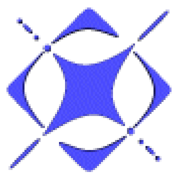
Fachhochschule
Münster University of
Applied Sciences

Java Reflection

Meta-Programmierung mit der *java.lang.reflection* API.

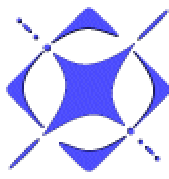
Prof. Dr. Nikolaus Wulff

Java Reflection

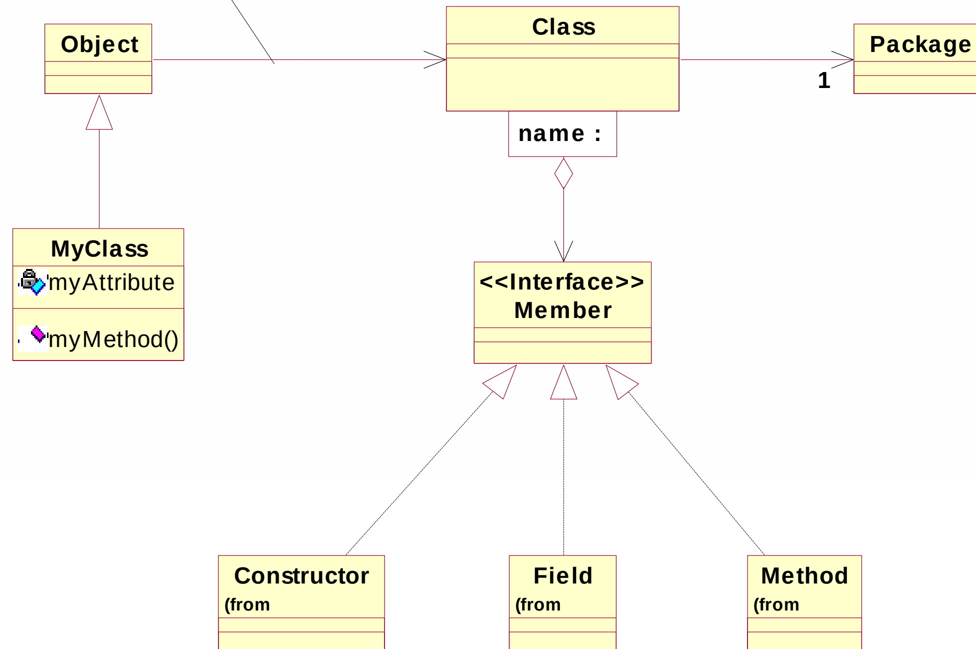


- Die Java Reflection API liefert per ***Introspection*** Informationen über Klassen => Meta-Daten.
- Zu jedem Java Objekt ***x*** vom Typ ***X*** gehört ein Class-Objekt ***X*** als Instanz der Klasse Class<***X***>=***X***.class.
- Das Class Objekt ***X*** ist „die Blaupause“ für alle Operationen die ***x*** ausführen kann, liefert Informationen über alle Annotationen, Parameterisierungen und alle Felder von ***x***.
- Reflection erlaubt schreibenden und lesenden Zugriff auf Felder von ***x*** und kann Methoden von ***x*** ausführen.
- Besonders Codegeneratoren verwenden Reflection.

Java Reflection API



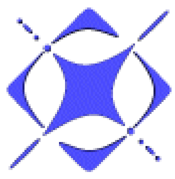
```
this.getClass()
```



Weitere Metainformationen die hinzugekommen sind:

- Annotation und Enum seid JDK 1.5
- Module seid JDK 1.9

Meta-Informationen

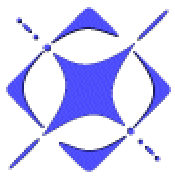


Typische Meta-Informationen zu einem Objekt ***x*** sind:

- isAnnotation: ist ***x*** eine Annotation?
- isEnum: ist ***x*** ein Aufzählungstyp?
- isPrimitive: ist der Typ von ***x*** ein Java Primitive?
- isInterface: ist ***x*** eine Schnittstelle?
- getName: wie heißt die Klasse von ***x***?
- isArray: handelt es sich bei ***x*** um ein Feld?

```
public static void introspect(Object x) {
   Class<?> X = x.getClass();
   out.printf("getName %s \n",X.getName());
   out.printf("isArray %b \n",X.isArray());

}
```

Felder und Methoden

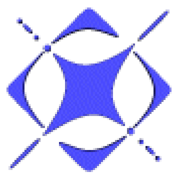


Neben den reinen Informationen ist auch der Zugriff auf Methoden oder Felder vom Objekt ***x*** möglich:

- getAnnotation(s): liefert (alle) Annotationen von ***x***.
- getMethod(s): liefert (alle) Methoden von ***x***.
- getField(s): liefert (alle) Felder/Attribute von ***x***.

```
public static void introspect(Object x) throws Exception {
   Class<?> X = x.getClass();

   Method m = X.getMethod("toString", new Class<?>[0]);
   out.printf("Method: x.toString %s \n",m);

   Field f = X.getField("myField");
   out.printf("Field:  x.myField %s \n",f);

}
```

Manipulationen



- Felder von ***x*** lassen sich Auslesen und Verändern.

```
Field f = X.getDeclaredField("myField");
out.printf("Field:  x.myField %s \n",f);

f.set(x,"another Value");
out.printf("Field:  x.myField %s \n",f.get(x));
```

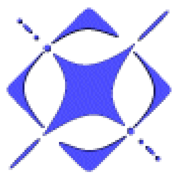
- Methoden von ***x*** lassen sich ausführen.

```
List<Integer> list = ...

Method m = X.getMethod("sort", List.class);
m.invoke(x, list);
```

  – Entspricht: `x.sort(list);`

Die Instanz ***x*** muss übergeben werden

Fabrik Methoden



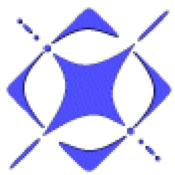
- Das Class Objekt liefert neben den Methoden als Meta-Informationen auch die Konstruktoren, die es gestatten neue Objekte vom Typ ***X*** zu erzeugen.
- Entweder mit Bean Konstruktor ohne Argumente:

```
Class<?> X = x.getClass();

Object z = X.newInstance();
out.printf("new Object %s\n",z);
assert(z.getClass() ==  x.getClass());
```

- oder mit Argumenten:

```
Object[] args = ...
Constructor<?>[] ctors = X.getConstructors();
Object y = ctors[0].newInstance(args);
out.printf("new Object %s\n",y);
```

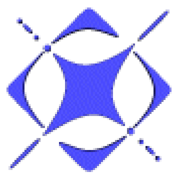
Class Objekte



- Ein spezifisches Class Objekt lässt sich auch ohne ein konkretes Objekt ***x*** zur Laufzeit erstellen, sofern der ***vollqualifizierte*** Name von ***X*** bekannt ist.
- Die statische Methode ***forName*** liefert eine Class Instanz passend zum übergebenen Namen:

```
Class<?> clazz = Class.forName("java.lang.Thread");

Object o = clazz.newInstance();
System.out.println("Object " + o);
Thread t = (Thread) o;
t.start();
```

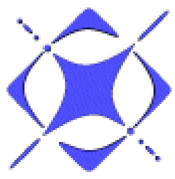
Code-Generatoren



- Code-Generatoren können zur Laufzeit eigenen Byte-Code generieren, diesen vom ***ClassLoader*** überprüfen lassen und dann Instanzen erzeugen.

```
class Generator extends ClassLoader {
    public Class<?> createClass(String className,byte[] code) {
        return defineClass(className,code,0,code.length);
    }
}

void loaderExample() throws InstantiationException, IllegalAccessExc
    Generator loader = new Generator();
    byte[] generated = ... // a miracle generates byte-code
    String className = "XYZ";

    Class<?> genClazz = loader.createClass(className, generated);
    Object instance = genClazz.newInstance();
    out.printf("generated Object %s\n",instance);
}
```

ASM Byte-Code Bibliothek



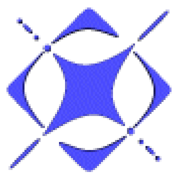
- Byte-Code kann z.B. mit Hilfe der ASM Bibliothek zur Laufzeit erstellt werden.

```
ClassWriter cw = new ClassWriter(ClassWriter.COMPUTE_FRAMES);
TraceClassVisitor cv = new TraceClassVisitor(cw, new PrintWriter(System.out));
cv.visit(V1_5,ACC_PUBLIC+ACC_SUPER,"test/Test",signature,"java/lang/Object",interfaces);
cv.visitField(ACC_PRIVATE, "x", "I", signature, null).visitEnd();
    ...
MethodVisitor mv = cv .visitMethod(ACC_PUBLIC, "getX", "()I", signature, exceptions);
mv.visitCode();
mv.visitVarInsn(ALOAD, 0);
mv.visitFieldInsn(GETFIELD,"test/Test", "x", "I");
mv.visitInsn(IRETURN);
mv.visitMaxs(1,1);
mv.visitEnd();
    ...
cv .visitEnd();

byte[] generated  = cw.toByteArray();

Generator generator = new Generator();

Class<?> clazz = generator.createClass("test/Test",generated);
System.out.println("clazz: " + clazz);
Object instance = clazz.newInstance();
```

```
// access flags 0x1
  public getX()I
    ALOAD 0
    GETFIELD test/Test.x : I
    IRETURN
```

```
    public int getX() {
        return x;
    }
```

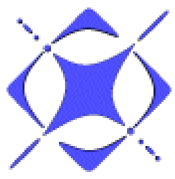
java.lang.reflection.Proxy



Reflection wurde z.B. zur Generierung von Proxies beim RessourcePool<X> eingesetzt (X ist ein Interface):

```
protected X createProxy(X x) {
    Class<?> clazz = x.getClass();
    X proxy = (X) java.lang.reflect.Proxy.newProxyInstance(
            clazz.getClassLoader(), clazz.getInterfaces(),
              new ResourceHandle(x));
    return proxy;
}

class ResourceHandle implements java.lang.reflect.InvocationHandler
    private X x;

    public Object invoke(Object proxy, Method m, Object[] args)
                                            throws Throwable {
        checkAccess();
        return m.invoke(x, args);
    }
```

Zusammenfassung



- Ausgehend vom Class Objekt lassen sich alle Felder und Methoden einer Klasse abfragen.
- Die Reflection API erlaubt die Analyse, Manipulation und Erzeugung von beliebigen Objekten.
- Mit Hilfe von Byte-Code Generatoren lassen sich neue Klassen, wie Proxies oder Adapter zur Laufzeit durch Übergabe des Byte-Code an den ClassLoader erzeugen.
- Der Java SecurityManager kann die Verwendung der Reflection API einschränken.