CHAPTER 8: POLYMORPHISM

LEARNING OBJECTIVES

- Know the principle of substitution for classes and interfaces
- Be able to explain polymorphism and dynamic binding
- Be able to explain the flexibility that can be achieved through polymorphism

8.1 SUBSTITUTION

Java supports the so-called substitution principle, which states that a subclass reference can also be used instead of a superclass reference. Given is:

```
interface Weighable { ... }
class Animal implements Weighable { ... }
class Dog extends Animal { ... }
class Cat extends Animal { ... }
```

... so the following statements are true or false (see comments):

```
Animal t = new Cat(); // OK - new Cat() returns subclass reference
Dog h1 = new dog(); // OK - same types
Dog h2 = t; // Not OK: Superclass ref. instead of subclass ref.
Cat k = new Cat(); // OK
k = h1; // Not OK, incompatible types!
t = k; // OK, so-called upcast
```

Note furthermore:

- `h2 = t` is not correct, because otherwise you could access the interface of a Dog object at runtime via `h2`. Since `h2` refers to a cat object in this case, you could call dog-specific methods that the object does not support!
- `t` has a smaller interface than `k`, i.e. you lose functionality through upcasts!

The substitution principle also applies to interfaces: if an interface is required, any object reference to a class that implements the interface can be used:

```
Figure f = new Circle(2.5);
```

This applies under the following assumption: Figure is an interface that is implemented by class Circle.

EXERCISE: SUBSTITUTION PRINCIPLE (SUBSTITUTABILITY)

Which statements are correct?

- [ ] Substitutability for interfaces means that an object reference whose underlying class implements the interface can be used.
- [ ] Substitutability for classes means that subclass references can be used instead of superclass references.
- [ ] Substitutability for interfaces means that another interface can be used instead of an interface.

8.2 DYNAMIC BINDING

Polymorphism means diversity.

In software design, this means that different behaviour can be used with the same interface. This property is used to achieve easy changeability or expandability of software by exchanging functionalities without adapting and testing the rest of the software.

In Java, this is achieved through the use of superclass references or interfaces utilising the substitution principle:

```
class GeoFigure {
    void print(){ System.out.print("GeoFigure ");
}
class Circle extends GeoFigure {
    void print(){ System.out.print("Circle ");
}
class Rectangle extends GeoFigure {
    void print(){ System.out.print("Rectangle ");
}
class PolymorphismTest {
    static public void main(String[] args) {
        GeoFigur[] figures = { new Circle(),
            new Rectangle(), new Circle() };
        for (GeoFigure geoFigure : figures)
            geoFigure.print(); // Output: Circle Rectangle Circle
    }
}
```

The compiler does not know which object is being referenced, it assumes a `GeoFigure` object or its derivatives. At runtime, the JVM decides which object and therefore which method is used!

The mechanism described on the last page is called dynamic binding. This makes it possible to add, for example, a `square` class without changing the rest of the code! The `for` loop would continue to work correctly if `figures` also had a reference of type `square`.

Please be aware: It also works if the `GeoFigur` class is abstract! You cannot then create objects of `GeoFigure`, but `GeoFigure` references may refer to concrete subclass objects (`Circle`, `R_Corner`).

The substitutability for interfaces means that polymorphism also works with interfaces, see the following example:

```
interface IGeoShape {
    void print();
}
class Circle implements IGeoShape {
    void print() { System.out.print("Circle "); }
}
class Rectangle implements IGeoShape {
    void print() { System.out.print("Rectangle "); }
}
class PolymorphismTest {
    static public void main(String[] args) {
        IGeoShape[] figures = {
            new Circle(), new Rectangle(), new Circle() };
        for (IGeoShape shape : figures)
            shape.print(); // Output: Circle Rectangle Circle
        }
}
```

EXERCISE: DYNAMIC BINDING

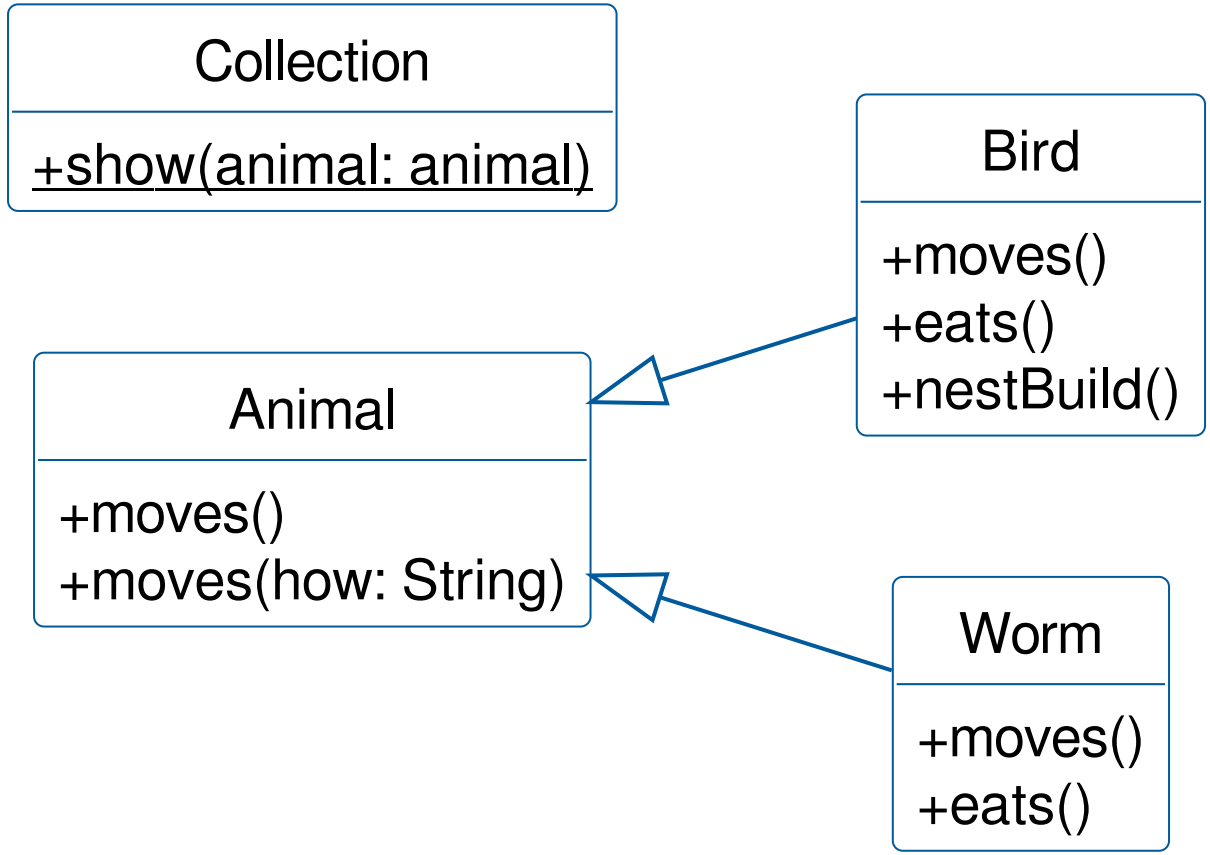
Which statements are correct?

- [ ] Dynamic binding means that dynamic new program parts are added.
- [ ] Dynamic binding only exists for classes with inheritance, but not in connection with interfaces.
- [ ] Dynamic binding means that the JVM decides which object and which method of the object is called.

8.3 EXERCISE

WORMS AND BIRDS AS ANIMALS

Given the following scenario with a class diagram and a corresponding implementation.



```
public class Animal {
    public void moves() { this.moves("???"); }
    public void moves(String how) {
      System.out.println("Moves by " + how); }

    public static void main(String[] args) {
        Animal[] animals = { new Bird(), new Worm() };
        for (Anamial animal : animals)
            moves(animal);
    }
    public static void moves(Animal animal) { animal.moves(); }
}
```

```
public class Bird extends Animal {
    public void moves() { this.moves("flying"); }
    public void eats() { System.out.println("Bird eats..."); }
    public void nestBuild() { System.out.println("Builds nest..."); }
}
```

```
public class Worm extends Animal {
    public void moves() { this.moves("crawling"); }
    public void eats() { System.out.println("Worm eats..."); }
}
```

EXERCISE: SUBSTITUTION

Substitution refers to...

- ☐ the possibility of using Bird and Worm in the animal array.
- ☐ the inheritance from Animal to Bird / Worm
- ☐ the possibility to overwrite moves
- ☐ the possibility of passing Bird and Worm in the static method `move(animal: Animal)`.
- ☐ the possibility to overload moves

EXERCISE: POLYMORPHISM

Polymorphism refers to...

- [ ] the possibility of using Bird and Worm in the animal array.
- [ ] the inheritance from animal to Bird / Worm
- [ ] the possibility to overload moves
- [ ] the possibility to pass Bird and Worm in the static method `move(animal: Animal)`.
- [ ] the possibility to overwrite moves