PJWSTK • INFORMATION SYSTEMS DESIGN

Design and Analysis of Information Systems

Comprehensive Lecture Handbook on Object-Oriented Modeling, Persistence, Advanced UML, Relational Mapping, and GUI Usability

---

Mariusz Trzaska, Ph.D.
Polish-Japanese Academy of Information Technology (PJWSTK / PJATK)
Warsaw, Poland

Table of Contents

00.	Course Overview & Rules	Foundations
01.	Classes in OOP	Object-Oriented Modeling
02.	Classes in OOP (Part 2)	Object-Oriented Modeling
03.	Classes in OOP (Part 3)	Object-Oriented Modeling
04.	Associations in OOP	Associations & Links
05.	Associations in OOP (Part 2)	Associations & Links
06.	Inheritance in OOP	Inheritance & UML
07.	Inheritance in OOP (Part 2)	Inheritance & UML
08.	Advanced UML Constructs	Inheritance & UML
09.	Relational Model in OOP	Persistence & Relational
10.	Relational Model in OOP (Part 2)	Persistence & Relational
11.	Relational Model in OOP (Part 3)	Persistence & Relational
12.	Relational Model in OOP (Part 4)	Persistence & Relational
13.	GUI Usability & Testing	GUI & Usability
14.	GUI Architecture & Design	GUI & Usability
15.	GUI Implementation & Logic	GUI & Usability

LESSON 00

Course Overview & Rules

Course regulations, grading criteria, mini-projects, final project specification, and recommended development environments.

Justification

Modern programming languages are largely based on the object-oriented paradigm. Therefore, a practical understanding of this paradigm is essential for anyone working in this field. Additionally, programmers frequently develop systems based on analytical materials, which often employ object-oriented graphical notations, such as UML. In this lesson and the next, you will deepen your theoretical knowledge, but more importantly, you will learn how to implement various object-oriented concepts. This will enable you to design and develop modern business applications effectively.

Introduction

This course focuses on embedding the conceptual model of a problem domain - the result of the analysis phase - and the specification requirements into a specific implementation environment, both object-oriented and relational. Students will learn how to construct the necessary structures to represent the model, including those not natively supported by a programming language. Topics related to usability, including testing of graphical user interfaces, are also covered. Theoretical knowledge is reinforced through practical implementation of data structures, business logic, and both simple and advanced graphical user interfaces (e.g., using dedicated editors). Each student is expected to carry out dynamic analysis, design, and implementation work based on individual user requirements. Specification requirements and static analysis are addressed during the Information Systems Design (PRI) course. Additionally, students entering the MAS course are expected to have proficiency in an object-oriented programming language, with Java as the default.

Mini-projects

The purpose of the mini projects is to practically verify students' understanding of how to implement specific constructs from the conceptual model (such as classes, associations, extents, etc.) and their interaction with the relational model. Additionally, they can serve as a "foundation" for the final project, which will later be expanded with additional elements, including a graphical user interface (GUI). Students are expected to implement various business-oriented constructs (technical details such as the number of objects in an extent, setters/getters, or object display do not apply) present in the conceptual model. Each mini project must be implemented as a **console application**. Students must propose their own business cases for the mini projects; examples from lectures, books, or other sources are not allowed. It is possible to submit mini projects at later sessions, but doing so will result in a 50% reduction in points. Elements of the mini projects subject to evaluation include:

MP1	MP2	MP3	MP4

Classes, attributes	Associations	Inheritance	Relational model (RM)
• the extent • Extent - persistency • Complex attribute • Optional attribute • Multi-valued attribute • Class attribute • Derived attribute • Class method • Override • Overload	• Binary • With attribute • Qualified • Composition *Each of the associations requires at least 1-* cardinality and automatic reverse connection.*	• Disjoint • Abstract • Polymorphic method call • Overlapping • Multi inheritance • Multi aspect • Dynamic	• RM - classes • RM - association (1:* or *:* cardinalities) • RM - inheritance *You are required to create a working program that uses a database and implements the above constructs. You must use an ORM (Object-Relational Mapping) tool, such as Hibernate or Entity Framework. The archive should contain only source files (no binaries/libraries) optionally including mappings (for ORMs). Do not forget about relevant examples.*

Mini-projects are a vital part of the final grade. Therefore, they should be created with a proper care. The MPx are delivered as a 7z/ZIP/RAR archive containing (a simple **console application**):

1. source codes with comments (describing implemented constructs, API and individual implementation steps). Please follow the guidelines for your platform, eg. Java requires separate files for each public class. You do not need to include binary files (libraries, images, etc);
2. among others, a **single class (file)** *Main.java* (*main.cs*, etc.) with the `main` method with commented examples showing how to use a particular construct, e.g.:

   ```java
   person1.setAddress(address2); // complex attribute
   ```

   Only commented items presented in the method will be evaluated. Projects without this class **will not be evaluated**;

The archive file (*MPx_LastName_FirstName_Number.zip* where x is the mini-project number) should be placed in the appropriate EDUX folder. The deadline - see below.

The final project

The project consists of two parts: a documentation part and an implementation part. The topic should be chosen so that it allows for the creation of a sufficient number of meaningful business classes (typically 12–

15). Therefore, tool-based applications, system utilities, media players, and similar types of software are generally not suitable. If you are unsure about your topic, you should consult the course instructor.

Documentation

1. The documentation may, but does not have to, be based on a project from the PRI course.
2. The project should be grounded in user requirements described as enumeration “user stories.”
3. Based on these requirements, you should develop: Functional requirements (represented as use case diagrams and optionally in textual form), Non-functional requirements (presented in some kind of enumeration).
4. A key component of the requirements is the analytical class diagram. Since it serves as the foundation for further work, it is essential that it is error-free. Without a correct analytical class diagram, it is impossible to prepare a proper design (implementation) class diagram.
5. At least one non-trivial use case (which references another use case) must be discussed in detail. The scenario for this use case should be described both in natural language and using activity diagrams.
6. User interface design: Based on the selected non-trivial use case, a user interface design should be created following the guidelines presented in the lectures.
7. Dynamic analysis: For the selected use case, perform dynamic analysis using activity diagrams and state diagrams. This analysis should conclude with an explicit discussion of its implications. The results of the dynamic analysis (e.g., new attributes, new associations) must be reflected in the implementation (design) class diagram. The project must utilize all the types of diagrams mentioned above.
8. Include elements that specify design and implementation decisions, illustrated with examples from the relevant diagrams (e.g., how class extents or inheritance are implemented).
9. Based on the analytical class diagram, an implementation (design) class diagram is created. It may include only elements derived from the analytical diagram and should:
    1. Be detailed and fully specified.
    2. Replace any constructs that do not exist in the chosen programming language according to the design decisions made.
    3. Optionally include methods resulting from the dynamic analysis.
10. The entire documentation, and especially the diagrams, must be **clear and readable**. It is recommended to use a vector format (e.g., Enhanced Metafile) or a lossless compression format (e.g., PNG). To ensure clarity, pay attention to the following:
    1. Correct use of UML notation (do not use “similar” graphical elements).
    2. Every diagram must be properly labeled.
    3. Diagrams should be prepared using an appropriate tool (e.g., UMLet).
    4. Font sizes on diagrams should be readable without excessive zooming (a diagram at 100% zoom should be legible; if necessary, increase the font size rather than the entire diagram).
    5. Diagrams should not be split into multiple parts.
    6. Elements (e.g., classes) must be properly arranged, e.g.:
        1. inheritances vertically,

2. associations horizontally,
3. avoid crossing lines,
4. associations should include names and multiplicities.

Note: Projects containing unreadable diagrams **will not be evaluated**.
A prepared guide can be helpful in creating readable diagrams.

11. The content of diagrams should not be duplicated in textual form, e.g., by providing a verbal description of the classes, their attributes, and their relationships. Such submissions **will not be evaluated**.
12. The project documentation should include the following elements:
    1. User requirements (“user story”)
    2. Use case diagrams (functional requirements)
    3. Non-functional requirements
    4. Analytical class diagram
    5. Design (implementation) class diagram
    6. Use case scenario (as an enumeration)
    7. Activity diagram for the use case
    8. State diagram for a class
    9. GUI design
    10. Discussion of design decisions and the results of dynamic analysis
13. The absolute maximum length of the documentation is **25 A4 pages**. Submissions exceeding this limit **will not be evaluated**.
14. Documentation should be delivered in **a single PDF file** ("*MAS_Group_Lastname_Firstname_StudentNo.pdf*") to a dedicated folder of the Gakko/EduX system. Deadline - see the attached schedule.

Implementation of the final project

1. The complete structure, including all classes with appropriate associations, inheritance, etc.
2. Methods required to implement the selected use case(s).
3. Graphical user interface (GUI) elements necessary to demonstrate a working implementation of the selected use case. Every project **must include a GUI**.
4. The **minimum GUI implementation must allow** interaction between at least two classes connected by an association (target multiplicity: “many”).
    1. a. For example, if the diagram shows two classes connected by an association, e.g., *Company* and *Employee*, the GUI should include: a widget displaying multiple items (e.g., a *ListBox*) showing the list of companies; when a company is selected, another widget (also displaying multiple items, e.g. *ListBox*) shows the list of its employees, retrieved through **the defined association** (this generally means no use of query languages, including SQL).
    2. b. A GUI that only creates connections between objects but does not allow the above interaction is **insufficient for passing** the project.

    3. c. Similarly, solutions with only one widget, a single TextBox, a target multiplicity of "," or filtering data from the extent instead of using the predefined association are also **insufficient**.

5. The implementation must include sample data that demonstrates correct functionality of the application.
6. Pay attention to the quality, ergonomics, and usability of the GUI (e.g., window scaling, color scheme, interaction design). This is an **important** component of the final grade. GUI design and implementation (using dedicated editors if desired) should follow usability principles presented in the lectures.
7. **All data** stored in the system must be persistent (e.g., using a file, database, or dedicated library). Ensure the chosen technology is appropriate (e.g., serialization is not suitable for web applications) and correctly implemented (e.g., serialization requires a single shared file for all extents).
8. Avoid including obvious comments describing the source code.
9. **Each project** will be individually reviewed. During the defense, students can expect detailed questions about the implementation (source code), such as: *"What would happen if…?"*, *"Why was this done this way…?"*, *"Please make the following modification…"*. Students who have completed the project independently should have no difficulty answering these questions. Failure to answer these questions will result in **not passing the course**.
10. Solutions that do not meet the above conditions will **not pass the exercises**.
11. Due to the anti-plagiarism procedure (see below), the project implementation (source codes only) must be uploaded to the designated folder in the Gakko/EduX system before the individual defense. The file should be named as follows: *"ProjectImpl_LastName_FirstName_Number.zip"*.
12. Solutions that do not meet the above conditions will **not pass the exercises**.
13. Evaluation criteria and deadlines: see below.

Deadlines

Detailed information regarding deadlines is provided in the course description. Late submissions will receive a reduced score, with mini-projects graded at 50% if submitted past the deadline.

Passing the tutorials

Note: According to the "blended studies" regulations, passing the tutorials depends on the exam results - see the "Exam" section.

The final grade of the tutorials consists of the following components:

1. Points for MPx (24 + 24 + 24 + 28 = 100 pts.; required minimum: 50% of the total points;)

   For each of the items listed in the table for each MP, you can get the number of points calculated from the formula: (Number of points for a particular MP) / (Number of constructs), eg. MP1 for each construction can be obtained 24/10 = 2.4 points (total: 10 x 2.4 points = 24 points)

The number of granted points for each issue will be determined based on the correctnes, quality, completness, selected approach and readability of the code.

2. Evaluation of the project (it is required to get at least 50% from each part):

    1. The evaluation for implementation will be issued according to the following table (prerequisite: fulfillment of the requirements in section 4.2):

Criterion	Maximum points
Complexity, scope and correctness of the implemented functionality	20
Scope and correctness of the implemented object-oriented structures	25
Code quality (names, comments, JavaDoc, etc.)	5
Elegance of the implemented solutions	15
Persistency	10
GUI implementation (including usability)	20
Project presentation	5
Total	**100**

    Based on the above scoring, an assessment is calculated: 50 pts. → 3,0; 60 pts.→ 3,5; 70 pts → 4,0; 80 pts → 4,5; 90 pts → 5,0.

    2. The assessment for the documentation will be issued in accordance with the following table (prerequisite: meeting the requirements from 4.1):

Criterion	Maximum points
Scope of the business field	10
Documentation of the use case (scenario and diagrams)	10
Correctness and completness of the design class diagram	35
Correctness and completness of the activity diagram	10
Correctness and completness of the state diagram	10
Design of the GUI	10
Discussion of the design decisions	10
Readability and organization of the document	5
Total	**100**

    Based on the above scoring, an assessment is calculated: 50 pts. → 3,0; 60 pts.→ 3,5; 70 pts → 4,0; 80 pts → 4,5; 90 pts → 5,0.

You must receive a positive grade from each part. Therefore, people who, for example, pass mini-projects and do not complete the project documentation, **will not pass the classes**.

In addition, you can earn 5% of the total points for solving each of the tasks given in the lectures. These bonus points are added to the total number of points only if you have at least 50% points.

Plagiarism

All submitted source codes (mini-projects, final project) will undergo **anti-plagiarism procedure**. If borrowings from other programs are detected:

1. all programs where common code was detected **receive 0 points**;
2. **it is not possible to send a corrected version of such a program**, which may lead to **failing the tutorials** without a chance to correct them in the current semester;
3. no determination will be made to investigate who copied the source codes from whom.

The final exam

The MAS exam consists of two parts (the sum of points counts):

1. Test. You should rate each question (Y/N). The correct answer is +2 points, wrong -2 points, no answer 0 points;
2. Assignment. Describe and briefly discuss how to implement selected constructs in the given class diagram.

There are no exam exemptions.

Sample MAS exam: https://www.mtrzaska.com/mas-exam/.

Note: in accordance with the regulations of the "blended" study programme, **to pass the tutorials** one must obtain at least 30% of the points in the exam (in the first or second sitting). Those who do not reach this threshold **must repeat the tutorials**. This "30%" rule does not apply to students who passed the tutorials in one of the previous semesters and are only retaking the exam in the current semester.

Complementary materials

1. Electronic version of lectures for full-time studies
2. Sample programming tasks
   These sample programming tasks are only intended to guide programming material (learned during previous courses) and their solutions will not be assessed as part of the subject MAS (Modeling and Analysis of Information Systems). Students entering the MAS course should be able to solve the vast majority of the tasks.

3. 

   Book (*Polish only*): M. Trzaska: „Modelowanie i implementacja systemów informatycznych 2.0”. Year 2025. Pages 487. ISBN 978-83-976442-0-5. This is a completely new edition of the book (published in 2025), updated and expanded with new topics (over 200 additional pages). It can be useful both as a supplement to knowledge of modeling (e.g., class diagrams) and as an extension of programming skills (e.g., the Java Spring framework in various configurations, end-to-end and integration testing, and software maintenance):
   - electronic version in ePub (e.g. Kindle readers) and PDF formats is available in the following online bookstores: eBookPoint, Helion, Empik,
   - more info on my web apge.

4. Free on-line books:
   - Bruce Eckel - Thinking in Java: http://www.mindview.net/Books/TIJ/
   - Allen B. Downey - How to Think Like a Computer Scientist: Java Version: http://www.greenteapress.com/thinkapjava/
   - Robert Sedgewick and Kevin Wayne - Introduction to Programming in Java: An Interdisciplinary Approach: http://introcs.cs.princeton.edu/home/
5. All books related to conceptual modeling (based on UML) - the corresponding list can be found in the description of the PRI
6. Programming books (e.g. for Java or C#).

Implementation tools

Due to the fact that there is quite a large choice of technology for project implementation, there is no mandatory tool list. However, below is a list of tools that may be useful:

- CASE tools (diagram editors):
    - UMLet (*open-source*, multi-platform): https://www.umlet.com//,
    - Modelio (*open-source*, multi-platform): https://www.modelio.org/
    - Visual Paradigm Community Edition: http://www.visualparadigm.com
    - ArgoUML: http://argouml.tigris.org/
    - MagicDraw Community Edition: http://www.magicdraw.com
    - StarUML: http://staruml.sourceforge.net/en/
    - NetBeans for Java: http://www.netbeans.org/ (allows, among others things, creating UML diagrams and generating source code)
    - MS Visio (available under ELMS for PJATK students)
    - Extensive list of tools: http://en.wikipedia.org/wiki/List_of_UML_tools
- IDE
    - Eclipse for Java: http://www.eclipse.org/
    - NetBeans for Java: http://www.netbeans.org/
    - IntelliJ IDEA Community Edition: http://www.jetbrains.com/idea/
    - MS Visual Studio
- GUI editors
    - included in IntelliJ Idea IDE,
    - included in MS Visual Studio,
    - included in NetBeans.

LESSON 01

Classes in OOP

Classes vs. objects, class extents, extent persistence, simple, complex, and optional attributes in Java.

Educational outcomes

Student:

- explains how to implement constructs such as class extents, extent persistence, complex attribute, optional attribute, mult-valued attribute, class attribute, derived attribute in natural language (understanding);
- identifies and corrects errors in the source code, which is an implementation of the above structure;
- lists constructs implemented in the source code (understanding);
- implements the above structures using the selected programming language (application);
- indicates functional differences between two selected programs based on code snippets source (analysis);
- creates a program that implements specific business requirements using the constructs learned from the lectures (synthesis);

Justification

Modern programming languages are mostly based on the object paradigm. Therefore, its practical understanding is a necessary basis for working in this profession. Moreover, programmers often make systems based on analytical materials. And these largely use object-oriented graphical notations (such as UML). During this and the next lessons, you will deepen your theoretical bond, but first of all you will learn how to implement different concepts from object-orientness. This will allow you to design and implement modern business applications.

Object and class



DEFINITION

Object It is a being that has well defined boundaries and properties and is distinctive in the analyzed fragment of the problem domain.

The concept of the object is related to the problem of its identity. It is assumed that the object is recognized on the basis of existence rather than using some combination of its features. Examples of objects may be: a chair, a building, an invoice, a student, a person, etc.

It is worth noting that the above definition does not take into account the specificity of computer systems. In practice (eg. in the Java or C# language), the object has its own identifier (reference) - usually it is simply an address in computer memory.



DEFINITION

Class is a named description of a group of objects that share the same set of properties (invariants).

According to this definition, a class is not a collection of objects (for that there is a separate concept). This is a very important statement, because it is often confused. In other words, the class is like a recipe for an object. It describes its properties (attributes) ,behavior (methods) and construction (in the context of programming languages). Class name should be a singular noun (eg. employee). You can use the plural form if a single instance of the class (object) describes many elements (eg *Employees* when we store information about their groups). It is also possible to use a few words if it has a business justification.

How it is related to popular object-oriented programming languages? In languages such as Java, MS C#, C++ classes occur in a manner consistent with the above definition.

Unfortunately, this does not apply to all concepts known from object-orientness (UML).

The following code in Java will create a new class whose objects will describe some of the movies.

```java
public class Movies {
        /* Body of the class: attributes, methods, etc. */
}
```

KNOWLEDGE CHECK — Which statements are true?

- A class is just a bigger object.
- Class instance is an object.
- Classes in Java are presented directly.

EXPLANATION & ANSWERS: **Explanation:**

- The class describes the object, but it is not an object.
- By definition, an instance of a class can be called an object.
- In Java, we have a specialclasskeyword that allows direct creation of classes.

Abstract class



DEFINITION

Abstract class is a class that can not have its own objects (direct instances).

This concept also occurs directly in popular programming languages. All we need to add is the `abstract` keyword.

```Java
public abstract class Movie {
        /* Body of the class: attributes, methods, etc. */
}
```

We will return to this issue in case of the inheritance.

Class extent



DEFINITION

Class extent is a set of currently existing objects (in other words, instances) of a given class.

Of course, in a working computer system there will be many class extents, approximately (see inheritance lesson) one for each existing business class, eg Movie, Actor, etc.

Unfortunately, we have come to a term that does not occur directly in popular programming languages. We mean the fact that a programmer does not have default access to all currently existing objects of a given class. So there is no special keyword for the language or any other construct that will make the relevant list available. We can implement class extent ourselves, using existing language constructs. Let us stress once again: what we do is a sort of a workaround, not a native solution. From the compiler point of view, there is no class extent, and only some constructs that the developer will treat in this way.



COMMENTARY

I think that everyone agrees that practically every business application requires a class extent. But should we always implement it on the programming language side? It seems we do not, for example, when using a database or an object-relational mapper. We will return to this topic in subsequent lectures.

I think that a fairly obvious solution is to create a dedicated container that will store references to individual instances of the class (ie, objects). The open questions are:

- Where to hold this container?
- When and how to add objects to it?

For the first question (where to hold this container) we have two general approaches:

- Within the same business class.
- Using an external class, e.g. for the Movie class, its extent Movies.

Both approaches have their advantages and disadvantages:

- One could say that the first solution introduces to the business classes some technical elements,
- Someone else could criticize the second way because it multiplies beings: for each business class it creates the corresponding class that manages extent.

As you can see, there is not single perfect solution. For reasons that will become clear soon, despite all else the first way seems to be a better one.

Implementation of a class extent within the same class

To implement class extent in Java or C#, we need to create a container that will store references to objects. Because we want to include it in the business class and all of its objects must have access to it, we will use an attribute with the keyword `static`. In addition to the container, it is also worth adding some methods which will facilitate the addition or removal of objects.

```Java

public class Movie {

    //   Business part

    private String title;
    private float price;
    private LocalDate additionDate;  // requires Java 8+

    public Movie() {
        // Add to the extent
        addMovie(this);
    }


    @Override
    public String toString() {
        return "Movie: " + title + ", id: " + super.toString();
    }

    private static List<Movie> extent = new ArrayList<>();

    private static void addMovie(Movie movie) {
        extent.add(movie);
    }

    private static void removeMovie(Movie movie) {
        extent.remove(movie);
    }

    public static void showExtent() {

        System.out.println("Extent of the class: " + Movie.class.getName());

        for (Movie movie : extent) {
            System.out.println(movie);
        }
    }
}
```

Some of the more interesting snippets (the following numbers in parentheses refer to the listing line):



COMMENTARY

Note that in line (20), we refer to the collection as `List` (an interface, not a specific class). Thanks to this, we can easily replace its specific implementation with eg `ArrayList` on any other class implementing this interface.

This is a fairly common technique that raises the level of abstraction and improves code flexibility. It's worth using it.

- (9). Let's get back to our second question: how to add objects to extent. The first solution to this problem is simply a manual call from the programmer, adding a newly created object to the container. Will this work? Of course - unless someone forgets to do it. Then some of the objects will be in the extent (because he/she remembered to call the method), and some of them will not. Can we do anything about it? We will use the constructor, and more specifically add the appropriate method call to the constructor. This approach ensures that each created object of this class will be placed in the container.

- (20). Here is the declaration of our container. We used the so-called a parameterized class that allows a particular type (and its sub-types) to be stored. We had to use the keyword `static` to access this item from all objects in this class. This implementation uses the *List* type, but you can use other types, such as *ArrayList*, depending on your needs.
- (22, 26). Methods to add and remove items. Because the container on which they operate is declared as `static`, then the methods must also be decorated with the keyword.
- (30). An auxiliary method that allows you to display (to the console) objects in class extent. It is worth to pay attention to the "new" for loops. This makes it unnecessary to use iterators, which are less convenient to use.

A sample test code for this solution is listed below along with the result of its operation. The reader who runs this program will receive other numbers (they are addresses in memory where the objects are stored).

```
Java

public static void main(String[] args) {
                var movie1 = new Movie("Terminator 1", LocalDate.now(), 29.90f);
                var movie2 = new Movie("Terminator 2", LocalDate.now(), 34.90f);

                Movie.showExtent();
                        }
```

```
Plain Text

Extent of the class: mt.mas.Movie
Movie: Terminator 1, id: mt.mas.Movie@61a485d2
Movie: Terminator 2, id: mt.mas.Movie@1810399e
```

Implementation of class extent using an external class

Another way to implement extent management is to create an additional "technical" class for each business class, such as Movies for the Movie business class or MovieExtent. With this approach, all extent functionality is placed in a separate class and does not "clutter" our business class. An additional potential benefit is the ability to manipulate many different extents for eg. the Movie class. Although the practical usefulness of this can be debatable.

```java
public class Movie {
    // Class body
}

public class MovieExtent {

    // Extent implementation

    /** The extent. */
    private List<Movie> extent = new ArrayList<>();

    public void addMovie(Movie movie) {
        extent.add(movie);
    }

    public void removeMovie(Movie movie) {
        extent.remove(movie);
    }

    public void showExtent() {

        System.out.println("Extent of the class: " + Movie.class.getName());

        for (Movie movie : extent) {
            System.out.println(movie);
        }
    }
}
```

Some interesting snippets of the implementation:

- (1). A business class whose extent we want to manage.
- (10). A container that holds references to specific objects belonging to the extent. In contrast to the previous solution, this attribute is not marked as `static` (although it may be). Thanks to that, creating more instances of the `MovieExtent` class, we can create multiple extents for one class.
- (12, 16). Methods for adding and removing objects to / from extent.
- (20). Auxiliary method to display extent.

Let's compare two implementations and let's see if there is something missing in the latter. Yes - there is no automatic addition to extent in the business class constructor. Is this oversight?,Of course not. With this implementation we do not know the extent management object, and therefore we can not write a direct reference. We can pass it, for example, as a parameter to the business object constructor - is it convenient? Probably not.

The intorduced approach in action. More interesting places:

- (4). We create a MovieExtent object that will manage the business class Movie.
- (6). We create a business object describing a single movie.

- (7). Manually adding a created object to class extent. Without this the new object will not be stored in the extent.

```java
public static void main(String[] args) {
    // Test: Implementacja ekstensji przy uzyciu klasy dodatkowej

    var movieExtent = new MovieExtent();

    var movie1 = new Movie("Terminator 1", LocalDate.now(), 29.90f);
    movieExtent.addMovie(movie1);

    var movie2 = new Movie("Terminator 2", LocalDate.now(), 34.90f);
    movieExtent.addMovie(movie2);

    movieExtent.showExtent();
}
```

As we mentioned before, the first approach (with the implementation of extent within the same business class) seems to be more convenient.

KNOWLEDGE CHECK — Which statements are true?

- Class extent and object mean the same thing.
- Class instance is an object.
- Extent of a class may change over time.
- When implementing class extent within the same business class, the collection must be declared asstatic.

EXPLANATION & ANSWERS: **Explanation:**

- Class extent is a collection of all objects.
- By definition, an instance of a class can be called an object.
- Of course, by adding / removing objects we change the extent.
- Implementing class extent within the same business class, the collection must be declared asstaticbecause it must be shared by all objects.

Simple attributes



DEFINITION

Attributes are used to describe the properties of objects belonging to a certain class. Due to the different criteria they can be divided into several categories.



DEFINITION

Simple attribute stores atomic values, e.g.: number, text, boolean, eg. salary, name, color name, weight.

This type of attribute occurs in all popular programming languages, the same way like in object-oriented UML. Listing presents an example of a price attribute for the Movie class.

```Java
public class Movie {
        private float price;
}
```

KNOWLEDGE CHECK — Which statements are true?

- An example of a simple attribute isAddress.
- In Java, simple attributes are implemented with thesimplekeyword.

EXPLANATION & ANSWERS: **Explanation:**

- Addressis an example of a complex attribute.
- There is nosimplekeyword in Java.

Complex attributes



DEFINITION

Complex attributes consists of simple attributes or other complex attributes, such as address (can include street, house number, city, zip code), position coordinates in 3D space (consist of values for x, y, z axis)



COMMENTARY

An attribute or a class with an association?

When creating a data model, think carefully about using a complex attribute or a separate class connected by association. The attribute will not be shared with other objects, i.e. each object will have its own copy, which makes it difficult to search. In the case of associations, the same value will be combined with many objects, which significantly simplifies the search.

The complex attribute is described using a dedicated class (eg, date). This class can be provided by developers of a given programming language, external library, or user-created (by a programmer). In effect:

- In the business class (eg Movie), we store a reference to its instance, not a (complex) value.
- As a result, we can share it, for example, another object can hold a reference to the same date. This is (somewhat) in contradiction to the "theoretical" semantics of a complex attribute that should not be shared. In C ++ the situation is a little different - you can store another class object as a "value".

```java
public class Movie {
    private LocalDate additionDate;
}
```

We may consider one more approach that involves direct placement of the content of the complex attribute in the class. For example, instead of a complex attribute, we can put simple attributes in the class: street, house number, apartment number, city and zip code. Sometimes such a solution can be more convenient than creating a separate, dedicated class.

KNOWLEDGE CHECK — Which statements are true?

- An example of a complex attribute is a double last name.
- A compound attribute can be implemented as a class object.

EXPLANATION & ANSWERS: **Explanation:**

- Double surname is not a complex attribute because it is treated as a whole.
- One way to implement a complex attribute is to create a dedicated class to store it.

Single and multivalued attributes



DEFINITION

Single attribute has exactly one value, eg name.

Exactly the same semantics as in object-orientness. One attribute holds one value, such as a name (provided that a person has only one, if not, see next).



DEFINITION

Multivalued attribute may have many values, eg. names or known foreign languages.

Many values for such an attribute are stored in a container or regular array. This first solution is preferred when we do not know how many of these elements we will need. The type of container you select may depend on how you work with that attribute, such as whether you add items more often, or rather read them, etc.

```Java
public class Person {
        private ArrayList<String> names;
}
```

Theoretically, one more approach can be imagined, but it is not recommended and I would classify them as "tricks" (and, as you know, sooner or later they are the cause of a trouble). We can store multiple names, for

example, in a single String, separated by commas. The level of comfort and safety with this type of "repetitive attribute" is very low.

KNOWLEDGE CHECK — Which statements are true?

- The repetitive attribute can be implemented usingArrayList.
- Always use Java arrays to implement repetitive attributes.

EXPLANATION & ANSWERS: **Explanation:**

- Repeated attributes can be implemented using a collection, for example.ArrayList.
- When implementing repetitive attributes, we avoid using of ordinary arrays (due to their constant size).

Mandatory and optional attributes



DEFINITION

Mandatory attribute must have value. The system should not allow you to create an object without that value, eg. a name in the Employee class.



DEFINITION

Optional attribute may not have value. This is a situation that is quite correct from a business point of view, such as a maiden name in a worker class (men usually do not have a maiden name).

In this case, we have to individually analyze two types of attributes:

- Simple. Each simple attribute holds a value - it can not be empty. Even if we put zero, it is some kind of value - just 0.
- Complex. A complex attribute holds a reference to an object "which is its value.". Because it is a reference, it may be null, or "no value". For security (to get no exception), you should "manually" check that it is null.

From the above explanation, it is clear that complex attributes can be either required or optional. How about simple attributes - can we do soomething to store optional information such as salary? Of course we can - but it will be a little more complicated. The simple attribute is implemented as a special class that wraps a

simple value. Thanks to this we will be able to substitute null, that is, information about the lack of value. There are already such wrapping classes in Java, such as `Integer` for `int`.

```Java
public class Employee {
        private Double salary;  // could be null (note the capitalized letter)
        private int number;     // can not be null
}
```

In addition to the discussed ability to remember missing information, the implementation of optional attributes should also take this fact into account during processing them. Instead of directly reading the value, first check if it exists, e.g.

```Java
if(salary != null) {
        // calculate something
}
else {
        // no value
}
```

Alternatively, you can use a dedicated construct called `Optional` (requires Java 8+). Note the sample usages in the `getIncome()` and `toString()` methods:

```java
public class Employee {

	// ...

	private Optional<Double> extraBonus = Optional.empty(); // initialization without a value

	public Optional<Double> getExtraBonus() {
		return extraBonus;
	}

	public void setExtraBonus(Optional<Double> extraBonus) {
		this.extraBonus = extraBonus;
	}

	public double getIncome() {
		return getSalary() + getExtraBonus().orElse(0d);
	}

	@Override
	public String toString() {
		return String.format("Emp '%s', sal: %s, bonus: %s", getName(), getSalary(), getExtraBonus().isPresent() ? getExtraBonus().get() : "(no bonus)");
	}
}
```

KNOWLEDGE CHECK — Which statements are true?

- TheArrayListcollection may not have any elements, so it is suitable for implementing the optional attributedouble.
- An optional and repetitive attribute may useArrayList.

EXPLANATION & ANSWERS: **Explanation:**

- Simple attributes always have some value. For this reason, they are not suitable for implementing optional attributes (use the wrapper class, e.g.Double).
- TheArrayListcollection may not have any elements, so it is suitable for implementation of the optional attribute.

Object and class attributes



DEFINITION

Attribute of an object can have its own value for each of the objects in the extent, eg. name, surname, date of birth.

The same semantics as in object-orientness (in UML). Each object within a particular class can store its own value.

```Java
public class Employee {
        private string firstName;
        private string lastName;
        private BigDecimal salary;
}
```



DEFINITION

Class attribute has the same value for all objects of a given class, eg. for an Employee class it could be: minimum age, maximum age.

The way of implementation depends on the approach to extensibility:

- Extent within the same class. We put class attributes in the same class with the `static` keyword,
- Extent as an extra class. We implement class attributes in the external class (without the `static` keyword).

```Java
public class Employee {
        private static int minAge = 18;
        private static int maxAge = 68;
}
```

KNOWLEDGE CHECK — Which statements are true?

- Regardless of the implementation of extent, the class attribute must be declared asstatic.
- Implementation of an object attribute must use only simple types.

EXPLANATION & ANSWERS: **Explanation:**

- The class attribute stored in an external class does not have to be declared asstatic.
- An attribute of an object can be, for example, a complex one and in that case its implementation may use a dedicated class.

Specific and derived (calculated) attributes



DEFINITION

Specific attribute is one whose value can not be determined in another (indirect) way and must be stored in the system, eg. date of birth.

```Java
public class Person {
        private Date birthDate;
}
```



DEFINITION

Derived attribute is one whose value can be determined by the state of the system (usually using other attributes), eg. gross price using net price.

In Java or C++ there is no native way to implement derived attributes. They are "simulated" using methods, which means that we really refer to methods in the code, not to attributes. In the case of orthodox encapsulation, this is not a big problem, since all attributes are still hidden and access is based on methods (for Java so called `setter` and `getter` ,that is, methods that allow you to change the value and read it). Special attribute treatment is implemented in the body of the method that reads (and sometimes also changes) its value.

```java
final static double taxVat = 1.23;      // VAT as a constant

/**
* Derived attribute implemented as a method.
* @return
*/
public double getPrice() {
        return netPrice * taxVat;
}
```

It is also worth noting that in most cases, derived attributes are read-only (eg. age). Therefore, we only implement a getter. Sometimes, we store the calculated value (due to performance improvements) - of course this means data redundancy (with all the consequences).



COMMENTARY

Generally speaking, encapsulation consists of hiding information with particular emphasis on attributes. Access to such hidden attributes is achieved through methods. This gives us greater control and, for example, we can detect unauthorized changes of their value. Developers have long argued that this approach is useful (because it takes a bit more work to write these methods - although more and more modern programming environments can help with this). Orthogonal encapsulation means that the decision about whether to hide any element (attribute, method) depends only on the developer. According to the orthodox encapsulation, all attributes are hidden and can not be shared. Of course, C++, C# and Java are compatible with the orthogonal encapsulation philosophy.

Although this lecture is mainly about Java, it is impossible not to mention on this occasion the excellent mechanism implemented in MS C#: *properties*). We can define special "methods" that we use exactly as attributes. The example for the `price` attribute is shown below (C#):

```C#
private double netPrice;
private const double vatTax = 1.23;

// Derived attribute implemented as a property in C#.
private double price {
        get {
                return netPrice * vatTax;
        }
        set {
                netPrice = value / vatTax;
        }
}
```

KNOWLEDGE CHECK — Which statements are true?

- Implementing a derived attribute may require calculating and storing its value.
- Implementing a derived attribute always means creating bothsetXXXandgetXXXmethods.

EXPLANATION & ANSWERS: **Explanation:**

- A derived attribute can be implemented as a method. In some cases, to increase the performance, we may want to store the result as a real attribute (as in memory)cache).
- Implementing a derived attribute usually means that we need to creategetXXXmethods. In rare cases alsosetXXX.

LESSON 02

Classes in OOP (Part 2)

Multi-valued attributes, class attributes, derived attributes, and class methods in object systems.

Educational outcomes

Student:

- explains how to implement constructs such as class extents, extent persistence, complex attribute, optional attribute, mult-valued attribute, class attribute, derived attribute in natural language (understanding);
- identifies and corrects errors in the source code, which is an implementation of the above structure;
- lists constructs implemented in the source code (understanding);
- implements the above structures using the selected programming language (application);
- indicates functional differences between two selected programs based on code snippets source (analysis);
- creates a program that implements specific business requirements using the constructs learned from the lectures (synthesis);

Justification

Modern programming languages are mostly based on the object paradigm. Therefore, its practical understanding is a necessary basis for working in this profession. Moreover, programmers often make systems based on analytical materials. And these largely use object-oriented graphical notations (such as UML). During this and the next lessons, you will deepen your theoretical bond, but first of all you will learn how to implement different concepts from object-orientness. This will allow you to design and implement modern business applications.

Object methods

Methods are used to describe the behavior of an object. Each of them can return a value (but does not have to) and retrieve parameters (also does not have to). Contrary to the attributes where the variation was large, here we have a fairly simple division:



DEFINITION

Object method has access and only works on a specific object (for which it was called).

More precisely, they have access to its attributes and other methods of this object (that is defined in the class to which this object belongs). We can not directly refer to another object (including its contents) - even from the same class. We can say that a single object is not aware of the existence of other objects of a given class (unless it is accessed through the extent). Simplified examples:

- `string getName()`, we will return the name of the specific employee (the one for whom we have called this method).
- `void setSalary(double newSalary)`, we change the salary of a particular employee.
- `bool isFemale()`, we ask if a particular employee is a woman.

And how about programming languages? Object methods in Java, C# or C++ have the same semantics as in the UML. The concrete method operates on the object to which it was invoked. It has access to all its

elements: attributes and other methods. In the body of the method, we can use the special keyword `this`, which is a reference to the object to which the method was invoked (more on this in the books on Java/C++/C#). The sample code for a very simple method is shown in the following listing:

```Java
public float getPrice() {
        return price;
}
```

KNOWLEDGE CHECK — Which statements are true?

- In order to create an object method in Java, you must use the keywordobject.
- The object method can only modify the attributes of the instance to which it was invoked.

EXPLANATION & ANSWERS: **Explanation:**

- In Java, there is noobjectkeyword.
- The object method has access only to the instance to which it was called.

Class methods



DEFINITION

Class method has access to the whole extent, and therefore to all objects belonging to that class

The mothods may operate on the extent, but they do not have to: for example, a class method that returns a value without looking at extent (therefore, class methods operate on the extent is not completely accurate). As in the case of attributes, class methods are invoked in favor of a class, not a particular object. This allows them to be used even when there are no objects of the same class. Examples:

- `static Employee findBestWorker()`,
- `static void setSalary(double newSalary)`, An analogous example was given for object methods, but this one has a different semantics: we increased the salary for one employee, here for all of them (whole extent).

Class methods in the sense of UML unfortunately do not appear in popular programming languages. Someone could protest: how about the `static` keyword? After all, it solved the problem of class attributes -

does not this help us here? Partially yes, but not quite - I suggest to remind ourselves what we just wrote about class methods (and extents).

Class method is characterized by the following properties:

- It can be called for class. This allows us to use it even when there are no objects of the class. And this can be achieved using the `static` keyword, occurring in Java, and in C++, as well as in C#.
- The second, very important feature is access to class extent. And here comes the problem. We said that in the languages we are interested in, there is no class extent, which obviously means that the method with the word `static` can not automatically access it.

As a result, we need to implement the class method ourselves. Our approach depends on how we manage the extent:

- An extent within the same class. Thus class methods in the same class with the keyword `static`. Because in the same class there is also a container also declared with the keyword `static`, our class method has trouble-free access to it, and thus extent.

- Extent as an additional class. Class methods are placed in the add-on class - this time without the keyword `static`.

Below is an example of two class methods implemented within the same business class:

Java

```
package mt.mas;

import java.util.ArrayList;
import java.util.List;
import java.util.stream.Collectors;

public class Employee {
	private static List<Employee> extent = new ArrayList<>();

	private double salary;
	private String name;

	// ...

	public Employee(String name, double salary) {
		this.salary = salary;
		this.name = name;

		// Add to the extent
		extent.add(this);
	}

	public double getSalary() {
		return salary;
	}

	public void setSalary(double newSalary) {
		this.salary = newSalary;
	}

	public String getName() {
		return name;
	}

	public void setName(String name) {
		this.name = name;
	}

	@Override
	public String toString() {
		return String.format("Emp '%s' sal: %s", getName(), getSalary());
	}

	/**
		* Class method returning all employees having a salary higher than the
given minimum.
		* @param minSalary
		* @return
		*/
	public static List<Employee> findEmployeesWithMinSalary(float minSalary) {
		// Using Java 8 Functional Streams to find/process data
		return extent.stream().filter(emp -> emp.getSalary() >
```

```
minSalary).collect(Collectors.toCollection(ArrayList::new));
        }

        /**
                * Class method setting a new salary for all the employees.
                * @param newSalary
                */
        public static void changeSalary(double newSalary) {
                // For each employee in the extent
                for (Employee p : extent) {
                        p.setSalary(newSalary);
                }
        }
}
```

KNOWLEDGE CHECK — Which statements are true?

- To create a class method in Java, use theclasskeyword.
- Class method can be used even when there is no object of the class.

EXPLANATION & ANSWERS: **Explanation:**

- In Java, class methods are created using thestatickeyword.
- The class method is called in favor of the class, so it can be used even if there is no object of the class.

Overloading methods

Method overloading is not purely object-oriented concept because it does not use any specific concepts of objectivity. After all, it is used in all modern object-oriented languages, and is therefore worth discussing.



DEFINITION

Method overloading is the creation of another method with the same name as the overloaded method, but different number and/or type of parameters.

Why the second "same" method? Will that help us? Let's look at the following example:

```java
public float getPrice() {
        return price;
}
public float getPrice(float vatTax) {
        return price * (1.0f + vatTax / 100.0f);
}
```

In a class, eg. *Product* there is a method that returns its net price. What if we want to find out about the gross price? We can create a method, eg getGrossPrice(). Another way is to overload the basic method with a parameter defining eg VAT rate (in percentage). Thanks to that, by referring to it in the code of the program, we do not have to recall its special name - we just call the method with the parameter.

Persistence of an extent



DEFINITION

Persist class extent occurs when its objects "survive" the system shutdown - once it is restarted we will have the same objects.

The feature is extremely important. In one form or another it occurs in almost all computer systems: starting with games (we remember the state of the game, which is just defined by the state of objects in the game), through office applications (eg. editor in which I write this lecture), and ending in databases.

In programming languages such as Java, MS C# or C++, this feature does not occur directly. We mean that there is no special keyword, the use of which with the definition of the class, for example, will ensure the persistence of its extent. That is why you need to implement it somehow. One way is to manually store the extent on the disk and then load it. Other, more unusual may be the use of such memory in a computer that does not lose its state after power off (eg. flash).

The most common way to achieve the persistency is to use a disk file. This requires storing data in the file and reloading it in memory, after a reloading a system. This approach has a certain problem. The shortest can be characterized as follows: after loading we do not have the same objects, but similar ones. Is this a problem? Yes it can be. First of all, it is not a perfect persistence, because we do not have the same objects in it. Secondly, more importantly, from a practical point of view, we can have problems related to the links between objects. If you use native programming language references, they represent specific places in memory where the connected objects reside. If we load them from the file and re-create them, 99% will be placed somewhere else, but the read references show the "old" places. You have to deal with it somehow - we'll show you how to do it later.

Fortunately, modern programming languages like Java and C# provide a mechanism called *serialization*. This technique significantly automates and facilitates data sustainability. Let's take a look at two approaches: manual and (semi) automatic.

KNOWLEDGE CHECK — Which statements are true?

- One way to achieve extent persistence is to save it to a file.
- Business applications do not need extent persistence.

EXPLANATION & ANSWERS: **Explanation:**

- Class extent can be persisted eg in a disk file.
- The vast majority (all of them?) of business applications must ensure data persistency.

Persistence of extent - manual solution

Compared to the use of serialization, manual implementation has almost the same advantages - aside from one. The only drawback is quite laborious - especially if you want to solve the problem of linking objects optimally. Returning to the advantages of this approach:

- Speed. We directly select the items to be saved and the mode (such as binary, text, structured text) of the file. This allows us to use the maximum speed offered by a particular programming language.
- Total control over the final effect. It is us - the programmers are responsible for the whole implementation, which allows us to completely control what we write (because we do not have to want to store everything) and how we write.
- Greater resistance to changes in classsource code. We'll get back to the topic later.
- Small file. Because we save only the selected items, the file can be as small as possible. In one of the projects I was working on, the files I saved were quite large. We had to remember the next position coordinates on the screen of certain elements (as the number of int type - they occupy 4 bytes each). Because their values were in the range 0 - 1000, we decided to use the short type (2 bytes). After analyzing the data we have applied even more clever solution. We only remember the

differences between the coordinates. Because they are relatively small, they are in the range 0-255. This way, we can save them using one byte. As you can see, sometimes the way data is written has little in common with their physical storage in the class.

Of course, there is a lot of manual approaches to data persistence implementation. One possible solution is presented below:

```Java

public class Movie {

	//	[...]

	private String title;
	private float price;
	private LocalDate additionDate;  // requires Java 8+

	private void write(DataOutputStream stream) throws IOException {
		stream.writeUTF(title);
		stream.writeFloat(price);
		stream.writeLong(additionDate.toEpochDay());    // count of days where day
0 is 1970-01-01 (ISO)
	}

	private void read(DataInputStream stream) throws IOException {
		title = stream.readUTF();
		price = stream.readFloat();
		long epochDay = stream.readLong();
		additionDate = LocalDate.ofEpochDay(epochDay);
	}
}
```

Remarks to the above listing:

- (5). Standard attributes for storing business data.
- (9). A method that saves the business data to a file. More specifically, to the stream. Thanks to this separated functionality, you can also save other extras to the same stream, so we get one file containing all the infromation from our system. In the method's body, we save one attribute after another.
- (15). Data reading method. Notice the order - it is identical to the order of writing.
- (12), (19). Standard simple types in Java can not handle the date. Therefore, we write the number of days (12), and then construct a date object (19).

In addition to methods of writing and reading the state of a single object, we need methods to write and read the state of the whole extent. Naturally, we place them in a class that manages extents.

```java
public class Movie {

        //       [...]

        private static List<Movie> extent = new ArrayList<>();

        public static void writeExtent(DataOutputStream stream) throws IOException {
                // Number of objects
                stream.writeInt(extent.size());

                for (Movie movie : extent) {
                        movie.write(stream);
                }
        }

        public static void readExtent(DataInputStream stream) throws IOException {
                Movie movie = null;

                // Get the number of written objects
                int objectCount = stream.readInt();

                // remove the current extent
                extent.clear();

                for (int i = 0; i < objectCount; i++) {
                        movie = new Movie();
                        movie.read(stream);
                }
        }
}
```

Comment to the code above:

- (3). Because of clarity, the constructor that adds the newly created object to the container is not shown.
- (7). A method that writes extent to a given stream.
- (9). At the beginning, we save the number of objects in the extent - it will be needed for reading.
- (11). Then we iterate through the entire container containing references to individual objects and call the method that writes the single object (shown previously).
- (16). The method that reads the contents of the extent from the specified stream.
- (20). First we read the number of saved objects.
- (23). We clean up the existing extent (we remove all existing objects of the class).
- (25). Knowing the number of objects stored, we create new ones (26) and read their contents by calling the method for each of them (27).

Let's look how we can use it:

```Java

// a test method

final String extentFile = "d:\\mas-extent.bin"; // TODO update the file location

var movie1 = new Movie("Terminator 1", LocalDate.now(), 29.90f);
var movie2 = new Movie("Terminator 2", LocalDate.now(), 34.90f);

// A test: persistency of the extent (manual impl.)
try {
        // Write the extent to the given stream
        var out2 = new DataOutputStream(new BufferedOutputStream(new
FileOutputStream(extentFile)));
        Movie.writeExtent(out2);
        out2.close();

        // Read the extent from the given stream
        var in2 = new DataInputStream(new BufferedInputStream(new
FileInputStream(extentFile)));
        Movie.readExtent(in2);
        in2.close();
} catch (IOException e) {
        e.printStackTrace();
}

Movie.showExtent();
```

Remarks:

- (3). Specify the location of the file on disk that will hold the extent.
- (5). Create two instance instances (objects) of the Movie class. Thanks to the special constructor design, objects are automatically added to the extent.
- (11).Create output stream connected to the file.
- (12). Calling a class method that writes extent to a given stream.
- (13). Closing the stream. This is a very important step which ommited can result in various errors, such as file lock or loss of "saved" data.
- (16). Creating a stream for reading from a file.
- (17). Calling a class method that reads the contents of a file.
- (18). Closing the stream.
- (20). Essential exception handling.

Below we see the contents of the console after launching the above program. It is worth noting that the numbers after the "@" (as we mentioned they are addresses in memory) are different before writing and after reading. This is an illustration that the use of writing to a file results in creating similar objects (because the titles of the movie match), but not the same objects (because their computer identity (references - addresses) is different).

```
Plain Text

Extent of the class: mt.mas.Movie
Movie: Terminator 1, id: mt.mas.Movie@61a485d2
Movie: Terminator 2, id: mt.mas.Movie@1810399e
Extent of the class: mt.mas.Movie
Movie: Terminator 1, id: mt.mas.Movie@4439f31e
Movie: Terminator 2, id: mt.mas.Movie@5dfcfece
```

KNOWLEDGE CHECK — Which statements are true?

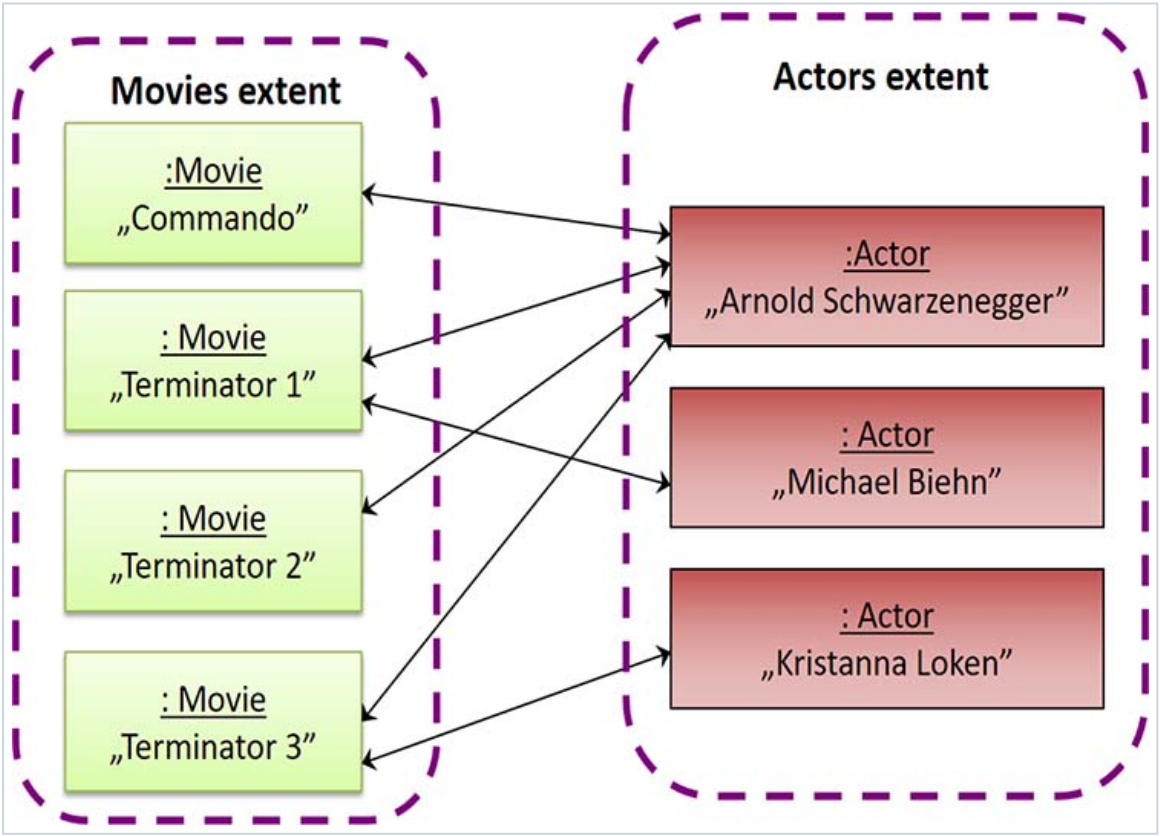
- The method storing a single object should, as a parameter, accept the file name.
- The method storing all extents should accept the file name as a parameter.

EXPLANATION & ANSWERS: **Explanation:**

- The method storing a single object should, as a parameter, accept a stream. This way, you can save a lot of data to one stream.
- The method storing all extents should accept the file name as a parameter. This will allow them to be stored in a single file.

Persistence of extent - problem with links

The manual implementation of persistence described above is very simple. It works pretty well when you save individual instances. Unfortunately, it does not allow proper treatment of linked objects. Let's look at the figure:



It presents a fragment of two extents of classes: *Movie* and *Actor*. The objects of these classes are linked in two directions, ie. the Movie has a connection to the Actors who played it, and the Actor is connected to all the Movies in which he/she played. What's the problem? Well, let's think about how we would write these extents to the file and then read them?

As in the presented approach, we are trying to save objects from the movie class - one by one. But how do we remember information about actors of a particular movie? We said earlier that writing a reference as a number would not work, since the object would be in a different place of memory when read (so the reference should be different). We can try to save this way:

- As before, we save individual attributes,
- When we come across an object, we call the `write()` method. The same way as we called this method from the saving extent. Will it work? There are two potential problems: looping (from the Movie we call the Actor, Actor from the Movie, etc.), and the same items are written multiple times. This is because the same object can be linked from many other objects. By using such a simple method, information about the actor "Arnold Schwarzenegger" will be recorded many times (because this actor played in many Movies).

As you can see, this is not the best way. You can try another approach:

- Each object has its own unique identifier.
- When we need to write information about an object, we will remember its id, not a reference.
- The read of this file is a two-step process: first load the data with the identifiers, and then replace the identifiers with the native references.

We will not give an example implementation because it would be rather complicated - instead we use an existing mechanism called *serialization*.

KNOWLEDGE CHECK — Which statements are true?

- The problem of extent persistence related to links is due to the poor performance of the Java platform.
- One of the ways to effectively maintain links between objects is to use references.

EXPLANATION & ANSWERS: **Explanation:**

- The problem of extent persistence, related to relationships, is caused by potential multiple retention of the same data.
- For effective persistence, variable references should be replaced with fixed identifiers.

Persistence of extent – serialization

Serialization is a mechanism implemented in Java/C# libraries. Allows you to automatically:

- saving graph of objects to a stream,
- reading the graph of objects from the stream.

What does it mean to write/read an entire object graph? TLet's get back to the example from the above drawing. During writing information about a movie class object named "Terminator 1", we also store information about all the objects it points to like "Arnold Schwarzenegger" and "Michael Biehn" (both from the `Actor` class). Also, information about all the objects pointed to by these objects is recorded, in this

example all the movies and all actors. In other words, the serialization mechanism makes sure that all reachable objects (regardless of how many intermediaries have to "pass") were in a valid state. And it does so smartly that it solves the problems we previously mentioned: looping and rewriting the same objects.



COMMENTARY

If you want the saving to take place with optimal utilization of resources, then we need all the serialized elements to be sent to a single stream - even those from different classes. Otherwise "computer" will not be able to recognize these repetitions. For this reason, serialization to multiple files (for example, one file per extents) is a fatal error - and so every file in the file stores the entire graph of related objects, including multiple extents (when multiple objects are interconnected).

The only requirement that must be met to use serialization is the "special" implementation of the `Serializable` interface by the class (and all its attributes). This implementation is "special" because in the simplest case we declare its implementation by the class, but we do not need to implement its methods manually. This will be taken care by the Java "compiler". This interface must be implemented not only by the

object from which the storing starts (object graph), but by all objects that are being written. Otherwise, we'll see an exception. The good news is that most classes provided by Java or C# implement this interface.

From the programmer's point of view we can list the following features of this approach to the persistency of an extent:

- Ease of use. The simplest case requires only adding two words to the class definition.
- Slower writng/reading.
- Larger file than with manual write. These last two features are due to the fact that together with the business data, various technical information are stored. Not only are they occupying the disk space, they also slows down the whole process. Another reason for the slowdown is the need to get information about building an object while running a program.
- Pretty high sensitivity to changes in class code. That is to say, most of the changes we make to the source code using serialization will result in incompatibility of the saved data. In other words: we serialize class objects, make changes to the source code, and when we try to read data we get an exception. This situation can occur even if we do not change the attributes of the class (because that would be quite obvious), but even if we add some method.

Fortunately, we have some control over how serialization works:

- Adding and overriding (own read/write implementation) of the following methods:
    - `private void writeObject(ObjectOutputStream stream) throws IOException`,
    - `private void readObject(ObjectInputStream stream) throws IOException, ClassNotFoundException`,
- Label the selected attributes with the `transient` keyword. This will not automatically save them.

Below there are two listings: a business class code with a declared implementation of the `Serializable` interface and how to use it. As we mentioned, it is worth noting that we do not provide bodies for the methods declared in this interface. So, as we promised, the whole thing is to add two words: `implements Serializable`.

```java
public class Movie implements Serializable {

        //       [...]

        private String title;
        private float price;
        private LocalDate additionDate;  // requires Java 8+

        private static List<Movie> extent = new ArrayList<>();

        public static void writeExtent(ObjectOutputStream stream) throws IOException {
                stream.writeObject(extent);
        }

        public static void readExtent(ObjectInputStream stream) throws IOException,
ClassNotFoundException {
                extent = (ArrayList<Movie>) stream.readObject();
        }
}
```

And the demonstration of serialization's utilization:

```java
final String extentFile = "d:\\mas-extent.ser"; // TODO update the file location

var movie1 = new Movie("Terminator 1", LocalDate.now(), 29.90f);
var movie2 = new Movie("Terminator 2", LocalDate.now(), 34.90f);

// A test: persistency of the extent (impl. using serialization)
try {
        // Write the extent to the given stream
        var out = new ObjectOutputStream(new FileOutputStream(extentFile));
        Movie.writeExtent(out);
        out.close();

        // Read the extent from the given stream
        var in = new ObjectInputStream(new FileInputStream(extentFile));
        Movie.readExtent(in);
        in.close();
} catch (IOException | ClassNotFoundException e) {
        e.printStackTrace();
}

Movie.showExtent();
```

KNOWLEDGE CHECK — Which statements are true?

- In order to improve performance, each extent should be serialized into a separate file.
- In order to use serialization, you just need to implement theSerializableinterface.

EXPLANATION & ANSWERS: **Explanation:**

- Serializing extents to multiple files causes the same data to be written multiple times (related objects).
- In order to use serialization, you just need to implement theSerializableinterface.

Other ways of achieving extent persistency

The above-described ways of implementing the persistency do not use any additional components - whether in the form of libraries or external applications. These are not the only options a programmer has. You can use at least two other approaches.

The first one is to use a database. Its greatest disadvantages include the need to map the Java constructs (classes) to structures from the database. Since currently most popular database solutions are based on relational technologies, the developer is forced to abandon most object-oriented constructs.

The second solution assumes utilization of libraries that make it easier to work with data (not just persistence). Often, under the intermediary layer (which is just a library), there is a database management system.

We will return to these ideas during next lessons.

LESSON 03

Classes in OOP (Part 3)

Encapsulation, access modifiers, method overloading, overriding, and object identity.

Educational outcomes

Student:

- explains how to implement constructs such as class extents, extent persistence, complex attribute, optional attribute, mult-valued attribute, class attribute, derived attribute in natural language (understanding);
- identifies and corrects errors in the source code, which is an implementation of the above structure;
- lists constructs implemented in the source code (understanding);
- implements the above structures using the selected programming language (application);
- indicates functional differences between two selected programs based on code snippets source (analysis);
- creates a program that implements specific business requirements using the constructs learned from the lectures (synthesis);

Justification

Modern programming languages are mostly based on the object paradigm. Therefore, its practical understanding is a necessary basis for working in this profession. Moreover, programmers often make systems based on analytical materials. And these largely use object-oriented graphical notations (such as UML). During this and the next lessons, you will deepen your theoretical bond, but first of all you will learn how to implement different concepts from object-orientness. This will allow you to design and implement modern business applications.

Extent Management - ObjectPlus class

The presented methods of implementing extent management will be (almost) the same for every business class in the system. Assuming an implementation within the same business class, we have to put almost the same elements in each of them:

- container that holds references to their instances,
- Methods to manage (add, delete, etc.).

Is it possible to unify it so that we do not have to write many times (almost) the same code? Fortunately, the answer to the above question is yes. We will use inheritance existing in Java. Although we have not discussed it yet, but for the time being, we will have enough basic information from the programming

lessons. We will create a base class from which will inherit all business classes in our application. Let us name it, for example, `ObjectPlus`, and equip it with:

- persistance,
- extent management,
- maybe some other elements.

We will use the first approache to implement extent, ie. within the same class.

KNOWLEDGE CHECK — Which statements are true?

- Our own implementation of extent management is due to the fact that native Java mechanisms have poor performance.
- By inheriting fromObjectPlus, business classes will receive extent management.

EXPLANATION & ANSWERS: **Explanation:**

- Our own implementation of extent management is due to the fact that Java does not provide such functionality.
- Inheritance from theObjectPlus, provides all functionality implemented here, including extent management.

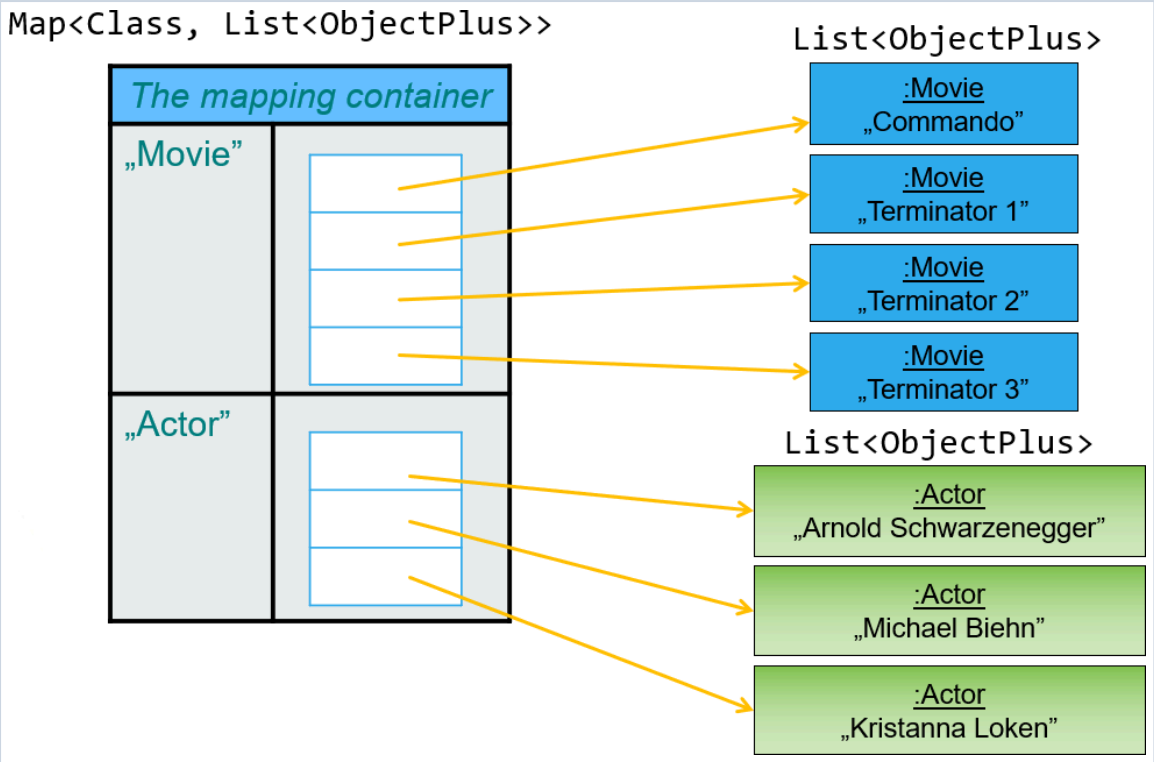
How to store many extents?

Referring to previous lessons, we know that we need to create a class attribute container (so that all objects in a class can access it). So at first glance it looks like an identical solution for a single class. Are you sure? Let's take a look: For example, we create a `Movie` object inherited from `ObjectPlus` class and, thanks to a special reference in the constructor, we add it to the extent (which is stored in the class attribute of the `ObjectPlus` class). Then we create an `Actor` object, which also inherits from the `ObjectPlus` class. Also, thanks to a special constructor, the object is added to the extent stored in the `ObjectPlus` class. As a result, we have two objects, `Actor` and `Movie`, which are in the same extent (in the `ObjectPlus` class). We did not want it. How to fix this?

Because all business classes inherit from one superclass (`ObjectPlus`), we can not use a simple container. We will use a mapping container that stores keys and values (dictionary):

- The key will be the name of the specific business class, such as `Actor` or `Movie`,
- The value will be a collection containing the references to its instances (the real extent).

In other words, this new container (dictionary) will contain many extents rather than a sinle one. The figure below shows schematically the content of the container. We see that there are currently two entries: films and actors. The extent of actors put in a dedicated collection stores references to three objects. Extent of movies saved in another collection store references to four movies.



The code declaring the class and the corresponding container (in this case it is a `Map` parametized with appropriate types) is shown below. Let's also take a look at the implementation of the `Serializable` interface - which will give us persistency of all the extents for free.

```java
public abstract class ObjectPlus implements Serializable {
                private static Map<Class<? extends ObjectPlus>, List<ObjectPlus>>
allExtents = new HashMap<>();

                                // ...
                        }
```

KNOWLEDGE CHECK — Which statements are true?

- Many class extents can easily be stored inArrayList.
- Hashtableis a mapping container.

EXPLANATION & ANSWERS: **Explanation:**

- You should use the map to store many extras. This will allow individual values to include collections with extents for specific classes.
- One of the maps provided by Java isHashtable.

The role of the constructor

In previous lessons, objects were added automatically to the extent, using the method call in the constructor. We will use this solution now, but due to the generality of our design, it will be a little more complicated.



COMMENTARY

Generic solution is one that can be applied to many cases. This is a fairly common trend in computer science: instead of creating a piece of the program just for a particular case, with a bit more work, we are able to write a library that can be used in many projects. This will save us work and reduce the number of errors. Unfortunately, this usually means (somewhat) more complicated implementation.

Adding to an extent

Let's analyze the following code:

```Java
public abstract class ObjectPlus implements Serializable {
        private static Map<Class<? extends ObjectPlus>, List<ObjectPlus>> allExtents = new
HashMap<>();

        public ObjectPlus() {
                List extent = null;
                Class theClass = this.getClass();

                if(allExtents.containsKey(theClass)) {
                        // An extent of this class already exist
                        extent = allExtents.get(theClass);
                }
                else {
                        // An extent does not exist - create a new one
                        extent = new ArrayList();
                        allExtents.put(theClass, extent);
                }

                extent.add(this);
        }
}
```

- (2). A private class attribute that is one of the Java mapping containers. As mentioned above, it stores keys (class names for specific extents) and values containing collections with references to instances of that class.
- (4). Constructor of the `ObjectPlus` class. Contains code that adds the object to a specific extents.
- (6). Thanks to a technique called reflection we get information about the membership of the object that is constructed. The returned value, which is an instance of class `Class`, will be used as a key in our `Hashtable`. We could use one of its methods to determine the class name, but it would be more efficient.
- (8). We are checking if the mapping container already contains a key describing our class.
- (10). If so, then the value of the key is recovered from the key, that is, the collection containing the extent.
- (14, 15). If not, then we create a new empty collection that will contain instances and add it to the main mapping container.
- (18). We add information about the new instance that is being created. It is worth noting that here we always have the correct extent - either because we retrieved it on the basis of a key, or created a new one.

KNOWLEDGE CHECK — Which statements are true?

- In the above listing, marking theextentvariable asstaticwould not affect the operation of the program.
- The above code, as the key in the map, uses an object ofClassclass.

EXPLANATION & ANSWERS: **Explanation:**

- In the above listing, markingextentasstaticwould result in a compilation error.
- The code above uses the object of theClassclass returned by thegetClass()method.

Ensure persistency

Thanks to serialization we do not have much to do. As we remember, a whole graph of objects is saved, so just send the `Hashtable` to our file .

```Java
public abstract class ObjectPlus implements Serializable {
	private static Map<Class<? extends ObjectPlus>, List<ObjectPlus>> allExtents = new
HashMap<>();

	// [...]

	public static void writeExtents(ObjectOutputStream stream) throws IOException {
		stream.writeObject(allExtents);
	}

	public static void readExtents(ObjectInputStream stream) throws IOException,
ClassNotFoundException {
		allExtents = (HashMap) stream.readObject();
	}
}
```

Actually, because all extents are in one map, we could even immediately create a stream inside the method and pass it the file name.

KNOWLEDGE CHECK — Which statements are true?

- Removing thestatickeyword from the above code would mean that the program would be faulty.
- Hashtableis one of several possible classes that can store a lot of extents in Java.

EXPLANATION & ANSWERS: **Explanation:**

- Removingstaticwould turn class invariants into an object (an extent and operations on it should be a class one).
- Hashtableis one of theMapinterface implementations that is convenient for storing multiple extents.

Getting an extent

Of course, sooner or later we will want to retrieve a certain extent and do something about its objects. The following method will do this:

```java
public abstract class ObjectPlus implements Serializable {
	private static Map<Class<? extends ObjectPlus>, List<ObjectPlus>> allExtents = new
HashMap<>();

	// [...]

	public static <T> Iterable<T> getExtent(Class<T> type) throws
ClassNotFoundException {
		if(allExtents.containsKey(type)) {
			return (Iterable<T>) allExtents.get(type);
		}

		throw new ClassNotFoundException(String.format("%s. Stored extents: %s",
type.toString(), allExtents.keySet()));
	}
}
```

Please note that this is a generic method, parameterized with a type. It returns the `Iterable<T>` type which gives two benefits:

- we can only iterate over it and it is not possible to add new objects,
- the result contains objects of the specific class, which means strong typing with all the benefits.

Sample usage (we iterate over the objects of the `Movie2` class, which makes possible, e.g. direct calls of its method, e.g. `getTitle()`):

```java
var movieExtent = ObjectPlus.getExtent(Movie2.class);
for (Movie2 movie : movieExtent) {
	System.out.println(movie.getTitle());
}
```

Helper methods

Our small library would not be useful without auxiliary methods, such as displaying extent:

```java
public class ObjectPlus implements Serializable {
	private static Map<Class<? extends ObjectPlus>, List<ObjectPlus>> allExtents = new
HashMap<>();

	// [...]

	public static void showExtent(Class theClass) throws Exception {
		List extent = null;

		if(allExtents.containsKey(theClass)) {
			// Extent of this class already exist
			extent = allExtents.get(theClass);
		}
		else {
			throw new Exception("Unknown class " + theClass);
		}

		System.out.println("Extent of the class: " + theClass.getSimpleName());

		for(Object obj : extent) {
			System.out.println(obj);
		}
	}
}
```

Important parts for this implementation and example of use in the business class:

- (6). As a method parameter, we provide an instance of the Class that we use to identify class membership.
- (9). We are checking to see if there is a key - that is, information about extent.
- (11). If so, then we return the value for this key, that is, the container containing the extent of that particular class.
- (14). If the key does not exist, then we do not have information about the extent. In this situation we throw an exception. This can be solved differently, eg. display a message, (silently) quit the method.
- (17). Display name of the class.
- (19). Then we iterate through the collection containing the extent and display information about each object. In fact, in this situation, it is implicitly called `toSring()` from a particular object.

```java
public class Movie2 extends ObjectPlus implements Serializable {
	// ...
	public static void showExtent() throws Exception {
		ObjectPlus.showExtent(Movie2.class);
	}
}
```

Examples of use

Now there will be a nice part - we'll show you how easy it is to use it. We just need to inherit from the `ObjectPlus` class (line 1) and call the constructor from the superclass (line 11). As you can see in the business class itself, we do not have much work involved in extent or persistency. We can focus on implementing business functionality.

```Java

public class Movie2 extends ObjectPlus implements Serializable {
        private String title;
        private float price;
        private Date additionDate;

        public Movie2(String title, Date additionDate, float price) {
                // Call the constructor from the super class
                super();

                this.title = title;
                this.additionDate = additionDate;
                this.price = price;
        }

        //       Business implementation
}
```

Potential improvements

The `ObjectPlus` class is only a simple prototype and therefore can be improved in many ways, for example:

- use *Java Generics*. This gives us better typological control. However, due to the generality of the solution, we will have to go to some compromises anyway;
- persistency for class attributes (class members marked as `static` are not saved during serialization);
- better management of adding to extent for inherited classes. We will return to this issue in the next lessons.

Source codes



Download all sources for MAS lectures.

LESSON 04

Associations in OOP

Binary associations, multiplicity constraints (1..1, 1..*, *..*), navigation, and bidirectional link management.

Educational outcomes

Student:

- explains how to implement constructs such as: association and link, cardinalities and association roles, directed and recursive association, attribute association and qualifed in natural language (understanding);
- identifies and corrects errors in the source code, which is an implementation of the above structure;
- lists constructs implemented in the source code (understanding);
- implements the above structures using the selected programming language (application);
- indicates functional differences between two selected programs based on code snippets source (analysis);
- creates a program that implements specific business requirements using the constructs learned from the lectures (synthesis);

Justification

Modern programming languages are mostly based on the object paradigm. Therefore, its practical understanding is a necessary basis for working in this profession. Moreover, programmers often make systems based on analytical materials. And these largely use object-oriented graphical notations (such as UML). During this and the next lessons, you will deepen your theoretical bond, but first of all you will learn how to implement different concepts from object-orientness. This will allow you to design and implement modern business applications.

Links and associations



DEFINITION

Link connects objects. Most often we deal with binary relationships, ie. those that only connects two objects at a time.



DEFINITION

Association is a description of a group of links with the same semantics. It is used to describe connections between classes.

The relationship between an association and a link can be thought the same way as a class and an object. The link is something "real", between existing objects, eg `Employee` Blacksmith is `hired by` a `Company` Softmakers. By contrast, an association generally describes `employment` between an `Employee` and a `Company`. In other words, the link is an instance of an association (just as the object is an instance of a class).

The association name should be matched to make a meaningful expression in conjunction with the class names, eg Employee *works in* Company.

KNOWLEDGE CHECK — Which statements are true?

- Association is an instance of a link.
- Association describes a link.

EXPLANATION & ANSWERS: **Explanation:**

- Link is an instance of an association (the same as an object is an instance of a class).
- The association describes a link (similarly as a class describes an object).

Cardinalities of associations

In addition to the name, the association is described by the cardinalities. It defines how many objects can be connected to a particular object. We will remember two values: one for each side.



The information from the above figure should be read as follows:

- The specific object of the A class is connected with *Y* objects of the B class,
- The concrete object of the B class is connected with *X* objects of the A class.
- Instead of *X, Y*, we put one of the following values (in the older version of UML 1.x combinations were also allowed):
  - 0,
  - 0..1,
  - * (means zero or more),
  - 1…* (means one or more),
  - particular number, eg. 4.
- In the older versions of UML (1.x), cardinalities could be arbitrarily merged using the notation, eg. "0 ...4, 7, 8, 13, 45...*". Of course this did not always have a business sense. Therefore, in UML2, such "combined" syntax is excluded. If no information is given, it means that the cardinality is 1. Usually we use: *1*, *"0, 1"*, "1...*".

You might wonder why you should give the cardinality information. After all, you could make an assumption that everywhere is "*" and we can store every business situation. The main reason is the desire to fine-tune the model, because there are differences in implementation of the "one" and "many" (see below). An additional benefit is to validate some rules, eg. if there is a "1" somewhere, then we will not be able to create an object ("system" will not allow it), which will not be associated with exactly one object.



COMMENTARY

Particularly cautious must be in the case of the 1 - 1 combination. From the data storage point of view, such a connection is not needed at all and both classes can be simply merged together. However, in some situations, we want to stress the existence of two different beings. In other words, when we see an association of 1 - 1 in the diagram, let's check it twice if it is really needed.

KNOWLEDGE CHECK — Which statements are true?

- The way in which associations and links are implemented varies depending on the defined cardinalities.
- Correctly defined cardinalities reduce the risk of making an error.

EXPLANATION & ANSWERS: **Explanation:**

- We store many bindings using the collection, and "1" with a single reference.
- One of the reasons for defining cardinalities is the ability to control them by implementing business rules.

Association roles

Placing a name is not the only way to describe an association. Instead, you can use role names. Note that the names are located at the target class, e.g. the *Employer* role with the `Employee` class. With this approach, we have a comprehensive description: from the point of view of each class. This is particularly useful when, at a later stage, we will transform the diagram into an implementation version (see below).

Two approaches to association implementation

Unfortunately in Java, MS C#, C++ associations do not occur directly. Therefore, we will have to implement them somehow. To do this we can use one of two approaches. The main difference between them is the way we store information about connections - we can use:

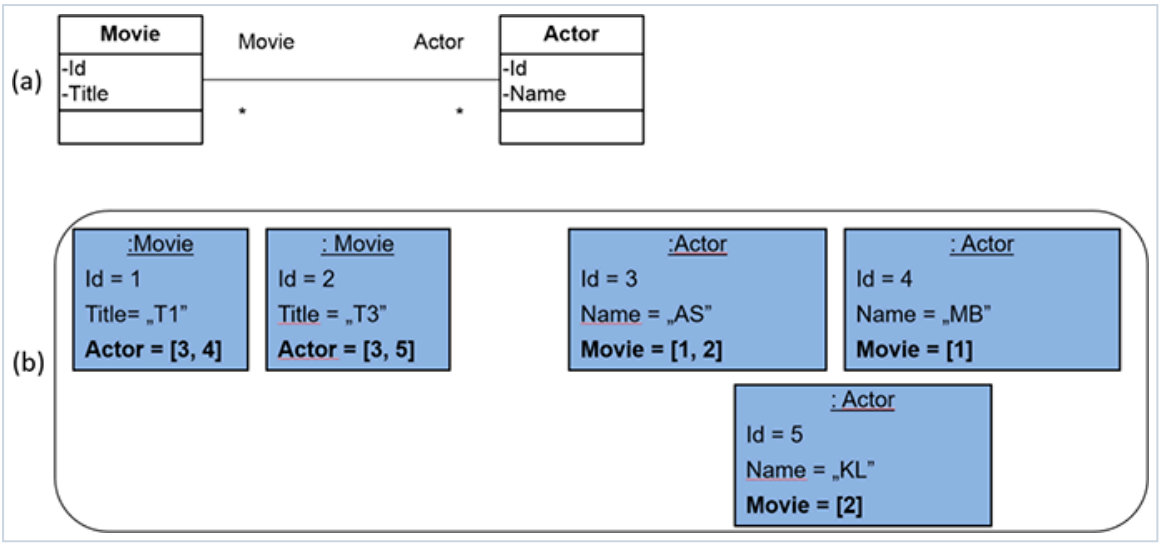
- custom identifiers, such as numbers,
- native programming language references (Java, C#) or pointers (C++).

Immediately the question arises: which approach is better? Which one to use? Traditionally there is no one universal answer. Depending on our needs the first or the second solution may seem more useful. Although it should be noted that in most cases we will use native programming language references - see further.

Implementation using identifiers

Let's start by discussing the implementation. For each business class we add an attribute that is an identifier, such as the `int` number. This number will allow to uniquely identify the object. Depending on business needs, its uniqueness may be within a single extent or all objects in the system. We keep information about connected objects by remembering their identifiers. Depending on cardinalities the storage will be a bit different.

Of course, in order to use bi-directional associations, we need to use pairs of identifiers. Let's look at the figure below. Part (a) shows a simple class diagram that allows you to store movie information and actors (or, in other words, the actors and movies they played). Part (b) contains a fragment of a sample situation that we have in the system using identifiers. Therefore, there are "actor" and "movie" attributes in the "objects" that are not in the class diagram. This is because they are equivalent to association roles. As you can see, the movie "Terminator 1" has an identifier of 1 and its third part "2". Note the attribute named "actor". It contains numbers that are the identifiers of the actors describing the actors playing the movie, eg "Id = 3" means actor "AS". And the other way: The actor "AS" has a "Movie" attribute that stores the IDs of the movies in which a particular actor was playing, for example Id = 3 would be "Movie = [1, 2]".



As you may know, the way is simple enough but effective. It allows you to store any connections between objects. How to implement it in Java? Let's look at the code:

Java

```
public class Actor {
	private int id;
	public String name;	// public for simplicity

	public int[] movieIds;

	private static List<Actor> extent = new ArrayList<>();

	public Actor(int id, String name, int[] movieIds) {
		// Add to the extent
		extent.add(this);

		this.id = id;
		this.name = name;
		this.movieIds = movieIds;
	}

	public static Actor findActor(int id) throws Exception {
		for(Actor actor : extent) {
			if(actor.id == id) {
				return actor;
			}
		}

		throw new Exception("Unable to find an actor with the id = " + id);
	}
}

public class Movie {
	public int id;
	public String title; // public for simplicity

	public int[] actorIds;

	private static ArrayList<Movie> extent = new ArrayList<Movie>();

	public Movie(int id, String title, int[] actorIds) {
		// Add to the extent
		extent.add(this);

		this.id = id;
		this.title = title;
		this.actorIds = actorIds;
	}

	public static Movie findMovie(int id) throws Exception {
		for(Movie movie : extent) {
			if(movie.id == id) {
				return movie;
			}
		}
```

```
                throw new Exception("Unable to find a movie with the id = " + id);
        }
}
```

Some important places to comment:

- (1). Definition of the business class Actor.
- (2). A private attribute of the type `int` that acts as an identifier.
- (3). Normal business attribute.
- (5). An array of `int` numbers that holds the movie IDs in which the actor was playing. It is worth noting that a single number is not sufficient because the size of the association informs us about the possibility of multiple associations. It's better to use a collection here (instead of an array), but we wanted to create the simplest possible example.
- (7). Class attribute that stores extent. This type of element is essential in our approach. Why? Because we describe each connection with identifiers and we must have a way to get the corresponding object. To do this, we search the extent until we find the object with the given identifier. Is it possible to make it in a more efficient way? Of course - we can use a mapping structure, where the key is id, and the value is the object reference.
- (9). A constructor.
- (11). Within the constructor we add the object to the extent.
- (13). Further, business part of the constructor.
- (18). The method that returns a reference to the corresponding object based on the identifier (number) - see note to point (5).
- (19). Iterate over all the objects in the extent. When we find an instance with the given identifier, we return it (18).
- (25). If we look through all the extent and we do not find the object with the specified identifier, we throw an exception. Instead of an exception, we could use something less invasive, such as null or a message to the console.
- (29). Movie class - similar to the Actor class.

And how can we use such classes?

```Java

var movie1 = new mt.mas.associationsid.Movie(1, "T1", new int[]{3, 4});  // The 'var'
requires Java 10+
var movie2 = new mt.mas.associationsid.Movie(2, "T3", new int[]{3});

var actor1 = new mt.mas.associationsid.Actor(3, "AS", new int[]{1, 2});
var actor2 = new mt.mas.associationsid.Actor(4, "MB", new int[]{1});
var actor3 = new mt.mas.associationsid.Actor(5, "KL", new int[]{2});

// Show information about the movie1
System.out.println(movie1.title);
for(int i = 0; i < movie1.actorIds.length; i++) {
        System.out.println("   " +
mt.mas.associationsid.Actor.findActor(movie1.actorIds[i]).name);
}

        // Show information about the actor1
System.out.println(actor1.name);
for(int i = 0; i < actor1.movieIds.length; i++) {
        System.out.println("   " +
mt.mas.associationsid.Movie.findMovie(actor1.movieIds[i]).title);
}
```

Let's look at:

- (1). Creating an instance of the `Movie` class. We provide the identifier, title, and array of `int` numbers referring to the identifiers of the actors that are playing in it.
- (4). Creating objects describing actors. As in the case of movies, we provide: id, initials, and array of numbers containing references to the movies. Note that if the "3" actor played in the movie "T1", actress "3" must also have information about the movie "1". This makes our relationships bi-directional.
- (9). View movie information. In the loop, we call the `findActor(...)` method, which returns the reference to the object describing it.

As a result, we get the state of the console shown below:

```Plain Text

T1
        AS
        MB
AS
        T1
        T3
```

How can we summarize the presented solution? Let's start with the flaws, or rather the essential one. It is necessary to search for an object based on its identification number, which can lead to performance problems. You can definitely improve it by using mapping containers instead of a regular list or an array. The sample code illustrating this modification is shown below.

```java
public class Movie {

        // [...]

        private static Map<Integer, Movie> extent = new TreeMap<>();

        public Movie(int id, String title, int[] actorIds) {
                // Add to the extent
                extent.put(id, this);

                // [...]
        }

        public static Movie findMovie(int id) throws Exception {
                return extent.get(id);
        }
}
```

- (5).The class attribute is a map where the key is `Integer`, and the value is a reference to the corresponding object.
- (9). We see that in the constructor we also add the object to the extent, but due to the specificity of the map we do it differently than before.
- (15). By using the method built into the map our search has also been simplified (and it is much faster as well).

Remember that despite the use of the mapping structure, we still need to search in the extent. The map is much faster than the iterative browsing of the whole extent, but it may take a little while.

Let's move on to the advantages. It can be summarized in one sentence: independence of individual objects from each other. Because we do not use Java references, the virtual machine knows nothing about our connections. This is very important when, for example, we want to read one object from a database or upload it over a network.

Implementation with references

Now let's take a second approach to storing information about related objects. This solution is probably more common, and can be summarized in the following points:

- We use native references (Java, C#) or pointers (C++) to process related objects. This way we do not have to search for objects.
- Having a reference (or a pointer) pointing to the object, we have an instant access to it (faster is no longer possible). Simply virtual machine reads from a specific memory location, appropriately interpreting the data contained there.
- As in the previous case, this way also requires creating reference pairs (if we want to navigate in two directions - and mostly we want).

Depending on the cardinalities of the association we use different constructions:

- Cardinality 1 to 1. A single reference on each side. As you can see in the code below, in each class we have a single reference pointing to the connected object. For the `Actor` class, this is a reference *movie* of the type `Movie`, and for the `Movie` class: *actor* of type `Actor`.

```java
public class Actor {
        public String name;     // public for simplicity
        public Movie movie; // implementation of an association, cardinality "1"

        public Actor(String name, Movie movie) {
                this.name = name;
                this.movie = movie;
        }
}

public class Movie {
        public String title; // public dla uproszczenia
        public Actor actor; // implementation of an association, cardinality "1"

        public Movie(String title, Actor actor) {
                this.title = title;
                this.actor = actor;
        }
}
```

- Cardinality 1 do *. Single reference and a container. Sample source code is below. In the `Actor` class, the situation is the same as in the previous case. By contrast, in the `Movie` class, we have a container that holds the `Actor` reference (because we need to be able to show the "*" count for many instances). Here we have used the parameterized `ArrayList` class, but it may be another container, depending on your needs.

```Java

public class Actor {
	public String name; // public for simplicity
	public Movie movie; // implementation of an association, cardinality "1"

	public Actor(String name, Movie movie) {
		this.name = name;
		this.movie = movie;
	}
}

public class Movie {
	public String title; // public for simplicity
	public ArrayList<Actor> actors; // implementation of an association,
cardinality "*"

	public Movie(String title) {
		this.title = title;
		this.actors = new ArrayList();
	}
}
```

- Cardinality * to *. Two containers (one on each side). Below is an example source code. Note that in both classes there are collections - just like in the previous example for the "*" count.

```Java

public class Actor {
	public String name; // public for simplicity
	public ArrayList<Movie> movie; // implementation of an association,
cardinality "*"

	public Actor(String name) {
		this.name = name;
	}
}

public class Movie {
	public String title; // public for simplicity
	public ArrayList<Actor> actor; // implementation of an association,
cardinality "*"

	public Movie(String title) {
		this.title = title;
	}
}
```

KNOWLEDGE CHECK — Which statements are true?

- Implementing an association with IDs provides better performance than using references.
- Implementation of cardinalities * - * enforces use of two containers (eg.ArrayList).

EXPLANATION & ANSWERS: **Explanation:**

- Using identifiers reduces performance because we have to search an extent.
- The cardinality * - * is implemented by two collections (one on each side).

Directed association

The implementation of the directed association is obviously simpler than bidirectional and probably does not require comment. Just remember that its practical usefulness is quite limited, because we have difficulty accessing information, eg `Actor` does not know in which movies was playing.

```java
public class Actor {
        public String name; // public for simplicity

        public Actor(String name) {
                this.name = name;
        }
}

public class Movie {
        public String title; // public for simplicity
        public ArrayList<Actor> actor; // implementation of an association, cardinality "*"

        public Movie(String title) {
                this.title = title;
        }
}
```

KNOWLEDGE CHECK — Which statements are true?

- In the above code, you can useVectorinstead ofArrayLis.
- In the code above, addingstaticto theactorcollection will not change the behavior of the program.

EXPLANATION & ANSWERS: **Explanation:**

- From a functional perspective, we can useArrayListorVectorcode.
- In the above code, addingstaticto theactorcollection will make it class invariant (shared by all objects).

Recursive association

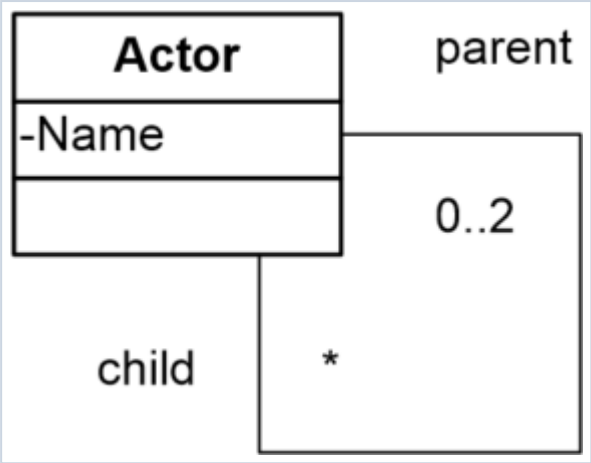


DEFINITION

Recursive association describes the connection between objects of the same class.

Because of this specificity, the role names in the case are mandatory. Otherwise we would not be able to tell which side the role plays.

The implementation is based on the same principles as the "classical" association. Let's look at the diagram and the code. We still have two containers (for * - *), but this time they are located in one class:




```java
public class Actor {
        public String name;     // public dla uproszczenia

        private ArrayList<Actor> parents = new ArrayList<Actor>();
        private ArrayList<Actor> children = new ArrayList<Actor>();

        // ...
}
```

KNOWLEDGE CHECK — Which statements are true?

- In the code above, changing the attributenametosalarywill change the way the recursive association works.
- In the above code, deleting one of the collections will not change the scope of the association information.

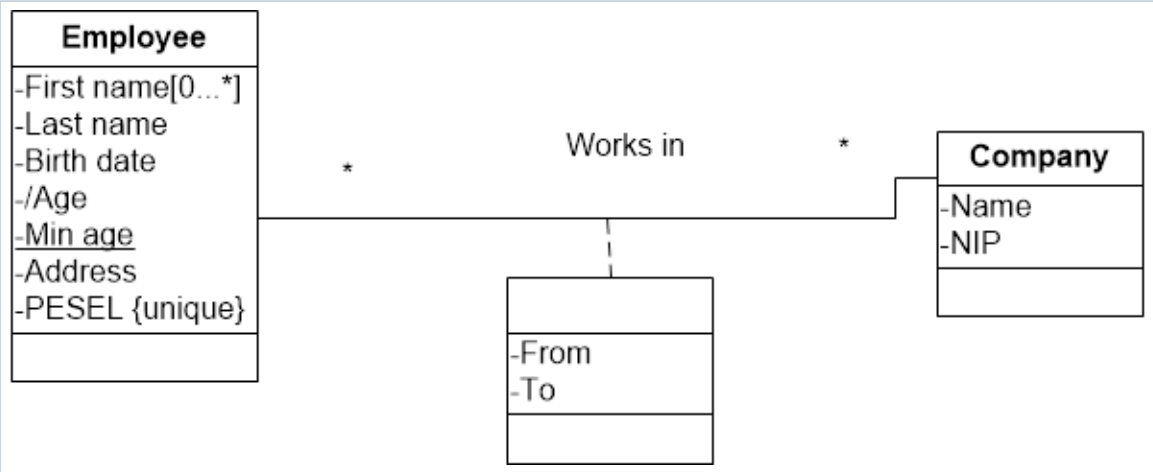
EXPLANATION & ANSWERS: **Explanation:**

- Replacing these attributes does not affect the operation of the association.
- In order to have complete information (parents and children) both collections are needed

Association with attribute

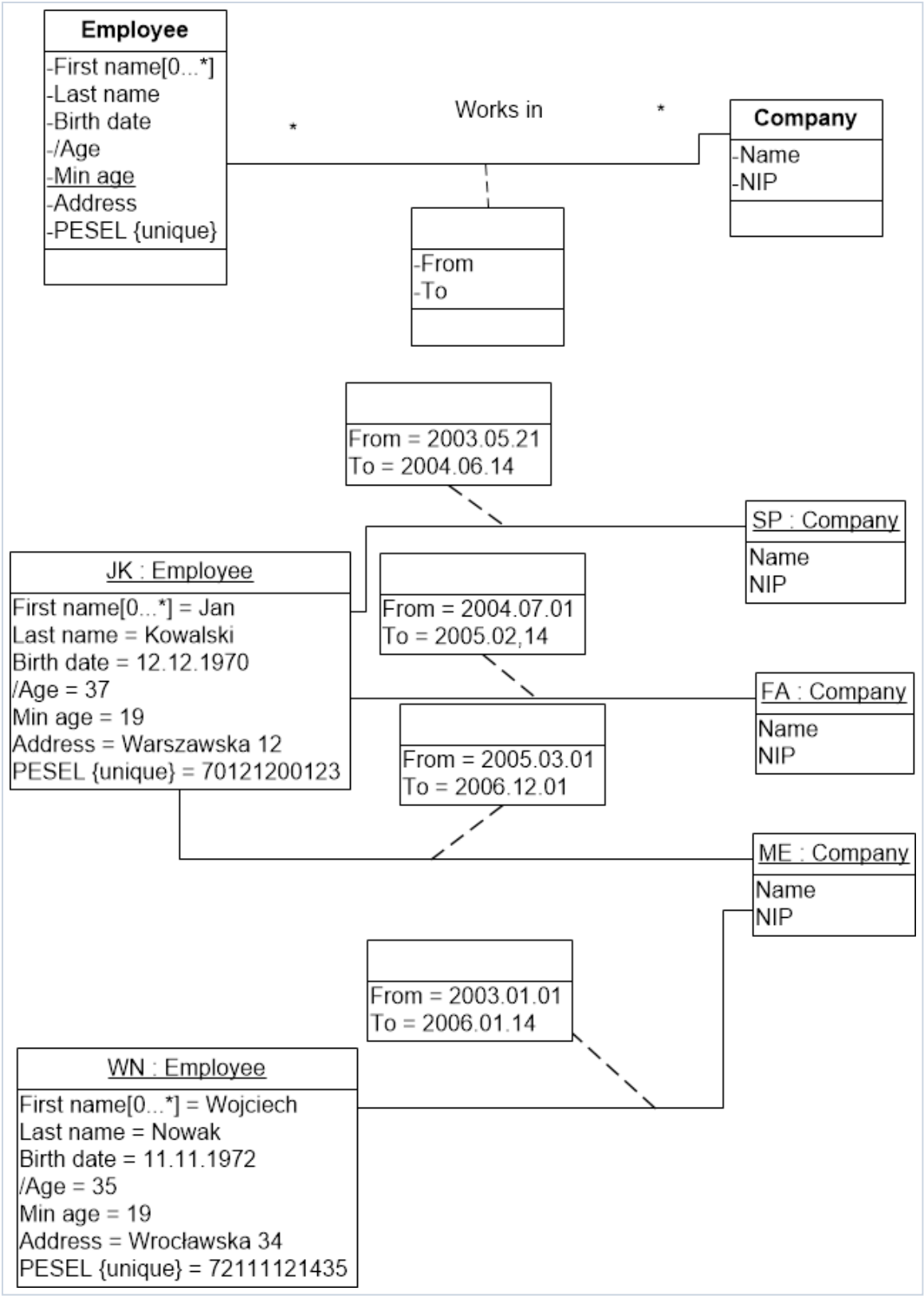
In some situations it is necessary to store additional information about a particular association/link (connection of objects). Let's suppose we want to keep the information about companies and their employees, but one employee may have several jobs. Additionally, we would like to keep the period of time spent working in individual companies. How can we do that? The simplest idea that comes to mind is to insert the appropriate attributes into one of the classes. Unfortunately this will not work because we will have many values (for different companies / employees) and connected to another class.

The best way to solve this problem is to use the association (class) attribute (see figure below). This is a normal class (has the same characteristics as any other), but "connected" to the association. Instances of an association class carry additional information about a particular link. You can think of it as a "sticky note" attached to the thread that represents the link.



To make it easier to understand the mechanism of the association class, let's look at the figure below, which contains an object diagram (not a class one). We see two Employee class instances, three Company class instances, and four instances of association classes. Based on the diagram we can say that:

- Jan Kowalski worked in three companies:
    - „Speedo” from 2003.05.21 to 2004.06.14,
    - „Fasto” from 2004.07.01 to 2005.02.14,
    - „Medio” from 2005.03.01 to 2006.12.01,
- Wojciech Nowak worked only in "Medio" company from 2003.01.01 to 2006.01.14. Let us also point out that it is the same company in which Jan Kowalski also worked. In other words, a particular object can be connected to many other objects. Of course, only when cardinalities allow - in our example we have "many to many", so everything is OK.



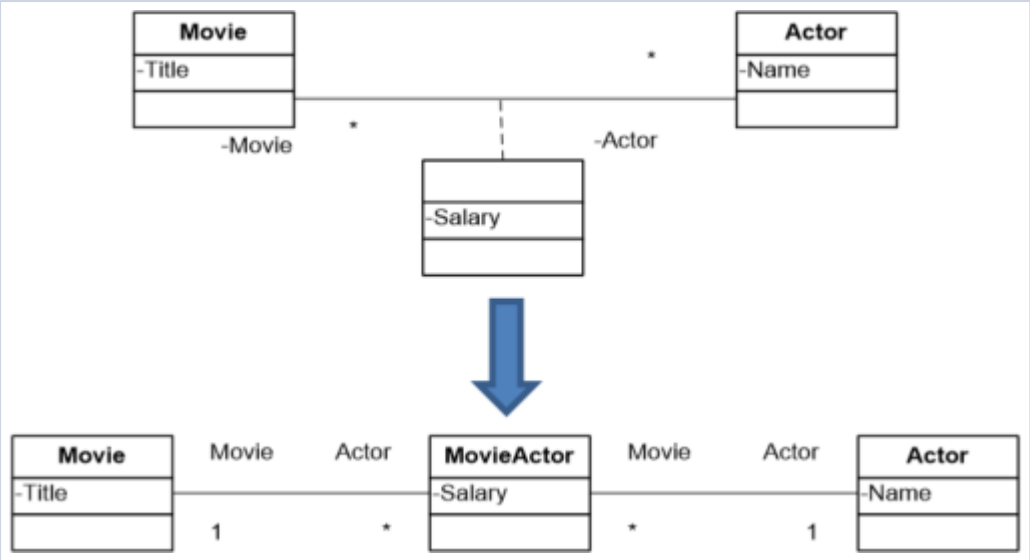
As we mentioned, the association attribute is a fully qualified class, so you can connect other associations and class-specific elements. However, it seems that this should be avoided. If this is business-justified and we want to add an association with an association class, then it's worth considering whether to make a "normal" class of it.



COMMENTARY

Let's remember that the association attribute must be used when we have the cardinality * - *. In other cases it is unnecessary (the informationcould be put in the class, on the "1" side) and we should avoid it.

Implementing the association attribute requires a prior transformation of the diagram. The association attribute is replaced by the middle class and the two associations 1-* and *-1 - as in the diagram below.



As a result, by replacing the association attribute with the intermediate class, we have received two "regular" associations. These "new" associations are implemented using one of the already known methods discussed earlier. Unfortunately, after this transformation we can expect minor problems with the semantics of this new class:

- What name to give it? If we do not come across any meaningful business name, then we can create a new as a cluster of neighboring class names. Some argue that this should always be done - so you can immediately see that it is an intermediate class;
- Another problem may be the names of "old" and "new" roles. Previously, we had two well-defined business roles. After the transformation we received two more. The good news is that we usually create new names by duplicating old ones.

The disadvantage of this approach is the difficulty of accessing the target. Previously, from the object `Movie` through the *actor* role, we got to the `Actor` instance. After the change we have to "pass" through the intermediate class object. Alternatively, you can implement some helper methods that will return the target after the call from the mediator class.

KNOWLEDGE CHECK — Which statements are true?

- One way to implement an association attribute is to add it to one of the classes.
- Introducing an intermediary class makes it difficult to access information stored in a combined class.

EXPLANATION & ANSWERS: **Explanation:**

- In the case of * - *, it is impossible to store the required information in one of the classes.
- Adding another structure between classes (the intermediate class) obviously impedes navigation and access to the associated object.

Qualified association

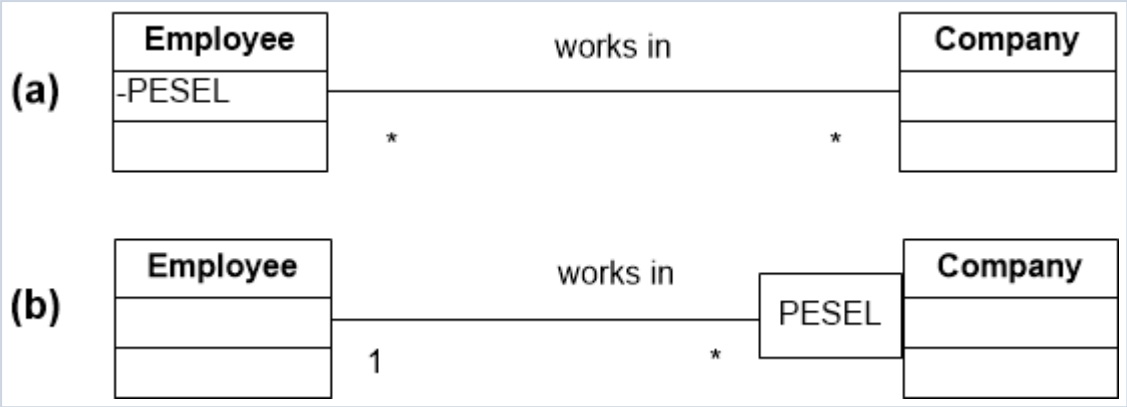
In a classic binary association, to find a particular object on the other side, we have to look through all the links until we find one that meets our criteria (eg, a PESEL employee). On contrary, in the qualified association, we can retrieve the target object based on the qualifier very quickly. Let's look at the definition:



DEFINITION

Qualifier is an attribute (or combination of attributes) that uniquely identifies the target object.

The difference between a simple association and qualified one is well illustrated by the drawing consisting of two parts:



- (a) describes the employee's relationship with the company through a classic association. In order to find an employee with a particular PESEL number, we have to look through all the employees available through the links and for each of them we check whether he/she has the required PESEL.
- (b) It also describes the employee's relationship with the company, but using a qualified association. Thanks to this solution, having a specific "qualifier" (PESEL number), we immediately get the corresponding object (employee).

The simplest implementation of a qualified association (actually its functionality) can be implemented as follows:

- we use the existing association implementation approach (no difference, if we use identifiers or references),
- we add a method, which based on the PESEL, returns an object of the `Employee` class.

Note that this approach is characterized by poor performance, because to find a specific PESEL, we need to "manually" view all the connected employees.

A better solution (and it is recommended) can be characterized as follows:

- We do not use a standard container (such as `ArrayList`),
- We use a mapping container where the key is the qualifier (eg PESEL) and the value is the reference to the target object (for example, e.g. of the Employee class).
- Returning link is stored in the "standard" way (e.g. `ArrayList`).

Example code along with diagram and explanations:



```java
public class Actor {

	// [...]

	private Map<String, Movie> moviesQualif = new TreeMap<>();

	public void addMovieQualif(Movie newMovie) {
		// Check if we already have the info
		if(!moviesQualif.containsKey(newMovie.title)) {
			moviesQualif.put(newMovie.title, newMovie);

			// Add the reverse connection
			newMovie.addActor(this);
		}
	}

	public Movie findMovieQualif(String title) throws Exception {
		// Check if we have the info
		if(!moviesQualif.containsKey(title)) {
			throw new Exception("Unable to find a movie: " + title);
		}

		return moviesQualif.get(title);
	}
}
```

- (1). A class that stores actor information.
- (5). A map that is a key element in the implementation of a qualified association. The key is the qualifier (in our case the title of the movie), and the value is a reference to the target (movie).
- (7). A method to create a connection within a qualified association. It also supports the returning link.
- (9). We are checking to see if there is already information about the movie with such a title.
- (10). If not, then we add a new entry to the map (the key is the title and the value is the reference to the movie object)
- (13). Then we create a returning link by calling the appropriate method from the Movie class.
- (17). The method that retrieves the target (movie) based on the qualifier (movie title).
- (20). If the map does not contain a given key, it means that we do not have the information. In this situation we throw an exception. As we have already mentioned, this can be handled differently, eg by returning null.
- (23). We use an already existing method from the map, which returns a value based on a key.

KNOWLEDGE CHECK — Which statements are true?

- The most efficient way to implement a qualified association is to replaceArrayListwithVector.
- One way to improve the performance of a qualified association is to replaceArrayListwithHashtable.

EXPLANATION & ANSWERS: **Explanation:**

- Search performance can be improved using a map, not a simple collection e.g.Vector.
- Therefore, after usingHashtable, we can expect better performance.

LESSON 05

Associations in OOP (Part 2)

Associations with attributes, qualified associations, composition vs. aggregation, and recursive associations.

Educational outcomes

Student:

- explains how to implement constructs such as: association and link, cardinalities and association roles, directed and recursive association, attribute association and qualifed in natural language (understanding);
- identifies and corrects errors in the source code, which is an implementation of the above structure;
- lists constructs implemented in the source code (understanding);
- implements the above structures using the selected programming language (application);
- indicates functional differences between two selected programs based on code snippets source (analysis);
- creates a program that implements specific business requirements using the constructs learned from the lectures (synthesis);

Justification

Modern programming languages are mostly based on the object paradigm. Therefore, its practical understanding is a necessary basis for working in this profession. Moreover, programmers often make systems based on analytical materials. And these largely use object-oriented graphical notations (such as UML). During this and the next lessons, you will deepen your theoretical bond, but first of all you will learn how to implement different concepts from object-orientness. This will allow you to design and implement modern business applications.

Aggregation and composition



DEFINITION

Aggregation is a special association emphasizing the relationship part - whole.

DEFINITION

Composition is a stronger form of aggregation modeling the part-to-whole relationship and additionally imposes certain constraints:

- Part can not be shared,
- Part can not exist without a whole,
- Removing the whole also means removing all parts. Removing a part does not necessarily mean deleting the whole.

The following figure shows an example of the composition between a building and its rooms.



The aggregation implementation is exactly the same as the association. This is because aggregation does not impose any specific conditions on the relationship.

The issue with the implementation of the composition is different. We have some problems to resolve:

- Block creating unconnected parts - because part can not exist without the whole,
- Prohibition of sharing parts,
- Deleting a part when deleting a whole - again: because part can not exist without the whole.

It seems that we can use at least two general approaches:

- Modification of existing solution implemented on the basis of identifiers or native references,
- Take advantage of a completely new construction, so far unannounced - see the next section.

Let's start by modifying the solution with native references. How do we block "alone" parts? Let's start by answering the following question: what to do in order to block creating an object at all? You need to make their conctructor private. Thus, it will not be available for code outside the class. Why give us a class for which objects can not be created (we can do it as abstract as well). It can not be outside, but we will add a special method - a class one (because we want to use it to create objects). It should have the following features:

- Get a reference to the whole. In addition, it must also check if the reference is valid. Someone could give, for example `null`.
- Create a part object and possibly return it. In some situations, sharing references to object-parts may not be desirable, because with such a reference, the programmer can do with the object what he/she likes (for example, combine it with another whole, and this is against the semantics of the composition). In this situation, we create methods that work on parts from the whole.
- Add returning information - to keep all association features.

Let's look at the sample code:

Java

```
public class Part {
	public String name;	// public for simplicity
	private Whole whole;

	private Part(Whole whole, String name) {
		this.name = name;
		this.whole = whole;
	}

	public static Part createPart(Whole whole, String name) throws Exception {
		if(whole == null) {
			throw new Exception("The given whole does not exist!");
		}

		// Create a new part
		Part part = new Part(whole, name);

		// Add to the whole
		whole.addPart(part);

		return part;
	}
}

public class Whole {
	private List<Part> parts = new ArrayList<>();

	private String name;

	public Whole(String name) {
		this.name = name;
	}

	public void addPart(Part part) throws Exception {
		if(!parts.contains(part)) {
			parts.add(part);
		}
	}

	@Override
	public String toString() {
		String info = "Whole: " + name + "\n";
		for(Part part : parts) {
			info += "    " + part.name + "\n";
		}

		return info;
	}
}
```

And traditionally the discussion of the most important parts:

- (1) Class describing part.
- (3) Because a part can only be connected to one entity, we use a single reference instead of a container.
- (5) A constructor. Please note that it is private. This allows only methods of the same class to create instances of that class.
- (10) Method creating a part based on the given whole.
- (11) First we check whether the whole exists. If not, then we throw an exception. Instead, you can eg return `null` or display a message.
- (16) The whole is correct, so we create a part.
- (19) We add a part to the whole. For this purpose we call a special method from the Whole class.
- (21) We return the reference to the newly created part
- (25) Class describing the whole.
- (26) A container storing links (references) to parts. Here we can not use a single reference, because there could be many parts.
- (34) A method adding a part to the whole. Checks whether a part is no longer connected to the whole. If not, it adds; If there is alsready a link, it does nothing.
- (41) A special version of the method for displaying a textual description of an object.

Another problem that we have to solve when implementing the composition is to prohibit the sharing of parts. How we do it? We will modify the method that adds a part:

- It will check if a part is no longer added somewhere,
- In addition to adding information related to a given section, it remembers (globally - that is, for the whole extent) the fact that a part is already associated with the whole. For this we will need a class attribute that stores information about all parts associated with the whole.

Modified snippets of the previous code and some comments:

```Java
public class Whole {
        private List<Part> parts = new ArrayList<>();

        private static Set<Part> allParts = new HashSet<>();

        // [...]

        public void addPart(Part part) throws Exception {
                if(!parts.contains(part)) {
                        // Check if the part has been already added to any wholes
                        if(allParts.contains(part)) {
                                throw new Exception("The part is already connected with a
whole!");
                        }

                        parts.add(part);

                        // Store on the list of all parts
                        allParts.add(part);
                }
        }
}
```

- (2) A container that holds references to object-parts for a particular object-whole.
- (4) A class attribute that is a container that holds references to all parts (connected to all object-bodies). Thanks to it, we are able to see if any part is already connected to any object-all.
- (8) Modified version of part adding method.
- (9) First we check whether the given part is no longer connected to us. If so, this method ends its operation.
- (11) If not, then we check whether the part is connected to any object of the whole. If so, we throw an exception with the appropriate message.
- (15) We add a part to our whole.
- (18) We store the connection of a given part on the global list of all parts.

The last problem to solve is to remove the part when removing the whole. Unfortunately, in languages such as Java or C#, we do not manualy delete objects from memory. As a result, we are unable to obtain the exact functionality required. For our implementation, it is useful to create a class method that removes the whole from an extent along with the associated parts (remember to remove the information from the global part list).

KNOWLEDGE CHECK — Which statements are true?

- In the code above, replacingHashSetwithVectorwill result in performance degradation.
- In the above example, thecreatePartmethod does not have to be marked with a keywordstatic.

EXPLANATION & ANSWERS: **Explanation:**

- Vectoris not optimized for quickly checking the existence of a given object.
- In the proposed implementation, thecreatePartmethod must be marked with the keywordstaticas it deals with creating new objects within the class containing it.

Composition using inner classes

Another approach to composition implementation is to use Java inner (nested) classes. This mechanism is characterized by the following features:

- An inner class object can not exist without the (enclosing) outer class object.
- The inner class object has a direct access to the invariants of the outer class object.

Because not all of the features of the composition are provided by the inner class mechanism, some of them must be implemented manualy. Let's look at the code and comments:

```Java
public class Whole {
        private String wholeName;

        private List<Part> parts = new ArrayList<>();

        public Whole(String wholeName) {
                this.wholeName = wholeName;
        }

        public Part createPart(String partName) {
                Part part = new Part(partName);
                parts.add(part);

                return part;
        }

        @Override
        public String toString() {
                return wholeName;
        }

        // The inner class (the part).
        public class Part {
                private String partName;
                // Because of Java inner class properties, we do not need a reference
pointing at the whole.

                public Part(String partName) {
                        this.partName = partName;
                }

                @Override
                public String toString() {
                        return "Part: " + partName;
                }
        }
}
```

- (4) A container that holds references to object-parts. We need to add it because the connection to this side is not provided by the inner class semantics.
- (10) A method that creates a part and returns a reference to it. It is worth noting that it is not a class one and it isinside the whole, not as before, in the part.
- (11) Physical creation of a part-object as an inner class instance.
- (12) Remember reference to it.
- (23). Inner class (note that it is actually inside the main class - see the brackets). Because each inner class instance automatically has access to its surrounding object, we do not need to create a manual linking (as we did before).

KNOWLEDGE CHECK — Which statements are true?

- In the above solution, internal classes were used to improve program performance.
- In the above example, thePartclass definition can be located outside theWholeclass.

EXPLANATION & ANSWERS: **Explanation:**

- The reason for using inner classes was their special properties (including the requirement for the existence of an object containing them) rather than performance.
- The proposed implementation uses Java inner classes, so the definition must be placed insideWhole.

Automatic return link creation

The approach proposed so far required manually adding return link information. So to connect two objects (such as an actor and a movie) you need to separately add references showing from the movie to the actor and from the actor to the movie. It is worth to automate it so that we do not have to do it manually. We will create a method that adds association information:

- in the main class,
- in the class associated with it.

The whole must be designed so that it does not create an endless loop (more on that later).

In other words, it is enough, for example, to add information about the actor to the movie, and the actor's link to the movie will be created automatically. Sample code implementing this functionality:

Java

```
public class Actor {
	public String name;	// public for simplicity

	// For standard associations
	private List<Movie> movies = new ArrayList<>(); // implementation of the
association, cardinality *

	public Actor(String name) {
		this.name = name;
	}

	public void addMovie(Movie newMovie) {
		// Check if we already have the info
		if(!movies.contains(newMovie)) {
			movies.add(newMovie);

			// Add the reverse connection
			newMovie.addActor(this);
		}
	}

	@Override
	public String toString() {
		var info = "Actor: " + name + "\n";

		// Add info about his/her movies
		for(Movie movie : movies) {
			info += "   " + movie.title + "\n";
		}

		return info;
	}
}

public class Movie {
	public String title; // public for simplicity

	private List<Actor> actors = new ArrayList<>(); // implementation of the
association, cardinality: *

	public Movie(String title) {
		this.title = title;
	}

	public void addActor(Actor newActor) {
		// Check if we have the information already
		if(!actors.contains(newActor)) {
			actors.add(newActor);

			// Add the reverse connection
			newActor.addMovie(this);
		}
```

```
        }

        @Override
        public String toString() {
                var info = "Movie: " + title + "\n";

                // Add info about titles of his/her movies
                for(Actor actor : actors) {
                        info += "   " + actor.name + "\n";
                }

                return info;
        }
}
```

Comments:

- (1) Class describing an actor.
- (5) Private container storing movie information associated with specific actors.
- (11). Method to add a movie information to specific actors. This is where "automation" is implemented, which we talked about earlier.
- (13). First, we check to see if the movie collection already contains a reference to the specified object describing the movie. In other words, we want to find out if there is already a link between this actor and the movie. Why are we doing this? Imagine that we do not check this condition. What is going on?
    - We are adding information about this movie (12),
    - Then we call the method from the object on the other side (15), which adds return info (38). Its code is almost identical to the method we are analyzing. That is, at some point, the method will be invoked from the object on the other side; from the inside of this method again will be called the method on the other side, etc. - we have endless looping, resulting in the "hang" of the program.
- As you can see, checking this condition works as a fuse - breaks the chain of reciprocal calls.
- (22). Override the method responsible for returning a textual description of an object. What's happening here? We first construct a single string containing names of actors (20). Then we add information about the movies in which he/she played. We do this by iterating through the movie collection (23) and reading their titles, which are separated by the end of the line.

And how can we use the code we just created? Here is an example, comment and the result:

```java
// Create new business objects (without connections)
var movie1 = new mt.mas.associationsref.Movie("T1");
var movie2 = new mt.mas.associationsref.Movie("T3");

var actor1 = new mt.mas.associationsref.Actor("AS");
var actor2 = new mt.mas.associationsref.Actor("MB");
var actor3 = new mt.mas.associationsref.Actor("KL");

// Add info about connections
movie1.addActor(actor1);
movie1.addActor(actor2);
movie2.addActor(actor1);
movie2.addActor(actor3);

actor1.addMovieQualif(movie1);

// Show info about movies
System.out.println(movie1);
System.out.println(movie2);

// Show info about actors
System.out.println(actor1);
System.out.println(actor2);
System.out.println(actor3);
```

- (1) We create objects that describe the movies. It is worth noting that for the time being we have no information about links with actors.
- (5) We create objects describing actors. As in the above case, we have no links yet.
- (10) We add actors to the movies. Equally good you can do the opposite: actors assign to movies. Because of automatically creating links, the effect will be exactly the same.
- (18) Show info about movies.
- (22) Show info about actors.

```
Plain Text

Movie: T1
        AS
        MB

Movie: T3
        AS
        KL

Actor: AS
        T1
        T3

Actor: MB
        T1

Actor: KL
        T3
```



COMMENTARY

Does anyone know what business case can not be realized this way? Well, we will not be able to store many links between the same objects. Why? Because when you try to add the same object, our method will notice that it is already in the collection and will not do anything.

KNOWLEDGE CHECK — Which statements are true?

- In the above solution, thecontainsmethod can be replaced with a loop that checks the contents of the collection.
- Return links make it easy to meet your business requirements.

EXPLANATION & ANSWERS: **Explanation:**

- Thecontainsmethod can be replaced with a collection loop, but better (easier and more efficiently) is to usecontainsmethod.
- Reverse link allows you to navigate two ways (from acompanyto anemployeeand vice versa), so it certainly helps you meet your business requirements.

Removing links

Deleting association information is accomplished on similar terms as the above two-way linking. Let's look at the code of the dedicated methods and discuss them:

```java
public class Actor {
	public String name;	// public for simplicity

	// [...]

	private List<Movie> movies = new ArrayList<>(); // implementation of the
association, cardinality *

	public void removeMovie(Movie movieToRemove) {
		if(movies.contains(movieToRemove)) {
			movies.remove(movieToRemove);

			// Remove the reverse link
			movieToRemove.removeActor(this);
		}
	}
}

public class Movie {
	public String title; // public for simplicity

	// [...]

	private List<Actor> actors = new ArrayList<>(); // implementation of the
association, cardinality: *

	public void removeActor(Actor actorToRemove) {
		if(actors.contains(actorToRemove)) {
			actors.remove(actorToRemove);
			// Remove the reverse link
			actorToRemove.removeMovie(this);
		}
	}
}
```

- (1). Fragment of the Actor class.
- (6). Collection keeping links.
- (8). Mthod removing the given movie.
- (9). First we check whether we even have information about the given object. If not, we do not do anything. It is also protection against endless looping.
- (10). If so, we remove it locally and call the method that will do it on the other side (13).
- Similarly, a method is used to remove actors from a movie.

Sample code is below (line 25).

```Java

// Create new business objects (without connections)
var movie1 = new mt.mas.associationsref.Movie("T1");
var movie2 = new mt.mas.associationsref.Movie("T3");

var actor1 = new mt.mas.associationsref.Actor("AS");
var actor2 = new mt.mas.associationsref.Actor("MB");
var actor3 = new mt.mas.associationsref.Actor("KL");

// Add info about connections
movie1.addActor(actor1);
movie1.addActor(actor2);
movie2.addActor(actor1);
movie2.addActor(actor3);

actor1.addMovieQualif(movie1);

// Show info about movies
System.out.println(movie1);
System.out.println(movie2);

// Show info about actors
System.out.println(actor1);
System.out.println(actor2);
System.out.println(actor3);

System.out.println("Remove info about actor: " + actor1.name);
movie1.removeActor(actor1);

System.out.println(movie1);
System.out.println(actor1);
```

Here is the effect of the use. Notice that calling the remove method causes the deletion of information from both the movie (line 20) and actors (line 23).

```
Plain Text

Movie: T1
        AS
        MB

Movie: T3
        AS
        KL

Actor: AS
        T1
        T3

Actor: MB
        T1

Actor: KL
        T3

Remove info about actor: AS
Movie: T1
        MB

Actor: AS
        T3
```

KNOWLEDGE CHECK — Which statements are true?

- In the above solution, marking the attributeArrayList actorsasstaticwill improve performance.
- In the above solution, theactorToRemove.removeMovie(this)could be replaced withactorToRemove.removeMovie(actors).

EXPLANATION & ANSWERS: **Explanation:**

- Marking theArrayList actorsattribute asstaticwill change its semantic and will result in a wrong execution of the program.
- Relacingthiswithactorswill result in a compilation error (actorsis a collection).

Managing links with the ObjectPlusPlus class

As in the case of the class extent, let's make an observation. So the way to implement association management will be (almost) the same for every business class in the system. Moreover, they will (almost) the same for each association even in the same class. In other words, many times we will have to duplicate the same code. Therefore, the question arises: can this be unified?

Similarly to the management of the extent, we will create a special class that groups functionality for association management. What's more, our new class, let's call it `ObjectPlusPlus`, will have "free" all functionality for extent, persistence, etc. How do we achieve this? ,The `ObjectPlusPlus` class will inherit from the `ObjectPlus` class.

The requirements for this new class (*ObjectPlusPlus*) can be summarized as follows:

- We will create a class that will inherit all business classes in our application.
- This new class will inherit from an existing *ObjectPlus* class. This will also make it easier for us to manage extent, persistence, and more.
- We will equip it with functionality that facilitates management:
    - ordinary binary associations,
    - qualified associations,
    - compositions (partially - condition no. 2 only).
- We will use the second of the discussed approaches for association implementation: based on references.

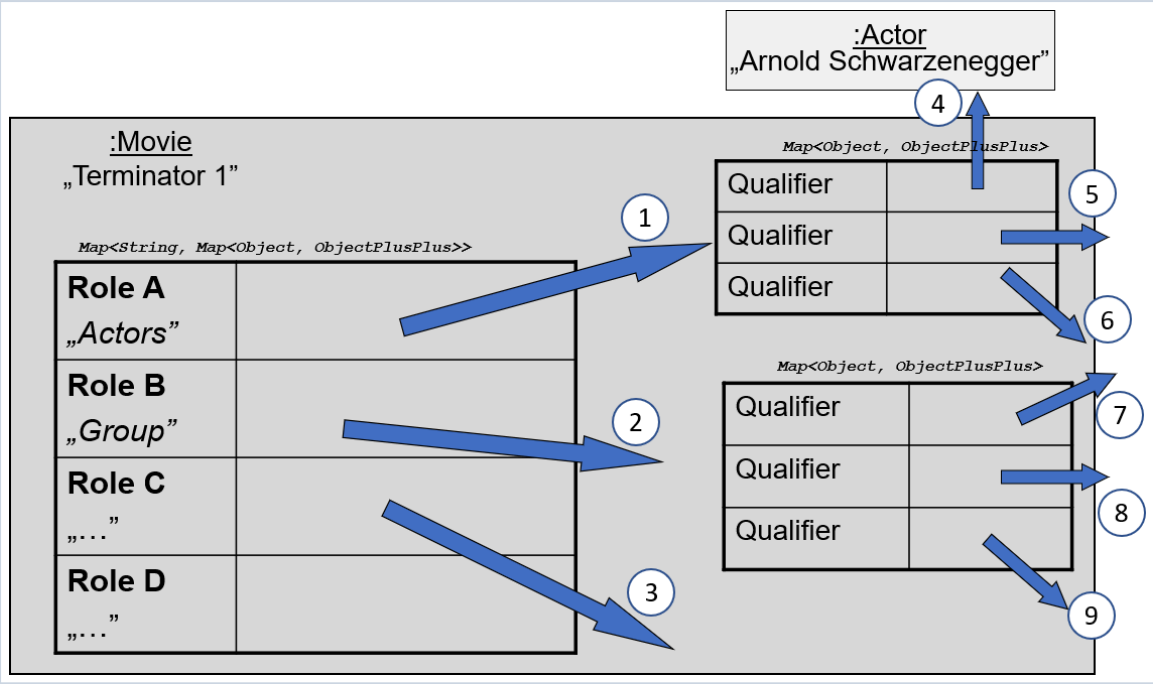
Storing links in ObjectPlusPlus

Because all associations within a single object will be stored in one collection, we can not use a simple `Vector` or `ArrayList` container. We will use a container that holds keys and values:

- The key will be the name of the role of the association. So in one object there will be as many keys as there are connected roles.
- The value will be a map storing:
    - qualifier key. When we do not want to use a qualified association, the qualifier will be the same as the target object.
    - A value that is a reference to a particular link.

The above assumption requires a brief comment. So if we did not want to implement a qualified association, then instead of the above map, we could use a regular container, eg `ArrayList`. Because we want to be able to reach the target quickly in the case of a qualified association, we use the map. Someone may blame us for duplicating the key and value in the case of storing information about classic bindings. This is true, but otherwise we would have to have separate constructs for qualified and classical associations.

The following figure should make it easier to understand implementation assumptions:



The main part of the drawing occupies a particular Movie object called "Terminator 1". Within it, we see a mapping collection (parameterized `Hashtable`) containing:

- keys with names of association roles ("Actors", "Group")
- and corresponding values in the form of references (1, 2, 3) to subsequent mapping containers (parameterized `HashMap`) - one for each role:
    - the key is the qualifier,
    - the value is a reference showing a specific target, eg for the role of "Actors", through subsequent calls (1, 4) we arrive at an instance of `Actor` named "Arnold Schwarzenegger". Further references (5, 6) lead to other actors (not shown) playing in this movie. Other references (7, 8, 9) show the target as a "Group" (describing the affiliation of a movie to some group).

Since we already have a well-conceived concept, we just have to put it into action - below is the right program and explanations.

Java

```
public abstract class ObjectPlusPlus extends ObjectPlus implements Serializable {

	/** Stores information about all connections of this object. */
	private Map<String, Map<Object, ObjectPlusPlus>> links = new Hashtable<>();

	/** Stores information about all parts connected with any objects. */
	private static Set<ObjectPlusPlus> allParts = new HashSet<>();

	public ObjectPlusPlus() {
		super();
	}

	private void addLink(String roleName, String reverseRoleName, ObjectPlusPlus
targetObject, Object qualifier, int counter) {
		Map<Object, ObjectPlusPlus> objectLinks;

		// Protection for the reverse connection
		if(counter < 1) {
			return;
		}

		// Find a collection of links for the role
		if(links.containsKey(roleName)) {
			// Get the links
			objectLinks = links.get(roleName);
		}
		else {
			// No links ==> create them
			objectLinks = new HashMap<>();
			links.put(roleName, objectLinks);
		}

		// Check if there is already the connection
		// If yes, then ignore the creation
		if(!objectLinks.containsKey(qualifier)) {
			// Add a link for the target object
			objectLinks.put(qualifier, targetObject);

			// Add the reverse connection
			targetObject.addLink(reverseRoleName, roleName, this, this, counter
- 1);
		}
	}

	public void addLink(String roleName, String reverseRoleName, ObjectPlusPlus
targetObject, Object qualifier) {
		addLink(roleName, reverseRoleName, targetObject, qualifier, 2);
	}

	public void addLink(String roleName, String reverseRoleName, ObjectPlusPlus
targetObject) {
		addLink(roleName, reverseRoleName, targetObject, targetObject);
```

```
        }

        public void addPart(String roleName, String reverseRoleName, ObjectPlusPlus
partObject) throws Exception {
                // Check if the part exist somewhere
                if(allParts.contains(partObject)) {
                        throw new Exception("The part is already connected to a whole!");
                }

                addLink(roleName, reverseRoleName, partObject);

                // Store adding the object as a part
                allParts.add(partObject);
        }

        public ObjectPlusPlus[] getLinks(String roleName) throws Exception {
                Map<Object, ObjectPlusPlus> objectLinks;

                if(!links.containsKey(roleName)) {
                        // No links for the role
                        throw new Exception("No links for the role: " + roleName);
                }

                objectLinks = links.get(roleName);

                return (ObjectPlusPlus[]) objectLinks.values().toArray(new
ObjectPlusPlus[0]);
        }

        public void showLinks(String roleName, PrintStream stream) throws Exception {
                Map<Object, ObjectPlusPlus> objectLinks;

                if(!links.containsKey(roleName)) {
                        // No links
                        throw new Exception("No links for the role: " + roleName);
                }

                objectLinks = links.get(roleName);

                Collection col = objectLinks.values();

                stream.println(this.getClass().getSimpleName() + " links, role '" +
roleName + "':");

                for(Object obj : col) {
                        stream.println("   " + obj);
                }
        }

        public ObjectPlusPlus getLinkedObject(String roleName, Object qualifier) throws
Exception {
                Map<Object, ObjectPlusPlus> objectLinks;

                if(!links.containsKey(roleName)) {
                        // No links
```

```
                        throw new Exception("No links for the role: " + roleName);
                }

                objectLinks = links.get(roleName);
                if(!objectLinks.containsKey(qualifier)) {
                        // No link for the qualifer
                        throw new Exception("No link for the qualifer: " + qualifier);
                }

                return objectLinks.get(qualifier);
        }
}
```

- (1). The `ObjectPlusPlus` class implementing links management.
- (4). The main mapping collection declared as a parameterized `Map`. The key is the name of the association role, and the value is another map (parameterized `Map`) that holds:
    - The key as a qualifier,
    - The value that is a reference to the target object.
- (7). Collection (parameterized `HashSet`) that holds references to all part objects connected to any object. This element is needed because of the implementation of condition no. 2 (part may belong only to one whole) of the composition.
- (9). A constructor. All objects that are (inheriting) from ObjectPlusPlus must call it. Similarly, in its body is the constructor call with `ObjectPlus` (including adding the object to the extent). Without this call our classes will not work properly.
- (13). The basic version of the linking method. It is worth noting that it is declared private. This is because we use it only internally. For developers using our class, there are overloaded and more friendly (simpler) versions.
- (17). The counter is used as an endless anti-loop. Previously, we used to make sure that the object was already stored in the container. Now we will introduce another solution that is probably also more efficient. You also need to use it if you want to remember multiple links (within the same role) to the same object. This works in the same way that when calling the same method that adds a link from the perspective of the target, we decrease its value. As a result, at the next call, there is a break in the method and no looping.
- (22). We are checking to see if there is a collection for the specified role name.
- (24). If so, we return it.
- (28). If not, then we create the appropriate collection (parameterized `HashMap`) and insert it into the main map (34).
- (34). If we do not have such qualifier (because they must be unique), then we add link information (37).
- (39). We call the same method, but in favor of the target to create a return link. We provide a reduced counter value so that the next call does not loop.
- (43). Public version of the method to add a qualified association. As you can see, there is no parameter for a counter anymore. These are our internal implementation issues and should not be bothered by programmers who use our library.
- (44). This method uses an internal implementation, passing the appropriate parameters.



IMPORTANT NOTE

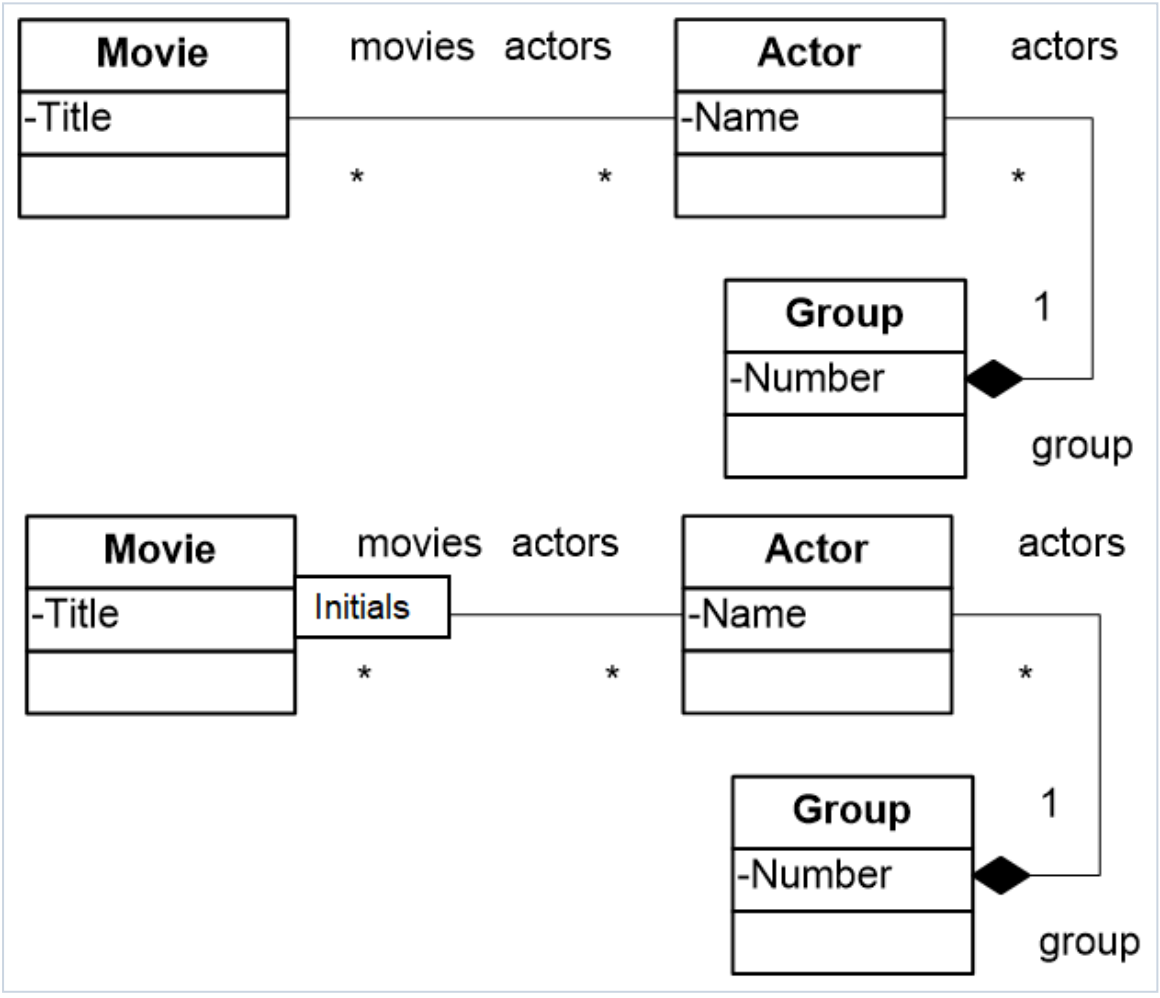
Creating one (or more) parameterized method and many of its public (overloaded) versions is a common practice. This makes the code in one place, and the programmers using our work have an easier life. It is also important for making corrections - we do it in one place.

- (47). Public version of the method to add a link without a qualifier.
- (48). This time we call the method (45), giving as the target qualifier (because we have to give something).
- (51). Method adding a part in a composition.
- (53). We are checking whether the given object-part no longer exists somewhere as a part. If so, we throw an exception.
- (57). We call the standard method of an association. So we have another example for reuse. This is even consistent with the semantics of the composition - after all, it is finally an association.
- (60). We will remember the object-part in the list of all parts (to prevent part sharing).
- (63). Method to return a link table for a given role name. Of course, instead of an array, we can return another container, but the arrays are used by many other methods.
- (66). We're checking to see if we have links for this role. If not, then we throw an exception (71).
- (71). We read the map links (based on the key).

- (73). We return an array using the method built into the map
- (76). A method that displays links for a given role on a given stream. This allows them to be displayed on, for example, the console, but also directed to, for example, a file.
- (79). We're checking to see if we have links for this role. If not, then we throw an exception.
- (84). We will retrieve the links for this role and retrieve them as values (16).
- (88). Display class name using reflection.
- (90). Iterate on all target objects by sending their descriptions to the stream.
- (95). Method returning the target based on the qualifier. Used only for qualified associations. For "classic" associations we have the `getLinks(...)` method.
- (98). We're checking to see if we have links for this role. If not, then we throw an exception.
- (103). We will retrieve a map of the links of this role.
- (104).We check if this map contains the given qualifier. If not, then we throw an exception (5).
- (109). We return the object based on the key (qualifier).

Example of use

Consider the following business case:



Let's look at what simple construction our business class is. By inheriting from `ObjectPlusPlus` we do not need to include any functionality related to extent, associations, etc.

```java
public class Actor extends ObjectPlusPlus {
        private String name;

        public Actor(String name) {
                super();        // call the super constructor
                this.name = name;
        }

        @Override
        public String toString() {
                return "Actor: " + name;
        }
}

public class Movie extends ObjectPlusPlus {
        private String title;

        public Movie(String title) {
                super();        // call the super constructor
                this.title = title;
        }

        @Override
        public String toString() {
                return "Movie: " + title;
        }
}

public class Group extends ObjectPlusPlus {
        private int number;

        public Group(int number) {
                super();        // call the super constructor
                this.number = number;
        }

        @Override
        public String toString() {
                return "Group: " + number;
        }
}
```

Using them is just as trivial and does not require any special comment:

```java
// Create new objects (no links)
Actor a1 = new Actor("Arnold Schwarzenegger");
Actor a2 = new Actor("Michael Biehn");
Actor a3 = new Actor("Kristanna Loken");

Movie f1 = new Movie("Terminator 1");
Movie f3 = new Movie("Terminator 3");

Group g1 = new Group(1);
Group g2 = new Group(2);

// Add info about links
f1.addLink("actor", "movie", a1);
// f1.addLink("actor", "movie", a2);
f1.addLink("actor", "movie", a2, "MB"); // use the qualified association
f3.addLink("actor", "movie", a1);
f3.addLink("actor", "movie", a3);

g1.addPart("part", "whole", a1);
g1.addPart("part", "whole", a2);
g2.addPart("part", "whole", a3);
// g2.addPart("part", "whole", a1); // an exception because the part already belongs to
another whole (group)


// Show infos
f1.showLinks("actor", System.out);
f3.showLinks("actor", System.out);

a1.showLinks("movie", System.out);

g1.showLinks("part", System.out);

// Test the qualified association
System.out.println(f1.getLinkedObject("actor", "MB"));
```

Effect of the above code:

```
Plain Text

Movie links, role 'actor':
        Actor: Arnold Schwarzenegger
        Actor: Michael Biehn
Movie links, role 'actor':
        Actor: Arnold Schwarzenegger
        Actor: Kristanna Loken
Actor links, role 'movie':
        Movie: Terminator 1
        Movie: Terminator 3
Group links, role 'part':
        Actor: Arnold Schwarzenegger
        Actor: Michael Biehn
Actor: Michael Biehn
```

Potential improvements

Unfortunately, our `ObjectPlusPlus` class is not ideal. Looking at the example of its use, you can see potential sources of problems.

First of all, links are identified by the regular `string`. This is bad for several reasons:

- By typing this name we can make a typo and we will not get any error message. For a computer, each name is equally good.
- We can pass an inappropriate object, eg an instance of `Actor` instead of `Movie`. And again there will be no exception.

How to deal with this? There are several possibilities (eg predefined `enum`s instead of `string`s), but to do so you have to identify the source of our problems. Well, "computer" has no additional information (metadata) to capture these errors.



IMPORTANT NOTE

Metadata is data about the data. This may be, for example, the author of the document, the date of the update. In our case, they will be associations that describe the links.

Adding them to the `ObjectPlusPlus` class would make our library able to validate the links. Using this functionality would take place in two steps:

- We first define allowed associations. As a minimum we should give the name and types of the associated classes. In addition, one can be tempted to determine the cardinalities or other more sophisticated requirements;
- Then, based on the defined association, we create links, such as the `addLink(..)` method takes as parameters the object describing the association and the business target.



AUXILIARY ASSIGNMENT

Develop an enhancement for the `ObjectPlusPlus` class to eliminate above issues. It may be based on the above idea or some other improvements.

The provided solution (along with a brief description, source code and example of utilization) can receive up to 5% of the points planned for mini-projects.

Source codes



Download all sources for MAS lectures.

LESSON 06

Inheritance in OOP

Generalization hierarchies, abstract classes, polymorphism, interface inheritance, and multi-inheritance workarounds.

Educational outcomes

Student:

- explains how to implement constructs such as: disjoint inheritance, abstract classes, overriding methods, abstract methods, overlapping, incomplete inheritance, multiinheritance in natural language (understanding);
- identifies and corrects errors in the source code, which is an implementation of the above structure;
- lists constructs implemented in the source code (understanding);
- implements the above structures using the selected programming language (application);
- indicates functional differences between two selected programs based on code snippets source (analysis);
- creates a program that implements specific business requirements using the constructs learned from the lectures (synthesis);

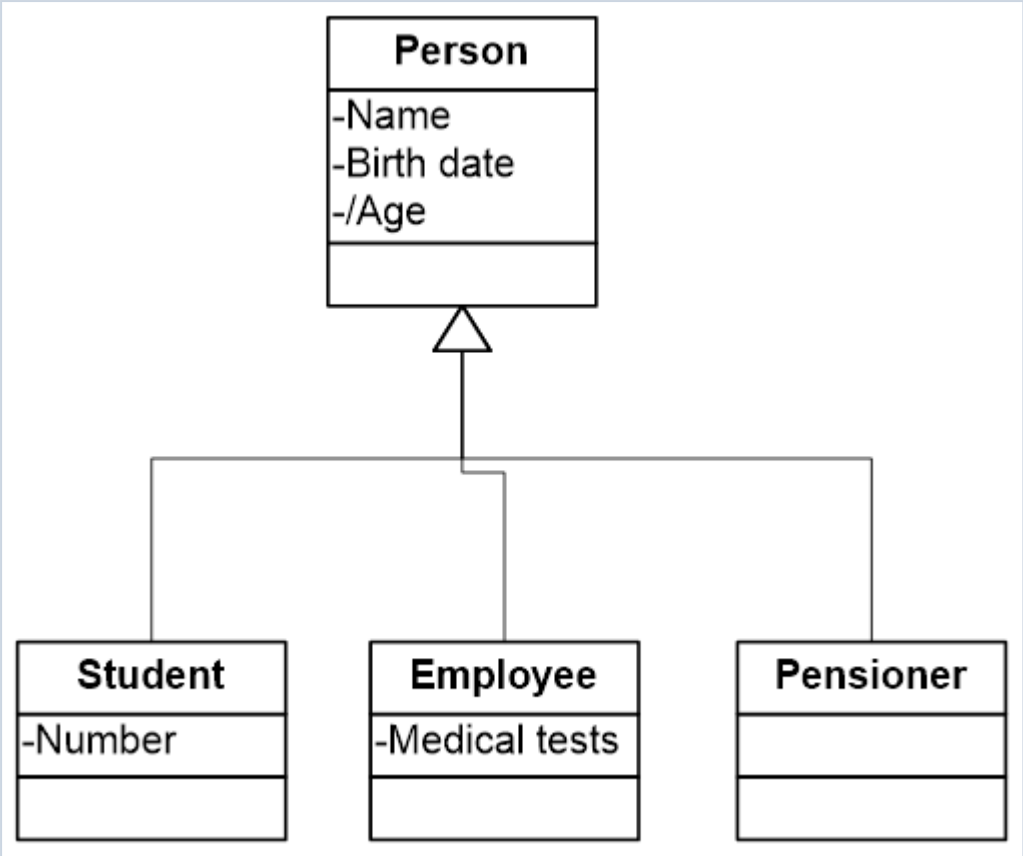
Justification

Modern programming languages are mostly based on the object paradigm. Therefore, its practical understanding is a necessary basis for working in this profession. Moreover, programmers often make systems based on analytical materials. And these largely use object-oriented graphical notations (such as UML). During this and the next lessons, you will deepen your theoretical bond, but first of all you will learn how to implement different concepts from object-orientness. This will allow you to design and implement modern business applications.

Disjoint inheritance

Inheritance is one of the most important concepts in object-orientness and implies a certain relationship between classes. This dependency allows you to create a new class (we say "subclass") based on "old" (superclass). The subclass has all the features of its superclass plus, possibly, its own. A subclass is a special case of a superclass, that is, you can inherit the class *Employee* from the *Person* class (because the employee is also a person unless we describe the robot factory). Thus, wrong examples: room from home (because room is not a special case of home) or a student from a university.

The simplest type of inheritance is called disjoint. The class instance (object) belongs to one of the classes. Let's look at the figure below:



Let's start by checking whether we were allowed to use inheritance at all: are all subclasses special cases of the superclass? Of course, yes: an employee is a person, just like a student and a retired person.

But is such a construction needed at all? After all, all these dedicated attributes can be placed in the superclass and it would also "work". Of course, it would, but:

- Depending on what kind of person we would describe, attributes that are specific to others would be "unused" (occuping memory),
- The diagram would be less descriptive of our business because it could not be stated that there is a pensioner, a student and an employee,
- We could not use the polymorphism of methods and overrides (see below).

Another doubt may be related to the fact that the *Retirement* class seems empty. It is not really "empty" - it contains important information that this person is a retired one. It is worth pointing out that in general (this is not quite true - we will go back to that), we inherit all elements in the class:

- attributes,
- methods,
- Associations - it is worth remembering especially because those who avoid the "empty" subclass (see above) sometimes forget it.

This kind of inheritance occurs directly in popular programming languages (eg Java) and can be implemented eg:

```java
// For simplicity all attributes are public (no getters/setters)
Public class Person {
        public String firstName;
        public String lastName;
        public LocalDate birthDate;
}

public class Employee extends Person {
        public boolean medicalTest;
}

public class Student extends Person {
        public int number;
}

public class Pensioner extends Person {
}
```

KNOWLEDGE CHECK — Which statements are true?

- In the diagram above, thePensionerclass can be deleted without loss for business information (it is empty).
- Associations in the superclass are not inherited.

EXPLANATION & ANSWERS: **Explanation:**

- Of course we are not allowed to remove a class from a diagram because we would lose information about one of the specializations.
- All invariants (including associations) are inherited by a subclass.

Inheritance and class extent

When discussing class extent, we mentioned that in conjunction with inheritance it raises some problems. Well, as we recall, objects in the subclass are indirectly also instances of the superclass (an *Employee* is also a *Person*). And that is exactly the problem. Since we are extending the *Person* class that contains all its objects, and the *Employee* class inherits it, should it be in the *Person* extent? Should *Person* objects be in the *Employee* extent (it seems to be the least appropriate)? The opinions of the scientists are divided. The most popular view is that *Employee* objects should be in the *Person* extent, but "cut" to a set of person properties (without additional attributes, methods, and associations). In practice, we can decide what are our business needs and choose the right solution.

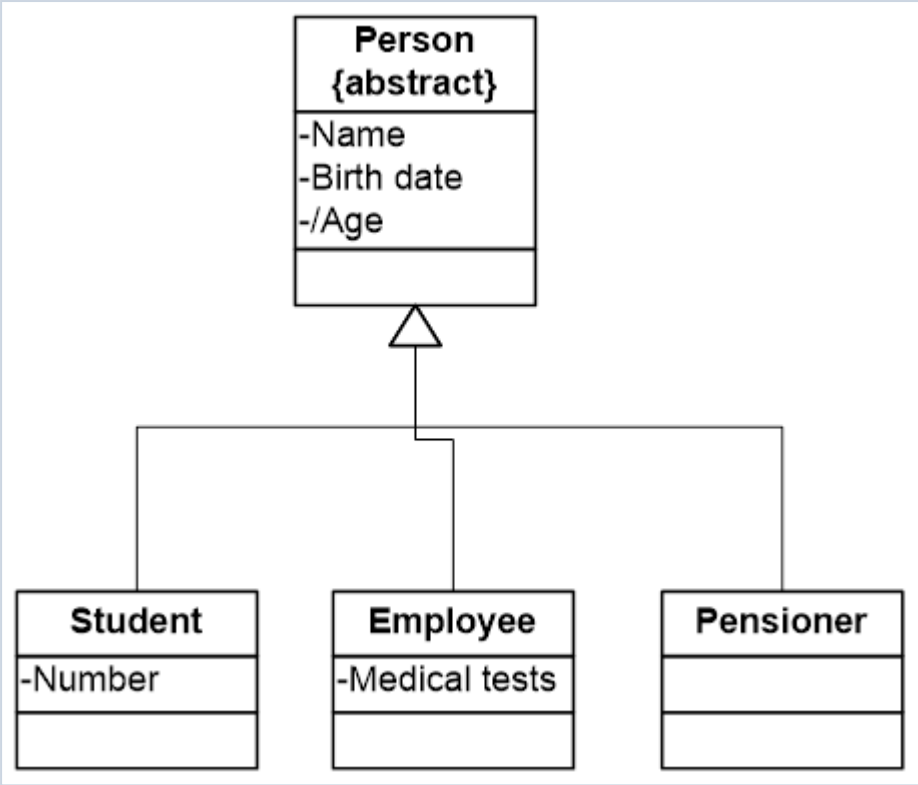
Classes and abstract methods



DEFINITION

Abstract class is a class that can not have direct instances (there can not be objects belonging to this class).

By reading the definition above you might wonder why we have a class that can not have objects? The reason is that it will be utilized to create a hierarchy of inheritance. The system described in the above example will not have a *Person* as such, but their specializations: *Student, Employee*, etc. The abstract class *Person*, is our assurance that nobody (intentionally or wrongly) will create such objects. We do this by writing the name in italics (which is not very readable) or by using the *{abstract}* label value.

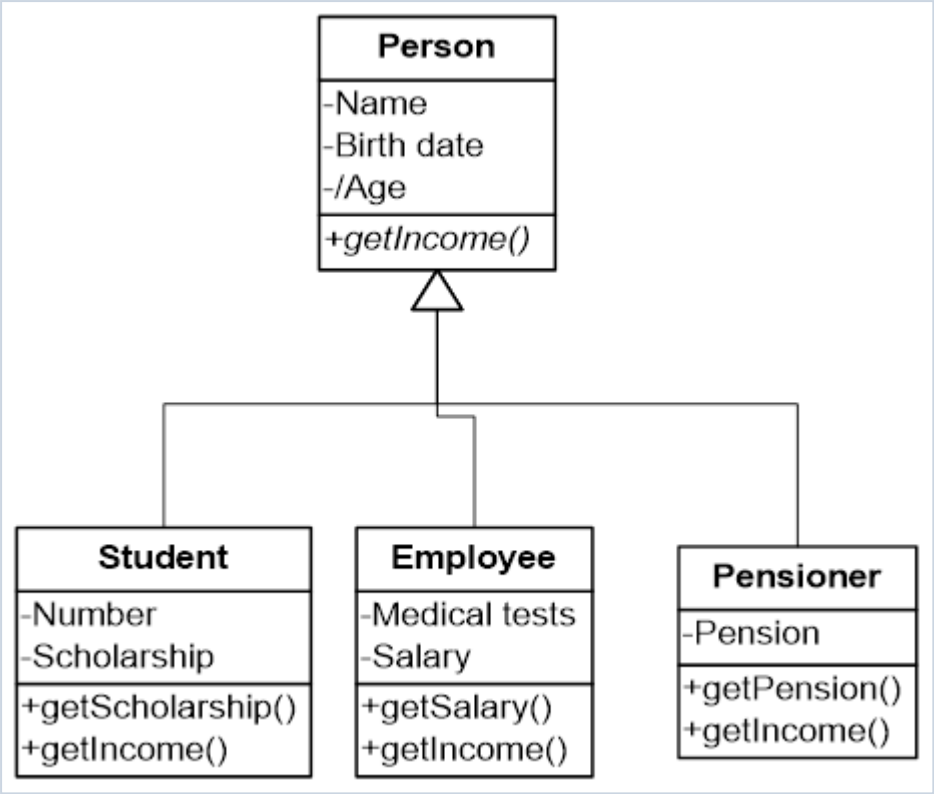

Person
{abstract}
-Name
-Birth date
-/Age
Student
-Number
Employee
-Medical tests
Pensioner



DEFINITION

Abstract method is a method that only has a declaration but has no definition (body). It can only occur in an abstract class.

Abstract class can contain concrete methods (just like any other class), but also abstract ones. As in the case of the abstract class, you might wonder why we have a method that does not have a body? We will provide them in subclasses, in different forms, depending on the specific business requirements. For example, for the above case, we may want to ask our people for their income. The problem is that the person, as such, has no income - they have specializations such as *Student* has a scholarship, *Employee* salary, etc. As a result, our diagram could look like this:


Person
-Name
-Birth date
-/Age
+getIncome()
Student
-Number
-Scholarship
+getScholarship()
+getIncome()
Employee
-Medical tests
-Salary
+getSalary()
+getIncome()
Pensioner
-Pension
+getPension()
+getIncome()

```java
public abstract class Person {

	// [...]

	private String firstName;
	private String lastName;
	private LocalDate birthDate;

	public Person(String firstName, String lastName, LocalDate birthDate) {
		this.firstName = firstName;
		this.lastName = lastName;
		this.birthDate = birthDate;
	}

	public abstract float getIncome();
}

public class Employee extends Person {

	// [...]

	private float salary;

	public float getSalary() {
		return salary;
	}

	public void setSalary(float salary) {
		this.salary = salary;
	}

	@Override
	public float getIncome() {
		return getSalary();
	}
}
```

KNOWLEDGE CHECK — Which statements are true?

- The concrete class may contain both abstract and concrete methods.
- In the abstract class there can be both abstract and concrete methods.

EXPLANATION & ANSWERS: **Explanation:**

- Specific classes may containonlyspecific classes.
- In the abstract class there can be both abstract and concrete methods.

Methods overriding

An important issue that is connected with the inheritance is the overriding and polymorphism of methods (there are several other types of polymorphism, but we will not deal with them). It allows for multiple versions of the same name (and the number and type of parameters) to be placed in each class of the same inheritance hierarchy (similar to our `getIncome()`) method. Mechanism of the polymorphic call of the method, allows you to run the proper of them.

Let's look at the code:

Java

```
public abstract class Person {

	// [...]

	private String firstName;
	private String lastName;
	private LocalDate birthDate;

	public Person(String firstName, String lastName, LocalDate birthDate) {
		this.firstName = firstName;
		this.lastName = lastName;
		this.birthDate = birthDate;
	}

	public abstract float getIncome();
}

public class Employee extends Person {

	// [...]

	private float salary;

	public float getSalary() {
		return salary;
	}

	public void setSalary(float salary) {
		this.salary = salary;
	}

	@Override
	public float getIncome() {
		return getSalary();
	}
}

public class Student extends Person {
	private int number;
	private float scholarship;

	// [...]

	@Override
	public float getIncome() {
		return getScholarship();
	}

	public float getScholarship() {
		return scholarship;
	}
```

```
        public void setScholarship(float scholarship) {
                this.scholarship = scholarship;
        }
}
```

The easiest way to understand this mechanism is by looking at how it is used:

```Java

Person p1 = new Employee("John", "Smith", LocalDate.of(1990, 12, 20), true, 4000.0f);
Person p2 = new Student("Adam", "Black", LocalDate.of(1995, 10, 18), 1212, 2000.0f);

System.out.println(p1.getIncome());
System.out.println(p2.getIncome());
```

- (1). We create `employee`. Notice that we use a reference of type `Person`, which is simply a person.
- (2). We create a `student`. Like the above, we use a reference of type `Person`.
- (4). We show the result of the call for a person's income method. But what happened? In person as such, there is only an abstract method that has no body. Now there is a polymorphic call to the method: before the virtual machine actually invokes the method, it checks what object it is dealing with (and not just what we show). If it does not find a method with the given signature in the same class, it is looking for superclasses ("going up"). This allows us to remain at a relatively high level of abstraction - we do not see who we are dealing with (student, employee or retired) - we simply ask a person for income.
- (5). The situation is similar to that in (4), but it concerns a student.

It is worth emphasizing that this mechanism is used sparsely in object languages, eg in a very popular method `toString()`.

KNOWLEDGE CHECK — Which statements are true?

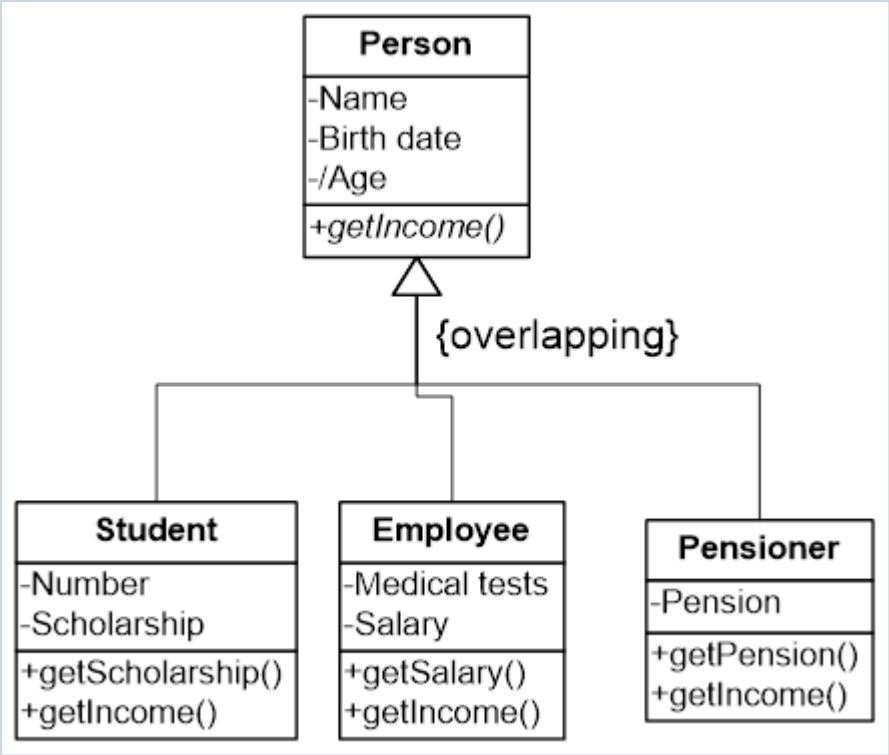
- The method override may exist without the polymorphic method call.
- Polymorphic calls to methods can exist without overriding methods.

EXPLANATION & ANSWERS: **Explanation:**

- Theoretically, the overriding may exist without a polymorphic call, but its usefulness would be rather limited.
- Conversely, the polymorphic call of the method can not exist without overriding (because there would be no alternative method to call).

Overlapping inheritance - flattening the hierarchy

Let's look at the diagram below. It looks almost the same as the previous one. The only difference is *overlapping*, meaning that we can expect objects of several classes at once in the system: in this case the object may belong to the *Employee* class and/or *Student* and/or *Pensioner*. The object can be any combination of them, which means that they will have all their features. It is worth noting that this can lead to similar problems as in multiinheritance (see below).



Unfortunately, this type of inheritance does not occur directly in any of the popular programming languages. Therefore, we need to implement it manually. Actually, we have the following ways:

- Replace the entire hierarchy of inheritance with one class (flattening),
- Use of aggregation or composition,

The first way is very simple, both conceptually and implementably. It depends on the specific grouping of all classes from the overlapping hierarchy within a single class. It usually has a superclass name. In other words:

- All invariants (attributes, methods, associations) are placed in one superclass,
- We add a discriminator (special attribute) that informs us about the object type. In our particular case we want to be able to remember several values at once. Therefore, we will use a special collection offered by Java called `EnumSet`.



IMPORTANT NOTE

In old programming languages, onstants has been used to distinguish attribute states, which has brought numerous problems, including: mixing business constants with plain numbers. For the compiler it was exactly the same. Fortunately, newer programming languages offer a special enumeration type (`enum`), which is not only safer, but also more comfortable to use.

Relevant source code:

```java
enum PersonType {Person, Employee, Student, Pensioner};

public class Person {
        private String firstName;
        private String lastName;
        private LocalDate birthDate;

        private boolean medicalTest;

        private int number;

        // We need to use EnumSet rather then PersonType because we would like
        // to have a possibility of storing combinations of the Person, e.g. Employee +
Student
        private EnumSet<PersonType> personKind =  EnumSet.of(PersonType.Person);
}
```

- (1). We have introduced an enumeration type that allows us to remember what kind of person we are dealing with.
- (3). In one class we grouped the attributes of person (4), employee (8) and student (10).
- (14). A special kind of collection acting as a discriminator. We include the appropriate enumeration types, such as `PersonKind.Employee`, `PersonKind.Student`. This configuration means that we are dealing with a working student.

How can we summarize such a solution? It certainly simpliy to implement and ease of use. At the same time, it also has some disadvantages. One of them is the inability to use inheritance constructs, such as method override, polymorphic methods calls, etc. Another one is not to use invariants belonging to another specialization - even though they occupy space. For example: if the object of our new class *Person* would describe a working student, then the other attributes that hold other attributes will not be used. And finally a considerable confusion of the class diagram. All attributes and methods will be in one class (it may become quite large), and associations that have so far come to the subclasses will be connected to that one.

KNOWLEDGE CHECK — Which statements are true?

- In the above example, the primary reason for usingEnumSetinstead ofEnumis the difference in performance.
- In inheritance overalpping object can belong to several classes at once.

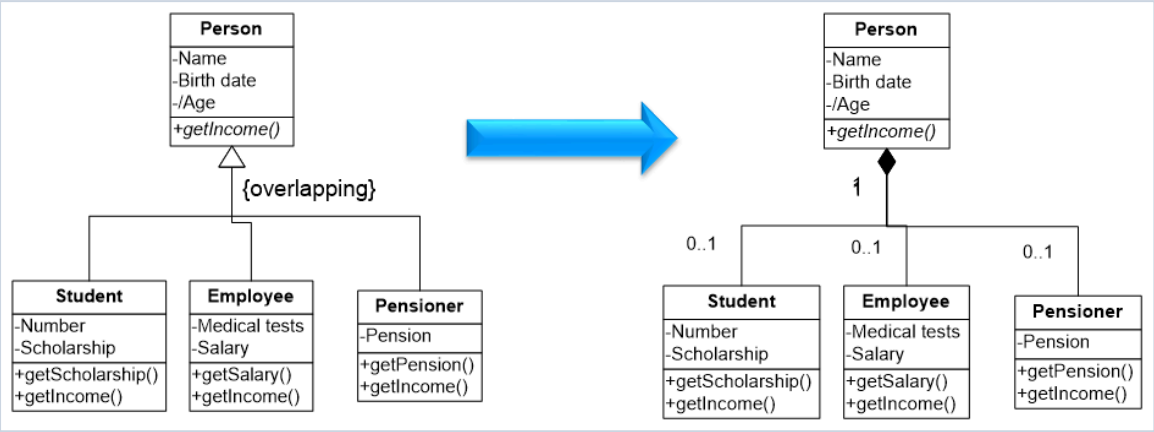
EXPLANATION & ANSWERS: **Explanation:**

- The use ofEnumSetis due to the need to store severalEnumvalues at the same time.
- As defined, theoveralppinginheritance can belong to several classes at once.

Overlapping inheritance - Composition

Another way to apply overlapping inheritance in popular programming languages is to use aggregation or composition. We usually choose composition, but in some specific business cases, aggregation may be

better. The following figure shows the transformation of our previous example into a form of inheritance overlapping.



As you can see, the matter is quite simple. Instead of inheritance we insert a composition where the superclass becomes a whole and subclasses are parts. As a result we got rid of inheritance fairly radical, because it no longer exists in any form. You also need to decide how we will treat the associations originally existing in the subclasses. The problem is that with "true" inheritance pointing to an instance of an *Employee* class gives us direct access to invariants (attributes, methods, associations) from the superclass. At the moment when the inheritance is replaced by the composition, such possibilities disappear and they have to be "simulated" somehow. We have two ways to do this after conversion. Subclass associates can show on:

- Whole. In this situation, we can add appropriate methods to make it easier for us to access the invariants in the subclasses. These will be methods that propagate the execution from class-whole to class-part. Thanks to this, the programmer who will use it will not have to navigate through the whole class.
- Part. In this case, object-parts can not be hidden. There must be direct access to them (not by object-whole), which involves a specific way of implementing the composition.

Of course, the composition can be implemented in one of the previously discussed ways. The following listing contains an example implementation of the overlapping inheritance with composition and `ObjectPlusPlus` class:

Java

```
public class Person extends ObjectPlusPlus {
	private String firstName;
	private String lastName;
	private LocalDate birthDate;

	public Person(String firstName, String lastName, LocalDate birthDate) {
		super();	// Required by the ObjectPlusPlus

		this.firstName = firstName;
		this.lastName = lastName;
		this.birthDate = birthDate;
	}

	public Person(String firstName, String lastName, LocalDate birthDate, boolean
medicalTest) {
		super();	// Required by the ObjectPlusPlus

		this.firstName = firstName;
		this.lastName = lastName;
		this.birthDate = birthDate;

		// "Changes" a person into an employee
		addEmployee(medicalTest);
	}

	private void addEmployee(boolean medicalTest) {
		// Creation of the employee part
		Employee p = new Employee(medicalTest);

		// Adding an employee as a link
		// We use a method provided by the ObjectPlusPlus
		this.addLink(roleNameEmployee, roleNameGeneralization, p);
	}

	private void addStudent(int number) throws Exception {
		// Creation of the student part
		Student s = new Student(number);

		// Adding a student as a link
		// We use a method provided by the ObjectPlusPlus
		this.addLink(roleNameStudent, roleNameGeneralization, s);
	}

	private void addPensioner() throws Exception {
		// Creation of the pensioner part
		Pensioner e = new Pensioner();

		// Adding a pensioner as a link
		// We use a method provided by the ObjectPlusPlus
		this.addLink(roleNamePensioner, roleNameGeneralization, e);
	}
```

```
        public boolean hasMedicalTest() throws Exception {
                // get an employee object
                try {
                        ObjectPlusPlus[] obj = this.getLinks(roleNameEmployee);
                        return ((Employee) obj[0]).isMedicalTest();
                } catch (Exception e) {
                        // Probably this is an exception telling that this is not an
employee
                        // (we should introduce different exception classes)
                        throw new Exception("The object is not an employee!");
                }
        }

        public int getNumber() throws Exception {
                // get a student object
                try {
                        ObjectPlusPlus[] obj = this.getLinks(roleNameStudent);
                        return ((Student) obj[0]).getNumber();
                } catch (Exception e) {
                        // Probably this is an exception telling that this is not a student
                        // (we should introduce different exception classes)
                        throw new Exception("The object is not a Student!");
                }
        }

        private static String roleNameEmployee = "specializationEmployee";
        private static String roleNamePensioner = "specializationPensioner";
        private static String roleNameStudent = "specializationStudent";
        private static String roleNameGeneralization = "generalization";
}
```

Important fragments:

- (2). Standard business characteristics specific to the person.
- (6). A constructor that creates an instance of a person class.
- (7). Since we use the `ObjectPlusPlus` class, we must call the constructor from the superclass.
- (9). Storing business data.
- (14). Constructor creating an employee.
- (22). Calling the method (25) that creates the object-part describing the *employee* and the association within the composition.
- (27). Create a part-object describing the employee.
- (31). Add it as a part to an existing whole. Uses the method provided by `ObjectPlusPlus`.
- (43). A Method analogous to (25), but creating a pensioner.
- (52). A method to check if an employee has medical tests.
- (54). The main part of the method is executed within the exception catch block.
- (55). Based on the role name we are trying to recover related objects. There really should be only one.
- (57). In case of any problems (e.g. no link, the person is not an employee), we receive an exception. In the real implementation one should distinguish different types of errors, eg by

introducing different classes of exceptions.

- (64). Method analogous to (52), but returning information specific to *student*.
- (76). Role names used to create associations within a composition.

The undoubted advantage of this method is that by breaking down functionality into different classes we use only those parts that we really need. Compared to the previous way, where in all classes were always all attributes or methods, this is definitely a useful feature. Depending on the implementation of auxiliary methods, you can also talk about the ease of use.

The main disadvantage of this method, which also occurred in the previous approach, is the lack of inheritance constructs, such as overriding methods or their polymorphic call. And of course, bigger disorder on the diagram. Someone who will look first and see compositions will not associate this with an inheritance.

KNOWLEDGE CHECK — Which statements are true?

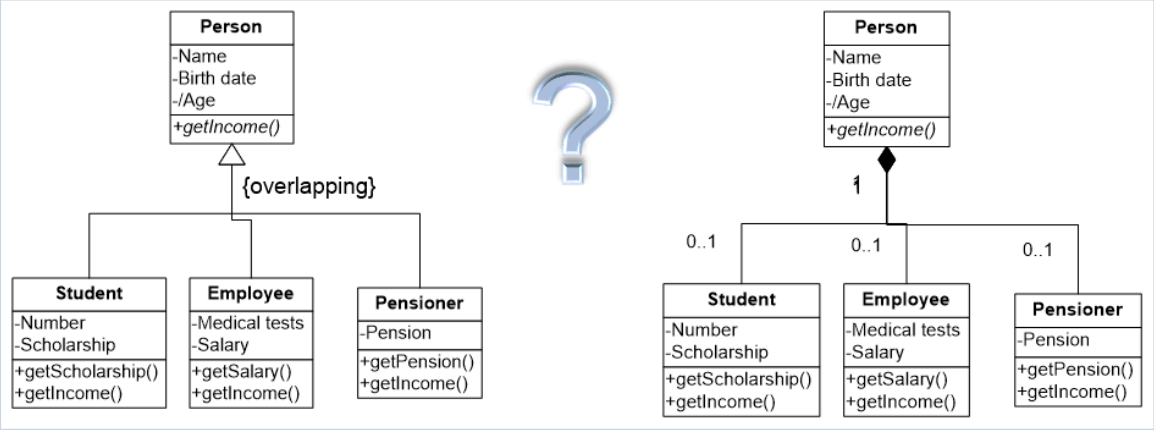
- Thanks to the use of composition instead of inheritance we have easy access to inherited invariants.
- When replacing the overlapping inheritance with a composition, we need to implement it with inner classes.

EXPLANATION & ANSWERS: **Explanation:**

- Unfortunately, the introduction of compositions made it difficult to access invariants (because they are not 'automatically' inherited).
- The composition can be implemented in any way - it does not always mean using inner classes.

Polymorphism in overlapping inheritance

Mechanism of overriding methods and their polymorphic call seems very useful. Unfortunately, for more complex hierarchies (eg, inheritance), there is a problem with choosing the right method. It consists in that, depending on the subclass, we have different definitions of the same operation. At the same time our object belongs to several classes at once, so theoretically also has access to several versions of methods - which one to choose? It is difficult to imagine an algorithm that does this automatically for all business cases.



Which version of the `getIncome()` method should be called to calculate the income of the employee and *employee* and *student*? Probably none of them... The solution is to create a new method which, depending on the types of objects, will take into account the corresponding incomes. Sample code is pretty simple:

```java
public float getIncome() throws Exception {
        float income = 0.0f;

        if(this.anyLink(roleNameEmployee)) {
                // Employee
                ObjectPlusPlus[] obj = this.getLinks(roleNameEmployee);

                // add employee's income
                income += ((Employee) obj[0]).getIncome();
        }

        if(this.anyLink(roleNameStudent)) {
                // Student
                ObjectPlusPlus[] obj = this.getLinks(roleNameStudent);

                // add Student's income
                income += ((Student) obj[0]).getIncome();
        }

        if(this.anyLink(roleNamePensioner)) {
                // Pensioner
                ObjectPlusPlus[] obj = this.getLinks(roleNamePensioner);

                // add pensioner's income
                income += ((Pensioner) obj[0]).getIncome();
        }

        return income;
}
```

KNOWLEDGE CHECK — Which statements are true?

- In the code above we can replace theObjectPlusclass with our implementation.
- The above implementation can be more efficiently implemented by overriding methods.

EXPLANATION & ANSWERS: **Explanation:**

- Obviously, usingObjectPlusis not mandatory.
- Unfortunately we can not use the overriding because we do not use 'real' inheritance.

Complete and incomplete inheritance

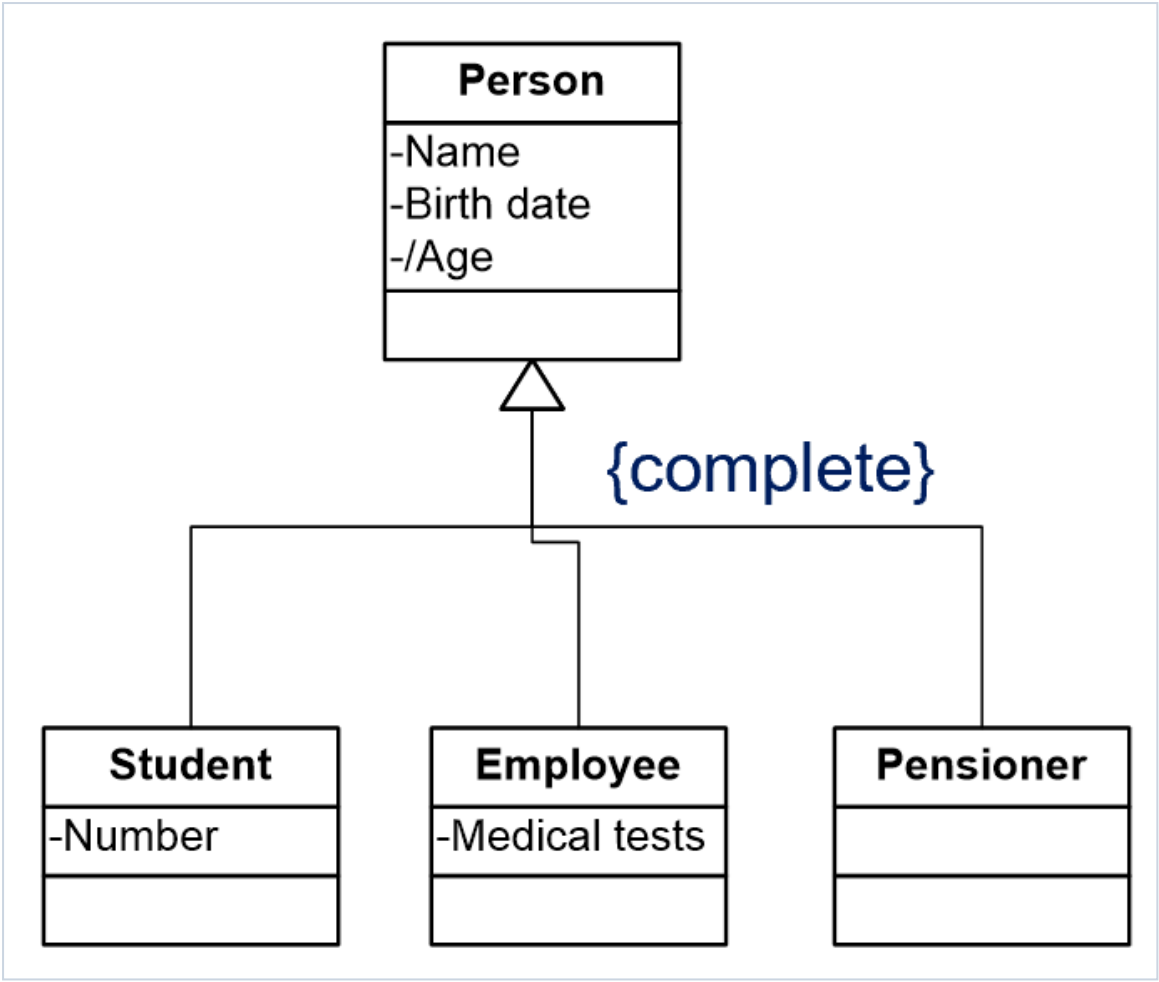
Incomplete inheritance means that we are aware that we will have more subclasses but we do not know them yet. A similar construction is an ellipse (placed as "..." instead of a subclass) that says we know additional subclasses, but for some reason (eg, readability) they are not presented.

Let's consider - what this kind of inheritance means for implementation? First of all, we need to ensure that we can inherit, for example, we should not use the word `final`, which does not allow Java to inherit from a class declared with that expression.



IMPORTANT NOTE

Let's keep in mind that we usually have several ways to meet a business requirements. Knowing that in the future we can have more subclasses, it is worth to try a solution, which may now make us a little more trouble, but later will significantly facilitate implementation.



And the issue of using the `final` keyword to implement the *{Complete}* constraint in a Java program. It blocks the creation of subclasses, e.g.

- `final class Person { ... }` → no subclasses for the `Person` class,
- `final class Employee { ... }` → no subclasses for the `Employee` class.

In the diagram above, the *{Complete}* constraint means no more `Person` subclasses (beyond the existing ones), so using `final` in our code won't help us.

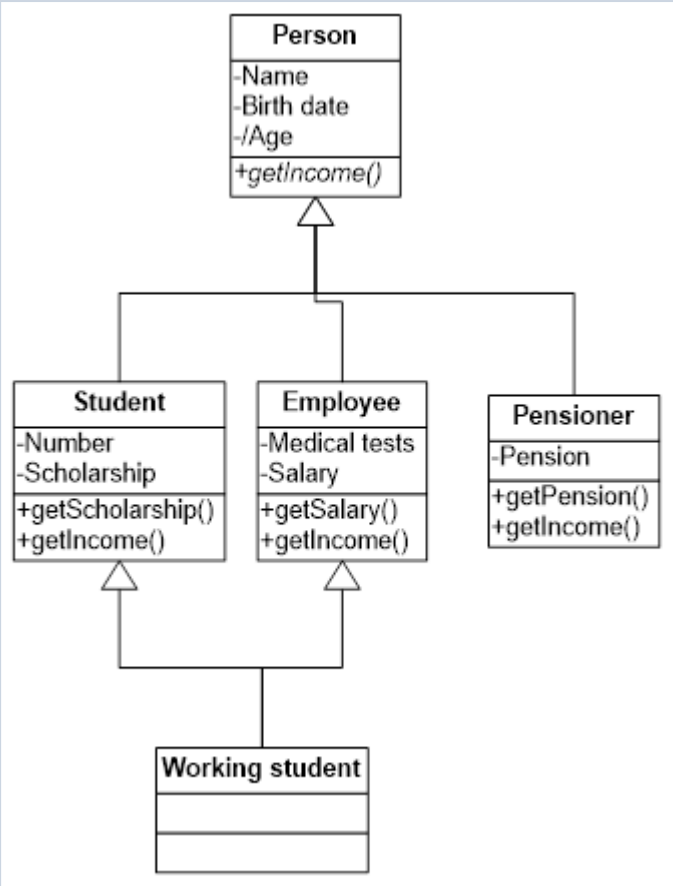
Multi-inheritance - flattening or composition



DEFINITION

Multi-inheritance (multiple inheritance) occurs when a subclass has more than one superclass.

Let's look at the above definition. This one, seemingly minor change, causes a lot of trouble. Let's analyze the following example:



The main change is to add a class inheriting from *Employee* and *Student* (*Working Student*). First let's see if we can do that. Is the subclass a special case of both superclasses? Naturally, yes: a working student is both an employee and a student. Since we have already tested this, let's go back to the problem we mentioned earlier. In accordance with the rules of superclass inheritance we get all invariants (constants such as attributes, methods, associations). Both classes have the same attributes (inherited from class *Person*): name, lastName, etc. In this case, class *Employed student* inherits two sets of attributes. We do not want to have two last names in the class, two first names, etc. The problem with attributes can be solved somehow, thinking this way: what does it mean that we inherit the attribute? This means the ability to store a certain value, that is, if the class already has that ability (eg attribute name), then placing the same attribute one more time could be ignored(because they are all the same). More serious problems occur with methods, because they can vary (and most likely they are). In our example, there are two versions of `getIncome()` (derived from *student* and *employee*) - which one to choose? In business terms, none is appropriate. We should create another version that sums up the income from the "employee" part and the "student" part.

What is the solution to the above problem with duplication of invariants? Unfortunately, there is no one good way - we usually have to determine exactly which element we are dealing with, or manually eliminate the conflict.

How does all this have to do with programming languages? Multi-inheritance

- occurs directly in C++. In the case of a name conflict that we wrote before, we use the scope operator. This allows the compiler to know which invariant to use;
- does not exist in Java or MS C#.

How can we implement them? Traditionally, there are several ways to deal with this problem:

- We can use the approaches we used to overlapping inheritance: we leave one hierarchy of inheritance, and the rest are grouped in superclasses or implemented by composition or aggregation. The basic question that arises is about the selection criteria: which hierarchy of inheritance to leave and which to replace? The answer is quite intuitive: the branch, where we use more inheritance-related constructions (overriding, polymorphic calls, etc.) leave intact, but remaining branches should be "converted".
- The new way we have not used so far is to utilize interfaces available in both Java and MS C# - see below.

KNOWLEDGE CHECK — Which statements are true?

- In Java, the inheritance hierarchy from the above figure can be implemented directly.
- For the above inheritance hierarchy, the Java virtual machine always invokes thegetIncome()method from thePersonclass.

EXPLANATION & ANSWERS: **Explanation:**

- Unfortunately, this is not possible due to the multi inheritance.
- Unfortunately, the hierarchy in the drawing is not directly executable in Java.

Multi-inheritance – interfaces

Let's start by reminding what interfaces are and what is they purpose. The interface contains a list of method headers (without their bodies or definitions). Facilitate the use of objects (method calls) without being interested in their "class membership". Implementing an interface by a class simply means delivering bodies to all the methods it defines. A class can implement any number of interfaces. By contrast, because of their limitations, we have no way of including attributes in them, so we use only methods. Part of this problem can be solved using getters and setters, that is, methods with the prefix get/set, which are used to retrieve attribute values and changing its value.



IMPORTANT NOTE

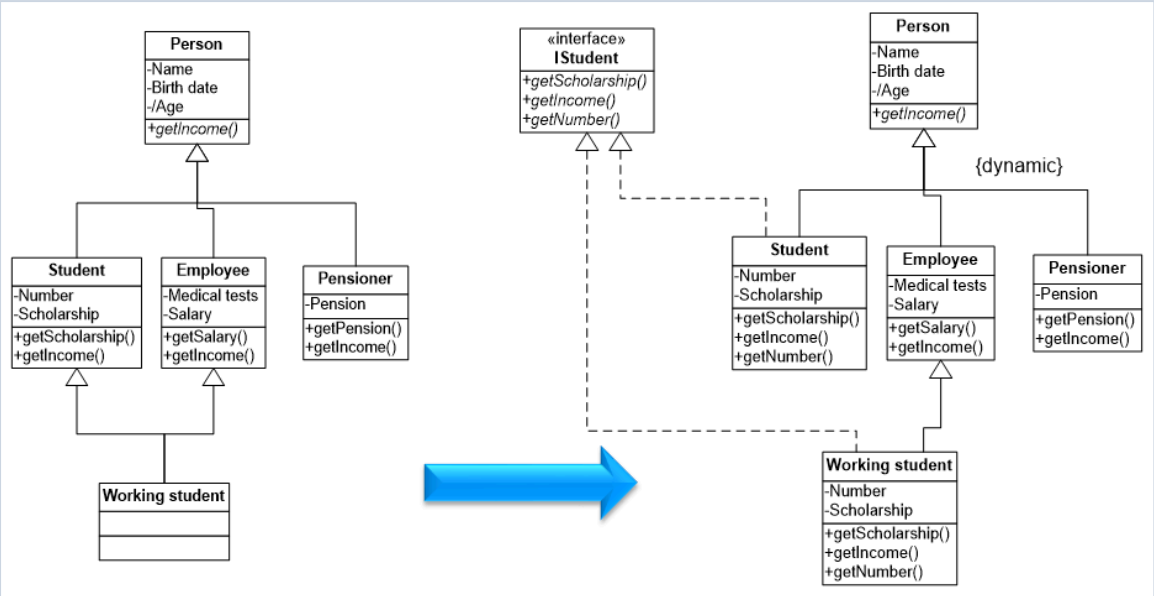
Theoretically, you can place attributes in the interface, but by definition they become `public` and `final`. So they really serve as constants, eg to diversify the kind of an element.

Unfortunately, a serious drawback of using interfaces is the need to implement the same methods repeatedly and in the same way. This is the case when in different classes we have, for example, the `getName()` method, which simply returns its value. Part of this problem is solved by modern IDEs that can automatically generate the appropriate sets of setters and getters for the specified attributes. After all, this is only a partial solution because:

- it relates to specific types of methods and does not help in "normal" methods,
- in case of modifications, we need to manually update each implementation

Another approach to solving the problem of multiple implementation methods can be a solution that use design patterns, such as delegations or commands. In short, by implementing the same method in different classes, instead of directly duplicating the code in its body, we delegate its execution (from within it in many classes) to some (the same) object.

Let's take a look at the following figure illustrating the conversion of interface (in UML interfaces are similar to inheritance, but with dashed lines).



After the conversion, the *Employed Student* class inherits from the *Employee* class and implements the `IStudent` interface. This interface is also implemented by the *Student* class. There is also a situation we have described previously. The `getScholarship()` and `getNumber()` methods will probably have identical bodies in both implementations. `getIncome()` is no longer available - for a student it will simply return the scholarship value, but the employee will still have his paycheck.

The appropriate code implementing the `WorkingStudent` class is shown below

Java

```
public interface IStudent {
	public abstract float getIncome();
	public abstract float getScholarship();
	public abstract void setScholarship(float scholarship);
	public abstract int getNumber();
}

public class WorkingStudent extends Employee implements IStudent {
	private int number;
	private float scholarship;

	public WorkingStudent(String firstName, String lastName, LocalDate birthDate, int
number, float scholarship, boolean medicalTest, float salary) {
		super(firstName, lastName, birthDate, medicalTest, salary);

		this.number = number;
		this.scholarship = scholarship;
	}

	@Override
	public float getScholarship() {
		return scholarship;
	}

	@Override
	public void setScholarship(float scholarship) {
		this.scholarship = scholarship;
	}

	@Override
	public float getIncome() {
		return super.getIncome() + getScholarship();
	}

	@Override
	public int getNumber() {
		return number;
	}

	@Override
	public String toString() {
		return String.format("Working student: %s %s, scholarship: %s, salary: %s",
getFirstName(), getLastName(), getScholarship(), getSalary());
	}
}
```



IMPORTANT NOTE

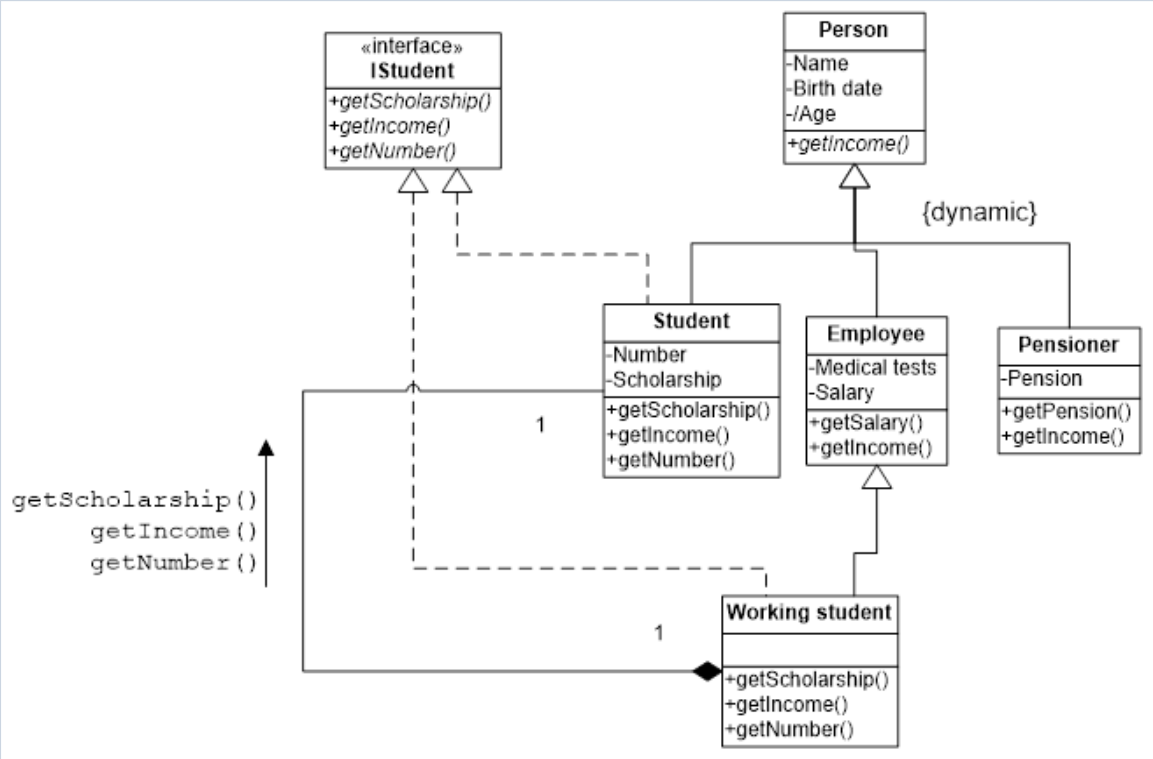
In some programming languages, or rather some programmers, begin interface names with capital letter "I", eg `IStudent`. This allows you to see at a glance what we are dealing with and there is no name conflict problem (if we wanted to have a class and interface named Student). Java is a bit different: interfaces are called as the ability to perform certain operations, eg `Serializable`.

Notes for implementation:

- (1). The `IStudent` interface declaration. As we can see, it only contains method headers. Their bodies will be added in the classes that implement it.
- (8). The `WorkingStudent` class inherits from `Employee` and implements `IStudent` (`implements`).
- (12). Class constructor retrieving the relevant business parameters.
- (13). Call the superclass constructor.
- (19). Implementation of the interface method. Probably identical to that in the `Student` class.

- (23). Implementing a method from an interface, but in a different version than the `Student` class.

We mentioned the problem of multiple functionality duplication in classes implementing the same interface. Can it be fixed? Of course - we already wrote about it. Partial solution to the problem of multiple implementation of the same methods may look like the following (diagram, code and comments):



```Java
public class WorkingStudent extends Employee implements IStudent {
        Student student;

        public WorkingStudent(String firstName, String lastName, LocalDate birthDate, int
number, float scholarship, boolean medicalTest, float salary) {
                super(firstName, lastName, birthDate, medicalTest, salary);

                student = new Student(null, null, null, number, scholarship);   // pass
nulls to prevent data redundancy (firstName, etc.)
        }

        @Override
        public float getIncome() {
                return super.getIncome() + getScholarship();
        }

        @Override
        public int getNumber() {
                return student.getNumber();
        }

        @Override
        public float getScholarship() {
                return student.getScholarship();
        }

        @Override
        public void setScholarship(float scholarship) {
                student.setScholarship(scholarship);
        }
}
```

- (2). A reference to the object to which we will redirect the student function calls.
- (4). Constructor.
- (5). Call the superclass constructor.
- (7). Create a student object that will be used to provide "student" functionality.
- (10). An interface method. As you can see, we call it from the student object and return the result.
- (18). An interface method that returns an income. First, we call the superclass method (to give us the employee's income), then we add the scholarship.

Quite a concern may be remembering some of the attributes twice (eg first name, last name): once in class `WorkingStudent` and the second time in the attached `Student` class. How can we remedy this? The answers are actually two: modifying the `Student` class (and/or inheritance hierarchy), or passing some special value to the `Student` object, eg `null`.

KNOWLEDGE CHECK — Which statements are true?

- In Java, interfaces can directly serve to define object attributes, firstName.
- The operation propagation causes the business code of the method to be in a single place.

EXPLANATION & ANSWERS: **Explanation:**

- Interfaces can not contain attribute declarations (such as in classes).
- Thanks to the propagation of the operation, the business code is located in one method, and the rest of the code only calls it.

Source codes



Download all sources for MAS lectures.

LESSON 07

Inheritance in OOP (Part 2)

Overlapping inheritance, dynamic inheritance, multi-aspect inheritance, and structural design patterns.

Educational outcomes

Student:

- explains how to implement constructs such as: disjoint inheritance, abstract classes, overriding methods, abstract methods, overlapping, incomplete inheritance, multiinheritance in natural language (understanding);
- identifies and corrects errors in the source code, which is an implementation of the above structure;
- lists constructs implemented in the source code (understanding);
- implements the above structures using the selected programming language (application);
- indicates functional differences between two selected programs based on code snippets source (analysis);
- creates a program that implements specific business requirements using the constructs learned from the lectures (synthesis);

Justification

Modern programming languages are mostly based on the object paradigm. Therefore, its practical understanding is a necessary basis for working in this profession. Moreover, programmers often make systems based on analytical materials. And these largely use object-oriented graphical notations (such as UML). During this and the next lessons, you will deepen your theoretical bond, but first of all you will learn how to implement different concepts from object-orientness. This will allow you to design and implement modern business applications.

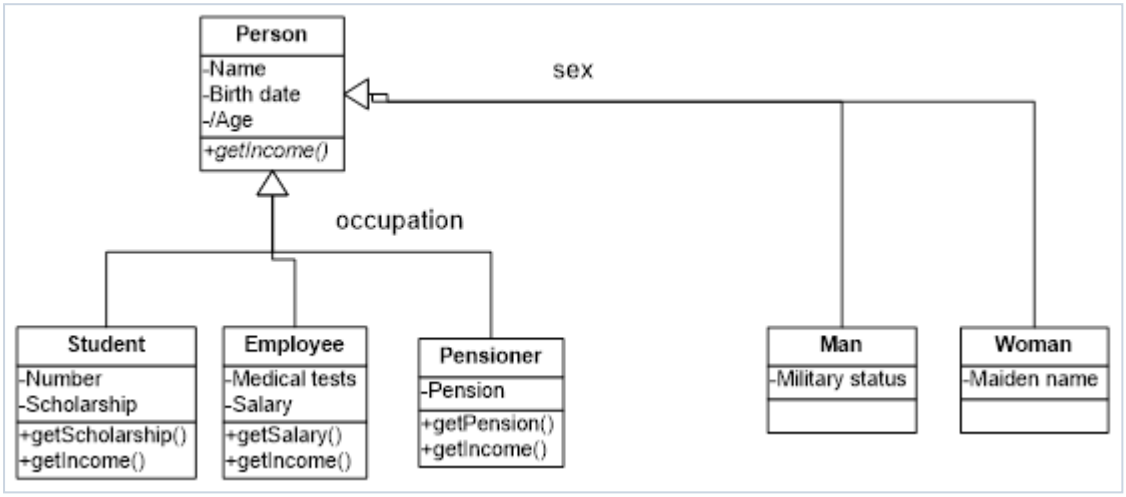
Multi-aspect inheritance

The types of inheritance discussed so far covered only one aspect (division criterion). In most cases this is sufficient, but not always. Let's go back to the previous example (specializations *Persons: Employee, Student, Pensioner*). Suppose we would like to include the gender of the people mentioned there. And depending on it, store some additional information and perform specific operations. For example, for women, this will be the maiden name and birth of the child, and for men the attitude to military service and going to the army. Let's see how we can accomplish this:

- In the Person class put appropriate attributes (and for men and women) and methods. If the object will describe a man, then the "feminine" features will not be used, and vice versa. As easy to guess, this is not the best way.

- Modify existing subclasses and add new ones, including gender: "Female Worker", "Male Worker", "Female Student", "Student Male", etc. And again tt's probably not the best idea: duplicate attributes and methods repeatedly. For our three classes and two genders this does not lead to a large number of combinations, but if we needed to divide the three categories, each of which has three options, then the number is going to be substantial.

As you can easily guess, we will be assisted by multi-aspect inheritance. Let's look at the figure below:



We have added two classes: *Woman* and *Man*. They contain the appropriate attributes and methods. The aspect names are placed at the inheritance sign (they are mandatory). Seeing such a diagram, we can expect objects belonging to the class, such as *Female Employee*, *Male Pensioner*, etc. The principle is simple: we take one class from every aspect (unless there is, for example, *overlapping*, then we can use any combination of classes from this aspect) and stick them together.

Unfortunately, this kind of generalization does not occur in any of the popular programming languages. Therefore, we need to implement them somehow, using the following suggestions:

- One aspect is inherited using built-in (simple) inheritance mechanisms for a particular programming language.
- Other aspects are implemented using one of the previously discussed ways.
- We can also try to remove some of the hierarchy or, in other words, group in one class, eg by adding flags to the main class. So we can do it when we do not need to store different information depending on the value of this flag. In other words, this flag is simply a kind of discriminator.

The question immediately arises: what aspect should we inherit, and which one model in another way? As in the case of multiple inheritance, we leave intact this aspect which:

- There is a method override and polymorphic call,
- There is a greater diversity of attributes in each subclass. In other words - the most complicated/advanced hierarchy.

In some situations when we do not store information that is specific to a particular aspect, and only the object type information, we can replace the inheritance with, for example, a flag located in the superclass. In such a situation, it may be better to place one or two attributes in the superclass, even if they are not always used, rather than complicate the model with eg a composition.

KNOWLEDGE CHECK — Which statements are true?

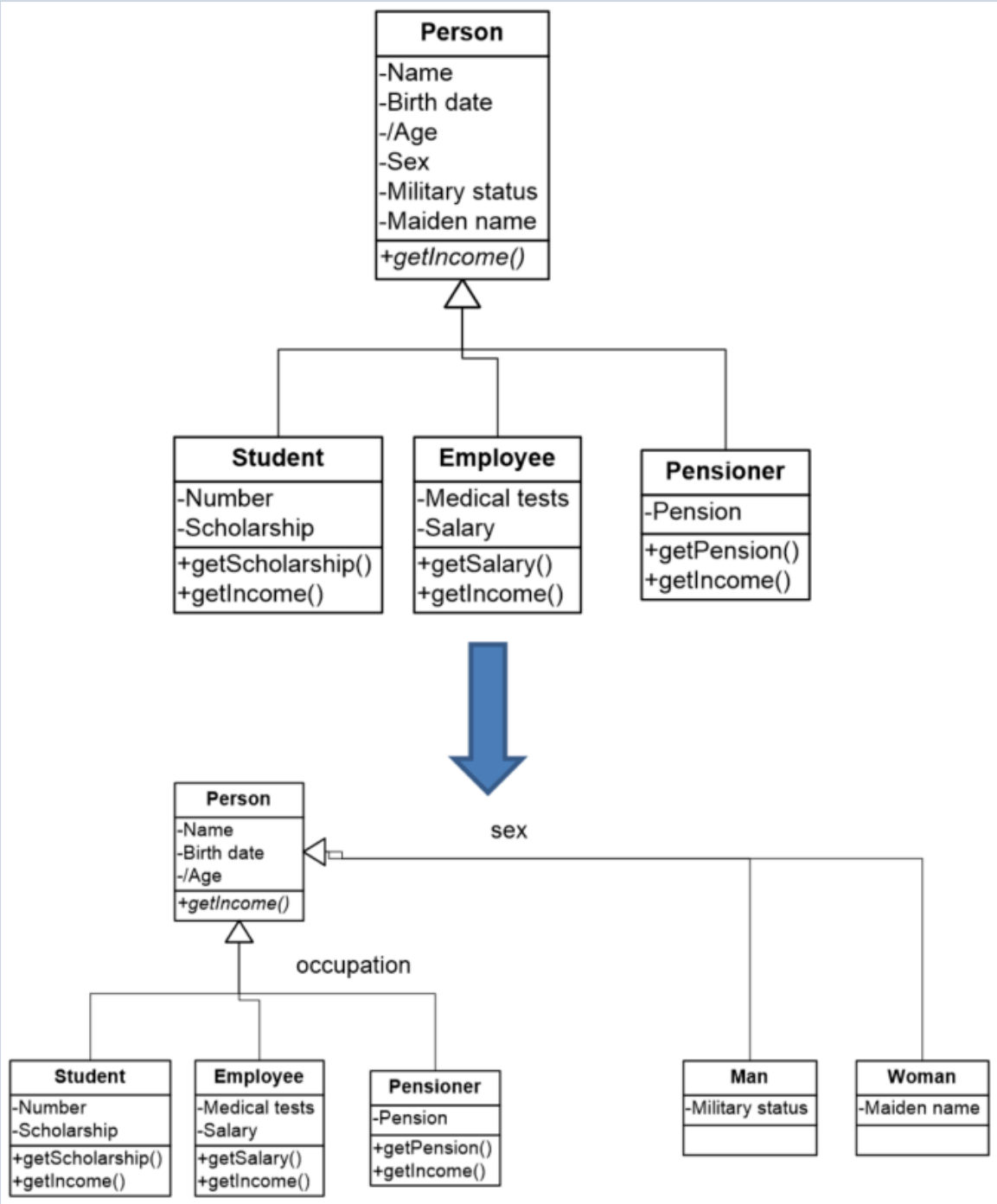
- Multi-aspect inheritance is the same as multi-inheritance.
- Multi-aspect inheritance occurs in all popular programming languages.

EXPLANATION & ANSWERS: **Explanation:**

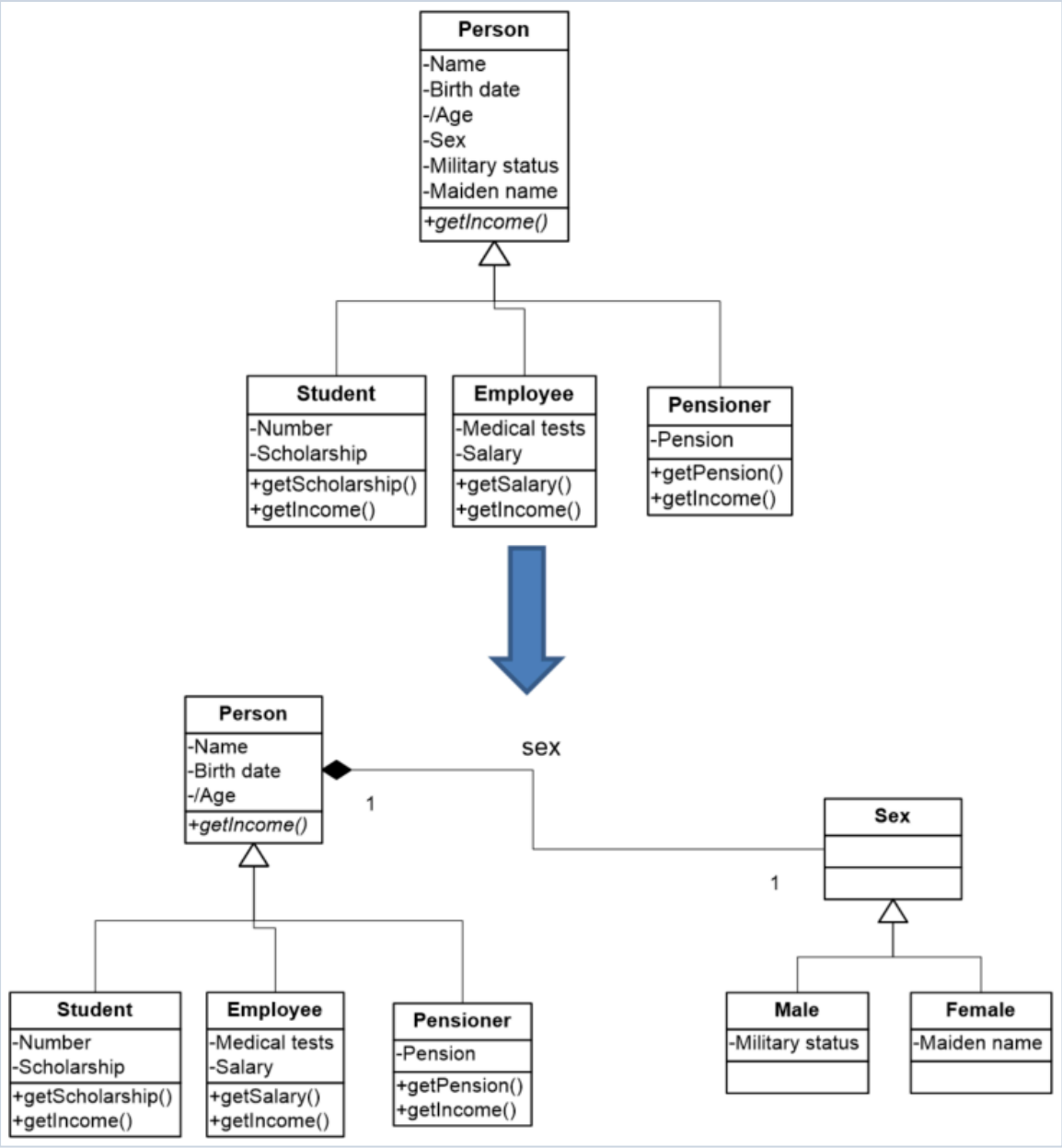
- Multi-aspect inheritance is different from multi-inheritance.
- Multi-aspect inheritance does not occur in any of the popular programming languages.

Examples of multi-aspect inheritance realization

Let's look at the examples. The following figure shows one possible solution to the problem with multi-aspect inheritance. The idea is to put attributes and methods from the removed aspect to the superclass. It is worth noting the notation of the attributes. Both *maiden name*, and *attitude to military service* are marked as optional. This is because, depending on the sex of the person, one of them will have no value.



Another solution for the same initial diagram is shown below. Instead of the subclass *Woman* and *Man* we have the composition and classes *Female* and *Male*. Note that class semantics have also changed. Sex is not a person (like a class previously), but only a certain set of information about it. Of course, you can discuss whether this change was needed. In my opinion yes, but if someone did not do it, then I would not consider it a mistake. Just in this type of a case, many designers have so many solutions. It is also worth noting that thanks to the cardinality of composition equal to 1 we will always be connected to exactly one object storing such information. And unlike the previous approach, this time no attribute is unnecessarily occupying space. The price for this is probably a little more complicated diagram, and what is usually followed by a more difficult implementation.



KNOWLEDGE CHECK — Which statements are true?

- The simplest way to implement multi-aspect inheritance is to flatten the hierarchy.
- Implementing multi-aspect inheritance one of the aspect should be implemented using a "real" inheritance of a given programming language.

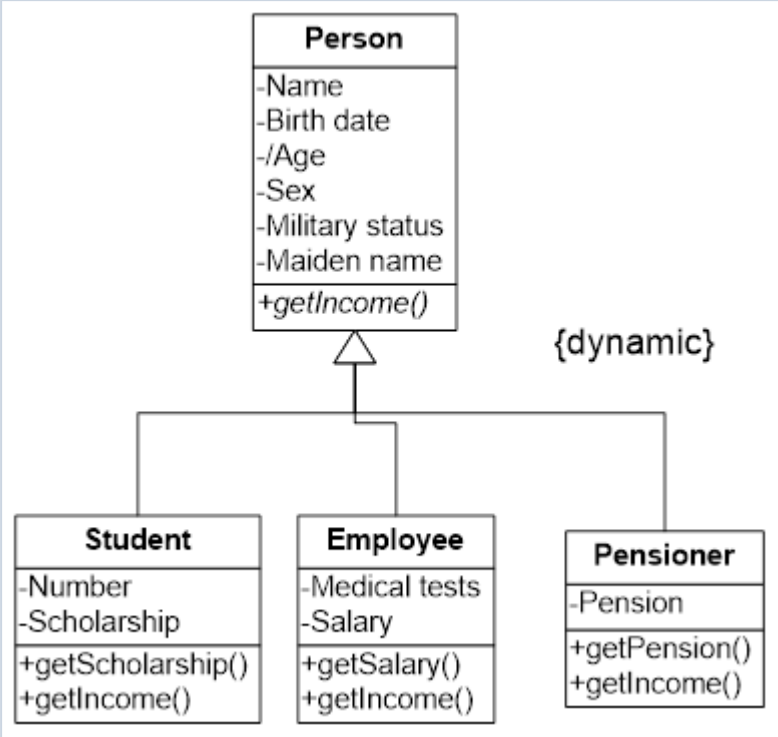
EXPLANATION & ANSWERS: **Explanation:**

- Flattening of the hierarchy is the easiest way to implement any kind of inheritance (including multi-aspect).
- Wherever possible, we use 'real' inheritance to use its facilities.

Dynamic inheritance - flattening hierarchy

Dynamic inheritance is quite unusual. So far as the object was created as *Employee*, it also ended its computer life as *Employee*. This is not the most convenient approach. In life it happens that someone is eg a student, later the same person is an employee and then, for example, a pensioner. Since we are modeling

real business situations, it would be good to have a mechanism that allows it. And that's what dynamic inheritance is for. Let's look at the diagram below. Stereotype <<dynamic>> means what we mean: dynamic inheritance, that is, subclass objects can freely change their membership while the system is running.



Like most of its predecessors, it does not exist in C++, nor in C#, nor in Java. Traditionally, we will have to implement them somehow manually using different techniques.

One is to place all invariants in the superclass and add the discriminator. We used this method in other types of inheritance, so we will not discuss it here in detail. Actually, this approach has only one advantage: easy implementation. The rest is probably only disadvantages.

KNOWLEDGE CHECK — Which statements are true?

- The simplest way to implement dynamic inheritance is to flatten the hierarchy.
- Flattening of the hierarchy should be combined with the addition of the discriminator.

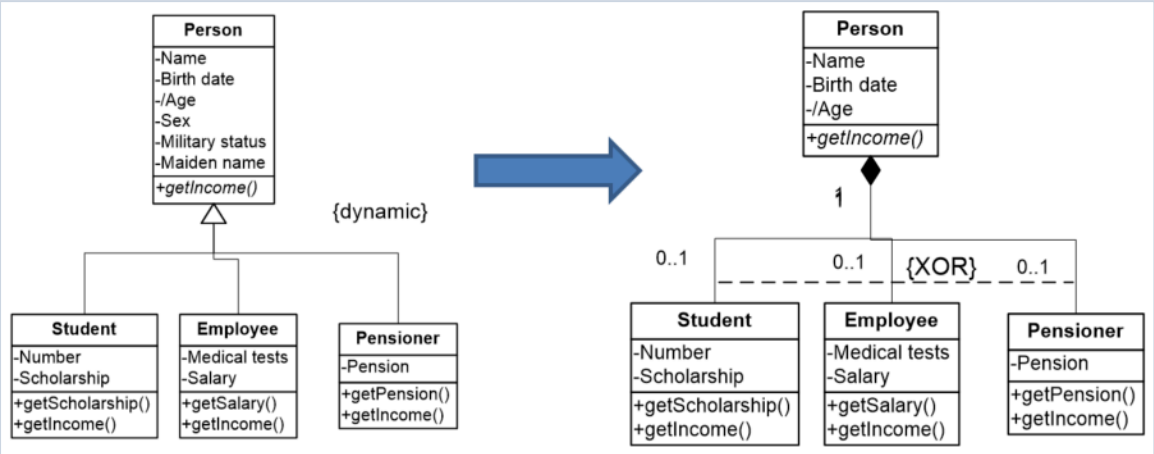
EXPLANATION & ANSWERS: **Explanation:**

- Flattening of hierarchy is the easiest way to implement any kind of inheritance (including dynamic).
- Adding a discriminator is a necessary requirement when flattening the hierarchy (so we know the 'type' of an object).

Dynamic inheritance - composition

One way to implement dynamic inheritance is to use aggregation/composition with the constraint *{xor}* (example shown below). We use the code created in case of the overlapping inheritance. In addition, we have included methods to facilitate the "change of class". Their task is to remove the existing object-part (responsible for the "dynamic aspect" of the object) and create a new one. For the example this could be a

transforming worker into a pensioner or a student into an employee. These methods are placed in the object-whole (class Person).



KNOWLEDGE CHECK — Which statements are true?

- Implementing a dynamic inheritance based on a composition should use the {or} constraint.
- In the above case, the composition must be done by inner classes.

EXPLANATION & ANSWERS: **Explanation:**

- We should use the {xor} constraint (because {or} allows two connections at a time).
- The composition can be realized in any way.

Dynamic inheritance - copying

"Smart" copying objects. The idea is to replace the old object belonging to the original class, which is an instance of a new class - this "after the transformation". To do this, in each subclass we create additional constructors. Each of them accepts a reference to the superclass object as a parameter (plus possibly additional business-specific information specific to the class). Information common to all objects located at a given hierarchy level is copied from the inside of the received object into the newly created one. Thus for our example, by changing the "old" *Student* class in the "new" class *Employee*, we will copy *name* and *birth date*. Other information specific to *Student* (*index number, scholarship*) is lost. If they are needed, then they can be preserved, but then it will not be "pure" dynamic inheritance.

At first glance this solution looks very good. It is quite easy to implement, and "works". Unfortunately not quite. The problem may be updating relevant references pointing to an "old" object (ie, links) to point to the new one. "Appropriate" references mean those that are common to the "old" and "new" class. Other references (those specific for the "old" class) "disappear" - as well as attribute values.

If you use the `ObjectPlusPlus` class, the solution to this problem is much easier. This is because we have information about objects that are pointing at us - because all the links in `ObjectPlusPlus` are bi-directional. You also need to remember to take care of the extent, that is removing "old" objects from it and adding the "new" one.

Sample code implementing part of this functionality is shown below. As we can see, the `Employee` class has a constructor (24) that retrieves an instance of the `Person` class (actually its subclass, because

`Person` is an abstract class) and `Employee`-specific parameters. And where is the copying attributes code we wrote about? Thanks to the call of the super class (26) we do not have to do this manually. Then the employee-specific information is stored (29).

```Java

public abstract class Person {
	protected String firstName;
	protected String lastName;
	protected LocalDate birthDate;

	public Person(String firstName, String lastName, LocalDate birthDate) {

		this.firstName = firstName;
		this.lastName = lastName;
		this.birthDate = birthDate;
	}

	public abstract float getIncome();

	@Override
	public String toString() {
		return this.getClass().getSimpleName() + ": " + firstName + " " + lastName;
	}
}

public class Employee extends Person {
	private boolean medicalTest;
	private float salary;

	public Employee(String firstName, String lastName, LocalDate birthDate, boolean
medicalTest, float salary) {
		super(firstName, lastName, birthDate);

		this.medicalTest = medicalTest;
		this.salary = salary;
	}

	public Employee(Person prevPerson, boolean medicalTest, float salary) {
		// Copy the old data
		super(prevPerson.firstName, prevPerson.lastName, prevPerson.birthDate);

		// Remember the new one
		this.medicalTest = medicalTest;
		this.salary = salary;
	}
}
```

And how can it be used? The example and effect are shown below. First we create the object describing the student (2), then convert it into the employee (6) and then into the pensioner (10).

```Java

// Create a student
Person p1  = new mt.mas.inheritance_dynamic.copying.Student("John", "Smith",
LocalDate.of(1994, 9, 14), 1212, 2000.0f);
System.out.println(p1);

// Create an Employee based on the Student
p1 = new mt.mas.inheritance_dynamic.copying.Employee(p1, true, 4000.0f);
System.out.println(p1);

// Create a Pensioner based on the Employee
p1 = new mt.mas.inheritance_dynamic.copying.Pensioner(p1, 3000.0f);
System.out.println(p1);
```

```Plain Text

Student: John Smith
Employee: John Smith
Pensioner: John Smith
```

KNOWLEDGE CHECK — Which statements are true?

- In the above code, removing a constructor from thePersonclass will speed up the program without changing its semantics.
- In the practical implementation of the discussed method, updating references does not matter.

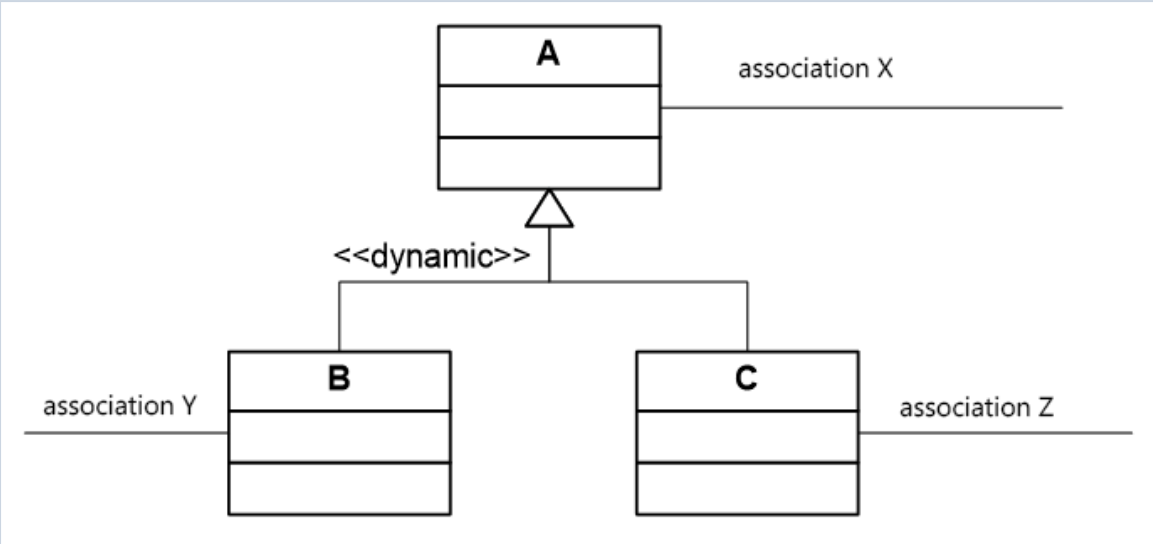
EXPLANATION & ANSWERS: **Explanation:**

- The constructor of thePersonclass is used to copy attribute values.
- In the 'real' program updating of references between objects is essential.

Problems related to the implementation of dynamic inheritance with the use of copying

What about updating the links pointing at the "old" object? This should not be difficult, because if you use the `ObjectPlusPlus` class, most of the required information is already in your system. In addition to copying the links you need to take care of a few things:

- from the "old" object, we should copy only data belonging to the "superclass" (eg A) - the figure below. We do not copy associations belonging to its association (eg B, C).
- To correctly update bindings showing the "new" object, we need information about the relationships of roles. Thus which *roleName* corresponds *reverseRoleName*. Theoretically, we can replace all instances (in all roles) *targetObject* showing an "old" object, but some of these updates are unnecessary because some of the links are "abandoned". In other words, when there are several associations (belonging to different associations / roles) between the same objects, we are unable to determine which items form pairs.


A
association X
<<dynamic>>
B
C
association Y
association Z



AUXILIARY ASSIGNMENT

Develop an enhancement for the `ObjectPlusPlus` class that implements dynamic inheritance. It may be based on the above idea or some other improvements.

The provided solution (along with a brief description, source code and example of utilization) can receive up to 5% of the points planned for mini-projects.

Three universal ways of implementing inheritances

In popular programming languages there is only the simplest type of inheritance. In fact, all other types of inheritance can be implemented by one or more of the following techniques:

- Replace the hierarchy with a single class. The undoubted advantage of this solution is the great ease of implementation. Sometimes it may be apparent, for example, when you need to override

the methods (and polymorphic call) using multiple conditional statements (eg `case` or `ifs`). Another disadvantage is the underutilization of resources.

- Use of aggregation / composition. We use the resources in an optimal way. Depending on the implementation details, especially when we do not use pre-built libraries (eg `ObjectPlusPlus`), we can expect a fairly laborious implementation (including "wrapping" methods), although it is usually not too difficult.
- Interfaces. The disadvantage is that it is quite laborious even when we apply some of the design patterns. The advantage is the wide possibilities and the fact that it is a very common technique (and therefore quite familiar to most programmers).

Summary

Remember that all the presented approaches bypass inheritance and they are not equivalent constructs. Therefore, wherever possible it is necessary to use inheritance instead of its substitutes.

Unlike associations, where we just have to decide whether we use identifiers or references, there is no perfect solution in the case of inheritance. Each case should be treated individually.

Source codes



Download all sources for MAS lectures.

LESSON 08

Advanced UML Constructs

UML constraints: {ordered}, {bag}, {xor}, {subset}, and business rule implementation in code.

Educational outcomes

Student:

- explains how to implement constructs such as: different types of constraints in UML (dynamic and static) in natural language (understanding);
- identifies and corrects errors in the source code, which is an implementation of the above structure;
- lists constructs implemented in the source code (understanding);
- implements the above structures using the selected programming language (application);
- indicates functional differences between two selected programs based on code snippets source (analysis);
- creates a program that implements specific business requirements using the constructs learned from the lectures (synthesis);

Justification

Modern programming languages are mostly based on the object paradigm. Therefore, its practical understanding is a necessary basis for working in this profession. Moreover, programmers often make systems based on analytical materials. And these largely use object-oriented graphical notations (such as UML). During this and the next lessons, you will deepen your theoretical bond, but first of all you will learn how to implement different concepts from object-orientness. This will allow you to design and implement modern business applications.

UML extensibility mechanisms

When discussing attribute types, we mentioned a special type of UML expressions called constraints. This is one of the extensibility mechanisms that makes it possible to refine the model. In other words, all the information we are unable to put on the diagram in the "standard" way, we can save as constraints. For this purpose, we can use a special OCL (Object Constraint Language), mathematical expressions (arithmetic, logic), plain text or combinations of these techniques. Regardless of the chosen technique, the content of the constraint is enclosed in curly braces.

Constraints can be divided into dynamic and static.

KNOWLEDGE CHECK — Which statements are true?

- The most precise way to write constraints is the natural language.
- In practice, no constraints are used because everything can be expressed as a "standard" notation.

EXPLANATION & ANSWERS: **Explanation:**

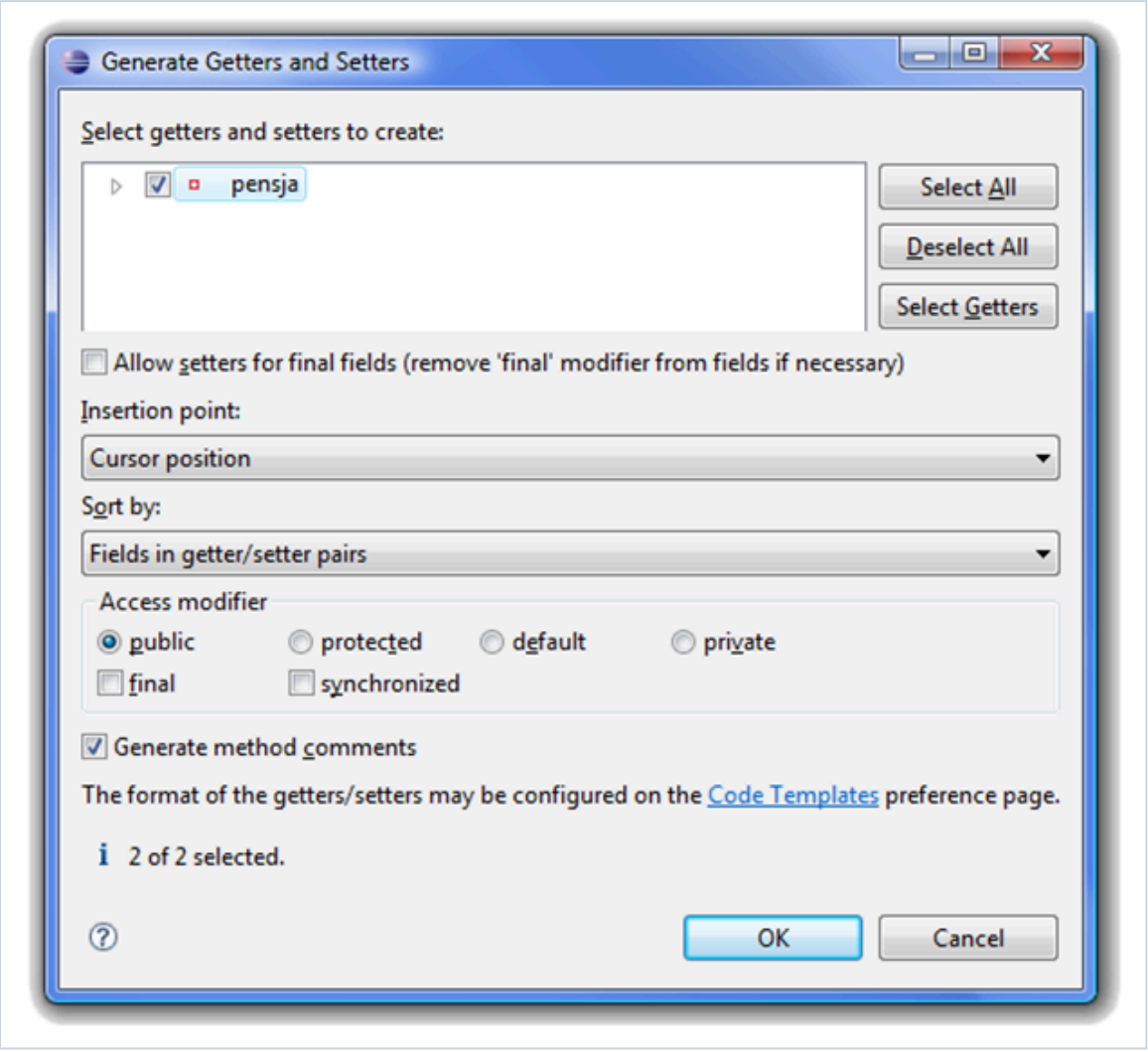
- Unfortunately, natural language is one of the least restrictive ways of defining constraints.
- Of course, usually you can not express all system requirements with standard notation.

Implementation of constraints

Unfortunately, in popular programming languages (Java, C#, C++) constraints are not directly present. One way to apply them is to implement them manually in the form of methods. The degree of complexity of such implementations depends on the type of constraint. There are some OCL implementations, but they usually work with model libraries (eg EMF) and therefore we will not be dealing with them.

Implementing constraints is best to assume that the constraint element (eg attribute) is hidden (`private`) and any modifications are made by dedicated methods (eg *settery*). The above rule also applies within a class that has attributes. So having a method in a class that references an attribute in the same class, we should do it using the appropriate getter or setter rather than directly. Unfortunately, this assumption can not be controlled beacuse even if the attribute is private, then invariants in its class have access to it.

As you can see, this approach requires the use of orthodox encapsulation. The consequence of this decision is the need to write appropriate methods for each of the attributes in the class. Fortunately, modern IDEs support the creation of methods for handling "hidden" attributes. For example, in Eclipse there is a dedicated option in the context menu: *Source – Generate Getters and Setters*.



We have to decide what to do when the constraint is broken? In the examples below we throw exceptions, but of course this will not always be the best solution. Sometimes it is better to use the assertion mechanism (available in some programming languages) or to display the appropriate message.

KNOWLEDGE CHECK — Which statements are true?

- The constraints in Java use the specialconstraintkeyword.
- The basic principle used to implement constraints is to create public attributes and private methods.

EXPLANATION & ANSWERS: **Explanation:**

- There is noconstraintkeyword in Java.
- The basic principle used to implement constraints is to create private attributes and public methods (not vice versa)

Dynamic constraints

In the case of dynamic constraints, the previous condition of the constraint element is important during validation. Examples may be: "{salary can not go down}" or "{salary increase must be smaller than 10%}".

Let's take a look at methods implementing this first constraint. Let's note that the `salary` attribute is private (2), and any external changes are only possible using the `setSalary(float salary)` (8) method. This makes it possible to meet the imposed requirements.

```Java
public class Employee {
        private float salary;

        public float getSalary() {
                return salary;
        }

        public void setSalary(float salary) throws Exception {
                // Validate all the constraints
                if(salary < this.salary) {
                        throw new Exception(String.format("The salary (%s) cannot be
decreased!%d", salary));
                }

                if(this.salary * (1d + maxSalaryChangePercentage /100d) < salary) {
                        throw new Exception(String.format("The salary (%s => %s) increase
cannot be more then %s%%", this.salary, salary, maxSalaryChangePercentage));
                }

                this.salary = salary;
        }

        public final static int maxSalaryChangePercentage = 10;
}
```

Static constraints

Unlike the above case, when checking the static constraint, the previous state of the constraint element is irrelevant, for example, "{salary > 2000,00 PLN}". Here is an example implementation based on similar rules. Note the provided constant (`minimalSalary`) - so that any updates will go smoothly.

```java
public class Employee {
        private float salary;

        public Employee(float salary) {
                this.salary = salary;   // required direct assignment
        }

        public float getSalary() {
                return salary;
        }

        public void setSalary(float salary) throws Exception {
                // Validate all the constraints

                if(salary < minimumSalary) {
                        throw new Exception(String.format("The new salary (%s) has to be at
least %s", salary, minimumSalary));
                }

                this.salary = salary;
        }

        public final static float minimumSalary = 2000;
        public final static float maximumSalary = 5000;
        public final static int maxSalaryChangePercentage = 10;
}
```

KNOWLEDGE CHECK — Which statements are true?

- Changing the way of accessingsetSalarytoprivatewill improve performance and will not change the program semantics.
- Declaring constants in constraints is a good practice.

EXPLANATION & ANSWERS: **Explanation:**

- Changing the way of accessing thesetSalarymethod onprivatewill not change performance, but it will certainly change the semantics of the program (it will prevent calling from outside of the class).
- By using the declared constants, we can easily change them in one place in the future.

Constraint {Unique}

The {unique} constraint is imposed on the attribute and ensures its uniqueness throughout the extent. In other words, the system will not create an object whose attribute value will be duplicated. An example might be *PESEL*.

For the implementation of {unique} we need to obtain a unique (in extent) value of a particular attribute of the object, eg the mentioned PESEL. This can be done by modifying the "setter" and adding a class method that looks through the whole extent and verifying that the given value no longer exists. This solution is

characterized by quite poor performance. It would be better for each unique attribute to add a class attribute (eg `Set`) to store all the valuesthat are already present.

An alternative to this is to add a single `map` class attribute with the key as an attribute name and the `Set` as values.

Another way is to use a `static Map`, but with the key being a unique attribute and the value of the object from the extent. Thanks to this, we ensure not only the uniqueness of the attribute, but also a quick retrieval of the object based on the value of the attribute. And that's the common reason for using `unique`.

KNOWLEDGE CHECK — Which statements are true?

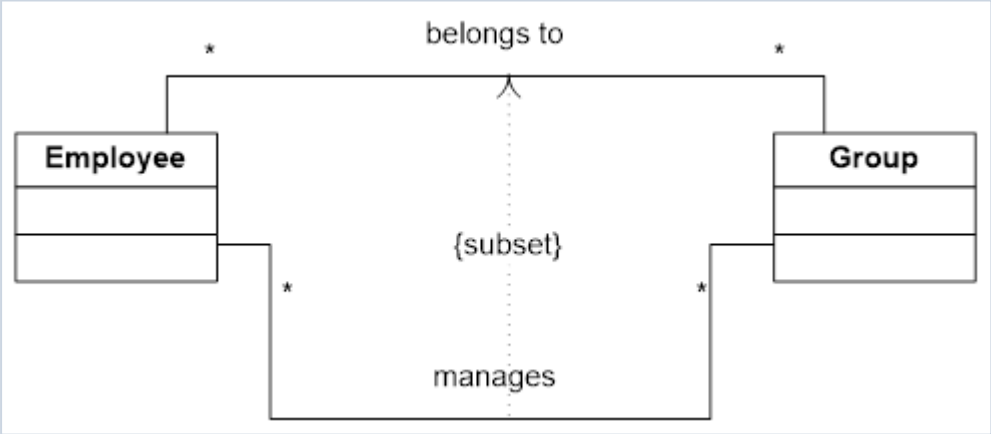
- Implementing {unique} with the class attributeSetwill improve performance at the expense of memory.
- Implementing {unique} using the extent reduces memory consumption at the expense of performance.

EXPLANATION & ANSWERS: **Explanation:**

- TheSetcontainer will speed up the search, but of course it will take extra memory.
- The second approach will work (significantly) slower, but without consuming extra memory.

Constraint {Subset}

The {subset} constraint can be applied to two associations (or aggregations). Let's look at the figure below. In order to be able to create a link within an association B ("direct"), there must already be a link within association A ("belongs"). Of course, both associations should be between the same classes and both should be between the same objects.



Links within the association described by the roles *"consists of","belongs to"* we add "normally". By contrast, before adding a link within the association described by the roles *"driven by", "direct"*, we check to see if there is a link to the object being added to the "parent" association. To automate the whole process, we will add new functionality to the `ObjectPlusPlus` class. We will create a method that will determine if there is a link to the given object within a given role. We will put all the functionality of the constraints in a new class called `ObjectPlus4`. In order not to lose our existing "facilities", it will inherit from `ObjectPlusPlus`.

Let's start with the method in the `ObjectPlusPlus` class, which checks for a link to the given object within a given role.

```Java
public boolean isLink(String roleName, ObjectPlusPlus targetObject) {
        Map<Object, ObjectPlusPlus> objectLink;

        if(!links.containsKey(roleName)) {
                // No links for the role
                return false;
        }

        objectLink = links.get(roleName);

        return objectLink.containsValue(targetObject);
}
```

- (4). We check whether we have information about the given role (in the method parameter)
- (6). If not, then we throw an exception.
- (8). At this point in the program, we are sure there is an entry of the links for the given role name. Thanks to that, we get the right collection.
- (10). We are investigating whether the mapping container storing the link information contains a value representing the object being inspected. The result is returned.

The next element that we need to implement is a method that adds a link together with the *{subset}* constraint. One possible solution is shown in the following listing:

```Java
public abstract class ObjectPlus4 extends ObjectPlusPlus {

        // [...]

        public ObjectPlus4() {
                super();
        }

        public void addLink_subset(String roleName, String reverseRoleName, String
superRoleName, ObjectPlusPlus targetObject) throws Exception {
                if(isLink(superRoleName, targetObject)) {
                        // There is a (super) link to the added object in the super role
                        // Create the link
                        addLink(roleName, reverseRoleName, targetObject);
                }
                else {
                        // No super link ==> exception
                        throw new Exception("No link to the '" + targetObject + "' object
in the '" + superRoleName + "' super role!");
                }
        }
}
```

- (2). A constructor that refers in its body to the superclass constructor (we remind you that this is a prerequisite for the proper behavior of objects belonging to classes inherited from `ObjectPlus`).

- (7). We call a method to check the existence of a link for a given role (discussed earlier).
- (10). We add a link using existing ObjectPlus mechanisms.
- (14). If there is no link within the parent association (the constraint is violated), then we throw an exception.

An example of using the above constructs. We create a worker (1) and a group (2). Then we add parent (5) and subordinate associations using a special method (7). And for validation we display information (10).

```Java

var employee = new Employee("John", "Smith");
var group = new Group("Group no 1");

try {
        // Add the ordinary link
        employee.addLink(roleBelongsTo, roleConsistsOf, group);

        employee.addLink_subset(manages, roleManagedBy, roleBelongsTo, group);

        // Show links info
        employee.showLinks(roleBelongsTo, System.out);
        employee.showLinks(manages, System.out);

} catch (Exception e) {
        // Error
        e.printStackTrace();
}
```

KNOWLEDGE CHECK — Which statements are true?

- The {subset} implementation requires a method to check for the existence of a given link.
- The {subset} constraint holds the associations in a fixed order.

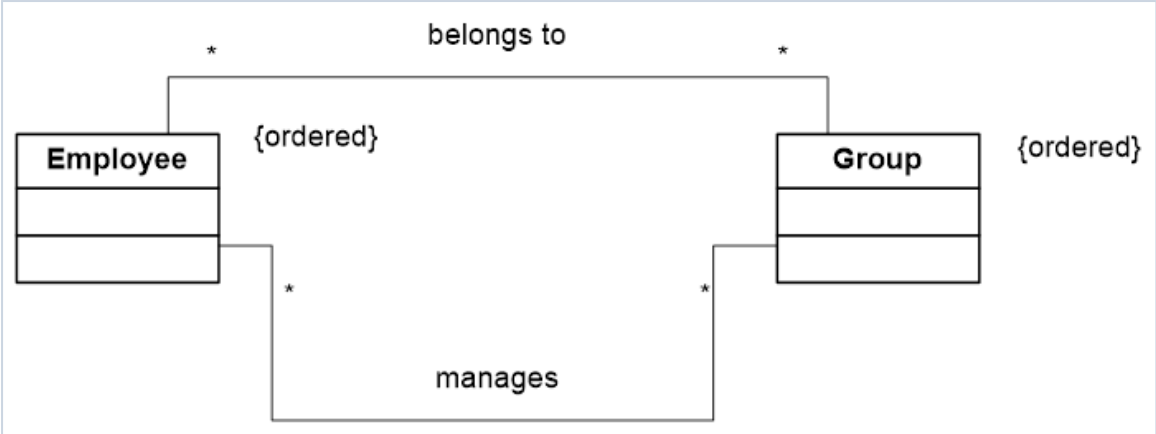
EXPLANATION & ANSWERS: **Explanation:**

- One of the basic activities that needs to be done is to check the existence of a particular link.
- Using {subset} does not affect the order of links.

Constraint {Ordered}

Constraint {ordered} may concern:

- Association. In this case, the links are stored (and received and processed) in a certain order.
- Class. In this situation, the objects in the extent are stored (and received and processed) in a certain order.



Implementing the {ordered} constraint is relatively straightforward. All we have to do is ensure that the order of the items stored does not change. To do this, you need to use a specific container type. If we want elements to be stored in the order of addition, we use eg `List`. However, when you need specific sorting criteria, you may want to use a container with a comparator (eg `TreeSet`).



COMMENTARY

A comparator is a special object whose purpose is to determine the order of the items being compared (it implements the *Comparator* interface). This allows the sorting method to know in what order to place the objects in the specified container.

By default, the `ObjectPlusPlus` class uses the `Vector` class for extent (guaranteed order) and `HashMap` for links. In summary, implementation of the {ordered} restriction is fairly straightforward, and in practice it boils down to the use of a certain type of container (guaranteeing the order of items).

KNOWLEDGE CHECK — Which statements are true?

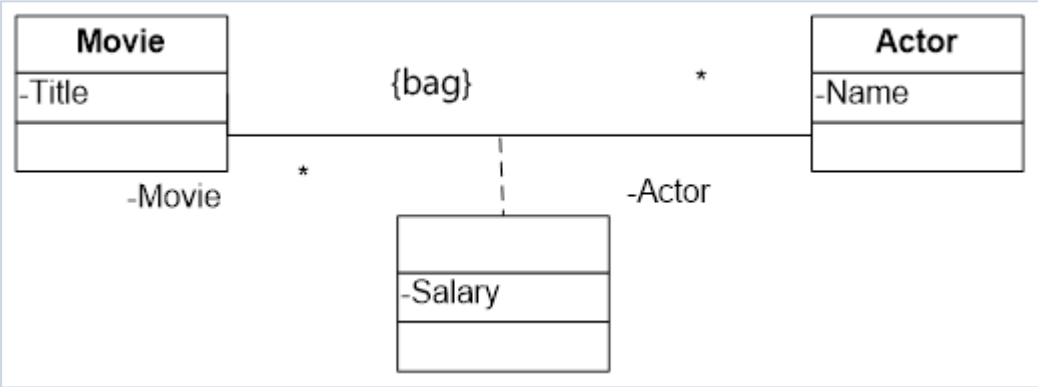
- The {ordered} implementation requires a method to check for the existence of a given link.
- By using {ordered}, we can store duplicate links.

EXPLANATION & ANSWERS: **Explanation:**

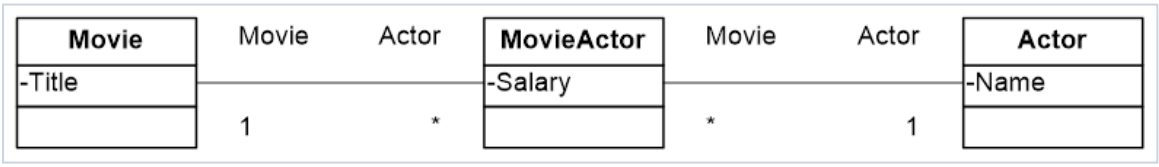
- Implementation {ordered} does not require checking the existence of the specified link.
- Constraint {ordered} does not affect the ability to store duplicates.

Constraints {Bag} and {History}

A classic association, without any additional markings, does not allow for more than one link between the same objects. If for some reason we require it, we need to apply a {bag} or {history} constraint. They are similar to each other. The difference between them is quite blurry. Both constraints allow for duplicate links, but {history} highlights the aspect of change over time. Typically, this is the case for association with attribute, as shown below:



But as we recall, the implementation of such an association is based on transforming it into two binary associations and an intermediate class:



Let's look at what happened - the problem of the links between the same objects simply disappeared. Instead of a recurring connection between *actor* and *movie*, we have links between *actor* and *MovieActor* and *MovieAktor* and *actor*. Thus we do not need to do anything special. In this case, the {bag} constraint simply could be ignored and normally implements the association. However, it may happen that someone would like to store duplicates, eg to count how many times and how much someone drank a drink (for *Person* and *Drink* classes). In this situation you have to choose a container that allows you to remember the same reference many times, eg `Vector`. The `ObjectPlusPlus` class utilizes the `HashMap` class for links that guard the uniqueness of keys. You can add duplicate values, but with unique keys. Or you need to modify the design assumptions for the `ObjectPlusPlus` class. The {history} restriction is implemented the same way. We said earlier that it meant more or less the same as {bag}, but emphasizes the temporal aspect of the

information. Consequently, from the implementation point of view, we do exactly the same thing as for {bag} restrictions.

KNOWLEDGE CHECK — Which statements are true?

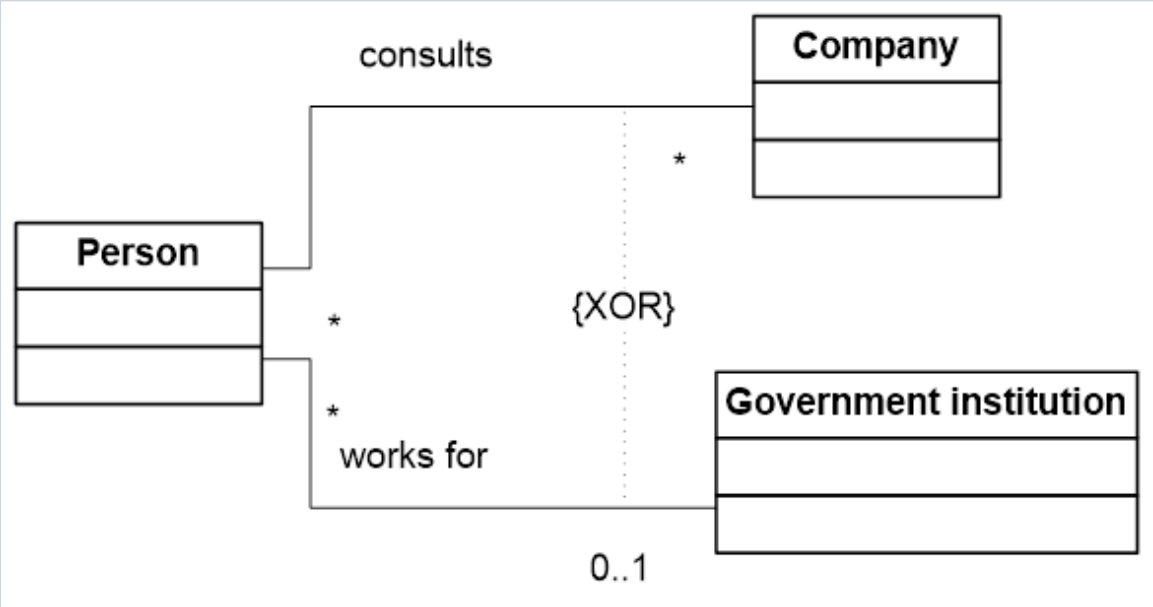
- Implementing {bag} requires a method that checks for the existence of a given link.
- By using {bag} constraint we can store duplicate links.

EXPLANATION & ANSWERS: **Explanation:**

- The {bag} constraint allows you to store any object, so you do not need to check for the existence of any links.
- The {bag} constraint allows you to store any links (including duplicates).

Constraint {XOR}

The constraint {xor} affects at least two associations. Ensures that there will be only one link within them. Let's look at the example below: If *person* is working in *government*, it can not work with *company* and vice versa.



What do we need to do to implement this type of a constraint? This case is quite obvious - it is important to ensure that there is only one link in the defined association group (roles). We will show an example solution for the `ObjectPlus4` class:

```Java
public abstract class ObjectPlus4 extends ObjectPlusPlus {

        private List<String> rolesXOR = new LinkedList<>();

        // [...]

        public void addXorRole(String xorRoleName) {
                rolesXOR.add(xorRoleName);
        }

        public void addLinkXor(String roleName, String reverseRoleName, ObjectPlusPlus
targetObject) throws Exception {
                if(rolesXOR.contains(roleName)) {
                        // The currently adding role is XOR'ed

                        // Check if there is a link for XOR'ed roles
                        if(isXorLink()) {
                                throw new Exception("There is a link for a XOR roles!");
                        }

                        // There is no link ==> add a link using an already existing method
from a super class
                }

                // Add the link
                super.addLink(roleName, reverseRoleName, targetObject);
        }

        private boolean isXorLink() {
                for(String role : rolesXOR) {
                        if(this.anyLink(role)) {
                                return true;
                        }
                }

                return false;
        }
}
```

The modification begins with the addition of a container (2) storing the names of the constraint roles. Then we create a method which adds the given role name (6). This addition means that association within this role is constrained. In the main method of creating the connection (10), we do the following:

- (11). We are checking whether the role being added is {xor} restricted.
- (15). For constrained roles, we check that if there is already a link. If so, we throw an exception. If there are no links, the control goes to the standard method that creates the connection between the objects.
- (23).Calling a "normal" method (from the superclass) that creates a link between objects.

This sample implementation can be supplemented by the support for adding links from the other side.

KNOWLEDGE CHECK — Which statements are true?

- Implementing {xor} requires a method to check the existence of a given link.
- By using {xor} we can store duplicate links.

EXPLANATION & ANSWERS: **Explanation:**

- The constraint {xor} ensures that there is only one link at the same time, so it needs to verify the existence of the given link.
- The constraint {xor} does not specify the uniqueness of the related objects.

Custom constraints

In this lesson, we outlined the most common "standard" constraints defined within UML notation. Note, however, that the analyst has complete freedom in formulating constraints and does not need to use only those provided by the UML creators. What to do in that case? For any other constraints, such as "plain text", you must:

- Understand "what the author meant". This part will probably be the most difficult, especially when the restriction is written without the use of formalism, ie, plain text. If in doubt, ask the person who formulated it. It may turn out (for example, at the test stage) that due to misunderstanding of the system requirements it is necessary to make quite significant changes in the code of the program.
- Implement them using the appropriate method.

Sometimes a seemingly minor limitation may even require a change in the approach to program development, such as the use of orthodox encapsulation. And it is important to realize that we will rarely be able to use ready-made solutions.

Source codes



Download all sources for MAS lectures.

LESSON 09

Relational Model in OOP

Relational vs. object paradigms, impedance mismatch, table representations of classes, and primary/foreign keys.

Educational outcomes

Student:

- explains how to implement different concepts of objectivity in relational model using natural language (understanding);
- identifies and corrects errors in the source code, which is an implementation of the above structure;
- lists constructs implemented in the source code (understanding);
- implements the above structures using the selected programming language (application);
- indicates functional differences between two selected programs based on code snippets source (analysis);
- creates a program that implements specific business requirements using the constructs learned from the lectures (synthesis);

Justification

Modern programming languages are mostly based on the object paradigm. Therefore, its practical understanding is a necessary basis for working in this profession. Moreover, programmers often make systems based on analytical materials. And these largely use object-oriented graphical notations (such as UML). During this and the next lessons, you will deepen your theoretical bond, but first of all you will learn how to implement different concepts from object-orientness. This will allow you to design and implement modern business applications.

Popularity of the relational model

Let's start with the fact that practically every designer or programmer who creates a little bigger system uses a database. And nowadays the term "database" still means "relational database". From the point of view of the designer or a programmer, the reasons for this are not very important. The most important thing is to be able to deal with this situation. What is the problem with the relational model? We must combine two, or rather three, worlds: object-oriented used in analysis, object-oriented used in programming languages (smaller) with the relational world used in the data warehouse. In other words: by analogy with the transition from a rich object model (analysis) to a poorer object model (Java, C#, C++), we need to provide object-oriented mapping to relational one and vice versa.



COMMENTARY

What is the reason for the popularity of relational databases? There are many factors and you could write about this quite extensive paper, but let's briefly list the most important ones:

- Investments made. Many companies have (probably) billions of dollars spent on tools and training. It is hard to imagine that they would want to abandon it quickly without very good reasons.
- Legacy systems. There are currently a large number of operating systems that use relational databases. Their replacement, or even upgrading, would be a very costly process, and in some cases practically impossible ("Please do not use our services for a month because we are implementing a modern database management system").
- Successors. It is difficult to indicate an unambiguous succession. For years, it was expected to be object-oriented databases, but so far nothing has worked out. It is, due to the lack of an acceptable standard (there are many "facial" objects). Perhaps it will be technologies called *No SQL*.

KNOWLEDGE CHECK — Which statements are true?

- Relational model is popular because of the richness of its structure.
- Relational databases for sure will be replaced by object-oriented.

EXPLANATION & ANSWERS: **Explanation:**

- There are not many constructs in the relational model.
- At the moment it is difficult to judge which technology/concept will replace the relational solutions.

Assumptions of a relational model

The exact description of the relational model is beyond the scope of the MAS lecture, but we will briefly recall the basic concepts:

- The equivalent of classes are tables.
- Each table can have any number of columns storing attribute values. Each such column is of a certain type, eg number, text, binary value. There are actually more types, but we will not take care of them.
- Table rows (called tuples or records) describe instances - they are equivalent to objects.
- The order in which the records appear in the table is not business-relevant.
- Each table can have a primary key and foreign keys.
- The primary key is used to uniquely identify a particular record, and is essential for connecting tables (the equivalent of an association).
- Foreign key identifies the associated tuple, usually from another table, such as *Person* and *Company* tables. One of the people has a primary key value, such as 12345. In the *Company* table, there is a column containing the foreign key relating to the person and telling us that the person with the key value of their primary key (12345) owns the company.
- Some columns in tables may have a special *NULL* value. This means that there is no value, eg no maiden name.
- Multi-to-many relationships must be modeled using intermediate tables.

As you can see, the above model is very simple and devoid of most constructions known from object-orientness. I think we will agree that using it means working at a lower abstraction level, more technical than the object model.



IMPORTANT NOTE

In our considerations regarding the relational model and its relationships with the object model, we take into account its classical assumptions. We do not use specific solutions found in some database management systems.

KNOWLEDGE CHECK — Which statements are true?

- Relational model uses the concept of object identity.
- One of the basic relational model concepts is the primary key and the foreign key.

EXPLANATION & ANSWERS: **Explanation:**

- The identity concept of an object exists in an object model, not a relational one.
- Indeed, one of the basic relational model concepts is the primary key and the foreign key used to model the connection.

Databases, as a layer of persistency

Let us consider for a moment what are the main reasons for using databases. It seems that one of the most important is the query language. It enables very easy data processing, saving you a lot of work.

Another important argument is obviously performance. Over the years, database management systems have been equipped with a variety of mechanisms (including indexes) that dramatically accelerate the performance of any operation. In many systems, it is true that modern computers are so fast that in many cases they would have managed without a dedicated BD, using only the built-in mechanisms of the development platform.



COMMENTARY

Indexes in databases speed up access to data. Unfortunately, this is at the expense of more memory consumption (these are some additional data structures that the system needs to store somewhere). Some operations (such as insertion or updates) are also slower (because the system needs to take care of the data and the index for them).

The explosion of popularity of web-based applications has also serious impact on the databases. In such cases, some data persistence mechanism is needed, and BD is usually the most obvious choice.

Other reasons are related to security. Thanks to the built-in mechanisms, we have the ability to control access for different users.

And of course the ability to store a lot of information, but as we have said sometimes it is abused.

We have exchanged so many advantages of relational databases, but do they have any disadvantages? Perhaps the most important is the relational model itself...

KNOWLEDGE CHECK — Which statements are true?

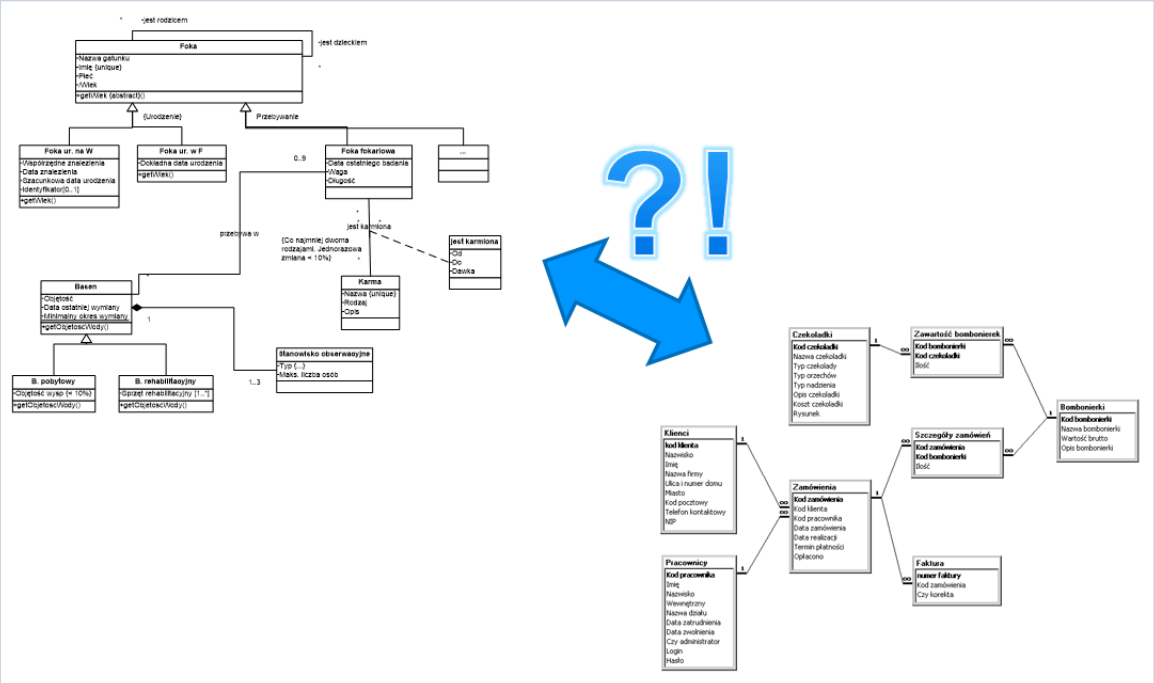
- Indexes increase the speed of all operations in the database.
- The explosion of popularity of web applications has contributed to the popularization of databases.

EXPLANATION & ANSWERS: Explanation:

- Indexes increase the speed of performing certainoperationsin a database (eg, search). On the other hand, they usually slow down updates.
- Most web applications use databases.

Impedance mismatch

The consequence of using relational databases from object-oriented programming languages is a phenomenon known as **impedance mismatch**. It is due to the need to match "two different worlds": relational and object-oriented.



In the next sections we will discuss various ways of mapping (translating) object constructs from programming language to relational data model. In the remainder of this tutorial we will also describe the tools that we can use to facilitate these operations: Object-Relational Mappers (ORMs).

KNOWLEDGE CHECK — Which statements are true?

- The impedance mismatch is due to the poor performance of databases.
- The impedance mismatch does not occur when you create a system entirely on the database side.

EXPLANATION & ANSWERS: Explanation:

- Impedance mismatch is not related to performance.
- Actually, when dealing with only one model, there is no impedance mismatch.

Class mapping

Mapping classes into tables is relatively straightforward, but you need to be aware of some restrictions. In general, we can say that classes are replaced by tables. Each attribute is stored in a separate column. Its type is similar to the attribute type in the object model. This rule only applies to simple types. In other cases we have to apply dedicated solutions - see the next lesson.

KNOWLEDGE CHECK — Which statements are true?

- Classes are mapped to relational tables.
- Simple attributes are directly mapped to columns in the table.

EXPLANATION & ANSWERS: **Explanation:**

- In fact, classes are mapped to relational tebels.
- Simple attributes are directly mapped to columns in the table.

Keys

As mentioned earlier, in the relational model we need keys to identify tuples. Therefore, for each table we add a special attribute - the key uniquely identifying our "object".

There are two approaches to these keys. Some programmers claim that to reduce the size of the data model, we should use business attributes. If we do not have a unique attribute, we should use their combination (the key may consist of several columns).

Another solution is to add a dedicated, non-business-related column. For performance reasons, it is best to have a number (of a sufficiently large type). Moreover, consistently using some convention (eg *IDNameTable* we will always know what our key is - and this is of course important when writing queries.

This second approach seems to be better: it provides better performance and makes it easier to memorize model details.

KNOWLEDGE CHECK — Which statements are true?

- The primary key is a combination of several columns of text type that provides better performance than a single-column numeric key.
- The primary key is used to identify a specific tuple.

EXPLANATION & ANSWERS: **Explanation:**

- In most cases, the greatest performance is provided by a single (large) number key.
- According to the assumptions of the relational model, the primary key is used to identify a particular tuple.

LESSON 10

Relational Model in OOP (Part 2)

Mapping complex and multi-valued attributes to relational database tables and normalization.

Educational outcomes

Student:

- explains how to implement different concepts of objectivity in relational model using natural language (understanding);
- identifies and corrects errors in the source code, which is an implementation of the above structure;
- lists constructs implemented in the source code (understanding);
- implements the above structures using the selected programming language (application);
- indicates functional differences between two selected programs based on code snippets source (analysis);
- creates a program that implements specific business requirements using the constructs learned from the lectures (synthesis);

Justification

Modern programming languages are mostly based on the object paradigm. Therefore, its practical understanding is a necessary basis for working in this profession. Moreover, programmers often make systems based on analytical materials. And these largely use object-oriented graphical notations (such as UML). During this and the next lessons, you will deepen your theoretical bond, but first of all you will learn how to implement different concepts from object-orientness. This will allow you to design and implement modern business applications.

Relational model - simple and complex attributes

As mentioned in the previous lesson, classes are mapped to tables, and attributes to columns. The latter rule applies, of course, to simple types only.

For complex attributes, the case is more complicated. We have several options, none of which is ideal. Consider the sample attribute *address*. We can store it as:

- one "flattened attribute" of the text type with content, eg *"Marszałkowska 10, 03-333 Warsaw, Poland*. The basic problem with this approach is the difficult access to its individual elements. For example, searching for any business in a city can be, in extreme cases, impossible;
- one text attribute but using special notation eg XML (below). Thanks to such structured construction it will be easier to process it. It seems that the problem is resolved, for example, from Java we can parse XML and get the particular entries. Remember, however, that we are referring to

information stored in the database, and in that case we mostly use query languages, such as SQL. And parsing XML from SQL, may not be possible. As a result, you would have to retrieve all of the addresses using SQL (as a text field) and then parse and filter data on the programming language side (eg Java). This will obviously seriously affect performance.

```XML
<Address>
        <Street>Marszałkowska</Street>
        <Number>12</Number>
        <ZipCode>03-333</ZipCode>
        <Country>Polska</Country>
</Address>
```

- additional columns (formed from simple attributes in the complex attribute) in the main table. This is, of course, only possible where the cardinality of the complex attribute is not greater than*1*.
- separate the *Address* table and connect it to the "main" entry with the key (figure below). It seems that this solution is the most feasible because it enables trouble-free processing of data using query language (eg SQL). You will sometimes have to use `JOIN` statement (when you want to get full company information, eg *name, tax ID, address*) and they have quite negative impact on efficiency. Unfortunately, also our data model is a bit complicated: for each complex attribute, we add an additional table.

IdAddress	Street	Number	ZipCode	Country
328	Marszałkowska	12	03-333	Poland

KNOWLEDGE CHECK — Which statements are true?

- Simple attributes are implemented in RM as columns in the table.
- Complex attributes can be implemented in RM as a separate table with a foreign key.

EXPLANATION & ANSWERS: **Explanation:**

- Indeed, simple attributes are implemented in RM as columns in the table.
- One way to implement complex attributes is to use an external table with the foreign key.

Relational model - manadatory and optional attributes

For optional attributes, the column in the table must allow the `null` value to be stored. In this situation, let's remember two things. First of all, our data will take up a little bit more space (because we have to remember the data itself and information about lack of value). Secondly, the value `null` can be a bit of a hassle in SQL queries.

The manadatory attribute will be defined as a column that does not allow `null`, and as a result, the database management system will not allow you to create an entry without the value.

KNOWLEDGE CHECK — Which statements are true?

- Optional attributes must be implemented as a separate table.
- The manadatory attributes are implemented as a column allowingNULLvalue.

EXPLANATION & ANSWERS: **Explanation:**

- The easiest way to include optional values is to allowNULLin the table column.
- The required attributes are implemented as a column that does not allownull(and not vice versa).

Relational model - single and multi-valued attributes

The single attribute does not cause any problems and is simply implemented as one column in the table.

Unfortunately, the multi-valued attribute gives us similar problems as complex one. Similarly, we have several options and we can implement it as:

- one column of text type with special syntax. We separate the values by commas, for example, or use some form of structure, such as XML, just like with complex attributes. Similar problems are also associated with this approach; In short, they are reduced to difficult access to individual values.
- separate table. Each of its row will be one business value with the foreign key referring to the "parent" table. This way you can store any data combinations, but at the cost of using SQL `JOIN` constructs, which can hinder access and/or reduce performance.

KNOWLEDGE CHECK — Which statements are true?

- One way to implement a repetitive attribute is to store its value in a text field.
- One way to implement repetitive attributes is to store values in a separate table.

EXPLANATION & ANSWERS: **Explanation:**

- This is a acceptable solution, but not very good.
- Thanks to this solution we have convenient access to individual values.

Relational model - object and class attributes

Unlike classes, the relational model does not provide a way for storing a single common value for all tuples of a given table. Therefore, we do not have anything close to semantics of the class attribute. For this reason, we need to apply some of our own solutions, none of which is ideal, e.g.:

- Create a special table that contains class attribute values from all classes. As a result, we will have many columns (for each class attribute) and only one row (because the class attribute has only one value). Such a table will semantically "occupy space" on the schema, really storing very little information.
- Declaring a special table with two columns: keys (class attribute names) and values. The disadvantage of this solution is also the type problem. Since we have only one column with a value, then all class attributes will have a common type, possibly a text. The alternative will be

several columns with values, such as date, number, and text. Of course, in a given row only one of them will be used.

- Save it outside the database, eg in a program code or configuration file.

We have mentioned that none of these methods is ideal, primarily because it means problems with the processing of a class attribute. Imagine any SQL query that includes such an attribute (eg maximum worker age) - in any case, it will be a problem.

KNOWLEDGE CHECK — Which statements are true?

- The best way to implement a class attribute is to define it on the programming language side without worrying about data persistency.
- The class attribute can be written to a dedicated table.

EXPLANATION & ANSWERS: **Explanation:**

- This approach is incorrect because it does not ensure the sustainability of business data (the value can change).
- This is one possible solution.

Relational model - derived attributes

In programming languages, the derived attributes were supported by methods. For a relational model, and a suitable relational database, we can create a dedicated:

- View - of course if your database management system supports it,
- Method in database programming language, eg in SQL,
- Method in programming language (outside of database).



COMMENTARY

Database views are virtual tables, from the user's perspective, behaving as standard ones. They are usually defined using a query language (eg SQL). This allows us to put in practically any functionality, e.g. one that is bound to derivative attributes, e.g. *age*.

Similarly as in previous cases, none of the above methods are ideal and involve larger (programmable language method) or smaller (procedural or SQL views) constraints.

KNOWLEDGE CHECK — Which statements are true?

- Derived attributes can be implemented as a special type of column in a regular table.
- The database view allows us to define, for example, the age of a person calculated on the basis of the date of birth.

EXPLANATION & ANSWERS: **Explanation:**

- Relational model does not give such an opportunity.
- This is the most feasible in modern BD.

Relational model - object and class methods

Unfortunately, in the classical relational model, methods do not exist at all. Fortunately, modern database management systems provide SQL extensions (PL / SQL or T-SQL) that allow them to be created. This is not always compatible with object-oriented semantics, especially with regard to object methods (access to specific information-entities).

As an alternative, you can take the implementation of these methods on the "classical" programming language (Java or C#). This may involve performance issues due to the need to (manually) process information derived from DB.

It seems that you often encounter a hybrid approach where basic business logic is implemented in the database programming language (eg SQL), and it is called from eg Java. Thanks to this, critical operations are fully implemented in a fast and optimized database environment.

KNOWLEDGE CHECK — Which statements are true?

- Every database makes it easy to define methods in tables.
- Each database makes it easy to define class methods in tables.

EXPLANATION & ANSWERS: **Explanation:**

- Defining methods in tables is not a standard feature of a relational model.
- Defining class methods in tables is not a standard feature of a relational model.

LESSON 11

Relational Model in OOP (Part 3)

Mapping associations (1..1, 1..*, *..*) and association classes to relational foreign keys and join tables.

Educational outcomes

Student:

- explains how to implement different concepts of objectivity in relational model using natural language (understanding);
- identifies and corrects errors in the source code, which is an implementation of the above structure;
- lists constructs implemented in the source code (understanding);
- implements the above structures using the selected programming language (application);
- indicates functional differences between two selected programs based on code snippets source (analysis);
- creates a program that implements specific business requirements using the constructs learned from the lectures (synthesis);

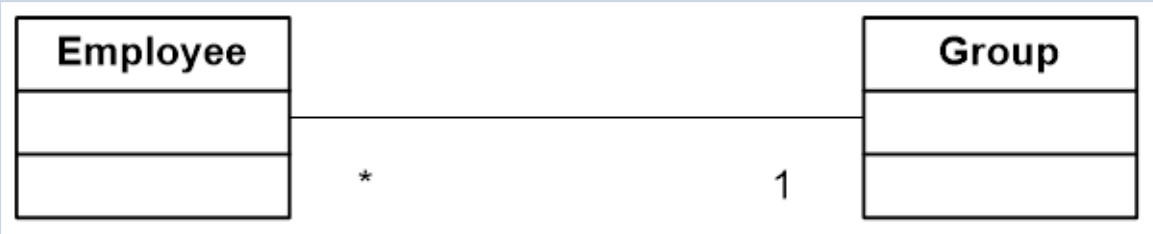
Justification

Modern programming languages are mostly based on the object paradigm. Therefore, its practical understanding is a necessary basis for working in this profession. Moreover, programmers often make systems based on analytical materials. And these largely use object-oriented graphical notations (such as UML). During this and the next lessons, you will deepen your theoretical bond, but first of all you will learn how to implement different concepts from object-orientness. This will allow you to design and implement modern business applications.

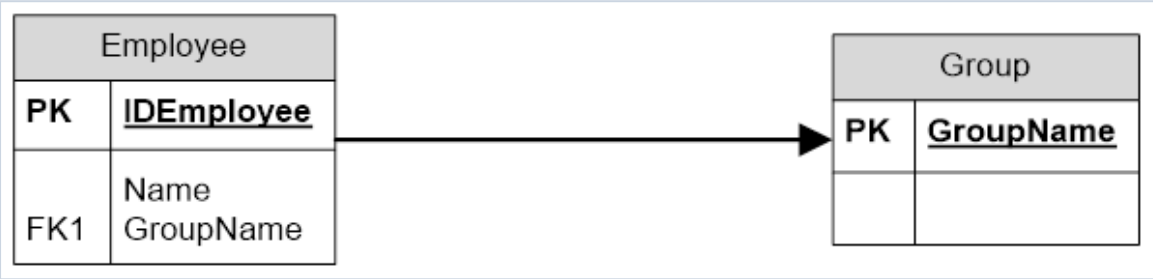
Association mapping 1:1 and 1:*

As we mentioned, connections between tables are described in the relational model using the primary key and the foreign key. Compared to the object model, this is a big inconvenience because it forces us to introduce purely technical elements to the business model. As if that was not enough, we do not have more advanced constructs, such as association with attribute, etc.

The basic type of association is replaced by a single relationship. This is for 1 - 1 and 1 - *. Let's look at the figure below showing the relationship between a n employee and a group.



The above business situation can be presented in the relational model as shown below:



As you can see, we created two tables. The employee table contains:

- PK – primary key,
- Column storing *name*,
- FK – foreign key storing the *group*identifier of the *employee*.

There are only two columns in the *Group*table: *primary key*and *name*.

Let's consider for a moment how will we get a list of all employees belonging to a particular group? In the object model, having an *Group*class instance (i.e. a reference or pointer to a *Group*class object), we immediately have access to all objects associated with it. Unfortunately, in the relational model, in order to get the same information, you need to look at all the employees and filter those who have the proper group ID. As you can see, this is a very serious fault. Fortunately, thanks to indexing, such searches are done fairly quickly.

KNOWLEDGE CHECK — Which statements are true?

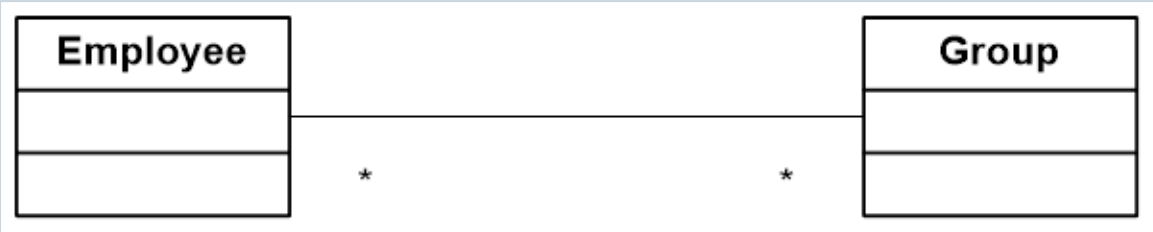
- Associations of 1:1 cardinality can be realized with a foreign key.
- Association 1:* can be realized with a foreign key.

EXPLANATION & ANSWERS: **Explanation:**

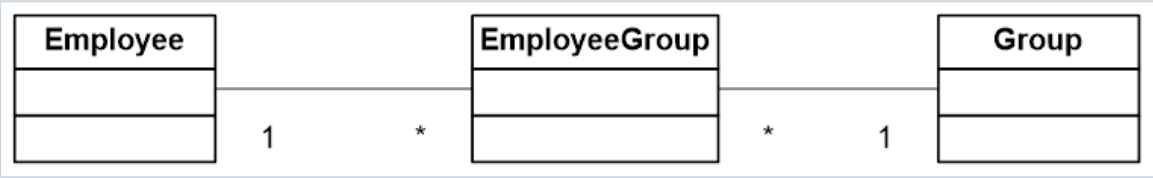
- Associations of 1:1 cardinality can be created with a foreign key. On the other hand, for such values, you might want to consider merging two tables.
- Association 1:* can be stored with a foreign key. Of course, with such numbers, it is not possible to merge two tables.

Association mapping *:*

The above solution works for the "1 - *" range. And what about the "* - *" case? Let's look at the modified class diagram:



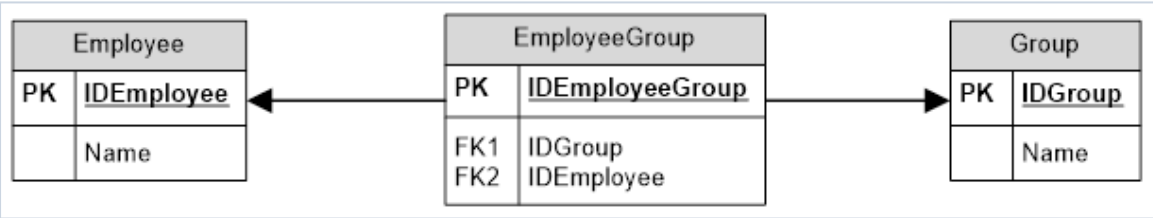
In the situation described here, the group may have many employees (as before) but the employee may belong to several groups. In order to present this situation in a relational model, we must first introduce an intermediate class. Thanks to this, one association "* - *" will be replaced by two associations with the numbers "1 - *". The effect of this transformation is shown below:



Since we now have two associations "1 - *", we can easily present them in the relational model:

- Three classes are mapped to three tables.
- In the "middle" table we put the foreign keys.

Example relational diagram for the above procedure:



In the intermediate table, there will be only foreign keys connecting *employees*with *groups*. In some cases we can add extra primary key for such pair. As it happens in the relational model, to find:

- *employees*belonging to a *group*,
- *groups*to which a specific *employee*belongs

we need to look through all the tuples in the *EmployeeGroup*table. Of course, the performance of this solution can be significantly improved by using proper indexing (on the database side).

KNOWLEDGE CHECK — Which statements are true?

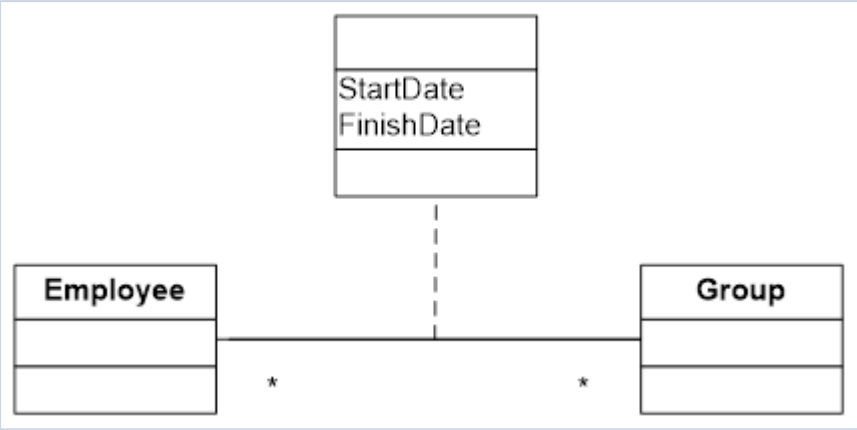
- Associations *:* can be accomplished using an intermediary table.
- Introducing an intermediary table facilitates understanding of business rules.

EXPLANATION & ANSWERS: **Explanation:**

- The only way to accomplish card. * - * is to use an intermediary table.
- Unfortunately, the introduction of the intermediate table 'obfuscates' the model: we have an extra table in the diagram, which has no business application.

Mapping association with attribute

Similarly the binary association of "* - *", we are dealing with an association with an attribute. Let's look at the class diagram:

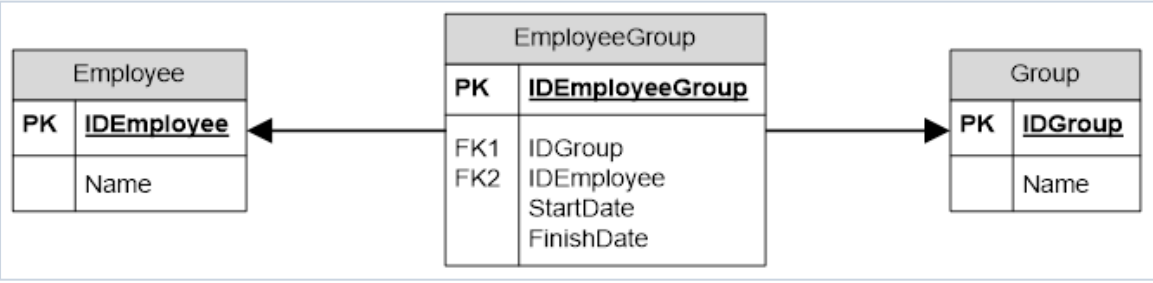


It expands the previous example to include time membership information. What do we have to do to be able to store such relationships in a relational model? Well, we do not have much choice:

- We introduce an intermediate class with attributes from the association class.
- Three classes are mapped to three tables.
- In the "middle" table we place foreign keys and attributes derived from the intermediate class (association class).



Relational version is in the drawing below. We see that apart from the foreign keys (as in the previous example), we also added columns describing the start and end dates.



KNOWLEDGE CHECK — Which statements are true?

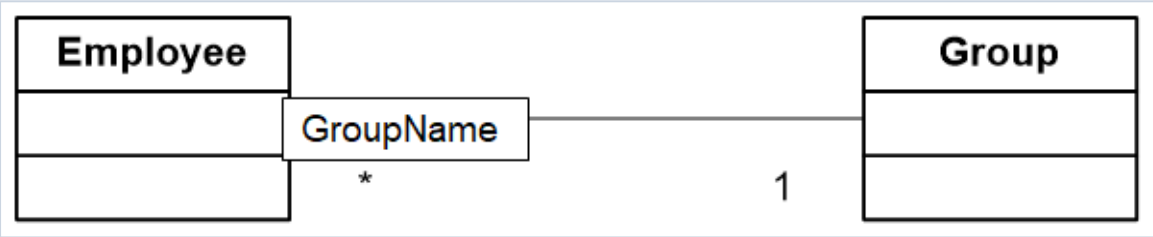
- You can add two primary keys and 4 foreign keys to avoid entering the intermediate table.
- Introducing an intermediary table facilitates understanding of business rules.

EXPLANATION & ANSWERS: Explanation:

- Of course, the only way to achieve an association with an attribute is to introduce an intermediary table.
- Unfortunately, the introduction of the 'obfuscate' intermediate table: we have an extra table on a diagram that looks identical to the other tables, making it difficult to analyze the model.

Mapping qualified associations

Qualified associations do not have a direct counterpart in the relational model. We can partially "simulate" it using a foreign key qualifier. Let's look at an example class diagram containing a qualified association:



An analyst has created such a design to quickly reach a group of specific employees based on its name. We can represent this in a relational model by using the *group name*as the primary key (in the *Group*) table and as the foreign key in the *Employee*table. Sample diagram containing two tables and one relation:



We see that the *Employee*table, in addition to primary *IDEmployee*, has a *Name*column and a foreign key (*GroupName*) being the group name. Thanks to this we are able to navigate from employee table to group table. One can argue whether a textual primary key is the right thing to do. It is certainly acceptable (by database management system).

KNOWLEDGE CHECK — Which statements are true?

- A design that is close to a qualified association can be implemented by selecting the correct primary and the foreign keys.
- The primary purpose of using a qualified association is to speed up the search for connected objects.

EXPLANATION & ANSWERS: **Explanation:**

- One way to 'simulate' a qualified association is to use a qualifier as a key.
- Indeed, the primary task of a qualified association is to speed up the search for connected objects.

Mapping of aggregation and composition

Similarly to the transition from the analytical to the implementation model, the implementation of aggregation is exactly the same as the association. So, depending on the specific situation, we use relationships, possibly with intermediary tables (see previous blocks). For compositions, we can try to have a more sophisticated approach. Depending on the capabilities of the database management system we use:

- An appropriate view that more accurately describes our design needs.
- Specific integrity constraints and dedicated triggers. This allows us to carefully follow the principles governing the composition. By creating appropriate SQL expressions (and sometimes even PL-SQL), we are able to enforce, for example, cascading delete or block sharing of parts.

Another solution, but definitely worse, is the implementation of the features of the composition on the classical programming language side. In other words: the database treats the connection as a normal relationship, and the special behaviour is implemented in eg. Java or C++.

KNOWLEDGE CHECK — Which statements are true?

- Implementation of aggregation and composition in RM is based on similar assumptions as for Java.
- It is worthwhile to use built-in DB mechanisms such as triggers.

EXPLANATION & ANSWERS: **Explanation:**

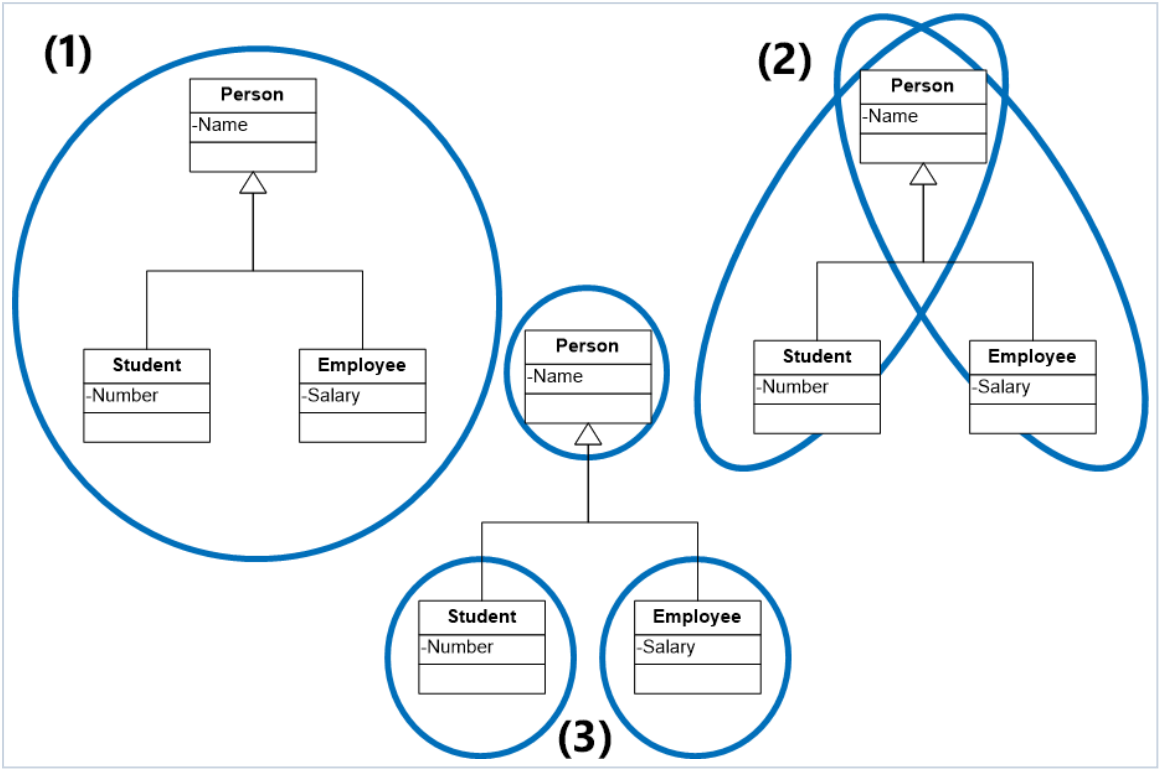
- As in the case of Java, the implementation of aggregation consists in providing a link (association) and a composition on the completion of specific rules.
- Triggers allow certain actions to be performed automatically by DB. They may, for example, be in line with the rules of composition.

Inheritance mapping

In traditional relational databases, inheritance does not exist as such. In some newer systems inheritance occurs, but mostly in the simplest form: disjoint and single. In the absence of it, you can try different workarounds. Let us emphasize once again that wherever a given construction takes place in its native form, it should be used. As the name implies, circumvention is not the solution. Therefore, if your database management system allows you to inherit, usually this should be used. In the remainder of this chapter we will deal with the situation when we do not have such an opportunity, and we must replace the inheritance somehow.

It can be said that the ways of circumventing inheritance in relational databases are similar to those used in transition from conceptual to implementation. In general, we include two solutions:

- Use relationships (associations) to describe relationships between tables (classes),
- Flattening hierarchy.



The above figure shows three approaches to implementing inheritance in a relational model. We can summarize them as:

1. One Table per Hierarchy (TPH). Complete flattening of the entire hierarchy and placement of all classes in one table. It will contain all the attributes from all classes in the inheritance tree. In addition, we add a column that specifies the type of the tuple. In other words, this discriminator informs us "to what class belongs this object". The undoubted advantage of this approach is the great simplicity of implementation. Unfortunately, some of the attributes will be not used. These features are very similar to those occurring when solving the overlapping inheritance problem.
2. Table Per Concrete Class (TPC). Partial flattening of the hierarchy and creation of two tables. The idea is to create one table for each bottom subclass. We include elements from the replaced subclass and all its superclasses. The advantage is the relative simplicity of the implementation. The drawback is that for each subclass we repeat invariants from the common superclass. In this particular case, the *Names*column will appear in each table. In addition, we have difficulty processing the `persons`ubclass as just people. A special SQL operator may be helpful: `UNION`.
3. Table Per Type (TPT). Create three tables and combine their relationships. Each class has its own table. Inheritance is replaced by aggregations, that is, in the relational model, relations between tables. We add the corresponding primary and foreign keys. The advantage of this solution is the full use of all the columns. Nothing is wasted, because in the table there are only elements belonging to a particular class. Usually there are no flawless solutions, so here we will find some. Particularly one: quite elaborate implementation and complicated management of such a structure, eg to find out the student's name, they need to read from the connected table *Person*. Of

course, in the case of a more advanced hierarchy, going through many tables/relationships makes life harder and decreases the performance (SQL `JOIN`).

It is clear from the above that there is no one ideal solution. Each of them has its advantages, such as ease of implementation at the expense of underutilized columns (approach #1), and defects, such as fairly difficult to use, at the cost of optimal resource management (solution #3). The concrete approach needs to be adapted to the individual situation.

KNOWLEDGE CHECK — Which statements are true?

- In the classical relational model there are all kinds of inheritance exceptoverlapping.
- One way to implement inheritance in RM is to flatten the hierarchy.

EXPLANATION & ANSWERS: **Explanation:**

- In the classical relational model, inheritance does not occur.
- Flattening the heirarchy is one way to accomplish any kind of inheritance.

Working with databases in Java

Each of the popular programming languages provides libraries for relational databases. For Java there are two general categories:

- JDBC (Java Database Connectivity). This is a standard interface supported by most databases available on the market. It enables all kinds of operations on a database system, including data retrieval and metadata, data updates, tables, relationships, etc. It was intended to isolate the functionality of a particular database from Java code. This way, our program should work in the same way, regardless of the database attached. Unfortunately, as in real life, the reality is not so perfect and sometimes after changing the DB engine you need to make some changes to the code of the program. Due to their versatility they are not always the best choice in terms of performance or ease of use.
- Native solutions for specific databases provided by their manufacturers. Because specific APIs are for a particular database management system, they are usually better optimized than JDBC. Therefore, if we do not care about cross-platform (DB), then it is worth checking to see if there are adequate programming libraries for our database.

As we mentioned, the most common way to connect to a Java database is to use the functionality provided by JDBC. Let's have a look at the code and comments below:

```java
// Attempting to load the driver
try
{
        Class.forName("sun.jdbc.odbc.JdbcOdbcDriver").newInstance();
}
catch (ClassNotFoundException ex)
{
        // Error
        System.err.println ("Exception: " + ex);
        System.exit(0);
}

Connection db_conn = DriverManager.getConnection ("jdbc:odbc:dbname", "user", "pass");

Statement db_statement = db_conn.createStatement();

db_statement.executeUpdate("create table employee { int id, char(50) name};");
db_statement.executeUpdate("insert into employee values (1, 'Jan Kowlski');");
db_connection.commit();

// [...]

// Execute query
ResultSet result = db_statement.executeQuery("select * from employee");

// Process the results
while (result.next() )
{
        System.out.println ("ID : " + result.getInt("id"));
        System.out.println ("Name : " + result.getString("name"));
}
```

- (4). It may seem that this code does nothing (because we do not use the created object). In fact it checks if a suitable JDBC driver is in your system. If not, then we will get an exception.
- (13). We open the connection to the database.
- (15). We are preparing an instance of an object for creating commands for BD.
- (17). We are sending a SQL command that creates a new table for the employees.
- (18). We send another command to the database. This time we put records into the table - we create employee information.
- (19). Calling this method will execute previously sent commands.
- (24). We send a query to the database, returning all our employees.
- (27). As a result, we got a `ResultSet` object, which is sort of a collection. It provides methods for navigating through individual records.
- (9). And here we come to the main part of working with data. It is also a specific example illustrating the impedance mismatch. Notice how we work with the records we receive. We retrieve the values of individual columns using the appropriate methods: different for text, numbers, and

other types of data. As a result, we have different values of type `String`, `int`, etc.. It would by much better to have real objects, e.g. `Person` class.

As you can see, the above method of working with data is not the most convenient, especially when compared with an object model. We will return to this problem in the next lesson.

KNOWLEDGE CHECK — Which statements are true?

- API libraries were designed exclusively for working with DB.
- JDBC technology makes it easy to write portable (between different BDs) applications.

EXPLANATION & ANSWERS: **Explanation:**

- API (Application Programming Interface) means any application programming interface, such as database, GUI, graphics.
- One of the basic assumptions of JDBC is to provide a unified way of working with any DB.

LESSON 12

Relational Model in OOP (Part 4)

Mapping inheritance strategies: Table per Hierarchy, Table per Concrete Class, Table per Subclass, and ORM frameworks (Hibernate / JPA).

Educational outcomes

Student:

- explains how to implement different concepts of objectivity in relational model using natural language (understanding);
- identifies and corrects errors in the source code, which is an implementation of the above structure;
- lists constructs implemented in the source code (understanding);
- implements the above structures using the selected programming language (application);
- indicates functional differences between two selected programs based on code snippets source (analysis);
- creates a program that implements specific business requirements using the constructs learned from the lectures (synthesis);

Justification

Modern programming languages are mostly based on the object paradigm. Therefore, its practical understanding is a necessary basis for working in this profession. Moreover, programmers often make systems based on analytical materials. And these largely use object-oriented graphical notations (such as UML). During this and the next lessons, you will deepen your theoretical bond, but first of all you will learn how to implement different concepts from object-orientness. This will allow you to design and implement modern business applications.

Object-relational mappers

I think we will agree that direct work with a relational database is not very convenient. The same opinion have creators of a number of additional libraries, whose task is to facilitate access to the relational database. One such library is Hibernate (http://www.hibernate.org/). This project is developed as open source since 2001. At the moment it is already mature and used by hundreds of thousands of developers all over the world. As a curiosity you can note that the core part of the source code is 76k lines, and each day it is downloaded 3000 times. There is also a version for Microsoft .NET. In the remainder of this chapter, we will take a brief look at this interesting technology.

The Hibernate library (and most other ORMs) does not completely eliminate the problem of impedance mismatch, but only reduces it. The programmer is constantly aware of the existence of a relational database and is partially limited by it. Nevertheless, it still greatly facilitates his/her work.

In every major project, and to such Hibernate is also used, the performance of the solution is crucial. If it is insufficient, even the most wonderful library will remain only a curiosity. Fortunately, its creators have applied different optimization techniques:

- Caching objects. It depends on "smart" access to objects. If it has not changed since the last use, it is not extracted from the database (which would take a while), but from a special structure existing in the programming language (eg Java or C#).
- Cache query results. A method similar to the above, but related to query results. If the requested data has not changed, the result is not retrieved from the database but from a special "container" stored on the programming language side.
- SQL statements are executed only when they are really needed, eg when "consuming" the result.
- Not updating unmodified objects. Hibernate has a special method for updating an object and sending it to the database. If the content of the object has not been changed, there is no need to send it to the database.
- Effective collection management.
- Merge multiple changes into one UPDATE. Use optimizations provided by native database language (eg SQL).
- Lazy initialization of objects and collections.

The big advantage of the above optimization is that they are transparent to the developer. This does not require any special action to use them.

KNOWLEDGE CHECK — Which statements are true?

- The main reason for using ORM tools is to improve the performance of BD access.
- ORMs, to some extent, suppress the impedance mismatch.

EXPLANATION & ANSWERS: **Explanation:**

- The basic premise of ORM is to make it easier (transparent) to work with any data source.
- ORMs do not completely eliminate impedance mismatches but significantly reduce them.

Microsoft Entity Framework i LINQ

While discussing the object-relational mappers, it is also worth mentioning LINQ technology (Language Integrated Query) developed by Anders Hejlsberg from Microsoft. The idea is to add to the existing programming language (eg MS C#) the syntax and functionality for writing queries similar to SQL. Emphasize: this is all part of the programming language (similar to other structures in C# like, for example Conditional statement `if` or `for` loop:

```C#
// A collection of customers from the city of London
var q = from c in db.Customers
        where c.City == "London"
        select c;

// For each customer we display the company name
foreach (Customer c in q)
        Console.WriteLine(c.CompanyName);

// Orders containing a specific code
var q = from o in db.Orders
        where o.ShipVia == 3
        select o;

// Browsing orders and executing C# methods
foreach (Order o in q) {
        if (o.Freight > 200)
                SendCustomerNotification(o.Customer);
        ProcessOrder(o);
}
```

This avoids sending commands (such as SQL) as expressions of type `string`. As a result, we got strong typological control (errors detected at compile stage, not execution), IDE hints, refactoring. What's more, there are LINQ versions that run on non-standard *LINQ to Objects* collections, but also on XML, JSON, and databases. Especially the latter approach in conjunction with ORM: Entity Framework is interesting to us (because it is similar to Hibernate). Interested parties are encouraged to read EF docs for this interesting technology.

KNOWLEDGE CHECK — Which statements are true?

- The biggest advantage of LINQ is the efficiency of accessing databases.
- One version of LINQ allows you to work with native .NET collections.

EXPLANATION & ANSWERS: **Explanation:**

- The main purpose of LINQ is not to improve performance, but to facilitate working with data sources (collections, DB, XML, JSON, etc.).
- Indeed, there is LINQ2Objects, created for working with native .NET collections.

Hibernate - introduction

Let's get back to Java and Hibernate. We will use the official tutorial, but with custom classes.

Database

Hibernate with Java works by default with a database via JDBC. As a database, we can use:

- Standard solutions, e.g. MariaDB, MySQL,
- Light systems written in Java, e.g. H2, HSQL or Apache Derby.

    - Usually you only need to use a small jar file containing all the necessary components;
    - For testing purposes, it is convenient to use the *in-memory* mode (data is not saved to a file and may be lost).

We will use the database H2 written entirely in Java. Thanks to this, we will avoid (sometimes) complicated installation and configuration. You can do it in several ways:

- "manually" i.e. downloading a distribution archive, unpacking it and adding it as a library to our IDE project;
- configure the IDE to download the appropriate library from the Maven repository, e.g. *com.h2database:h2:1.4.199* (this is how it is configured source files for this lecture).

We can run the database in two ways:

- classic in server mode - file *bin/h2.bat* or *bin/h2.sh* (available simple configuration console via browser);
- simplified mode - it will be automatically started by the Hibernate after proper configuration (see below). This requires adding the appropriate jar file to the project launch libraries in the IDE.

Configuration file

A single configuration file contains all settings related to Hibernate, including the database.

```xml
<!DOCTYPE hibernate-configuration PUBLIC
"-//Hibernate/Hibernate Configuration DTD 3.0//EN"
"http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">

<hibernate-configuration>
        <session-factory>
                <!-- Database connection settings -->
                <property name="connection.driver_class">org.h2.Driver</property>
                <property
name="connection.url">jdbc:h2:mem:db1;DB_CLOSE_DELAY=-1;MVCC=TRUE</property>
                <!--<property name="connection.url">jdbc:h2:~/db-test.h2</property>-->
                <!--<property name="connection.url">jdbc:h2:tcp://localhost/~/db-
test.h2</property>-->
                <property name="connection.username">sa</property>
                <property name="connection.password"></property>

                <!-- JDBC connection pool (use the built-in) -->
                <property name="connection.pool_size">1</property>

                <!-- SQL dialect -->
                <property name="dialect">org.hibernate.dialect.H2Dialect</property>

                <!-- Disable the second-level cache  -->
                <property
name="cache.provider_class">org.hibernate.cache.internal.NoCacheProvider</property>

                <!-- Echo all executed SQL to stdout -->
                <property name="show_sql">true</property>

                <!--Drop and re-create the database schema on startup-->
                <property name="hbm2ddl.auto">create</property>

                <!-- Enable Hibernate stats in the logs -->
                <property name="hibernate.generate_statistics">false</property>

                <!-- Full names the annotated entity class -->
                <mapping class="mt.mas.hibernate.Movie"/>
                <mapping class="mt.mas.hibernate.Actor"/>

        </session-factory>
</hibernate-configuration>
```

Depending on the type/mode of operation of the H2 database (note the selection of the appropriate driver in `connection.driver_class`), e.g.:

- *in-memory* mode. Convenient for testing (without preserving data permanently), the BD management system is started automatically.

```xml
<!-- Database connection settings -->
<property name="connection.driver_class">org.h2.Driver</property>
<property
name="connection.url">jdbc:h2:mem:db1;DB_CLOSE_DELAY=-1;MVCC=TRUE</property>
<property name="connection.username">sa</property>
<property name="connection.password"></property>
```

- file mode. Requires the server H2 to be started. You may also need to provide additional settings, e.g. a password.

```xml
<!-- Database connection settings -->
<property name="connection.driver_class">org.h2.Driver</property>
<property name="connection.url">jdbc:h2:~/db-test.h2</property>
<property name="connection.username">sa</property>
<property name="connection.password">sa</property>
```

Data mapping

Unfortunately, Hibernate does not work fully automatically (just like other ORMs) and you need to define the way the (relational) DB structure is mapped to object-oriented (Java). We have several options to choose from, among others:

- native Hibernate annotations,
- JPA annotations,
- external XML mapping file (for each Java class).

It seems that the best (especially for new projects) is to use annotations that comply with the JPA standard. Thanks to this, all information will be in one place (Java source code), and the possible ORM change should be quite painless.

Hibernate - class and attribute mapping

It is recommended to use the *JavaBean* convention, i.e. create dedicated `set/get` methods for changing and reading attribute values. You will also need a private, parameterless constructor.

In our example, we'll create a simple application that allows you to storing information about movies and combining movies with actors. The sample code for the `Movie` class is shown below. As you can see, we have information about the title of the movie (2) and its date (3). For each of the attributes we have created appropriate methods for changing its value and reading (so-called getters and setters). It is worth noting that this class does not have to inherit from any special superclass or implement dedicated interfaces.

```java
public class Movie {
        private String title;
        private LocalDate releaseDate;

        public Movie() {
        }

        public Movie(String title, LocalDate releaseDate, MovieCategory movieCategory) {
                this.releaseDate = releaseDate;
                this.title = title;
                this.movieCategory = movieCategory;
        }

        public LocalDate getReleaseDate() {
                return releaseDate;
        }

        public void setReleaseDate(LocalDate releaseDate) {
                this.releaseDate = releaseDate;
        }

        public String getTitle() {
                return title;
        }

        public void setTitle(String title) {
                this.title = title;
        }
}
```

Object ID

Each class that wants to use the full capabilities of the library must have a specific attribute to identify instances. It is usually private and numeric, e.g. `private long id`. Hibernate is responsible for changing its value.

As we can see in the list below, we have a dedicated attribute required by Hibernate, and used to store the object identifier. We also added the required annotations: `@Id`, `@GeneratedValue`, `@GenericGenerator`.

```java
@Entity(name = "Movie")
public class Movie {

        // [...]

        private long id;

        @Id
        @GeneratedValue(generator="increment")
        @GenericGenerator(name="increment", strategy = "increment")
        public long getId() {
                return id;
        }

        private void setId(long id) {
                this.id = id;
        }
}
```

Annotations can be applied to attributes or *getters* .

Simple attributes

As we mentioned earlier, we must provide information on which elements of the class are to be reflected in the relational model. The `@javax.persistence.Basic` annotation applies to:

- simple types,
- several others, including String and related dates.

It can be omitted because this is the default behavior of Hibernate.

In special situations, we can provide optional parameters:

- `optional`. Specifies the acceptability of nulls (`True`),
- `fetch`. The method of retrieving values (`Eager`).

Sometimes, the `@javax.persistence.Column` annotation may also be useful, which provides more customization options, e.g. column name in DB and an `@javax.persistence.Type` annotation allowing you to choose data type in DB.

Below is the code for the class describing the movie along with the required annotations to define the attributes to map.

```java
@Entity(name = "Movie")
public class Movie {

	// [...]

	private LocalDate releaseDate;
	private String title;

	@Basic
	public LocalDate getReleaseDate() {
		return releaseDate;
	}

	public void setReleaseDate(LocalDate releaseDate) {
		this.releaseDate = releaseDate;
	}

	@Basic
	public String getTitle() {
		return title;
	}

	public void setTitle(String title) {
		this.title = title;
	}
}
```

Derived attributes

Sometimes there is a need to use a derivative (computable) attribute. As we remember, it is usually not stored but calculated if necessary. In the case of Hibernate, we need to use the annotation `@javax.persistence.Transient`, which means ignoring the specified attribute or getter and the associated attribute. The following example shows this for attributes derived from `Age` and `Name` (implemented as *getters*).

```Java
@Entity(name = "Actor")
public class Actor {

	// [...]

	private String firstName;
	private String lastName;
	private LocalDate birthDate;

	@Transient
	public String getName() {
		return getFirstName() + " " + getLastName();
	}

	@Transient
	public int getAge() {
		return Period.between(getBirthDate(), LocalDate.now()).getYears();
	}

	@Basic
	public String getFirstName() {
		return firstName;
	}

	public void setFirstName(String firstName) {
		this.firstName = firstName;
	}

	@Basic
	public String getLastName() {
		return lastName;
	}

	public void setLastName(String lastName) {
		this.lastName = lastName;
	}

	@Basic
	public LocalDate getBirthDate() {
		return birthDate;
	}

	public void setBirthDate(LocalDate birthDate) {
		this.birthDate = birthDate;
	}
}
```

The *enum* attributes

In situations where we need to distinguish certain things in the application, it is worth to use the enum structure. It can also be mapped in Hibernate using the `@javax.persistence.Enumerated` annotation.

We can also define how these values will be remembered in the database:

- `EnumType.ORDINAL`. Uses a numeric value
- `EnumType.STRING`. Uses the enum's name.

Below is a code that adds movie categories made using a defined enum.

```Java
@Entity(name = "Movie")
public class Movie {

	// [...]

	public enum MovieCategory {Unknown, Comedy, SciFi}

	@Enumerated
	public MovieCategory getMovieCategory() {
		return movieCategory;
	}

	public void setMovieCategory(MovieCategory movieCategory) {
		this.movieCategory = movieCategory;
	}

	@Override
	public String toString() {
		var sb = new StringBuilder();

		sb.append(String.format("Movie: %s released on %s as %s (#%s @%s)",
getTitle(), getReleaseDate(), getMovieCategory(), getId(), super.hashCode()));

		return sb.toString();
	}
}
```

Note also the override of the `toString()` method, which allows displaying information about the mapped object.

Working with Hibernate

Working with the Hibernate library is quite easy, but you need to follow a few rules and you should use the auxiliary classes.

Sessions in Hibernate

Working with a database takes place in the so-called sessions in which individual operations (or sets of them) are wrapped in transactions. This is quite a standard way of working with a DB. The following is an example of how to do it:

```Java

StandardServiceRegistry registry = null;
SessionFactory sessionFactory = null;

try {
        registry = new StandardServiceRegistryBuilder()
                        .configure() // configures settings from hibernate.cfg.xml
                        .build();
        sessionFactory = new
MetadataSources(registry).buildMetadata().buildSessionFactory();

        Session session = sessionFactory.openSession();
        session.beginTransaction();

        // Do something within the session, e.g. create/retrieve objects, etc.

        session.getTransaction().commit();
        session.close();
}
catch (Exception e) {
        e.printStackTrace();
        StandardServiceRegistryBuilder.destroy( registry );
}
finally {
        if (sessionFactory != null) {
                sessionFactory.close();
                sessionFactory = null;
        }
}
```

- Create a `registry` .
- Create a `session factory` .
- Open the session.
- Start the transaction.
- Perform operations, e.g. creating, updating, deleting data.
- Commit the transaction and close the session.

Identity of objects

Hibernate ensures that the same object (primary key) retrieved from the database in a single session will have the same instance in the Java environment. Sometimes it may be helpful to implement your own `equals()` and `hashCode()` methods with the annotation attribute: `@NaturalId`. More in the Hibernate documentation.

Creating and saving objects

Time to check the library in action. Let's just remember about the proper placement of the XML configuration file. Let's look at the code below and comments:

```Java

var movie1 = new Movie("Terminator 1", LocalDate.of(1984, 10,26),
Movie.MovieCategory.SciFi);
var movie2 = new Movie("Terminator 3", LocalDate.of(2003, 8,8), Movie.MovieCategory.SciFi);

System.out.println(movie1);
System.out.println(movie2);

Session session = sessionFactory.openSession();
session.beginTransaction();
session.save(movie1);
session.save(movie2);
session.getTransaction().commit();
session.close();
```

As you can see, we create objects with the standard new (1) operator, save in the session (9) and confirm the transaction (11). From that moment they are already in the database and we can use them normally.

Retrieving data

In order to retrieve data using Hibernate, we can use the SQL-like query language. As shown in the list below, we work on complete objects (class `Movie`) and not (as in the case of "pure" JDBC) on plain values (eg `String`) from the database. Of course, the objects contain the correct primary keys.

```Java

System.out.println("\nMovies from the db:");

session = sessionFactory.openSession();
session.beginTransaction();
List<Movie> moviesFromDb = session.createQuery("from Movie").list();
for ( var movie : moviesFromDb) {
        System.out.println(movie);
}
session.getTransaction().commit();
session.close();
```

The log file

The activities of Hibernate can also be seen in the *log* file. Thanks to this, we can follow the actions performed by the library, which is an invaluable help in correcting errors. Also notice that we can preview SQL commands sent to DB (lines: 8, 10, 17-19, 22):

```
Plain Text

1:01:10 PM org.hibernate.Version logVersion INFO: HHH000412: Hibernate Core {5.4.1.Final}
1:01:11 PM
org.hibernate.engine.jdbc.connections.internal.DriverManagerConnectionProviderImpl
configure WARN: HHH10001002: Using Hibernate built-in connection pool (not for production
use!)
1:01:11 PM
org.hibernate.engine.jdbc.connections.internal.DriverManagerConnectionProviderImpl
buildCreator INFO: HHH10001005: using driver [org.h2.Driver] at URL
[jdbc:h2:mem:db1;DB_CLOSE_DELAY=-1;MVCC=TRUE]
1:01:11 PM
org.hibernate.engine.jdbc.connections.internal.DriverManagerConnectionProviderImpl
buildCreator INFO: HHH10001001: Connection properties: {password=****, user=sa}
1:01:11 PM
org.hibernate.engine.jdbc.connections.internal.DriverManagerConnectionProviderImpl
buildCreator INFO: HHH10001003: Autocommit mode: false
1:01:11 PM
org.hibernate.engine.jdbc.connections.internal.DriverManagerConnectionProviderImpl$PooledConn
<init> INFO: HHH000115: Hibernate connection pool size: 1 (min=1)
1:01:11 PM
org.hibernate.resource.transaction.backend.jdbc.internal.DdlTransactionIsolatorNonJtaImpl
getIsolatedConnection INFO: HHH10001501: Connection obtained from JdbcConnectionAccess
[org.hibernate.engine.jdbc.env.internal.JdbcEnvironmentInitiator$ConnectionProviderJdbcConnec
for (non-JTA) DDL execution was not in auto-commit mode; the Connection 'local transaction'
will be committed and the Connection will be set into auto-commit mode.
Hibernate: drop table Movie if exists
1:01:11 PM
org.hibernate.resource.transaction.backend.jdbc.internal.DdlTransactionIsolatorNonJtaImpl
getIsolatedConnection INFO: HHH10001501: Connection obtained from JdbcConnectionAccess
[org.hibernate.engine.jdbc.env.internal.JdbcEnvironmentInitiator$ConnectionProviderJdbcConnec
for (non-JTA) DDL execution was not in auto-commit mode; the Connection 'local transaction'
will be committed and the Connection will be set into auto-commit mode.
Hibernate: create table Movie (id bigint not null, movieCategory integer, releaseDate date,
title varchar(255), primary key (id))
1:01:11 PM org.hibernate.engine.transaction.jta.platform.internal.JtaPlatformInitiator
initiateService INFO: HHH000490: Using JtaPlatform implementation:
[org.hibernate.engine.transaction.jta.platform.internal.NoJtaPlatform]

Created movies:
Movie: Terminator 1 released on 1984-10-26 as SciFi (#0 @1499588909)
Movie: Terminator 3 released on 2003-08-08 as SciFi (#0 @1339052072)

Hibernate: select max(id) from Movie
Hibernate: insert into Movie (movieCategory, releaseDate, title, id) values (?, ?, ?, ?)
Hibernate: insert into Movie (movieCategory, releaseDate, title, id) values (?, ?, ?, ?)

Movies from the db:
Hibernate: select movie0_.id as id1_1_, movie0_.movieCategory as movieCat2_1_,
movie0_.releaseDate as releaseD3_1_, movie0_.title as title4_1_ from Movie movie0_
Movie: Terminator 1 released on 1984-10-26 as SciFi (#1 @1989924937)
Movie: Terminator 3 released on 2003-08-08 as SciFi (#2 @1842571958)
1:01:12 PM
org.hibernate.engine.jdbc.connections.internal.DriverManagerConnectionProviderImpl$PoolState
```

```
stop
INFO: HHH10001008: Cleaning up connection pool
[jdbc:h2:mem:db1;DB_CLOSE_DELAY=-1;MVCC=TRUE]
```

Database scheme

Of course, Hibernate automatically creates an appropriate relational data model in DB. After "generating" data (stable version of the data model), the fragment of the file *hibernate.cfg.xml* should be commented or removed (otherwise the schema will be updated and existing data will be deleted after the program starts).

```XML

<!-- Drop and re-create the database schema on startup -->
<!-- <property name="hbm2ddl.auto">create</property> -->
```

Hibernate - mapping directed associations

So far, we only dealt with a very simple class that was not connected to other classes. Of course, in real projects of this type, the situation is extremely rare. Fortunately, Hibernate also enables us to work with associations that are (almost) automatically mapped to database relationships. Association elements that need to be considered are:

- direction
- cardinalities
- behavior of the collection (on the Java side).

To extend our example, we will create an `Actor` and associate it with movies (`Movie`). The presented example code is quite simple and does not require any special comments. Just note that we have placed the private ID required by Hibernate. We also need to make a new entry in the main configuration file (equivalent to `<mapping class="mt.mas.hibernate.Actor"/>`)

Java

```
@Entity(name = "Actor")
public class Actor {
	private long id;
	private String firstName;
	private String lastName;
	private LocalDate birthDate;

	/**
		* Required by Hibernate.
		*/
	public Actor() {
	}

	public Actor(String firstName, String lastName, LocalDate birthDate) {
		this.firstName = firstName;
		this.lastName = lastName;
		this.birthDate = birthDate;
	}

	@Id
	@GeneratedValue(generator="increment")
	@GenericGenerator(name="increment", strategy = "increment")
	public long getId() {
		return id;
	}

	private void setId(long id) {
		this.id = id;
	}

	@Basic
	public String getFirstName() {
		return firstName;
	}

	public void setFirstName(String firstName) {
		this.firstName = firstName;
	}

	@Basic
	public String getLastName() {
		return lastName;
	}

	public void setLastName(String lastName) {
		this.lastName = lastName;
	}

	@Basic
	public LocalDate getBirthDate() {
		return birthDate;
	}
```

```
        public void setBirthDate(LocalDate birthDate) {
                this.birthDate = birthDate;
        }

        @Transient
        public String getName() {
                return getFirstName() + " " + getLastName();
        }

        @Transient
        public int getAge() {
                return Period.between(getBirthDate(), LocalDate.now()).getYears();
        }
}
```

For the `Movie` class we will add information about the actors playing in it (`actors`). We use a `List` container (other types are also supported). We will also use the `@OneToMany` annotation. The relevant code snippets is below:

Java

```
@Entity(name = "Movie")
public class Movie {

        // [...]

        private List<Actor> actors = new ArrayList<>();

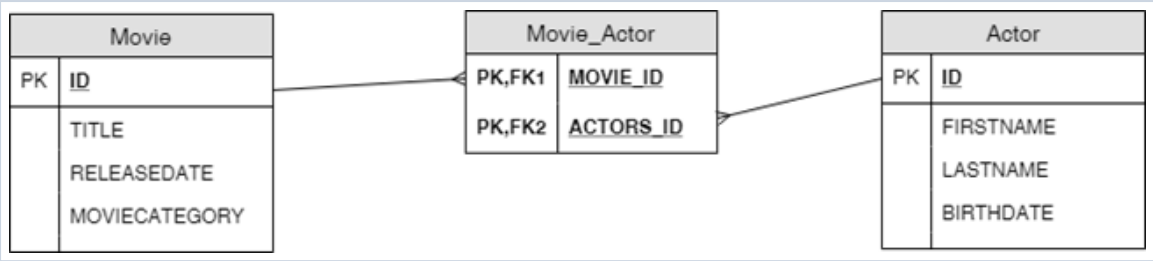
        @OneToMany(mappedBy = "movie", cascade = CascadeType.ALL, orphanRemoval = true)
        private List<Actor> getActors() {
                return actors;
        }

        private void setActors(List<Actor> actors) {
                this.actors = actors;
        }

        public void addActor(Actor actor) {
                getActors().add(actor);
        }

        public void removeActor(Actor actor) {
                getActors().remove(actor);
        }
}
```

As a result of these operations, Hibernate generated the following database schema. Note that we have also received an intermediary table, which is not needed for 1: *, but it will be useful later.



At the Java language level, there is a connection (Java references) from the `Movie` to `Actor` (but note: not the other way). We add the links by modifying the Java container. Hibernate automatically detects this and updates the database upon saving. Similar "automation" exists for attributes.

And yet our new association in action: we create movies (7) and actors (13). Next, we add the actors to the movies (21), save to the session (27) and commit the transaction (32). Then we retrieve the objects.

Java

```
registry = new StandardServiceRegistryBuilder()
                .configure() // configures settings from hibernate.cfg.xml
                .build();
sessionFactory = new MetadataSources(registry).buildMetadata().buildSessionFactory();

System.out.println("Created movies:");
var movie1 = new Movie("Terminator 1", LocalDate.of(1984, 10,26),
Movie.MovieCategory.SciFi);
var movie2 = new Movie("Terminator 3", LocalDate.of(2003, 8,8), Movie.MovieCategory.SciFi);

System.out.println(movie1);
System.out.println(movie2);
System.out.println("Created actors:");
var actor1 = new Actor("Arnold", "Schwarzenegger", LocalDate.of(1947, 7, 30));
var actor2 = new Actor("Claire", "Danes", LocalDate.of(1979, 4, 12));
var actor3 = new Actor("Kristanna", "Loken", LocalDate.of(1979, 10, 8));

System.out.println(actor1);
System.out.println(actor2);
System.out.println(actor3);

movie2.addActor(actor1);
movie2.addActor(actor2);
movie2.addActor(actor3);

Session session = sessionFactory.openSession();
session.beginTransaction();
session.save(movie1);
session.save(movie2);
session.save(actor1);
session.save(actor2);
session.save(actor3);
session.getTransaction().commit();
session.close();

System.out.println("\nMovies and actors from the db:");

session = sessionFactory.openSession();
session.beginTransaction();

List<Movie> moviesFromDb = session.createQuery( "from Movie" ).list();
for ( var movie : moviesFromDb) {
        System.out.println(movie);
}

session.getTransaction().commit();
session.close();
```

KNOWLEDGE CHECK — Which statements are true?

- The@OneToManyannotation is used to map the association.
- When working with Hibernate, you need to create and update the DB schema manually.

EXPLANATION & ANSWERS: **Explanation:**

- Indeed, the@OneToManyannotation is used to map the association.
- Fortunately, Hibernate manages the database schema itself by creating the appropriate tables and connections.

Hibernate - mapping bidirectional associations

So far, we have used directed association. Now it is the time to use two-way association. Thanks to this we will be able to navigate in both directions. To do this, we add a single `Movie` reference to the `Actor` class and the required `@ManyToOne` annotation. One should also implement dedicated logic in the methods that create the connections (examples in previous lectures).

```Java

@Entity(name = "Actor")
public class Actor {

	// [...]

	private Movie movie;

	@ManyToOne
	public Movie getMovie() {
		return movie;
	}

	public void setMovie(Movie movie) {
		this.movie = movie;
	}
```

Similarly, we map other cardinalities using `@ManyToMany` and `@OneToOne` annotations.

It is worth noting that despite the changes in the configuration files and the source code, the relational scheme does not change.



COMMENTARY

Remember that in all places of the program, where we need to use navigation along the association, use these two-way constructions. They are much clearer and often more efficient than manually written queries in SQL or HQL.

Of course, this also applies to MAS final projects.

KNOWLEDGE CHECK — Which statements are true?

- Transforming a directed association into a bidirectional one requires, among other things, adding dedicated constructs on the Java side.
- The data schema for the directed and bidirectional association is identical.

EXPLANATION & ANSWERS: **Explanation:**

- In order to convert the association to bidirectional, we actually need to add the appropriate collection on the Java side.
- Indeed, the data scheme for directed and bidirectional association is identical.

Hibernate - multi-value attributes

Hibernate also has the option of using multi-value attributes. They are called *collection of values*. The difference with the association is that the values are not shared (and the objects, of course, are shared). They must be declared using an interface, not a specific implementation. The behavior of this attribute depends on the interface type (e.g., `List`, `Set`).

For the `Actor` class we will add a list of web addresses. For this purpose, we will use the annotation `@javax.persistence.ElementCollection`. It means that the collection does not contain references to other instances, but a list of items such as `String`.

We've included the code below.

```Java

@Entity(name = "Actor")
public class Actor {

	// [...]

	private List<String> urls;

	@ElementCollection
	public List<String> getUrls() {
		return urls;
	}

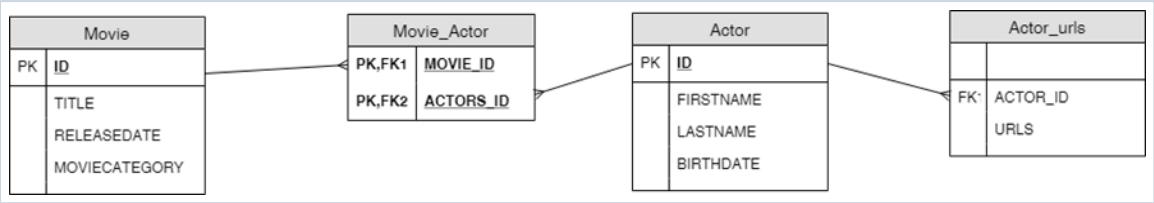
	public void setUrls(List<String> urls) {
		this.urls = urls;
	}
}
```

Sample utilization:

```Java

actor1.setUrls(List.of("http://www.schwarzenegger.com/",
		"https://pl.pinterest.com/schwarzenegger/",
		"https://www.facebook.com/arnold"));
```

Of course, the database schema has also changed (automatically generated by Hibernate). Now it looks like this:



KNOWLEDGE CHECK — Which statements are true?

- Implementing multi-valued attributes with Hibernate is no different than the implementation of the association.
- The@ElementCollectionannotation is used for the implementation of multi-valued attributes.

EXPLANATION & ANSWERS: Explanation:

- The Hibernate library provides a special tag (element) to implement multi-valued attributes.
- The@ElementCollectionannotation is used for the implementation of multi-valued attributes.

Summary

The impedance mismatch (difficulty in mapping the relational and the object world), is a very serious, practical problem. There are two general approaches to its solution:

- Modify the programming language so that it has as many advantages of databases as possible. An example might be Microsoft C# along with LINQ. Query language (functionally similar to SQL) has become part of the programming language. The impedance mismatch disappears completely, so we do not need to map object constructs to relational and vice versa.
- Creation of additional libraries facilitating data work (eg Hibernate for Java and MS .NET). The library actually makes it easy to work with databases. Unfortunately, in some situations, instead of native references use identifiers.

It seems that a far better approach is to add database features to the programming language, as in Microsoft LINQ. This, however, requires the involvement of language developers and can not be done solely by creating additional libraries.

Source codes



Download all sources for MAS lectures.

LESSON 13

GUI Usability & Testing

Principles of user interface usability, Jakob Nielsen heuristics, cognitive load, user testing, and evaluation metrics.

Educational outcomes

Student:

- explains how to take into account the principles of usability through natural language (understanding);
- identifies and corrects errors in projects that violate usability rules (understanding);
- designs user interfaces according to known rules (application);

Justification

Modern programming languages are mostly based on the object paradigm. Therefore, its practical understanding is a necessary basis for working in this profession. Moreover, programmers often make systems based on analytical materials. And these largely use object-oriented graphical notations (such as UML). During this and the next lessons, you will deepen your theoretical bond, but first of all you will learn how to implement different concepts from object-orientness. This will allow you to design and implement modern business applications.

Graphical User Interfaces

Let's start with the question: what is (or was actually) the most underrated component of the software produced? This is probably the user interface, which for most modern software means: Graphical User Interface (GUI). Why is it like that? Why do programmers mostly focus on business application functionality by treating the GUI as a necessary evil? It is even more weird because users evaluate the application just through its interface, especially nowadays when programs with similar functionality are very common. Which one would you choose if their features are very similar? Probably the one that will be more suited to the price and the one that is easier to use. Fortunately, for the past few years, there has been an increase in the importance of application interfaces.

Graphical user interfaces are a cross-cutting topic in computer science (eg programming), psychology, ergonomics, and probably a few areas we did not mention here. Due to its interdisciplinarity, this subject is quite complex, and worst of all, sometimes difficult to measure. We mean here that it is not always easy to judge which solution is better. Fortunately, for some time, in computer science, and more specifically in software engineering, there is the usability concept.

There are many web portals devoted to this area. The most interesting ones would be http://www.uzytecznosc.pl and http://www.useit.com . Therefore, we will use some of the information contained there.

KNOWLEDGE CHECK — Which statements are true?

- The most important in the application is functionality, and the interface is a minor add-on.
- It is very easy to measure the quality of the interface.

EXPLANATION & ANSWERS: **Explanation:**

- Of course, creating an application should pay attention to both the functionality and the interface to support it.
- Interface evaluation is dependent on many factors, including tastes, habits. Therefore, it is quite difficult to carry out.

Usability

Generally speaking, usability is the ability to meet user needs by:

- a device
- an application
- an interface.

It concerns the combination of factors shaping the user's experience of working with the product. These include:

- Practical usefulness. Does the system perform tasks that meet the needs of the user?
- Easy to learn and operate. How fast can you learn to use the system? Is it easy for most people to operate the system?
- Effectiveness. Will the system ensure the outcome of the task as expected by the user?
- Efficiency. Is the desired result attainable with moderate effort from the user?
- Satisfaction. Does the user like to work with the system and recommend it to others?

As we can see, these factors cover many different aspects, ranging from utilitarian (practical), educational, to subjective (such as satisfaction).

KNOWLEDGE CHECK — Which statements are true?

- Usability of interfaces is an interdisciplinary issue.
- One of the most important aspects of usability is the practical usefulness of the thing/service.

EXPLANATION & ANSWERS: **Explanation:**

- Interdisciplinary is a term that concerns several disciplines. Of course the GUI belongs to this category.
- Practical application is an important assessment factor.

Shaping usability

How can we shape usability? In general, we can say that the key is cooperation with users. During the entire development process we must have good contact with the people who will use our software. For this purpose it is worth:

- Conduct frequent user feedback surveys while working on the project

- Test prototypes (because they will be more then one) with target users - see below
- Observe the way users work with the system under real conditions
- Use survey techniques: interviews and questionnaires.

KNOWLEDGE CHECK — Which statements are true?

- One of the basic elements facilitating shaping usability is frequent contacts with users.
- Usability can be shaped in complete isolation from the users.

EXPLANATION & ANSWERS: **Explanation:**

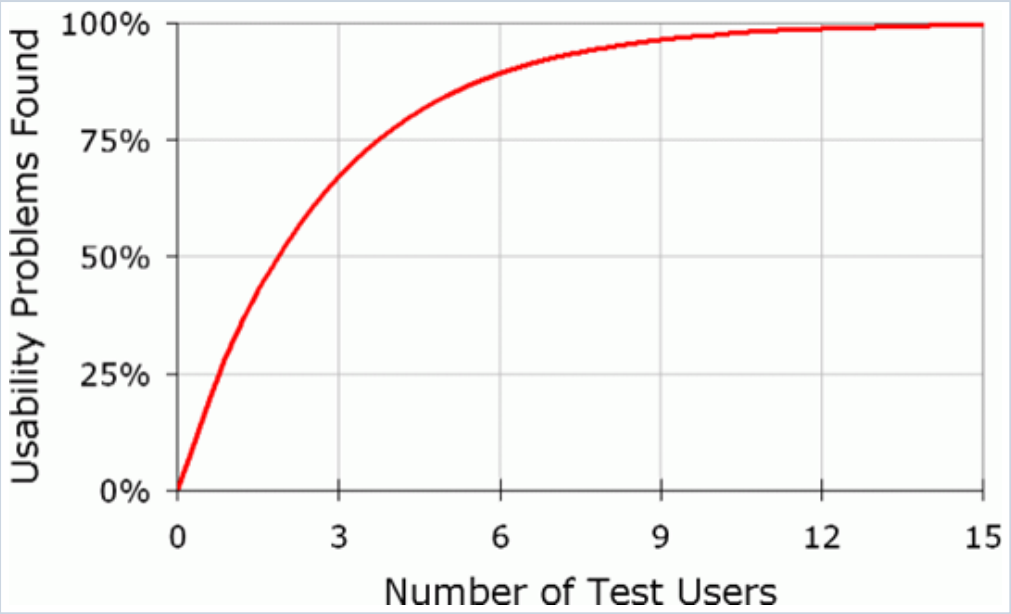
- Contacts with your users let you know their GUI feedback and, accordingly, make corrections.
- It is difficult to shape product usability (GUI) without knowing user feedback.

Usability tests

Once we have a piece of the testable item (usually some version of the application, but may be a paper GUI design as well), we need to perform a usability test. How to do it? Follow these simple rules:

- Testers must be representative. In other words, if we are doing an application for lawyers who are not computer experts, then we need to form such a test group. Usually, we pay for these types of tests (in the end, people do some work for us).
- Tests consist of performing specific (typical) tasks, eg for a text editor it may be writing a specificaly formatted letter.
- The essence of these tests is watching users. We evaluate, among others, the following aspects:
    - What users do?
    - Where are the difficulties?
- The whole test procedure should be recorded. We can use a video camera and/or special programs. Usually it is best to use both methods because simply recording the content of the screen will not provide us with full information on the reactions of users.
- To get as close as possible to the real working conditions, testers should be left alone. Therefore, during the entire test, it is strictly forbidden to give advice, help, etc.

Fortunately for us, in this type of test, unlike in statistical surveys, it is usually not required a large numbers of people (see the below chart). Typically, a group of 5, 10 people are able to capture the main problems of usability. People in more groups usually identify the same problems. Therefore, it is better to iteratively improve the project and test it in small groups (eg 10 times in 10 people) than do one test per 100 people.



The number of users and a factor of identified usability problems. Source: Jakob Nielsen, Why You Only Need to Test with 5 Users

You can also risk saying that it is more important to observe what the testers do than to listen to their later comment (which of course can also be useful). The reason for this is that the "current" reactions of people are more spontaneous. Later their opinions may change, for example due to the fact that we pay them (and they do not want to talk bad about our product).

KNOWLEDGE CHECK — Which statements are true?

- You do not have to watch your users during the usability tests - just listen to their comments.
- It is worth doing usability tests for very large user groups.

EXPLANATION & ANSWERS: **Explanation:**

- User comments may be misleading. Therefore, it is important to observe their behavior.
- Large user groups repeatedly report the same errors. For this reason, it is better to conduct more frequent tests with smaller teams.

Usability and business

As you can see from the above description, usability tests can be quite expensive. With a professional approach you need to have a special laboratory equipped with computer desks, a Venetian mirror, cameras and specialized software. Is it worth it? Let's get back to what we said earlier. In today's world many applications have very similar functionality. In that case, what criteria do the customers follow? They certainly take into account the price, but also ease of use of the system. And this comfort translates into satisfaction with the use of the application. And that, hopefully, will translate into our profits. For couple of years, we can observe a new trend: updated versions of applications do not focus only on adding new features, but facilitating access to existing ones, such as Microsoft Office 2007. You can discuss whether this new interface appeals to you or not. But the fact is: the main reason given by Microsoft to buy a new version of MS Office is a new, more useful interface.

Are there any ways to reduce usability costs? Of course! Before we start a new project, let's test its previous version (when we had one). Identify positive and negative elements. Let's strength the first and eliminate the latter. It is also worth to look at the solutions of competition (because usually we have one). In our work we should use (many) prototypes. They may also be "paper", but in the age of RAD tools it is not a problem to quickly produce a "fake" system prototype. In the case of iteration work each of them should end with tests. And do not forget to check the project in the context of known usability recommendations (see below).

And what are the benefits of high usability? We can certainly include:

- Shorter time to complete a task
- Reduce the number of mistakes made by the user
- Shorter learning time and less effort needed to master the product
- Confidence in the product and its willingness to further develop
- Persistent product satisfaction and the intention to recommend it to other buyers

Let's summarize optimistically: good usability just pays off!

KNOWLEDGE CHECK — Which statements are true?

- The usability level presented by the application has no impact on user productivity.
- Before you start your own design work, you should test your competitor solutions.

EXPLANATION & ANSWERS: **Explanation:**

- Of course, the usability of the interface has a huge impact on users.
- By testing competitive solutions, we can avoid making mistakes.

GUI - good advices

Basic and general GUI guidelines can be summarized as follows:

- Consistency. The entire application should look and be used in the same way.
- Transparency and intuitiveness. In most cases, this means referring to existing GUI habits.
- Follow the guidelines for a specific platform. Every platform (such as Windows, Linux, Mac) has its own rules for building applications. In general, they are similar, but may vary in detail, such as treating application windows or using context menus. For this reason, they should be taken into account. Otherwise, the user will be annoyed that the application is not working properly. It seems that this principle must be applied especially in business applications. While the program is more "entertaining" (eg game), the greater the deviation can be made (up to a completely own GUI).
- Proper management of the screen space. Usually, this is down to the proper scaling of windows. Naturally, this does not always make sense, for example, when we only provide a few short information (eg login window), there is no need to resize the window. Nevertheless, this is a very important aspect of GUI quality evaluation, but sometimes it is quite difficult to implement.
- Smooth work. Users do not like when the computer does not respond to their commands. This may happen when it is busy doing complex calculations or for example waiting for a response from the database. In such situations, it is useful to inform the user about the progress of the operation (eg, current percentage, remaining time) or even to display a simple animated image of the occupancy.

KNOWLEDGE CHECK — Which statements are true?

- Instead of creating a consistent GUI design, it is better to include a comprehensive manual.
- Modern monitors are large, so you do not have to worry about proper window space management.

EXPLANATION & ANSWERS: **Explanation:**

- Users do not like to read manuals. Therefore, let's try to design a GUI so that it is not needed.
- Precisely because of the varying screen sizes, scaling techniques should be considered.

GUI Checklist

Below is a GUI checklist (partially taken from http://www.uzytecznosc.pl) broken down into categories. Follow it during designing of your interafces, and when the implementation (or design) is complete, recheck that all elements of our interface are compatible with it. Some of these recommendations may seem obvious and trivial, but unfortunately you may come across solutions that break many of these rules.

Functional requirements

- Are you paying close attention to the functionality and appearance of the application?
- Is your window 100% consistent? Does it correspond to the design?
- Does it have all the designed functions?
- Does it have only designed functions? Do not place any additional (not included in the project) functionality.
- Do you help the user avoid common mistakes by skilfully designing the window? Is wrong data being handled properly?
- Is your window compatible with other windows of the same application (layout, controls, conventions)?

Platform Requirements

Is your project fully compliant with the interface/operating system guidelines?

- Uses appropriate controls, fonts, colors
- Proper window dimensions (including minimum)
- Correctly aligned elements
- Behavior of windows and controls

Deviations from the above recommendations may indicate a faulty GUI operation and/or lack of "system" modifications. As a result, for example, it may not work to scale the font size or change the graphic theme.

Requirements for windows

- Does your window have a title? Windows without or with a default title are difficult to identify, eg in the taskbar.
- Is this title the same or similar to the button name or option from the menu that opened the window?

- Are there buttons in the title bar (including minimize, maximize, restore, help)? If so, do they work properly?
- Is it possible to change the size (maximize, minimize) the window? If yes, is this justified? Is the entire contents of the window properly scaled?
- Is there a contextual help button?
- Have you selected the right frame type for your window (eg dialog)?
- Is your window properly placed in the hierarchy?
- Do you use modality (blocking interaction with other windows programs) where it is required?

Dialog window requirements

- Is there a default button in your dialog? Is it marked?
- Can your window be closed in the standard way? E.g.:
    - button in the title bar
    - Esc key
    - "Cancel" button.
- Is it possible to close the window or select "Cancel" without making changes?
- Are the "OK" buttons, "Cancel" and other related window manipulation separated from other buttons?

Remember that the user always has an emergency exit from any, even the strangest situation. In other words, he/she does not be afraid to work with the program.

Widget requirements

- Do you use the intuitive layout of the controls and choose their types accordingly, instead of artificially describing activities and creating instructions?
- Are the controls positioned according to the way the user reads or scans the screen?
- Are the most commonly used options placed "at the earliest"?
- Is the number of controls in the window not too large? If so, is it better to delete some less frequently used or to show them only after pressing the corresponding button?
- Are you using the controls according to their purpose as described in the guidelines and used in popular applications? Eg.
    - exclusive options: *radio buttons*
    - multi options: *check boxes*.
    - lists with multiple choice.
- Are your controls responding in a standard way? e.g.:
    - press the buttons to execute the command or open the window
    - radio buttons or check boxes should only change status.
- Are semantically-related controls grouped using frames or related interface elements?
- Are you not using frames excessively to group only individual controls?
- Does the software properly enable or disable inactive controls when needed?

Menu Requirements

- Is there a default option in the menu (and in the *combo boxes*)?
- Is the opening list large enough (vertically) to save user scrolling?
- Is the opening list wide enough so that each item is visible in full?

Label requirements

- Are there appropriate labels/descriptions (*labels*) that describe their purpose or content (especially for lists)?
- Are all the displayed data described?
- Are the descriptions aligned equally (in relation to the text of the control and to other descriptions)?
- Are colons in descriptions are consistently applied or not used?
- Do you use tooltips to explain some of the interface elements? Remember that the tooltips that repeat the standard description of the control do not make much sense.
- Do you use national (e.g. Polish) letters? Descriptions and names of controls without them look sloppy and less readable.

Keyboard requirements

- Are there any shortcuts in the window that allow advanced users to take advantage of some of the options?
- Can the user navigate the cursor using the Tab and Shift-Tab keys?
- Is the order of movement consistent with the layout of the box?
- Can you reach all the controls in this way?

KNOWLEDGE CHECK — Which statements are true?

- Checklists should be included with each interface project.
- All platforms have identical interface requirements.

EXPLANATION & ANSWERS: **Explanation:**

- Checklists allow you to avoid common mistakes and therefore should be used when designing.
- Each platform has its own interface requirements. Their inclusion will make it easier for users to use applications.

LESSON 14

GUI Architecture & Design

Designing desktop and web graphical interfaces, layout managers, declarative UI, MVC/MVP patterns, and dialog flow.

Educational outcomes

Student:

- explains how to design and implement graphical interfaces using natural language (understanding);
- identifies and corrects errors in the source code that implements the above interfaces (understanding).;
- lists the constructs implemented in the source code (understanding);
- implements the aforementioned constructs using the selected programming language (application);
- indicates differences in functionality between two selected programs based on fragments of source code (analysis);
- creates a program that implements specific business requirements using the structures learned in the lectures (synthesis);

Justification

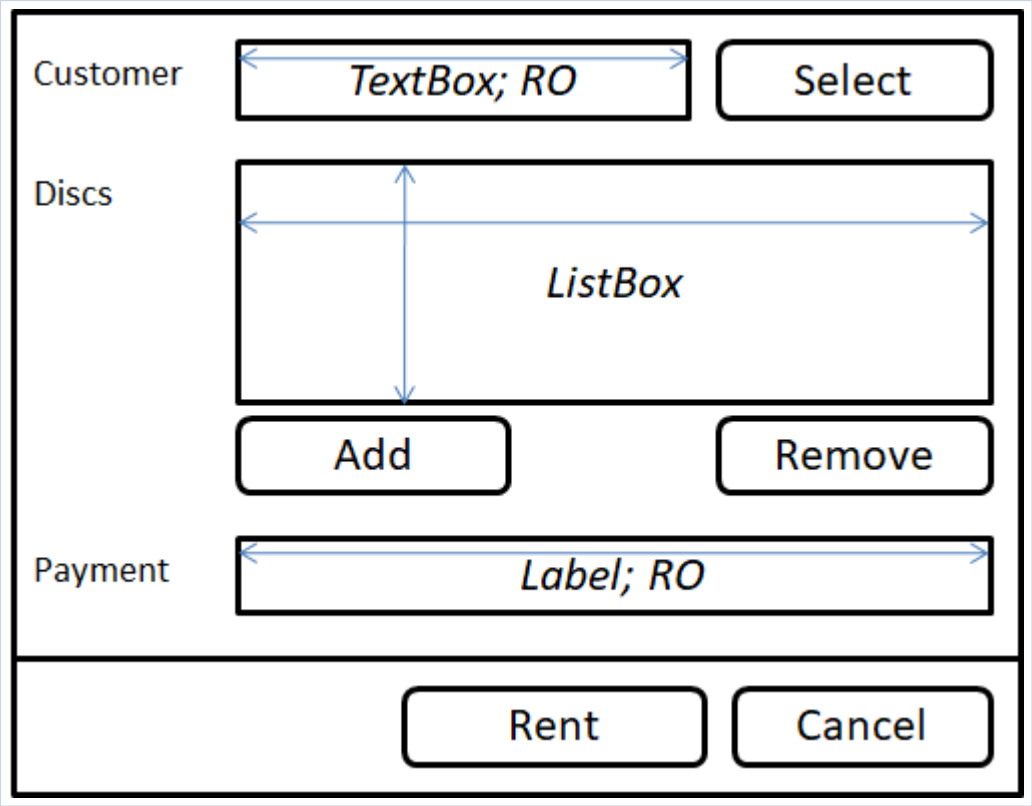
Modern programming languages are mostly based on the object paradigm. Therefore, its practical understanding is a necessary basis for working in this profession. Moreover, programmers often make systems based on analytical materials. And these largely use object-oriented graphical notations (such as UML). During this and the next lessons, you will deepen your theoretical bond, but first of all you will learn how to implement different concepts from object-orientness. This will allow you to design and implement modern business applications.

User interface design

Graphical user interface design can be done in several ways. Regardless of the method used, please observe the usability guidelines (see previous lesson). The most popular approaches use diagrams containing forms an/or windows. They include individual widgets (*visual gadgets*- GUI elements/controls) represented as geometric figures or existing controls (eg Swing or Windows Forms). The project should include information about:

- placement of individual items,
- their description,
- the widget type (eg *Combo Box*, *List Box*),

- how to react to window size changes (this is a very important aspect, unfortunately quite often neglected),
- possible use of hints (*tooltips*),
- any additional information,
    - read-only control
    - how to select data, eg in ListBox, you can allow multiple selections,
    - behavior of the window (modality).



The above figure shows the design of a form that allows rental of film to customers. At the time of its creation, we have adopted several assumptions about notation. Naturally, this is not a formal standard, but a certain arbitrary decision. Depending on your needs, you can make your own modifications, especially verbal descriptions could be replaced with icons. Assumptions:

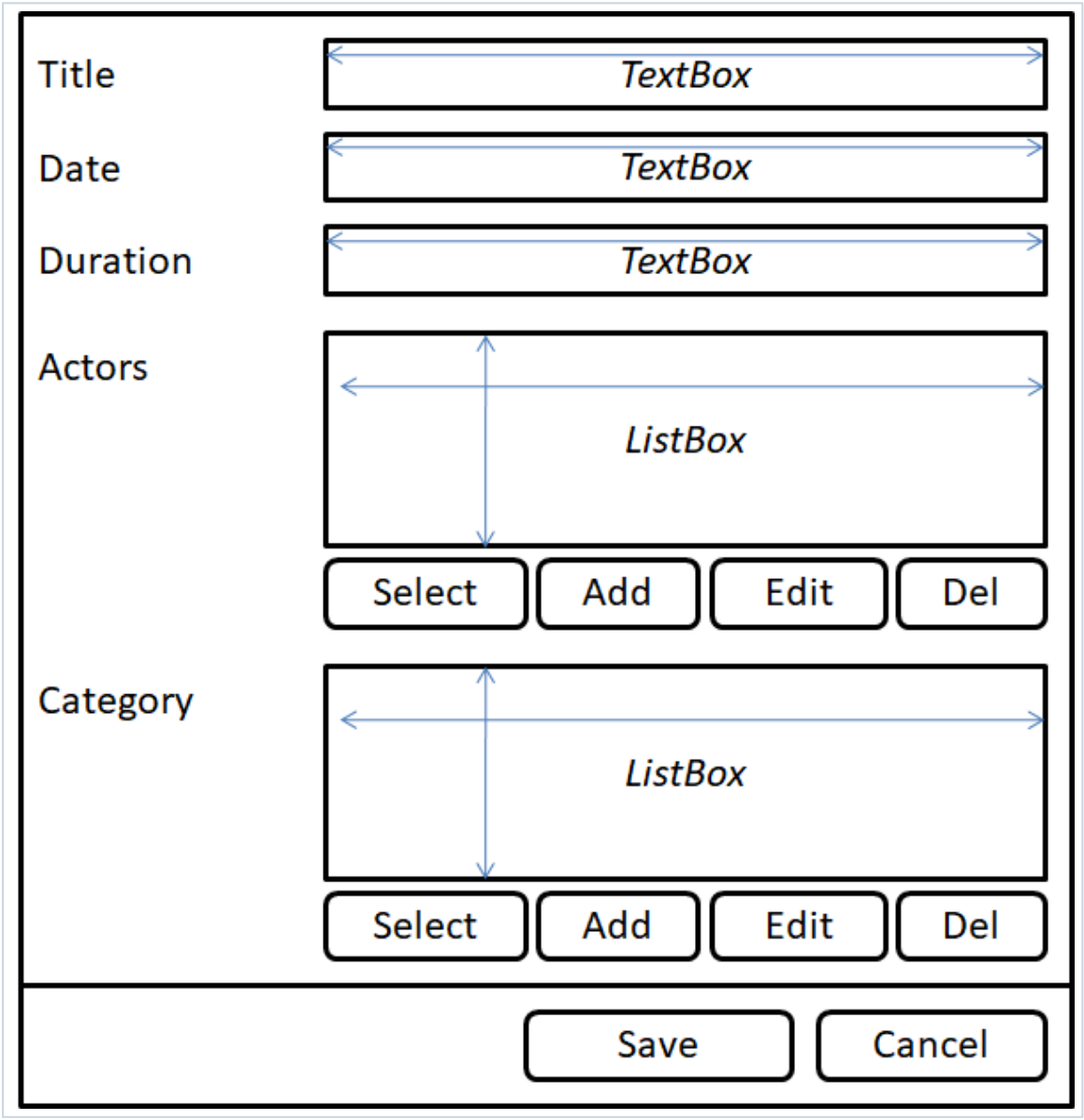
- We have described the type of controls that we want to use. It is worth noting that, for example, the abbreviation *RO*stands for "Read Only". In other words, the user is not able to directly modify the content of the widget, for example, to change the client, you have to select it with the appropriate function. Similarly, the rental fee is billed by the system and can not be manually modified.
- Rectangles with rounded corners are buttons.
- Thin lines with arrows show how the control is supposed to react to changes in the size of the entire window:
    - A single horizontal line means that the control only expands (does not change its height),
    - Single vertical and horizontal lines tell us that the widget should change its height and width. Remember that if there were more than one element on the label, it would be

necessary to establish a mutual relationship when scaling the window, eg proportionally (50% - 50% or 80% - 20%).

Depending on the "typicalness" of the given window, additional designs or verbal descriptions may be required.

This type of project is best for all windows in the application. Thanks to that, we will avoid surprises in the form of unthinkable format. The exception may be a library file that automatically generates a GUI based on a data model. In this situation, the general rules for creating individual elements are "put" inside the library. This approach significantly reduces the amount of work required to design and implement a GUI. This is usually done at the expense of the solution's generality. Generated windows are quite similar to each other and may not always be suitable for a particular type of application. That's why this approach works best for typical business applications (see the next lesson).

The following figure shows the design of another window with a slightly more complicated construction. There are two elements that change their vertical and horizontal size (*actors* and *categories*). Please note that the data that they display is associated with the *Movie*object by association. Typically, in these types of situations (when we show other objects available through association) we will use *ListBox*.



KNOWLEDGE CHECK — Which statements are true?

- The interface design should include information about scaling controls.
- The interface design may contain information about the type of controls.

EXPLANATION & ANSWERS: **Explanation:**

- According to the usability guidelines, a GUI project should take into account information about window scaling and its contents.
- Of course, the designer should also include the types of controls for displaying different types of data.

Different ways of implementation

As we have seen in the previous lesson, designing a good GUI is not a simple matter. Similarly, its implementation is usually quite complicated as well. In general, we can distinguish two most popular approaches to this issue: manual (the programmer writes the code for particular widgets separately) and automated (with a dedicated editor).

Manual GUI implementation usually is:

- time-consuming,
- error-prone,
- complicated.

Consequently, in most cases you should use visual editors that allow you to create GUIs from the components and connect them to each other. But remember that sometimes you may need to make manual modifications even though you use the editor. Therefore, it is important to check how the editor treats the generated code. It is mainly about determining whether manual adjustments are reflected in the visual design. If not, and the entire code is regenerated on the basis of a graphic design, then usually such editor will not be useful for us. Also check that the generated code must have some particular structure or naming. It is better not to have such requirements.

Another way to create a GUI is the declarative approach. The programmer concentrates on what should be done, not how to achieve it. As in the case of SQL, for example, we remain at a higher level of abstraction without delving into implementation details. All ungrateful work is done by a specialized library, which, by building data and some additional information, generates the appropriate widgets. More on this subject will be on the next lesson.

KNOWLEDGE CHECK — Which statements are true?

- Manual GUI implementation is the fastest way to implement it.
- Using graphical editors accelerates the GUI creation process.

EXPLANATION & ANSWERS: **Explanation:**

- Manually implementing a GUI usually gives you the best results, but it's not the fastest way to accomplish it.
- The graphical editor allows you to design and test the interface quickly.

Library selection criteria

Some platforms allow you to work with a variety of alternative GUI libraries. In such situations, it is important to make an informed choice taking into account the following factors:

- Ease of use. Probably one of the most important criteria. Sometimes complicated usage is the result of a great deal of potential and, after all, you should consider using it anyway. In other situations, for example, it is only from some historical past and better search for a more modern framework.
- Modifiability. Specifies the possible degree of influence on appearance and behavior of a given solution. If we want to create a typical interface, it will not matter to us.
- Efficiency. In typical applications, given the high performance of modern hardware, we do not have to worry so much about it. Of course, if you plan to display thousands of items in eg `ListBox`(a bad idea), then you should look for the most "native" solution, mostly created in C ++.
- Cross-platform. It only matters if we want to run our program on different hardware platforms and/or operating systems. There are many solutions on the market that allow this (see below).
- Price. Fortunately, for almost any platform, there are free solutions and they are usually the primary choice. Sometimes, we can buy some add-ons (such as controls) that offer greater visibility, improved appearance or better performance.
- Aesthetics. Basic versions of libraries generally offer a unified (compatible with the operating system) appearance. In particular, modern solutions (Windows Phone, MS WPF, Qt, Android) offer a wide range of possibilities, conceptually similar to CSS stylesheets (for example, for Java - *Look&Feel*.

It is worth noting that the set of components used to build the interface is quite similar in all languages and libraries. However, there are different layout managers, which are the special components responsible for the placement of individual widgets in the window. This is important when the window can change its size. In such a situation, the position and size of individual elements should be adjusted accordingly. It seems that one of the best managers is "anchor" used in MS .NET and lately also in Java.

KNOWLEDGE CHECK — Which statements are true?

- The only significant criterion for choosing a GUI library is its price.
- The criterion of cross-platform is not always relevant.

EXPLANATION & ANSWERS: **Explanation:**

- Of course there are many important criteria (eg performance, portability, price, capabilities) of library selection for GUI creation.
- If we do the system on one platform, the cross-platform criterion may be omitted.

GUI libraries for Java

Below is a brief description of the main Java GUI libraries.

AWT (*Abstract Window Toolkit*)

Historically, the first library for Java was AWT (Abstract Window Toolkit; http://java.sun.com/products/jdk/awt/) published in 1995. This solution used native widgets for the platform, resulting in different looks of the same application on different operating systems. Developers

could use a basic set of widgets (buttons, text boxes, menus, etc.) that respond to various events. AWT was actually a development interface between native operating system and Java application. It contained several layout managers, quite complicated to handle. It was also slow in action. You may have heard of the poor performance of Java when displaying graphics.

Swing

Two years later (in 1997), the Swing library saw the light of day (http://www.javaswing.net/) by introducing revolutionary changes. First of all, the look and behavior of widgets was determined by Java, resulting in the same appearance and behavior of the application across platforms. This was made possible by the creation of a special Java2D technology, which offered many times better performance than AWT. Widgets and installable graphic themes called *Look&Feels* have also been expanded. This approach has definitely improved platform independence and increased extensibility. The whole library has become component oriented using them to draw fragments of controls such as borders or decorations.

The MVC pattern (Model-View-Controller) is widely used. Most widgets have their own model (defined by the Java interfaces) that influences the behavior of the data. You can also use their basic implementation provided by Swing.

The Swing supports two categories of events: "physical" (eg mouse click) and logic ones related to the model (eg adding a row of data to a table or selecting an item). Of course it is better to work with the latter category.

Swing was the primary and default Java GUI library for desktop application. Currently (since Java 8) it is replaced by JavaFX.

JavaFX

JavaFX has been introduced as a modern way of working with the GUI. It uses the declarative approach similar to that used on Android. Interface is defined in a special language that utilizes XML.

Like its predecessor, this library uses the MVC pattern. Because the definition of GUI is in XML files, the division is even stronger than in Swing.

The creators also thought of improving the appearance and applied a well-known approach with CSS files. In addition, we can work with a web designer who is not a developer, but will be able to modify the look of the application.

More information can be found in the official Java documentation.

SWT (Standard Widget Toolkit)

An alternative to Swing is the SWT library (Standard Widget Toolkit; http://www.eclipse.org/swt/) originally developed by IBM, and is currently being developed by the Eclipse Foundation (their flagship IDE is using SWT). The solution utilizes the native capabilities of the operating system to draw the controls, which improves their drawing speed. Therefore, you can meet with the opinion that it is much faster than Swing. To some extent, this is true (drawing the controls), but overall performance is negatively affected by the need to "translate" the data from Java into the format "understood" by the native controls.

Programs that use SWT are portable, but require a dedicated SWT implementation for each operating system (even for Windows x86 and x64). It is not always available for all platforms on which the JVM exists.

SWT does not use MVC, but provides the ability to collaborate with the *JFace* library, which supports this approach. However, because of the use of native widgets, their customization is not always easy (as opposed to Swing).

Another drawback, especially for languages like Java, is the need to manually release the resources used by widgets: `dispose()`. In practice these are usually subclasses `Resource`such as `Image`, `Color`and `Font`.

KNOWLEDGE CHECK — Which statements are true?

- The Swing library is currently (still) the most popular desktop application development solution for Java.
- The SWT library is always faster than Swing.

EXPLANATION & ANSWERS: **Explanation:**

- In fact, Swing is currently the most popular desktop application development solution for Java. This is because it is integrated with the platform.
- In some situations the SWT library is faster than the Swing ('drawing' component), but in others the slower ('converting' the data to a form acceptable by native controls).

Custom GUI library

Sometimes it may be necessary to implement a custom graphical user interface. We do not mean to add a component to eg Swing, but develop the entire framework that will replace it. It's a good idea to start by reviewing existing solutions and trying to select one with an optional modifications. Thanks to this we can gain a whole new look and behavior with much less work.

If it turns out that we have to create such a library ourselves, we have to make some important decisions that affect our further actions:

- Choice of event model: "physical" or "logical". Both have their advantages and disadvantages, but it seems that the more universal is the latter. Thanks to it, we will be able to abstain from the way of doing things (mouse, keyboard) and focus on its semantics (eg selecting an item from the list);
- Widget set. Due to the laboriousness, find out what we really need. Let's also enjoy the benefits of objectivity: larger controls (*MessageBox*) can consist of smaller (previously created) , e.g. *button*, *label*, *window*);
- How to draw the controls. This is quite a laborious part because we usually have only so-called basic operations like drawing lines, rectangles, etc. You also need to consider the way windows are scaled and thus drawn regardless of the screen size or resolution;
- Link events to their operation. We need to determine how we will combine business logic with individual controls. For this purpose, we can define, for example, some programming interfaces that need to be implemented by the appropriate classes.

As you can see from the above, quite sketchy argument, this is quite a complicated task, which means the need to work at a very low level of abstraction. For this reason, if possible, better use existing solutions.

KNOWLEDGE CHECK — Which statements are true?

- The workload needed to create your own GUI library is comparable to that required for existing modification.
- Creating your own GUI library requires first of all the ability to work at a high level of abstraction.

EXPLANATION & ANSWERS: Explanation:

- Modifying an existing library is a lot simpler than creating your own from scratch.
- Creating your own GUI library requires working on a fairly low level of abstraction (operating system events).

GUI implementation - remarks

During many years of my professional work, I have seen a lot of different implementations of graphical user interfaces. Unfortunately, in some of them, the same, relatively small, but significant **list of errors** was repeated:

- Adding `String` instead of objects to widgets. It's the absolute number one on the wrong list. This approach means that we first add text (for example, the name of the product) and then find the object (usually the `getSelectedIndex()` method) to do something about it. Same drawbacks. And if we immediately add an object (such as our created `Product` class) instead of its name, then we could use `getSelectedValue()` and immediately get an instance of the added class (without any search). What to do to get the objects properly displayed? Of course, override the `toString()` method in the business class. Let's remember: add an object, not a text to a widget!
- Placing business code directly in event handling methods. Why this is not a good approach? Because usually the same functionality we can run in different ways, from different places in the program. Therefore, it is best to separate it to some business method and call it from *listener*.
- Share listener without a good reason. This approach will lead to unreadable code: a big method with many `if` conditions, such as the source of the event and the execution of the various actions. Instead, it is better to use dedicated event handlers for each widget. Exceptions may include collections of controls with identical operating procedures.
- Use physical/hardware events/listeners (such as `MouseListener`) instead of "logical" (eg `actionPerformed`). What is the problem? Well, when we react to clicking a mouse in eg `ListBox`'ie it will all work fine but only in conjunction with the mouse. And yet the user can (and often does) use the cursor keys and eg *Enter* key.

KNOWLEDGE CHECK — Which statements are true?

- AddingStringinstead of objects to the control decreases the performance of the application and increases the required developer workload.
- Using the dedicatedlistenerfor handling controls improves code readability.

EXPLANATION & ANSWERS: Explanation:

- Of course, addingStringinstead of objects to the control decreases application performance and increases the developer's workload. This is because the object needs to be converted to text and subsequently searched for in the extent.
- Dedicatedlisteners sprawiają, że mamy mniejsze fragmenty kodu do zarządzania i zrozumienia.

LESSON 15

GUI Implementation & Logic

Step-by-step implementation of interactive Java GUI windows, event listeners, extent data binding, and video demonstrations.

Educational outcomes

Student:

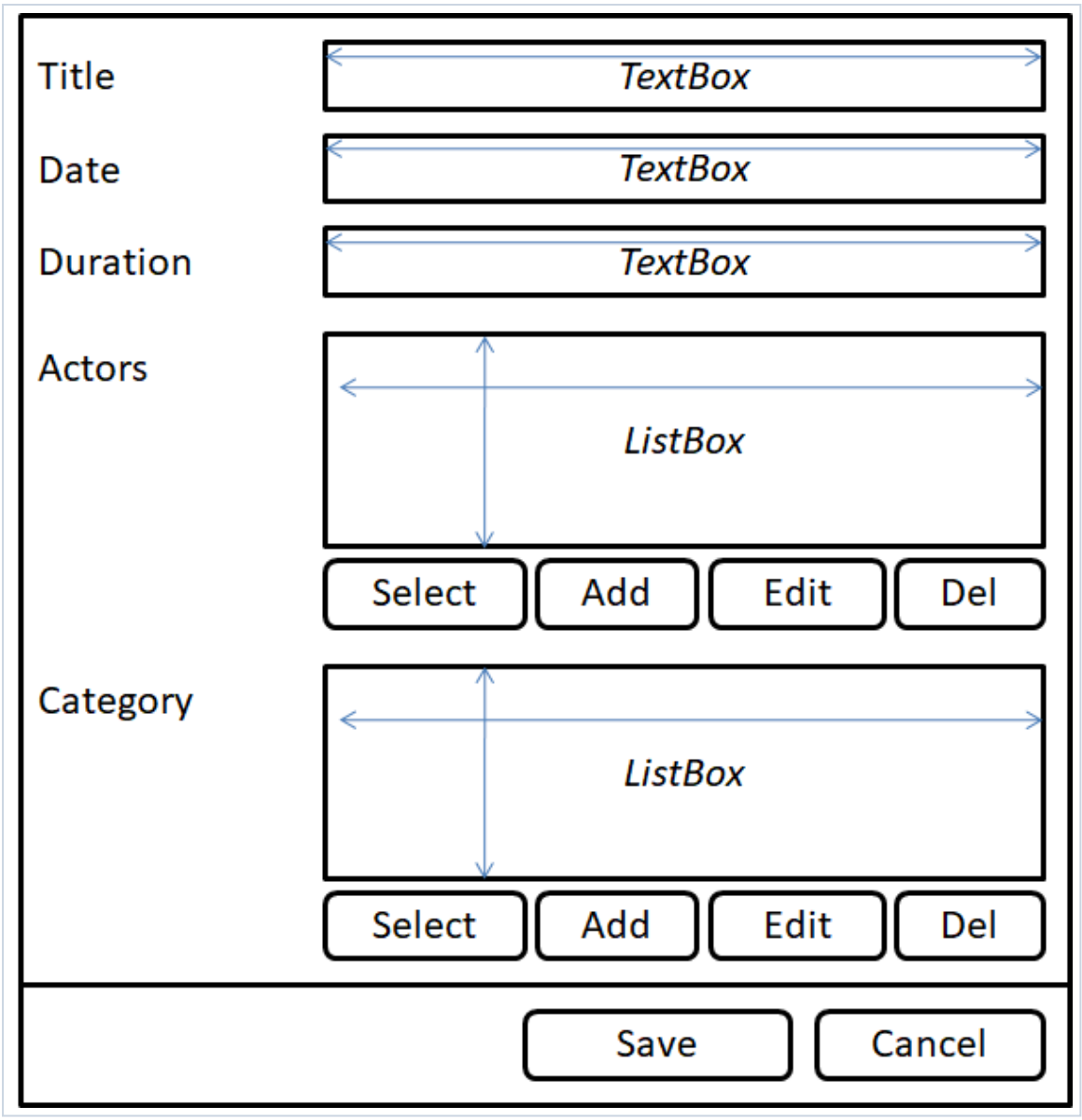
- explains how to design and implement graphical interfaces using natural language (understanding);
- identifies and corrects errors in the source code that implements the above interfaces (understanding).;
- lists the constructs implemented in the source code (understanding);
- implements the aforementioned constructs using the selected programming language (application);
- indicates differences in functionality between two selected programs based on fragments of source code (analysis);
- creates a program that implements specific business requirements using the structures learned in the lectures (synthesis);

Justification

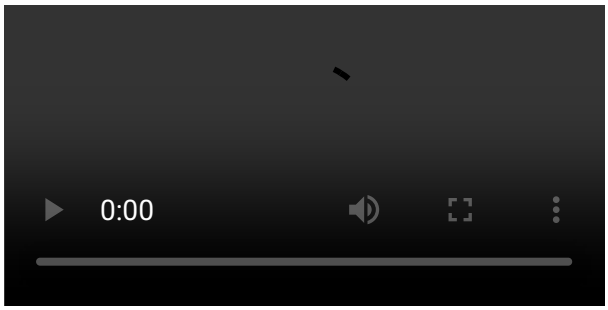
Modern programming languages are mostly based on the object paradigm. Therefore, its practical understanding is a necessary basis for working in this profession. Moreover, programmers often make systems based on analytical materials. And these largely use object-oriented graphical notations (such as UML). During this and the next lessons, you will deepen your theoretical bond, but first of all you will learn how to implement different concepts from object-orientness. This will allow you to design and implement modern business applications.

Creating a GUI in the editor

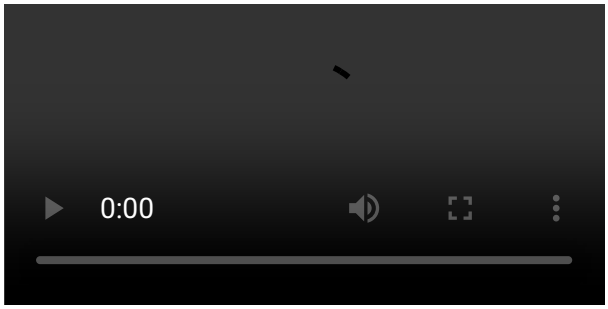
As mentioned in the previous lesson, in most cases GUI editors are used. One of them, called *Windows Builder*, is available for Eclipse. We will use it to create a piece of the program that implements the previously designed window. For convenience, once again the same design:



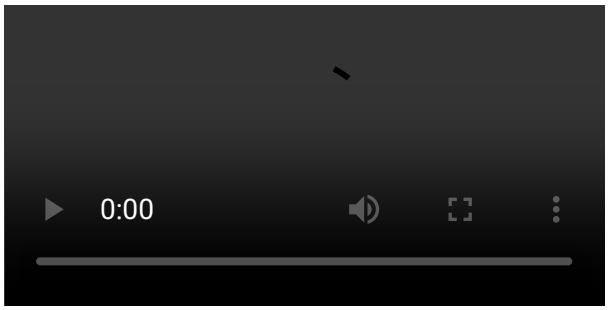
The following movie shows how to use a GUI editor to design a window with proper scaling of its components.



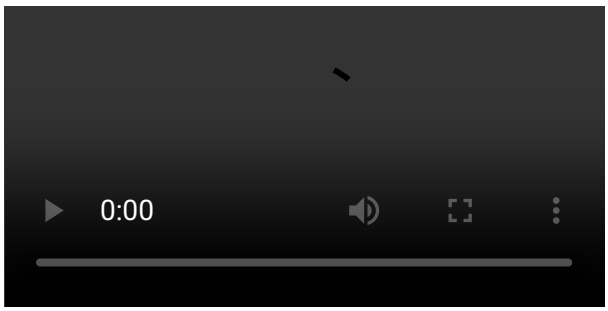
On the next material we see how to rename variables related to attributes and label captions.



And yet another movie showing how to modify the layout manager's settings so that the actor list occupies more space (at the expense of the category).



Our window requires a way to connect particular controls (e.g. buttons) with a source code to execute on click. See the video:



Actually, we should still modify the way the two lists are displayed. We would like to see a scrollbar in the case of a large number of entries. For this purpose, the `JList` instance should be embedded within `JScrollPane`. The appropriate modification can easily be done with the editor:

- We select a specific list,
- Click the right mouse button and select *Surround with => JScrollPane* from the menu.

KNOWLEDGE CHECK — Which statements are true?

- The programmer, in all circumstances, should use only one distribution manager.
- All the controls we refer to in the code should have the correct names given.

EXPLANATION & ANSWERS: **Explanation:**

- Layout managers should be selected according to needs. Moreover, it often happens that in the same window, we use several different layouts.
- Of course, in order for the program to be readable, references in the source code to controls should have meaningful names - often with a specific convention.

Adding business logic to the application

Before we start adding functionality to our application, let's take a look at sample date. For our purposes it is enough to have ordinary arrays with predefined movies and actors.

```Java
private static Movie[] movies = new Movie[] {new Movie("Title 1"), new Movie("Title 2"),
new Movie("Title 3")};
private static Actor[] actors = new Actor[] {new Actor("AB"), new Actor("CD"), new
Actor("EF"), new Actor("GH"), new Actor("IJ"), new Actor("KL")};
```

And another snippet of code that will make links between the movies and the actors. We use methods implemented in [association discussions](#).

```Java
movies[0].addActor(actors[1]);
movies[0].addActor(actors[3]);
movies[0].addActor(actors[5]);

movies[1].addActor(actors[0]);
movies[1].addActor(actors[1]);

movies[2].addActor(actors[1]);
movies[2].addActor(actors[2]);
movies[2].addActor(actors[4]);
```

Now we can show something. We will create a method that will display information about a given video in a previously created window. Initially we will remember the video instance (4), then update the text boxes (7 - 10), and finally create the appropriate model for the actors display list.

Java

```
public class MovieFrame extends JFrame {
        // ...

        public void setMovie(Movie movie) {
                this.movie = movie;

                // Update the GUI
                setTitle("Movie: " + movie.title);
                txtTitle.setText(movie.title);
                txtLength.setText(Integer.toString(movie.duration));
                txtDate.setText(movie.date.toString());

                ObjectsModel<Actor> modelActor = new ObjectsModel<Actor>
(movie.getActors());   // see below for the ObjectsModel class
                lstActors.setModel(modelActor);

                // TODO add categories
        }
}
```



COMMENTARY

Most modern IDEs recognize special markings in the source code. One of these is TODO in a comment that is treated as a task to create. It will be displayed in a special view (in the Eclipse: *Window/Show view => Tasks* menu). It is worth to keep them up to date so that you do not forget to realize some functionality.

As we mentioned at the presentation of the Swing library, a large part of the components work on data models. This is also the case with `JList`. The simplest way to display items in this list is to use the ready class that implements the default model: `DefaultListModel`. By contrast, this is not always the best solution and to show another possibility, we will create our own model. It inherits from `AbstractListModel` and must implement two abstract methods: `public T getElementAt(int index)` and `public int getSize()`. Let's look at the code below and some comments:

```java
public class ObjectsModel<T> extends AbstractListModel<T> {
        private static final long serialVersionUID = -7858066195290617794L;
        ArrayList<T> objects;

        public ObjectsModel(List<T> objs) {
                objects = new ArrayList<T>(objs);
        }

        @Override
        public T getElementAt(int index) {
                return objects.get(index);
        }

        @Override
        public int getSize() {
                return objects.size();
        }

        public void addObject(T newObj) {
                objects.add(newObj);

                // Inform the model about the change
                fireContentsChanged(this, getSize() - 1, getSize() - 1);
        }

        public void removeObject(T obj) {
                objects.remove(obj);
                // Inform the model about the change
                fireContentsChanged(this, getSize() - 1, getSize() - 1);
        }

        public List<T> getObjects() {
                return objects;
        }
}
```

- (1). Our class is parameterized by the type of object that will be stored in the model. This will allow us to use it not only for the movies (what we did) but also, for example, for the category (which we leave as one of the tasks to do).
- (3). A container holding our objects.
- (6). Make a copy of the specified (in the constructor) container (see below).
- (10), (15). Implementation of the required abstract methods.
- (19). A method that adds an object to the model (2) and announces a list of its changes (23) - so that the contents of the list are automatically refreshed.
- (26). Method as above but removing objects.
- (32). Meotda returning all objects from the model. You can discuss whether instead of it would not be better to keep them hidden and put up only operations on them - for simplicity we simply return the collection.

While discussing the usability of graphical interfaces, we noticed that it is always worthwhile that the user always has the ability to cancel their operations. For this reason, we designed the *Save* and *Cancel* buttons in our window. Unfortunately, this requirement significantly complicates implementation, especially the list of related objects. Without it, we could just plug in the appropriate business methods to create two-way links. Unfortunately, we can not do this because there would be no possibility of cancelling these operations.

In order to be able to cancel the changes, we will store the association locally (in the model - see above) and only after the approval update the business object. This operation must include not only adding new actors, but also removing them. Code of the corresponding method ("connected" to the *Save* button) and comments:

```Java
public class MovieFrame extends JFrame {
        // ...

        protected void SaveData() {
                // Save changes
                movie.title = txtTitle.getText();

                // TODO add data validation (calendar, duration)
                try {
                        movie.duration = Integer.parseInt(txtLength.getText());
                        movie.date = DateTime.parse(txtDate.getText());
                }
                catch(Exception ex) {
                        JOptionPane.showMessageDialog(null, "Please check the correctness
of the entered data.");
                        return;
                }

                // Update links (due to support 'Cancel' in the window)
                List<Actor> linkedObjects = ((ObjectsModel<Actor>)
lstActors.getModel()).getObjects();
                // Deleted objects (we iterate over the copy)
                for (Actor actor : new ArrayList<Actor>(movie.getActors())) {
                        if(!linkedObjects.contains(actor)) {
                                movie.removeActor(actor);
                        }
                }
                //Added objects
                for (Actor actor : linkedObjects) {
                        movie.addActor(actor);
                }

                // quitApp(false);
        }
}
```

- (5). We are updating the title of the movie.
- (9), (10). Updating time and date information (stored as a dedicated class instance `org.joda.time.DateTime`; we could also use an instance of Java8 class: `LocalDate`). In both

cases we can expect errors related to badly entered values. A good validation of data and/or specialized controls should be included in the final implementation.

- (12). Display messages in case of exception.
- (18). Recover object collection from model.
- (20). A loop that looks at a copy of the current link list and removes those that are not in our model.
- (26). Loop that adds links in the model. Due to the way of implementation, the attempt to add a duplicate will be ignored.

We also have a simple way to add new actors to the movie (*Add* button). A target window would be needed for the implementation of the actor data. Instead, we will simply generate it with a random name. Let's look at the following method (called from the *Add* button listener): first create a new actor (5), then recover the model from `JList`, cast to `ObjectsModel<` and add object. The model will update the list so that it will be refreshed.

```Java
public class MovieFrame extends JFrame {
        // ...

        protected void addNewActor() {
                Actor actor = new Actor("name " + (int)(Math.random() * 10000));

                ((ObjectsModel<Actor>) lstActors.getModel()).addObject(actor);
        }
}
```

It is also worth adding the ability to display information about the selected actor. For this purpose, we will create a simple window (eg `ActorFrame`) containing the name of the chosen actor and his/her movie list, eg:



Same as for movie information, in the `ActorFrame` class we will add a method to display the information. Notice the reuse of our `ObjectsModel` class, but this time for `Movie`:

```Java
public class ActorFrame extends JFrame {
	// ...

	public void setActor(Actor actor) {
		setTitle("Actor: " + actor.name);
		lblName.setText(actor.name);
		lstMovies.setModel(new ObjectsModel<Movie>(actor.getMoviey()));
		setLocationRelativeTo(null);
	}
}
```

And how to display information about selected actor? It's pretty simple: add a listener for a double click on the actor. To do this, simply select the specific event in the property pane and double-click on its name - the IDE will generate the required code, which looks something like this:

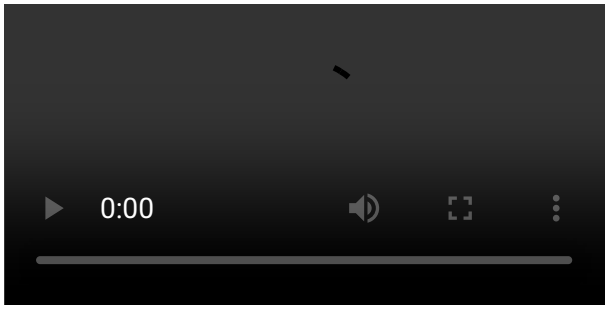
```Java
public class MovieFrame extends JFrame {
	// ...

	public MovieFrame() {
		// ...

		lstActors = new JList<Actor>();
		lstActors.addMouseListener(new MouseAdapter() {
			@Override
			public void mouseClicked(MouseEvent arg0) {
				if(arg0.getClickCount() == 2) {
					showActor(lstActors.getSelectedValue());
				}
			}
		});
	}

	// Displays information about the specified actor
	protected void showActor(Actor actor) {
		ActorFrame actorFrame = new ActorFrame();
		actorFrame.setActor(actor);
		actorFrame.setVisible(true);
	}
}
```

Let's see what we have achieved. Of course, I encourage you to go through the above steps personally.


0:00



AUXILIARY ASSIGNMENT

Extend the implemented GUI fragment with additional functionality, e.g.

- category support (similarly to actors),
- managing actors,
- selecting Actors for linking (from their extent)
- browsing all the movies.

The provided solution (along with a brief description, source code and example of utilization) can receive up to 5% of the points planned for mini-projects.

Long-lasting actions and the GUI

As we mentioned during the usability discussion, users do not like when the computer is not responding to their commands. This may happen when an application performs some long-lasting action, such as:

- search for items,
- calculations,
- data processing,
- download/upload resources from/to the network,
- waiting for a service availability.

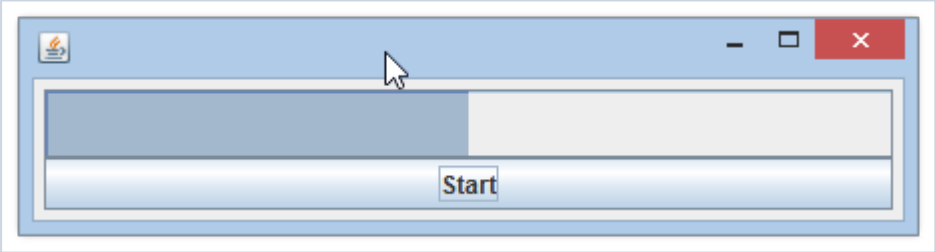
In such a situation, you should inform the user of ongoing activities, keep up to date on their progress, and allow them to stop. Studies show that even long-lasting activities, when accompanied by the corresponding GUI, seems to last less. If we are able to predict that an operation can take a long time, it will not hurt to get confirmation from the user about starting it.

How to do it? Unfortunately the following (pseudo) code will not work as expected:

```Java
Public void longLastingMethod() {
        For(...) {
                // Performing long-lasting action,
                // eg. searching for information

                // Refresh the GUI
                progressBar.setProgress(...);
        }
}
```

Just in case, let's create this simple window with progress bar (it can be created eg in the previously described editor) and adding the code:



```Java

public class ProgressFrame extends JFrame {
	// ...

	JProgressBar progressBar;

	// Update progress bar
	public void setProgress(int val) {
		progressBar.setValue(val);
	}

	// Example method for complex calculations
	protected void startJob() {
		List<Integer> numbers;
		int maxCount = 10000000;

		final int maxNumber = 100000000; //1000000;
		int i = 1;

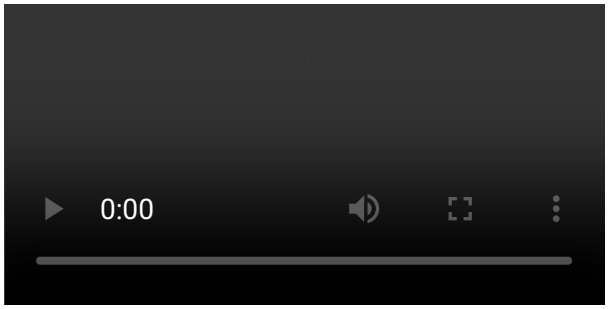
		numbers = new LinkedList<Integer>();

		while(i < maxCount && isCancelled() == false) {
			Integer curInt = new Integer((int)(Math.random() * (double)
maxNumber));
			numbers.add(curInt);

			if(i % 100 == 0) {
				// Possible information on progress
				setProgress((int)(100.0 * (float)i / maxCount));
			}

			i++;
		}

		System.out.println("Finished: " + numbers.size());
	}
}
```



As you can see in the above movie, according to our assumptions, the refresh of information does not work. The reason is that in popular programming languages (Java, C #), the default implementation performs computations in the same thread as the GUI upgrade. As a result, it will be refreshed when the computer will have "some free time". In the extreme case only when all calculations are completed - that is definitely too late. How to remedy this? You need to use threads of a programming language.

KNOWLEDGE CHECK — Which statements are true?

- The above lack of information refresh is due to the poor performance of Java.
- Moving calculations to a separate thread can improve user interface performance.

EXPLANATION & ANSWERS: **Explanation:**

- The problem with refreshing progress information is due to the use of the same thread by calculation and GUI library.
- The best way to solve this problem is to move the calculation to a separate thread.

Utilization of the `SwingWorker`

As we mentioned, in order to properly implement a long-term action, the developer must apply the thread. Usually, for this type of implementation, you should use the auxiliary class that comes with the language:

- `SwingWorker` for Java (a few years ago it was not a part of official distribution),
- `BackgroundWorker` for MS C#.

Both classes are constructed using a similar concept. The programmer must override the method:

- executing long-term action: `SwingWorker.doInBackground()`,
- done after the action is finished: `SwingWorker.done()`.

In addition, there are several other methods available, including.

- cancelling operation,
- getting partial results.

The SwingWorker class is parameterized with two types:

- The return type of the method (longtime action result): `doInBackground()` and `get()`. The latter waits for the end of the whole action and then returns the value of the calculation (note: this waiting blocks the GUI).
- Type of values passed as partial results (as a list).

Let's start with something simple - for now without a GUI, but of course using `SwingWorker`. Let's look at the code below: we inherit the SwingWorker class by parameterizing it with the value of our calculation (`String`) and `Void` (we do not return a partial results). Let's override the `doInBackground()` method with the business logic that implements our long-term operation.

```java
public class Worker extends SwingWorker<String, Void> {

	@Override
	/**
	* Counts the sum of numbers from 1 to maxNumber and return it as String.
	*/
	protected String doInBackground() throws Exception {
		// Count the sum of numbers from 1 to max
		final int maxNumber = 500000000;
		int sum = 0;

		for (int i = 1; i < maxNumber; i++) {
			sum += i;
		}

		return Integer.toString(sum);
	}
}
```

And how to use it? Eg, as shown below:

```java
public static void main(String[] args) {
	Worker worker = new Worker();

	System.out.println("Starting calculations...");

	// We are launching our long-term actions
	worker.execute();

	// But the control comes back and we can do something else
	System.out.println("Waiting for the result...");

	// Exceptions required by SwingWorker
	try {
		// We are waiting, checking if it is finished
		while(!worker.isDone()) {
			Thread.sleep(1000);
		}

		// Calculations completed ==> get the result
		String sum = worker.get();

		System.out.println("Result: " + sum);
	} catch (InterruptedException ex) {
		Logger.getLogger(Main.class.getName()).log(Level.SEVERE, null, ex);
	} catch (ExecutionException ex) {
		Logger.getLogger(Main.class.getName()).log(Level.SEVERE, null, ex);
	}
}
```

- (2). We are creating an instance of our worker;
- (7). We start calculations;
- (10). Due to the fact that they are implemented in a separate thread, the control immediately goes to the next line (without waiting for the result). Otherwise we would have used the `get()` method;
- (15). In the loop, "we ask" our worker if has already finished;
- (20). If so, we will get the result.

Now, to the above program, let's add progress information (still on the console). Modification, usage and comments in code:

```Java
public class WorkerPlus extends SwingWorker<String, Void> {

	@Override
	/**
	* Counts the sum of numbers from 1 to maxNumber and return it as String.
	*/
	protected String doInBackground() throws Exception {
		final float maxNumber = 500000000;
		int sum = 0;

		for (int i = 1; i < maxNumber; i++) {
			sum += i;

			if(i % 100 == 0) {
				// Optional progress information
				setProgress((int)(100.0 * (float)i / maxNumber));
			}
		}

		return Integer.toString(sum);
	}
}
```

```Java
public static void main(String[] args) {
        WorkerPlus worker = new WorkerPlus();

        System.out.println("Starting calculations...");

        // We are launching our long-term actions
        worker.execute();

        // But the control comes back and we can do something else
        System.out.println("Waiting for the result...");

        // Exceptions required by SwingWorker
        try {
                // We are waiting, checking if it has ended and displaying info about the
progress
                while(!worker.isDone()) {
                        System.out.println("Progress: " + worker.getProgress());
                        Thread.sleep(1000);
                }

                // Calculations completed ==> get the result
                String sum = worker.get();

                System.out.println("Result: " + sum);
        } catch (InterruptedException ex) {
                Logger.getLogger(Main.class.getName()).log(Level.SEVERE, null, ex);
        } catch (ExecutionException ex) {
                Logger.getLogger(Main.class.getName()).log(Level.SEVERE, null, ex);
        }
}
```

Now, let's take advantage of our new knowledge to implement progressive GUIs. First, we need to create the appropriate window (eg in the editor using the description given at the beginning of the lesson) with a progress indicator, eg `javax.swing.JProgressBar`. Note that our window is an instance of `JDialog` (modal dialog), not a `JFrame`. And of course the code:

```Java
public class ProgressDialog extends javax.swing.JDialog {
        // ...

        private SwingWorker worker;

        /**
        * Sets a worker connected to the dialog.
        * @param worker
        */
        public void setWorker(SwingWorker worker) {
                this.worker = worker;
        }

        public void setValue(Integer progress) {
                progressBar.setValue(progress);
        }

        private void btnCancelActionPerformed(java.awt.event.ActionEvent evt) {
                if(worker != null) {
                        worker.cancel(false);
                }
                this.setVisible(false);
        }
}
```

Important snippets:

- (4). A reference to the *worker* instance. We will need it to cancel any calculations.
- (18). This method is called when the user presses the `Cancel` button. First, we may call the dedicated *worker* method (20) and then hide the window.

Now the code that executes our complex calculations. It is very similar to what we have already seen, so let's emphasize only differences:

- (4). Window instance;
- (24). Note that in the loop, in addition to checking the number of repetitions, we also check the cancelation of operations;
- (30). Call the `setProgress()` method, which safely updates progress information - see next.

Java

```
public class NumberGenerator extends SwingWorker<List<Integer>, Void> {
        private List<Integer> numbers;
        private int maxCount;
        ProgressDialog progressDialog;

        public NumberGenerator(int maxCount, ProgressDialog progressDialog) {
                super();

                this.maxCount = maxCount;
                this.progressDialog = progressDialog;
        }

        /**
        * Generates numbers and add them to the list.
        */
        @Override
        protected List<Integer> doInBackground() throws Exception {
                final int maxNumber = 1000000;
                int i = 1;

                numbers = new LinkedList<Integer>();

                while(i < maxCount && isCancelled() == false) {
                        Integer curInt = new Integer((int)(Math.random() * (double)
maxNumber));
                        numbers.add(curInt);

                        if(i % 100 == 0) {
                                // Optional progress information
                                setProgress((int)(100.0 * (float)i / maxCount));
                        }

                        i++;
                }

                return numbers;
        }

        @Override
        protected void done() {
                super.done();

                // Hide GUI
                progressDialog.setVisible(false);
        }
}
```

Below is a code that implements the main window, which allows starting the whole operation. Let's look at how are sent notifications about progress (21 - 27).

```Java
public class MainFrame extends javax.swing.JFrame {
        // ...

        /** Creates new form MainFrame */
        public MainFrame() {
                initComponents();
        }

        private void generate() {
                final int count = 1000000;

                // Create GUI to display info about progress
                final ProgressDialog dialog = new ProgressDialog(this, "Please wait...");

                // Create a generator
                NumberGenerator generator = new NumberGenerator(count, dialog);

                dialog.setWorker(generator);

                // Add a listener to have information about the progress of a long-term
action
                generator.addPropertyChangeListener(new PropertyChangeListener() {
                        public void propertyChange(PropertyChangeEvent evt) {
                                if ("progress".equals(evt.getPropertyName())) {
                                        dialog.setValue((Integer) evt.getNewValue());
                                }
                        }
                });

                // Start the action
                generator.execute();

                dialog.setVisible(true);
        }
}
```

The effect of the above code we saw on the last movie.

KNOWLEDGE CHECK — Which statements are true?

- TheSwingWorkerclass displays the default version of the progress bar.
- Using theSwingWorkerclass speeds up execution of calculations about 3 times.

EXPLANATION & ANSWERS: **Explanation:**

- The default implementation of theSwingWorkerclass does not contain any GUI elements.
- Using theSwingWorkerclass is intended to improve GUI performance, rather than speeding up computation.

Utility class `ProgressWorker`

Paradoxically, contrary to its name, the `SwingWorker` class does not directly provide any GUI for the Swing library. Therefore, it is worth trying to change this situation. Let's create a class that makes it easy to work with the GUI and SwingWorker. We include data processing functionality (SwingWorker) and display progress information using the GUI. We'll call it `ProgressWorker<finalResult, partialResult>`. It will inherit from `SwingWorker` and contain the modified `ProgressDialog` instance.

Java

```
public abstract class ProgressWorker<finalResult, partialResult> extends
SwingWorker<finalResult, partialResult> {
	private ProgressDialog progressDialog;

	/**
	* Creates the ProgressWorker instances to display the given message during
processing. Invoked from the subclass constructor.
	* @param progressMessage
	*/
	public ProgressWorker(String progressMessage) {
		super();

		progressDialog = new ProgressDialog(null, progressMessage);
		progressDialog.setWorker(this);

		// Add a listener to have information about the progress of a long-term
action
		addPropertyChangeListener(
				new PropertyChangeListener() {
					public void propertyChange(PropertyChangeEvent evt)
{
						if
("progress".equals(evt.getPropertyName())) {
							progressDialog.setValue((Integer)
evt.getNewValue());
						}
					}
				}
		);
	}

	/**
	* Runs a long-lasting action and shows a modal dialog.
	* Should be called instead of execute().
	*/
	public void start() {
		execute();

		progressDialog.setVisible(true);
	}

	/**
	* Called after processing (or canceling).
	* Calls the superclass method and hide the dialog.
	*/
	@Override
	protected void done() {
		super.done();

		// Hide the GUI
		progressDialog.setVisible(false);
```

```
        }
}
```

And how can we use it? Actually almost like the usual `SwingWorker`. We inherit from the superclass and provide the necessary `doInBackground()` method. Its implementation is similar to what we have done before, so it does not require additional comment.

Java

```
public class NumberGenerator extends ProgressWorker<List<Integer>, Void> {
        private List<Integer> numbers;
        private int maxCount;

        public NumberGenerator(int maxCount) {
                // Necessary constructor call from superclass with the text to display
                super("Generating numbers (" + maxCount + ")... Please wait");

                // System.out.println("Generating numbers (" + maxCount + ")... Please
wait");

                this.maxCount = maxCount;
        }

        /**
        * Generates numbers and add to list.
        */
        @Override
        protected List<Integer> doInBackground() throws Exception {
                final int maxNumber = 1000;
                int i = 1;

                numbers = new LinkedList<Integer>();

                while(i < maxCount && isCancelled() == false) {
                        Integer curInt = new Integer((int)(Math.random() * (double)
maxNumber));
                        numbers.add(curInt);

                        if(i % 100 == 0) {
                                // Possible information on progress
                                setProgress((int)(100.0 * (float)i / maxCount));
                        }

                        i++;
                }

                return numbers;
        }
}
```

Traditionally an example of the utilization for the above implementation: create an instance (1) and start calculations and display the progress window (2). Verify that the process has not been canceled (5) and

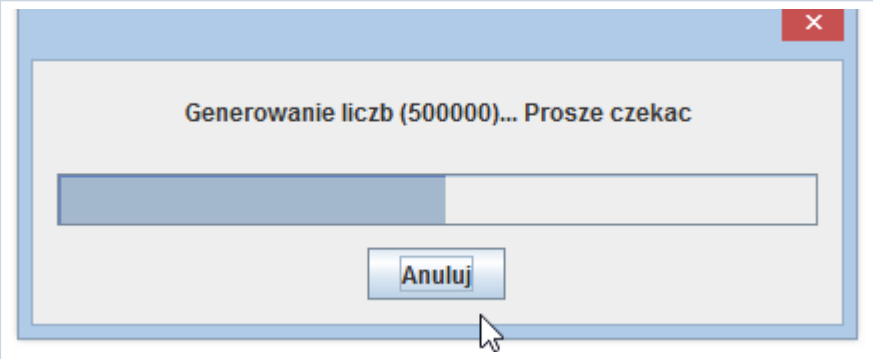
retrieve the result (11). As you can see, implementing almost the same as the standard SwingWorker code, we received "free" graphical information about the progress of the operation.

```Java

NumberGenerator gen = new NumberGenerator((Integer) spnCount.getValue());
gen.start();

// If canceled we will get an exception when the results are retrieved
if(gen.isCancelled()) {
        JOptionPane.showMessageDialog(null, "Operation canceled.");
        return;
}

// Get the generated list
List<Integer> numbers = gen.get();
```



KNOWLEDGE CHECK — Which statements are true?

- TheProgressWorkerclass displays the progress bar.
- TheProgressWorkerclass just likeSwingWorkersupports interrupting the operation.

EXPLANATION & ANSWERS: **Explanation:**

- One of the improvements to theProgressWorkeris to add a progress bar.
- Both classes:ProgressWorkerandSwingWorkerhave support for interrupting the operation.

Creating a GUI using semantic declarativeness

One of the most recently adopted paradigms used to create GUIs is the declarative approach. Its main feature is that the developer focuses on ***what*** should done, leaving ***how*** to do it for the computer. In other words, we define the final effect we are interested in (some GUI), and the relevant component/module will take care of its preparation.

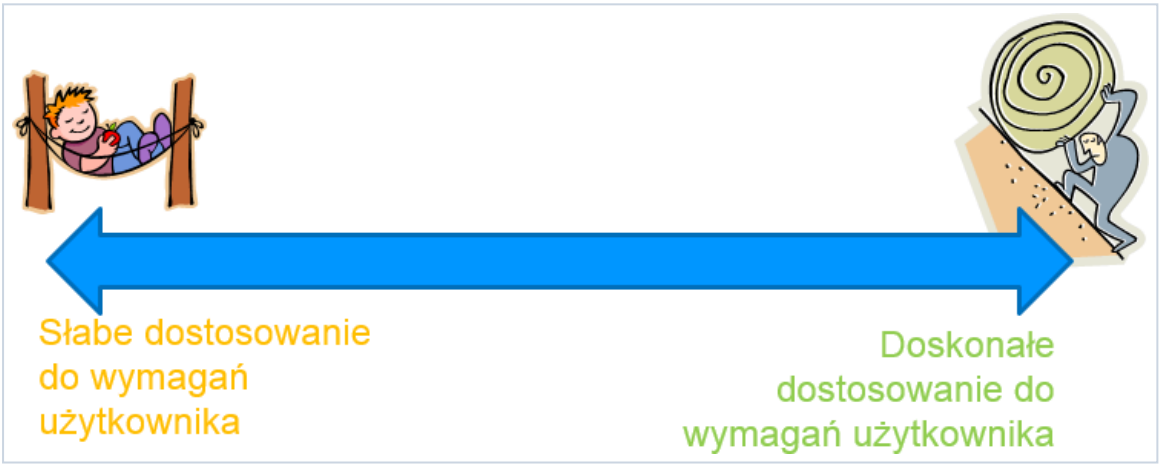
We can distinguish two approaches:

- **Component declarativeness**. The developer is working with UI components. Using a dedicated language (or a configuration file) determines what a window or panel should be. As a result, the amount of information needed is quite similar to the classical GUI creation.Another is the way of providing them. Examples include:

    - Microsoft XAML used in Universal Applications, Xamarin Forms, Silverlight, Windows Presentation Foundation (WPF), Windows Phone and Windows Store applications (former Metro);
    - Java FXML used in JavaFX;
    - Android - XML files defining the GUI.

- **Semantic declarativeness**. The programmer operates on a higher level of abstraction, such as a data model. As a result, the amount of information required is considerably smaller, but of course at the price of freedom in the interface design. This approach seems to be much more interesting, so we focus on our reflections on it.

What we have written so far, sounds great, but how to put it into practice? It seems that the key to success is finding the balance between our engagement (providing details) and the end result matching our expectations - just like the figure below. We see that the end result is heavily dependent on the amount of work involved. Of course, the goal of using this approach is approaching this man on the left ;)



Of course, if the system needs too much information (necessary to adapt it to specific needs) it will be more difficult to use than the classic GUI. Consequently, to reduce our workload we have to accept a certain "universality" of the solution, ie the generated GUIs will be quite similar to each other. However, in typical cases, they can be very useful, such as all kinds of business applications where many forms (windows) are used.

The assumptions of the proposed declarative approach can be formulated as follows:

- The programmer specifies elements of the Java data model (classes) that should have GUIs created for attributes and methods;
- The programmer calls the method that displays the form (window) for the given object.
- The system itself generates forms that allow you to create new data instances, edit them, etc.
- There is an option to further refine details, including: text labels, specific types of widgets;
- We mainly use simple types (but not only): numbers and texts, true/false, date, enumeration type.

We need to solve a few problems to implement this approach. Subsequent sections will be devoted to them.

Read the structure of the business class

Actually, the only way to read class constructs is to use a technology called reflection. It provides information on class structures, methods, attributes, etc. It also allows you to create object instances, call methods, and modify and read attribute content. The key is the Java class named `Class` and its methods: `getDeclaredFields()` and `getDeclaredMethods()`.

Defining interface elements

The next question is how to define the model elements to generate GUI controls? probably we would not want to do it for all items. It is also hard to imagine some clever algorithm which can automatically determin it for the general case. Actually, we have the following possibilities:

- Configuration file, such as XML, or Java properties;
- passing the appropriate parameters to the method;
- use Java annotation. They can be defined by the programmer and affect almost all language constructs, including classes, attributes, methods. The popular example is `@Override`

  ```
  Java

  @Override
  public String toString() {
          return "My own implementation of toString()";
  }
  ```

  During the operation of the program we have the ability to read them (eg for an attribute) and react (eg generate a control).
- Development of dedicated domain language (*Domain Specific Language - DSL*). By creating your own DSL embedded in the host language (eg Java), you need to decide they will interact with each other. This may be plain text, such as SQL in JDBC. This solution obviously has a number of drawbacks (eg lack of typological control). That is why it is better to use the so-called *Fluent API*. The idea is to "cleverly" define classes, methods, and attributes so that their calls would look like DSL commands (see below).

Later in this lesson we will show examples of annotations and DSL solutions.

Customizing the Generated GUI

As we mentioned, we will create appropriate annotations to determine which model elements are to have generated GUIs. They must have certain parameters to customize the generated widgets. Of course, the number of these parameters should be as small as possible. Moreover, in order to further facilitate the use of the created annotations, the parameters will have default values. By introducing a breakdown into attributes and methods, we can individualize the annotation functionality - the set of parameters will be similar but the default values are different. Let's look at the possible annotation parameters:

- `label`. Describes the label attached to the widget. If it is empty (the default value), then the name of the attribute or method will be used. This parameter is required because sometimes we need to enter special names (eg, space or national characters);

- `widgetClass`. A widget class that will be used to edit and display data. The default value is the use of a text box (`JTextBox`);
- `tooltip`. Short text displayed when the cursor is placed on an element.
- `getMethod`. The method used to read attribute values. The default value contains an empty string and indicates a getter and setter approach;
- `setMethod`. Same as for getMethod, but for writing.
- `showInFields`. Flag indicating whether this element should be visible in the field view;
- `showInTable`. Flag indicating whether this element should be visible in tabular view;
- `showInSearch`. A flag indicating whether this item should be visible in the search view.
- `order`. The number defining the order of the widget in the form;
- `readOnly`. A flag that defines whether data is read-only (without modification);
- `scaleWidget`. Specifies whether the widget should change its size when modifying the form size.

Linking GUI controls with business model

We need to determine which set of controls will support our library. The simplest and most universal approach is using `JTextBox`. Of course, they can be improved. Using the special `ComboBox` to work with the enumeration type, `CheckBox` supports boolean type and any other - defined by the programmer.

Another issue is the transfer of data to/from the control. Because our solution is generic (ie capable of working with different types of data) we need to establish a common "type" of data being sent: of course, this will be text (`String`). We require each of the controls used to handle two methods: `void setText(String)` - write data to the control and `String getText()` - read them from the control. Fortunately, most of the controls provided by Swing support these methods. Sometimes we need to create a simple wrapper for the existing control so that it works with the above methods. In a few situations, you may need to make significant modifications or implement your own control.

Examples for *senseGui*

Below are examples of ***senseGUI*** library created by me as part of my research conducted in PJATK. It allows you to generate GUI based on annotated Java classes. We will not discuss its implementation in detail (I used the ideas discussed above), but of course I encourage you to experiment and consult with me.

Let's say we have a simple business class for which we need a GUI:

```Java

public class Person {
        private String firstName;
        private String lastName;
        private Date birthDate;
        private boolean higherEducation;
        private String remarks;
        private int SSN;
        private double annualIncome;

        public int getAge() {
                // [...]
        }
}
```

To use *senseGUI*, just decorate with the annotations and call up a simple method. Notice that we are dealing with so-called POJO without any special interfaces or superclasses.

```Java

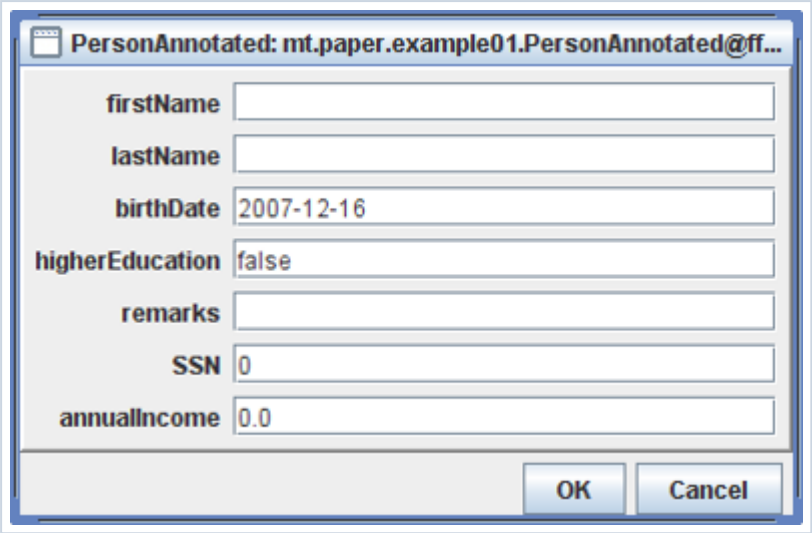
public class PersonAnnotated {
        @GUIGenerateAttribute
        private String firstName;
        @GUIGenerateAttribute
        private String lastName;
        @GUIGenerateAttribute
        private Date birthDate = new Date();
        @GUIGenerateAttribute
        private boolean higherEducation;
        @GUIGenerateAttribute
        private String remarks;
        @GUIGenerateAttribute
        private int SSN;
        @GUIGenerateAttribute
        private double annualIncome;
        @GUIGenerateMethod
        public int getAge() {
        // ...
        }
        // Standard getters/setters methods
}
```

As a result of running the following code we get a ready-made pane to display and edit the business object.

```Java

JDesktopPane jdp = new JDesktopPane();

PersonAnnotated person = new PersonAnnotated();
SingleObjectPlusFrame personFrame = GUIFactory.getObjectFrame(person, jdp, false, true,
true);
```

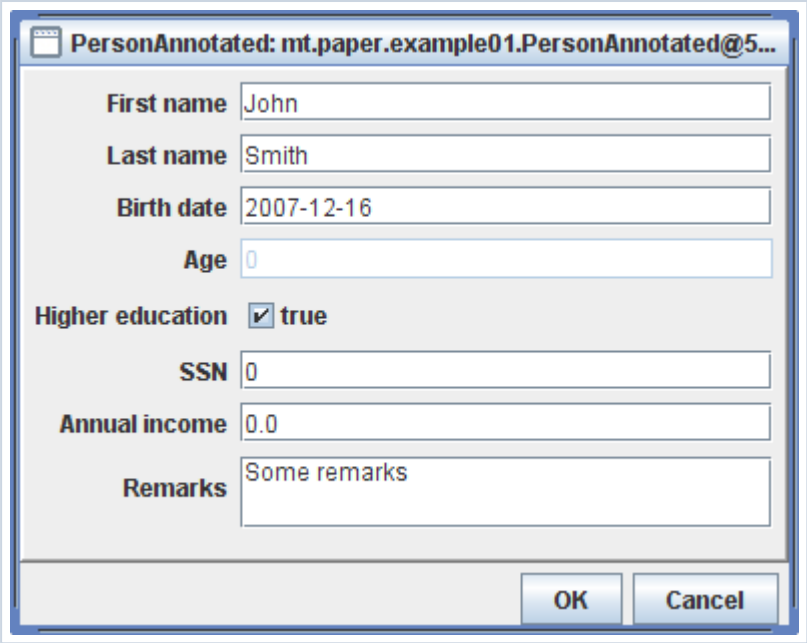


As for one line of code (#4), the effect is good, but we can correct it by modifying the default annotation parameters. Thanks to that we have valid attribute names, proper order, the age is read only, field *Remarks* is scaled, and boolean is displayed in the *checkbox*.

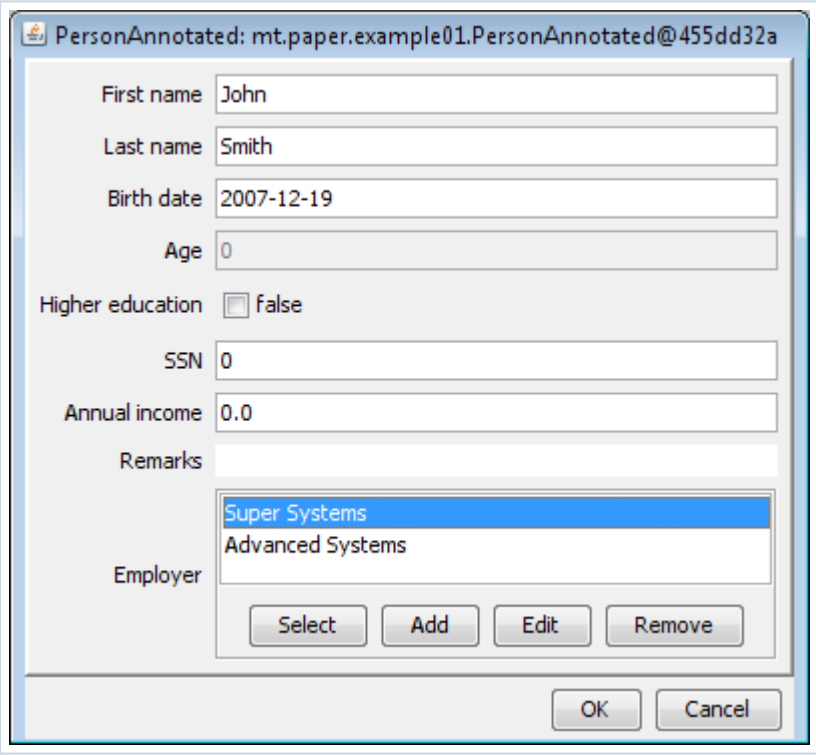
```Java

public class PersonAnnotated {
        @GUIGenerateAttribute(label = "First name", order = 1)
        private String firstName;
        @GUIGenerateAttribute(label = "Last name", order = 2)
        private String lastName;
        @GUIGenerateAttribute(label = "Birth date", order = 3)
        private Date birthDate = new Date();
        @GUIGenerateAttribute(label = "Higher education",
widgetClass="mt.mas.GUI.CheckboxBoolean", order = 5)
        private boolean higherEducation;
        @GUIGenerateAttribute(label = "Remarks", order = 50,
        widgetClass="javax.swing.JTextArea",scaleWidget=false)
        private String remarks;
        @GUIGenerateAttribute(order = 6)
        private int SSN;
        @GUIGenerateAttribute(label= "Annual income", order=7)
        private double annualIncome;
        @GUIGenerateMethod(label="Age", showInFields = true, order = 4)
        public int getAge() { ...}

        // Standard getters/setters methods
}
```



It is also possible to combine *senseGUI* with the previously created *ObjectPlus* library. This will allow us to manipulate extent, linked objects and search information. Sample window (of course, also automatically generated):



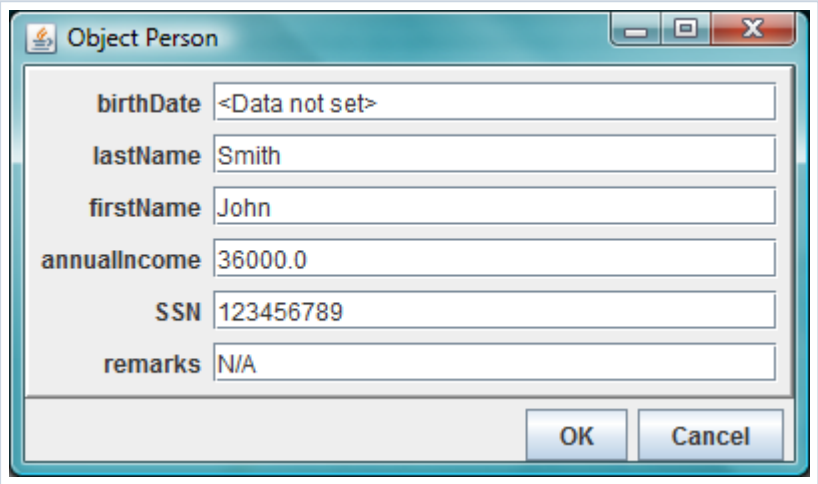
Example utilization of *GCL*

Another solution I've created is GCL (GUI Creating Language). Its effects are similar, but instead of using annotations we use dedicated DSL (and the *Fluent API* concept). This approach may be better when, for example, we want to display different interfaces for the same business class (writing other DSL commands). Of course, I encourage you to experiment and analyse the attached sources.

We will reuse the usual `Person` class, defined above. To view the GUI for editing, simply write the following code snippet. Let's take a look at how to combine each of our DSL commands - reading it together makes sense: `create.frame.usingOnly(person)` and of course the correct part of Java. On the other hand, if someone has taken a look at the implementation, they will see the strange names of the constructs, such as the `usingOnly (...)` method.

```
Java

JFrame frame1 = create.
                                frame.
                                usingOnly(person);
```



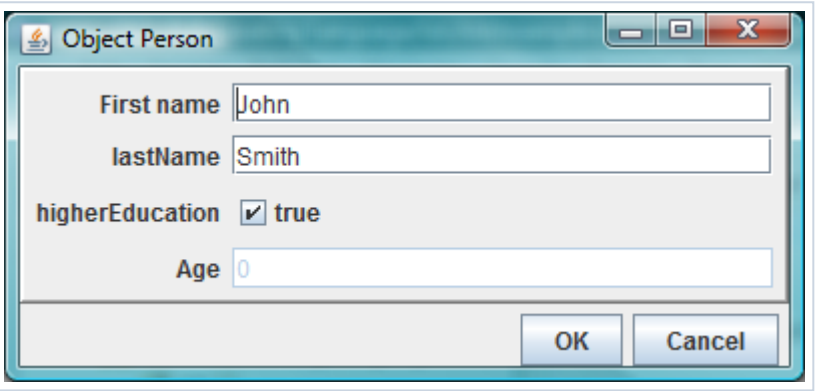


COMMENTARY

The **Fluent API** is based on returning the same (or similar) object type in consecuive calls. In addition, we use specific naming of individual constructs (classes, methods, attributes). This makes it possible to write language commands that are readable. Of course, at the same time we have strong typological control, IDE hints, etc. Examples of such approaches are the jQuery library, one of the ways to use LINQ and my GCL.

And here is another example - this time we changed the look of our window and added a validator for the `lastName` attribute:

```java
JFrame frame =  create.
                              frame.
                              using(person).
                              containing(
                                                attribute("firstName").as("First name"),
                                                attribute("lastName").validate(new
ValidatorNotEmpty()),
                                                attribute("higherEducation"),
                                                method("getAge").as("Age"));
```



KNOWLEDGE CHECK — Which statements are true?

- Declarative GUI creation is an absolute novelty and no commercial company provides such capabilities.
- Semantic declarations make it possible to work on a higher level of abstraction than a component.

EXPLANATION & ANSWERS: **Explanation:**

- There are several commercial implementations of declarative GUIs, such as XAML.
- Semantic declarations use a data model, not GUI controls. For this reason, we can talk about a higher level of abstraction.