

Reflection

Reflection API

- ❖ dient dazu, die Bestandteile von Klassen(Datenelemente, Konstruktoren und Methoden) dynamisch zur Laufzeit zu analysieren und zu benutzen.
- ❖ Es gestattet somit, in einem Java-Programm Klassen und Interfaces zu benutzen, die zum Zeitpunkt des Kompilierens noch nicht bekannt sind.

Anwendungsfälle

- ❖ dynamisches Laden von Klassen in eine Anwendung
 - ❖ Beispiel (Treiber)
- ❖ Java-Beans, um die Datenelemente der Bean-Klassen zu ermitteln und deren Werte auszulesen bzw. neu zu setzen
 - ❖ Frameworks

Allgemeiner formuliert...

- ❖ ...wenn zur Laufzeit Klassen eingebunden werden sollen
 - ❖ die zum Compile-Zeitpunkt noch nicht bekannt sind
 - ❖ oder deren Schnittstellen nicht durch Interfaces oder abstrakte Klassen definiert sind, sondern nur Schnittstellenkonventionen und -mustern folgen.

Wichtige Klassen

- ❖ Constructor
- ❖ Method
 - ❖ verfügt über Methoden zum Aufrufen
- ❖ Field
 - ❖ verfügt über Methoden zum Setzen und Lesen.

Erzeugung erfolgt über ...

- ❖ `getConstructor()`, `getMethod()`, `getField()` der Klasse `Class`.
- ❖ Es muss also zunächst immer das `Class`-Objekt für die Klasse ermittelt werden, auf deren Bestandteile dynamisch zugegriffen werden soll.

Wie komme ich an ein Class-Objekt?

- ❖ Class-Objekte kann nur die JVM erzeugen über
 - ❖ `getClass()`, wenn Exemplar der Klasse verfügbar.
 - ❖ .class Klassenvariable vom Typ Class
 - ❖ `Class.forName(String)`

Beispiele

```
public class GetClassObject {
    @SuppressWarnings("rawtypes")
    public static void main(String[] args) {
        Class<ArrayList> c1 = java.util.ArrayList.class;
        System.out.println( c1 );          // class java.util.ArrayList
        Class<?> c2 = new java.util.ArrayList().getClass();

        System.out.println( c2 );          // class java.util.ArrayList
        try {
            //immer voll qualifiziert
            Class<?> c3 = Class.forName( "java.util.ArrayList" );
            System.out.println( c3 );          // class java.util.ArrayList
        }
        catch ( ClassNotFoundException e ) { e.printStackTrace(); }
    }
}
```

Ein Class Objekt beschreibt...

- ❖ eine Schnittstelle
 - ❖ isInterface()
- ❖ eine Klasse
- ❖ einen primitiven Datentyp
 - ❖ isPrimitive()
- ❖ einen Array-Typ
 - ❖ isArray()

Beispiel

```
class CheckClassType
{
    public static void main( String[] args )
    {
        checkClassType( Observer.class );
        checkClassType( Observable.class );
        checkClassType( (new int[2][3][4]).getClass() );
        checkClassType( Integer.TYPE );
    }

    static void checkClassType( Class<?> c )
    {
        if ( c.isArray() )
            System.out.println( c.getName() + " ist ein Feld." );
        else if ( c.isPrimitive() )
            System.out.println( c + " ist ein primitiver Typ." );
        else if ( c.isInterface() )
            System.out.println( c.getName() + " ist ein Interface." );
        else
            System.out.println( c.getName() + " ist eine Klasse." );
    }
}
```

Der Name der Klasse

- ❖ vollqualifizierter Name über getName()
 - ❖ new java.util.Date().getClass().getName();
- ❖ Felder werden über getName [kodiert
 - ❖ System.out.println((new int[2][3]
[4]).getClass().getName()); // [[[I
 - ❖ Kürzel B für Byte, C für Char, I für Int etc.
 - ❖ LElementtyp; Klasse oder Schnittstelle

instanceof

- ❖ binärer Operator instanceof testet ob ein Objekt ein Exemplar einer Klasse oder Oberklasse ist.
 - ❖ Prüfung ob Objekte Zuweisungskompatibel
 - ❖ Normalerweise sollte Typname(rechter Operator) bekannt sein
 - ❖ ist er das vielleicht mal nicht kann die Methode `isInstance` weiterhelfen

instanceof

```
public class InstanceOf {  
    public static void main(String[] args) {  
        String str = "test";  
        System.out.println(str instanceof String);  
  
        //oder  
        System.out.println(String.class.isInstance(str));  
    }  
}
```

Oberklasse finden

- ❖ `getSuperClass()`

```
public class ShowSuperClasses {  
    public static void main(String[] args) {  
        Class<?> subclass = javax.swing.JPanel.class;  
        Class<?> superclass = subclass.getSuperclass();  
  
        while (superclass != null)  
        {  
            String className = superclass.getName();  
            System.out.println(className);  
            subclass = superclass;  
            superclass = subclass.getSuperclass();  
        }  
    }  
}
```

|

javax.swing.JComponent
java.awt.Container
java.awt.Component
java.lang.Object

Interfaces

- ❖ `getInterfaces()`

```
public class ShowInterfaces {  
    public static void main(String[] args) {  
        for ( Class<?> theInterface: java.io.File.class.getInterfaces() )  
            System.out.println( theInterface.getName() );  
    }  
}
```

`java.io.Serializable`
`java.lang.Comparable`

Modifizierer

```
public class ShowModifizierer {  
    public static void main(String[] args) {  
        System.out.println( Date.class.getModifiers() ); // 1  
        System.out.println( Modifier.toString(Modifier.class.getModifiers()) ); // public  
    }  
}
```

Modifier `toString()`-Methode

```
public static String toString(int mod) {  
    StringBuilder sb = new StringBuilder();  
    int len;  
  
    if ((mod & PUBLIC) != 0)          sb.append("public ");  
    if ((mod & PROTECTED) != 0)       sb.append("protected ");  
    if ((mod & PRIVATE) != 0)         sb.append("private ");  
  
    /* Canonical order */  
    if ((mod & ABSTRACT) != 0)        sb.append("abstract ");  
    if ((mod & STATIC) != 0)          sb.append("static ");  
    if ((mod & FINAL) != 0)           sb.append("final ");  
    if ((mod & TRANSIENT) != 0)       sb.append("transient ");  
    if ((mod & VOLATILE) != 0)        sb.append("volatile ");  
    if ((mod & SYNCHRONIZED) != 0)    sb.append("synchronized ");  
    if ((mod & NATIVE) != 0)          sb.append("native ");  
    if ((mod & STRICT) != 0)          sb.append("strictfp ");  
    if ((mod & INTERFACE) != 0)       sb.append("interface ");  
  
    if ((len = sb.length()) > 0)      /* trim trailing space */  
        return sb.toString().substring(0, len-1);  
    return "";  
}
```

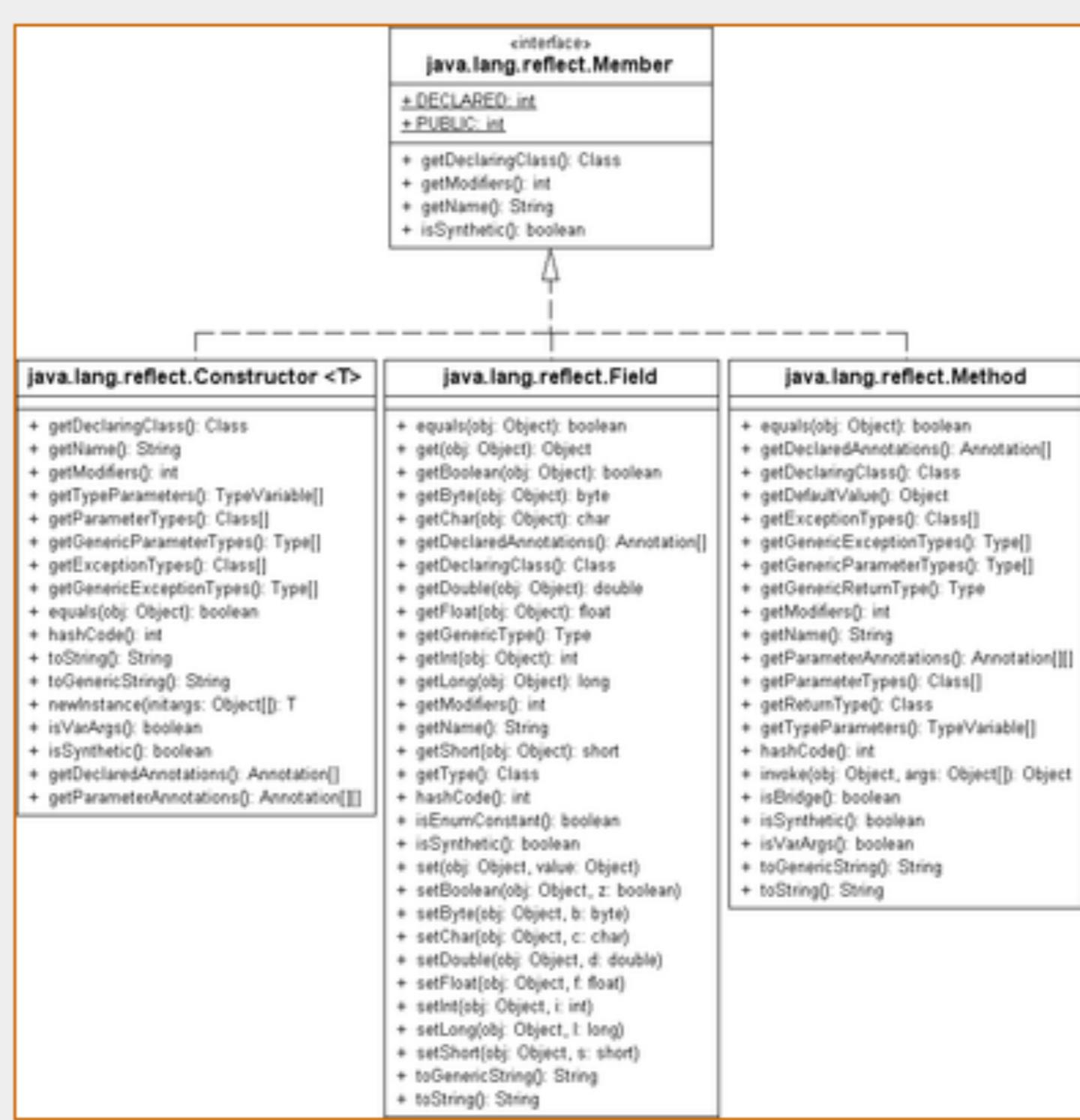
Class-Objekt bietet weiterhin

- ❖ `getConstructors()`
 - ❖ Feld von Konstruktoren
- ❖ `getFields()`
 - ❖ Feld von Objekt- und Klassenvariablen
- ❖ `getMethods()`
 - ❖ Feld von Methoden

...

- ❖ `getAnnotations()`
 - ❖ Feld von Annotationen
- ❖ `getPackage`
 - ❖ liefert ein Package-Objekt für die Klasse, die eine Versionsnummer beinhaltet, wenn diese im Manifest gesetzt wurde.

Klassenhierarchie



getConstructors()

```
public class ConstructorsDemo {  
    public static void main(String[] args) {  
        try {  
            Class cls = Class.forName("java.util.ArrayList");  
            System.out.println("ArrayList Constructors =");  
  
            /* returns the array of Constructor objects representing the public  
            constructors of this class */  
            Constructor c[] = cls.getConstructors();  
            for(int i = 0; i < c.length; i++) {  
                System.out.println(c[i]);  
            }  
        }  
        catch (Exception e) {  
            System.out.println("Exception: " + e);  
        }  
    }  
}
```

```
ArrayList Constructors =  
public java.util.ArrayList(java.util.Collection)  
public java.util.ArrayList()  
public java.util.ArrayList(int)
```

Objekte erzeugen

- ❖ new fällt zur Laufzeit flach
- ❖ passendes Class-Objekt wird benötigt
- ❖ getConstructor() und dann
- ❖ newInstance(Object[])
- ❖ IllegalAccessException (private),
IllegalArgumentException (falsche Anzahl an Parameter)
- ❖ InvocationTargetException als Wrapper um mögliche Exception

Beispiel

```
public class StudentTest2
{
    public static void main(String[] args)
    {
        try
        {
            Class cls = Class.forName("reflection.Student");
            Constructor constructor = cls.getConstructor(String.class);
            System.out.println("Constructor Name--->>>" + constructor.getName());
            constructor.setAccessible(true);
            System.out.println("Object creation--->>>" + constructor.newInstance( "Peter" ));
        }
        catch( Exception e )
        {
            e.printStackTrace();
        }
    }
}
```

Objekt erzeugen

```
public class Student
{
    private String name;
    private Student(String name) {
        System.out.println("Student erzeugt");
        this.name = name;
    }
}
```

```
public class StudentTest
{
    public static void main(String[] args)
    {
        try
        {
            Class cls = Class.forName("reflection.Student");
            Constructor[] constructors = cls.getDeclaredConstructors();
            System.out.println("Constructor Name--->>>" + constructors[0].getName());
            constructors[0].setAccessible(true);
            System.out.println("Object creation--->>>" + constructors[0].newInstance( "peter"));
        }
        catch( Exception e )
        {
            e.printStackTrace();
        }
    }
}
```

Übung

- ❖ UebReflection.listConstructors
- ❖ UebRefelection.useConstructors

getFields()

```
public class FieldsDemo {  
  
    public static void main(String[] args) {  
        Class<?> clazz = java.lang.String.class;  
        Field[] fields;  
  
        // list with all the accessible public fields of the class or interface  
        fields = clazz.getFields();  
        for (int i = 0; i < fields.length; i++) {  
            System.out.println("Found public field: " + fields[i]);  
        }  
  
        System.out.println();  
  
        // list with all the fields declared by this class or interface  
        fields = clazz.getDeclaredFields();  
        for (int i = 0; i < fields.length; i++) {  
            System.out.println("Found field: " + fields[i]);  
        }  
  
    }  
}
```

Found public field: public static final java.util.Comparator java.lang.String.CASE_INSENSITIVE_ORDER

Found field: private final char[] java.lang.String.value

Found field: private int java.lang.String.hash

Found field: private static final long java.lang.String.serialVersionUID

Found field: private static final java.io.ObjectStreamField[] java.lang.String.serialPersistentFields

Found field: public static final java.util.Comparator java.lang.String.CASE_INSENSITIVE_ORDER

auch die Inhalte sind interessant...

```
public class ClassFieldTest {  
  
    public static void main(String args[]) throws Exception {  
        Student st = new Student("Peter", 3);  
        Class stClass = st.getClass();  
  
        //wenn public dann  
        //Field f1 = stClass.getField("name");  
        //ansonsten  
        Field f1 = stClass.getDeclaredField("name");  
        f1.setAccessible(true);  
        f1.set(st, "reflecting on life");  
        String str1 = (String) f1.get(st);  
        System.out.println("name field: " + str1);  
  
        Field f2 = stClass.getDeclaredField("matNr");  
        f2.setAccessible(true);  
        f2.set(st, 5);  
        //Erfragt den Wert eines Attributs.  
        //Die Referenz von obj zeigt auf das Objekt, das das Attribut enthält.  
        //Es sollte null übergeben werden, wenn es sich um eine statische Variable handelt.  
        int i = f2.getInt(st);  
        System.out.println("name field: " + i);  
  
        try {  
            System.out.println("\nThis will throw a NoSuchFieldException");  
            Field f3 = stClass.getField("matNr");  
        } catch (Exception e) {  
            e.printStackTrace();  
        }  
    }  
}
```

Frameworks nutzen Reflection

- ❖ beispielsweise `toString` könnte generisch geschrieben werden
 - ❖ `ToStringHelper`
 - ❖ oder ein Debugger, der zur Laufzeit Werte ändert.
 - ❖ Bean properties kopieren
 - ❖ `org.apache.commons.beanutils.BeanUtils.copyProperties(Object dest, Object orig)`

Übung

- ❖ UebReflection.fields

getMethods

```
public class MethodDemo {  
    public static void main(String[] args) {  
        try {  
            Class cls = Class.forName("java.util.ArrayList");  
            System.out.println("Methods =");  
  
            /* returns the array of Method objects representing the public  
            methods of this class */  
            Method m[] = cls.getMethods();  
            for(int i = 0; i < m.length; i++) {  
                System.out.println(m[i]);  
            }  
        }  
        catch (Exception e) {  
            System.out.println("Exception: " + e);  
        }  
    }  
}
```

Methods =

```
public boolean java.util.ArrayList.add(java.lang.Object)  
public void java.util.ArrayList.add(int,java.lang.Object)  
public boolean java.util.ArrayList.remove(java.lang.Object)  
public java.lang.Object java.util.ArrayList.remove(int)  
public java.lang.Object java.util.ArrayList.get(int)  
public java.lang.Object java.util.ArrayList.clone()  
public int java.util.ArrayList.indexOf(java.lang.Object)  
public void java.util.ArrayList.clear()  
public boolean java.util.ArrayList.isEmpty()
```

Methoden aufrufen

- ❖ `Object invoke(Object obj, Object... args) throws
IllegalAccessException, IllegalArgumentException,
InvocationTargetException`
- ❖ Ruft eine Methode des Objekts `obj` mit den gegebenen Argumenten auf. Wie schon beim Konstruktor löst die Methode eine `InvocationTargetException` aus, wenn die aufzurufende Methode eine Exception auslöst.

Beispiel

```
public class ClassMethodTest {  
  
    public static void main(String args[]) throws Exception {  
  
        Student s = new Student("Thomas", 6);  
        Class tClass = s.getClass();  
  
        Method gs1Method = tClass.getMethod("rufeHallo", new Class[] {});  
        String str1 = (String) gs1Method.invoke(s, new Object[] {});  
        System.out.println("rufeHallo returned: " + str1);  
  
        Method ss1Method = tClass.getMethod("getMatNr", new Class[] {});  
        System.out.println("calling getMatNr");  
        int mNr = (int) ss1Method.invoke(s, new Object[] {});  
        System.out.println("getMatNr returned: " + mNr);  
  
        Method ss2Method = tClass.getMethod("setMatNr", new Class[] { int.class });  
        System.out.println("calling setSMatNr with 9");  
        ss2Method.invoke(s, new Object[] { 9 });  
  
    }  
}
```

Übung

- ❖ UebReflection.methods

Annotationen

- ❖ @Override, @SuppressWarnings als bekannte Annotationen
- ❖ @WebService, @Stateless, @Inject als weitere Beispiele

drei Arten von Annotationen...

- ❖ Markerannotation
 - ❖ keine Parameterübergabe, dient nur zur Markierung, z.B. @Override, @Deprecated
- ❖ Single Member Annotation
 - ❖ Annotation mit nur einem Parameter. Parameternname value(muss nicht angegeben werden), z.B. @Retention, @Target

drei Arten von Annotationen...

- ❖ Normal Annotation
- ❖ Eine Annotation mit möglicherweise mehreren Parametern. Den Parametern muss jeweils der Parametername vorangestellt werden. Alle Parameter, für die die Deklaration keinen Defaultwert vorsieht, müssen gesetzt werden, z.B.: @Resource

Erläuterung zu einigen Standard Annotationen

- ❖ `@Documented`
 - ❖ Steuert JavaDoc Dokumentationen.
- ❖ `@Inherited`
 - ❖ Steuert die Vererbbarkeit der Annotation.
- ❖ `@Retention`
 - ❖ steuert die Beibehaltung und Verfügbarkeit der Annotation-Informationen

Erläuterung zu einigen Standard Annotationen

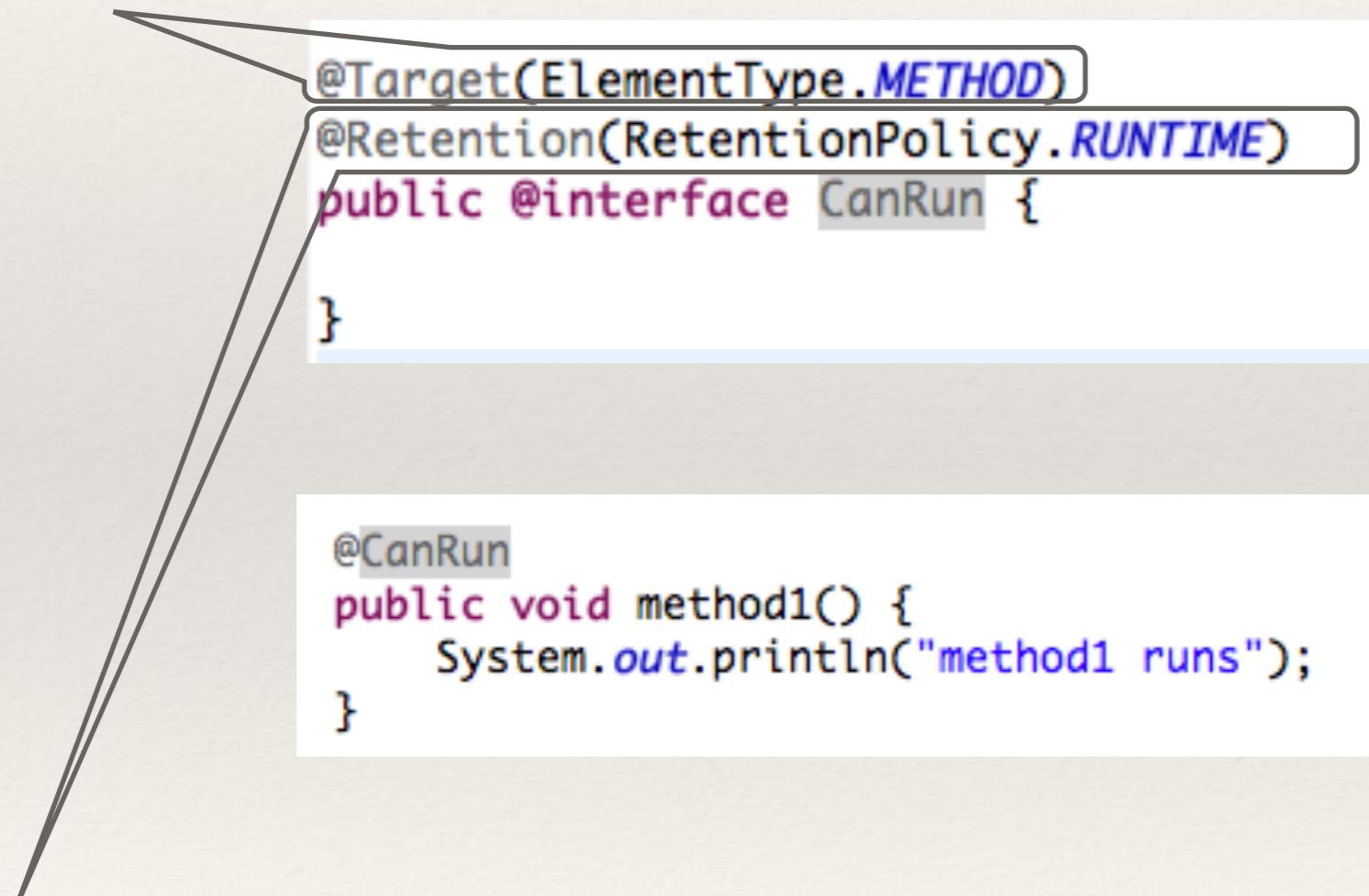
- ❖ @Target
 - ❖ steuert, welche Programmelemente anmontiert werden können.
- ❖ @Deprecated
 - ❖ markiert Programmelemente, die nicht mehr verwendet werden sollen.
- ❖ @Override
 - ❖ Markiert Methoden, die eine Methode der Superklasse überschreiben. Dies ist sinnvoll zur Dokumentation und damit sonst schwer zu findende Tippfehler vom Compiler gefunden werden.
- ❖ @SuppressWarnings
 - ❖ Unterdrückt einzelne oder mehrere Warnmeldungen.

Definition eigener Annotationen

- ❖ @CanRun -> Markerannotation
- ❖ @DefaultHallo -> Single member Annotation
- ❖ @IntegerSetter -> Normal Annotation

Neue Annotation deklarieren

wo kann ich meine Annotation verwenden



Wo ist die Annotation sichtbar?
Nur für den Compiler oder auch für die Laufzeitumgebung?

Annotieren von Annotationstypen

Annotation	Beschreibung
@Target	Was lässt sich annotieren? Klasse, Methode ...?
@Retention	Wo ist die Annotation sichtbar? Nur für den Compiler oder auch für die Laufzeitumgebung?
@Documented	Zeigt den Wunsch an, die Annotation in der Dokumentation zu erwähnen.
@Inherited	Macht deutlich, dass ein annotiertes Element auch in der Unterklasse annotiert ist.

Annotieren von Annotationstypen

ElementType	Erlaubt Annotationen ...
ANNOTATION_TYPE	an anderen Annotationstypen, was <code>@Target(ANNOTATION_TYPE)</code> somit zu einer Meta-Annotation macht.
TYPE	an allen Typdeklarationen, also Klassen, Schnittstellen, Aufzählungen.
CONSTRUCTOR	an Konstruktoren.
METHOD	an statischen und nicht-statischen Methoden.
FIELD	an statischen Variablen und Objekt-Variablen.
PARAMETER	an Parametervariablen.
LOCAL_VARIABLE	an lokalen Variablen.
PACKAGE	an package-Deklarationen.

Retention

- ❖ Die Annotation @Retention steuert, wer die Annotation sehen kann. Es gibt drei Typen, die in der Aufzählung `java.lang.annotation.RetentionPolicy` genannt sind:
 - ❖ SOURCE: Nützlich für Tools, die den Quellcode analysieren, aber die Annotationen werden vom Compiler verworfen, sodass sie nicht den Weg in den Bytecode finden.
 - ❖ CLASS: Die Annotationen speichert der Compiler in der Klassendatei, aber sie werden nicht in die Laufzeitumgebung gebracht.
 - ❖ RUNTIME: Die Annotationen werden in der Klassendatei gespeichert und sind zur Laufzeit in der JVM verfügbar.

Markerannotation

```
public class CanRunUsage {  
    public static void main(String[] args) {  
        CanRunUsage annotationUsage = new CanRunUsage();  
        annotationUsage.method1();  
        annotationUsage.method2();  
    }  
  
    @CanRun  
    public void method1() {  
        System.out.println("method1 runs");  
    }  
  
    public void method2() {  
        System.out.println("method2 runs");  
    }  
  
    @CanRun  
    public void method3() {  
        System.out.println("method3 runs");  
    }  
}
```

Markerannotation

```
public class CanRunTest {  
    public static void main(String[] args) {  
        CanRunUsage annotationUsage = new CanRunUsage();  
        Method[] methods = annotationUsage.getClass().getMethods();  
        for (Method method : methods) {  
            CanRun annos = method.getAnnotation(CanRun.class);  
            if(annos != null) {  
                try {  
                    method.invoke(annotationUsage);  
                } catch(Exception ex) {  
                    ex.printStackTrace();  
                }  
            }  
        }  
    }  
}
```

SingleMemberAnnotation

Attributtyp

```
@Target(ElementType.FIELD)
@Retention(RetentionPolicy.RUNTIME)
public @interface DefaultHallo {
    String value() default "Hallo";
}
```

SingleMemberAnnotation

```
public class DefaultHalloUsage {  
    private String str1;  
    @DefaultHallo  
    private String str2;  
    @DefaultHallo(value = "Test")  
    private String str3;  
    public String getStr1() {  
        return str1;  
    }  
    public void setStr1(String str1) {  
        this.str1 = str1;  
    }  
    public String getStr2() {  
        return str2;  
    }  
    public void setStr2(String str2) {  
        this.str2 = str2;  
    }  
    public String getStr3() {  
        return str3;  
    }  
    public void setStr3(String str3) {  
        this.str3 = str3;  
    }  
}
```

value = kann auch weggelassen werden

SingleMemberAnnotation

```
public class DefaultHalloResource {  
    public static <T> T getInitializedResourcesFor(Class<T> ressources) {  
        try {  
            T newInstance = ressources.newInstance();  
  
            //mit getFields nur auf public fields  
            for (Field field : ressources.getDeclaredFields()) {  
                field.setAccessible(true);  
                if (field.isAnnotationPresent(DefaultHallo.class)) {  
                    field.set(newInstance,  
                            field.getAnnotation(DefaultHallo.class).value());  
                }  
            }  
  
            return newInstance;  
        } catch (Exception e) {  
            e.printStackTrace();  
            return null;  
        }  
    }  
}
```

SingleMemberAnnotation

```
public class DefaultHalloExample {  
    public static void main( String[] args )  
    {  
        DefaultHalloUsage stringClass =  
            DefaultHalloResource.getInitializedResourcesFor( DefaultHalloUsage.class );  
        System.out.println( stringClass.getStr1() );  
        System.out.println( stringClass.getStr2() );  
        System.out.println( stringClass.getStr3() );  
    }  
}
```

null
Hallo
Test

Normal Annotation

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface IntegerSetter {
    int min();
    int max();
    int start() default 1;
}
```

```
public class Point {
    private int x, y;

    @IntegerSetter(min = 0, max = 5)
    public void setX(int x) {
        this.x = x;
    }

    @IntegerSetter(min = 0, max=20, start = 5)
    public void setY(int y) {
        this.y = y;
    }
}
```

Normal Annotation

```
public static void main(String[] args) {
    Point.list(Point.class);
}
```

```
public static void list(Class<Point> class1) {
    for(Method method : class1.getMethods()) {
        IntegerSetter value = method.getAnnotation(IntegerSetter.class);
        if(value != null) {
            System.out.printf("%s: min=%d, max=%d, start=%d\n",
                method.getName(), value.min(), value.max(), value.start());
        }
    }
}
```

Nachteile von Annotationen

- ❖ Die Annotationen sind stark mit dem Quellcode verbunden, können also auch nur dort geändert werden. Ist der Original-Quellcode nicht verfügbar, etwa weil der Auftraggeber ihn geschlossen hält, ist eine Änderung der Werte nahezu unmöglich.
- ❖ Wenn Annotationen allerdings nach der Übersetzung nicht mehr geändert werden können, stellt das bei externen Konfigurationsdateien kein Problem dar. Externe Konfigurationsdateien können ebenso den Vorteil bieten, dass die relevanten Informationen auf einen Blick erfassbar sind und sich mitunter nicht redundant auf unterschiedliche Java-Klassen verteilen.
- ❖ Klassen mit Annotationen sind invasiv und binden auch die Implementierungen an einen gewissen Typ, wie es Schnittstellen tun. Sind die Annotationstypen nicht im Klassenpfad, kommt es zu einem Compilerfehler.
- ❖ Bisher gibt es keine Vererbung von Annotationen: Ein Annotationstyp kann keinen anderen Annotationstyp erweitern.
- ❖ Die bei den Annotationen gesetzten Werte lassen sich zur Laufzeit erfragen, aber nicht modifizieren.
- ❖ Warum werden Annotationen mit @interface deklariert, einer Schreibweise, die in Java sonst völlig unbekannt ist?

Übung

- ❖ UebAnnotationen/Uebung1.txt