Обектно Ориентирано Програмиране с Java

Лектор: ***проф. д-р Е. Кръстев*** (eck@fmi.uni-sofia.bg)

Асистент: **Петя Григорова**(petya.bgr@gmail.com)

[1] P. Deitel, H. Deitel, "Java 9 for Programmers", Prentice Hall 4th ed. **2017**, ISBN-13: 978-0-13-477756-6 ISBN-10: 0-13-477756-5 (**основна)**

[2] Y. Daniel Liang, "Java Programming and Data Structures. Comprehensive version"**, 11th** ed.,Pearson **2019** ISBN-10: 1-292-22187-9 ISBN-13: 978-1-292-22187-8

Technology

- JDK 17.x
- IDE- IntelliJ Ultimate 2022
- UML Modeling
- JavaFX

Methodology

- Tutorial classes and Labs
- One Midterm test
- Course Project
- Final Exam

Evaluation

Final grade components:

Coursework (60%)

- **Midterm written exam (40%)**
- **Project (20%)**

Written examination (40%)

- **A final written exam**

Evaluation methods

Grades:

2 ***from 0 to 54 marks***

3 ***from 55 to 64 marks***

4 ***from 65 to 74 marks***

5 ***from 75 to 84 marks***

6 ***from 85 to 100 marks***

Забележка:

Писменият изпит през семестъра (midterm) и защитата на курсовия проект не се повтарят след приключване на семестъра

Evaluation methods

В случай, че в края на семестъра средната оценка от писменият изпит през семестъра (midterm) и проекта, взети със съотвентите тежести, е по-малка от 55 точки, то студентът трябва да повтори курса

Evaluation methods

Посещението на лекции и практически занятия, както и предаване на работата от практическите занятия е задължително условие за ползването на сайта на курса.

Достъпът до сайта се прекъсва при липса на активност за повече от 4 седмици.

Моите очаквания и изисквания

- Да **посещавате редовно** лекции и да **изпълнявате старателно** практическите занятия
- **Да питате, ако не знаете как да …**
- **Да следите редовно материала и да идвате подготвени в час**
- **Да нараства интереса Ви към курса**

Course Goals

- Presents the Concepts of OOP
- Advanced Java Programming
- Solve typical Business Problems

Description

Fundamentals of **OOA**; **Data structures and algorithms**; **Style of programming** and profiling Java applications; **Inheritance** and applications; **Polymorphism- abstract classes and abstract methods**, **interfaces**, **closure**, **callback**, **lambda** expressions, Handle **exceptions**; **Inheritance** and **Polymorphism-** building interactive GUI with **JavaFX**; generics with **Java Collections Framework**, processing files, **Streams** API, **Multithread programming**; asynchronous execution of tasks, **RMI** and **web services**.

OO Program development – Basic Concepts

- Build up a program according to the objects it involves rather than the functions it supports
- Build up a program of a number of well-delimited units called objects
- Involves OOA, OOD and OOP

Advantages of Java

- Java offers higher **cross- functionality and portability** as programs written in one platform can run across desktops, mobiles, embedded systems.
- Java is free, simple, **object-oriented**, **distributed**, where **multithreading, multimedia** (JavaFX), data query language (Stream API and JPQL) and **networking** are inherently integrated into it.
- Java is **a mature language**, therefore more stable and predictable. The Java Class Library enables cross-platform development.
- Being highly popular at enterprise, embedded and network level, Java has a large **active user community and support** available.

Advantages of Java

- Unlike C and C++, Java programs are compiled independent of platform in *bytecode* language which allows the same program to **run on any machine that has a JVM installed**.
- Java has **powerful development tools** like Eclipse SDK and NetBeans which have debugging capability and offer integrated development environment.
- Increasing language diversity, evidenced by **compatibility** of Java with Scala, Groovy and JRuby.
- Java considers **security as part of its design**. The Java language, compiler, interpreter, and runtime environment were each developed with security in mind.

Basic Concepts

- Function- oriented view, Traditional approach- transforms a flow of data
- Object- Oriented view- uses “objects” as models of real or devised things in the program’s environment, the computer program manipulates these “objects”

Example

- Lift Object
  - status (direction, floor No)
  -behavior (go_to(),whereAmI(),stop())

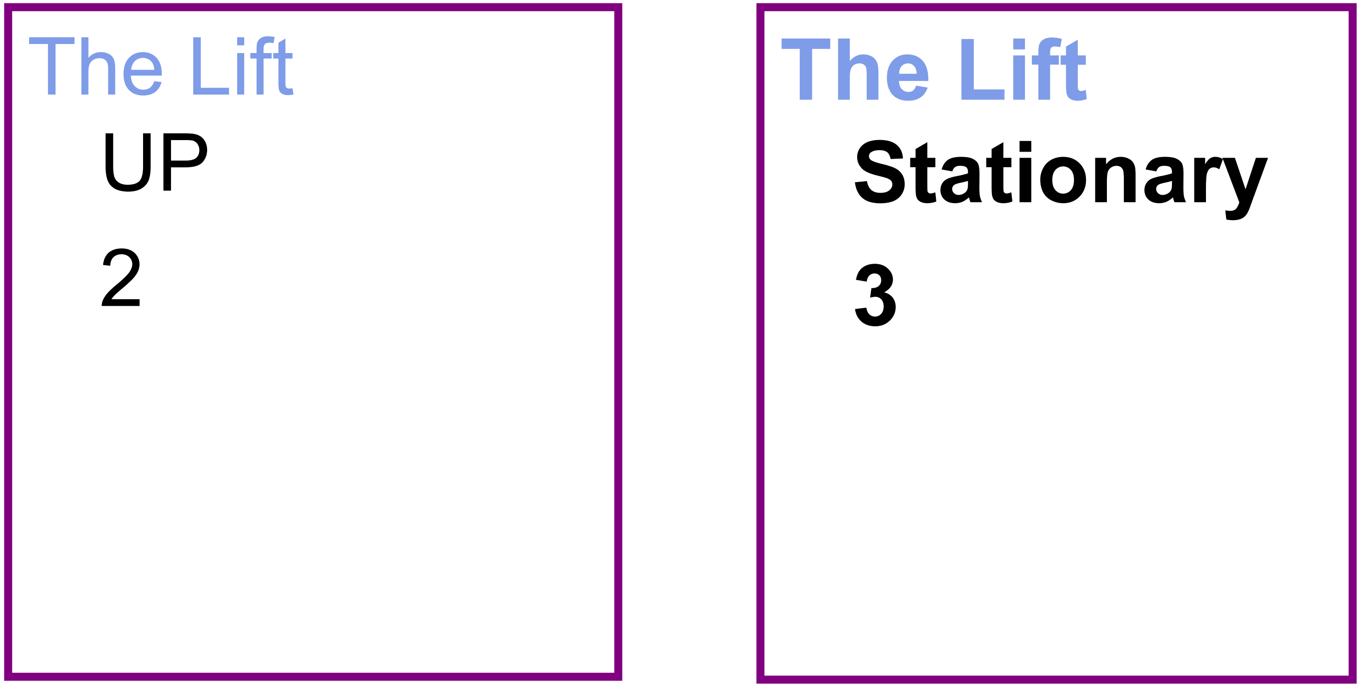
Description of Objects

- Each Object has unique identity
- A program makes access to objects via references (variables)
- Objects have two kind of attributes
  - status data (information hiding)
  - behavior operations

Classes

- A class is anything that represents existing people roles in human society *(Customer, Employee, Student etc)*, living creatures *(Bird, Snake, Rabbit etc)*, things *(Product, Book, Document etc)*, events *(Graduation, Exam etc.)* or abstract categories *(Person, Shape, Object, Exception etc)*.

Example- instances of the *Lift* Class



The Lift

UP

2

The Lift

Stationary

3

Classes and Objects

- A class is a pattern, which defines the appearance of a collection of objects with common construction and set of properties
- Objects the are instances of a given class.

Object Oriented Analysis

- Requirements Analysis
- Requirements specification

- find the objects which will be part of the model
- Define the object attributes
- Establish the relations between the different objects

Relations between Objects

- Has- A
- Knows- About (association)
- IS- A

IS- A relationship

- Definition
  States that an Object is constructed with the help of another object, that it has this object as one of its parts
- Example
  - A Car HAS a motor
  - A Book HAS chapters
- Graphical Notations (rhombus)
- Represented as data attributes

KNOWS- About < > HAS- A

- KNOWS- ABOUT is BIDIRECTIONAL
- HAS-A is UNIDIRECTIONAL
- Example
  -A Car has an Owner and the Owner has a Car (or many Cars)
  - A Person may be married (recursion)
- Graphical representation (Circle)
- Program representation

KNOWS- About (association)

- Definition
  - An Object knows about and communicates with another object, without necessarily stating that it is constructed with the help of another object
- Example
  -A person KNOWS his address
  - A Car KNOWS about its owner

IS- A relationship

- Definition
  -a class has certain general properties that can be common to other classes
- Example
  - A Circle is a Shape
  - A Student is a Person

IS- A relationship
INHERITANCE

- Start describing a class (subclass) with an already existing class (super class) by adding/ subtracting attributes
- Examples
  -Person (name, address)- super class
  -Student(+ major) – subclass
  -Vehicle(Car, Boat, Train, Bike)
- Graphical notation (crow foot)
- Program representation, class library

OOD
System design + Object design

- Second phase
  -plans are drawn up and drawings of the OOP
- OOA
  -what is to be done
- OOD
  -how these things are to be done
- Iterative process

OOP

- Requirements for a “good” program
  -a correct program (ops defined)
  -an effective program(resources)
  -it is reusable (development+maintenance costs, development time, quality)
  -it is adaptable (info hiding, packaging)

Modified Hungarian Notation

- It is very important to **keep the coding style consistent.**

- **short prefix mnemonics** that allowed programmers to easily identify the type of information a variable might contain .

- both types of code **interoperate**

Modified Hungarian Notation

Some commonly used prefixes in this course:

Control	Prefix
Button	btn
ComboBox	cbo
CheckBox	chk
Label	lbl
ListBox	lst
MainMenu	mnu
RadioButton	rdb
PictureBox	pic
TextBox	txt

Modified Hungarian Notation

As a **general rule**, notice that in **Java**:

- **class and interface** names start by a **Capital** letter
- **references** to classes and interfaces, as well, as **variables of primitive data types** such as ***int, boolean, double*** etc start by a **lowercase letter**
- the **names of methods** start by a **lowercase** letter
- the **names of controls have to follow the Modified Hungarian notation** explained above (*they have to be descriptive by means of introducing appropriate prefixes*)

Writing good code

Good programming **qualities**:

- Simplicity
- Readability
- Modularity
- Layering
- Design
- Efficiency
- Elegance
- Clarity

Writing good code

Simplicity

Means you *don't do in ten lines what you can do in five*. It means you make extra effort to be concise, but not to the point of obfuscation. It means you abhor open coding and functions that span pages. Simplicity- of organization, implementation, design- makes your code more reliable and bug free. There's less to go wrong

Writing good code

Readability

Means what it says: ***that others can read your code***. Readability means you bother to **write comments, to follow conventions**, and pause to **name your variables wisely**. Like choosing "*taxRate*" instead of "*tr*".

Writing good code

Modularity

Means ***your program is built like the universe***. The world is made of molecules, which are made of atoms, electrons, nucleons, quarks, and (if you believe in them) strings. Likewise, **good programs erect large systems from smaller ones**, which are built from even smaller building blocks. You can write a text editor with three primitives: move, insert, and delete. And **just as atoms combine in novel ways, software components should be reusable.**

Writing good code

Layering

Means that **internally, your program resembles a layer cake**. The app sits on the framework sits, the OS sits on the hardware. Even within your app, you need layers, like file-document-view-frame. **Higher layers call ones below, which raise events back up**. (Calls go down; events go up.) **Lower layers should never know what higher ones are up to.** The essence of an **event/callback** is to provide **blind upward notification**. .

Writing good code

Design

Means you **take time to plan your program before you build it**. Thoughts are cheaper than debugging. **A good rule of thumb is to spend half your time on design**. You need a functional spec (what the programs does) and an internal blueprint. APIs should be codified in writing.. .

Writing good code

Efficiency

Means **your program is *fast* and *economical*. It doesn't hog files, data connections, or anything else**. It **does what it should, but no more**. It **loads and departs without fuss**. At the function level, you can always optimize later, during testing. But at high levels, you must **plan for performance**. If the design requires a million trips to the server, expect a big problem.

Writing good code

Ellegance

Elegance is like beauty: hard to describe but easy to recognize. Elegance combines ***simplicity***, ***efficiency***, **and *brilliance*, and produces a feeling of pride**. Elegance is when you replace a procedure with a table, or realize that you can use recursion- which is almost always elegant:

```
int fact(int n) {
    return n==0 ? 1 : n * fact(n-1);
 }
```

Writing good code

Clarity

Clarity is the platinum quality all the others serve. The fundamental challenge of programming is **managing complexity**. ***Simplicity, readability, modularity, layering, design, efficiency, and elegance*** **are all time-honored ways to achieve clarity**, which is the antidote to complexity. **You must understand- really understand- what you're doing at every level**. Otherwise you're lost. **Bad programs are less often a failure of coding skill than of having a clear goal.**

Happy
Object Oriented Programming
with
Java