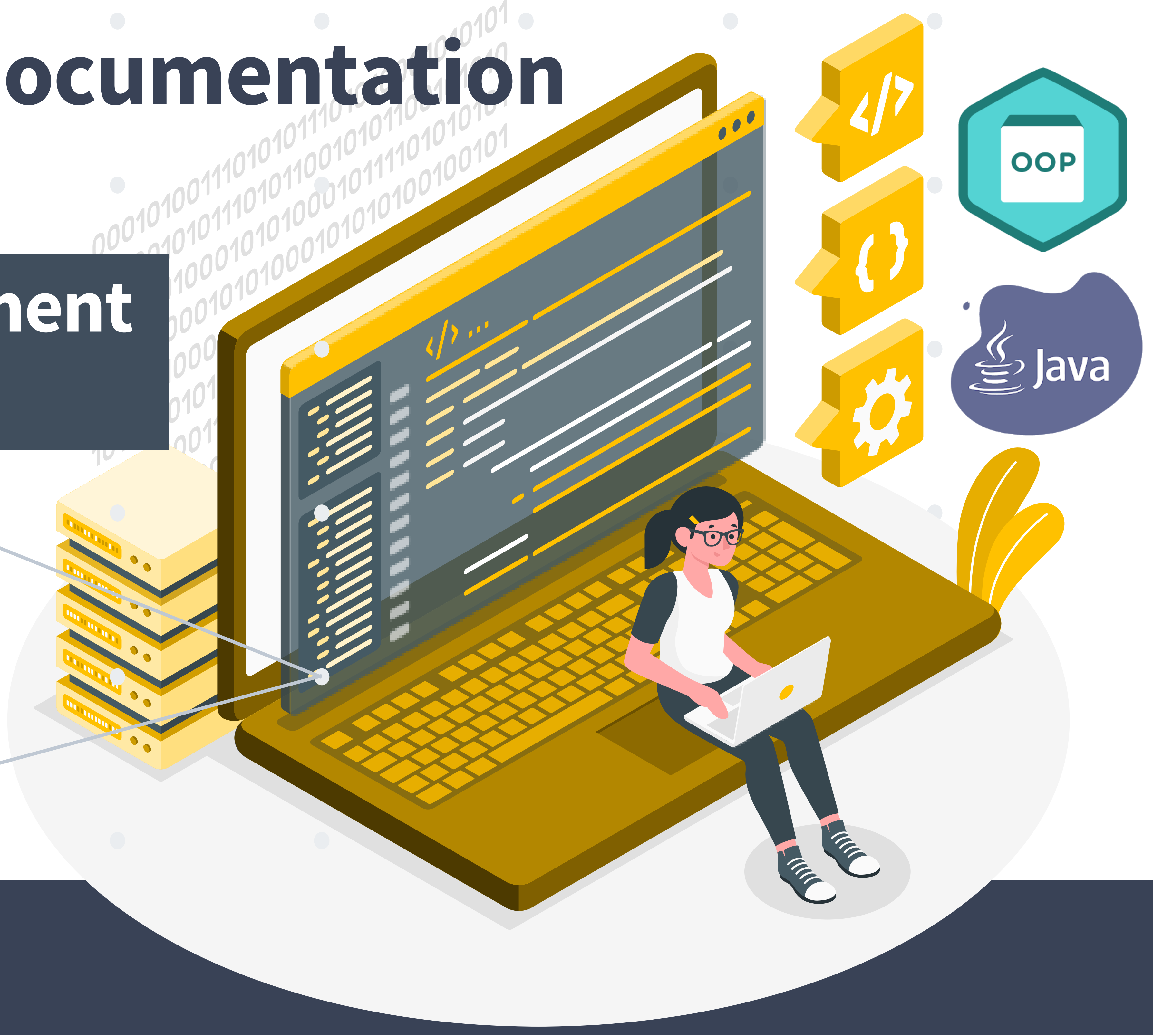
DEPAUL UNIVERSITY

# OOP Principles:

## Abstraction | Encapsulation | Software Documentation



**Object-oriented Software Development**
**SE 350– Spring 2021**

**Vahid Alizadeh**

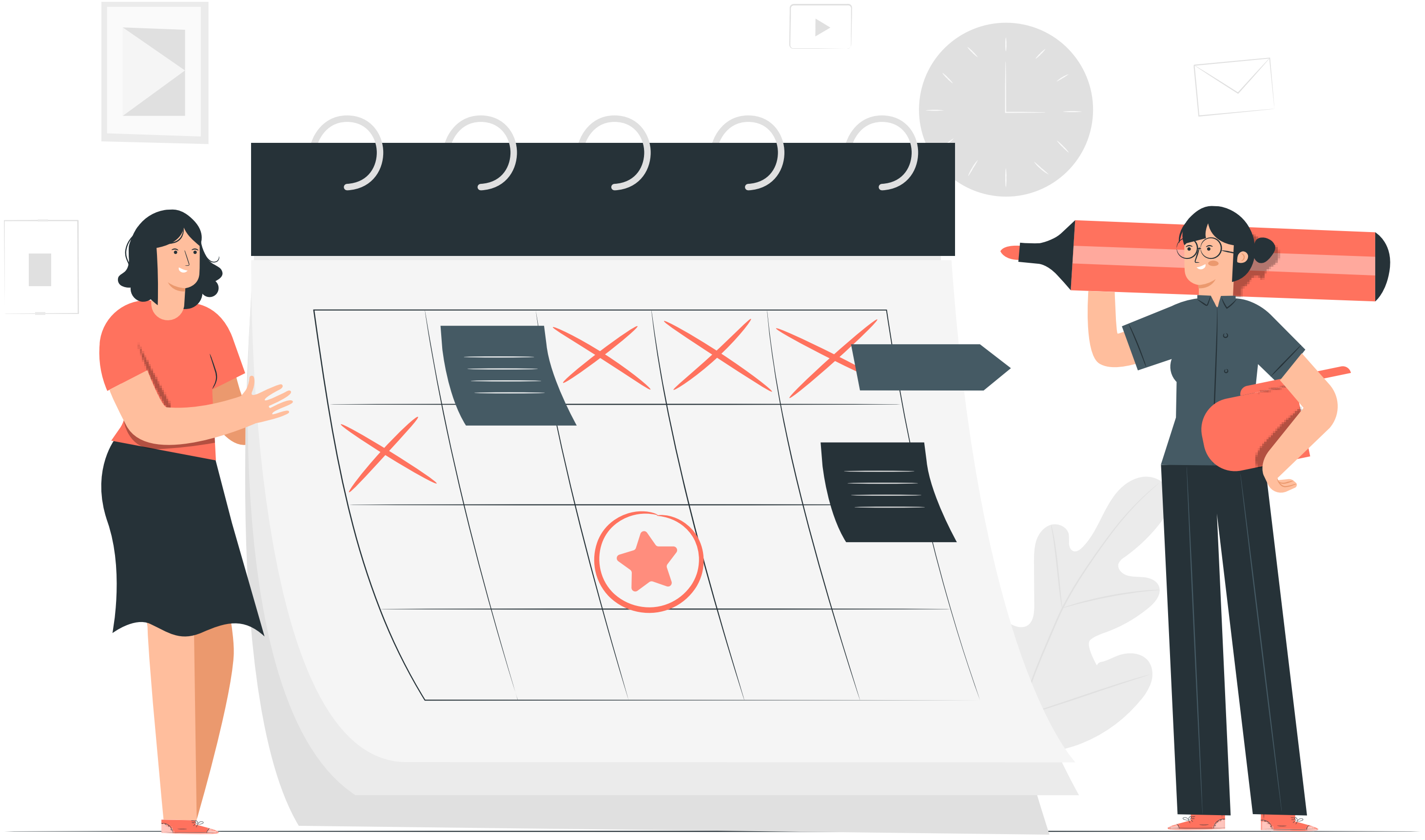
Week 4.2
April 22, 2021

# Announcements

# Future Schedule

## Assignment 2 Tips Video Recording Uploaded in D2L on Tuesday

- ~~**Assignment 1:**~~
  - ~~Release: Week 3.1~~
  - ~~Due: Week 3.1~~
- **Assignment 2:**
  - Release: Week 4.1 (Today)
  - Due: Week 5.1
- **Mid Term Exam:**
  - Week 5.2
  - Thursday – April 29, 2021
  - No Class
  - Take home exam
  - The Midterm questions and a video recording about it will be released 8 AM
  - You have the whole day to submit by 11:59 PM
  - No Limit on the amount of time you spend on the midterm



# Object-oriented Programming

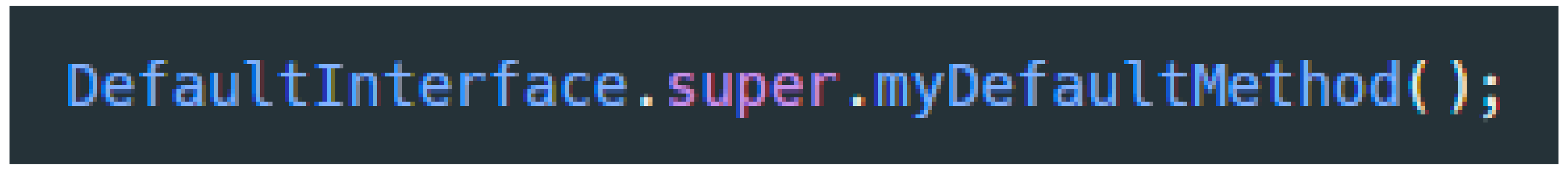
## Principles

# Object-oriented Programming Principles

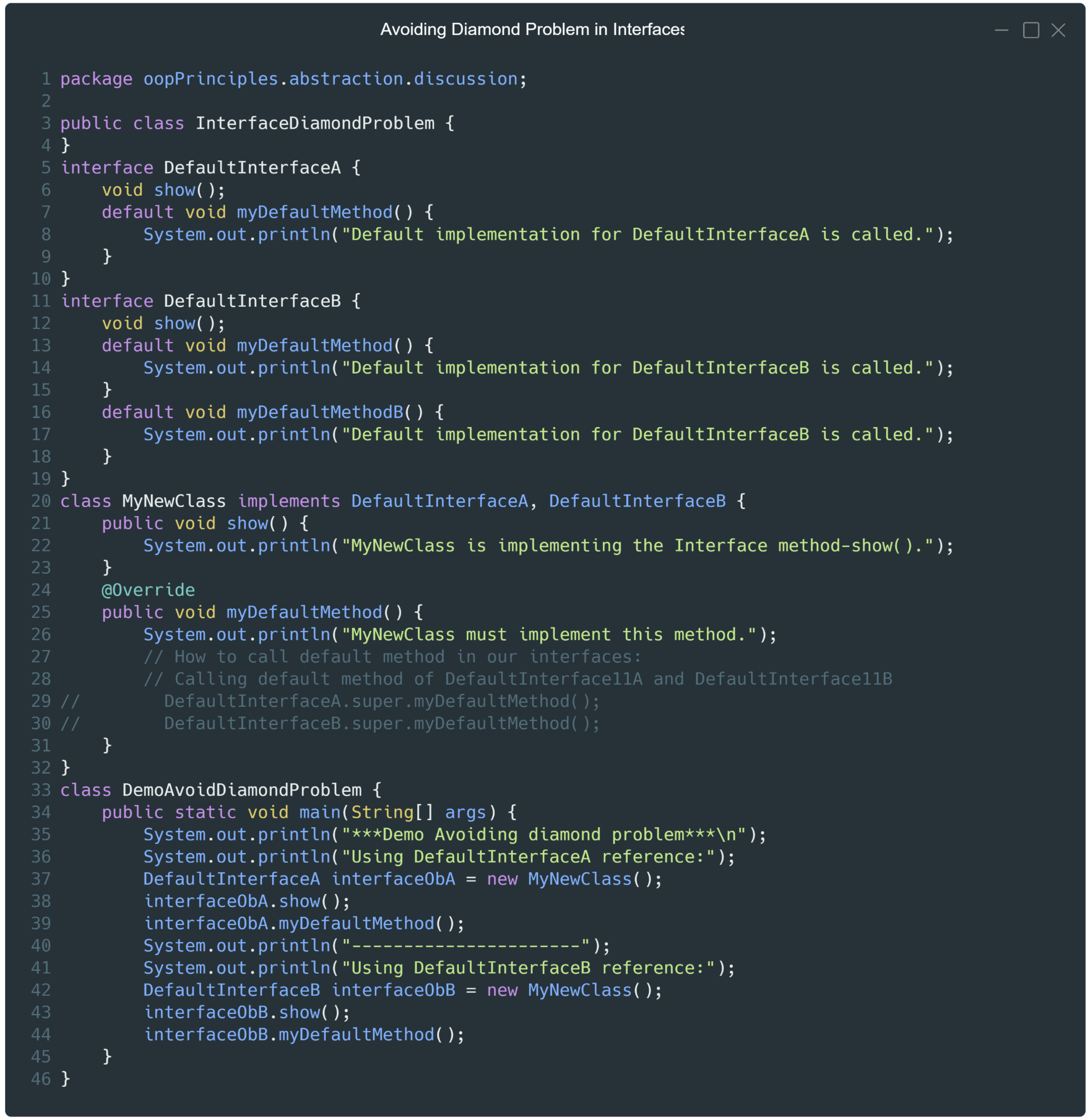
## ABSTRACTION

# Avoiding Diamond Problem in Interfaces with Default Methods

- **How to avoid the diamond problem?**
  - If a class implementing from multiple interfaces, and each interface has same default method, the class must override it.
- **Example** [package oopPrinciples.abstraction.discussion]
  - You can call the default interface method like follows:



```java
DefaultInterface.super.myDefaultMethod();
```

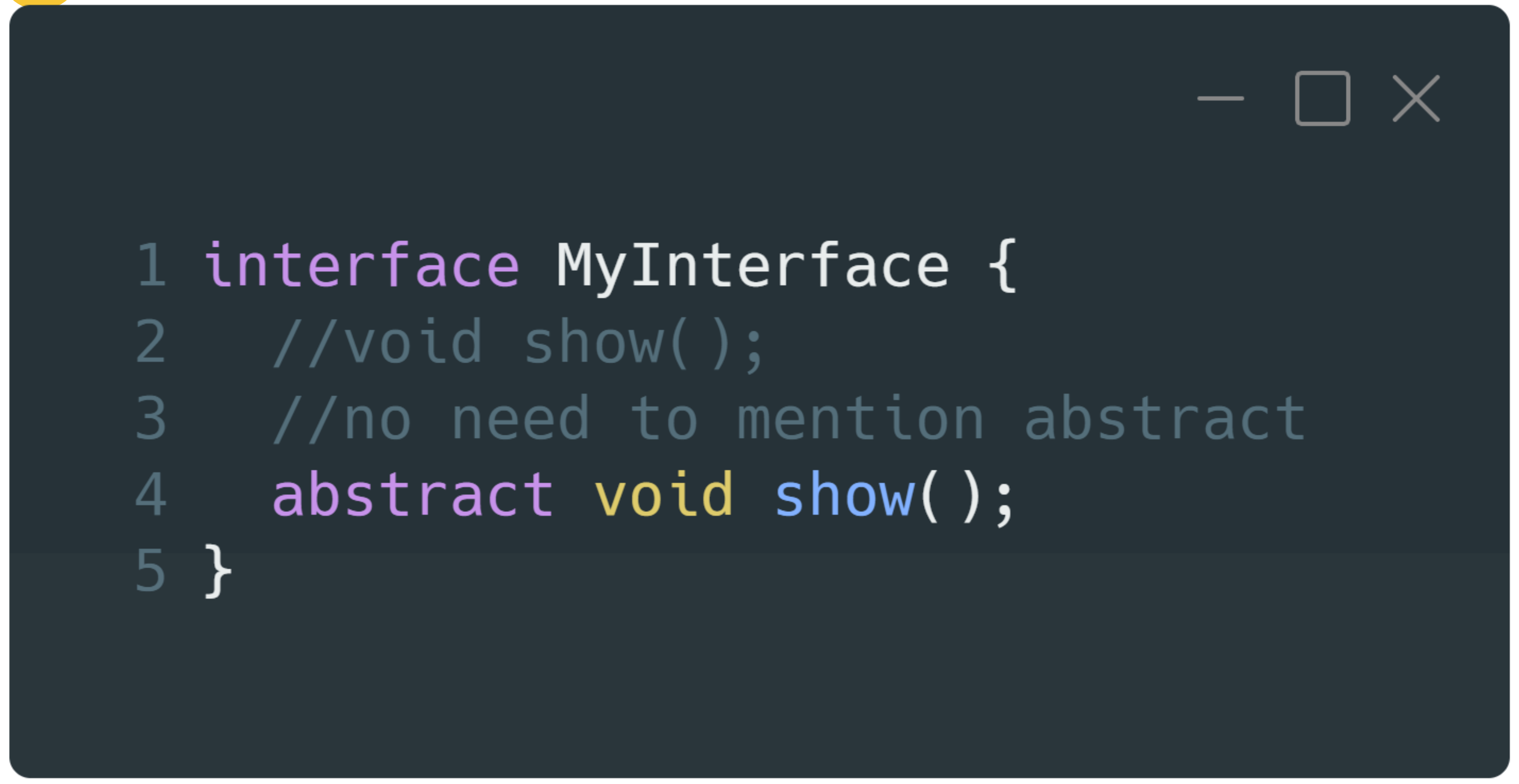


Avoiding Diamond Problem in Interfaces

```java
package oopPrinciples.abstraction.discussion;

public class InterfaceDiamondProblem {
}
interface DefaultInterfaceA {
    void show();
    default void myDefaultMethod() {
        System.out.println("Default implementation for DefaultInterfaceA is called.");
    }
}
interface DefaultInterfaceB {
    void show();
    default void myDefaultMethod() {
        System.out.println("Default implementation for DefaultInterfaceB is called.");
    }
    default void myDefaultMethodB() {
        System.out.println("Default implementation for DefaultInterfaceB is called.");
    }
}
class MyNewClass implements DefaultInterfaceA, DefaultInterfaceB {
    public void show() {
        System.out.println("MyNewClass is implementing the Interface method-show().");
    }
    @Override
    public void myDefaultMethod() {
        System.out.println("MyNewClass must implement this method.");
        // How to call default method in our interfaces:
        // Calling default method of DefaultInterface11A and DefaultInterface11B
//        DefaultInterfaceA.super.myDefaultMethod();
//        DefaultInterfaceB.super.myDefaultMethod();
    }
}
class DemoAvoidDiamondProblem {
    public static void main(String[] args) {
        System.out.println("***Demo Avoiding diamond problem***\n");
        System.out.println("Using DefaultInterfaceA reference:");
        DefaultInterfaceA interfaceObA = new MyNewClass();
        interfaceObA.show();
        interfaceObA.myDefaultMethod();
        System.out.println("----------------------");
        System.out.println("Using DefaultInterfaceB reference:");
        DefaultInterfaceB interfaceObB = new MyNewClass();
        interfaceObB.show();
        interfaceObB.myDefaultMethod();
    }
}
```

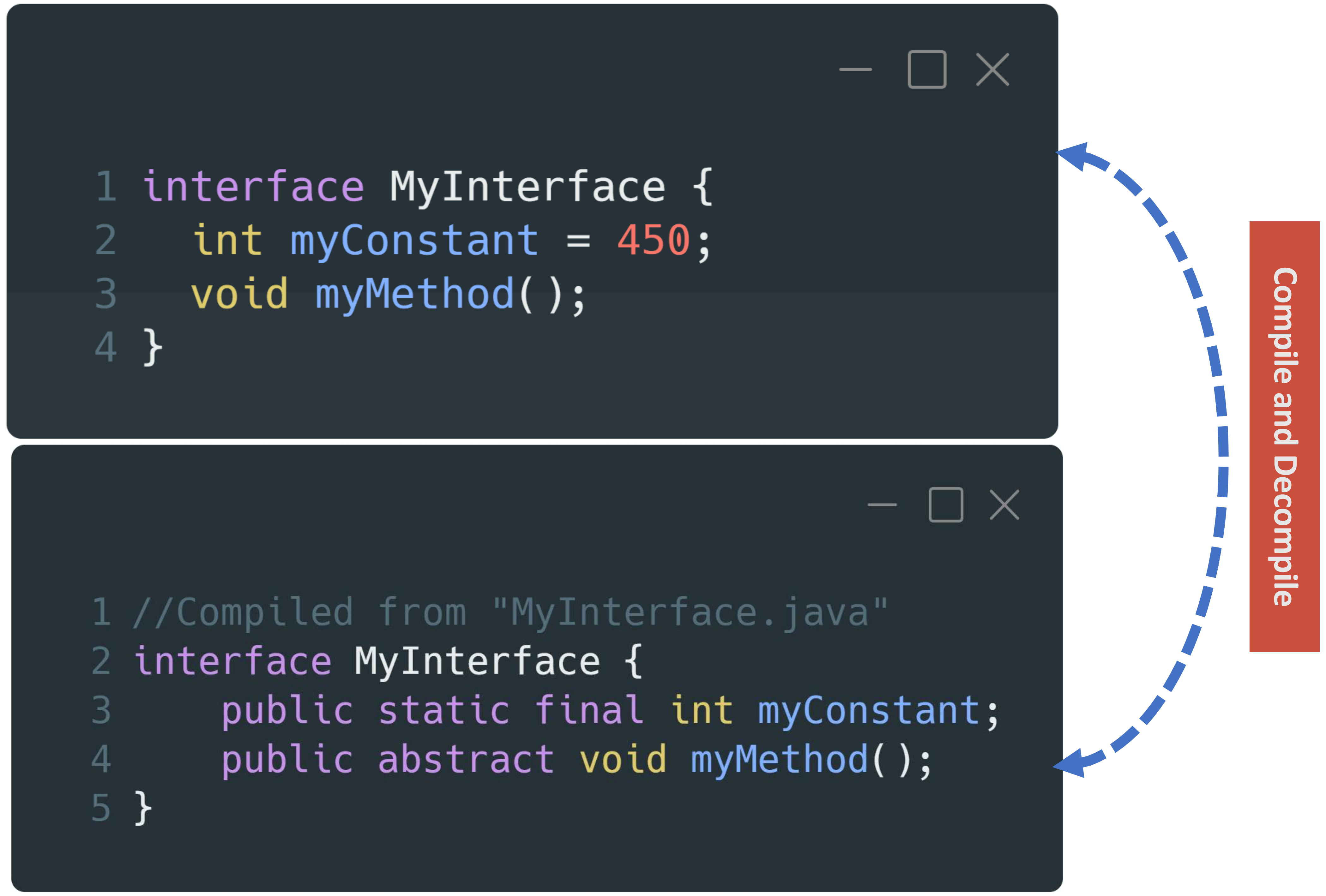
# Interface Discussions

- **You cannot make an interface *final*.**
- **You can use *abstract* before *interface* method (Optional) (Snippet 1)**
  - Interfaces by default are abstract.
- **You can use constants inside an interface. (Snippets 2 and 3)**
  - They are public, static, and final by default.
- **You cannot inherit an interface from a class.**
- **Difference between abstract class and interface?**
  - Concrete methods with *default* keyword
  - Abstract can have only one parent and can extend abstract or concrete classes
  - Interface can have multiple parents and can only extend interfaces.
  - Interface members are by default public.
  - Variables in interface are by default static final.
- **Summary of the interface's benefits.**
  - Polymorphism, Multiple inheritance, loosely coupled systems, parallel developments
- **Summary of this section..**



```
interface MyInterface {
  //void show();
  //no need to mention abstract
  abstract void show();
}
```

1



```
interface MyInterface {
  int myConstant = 450;
  void myMethod();
}
```

```
//Compiled from "MyInterface.java"
interface MyInterface {
    public static final int myConstant;
    public abstract void myMethod();
}
```

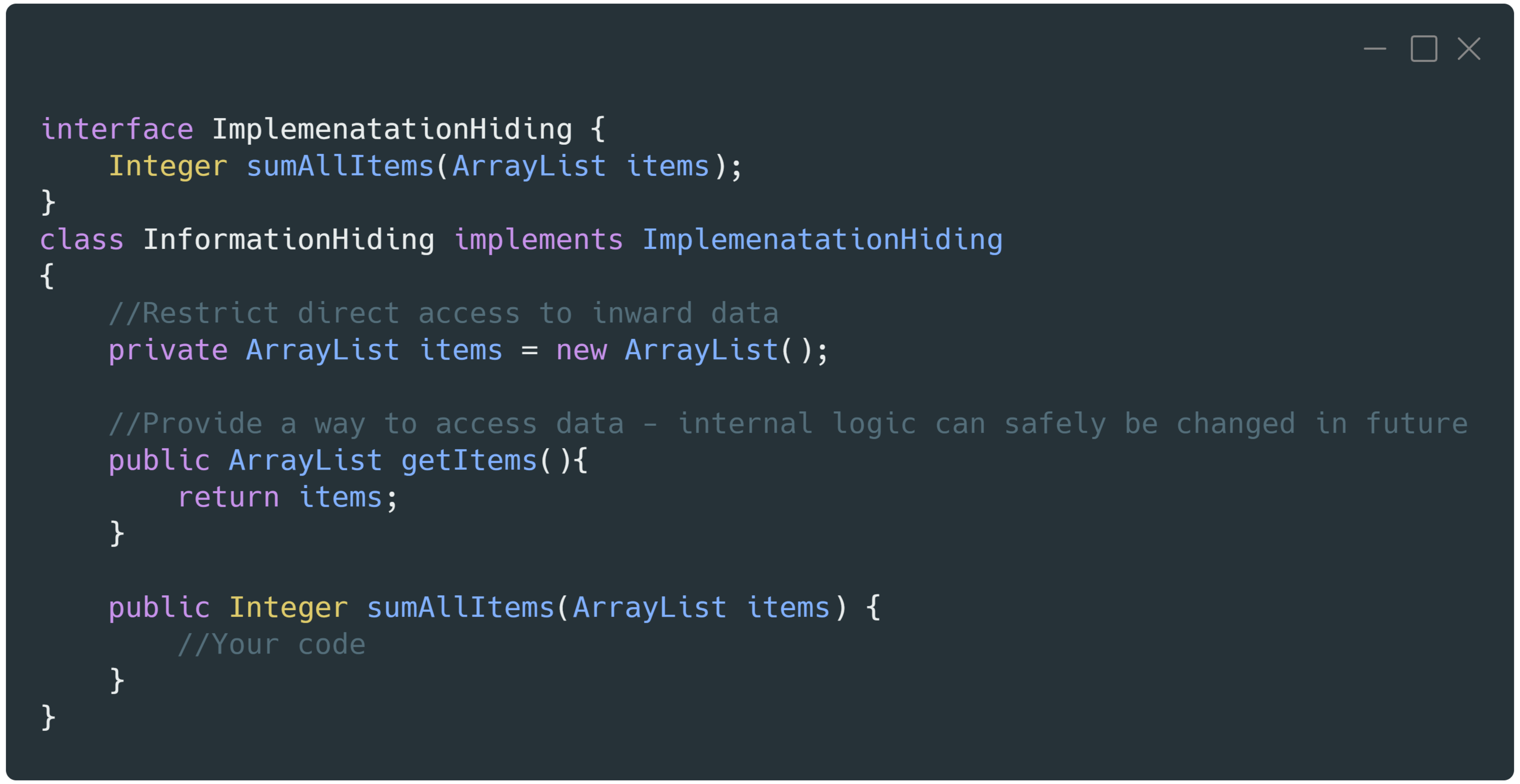
2

# Object-oriented Programming Principles

## ENCAPSULATION

# Encapsulation

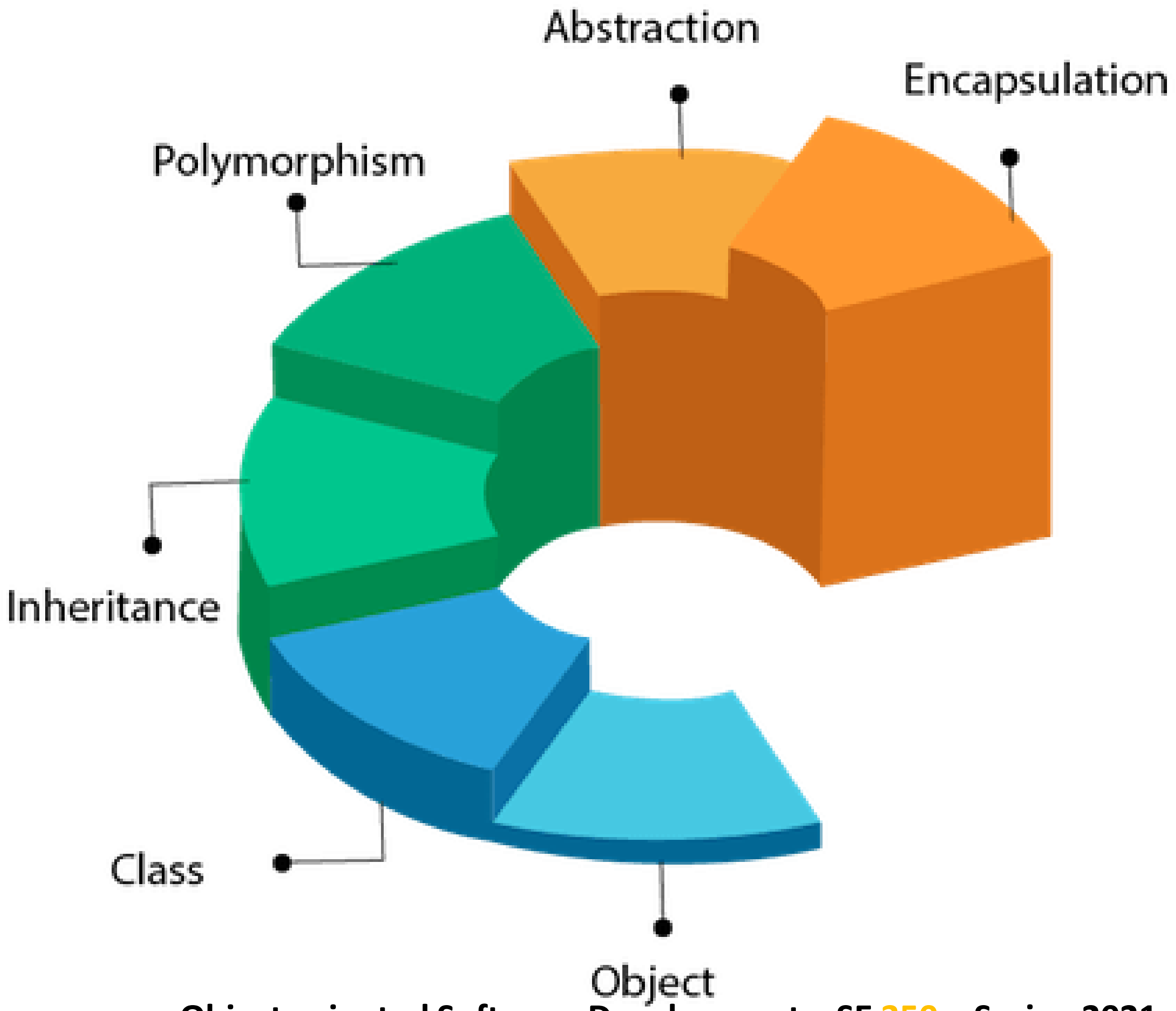
- **Encapsulation**
  - Wrapping code and data together into a single unit
  - **What** should you encapsulate in code?
    - *“Whatever changes encapsulate it”*
  - **Why** use encapsulation? Advantages?
    - Establish the freedom of the client programmer
    - Separating the interface from the implementation
    - Read-only and Write-only classes
    - Control over data
- **Encapsulation has both:**
  - **Information hiding :** Via access control modifiers
  - **Implementation hiding :** Via creation of interface for a class
- **Encapsulation vs Abstraction**
  - Abstraction is more about ‘**What**‘ a class can do. [Idea]
  - Encapsulation is more about ‘**How**‘ to achieve that functionality. [Implementation]



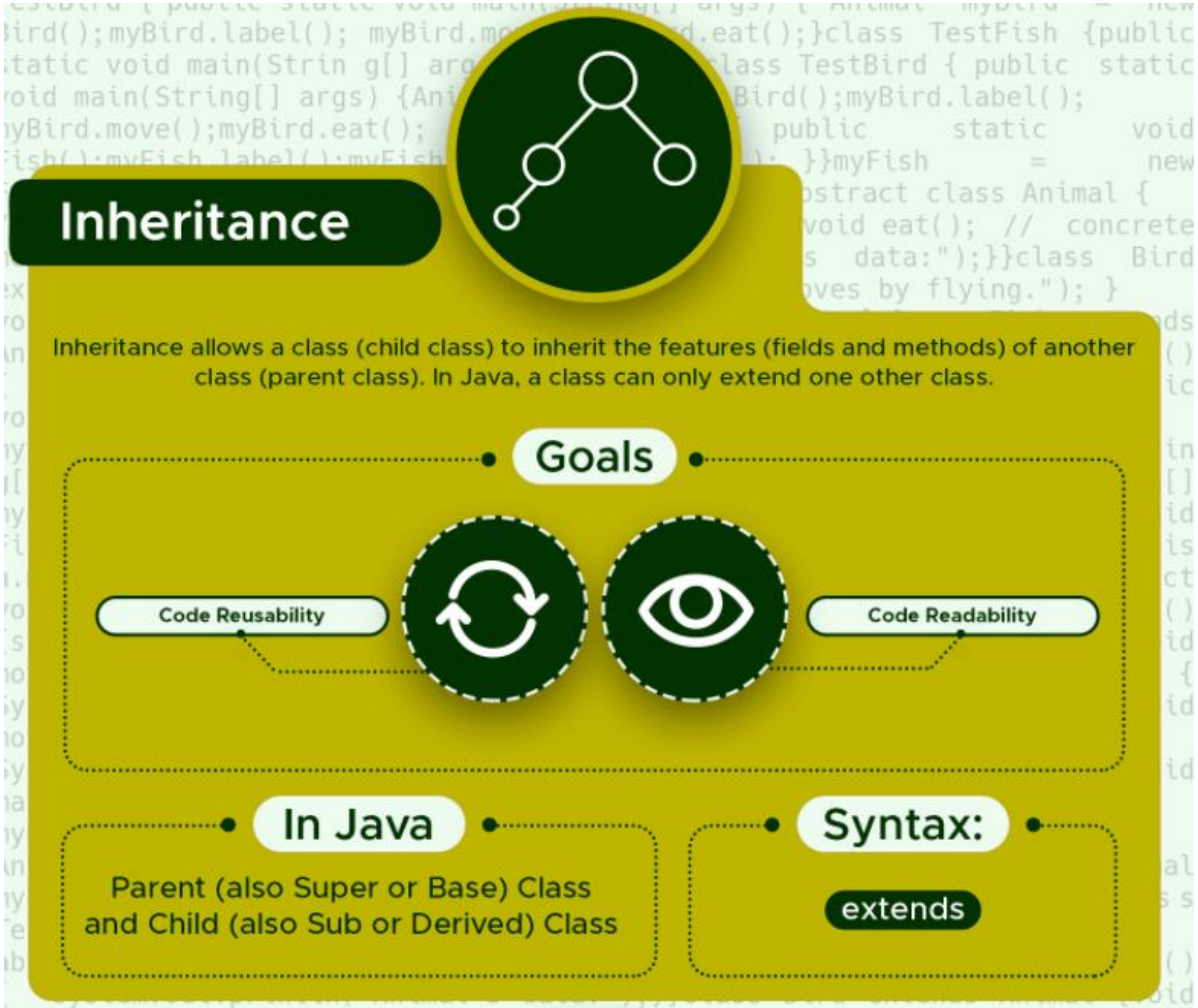
```
interface ImplemenatationHiding {
    Integer sumAllItems(ArrayList items);
}
class InformationHiding implements ImplemenatationHiding
{
    //Restrict direct access to inward data
    private ArrayList items = new ArrayList();

    //Provide a way to access data - internal logic can safely be changed in future
    public ArrayList getItems(){
        return items;
    }

    public Integer sumAllItems(ArrayList items) {
        //Your code
    }
}
```

# Object-oriented Programming Principles

## An overview of 4 pillars of OOP



# Inheritance



## Inheritance

Inheritance allows a class (child class) to inherit the features (fields and methods) of another class (parent class). In Java, a class can only extend one other class.

### Goals

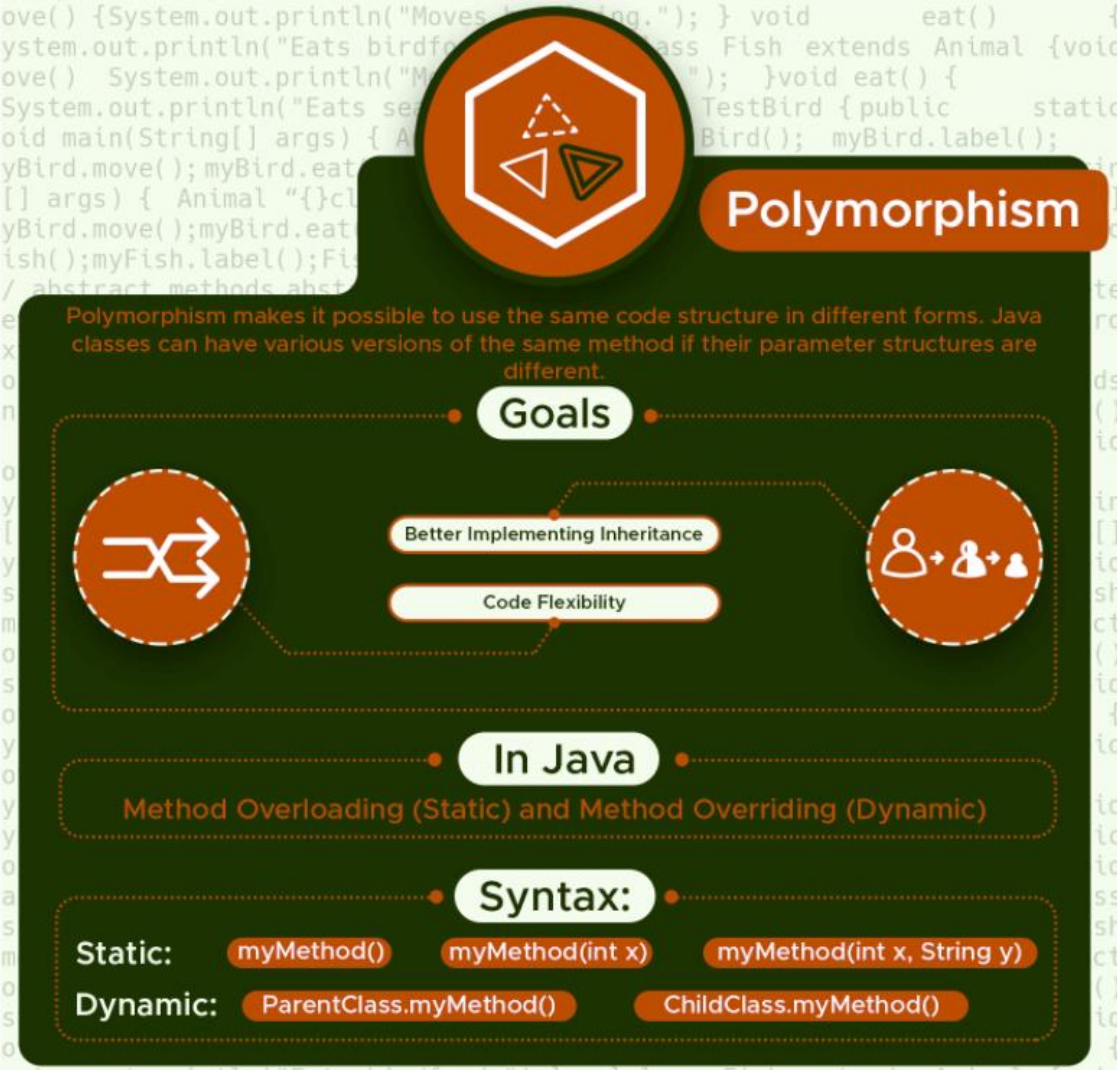
### In Java

Parent (also Super or Base) Class and Child (also Sub or Derived) Class

### Syntax:

`extends`

# Polymorphism



## Polymorphism

Polymorphism makes it possible to use the same code structure in different forms. Java classes can have various versions of the same method if their parameter structures are different.

### Goals

- Better Implementing Inheritance
- Code Flexibility

### In Java

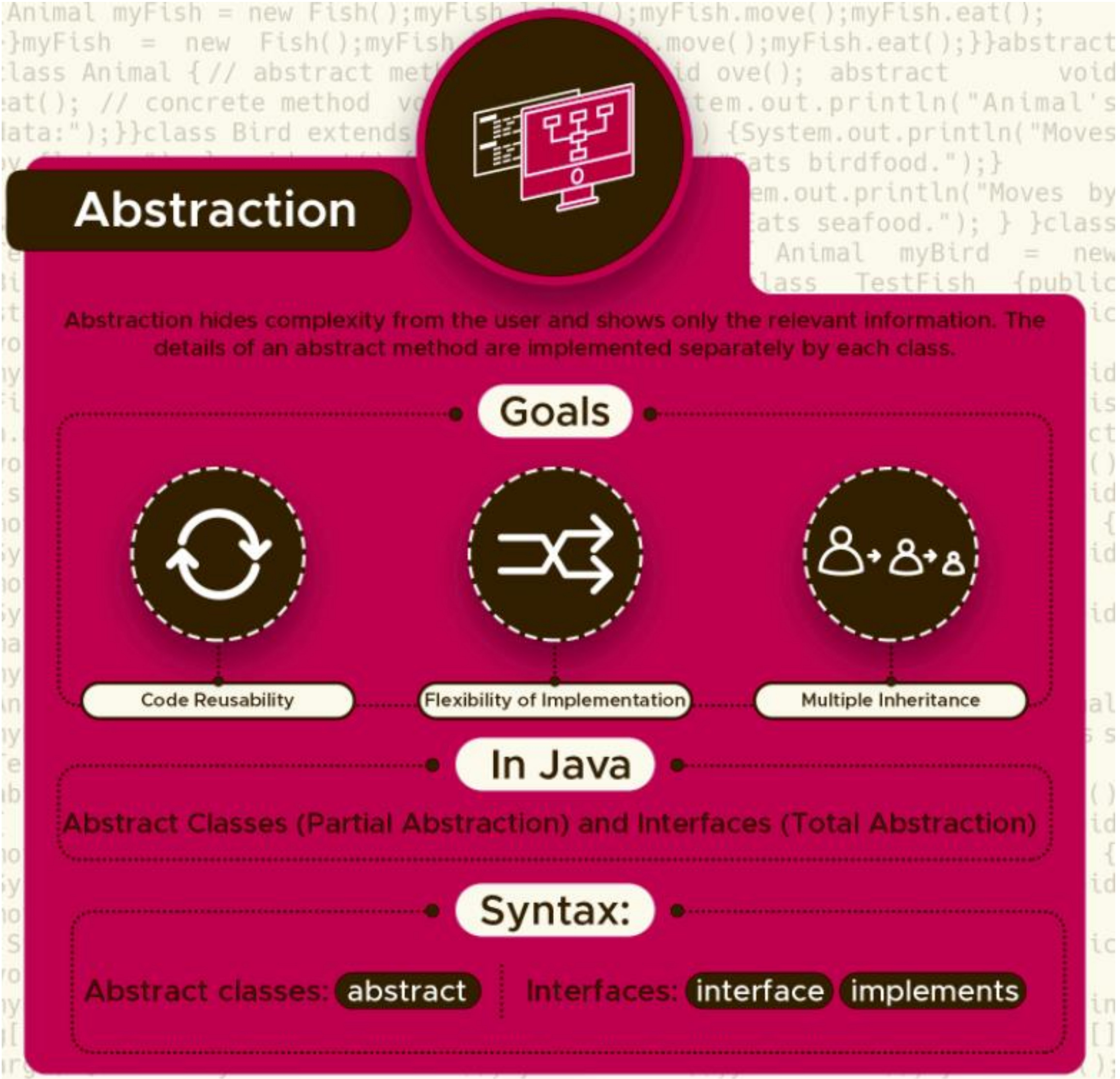
Method Overloading (Static) and Method Overriding (Dynamic)

### Syntax:

Static: myMethod() myMethod(int x) myMethod(int x, String y)

Dynamic: ParentClass.myMethod() ChildClass.myMethod()

# Abstraction



## Abstraction

Abstraction hides complexity from the user and shows only the relevant information. The details of an abstract method are implemented separately by each class.

### Goals

- Code Reusability
- Flexibility of Implementation
- Multiple Inheritance

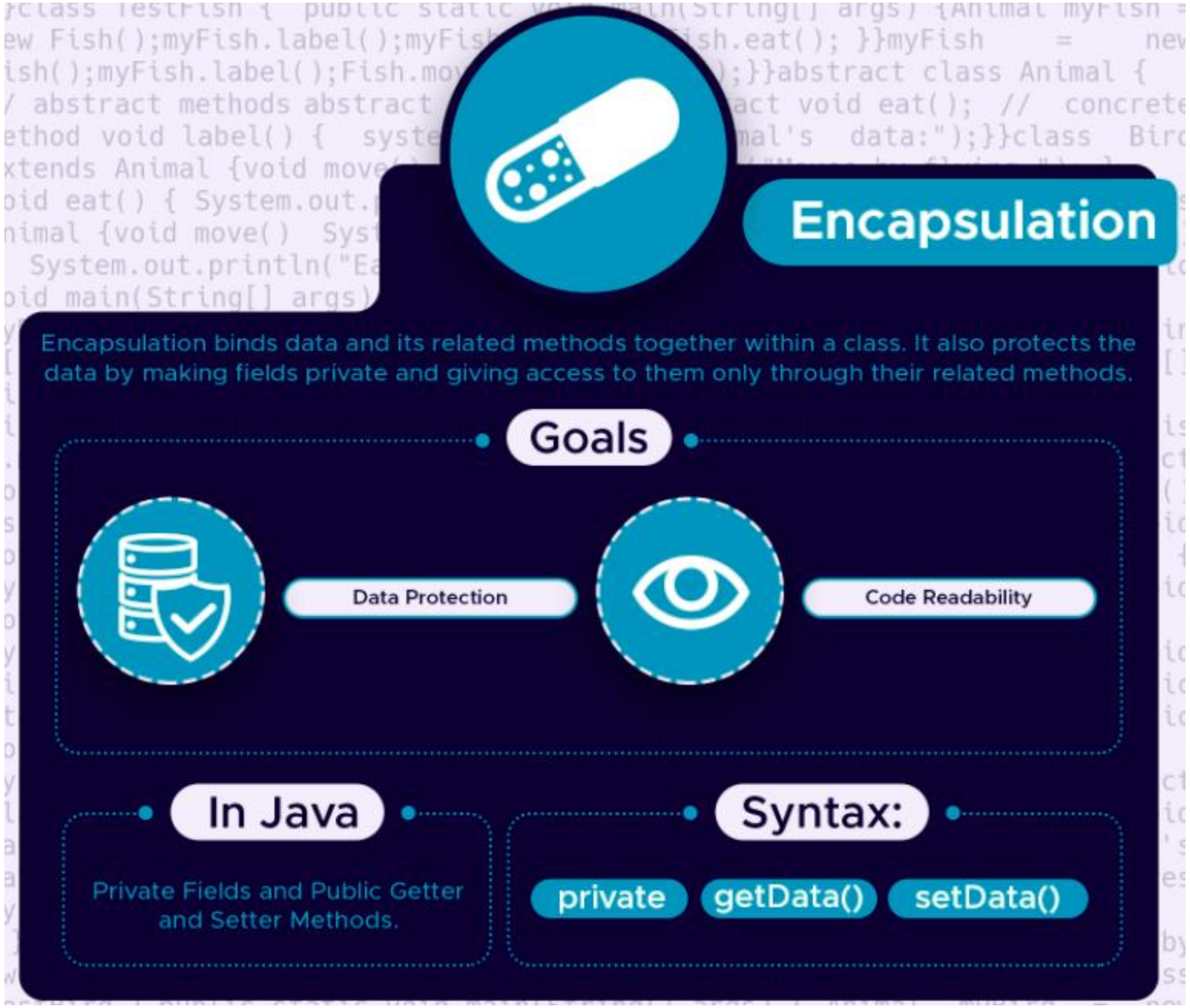
### In Java

Abstract Classes (Partial Abstraction) and Interfaces (Total Abstraction)

### Syntax:

Abstract classes: abstract | Interfaces: interface implements

# Encapsulation



## Encapsulation

Encapsulation binds data and its related methods together within a class. It also protects the data by making fields private and giving access to them only through their related methods.

### Goals

Data Protection

Code Readability

### In Java

Private Fields and Public Getter and Setter Methods.

### Syntax:

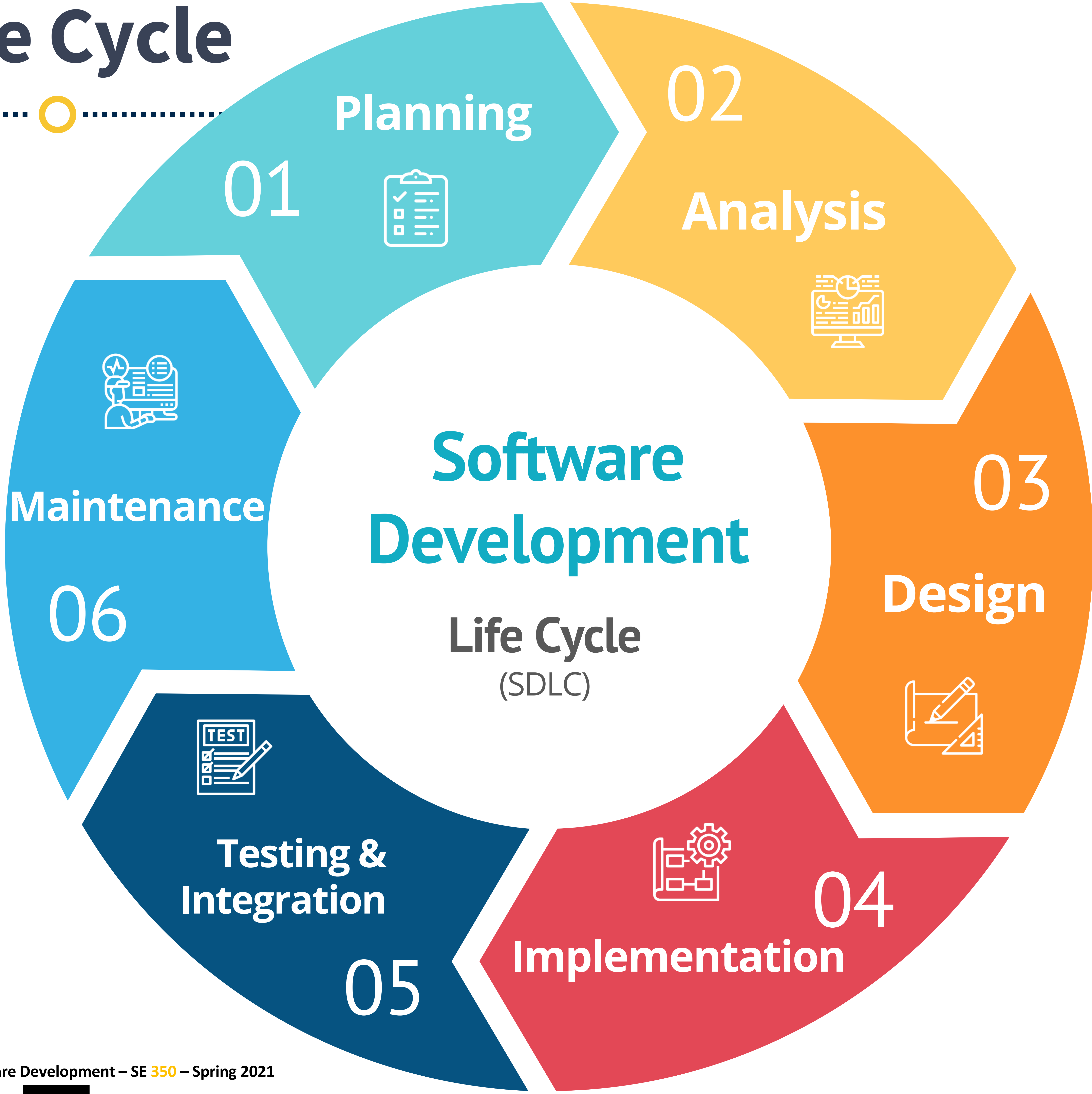
private getData() setData()

# Software Documentation

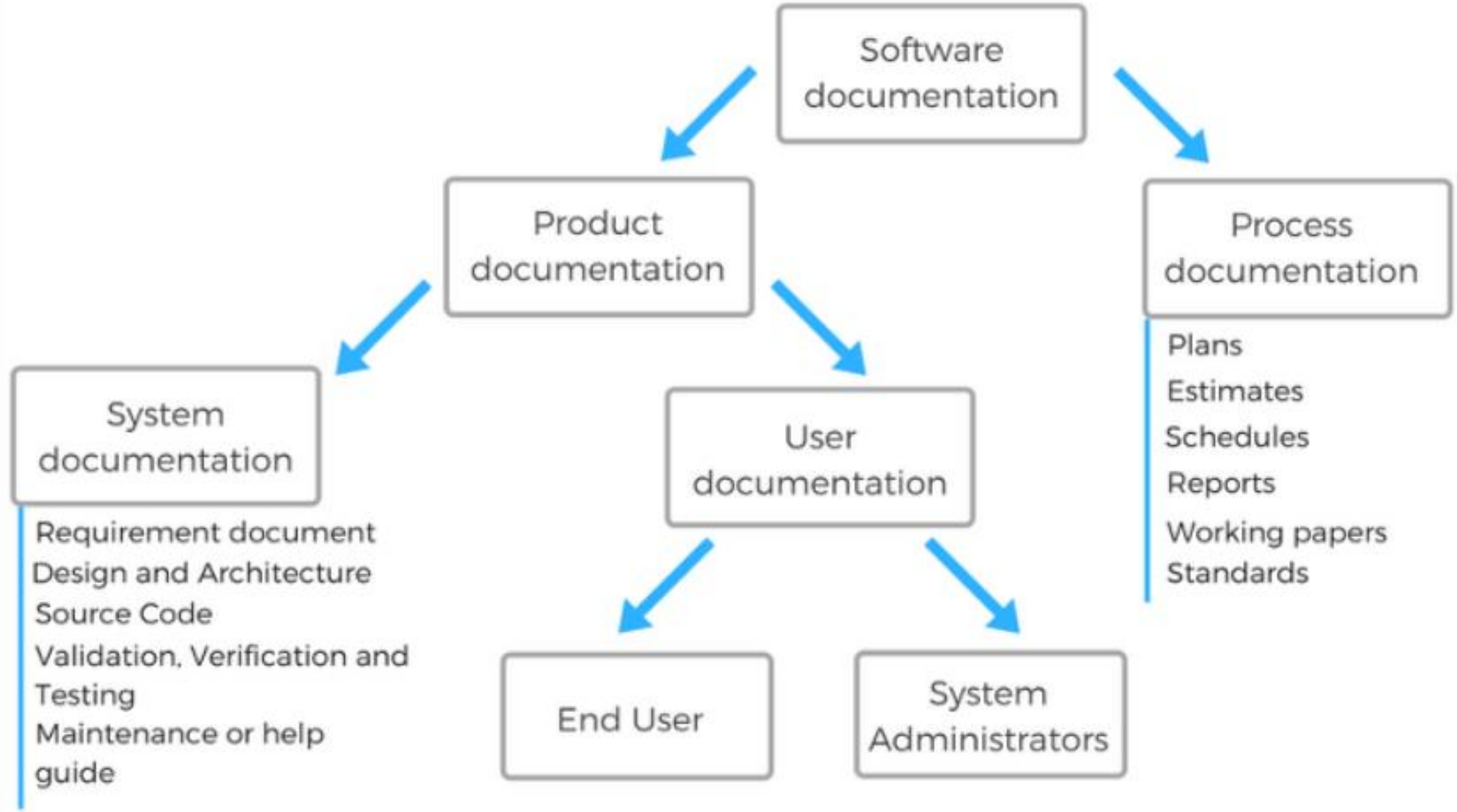
## Types

# Software Development Life Cycle

| Stage | Activity | Key Roles | Deliverables |
|---|---|---|---|
| **Planning** | Preliminary requirements' analysis, research, basic project vision and scope. | Customer, sales representatives, business analysts | Understanding the customer needs; basic project roadmap and technical recommendations. |
| **Analysis** | Identifying the project goals and functionality, finalizing the technical specifications, requirements and finding solutions to potentially challenging issues. | Customer, development team (business analysts, technical experts, project managers) | Complete functional and design specifications, Work Breakdown Structure (WBS), rough cost estimate. |
| **Design** | Creating basic system architecture and visual design (UI/UX). | Development team (architects, UX and UI experts, project managers) | Draft system architecture, final product design, and revised estimate. |
| **Implementation** | Software development process. | Development team (software engineers, architects, project managers) | Full-featured functioning software product. |
| **Testing** | Quality assurance process. | Development team (software engineers, QA engineers, project managers), Customer | Finalized software product of the required quality. |
| **Maintenance** | Deployment, support and updates. | Development team (software engineers, project managers) | Up-to-date software product. |



# Types of documentation

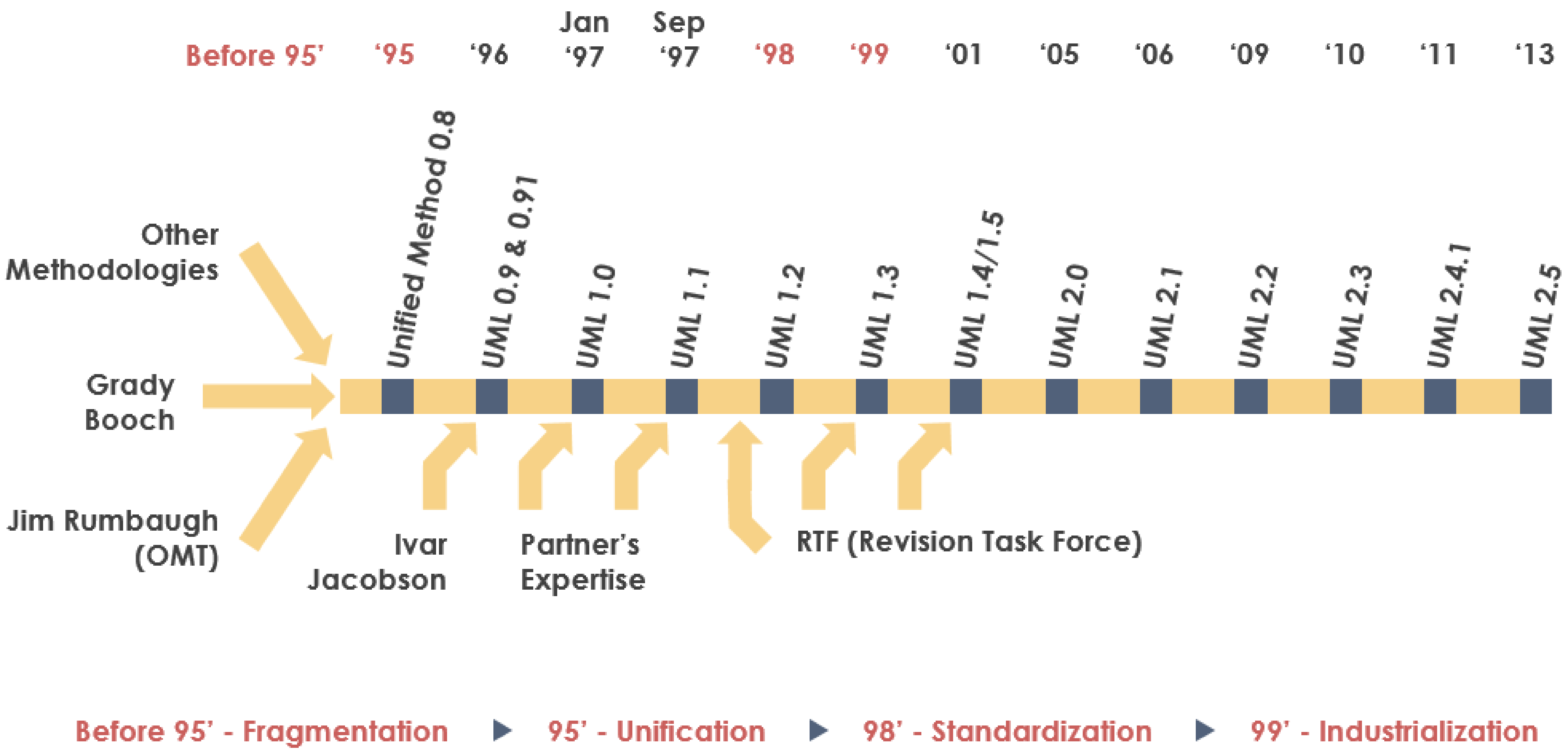


# Design Documentation

## UML

# UML Origin & History

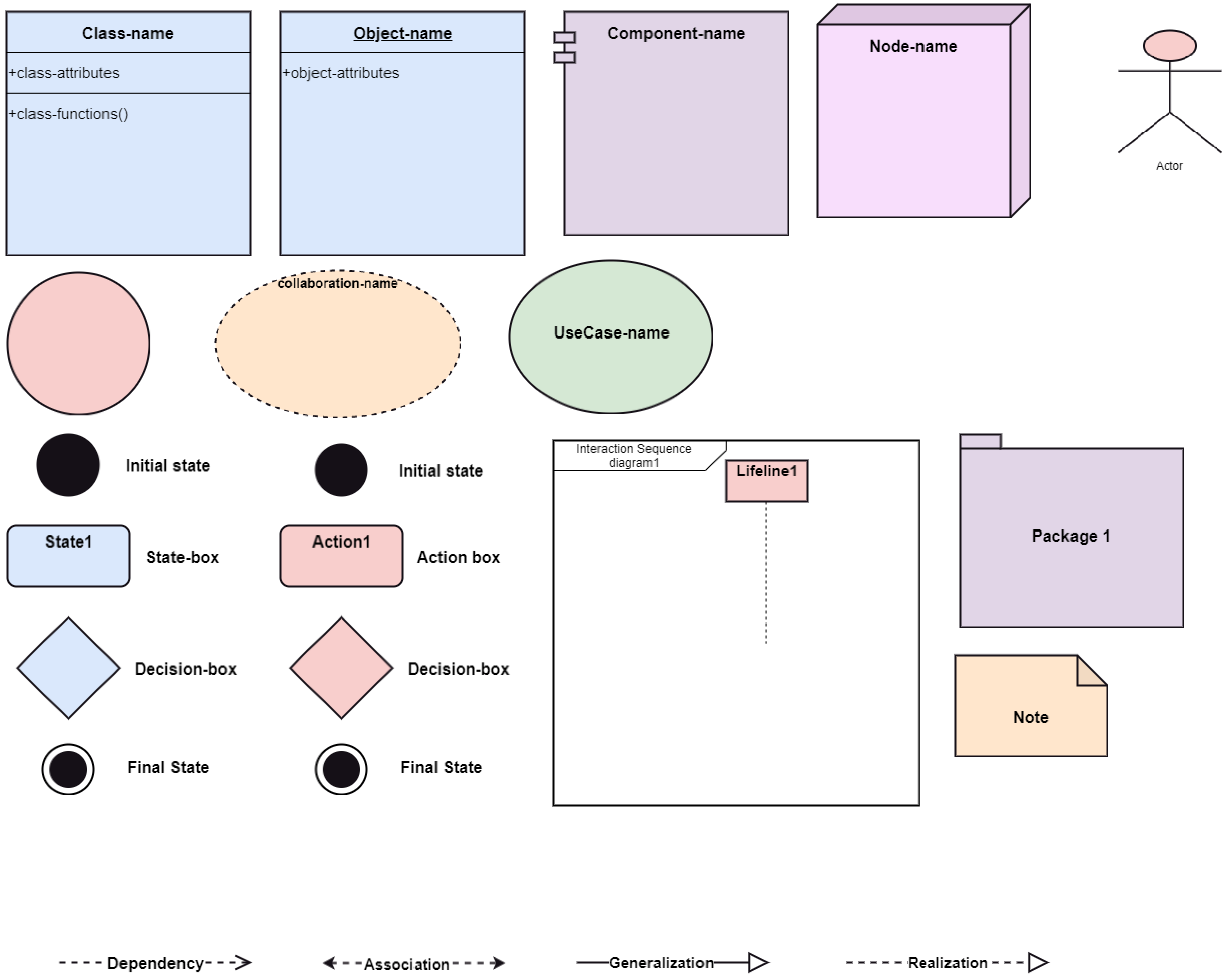
- **Unified Modeling Language**
  - visual modeling language
  - specifying, visualizing, constructing, and documenting
- **UML 1.0**
  - 1997
  - HP, IBM, Microsoft, Oracle, …
- **UML 2.5**
  - Current version
- **https://www.uml-diagrams.org/**
  - Version History
  - Complete References & Examples



Source: visual-paradigm.com

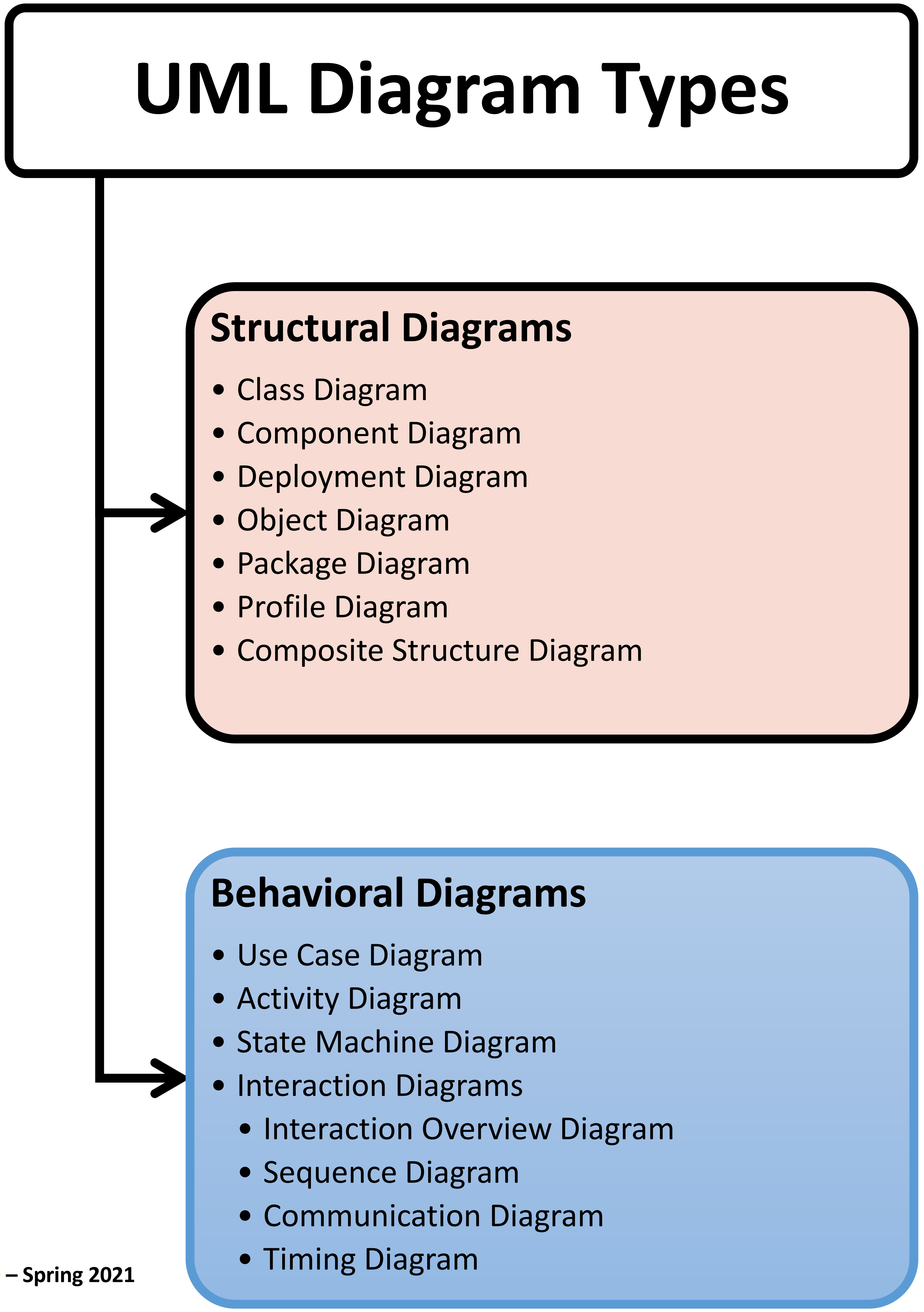
# UML Building Blocks

- **UML building blocks**
  - Things
    - Structural things
      - Class, Object, Interface, Collaboration, Use case, Actor, Component, Node
    - Behavioral things
      - State Machine, Activity Diagram, Interaction Diagram
    - Grouping things
      - Package
    - Annotational things
      - Note
  - Relationships
    - Dependency, Association, Generalization, Realization
  - Diagrams
    - Structural Diagram
    - Behavioral Diagram
      - Interaction Diagram



# UML Diagram Types

- **Structure diagrams**
  - Objects
  - Static view
- **Behavioral diagrams**
  - objects interaction
  - Dynamic view
  - **Interaction Diagrams**
    - flow between various use case elements of a system



## UML Diagram Types

### Structural Diagrams

- Class Diagram
- Component Diagram
- Deployment Diagram
- Object Diagram
- Package Diagram
- Profile Diagram
- Composite Structure Diagram

### Behavioral Diagrams

- Use Case Diagram
- Activity Diagram
- State Machine Diagram
- Interaction Diagrams
  - Interaction Overview Diagram
  - Sequence Diagram
  - Communication Diagram
  - Timing Diagram

# UML Class Diagram

## What is UML Class Diagram?

- static view of an application | types of objects | relationships between them

## Purpose?

- analyses and designs | responsibilities | base for other diagrams | forward and reverse eng.

## Benefits?

- complex systems | reduces the maintenance time | better understanding | desired code | helpful for the stakeholders

## Components

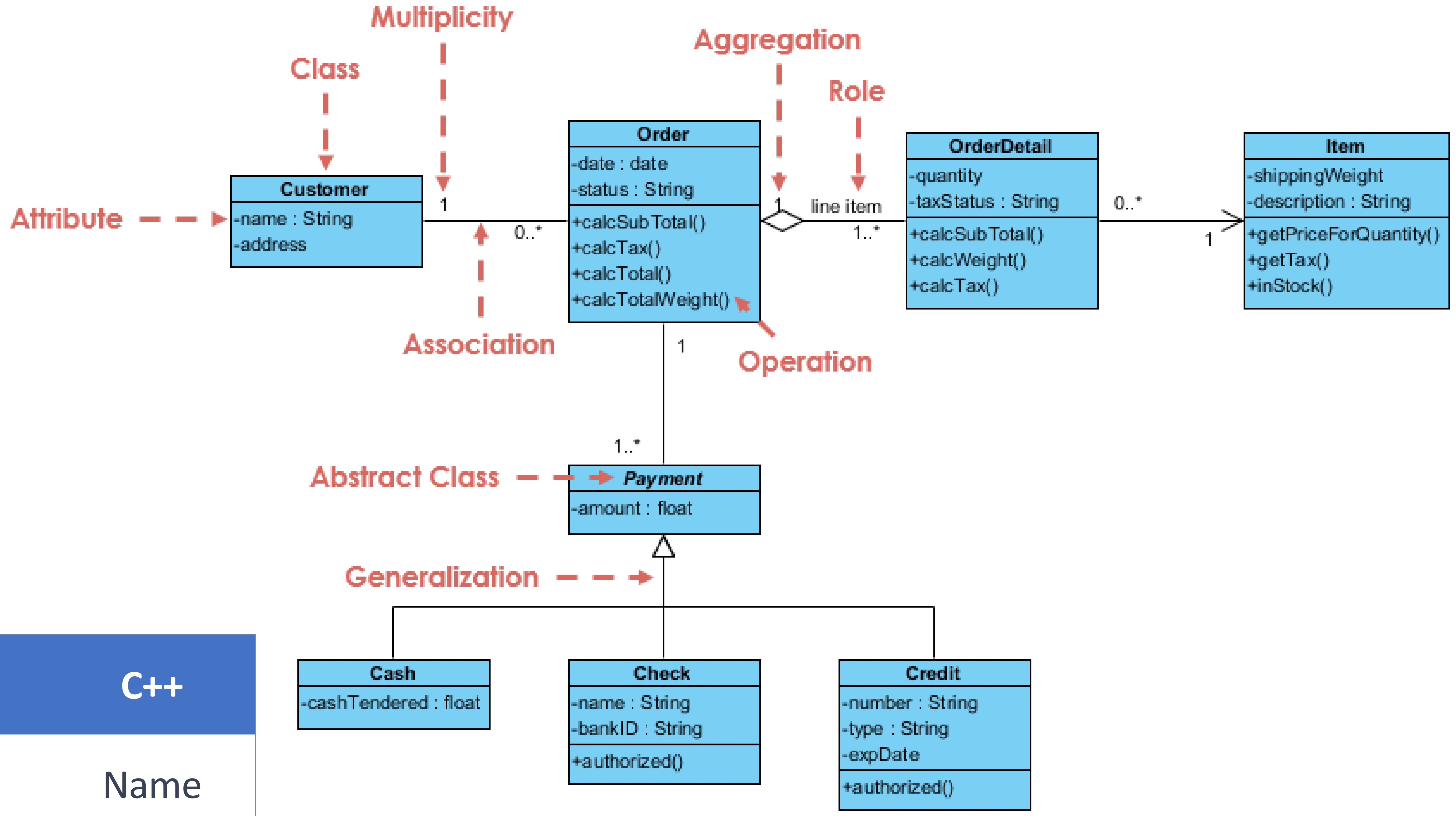
- Upper Section
- Middle Section
- Lower Section

ClassName

attributes

methods

| General | Java | C++ |
|---|---|---|
| Name | Name | Name |
| State | Variable | Members |
| Behavior | Methods | Functions |

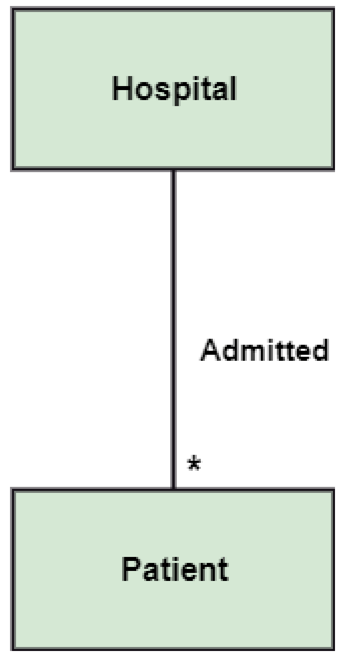


Class Diagram Example: Order System

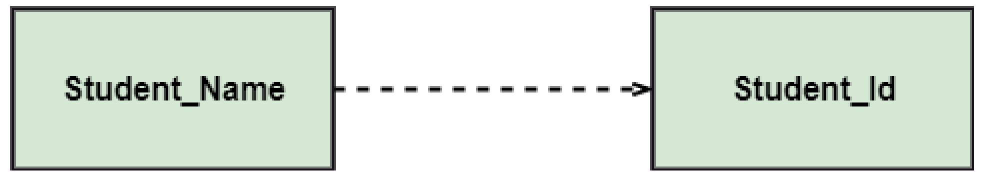
Source: visual-paradigm.com

# Class Diagram Relationships

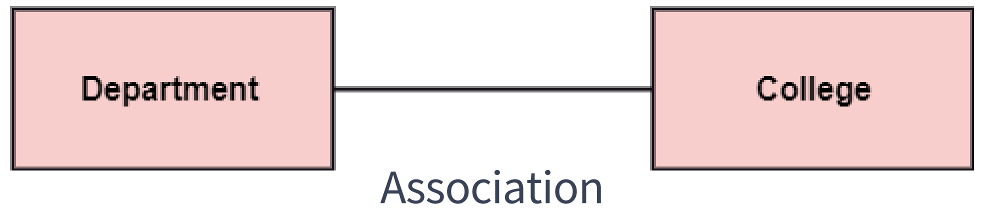
- **Relationships**
  - **Dependency**
    - One class is dependent to another class
    - “Uses-a”
    - Dotted line with arrowhead
  - **Generalization**
    - Inheritance
    - “Is-a-kind-of”, “is-a”
  - **Association**
    - “Has-a”
    - **Aggregation**
      - “Has-a”, “is part of”
    - **Composition**
      - has-a”, “part of”, “belongs-to”



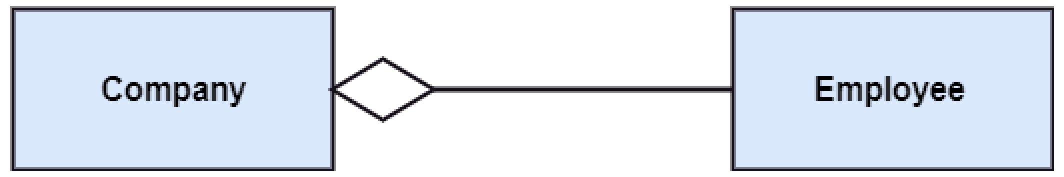
Multiplicity



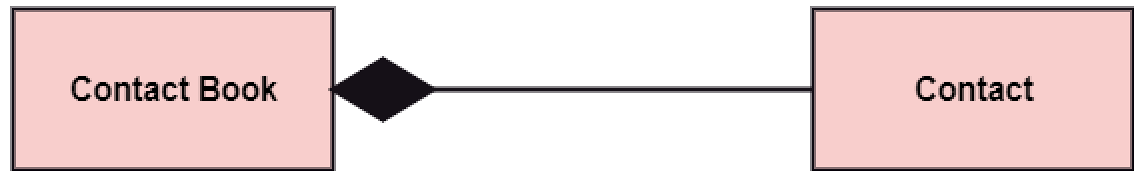
Dependency



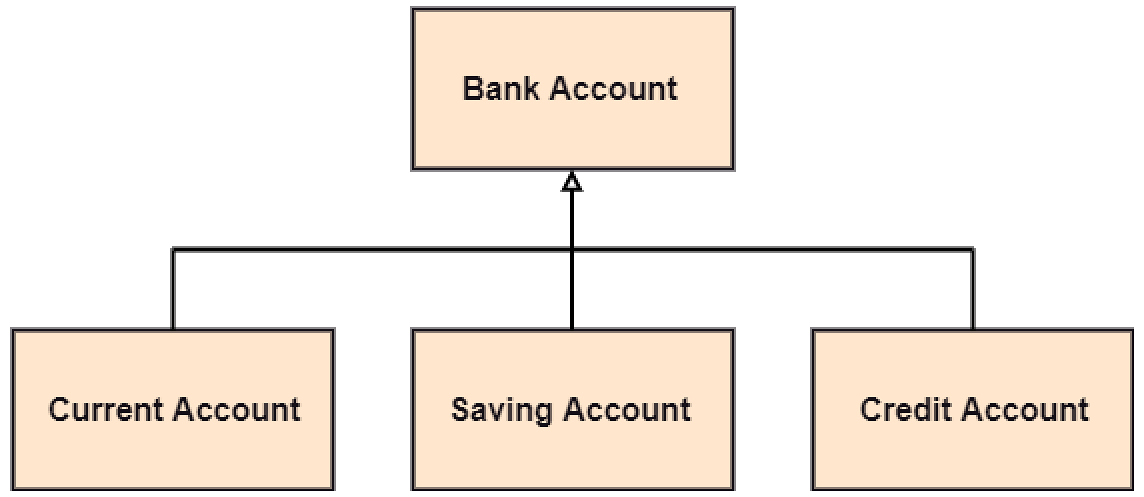
Association



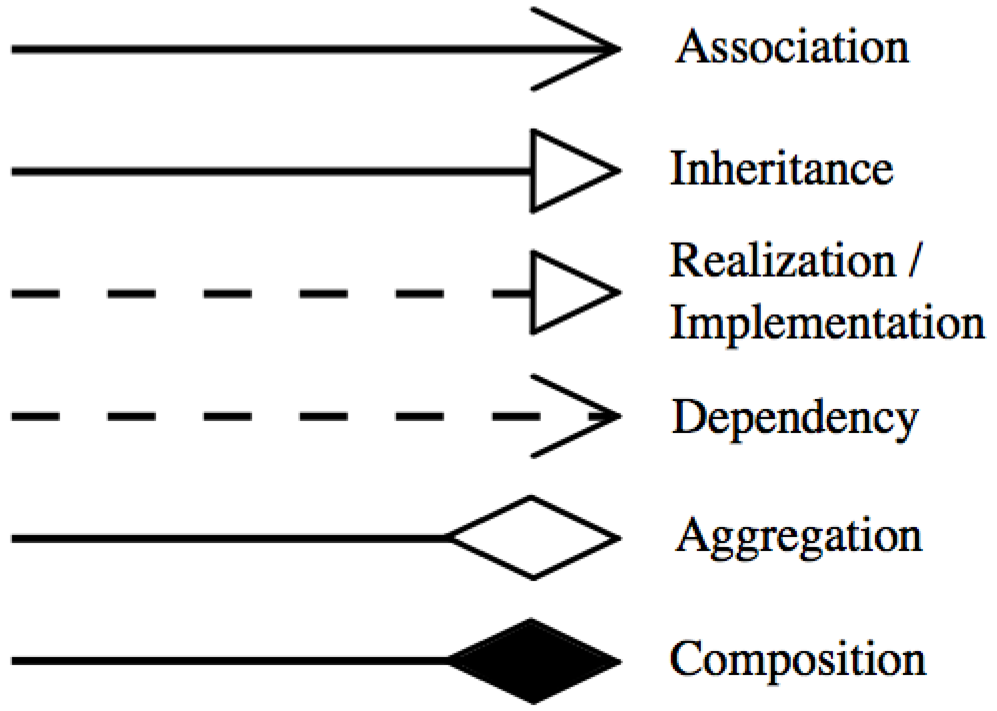
Aggregation



Composition



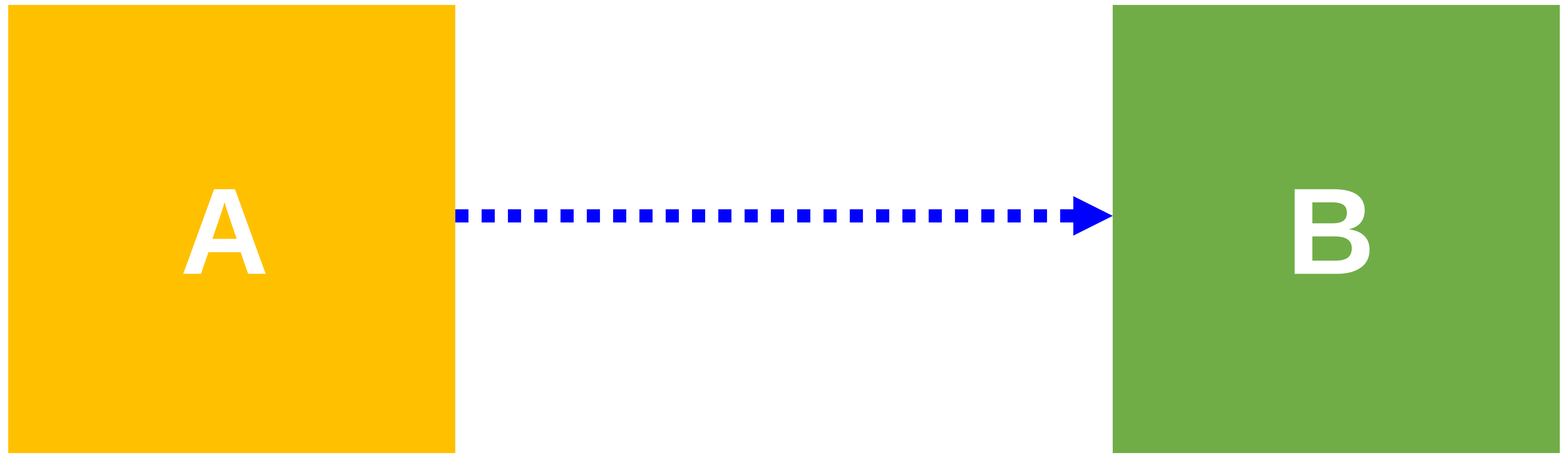
Generalization



Summary of types of relationships and their notation

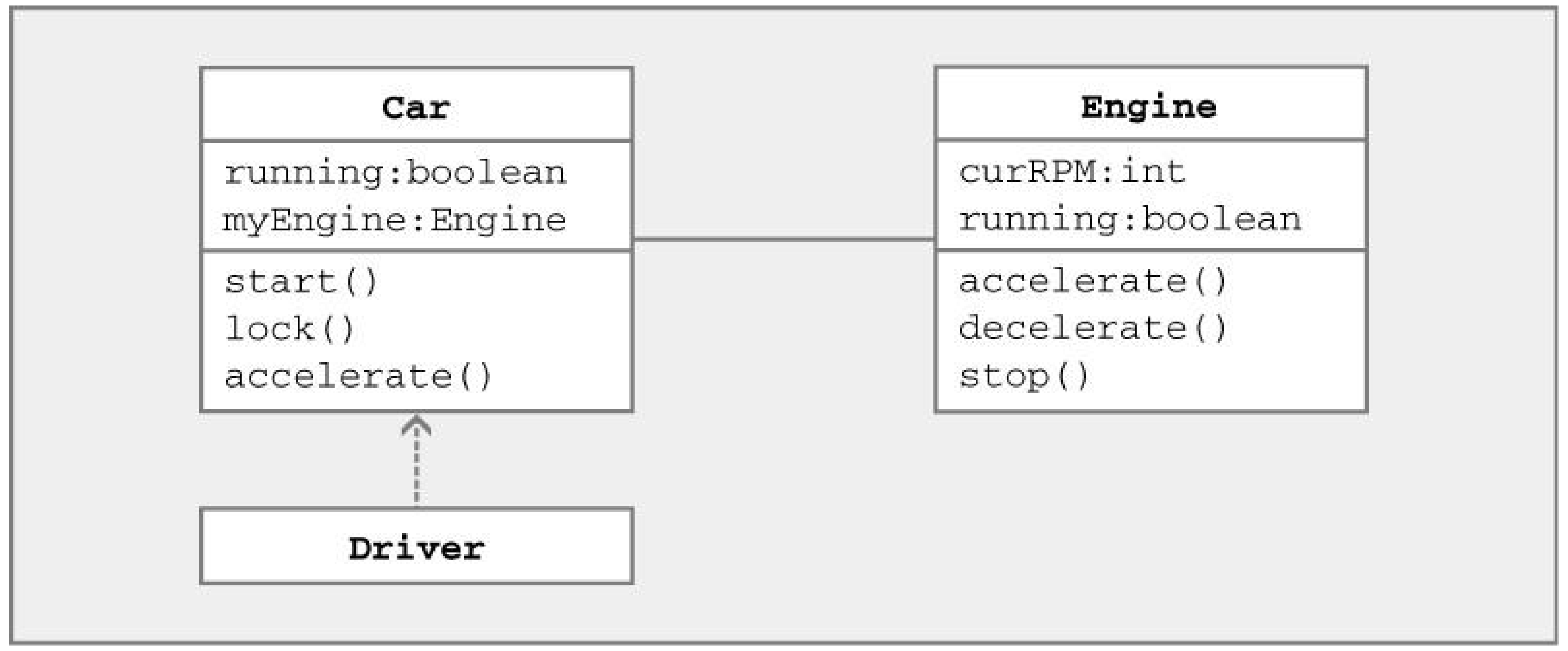
# Dependency

- **Dependency**
  - Denotes dependence between classes
  - Temporary
  - “uses a”
  - Dotted line with arrowhead
  - Always directed (Class A depends on B)
  - Caused by class methods. Maybe caused by:
    - Local variable
    - Parameter
    - Return value
  - Method in Class A temporarily “uses an” object of type Class B
  - Change in Class B may affect class A



**A** depends on **B**

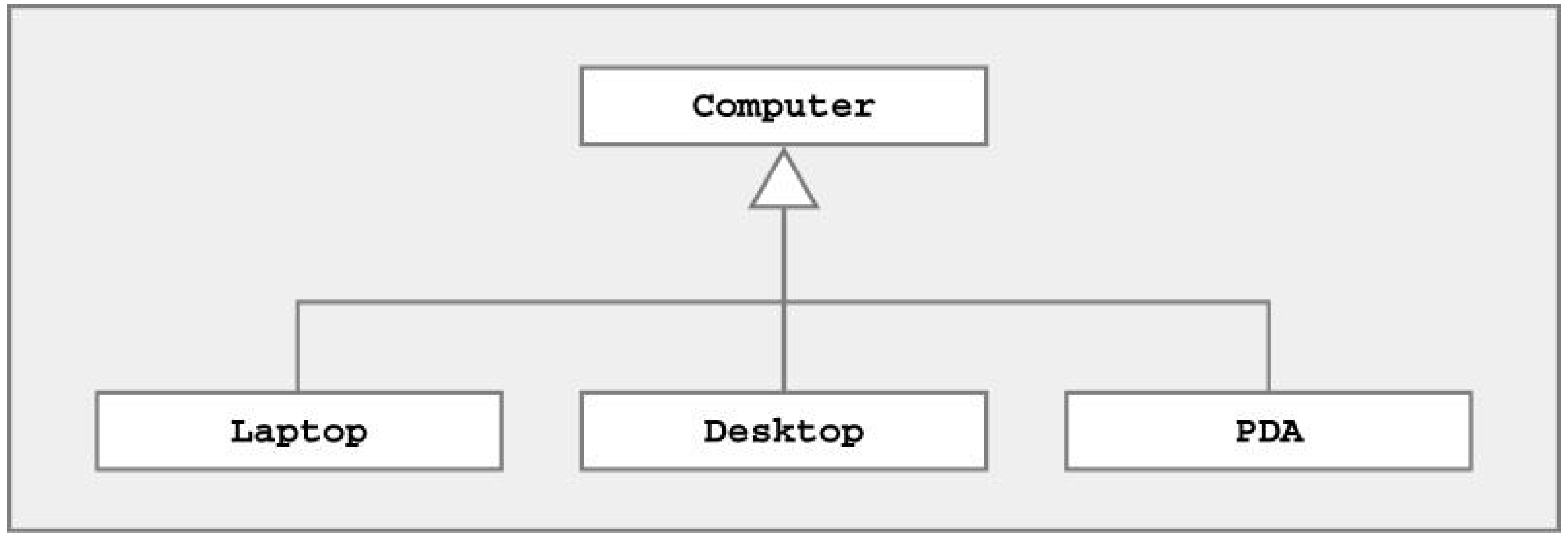
**A** uses object of class **B**



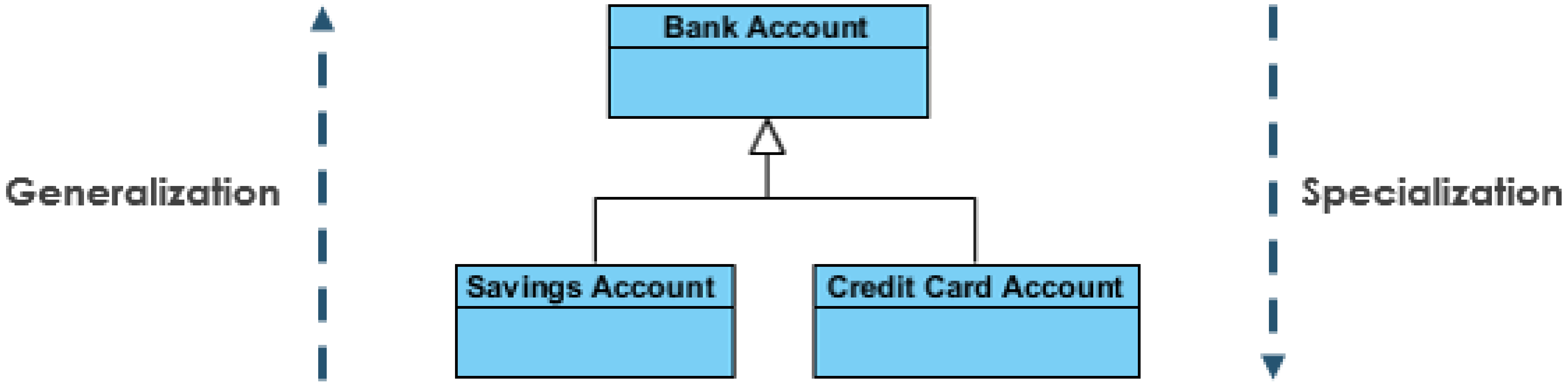
Example: Class Driver depends on Class Car

# Generalization

- **Generalization**
  - Denotes inheritance between classes
  - “is a” relationship
  - Solid line with open (triangular) arrowhead
- **Generalization vs Specialization**
- **Generalization vs Inheritance**

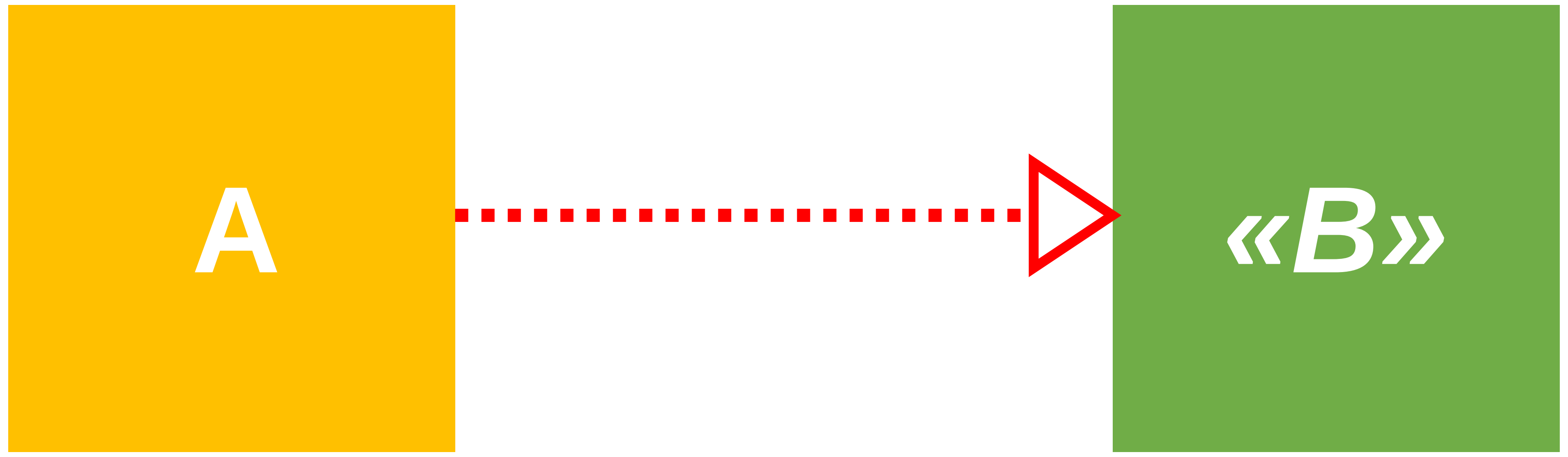


Example: Laptop, Desktop, PDA inherit state & behavior from Computers

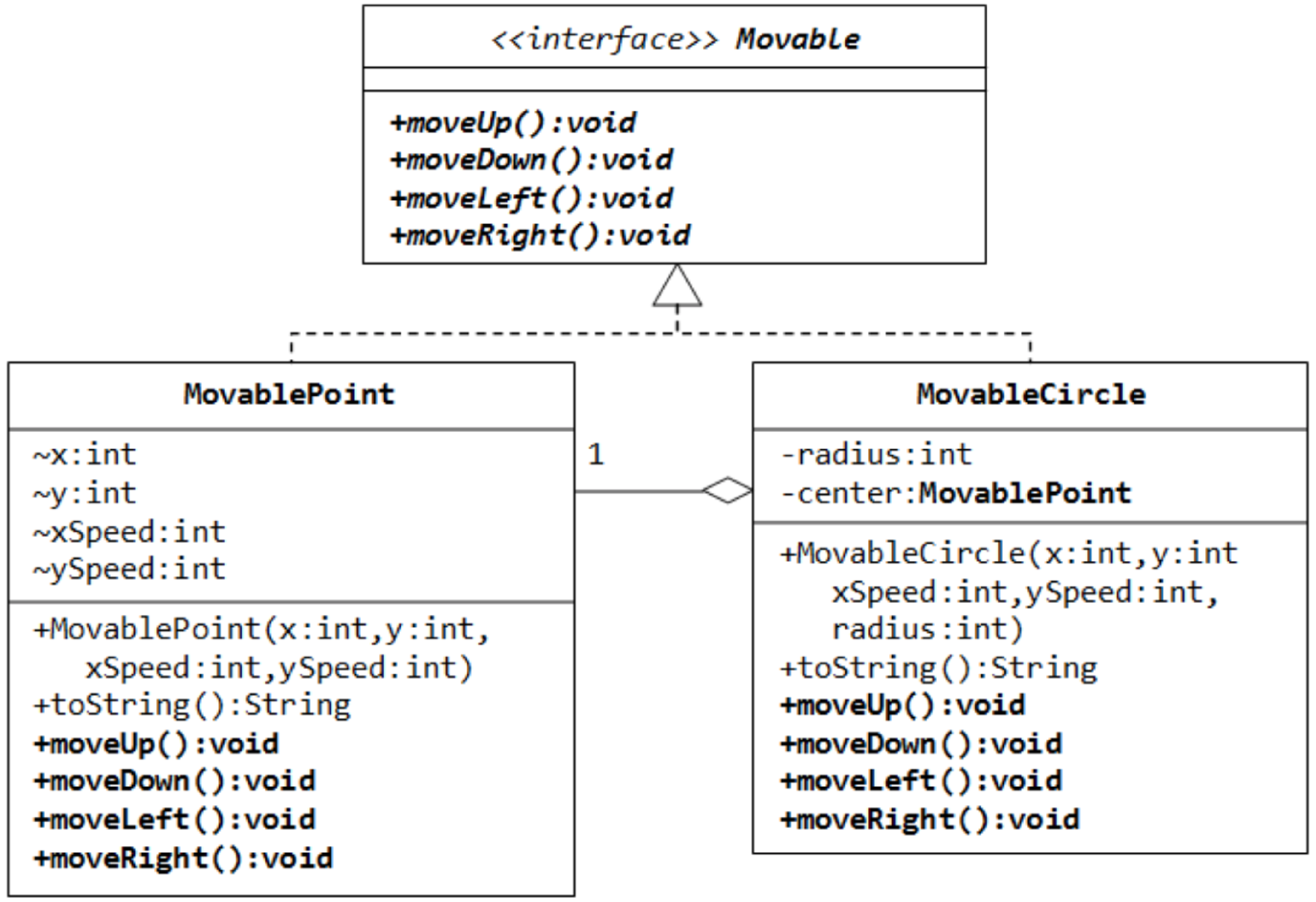


# Realization/Implementation

- **Realization/Implementation**
  - Denotes class implements Java interface
  - Dotted line with open (triangular) arrowhead



**A** implements interface **B**



*<<interface>>* ***Movable***

***+moveUp():void***
***+moveDown():void***
***+moveLeft():void***
***+moveRight():void***

**MovablePoint**

~x:int
~y:int
~xSpeed:int
~ySpeed:int

+MovablePoint(x:int,y:int,
xSpeed:int,ySpeed:int)
+toString():String
**+moveUp():void**
**+moveDown():void**
**+moveLeft():void**
**+moveRight():void**

1

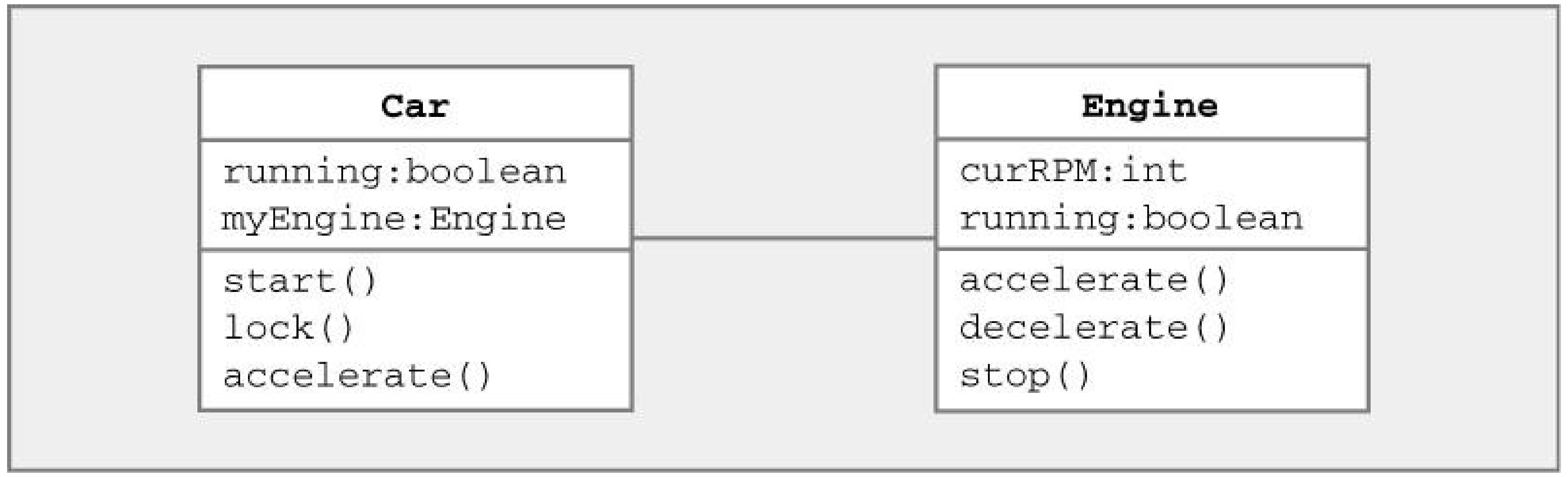
**MovableCircle**

-radius:int
-center:**MovablePoint**

+MovableCircle(x:int,y:int
xSpeed:int,ySpeed:int,
radius:int)
+toString():String
**+moveUp():void**
**+moveDown():void**
**+moveLeft():void**
**+moveRight():void**

# Association

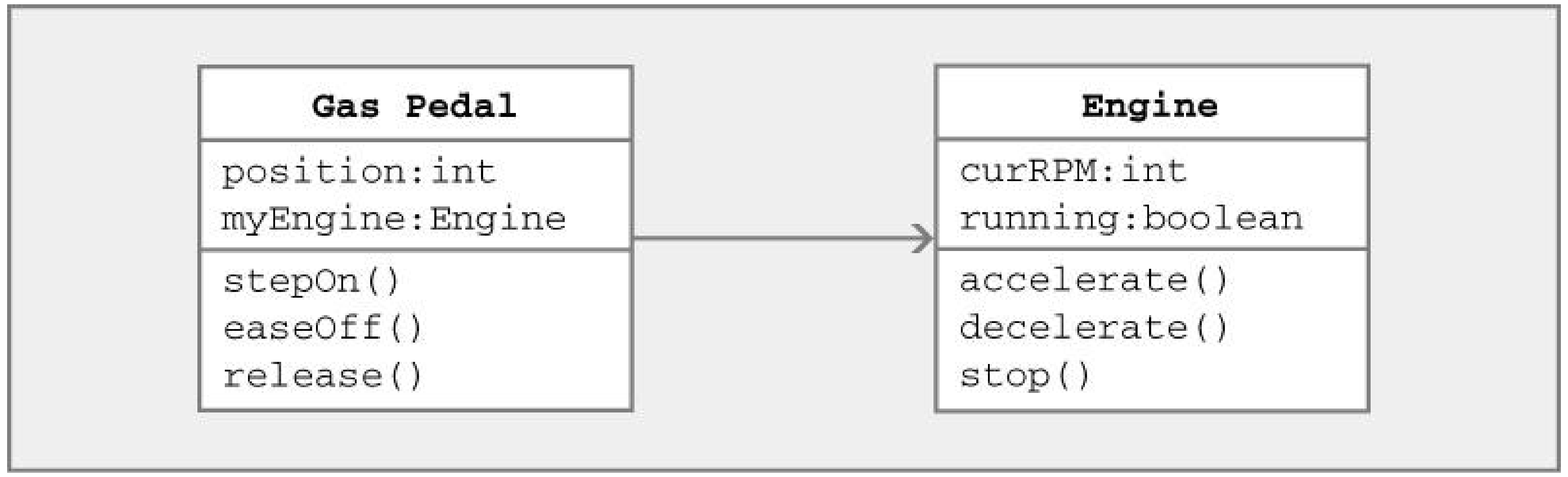
- **Association**
  - permanent, structural relationship
  - State of class A contains class B
  - Represented by solid line (arrowhead optional)
- **Navigation**
  - Represented by solid line with arrowhead
  - Denotes “has-a” relationship between classes
    - Example:
      - “Gas Pedal” has an “Engine”
      - **State** of Gas Pedal class contains instance of Engine class and can invoke its methods



```
Car
running:boolean
myEngine:Engine
start()
lock()
accelerate()
```

```
Engine
curRPM:int
running:boolean
accelerate()
decelerate()
stop()
```

Car and Engine classes know about each other



```
Gas Pedal
position:int
myEngine:Engine
stepOn()
easeOff()
release()
```

```
Engine
curRPM:int
running:boolean
accelerate()
decelerate()
stop()
```

Gas Pedal class knows about Engine class

Engine class doesn’t know about Gas Pedal class

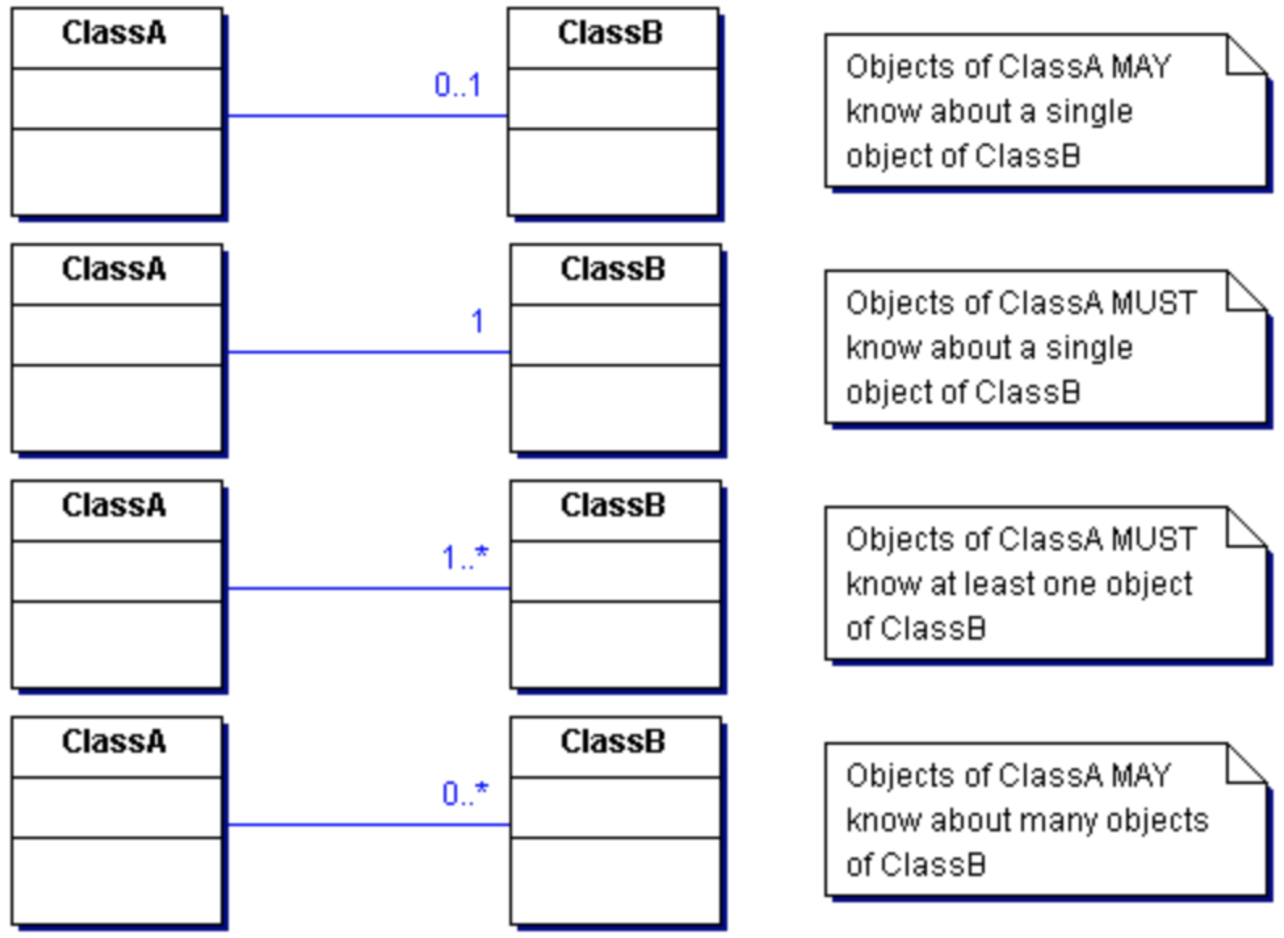
# Multiplicity (Cardinality) of Associations

## Multiplicity

- Denotes how many objects

## Notation

- ***** => 0, 1, or more
- **7** => exactly 7
- **3..8** => between 3 and 8, inclusive
- **2..*** => 2 or more



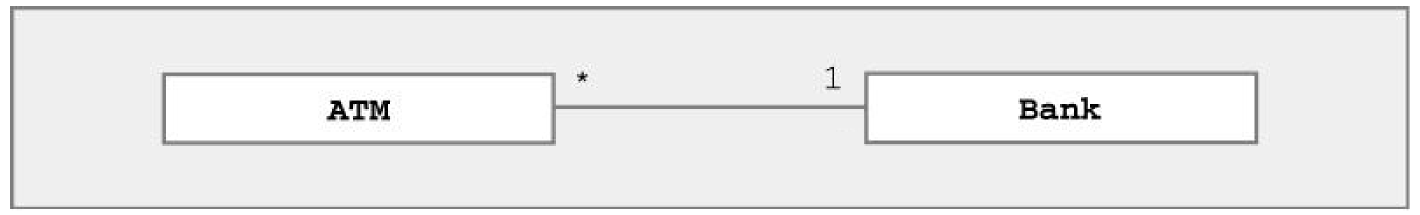
Objects of ClassA MAY know about a single object of ClassB

Objects of ClassA MUST know about a single object of ClassB

Objects of ClassA MUST know at least one object of ClassB

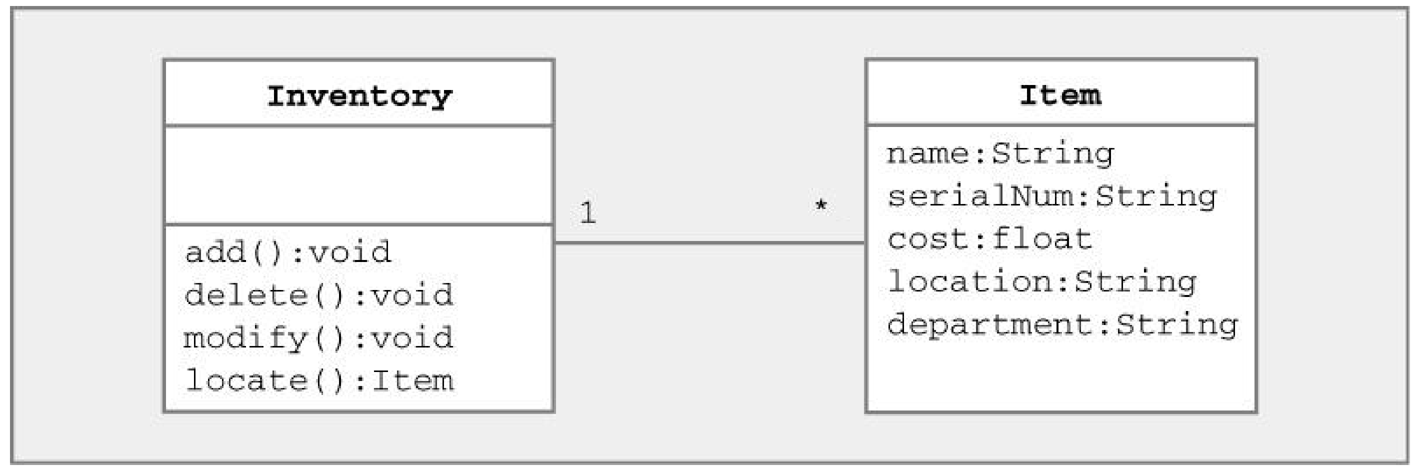
Objects of ClassA MAY know about many objects of ClassB

## Many to One

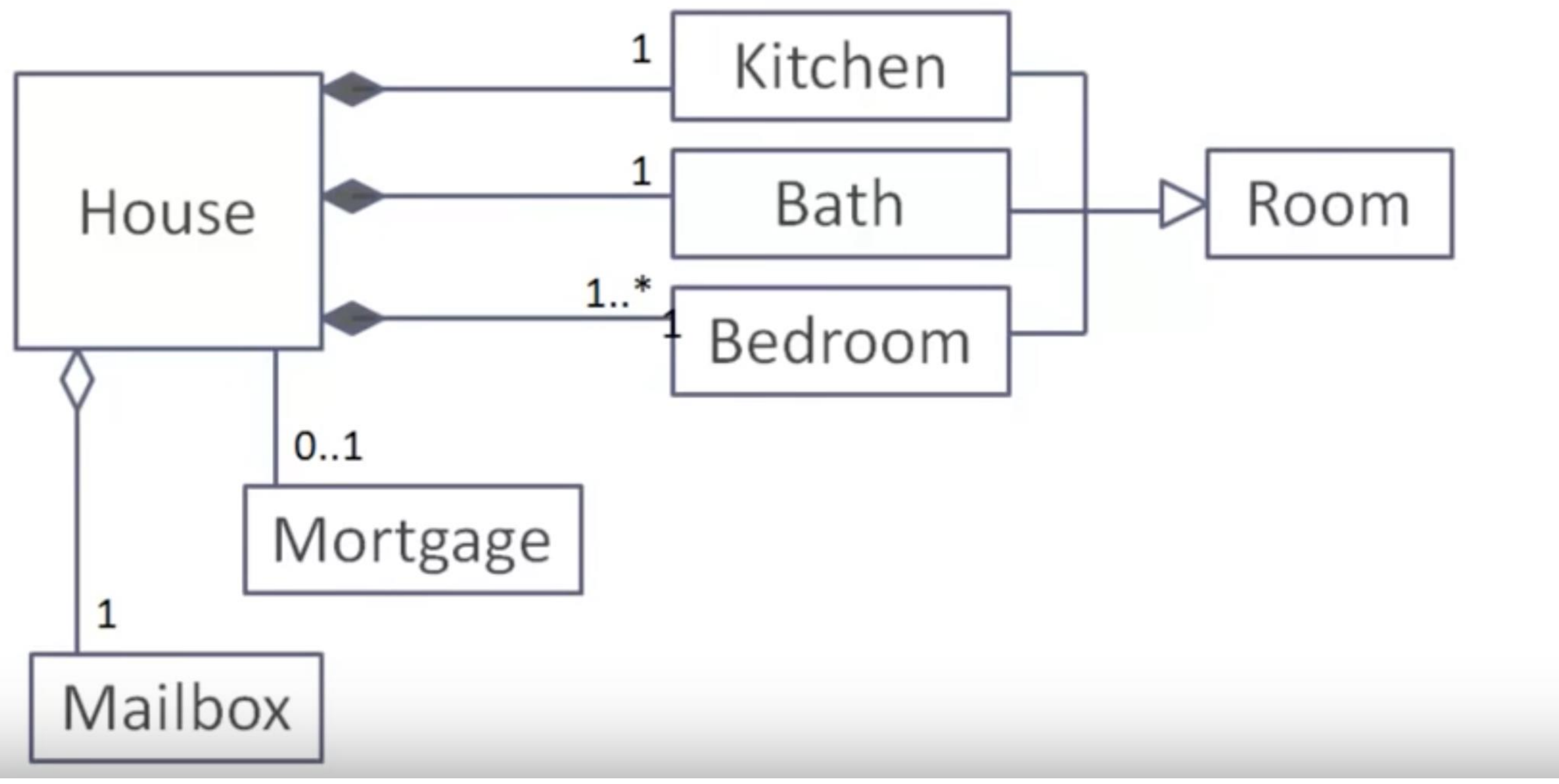


- Bank has many ATMs, ATM knows only 1 bank

## One-to-many



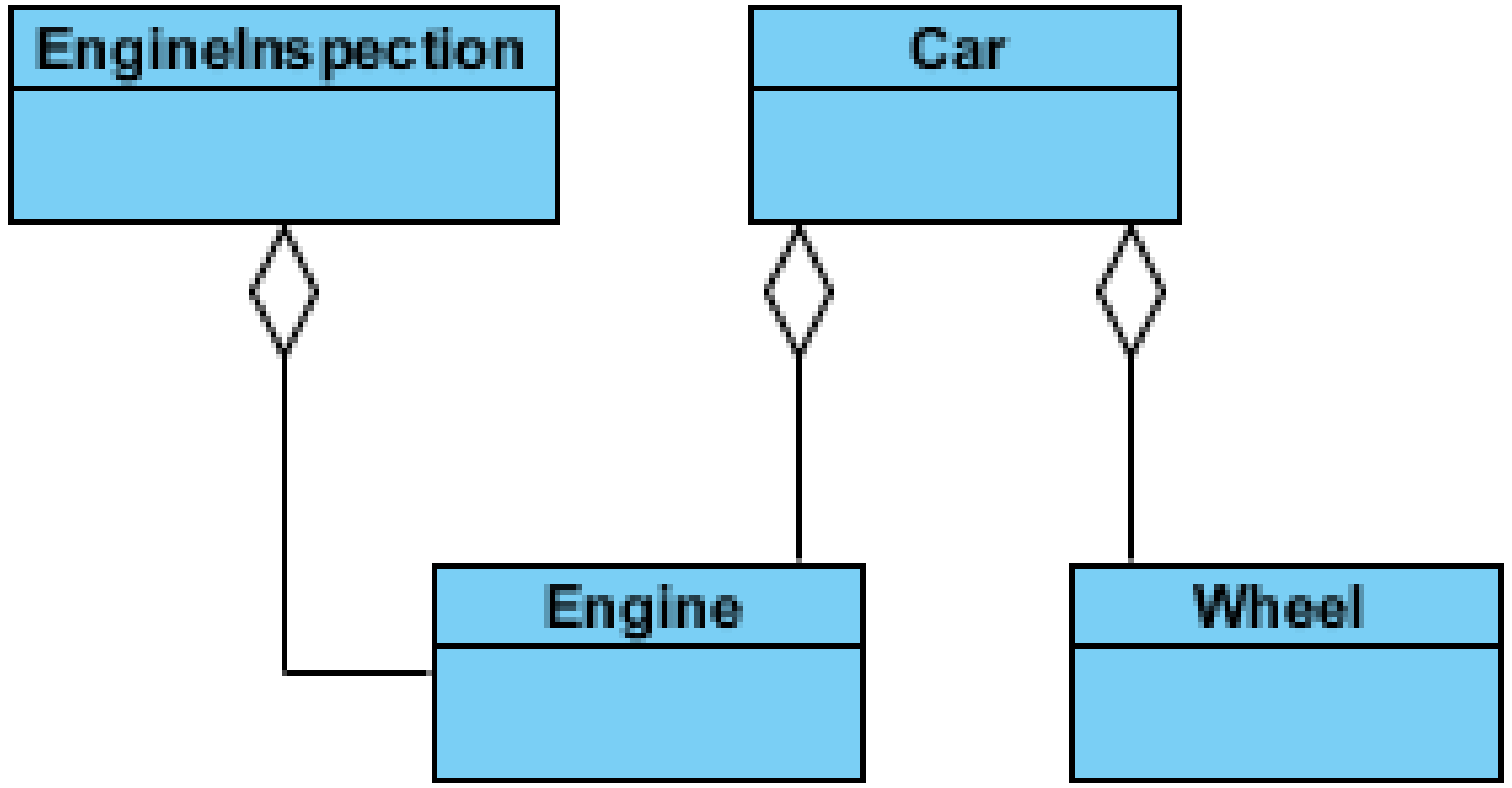
- Inventory has many items, items know 1 inventory



Example: A house has exactly one kitchen, exactly one bath, at least one bedroom (can have many), exactly one mailbox, and at most one mortgage (zero or one).

# Aggregation

- **Aggregation**
  - Special kind of association
    - whole- part model
  - unidirectional (One-way) relationship
    - “Has-a”, “is part of”
  - Only one class is dependent on the other
  - Both the entries can survive **individually**
  - Illustrate composition with a **hollow** diamond
    - The diamond end points toward the "whole" class
- **When to use Inheritance and Aggregation?**
  - Use property/behavior without modification or add functionality > Aggregation
  - Use and modify property/behavior and add functionality > Inheritance


EngineInspection
Car
Engine
Wheel