Woche 07

Generics und Polymorphie

Interface und Abstrackte Klasse

Generics

Bisher haben wir spezielle Klassen für jeden Datentyp neu implementiert:

```
1 class IntListElement {
2     private IntListElement next;
3     private int value;
4 }
```

```
1 class CharListElement {
2     private CharListElement next;
3     private char value;
4 }
```

Redundant, mühsam und fehleranfällig

Generics

Mit generics ermöglichen wir “variable” Datentypen:

```
1 public class List<T> {
2     private ListElement head;
3
4     private class ListElement {
5         private ListElement next;
6         private T value;
7     }
8 }
```

- List erwartet einen unbekannten “Typ-Parameter”
- T wird zur Kompilierzeit entschieden

Vorteile:

- Typ-Sicherheit
- Vermeidung von Type-Casts
- Code-Wiederverwendbarkeit

Anwendung von Generic:

- Generische Klassen:

```
public class A<T> {
    private T content;
    ⋮
}
```

* : var instanceof A<T> Error !!!
K extends T , aber : A<K> not extends A<T>

- Generische Interface:

```
public interface B<T>{
    public T methode1();
    ⋮
}
```

- Generische Methoden:

```
public <T> void method2(...){
    ⋮
}
```

* <T> vor Rückgabetyp : generische Methode

```
public T methode1(K k){
    ⋮
    return new T();
}
```

Rückgabetyp ist Generic:
nicht generische Methode

Generics

Generische Typen müssen Klassen/Interfaces sein, somit müssen für primitive Datentypen Wrapperklassen benutzt werden:

keine einfachen Datentypen.

```
1 List<Integer> intList = new List<Integer>();
2 List<Character> charList = new List<Character>();
3 List<Message> messageList = new List<Message>();
4
5 // kürzer:
6 List<Integer> intList = new List<>();
7
8 // unparametrisiert:
9 List objList = new List(); // == List<Object>
```

- konkrete Typ :

  entsprechende Einschränkung basierend auf dem Typ

- nicht konkrete Typ:

  beliebig

  mit T, K, ... deklaratieren.

Generics

Mehrere generische Typen:

```
1 public class Pair<T, U> {
2     private T first;
3     private U second;
4 }
```

Generics

Wir können auch festlegen, dass T gewisse Voraussetzungen erfüllen muss, d.h. von bestimmten Klassen erbt oder Interfaces implementiert.

```
1  public class ClassA<T extends ClassB> {
2      private T myT;
3
4      public ClassA(T myT) {
5          this.myT = myT;
6      }
7
8      public void doSomething() {
9          myT.methodFromB();
10     }
11 }
```

Wildcards.

? verwendet, um Flexibilität zu ermöglichen

	Typen	read	write
<?>	beliebig	yes	no (außer null)
<? extends T>	T, Subklassen	yes	no (außer null)
<? super T>	T, Superklassen	no	yes

Generics - Wildcards

Wildcards erlauben mehr Flexibilität beim schreiben einer Methode. Mit '?' kann der generische Typ eines Parameters spezifiziert werden.

```java
public double sum(List<? extends Number> producer) {
    double sum = 0;
    for (int i = 0; i < list.size(); i++) {
        Number n = producer.get(i);
        sum += n.doubleValue();
        // producer.add(n) funktioniert nicht!
    }
    return sum;
}

public void addIntegersToList(List<? super Integer> consumer) {
    int a = 10, b = 15;
    consumer.add(a);
    consumer.add(b);
    // Integer i = consumer.get(...) i.A. nicht möglich
}
```

PECS - Producer Extends, Consumer Super

Generics

<?> ist über unparametrisierten Parameter vorzuziehen, um Typsicherheit zu gewährleisten:

```
1  public void print(List<?> list) {
2      System.out.println(list.size());
3
4      // read mit get() ohne cast nur auf Object möglich
5      for (Object o : list) {
6          System.out.println(o);
7      }
8
9      // write mit mit list.add() nicht möglich
10 }
```

Generics

Für statische Methoden muss man den generischen Typ in der Methodensignatur definieren:

```
1 public static <T> List<T> arrayToList(T[] array) {
2     List<T> list = new List<>();
3     for (T t : array) {
4         list.add(t);
5     }
6     return list;
7 }
```

W07P01 - Generischer Stack

Bearbeite nun die Aufgabe W07P01 - Generischer Stack

W07P02 - TUMobile

Bearbeite nun die Aufgabe W07P02 - TUMobile

Abstrakte Klassen und Interfaces

Interfaces

Interfaces definieren eine Menge von Funktionalitäten, die eine Klasse bereitstellen muss.

```java
public interface VectorOperations<T> {
    // alle Methoden per default public
    void add(T other);
    void subtract(T other);
    double scalarProduct(T other);
}
```

Interfaces

Variablen bei Interfaces sind immer public, static und final

```java
public interface Distance<T>  {
  int MIN_DISTANCE = 0; // == public static final int MIN_DISTANCE = 0;

  double distanceTo(T other);

  static <T extends VectorOperations<T>> double getDistance(T a, T b) {
    T vec = a.subtract(b);
    return Math.sqrt(vec.scalarProduct(vec));
  }
}
```

zugreifen, nicht ändern

Interfaces

Klassen können mehrere Interfaces implementieren. Hierbei müssen alle Methoden aus allen Interfaces implementiert werden.

```java
public class Vector2 implements Distance<Vector2>,
                                VectorOperations<Vector2> {
    public Integer[] values = new Integer[2];

    @Override
    public Vector2 add(Vector2 other) { ... }

    @Override
    public Vector2 subtract(Vector2 other) { ... }

    @Override
    public double scalarProduct(Vector2 other) { ... }

    @Override
    public double distanceTo(Vector2 other) { ... }
}
```

Interfaces

Klassen können mehrere Interfaces implementieren, jedoch nur von einer Klasse erben.

```
1 public class ClassB extends ClassA implements Interface1, Interface2 {
2     ...
3 }
```

Somit müssen auch generische Typen mehrere Interfaces implementieren können. Achtung: “&”-Operator funktioniert nur wenn ein Interface rechts davon steht.

```
1 public class ClassB<T extends ClassB & Interface1 & Interface2> {
2     ...
3 }
```

Generische Interface :

```
public interface B<T>{
    public T methode1();
    ⋮
}
```

- ohne konrektem Typ:

```
classe Mammal<T> implements Animal <T>{
    ⋮
}
```

- mit konrektem Typ:

```
classe Mammal implements Animal<String>{
    ⋮
}
```

Abstrakte Klassen

Um eine Basis mit nicht nur Methoden (=> Interfaces), sondern auch Membervariablen erstellen zu können, gibt es abstrakte Klassen:

```java
public abstract class VectorBase {
    protected Integer[] values;
    private int dimension;

    public VectorBase(Integer[] values) {
        this.values = values;
        dimension = values.length;
    }

    public abstract void normalized();

    public int getDimension () {
        return dimension;
    }
}
```

abstrakte Methode: ohne Methodenköper keyword : abstract.

Abstrakte Klassen können, wie Interfaces, nicht instanziiert werden, können jedoch sehr wohl Konstruktoren besitzen!

Abstrakte Klassen

Eine konkrete Implementierung von Vector Base sieht dann so aus:

```
1  public class Vector2 extends VectorBase implements Distance<Vector2>,
2                                                      VectorOperations<Vector2>
3      public Vector2() {
4          super(new Integer[2]);
5      }
6      @Override
7      public void normalized() { ... }
8
9      @Override
10     public void add(Vector2 other) { ... }
11     @Override
12     public void subtract(Vector2 other) { ... }
13     @Override
14     public double scalarProduct() { ... }
15
16     @Override
17     public double distanceTo(Vector2 other) { ... }
18 }
```

Polymorphie

- Methodensignatur
  - Name der Methode und Parameterliste (ohne Name vom Parameter)
  - VERBOTEN: Eine Klasse mit extra derselben Signatur

z.B public void eat (Meat beef){...}

signatur: eat(Meat).

---

public void eat (Meat beef) {...}

public void eat (Meat chicken) {...} Compile Error !!!

- Override (Überschreiben) @Override
  - Methode in abgeleiteten Klasse neu definiert wird.
  - Voraussetzung: derselben Methodensignatur & Rückgabetyp
    überschreibene Methode in Supermethod nicht private.

z.B:
```
classe Animal {
    void makeSound(){ System.out.println(" make a sound."); }
}
classe Dog extends Animal{
  @Override
   void makeSound() { System.out.println(" Wuf Wuf"); }
}
```

- Overload (Überladen) @Overload
  - mehrere Methoden mit demselben Namen, aber unterschiedlichen Parameterlisten.
  - Rückgabetyp von überladener Methode verändert werden kann.

z.B
```
classe Animal {
    void eat() { System.out.println(" yammy");
    String eat() { return new String("yammy"); }
    void eat(Meat beef) { System.out.println( beef.toString + " is yammy"); }
}
```

- Statischer Typ
  - zur Compile-Zeit festlegen
  - auf Deklaration der Variablen basieren.
  - in "LINK" Seite

z.B: `Animal animal = new Dog();`
↑ statischer Typ

- Dynamischer Typ
  - zur Laufzeit festlegen
  - auf Klasse des tätsachlichen Objekts basieren.
  - bestimmte: welsche Methode tatsächlich ausgefürt wird. insbesondere Override.
  - in "RECHT" Seit

z.B: `Animal animal = new Dog();`
↑ dynamischer Typ.

* dynamischer Typ instanceof statischer Typ => true

- Dynamische Dispatch
  - bestimmt: zur Laufzeit, welsche Methode aufgerufen wird.
  - Aktionen:
    - Compiler überprüft die Signatur anhand des statischen Typs.
    - Zur Laufzeit wird die Methode des dynamisches Typs un Objekt aufgerufen

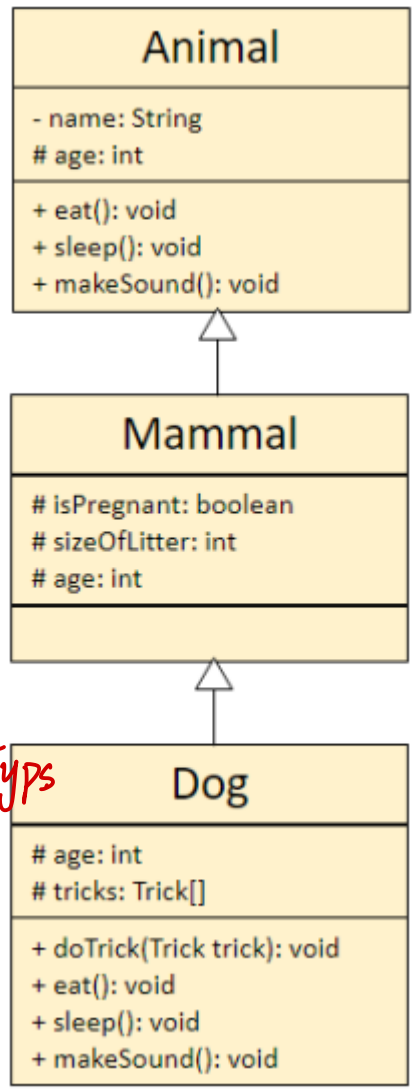
z.B
```
Animal animal = new Dog();
    animal.makeSound();
}
```
Output: Wof Wof

Polymorphie

```java
class Animal {}
class Mammal extends Animal {}
class Dog extends Mammal {}

public class Main {
    public static void main(String[] args) {
        Dog dog = new Dog();
        Animal dogAnimal = dog;
        Animal otherAnimal = new Animal();
        System.out.println(dog instanceof Dog);             // true
        System.out.println(dog instanceof Animal);          // true
        System.out.println(dogAnimal instanceof Dog);       // true
        System.out.println(dogAnimal instanceof Animal);    // true
        System.out.println(otherAnimal instanceof Animal);  // true
        System.out.println(otherAnimal instanceof Dog);     // false
    }
}
```

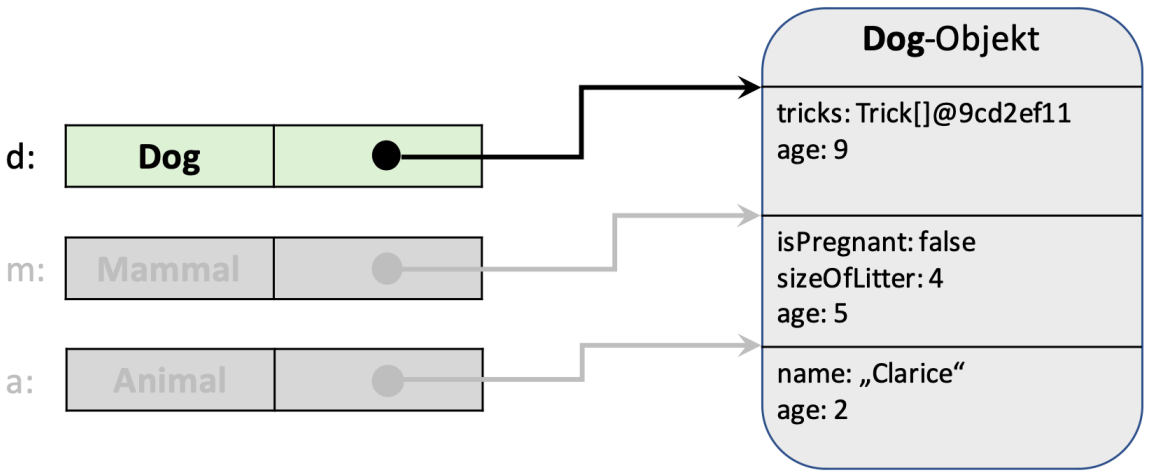
Polymorphie

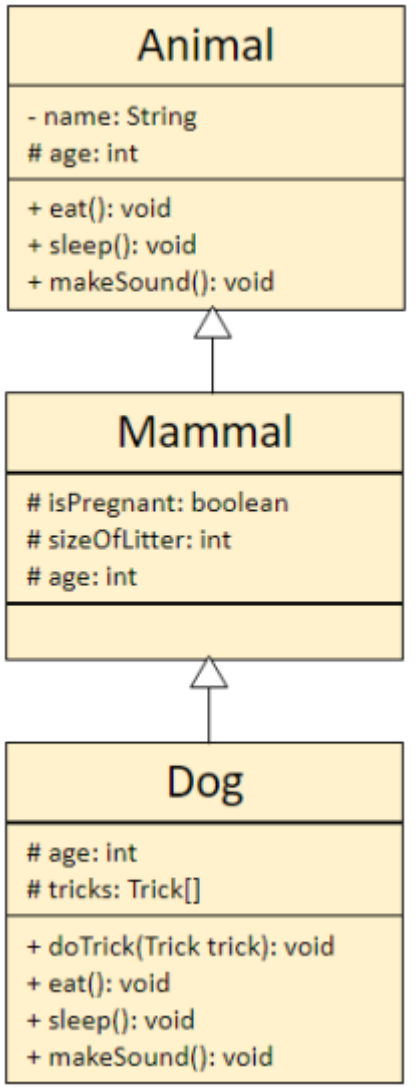
```java
// statischer Typ: Animal, dynamischer Typ: Dog
Animal animal = new Dog();

// -----

Dog dog = new Dog();
Mammal mammal = dog;
Animal animal = dog;

dog.doTrick(new Trick());
System.out.println(mammal.sizeOfLitter);
animal.makeSound();

// compile-fehler, Methode nicht verfügbar
animal.doTrick(new Trick()); überprüft anhand des statisches Typs
```



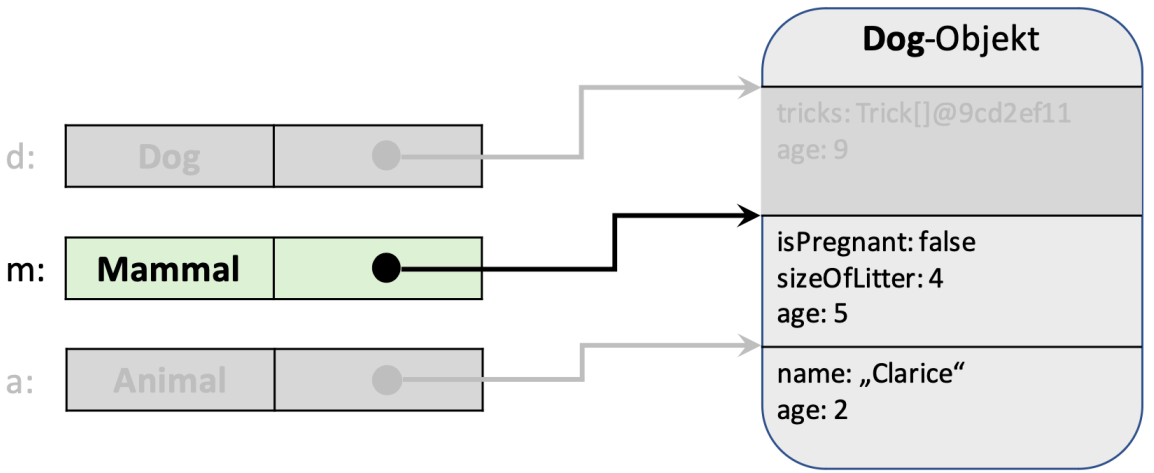
Polymorphie

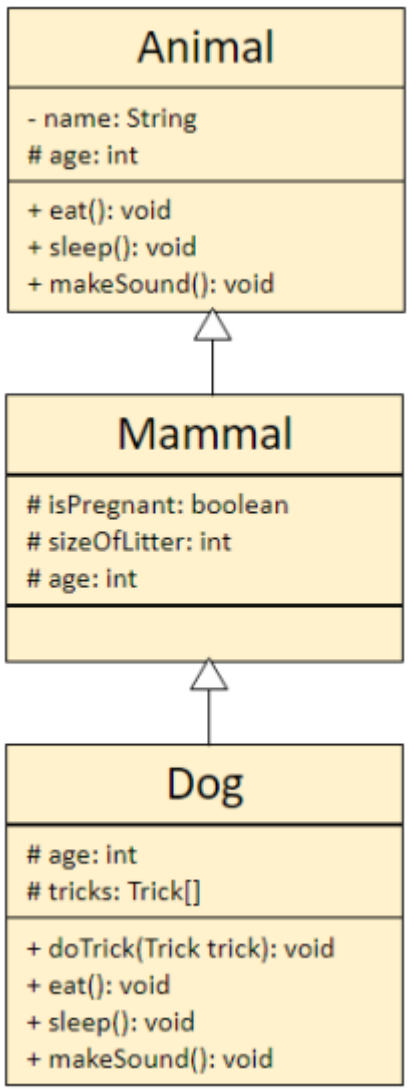
. Aufrufe von Variable: anhand des statischen Typs.



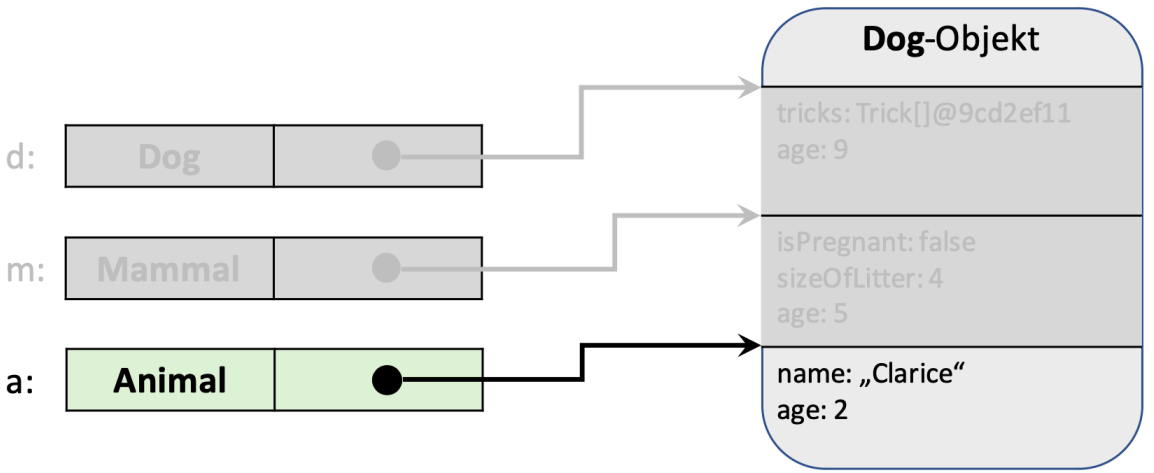


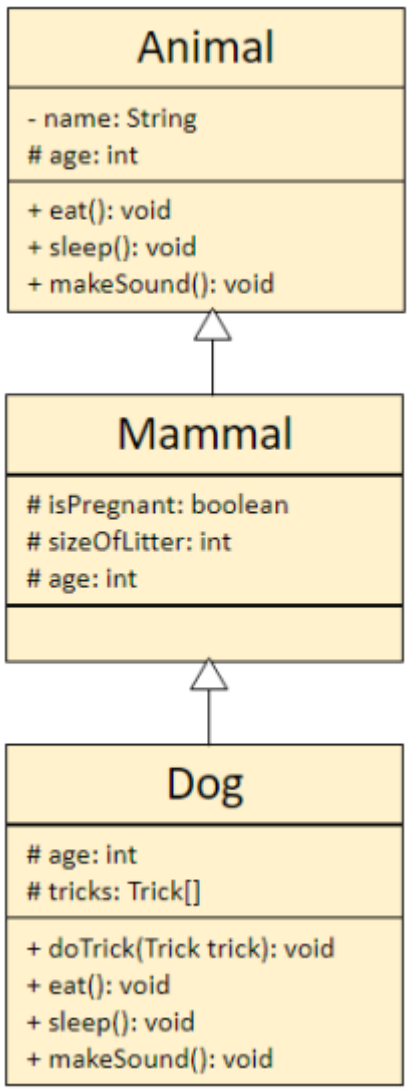
Polymorphie





Polymorphie





Polymorphie

```
1  // Mammal extends Animal
2  // Cat extends Mammal
3  // Dog extends Mammal
4  // Cow extends Mammal
5  Animal cat = new Cat();
6  Animal dog = new Dog();
7  Animal cow = new Cow();
8
9  cat.makeSound();
10 dog.makeSound();
11 cow.makeSound();
```

Angenommen makeSound() ist in Animal, Mammal und allen drei Unterklassen definiert. Welche wird tatsächlich aufgerufen?

Polymorphie

Method *overloading*: Gleiche Methodennamen, verschiedene Parametertypen - Auswahl der Methode hängt vom statischen Typ des Parameters ab:

```java
public class Main {
    public static void main(String[] args) {
        Dog dog = new Dog();
        Cat cat = new Cat();
        Animal dogAnimal = dog;

        printAnimal(dog);           // Dog
        printAnimal(cat);           // Animal
        printAnimal(dogAnimal);     // Animal
    }
    static void printAnimal(Animal animal) {
        System.out.println("Animal");
    }
    static void printAnimal(Dog dog) {
        System.out.println("Dog");
    }
}
```

Polymorphie

Casts und instanceof:

```
1 if (animal instanceof Dog) {
2     Dog animalAsDog = (Dog) animal;
3     animalAsDog.doTrick(new Trick());
4 }
```

instanceof prüft den dynamischen Typ, ohne diesen Check wirft der Cast möglicherweise eine Exception.

Polymorphie

Regeln für Casts: statischer Typ wird castet.

1. Upcast (von Subtyp zu Supertyp) immer implizit möglich

```
1 Dog d = new Dog;
2 Animal a = d;
```

2. Downcast (von Supertyp zu Subtyp) nur explizit möglich

```
1 Animal a = new Dog;
2 Dog d = (Dog) a;
```

3. Casts zwischen nicht verwandten Typen kompilieren (auch mit Cast) nicht

```
1 Cat c = new Dog(); // Fehler
2 Cat c = (Cat) new Dog(); // Fehler
```

4. Casts, die gegen die Hierarchie verstoßen, sind nicht möglich Beispiele hierfür sind.

```
Animal a = new Animal();
(Dog) a; // Fehler
```

W07P03 - Polymorphie

Bearbeite nun die Aufgabe W07P03 - Polymorphie

- Tipps:
  - Aufrufe von statischer Methode: anhand des statischen Typs
  - statischen Typ, kompatible Methoden, statischen Methoden, Signatur bestimmen: anhand nur des statischen Typs von Object oder/und Parameter.
    - kompatible Methoden:

      Methoden von Superklassen des Objekts.

      Methoden, statischer Typ des Parameters Superklasse des Parameter.

Auf-ruf	Zei-le	stat. Typ	kompati-ble Methoden	statisch gewählte Methode	Si-gna-tur	Begründung	dyn. Typ	zur Lauf-zeit aus-geführt	Begründung	Ausgabe
1	62	B	41, 14, 36, 9	41	f(B)	Speziellste Signatur, niedrigste Unterklasse	B	41	static, kein Dispatch	9-
	43	B	36, 9	36	f(A)	Speziellste Signatur	B	36	static, kein Dispatch	3-
	37	A	19	19	g(A)	Eindeutig	A	19	stat. Typ = dyn. Typ	3
1										9-3-3
2	63	A	19, 23	23	g(B)	Speziellste Signatur	A	23	stat. Typ = dyn. Typ	9-
	25	A	10	10	f(A)	Eindeutig	A	10	static, kein Dispatch	0-
	11	A	19	19	g(A)	Eindeutig	A	19	stat. Typ = dyn. Typ	3
2										9-0-3
3	64	A	9, 14	14	f(B)	Speziellste Signatur	B	14	static, kein Dispatch	9-
	16	B	51, 46, 23, 19	51	g(B)	Speziellste Signatur, niedrigste Unterklasse	B	51	stat. Typ = dyn. Typ	1-
	53	B	41, 14, 36, 9	41	f(B)	Speziellste Signatur, niedrigste Unterklasse	B	41	static, kein Dispatch	9-
	43	B	36, 9	36	f(A)	Speziellste Signatur	B	36	static, kein Dispatch	3-

	37	A	19	19	g(A)	Eindeutig	A	19	stat. Typ = dyn. Typ	3
3										9-1-9-3-3
4	65	A	19	19	g(A)	Eindeutig	B	46	dynamisch ein B	3-
	48	A	9, 14	14	f(B)	Speziellste Signatur	A	14	static, kein Dispatch	Exception
4								ClassCastException: A cannot be cast to B		
5	66	A	9	9	f(A)	Eindeutig	B	9	static, kein Dispatch	1-
	11	A	19	19	g(A)	Eindeutig	B	46	dynamisch ein B	3-
	48	A	9, 14	14	f(B)	Speziellste Signatur	B	14	static, kein Dispatch	9-
	16	B	51, 46, 23, 19	51	g(B)	Speziellste Signatur, niedrigste Unterklasse	B	51	stat. Typ = dyn. Typ	1-
	53	B	41, 14, 36, 9	41	f(B)	Speziellste Signatur, niedrigste Unterklasse	B	41	static, kein Dispatch	9-
	43	B	36, 9	36	f(A)	Speziellste Signatur	B	36	static, kein Dispatch	3-
	37	A	19	19	g(A)	Eindeutig	A	19	stat. Typ = dyn. Typ	3
5										1-3-9-1-9-3-3