FARABI UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE



Object-Oriented Analysis and Design

Lecture 7
Expressing Design Structure in Java and UML Class Diagrams

Part 1. Class structure and encapsulation

Dr. Bazargul Matkerim, Professor-Researcher

Learning Outcomes

Read

Interpret class names, attributes, operations and visibility.

Translate

Map a small UML class to a Java implementation.

Protect

Design operations that preserve valid object state.

Part 1 focuses on individual classes, abstraction and encapsulation.

From Responsibilities to Class Structure

CRC view

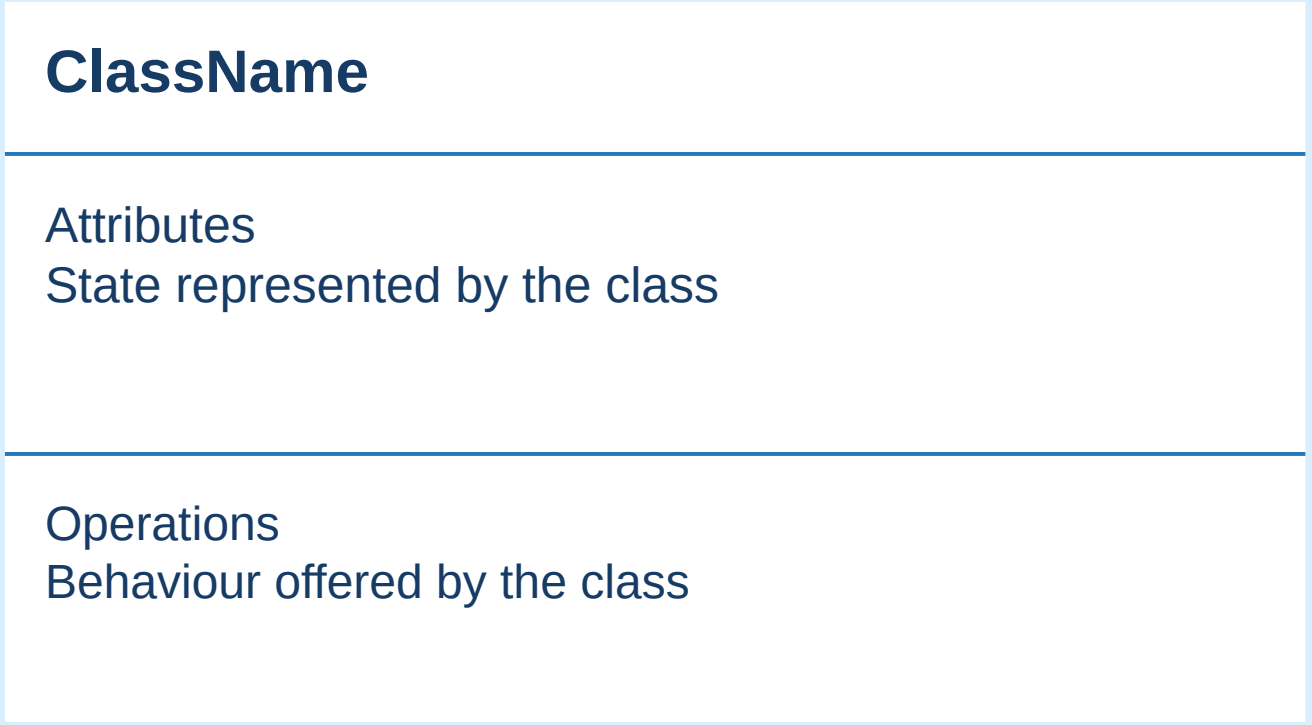
What does the class know?
What does it do?
Which collaborators help?

UML class view

Which attributes represent state?
Which operations express behaviour?
What is visible to callers?

CRC cards support design exploration; class diagrams make structural decisions more precise.

Three Compartments of a UML Class



- Top compartment: name.
- Middle compartment: attributes.
- Bottom compartment: operations.

A class diagram can omit details that are irrelevant to its purpose.

Attribute Notation

UML form

visibility name: Type = default

− count: int = 0
− name: String

Java form

private int count = 0;

private String name;

Java places the type before the name.

The default value is optional. Use a consistent type vocabulary in the diagram.

Choosing Types

Primitive Java types

int: whole-number count
boolean: true or false
double: floating-point value

Reference types

String: text
LocalDate: calendar date
Food: a domain object

A conceptual model may use Integer or Boolean; this lecture uses Java-oriented type names.

Operation Notation

UML form

+ getCount(): int
+ reset(): void
+ setCount(value: int): void

Read the notation

Operation name before parentheses
Parameters inside parentheses
Return type after the colon

An operation describes a callable contract; a Java method supplies its implementation.

Parameters and Return Types

+ isExpired(onDate: LocalDate): boolean

Input	Output
onDate is the date being checked. Its type is LocalDate.	The result is true or false. Its type is boolean.

Parameter names clarify meaning; parameter types constrain the accepted values.

Visibility Symbols

+ public	Accessible to clients when the class itself is accessible.
– private	Restricted to the declaring class; Java nested-class access is a nuance.
# protected	Java permits same-package access and specified subclass access.
~ package	Java has no modifier: accessible within the same package.

Java protected also permits same-package access; cross-package subclass access has restrictions.

Food: Select the Relevant State



Private state

Public queries

No setters required for this example

Scope: track a food item’s identity, name and expiry date. Pricing is outside this example.

Food in Java: State and Construction

```
import java.time.LocalDate;

class Food {
  private final String groceryId;
  private final String name;
  private final LocalDate expiryDate;

  Food(String id, String name, LocalDate expiryDate) {
    this.groceryId = id;
    this.name = name;
    this.expiryDate = expiryDate;
  }
}
```

Final fields

Constructor assigns state

Assume non-null inputs

This skeleton is expanded with operations on the next slide; input validation is omitted here.

Food in Java: Operations

```
// Methods inside Food
public String getName() {
  return name;
}

public boolean isExpired(LocalDate onDate) {
  return onDate.isAfter(expiryDate);
}
```

Query returns the name

Date parameter supplies context

Comparison returns a boolean

Example rule: the item is expired after its expiry date, not on that date. Confirm domain rules.

Keep UML and Java Consistent

UML decisions

− expiryDate: LocalDate
+ getName(): String
+ isExpired(onDate: LocalDate): boolean

Implementation checks

Is the field private?
Do method names and types match?
Does the date comparison
implement the stated rule?

A matching signature does not prove matching behaviour; review contracts and tests too.

ClickCounter: A Small Encapsulated Class

ClickCounter

− count: int = 0

+ getCount(): int
+ click(): void
+ reset(): void

Each object has its own count.

The getter reports the current value.

Commands control valid changes.

The count is private. Clients change it through approved operations.

ClickCounter in Java

```
class ClickCounter {
  private int count = 0;

  public int getCount() { return count; }

  public void click() {
    count = Math.addExact(count, 1);
  }

  public void reset() { count = 0; }
}
```

Query current state

Increment through a command

Reset through a command

Math.addExact reports integer overflow instead of silently wrapping the count.

Why Unrestricted Fields Are Risky

Unrestricted mutation

```
counter.count = -100;
```

A caller can create a state
that the class did not intend.

Controlled operations

```
counter.click();
counter.reset();
```

The class owns valid state changes.

Encapsulation places state transitions behind operations that can enforce invariants.

Getters Are Queries

Purpose

Expose information needed
by callers.

Example: getCount()

Contract

Specify what the result means.
Avoid surprising side effects.
Consider whether a mutable
result exposes internal state.

A getter may compute a result; it does not have to return a stored field directly.

Setters Need a Reason and a Rule

Weak setter

setCount(value) accepts
any integer without checking it.

Invalid values remain possible.

Better interface

Prefer click() and reset()
when these express real actions.
If direct assignment is required,
validate the allowed values.

Creating a setter for every field is not a requirement of encapsulation.

Invariant and Validation

```
// Optional operation inside ClickCounter
public void setCount(int value) {
  if (value < 0) {
    throw new IllegalArgumentException(
        "count must be non-negative");
  }
  count = value;
}
```

Rule: count >= 0

Validate before assignment

On rejection, preserve old state

A validation rule belongs to the class contract and should be reflected in tests.

Exercise: Encapsulate a Person Name

Required behaviour

Store a name privately.
Reject null or blank input.
Remove leading and trailing spaces.
Expose a query and a rename operation.

Deliverables

Draw the UML class.
Write a constructor.
Implement getName() and rename().
Explain what happens on invalid input.

Assume this teaching model requires one non-empty display name, not a legal-name schema.

Person: A Possible Java Solution

```
class Person {
  private String name;
  Person(String name) { rename(name); }
  public String getName() { return name; }
  public final void rename(String value) {
    if (value == null || value.trim().isEmpty()) {
      throw new IllegalArgumentException("name required");
    }
    name = value.trim();
  }
}
```

Validate creation and updates

One rule, reused

Final method prevents overriding

String is immutable, so returning the stored name does not expose mutable internal data.

Mutable Collections Need Care

Leaky getter

return enrolledStudents;

Callers may change the internal list without using the class’s rules.

Safer option

Return a defensive copy or a suitable read-only view.

Consider whether the elements themselves are mutable.

An unmodifiable collection wrapper is not automatically a deep immutable object graph.

Test the Behaviour Behind the Diagram

ClickCounter checks

New counter returns zero.
Two clicks return two.
Reset returns zero.
Negative setCount input is rejected.

Person and Food checks

Null or blank names are rejected.
Invalid rename preserves the old name.
Expiry-day rule is checked
on, before and after the date.

Tests provide evidence for the contract; a class diagram alone cannot establish correctness.

Design Workshop: A Student Class

Build the model

Private student ID and programme
A query for each value
An operation to change programme
An explicit validation rule

Review the result

Should the ID be changeable?
Does every field need a setter?
Do UML and Java signatures match?
What failure case will you test?

State assumptions and justify the public interface from the class’s responsibilities.

Review and Key Ideas

Express the structure

UML names classes and their members.
Attributes use name: Type.
Operations include parameters
and return types.

Preserve the behaviour

Visibility defines access.
Operations protect invariants.
Getters and setters need contracts.
Tests check observable outcomes.

Exit question: why can a class with private fields and public setters still be poorly encapsulated?