Recap

- Object-Oriented Programming
    - Dealing with complexity by structuring your code
    - Classes and objects
- Classes
    - Unit of code to define some type of object
    - Contains fields (= variables, data) and methods (= behaviour)
- Objects
    - Concrete individuals of a certain class

→ Horse.java

```java
public class Horse {
  // the fields/variables of a class to store data about an instance
  String color;
  String name;
  int currentSpeed;

  // constructor to define what happens when a new object is created
  public Horse(String name) {
    this.name = name; // "this" to distinguish field and local variable
  }

  // methods of the class to define their behaviour
  public Horse mate(Horse partner) {
    // two horses meet and make a new horse
  }

  public static void main(String[] args) {
    // create an instance of type horse
    Horse h1 = new Horse("Joe");
    // create a second instance of type horse
    Horse h2 = new Horse("Jane");
  }
}
```

Eigenschaft

~~static~~

demo

Exercise 8

UNIVERSITÄT
ZU KÖLN

Session 9: Methods and Inheritance

Softwaretechnologie: Java I

Nils Reiter
nils.reiter@uni-koeln.de

December 11, 2024

INSTITUT FÜR
DIGITAL HUMANITIES
UNIVERSITÄT ZU KÖLN

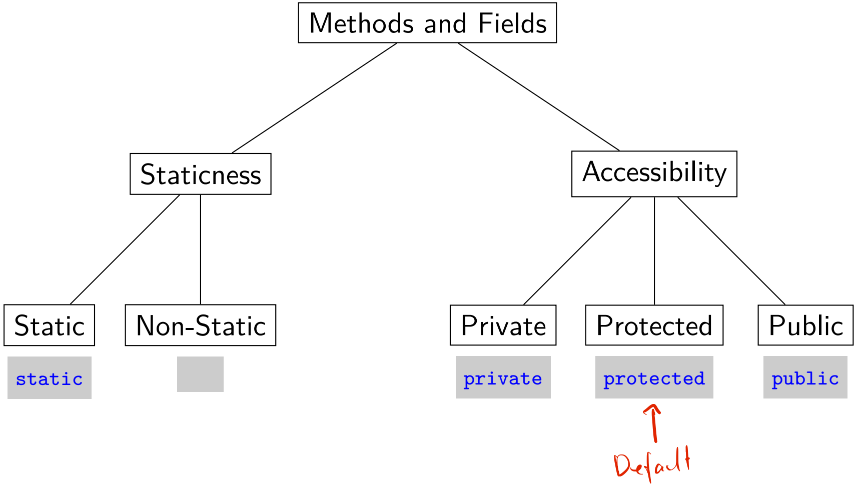
Packages
(leftover from last week)

- Multiple classes often belong conceptually together
- Packages can be used to group classes (and files)
- Package declaration: `package de.nilsreiter.java.bla;`
  - First statement within a file
  - Package hierarchy must reflect directory hierarchy
    - Eclipse hides that from us
- Package name conventions
  - Lower-cased
  - ‘Reversed URLs’ to be globally unique

Section 1

Methods

Introduction



Staticness

Non-static

- Methods can only be used with an object of the class in which they are defined
  - E.g., in order to call method `mate(Horse)`, we need an object of type Horse
- Default behaviour (unmarked methods are non-static)
- Also applies to fields
- E.g.: `INSTANCE.METHOD()`

Staticness

Non-static

- Methods can only be used with an object of the class in which they are defined
    - E.g., in order to call method `mate(Horse)`, we need an object of type Horse
- Default behaviour (unmarked methods are non-static)
- Also applies to fields
- E.g.: `INSTANCE.METHOD()`

Static

- Methods can be used without an object
    - E.g., marking a species as endangered is something for the class, not for instances of it
- Java keyword `static`
- E.g.: `CLASS.METHOD()`

```java
public class Horse {
  // the fields/variables of a class to store data about an instance
  int age;    ← non-static

  // boolean field to store what horses eat
  static String diet = "herbivore";

  public void birthday() {
    // it's the horse's birthday
    age = age + 1;
  }

  public static boolean isCarnivore() { return diet.equals("carnivore"); }
  public static boolean isHerbivore() { return diet.equals("herbivore"); }

  public static void main(String[] args) {
    Horse h1 = new Horse();

    // call a non-static method
    h1.birthday();

    // call a static method
    Horse.isHerbivore();
  }
}
```

Accessibility

- Public access – `public`
  - Method/field can be accessed from anywhere
- Protected access – `protected`
  - Method/field can only be accessed from within the same package
  - If no access is specified, it's protected
- Private access – `private`
  - Method/field can only be accessed from within the same class

Accessibility

- Public access – `public`
  - Method/field can be accessed from anywhere
- Protected access – `protected`
  - Method/field can only be accessed from within the same package
  - If no access is specified, it's protected
- Private access – `private`
  - Method/field can only be accessed from within the same class

Why?

- Modularization is important for dealing with complexity
- A complex program consists of many small parts that are not as complex
- Small parts are only maintainable if they have restricted interfaces
- Access restrictions can enfore that

demo

Horse with static and private fields/methods

Section 2

Inheritance

Introduction

Inheritance – "Vererbung"

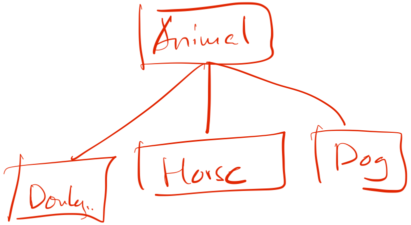
- Important concept in object-oriented programming
- Classes represent kinds of things, because they show similar behaviour
  - Not all kinds are totally unique
  - Many kinds share certain properties
- E.g. Donkeys move in a similar way as horses do and both are mammals etc.

Introduction

Inheritance – “Vererbung”

- Important concept in object-oriented programming
- Classes represent kinds of things, because they show similar behaviour
    - Not all kinds are totally unique
    - Many kinds share certain properties
- E.g. Donkeys move in a similar way as horses do and both are mammals etc.
- Inheritance allows us to model this
- Many domains have hierarchical structures
    - E.g., animal species, companies, kitchen equipment

Class Inheritance



- A class inherits from another class
- New keyword: `extends`, used in the class declaration:

```
public class Horse extends Animal { ... }
```

- Horse: sub class
- Animal: super class

Class Inheritance

Meaning

- No change in accessibility/visibility rules
  - private fields/methods still not visible, protected only within the same package etc.
- Objects of sub class can call methods defined in super class
  - E.g., the class Animal can define a walk-method for *all* sub classes

Class Inheritance

Meaning

- No change in accessibility/visibility rules
  - private fields/methods still not visible, protected only within the same package etc.
- Objects of sub class can call methods defined in super class
  - E.g., the class Animal can define a walk-method for *all* sub classes
- Objects of the sub class can be assigned to variables of the super class
  - `Animal someAnimal = new Horse();`

    ~~Horse h = new Animal();~~
  - `Animal[] zooAnimals = new Animal[2] { new Horse(), new Donkey() };`
- Casting from sub class to super class (“upwards”) always works
  - `Animal someAnimal = (Animal) myHorse;`

demo

Animal and Hippo

Inheritance

Method Overriding

```
class Animal {
  public void step(int size) { /*...*/ };
}

class Horse extends Animal {
}

class Main {
  public static void main(String[] args) {
    Horse h = new Horse();
    h.step(5);
  }
}
```

Inheritance

Method Overriding

```java
class Animal {
  public void step(int size) { /*...*/ };
}

class Horse extends Animal {
}

class Main {
  public static void main(String[] args) {
    Horse h = new Horse();
    h.step(5);
  }
}
```

- Objects of the sub class can call methods defined in super class

Inheritance

Method Overriding

```java
class Animal {
  public void step(int size) { /*...*/ };
}

class Horse extends Animal {
  public void step(int size) { /*...*/ };
}

class Main {
  public static void main(String[] args) {
    Horse h = new Horse();
    h.step(5);
  }
}
```

Inheritance

Method Overriding

```java
class Animal {
  public void step(int size) { /*...*/ };
}

class Horse extends Animal {
  public void step(int size) { /*...*/ };
}

class Main {
  public static void main(String[] args) {
    Horse h = new Horse();
    h.step(5);
  }
}
```

- Methods in sub class override methods in super class
- Calling super method explicitly
  - Outside of sub class by casting: `((Animal)h).step(5);`
  - Inside of sub class with `super`: `super.step(5);`
    - Think of super as `((Animal) this)` (in this case)

Variable Type != Object Type

- Each variable has a type
  - E.g., `int`, `String`, `Horse`, ...
- Each object and value has a type
  - E.g., `int`, `String`, `Horse`, ...

Variable Type != Object Type

- Each variable has a type
  - E.g., `int`, `String`, `Horse`, ...
- Each object and value has a type
  - E.g., `int`, `String`, `Horse`, ...
- If object/value type and variable type match, we can make an assignment
  - E.g., `int i = 5;`
  - E.g., `Horse h = new Horse();`

Variable Type != Object Type

- Each variable has a type
  - E.g., `int`, `String`, `Horse`, ...
- Each object and value has a type
  - E.g., `int`, `String`, `Horse`, ...
- If object/value type and variable type match, we can make an assignment
  - E.g., `int i = 5;`
  - E.g., `Horse h = new Horse();`
- It's a compile error, if they do not match
  - E.g., `int i = true;` ⚠
  - E.g., `Horse h = new Donkey();` ⚠

Variable Type != Object Type

- Each variable has a type
  - E.g., `int`, `String`, `Horse`, ...
- Each object and value has a type
  - E.g., `int`, `String`, `Horse`, ...
- If object/value type and variable type match, we can make an assignment
  - E.g., `int i = 5;`
  - E.g., `Horse h = new Horse();`
- It's a compile error, if they do not match
  - E.g., `int i = true;` ⚠
  - E.g., `Horse h = new Donkey();` ⚠
- But we can assign a object of a sub class to a variable of a super class
  - E.g., `Animal a = new Horse(); //if Horse extends Animal`

`java.lang.Object`

- All classes inherit automatically from `java.lang.Object`
  - I.e., every object is in an instance of `java.lang.Object` (though maybe indirectly)
- Class provides a few methods

  Javadoc
  - `Object clone()`
  - `boolean equals(Object obj)`
  - `int hashCode()`
  - `String toString()`
  - `void wait()`, `void wait(long timeout)`, `void wait(long timeout, int nanos)`
  - `void notify()`, `void notifyAll()`
  - `void finalize()`
  - `Class<?> getClass()`

Testing Inheritance

- New operator: `isinstance`

```java
Horse h = new Horse();

h instanceof Horse;  // true
h instanceof Object; // true
h instanceof String; // false
h instanceof Animal; // true if Horse extends Animal
```

Remarks on Inheritance

- Why inheritance?
  - Model commonalities in our domain
  - The same behaviour can be implement as high as possible in the hierarchy, and only once
  - Again, reducing complexity

Remarks on Inheritance

- Why inheritance?
  - Model commonalities in our domain
  - The same behaviour can be implement as high as possible in the hierarchy, and only once
  - Again, reducing complexity
- Multiple inheritance: Can a class inherit from multiple classes?
  - In Java: No
    - Because method calls then become ambiguous

Remarks on Inheritance

- Why inheritance?
  - Model commonalities in our domain
  - The same behaviour can be implement as high as possible in the hierarchy, and only once
  - Again, reducing complexity
- Multiple inheritance: Can a class inherit from multiple classes?
  - In Java: No
    - Because method calls then become ambiguous
  - In C++/Python: Yes!
    - C++: Programmer has to resolve ambiguity with additional syntax
    - Python: Depends on the order in which inheritance has been specified

demo

Exercise