Technische Hochschule Rosenheim
Technical University of Applied Sciences

Object-oriented programming
Chapter 11 – Abstract base classes

Prof. Dr Kai Höfig

Contents

Technische Hochschule Rosenheim
Technical University of Applied Sciences

- Abstract base classes
- Interfaces
- Abstract classes vs. interfaces
- ***Deadly Diamond of Death***: multiple inheritance from Java 8 onwards

Abstract classes – what for?

Technische Hochschule Rosenheim
Technical University of Applied Sciences

- Abstract classes are an important technical concept of object-oriented software development
- They are used to define the interfaces of future subclasses (for example, if we cannot or do not want to specify an implementation yet)
- They are a kind of pattern (template method pattern) that specifies which methods must be implemented in subclasses
- Only the subclasses know exactly how the methods should behave
- An abstract class works like a slot into which objects of the subclasses are inserted
- Software that provides such slots = framework
- Often the topmost or root element of an inheritance hierarchy

Abstract classes (1)

Technische Hochschule Rosenheim
Technical University of Applied Sciences

- Definition:
  - Class that **cannot be instantiated**

- Two different types are possible:
  (1) All operations are fully implemented – as with specific classes
  (2) At least one operation is not implemented (**abstract operation**)
    - Only defines the method signature – the method body is empty
    - Only specifies the interface
    - Derived classes must implement all abstract operations of the base (top-level) class

Abstract classes (2)

Technische Hochschule Rosenheim
Technical University of Applied Sciences

- Notation in UML
  - Keyword `<<abstract>>`
  - Italic font
  - Better with keyword for handwritten representation!

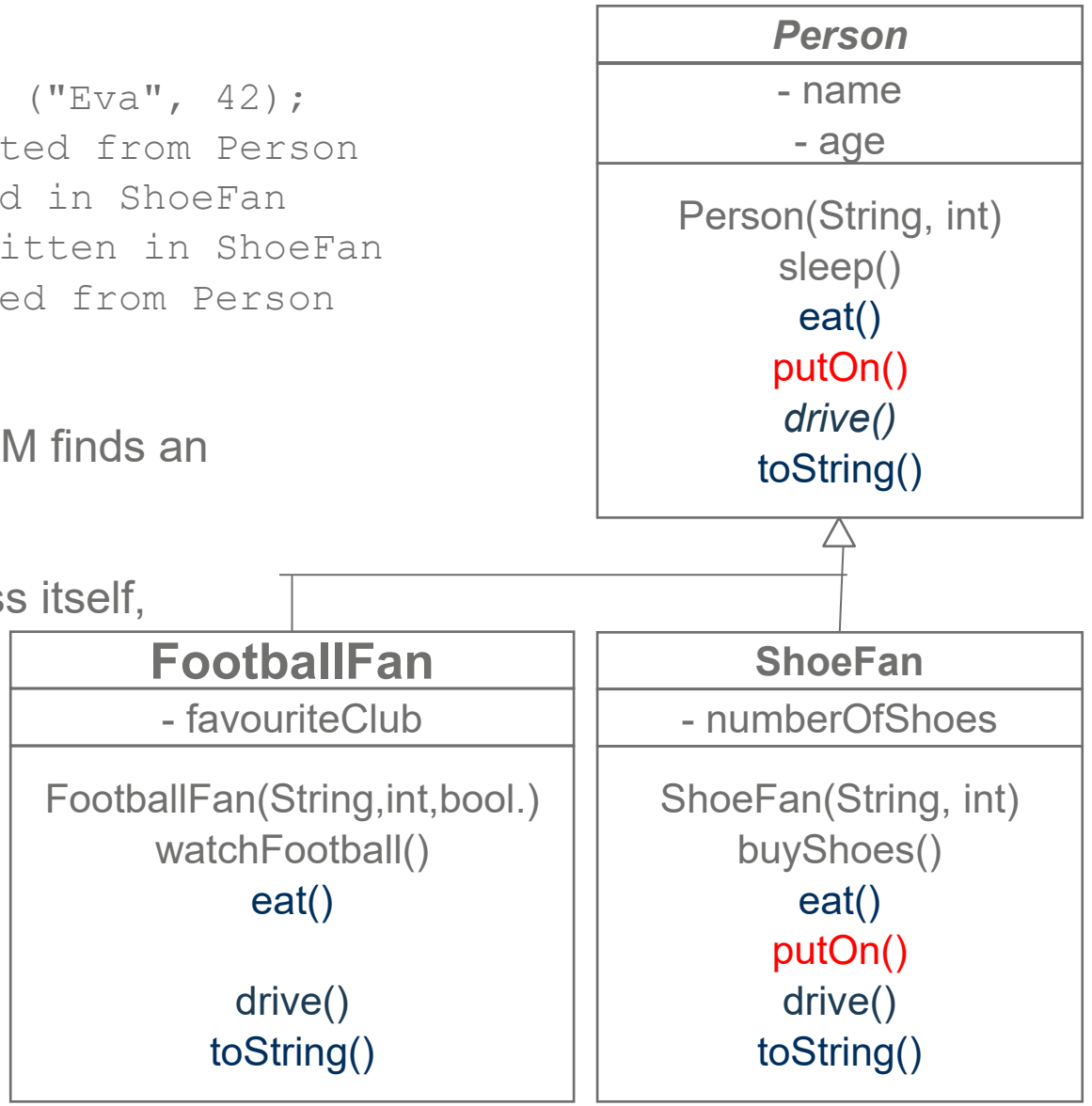
<<abstract>> **Person**
drive() {abstract}

Person
drive()

Calling inherited methods

Technische Hochschule Rosenheim
Technical University of Applied Sciences

- Example:
  - `ShoeFan eva = new ShoeFan ("Eva", 42);`
  - `eva.sleep();  // inherited from Person`
  - `eva.buyShoes();// defined in ShoeFan`
  - `eva.putOn();  // overwritten in ShoeFan`
  - `eva.drive();  // realised from Person`
- Processing:
  - Compiler ensures that the JVM finds an implementation at runtime
  - JVM first searches in the class itself, then in the base class, then in its base class, and so on.



Modifying the subclasses

Technische Hochschule Rosenheim
Technical University of Applied Sciences

- **Extend**
- Add something completely new
- ***Overload*** an existing method
- Subclass extends the top-level base class with additional attributes, operations and/or relationships

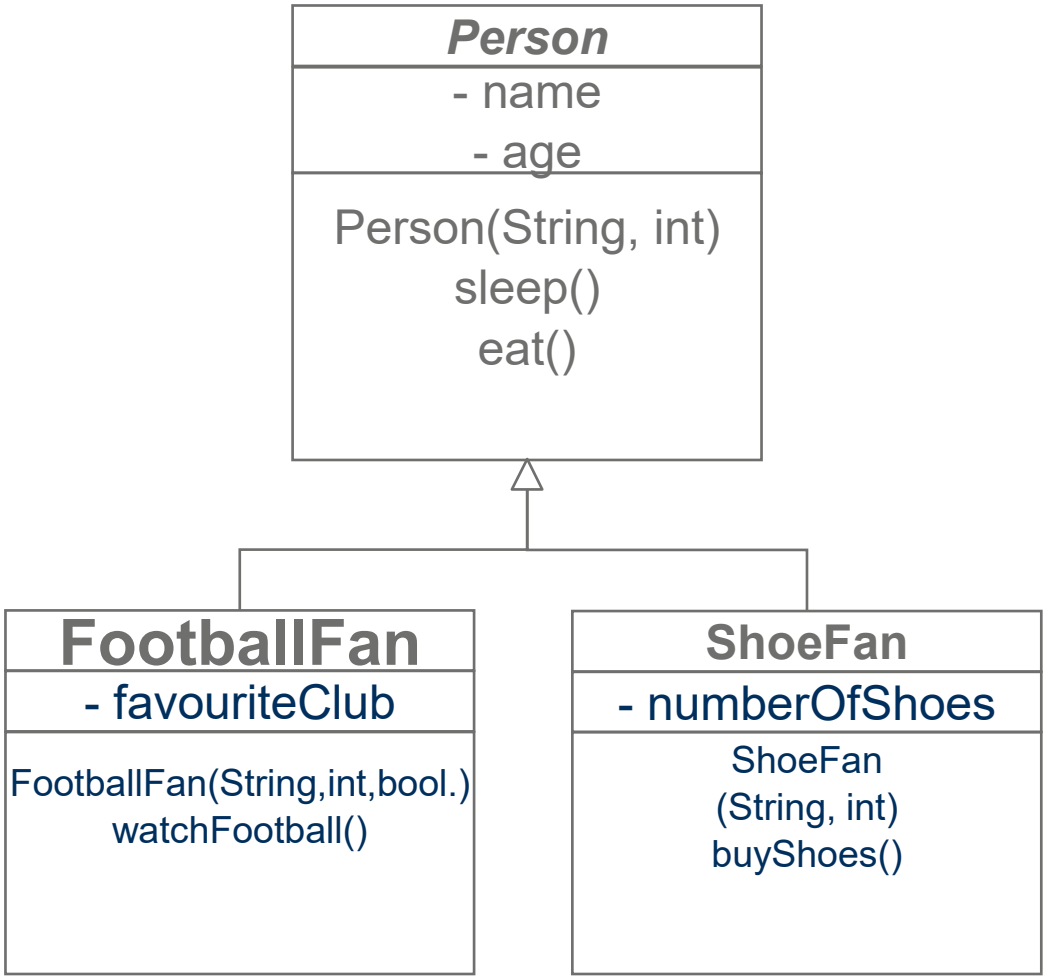
- **Redefine**
- Behave in a similar way
- If required, ***overwrite*** methods inherited in the subclass from the top-level base class with own specific implementation
- If necessary, use inherited implementations

- **Define**
- ***Realise*** something promised
- Implement abstractly declared operations of the top-level base class in the subclass

Extend properties in subclass

Technische Hochschule Rosenheim
Technical University of Applied Sciences

- **Starting point**
  - Base class Person
  - Common attributes
  - Basic common methods
  - Constructor
- **Extension in subclasses**
  - Specific attributes
  - Specific methods
  - In particular, own constructors



Overloading methods

Technische Hochschule Rosenheim
Technical University of Applied Sciences

- When **overloading** a method, we consider methods of the same name within a single class

- A method is specified by three properties:
  - Method name
  - Parameter list
  - Return value

- If we overload a method, it means that we have at least two methods of the same name within one class.

- **A differentiation using the return value does not work.**
- Therefore, we only have the parameter list for differentiation

Overloading methods: example

Technische Hochschule Rosenheim
Technical University of Applied Sciences

```
public static void foo(int x,double y,double z){
        System.out.println("foo 1");
}

public static void foo(int x,double y){
        System.out.println("foo 2");
}

public static void foo(double x,int y){
        System.out.println("foo 3");
}
```

- What happens now?

```
> foo(6,7.0,6.7);
> foo(3.1,5);
> foo(5,3.1);
> foo(5.0,5.0);
```

Overwriting methods

Technische Hochschule Rosenheim
Technical University of Applied Sciences

- **Overwriting** methods involves methods of the same name that are distributed among different classes within an inheritance hierarchy
- As with **overloading**, when **overwriting** methods the parameter list is also the decisive criterion for the unique assignment of a call to a method.

```
class Circle {
   ...
 void draw() {
  for( .... )
        drawPoint( x1, y1 );
   }
}

class ColouredCircle extends
Circle{
    ...
    @Override
  void draw() {
   for( .... )
    drawColourPoint(x1,y1,c);
    }
}
```

Overwriting vs. overloading

Technische Hochschule Rosenheim
Technical University of Applied Sciences

- **Overloading**: two methods have the same name, but different parameters. Calling is differentiated at compile time.
- **Overwriting**: two methods with the same name and parameters, but one of them is in the base class and one in the derived class. Only ***at runtime*** is the type of the object checked and the correct method selected.
- The binding of overloaded methods is **static binding**. The binding of overwritten methods is **dynamic binding** (or late binding).

Shadowing (1)

Technische Hochschule Rosenheim
Technical University of Applied Sciences

- Already known from "normal" classes
  - Local variables and/or parameter names shadow attributes
- New type of shadowing with inheritance
  - Subclass attribute shadows base class attribute
  - Subclass method shadows base class method
- Access
  - To shadowed element `x` of the base class: `super.x`
  - To shadowed element `x` of the current class: `this.x`

Shadowing (2)

Technische Hochschule Rosenheim
Technical University of Applied Sciences

- Base class

```
public abstract class Person {
   …
 public String eat() {
 return "eat  : Mmmmh, delicious.\n";
 }
}
```

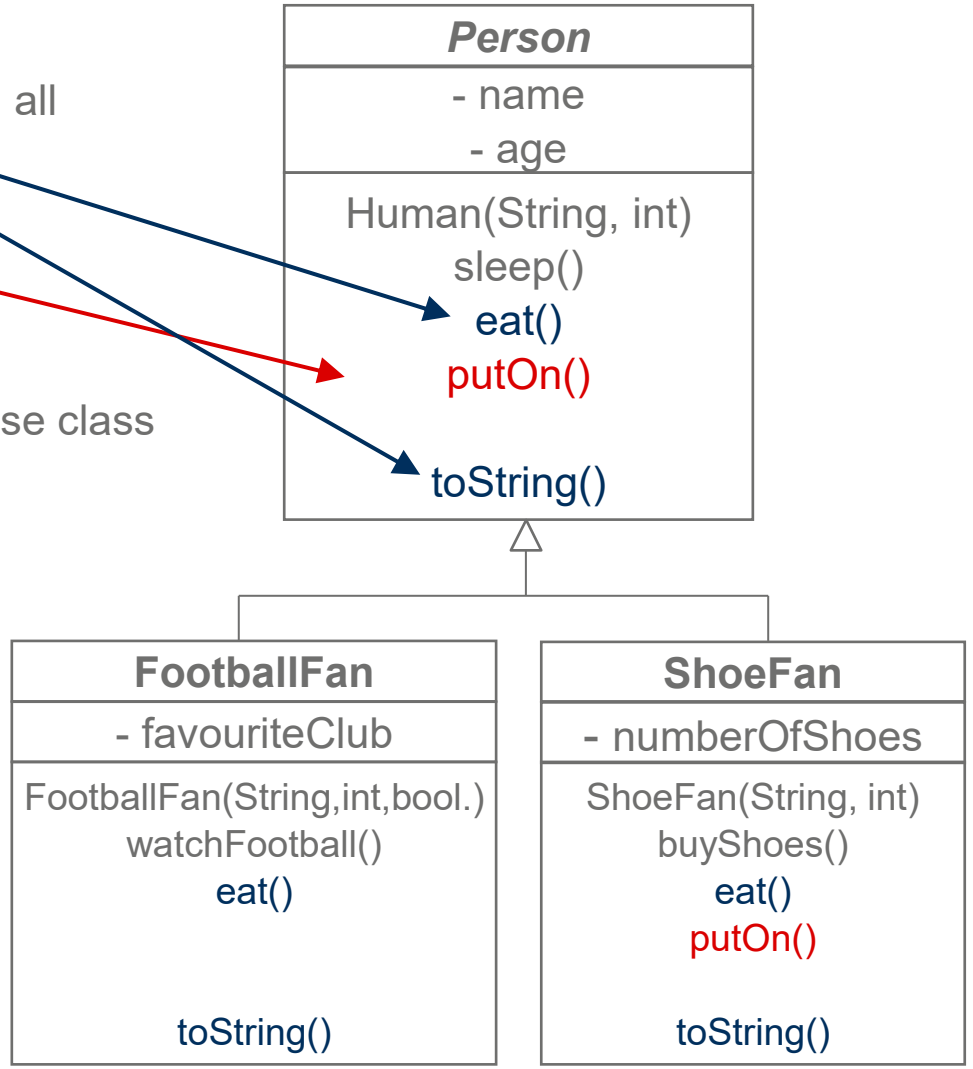
- Subclass

```
 public class FootballFan extends Person {
    …
  public String eat() {
   return super.eat() +
      "Can I have some more?";
  }
 }
```

Redefining properties in subclass

- **Starting point**
- Method with basic functionality that occurs in all subclasses, but is supplemented
- Method with standard functionality that is sufficient as is for some subclasses

- **Redefinition in subclasses**
- Overwriting the method from the top-level base class by specific implementation
- Integrating the implementation from the top-level base class via `super.<methodName>()`



Example: Redefining

Technische Hochschule Rosenheim
Technical University of Applied Sciences

- Extending the `Person` class with the `putOn()` method

```
public abstract class Person { …
 public String putOn() {
  return "put on: underwear and socks";
 } …
}
```

- Redefining the `ShoeFan` class

```
public class ShoeFan extends Person { …
 public String putOn() {
  return "put on: underwear, socks and shoes";
  } …
}
```

Example: Redefining, using basic function (1)

Technische Hochschule Rosenheim
Technical University of Applied Sciences

- Changes compared to the above example
  - `Person` class unchanged
  - Overwrite `eat()` method in both `ShoeFan` and `FootballFan` classes

- Redefining the `ShoeFan` class

```
public class ShoeFan extends Person {
  …
 public String eat() {
    String result = super.eat();
  result = result + "\n        Such a shame
      that this has so many calories...";
  return result;
  }…
}
```

Example: Redefining, using basic function (2)

Technische Hochschule Rosenheim
Technical University of Applied Sciences

- Redefining the `FootballFan` class

```
public class FootballFan extends Person {
  …
 public String eat() {
    String result =      .eat();
  result = result + "\n Can I have
      some more?";
  return result;

  }
  …
}
```

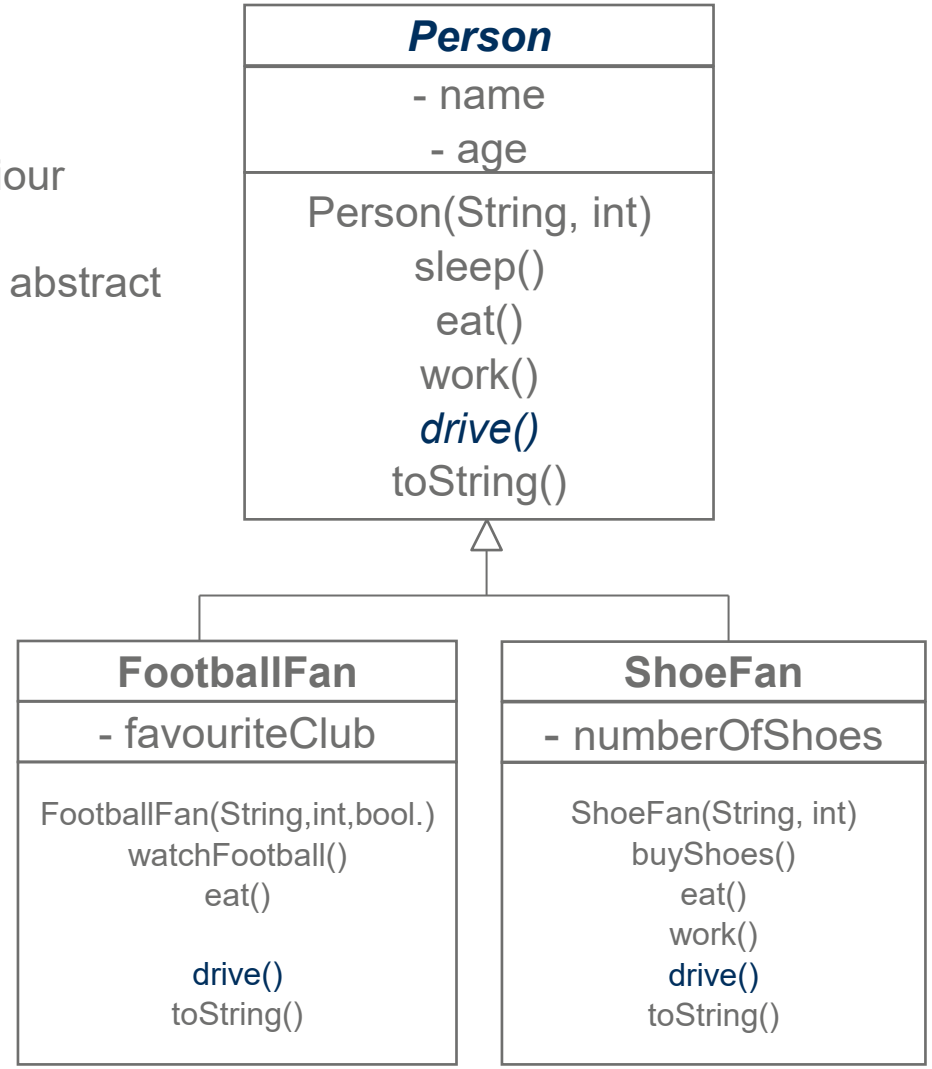
Defining properties in subclass

Technische Hochschule Rosenheim
Technical University of Applied Sciences

- **Starting point**
- Operation `drive()` is abstract, i.e. defines only the signature
- This ensures the existence of this behaviour
- No implementation!
- This also makes the top-level base class abstract

- **Definition in subclasses**
- Define specific implementations for the abstract operations

- Required in every non-abstract subclass



Example: Defining (1)

Technische Hochschule Rosenheim
Technical University of Applied Sciences

- Changes compared to the above example
  - `Person` class only defines the interface of the `drive()` method; thus becomes abstract class
  - `ShoeFan` and `FootballFan` classes both extended by implementation of the `drive()` method

- New version of the `Person` class

```
public abstract class Person {
   …
 public abstract String drive();
}
```

Example: Defining (2)

Technische Hochschule Rosenheim
Technical University of Applied Sciences

- Extension of `FootballFan` with definition of `drive()`

```
public class FootballFan extends Person {
  …
 public String drive() {
  return "Please start driving, I want to go to the game!";
}…}
```

- Extension of `ShoeFan` with definition of `drive()`

```
public class ShoeFan extends Person {
  …
 public String drive() {
  return "Then off to the shoe shop!";
}…}
```

Interface – meaning

Technische Hochschule Rosenheim
Technical University of Applied Sciences

- Definition: **interface**
- Special form of class
- No objects can be derived directly from the interface

- **Behaviour**
- Only defines abstract operations, no implementations
  Exception: `default` implementations
- Thus only sets requirements
- No executable statements (since Java8: `static` and `default` possible)
- No constructors

- **Properties**
- Contains no modifiable attributes
- Publicly visible constants are possible as attributes

- **All methods / data elements have implicit public visibility!**

Interface – implementation in Java

Technische Hochschule Rosenheim
Technical University of Applied Sciences

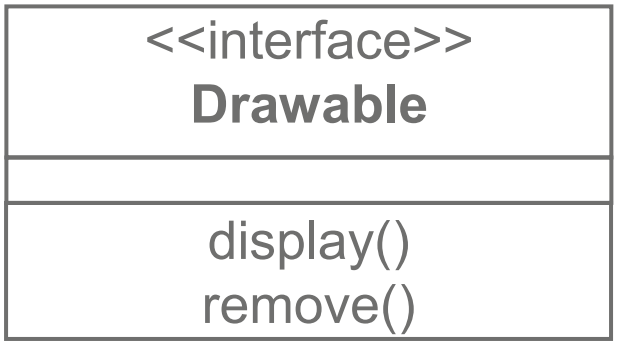
- Significance of Java
  - Enables clear separation of implementation and interface
  - Multiple inheritance of specific classes is not allowed in Java
  - However, implementation of multiple interfaces is possible!!!

- Implementation in Java:
  - Reserved keyword `interface` (instead of `class`)
  - Each interface has its own `.java` file which is translated into a `.class` file

Interface in UML

Technische Hochschule Rosenheim
Technical University of Applied Sciences

- Interface in UML
  - Symbol analogous to class
  - Stereotype `<<interface>>` above the class name
  - Interface is always also abstract, doesn't have to be explicitly marked as abstract



Terms – supplier and client

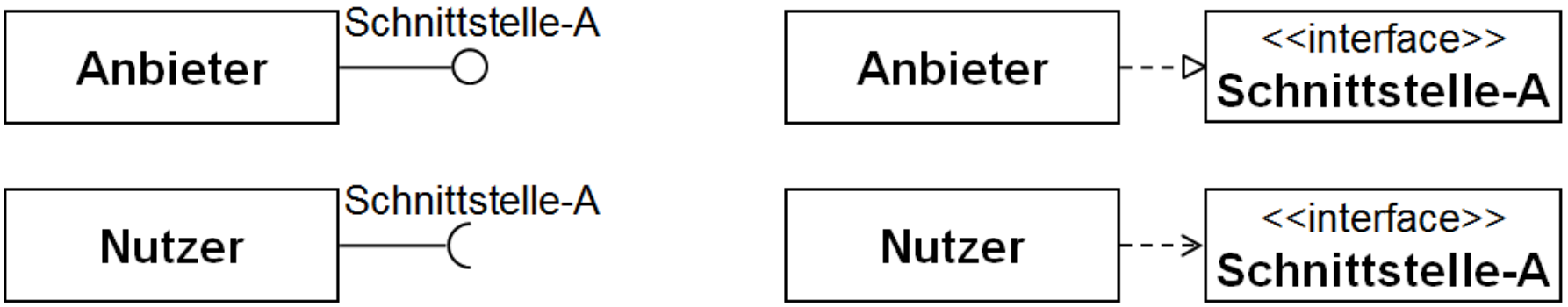
Technische Hochschule Rosenheim
Technical University of Applied Sciences

- Supplier of an interface
  - Realises the interface, i.e. implements the operations
- Client of an interface
  - Uses the interface, i.e. calls the operation
  - Does not know specific implementation!

Suppliers and clients in UML

Technische Hochschule Rosenheim
Technical University of Applied Sciences

Provision and use of interface A



Interaction via interface A

Terms – realisation and inheritance

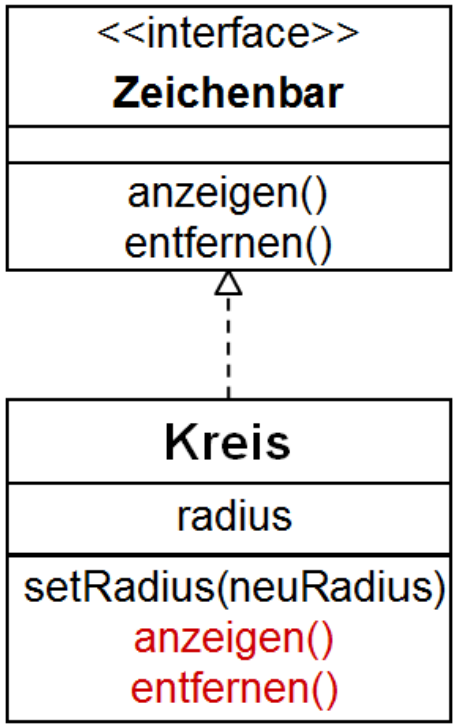
Technische Hochschule Rosenheim
Technical University of Applied Sciences

- Realisation
  - Interface not executable on its own
  - Specific class is derived from interface
  - Simply put: "Specific class implements the interface"
  - In doing so, implements all defined operations of the interface

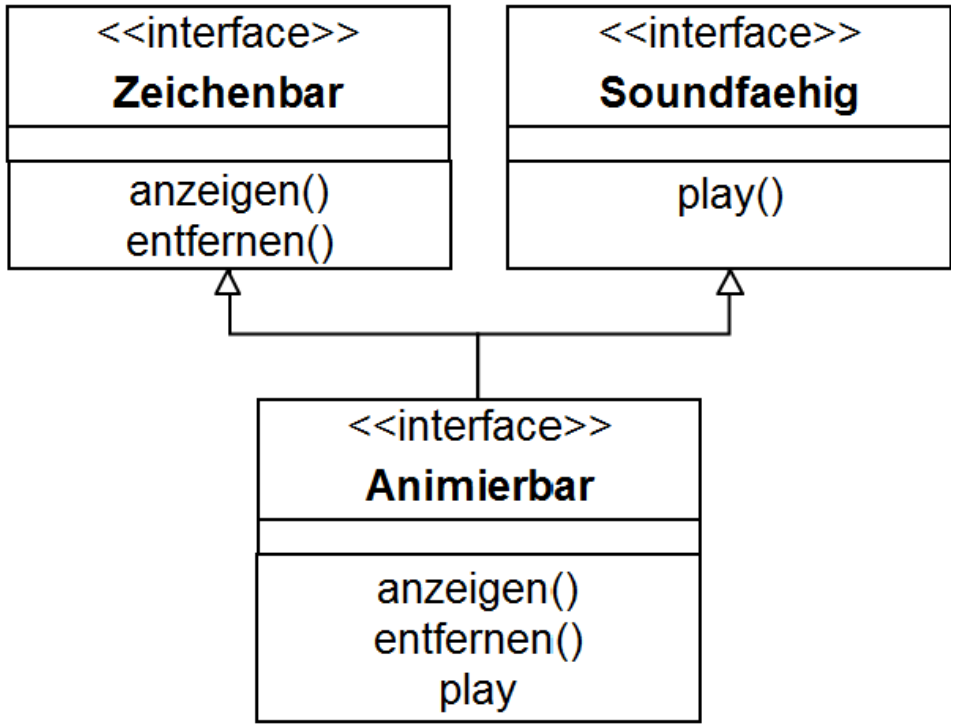
- Inheritance between interfaces
  - New interface extends old interface
  - In doing so, only adds abstract operations
  - In Java: interface can extend multiple interfaces
  - i.e. multiple inheritance between interfaces is possible!

Example: realisation and inheritance

Technische Hochschule Rosenheim
Technical University of Applied Sciences



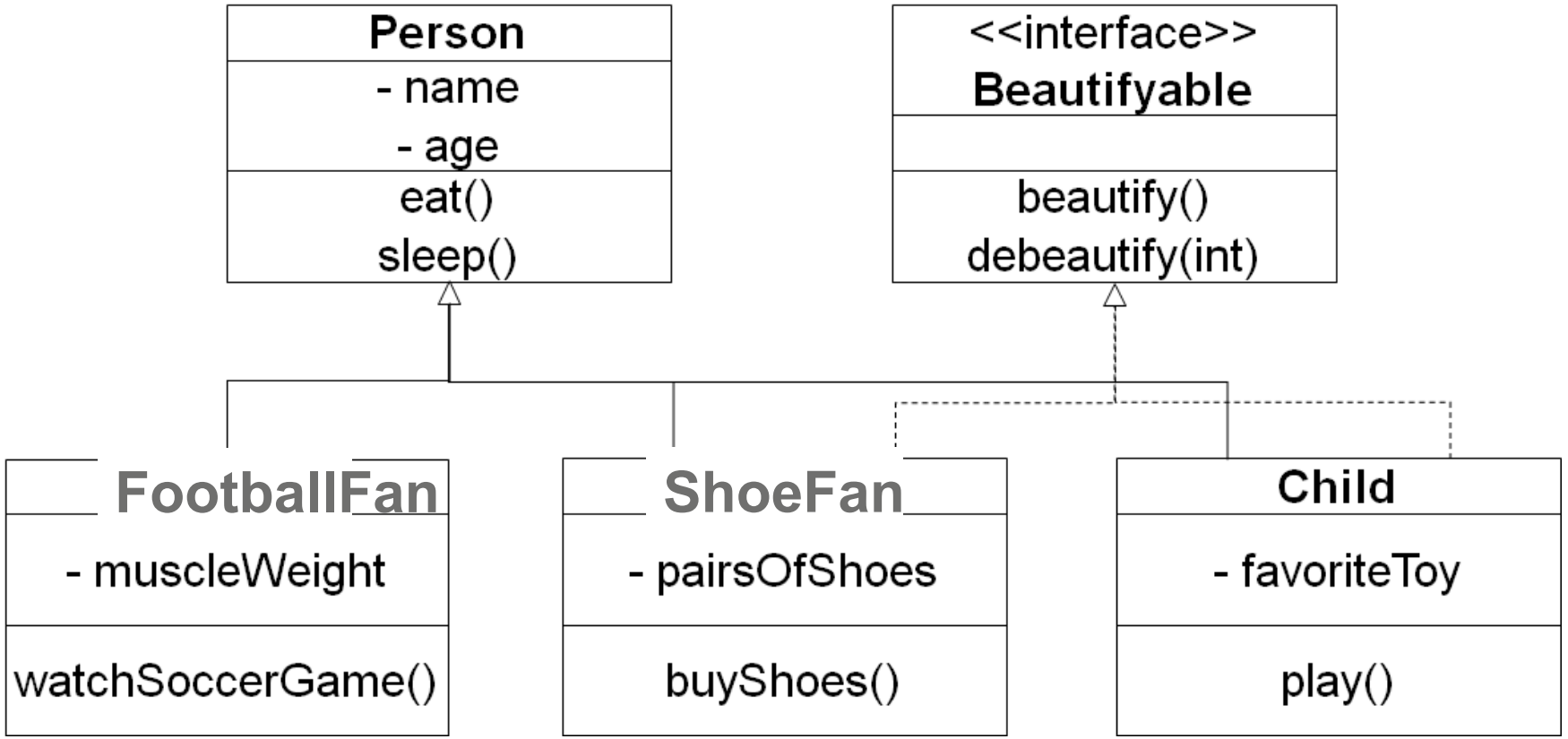
Realisation



Inheritance

Example

Technische Hochschule Rosenheim
Technical University of Applied Sciences



Implementation of the interface

Technische Hochschule Rosenheim
Technical University of Applied Sciences

- Changes compared to the above example
  - `FootballFan` and `Person` classes unchanged
  - Extension of the `Main` class
  - New `Beautifyable` interface
  - `ShoeFan` class implements `Beautifyable` interface
  - New `Child` class implements `Beautifyable` interface

- New `Beautifyable` interface

```
public interface Beautifyable {
 public String beautify();
 public String debeautify(int minutes);
}
```

Interfaces vs. abstract classes

Technische Hochschule Rosenheim
Technical University of Applied Sciences

- During the design phase of a complex software system: often difficult to choose one of the two variants
- **In favour of interfaces**: greater flexibility through the ability to be used in different class hierarchies
- **In favour of classes**: possibility to realise already formulatable parts of the implementation, and the ability to accommodate static components and constructors
- Combination of both approaches: first provide an interface, then simplify its application by using a helper class

Interfaces vs. abstract classes

Technische Hochschule Rosenheim
Technical University of Applied Sciences

- Example: subsequent changes
  - Subsequent changes to interfaces are also not easy: an abstract class can be given a specific method, which does not result in any source code adaptation for subclasses.

- What should we do if we want to implement a `getTimeInSeconds()` helper function ?
- In the interface → all implementing classes must reimplement this implementation.
- In the abstract base class → simply insert into the abstract class, no changes necessary in the class users.

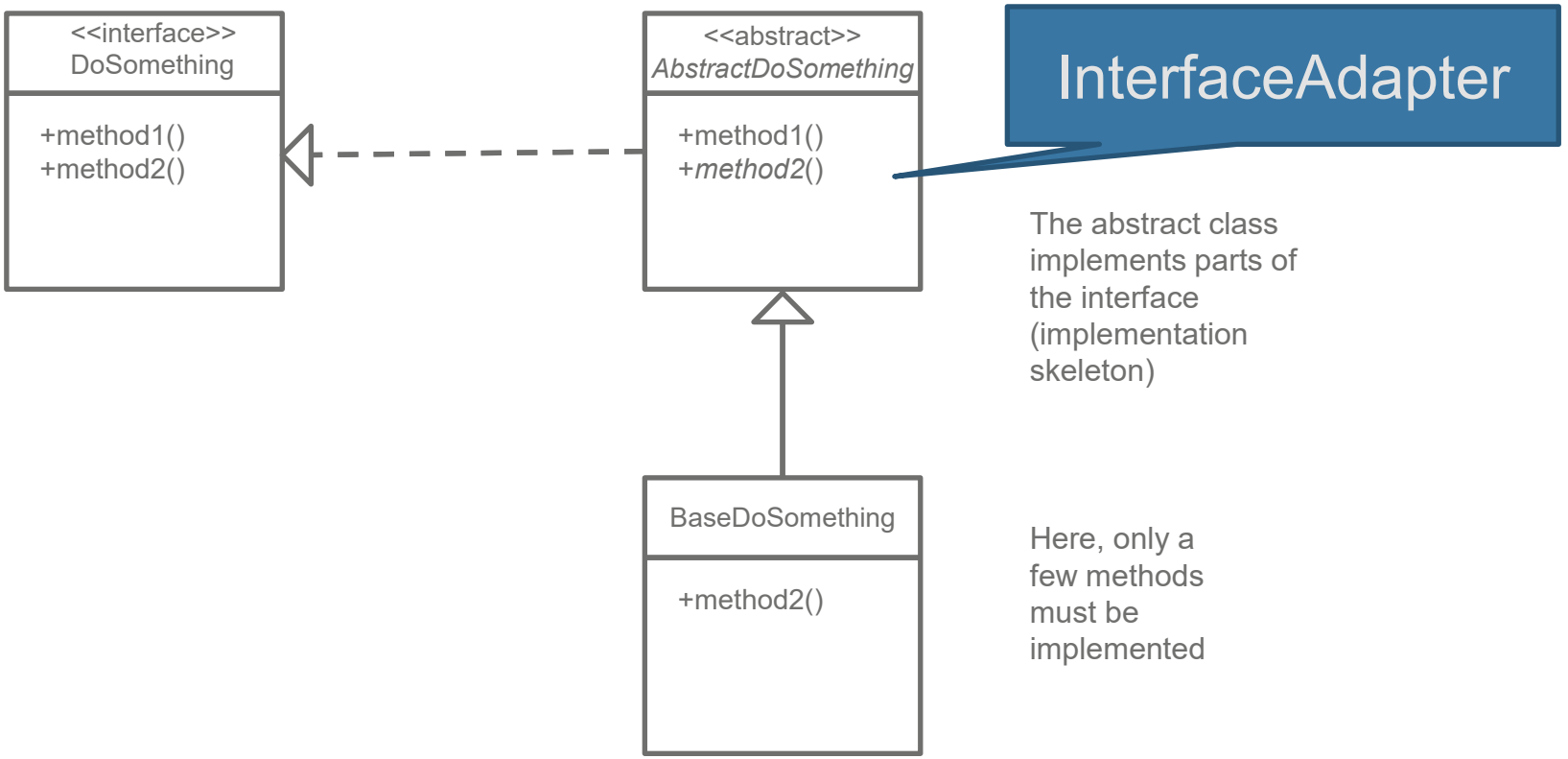
```
abstract class Timer {
 abstract long getTimeInMillis();

 long getTimeInSeconds() {
  return getTimeInMillis() / 1000;
  }
}
```

Combination of interfaces and abstract classes

Technische Hochschule Rosenheim
Technical University of Applied Sciences

- Adapters provide a basic implementation of an interface



Type information at runtime

Technische Hochschule Rosenheim
Technical University of Applied Sciences

```
AbstractBasis ab1 = new NormalBasis();
AbstractBasis ab2 = new SpecificA();
NormalBasis nb1 = new SpecificB();
SpecificA sa1 = new SpecificA();
SpecificB sb1 = new SpecificB();
```

- Instances of subclasses can be stored in base classes
- Runtime information through
  - **`instanceof`**: e.g. `if(ref instanceof MyClass){ … }`
  - Via `Object.getClass()`

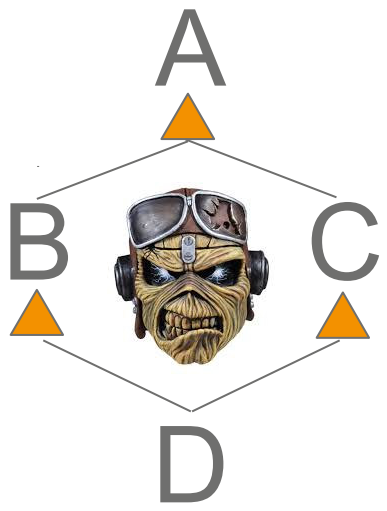
Deadly Diamond of Death

Technische Hochschule Rosenheim
Technical University of Applied Sciences

Problems that occur with such an inheritance structure

1. If class `A` provides an attribute `a`, do type `D` objects then have two `a` attributes, one through inheritance from `B` and one through inheritance from `C`?

2. If classes `B` and `C` both provide the method `public void foo()` with different implementations, which method then applies in `D`?



Multiple inheritance is shown here. Class D inherits from B and C (and twice from A). This is not possible without restrictions in Java.

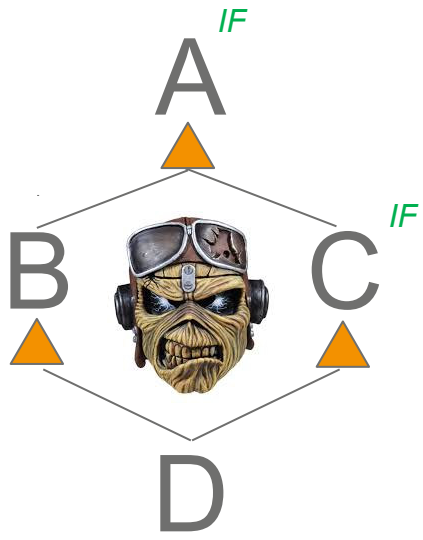
How are these problems handled in Java?

Technische Hochschule Rosenheim
Technical University of Applied Sciences

There is still no multiple inheritance of classes in Java. This means that exactly one class always comes after `extends`. It follows that either `B` or `C` must be an interface. Since interfaces cannot inherit from classes, then `A` is also an interface.

1. An attribute `a` in `A` must be `static final` because `A` is an interface. This means that `a` cannot be overwritten by `B` or `C`. Consequently, `D` has exactly one `a`, which is defined by `A`, and that's all!

2. If `B` provides a `void foo()` method and `C` (as an interface) provides a `default void foo()` method, then the method of `B` applies.
   a) if `A` and `C` provide `default void foo()`, then `C` applies (similar to overwriting)
   b) if `B` and `C` are both interfaces and provide `default void foo()`, then it must be explicitly specified which applies, otherwise a compiler error will occur.




```
public class D implements B,C{

    public void foo(){
        B.super.foo();
    }
}
```