FARABI UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE



Object-Oriented Analysis and Design

Lecture 9
Generalization with Interfaces

Java contracts, UML realization and polymorphism

Dr. Bazargul Matkerim, Professor-Researcher

Learning Outcomes

Model contracts

Explain an interface as a type and a behavioural contract.

Distinguish UML realization from generalization.

Implement and use interfaces

Write interchangeable implementations.

Extend interfaces and resolve competing default methods.

Types and Subtyping

A type describes usable operations

A class defines a type for its objects. An interface also defines a reference type.

Clients rely on the operations exposed by the declared type.

An object can have several types

A Dog that implements AnimalSound is both a Dog and an AnimalSound.

AnimalSound animal = new Dog();

Class Inheritance and Interface Contracts

Class inheritance

class Dog extends Animal

A subclass inherits eligible implementation and becomes a subtype.

Java permits one direct superclass.

Interface implementation

class Dog implements AnimalSound

The class promises to satisfy the interface contract.

A class may implement several interfaces.

What a Java Interface Can Declare

Contract members

Abstract methods
Constants: public static final fields
Nested types

Interfaces have no constructors or per-object instance fields.

Methods with bodies

default methods: inherited instance behaviour
static methods: called on the interface
private methods: internal helpers

Private interface methods require Java 9+.

A Minimal Interface

```
interface AnimalSound {
  String sound();
}
```

sound() is implicitly public and abstract.

The declaration specifies the return type.

A contract also describes what the result means.

Example contract: sound() returns a non-null description of the animal’s vocal sound.

Implementing the Contract

```
class Dog implements AnimalSound {
  @Override
  public String sound() {
    return "Woof";
  }
}

class Cat implements AnimalSound {
  @Override
  public String sound() { return "Meow"; }
}
```

Each class supplies its behaviour.

Implementing methods must be public.

@Override catches signature mistakes.

Implementation Requirements

Concrete class

Every required abstract method needs an implementation.

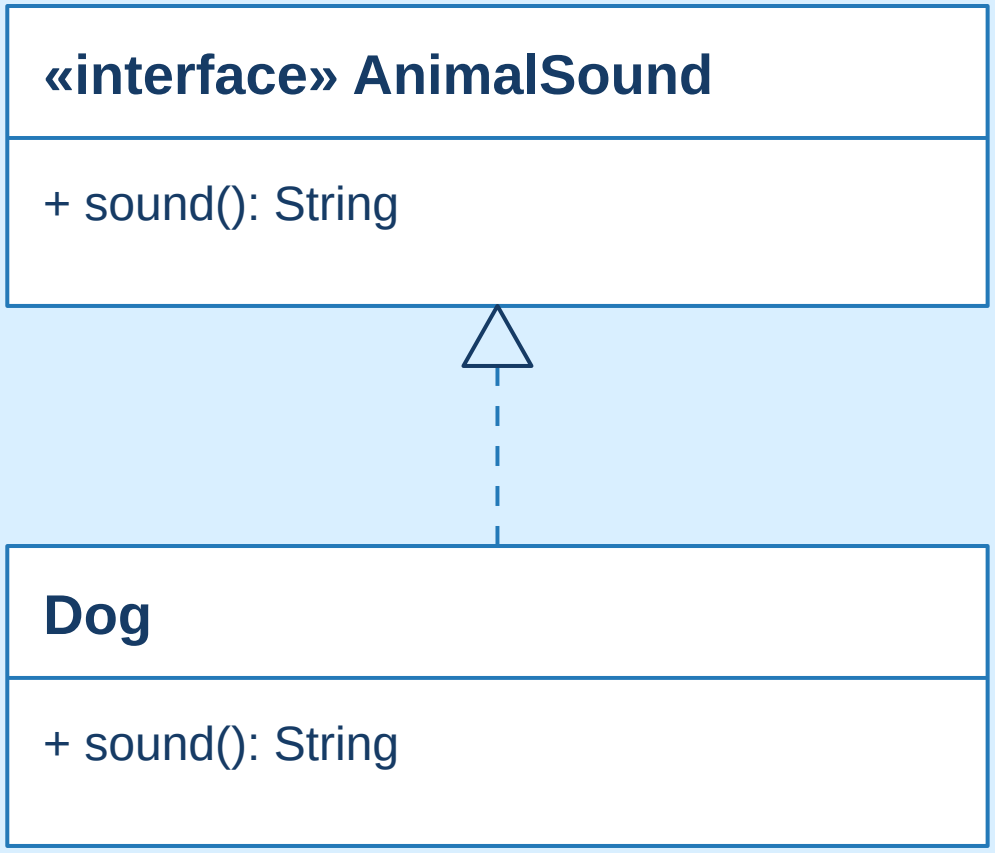
A compatible inherited method may already satisfy the requirement.

Abstract class

An abstract class can leave methods unimplemented.

A default method can provide an inherited body unless a conflict or stronger requirement applies.

UML Interface Realization



Dashed line with a hollow triangle: the class realizes the interface. The triangle points to the interface.

Polymorphism Through an Interface

```
AnimalSound[] animals = { new Dog(), new Cat() };

for (AnimalSound animal : animals) {
  System.out.println(animal.sound());
}

// Output:
// Woof
// Meow
```

One operation in client code.

Different runtime implementations.

The client needs the interface contract.

Reference Type and Runtime Type

Declared type

AnimalSound animal = new Dog();

animal.sound() is available because AnimalSound declares it.

Dog-only methods are unavailable through this reference.

Runtime type

The object is a Dog.

Dynamic dispatch selects Dog.sound().

A cast can fail if the object is not of the target type.

A Client That Depends on a Contract

```
class SoundPlayer {
  public void play(AnimalSound source) {
    System.out.println(source.sound());
  }
}

SoundPlayer player = new SoundPlayer();
player.play(new Dog());
player.play(new Cat());
```

The client accepts a capability.

Adding another implementation need not change play().

Assumption: source is non-null.

The class declaration and example calls are separate snippets. Put calls inside a method.

Interface Generalization

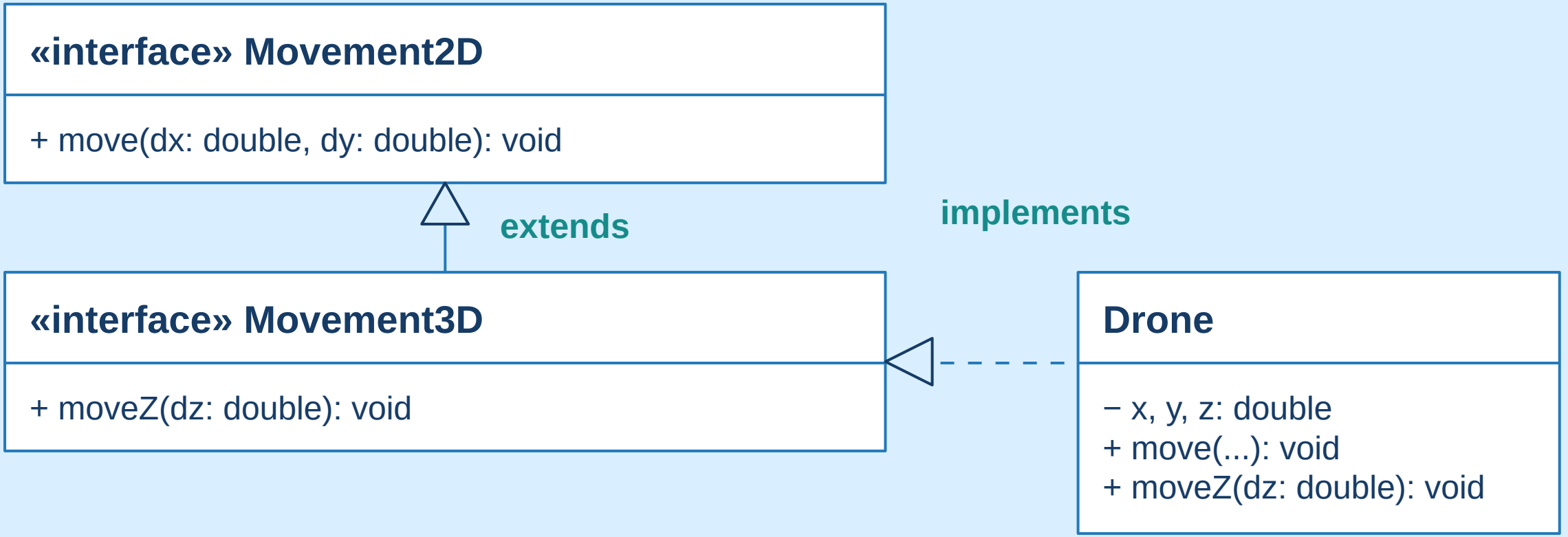
```
interface Movement2D {
  void move(double dx, double dy);
}

interface Movement3D extends Movement2D {
  void moveZ(double dz);
}
```

An interface extends another interface.

Movement3D includes move() and moveZ().

A Movement3D can serve as a Movement2D.

Generalization Versus Realization



Solid line: interface generalization. Dashed line: class realization. Both triangles point to the supertype.

A Three-Dimensional Implementation

```
class Drone implements Movement3D {
  private double x, y, z;

  @Override
  public void move(double dx, double dy) {
    x += dx;
    y += dy;
  }

  @Override
  public void moveZ(double dz) { z += dz; }
}
```

Implements both inherited and new operations.

Position is private object state.

Teaching model assumes finite inputs.

Substitution in an Interface Hierarchy

Movement2D contract

move(dx, dy) changes horizontal position by the given offsets.

A Movement3D implementation must preserve that meaning.

Design warning

Do not extend an interface only to collect unrelated operations.

A subtype must honour the parent’s accepted inputs and promised behaviour.

Implementing Multiple Interfaces

```
interface Printable { void print(); }
interface Scannable { void scan(); }

class AllInOne implements Printable, Scannable {
  @Override
  public void print() { System.out.println("Printing"); }

  @Override
  public void scan() { System.out.println("Scanning"); }
}
```

One object supports both roles.

A client may use either interface type.

This is multiple inheritance of type.

Extending Multiple Interfaces

```
interface MultifunctionDevice
    extends Printable, Scannable {
}

class OfficeDevice implements MultifunctionDevice {
  @Override
  public void print() { System.out.println("Print"); }

  @Override
  public void scan() { System.out.println("Scan"); }
}
```

The child interface combines compatible parent contracts.

An interface extends interfaces.

A class implements interfaces.

Overlapping Abstract Methods

Compatible signatures

```
interface A { void reset(); }
interface B { void reset(); }
```

One public reset() implementation can satisfy both contracts.

Compatibility has limits

Same name alone is insufficient. Parameter and return types matter.

Incompatible return types can prevent implementation. Behavioural promises must also agree.

Default Methods

```
interface Named {
  String name();

  default String label() {
    return "Name: " + name();
  }
}

class Sensor implements Named {
  public String name() { return "Temperature"; }
}
```

A default method has a body.

Sensor inherits label().

A class may override the default to specialize behaviour.

Resolving a Default-Method Conflict

```
interface A {
  default String label() { return "A"; }
}
interface B {
  default String label() { return "B"; }
}
class Both implements A, B {
  @Override
  public String label() {
    return A.super.label();
  }
}
```

Unrelated defaults conflict.

Both must explicitly override label().

This solution chooses A’s default.

Interfaces can create ambiguity. Java resolves some cases by rules and rejects unresolved conflicts.

Interface or Abstract Class?

Interface

Useful for a capability shared by unrelated classes.

Allows multiple interface types.

No instance fields or constructors.

Abstract class

Useful for related classes sharing state and construction rules.

Can combine abstract and concrete methods.

Uses the class’s single superclass slot.

Design Exercise: University Notifications

Required contract

Define Notifier with:
+ send(message: String): void

Assume message must be non-null and non-blank.
Implement EmailNotifier and SmsNotifier.

Deliverables

Draw a UML interface and both realization links.

Write a client accepting Notifier.

Use console output as a classroom simulation.

Exercise Solution and Review

```
interface Notifier { void send(String message); }
class EmailNotifier implements Notifier {
  public void send(String message) {
    if (message == null || message.isBlank())
      throw new IllegalArgumentException("message required");
    System.out.println("Email: " + message);
  }
}
class Announcement {
  void publish(Notifier channel, String text) {
    channel.send(text);
  }
}
```

Implement SmsNotifier with the same input rules.

Check valid and invalid messages.

Verify the client works with either implementation.

The example simulates delivery. Real messaging needs a separate delivery and failure contract.

Key Ideas and Further Reading

Review questions

What does implements promise?
Which UML link is dashed?
Can an interface extend two interfaces?
How do default methods cause conflicts?

Further reading

Oracle Java Tutorials: Interfaces
Dev.java: Defining Interfaces
OMG UML 2.5.1: Interface Realization

Source links are in the slide notes.