ... };    // illegal in IDL
```

- Sinn:

- ◆ Erlaubte Abbildung von IDL zu Sprachen, die nicht "case-sensitive" sind
- ◆ Erhalte Schreibweise von Bezeichner für "case-sensitive" Sprachen

## 3 Module

- Namensraum (scope) für IDL-Deklarationen

- Syntax:

```
module Name {
    Deklarationen
};
```

- Zugriff auf andere Namensräume über den "::"-Operator

- Beispiel:

```
module Beispiel1 {
    typedef long IDNumber;
};
module Beispiel2 {
    typedef Beispiel1::IDNumber MyID;    // typedef long MyID;
};
```

## 4 Datentyp-Deklarationen

- Alias für einen existierenden Datentyp

- Syntax:

```
typedef existing_type alias;
```

- Beispiel:

```
typedef long IDNumber;
```

## 5 Primitive Datentypen

- Integer-Zahlen

◆ <b>short</b>	$-2^{15}$ to $2^{15}-1$
◆ <b>unsigned short</b>	0 to $2^{16}-1$
◆ <b>long</b>	$-2^{31}$ to $2^{31}-1$
◆ <b>unsigned long</b>	0 to $2^{32}-1$
◆ <b>long long</b>	$-2^{63}$ to $2^{63}-1$
◆ <b>unsigned long long</b>	0 to $2^{64}-1$

- Fließkommazahlen (IEEE-Standard für binäre Fließkommazahlen-Arithmetik, ANSI/IEEE Std 754-1985)

◆ <b>float</b>	einfache Genauigkeit
◆ <b>double</b>	doppelte Genauigkeit
◆ <b>long double</b>	erweiterte Genauigkeit (mindestens 15 Bit Exponent und 64 Bit Basis)

## 5 Primitive Datentypen (2)

- Zeichen
  - ◆ **char** ISO 8859-1 (Latin1) Zeichen
  - ◆ **wchar** multi-byte character (z.B. Unicode)
  - ◆ Länge hängt von Implementierung und Programmiersprache ab.
- **boolean**
  - ◆ Nur die Werte TRUE und FALSE
- **octet**
  - ◆ Länge 8 bit
  - ◆ Keine Konvertierung bei der Übertragung
- **any**
  - ◆ Kapselung für beliebigen CORBA-Datentyp
- **void**

## 6 Strukturen

- Gruppierung von mehreren Typen in einer Struktur
- Syntax:

```
struct Name {
    Deklaration von Struktur-Elementen
};
```

- Beispiel:

```
struct AmountType {
    float value;
    char currency;
};
```

- Verwendung:

```
AmountType amount;
```

## 6 Geschachtelte Strukturen

- Strukturen können innerhalb anderer Strukturen definiert werden.

- Beispiel:

```
struct AmountType {
    struct ValueType {
        long integerPart;
        short fractionPart;
    } amount;
    char currency;
};
```

- Strukturen erzeugen einen eigenen Namensraum (scope)!

- Kompletter Name von obigem Typ:

```
AmountType::ValueType
```

## 7 Aufzählungen

- Aufzählungen mit festgelegter Menge an möglichen Werten

- Syntax:

```
enum name {
    value1, value2, ...
};
```

- Beispiel:

```
enum Color {
    GREEN, RED, BLUE
};
```

## 7 Aufzählungen

- Achtung: Aufzählungen erzeugen keinen eigenen Namensraum!
- Zugriff auf Werte von Aufzählungen:

```
GREEN
not Color::GREEN
```

- Nicht erlaubt::

```
interface A {
    enum E { E1, E2, E3 };           // line 1
    enum BadE { E3, E4, E5 };       // Fehler: E3 is bereits
}                                   // definiert
```

## 8 Arrays

- Ein- und Mehrdimensionale Arrays
  - ◆ Feste Grösse in jeder Dimension

- Syntax:

```
typedef element_type name[positive_constant][positive_constant] ...;
```

- Beispiel:

```
typedef long Matrix[3][3];
```

- Achtung:  
Array-Datentypen müssen mit **typedef** deklariert werden, bevor man sie verwenden kann!

## 9 Sequences

- Eindimensionales Array
  - ◆ Variable Grösse
  - ◆ Optional maximale Grösse ("bounded sequence")

- Syntax:

```
typedef sequence<element_type> name;           // unbounded
typedef sequence<element_type, positive_constant> Name; // bounded
```

- Beispiel:

```
typedef sequence<long> Longs;
typedef sequence< sequence<char> > Strings;
```

- Achtung:  
Auch Sequence-Datentypen müssen vor Verwendung mit **typedef** deklariert werden!

## 10 Zeichenketten

- Zeichenketten
  - ◆ Ähnlich zu **sequence<char>** und **sequence <wchar>**
  - ◆ Spezieller Datentyp aus Performance-Gründen
  - ◆ Zeichenketten müssen nicht mit **typedef** deklariert werden.
  - ◆ Ebenfalls optional maximale Grösse festlegbar

- Syntax:

```
typedef string name;           // unbounded
typedef string<positive_constant> name; // bounded
typedef wstring name;         // unbounded
```

- Beispiel:

```
typedef string<80> Name;
```

## 11 Konstanten

- Symbolische Namen für spezielle Werte

- Syntax:

```
const type Name = Konstantenausdruck;
```

- Konstantenausdruck

- ◆ Konstante Werte (Zahlen/Zeichen/Zeichenketten/Enums je nach **type**)

- Beispiel:

```
const Color WARNING = RED;
```

## 12 Schnittstellen (Interfaces)

- Sichtbare Schnittstelle von CORBA-Objekten

- Enthält:

- ◆ Attribute
- ◆ Operationen
- ◆ Lokale Typen, Konstanten, Exceptions

- Syntax:

```
interface name {  
    Deklaration von Attributen und Operationen (sowie Typen und Exceptions)  
}
```

- Schnittstellen definieren ebenfalls einen eigenen Namensraum.

- Die Namen von Attributen und Operationen müssen eindeutig sein!

- ◆ Kein "Overloading!"



## 12 Schnittstellen - Attribute

- Öffentliche Objektvariablen
  - ◆ Schreibzugriff kann verhindert werden (Nur-Lese-Attribute)
  - ◆ Keine Instanzvariablen

- Syntax:

```
attribute type name;           // read & write
readonly attribute type name;  // read-only
```

- Beispiel:

```
interface Account {
    readonly attribute float balance;
};
```

## 12 Schnittstellen – Operationen

- Methoden von CORBA-Objekten mit:

- ◆ Methoden-Name
- ◆ Rückgabe-Datentyp
- ◆ Aufruf-Parameter
- ◆ Exceptions
- ◆ (Aufruf-Kontext)

- Syntax:

```
return_type name( parameter_list ) raises( exception_list );
```

- Nur der Methodenname ist signifikant

- ◆ Kein Overloading durch Parametertypen

- Methodenaufruf mit "best-effort"-Semantik (keine Rückgabe-Werte und keine Exceptions erlaubt)

```
oneway void name( parameter_list );
```

## 12 Schnittstellen – Parameterübertragung

- Für jeden Parameter muss die Übertragungsrichtung angegeben werden:

- ◆ **in** nur vom Klienten zum Server
- ◆ **out** nur vom Server zum Klienten
- ◆ **inout** in beiden Richtungen

- Syntax:

```
( copy_direction1 type1 name1, copy_direction2 type2 name2, ... )
```

- Beispiel:

```
interface Account {
    void makeDeposit( in float sum );
    void makeWithdrawal( in float sum,
                        out float newBalance );
};
```

## 12 Schnittstellen – Vererbung

- Ableitung von neuen Schnittstellen von existierenden

- Mehrfache Vererbung möglich

- Syntax:

```
interface name : inherited_interface1, inherited_interface2, ... {
    Declaration of additional attributes and operations
};
```

- Namen von geerbten Attributen und Operationen müssen eindeutig sein.

- ◆ Ausnahme: Bezeichner, die auf verschiedenen Pfaden geerbt werden, aber von der selben Basisklasse stammen, sind erlaubt.

## 12 Schnittstellen – Vererbung (2)

- Weder "Overloading" noch "Overriding" ist erlaubt:

```
module Foo {
    interface A {
        void draw( in float num );
    };

    interface B {
        void print( in float num);
        void print( in string str); // Fehler: Overloading
    };

    interface C: A, B {
        void draw( in float num);    // Fehler: Overriding
    };
};
```

## 12 Schnittstellen – Vererbung (3)

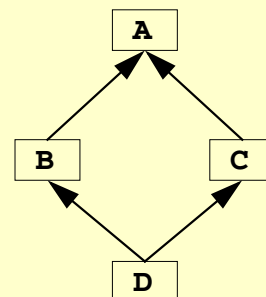
- Erlaubter Vererbungsgraph in CORBA:

```
module Foo {
    interface A {
        void draw( in float num );
    };

    interface B : A {
    };

    interface C : A {
    };

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



## 13 Exceptions – User Exceptions

- Benutzer-Exceptions werden im Benutzercode auf Serverseite erzeugt und zum Klienten weitergereicht.

- Syntax:

```
exception name {
    Declaration of data elements
};
```

- Exceptions sind eine spezielle Form von Strukturen

- ◆ Nur Datenelemente, keine Operationen

- ◆ Keine Vererbung von Exceptions!

- Beispiel:

```
interface Account {
    exception Overdraft { float howMuch; };
    void makeWithdrawal( in float sum )
        raises( Overdraft );
};
```

## 13 Exceptions – System Exceptions

- "System Exceptions" werden vom ORB bei internen Fehlern erzeugt.

```
module CORBA {
    enum completion_status { COMPLETED_YES, COMPLETED_NO,
                            COMPLETED_MAYBE};

    exception UNKNOWN {
        unsigned long    minor;
        completion_status completed;
    };
    exception BAD_PARAM {
        unsigned long    minor;
        completion_status completed;
    };
    exception NO_MEMORY {
        unsigned long    minor;
        completion_status completed;
    };
    exception COMM_FAILURE {
        unsigned long    minor;
        completion_status completed;
    };
};
```

## 14 Vorwärtsdeklarationen

- Problem: Zirkuläre Abhängigkeiten in den Deklarationendeklarationen
  - ◆ Schnittstelle **A** enthält Operation **op\_b()**, die Objekt vom Typ **B** liefert
  - ◆ Schnittstelle **B** enthält Operation **op\_a()**, die Objekt vom Typ **A** liefert
- Lösung: Vorwärtsdeklaration
  - ◆ Deklariere einen Bezeichner für einen Typ, aber nicht den Typ selbst
- Beispiel:

```
interface B;           // Forward declaration
interface A {
    B op_b();
};
interface B {
    A op_a();
};
```

## 15 IDL-Zusammenfassung

- Beschreibung von Datentypen und Schnittstellen von CORBA-Objekten
- C++-ähnliche Syntax
- Primitive Datentypen (**short**, **long**, **boolean**, **char**, ...)
- Zusammengesetzte Datentypen (**struct**, **union**, **enum**)
- Arrays
- "Template types" (**sequence**, **string**, **fixed**)
- Objekt-**interface** mit Attributen und Operationen
- Fehlersignalisierung mit Exceptions

## F.4 Abbildung von IDL nach Java

- Abbildung der IDL-Datentypen nach Java
- Abbildung von Objektschnittstellen nach Java

### 1 Allgemeine Bemerkungen

- Abbildung von IDL-Datentypen auf Java-Schnittstellen und Klassen
- Abbildungs-Spezifikation Version 1.2 vom August 2002
- Enthält Abbildung für die POA-Schnittstellen.
- Wichtige Ziele:
  - ◆ Portable Stubs
    - Stubs können über das Netzwerk geladen werden.
    - Stubs müssen mit jedem lokal installierten ORB funktionieren, unabhängig vom ORB-Kern.
    - Schnittstelle zwischen Stubs und ORB festgelegt, um Austauschbarkeit zu garantieren.
  - ◆ Umgekehrte Abbildung von Java nach IDL sollte möglich sein.
- Bezeichner von IDL werden unverändert in Java verwendet.
  - ◆ Bei Namenskollisionen wird \_ (underscore) vorangestellt.

## 2 Primitive Typen

### ■ Ganzzahlen

- ◆ **short** wird zu **short**
- ◆ **unsigned short** wird zu **short**
- ◆ **long** wird zu **int**
- ◆ **unsigned long** wird zu **int**
- ◆ **long long** wird zu **long**
- ◆ **unsigned long long** wird zu **long**
- ◆ Grosse **unsigned**-Werte werden in Java negativ!

### ■ Fließkommazahlen

- ◆ **float** wird zu **float**
- ◆ **double** wird zu **double**
- ◆ **long double** bisher nicht spezifiziert

## 2 Primitive Types (2)

### ■ Zeichen

- ◆ **char** wird zu **char**
- ◆ **wchar** wird zu **char**
- ◆ Weil Java **char** eine Obermenge von IDL **char** enthalten kann, kann das Marshalling eine **CORBA::DATA\_CONVERSION**-Exception auslösen.

### ■ **boolean** bleibt **boolean**

### ■ **octet** wird zu **byte**

### ■ **any**

- ◆ Klasse **org.omg.CORBA.Any**
- ◆ **insert**- und **extract**-Methoden für primitive Datentypes
- ◆ Für andere Datentypen **insert**- und **extract**-Methoden in der Helper-Klasse dieses Types

### 3 Helper-Klassen

- Eine Helper-Klasse für jeden IDL-Typ (hier *name*)

```
public class nameHelper {
    public static void insert( org.omg.CORBA.Any a, Name t )
        {...}
    public static name extract( org.omg.CORBA.Any a ) {...}
    public static org.omg.CORBA.TypeCode type() {...}
    public static String id() {...}
    public static name read(
        org.omg.CORBA.portable.InputStream istream ) {...}
    public static void write(
        org.omg.CORBA.portable.OutputStream ostream,
        name value ) {...}

    // nur für Helper von einem Interface
    public static name narrow( org.omg.CORBA.Object obj )
        {...}
}
```

### 3 Helper-Klassen (2)

- **extract** und **insert**
  - ◆ Methoden zum Einfügen und Entnehmen dieses Tyes in/aus **any**-Object
- **type** und **id**
  - ◆ Erfragen von Typ-Code und Typ-Information (Repository-ID) für diesen Typ
- **read** und **write**
  - ◆ Methoden zum Marshalling und Demarshalling in portablen Stubs
- **narrow**
  - ◆ Existiert nur in Helper-Klassen für **interface**
  - ◆ Modifikation der sichtbaren Schnittstelle einer Objektreferenz (Casting)



## 4 Holder-Klassen

- Java besitzt nur call-by-value-Semantik (Objektreferenzen können nicht verändert werden!).
- **out**- und **inout**-Parameters benötigen call-by-reference.
- Kapselung der Parameter in einem Holder-Objekt (hier für Typ *name*)

```
final public class nameHolder
    implements org.omg.CORBA.portable.Streamable {
    public name value;

    public nameHolder() {...}
    public nameHolder( name initial ) {...}

    public void _read( org.omg.CORBA.portable.InputStream i )
        {...}
    public void _write( org.omg.CORBA.portable.OutputStream o )
        {...}
    public org.omg.CORBA.TypeCode _type() {...}
}
```

## 5 IDLEntity

- Leere Markierungs-Schnittstelle
- Geerbt bei allen aus IDL generierten Schnittstellen und Klassen
- Deklaration:

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

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

## 6 Module

### ■ IDL:

```
module name {
    Deklarationen
};
```

### ■ Abbildung zu Java "package"

### ■ Java:

```
package name;
```

Abbildung für Deklarationen

### ■ Abbildung vom Modul CORBA zum Package org.omg.CORBA

## 7 Typ-Deklarationen

### ■ IDL:

```
typedef existing_type alias;
```

### ■ Java:

- ◆ Nur Helperklasse für *alias*
- ◆ Für den Typ selbst muss Abbildung von *existing\_type* benutzt werden.

### ■ Beispiel:

```
// IDL
typedef long IDNumber;

// Java
public class IDNumberHelper {...}
final public class IDNumberHolder
    implements org.omg.CORBA.portable.Streamable {
    public int value;
    ...
}
```

## 8 Strukturen

### ■ IDL:

```
struct name {
    Deklaration von struct-Elementen
};
```

### ■ Abbildung zu einer "public final"-Klasse mit Helper und Holder

- ◆ Elemente werden zu "public" Variablen
- ◆ Leerer Konstruktor, sowie Konstruktor, der alle Variablen initialisiert

### ■ Java:

```
public final class name
    implements org.omg.CORBA.portable.IDLEntity
{
    Mapping for structure elements as public variables
    public name() {}
    public name( Mapping_for_structure_elements ) {...}
}
```

## 8 Strukturen (2)

### ■ Beispiel:

```
// IDL
struct Beispiel {
    float value;
    char currency;
};

// Java
final public class Beispiel
    implements org.omg.CORBA.portable.IDLEntity
{
    public float value;
    public char currency;
    public Beispiel() {}
    public Beispiel( float value, char currency ) {
        this.value = value;
        this.currency = currency;
    }
}
```

## 9 Aufzählungen

### ■ IDL:

```
enum name {
    value1, value2, ...
};
```

### ■ Abbildung auf eine "public final"-Klasse mit **Helper** and **Holder**

- ◆ Werte werden auf Integer-Werte abgebildet (Bezeichner `_value1`, ...).
- ◆ Statische Instanzen innerhalb der Enumeration-Klasse

### ■ Java:

```
public final class name
    implements org.omg.CORBA.portable.IDLEntity
{
    public static final int _value1 = int_value1;
    public static final name value1 = new name( _value1 );
    ...
    private final int __value;
    private name( int value ) { this.__value = value; }
    public int value() { return __value; }
    public static name from_int( int value ) {...};
}
```

## 9 Aufzählungen

### ■ Beispiel:

```
// IDL
enum Color { GREEN, RED, BLUE };

// Java
final public class Color
    implements org.omg.CORBA.portable.IDLEntity
{
    private int __value;
    private int __size = 3;

    final public static int _GREEN = 0;
    final public static Color GREEN = new Color( _GREEN );
    final public static int _RED = 1;
    final public static Color RED = new Color( _RED );
    final public static int _BLUE = 2;
    final public static Color BLUE = new Color( _BLUE );

    private Color( int value ) { this.__value = value; }
    public int value() { return __value; }
    public static Color from_int( int value ) {
        switch( value ) {...}
    }
}
```

## 10 Arrays

### ■ IDL:

```
typedef element_type name[positive_constant] [positive_constant] ...;
```

### ■ Abbildung auf Java Arrays, sowie **Helper** und **Holder**

- ◆ Array-Elemente haben den Datentyp, der sich aus der Java-Abbildung von *element\_type* ergeben.

### ■ Beispiel:

```
// IDL
typedef long Matrix[3][3];

// Java
public class MatrixHelper {...}
final public class MatrixHolder
    implements org.omg.CORBA.portable.Streamable {
    public int[][] value;
    ...
}
```

## 11 Sequences

### ■ IDL:

```
typedef sequence<element_type> name; // unbounded
typedef sequence<element_type, positive_constant> name; // bounded
```

### ■ Abbildung genauso wie bei eindimensionalen Arrays

- ◆ Längenüberprüfung für "bounded sequences" wird nur beim Marshalling gemacht.

### ■ Beispiel:

```
// IDL
typedef sequence<long> Longs;

// Java
public class LongsHelper {...}
final public class LongsHolder
    implements org.omg.CORBA.portable.Streamable {
    public int[] value;
    ...
}
```

## 12 Zeichenketten

### ■ IDL:

```
typedef string name;           // unbounded
typedef string<positive_constant> name; // bounded
typedef wstring name;         // unbounded
```

### ■ Abbildung auf `java.lang.String`

- ◆ Exceptions beim Marshalling, wenn die Länge überschritten wird oder Zeichen nicht auf CORBA `char` abgebildet werden können
- ◆ Holderklasse: `org.omg.CORBA.StringHolder`

### ■ Beispiel:

```
// IDL
typedef string<80> Name;

// Java
public class NameHelper {...}
```

## 13 Konstanten

### ■ Symbolische Namen für spezielle Werte

### ■ IDL:

```
const type name = constant_expression;
```

### ■ Abbildung von lokalen Konstanten in IDL-Schnittstellen

- ◆ `"final public static"`-Variablen im Java-Interface
- ◆ Beispiel:

```
// IDL
interface Beispiel {
    const Color WARNING = RED;
};

// Java
public interface Beispiel ... {
    final public static Color WARNING = (Color) Color.RED;
    ...
}
```

## 13 Konstanten (2)

- Abbildung von Konstanten außerhalb von IDL-Schnittstellen
  - ◆ Eigene Klasse mit Namen der Konstanten und lokalen Wert *value*
  - ◆ Beispiel:

```
// IDL
module Beispiel {
    const Color WARNING = RED;
};

// Java
package Beispiel;
public interface WARNING {
    final public static Color value = (Color) Color.RED;
};
```

## 14 Schnittstellen

- IDL:
 

```
interface name {
    Deklaration von Attributen und Operationen (sowie Typen und Exceptions)
};
```
- Abbildung auf:
  - ◆ **public** Java interface *nameOperations*
  - ◆ **public** Java interface *name*
  - ◆ *nameHelper*- und *nameHolder*-Klassen
  - ◆ Stub- und Skeleton-Klasse

## 14 Schnittstellen (2)

### ■ Java:

```
public interface nameOperations
{
    Mapping for attributes and operations
}

public interface name extends org.omg.CORBA.Object,
    nameOperations, org.omg.CORBA.portable.IDLEntity
{...}

final public class nameHolder
    implements org.omg.CORBA.portable.Streamable {...}
public class nameHelper {...}
```

## 14 Schnittstellen – Vererbung

### ■ IDL:

```
interface name : inherited_interface1, inherited_interface2, ... {
    Declaration of additional attributes and operations
};
```

### ■ Abbildung auf Vererbung von Java-Schnittstellen

### ■ Java:

```
public interface nameOperations
    extends inherited_interface1Operations,
        inherited_interface2Operations, ...
{
    Mapping for additional attributes and operations
}

public interface name
    extends inherited_interface1, inherited_interface2, ...,
        nameOperations, org.omg.CORBA.portable.IDLEntity
{}

final public class nameHolder
    implements org.omg.CORBA.portable.Streamable {...}
public class nameHelper {...}
```



## 14 Schnittstellen – Attribute

### ■ IDL:

```
attribute type name;           // read & write
readonly attribute type name;  // read-only
```

### ■ Abbildung auf ein Paar von Zugriffsmethoden

### ■ Java:

```
public Mapping_for_type name();           // get attribute
public void name( Mapping_for_type );     // set attribute (not if read-only)
```

### ■ Beispiel:

```
// IDL
interface Account {
    readonly attribute float balance;
};

// Java
public interface AccountOperations {
    public float balance();
}
```

## 14 Schnittstellen – Operationen

### ■ IDL:

```
return_type name( parameter_list ) raises( exception_list );
```

### ■ Abbildung auf Methoden im Java-Interface

### ■ Java:

```
public Mapping_for_return_type name( Mapping_for_parameter_list )
    throws Mapping_for_exception_list;
```

## 14 Schnittstellen – Parameterübertragung

### ■ IDL:

```
( copy_direction1 type1 name1, copy_direction2 type2 name2, ... )
```

### ■ Abbildung von Parametertypen hängt von der Kopierrichtung ab:

- ◆ **in** nach *Mapping\_for\_type*
- ◆ **out** und **inout** nach *typeHolder*

### ■ Beispiel:

```
// IDL
interface Account {
    void makeWithdrawal( in float sum,
                        out float newBalance );
};

// Java
public interface AccountOperations {
    public void makeWithdrawal( float sum,
                               FloatHolder newBalance );
}
```

## 14 Schnittstellen

### ■ Beispiel:

```
//IDL
module Bank {
    interface Account {
        void withdraw(in double amount);
        void deposit(in double amount);
        void transfer(inout Account src, in double amount);
        readonly attribute double balance;
    };
};

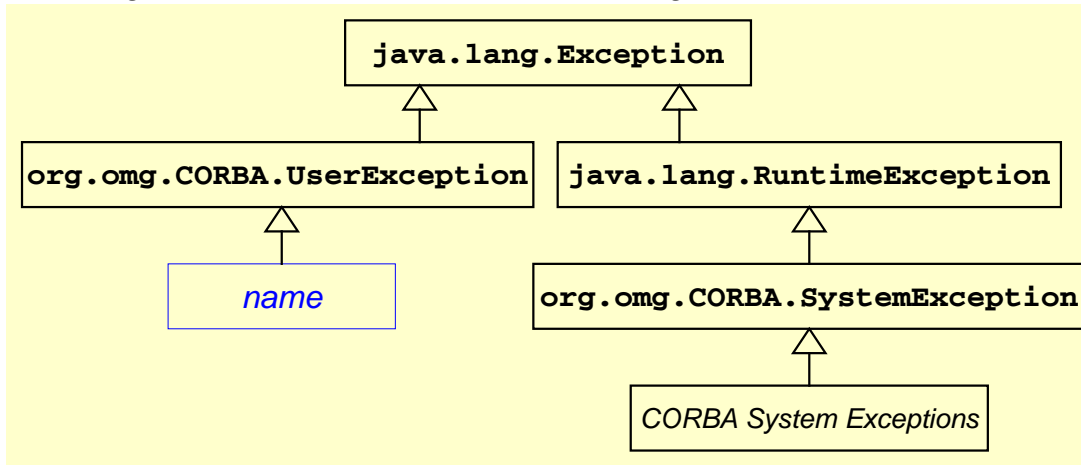
// Java
package Bank;
public interface AccountOperations {
    public void withdraw(double amount);
    public void deposit(double amount);
    public void transfer(AccountHolder src, double amount);
    public double balance();
}
public interface Account extends org.omg.CORBA.Object,
    AccountOperations, org.omg.CORBA.portable.IDLEntity {}
```

## 15 Exceptions

### ■ IDL:

```
exception name {
    Declarations of data elements
};
```

### ■ Abbildung auf "final public"-Klasse in folgender Klassenhierarchie:



## 15 Exceptions (2)

### ■ Abbildung von lokalen Exceptions innerhalb eines IDL-Interface *name* auf "final public"-Klassen im Paket *namePackage*

### ■ Beispiel:

```
// IDL
interface Account {
    exception Overdraft { float howMuch; };
    void withdraw( in double amount )raises( Overdraft );
};

// Java
public interface AccountOperations {
    public void withdraw( double amount )
        throws AccountPackage.Overdraft;
}
public interface Account extends ... {}

package AccountPackage;
final public class Overdraft
    extends org.omg.CORBA.UserException {
    public float howMuch;
    ...
}
```

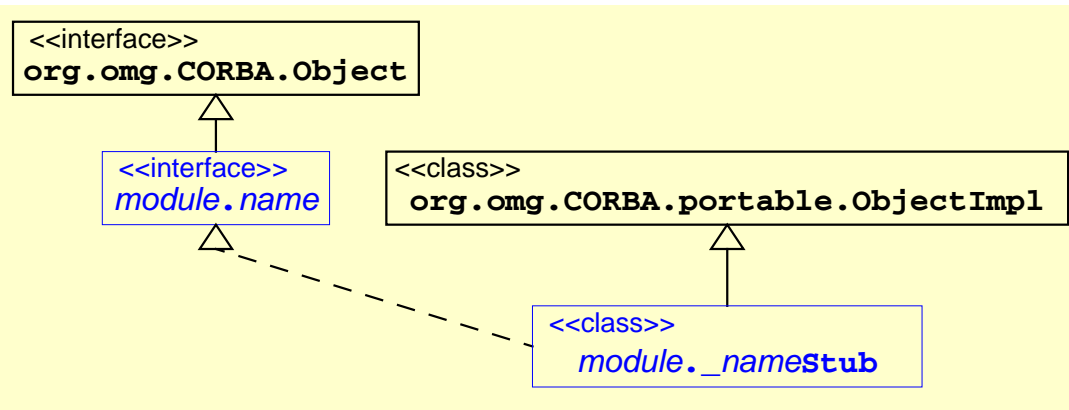
## 15 Exceptions (3)

- Zusätzlich: Erzeugung von Helper- und Holder-Klassen
- Abbildung von CORBA-Systemexceptions auf "final public"-Unterklassen von `org.omg.CORBA.SystemException`

```
package org.omg.CORBA;
abstract public class SystemException
    extends java.lang.RuntimeException {
    public int minor;
    public CompletionStatus completed;
    protected SystemException(        String reason, int minor,
                                     CompletionStatus status) {
        super(reason); this.minor = minor;
        this.status = status;
    }
}
// CORBA::UNKNOWN
final public class UNKNOWN
    extends org.omg.CORBA.SystemException {
    public UNKNOWN() ...
    public UNKNOWN(String reason) ...
    ...
}
```

## 16 Stubs

- Klient hat lediglich eine Java-Referenz auf das lokale Proxy-Objekt.
  - ◆ Stub-Objekt
  - ◆ Stub-Klasse wird automatisch aus der IDL-Beschreibung erzeugt
  - ◆ Stub-Objekte sind transparent für den Benutzer –  
Automatische Erzeugung und Vernichtung durch das CORBA-System.
- Klassenhierarchie für IDL-*interface module::name*



## 17 Zusammenfassung der Java-Abbildung

- Für jede IDL-Definition gibt es zwei Klassen:
  - ◆ Eine Holder-Klasse zur Übertragung von "out"- und "inout"-Parametern
  - ◆ Eine Helper-Klasse zum Marshalling und zum Einfügen/Extrahieren in/aus **any**-Objekten
- Primitive Datentypen werden auf Java-Typen abgebildet (Achtung: keine eins-zu-eins-Beziehung).
- IDL-Arrays und Sequenzen sind Java-Arrays.
- Für andere konstruierte Typen gibt es spezielle Klassen.
- IDL-Schnittstellen werden auf Java-Schnittstellen abgebildet.
- IDL-Exceptions sind spezielle Java-Exception-Klassen.
- Klienten besitzen Java-Referenzen auf Stub-Objekte.

## F.5 Java-Klienten

- Was fehlt noch für einen vollständigen Klienten?
- Wie bekommt man die erste Referenz auf ein CORBA-Objekt?
- Beispiel "Hello World!"

# 1 CORBA Pseudo-Objekte

- Keine echten CORBA-Objekte
  - ◆ Nur lokal sinnvoll
  - ◆ Keine entfernter Zugriff möglich.
- Beispiele:
  - ◆ **CORBA::Object**                      Features von CORBA-Objektreferenzen
  - ◆ **CORBA::ORB**                        Schnittstelle zu ORB-Features
  - ◆ **PortableServer::POA**           Schnittstelle zum Portable Object Adaptor
- Seit CORBA 2.5: Spezifikation von lokalen Schnittstellen in IDL ("local interface")

## 2 Objektreferenzen – CORBA::Object

- Operationen, die ein Klient an jedem CORBA-Objekt aufrufen kann:

```
module CORBA {
    interface Object {                                     // PIDL
        InterfaceDef get_interface();
        boolean is_nil();
        Object duplicate();
        void release();
        boolean is_a( in string logical_type_id );
        boolean non_existent();
        boolean is_equivalent( in Object other_object );
        unsigned long hash( in unsigned long maximum );
        Status create_request(    in Context ctx,
                                in Identifier operation,
                                in NVList arg_list,
                                inout NamedValue result,
                                out Request request,
                                in Flags req_flags );
        ...
    };
};
```

## 2 Objektreferenzen – CORBA::Object

### ■ **InterfaceDef get\_interface()**

- ◆ Liefert eine Interface-Beschreibung (vom Interface Repository) für dieses Objekt zurück.
- ◆ Normalerweise in Verbindung mit dem Dynamic Invocation Interface (DII) verwendet

### ■ **boolean is\_nil()**

- ◆ Überprüft, ob dies eine gültige Objektreferenz ist.

### ■ **Object duplicate()** **void release()**

- ◆ Kopieren und Löschen von Objektreferenzen
- ◆ Referenzzählung nur lokal im Klienten
- ◆ Die Objektkomplementierung wird nicht informiert.

## 2 Objektreferenzen – CORBA::Object

### ■ **boolean is\_a( in string logical\_type\_id )**

- ◆ Überprüft, ob das Objekt eine bestimmtes Interface implementiert.
- ◆ Interface Repository ID als String, z.B. **IDL:Bank/Account:1.0**

### ■ **boolean non\_existent()**

- ◆ Überprüft, ob dies eine Implementierung für dieses Objekt ist.

### ■ **unsigned long hash( in unsigned long maximum )**

- ◆ Hash, um Objektreferenzen zu unterscheiden

### ■ **boolean is\_equivalent( in Object other\_object )**

- ◆ Überprüft, ob zwei Referenzen auf das selbe CORBA-Objekt zeigen.
- ◆ Achtung: nur "best-effort"-Semantics
  - true: Referenzen zeigen auf das selbe Objekt.
  - false: Referenzen zeigen wahrscheinlich auf verschiedene Objekte.

### ■ **Status create\_request( ... )** Erzeuge einen DII-Request

### 3 Objektreferenzen – org.omg.CORBA.Object

#### ■ Java-Abbildung auf das Interface `org.omg.CORBA.Object`

```
package org.omg.CORBA;

public interface Object {
    boolean _is_a( String Identifier );
    boolean _is_equivalent( Object that );
    boolean _non_existent();
    int _hash( int maximum );
    org.omg.CORBA.Object _duplicate();
    void _release();
    ImplementationDef _get_implementation();
    InterfaceDef _get_interface();
    ...
}
```

#### ■ `duplicate` und `release` nicht wirklich notwendig.

- ◆ Java verwendet internen Garbage Collector anstatt Referenzzählung.

#### ■ Achtung: Einfach "Object" bedeutet immer `java.lang.Object`!

### 4 Die ORB-Schnittstelle – CORBA::ORB

#### ■ Standard-Operationen des ORBs

```
module CORBA {
    interface ORB {                                     // PIDL
        string object_to_string( in Object obj );
        Object string_to_object( in string str );

        typedef string ObjectId;
        typedef sequence<ObjectId> ObjectIdList;
        exception InvalidName {};
        ObjectIdList list_initial_services();
        Object resolve_initial_references(
            in ObjectId identifier ) raises (InvalidName);

        ...
    };
};
```



## 4 Die ORB-Schnittstelle – CORBA::ORB

- `string object_to_string( in Object obj )`  
`Object string_to_object( in string str )`
  - ◆ Umwandlung von Objektreferenzen in eindeutige Strings und umgekehrt
  - ◆ String-Format: `IOR:00202020...`
- `ObjectIdList list_initial_services()`
  - ◆ Liste von Diensten, die der ORB kennt, z.B. Namensdienst
- `Object resolve_initial_references( in ObjectId identifier ) raises (InvalidName)`
  - ◆ Liefert Objektreferenz auf den angeforderten ORB-Dienst zurück.
  - ◆ `ObjectId` ist ein String, z.B. `"NameService"`.

## 5 Die ORB-Schnittstelle – org.omg.CORBA.ORB

- Java-Abbildung auf abstrakte Klasse `org.omg.CORBA.ORB`

```
package org.omg.CORBA;

public abstract class ORB {

    public abstract String[] list_initial_services();
    public abstract org.omg.CORBA.Object
        resolve_initial_references( String object_name )
            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 );

    ...
}
```

## 6 ORB-Initialisierung

- Erster Schritt in jeder CORBA-Anwendung
- Liefert Referenz auf ein **CORBA::ORB**-Objekt.
- PIDL-Spezifikation

```
module CORBA {                                     // PIDL
    typedef string ORBId;
    typedef sequence<string> arg_list;
    ORB ORB_init( inout arg_list argv,
                  in ORBId orb_identifizier);
};
```

- Auswahl eines ORBs (falls es mehr als einen gibt) via ORBId
- ORB-Parameter in Kommandozeilenargumenten
  - ◆ z.B. **-ORB<suffix> <value>**

## 7 ORB-Initialisation – org.omg.CORBA.ORB

- Java: Initialisierung über statische Methoden in **org.omg.CORBA.ORB**

```
public abstract class ORB {
    ...
    public static ORB init(Strings[] args,
                           Properties props );
    public static ORB init( Applet app, Properties props );
    public static ORB init();
    ...
}
```

- ◆ Spezielle **init**-Methode für ORB innerhalb eines Applet
- ◆ **init()** ohne Parameter liefert nur einen Singleton-ORB.
  - Kann nur spezielle Strukturen wie Typecodes erzeugen
  - Nicht geeignet für entfernte Methodenaufrufe

- Java-Properties zur Auswahl von ORB-Features, z.B.

- ◆ **org.omg.CORBA.ORBClass** Klasse, die von **init** geliefert wird  
(implementiert **org.omg.CORBA.ORB**),  
z.B. **com.ooc.CORBA.ORB**

## 8 "Hello World"-Client - Interface

### ■ IDL-Interface: Hello.idl

```
module Example {
    interface Hello {
        string say( in string msg );
    };
};
```

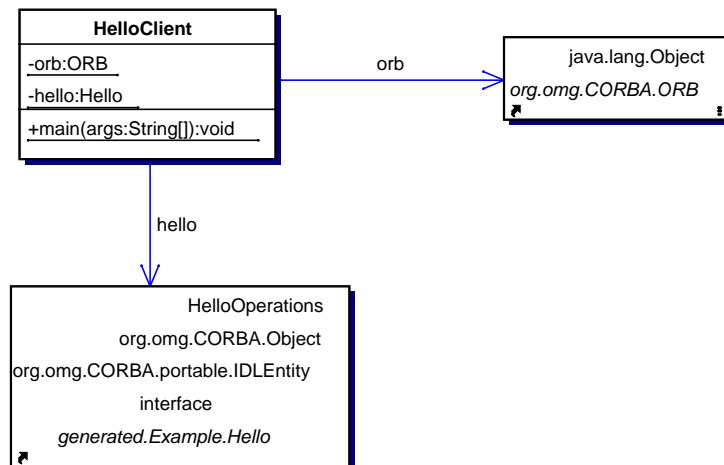
## 8 "Hello World"-Client - Implementierung

```
// client/HelloClient.java
package client;

import generated.Example.*;
import org.omg.CORBA.*;

public class HelloClient {
    public static void main( String[] args ) {
        try {
            // Initialise ORB
            ORB orb = ORB.init( args, null );
            // Read object reference from file Hello.ior
            BufferedReader br = new BufferedReader(
                new FileReader("Hello.ior"));
            String s = br.readLine();
            // Create a stub object
            org.omg.CORBA.Object o = orb.string_to_object(s);
            // Narrow to the Hello interface
            Hello hello = HelloHelper.narrow( o );
            // Do the call
            System.out.println( hello.say( " world!" ) );
        } catch( Throwable t ) {
            t.printStackTrace();
        }
    }
}
```

## 8 "Hello World"-Client - Klassen



## 8 "Hello World"-Client - Starten

### ■ Compilieren&Starten mit JDK 1.5 (/local/java-1.5/)

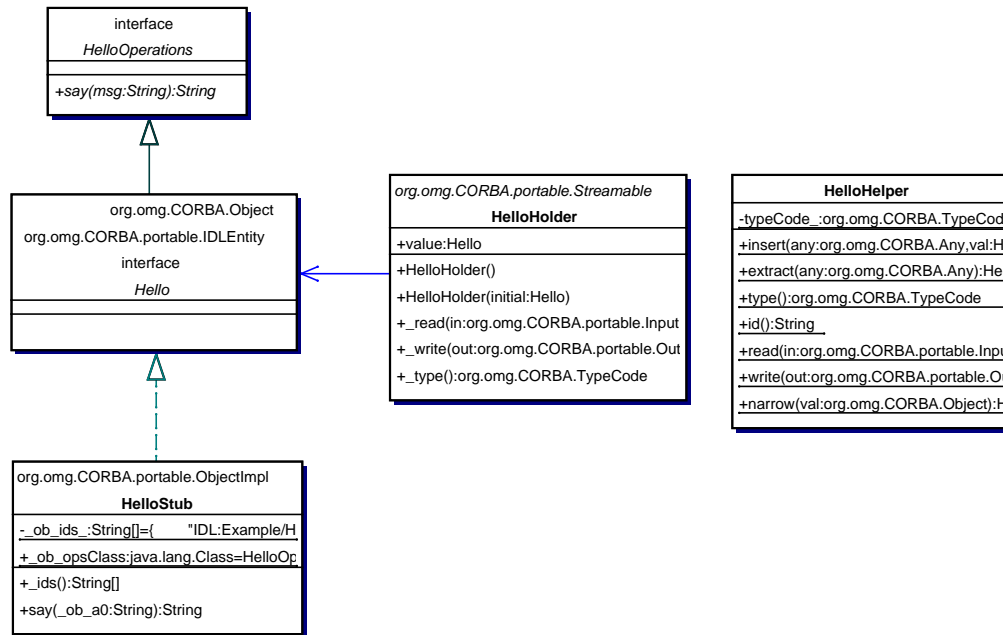
```
> idlj [-fclient] Hello.idl
> javac HelloClient.java
> java HelloClient
```

### ■ Compilieren&Starten mit JacORB (/local/JacORB/)

```
> idl Hello.idl
> javac HelloClient.java
> jaco HelloClient

> java -Xbootclasspath:${JACORB_HOME}/lib/jacorb.jar:\
    ${JRE_HOME}/lib/rt.jar:${CLASSPATH} \
    -Dorg.omg.CORBA.ORBClass=org.jacorb.orb.ORB \
    -Dorg.omg.CORBA.ORBSingletonClass=
        org.jacorb.orb.ORBSingleton HelloClient
```

## 8 "Hello World"-Client – Generierte Klassen

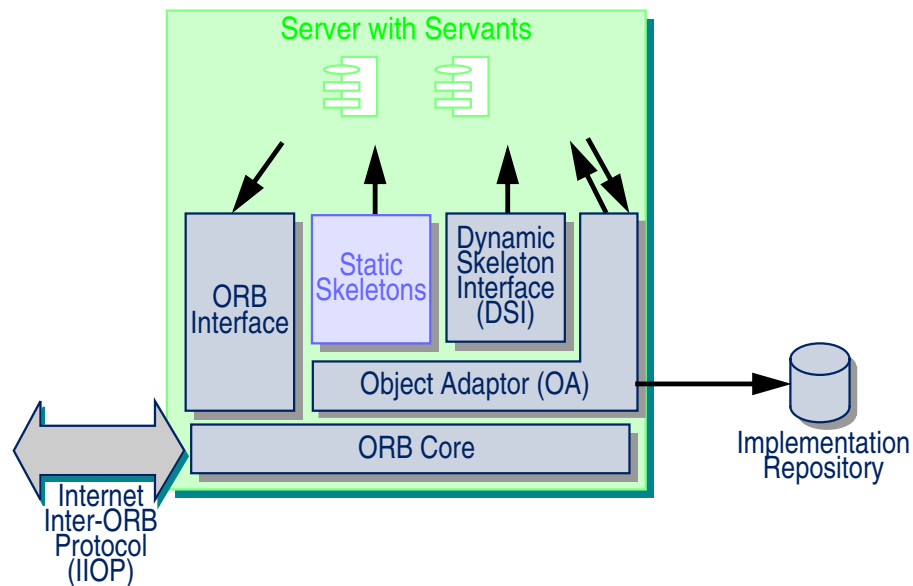


## F.6 Implementierung von CORBA-Objekten

### 1 CORBA-Objekte

- Zugriff auf CORBA-Objekte über Objektreferenzen
  - ◆ CORBA-Objekt existiert, sobald die erste Referenz erzeugt wird.
- Funktionalität von CORBA-Objekten durch Servants bereitgestellt
  - ◆ Servants sind in einer Programmiersprache geschrieben.
  - ◆ Servant existiert möglicherweise noch nicht, wenn das CORBA-Objekt erzeugt wird.
  - ◆ Zu jeder Zeit höchstens ein Servant pro CORBA-Objekt
  - ◆ Aber verschiedene Servants zu verschiedenen Zeitpunkten möglich

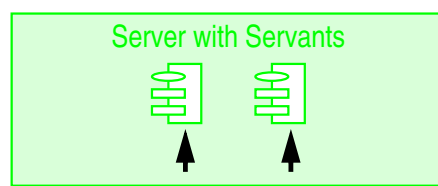
## 2 Server-Architectur



## 3 Der Server und Servants

### ■ Server

- ◆ Prozess, der Implementierungen (Servants) von CORBA-Objekten verwaltet

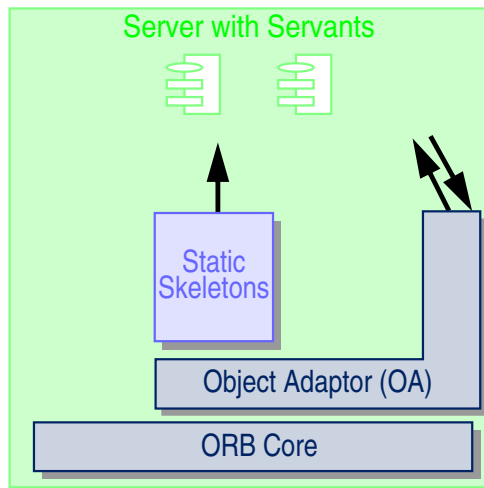


### ■ Servant

- ◆ Implementierung von genau einem CORBA-Objekt
- ◆ In OO-Sprachen: Spezielles Objekt, das die IDL-Schnittstelle implementiert
- ◆ In Nicht-OO-Sprachen: Menge von Funktionen, die die IDL-Schnittstelle implementieren, und eine Datenstruktur zur Identifikation der Instanz
- ◆ Viele Servants pro Server möglich

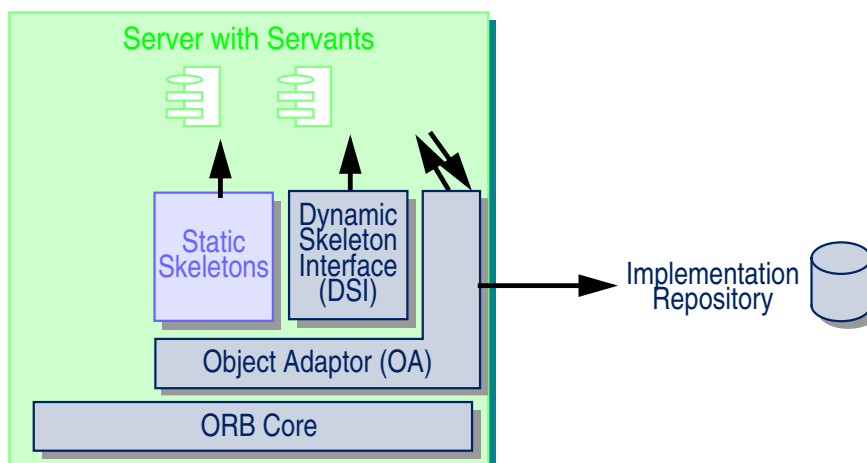
## 4 Statische Skeletons

- Können ebenfalls aus der IDL-Schnittstelle erzeugt werden
- Demarshalling von Aufrufparametern, Weiterleitung an den Servant
- Marshalling von Rückgabewerten oder Exceptions des Aufrufs



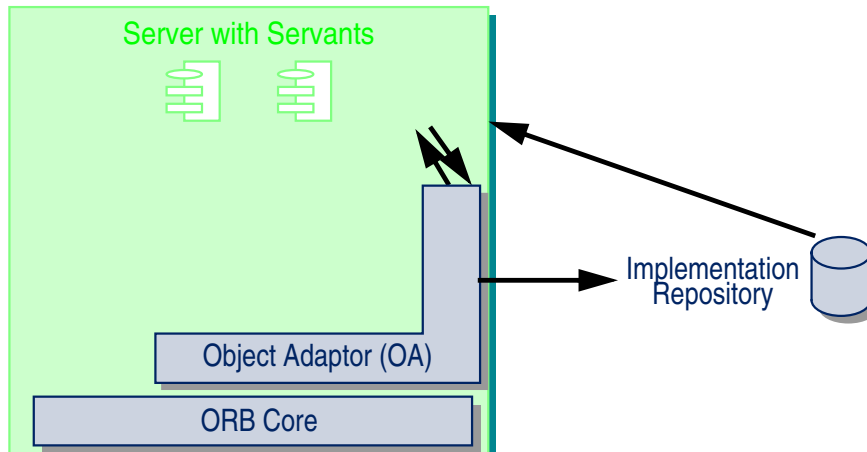
## 5 Object Adaptor

- Aufrufweiterleitung vom ORB-Core an die Skeletons
- Erzeugung und Verwaltung von Objektreferenzen
- Dynamische Aktivierung von Servants
- bis CORBA 2.1: Basic Object Adaptor (BOA), jetzt Portable OA (POA)



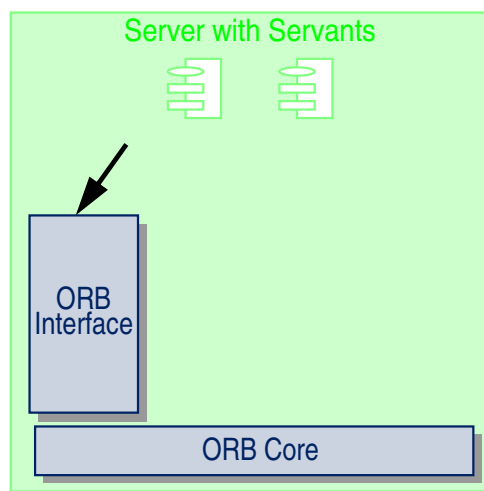
## 6 Implementation Repository

- Datenbank für Implementierungen von CORBA-Objekten
  - ◆ Informationen über welches Objekt von welchem Servant implementiert wird
- Oft mit "Location Forwarding Service" kombiniert
  - ◆ Dynamische Erzeugung von Server-Prozessen
  - ◆ Aufrufweiterleitung zu den dynamisch gestarteten Servern



## 7 ORB-Schnittstelle

- Export von initialen Objektreferenzen (ORB, OAs, Naming Service, ...)
- Manipulation von Objektreferenzen (Umwandlung in Strings und zurück)





## 8 Zusammenfassung Server

- Servants implementieren CORBA-Objekte.
- Server-Prozess verwaltet Servants.
- Skeletons leiten Aufrufe an Servants weiter.
- Object Adaptor verwaltet Lebenszyklus von Servants.
- Implementation Repository enthält Informationen über aktive Server und deren Servants.

### F.7 Einfacher Java-Server (POA)

## F.7 Einfacher Java-Server (POA)

- Verwendung des "Portable Object Adaptor" (POA)
- Erzeugen und Aktivieren von Servants

## 1 Servant

- Objekt, das die Funktionalität des CORBA-Objekts implementiert
- Semantik:
  - ◆ Jede Erzeugung eines Servant erzeugt ein zugeordnetes CORBA-Objekt.
  - ◆ Nach Zerstörung des Servant existiert das zugeordnete CORBA-Objekt nicht mehr.
- IDL:

```
module PortableServer {                               // PIDL
    native Servant;
};
```

- Java-Abbildung in eine abstrakte Klasse:

```
package org.omg.PortableServer;

public abstract class Servant {
    ...
};
```

## 2 Skeleton

- Basisklasse für die Implementierung des Servants
- IDL:

```
module module {
    interface name { ... };
};
```

- Java-Abbildung in eine abstrakte Klasse *namePOA*:

```
package module;

public abstract class namePOA
    extends org.omg.PortableServer.Servant
    implements org.omg.CORBA.portable.InvokeHandler,
               nameOperations
{
    public org.omg.CORBA.portable.OutputStream _invoke(
        String op, org.omg.CORBA.portable.InputStream i,
        org.omg.CORBA.portable.ResponseHandler handler)
    { ... }
    public name _this( org.omg.CORBA.ORB orb ) { ... }
    ...
}
```

## 2 Skeleton

### ■ `OutputStream _invoke( String op, InputStream i, ... )`

- ◆ Unmarshalling der Parameter von `InputStream`
- ◆ Weiterleitung an die Operation namens `op`
- ◆ Marshalling der Rückgabewerte nach `OutputStream`

### ■ `name _this( org.omg.CORBA.ORB orb )`

- ◆ Aktiviert ein neues CORBA-Objekt.
- ◆ Ordnet den Servant dem neuen CORBA-Objekt zu.
- ◆ Liefert eine Objektreferenz zurück (einen lokalen Stub).
- ◆ Mechanismus wird auch "*implizite Aktivierung*" genannt.

### ■ Skeleton wird automatisch durch den IDL-Compiler erzeugt, z.B. mit

```
> idlj -fserver Hello.idl
```

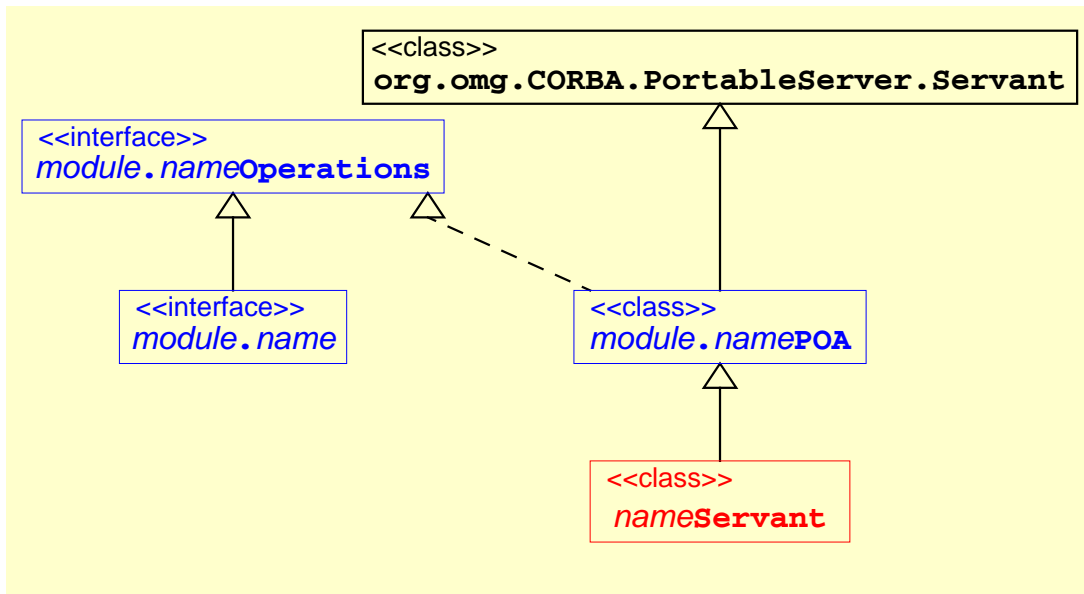
## 3 Implementierung Servant

- Eigene Implementierung der Objektfunktionalität
- Erbt von der Skeleton-Klasse
- Namenskonvention: `nameServant`

```
public class nameServant extends module.namePOA {
    Implementation of methods for attributes and operations
}
```

## 4 Hierarchie der Servant-Implementierung

- Klassenhierarchie für das IDL-Interface *module::name*



## 4 Hierarchie der Servant-Implementierung

- Beispiel: Erzeugte Schnittstellen und Klassen

```

public interface AccountOperations {
    public void withdraw( double amount )
        throws AccountPackage.Overdraft;
}

public interface Account extends org.omg.CORBA.Object,
    AccountOperations, ...
{
}

public abstract class AccountPOA
    extends org.omg.PortableServer.Servant
    implements AccountOperations, ...
{
    ...
}
  
```

- Beispiel: Servant

```

public class AccountServant extends AccountPOA {
    public void withdraw( double amount )
        throws AccountPackage.Overdraft {
        // Implementation of withdraw
    }
}
  
```

## 5 Was fehlt noch?

### ■ ORB-Initialisierung (siehe Klient)

### ■ POA-Aktivierung

#### ◆ Referenz auf RootPOA via **resolve\_initial\_references**

```
org.omg.CORBA.Object o =
    orb.resolve_initial_references( "RootPOA" );
```

#### ◆ "Narrow" auf das Interface **org.omg.PortableServer.POA**

```
org.omg.PortableServer.POA poa =
    org.omg.PortableServer.POAHelper.narrow( o );
```

#### ◆ POA aktivieren via **org.omg.PortableServer.POAManager**

```
poa.the_POAManager().activate();
```

### ■ ORB-Hauptschleife

#### ◆ Verarbeitung von Anforderungen starten: Methode **run()** in

```
org.omg.CORBA.ORB
orb.run();
```

## 6 "Hello World"-Server - Interface

### ■ IDL-Interface: Hello.idl

```
module Beispiel {
    interface Hello {
        string say( in string msg );
    };
};
```

## 6 "Hello World"-Server - Implementierung

### ■ Servant-Implementierung mittels Vererbung

```
// server/HelloServant.java

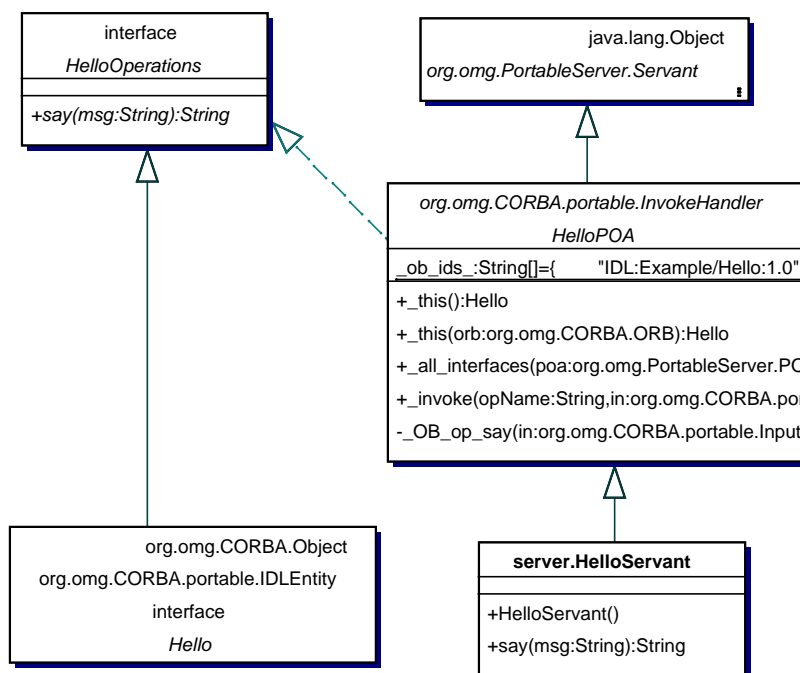
import generated.Example.*;

public class HelloServant extends HelloPOA {

    // Constructor
    public HelloServant() {
        super();
    }

    // Operation Beispiel::Hello::say from IDL
    public String say( String msg ) {
        System.out.println( "say() called" );
        return "Hello" + msg;
    }
}
```

## 6 "Hello World"-Server - Klassen



## 6 "Hello World"-Server - Startup

```
// server/HelloServer.java
import generated.Example.*;
import org.omg.CORBA.*;
import java.io.*;
import org.omg.PortableServer.*;

public class HelloServer {
    public static void main( String[] args ) {
        try {
            // Initialize ORB
            ORB orb = ORB.init( args, null );

            // Get the RootPOA
            POA poa = POAHelper.narrow(
                orb.resolve_initial_references( "RootPOA" ) );

            // Activate the RootPOA
            poa.the_POAManager().activate();

            // Create the hello servant object
            HelloServant hServ = new HelloServant();

            org.omg.CORBA.Object obj =
                poa.servant_to_reference(hServ);

            // to be continued ...
        }
    }
}
```

## 6 "Hello World"-Server - Startup (2)

```
// Write reference
PrintWriter pw = new PrintWriter(new FileWriter(
    "/proj/i4oovs/pub/hello/Hello.ior"));
pw.println( orb.object_to_string( obj );
pw.close();

// Wait for request
orb.run();

} catch( org.omg.CORBA.SystemException e ) {
    //...
} catch( Throwable t ) {
    //...
}
}
```

## 7 Zusammenfassung

---

- Initialisierung des ORB
- Aktivierung des POA
- Instantiierung von Servant(s)
- Aktivierung von Servant(s)
- Registrierung von Servant(s) beim Name Service
- Starten der ORB-Hauptschleife