Introduction to Programming

04 Object Orientation I

Stephan Krusche

22 November 2023
Technical University of Munich

Schedule

#	Date	Subject
1	18.10.23	Introduction
1b	25.10.23	Central exercise
	01.11.23	**No lecture**
2	08.11.23	Control Structures
3	15.11.23	Data Types
4	**22.11.23**	**Object Orientation I**
5	29.11.23	Object Orientation II
6	06.12.23	Object Orientation III
7	13.12.23	Algorithms
	20.12.23	**No lecture**
8	10.01.24	Programming Languages
9	17.01.24	Graphical User Interfaces
10	24.01.24	Recursion
11	31.01.24	Beyond Programming
12	07.02.24	Course Review

Roadmap of today’s lecture

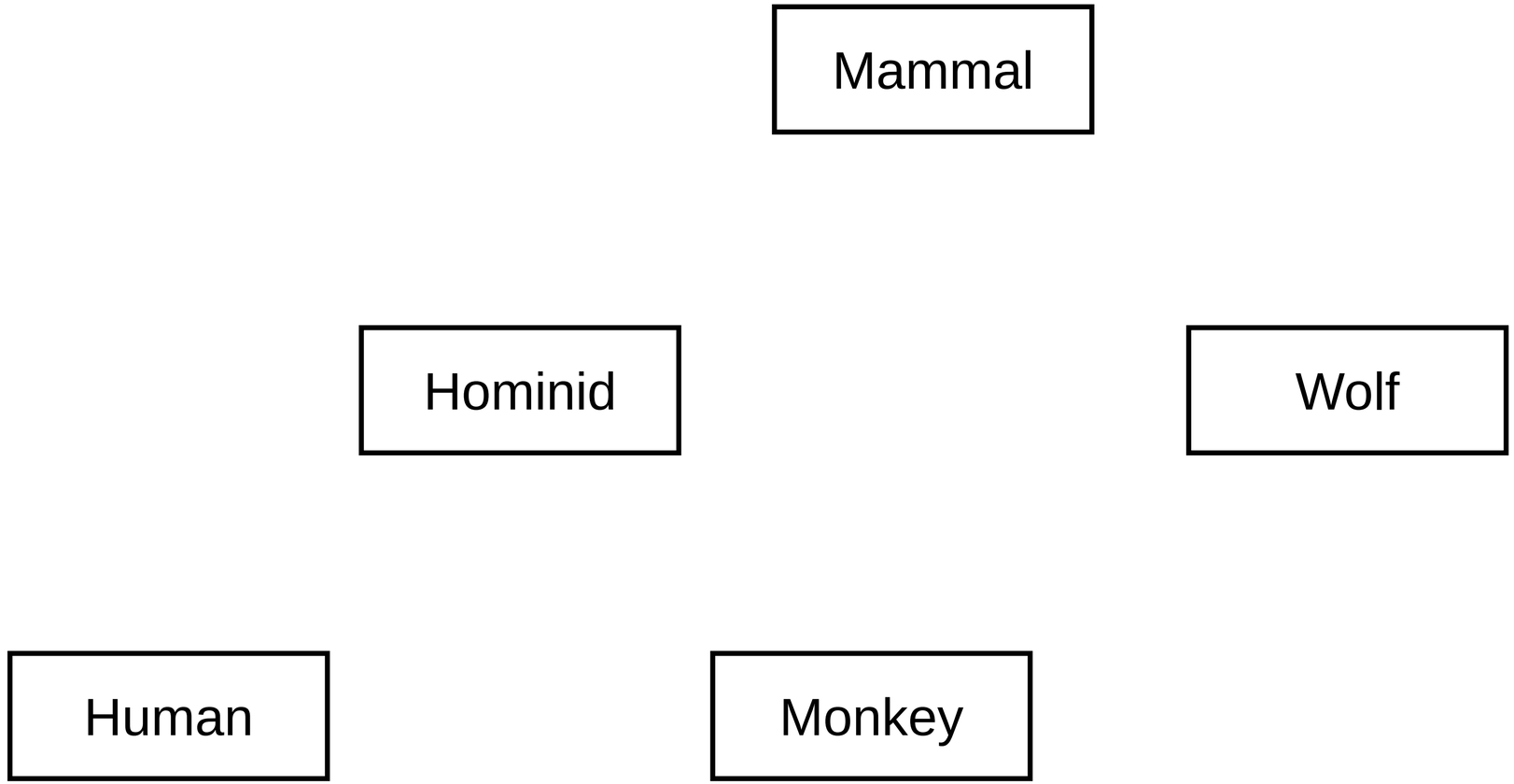
- **Context**
  - Apply the basics of object oriented programming
  - Use basic control structures (**`if`**, **`switch`**, **`for`**, **`while`**)
  - Implement and use basic data types (**`List`**, **`Stack`**, **`Queue`**)
- **Learning goals**
  - Explain the differences between **overriding** and **overloading**
  - Implement **inheritance** taxonomies with **abstract classes** and **interfaces**
  - Explain the concept of **polymorphism** and how it is used in Java (as an example)
  - Explain the difference between **compile time** type and **runtime** type

Outline

- **Inheritance (part 1)**
- Inheritance (part 2)
- Abstract classes and interfaces
- Polymorphism

Observation

- Often, several classes of objects are needed, which are similar but different

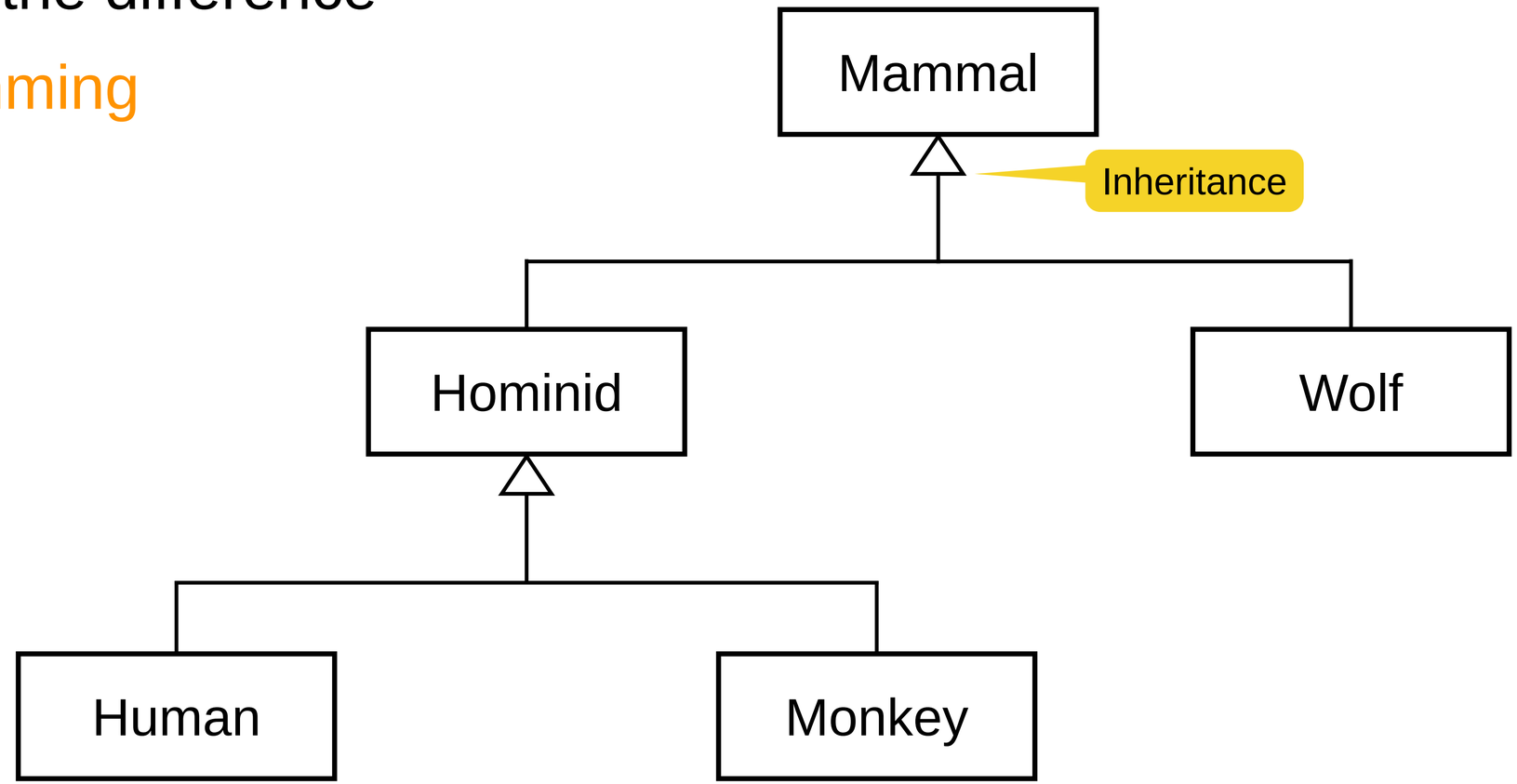


Idea

- Find out commonalities and organize them in a hierarchy
- Implement first what is common to all
- Then implement only the difference

→ Incremental programming

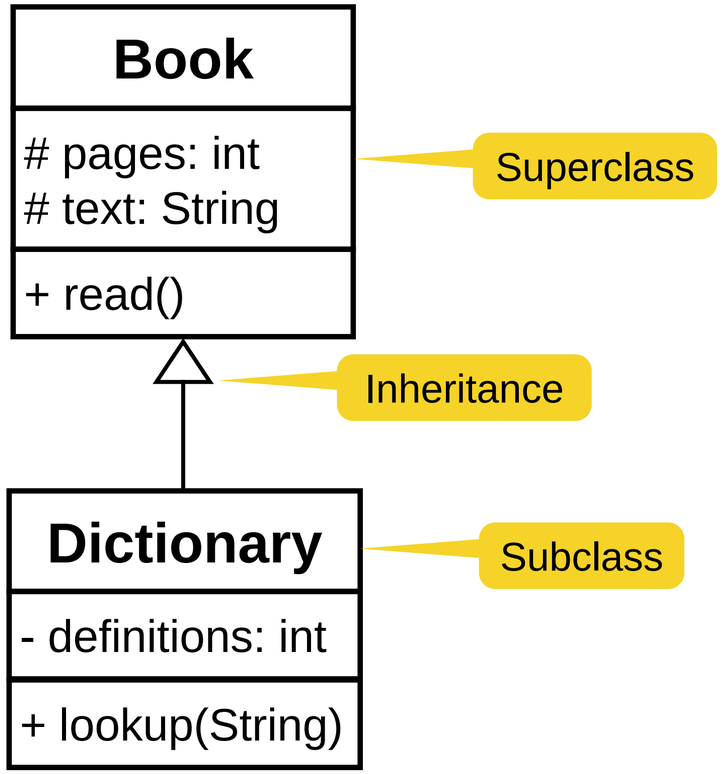
→ Software reuse



Inheritance

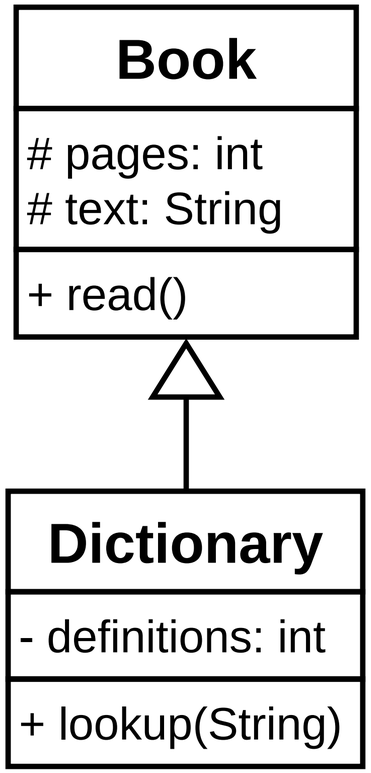
Attributes + constructors + methods

- The subclass has the members of the superclass and possibly additional ones as well
- Transfer of members of the superclass into the subclass is called inheritance
- Example



Implementation

Directly accessible by subclasses




```java
public class Book {
    protected int pages;
    protected String text;

    public Book(int pages, String text) {
        this.pages = pages;
        this.text = text;
    }

    public void read() {
        System.out.print(text);
    }
}
```

`extends` realizes inheritance in Java

```java
public class Dictionary extends Book {
    private int definitions;

    public Dictionary(int pages, String text, int definitions) {
        super (pages, text);
        this.definitions = definitions;
    }

    public void lookup(String word) {
        if(text.contains(word)) {
            System.out.println("Definition for  " + word + ": " + " ... ");
        }
    }
}
```

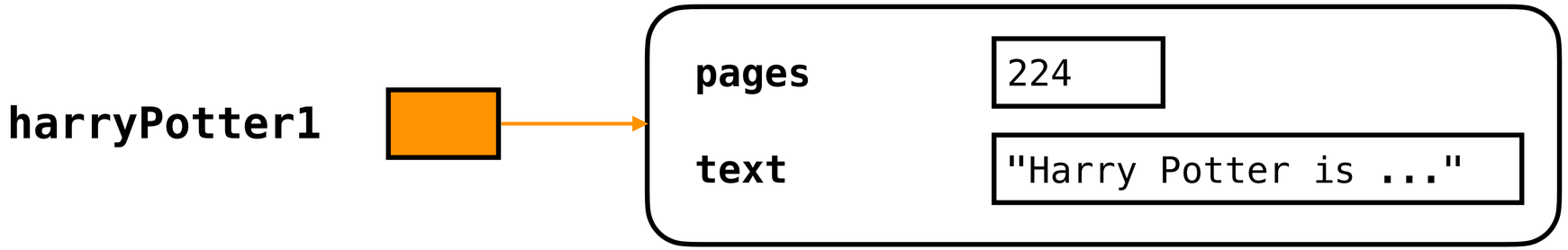
`super(...)` calls the constructor of the superclass

Comments

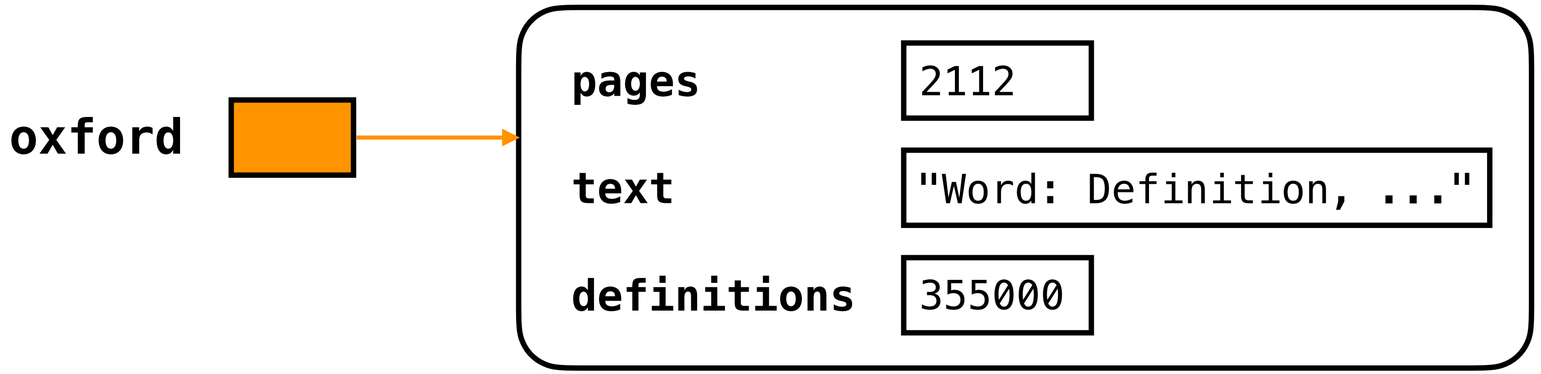
- `class Sub extends Super { ... }` declares class `Sub` as a subclass of class `Super`
  - All members of `Super` are thus automatically available to `Sub` as well
  - Members classified as **`protected`** are visible in the subclass
  - Members declared as **`private`** **cannot** be called directly (they are not visible)
- If a constructor of `Sub` is called, the constructor **`Super()`** of `Super` is implicitly called first if a default constructor is available
  - Otherwise, a constructor has to be called (like in the example)

Usage

```
Book harryPotter1 = new Book(224, "Harry Potter is ...");
```




```
Dictionary oxford = new Dictionary(2112, "Word: Definition, ...", 355000);
```



The keyword **`super`**

- Sometimes, it is necessary to explicitly call the constructors or object methods of the superclass within the subclass
- This is the case when
  - Constructors of the superclass have parameters
  - Object methods or attributes of the superclass and subclass have the same names
- The keyword **`super`** is used to distinguish the current class from the superclass
- **`super(...)`** calls the corresponding constructor of the superclass
  - Similarly, **`this(...)`** allows to call the corresponding constructor of the own class
  - Such an explicit call must always be at the beginning of a constructor

The keyword **`super`**

- **`super.attribute`** accesses an attribute of the superclass
- **`super.method(...)`** accesses an object method of the superclass
- **`super(...)`** invokes a constructor of the superclass
- Example
  - The new object **`oxford`** contains the attributes **`pages`**, **`text`** and **`definitions`** which can be accessed within the class **`Dictionary`**
  - **`super(...)`** invokes the constructor **`Book(...)`** of the Book class
- **`super`** cannot access private attributes, constructors, or methods
- Any other use of **`super`** is **not allowed**

Use of methods

```
Dictionary oxford = new Dictionary(2112, "Word: Definition, ...", 355000);
oxford.read();
oxford.lookup("Computer Science");
```

- The new object **`oxford`** also contains the object methods **`read()`** and **`lookup(String)`**
- Inheritance represents an **`"is a"`** relationship, this means the subclass is also of the type of the superclass, a dictionary **`is a`** (specialized) book

Liskov substitution principle (LSP)

- A superclass object should be replaceable with a subclass object without breaking the functionality of the software
  - **Note:** sometimes, developers do **not** follow this semantic principle and add methods to superclasses, which are not functional in subclasses
  - Then subclasses might need to redefine the method (bad practice!)

* 1939 in Californiia
MIT, ACM Turing Award Laureate 2008
https://amturing.acm.org/award_winners/liskov_1108679.cfm

Visibility and modifiers

Keyword		Accessible to			
		Class	Package	Subclass	World
-	private	yes	no	no	no
	default[1]	yes	yes	no[2]	no
#	protected	yes	yes	yes	no
+	public	yes	yes	yes	yes

[1]: without keyword
[2]: except the subclass is within the same package

→ Means of realization of the principle of **information hiding**

Example Shape

```java
public class Shape {
    protected String color;
    public double area() {
        double area = 0.0;
        // TODO: calculation of the area
        return area;
    }
}

public class Circle extends Shape {

    private final double radius;
    public Circle(double radius) {
        this.radius = radius;
    }

    @Override
    public double area() {
        return Math.PI * radius * radius;
    }
}
```

Cannot be changed after the first assignment of a value

Optional keyword for methods in subclasses that change the implementation of the superclass

L04E02 Simple Inheritance

Not started yet.

Start exercise

Easy

Due date tonight

10 min

3 pts

- Problem statement
  - Create a new subclass `Rectangle` of the superclass `Shape`
    - Implement a constructor based on the `width` and `height` of the rectangle
    - Override the `area()` method with the correct calculation
  - **Optional challenge:** create another subclass `Square`
    - Implement a meaningful constructor
    - **Question:** should `Square` extend `Shape` or `Rectangle`?

Example solution

```java
public class Rectangle extends Shape {
    private final double width;
    private final double height;

    public Rectangle(double width, double height) {
        this.width = width;
        this.height = height;
    }

    @Override
    public double area() {
        return width * height;
    }
}
```

```java
public class Square extends Rectangle {
    public Square (double length) {
        super(length, length);
    }
}
```

Example usage:

```java
Rectangle rectangle = new Rectangle(5, 7);
System.out.println("Rectangle area: " + rectangle.area());

Square square = new Square(5);
System.out.println("Square area: " + square.area());
```

Example solution (with comments)

```java
public class Rectangle extends Shape {
    private final double width;
    private final double height;

    public Rectangle(double width, double height) {
        this.width = width;
        this.height = height;
    }

    @Override
    public double area() {
        return width * height;
    }
}
```

```java
public class Square extends Rectangle {
    public Square (double length) {
        super(length, length);
    }
}
```

A square is a special rectangle

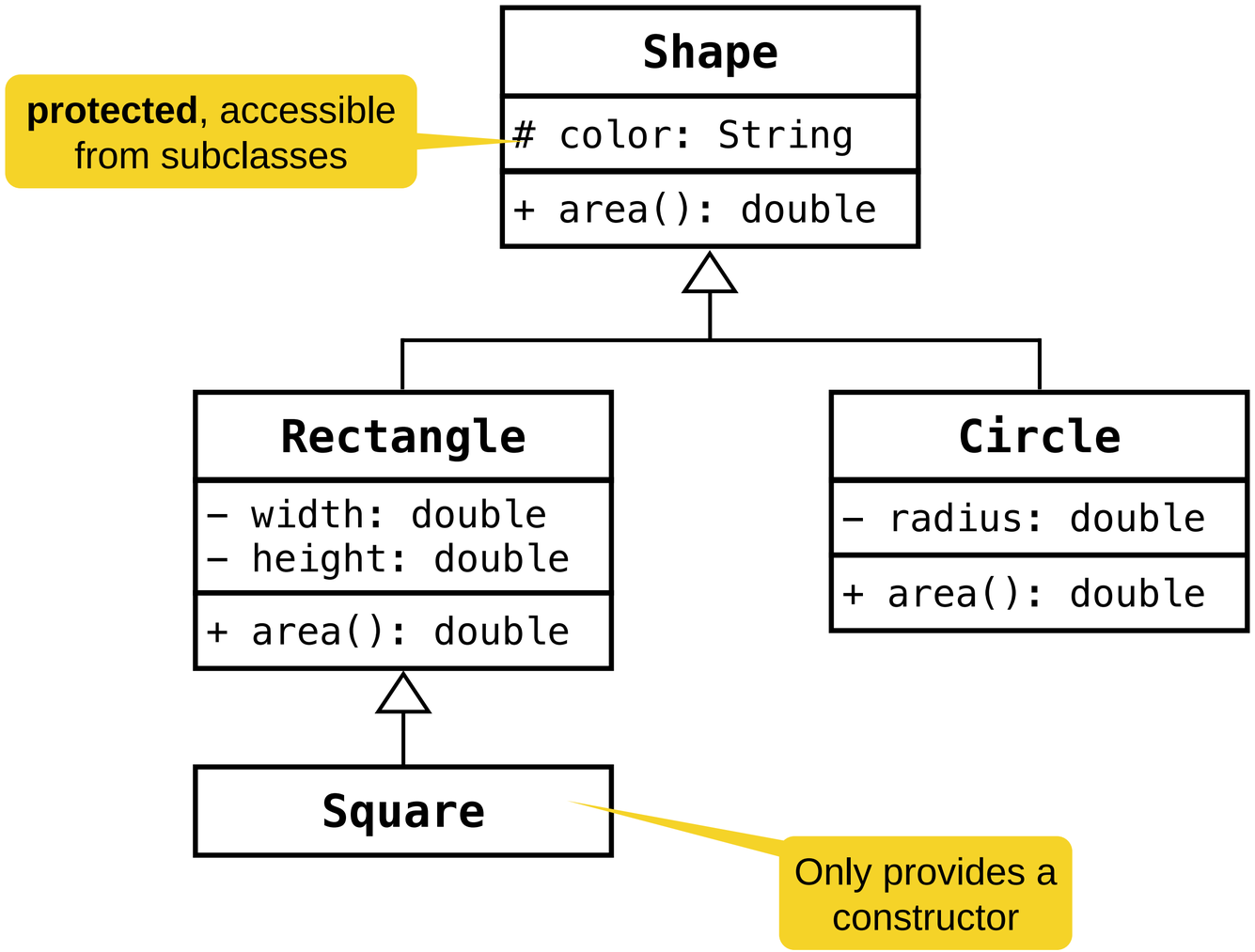
Reuse **`area()`** implementation of **`Rectangle`** in **`Square`** → no need to override

Simply call the constructor of **`Rectangle`**

Example usage:

```java
Rectangle rectangle = new Rectangle(5, 7);
System.out.println("Rectangle area: " + rectangle.area());

Square square = new Square(5);
System.out.println("Square area: " + square.area());
```

Example solution: model



Break

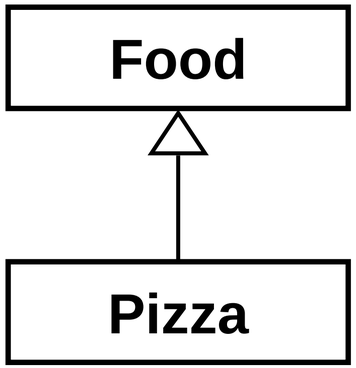
10 min

The lecture will continue at **15:00**

Outline

- Inheritance (part 1)
- **Inheritance (part 2)**
- Abstract classes and interfaces
- Polymorphism

Example `Eating`



The program **`Eating`** should output the number of calories per meal

Example Eating

```java
public class Food {
    private final int CALORIES_PER_GRAM = 9;
    private final int fat;
    private final int servings;

    public Food (int numFatGrams, int numServings) {
        fat = numFatGrams;
        servings = numServings;
    }
    private int calories() {
        return fat * CALORIES_PER_GRAM;
    }
    public int caloriesPerServing() {
        return (calories() / servings);
    }
}
```

Constant

Hidden for Pizza objects

Hidden for Pizza objects

```java
public class Pizza extends Food {

    public Pizza(int amountFat) {
        super(amountFat, 8);
    }
}
```

Has all the members of the superclass **Food** - although not all are directly accessible

```java
public class Eating {
    public static void main(String[] args) {
        Pizza special = new Pizza(275);
        System.out.print("Calories per serving: " + special.caloriesPerServing());
    }
}
```

Nevertheless, the private attributes can be used by the public object method **caloriesPerServing** (why?)

Program output: **Calories per serving: 309**

Binding and scope

- **Binding**: connect an **expression** (or value) to a variable or attribute **name**
- **Scope**: part of the program where the name of the binding is valid
  - In Java, scopes are bounded by curly braces
- Examples of scopes

Formal parameter	Within the entire method
Local variable	From declaration to end of block
Instance and class variable	Anywhere in class, accessibility varies
Block	Only inside the block (like if-statements, loops)
Loop Variable	Only within the loop
Method	Entire method body
Class	Throughout the class
Package	Available within the same package
Global	Public static variables/methods, accessible application-wide

Scopes

Scope for the 3 attributes of **Queue**

```
public class Queue {
    private int first, last;
    private int[] array;

    // ...

    public void enqueue (int x) {
        if (first == -1) {
            first = 0;
            last = 0;
        } else {
            int length = array.length;
            last = (last + 1) % length;
            if (last == first) {
                // queue full
                int[] newArray = new int[2 * length];
                for (int i = 0; i < length; i++) {
                    newArray[i] = array[(first + i) % length];
                }
                first = 0;
                last = length;
                array = newArray;
            }
        }
        array[last] = x;
    }
}
```

Scope for the counter **i**

Scope for the parameter **x**

The lifetime and scope of a binding

- **Multiple declarations**: an identifier can be declared and bound to different values multiple times
- **Lifetime vs. scope**
  - **Lifetime of a binding**: the time during which the binding is valid
  - **Scope of a binding**: the regions of the program where the binding is visible
- **Shadowing**
  - When a **new declaration** of an identifier hides an **existing one** within its scope
  - The original binding still exists but is temporarily obscured

Example: attribute shadowing

```
class Person {
    protected int age;
    public int getAge() {
        return age; // relates to Person class
    }
}
class Child extends Person {
    public int age;
    public void setAge(int theAge) {
        age = theAge; // relates to Child class
    }
    public void setPersonAge(int theAge) {
        super.age = theAge;
    }
}
public class Playground {
    public static void main(String[] args) {
        Child child = new Child();
        child.setAge(2);
        child.setPersonAge(3);
        System.out.println(child.getAge() + " – " + child.age);
    }
}
```

When attributes of the **same name and type** are redefined in a child class, they **shadow** the attribute of the person class

Used to bypass shadowing

Program output: **3 – 2**

Example: method shadowing

```
class Person {
    protected int age;
    public int getAge() {
        return age;          // this person class
    }
}
class Child extends Person {
    public int age;
    public void setAge(int theAge) {
        age = theAge;        // relates to this class
    }
    public int getAge() {
        return age;          // age of this class
    }
    public void setPersonAge(int theAge) {
        super.age = theAge;
    }
}
public class Playground {
    public static void main(String[] args) {
        Child child = new Child();
        child.setAge(2);
        child.setPersonAge(3);
        System.out.println(child.getAge() + " – " + child.age);
    }
}
```

When methods of the **same name and arguments** are redefined in a child class, they **shadow** the method of the person class

Program output: **2 – 3**

Example: attribute shadowing: different types

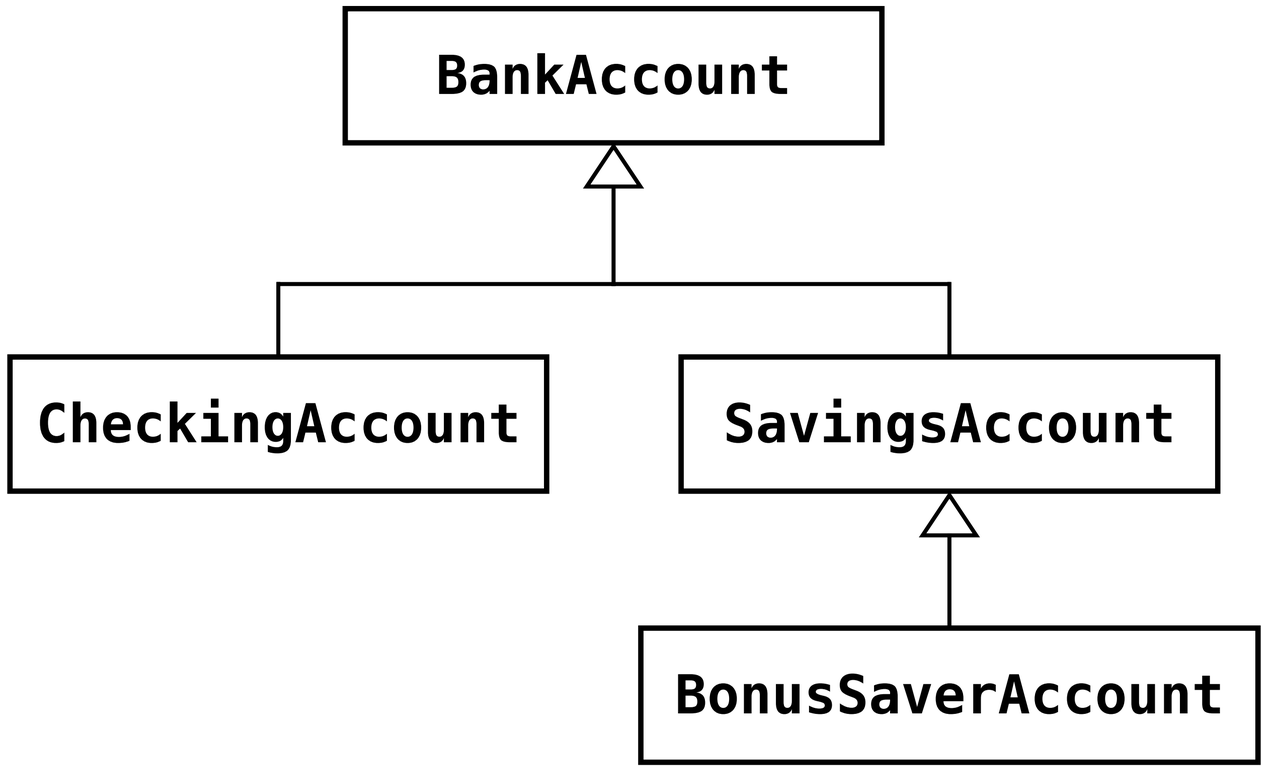
```java
class Person {
    int age;
}
class Child extends Person {
    float age;
    void setAge(float theAge) {
        age = theAge;         // age is this.age
        super.age = (int) theAge;
    }
    float getAge() {
        System.out.println("age in person: " + super.age);
        return age;           // age of this class
    }
}
public class Playground {
    public static void main(String[] args) {
        Child child = new Child();
        child.setAge(17.3f);
        float ageInChild = child.getAge();
        System.out.println("age in child: " + ageInChild);
    }
}
```

Program output:

```
age in person: 17
age in child: 17.3
```

Overriding of methods - example

- Implementation of mutually derived forms of bank accounts
- Each account can be set up, allowed deposits and withdrawals
- Different accounts behave differently in terms of interest and cost of account movements



BankAccount

```
public class BankAccount {
    protected final int accountId;
    protected double balance;

    public BankAccount(int accountId, double initial) {
        this.accountId = accountId;
        balance = initial;
    }

    public void deposit(double amount) {
        balance = balance + amount;
        System.out.println("Deposit into account " + accountId + "\n" + "Amount:\t\t\t"
                + amount + "\n" + "New balance:\t" + balance + "\n");
    }

    public boolean withdraw(double amount) {
        System.out.println("Withdrawal from account " + accountId + "\n" + "Amount:\t\t\t" + amount);
        if (amount > balance) {
            System.out.println("Sorry, insufficient funds...\n");
            return false;
        }
        balance = balance - amount;
        System.out.println("New balance:\t" + balance + "\n");
        return true;
    }
}
```

BankAccount (with comments)

```
public class BankAccount {
    protected final int accountId;
    protected double balance;

    public BankAccount(int accountId, double initial) {
        this.accountId = accountId;
        balance = initial;
    }

    public void deposit(double amount) {
        balance = balance + amount;
        System.out.println("Deposit into account " + accountId + "\n" + "Amount:\t\t\t"
                + amount + "\n" + "New balance:\t" + balance + "\n");
    }

    public boolean withdraw(double amount) {
        System.out.println("Withdrawal from account " + accountId + "\n" + "Amount:\t\t\t" + amount);
        if (amount > balance) {
            System.out.println("Sorry, insufficient funds...\n");
            return false;
        }
        balance = balance - amount;
        System.out.println("New balance:\t" + balance + "\n");
        return true;
    }
}
```

The attributes are **protected**, i.e. can only be accessed by object methods of the class or its subclasses

Creating a new **BankAccount** stores a new account number and an initial deposit

Puts money on the account, i.e. modifies the value of balance and prints the account movement

Makes a payout: if the payout fails, it prints a message; the **return** value specifies whether the payout was successful

SavingsAccount

```java
public class SavingsAccount extends BankAccount {

    protected final double interestRate;

    public SavingsAccount(int accountId, double initial, double rate) {
        super(accountId, initial);
        interestRate = rate;
    }

    public void addInterest() {
        balance = balance * (1 + interestRate);
        System.out.println("Interest added to account: " + accountId
                + "\nNew balance:\t" + balance + "\n");
    }
}
```

SavingsAccount (with comments)

```
public class SavingsAccount extends BankAccount {

    protected final double interestRate;

    public SavingsAccount(int accountId, double initial, double rate) {
        super(accountId, initial);
        interestRate = rate;
    }

    public void addInterest() {
        balance = balance * (1 + interestRate);
        System.out.println("Interest added to account: " + accountId
                + "\nNew balance:\t" + balance + "\n");
    }
}
```

Extends the superclass with the additional attribute **`interestRate`** and an object method **`addInterest`**

Additional method (not available in superclass)

All other attributes and object methods are inherited from the superclass

CheckingAccount

```java
public class CheckingAccount extends BankAccount {

    private final SavingsAccount overdraft;

    public CheckingAccount(int accountId, double initial, SavingsAccount savings) {
        super(accountId, initial);
        overdraft = savings;
    }

    @Override
    public boolean withdraw(double amount) {
        if (!super.withdraw(amount)) {
            System.out.println("Using overdraft...");
            if (!overdraft.withdraw(amount - balance)) {
                System.out.println("Overdraft source insufficient.\n");
                return false;
            }
            else {
                balance = 0;
                System.out.println("New balance on account " + accountId + ": 0\n");
            }
        }
        return true;
    }
}
```

CheckingAccount (with comments)

```
public class CheckingAccount extends BankAccount {

    private final SavingsAccount overdraft;

    public CheckingAccount(int accountId, double initial, SavingsAccount savings) {
        super(accountId, initial);
        overdraft = savings;
    }

    @Override
    public boolean withdraw(double amount) {
        if (!super.withdraw(amount)) {
            System.out.println("Using overdraft...");
            if (!overdraft.withdraw(amount - balance)) {
                System.out.println("Overdraft source insufficient.\n");
                return false;
            }
            else {
                balance = 0;
                System.out.println("New balance on account " + accountId + ": 0\n");
            }
        }
        return true;
    }
}
```

Improves a normal account by drawing on the reserve of a savings account (called overdraft) in case of debt

The method **`deposit(...)`** is inherited

Redefined

Withdrawal occurs as a **side effect** when testing the if condition by calling the method of the superclass

If **`withdraw` fails**, the difference is withdrawn from the overdraft account

If this also **fails**, there is no account movement, but an error message

Otherwise, the current account **balance drops to 0** and the reserve is decreased

BonusSaverAccount

```java
public class BonusSaverAccount extends SavingsAccount {

    private final int penalty;
    private final double bonus;

    public BonusSaverAccount(int accountId, double initial, double rate) {
        super(accountId, initial, rate);
        penalty = 25;
        bonus = 0.03;
    }

    @Override
    public boolean withdraw(double amount) {
        System.out.println("Penalty incurred:\t" + penalty);
        return super.withdraw(amount + penalty);
    }

    @Override
    public void addInterest() {
        balance = balance * (1 + interestRate + bonus);
        System.out.println("Interest added to account: "
                + accountId + "\nNew balance:\t" + balance + "\n");
    }
}
```

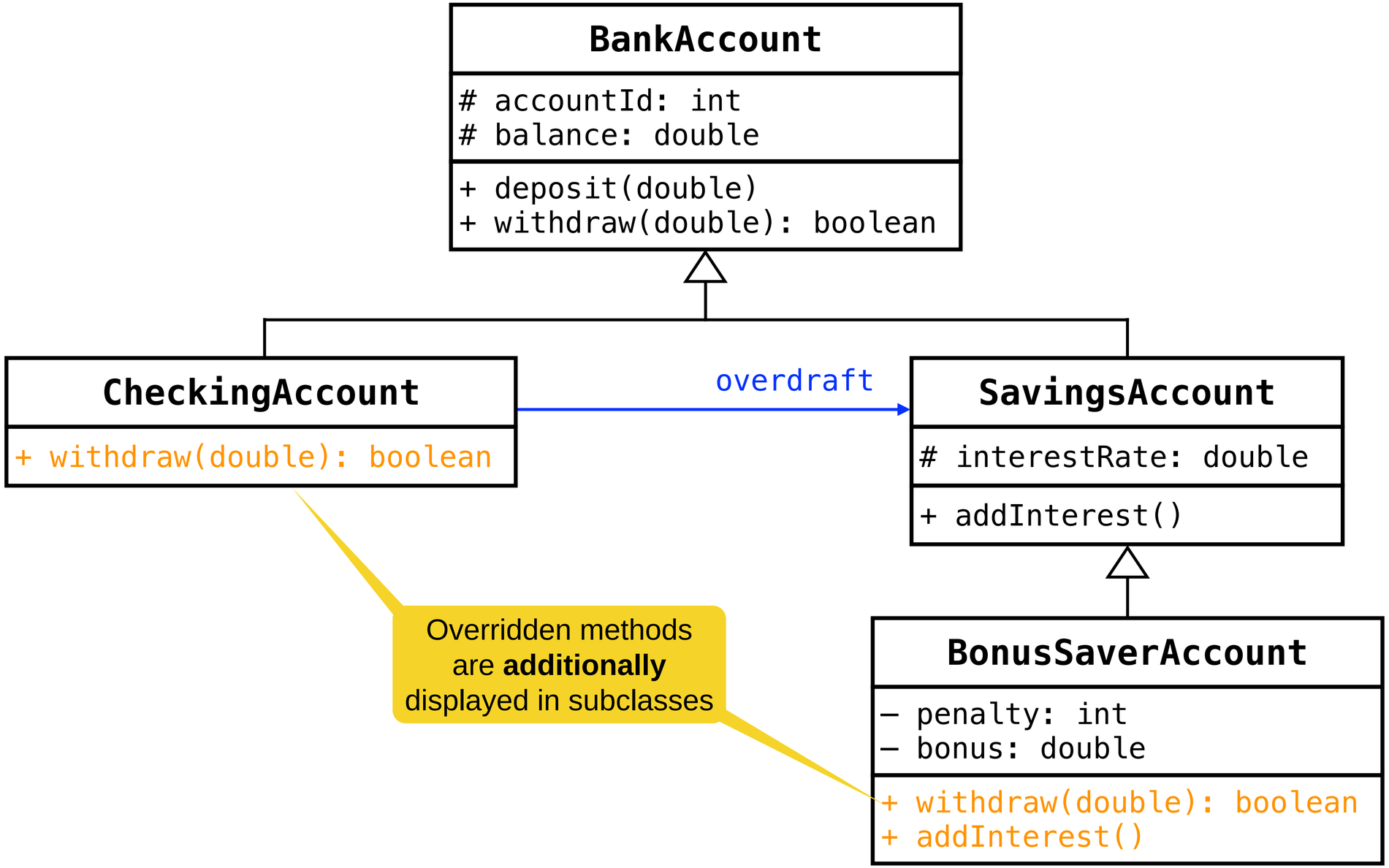
BonusSaverAccount (with comments)

```java
public class BonusSaverAccount extends SavingsAccount {

    private final int penalty;
    private final double bonus;

    public BonusSaverAccount(int accountId, double initial, double rate) {
        super(accountId, initial, rate);
        penalty = 25;
        bonus = 0.03;
    }

    @Override
    public boolean withdraw(double amount) {
        System.out.println("Penalty incurred:\t" + penalty);
        return super.withdraw(amount + penalty);
    }

    @Override
    public void addInterest() {
        balance = balance * (1 + interestRate + bonus);
        System.out.println("Interest added to account: "
                + accountId + "\nNew balance:\t" + balance + "\n");
    }
}
```

Additionally increases the interest rate, but introduces penalty charges for withdrawal

Small change for quite some duplicated code.
Challenge: can you change the implementation to avoid overriding this method?

Refined UML class diagram



Example usage: Bank

```java
public class Bank {

    public static void main(String[] args) {
        // ID 4321, initial sum 5028.45, 2% interest
        SavingsAccount savings =
                new SavingsAccount(4321, 5028.45, 0.02);
        // ID 6543, initial sum 1475.85, 2% interest
        BonusSaverAccount bonusSaver =
                new BonusSaverAccount(6543, 1475.85, 0.02);
        // ID 9876, initial sum 269.93, belongs to savings
        CheckingAccount checking =
                new CheckingAccount(9876, 269.93, savings);
        savings.deposit(148.04);
        bonusSaver.deposit(41.52);
        savings.withdraw(725.55);
        bonusSaver.withdraw(120.38);
        checking.withdraw(320.18);
    }
}
```

Example output:

```
Deposit into account 4321
Amount:         148.04
New balance:    5176.49

Deposit into account 6543
Amount:         41.52
New balance:    1517.37

Withdrawal from account 4321
Amount:         725.55
New balance:    4450.94

Penalty incurred:  25
Withdrawal from account 6543
Amount:         145.38
New balance:    1371.9899999999998

Withdrawal from account 9876
Amount:         320.18
Sorry, insufficient funds...

Using overdraft...
Withdrawal from account 4321
Amount:         50.25
New balance:    4400.69

New balance on account 9876: 0
```

L04E03 BankAccount

Not started yet.

Start exercise

Medium

Due date tonight

15 min

4 pts

- Problem statement
  - Create a new subclass of `BankAccount` called `CreditCardAccount`
    - It should keep track of the negative `creditBalance`
    - It should allow customers to pay (new method) until a certain, predefined `limit` (e.g. 1000)
    - **Hint 1:** the **`withdraw`** functionality stays the same
    - **Hint 2:** create an object `creditCard` and invoke the pay method several times to test it
  - **Optional challenge 1:** implement a method `compensate()` that subtracts the negative balance from the amount of the bank account
    - The amount could become negative through this
    - In case of a negative amount, the customer can only use money to pay until the limit (including the negative amount) is reached
  - **Optional challenge 2:** implement a method `handleOverdraftInterest()` that subtracts 5% overdraft interest in case of a negative balance

Example solution

```java
public class CreditCardAccount extends BankAccount {
    private final double limit;
    private double creditBalance = 0;

    public CreditCardAccount(int accountId, double initial, double limit) {
        super(accountId, initial);
        this.limit = limit;
    }

    public boolean pay(double amount) {
        System.out.println("Pay from account " + accountId + "\n" + "Amount:\t\t\t" + amount);
        if (-creditBalance + amount > limit) {
            System.out.println("Sorry, insufficient balance...\n");
            return false;
        }
        creditBalance = creditBalance - amount;
        System.out.println("New credit balance:\t" + creditBalance + "\n");
        return true;
    }
}
```

Example solution (with comments)

```java
public class CreditCardAccount extends BankAccount {
    private final double limit;
    private double creditBalance = 0;

    public CreditCardAccount(int accountId, double initial, double limit) {
        super(accountId, initial);
        this.limit = limit;
    }

    public boolean pay(double amount) {
        System.out.println("Pay from account " + accountId + "\n" + "Amount:\t\t\t" + amount);
        if (-creditBalance + amount > limit) {
            System.out.println("Sorry, insufficient balance...\n");
            return false;
        }
        creditBalance = creditBalance - amount;
        System.out.println("New credit balance:\t" + creditBalance + "\n");
        return true;
    }
}
```

New attributes

If there is **not** enough money

Reduce the amount from the **`creditBalance`**

Example solution (optional challenge 1)

```
public class CreditCardAccount extends BankAccount {
    private final double limit;
    private double creditBalance = 0;

    public CreditCardAccount(int accountId, double initial, double limit) {
        super(accountId, initial);
        this.limit = limit;
    }

    public boolean pay(double amount) {
        System.out.println("Pay from account " + accountId + "\n" + "Amount:\t\t\t\t" + amount);
        if (-creditBalance + amount > limit || -creditBalance - balance + amount > limit) {
            System.out.println("Sorry, insufficient balance...\n");
            return false;
        }
        creditBalance = creditBalance - amount;
        System.out.println("New credit balance:\t" + creditBalance + "\n");
        return true;
    }

    public void compensate() {
        System.out.println("Compensate account " + accountId);
        balance = balance + creditBalance;
        creditBalance = 0;
        System.out.println("New balance:\t\t" + balance);
        System.out.println("New credit balance:\t" + creditBalance);
    }
}
```

Example solution (optional challenge 1, with comments)

```java
public class CreditCardAccount extends BankAccount {
    private final double limit;
    private double creditBalance = 0;

    public CreditCardAccount(int accountId, double initial, double limit) {
        super(accountId, initial);
        this.limit = limit;
    }

    public boolean pay(double amount) {
        System.out.println("Pay from account " + accountId + "\n" + "Amount:\t\t\t\t" + amount);
        if (-creditBalance + amount > limit || -creditBalance - balance + amount > limit) {
            System.out.println("Sorry, insufficient balance...\n");
            return false;
        }
        creditBalance = creditBalance - amount;
        System.out.println("New credit balance:\t" + creditBalance + "\n");
        return true;
    }

    public void compensate() {
        System.out.println("Compensate account " + accountId);
        balance = balance + creditBalance;
        creditBalance = 0;
        System.out.println("New balance:\t\t" + balance);
        System.out.println("New credit balance:\t" + creditBalance);
    }
}
```

Additional check for potential negative balances

Reduce the amount from the **`creditBalance`**

Example solution (optional challenge 2)

```
public void handleOverdraftInterest() {
    System.out.println("Handle overdraft interest for account" + accountId);

    if (balance < 0) {
        double overdraftInterest = 0.05 * -balance;
        System.out.println("Overdraft interest:\t" + overdraftInterest);
        balance = balance - overdraftInterest;
    }
    System.out.println("New balance:\t\t" + balance);
}
```

Example solution (optional challenge 2, with comments)

```
public void handleOverdraftInterest() {
    System.out.println("Handle overdraft interest for account" + accountId);

    if (balance < 0) {
        double overdraftInterest = 0.05 * -balance;
        System.out.println("Overdraft interest:\t" + overdraftInterest);
        balance = balance - overdraftInterest;
    }
    System.out.println("New balance:\t\t" + balance);
}
```

Only when the balance is negative

The interest should be positive…

…so it can be subtracted from the balance

Example usage

```
CreditCardAccount creditCard =
    new CreditCardAccount(7391, 300.0, 1000.00);

creditCard.pay(532.45);
creditCard.pay(467.54);
creditCard.pay(0.01);

creditCard.pay(0.01);

creditCard.compensate();

creditCard.pay(23.01);

creditCard.pay(532.45);

creditCard.handleOverdraftInterest();
```

Program output:

```
Pay from account 7391
Amount:                532.45
New credit balance:    -532.45

Pay from account 7391
Amount:                467.54
New credit balance:    -999.99

Pay from account 7391
Amount:                0.01
New credit balance:    -1000.0

Pay from account 7391
Amount:                0.01
Sorry, insufficient balance...

Compensate account 7391
New balance:           -700.0
New credit balance:    0.0

Pay from account 7391
Amount:                23.01
New credit balance:    -23.01

Pay from account 7391
Amount:                532.45
Sorry, insufficient balance...

Handle overdraft interest for account 7391
Overdraft interest:    35.0
New balance:           -735.0
```

Example usage (with comments)

```
CreditCardAccount creditCard =
    new CreditCardAccount(7391, 300.0, 1000.00);

creditCard.pay(532.45);
creditCard.pay(467.54);
creditCard.pay(0.01);

creditCard.pay(0.01);

creditCard.compensate();

creditCard.pay(23.01);

creditCard.pay(532.45);

creditCard.handleOverdraftInterest();
```

Should succeed

Should fail

Should succeed

Should fail

```
Program output:

Pay from account 7391
Amount:                 532.45
New credit balance:    -532.45

Pay from account 7391
Amount:                 467.54
New credit balance:    -999.99

Pay from account 7391
Amount:                 0.01
New credit balance:    -1000.0

Pay from account 7391
Amount:                 0.01
Sorry, insufficient balance...

Compensate account 7391
New balance:           -700.0
New credit balance:    0.0

Pay from account 7391
Amount:                 23.01
New credit balance:    -23.01

Pay from account 7391
Amount:                 532.45
Sorry, insufficient balance...

Handle overdraft interest for account 7391
Overdraft interest:    35.0
New balance:           -735.0
```

30 min

The lecture will continue at **16:10**

Outline

- Inheritance (part 1)
- Inheritance (part 2)
- **Abstract classes and interfaces**
- Polymorphism

Abstract methods and classes

- An **abstract** object method is a method for which no implementation is provided
- A class that contains **abstract** object methods is also called **abstract**
- No objects can be created for an abstract class
- **Abstract** classes allow to group subclasses with different implementations of the same object methods

Abstract methods and classes

Example: evaluation of expressions

```
public abstract class Expression {

    private int value;
    private boolean evaluated = false;

    public int getValue() {
        if (evaluated) {
            return value;
        } else {
            value = evaluate();
            evaluated = true;
            return value;
        }
    }
    abstract protected int evaluate();
}
```

- Subclasses of **`Expression`** represent different types of expressions
- All subclasses have an object method **`evaluate()`** in common
  - Always with a different implementation
  - Also called specification inheritance

Stores the result in **`value`** and remembers that the expression has already been evaluated

Evaluates the expression

Abstract methods and classes

- An abstract object method is identified by the keyword **`abstract`**
- A class that contains an abstract method must itself also be marked as **`abstract`**
- In an abstract method, the complete signature must be specified - including the parameter types and the (possibly) thrown exceptions
- An abstract class can contain concrete methods
  - In the previous example: **`int getValue()`**

Exceptions are covered in **L05**

Example for an expression: Const

The class is declared as **`final`**: no subclasses of **`Const`** may be declared

```
public final class Const extends Expression {
    private final int constValue;

    public Const(int constValue) {
        this.constValue = constValue;
    }

    @Override
    protected int evaluate() {
        return constValue;
    }
}
```

Final attribute can only be initialized, but **not** modified

Requires an argument which is stored in a private variable

- For security and efficiency reasons, classes should be declared as **final** (if possible)
- Attributes or methods can be declared as **final** → must not be redefined in subclasses
- **Final** variables (constants) may additionally only be initialized, but not be modified

Other expressions

```java
public final class Addition extends Expression {
    private final Expression left, right;

    public Addition(Expression left, Expression right) {
        this.left = left;
        this.right = right;
    }
    @Override
    protected int evaluate() {
        return left.getValue() + right.getValue();
    }
}
```

Example usage:

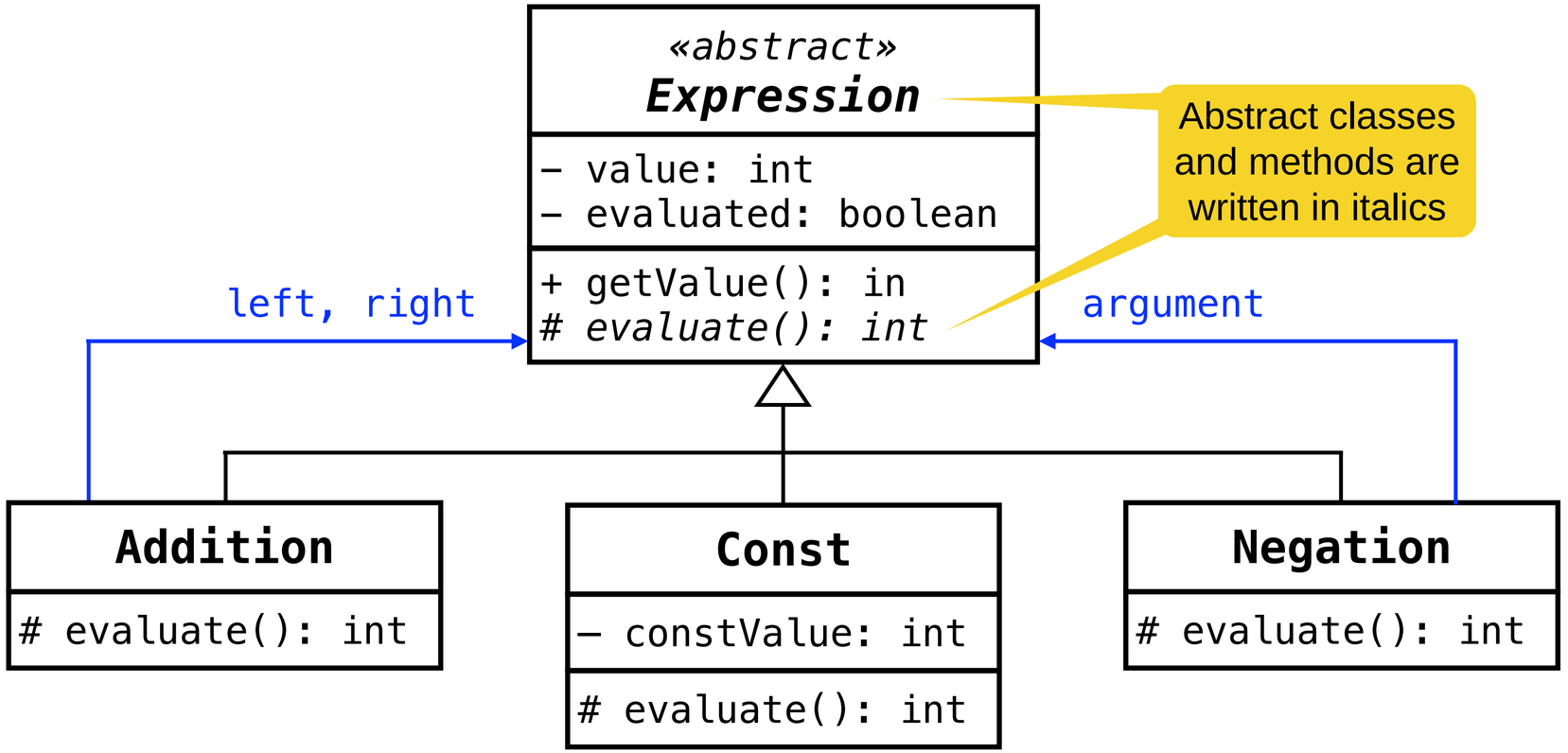
```java
public static void main(String[] args) {
    Expression exp = new Addition(
        new Negation(new Const(8)),
        new Const(16));
    System.out.println(exp.getValue());
}
```

```
Expression: – 8 + 16 = 8
```

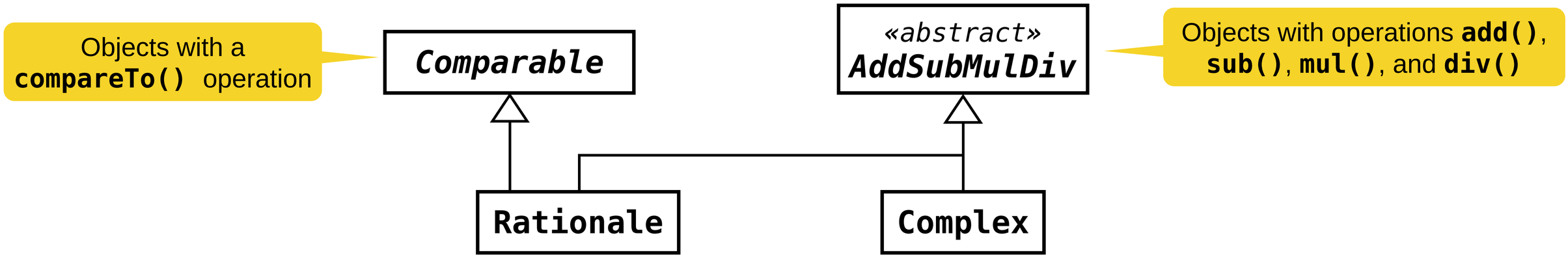
```java
public final class Negation extends Expression {
   private final Expression argument;

    public Negation(Expression argument) {
        this.argument = argument;
    }
    @Override
    protected int evaluate() {
        return -argument.getValue();
    }
}
```

- **`getValue()`** calls **`evaluate()`** successively for each partial expression of **`exp`**
- Which concrete implementation is chosen in each case depends on the concrete class of the respective partial expression, i.e. is decided only **at runtime**
- This is also called **dynamic binding**

UML class diagram



Example



- Multiple direct superclasses of a class lead to conceptual problems
  - To which class does **`super`** refer?
  - Which object method is meant if several superclasses provide an implementation?

→ Multiple inheritance (of classes) is **not** possible in Java

Interfaces

- No problem arises if an object method is **abstract** in all superclasses or at least has an implementation in at most one superclass
- An **interface** can be thought of as an **abstract** class, where
  - All object methods are **abstract** (specification, no implementation)
  - There are no “class” methods (an interface can still provide static methods)
  - All variables are **constants** → there is **no** mutable state

Example

```
public interface Comparable {
    int compareTo(Object x);
}
```

- **`Object`** is the common superclass of all classes (no need to specify it)
- Methods in interfaces are automatically object methods and **`public`**
- A superset of the exceptions thrown in implementations must be specified (covered later!)
- Any constants that occur are automatically **`public static`**

Example

One superclass

Multiple interfaces are possible

Keyword for interfaces

```
public class Rationale extends AddSubMulDiv implements Comparable {

    private long numerator, denominator;

    public int compareTo(Object other) {
        Rationale fraction = (Rationale) other;
        long left = numerator * fraction.denominator;
        long right = denominator * fraction.numerator;
        if (left == right) {
            return 0;
        }
        else if (left < right) {
            return -1;
        }
        else {
            return 1;
        }
    }
}
```

Explanations

- `class A extends B implements C1, C2, ..., Ck` means
  - `A` is a subclass of `B`
  - `A` implements interfaces `C1, C2, ..., Ck`
- Inheritance rules
  - Only one superclass per class
  - Unlimited implemented interfaces
- Interface usage
  - Constants from interfaces can be used in implementing classes
  - Interfaces as **types**: formal parameters, variables, return types
- Interface and objects
  - Instantiated objects of interface types must be from implementing classes
  - Explicit casting to the implementing class is often required (and can be cumbersome)

Interfaces among each other

- Interfaces can extend other interfaces or even combine several other interfaces
- Extending interfaces can redefine constants
- If a constant with the same name occurs in different implemented interfaces **A** and **B**, it can be distinguished by **A.constant** and **B.constant**

Example

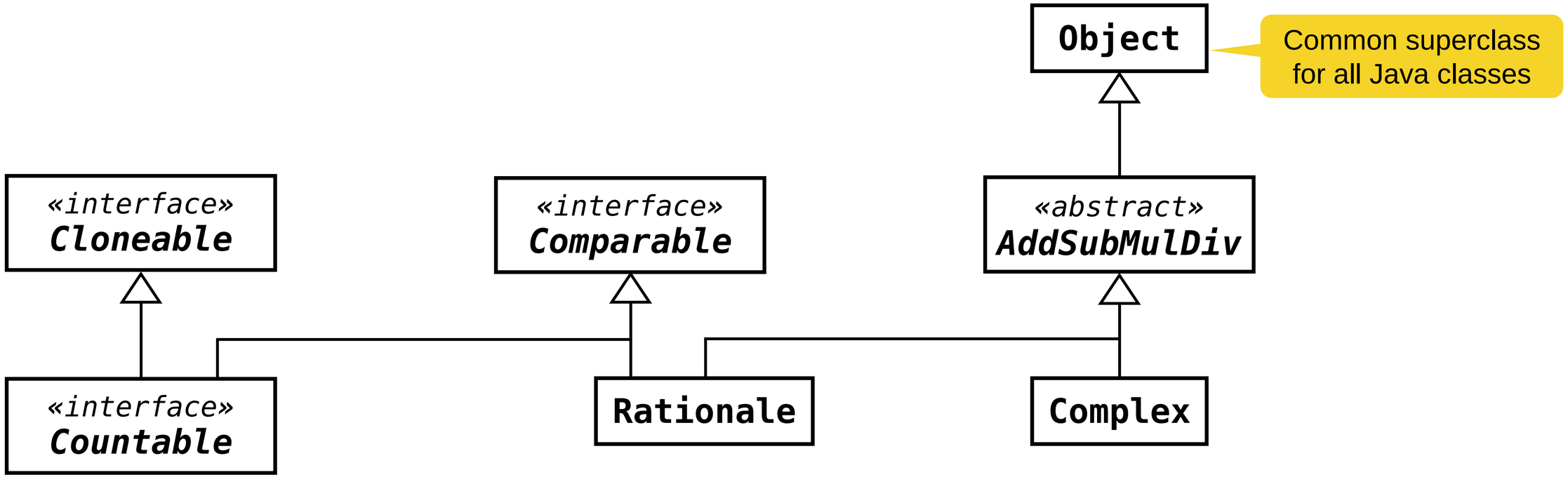
Includes the (predefined) interfaces **Comparable** and **Cloneable**

```
public interface Countable extends Comparable, Cloneable {
    Countable next();
    Countable prev();
    int number();
}
```

Cloneable requires an object method **public Object clone()** which creates a copy of the object

A class that implements **Countable** must implement (or inherit) the object methods **compareTo()**, **clone()**, **next()**, **prev()**, and **number()**

Overview


Object
Common superclass for all Java classes
«interface»
Cloneable
«interface»
Comparable
«abstract»
AddSubMulDiv
«interface»
Countable
Rationale
Complex

Exercise L05E04

10 min

- Problem statement
  - Rewrite the example **`Shape`** (with subclasses **`Circle`**, **`Rectangle`**, **`Square`**) using an abstract class for **`Shape`**
    - Reuse the same code as implemented in from L05E02
  - Add a second alternative (e.g. in a different package) of the example with an interface for `Shape`
  - Discuss with your neighbor which alternative is better in this case

Example solution

```
abstract class Shape {
    protected String color;
    public abstract double area();
}

class Circle extends Shape {
   //...
}

class Rectangle extends Shape {
   //...
}

class Square extends Rectangle {
   //...
}
```

```
interface Shape {
    double area();
}

class Circle implements Shape {
    private String color;
    //...
}

class Rectangle implements Shape {
    private String color;
    //...
}

class Square extends Rectangle {
   //...
}
```

Example solution (with comments)

Change

Change

```
abstract class Shape {
    protected String color;
    public abstract double area();
}

class Circle extends Shape {
   //...
}

class Rectangle extends Shape {
   //...
}

class Square extends Rectangle {
   //...
}
```

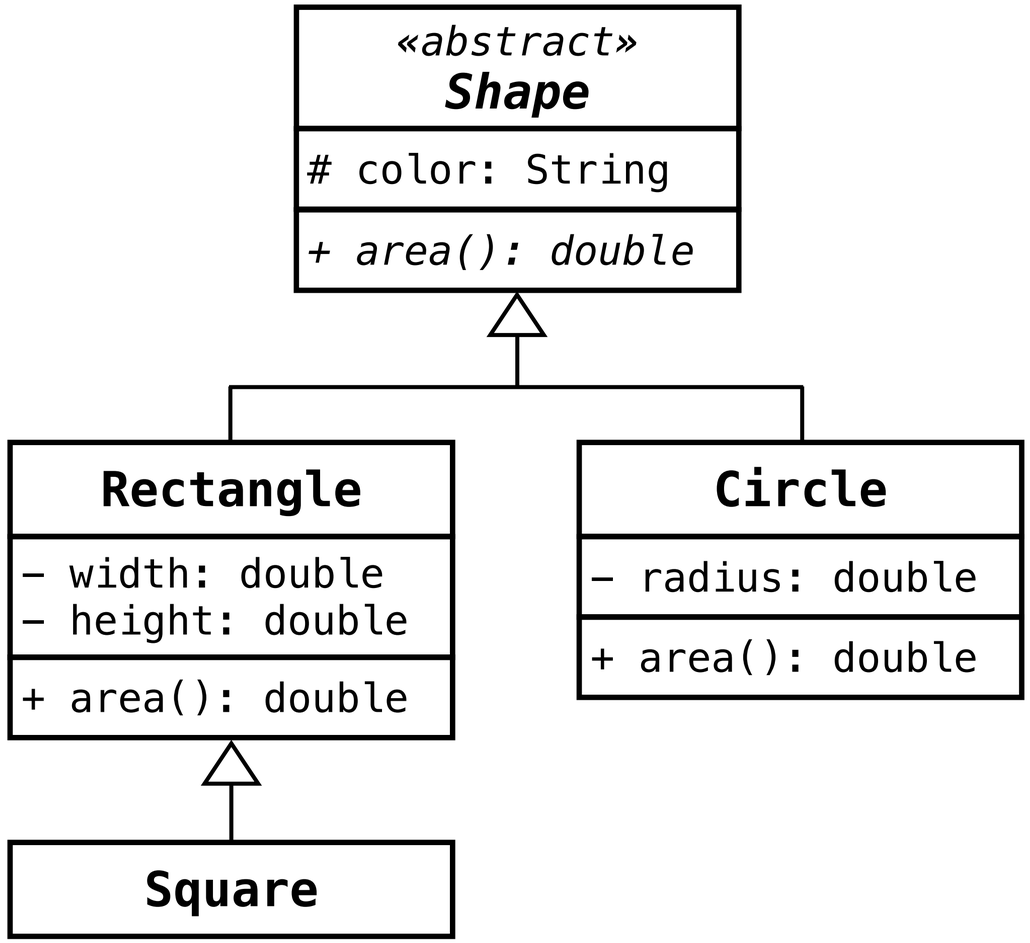
Change

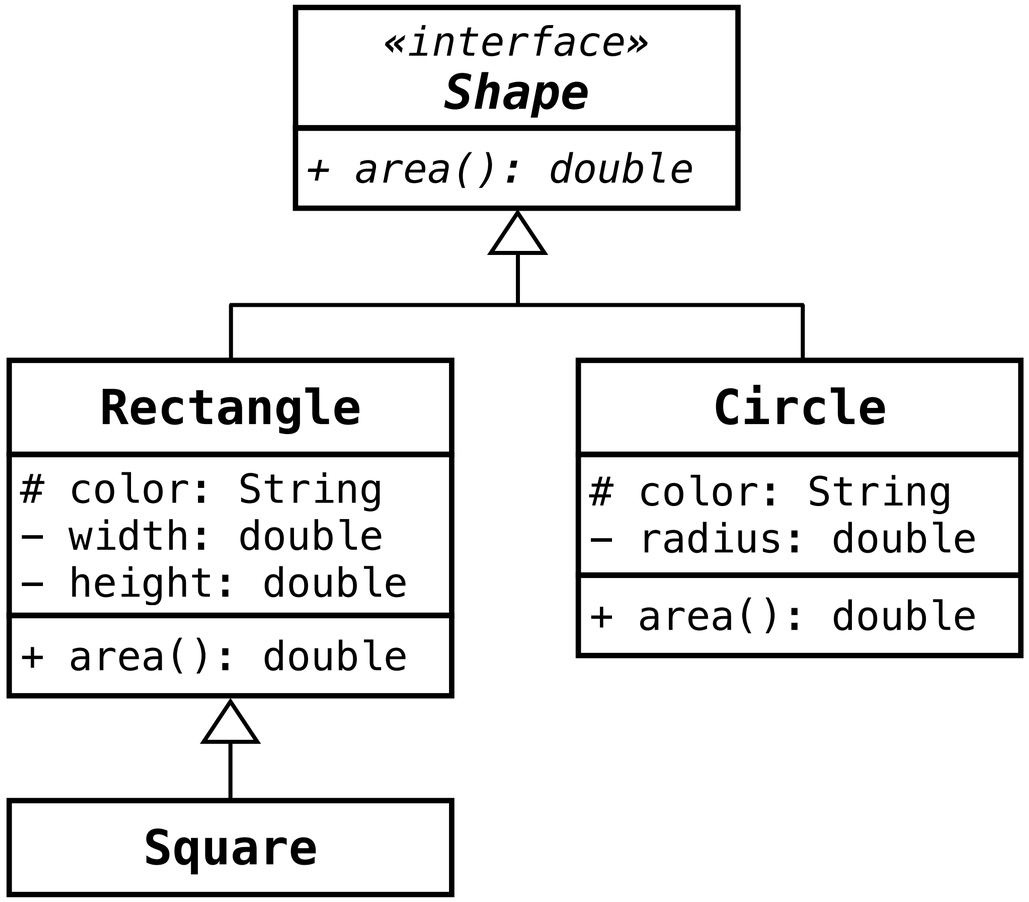
Change: all object methods are automatically **public** and **abstract**

```
interface Shape {
    double area();
}

class Circle implements Shape {
    private String color;
    //...
}

class Rectangle implements Shape {
    private String color;
    //...
}

class Square extends Rectangle {
   //...
}
```

The attribute needs to be defined in the subclasses and can be private

Example solution: models





→ In this example, an **abstract class** is preferred, because **Shape** contains state (i.e. attributes)

Abstract classes vs. interfaces

Property	Interface	Abstract class
Inheritance	Implement several interfaces	Only one abstract class
Structure	**Not available:** attributes (state), constructors **Available:** abstract, static and default (object) methods	**Available:** attributes (state), constructors **Available:** abstract, static, and object methods
Design purpose	Future enhancement: define a contract that can be implemented by any class in the inheritance tree	Avoid independence: provide a common base with shared code to enforce a structure on the subclass
Default implementation	Provides a default implementation that does not require subclassing	Offers a partially completed template that subclasses can build upon
Access modifiers	Everything is assumed public	Can have other access modifiers
Flexibility	Various implementations share only method signature, polymorphic hierarchy of value types	Various implementations of the same kind share a common behavior
State management	Cannot hold state; it defines behavior without implementation details	Can hold state (based on attributes) and provide a consistent implementation base
Role in code	Acts as a capability (“features”) or contract that a class can promise to fulfill	Serves as a skeletal foundation (identity) that defines and partially implements a common family of classes

Break

10 min

The lecture will continue at **17:00**

Outline

- Inheritance (part 1)
- Inheritance (part 2)
- Abstract classes and interfaces
- **Polymorphism**

Polymorphism

- Object oriented programming (OOP) languages exhibit four basic characteristics
  1. Abstraction ✓
  2. Encapsulation ✓
  3. Inheritance ✓
  4. Polymorphism

- Polymorphism (Greek): **many** (poly) **forms** (morph)
- Polymorphism extends inheritance: allows to modify functionality by **overriding** methods of a superclass

Polymorphism

- A **polymorphic** method has the same name in (different) classes, but different behavior
- Two core types are distinguished
  1. **Static** or **compile time** polymorphism: enforced at compile time → **overloading**
  2. **Dynamic** or **runtime** polymorphism: realized at runtime → **overriding**
- Typically refers to dynamic polymorphism
- Allows a child class to share the information and behavior of its parent class while also incorporating its own functionality
  - Simplifies syntax and reduces the cognitive overload for developers
  - Does the compiler always know which method to call?

Static polymorphism: **overloading**

- During code compilation, the compiler verifies that all invocations of a (overloaded) method correspond to at least one defined method: **static binding**

```
class Vehicle {
    public void move() {
        System.out.println("Vehicles can move");
    }
    public void move(int speed) {
        System.out.println("Vehicles can move with " + speed + "km/h");
    }
}

class Test {
    public static void main(String[] args) {
        Vehicle vehicle1 = new Vehicle();
        vehicle1.move();      // prints Vehicles can move
        Vehicle vehicle2 = new Vehicle();
        vehicle2.move(10);    // prints Vehicles can move with 10km/h
    }
}
```

Dynamic polymorphism: **overriding**

- Detect the appropriate method to execute when a subclass is assigned to its parent form
- This is necessary because the subclass may override some or all of the methods defined in the parent class
- **Overriding**: defining methods with the same signature
  - Overriding inside one class is **not** possible
  - Overriding in a class hierarchy is possible (called dynamic polymorphism)
- Possible due to **late binding** (also called **dynamic binding**): the compiler resolves the method call binding during the execution (runtime) of the program

Dynamic polymorphism: overriding - example

```java
class Vehicle {
    public void move() {
        System.out.println("Vehicles can move");
    }
}

class MotorBike extends Vehicle {
    public void move() {
        System.out.println("MotorBike can move and accelerate too");
    }
}

class Test {
    public static void main(String[] args) {
        Vehicle vehicle1 = new MotorBike();
        vehicle1.move();    // MotorBike can move and accelerate too
        Vehicle vehicle2 = new Vehicle();
        vehicle2.move();    // Vehicles can move
    }
}
```

Static type (**compile** time type)

Dynamic type (**runtime** type)

- The program uses the **dynamic** type at **runtime** to find the appropriate method
- If the **dynamic** type does not implement the method, the program navigates to the **superclass** of the current type until a method with the correct signature is found

Problems with polymorphism

- **Type identification during down casting**: the compiler allows down casting even if it might not be possible
  - This can lead to **`ClassCastException`**s
  - Example

There is no compile error, only a runtime exception

```
Vehicle vehicle = new Vehicle();
MotorBike motorBike = (MotorBike) vehicle;
motorBike.move();
```

  - **Recommendation:** always use an **`instance of`** check before down casting

```
Vehicle vehicle = new Vehicle();
if(vehicle instanceof MotorBike) {
    MotorBike motorBike = (MotorBike) vehicle;
    motorBike.move();
}
```

Or even shorter:

```
if(vehicle instanceof MotorBike motorBike) {
    motorBike.move();
}
```

Polymorphism quiz 1

3 min

- What does the following program print and why?

```
class Car {
}
class SportsCar extends Car {
}
public class Exercise {

    public void drive(Car car) {
        System.out.println("Using Car");
    }
    public void drive(SportsCar sportsCar) {
        System.out.println("Using SportsCar");
    }
    public static void main(String[] args) {
        Car car = new Car();
        SportsCar sportsCar = new SportsCar();
        Car anotherSportsCar = new SportsCar();
        Exercise exercise = new Exercise();
        exercise.drive(car);
        exercise.drive(sportsCar);
        exercise.drive(anotherSportsCar);
    }
}
```

Solution

```
Using Car
Using SportsCar
Using Car
```

Explanation

- The choice of which overloaded method should be called is decided at **compile time**
- The **static type** decides which **overloaded** method is invoked
- The compiler uses the **static type** of `Car` to decide which overloaded method is invoked

Polymorphism quiz 2

3 min

- What does the following program print and why?

```
class Car {
    public int getSpeed() {
        return 100;
    }
}
class SportsCar extends Car {
    public int getSpeed() {     // overridden method
        return 500;
    }
}
public class Exercise {
    public void drive(Car car) {
        System.out.println("Using Car and driving at " + car.getSpeed());
    }
    public void drive(SportsCar sportsCar) {    // overloaded
        System.out.println("Using SportsCar and driving at " + sportsCar.getSpeed());
    }
    public static void main(String[] args) {
        Car car = new Car();
        SportsCar sportsCar = new SportsCar();
        Car sportsCarCar = new SportsCar();
        Exercise myTestClass = new Exercise();
        myTestClass.drive(car);
        myTestClass.drive(sportsCar);
        myTestClass.drive(sportsCarCar);
    }
}
```

Solution

```
Using Car and driving at 100
Using SportsCar and driving at 500
Using Car and driving at 500
```

Explanation

- The output of line 1 and line 2 are straight-forward
- Output line 3: even though the actual object at **runtime** is a **`SportsCar`**, the choice of which **overloaded method** is called is decided at **compile time**
- The **static type** decides which overloaded method is invoked

Polymorphism quiz 3

3 min

- What does the following program print and why?

```
class Car {
    void drive() {
        System.out.println("Drive a car");
    }
}
class SportsCar extends Car {
    void drive() {
        System.out.println("Drive a sports car");
    }
}
public class Exercise {

    public static void main(String[] args) {
        Car car = new Car();
        SportsCar sportsCar = new SportsCar();
        Car anotherSportsCar = new SportsCar();
        car.drive();
        sportsCar.drive();
        anotherSportsCar.drive();
    }
}
```

Solution

```
Drive a car
Drive a sports car
Drive a sports car
```

Explanation

- The output of line 1 and line 2 are straight-forward
- Output line 3: for **overridden** methods, the **dynamic** type is used at runtime to decide which method is used

Object oriented programming (OOP) concepts

1. Abstraction ✓
2. Encapsulation ✓
3. Inheritance ✓
4. Polymorphism ✓

Next steps

- **Tutor group** exercises
  - T03E01 - Fairytale Gone Inheritance
  - T03E02 - InheriTale of Bremen
- **Homework** exercises
  - H04E01 - Code MacDonald and the Inherited Farm
  - H04E02 - The Chamber of Coding Secrets
- Read the following articles
  - https://www.programiz.com/java-programming/inheritance
  - https://www.mygreatlearning.com/blog/polymorphism-in-java

→ Due until **Wednesday, November 29, 13:00**

Summary

- **Inheritance** allows to structure objects in a taxonomy / hierarchy (reuse)
  - Prevent code duplication
  - Only use it if you can follow the Liskovs substitution principle, i.e. if a subclass truly is a superclass
- **Abstract classes** capture abstract types (including state) which cannot be instantiated
  - Each class can only have one (abstract) superclass
- **Interfaces** allow to divide functionality (methods) into features (groups) and to extend functionality of classes incrementally
  - Classes can implement multiple interfaces
- **Polymorphism** means that objects can have many forms
  - Static (compile time) polymorphism (overloading) vs. dynamic (runtime) polymorphism (overriding)
  - Static binding vs. dynamic binding

References

- https://www.programiz.com/java-programming/inheritance
- https://www.programiz.com/java-programming/method-overriding
- https://www.programiz.com/java-programming/abstract-classes-methods
- https://www.programiz.com/java-programming/interfaces
- https://www.programiz.com/java-programming/polymorphism
- https://www.w3schools.com/java/java_inheritance.asp
- https://www.w3schools.com/java/java_polymorphism.asp
- https://www.javatpoint.com/inheritance-in-java
- https://www.javatpoint.com/method-overloading-in-java
- https://www.javatpoint.com/abstract-class-in-java
- https://www.javatpoint.com/interface-in-java
- https://www.mygreatlearning.com/blog/polymorphism-in-java
- https://www.codecademy.com/learn/learn-java/modules/learn-java-inheritance-and-polymorphism/cheatsheet