FARABI UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE



Object-Oriented Analysis and Design

Lecture 8
Relationships and Inheritance in Java and UML

Association, aggregation, composition and generalization

Dr. Bazargul Matkerim, Professor-Researcher

Learning Outcomes

Model relationships

Distinguish association, aggregation and composition.

Read multiplicities and ownership symbols.

Translate to Java

Represent links with references and collections.

Use inheritance, constructors and overriding correctly.

Decomposition and Generalization

Decomposition

Divide a system into collaborating objects.

Ask which objects refer to or own other objects.

Generalization

Describe a more specific type using a general type.

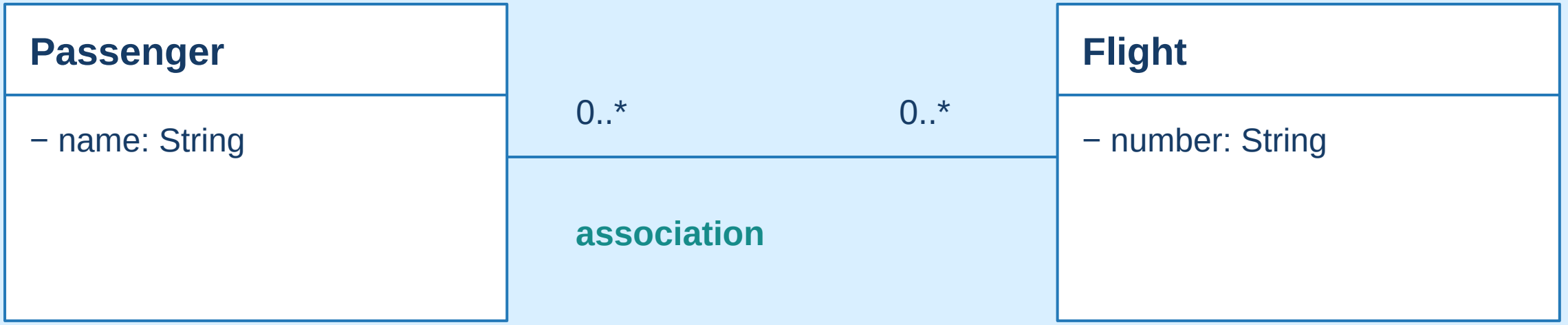
Ask whether a subtype can fulfil the parent contract.

Relationship Notation

Association		Solid line
Shared aggregation		Hollow diamond at whole
Composition		Filled diamond at whole
Generalization		Hollow triangle at parent

The symbol expresses a design decision. Java has no aggregation or composition keyword.

Association



An association describes links between instances. Its meaning and multiplicities come from the domain.

Multiplicity at Each End

Common bounds

1 exactly one
0..1 zero or one
0..* any number
1..* one or more

Reading a relationship

The number near Flight states how many flights one passenger may link to.

A multiplicity alone does not imply ownership.

A Stored Association in Java

```
class Food { }

class Wine {
  private Food pairing;

  public void pairWith(Food food) {
    pairing = food;
  }

  public Food getPairing() { return pairing; }
}
```

A Wine stores a Food reference.

null means no pairing.

Model: 0..1 Food per Wine.

Assumption: food and wine exist independently. This example allows a pairing to be cleared.

Dependency and Association

Temporary use

void recommend(Food food) { ... }

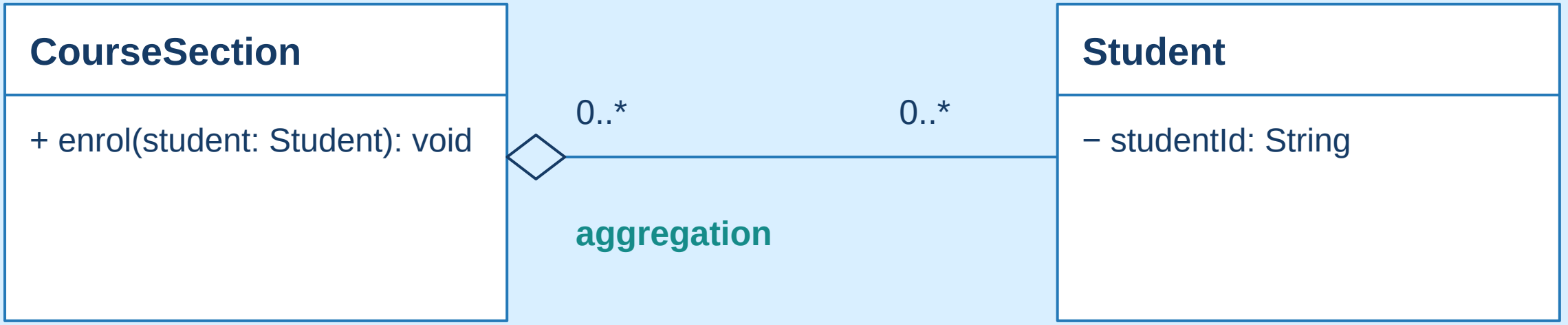
A parameter used only during a call commonly indicates a dependency.

Persistent link

private Food pairing;

A stored reference can implement a navigable association.
Other implementations are possible.

Shared Aggregation



A hollow diamond marks the whole. Shared aggregation needs explicit domain rules.

Pet Store and Pet

Model assumptions

A store maintains a collection of pets.

A pet can exist before joining or after leaving the store.

Ownership decision

The store does not control the pet’s existence.

A plain association may communicate enough. Use aggregation only when it adds meaning.

Aggregation in Java

```
import java.util.ArrayList;
import java.util.List;
class Pet { }
class PetStore {
  private final List<Pet> pets = new ArrayList<>();
  public void add(Pet pet) {
    pets.add(java.util.Objects.requireNonNull(pet));
  }
  public void remove(Pet pet) { pets.remove(pet); }
  public List<Pet> getPets() {
    return List.copyOf(pets);
  }
}
```

Caller supplies existing pets.

Removing a reference does not destroy a Pet.

List.copyOf requires Java 10+.

This simplified example permits duplicates. Add uniqueness rules if the domain requires them.

Composition



A filled diamond marks the whole. A part belongs to at most one composite at a time.

Composition and Lifecycle

Model semantics

The composite controls its parts' lifecycle.

Deleting a composite deletes the parts still contained in it.

Important distinctions

A whole may allow zero parts.
A part may be detached before deletion if the model permits it.

Deleting one part need not delete the whole.

Composition in Java

```
import java.util.ArrayList;
import java.util.List;
class Order {
  private final List<OrderLine> lines = new ArrayList<>();
  public void addLine(int qty) {
    if (qty <= 0) throw new IllegalArgumentException();
    lines.add(new OrderLine(qty));
  }
  private static final class OrderLine {
    private final int quantity;
    private OrderLine(int qty) { quantity = qty; }
  }
}
```

Order creates its parts.

No operation shares a line with another order.

Private nested class limits exposure.

Java references and access control support the design. Garbage collection follows reachability.

Employee and Salary: A Modeling Choice

Salary as a value

private BigDecimal salary;

An amount is usually an attribute.
A primitive or value field alone does not demonstrate object composition.

Salary as a domain object

An owned SalaryRecord could contain pay history and currency.

Composition is justified only when its identity and lifecycle belong to Employee.

Relationship Comparison

Relationship	Meaning	Lifecycle ownership
Association	Links between instances	Defined by the domain
Aggregation	Shared whole–part relation	No exclusive ownership implied
Composition	Exclusive composite ownership	Composite controls its parts

Select relationships from requirements, ownership and lifecycle. Names alone are insufficient.

Relationship Exercise

Classify the pairs

CourseSection and Student
Bookshelf and Book
Order and OrderLine
Dog and Collar

Explain your assumptions

Can the part be shared?
Can it exist independently?
Who controls its lifecycle?
What multiplicities are allowed?

Exercise: One Defensible Answer

Independent objects

CourseSection–Student: association or shared aggregation.

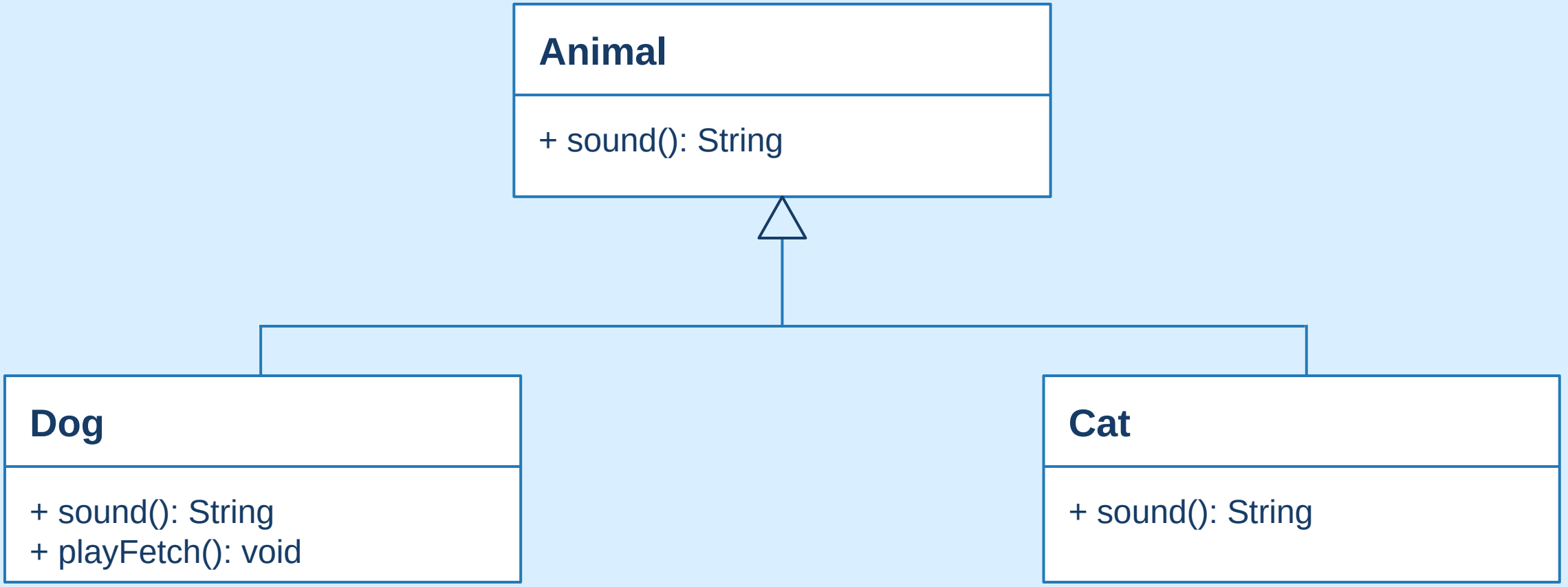
Bookshelf–Book and Dog–Collar: association under independent-lifecycle assumptions.

Owned parts

Order–OrderLine: composition when each line belongs to one order and ends with it.

Different domain rules may justify another model.

Generalization in UML



The hollow triangle points toward the superclass. Dog and Cat specialize Animal.

The Subtype Contract

“Is a” relationship

A Dog can be used where an Animal is expected.

Inherited operations keep their promised meaning.

Design check

Code reuse alone is insufficient.

A subtype should not reject valid parent inputs or violate parent guarantees.

Superclass State and Behaviour

```
abstract class Animal {
  private final String name;

  protected Animal(String name) {
    this.name = java.util.Objects.requireNonNull(name);
  }

  public final String getName() { return name; }

  public abstract String sound();
}
```

Abstract classes cannot be instantiated directly.

Private state remains controlled by Animal.

Subclasses implement sound().

Subclass Implementation

```
class Dog extends Animal {
  public Dog(String name) {
    super(name);
  }

  @Override
  public String sound() { return "Woof"; }

  public void playFetch() {
    System.out.println(getName() + " fetches");
  }
}
```

extends names one superclass.

super(name) initializes parent state.

@Override checks the intended override.

Access Across an Inheritance Hierarchy

private and protected

private members are not inherited. Parent methods can still use parent-private state.

protected also permits access in the same package.

Design implications

Outside the package, protected access follows subclass restrictions.

Prefer private fields and controlled operations over exposed mutable fields.

Default and Explicit Constructors

Compiler-provided constructor

A class with no declared constructor receives a default constructor.

Its implicit super() requires an accessible no-argument parent constructor.

Declared constructor

Declaring any constructor removes the automatic default constructor.

Constructors are not inherited. A subclass initializes its superclass through a constructor call.

Constructor Chaining

```
class Cat extends Animal {
  public Cat() {
    this("Unnamed");
  }

  public Cat(String name) {
    super(name);
  }

  @Override
  public String sound() { return "Meow"; }
}
```

this(...) delegates within Cat.

super(...) invokes Animal's constructor.

The chain initializes one object.

These examples use the conventional constructor form compatible with Java 17.

Overriding and Dynamic Dispatch

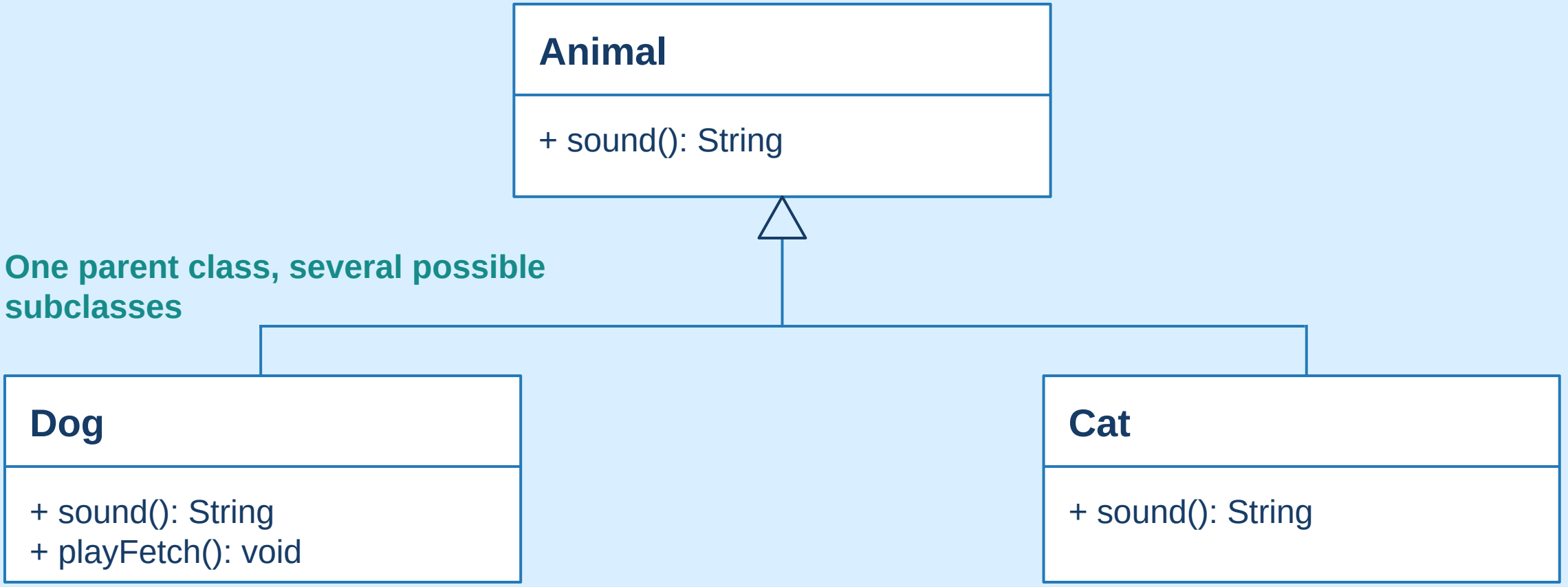
```
Animal first = new Dog("Rex");
Animal second = new Cat("Mimi");

System.out.println(first.sound());   // Woof
System.out.println(second.sound());  // Meow

// first.playFetch(); // compile-time error
```

Reference type controls available operations.

Runtime object type selects the overridden instance method.

Overloading uses different parameter lists. It is distinct from overriding an inherited method.

Single Class Inheritance



One parent class, several possible subclasses

A Java class has at most one direct superclass. It may implement multiple interfaces.

Inheritance or Composition?

Inheritance

Use when the subtype satisfies the parent contract.

Dog extends Animal

Changing the parent can affect every subclass.

Composition and delegation

Use when an object needs a collaborator’s behaviour.

Car has an Engine

The car delegates work without becoming an engine.

Design Workshop: Transport

Task

Model Car, Horse and Plane.
Choose a common operation.
Identify shared state only when it has the same meaning.

Review questions

Does fuel belong in every subtype?
Would a Movable interface be clearer?
Where would Engine composition fit?
Draw UML and sketch one Java class.

Review and Further Reading

Exit questions

Where does each diamond belong?
What does 0..* allow?
How does super(...) differ from this(...)?
Which sound() runs through an Animal reference?

Sources

OMG UML 2.5.1
Class relationships and aggregation

Oracle Java Tutorials
Inheritance and the super keyword

Full links appear in the slide notes.