Introduction to Programming I

Lab 8

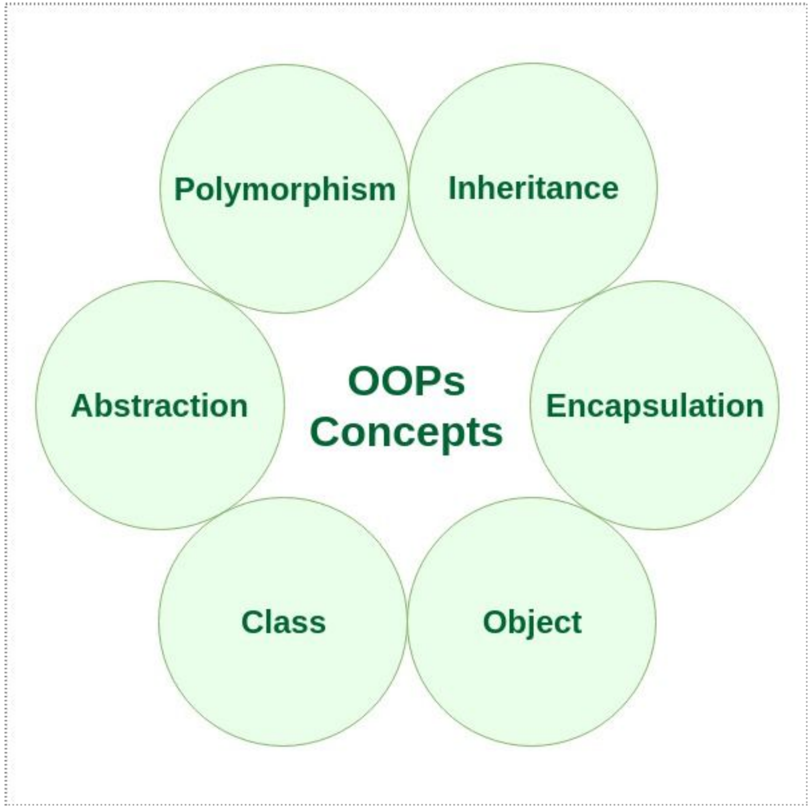
Alexey Shikulin, Munir Makhmutov, Sami Sellami and Furqan Haider

Agenda

Learning outcome:

- Practicing OOP in Java
- A little insight to UML
- Design complex systems using OOP

Java OOP


Polymorphism
Inheritance
Abstraction
OOPs Concepts
Encapsulation
Class
Object

Building Complex Systems

- Now you know OOP which should help you design complex systems
- But before starting the development of a system’s logic which requires multiple classes and relationships between them: you should design the logic in UML
- But you are not required to have a strong knowledge of UML nor to obtain it right now. It is just for you to have a convenient tool to design the logic
- You just need to learn inheritance between classes, and interactions between them (which class uses what etc).

UML

The **Unified Modeling Language** ensures that diagrams drawn by different people can be read and understood by everyone familiar with the language.

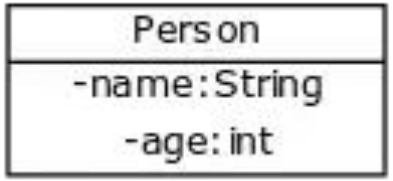
There are many diagrams defined by UML. In this lesson we will focus on the **Class Diagram**.

A **Class Diagram** describes classes, constructors, methods, attributes and the relationships between them.

UML - Class diagram

Describing class and class attributes

```
public class Person {
    private String name;
    private int age;
}
```

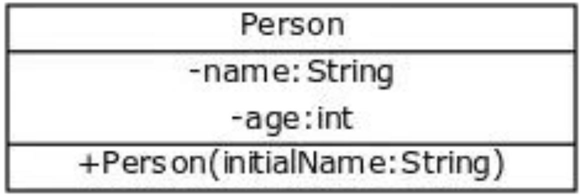


The plus sign (**+**) before the attribute (or method) means public, a (**-**) means that the attribute is private and the (**#**) means the attribute is protected.

UML - Class diagram

Describing class constructor

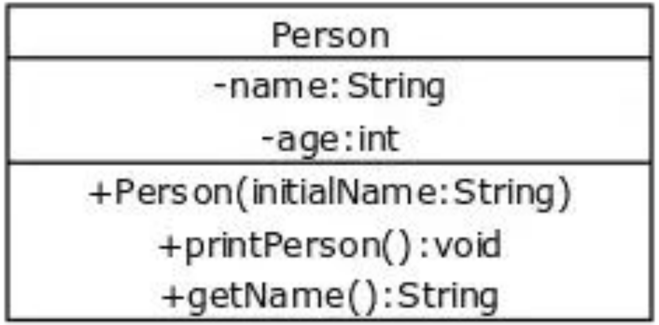
```
public class Person {
    private String name;
    private int age;

    public Person(String initialName)
{
        this.name = initialName;
        this.age = 0;
    }
}
```



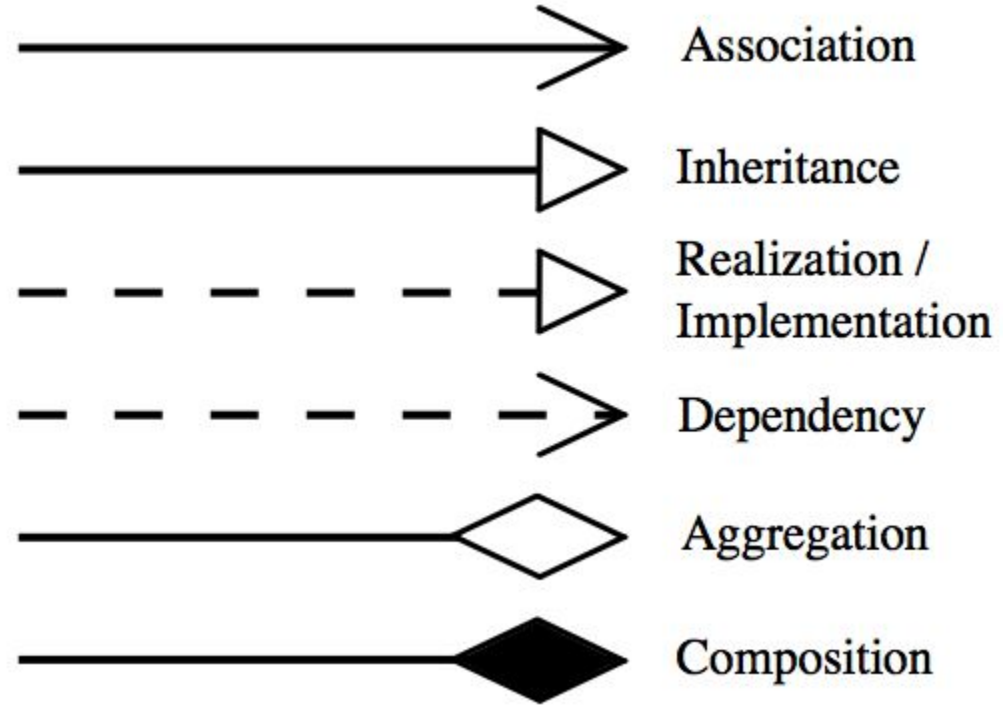
UML - Class diagram

Describing class methods

```
public class Person {
    private String name;
    private int age;

    public Person(String initialName) {
        this.name = initialName;
        this.age = 0;
    }

    public void printPerson() {
        System.out.println(this.name + ", age " +   this.age + " years");
    }

    public String getName() {
        return this.name;
    }
}
```



Relationships between Classes



UML - Class diagram

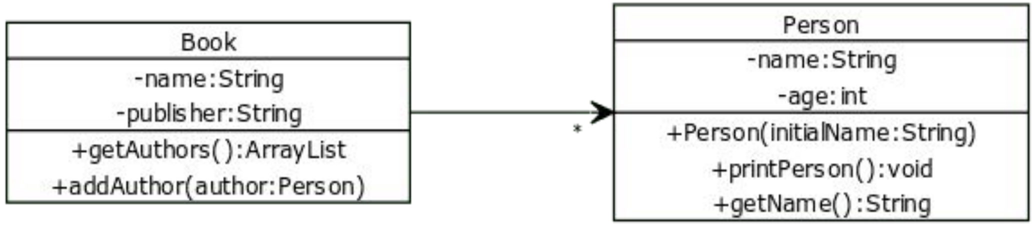
Connections between classes - Association

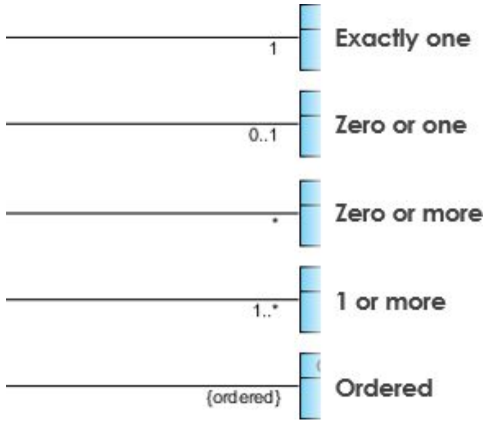
The **arrow** shows the direction of the connection. Here, a Book knows its author but a Person does not know about books they are the author of.

We can add a label to the arrow to **describe** the connection.

If there is **no arrowhead** in a connection, both classes know about each other.

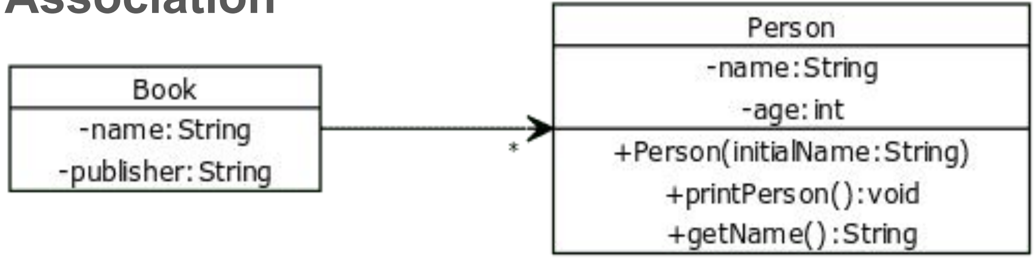
Cardinality is expressed in terms of: one to one, one to many, many to many.





UML - Class diagram

Connections between classes - Association



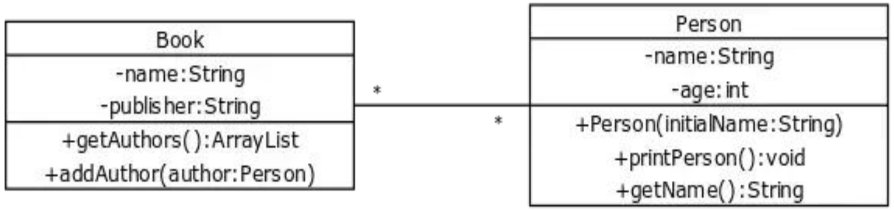
```
import java.util.ArrayList;

public class Book {
    private String name;
    private String publisher;
    private ArrayList<Person> authors;

    // constructor

    public ArrayList<Person> getAuthors() {
        return this.authors;
    }

    public void addAuthor(Person author) {
        this.authors.add(author);
    }
}
```

```
import java.util.ArrayList;

public class Person {
    private String name;
    private int age;

    // ...
}
```

UML - Class diagram

Connections between classes - Association

If a person can have **multiple** books and a book can have **multiple** authors, we add a star to both ends of the connection.



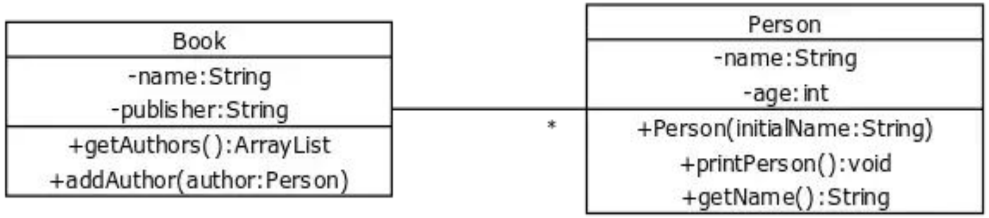
```
import java.util.ArrayList;

public class Person {
    private String name;
    private int age;
    private ArrayList<Book> books;

    // ...
}
```

```
import java.util.ArrayList;

public class Book {
    private String name;
    private int age;
    private ArrayList<Person> authors;

    // ...
}
```

UML - Class diagram

Connections between classes - Association

If there is **no arrowhead** in a connection, both classes know about each other.



```java
import java.util.ArrayList;

public class Person {
    private String name;
    private int age;
    private Book book;

    // ...
}
```

```java
import java.util.ArrayList;

public class Book {
    private String name;
    private int age;
    private ArrayList<Person> authors;

    // ...
}
```

UML - Class diagram

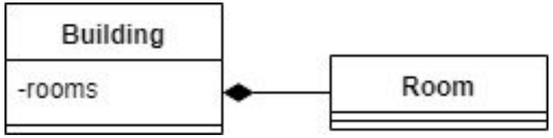
Connections between classes - Composition

Composition is a “**belongs-to**” type of relationship. It means that one of the objects is a logically larger structure, which contains the other object. In other words, it's part or member of the other object.

If we **destroy** the owner object, its members **also will be destroyed** with it**.**

→ Here the room is destroyed with the building.

```
class Building {
    List<Room> rooms;
    String address;
    class Room {
        String getBuildingAddress() {
            return Building.this.address;
        }
    }
}
```



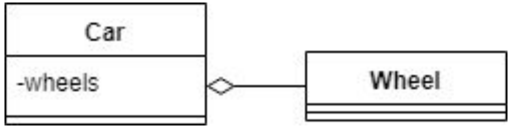
UML - Class diagram

Connections between classes - Aggregation

Aggregation is also a “**has-a**” relationship. What distinguishes it from **composition**, that it doesn't involve owning. As a result, the lifecycles of the objects aren't tied: every one of them **can exist** independently of each other.

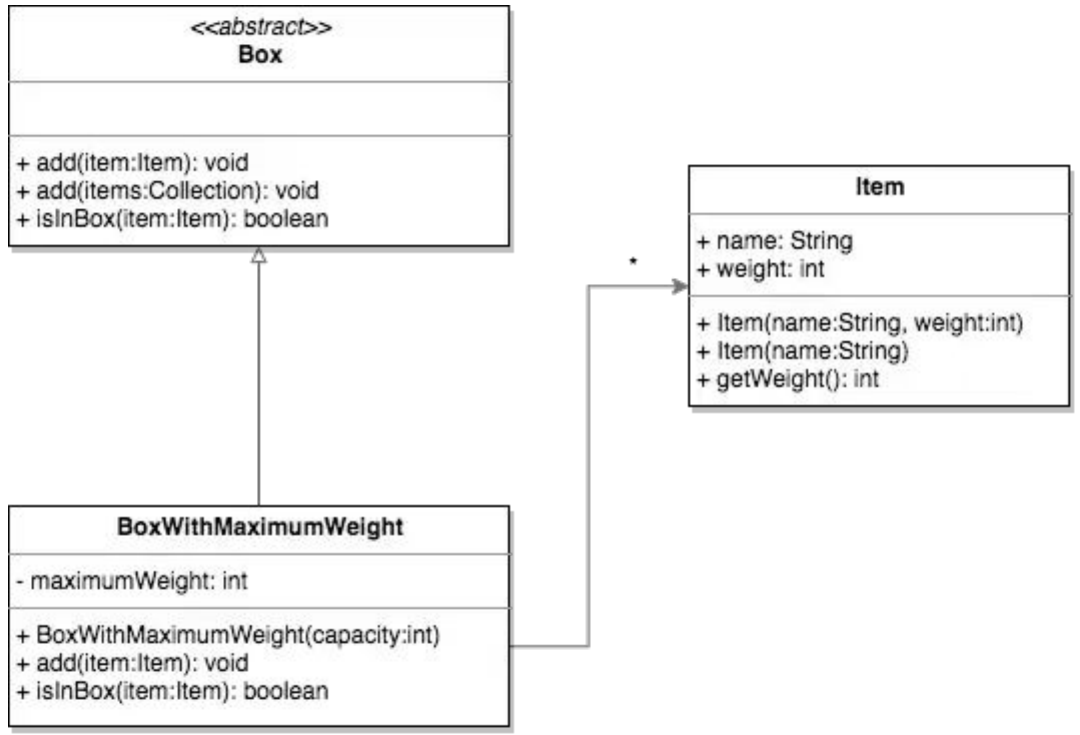
→ We can take off the wheels, and they'll still exist. We can mount other (preexisting) wheels, or install these to another car and everything will work just fine.

```
class Wheel {}

class Car {
   List<Wheel> wheels;
}
```



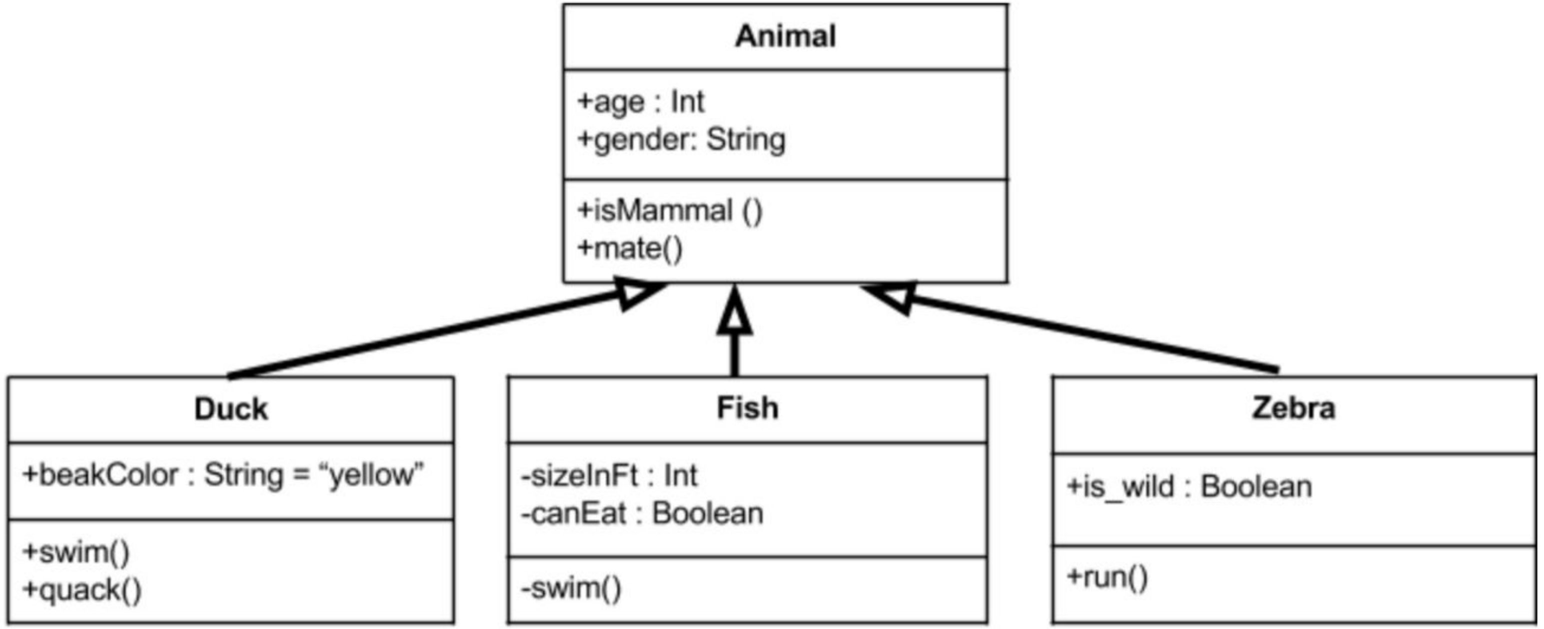
UML - Class diagram

Inheritance



A class can either inherit from another regular class or from an abstract class, marked with the keyword **<<abstract>>**.

Inheritance in UML



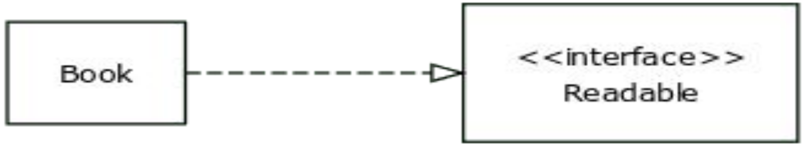
UML - Class diagram

Realization / Implementation (Interface)

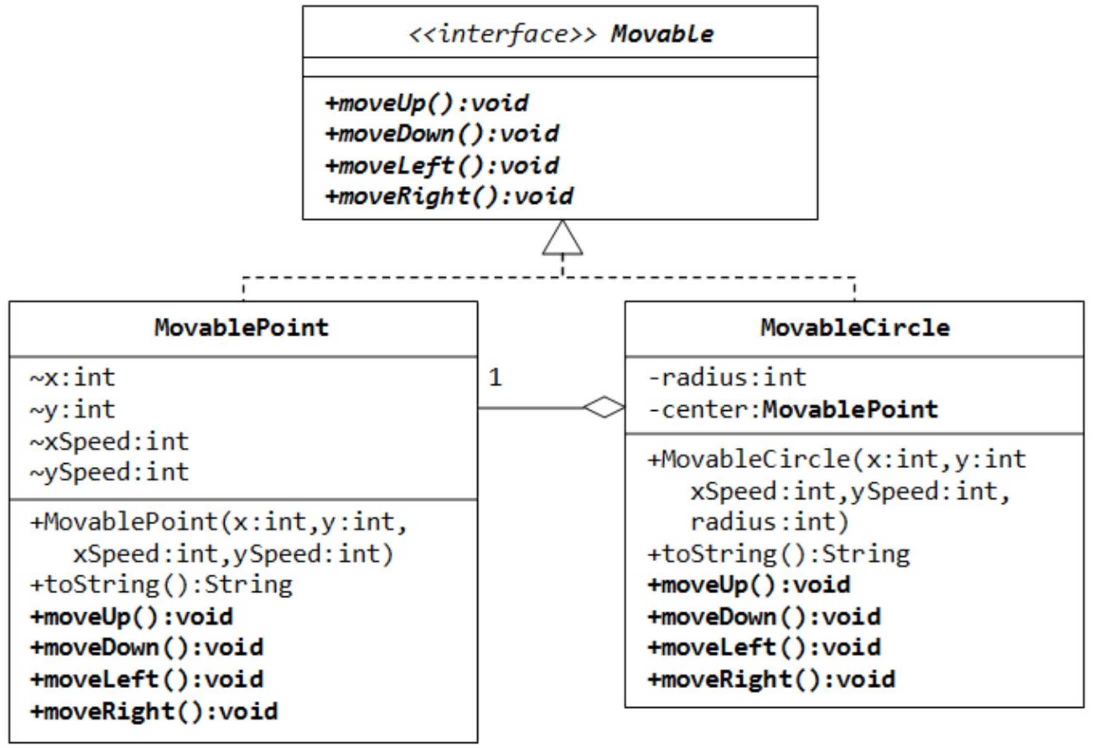
```
public interface Readable {
    // ...
}
```

```
public class Book implements Readable
{
    // ...
}
```

An interface is marked with the keyword **<<interface>>**.

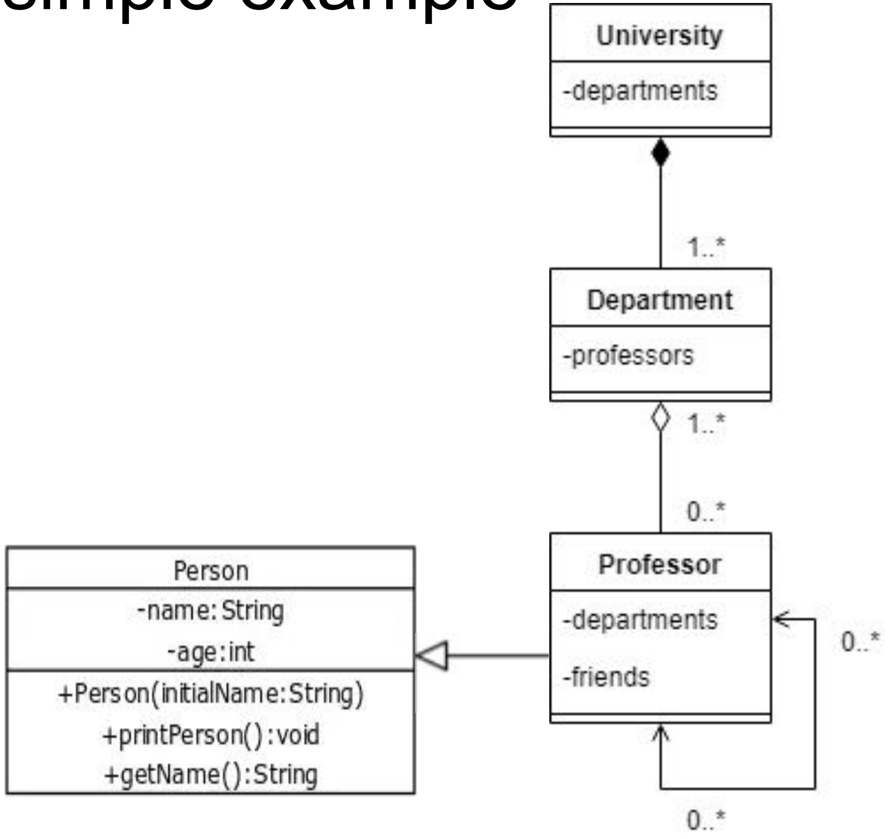


Realization/Implementation

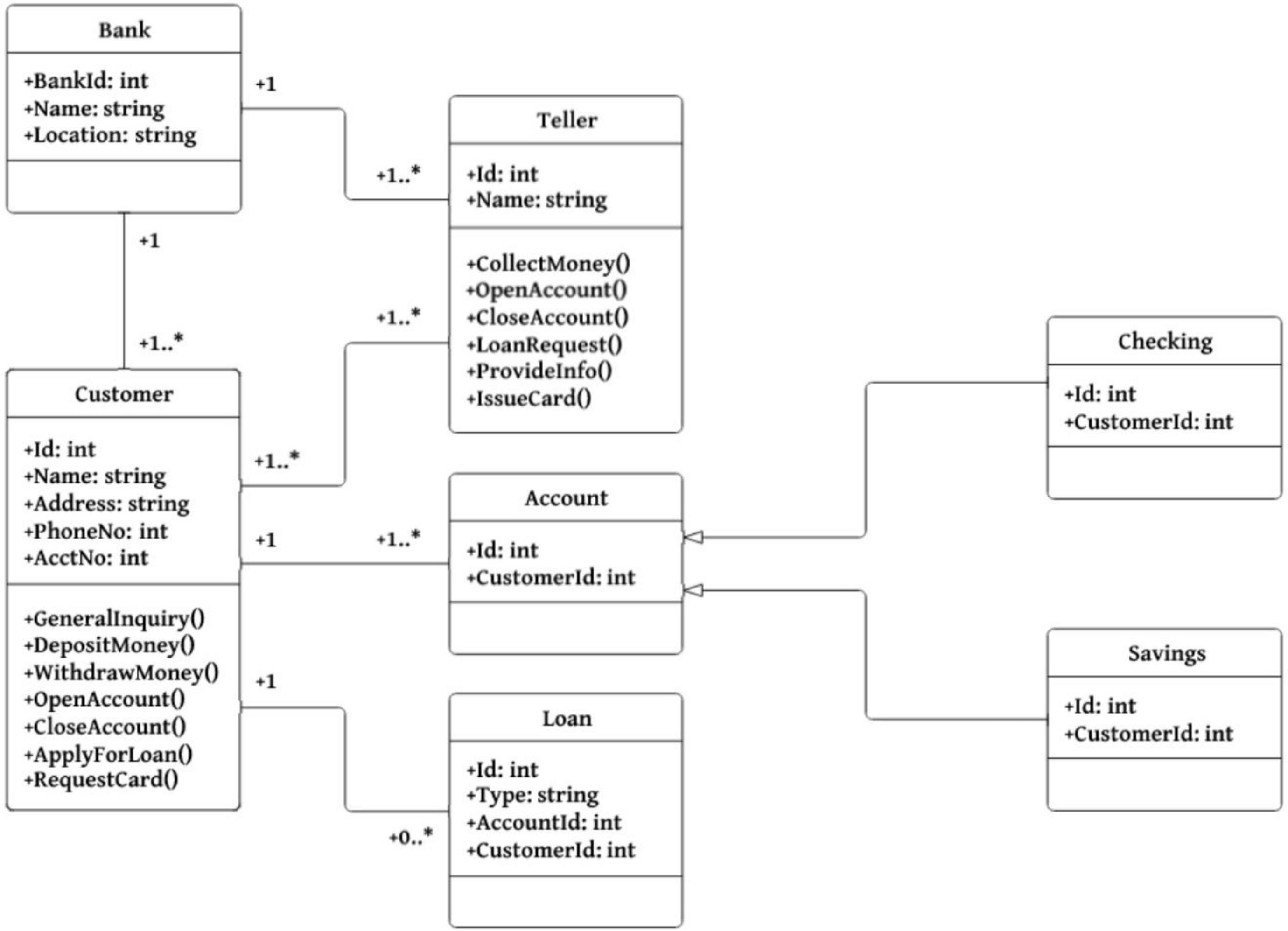


UML - Class diagram simple example

```
public class Person {
    private String name;
    private int age;

    // constructor
    // other methods
}

class University {
    List<Department> department;
}

class Department {
    List<Professor> professors;
}

class Professor extends Person {
    List<Department> department;
    List<Professor> friends;
}
```



UML


Bank
+BankId: int
+Name: string
+Location: string
+1
+1..*
Teller
+Id: int
+Name: string
+CollectMoney()
+OpenAccount()
+CloseAccount()
+LoanRequest()
+ProvideInfo()
+IssueCard()
+1
+1..*
+1..*
+1..*
Customer
+Id: int
+Name: string
+Address: string
+PhoneNo: int
+AcctNo: int
+GeneralInquiry()
+DepositMoney()
+WithdrawMoney()
+OpenAccount()
+CloseAccount()
+ApplyForLoan()
+RequestCard()
+1
+1..*
Account
+Id: int
+CustomerId: int
Checking
+Id: int
+CustomerId: int
Savings
+Id: int
+CustomerId: int
+1
+0..*
Loan
+Id: int
+Type: string
+AccountId: int
+CustomerId: int

Exercise 1: Hospital management system

We want to implement a **hospital management system** where we can manage appointments, bills, patients and doctors.

The **users** in this system can choose from the main menu what type of user they are, depending on that, they can make some actions.

We want to keep track of the bills. A **bill** is defined by a unique **ID**, it has a **name** and an **amount**.

Exercise 1: Hospital management system

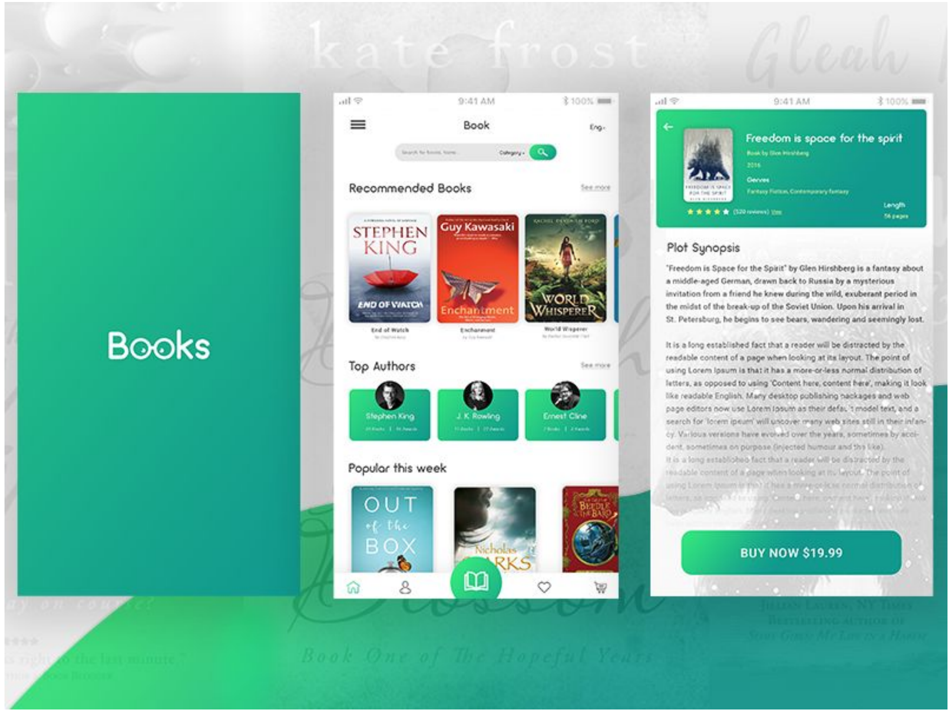
We have three types of users:

- **Patient**: this user is identified by a unique **ID** and have a **name**. They can pay the bill. A bill **belongs** to a patient and each patient has **one** bill.

- **Receptionist**: this user can give appointments to as **many** patients as they want. A patient gets an appointment from **one** receptionist.
  The receptionist can also generate **bills**. A bill is generated by **one** receptionist.

- **Doctor**: this user can **check** as **many** patients as he wants, a patient is checked by **one** doctor.

Exercise 2: Online Book Reader



Plot Synopsis

"Freedom is Space for the Spirit" by Glen Hirshberg is a fantasy about a middle-aged German, drawn back to Russia by a mysterious invitation from a friend he knew during the wild, exuberant period in the midst of the break-up of the Soviet Union. Upon his arrival in St. Petersburg, he begins to see bears, wandering and seemingly lost.

It is a long established fact that a reader will be distracted by the readable content of a page when looking at its layout. The point of using Lorem Ipsum is that it has a more-or-less normal distribution of letters, as opposed to using 'Content here, content here', making it look like readable English. Many desktop publishing packages and web page editors now use Lorem Ipsum as their default model text, and a search for 'lorem ipsum' will uncover many web sites still in their infancy. Various versions have evolved over the years, sometimes by accident, sometimes on purpose (injected humour and the like).

Asked in Amazon, Microsoft, and many more interviews

Exercise 2: Design an Online Book Reader System

Hint: Let’s assume we want to design a basic online reading system which provides the following functionality:

- Searching the database of books and reading a book.
- User membership creation and extension.
- Only one active user at a time and only one active book by this user

The class OnlineReaderSystem represents the body of our program. We could implement the class such that it stores information about all the books, deals with user management, and refreshes the display, but that would make this class rather hefty. Instead, we’ve chosen to tear off these components into Library, UserManager, and Display classes.

Exercise 2: Design an Online Book Reader System

- First try to design the logic with UML.
- Then start coding…

Exercise 1: Equipment Rental

Ski Resort, an equipment rental place, is hiring a Software Engineer to implement part of their software system. Your task is to implement in Java this system. The Ski Resort wants to keep track of who rents what equipment.

Exercise 1: Equipment Rental

Currently, the Resort counts with the following list of equipment to be rented (this is a current list and the developed software needs to be flexible enough so to add more equipment as desired):

- Primary Equipment
  - Skies
  - Snowboard
- Secondary Equipment
  - Helmet
  - Goggles
  - Ski sticks
  - Boots, that can be of two types
    - Ski or
    - Snowboard
- Ski pass

Exercise 1: Equipment Rental

To keep track the equipment, the Resort wants the software to have the following:

- stock: that stores whatever equipment the Resort has in the store;
- rentals: that maps a person with the list of equipment rented (including the date of renting and returning);
- toRent: that implements the action for a person to rent a specific (a list of) equipment;
- toReturn: that implements the action for a person to return the equipment he rented;
- outOfDateFee: that returns a fee in case the person returns the equipment after the returning date.

Exercise 1: Equipment Rental

There are some requirements in order to rent the equipment. The following are the strict rules for the rent:

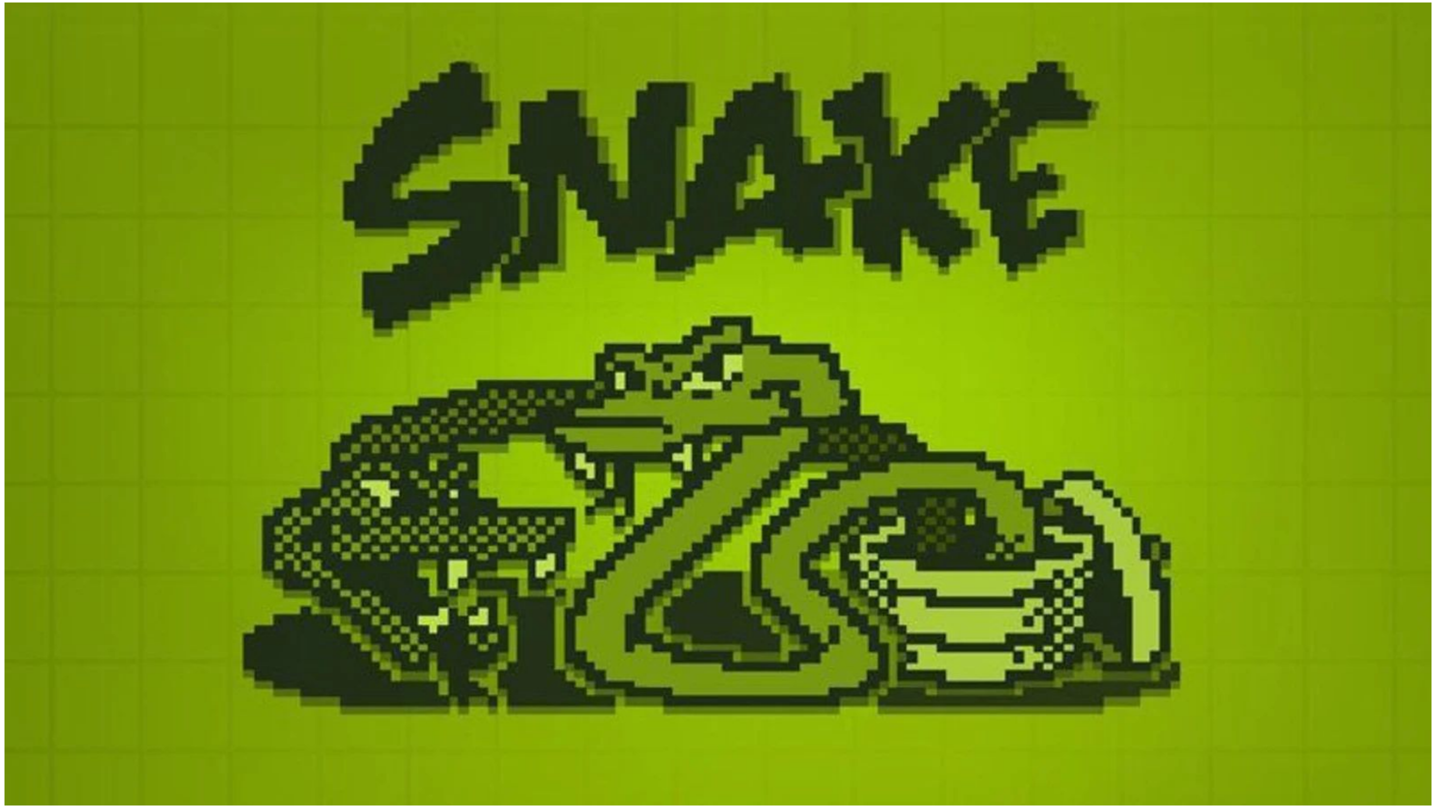
- A person can rent an equipment set: primary equipment (skies or snowboard) with optional secondary equipment and/or a ski pass (renting only ski pass is allowed).
- No secondary equipment is allowed to rent without primary.
- A person can rent skies with/without ski boots, helmet, goggles and ski sticks.
- A person can rent snowboard with/without snowboard boots, helmet and goggles.

Exercise 1: Equipment Rental

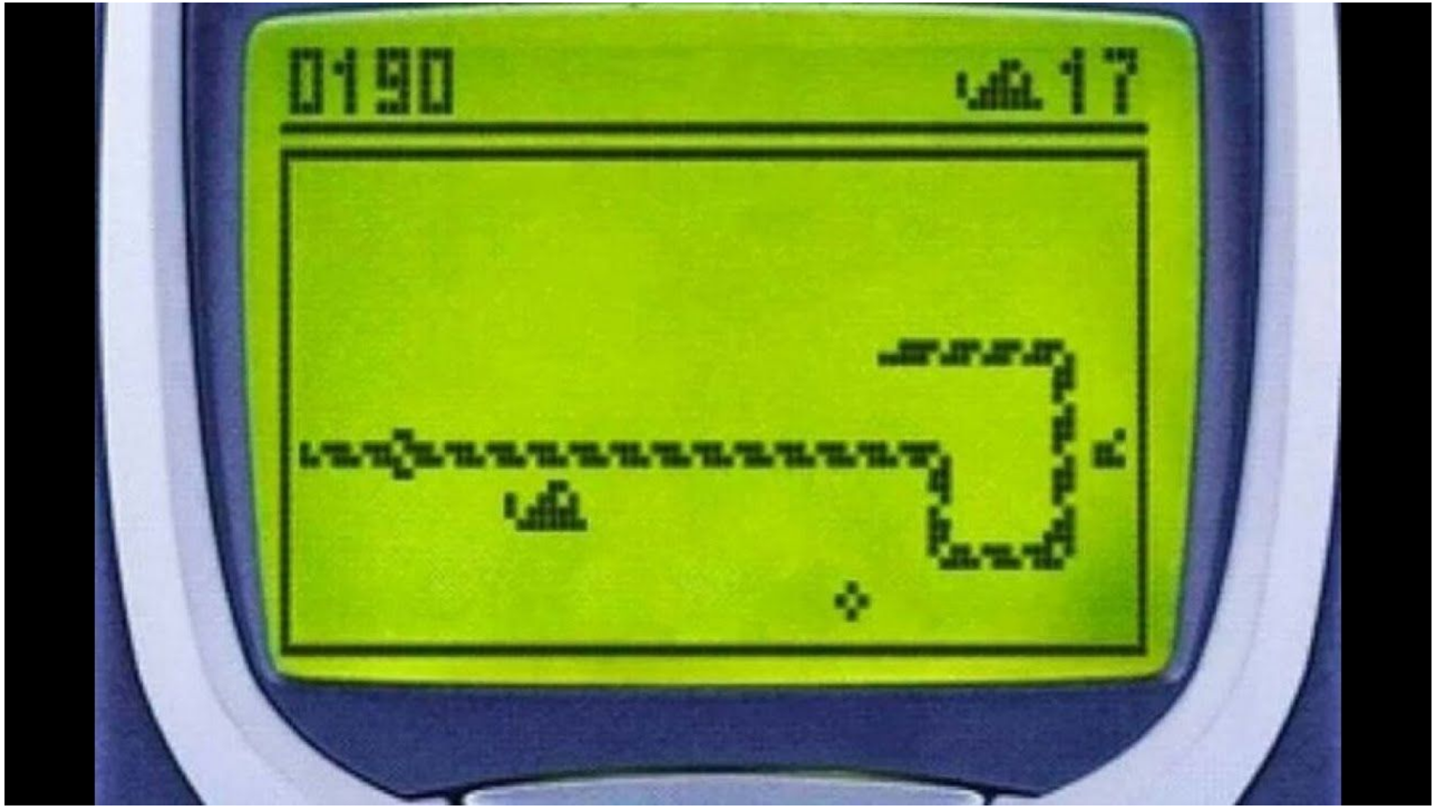
There are rules that the Ski Resort follows that need to be reflected in the implementation of the system:

- The Ski Resort always has at least one item of each equipment they offer. For instance, as it is right now, the Resort has at least a pair of skies, one snowboard, one helmet, one goggles, one pair of ski sticks, one pair of ski boots, one pair of snowboard boots and one ski pass. The Resort cannot be in a situation where there are zero items of a specific offered equipment. (Notice that the list of equipment can change in time – new equipment can arrive)
- A person can rent some equipment if there is enough items for him to rent.

Exercise 3: Snake Game



… childhood :))



Exercise 3: Design Snake Game

Let us see how to design a basic Snake Game which provides the following functionalities:

- Snake can move in a given direction and when it eats the food, the length of snake increases.
- When snake crosses itself, the game will over.
- Food will be generated at a given interval.

This question is asked in interviews to Judge the Object-Oriented Design skill of a candidate. So, first of all, we should think about the classes.

The main classes will be:

1. Snake
2. Cell
3. Board
4. Game

Game represents the body of our program. It stores information about the snake and the board.

Cell represents the one point of display/board. It contains the row NO, column NO and the information about it, i.e. it is empty or there is food on it or is it a part of snake body?

References

- Inheritance, abstract classes, interfaces https://medium.com/@isaacjumba/overview-of-inheritance-interfaces-and-abstract-classes-in-java-3fe22404baf8
- Polymorphism https://codegym.cc/groups/posts/99-how-to-use-polymorphism
- UML: https://medium.com/@smagid_allThings/uml-class-diagrams-tutorial-step-by-step-520fd83b300b
- Problems: https://www.geeksforgeeks.org/
- Problems: https://www.e4developer.com/2018/08/16/designing-an-object-oriented-chess-engine-in-java/
- 2d snake game: https://github.com/hexadeciman/Snake