Mahidol University
Faculty of Information
and Communication Technology

LECTURE 14

Object Oriented Design

ITCS123 Object Oriented Programming

Dr. Siripen Pongpaichet
Dr. Petch Sajjacholapunt
Asst. Prof. Dr. Ananta Srisuphab

(Some materials in the lecture are done by Aj. Suppawong Tuarob)

Ref: Java Concepts Early Objects by Cay Horstmann

Semester 2/2023 18 April 2024

Object-Oriented Development

- Object-oriented *Analysis*, *Design* and *Programming* are related but distinct.
  - **OOA** is concerned with developing an *object model* of the application domain.
    - What are different objects?
    - What should each object be able to do?
  - **OOD** is concerned with developing an object-oriented *system* model to implement requirements.
    - How different objects interact with each other?
  - **OOP** is concerned with *realizing (implementing)* an OOD using an OO programming language such as Java or C++.
    - How to implement the system?

Object-Oriented Development (OOD)

Program Development Processes

Program Development Processes

- The creation of software involves four basic activities:

1. Establishing the *requirements*
2. Creating a *design*
3. *Implementing* the code
4. *Testing* the implementation

- These activities are not strictly linear – they overlap and interact

1. Establishing the requirement

- *Software requirements* specify the tasks that a program must accomplish.
  - What to do, not how to do it.
- Often an initial set of requirements is provided, but they should be *critiqued and expanded.*
- It is *difficult* to establish detailed, unambiguous, and complete requirements.
- Careful attention to the requirements can *save significant time* and expense in the overall project.

2. Creating a design

- A *software design* specifies how a program will accomplish its requirements
- That is, a software design determines:
  - How the solution can be *broken down into manageable pieces ?*
  - What each piece will do ?

- [**High Level Design**] An object-oriented design determines which classes and objects are needed and specifies *how they will interact.*
- **[Low level design]** includes how individual methods will accomplish their tasks.

3. Implement the Code

- *Implementation* is the process of translating a design into source code.

- ***Novice programmers*** *often think that writing code is the heart of software development, but actually it should be the least creative step.*

- Almost all-important decisions are made *during requirements and design* stages.

- Implementation should focus on coding details, including style guidelines and *documentation.*

Testing the Implementation

- *Testing* attempts to *ensure* that the program will solve the intended problem under all the constraints *specified in the requirements.*
- A program should be thoroughly tested with the *goal of finding errors.*
  - *Corner cases*
- *Debugging* is the process of determining the cause of a problem and fixing it.

Object-Oriented Development (OOD)

OOP Development Activities

OOP Development Activities

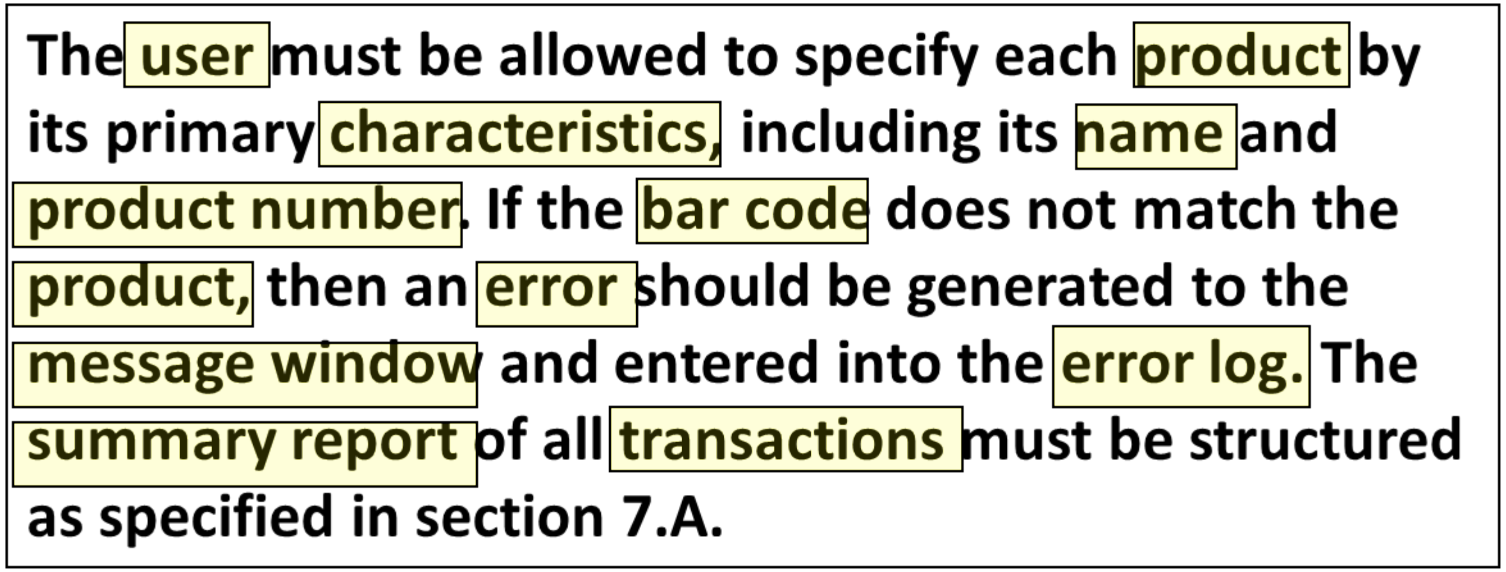
1. Identifying Classes and Objects
2. Identifying Variables and Methods
3. Identifying Class Relationships
4. ~~Interfaces~~
5. ~~Enumerated Types Revisited~~
6. ~~Method Design~~
7. ~~Testing~~
8. ~~GUI Design and Layout~~

1. Identifying Classes and Objects

- The core activity of object-oriented design is determining the classes and objects that will make up the solution
- The classes may be part of a class library, reused from a previous project, or newly written
- One way to identify potential classes is to identify the objects discussed in the requirements
- Objects are generally nouns, and the services that an object provides are generally verbs

1. Identifying Classes and Objects

- A partial requirements document:



The user must be allowed to specify each product by its primary characteristics, including its name and product number. If the bar code does not match the product, then an error should be generated to the message window and entered into the error log. The summary report of all transactions must be structured as specified in section 7.A.

Of course, not all nouns will correspond to a class or object in the final solution

1. Identifying Classes and Objects

Guidelines for Discovering Objects

1.1 Limit responsibilities of each analysis class.

1.2 Use *clear and consistent names* for classes and methods.

1.3 Keep analysis classes *simple.*

1. Identifying Classes and Objects

1.1 Limit Responsibilities

- Each class should have *a clear and simple purpose* for existence.
- Having classes with too many responsibilities make them difficult to understand and maintain.
- A good test for this is *trying to explain the functionality* of a class in a few sentences.

1. Identifying Classes and Objects

1.2 Use Clear and Consistent Names

- Companies sometimes spend *millions just to change their name* into a catchier one. You should give a similar effort to let your classes and methods have *suitable names*.
- Class names should be *nouns*.
- If you could not find a good name, this could mean the *boundaries of your class is too fuzzy.*
- Having too many simple classes is acceptable, but please ensure that they have good, descriptive names.

1. Identifying Classes and Objects

1.3 Keep Classes Simple

- To design a class, at the beginning, your imagination should not be crippled with *worrying about details* like object relationships.

1. Identifying Classes and Objects

Class Characteristic

- Remember that a class represents a *group (classification) of objects* with the same behaviors
- Generally, classes that represent objects should be given names that are *singular nouns.* Examples: `Coin`, `Student`, `Message`
- A class represents the concept of one such object.
- We are free to instantiate as many of each object as needed.

1. Identifying Classes and Objects

- Sometimes it is challenging to decide whether something should be represented as a class.
  - **For example:** Should an *employee's address* be represented as (1) *a set of instance variables* or as (2) *an* `Address` *object* ?
- The more you examine the problem and its details the clearer these issues become.
- When a class becomes too complex, it often should be *decomposed into multiple smaller classes* to distribute the responsibilities.

1. Identifying Classes and Objects

- In general, we typically define classes with an appropriate level of detail. Thus, it may not be necessary to create a small class to represent every single entity. **For example**: It may be unnecessary to create separate classes for each type of appliance in a house E.g. `Refrigerator, Microwave, DishWasher`.
- It may be sufficient to define a more general `Appliance` class with appropriate instance data E.g. `Appliance (type = "Refrigerator")`

"Designing class is all depends on the details of the problem being solved"

OOP Development Activities

1. Identifying Classes and Objects
2. Identifying Variables and Methods
3. Identifying Class Relationships

2. Identifying Variables and Methods

- Part of identifying the classes we need is the process of *assigning characteristics (variables) and responsibilities* (Method) to each class.

- Every *activity* that a program must accomplish must be represented by *one or more variables+methods* in one or more classes

- We generally use nouns for variables and *verbs* for the names of methods

- In early stages it is not necessary to determine every method of every class – begin with *primary responsibilities* and evolve the design.

“Perfection is the enemy of {progress, productiveness, good, etc.}” - **Many people**
“Good enough is better than perfect” - **Gretchen Rubin**

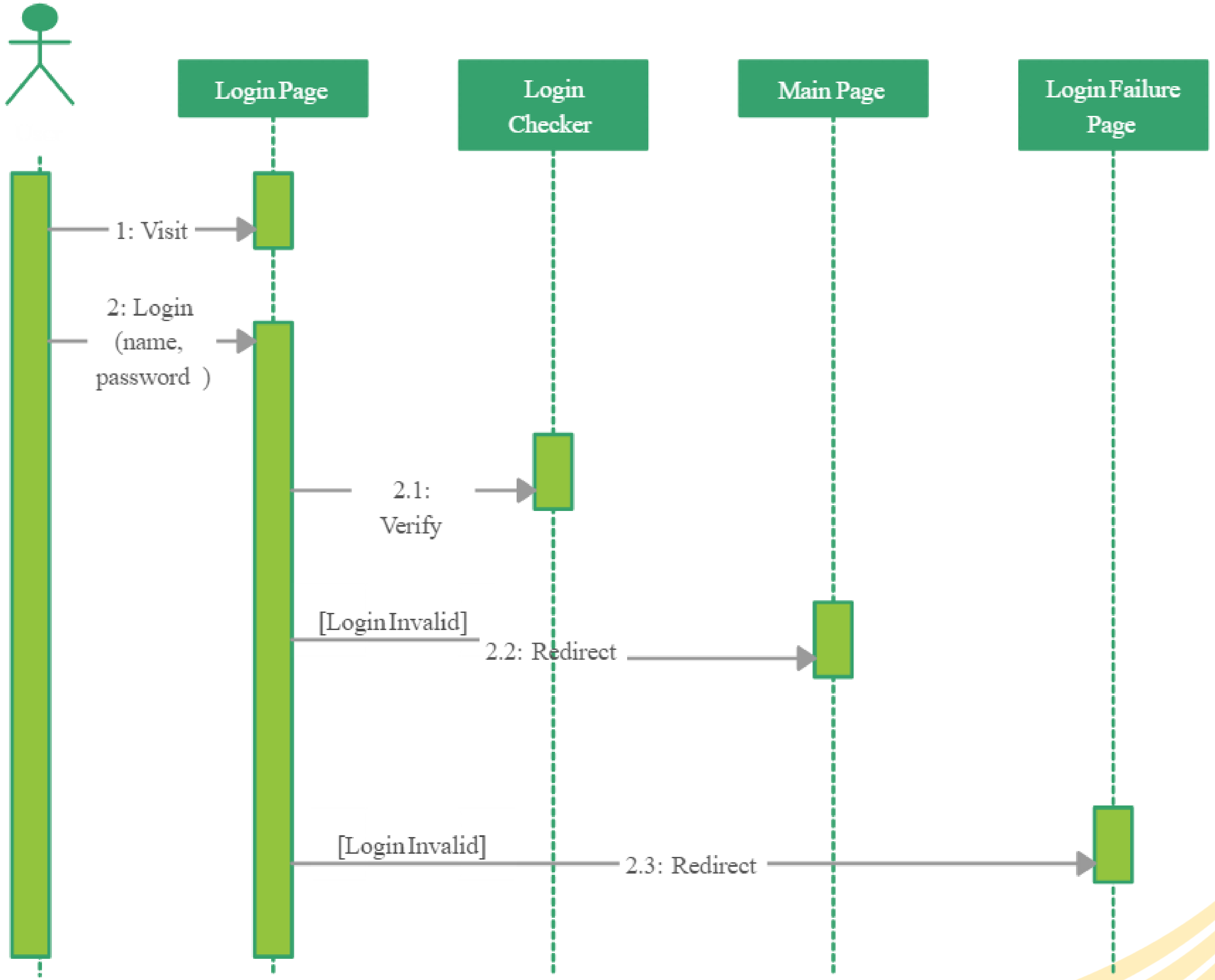
2. Identifying Variables and Methods

Describe Behavior (Method)

- The set of methods also dictate *how your objects interact* with each other to produce a solution.
- *Sequence diagrams is a tool that* can help tracing object methods and interactions.

2. Identifying Variables and Methods

Example of Sequence Diagram



2. Identifying Variables and Methods

Cohesion between Methods

- *Methods of an object should be in harmony*. If a method seems out of place, then your object might be better off by giving that responsibility to somewhere else. For example: The methods for the `class Car` are as follows. Which one seems strange?
  - `getPosition()`, `getVelocity()`, `getAcceleration()`, *`getAgeOfDriver()`*

- In this case the method *`getAgeOfDriver()`* may appropriate to other class such as *`class Driver`*.

2. Identifying Variables and Methods

Use clear and Unambiguous Method Names

- Having *good names* may prevent others to have a need for documentation.
- If you cannot find a good name, it might mean that your *object is not clearly defined*, or you are trying to *do too much* inside your method.

2. Identifying Variables and Methods

Static Class Members

- Recall that a static variable and method are those that can *be invoked through its class name.*

- For example, the methods of the `Math` class are static:

```
result = Math.sqrt(25)
```

- Determining if a *method* or *variable* should be static is an important design decision

2. Identifying Variables and Methods

The static Modifier

- We declare static methods and variables using the `static` modifier

- It associates the method or variable with the class rather than with an object of that class (it's shared among all objects).

2. Identifying Variables and Methods

Static Variables

- Normally, each object has its own data space, but if a variable is declared as static, *only one copy of the variable exists*

```
private static float price;
```

- Memory space for a static variable is created when the *class is first referenced*
- All objects instantiated from the class *share its static variables* that means Changing the value of a static variable in one object changes it for all others.

2. Identifying Variables and Methods

Student Id problem

- Let's suppose we have `a Student class`
- How do we assign unique student id's to each student object that we create?
- What if we also want to get the latest Student created? By the following method:

```
public static String getLatestStudent()
```

2. Identifying Variables and Methods

The this Reference

- The `this` reference allows an object to refer to *itself*
- That is, the `this` reference, used inside a method, refers to the object through which the method is being executed
- Suppose the `this` reference is used in a method called `tryMe`, which is invoked as follows:

```
obj1.tryMe();
obj2.tryMe();
```

- In the first invocation, the `this` reference refers to `obj1`; in the second it refers to `obj2` `(pass reference)`

2. Identifying Variables and Methods

The this Reference

- The `this` reference can be used to *distinguish the instance variables* of a class from corresponding method parameters with the same names

- The constructor of the `Account` class (from Chapter 4) could have been written as follows:

```
private String name;
private long acctNumber;
private double balance;

public Account (String name, long acctNumber, double balance)
{
   this.name = name;
   this.acctNumber = acctNumber;
   this.balance = balance;
}
```

OOP Development Activities

1. **Identifying Classes and Objects**
2. **Identifying Variables and Methods**
3. **Identifying Class Relationships**

3. Identifying Class Relationships

Class Relationships

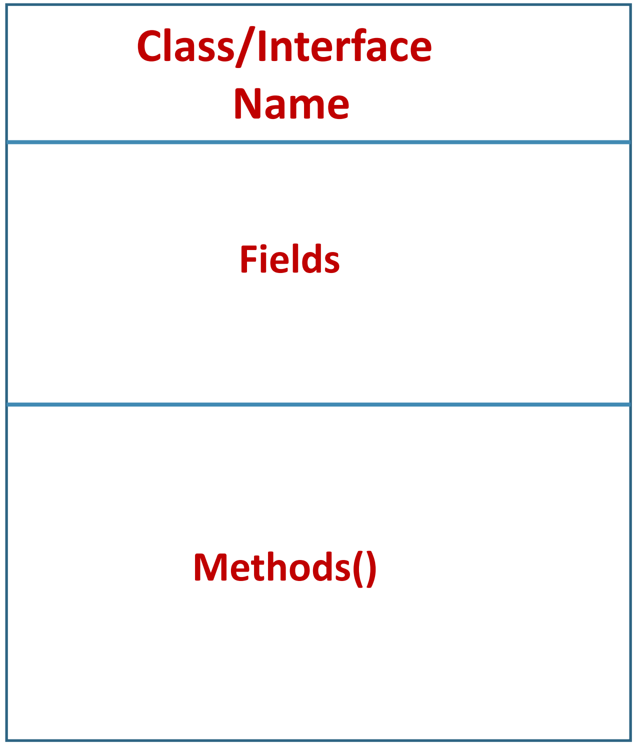
- Classes in a software system can have various types of relationships to each other
- To Design a Software the **UML Diagram** is used to represent Class Relationships

3. Identifying Class Relationships

- **UML Diagram is a picture of**
  - The Class in OOP system
  - *Fields* and *Methods*
  - Relationship between Classes

3. Identifying Class Relationships

- **Basic Diagram of UML is as follows**



Class/Interface Name

Fields

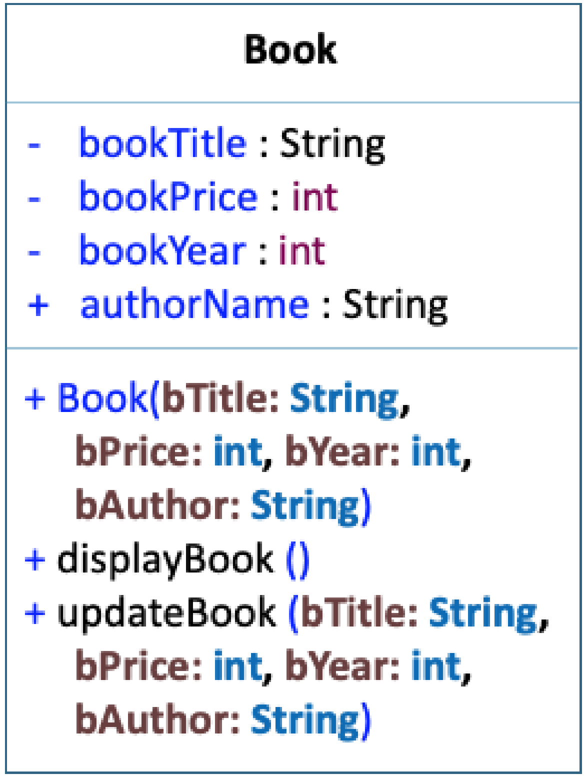
Methods()

Note that: in UML it is important to give **an access modifier** for Field and Methods

+	public
#	protected
-	private
~	package

3. Identifying Class Relationships

- **Example of UML diagram**



Book

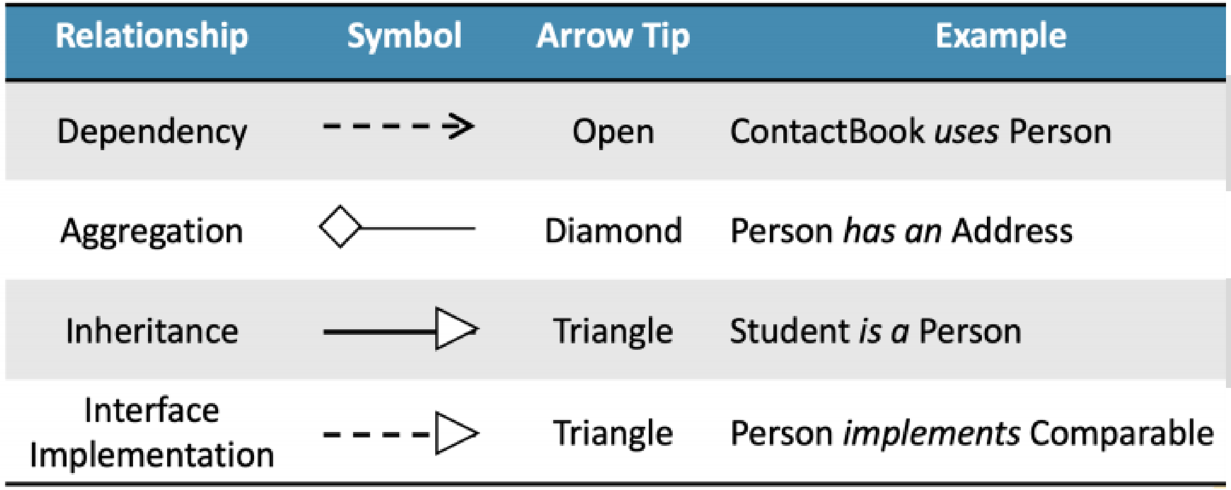
```
-   bookTitle : String
-   bookPrice : int
-   bookYear : int
+   authorName : String

+ Book(bTitle: String,
    bPrice: int, bYear: int,
    bAuthor: String)
+ displayBook ()
+ updateBook (bTitle: String,
    bPrice: int, bYear: int,
    bAuthor: String)
```

3. Identifying Class Relationships

Class Relationships

- Classes in a software system can have various types of relationships to each other
- Four of the most common relationships:



Relationship	Symbol	Arrow Tip	Example
Dependency	- - - - ->	Open	ContactBook *uses* Person
Aggregation	◇———	Diamond	Person *has an* Address
Inheritance	———▷	Triangle	Student *is a* Person
Interface Implementation	- - - -▷	Triangle	Person *implements* Comparable

- Let's discuss *dependency* and *aggregation* further

3. Identifying Class Relationships

Dependency

- A **dependency** exists when one class *relies* on another in some way, usually by *invoking* the methods of the other.
  - **For example**: If Class A uses objects of Class B as parameters in its methods or in it Class, then Class A has a dependency on Class B.
- We don't want numerous or *complex dependencies* among classes, nor do we want *complex classes* that don't depend on others (i.e. one class does all the jobs)
- A good design strikes the right balance.

3. Identifying Class Relationships

Dependency

- Some dependencies occur between *objects of the same class*
- A method of the class may accept an object of the same class as a parameter
  For example: the `concat` method of the `String` class takes as a parameter another `String` object

```
str3 = str1.concat(str2);
```

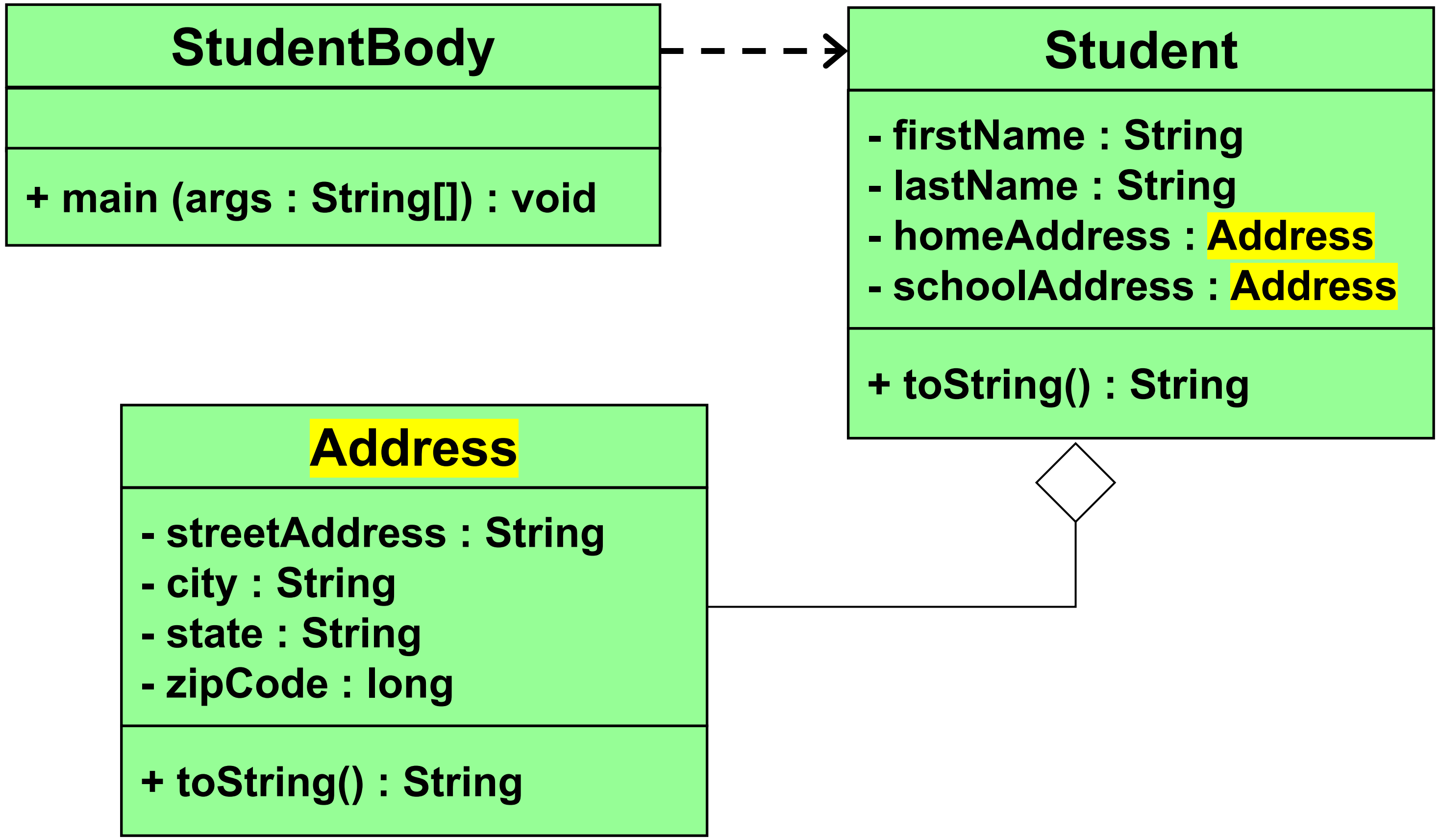
3. Identifying Class Relationships

Aggregation

- Aggregation represents a "whole-part" relationship between classes, where one class (the whole) contains or owns other classes (the parts). The parts can exist independently of the whole.
  - **For example:** a `Student` object (a whole) is composed in (part) of `Address` objects.
- A student ***has an address*** (in fact each student can have more than one addresses)
- An aggregation association is shown in a UML class diagram using an *open diamond* at the aggregate end

3. Identifying Class Relationships

- Aggregation in UML



StudentBody

+ main (args : String[]) : void

Student

- firstName : String
- lastName : String
- homeAddress : Address
- schoolAddress : Address

+ toString() : String

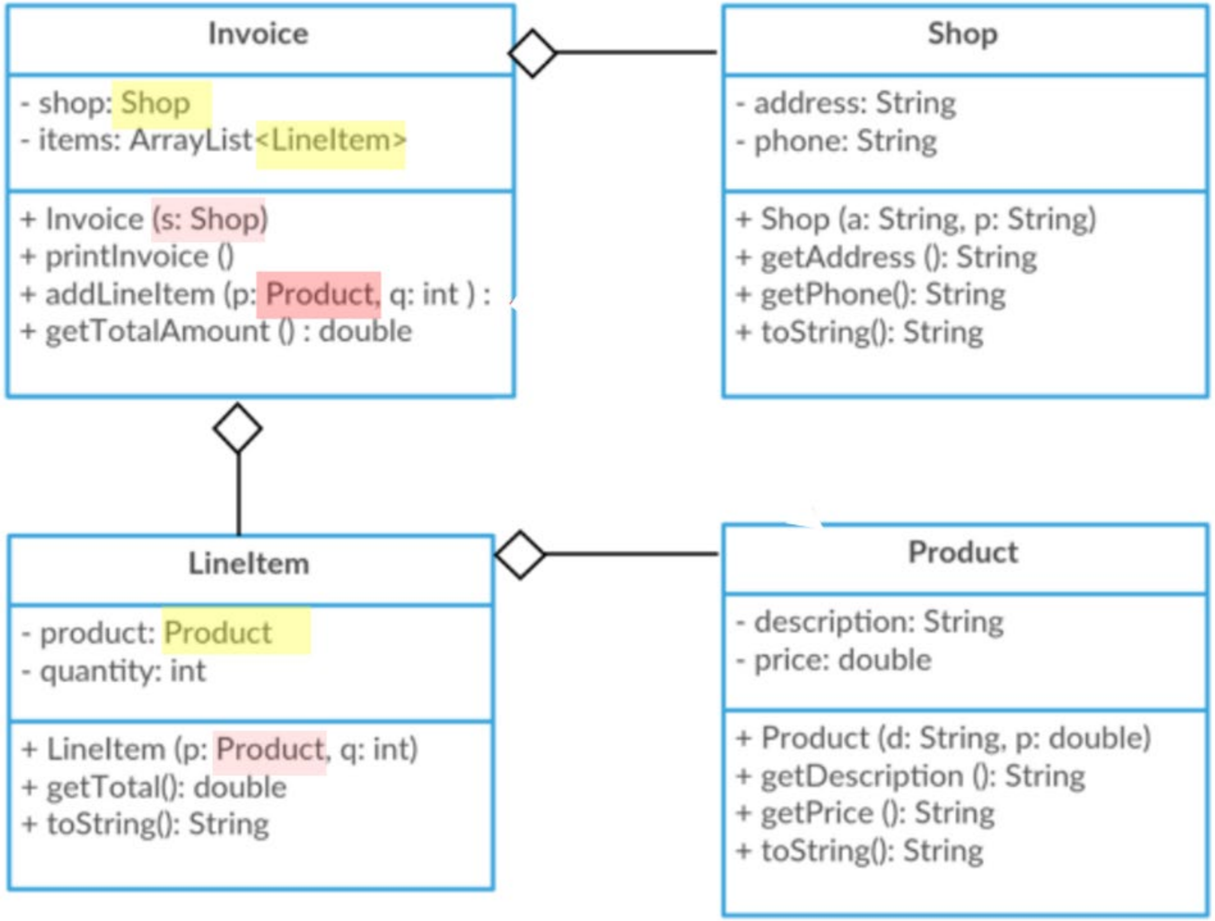
Address

- streetAddress : String
- city : String
- state : String
- zipCode : long

+ toString() : String

3. Identifying Class Relationships

- **Another Example**



Invoice

- shop: Shop
- items: ArrayList<LineItem>

+ Invoice (s: Shop)
+ printInvoice ()
+ addLineItem (p: Product, q: int) :
+ getTotalAmount () : double

Shop

- address: String
- phone: String

+ Shop (a: String, p: String)
+ getAddress (): String
+ getPhone(): String
+ toString(): String

LineItem

- product: Product
- quantity: int

+ LineItem (p: Product, q: int)
+ getTotal(): double
+ toString(): String

Product

- description: String
- price: double

+ Product (d: String, p: double)
+ getDescription (): String
+ getPrice (): String
+ toString(): String

Lab Exercise