

## Stubs & Skeletons

- Java Reflection API

- Dynamische Proxies als Stubs

- Generische Skeletons

- Aufgabe 2



- Bietet die Möglichkeit das Laufzeitverhalten von Applikationen zu analysieren und es gegebenenfalls sogar zu beeinflussen
- Tutorial: <http://docs.oracle.com/javase/tutorial/reflect/index.html>

“[...] This is a relatively advanced feature and **should be used only by developers** who have a **strong grasp of the fundamentals of the language**. [...]”

- Ermöglicht zur Laufzeit
  - Analyse von Attributen, Konstruktoren, Methoden, ...
  - Erzeugung neuer Objekte
  - Modifikation bestehender Objekte
  - Dynamische Methodenaufrufe
  - ...



- Zentrale Klasse: `java.lang.Class`
  - Pro Objekttyp existiert ein unveränderliches `Class`-Objekt
  - Beispiel

```
String x = "x";  
String y = "y";  
boolean b = (x.getClass() == y.getClass()); // -> b == true
```

- Zugriff auf `Class`-Objekte
  - Allgemein mit `class`-Attribut

```
Class c = <Klassenname>.class;
```

[Funktioniert auch bei primitiven Datentypen.]

- Objektreferenzen
  - Dynamisch mit `getClass()`

```
Class c = <Objekt>.getClass();
```

- Statisch mit `forName()`

```
Class c = Class.forName(<Klassenname>);
```



## ■ Analyse einer Klasse

```
public class Class<T> {  
    public Class<? super T> getSuperclass();  
    public Class<?>[] getInterfaces();  
    public Method[] getMethods();  
    [...]  
}
```

- `getSuperClass()` Zugriff auf `Class`-Objekt der Oberklasse
- `getInterfaces()` Zugriff auf `Class`-Objekte der von dieser Klasse direkt implementierten Schnittstellen
- `getMethods()` Rückgabe der öffentlichen Methoden dieser Klasse

## ■ Beispiel: Ausgabe aller Methoden der implementierten Schnittstellen

```
Class<?> c = <Objekt>.getClass();  
do {  
    for(Class<?> intf: c.getInterfaces()) {  
        for(Method m: intf.getMethods()) System.out.println(m);  
    }  
} while((c = c.getSuperclass()) != null);
```



## ■ Analyse einer Methode: java.lang.reflect.Method

```
public class Method {  
    public String getName();  
    public Class<?>[] getParameterTypes();  
    public Class<?> getReturnType();  
    public Class<?>[] getExceptionTypes();  
    public String toGenericString();  
    [...]  
}
```

- getName() Rückgabe des Methodennamens
- getParameterTypes() Zugriff auf Class-Objekte der Parameter
- getReturnType() Zugriff auf Class-Objekt des Rückgabewerts
- getExceptionTypes() Zugriff auf Class-Objekte der Exceptions
- toGenericString() Rückgabe der kompletten Methodensignatur

## ■ Dynamischer Aufruf einer Methode

```
public class Method {  
    public Object invoke(Object obj, Object... args);  
}
```



- Beispiel: get()-Methodenaufruf am VSBoard aus Übungsaufgabe 1

```
VSBoard board = new VSBoardImpl();  
[...] // Hinzufuegen von Nachrichten  
VSBoardMessage[] msgs = [...]; // Methodenaufruf (siehe unten)  
for(VSBoardMessage msg: msgs) System.out.println(msg);
```

- Gewöhnlicher Methodenaufruf

```
VSBoardMessage[] msgs = board.get(47);
```

- Methodenaufruf mit Java Reflection API

```
// Holen des Methoden-Objekts fuer get()  
Class<?> c = board.getClass();  
Class<?>[] paramTypes = new Class<?>[] { int.class };  
Method m = c.getMethod("get", paramTypes);  
  
// Zusammenstellung der Parameter und Aufruf der Methode  
Object[] params = new Object[] { 47 };  
VSBoardMessage[] msgs = (VSBoardMessage[]) m.invoke(board,  
                                                    params);
```

[Wie das Beispiel verdeutlicht, gibt es keinen Grund für den Aufruf einer Methode die Java Reflection API zu verwenden, solange alles Mögliche unternommen wurde, dies zu verhindern.]



## Stubs & Skeletons

Java Reflection API

Dynamische Proxies als Stubs

Generische Skeletons

Aufgabe 2



- Stellvertreter des entfernten Objekts beim Aufrufer einer Methode  
→ Implementierung der **Schnittstelle des entfernten Objekts**
- Zentrale Aufgabe: Umwandlung eines lokalen Methodenaufrufs am Stub in einen Fernaufruf am entfernten Objekt
  - Erzeugung einer Anfragenachricht
    - Eindeutige Kennung des Server-Prozesses
    - Eindeutige Kennung des entfernten Objekts
    - Eindeutige Kennung der aufzurufenden Methode
    - Einpacken der Aufrufparameter
  - Senden der Anfragenachricht über das Kommunikationssystem
  - Empfang einer Antwortnachricht über das Kommunikationssystem
  - Auspacken des Rückgabewerts
  - Übergabe des Rückgabewerts an den Aufrufer





## ■ Schnittstelle

```
public interface VSHelloInterface {
    public void setName(String name);
    public String getName();
    public void sayHello();
}
```

## ■ Implementierung

```
public class VSHelloImpl implements VSHelloInterface {
    private String name;

    public void setName(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }

    public void sayHello() {
        System.out.println("Hallo " + name);
    }
}
```



## ■ Stub für Beispiel-Schnittstelle

```
public class VSHelloStub implements VSHelloInterface {
    public void setName(String name) {
        // Anfrage (IDs, Parameter) erstellen und senden
        // Fuer synchronen Aufruf: Antwort empfangen
    }
    public String getName() {
        // Anfrage (IDs) erstellen und senden
        String s = [...] // Antwort empfangen und auspacken
        return s;
    }
    public void sayHello() {
        // Anfrage (IDs) erstellen und senden
        // Fuer synchronen Aufruf: Antwort empfangen
    }
}
```

## ■ Nachteile

- Hoher Implementierungsaufwand (vor allem bei Schnittstellenänderungen)
- Code-Duplikation
- Fehleranfällig



## ■ Grundidee

- Zur Laufzeit generierte Stellvertreterobjekte
- Konfigurierbare Schnittstellen
- Umleitung von lokalen Methodenaufrufen am Proxy auf einen zuvor registrierten **Invocation-Handler**
- Anwendungsspezifische Implementierung des Invocation-Handler
- Weiterführende Dokumentation
  - <http://docs.oracle.com/javase/6/docs/technotes/guides/reflection/proxy.html>
  - [http://www.roseindia.net/javatutorials/dynamic\\_proxies\\_tutorial.shtml](http://www.roseindia.net/javatutorials/dynamic_proxies_tutorial.shtml)

## ■ Dynamische Proxies als Stubs

- Implementierung beliebiger Schnittstellen → Proxies können als Stellvertreter für entfernte Objekte dienen
- Abfangen von lokalen Methodenaufrufen → Umwandlung in Fernaufrufe



- Implementierung eines Invocation-Handler
  - Bereitstellung einer `invoke()`-Methode, an die sämtliche am Proxy getätigten Methodenaufrufe delegiert werden
  - Informationen über den ursprünglichen Aufruf (Methode, Parameter) bleiben dabei erhalten
  - Rückgabewert von `invoke()` → Rückgabewert des ursprünglichen Aufrufs
- Schnittstelle: `java.lang.reflect.InvocationHandler`

```
public Object invoke(Object proxy, Method method,  
                    Object[] args) throws Throwable;
```

[Nicht zu verwechseln mit der `invoke()`-Methode der Java Reflection API.]

- `proxy` Der Proxy, an dem die `invoke`-Methode aufgerufen wurde
- `method` Das `Method`-Objekt der aufgerufenen Proxy-Methode
- `args` Array mit den Parametern des ursprünglichen Methodenaufrufs  
[Falls kein Parameter übergeben wurde: `args == null`]
- Die `invoke()`-Methode darf nur die Exceptions (`Throwable`) werfen, die in der Signatur der aufgerufenen Methode enthalten sind



- Proxy-Erzeugung mittels `Proxy.newProxyInstance()`

```
static Object newProxyInstance(ClassLoader loader,  
                               Class[] interfaces, InvocationHandler handler);
```

- `loader`      Class-Loader für die Proxy-Klasse  
    [Typischerweise der Class-Loader der zu implementierenden Schnittstelle; dieser lässt sich durch den Aufruf von `getClassLoader()` am Class-Objekt der Schnittstelle bestimmen.]
  - `interfaces`    Array der zu implementierenden Schnittstellen-Klassen
  - `handler`        Instanz des Invocation-Handler
- 
- Nach der Erzeugung des Proxy-Objekts kann dieses als Stellvertreter für die eigentliche Implementierung der vom Proxy bereitgestellten Schnittstellen genutzt werden



- Umleitung eines Methodenaufrufs auf ein lokales Objekt

```
public class VSHelloInvHandler implements InvocationHandler {
    private VSHelloInterface object;

    public VSHelloInvHandler(VSHelloInterface object) {
        this.object = object;
    }

    // Handler-Methode fuer alle lokalen Aufrufe am Proxy
    public Object invoke(Object proxy, Method m,
        Object[] args) throws Throwable
    {
        System.out.println("[Proxy] Methode: " + m.getName());
        if(args != null) {
            System.out.println("[Proxy] Args: " + args.length);
        }
        return m.invoke(object, args); // eigentlicher Aufruf
    }
}
```



- Main-Methode (der Klasse VSHelloTest) zum Testen des Proxy

```
public static void main(String[] args) {
    // Erzeugung des eigentlichen Objekts
    VSHelloInterface object = new VSHelloImpl();

    // Erzeugung eines Invocation-Handler
    VSHelloInvHandler handler = new VSHelloInvHandler(object);

    // Proxy-Erzeugung
    ClassLoader ldr = VSHelloInterface.class.getClassLoader();
    Class<?>[] intfs = new Class[] { VSHelloInterface.class };
    VSHelloInterface proxy = (VSHelloInterface)
        Proxy.newProxyInstance(ldr, intfs, handler);

    // Test: Methodenaufrufe am Proxy
    proxy.setName("Benutzer");
    proxy.sayHello();
    System.out.println(proxy.getName());
}
```



- Beispiel-Ausführung

```
> java VSHelloTest
[Proxy] Methode: setName
[Proxy] Args: 1
[Proxy] Methode: sayHello
Hallo Benutzer
[Proxy] Methode: getName
Benutzer
```

- Jeder Aufruf einer Methode an dem Objekt `proxy` wird durch den dynamisch generierten Proxy an die `invoke()`-Methode von `VSHelloInvHandler` weitergegeben
- Im verteilten Fall erfolgt im Invocation-Handler der eigentliche Fernaufruf am entfernten Objekt





## Stubs & Skeletons

Java Reflection API

Dynamische Proxies als Stubs

**Generische Skeletons**

Aufgabe 2



- Stellvertreter des Aufrufers einer Methode beim eigentlichen Objekt  
→ Nachbildung des Verhaltens eines lokalen Aufrufers
  
- Zentrale Aufgabe: Ausführung des eigentlichen Methodenaufrufs
  - Empfang einer Anfragenachricht über das Kommunikationssystem
    - Auspacken der Kennung des (jetzt lokalen) Objekts
    - Auspacken der Kennung der aufzurufenden Methode
    - Auspacken der Aufrufparameter
  - Bestimmung des Objekts mittels Kennung
  - Methodenaufruf am Objekt
  - Erzeugung einer Antwortnachricht mit dem Rückgabewert
  - Senden der Antwortnachricht über das Kommunikationssystem



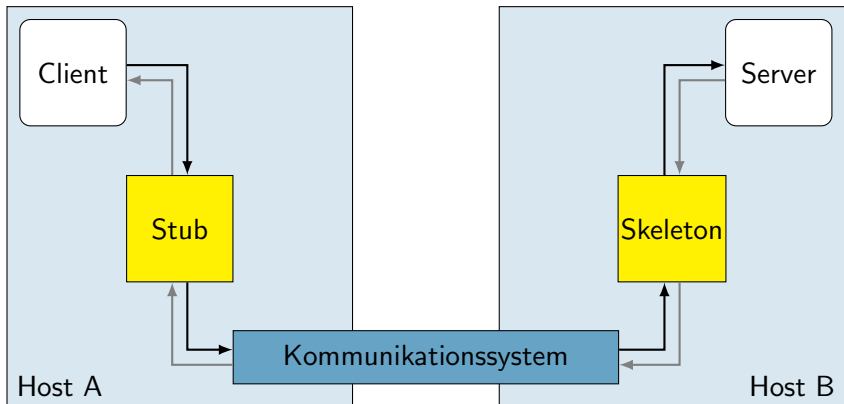
- Idee
  - Gemeinsame Skeleton-Implementierung für alle Fernaufrufe
  - Methodenaufrufe per Java Reflection API
- Problemstellung: Wie finde ich die richtige Methode?
  - Methodenname nicht eindeutig
  - Parameteranzahl nicht eindeutig
  - Eine eindeutige Kennung muss den Methodennamen sowie Anzahl und Typen sämtlicher Parameter berücksichtigen
- Lösungsansatz
  - Eindeutige Kennung per `Method.toGenericString()`, z. B. `VSBoard.get()`

```
public abstract vsue.rmi.VSBoardMessage[] vsue.rmi.VSBoard.get(int) throws java.lang.IllegalArgumentException,java.rmi.RemoteException
```
  - Bestimmung und Verwendung des richtigen Method-Objekts
    1. Abfrage aller Remote-Schnittstellen des Remote-Objekts
    2. Abfrage aller Methoden dieser Schnittstellen
    3. Vergleich der generischen Methoden-Strings mit dem in der Anfrage
    4. Aufruf von `invoke()` am gefundenen Methoden-Objekt



## Übungsaufgabe 2

- Dynamische Stubs und Skeletons
- Unterstützung von Rückrufen



- Implementierung als *Singleton*
  - Nur eine Instanz pro Java Virtual Machine
  - Zugriff über statische `getInstance()`-Methode
- Export von Objekten
  - Bereitstellung dynamischer Proxies für Fernaufrufe
  - Verwaltung der exportierten Remote-Objekte
- Aufruf von Methoden an exportierten Objekten
  - Suche des Objekts anhand der Objekt-ID
  - Bestimmung der Methode über ihren generischen Namen
  - Aufruf der Methode mit den übergebenen Parametern
  - Rückgabe des Rückgabewerts der aufgerufenen Methode

```
public class VSRemoteObjectManager {
    public static VSRemoteObjectManager getInstance();
    public Remote exportObject(Remote object);
    public Object invokeMethod(int objectID,
                               String genericMethodName, Object [] args);
}
```



## ■ Remote-Referenz: VSRemoteReference

```
public class VSRemoteReference implements Serializable {  
    private String host;  
    private int port;  
    private int objectID;  
}
```

- host Host-Name des Servers
- port Port-Nummer, auf dem der Server Verbindungen annimmt
- objectID Objekt-ID für Zugriff auf Remote-Objekt

## ■ Verwaltung von Verbindungen: Anpassung von VSServer

- Empfangen und Bearbeitung von Anfragen
- Erzeugung und Senden von Antworten



- Ziel: Transparente Fernaufrufe
    - Normalfall: Rückgabe des Ergebnisses
    - Fehlerfall: Abbruch der Ausführung auf Server-Seite (Exception)
      - Fernaufrufsystem muss Exception zum Aufrufer propagieren
  - Konsequenz für den Fehlerfall
    - Fangen der Exception beim Methodenaufruf auf Server-Seite
    - Weiterleitung der Exception zur Client-Seite
    - Werfen der Exception im Stub
  - Im Fernaufruf bedingte Fehler
    - Beispiele
      - Unerreichbarer Server
      - Verbindungsabbruch
      - ...
- Fernaufrufsystem muss Fehler (soweit möglich) behandeln  
[Näheres in der Tafelübung zu Übungsaufgabe 3.]



# Unterstützung von Rückrufen

- Parameterübergabe (analog zu Java RMI)
  - Call-by-Value
    - Übertragung einer Kopie des Parameters
    - Standardübergabeart
  - Call-by-Reference
    - Übertragung eines Stub für den Parameter
    - Voraussetzungen
      - \* Parameter implementiert Remote-Schnittstelle
      - \* Parameterobjekt wurde zuvor exportiert
- Implementierung
  - Erweitertes Marshalling im Invocation-Handler des Stub
    - Analyse der Aufrufparameter
    - Unterscheidung der Parameterübergabearten
  - Beachte: Call-by-Reference ist auch relevant für Rückgabewerte
- `Class.isAssignableFrom()`: Überprüfung, ob ein Objekt `o` eine bestimmte Schnittstelle (z. B. `Serializable`) implementiert

```
Object o = [...];  
if (Serializable.class.isAssignableFrom(o.getClass())) { [...] }
```

