Reflection

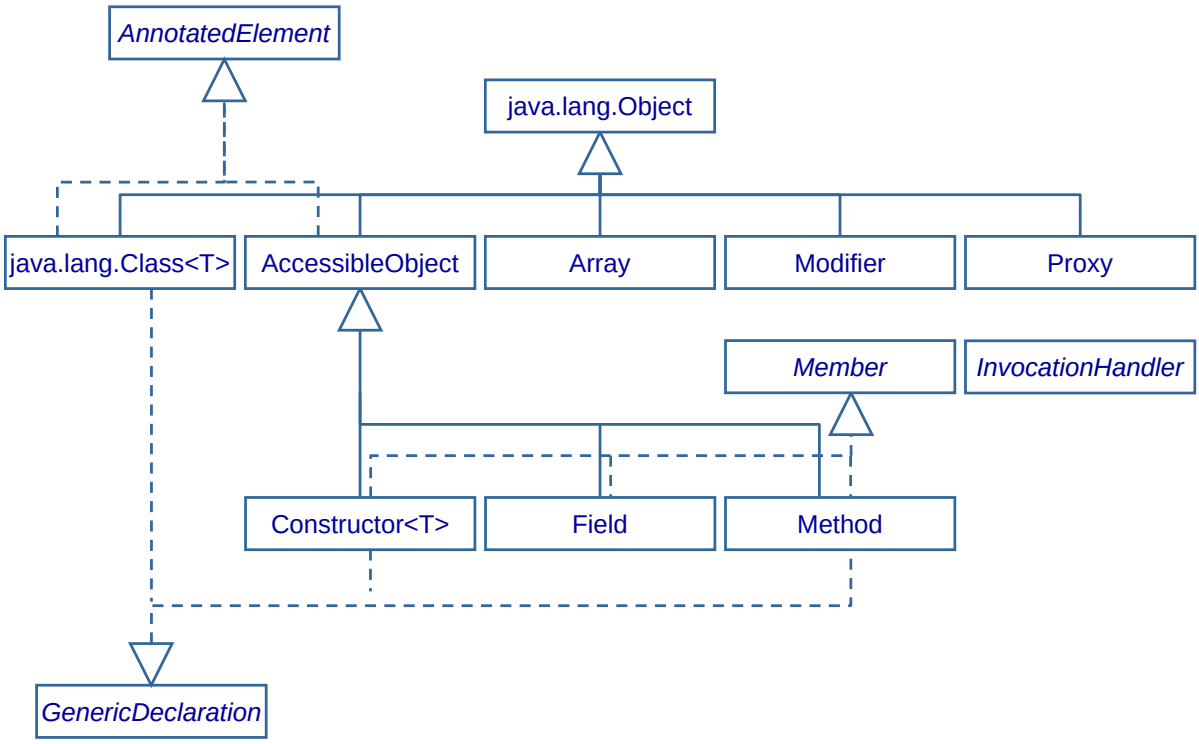
Reflection

- Laufzeittypinformationen
- Objekterzeugung und Methodenaufrufe über Reflection
- Dynamic Proxy
- Annotations

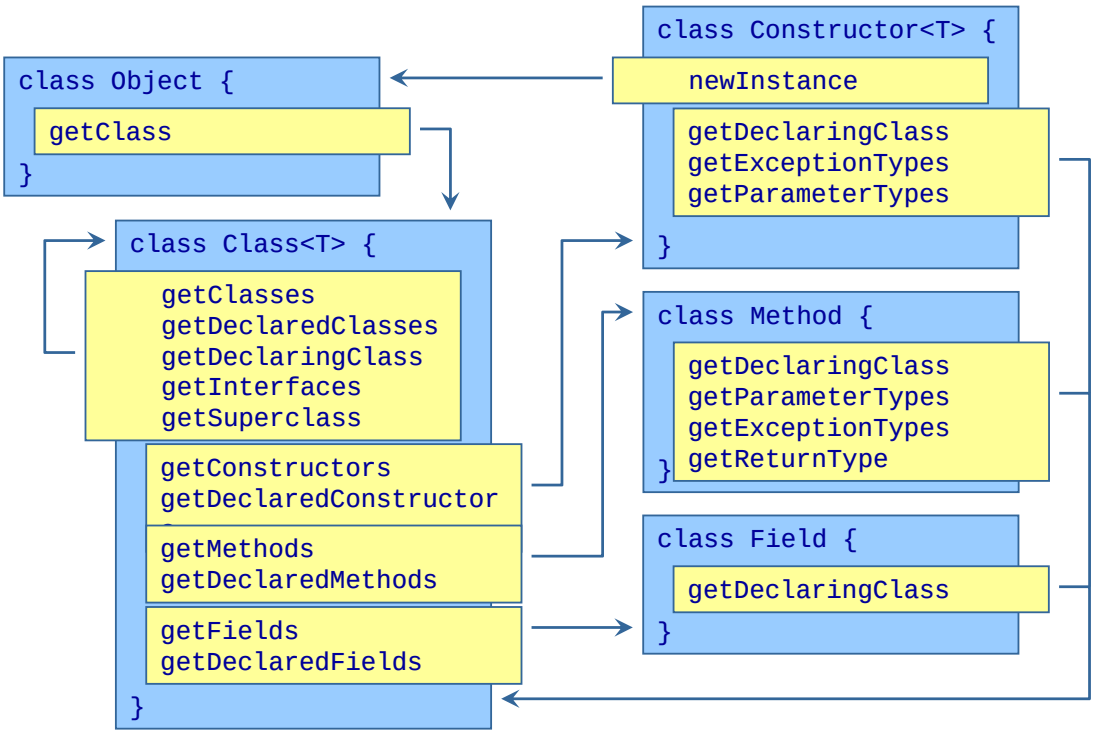
Reflection

- Programmatischer Zugriff auf Typinformationen
  - Über Klassen, Konstruktoren, Methoden und Felder
  - Zur Laufzeit
  - Zugriff erfolgt auf Basis des Byte-Codes
  - man kann auf alle Deklarationen zugreifen, nicht aber auf den ausführbaren Code
- Mechanismus
  - Untersuchen von Objekten
  - Erzeugen neuer Instanzen
  - Aufruf von Methoden
  - Lesen und Schreiben von Feldern
- Paket `java.lang.reflect`

Reflection Klassenhierarchie



Informationen über ein Objekt



Unterschied bei Zugriffen auf Members:

`get`*xxx*	… nur public, aber auch die geerbten
`getDeclared`*xxx*	… alle in dieser Klasse deklarierten

Klasse `Class<T>`

- JVM erzeugt für jede Klasse ein *eindeutiges* Objekt vom Typ `Class<T>`
- Typparameter `T` ist wiederum der Typ des Objekts, d.h.
  - `Class`-Objekt zur Klasse `Person` ist vom Typ `Class<Person>`
  - `Class`-Objekt zur Klasse `String` ist vom Typ `Class<String>`

```
Class<Person> personClass = Person.class;
```

- Zugriff auf `Class`-Objekt
  - mit `class` (siehe oben)
  - mit `obj.getClass()`

Beachte: Generischer Typ nicht bekannt

```
Class<?> personClass = person1.getClass();
```

aber Typcast möglich

```
Class<Person> personClass =
        (Class<Person>)person1.getClass();
```

  - mit `Class.forName` mit String des vollqualifizierten Namens

```
Class<?> stringClass =
        Class.forName("java.lang.String");
```

Vordefinierte Typen

- Für die primitiven Datentypen sind `Class`-Objekte als Konstante definiert
  - `byte` `Byte.TYPE`
  - `short` `Short.TYPE`
  - `int` `Integer.TYPE`
  - `long` `Long.TYPE`
  - `char` `Character.TYPE`
  - `boolean` `Boolean.TYPE`
  - `float` `Float.TYPE`
  - `double` `Double.TYPE`
  - `void` `Void.TYPE`

- Anwendung z.B.

```
if (method.getReturnType()) == Integer.TYPE) {
   int ret = ((Integer) method.invoke(obj, new Object[0]))).intValue();
}
```


Zugriff auf Members (1)

- Felder
  - Objekte der Klasse `java.lang.reflect.Field` repräsentieren Felder
  - Informationen über Sichtbarkeit und Typ
  - Lesen und Schreiben des Feldes für spezifiziertes Objekt

```
Field[] getFields()
Field getField(String name)
```

```
Person person1 = new Person();
Class<Person> personClass = Person.class;
try {
    Field nameField = personClass.getField("name");
    Field yearField = personClass.getField("year");
    System.out.println("Typ des Feldes " +
                       nameField.getName() + " ist " +
                       nameField.getType().toString());
    System.out.println("Typ des Feldes " +
                       yearField.getName() + " ist " +
                       yearField.getType().toString());
    nameField.set(person1, "Hans");
    yearField.setInt(person1, 1972);
    String name = (String)nameField.get(person1);
    int year = yearField.getInt(person1);
} catch (IllegalAccessException e) { …
} catch (SecurityException e) { …
} catch (NoSuchFieldException e) { …
}
```

Typ des Feldes name ist class java.lang.String

Typ des Feldes year ist int

Ausnahmen können auftreten !

Zugriff auf Members (2)

- Konstruktor
  - Objekte vom Typ `Constructor<T>` mit T ist Typ der Instanzen
  - Informationen über Modifizierer
  - erlaubt Erzeugen neuer Instanzen

```
Constructor<T>[] getConstructors()
Constructor<T> getConstructor(Class<T>... paramTypes)
```

```
Class<Person> personClass = Person.class;
try {
    Constructor<Person> personConst1 =
            personClass.getConstructor();
    person1 = personConst1.newInstance();

    Constructor<Person> personConst2 =
            personClass.getConstructor(String.class, Integer.TYPE);
    person2 = personConst2.newInstance("Hans", 1972);

} catch (SecurityException e1) { …
} catch (NoSuchMethodException e1) { …
} catch (IllegalArgumentException e) { …
} catch (InstantiationException e) { …
} catch (IllegalAccessException e) { …
} catch (InvocationTargetException e) { …}
```

```
public class Person {
  public Person() {
  }
  public Person(String name, int year) {
    this.name = name;
    this.year = year;
  }
```

Ausnahmen!

Zugriff auf Members (3)

- Methoden
  - Objekte vom Typ `Method`
  - Parametertypen, Ergebnistyp, Ausnahmetypen
  - Informationen über Modifizierer
  - Aufruf auf spezifiziertem Objekt (wenn erlaubt)

```
Method[] getMethods()
Method getMethod(String name, Class<T>... paramTypes)
```

```
try {
    Method setYearMethod =
        personClass.getMethod("setYear", Integer.TYPE);
    setYearMethod.invoke(person1, 1974);

    Method getAgeMethod = personClass.getMethod("getAge");
    System.out.println(getAgeMethod.invoke(person1));

} catch (SecurityException e1) {
} catch (NoSuchMethodException e1) {
} catch (IllegalArgumentException e) {
} catch (IllegalAccessException e) {
} catch (InvocationTargetException e) {
}
```

Beispiel: Ausgabe der Klassendefinition (1)

- Es werden alle Deklarationen (Klasse, Felder, Konstruktoren, Methoden, innere Klassen) ausgegeben

```
public static void printClass(Class clazz) {
```

// Beispielausgabe für java.util.LinkedList

- Ausgabe der Package-Deklaration

```
    // package
    if (clazz.getDeclaringClass() == null) { // no package for inner classes
        System.out.println("package " + clazz.getPackage().getName() + ";");
    }
```

package java.util;

- Ausgabe Class- oder Interface-Deklaration

```
    int modifiers = clazz.getModifiers();
    if (! clazz.isInterface()) {
        System.out.println(Modifier.toString(modifiers) + " class " +
            clazz.getName() + " extends " + clazz.getSuperclass().getName());
    } else {
        System.out.println(Modifier.toString(modifiers) + " interface " +
            clazz.getName());
    }
```

public class java.util.LinkedList extends java.util.AbstractSequentialList

Beispiel: Ausgabe der Klassendefinition (2)

- Ausgabe der implementierten oder erweiterten Interfaces

```
    StringBuffer line = new StringBuffer();
    Class<?>[] interfaces = clazz.getInterfaces();
    if (interfaces.length != 0) {
        StringBuffer line = new StringBuffer();
        if (! clazz.isInterface()) {
            line.append("    implements ");
        } else {
            line.append("    extends ");
        }
        for (int i = 0; i < interfaces.length; i++) {
            line.append(interfaces[i].getName());
            if (i < interfaces.length-1) {
                line.append(", ");
            }  else {
                line.append(" {");
            }
        }
        System.out.println(line);
    }
```

implements java.util.List, java.util.Deque, java.lang.Cloneable, java.io.Serializable {

Beispiel: Ausgabe der Klassendefinition (3)

- Ausgabe der statischen Felder

```
Field[] fields = clazz.getDeclaredFields();
for (i = 0; i < fields.length; i++) {
    if (Modifier.isStatic(fields[i].getModifiers())) {
        printField(fields[i]);
    }
}
```

```
private static final long serialVersionUID;
```

- und nicht-statischen Felder

```
for (i = 0; i < fields.length; i++) {
    if (! Modifier.isStatic(fields[i].getModifiers())) {
        printField(fields[i]);
    }
}
```

```
private transient java.util.LinkedList$Entry header;
private transient int size;
```

```
private static void printField(Field field) {
    System.out.println("  " + Modifier.toString(field.getModifiers()) +
        " " + field.getType().getName() + " " + field.getName() + ";");
}
```


Beispiel: Ausgabe der Klassendefinition (4)

- Ausgabe der Konstruktoren

```
Constructor<?>[] constructors = clazz.getDeclaredConstructors();
for (i = 0; i < constructors.length; i++) {
    printConstructor(constructors[i]);
}
```

```
public java.util.LinkedList();
public java.util.LinkedList(java.util.Collection);
```

```
private static void printConstructor(Constructor<?> c) {
    System.out.println("  " + Modifier.toString(c.getModifiers()) + " " +
        c.getName()+ constructParamsList(c.getParameterTypes()) + ";");
}
```

```
private static String constructParamsList(Class<?>[] paramTypes) {
    StringBuffer paramsString = new StringBuffer();
    paramsString.append("(");
    for (int i = 0; i < paramTypes.length; i++) {
        paramsString.append(paramTypes[i].getName());
        if (i < paramTypes.length-1) {
            paramsString.append(", ");
        }
    }
    paramsString.append(")");
    return paramsString.toString();
}
```

Beispiel: Ausgabe der Klassendefinition (5)

- Ausgabe der Methoden

```
Method[] methods = clazz.getDeclaredMethods();
for (i = 0; i < methods.length; i++) {
    print(methods[i]);
}
```

```
public java.lang.Object clone();
public int indexOf(java.lang.Object);
public int lastIndexOf(java.lang.Object);
…
```

```
private static void print(Method method) {
   System.out.println("  " + Modifier.toString(method.getModifiers())+" " +
         method.getReturnType().getName() +  " " + method.getName() +
         constructParamsList(method.getParameterTypes()) + ";");
}
```

- Ausgabe der inneren Klassen

```
Class<?>[] innerClasses = clazz.getDeclaredClasses();
for (i = 0; i < innerClasses.length; i++) {
    printClass(innerClasses[i]);
    System.out.println();
}
```


Reflection

- Laufzeittypinformationen
- Objekterzeugung und Methodenaufrufe über Reflection
- Dynamic Proxy
- Annotations

Beispiel: Methodenaufrufe über Reflection (1)

- Folgende Anweisungen ohne Verwendung von Reflection

  werden im Anschluss über Reflection ausgeführt

```
List list;
list = new LinkedList();
list.add("A");
list.add("C");
list.add(1, "B");

System.out.println(list.toString());

if (list.contains("A")) {
   System.out.println("A contained in list");
}
```

```
[A, B, C]
A contained in list
```


Beispiel: Methodenaufrufe über Reflection (2)

- Zugriff auf Klasse `LinkedList` und erzeugen des Objekts

```
try {
   List list;
//  list = new LinkedList();
   Class listClass = Class.forName("java.util.LinkedList");
   list = (List) listClass.newInstance();
```

- Zugriff auf Methode `add(Object)` und Aufrufe

```
//  list.add("A");
    Method addMethod =
        listClass.getMethod("add", Object.class);
    addMethod.invoke(list, "A");
//  list.add("C");
    addMethod.invoke(list, "C");
```

- Zugriff auf Methode `add(int, Object)` und Aufruf

```
//  list.add(1, "B");
    Method addMethod2 = listClass.getMethod("add",
                                Integer.TYPE, Object.class);
    addMethod2.invoke(list, 1, "B");
```

Beispiel: Methodenaufrufe über Reflection (3)

- Aufruf von `toString()`

```
// System.out.println(list.toString());
   Method toStringMethod =
      listClass.getMethod("toString");
   System.out.println(toStringMethod.invoke(list));
```

- Aufruf von `contains(Object)`

```
// if (list.contains("A")) {
//     System.out.println("A contained in list");
// }
   Method containsMethod =
    listClass.getMethod("contains", Object.class);
   Boolean aCont = (Boolean)containsMethod.invoke(list, "A");
   if (aCont.booleanValue()) {
      System.out.println("A contained in ll");
   }
 } catch (NoSuchMethodException e) {…
 } catch (IllegalArgumentException e) { …
 } catch (IllegalAccessException e) {…
 } catch (InvocationTargetException e) {…
 } catch (ClassNotFoundException e) {…
 } catch (InstantiationException e) {…
 } catch (SecurityException e) {…
 }
```


Reflection und Generizität

Beachte:

- Generische Klassen gibt es im Java-Byte-Code nicht !
- Generizität nur innerhalb des Java-Source-Codes und für Java-Compiler relevant
- Reflection kennt also keine generischen Typen (kennt nur die Rohtypen)
- Dadurch können sich sonderbare Phänomene ergeben

Beispiel ähnlich vorher

```
try {
   LinkedList<String> list = new LinkedList<String>();
   Class< LinkedList<String>> listClass=
                  (Class<LinkedList<String>>)list.getClass();  // ok
   list = listClass.newInstance(); // ok

   Method addMethod = listClass.getMethod("add", String.class); // Exception
   addMethod.invoke(list, "A");
    ...
} catch (NoSuchMethodException e) {
   ...
```

NoSuchMethodException, weil in Rowtype LinkedList keine Methode
`void add(String o)`
sondern nur Methode
`void add(Object o)`

Reflection

- Laufzeittypinformationen
- Objekterzeugung und Methodenaufrufe über Reflection
- Dynamic Proxy
- Annotations

Dynamic Proxy

- Platzhalterobjekt
  - Implementiert zur Laufzeit spezifizierte Liste von Interfaces
  - Methodenaufrufe werden an Invocation Handler weitergeleitet

```
public class Logger implements InvocationHandler {
  private PrintStream log;

  public Logger(PrintStream log) {
    this.log = log;
  }

  public Object invoke(Object proxy, Method method, Object[] args)
      throws Throwable {
    ...
  }
}
```

```
public interface Foo {
  public void something(int x);
}
```

```
Looger logger = new Logger(System.out);
Foo proxy = (Foo) Proxy.newProxyInstance(
    Foo.class.getClassLoader(), new Class[] { Foo.class }, logger);
proxy.something(333);
```

Reflection

- Laufzeittypinformationen
- Objekterzeugung und Methodenaufrufe über Reflection
- Dynamic Proxy
- Annotations

Annotations

- Mit 1.5 wurden in Java *Annotations* eingeführt
- Annotation von Programmelementen mit beliebiger Metainformation
- Annotationen
  - werden als `@interface` definiert
  - damit werden Programmelemente annotiert
  - können über Reflection API ausgelesen werden
- Annotations werden ähnlich wie Modifiers verwendet
- Nutzung von Annotations primär von Entwicklungswerkzeugen
- Package `java.lang.annotation`

Annotations-Typen

- Deklariert ähnlich einem Interface mit `@interface`
- mit Methoden für Eigenschaften wie folgt:
  - keine Parameter
  - keine throws-Klauseln
  - Rückgabewerte ausschließlich
    - primitive Datentypen (keine Wrapper)
    - String
    - Class
    - Enums
    - Annotationen
    - Arrays dieser Typen
  - keine Implementierung (wie bei Interfaces)
  - optional mit default-Klausel
- Annotation-Interfaces erweitern java.lang.annotation.Annotation

```
@interface SampleAnnotation {
   int intProp();
   String stringProp() default "default";
}
```


Beispiel: Annotation von Methoden mit Copyright

- Schreiben des Annotation-Typs mit Kodierung von Eigenschaften als Methoden

```
/**
 * Associates a copyright notice with the annotated API element.
 */
public @interface Copyright {
   int value();                    // Standardeigenschaft
   String owner() default "SSW"; // Standardwert
}
```

- Verwenden des Annotation-Typs
  - @-Zeichen plus Name des Annotation-Typs
  - In Klammern Angabe der Eigenschaftswerte (Name/Wert-Paare)
  - Werte müssen Compile-Time-Konstanten sein

```
public class UseCopyrightAnnotation {

   @Copyright(3)
   public void methodWithCopyright() {
      //...
   }
}
```

Annotation von Annotationen

- Annotation-Interfaces können selbst annotiert werden:
  - Welche Programmelemente annotiert werden können
    ➔ **@Target**
  - Wann die Annotation zur Verfügung stehen soll
    ➔ **@RetentionPolicy**
  - Ob diese in die JavaDoc aufgenommen werden sollen
    ➔ **@Documented**
  - Ob die Annotation vererbt werden soll
    ➔ **@Inherited**

Annotation-Typen für Annotationen (1)

@Target:

```
@Documented
@Retention(value=RUNTIME)
@Target(value=ANNOTATION_TYPE)
public @interface Target {
        public abstract ElementType[] value
}
```

```
public enum ElementType extends Enum<ElementType> {
        ANNOTATION_TYPE, TYPE, CONSTRUCTOR,
        METHOD, FIELD, LOCAL_VARIABLE, PACKAGE, PARAMETER
}
```

@Documented:

```
@Documented
@Retention(value=RUNTIME)
@Target(value=ANNOTATION_TYPE)
public @interface Documented {}
```

@Inherited:

```
@Documented
@Retention(value=RUNTIME)
@Target(value=ANNOTATION_TYPE)
public @interface Inherited {}
```

Annotation-Typen für Annotationen (2)

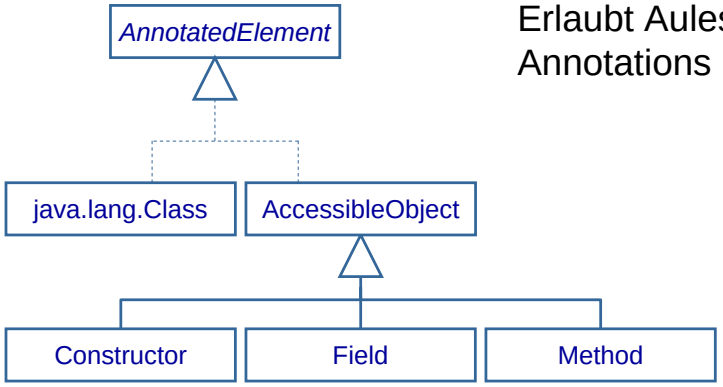
@RetentionPolicy:

```
@Documented
@Retention(value=RUNTIME)
@Target(value=ANNOTATION_TYPE)
public @interface Retention {
   public abstract RetentionPolicy value
}
```

```
public enum RetentionsPolicy extends Enum<ElementType> {
   CLASS, SOURCE, RUNTIME
}
```

- SOURCE: Compiler löscht die Annotation
- CLASS: in Byte-Code eingefügt aber nicht zur Laufzeit verfügbar
- RUNTIME: zur Laufzeit vorhanden

Interface AnnotatedElement



Erlaubt Aulesen der Annotations über Reflection API

```
interface AnnotatedElement {
    boolean isAnnotationPresent (Class<? extends Annotation> annotationType)
    <T extends Annotation> T getAnnotation(Class<T> annotationType)
    Annotation[] getAnnotations()
    Annotation[] getDeclaredAnnotations()
}
```

Beispiel: Annotation von Methoden mit Copyright (2)

- Schreiben des @interface-Interfaces mit RetentionPolicy und Target

```
/**
 * Associates a copyright notice with the annotated API element.
 */
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.TYPE, ElementType.METHOD})
public @interface Copyright {
    String owner() default "NN";
}
```

- Verwenden der Annotation

```
public class UseCopyrightAnnotation {

   @Copyright(owner = "2006 SSW")
   public void methodWithCopyright() {
      //...
   }

}
```


Beispiel: Annotation von Methoden mit Copyright (3)

- Auslesen der Annotation über Reflection-API

```
import java.lang.annotation.Annotation;
import java.lang.reflect.*;

public class ReadAnnotationInfo {

  public static void main(String[] args) {
    Class clazz = UseCopyrightAnnotation.class;
    try {
      Method method = clazz.getMethod("methodWithCopyright");
      Annotation annotation =
              method.getAnnotation(Copyright.class);
      Copyright copyright = (Copyright) annotation;
      System.out.println("Copyright: " + copyright.owner());
    } catch (SecurityException e) {
      e.printStackTrace();
    } catch (NoSuchMethodException e) {
      e.printStackTrace();
    }

  }
}
```

2 Spezialfälle bei Annotationen

Einzige Eigenschaft `value`:

- Gibt es nur eine Eigenschaftsmethode, so sollte diese `value` heißen
- Bei Verwendung kann dann der Name weggelassen werden

```
public @interface Copyright {
    String value() default "NN";
}
```

```
   @Copyright("2006 SSW")
   public void methodWithCopyright() { …
```

Annotation ohne Methoden:

- werden als *Marker* bezeichnet
- brauchen keine Klammern

```
public @interface Test {}
```

```
…
    @Test public static void testM2() {
        testObj.m2();
    }
```


Beispiel: Testwerkzeug (1)

- Annotation-Interface @Test

```
import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface Test {}
```

```
public class SystemUnderTestTest {

   static SystemUnderTest testObj = new SystemUnderTest();

    @Test public static void testM1() {
          testObj.m1();
    }
    @Test public static void testM2() {
          testObj.m2();
    }
    @Test public static void testM3() {
          testObj.m3();
    }
    @Test public static void testM4() {
          testObj.m4();
    }
}
```

```
public class SystemUnderTest {

    public void m1() { //...}
    public void m2() { //...}
    public void m3() {
        throw new RuntimeException("Boom");
    }
    public void m4() {
        throw new RuntimeException("Crash");
    }
}
```

Beispiel: Testwerkzeug (2)

- Ermitteln aller Testmethoden der Testklasse und Aufrufen der Tests

```
import java.lang.reflect.*;
public class TestTool {

   public static void main(String[] args) throws Exception {
      int passed = 0, failed = 0;
      for (Method m : Class.forName(args[0]).getMethods()) {
         if (m.isAnnotationPresent(Test.class)) {
            try {
               m.invoke(null);
               passed++;
            } catch (Throwable ex) {
               System.out.printf("Test %s failed: %s %n", m, ex.getCause());
               failed++;
            }
         }
      }
      System.out.printf("Passed: %d, Failed %d%n", passed, failed);
   }
}
```


Zusammenfassung

- Reflection
  - Programmatischer Zugriff auf Typinformationen zur Laufzeit
  - Erzeugen neuer Instanzen, Aufruf von Methoden, Zugriff auf Felder
- Dynamic Proxy
  - Platzhalterobjekt, das Liste von Interfaces implementiert
- Annotations
  - erlauben beliebige Zusatzinformationen bei Programmelementen
  - Können über Reflection ausgelesen werden