technische universität
dortmund

Software- Engineering für langlebige Systeme

Thomas Ruhroth | Dortmund
SoSe 2014

Anbinden von Alt-Systemen

VL6

- Anbindung neuer Entwicklungen an Altsysteme

- Ziele:
  - Anbindungsmöglichkeiten kennenlernen.
  - Das Java Native Interface in Grundzügen benutzen können.

Problem neuer Entwicklungen für Altsysteme

- Umgebung von Altsystemen unterstützen neue Technikentwicklungen nicht
  - SmallTalk und SOA
  - Fortran 77 und XML

Häufige Lösung:

- Entwicklung neuer Komponenten in „modernen“ Sprachen/ mit „modernen“ Bibliotheken
- Anbindung an das Altsystem

Verknüpfung von Systemen

- Übergabe über existierende Eingabe/Ausgabesysteme
  - Dateien
  - Netzwerksockel
  - Pipes
- Austausch über gemeinsame Datenhaltung
  - Dateien
  - Datenbanken
- Direkte Einbindung auf auf Programmiersprachen Ebene
  - Linken von kompatiblen Kompilaten (Objekt-Files, ...)
    - Statisch/Dynamisch
  - Verknüpfen durch Sprachbrücken
    - z.B. Java Native Interface

Übergabe über existierende Eingabe/Ausgabesysteme

- Physische Koppelung
  - Serielle Schnittstelle
  - USB
  - Netzwerke
- Logische Koppelung
  - Pipes
  - Temporäre Dateien

Austausch über gemeinsame Datenhaltung

- Gemeinsame Nutzung von
  - Dateien
  - Datenbanken

- Programmierung unabhängig
- Datenhaltung stark abhängig
- Verbindung mehrerer Systeme ist leicht
- Direkte Reaktion auf Events nicht gegeben

Direkte Einbindung auf Programmiersprachen Ebene

- Linken von binärkompatiblen Objectfiles und Bibliotheken
  - C, Pascal, C++, Modula 2
  - cc -L. -shared -Wall -Werror -fpic -o libneueBib.so -lAusModula AusC.c

- Optimal, da binärkompatible Dateien keine Sprachunterscheidung besitzen.
- Manchmal: Anpassung von Little-Endian und Big-Endian aufeinander nötig
  - Wird häufig durch den Compiler/Linker erzeugt – transparent für den Programmierer

Direkte Einbindung auf Programmiersprachen Ebene

- Einbindung durch Bridges in der Sprache
  - Programmiersprache/Bibliothek bietet Anbindung an andere Sprachen an
  - Häufig C, C++ oder allgemein Shared Object Files

- Häufig, wenn Programmierparadigmen unterschiedlich
  - z.B. Java – C , Smalltalk – C oder Lisp -C
- Problem: Anpassung der Paradigmen
- Häufig eine Richtung einfach (andere kompliziert)
- Geschwindigkeit häufig ein Problem

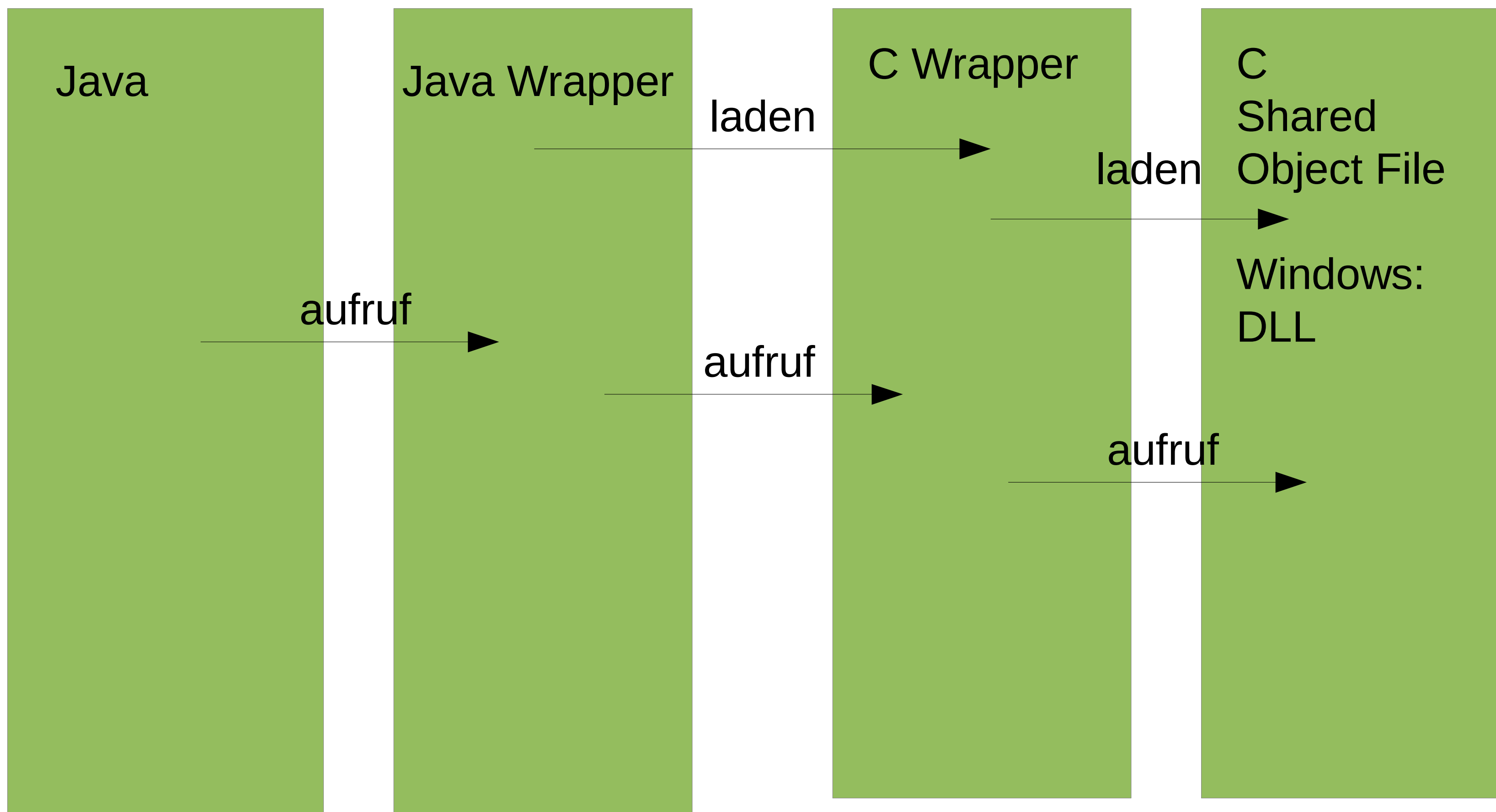
Java Native Interface

Zwei Richtungen

- Java → C
  - Recht einfach: laden, deklarieren, nutzen
  - Wird auch intern für die Anbindung an das Betriebssystem genutzt

- C → Java
  - Ein bisschen komplizierter
    - Objekte müssen durch imperativen Code simuliert werden
    - Strukturen für das Environment
    - Strukturen für Klassen
    - Suchen von Methoden

Java → C: Vorgehen



Laden von Shared Object Files in Java

```
public final class Wrapper {

    static{
        System.loadLibrary("xxxwrapper");
    }
```

- Name wird systemspezifisch angepasst
  - Linux: libxxxwrapper.so
  - Windows: wrapper.dll

Definieren der externen Methoden

- public native double fkt(int zahl);
-
- Native zeigt an, dass die Operation in einem geladenen Shared Object File gesucht werden soll (oder DLL)
- Die Namen werden für das Shared Object File angepasst:
  - Java_package1_package2_classname_fkt

Erzeugen einer Headerdatei

- Javah erzeugt eine für C und C++ geeignete Headerdatei
- In jni.h sind die genutzten speziellen Typen und Funktionen definiert
  - JNIEnv - Funktionen und Eigenschaften der Runtime-Umgebung
  - Jobject – Referenz auf ein Javaobjekt
  - Jdouble – Definition eines Java-kompatiblen double Wertes

Erstellen des C-Wrappers

- Für jede Operation aus dem erzeugten Headerfile muss die C-Operation aus dem existierenden Shared Object File zugeordnet werden
- Anpassungen von Typen (javakompatible und C-Typen)

Zusammenbauen der Teile

- Java normal kompilieren
- C: Bauen eines neues Shared Object Files
  - Original Shared Object File statisch oder dynamisch

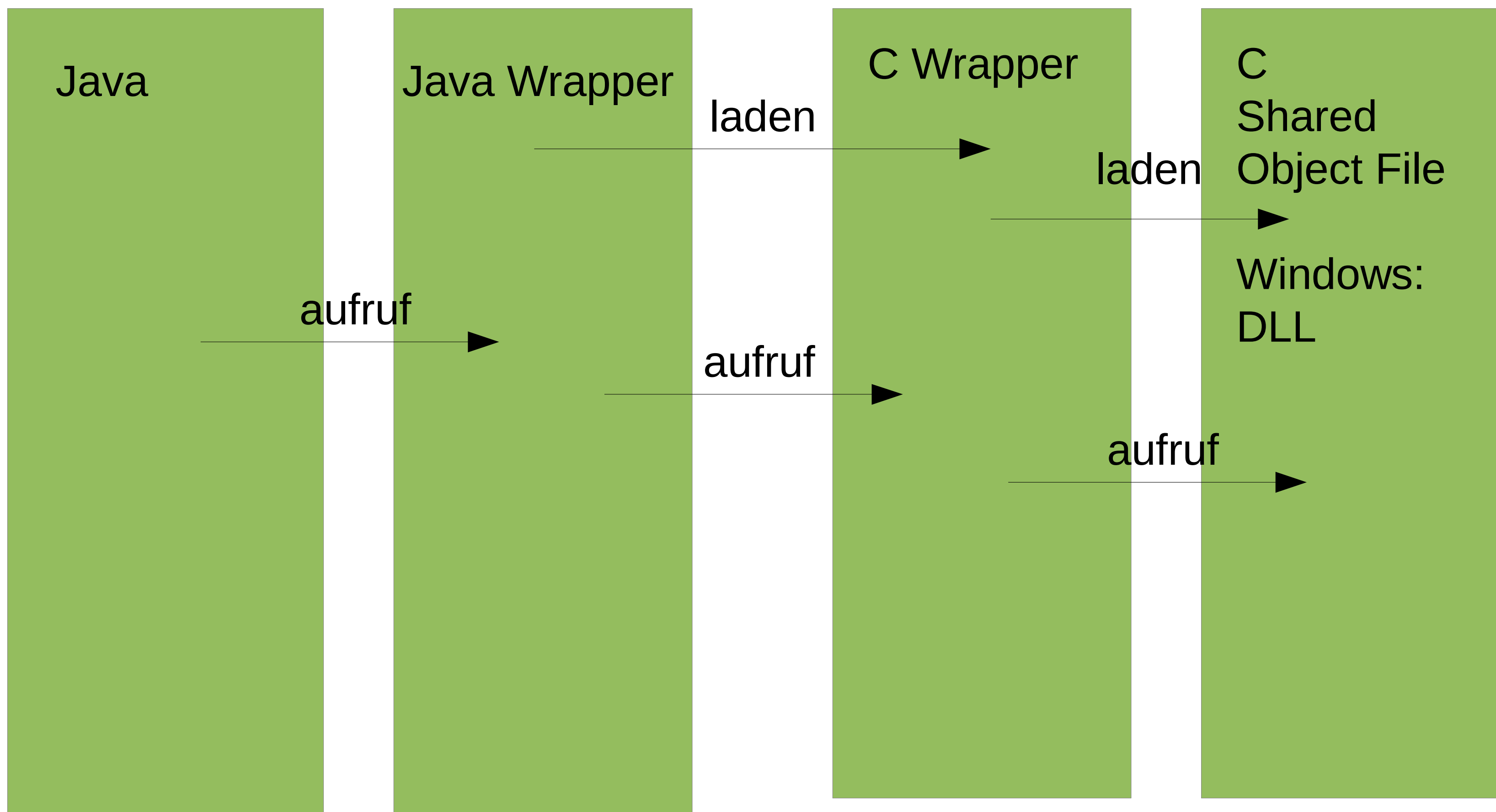
```
cc  -L.  -shared -Wall -I/usr/lib/jvm/java-7-oracle/include
-I/usr/lib/jvm/java-7-oracle/include/linux -o libreactorwrapper.so
-lreactor reactorwarapper.c
```

  - L setzen des Library Suchpfades
  - -shared = es soll ein Shared Object File (oder DLL) erstellt werden
  - -Wall alle Optionen werden zu den Subprozessen (Preprocessor, linker) weitergeleitet
  - -I Zusätzliche Include File Suchpfade
  - -o Name des zu erzeugenden Files
  - -l zu Verbindend Libaries
  -

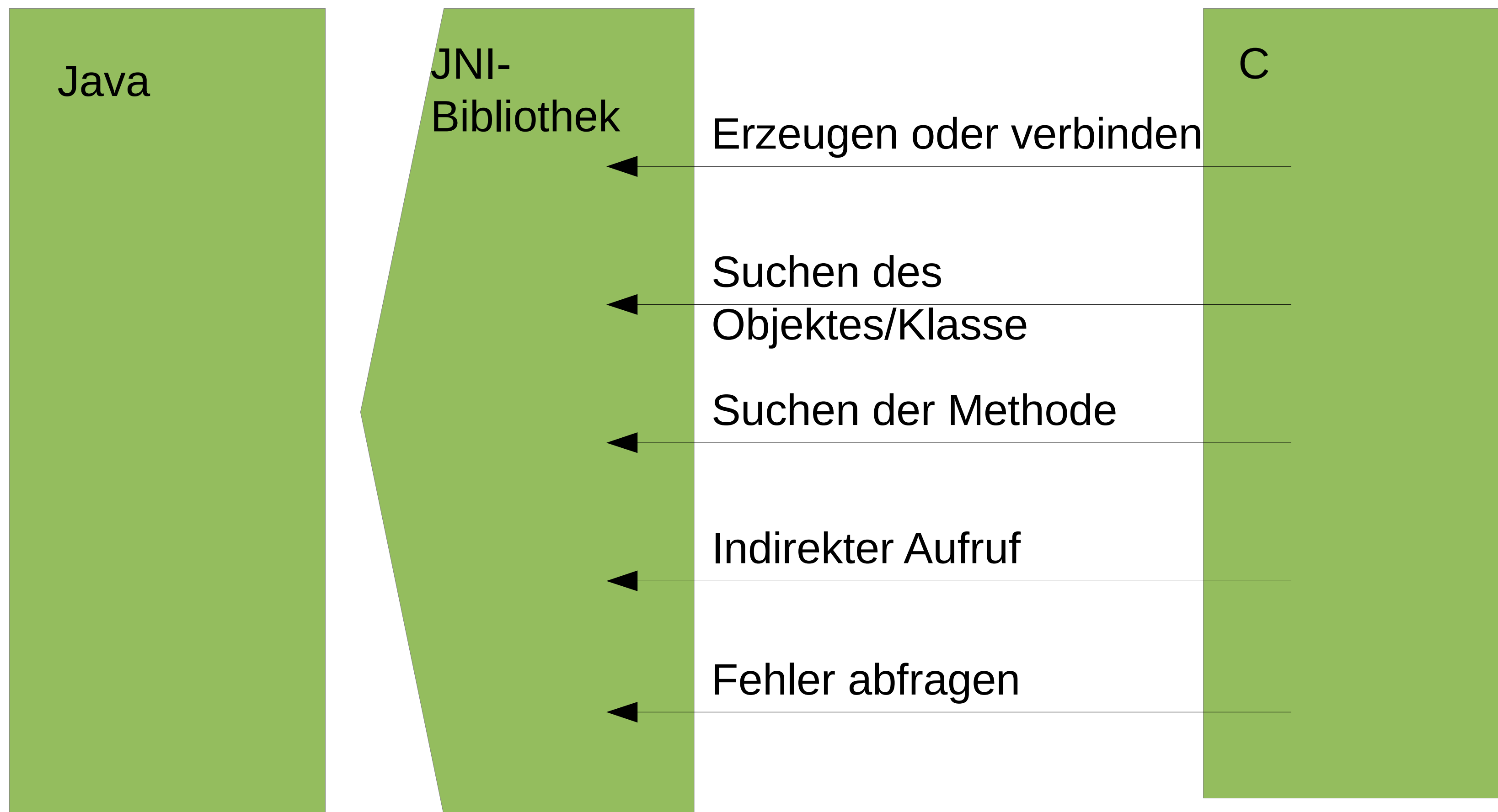
Nutzen

- java -Djava.library.path=/home/ruhroth/workspaces/web_1/LSys/csrc lsys.A1
- java.library.path – Suchpfade für Shared Object Files

Java → C: Vorgehen



C → Java: Vorgehen



Wiederholung: Objektorientierung

- Um die Simulation der Objektorientierung durch andere System verstehen zu können, muss man die Theorie der Objektorientierung verstanden haben.
- Problembegriffe:
  - Vererbung
  - Subtyping
  - Polymorphie
- Problem: In Programmiersprachen werden die Begriffe vermischt

Vererbung

- Ein Object s einer Subklasse S hat automatisch auch die Methoden, Variablen und Eigenschaften seiner Oberklasse O, sofern diese nicht explizit durch die Klasse S geändert werden.

- Vererbung sagt etwas über die Herkunft vor Objekteigenschaften aus, nicht über die Einsetzungbeziehung!

Subtyping

- Ein Typ C ist ein Subtype vom Typ A, gdw an allen Stellen, an denen ein Objekt vom Typ A genutzt wird kann ein Objekt vom Typ C genutzt werden.
- Kann durch Vor- und Nachbedingungen ausgedrückt werden:
- Pre C => Pre A
- Post A => Post C

- Subtyping ist in der Theorie unabhängig von der Vererbung

Subtyping und Vererbung

- Beide Konzepte sind in den meisten Programmiersprachen nicht sauber getrennt.
- Subtyping kann im begrenzten Umfang mit Interfaces und Erweiterungen zu vor- und Nachbedingungen erreicht werden.
- Viele Programmierer sehen Subtyping und Vererbung fälschlicherweise als identisch an.

Polymorphie

- Eine Methode ist polymorph, wenn sie in verschiedenen Klassen die gleiche Signatur hat, jedoch erneut implementiert ist. (Wikipedia)

Klassen und ihr Aufbau

- Klassen bestehen aus Daten
  - Daten auf Objektebene
  - Daten auf Klassenebene
- Klassen besitzen Operationen
  - Operationen auf Objektebene
  - Operationen auf Klassenebene
  - Erzeugungsoperationen
  - Zerstörungsoperationen

JNI – Zugriffe auf Klassen

- jclass FindClass(JNIEnv *env, const char *name);
  - z.B. java.lang.String ist "java/lang/String"
- jclass GetSuperclass(JNIEnv *env, jclass clazz);

  - ClassFormatError: if the class data does not specify a valid class.
  - ClassCircularityError: if a class or interface would be its own superclass or superinterface.
  - NoClassDefFoundError: if no definition for a requested class or interface can be found.
  - OutOfMemoryError: if the system runs out of memory.
  -
- Neue Fehler, da der Objectcode Fehler enthalten kann.

IsAssignableFrom

```
jboolean IsAssignableFrom(JNIEnv *env, jclass clazz1, jclass clazz2);
```

- Prüft ob ein Objekt der clazz1 sich in clazz2 gecasted werden kann.

Fehlerbehandlung

- Fehler blockieren JNI!
- jthrowable ExceptionOccurred(JNIEnv *env);
  Prüft ob eine Ecxeption aufgetreten ist. Die Exception bleibt aktiv bis sie durch den Native Code ExceptionClear() aufgerufen wurde oder die Exception im java-Code behandelt wurde.
- void ExceptionDescribe(JNIEnv *env);
  Gibt den Fehler einschließlich dem Stacktrace auf der Standartausgabe des C-Programmes aus
- void ExceptionClear(JNIEnv *env);
  Löscht die aktuelle Exception.

Neue Objecte

- jobject NewObject(JNIEnv *env, jclass clazz, jmethodID methodID, ...);
- jobject NewObjectA(JNIEnv *env, jclass clazz, jmethodID methodID, const jvalue *args);
- jobject NewObjectV(JNIEnv *env, jclass clazz, jmethodID methodID, va_list args);

- Erzeugt eine neues java Objekt. Die ID bezeichnet den Constructor der für die Erstellung genutzt werden soll. Diese ID muss durch die Operation GetMethodID() mit void (V) als Returntype.

Zugriffe auf Instanzvariablen

- GetObjectField(JNIEnv *env, jobject obj, jfieldID fieldID);
- GetBooleanField(JNIEnv *env, jobject obj, jfieldID fieldID);
- GetByteField(JNIEnv *env, jobject obj, jfieldID fieldID);
- GetCharField(JNIEnv *env, jobject obj, jfieldID fieldID);
- GetShortField(JNIEnv *env, jobject obj, jfieldID fieldID);
- GetIntField(JNIEnv *env, jobject obj, jfieldID fieldID);
- GetLongField(JNIEnv *env, jobject obj, jfieldID fieldID);
- GetFloatField(JNIEnv *env, jobject obj, jfieldID fieldID);
- GetDoubleField (JNIEnv *env, jobject obj, jfieldID fieldID);

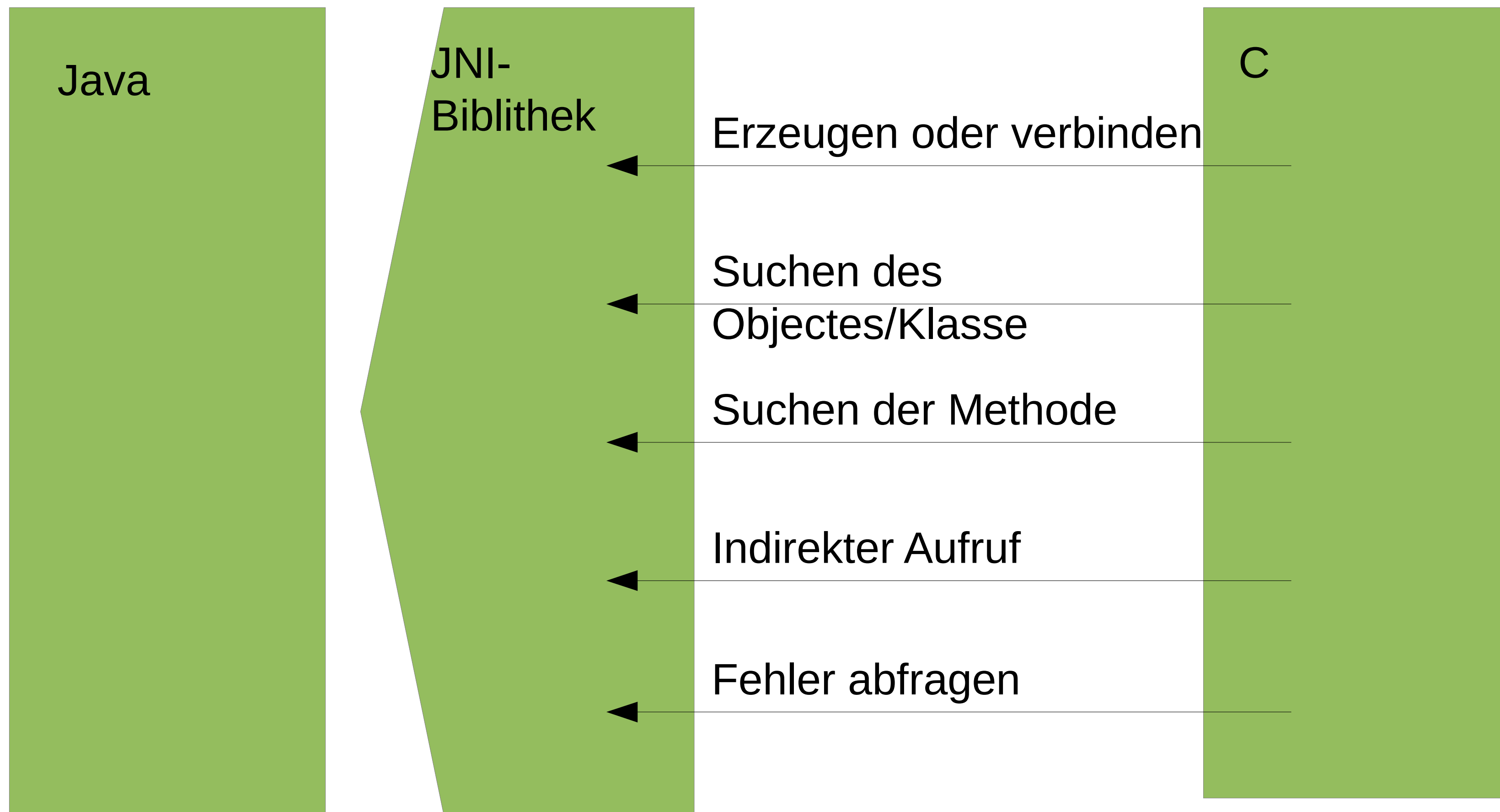
Zugriffe auf Instanzvariablen

- SetObjectField(JNIEnv *env, jobject obj, jfieldID fieldID, jobject value);
- SetBooleanField(JNIEnv *env, jobject obj, jfieldID fieldID, jboolean value);
- SetByteField(JNIEnv *env, jobject obj, jfieldID fieldID, jbyte value);
- SetCharField(JNIEnv *env, jobject obj, jfieldID fieldID, jchar value);
- SetShortField(JNIEnv *env, jobject obj, jfieldID fieldID, jshort value);
- SetIntField(JNIEnv *env, jobject obj, jfieldID fieldID, jint value);
- SetLongField(JNIEnv *env, jobject obj, jfieldID fieldID, jlong value);
- SetFloatField(JNIEnv *env, jobject obj, jfieldID fieldID, jfloat value);
- SetDoubleField (JNIEnv *env, jobject obj, jfieldID fieldID, jdouble value);

Methodenaufruf

- NativeType Call<type>Method(JNIEnv *env, jobject obj,
- jmethodID methodID, ...);
- NativeType Call<type>MethodA(JNIEnv *env, jobject obj,
- jmethodID methodID, const jvalue *args);
- NativeType Call<type>MethodV(JNIEnv *env, jobject obj,
- jmethodID methodID, va_list args);
- CallVoidMethod()
- CallVoidMethodA()
- CallVoidMethodV()
- CallObjectMethod()
- CallObjectMethodA()
- CallObjectMethodV()
- ...

C → Java: Vorgehen



Nächste Woche:

Reengineering