CHAPTER 3: INHERITANCE

LEARNING OBJECTIVES

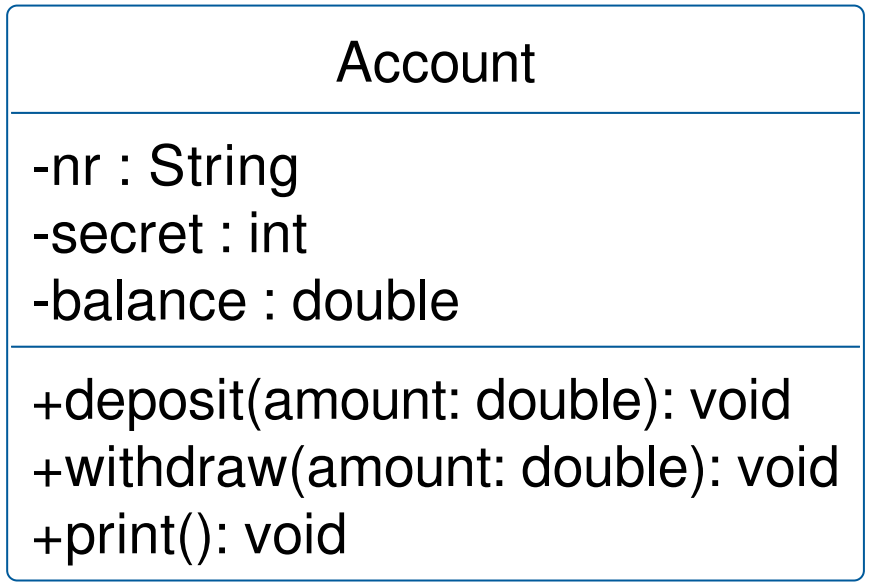
- explain the Java inheritance concept
- explain the principle of aggregation
- apply the visibility levels required for inheritance
- understand UML class models with inheritance relationships
- understand where inherited properties can be used
- use the keywords "this" and "super"
- understand and use constructor chaining
- know class "Object" and its methods
- use the "instanceof" operator
- know the keyword "final" and its effects
- know abstract classes and methods and when to use them
- know when to use aggregation and when to use inheritance

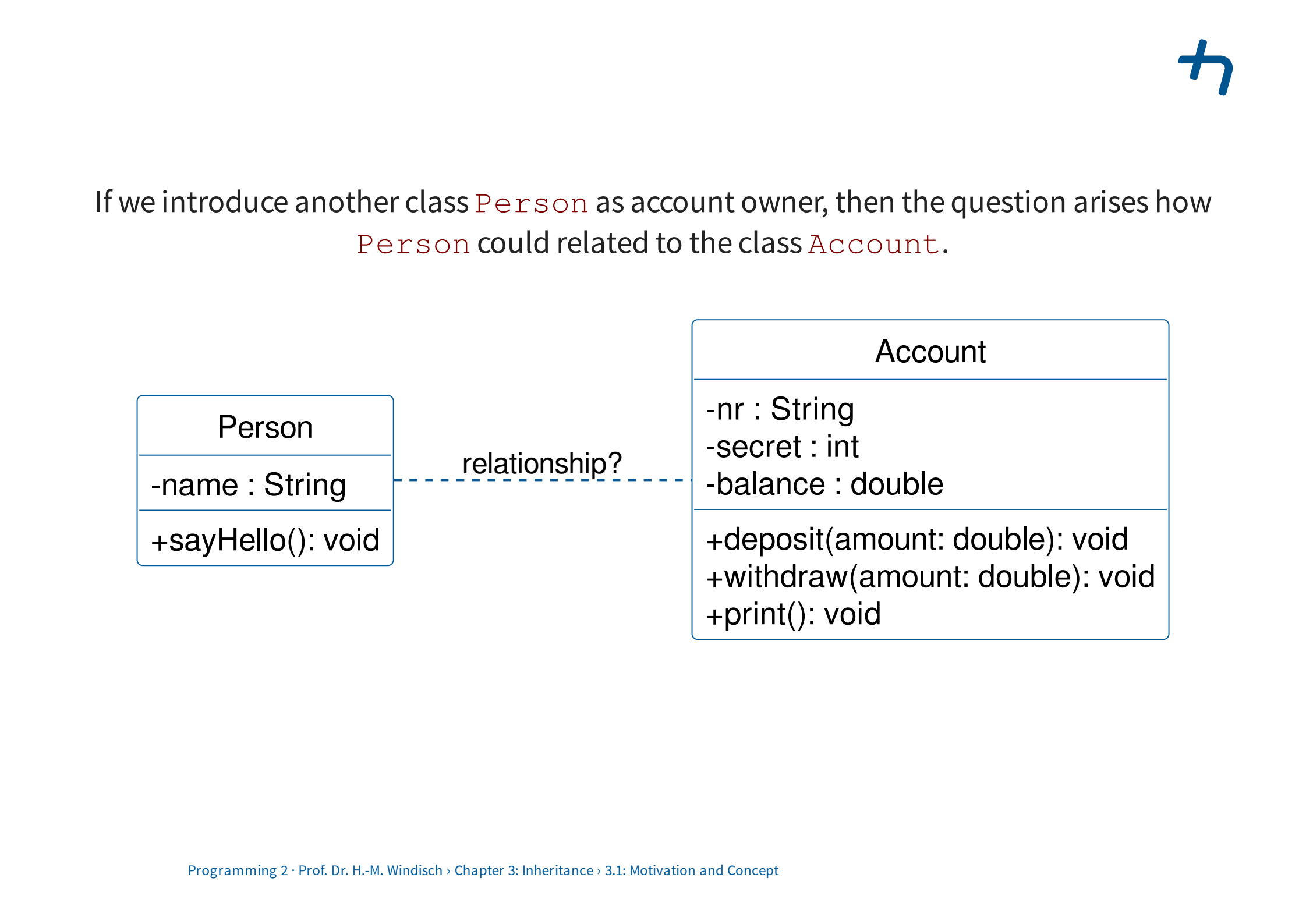
3.1 MOTIVATION AND CONCEPT

Real world Java applications do not consist of just one class

When developing large software systems, it is important for economic reasons to structure existing source code and make it reusable. How can we achieve that?

Let's look at a class for accounts.





If we introduce another class `Person` as account owner, then the question arises how `Person` could related to the class `Account`.

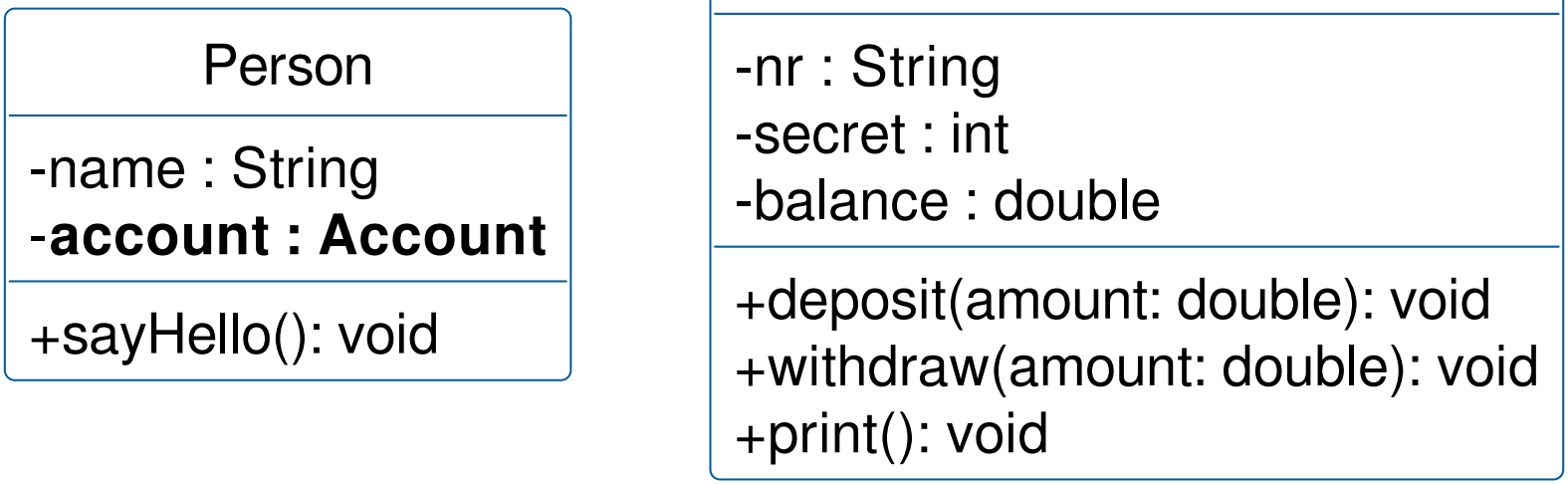
Basically we can distinguish between two possibilities:

aggregation

inheritance

AGGREGATION

Aggregation refers to building objects from other objects. For example, the `Person` class could have an attribute `account` of type `Account`.



AGGREGATION WITH CARS

A car has a steering wheel and 4 wheels.

... if we convert this into Java, we might get:

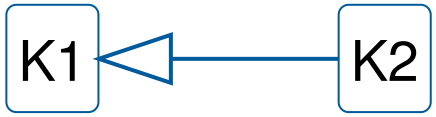
```
class Wheel { ... }
```

```
class SteeringWheel { ... }
```

```
class Car {
  private Wheel[] wheels = new Wheel[4];
  private SteeringWheel steeringWheel = new SteeringWheel();
  //...
}
```

INHERITANCE

The second type of reuse in object-oriented languages is inheritance. If a class `K2` inherits from a class `K1`, `K2` receives all **non-private** attributes and methods of `K1`. Typically `K2` defines more "own" attributes and methods and thus represents a specialisation of `K1`.



In UML class models the arrowhead points to the parent.

`K2` is a class derived from `K1`, syntactically this is expressed in Java with the keyword `extends`:

```
class K1 { ... }
```

```
class K2 extends K1 { ... }
```

AGGREGATION VS. INHERITANCE

Inheritance defines a "is-a" relationship. For example, "a share is an asset", but not "a share has an asset". Thus it makes sense to derive share from asset. Aggregation does not make sense because it makes less sense if a share object has a reference to an asset object.

Attention: Since aggregation is more flexible than aggregation, try to use aggregation first and inheritance only if it really "fits"!

INHERITANCE AND VISIBILITY LEVELS

Only non-private properties are inherited. For finer control of inheritance, a visibility level is introduced via the keyword `protected`, which describes properties that are non-public but are still inherited.

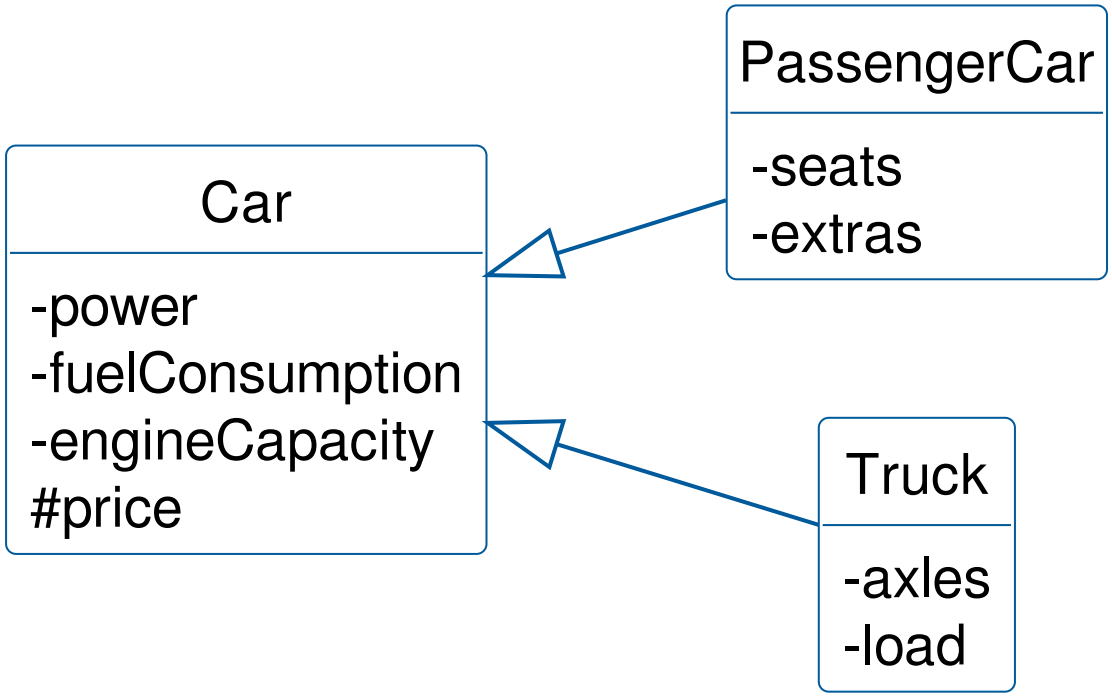
```
class Person {
    protected String name;
    // Aggregation of an Address object
    private Address address;
    public Address getAddress() {
        return address;
    }
}
// a Student is a Person
class Student extends Person {
    private int registrationNumber;
}
```

- The Student class inherits the attribute `name` and the method `getAddress` from the superclass Person.
- The methods of a Student object can accordingly use the inherited attributes as well as `registrationNumber`.
- Access to the address attribute is possible via the inherited getter `getAddress`!

In a UML class diagram, the visibility level `protected` is represented by the character # (hash) (see example in the next section).

3.2 UML CLASS MODELS AND INHERITANCE HIERARCHIES

The notation for UML class models is extended to include inheritance relationships.
The arrowhead is always pointing to the parent class!



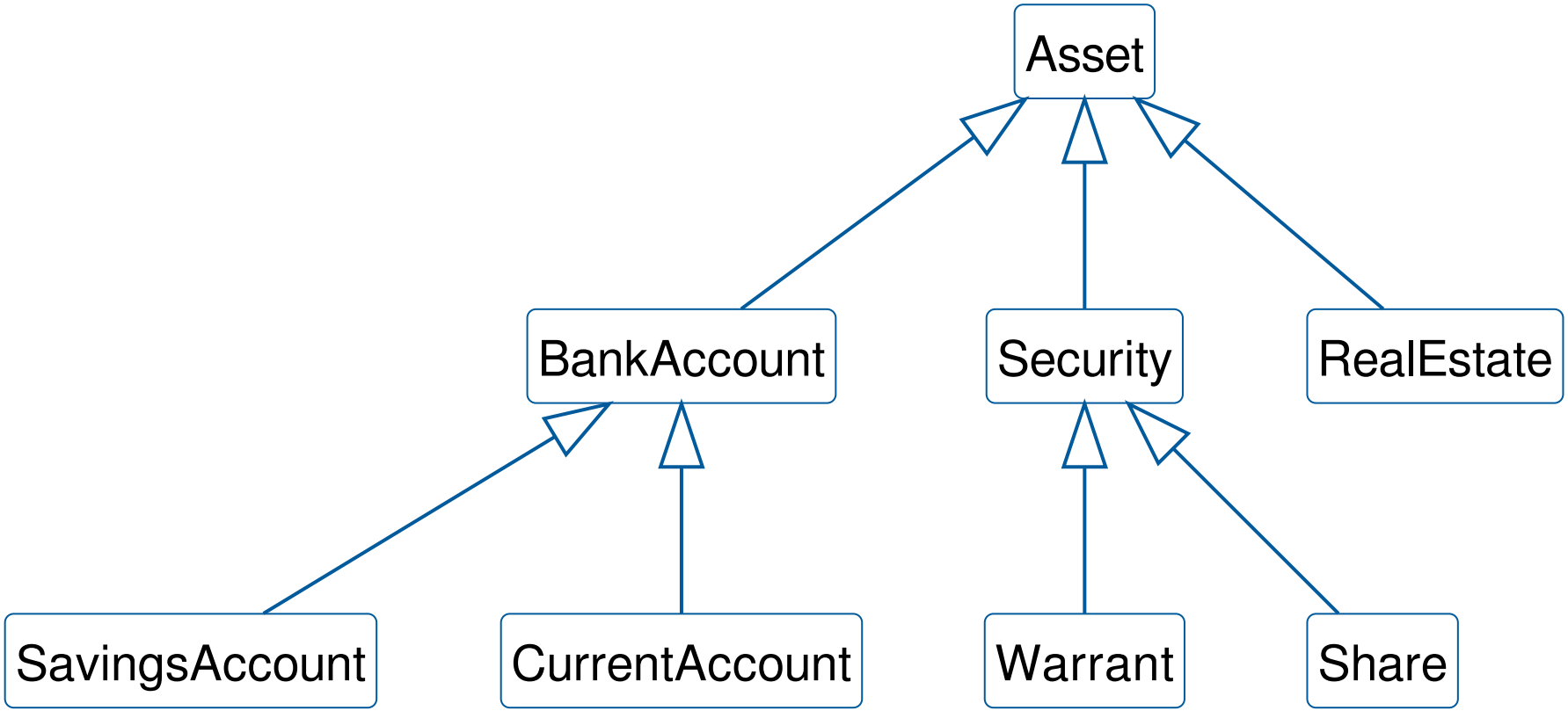
The model contains the following statements:

- `Car` and `Truck` inherit attributes of the superclass `Car`.
- Common attributes and methods (not shown here) are reusable by the superclass.
- The class `Car` has 4 attributes of which only `price` may be inherited, as the other attributes are private.
- The class `PassengerCar` has 3 attributes, one inherited and 2 "locally" defined: `price`, `seats`, `extras`.
- The class `Truck` has 3 attributes, one inherited and 2 "locally" defined: `price`, `axles`, `load`.

INHERITANCE HIERARCHIES

In Java, a class can have any number of subclasses, but only one superclass - this is called single inheritance. Other object-oriented languages, such as C++, allow multiple inheritance. Single inheritance creates a derivation tree in the graphical representation of a class model. The root of the tree contains the most abstract class, the closer you get to the leaves the more concrete the classes become.

If the classes of an abstraction level are arranged at the same tree level, an inheritance hierarchy is created.



3.3 ACCESSING INHERITED PROPERTIES

Look at method f() - which statements are correct?

```
public class Base {
  public int a;
  protected int b;
  int c;
  private int d;

  public void f() { … }
  protected void g() { … }
  void h() { … }
  private void k() { … }

}
```

```
public class Child extends Base {
  public void f() {
    a = 1;
    b = 2;
    c = 3;
    d = 4;
    f();
    g();
    h();
    k();
  }
}
```

Hint: The chapter on classes and objects contains information on visibility levels.

In the example, the class `Child` inherits the following properties from class `Base`:

- Attributes: `a`, `b`, `c`
- Methods: `f`, `g`, `h`

Attribute `d` and method `k` are private and are therefore not inherited by `Child`!
Accordingly, the following accesses are not permitted in method `f`:

- `d = 4;`
- `k();`

Remarks:

- Method `f` calls itself recursively, which leads to an infinite loop! With `super.f();` the overridden method of the superclass can be called.
- write access to attribute `d` could be realised through a public setter in class `Base`.

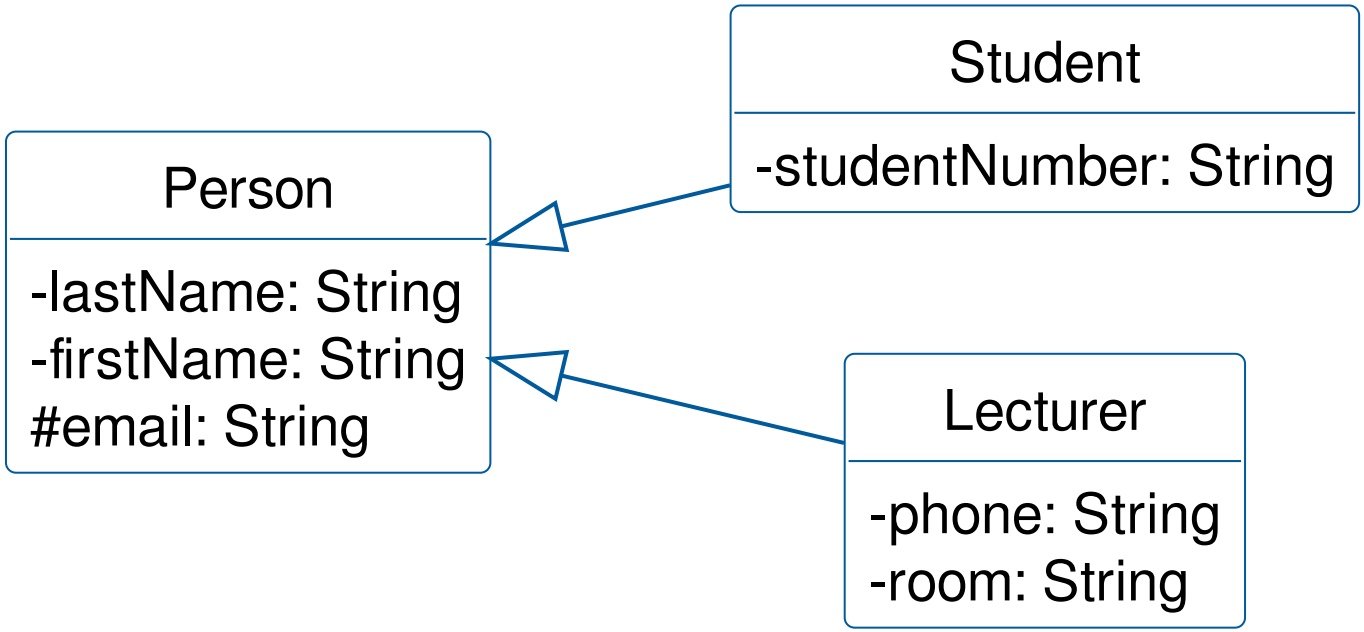
EXERCISE: METHOD INHERITANCE

How can one achieve that a method `m` is inherited but cannot be called from the outside?

- [ ] Set visibility to public so that `m` is public for derived classes
- [ ] Set visibility to protected, so that `m` is only visible within the inheritance hierarchy
- [ ] Set no visibility, so that `m` is not visible outside the class
- [ ] Set visibility to private, so that `m` is a private property for all classes within the inheritance hierarchy

EXERCISE: INHERITED ATTRIBUTES

Given the following class model:



Which attributes are accessible in objects of type `Student` and `Lecturer`?

- [ ] `Lecturer` can access only phone, rom, lastName, firstName and email
- [ ] `Student` can access only studentNumber, lastName, firstName and email
- [ ] `Student` can access only studentNumber and email
- [ ] `Student` can access only studentNumber, phone, room, lastName, firstName and email
- [ ] `Lecturer` can access only studentNumber, phone, room, lastName, firstName and email
- [ ] `Lecturer` can access only phone, room and email

Which attributes are stored in an object of class `Student` or `Lecturer`?

- ☐ `Lecturer` has email, phone and room
- ☐ `Student` has studentNumber and email
- ☐ `Student` has studentNumber, lastName, firstName and email
- ☐ `Student` has studentNumber, phone, room, lastName, firstName and email

Which attributes are stored in an object of class `Person` or `Lecturer`?

- [ ] `Lecturer` has phone, room, lastName, firstName and email
- [ ] `Lecturer` has studentNumber, phone, room, lastName, firstName and email
- [ ] `Person` has studentNumber, phone, room, lastName, firstName and email
- [ ] `Person` has lastName, firstName and email

3.4 CREATING OBJECTS WITH INHERITANCE

THIS AND SUPER WITH INHERITANCE

Every Java object has the predefined references `this` and `super`.

The meaning of `this` was explained in the chapter about classes and objects, it refers to the object itself.

In contrast, `super` serves as a reference to the superclass.

Remarks:

The reference `super` can only be used with a concrete access, i.e. in the form `super.superclassMethod()` or `super.superclassAttribute`.

The reference `this`, on the other hand, can also be used by itself, e.g. to parameterise a method call with the current object: `aMethod(this);`

Example with usage of `super`

```
class Car {
    public String toString() { ... }
}

class Convertible extends Car {
    public String toString() {
        return "Convertible - " + super.toString();
    }
}
```

The reference `super` is used to call the toString method of the superclass, which is overwritten due to the name identity. Without `super`, the `toString` call would lead to an endless recursion, as the local method would always be called!

DEFINING CONSTRUCTORS FOR DERIVED CLASSES

A constructor (CTOR) has the task of correctly initialising an object during object creation.

```
class SuperClass {
    public SuperClass() { ... }
}

class DerivedClass extends SuperClass {
    public DerivedClass() {
        super(); // Call super class CTOR
    }
}
```

When creating an object of a derived class, the constructors must be executed along the inheritance hierarchy from top to bottom to ensure correct initialisation of the inherited attributes.

Correct chaining of constructors is necessary to define the initialization order. This can, for example, prevent a non-initialized superclass reference from being used for a method call, which would lead to a `NullPointerException`, since references are always initialized with `null`. To ensure the above procedure, every constructor of a derived class must call a suitable (i.e. with a matching parameter list) superclass constructor in its first line.

Consequently, attribute initialization starts at the most abstract base class and proceeds along the hierarchy.

DEFAULT CONSTRUCTOR

As long as no custom CTOR exists, the compiler creates a parameterless CTOR to ensure constructor chaining. Also, the compiler will add a super call as the first line in any custom constructor, if necessary.

See example on the next page...

Class for Passenger Cars

```
public class PassengerCar {
    protected String name;

    public PassengerCar() {
        this("unknown");
    }
    public PassengerCar(
        String name) {
        this.name = name;
    }
}
```

Class for Convertibles

```
public class Convertible
    extends PassengerCar {
    private boolean fabricRoof;
    public Convertible(boolean fabricRoof) {
        super();
        this.fabricRoof = fabricRoof;
    }
    public String toString() {
        return "Convertible: fabric roof: "
            + fabricRoof + ", name = "
            + name;
    }
    public static void main(String[] args) {
        System.out.println(
            new Convertible(true));
    }
}
```

- Starting with the object creation triggered by `new`
- ... first the `Convertible` instance will be initialised
- ... then the `toString` method is implicitely called

Class for Passenger Cars

```
public class PassengerCar {
    protected String name;
    public PassengerCar() {
        this("unknown");
    }
    public PassengerCar(
        String name) {
        this.name = name;
    }
}
```

Class for Convertibles

```
public class Convertible
    extends PassengerCar {
    private boolean fabricRoof;
    public Convertible(boolean fabricRoof) {
        super();
        this.fabricRoof = fabricRoof;
    }
    public String toString() {
        return "Convertible: fabric roof = "
            +  fabricRoof
            + ", name = " + name;
    }
    public static void main(String[] args) {
        System.out.println(
            new Convertible(true));
    }
}
```

- CTOR `Convertible(boolean)` is calling `PassengerCar()`
- CTOR `PassengerCar()` is calling CTOR `PassengerCar(String)`
- Ouptut is: `Convertible: fabric roof = true, name = unknown`

EXERCISE: CTOR CHAINING

```
public class Hello {
    public Hello() { System.out.print("Hello, Unknown"); }
    public Hello(String name) { System.out.print("Hello, " + name); }
}

class Person extends Hello {
    public Person() { super("Unnamed"); }
    public Person(String name) { super(name); }
}

class Other extends Hello {
    public Other(String name) { super(); }
}
```

Which outputs are correct?

- ☐ `new Other("Unknown")` will produce `"Hello, Unnamed"`;
- ☐ `new Hello("World")` will produce `"Hello, World"`;
- ☐ `new Person()` will produce `"Hello, Unknown"`;
- ☐ `new Person("Lisa")` will produce `"Hello, Lisa"`;
- ☐ `new Other("World")` will produce `"Hello, World"`;

3.5 THE SUPERCLASS OBJECT

Every class inherits from Object - even if it is not explicitly stated.

The class `java.lang.Object` is the base class for all classes because the compiler implicitly adds the following inheritance relationship:

```
class MyNewClass extends Object { ... }
```

If a class inherits from another class, it also inherits indirectly from `Object`, since inherited properties are inherited at any depth. This approach ensures that every object inherits certain standard methods from `Object`.

IMPORTANT STANDARD METHODS

- `boolean equals():`
  Object comparison
- `String toString():`
  yields the object's string representation in format `<class>@<hashcode>`
- `Class getClass():`
  yields the class description
- `int hashCode():`
  returns a hash value for the object for storage in hash tables

Please be aware: Methods in custom classes usually need to be overridden (or redefined). For example, in the default implementation, the equals() method does not compare the contents, but the references of the objects to be compared.

3.6 OVERIDING INHERITED METHODS

Inherited methods can be overridden, i.e. redefined, by specifying a method with the same signature (parameter types + result type). By means of...

```
super.<Methode>(<Parameter>)
```

...the overridden method can still be called.

In the following example for the `Square` and `ColoredSquare` classes, the `main()` method calls the `ColoredlSquare.toString()` method and `ColoredSquare.toString()` calls `Square.toString()`.

Square

```
class Square {
    private double sideLength;
    public Square(double sideLength) {
        this.sideLength = sideLength;
    }
    public String toString() {
        return "sideLength=" + sideLength;
    }
    static public void main(String args[])
        String msg =
        new ColoredSquare(2.25, "blue")
        .toString();
        System.out.println(msg);
    }
}
```

ColoredSquare

```
class ColoredSquare extends Square {
    private String color;
    public ColoredSquare(
        double sideLength,
        String color) {
        super(sideLength);
        this.farbe = farbe;
    }
    public String toString() {
        return "ColoredSquare [color=" +
        color + ", " + super.toString() +
        "]";
    }
}
```

Some remarks:

- The instances of the `ColoredSquare` class can only call one `toString` method
- The methods that are most concrete in the inheritance hierarchy are always called on instances
- The call `super.super` doesn't work
- The call `super.toString()` calls the `toString` of the super class. If the super class does not override `toString`, it inherits `toString` from its super class. Thus, the call also works on inherited methods in super classes

EXERCISE: OVERRIDING METHODS

Given the following example:

```
class Person {
    void update(String name) { ... }
    void print() { ... }
    void sendMail(String text) { ... }
}
```

```
class Student extends Person {
    void update(String name,
        String registrationNumber) { ... }
    void print() { ... }
    void mail(String text) { ... }
}
```

Which statements are correct?

- ☐ `Person.print` overrides `Student.print`
- ☐ `Student.print` overrides `Person.print`
- ☐ `Student.Student` overrides `Person.Person`
- ☐ `Student.update` overrides `Person.update`
- ☐ `Student.mail` overrides `Person.sendMail`

Which statements are correct?

- ☐ The methods `update(String name)` and `update(String name, String registrationNumber)` can be called on an instance of `Person`
- ☐ The `mail` and `sendMail` methods can be called on an instance of `Student`
- ☐ The methods `update(String name)` and `update(String name, String registrationNumber)` can be called on an instance of `Student`

3.7 CHECKING DATA TYPES WITH „INSTANCEOF“

```
class Example {
    public void doSomething(Object value) {
        // ...
    }
}
```

How to check the type of `value` in method `doSomething`?

In many situations you are given an object reference (e.g. as a parameter) and you need to know what type it is. The `instanceof` operator can help here: it returns `true` if an object reference is of a particular type.

One possible use is in so-called "downcasts".

DOWNCAST EXAMPLE 1

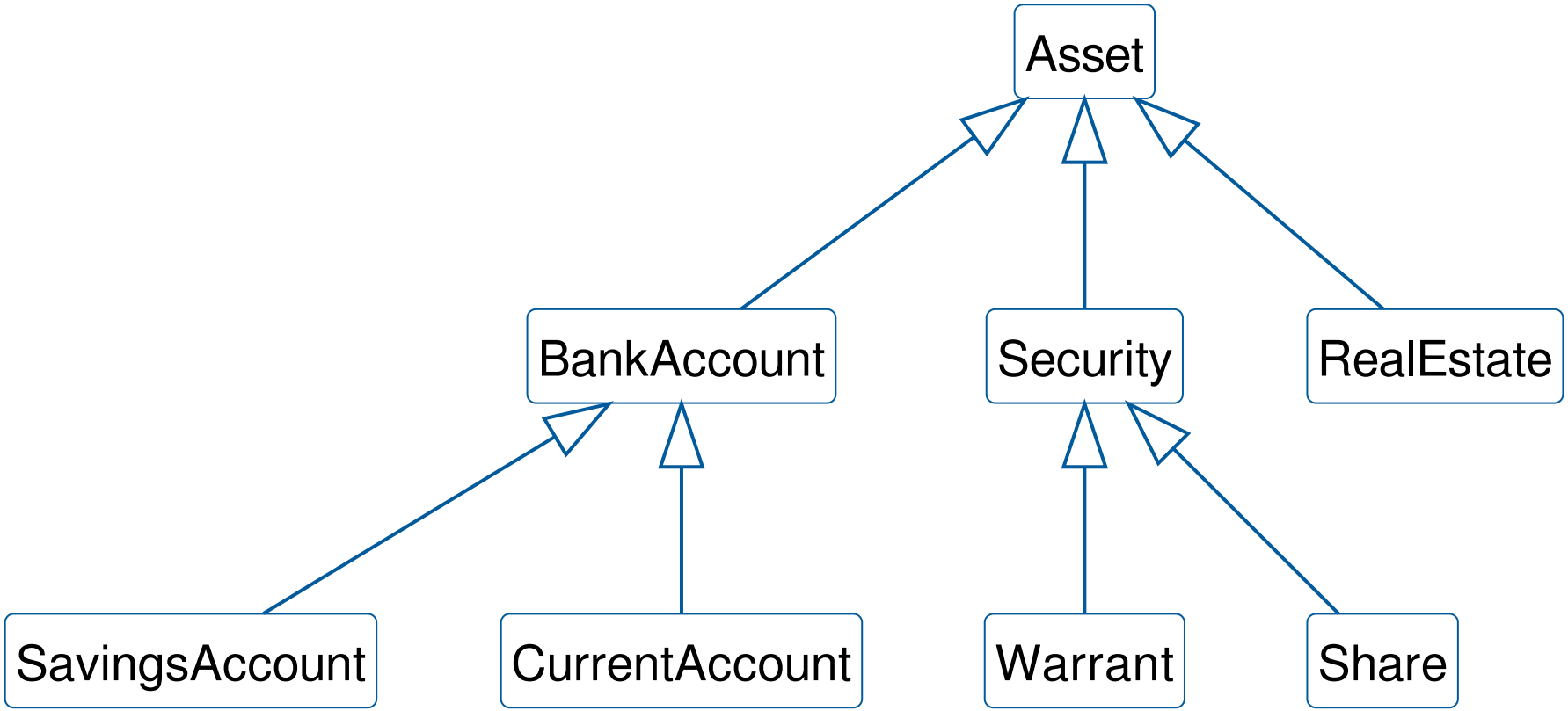
```
Object o = "abc";
if (o instanceof String) {
    /**
     * now it is save to downcast from Object to String!
     */
    String s = (String) o;
    System.out.println(s.length());
}
Square sq = new ColoredSquare(1., "yellow");
System.out.println(sq instanceof Square); // true
System.out.println(sq instanceof ColoredSquare); // true
```

DOWNCAST EXAMPLE 2

```
private String getSquareColor(Square sq) {
    if (sq instanceof ColoredSquare) {
        ColoredSquare coloredSquare = (ColoredSquare) sq;
        return coloredSquare.getColor();
    } else {
        return "unknown";
    }
}
```

EXERCISE: TYPE CHECKING

Evaluate statements with reference to the following inheritance hierarchy.



Which expression yields True?

```
Share share = new Share();
```

- ☐ `share instanceof RealEstate`
- ☐ `share instanceof Warrant`
- ☐ `share instanceof SavingsAccount`
- ☐ `share instanceof Asset`
- ☐ `share instanceof Security`
- ☐ `share instanceof BankAccount`

Which expression yields True?

```
BankAccount account = new BankAccount();
```

- ☐ `account instanceof BankAccount`
- ☐ `account instanceof RealEstate`
- ☐ `account instanceof Security`
- ☐ `account instanceof Asset`
- ☐ `account instanceof Warrant`
- ☐ `account instanceof SavingsAccount`

3.8 THE KEYWORD "FINAL"

The keyword `final` can be used in Java in class, method and attribute definitions to indicate that the respective property can no longer be changed. However, the exact meaning differs.

FINAL IN CLASS DEFINITIONS

```
final class MyClass {
    // ...
}
```

For class definitions, such as `final class MyClass`, the keyword means that `MyClass` can no longer be subclassed. Examples of this are the `String` and `StringBuffer` classes. The 'final' declaration of a class is usually justified by security or performance considerations.

FINAL IN METHOD DEFINITIONS

```
class Account {
    public final boolean checkSecret(int secret) {
        // ...
    }
}
```

Methods marked as `final` may not be overridden in subclasses. This ensures certain functionalities along an inheritance chain.

FINAL IN VARIABLE AND ATTRIBUTE DEFINITIONS

```
final Square sq = new Square(1.9);
sq.setSideLength(2.5); // changes square contents!
sq = null; // not possible since sq is final!
```

Variables and attributes marked as `final` cannot be changed after declaration and simultaneous initialization.

However, it should be noted that the referenced object can change its attribute values through method calls!

EXERCISE: FINAL

Which statements are correct regarding the following code snippet?

```
final class Square {
    final double addArea(final Square sq) {
        // ...
    }
}
```

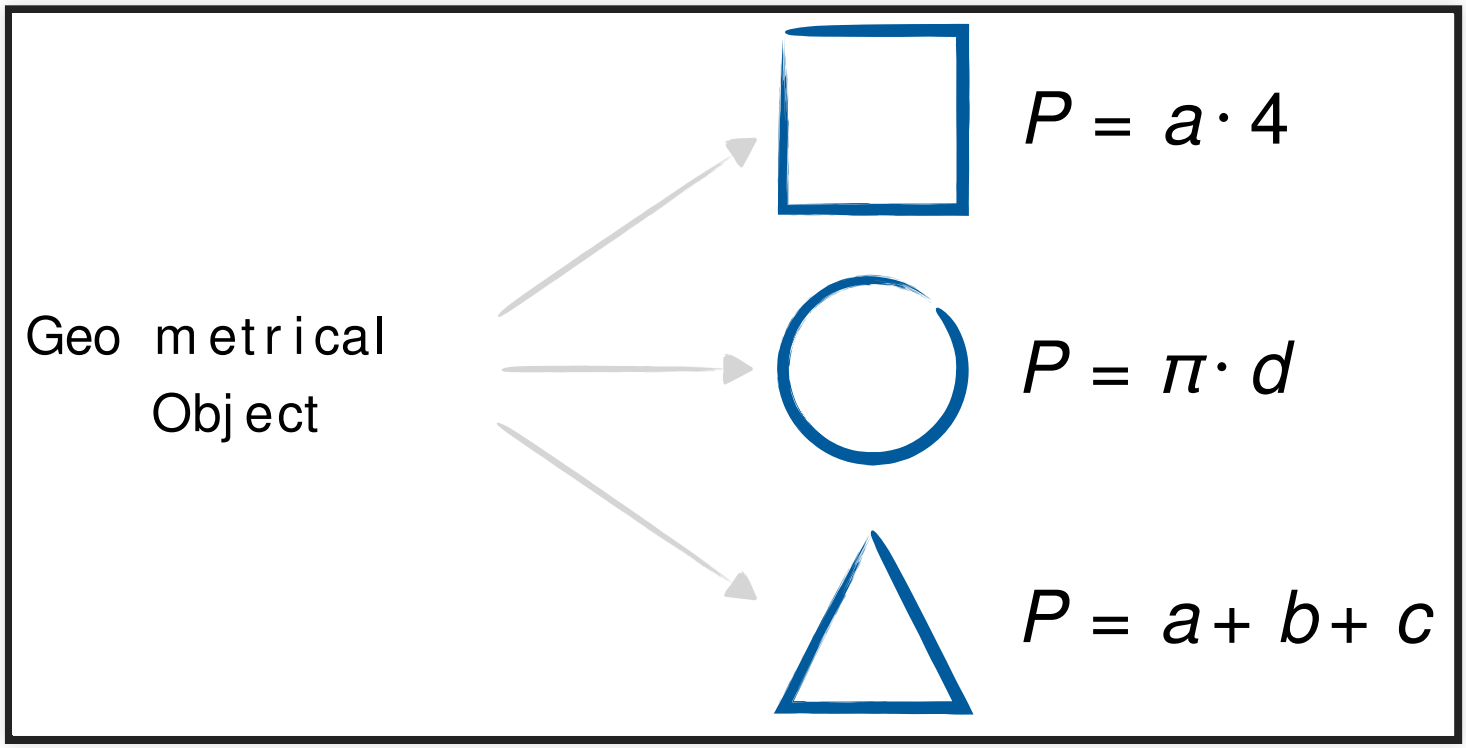
- [ ] The method `addArea` cannot be overridden
- [ ] The method `addArea` cannot be overloaded
- [ ] The parameter `q` can no longer be changed
- [ ] The `addArea` method does not need to calculate anything since the result is already marked with `final`
- [ ] No subclasses can be created from the class `Square`
- [ ] The parameter `q` does not need to be checked because it is already `final`

3.9 ABSTRACT CLASSES

Abstract classes, unlike concrete classes, are not ready to execute - they describe an *abstract* entity that can be reused in concrete classes.

```
public class GeometricObject {
    public double calculatePerimeter() {
        // ... how to calculate?
    }
}
```

As an example, the class `GeometricObject` could be the parent class of class `Square` and calculate the object's perimeter.


Geo metrical
Object
P = a · 4
P = π · d
P = a + b + c

Circles, triangles, squares and rectangles all have perimeters, but the calculation is different in each case. Abstract classes allow us to express that geometric objects can always calculate a perimeter, without knowing how the calculation is performed in each case.

THE KEYWORD "ABSTRACT"

With the keyword `abstract` we mark classes that are abstract and methods that are to be implemented by child classes.

```
public abstract class GeometricObject {
    public abstract double calculatePerimeter();
}
```

This also means that no instance of the class `GeometricObject` can be created with `new`. As a consequence, the implementation of `calculatePerimeter` may still be missing.

However, concreate subclasses need to implement the method to ensure that there is Java code to be executed when the method is called.

Based on GeometricObject a class for squares could be defined as follows:

```
public class Square extends GeometricObject {
    private double sideLength;

    public double calculatePerimeter() {
        return 4 * sideLength;
    }
}
```

... and for rectangles:

```
class Rectangle extends GeometricObject {
    double height;
    double width;

    public double calculatePerimeter() {
        return 2 * (height + width);
    }
}
```

Accordingly the following does not work:

```
GeometricObject o1 = new GeometricObject(); // not ok for abstract class
```

... and the following is possible:

```
GeometricObject o2 = new Square(); // concrete class can be instantiated
GeometricObject o3 = new Rectangle(); // concrete class
```

EXAMPLE STRING INSTRUMENT

Class StringInstrument specifies the attribute `name` and enforces the operation `getNumberOfStrings` from all subclasses. Accordingly, guitar and bass must override the abstract method and provide an implementation.

```
public abstract class StringInstrument {
    protected String name;

    public StringInstrument(String name) {
        this.name = name;
    }
    abstract int getNumberOfStrings();
}
```

Advantage: it is guaranteed that each string instrument can supply the number of strings.

Guitar

```
class Guitar extends StringIntstrument {
    public Guitar(String name) {
        super(name);
    }
    int getNumberOfStrings() {
        return 6;
    }
}
```

Bass Guitar

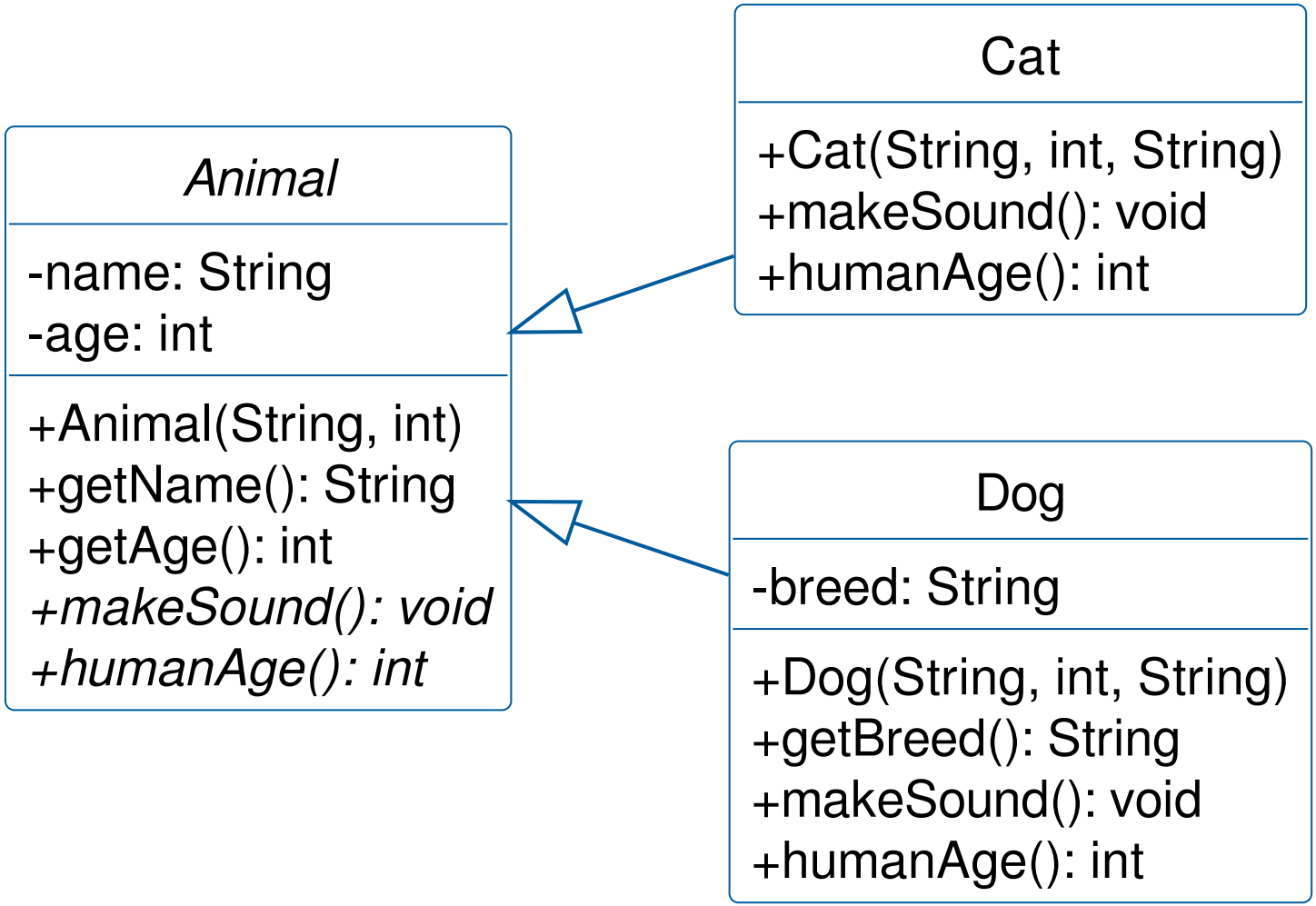
```
class Bass extends StringIntstrument {
    public Bass(String name) {
        super(name);
    }
    int getNumberOfStrings() {
        return 4;
    }
}
```

Application in Main method

```
static public void main(String[] args) {
    System.out.println("Number of strings: " +
        new Gitarre("GL-2NT").getNumberOfStrings()); // Output: 6
    System.out.println("Number of strings: " +
        new Bass("HBZ-2004").getNumberOfStrings()); // Output: 4
}
```

EXERCISE: ANIMAL ABSTRACTION

Animals have both common and different characteristics. In this exercise, a scenario will be implemented with an abstract base class `Animal` and two concrete classes `Dog` and `Cat`.


Animal
-name: String
-age: int
+Animal(String, int)
+getName(): String
+getAge(): int
+makeSound(): void
+humanAge(): int
Cat
+Cat(String, int, String)
+makeSound(): void
+humanAge(): int
Dog
-breed: String
+Dog(String, int, String)
+getBreed(): String
+makeSound(): void
+humanAge(): int

Unzip abstract-animal.zip and import the folder as "Existing Maven Project".

Base class `Animal`:

- Each animal has a name and an age
- The abstract method `makeSound()` outputs the sound of the animal to the console
- The abstract method `humanAge()` returns the human age of the animal

Class `Dog`:

- A dog also has an attribute `breed`
- Dogs go `woof!`
- An algorithm for calculating human age can be found, for example, in the if-else exercise in Chapter 1

Class `Cat`:

- Cats say `miao!`
- Calculation of human age: analogous to `Dog`, whereas 1 cat year corresponds to 16 human years, 2 corresponds to 24, then 4 years each

Test your solution with the following main method:

```
public static void main(String[] args) {
     Animal zoe = new Dog("Zoe", 2, "Mongrel");
     Animal paul = new Cat("Paul", 5);
     zoe.makeSound();
     paul.makeSound();
     System.out.printf("Zoe is %d human years old, Paul %d.%n",
             zoe.humanAge(), paul.humanAge());
}
```

To check the output:

```
woof!
miao!
Zoe is 22 human years old, Paul is 36.
```

3.10 DESIGN DECISION: INHERITANCE OR AGGREGATION?

When considering which classes should be defined for a task, you are typically faced with the decision to use *inheritance* or *aggregation*.

Class A has/consists of one/several Class B

```
class A {
    private B b;
    // or:
    private B[] bList;
}
```

Aggregation is preferable when composing one class from others, i.e. when there is a "has another class" or "consists of one or more other classes" relationship. *Aggregation* offers more possibilities than single inheritance and creates less tight dependencies between classes.

Class A is a Class B

```
class A extends B { ... }
```

Inheritance is useful when it comes to specializations, for example

- Person to Student or Lecturer;
- Assets in bank accounts, real estate or stocks;
- Geometric object to square, rectangle, circle or triangle.

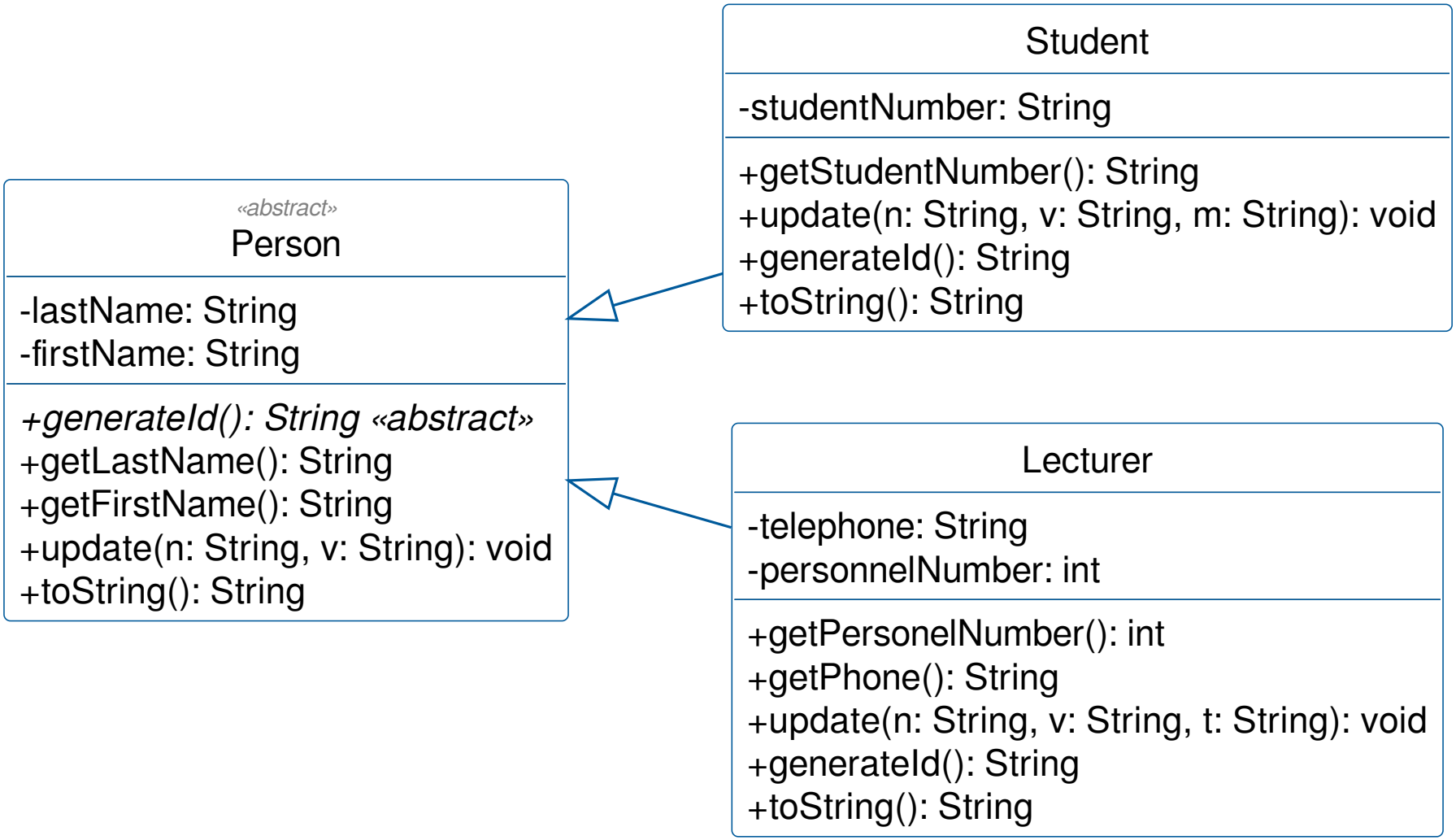
3.11 INHERITANCE EXERCISE

PERSONS, STUDENTS AND LECTURERS

Implement the following UML class structure consisting of `Person`, `Student` and `Lecturer`.

Unzip person-inherite.zip and import the folder as "Existing Maven Project".

CLASS HIERARCHY



Realize class structure in Java

- Create the class `Person`, `Student` and `Lecturer`
- Mark the class `Person` as `abstract`
- Let classes `Student` and `Lecturer` inherit from `Person`
- Add the attributes from the UML class diagram

CONSTRUCTORS

- Add a CTOR `Person` that expects `lastName` and `firstName` of type `String` as parameters
- Add a CTOR `Student` that expects `lastName`, `firstName` and `studentNumber` of type `String` as parameters
- Add a CTOR `Lecturer` that expects `lastName`, `firstName`, `personelNumber` and `telephone` as parameters, only the `personelNumber` is of type `int`, all other parameters are of type `String`
- Assign the parameters to the corresponding attributes in all constructors, call the super constructor if necessary

GETTER

- Add getters in the `Person` class for the `lastName` and `firstName` attributes
- Add getters in the `Student` class for the `studentNumber` attribute
- Add getters in the class `Lecturer` for the attributes `personelNumber` and `telephone`

METHODS

- Add an abstract method `generateId` to the `Person` class
- Implement the `generateId` method in the `Student` and `Lecturer` class:
  - `Student`: should return `<name>-<first name>-<studentNumber>`
  - `Lecturer`: should return `<name>-<firstname>-<personelNumber>`
- Implement the `update` method and assign the parameters accordingly:
  - `Person`: with the parameters `lastName` and `firstName`
  - `Student`: with the parameters `lastName`, `firstName` and `matrikelnummer`
  - `Lecturer`: with the parameters `lastName`, `firstName` and `telephone`

METHOD TOSTRING

- Implement the `toString` method in the `Person` class, return `id=<id>` and use the `generateId` method
- Implement the `toString` method in the `Student` class, return `Student(id=<id>)` and use the `toString` from the `Person` class
- Implement the `toString` method in the `Lecturer` class, return `Lecturer(id=<id>, phone=<phone>)` and use the `toString` from the `Person` class

EXERCISE: OBJECT CREATION

Which statements are correct?

Hint: try it in a Main method!

An instance can be created with...

- [ ] `new Student("Max", "Pattern")`
- [ ] `new Lecturer("Erika", "Pattern")`
- [ ] `new Student("Erika", "Pattern", "xy123")`
- [ ] `new Person("Max", "Pattern")`
- [ ] `new Lecturer("Max", "Pattern", 123, "0123 45677")`

EXERCISE: METHODEN ACCESS

Which of the following calls/statements is correct?

Hint: try it in a Main method!

```
Student s1 = new Student("Erika", "Muster", "xy123");
```

- ☐ `s1.toString()` yields `Student(id=Erika-Muster-xy123)`
- ☐ `s1.update("Erika", "Neumuster", "ab321")`
- ☐ `s1.update("Erika", "Neumuster")`
- ☐ `s1.toString()` yields `Erika-Muster-xy123`
- ☐ `s1.generateId()`

EXERCISE: TERMS

Which statements are correct?

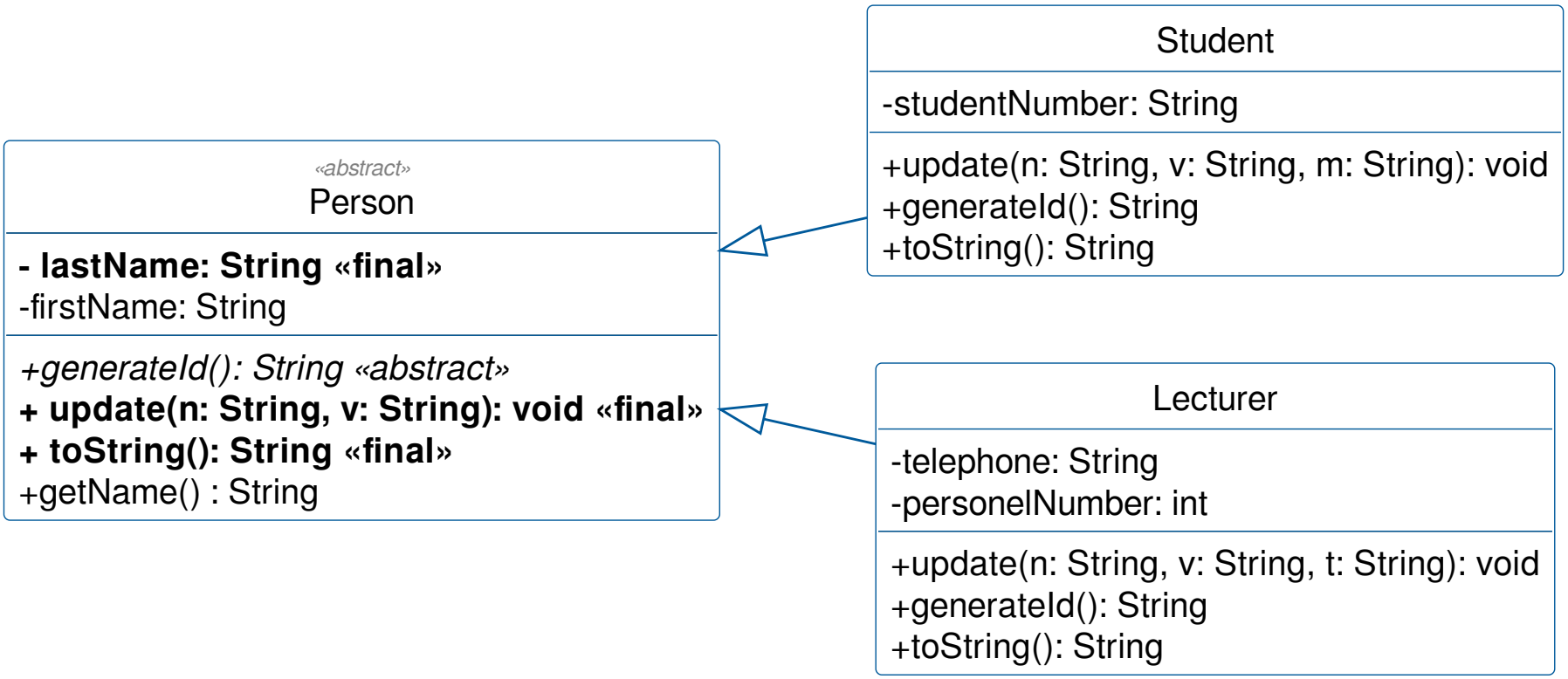
- ☐ The `generateId()` method in the `Person` class is overridden by the `generateId()` method in the `Student` class
- ☐ The `generateId()` method in the `Person` class overloads the `generateId()` method in the `Student` class
- ☐ The constructor in the `Student` class overloads the constructor in the `Person` class
- ☐ The constructor in the `Student` class must call the constructor in the `Person` class with the `super` keyword to chain

Which statements are correct?

- ☐ The method `update(String lastName, String firstName)` in the class `Person` overloads the method `update(String lastName, String firstName, String studentNumber)` in the class `Student`
- ☐ The method `update(String lastName, String firstName)` in the class `Person` is overridden by the method `update(String lastName, String firstName, String studentNumber)` in the class `Student`
- ☐ The `toString()` method in the `Person` class is overridden by the `toString()` method in the `Student` class
- ☐ The `toString()` method in the `Person` class overloads the `toString()` method in the `Student` class

EXERCISE: FINAL KEYWORD

Let's assume the method `toString` and `update` and the attribute `lastName` in the class `Person` become `final`.



Try the changes in your Java program, which statements are true?

- [ ] Overloading of the method `Lecturer.update(String, String, String)` is still possible.
- [ ] Overwriting the method `Student.toString` is no longer possible.
- [ ] Calling `getName()` is no longer possible - because the attribute `name` is `final
- [ ] The method `Person.update(String, String)` shows an error, because the attribute `name` is `final`.
- [ ] Overloading the method `Student.update(String, String, String)` is still possible
- [ ] Overwriting the method 'Lecturer.toString' is still possible.

EXERCISE: INSTANCEOF (1)

Given s, which expressions yield true?

```
Student s = new Student("Muster", "Erika", "xy123");
```

- ☐ `s instanceof Person`
- ☐ `s instanceof Lecturer`
- ☐ `s instanceof Student`
- ☐ `s instanceof Object`

EXERCISE: INSTANCEOF (2)

Given p, which expressions yield true?

```
Person p = new Lecturer("Muster", "Max", 123, "01234567890");
```

- ☐ `p instanceof Student`
- ☐ `p instanceof Person`
- ☐ `p instanceof Lecturer`
- ☐ `p instanceof Object`