



# Java Programming 2

OOP with Java. SOLID principles

# Course Schedule

- Block 1: 9:00 – 10:30
- Pause: 10:30 – 10:45
- Block 2: 10:45 – 12:15
- Lunch: 12:15 – 13:00
- Block 3: 13:00 – 14:30
- Pause: 14:30 – 14:45
- Block 4: 14:45 – 16:15

# Where to Find The Code and Materials?

Java Academy projects and examples are available @GitHub:

<https://github.com/iproduct/course-java-web-2021>

# OOP Principles. SOLID Principles





# Agenda for This Session

- **Encapsulation**
  - What is Encapsulation?
  - Validation of input data;
  - Mutable and Immutable Objects;
  - Keyword final;
- **Inheritance**
  - Class Hierarchies;
  - Inheritance in Java;
  - Accessing Members of the Base Class;
  - Types of Class Reuse;
  - When to Use Inheritance;
  - Code reuse strategies – choosing inheritance vs. composition.
- **Abstraction**
  - Implementing Interfaces;
  - Creating and extending Abstract Classes;
  - Interfaces vs Abstract Classes;
- **Understanding Polymorphism**
  - Differences between method *overriding* and *overloading*.
  - Depending on abstractions, not implementations – *Dependency Inversion (DI) Principle*;
- **SOLID principles**

# Where to Find the Code?

Java Web Development projects and examples are available @ GitHub:

<https://github.com/iproduct/course-java-fd>

# Basic Concepts in OOP and OOAD

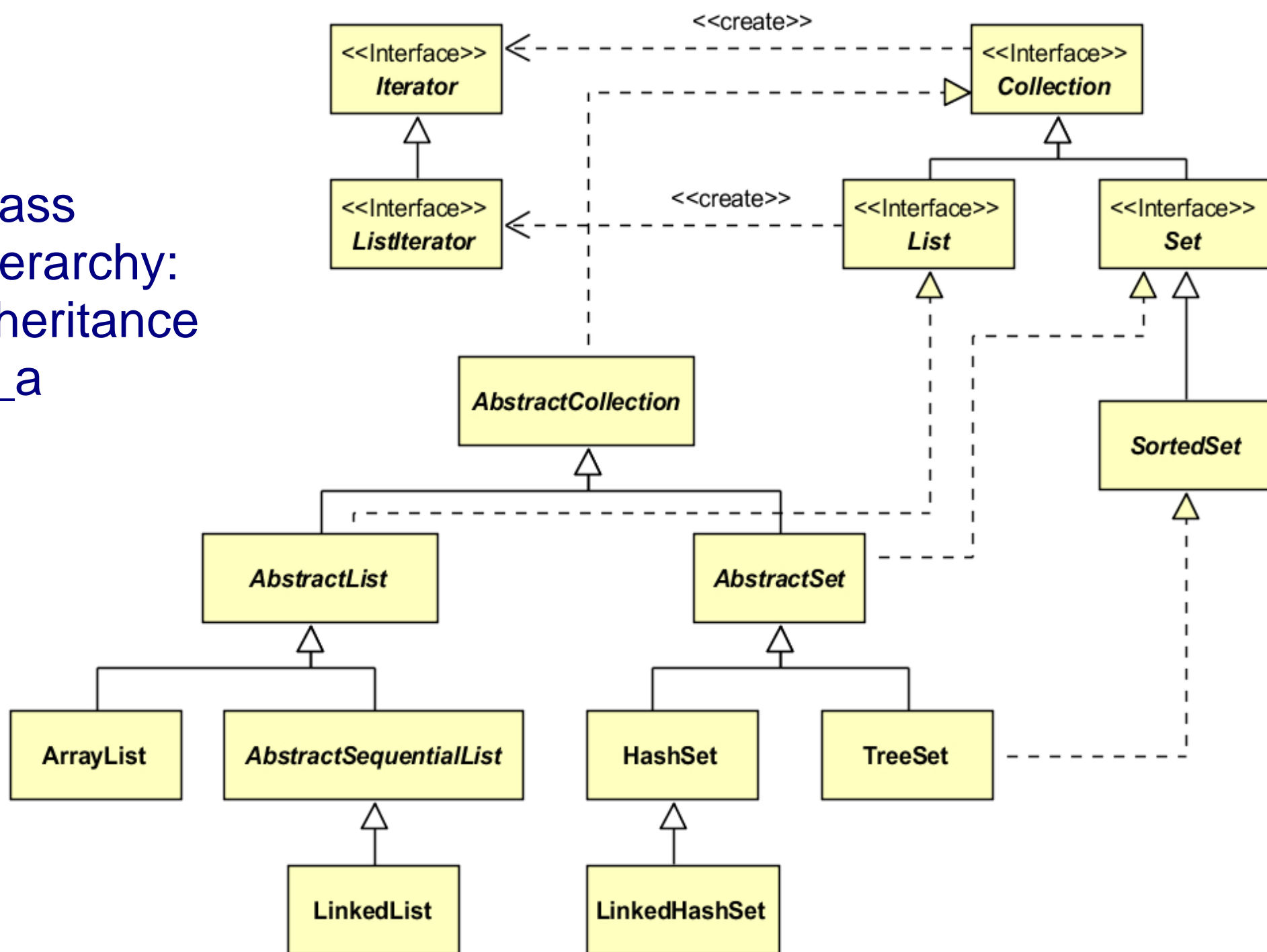
- **interface and implementation** – we divide what remains constant (contractual interface) from what we would like to keep our freedom to change (hidden realization of this interface)
- interface = **public**
- implementation = **private**
- This separation allows the system to evolve while maintaining backward compatibility to already implemented solutions, enables parallel development of multiple teams
- programming based on **contractual interfaces**

# Object-Oriented Approach to Programming

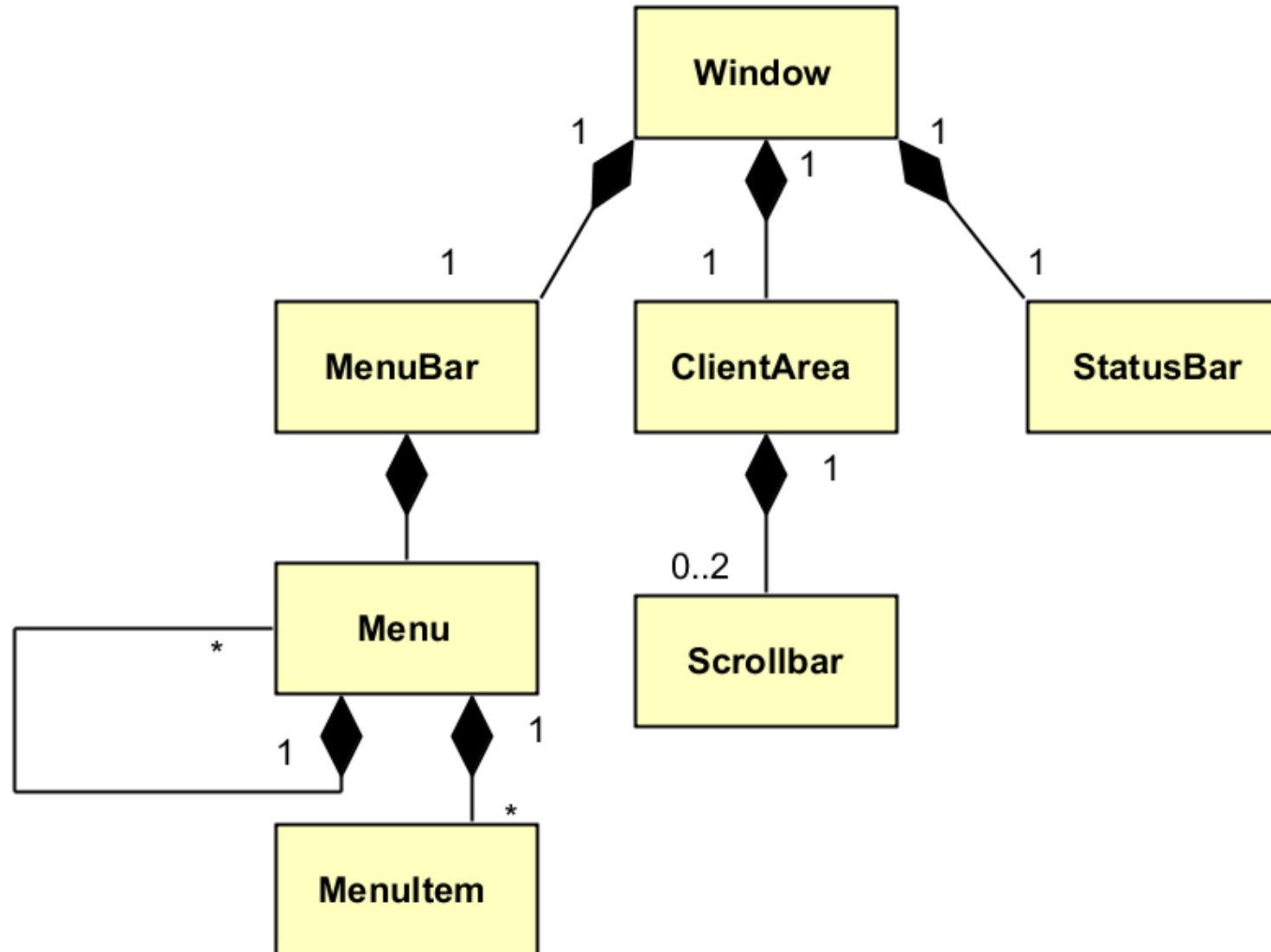
- Key elements of the object model [Booch]:
- **class, object, interface and implementation**
- **abstraction** – basic distinguishing characteristics of an object
- **encapsulation** – separating the elements of abstraction that make up its structure and behavior - interface and implementation
- **modularity** – decomposing the system into a plurality of components and loosely connected modules - principle: maximum coherence and the minimum connectivity
- **hierarchy** – class and object hierarchies



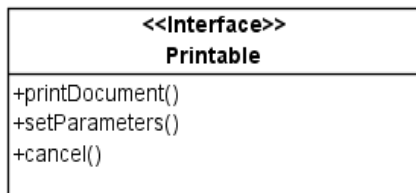
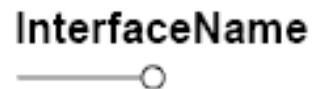
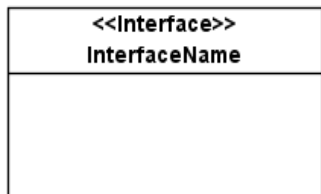
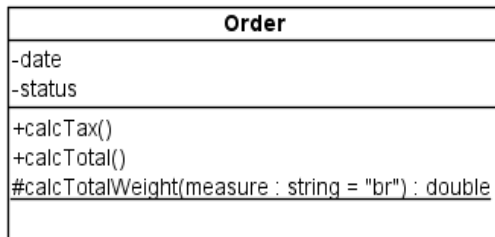
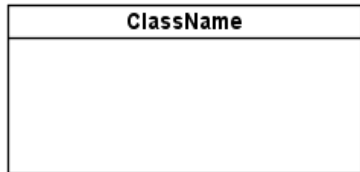
# Class Hierarchy: Inheritance is\_a



# Object Hierarchy: Composition, has\_a

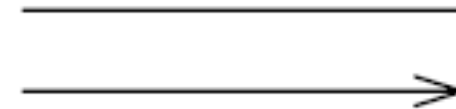


# Elements of Class Diagrams



- Types of connections:

- Association



- aggregation



- composition



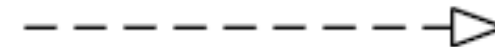
- dependence



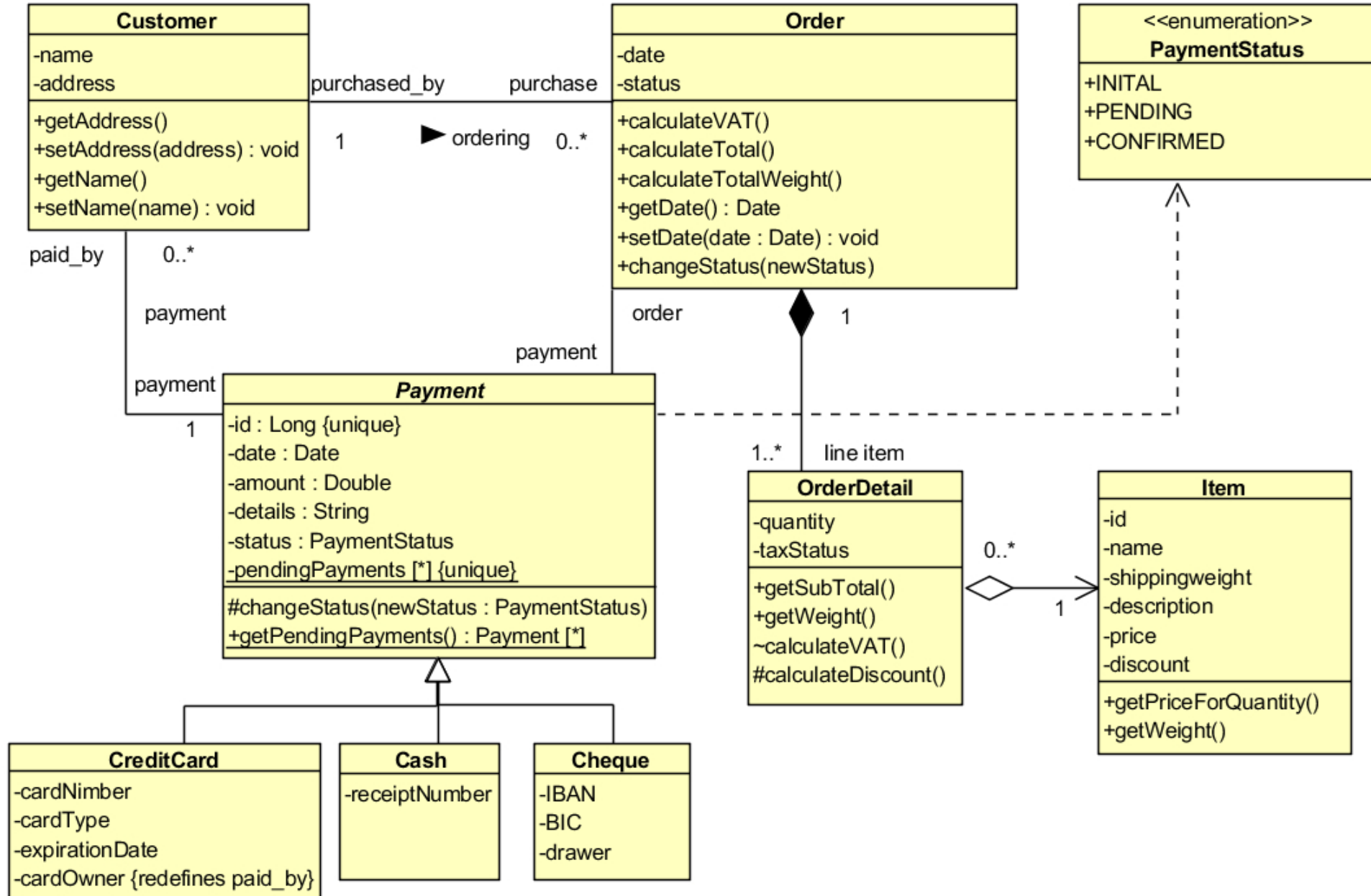
- generalization



- realization



# Class Diagram - 1



# Reusing Classes

- Advantages of **code reuse**
- Ways of implementation:
  - **Objects composition**
  - **Inheritance of classes (object types)**
- Building complex objects by **composition**
- Initializing the references:
  - **on declaration of the site**
  - **in the constructor**
  - **before using (lazy initialization)**

# Class Inheritance - I

- Inheritance realization in Java™ language
  - Keyword **extends**
  - Keyword **super**
- Initialization of objects inheritance:
  - 1) **base class**; 2) **inherited class**
  - Calling the default constructors
  - Calling constructors with arguments
- Combining **composition** and **inheritance**

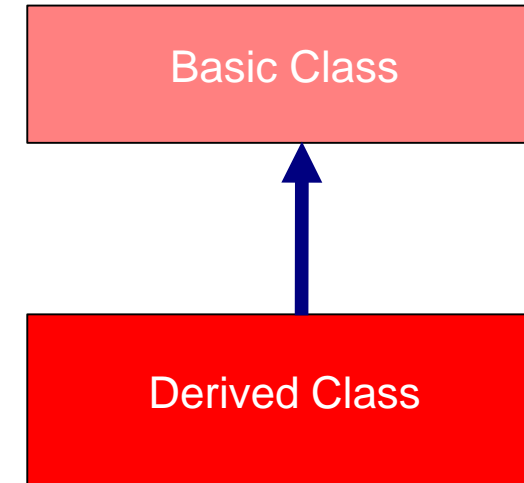


# Class Inheritance - II

- Clearing of objects – realization in Java™
- Overloading and overriding methods of base class in derived classes
- When to use composition and when inheritance?
  - ✓ Do we need the interface of the base class?
  - ✓ Connection Type - „there is“ and „it is“?

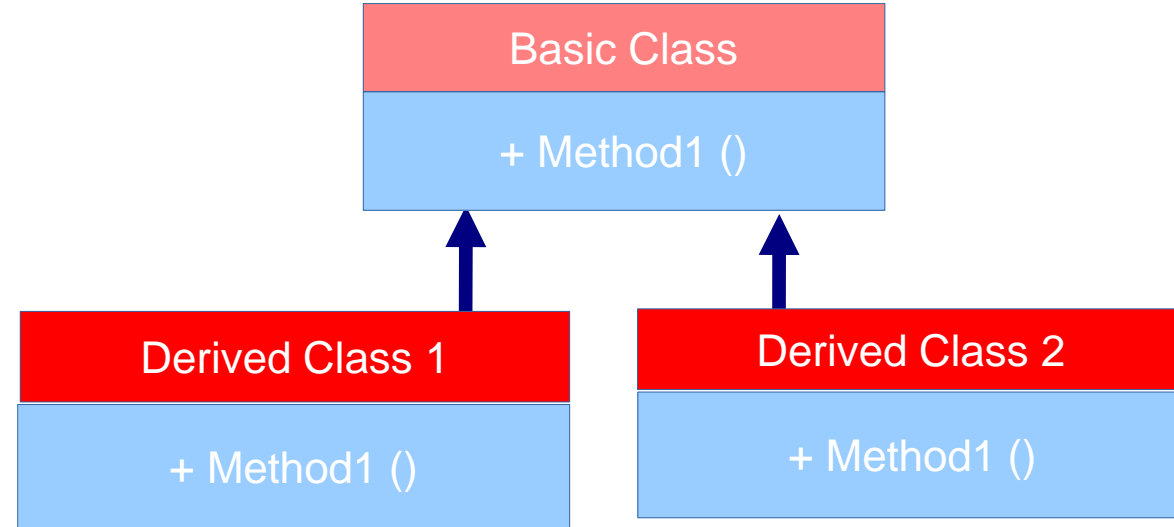
# Class Inheritance - III

- Protected methods
- Upcasting
- Keyword final
  - Final data – defining constants
    - simple data type
    - objects
    - empty fields
    - arguments
  - Final methods
  - Final classes



# Polymorphism - I

- Upcasting



- Abstract methods and classes – abstract
- Order of constructor calls
- Inheritance and expansion

# Polymorphism - II

- **Polymorphism** – by default, unless the method is declared as static or final (private methods become automatically final)
- When constructing objects with inheritance each object cares about its attributes and **delegate initialization of parental attributes on parental constructor or method**

# Interfaces and Multiple Inheritance

- Interfaces – keywords: **interface**, **implements**
- Multiple inheritance in Java
- Interface expansion through inheritance
- Constants (static final)
- Interface incorporation

# Advantages of Using Interfaces

- **Interfaces** cleanly separate requirements type of the object from many possible implementations and make our code more universal and usable
- **Reusable Design Pattern: Adapter** – It allows to adapt existing realization to an interface that is required in our application
- **Inheritance (expansion) of interfaces**
- **Reusable Design Pattern: Factory Method** – creating reusable client code, isolated from the specifics of the particular server implementation



# Inner Classes - I

- **Inner Classes** group logically related classes and control their visibility
- **Closures** – internal class has a constant connection to containing outside class and can access all its attributes and even final arguments and local variables (if defined in the method or block)
- Inner classes can be **anonymous** if used once in the program. Construction.
- Reference to the object from an external class - **.this** and creating an object from internal class in the context of containing object of the outer class - **.new**

# Inner Classes - II

- **Inner Classes**
  - defined in an external class
  - defined in method
  - defined in a block of operators
  - access to the attributes of the outer class and to the arguments of the method which are defined in
- **Anonymous inner classes**
  - implementing public interface
  - inheriting class
  - instance initialization
  - static inner classes

# SOLID design principles of OOP

- **Single responsibility principle** - a class should only have a single responsibility, that is, only changes to one part of the software's specification should be able to affect the specification of the class.
- **Open–closed principle** - software entities should be open for extension, but closed for modification.
- **Liskov substitution principle** - Objects in a program should be replaceable with instances of their subtypes without altering the correctness of that program.
- **Interface segregation principle** - Many client-specific interfaces are better than one general-purpose interface.
- **Dependency inversion principle** - depend upon abstractions, not concretions.

# Resources

- SOLID Principles in Wikipedia – <https://en.wikipedia.org/wiki/SOLID>

# Thank's for Your Attention!



Trayan Iliev

IPT – Intellectual Products & Technologies

<http://iproduct.org/>

<http://robolearn.org/>

<https://github.com/iproduct>

<https://twitter.com/trayaniliev>

<https://www.facebook.com/IPT.EACAD>