Design and Analysis of Information Systems (MAS)

Mariusz Trzaska, Ph. D.
mtrzaska@pjwstk.edu.pl

Lecture 05

Associations in Object-Oriented Programming Languages (1)

Outline

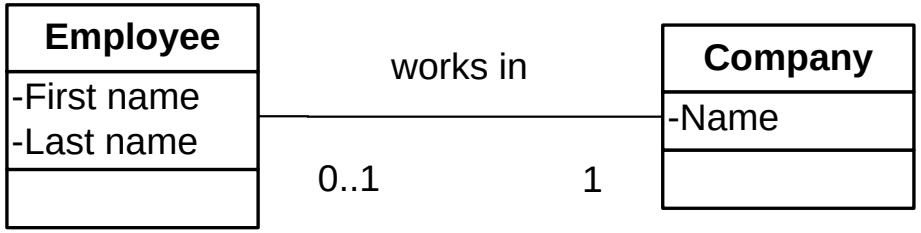
- Introduction
- Implementation of the associations using:
  - identifiers,
  - references.
- Implementation of the associations:
  - In relation to cardinalities,
  - binary,
  - attribute association,
  - qualified,
  - *n-ary,*
- *Implementation of an aggregation,*
- *Implementation of a composition,*
- *Generic associations management,*
- *Summary*

Links and Associations

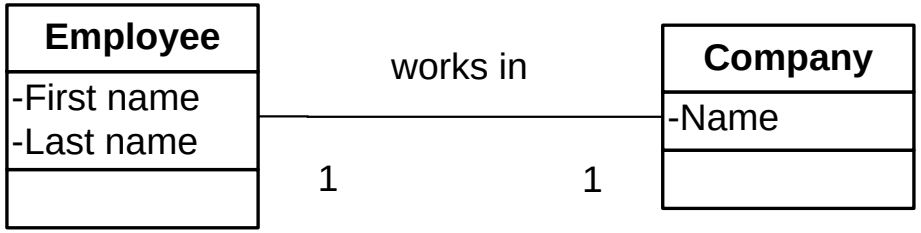
- **A link**. A dependency between objects.
- Binary links.
- **An Association**. Description of a group of links sharing the same semantics and structure.
- Used to describing class dependencies.
- A link is an instance of an association (similarly to an object and its class).

Cardinalities

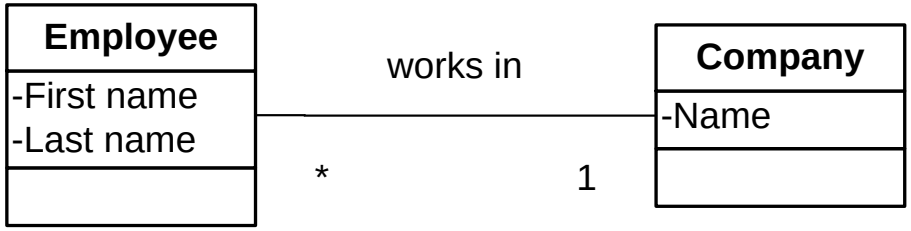
- 1 to 0, 1



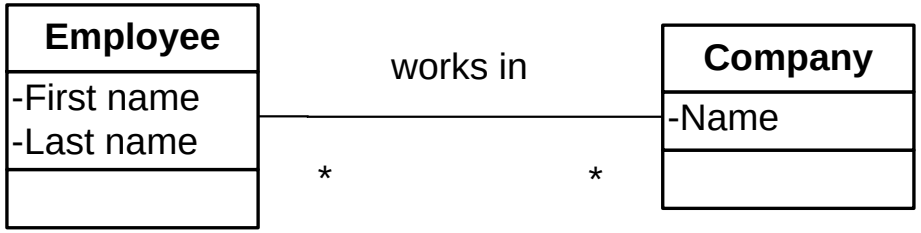
- 1 to 1



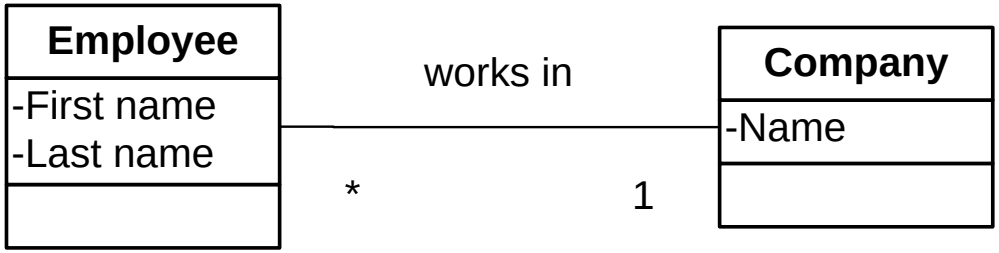
- 1 to *

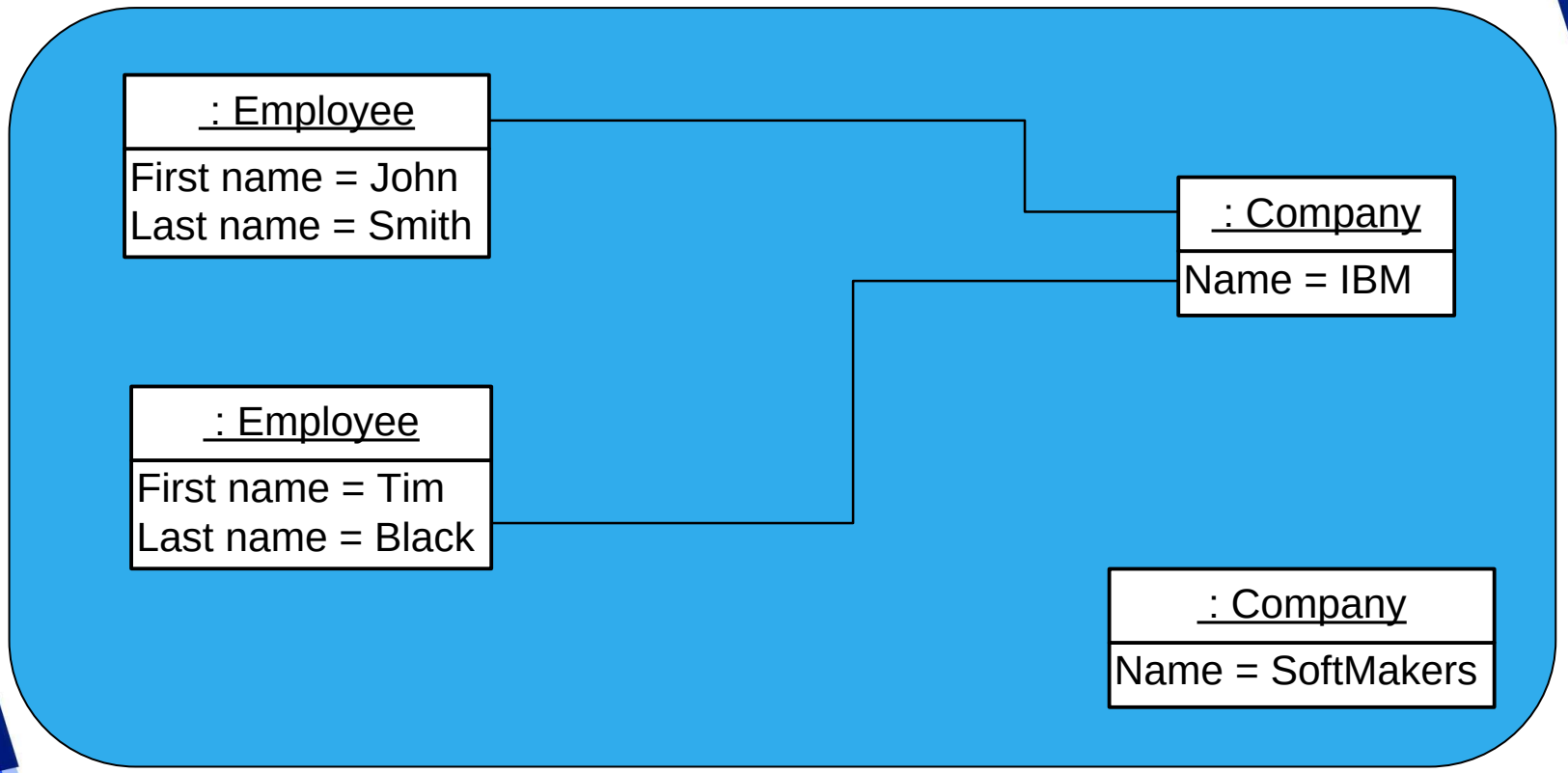


- * to *



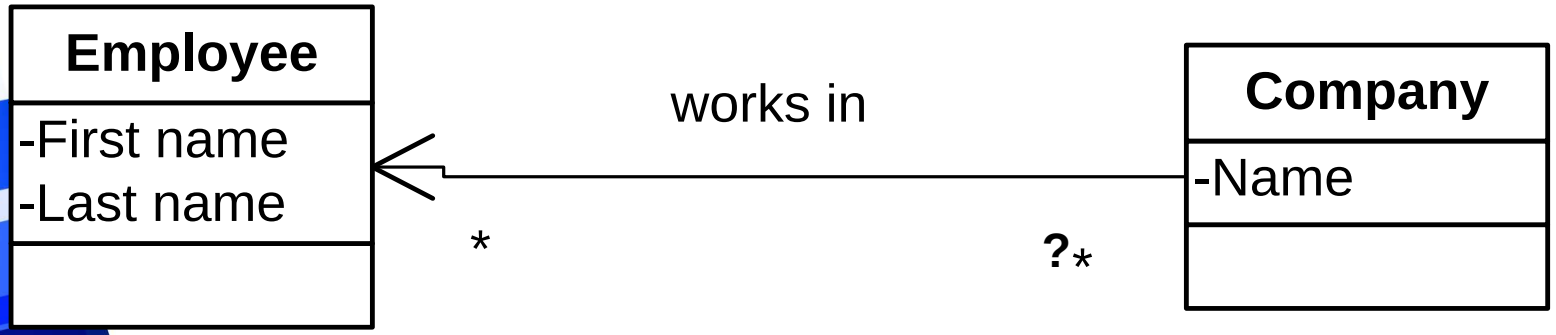
Links and Associations – an Example





Directed (unidirectional) association

- Business information (dependency) is stored only in a single direction.
- Purpose?
- Usability?

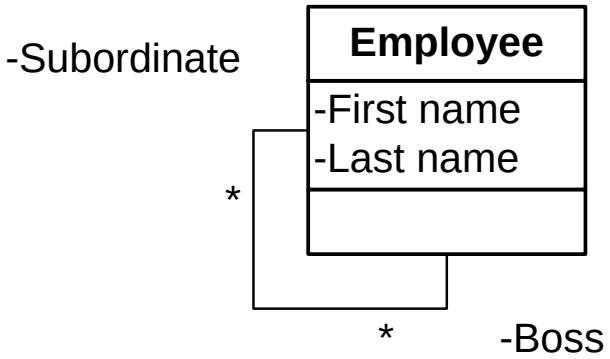


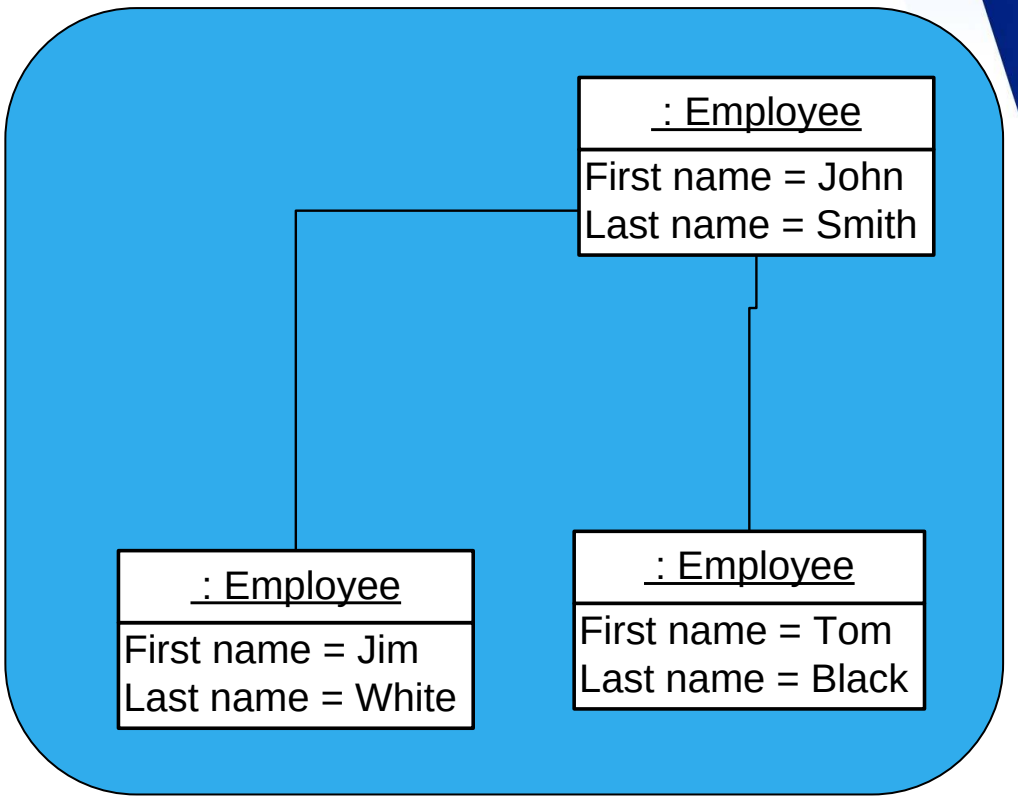
Association Roles

- An association can have roles with dedicated names.
- Naming convention
  - An association name: works in,
  - A role name: employer
- Roles
  - Optional (when),
  - Mandatory (when)?
- Usability during an implementation.

Recursive Association

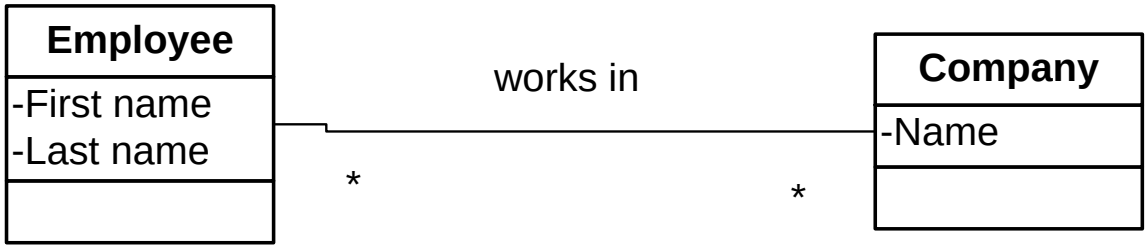
- In the same class





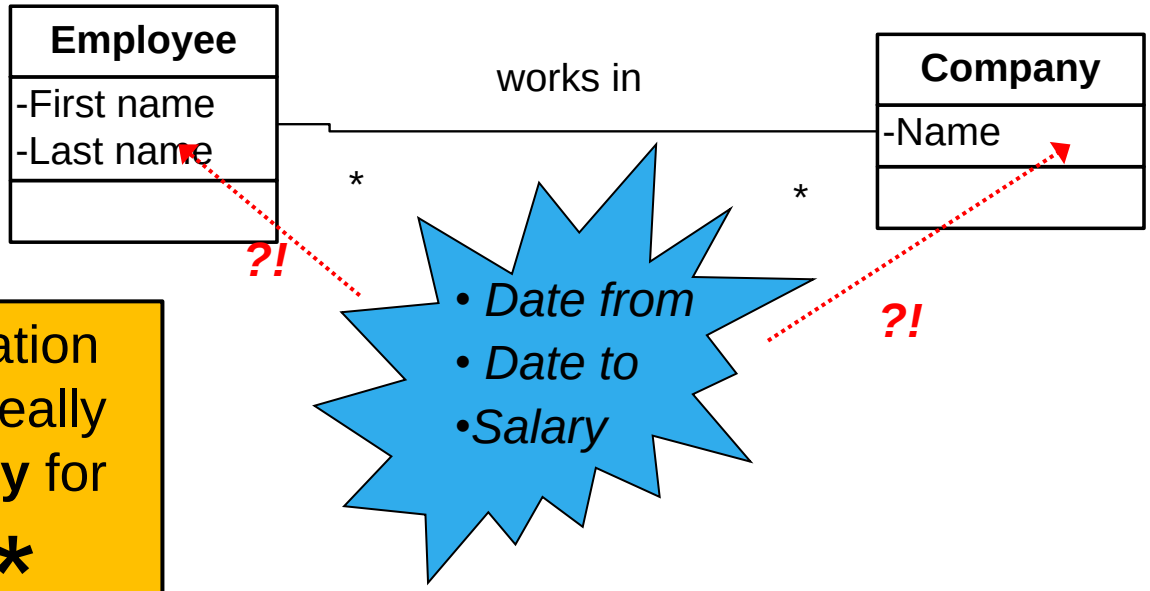
Association Class

- Let's assume that we have the following business case:



- We need to remember:
  - When an employee worked in a particular company,
  - How much did he/she earn.
- How and where we should put the information?

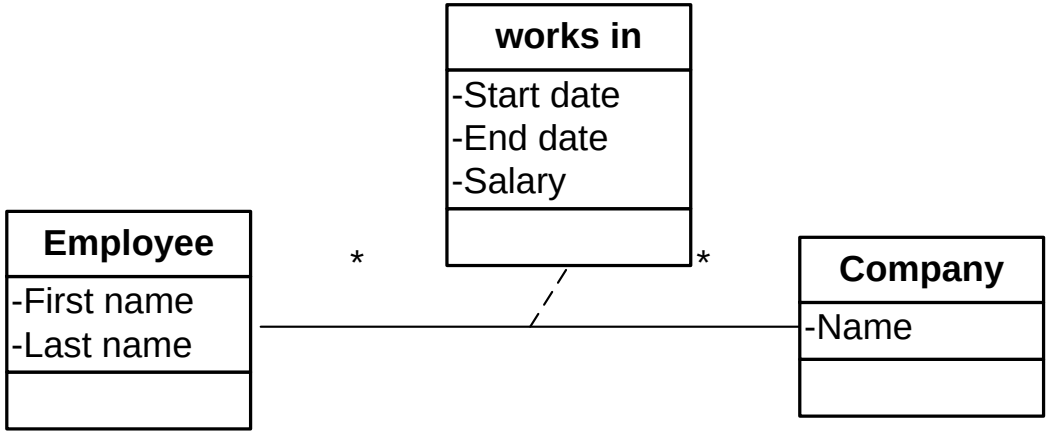
Association Class (2)



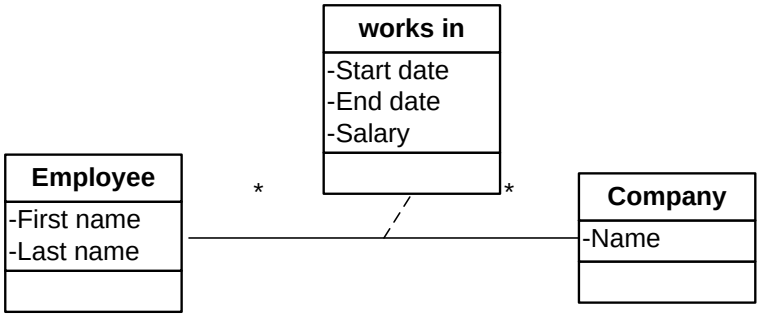
The association attribute is really **needed only** for

*** – *** cardinalities

- The solution: an association class.

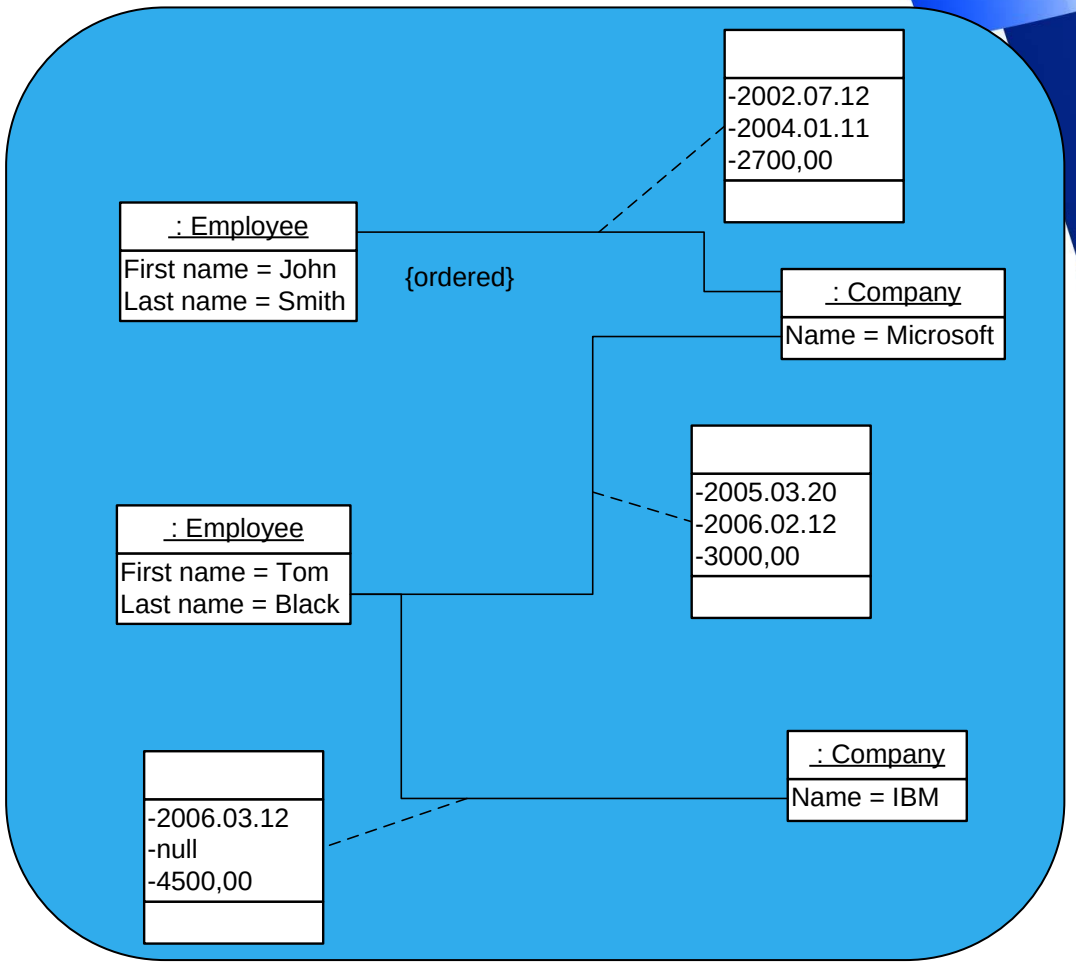


Association Class (3)



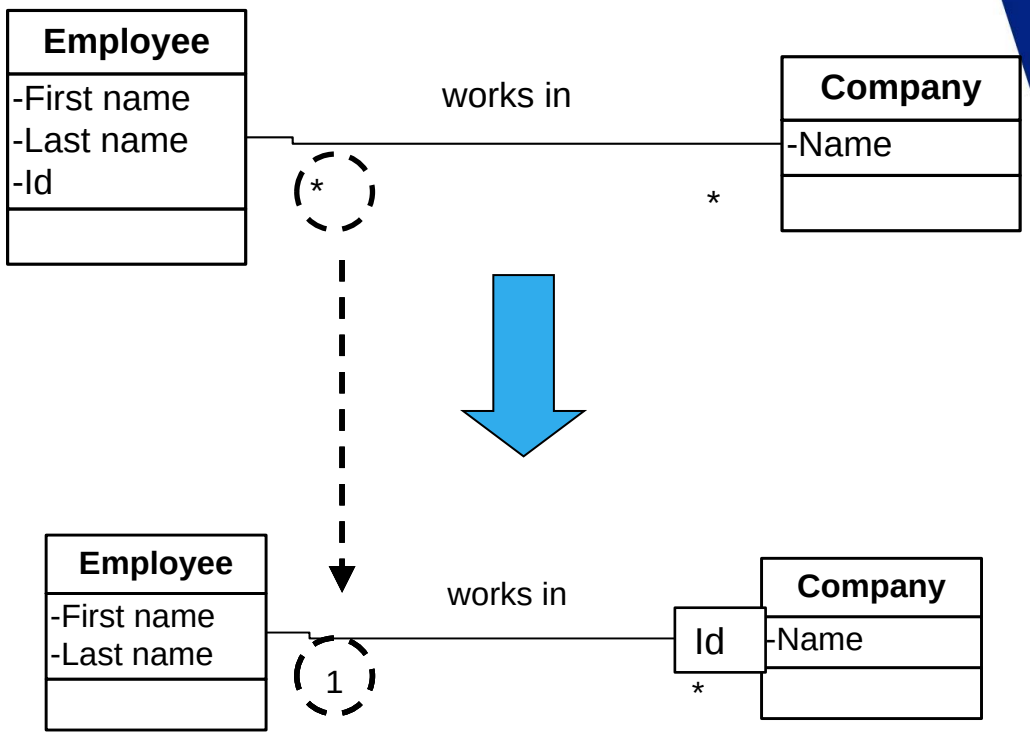
- The class diagram

- The object diagram



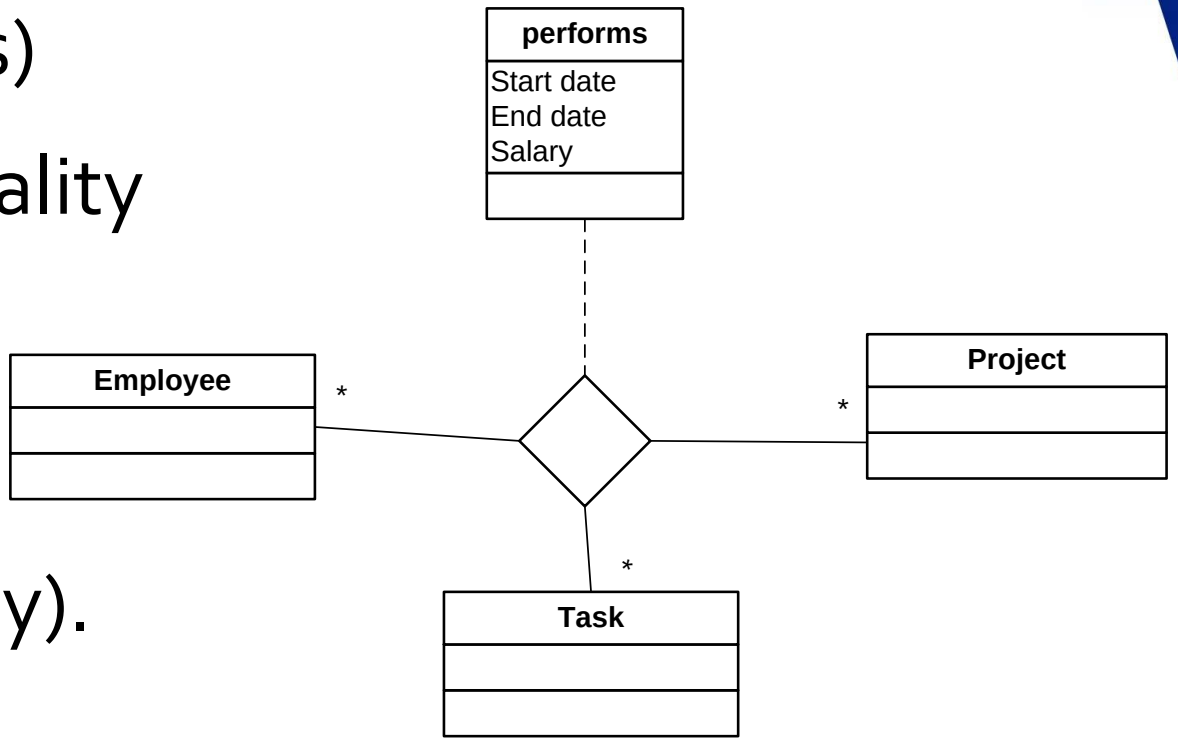
Qualified Association

- **Qualifier**: an attribute or a set of attributes explicitly defining the target object of the association.
- Allows quick access to the object based on the qualifier.
- Indexing (?)



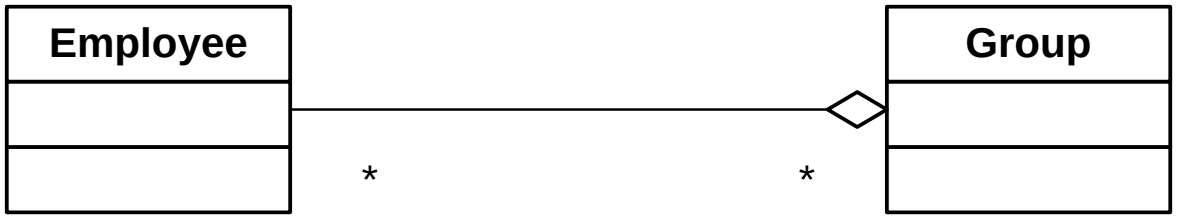
N-ary Association

- An association connecting n classes.
- Some conceptual problems (with cardinalities)
- Each cardinality should be more than 1 (many).



Aggregation

- An association describing the whole – part dependency:
  - Consists of,
  - Belongs to,
  - Is member of,
  - etc.
- All properties of an „ordinary" association.
- Because of the specific dependency the name should be omitted.
- There are no consequences among connected objects.

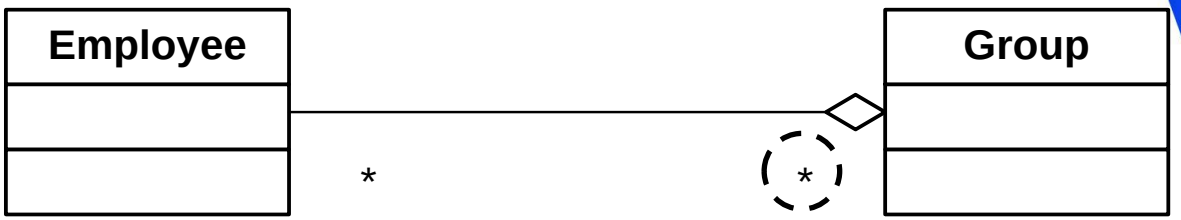


Composition

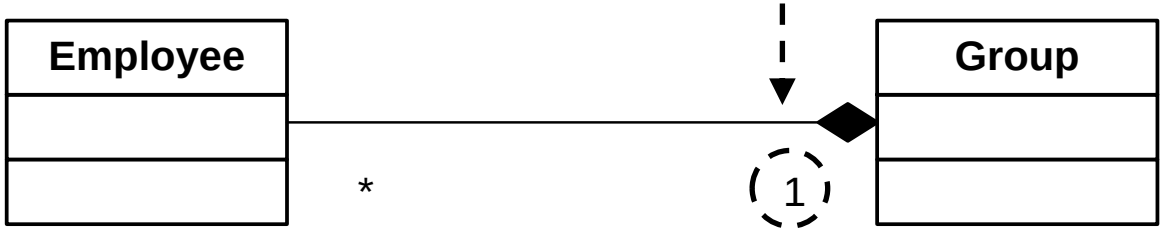
- A composition is a stronger version of an aggregation.
- Hence it is also an association.
- Utilization of the composition causes some consequences:
  - A part cannot be shared (cardinalities),
  - A part cannot exist without the whole,
  - Removing the whole means removing all its parts.

Composition (2)

- An aggregation



- A composition



Associations and Programming Languages

- How the term is related to popular programming languages?
- In languages:
  - Java,
  - MS C#,
  - C++

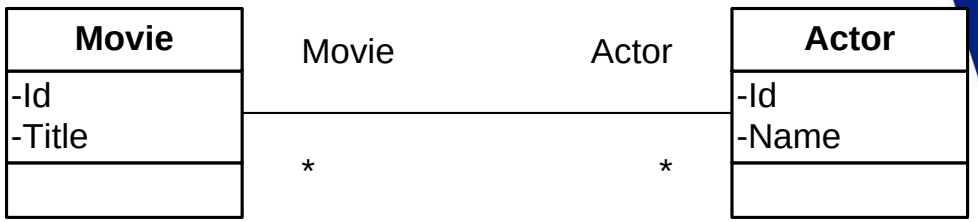
  associations do not exist.
- Fortunately it is possible to implement them manually.

Implementation of Associations

- We can use two approaches. The basic difference is related to the way of storing information about links among objects:
  - Identifiers, i.e. numbers,
  - References (Java, C#) or pointers (C++).
- Which one is better?

The Identifiers Approach

- Add an attribute which will be an identifier, i.e. `int` number.
- Information about linked objects is stored using their identifiers (be careful with cardinalities).
- Necessity of creating pairs of ids (for bidirectional associations).



:Movie	: Movie	:Actor	: Actor	: Actor
Id = 1	Id = 2	Id = 3	Id = 4	Id = 5
Title= „T1”	Title = „T3”	Name = „AS”	Name = „MB”	Name = „KL”
Actor = [3, 4]	**Actor = [3, 5]**	**Movie = [1, 2]**	**Movie = [1]**	**Movie = [2]**

The Identifiers Approach (2)

```
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);
   }
}
```

The Identifiers Approach (3)

```
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);
   }
}
```

The Identifiers Approach (4)

```
public static void testIdAssociations() throws Exception {
   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("   " + Movie.findMovie(actor1.movieIds[i]).title);
   }
}
```

```
T1
  AS
  MB
AS
  T1
  T3
```

The Identifiers Approach (5)

- Disadvantages:
  - Necessity of searching an object based on the id – performance problems. The performance could be improved with a map container rather then a list-like e.g.:

```
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);

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

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

  - Anyway we still need to search…

The Identifiers Approach (6)

- Advantages (actually just one (?), but sometimes very important):
  - All objects are independent from each other (from the JVM point of view).
  - Because we do not use references.
  - It is extremely important in some cases, i.e. reading just one object from a DB or transferring through a network:
    - The transferred object is created but without linked objects,
    - It could be designed in such a way that only in case of accessing an object with particular id, the linked object will be created.
    - We do not need to process entire graph of objects.
- In most cases, this approach should ba avoided.

The Reference Approach

- We will use references (Java, C#) or pointers (C++) for connecting objects.
- There is no need of searching for an object;
- Using a reference we have an instant access to it (this is the fastest possible way);
- Necessity of creating references' pairs (if we need a bidirectional connection – usually we do).

The Reference Approach (2)

- Depending on the cardinality of the association:
  - 1 to 1. A single reference on each side.

```
public class Actor {
      public String name;
      public Movie movie; // impl. Asoc., card 1
      public Actor(String name, Movie movie) {
            this.name = name;
            this.movie = movie;
      }
}
public class Movie {
      public String title;
      public Actor actor; // impl. Asoc., card 1
      public Movie(String title, Actor actor) {
            this.title = title;
            this.actor = actor;
      }
}
```

The Reference Approach (3)

- Depending on the cardinality of the association:
  - 1 to *. A single association and a container.

```
public class Actor {
      public String name;
      public Movie movie; // impl. Asoc., card 1
      public Actor(String name, Movie movie) {
            this.name = name;
            this.movie = movie;
      }
}

public class Movie {
      public String title;
      public List<Actor> actor; // impl. Asoc., card. *
      public Movie(String title) {
            this.title = title;
      }
}
```

The Reference Approach (4)

- Depending on the cardinality of the association:
  - * to *. Two containers.

```
public class Actor {
      public String name;
      public List<Movie> movie; // impl. Asoc., card. *
      public Actor(String name) {
            this.name = name;
      }
}

public class Movie {
      public String title;
      public List<Actor> actor; // impl. Asoc., card. *
      public Movie(String title) {
            this.title = title;
      }
}
```

The Improved Links Management

- The presented approach required manual adding of the reverse connection's information.
- It is worth automating.
- Let's create a method which add a link's information in:
  - The main class,
  - Target (connected) class.
- It has to be carefully designed to avoid never-ending execution.

The Improved Links Management (2)

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

   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;
   }
}
```

The Improved Links Management (3)

```
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;
   }
}
```

The Improved Links Management (4)

```
public static void testRefAssociations() throws Exception {
   // Create new business objects (without connections)
   var movie1 = new mt.mas.asocjacjeref.Movie("T1");
   var movie2 = new mt.mas.asocjacjeref.Movie("T3");

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

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

   // 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);
}
```

```
Movie: T1
  AS
  MB

Movie: T3
  AS
  KL

Actor: AS
  T1
  T3

Actor: MB
  T1

Actor: KL
  T3
```

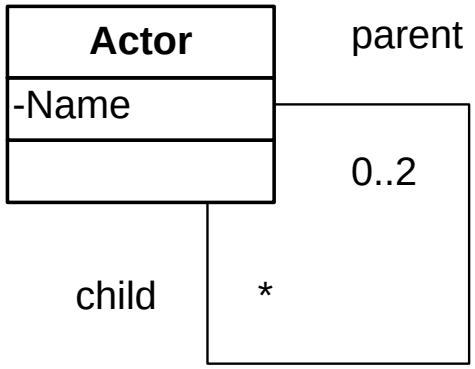
The Improved Links Management (5)

- Be careful with „1” cardinality:
  - There will be a setter (e.g. `setFilm`) on the „1” side (instead of `addFilm`).
  - Before we will insert the new value, we have to check if the current one is different then `null`.
  - If yes, then we need to remove the existing link **on both sides**.
  - Next we can create a new connection by setting the reference (the „1” side) and by adding a new one (the „*” side).
  - Remember about potential „self-looping”.

Implementation of a Directed Association

- The diagram means that for:
  - The particular movie we would like to know its actors,
  - The particular actor we do not need his movies.
- The implementation is very similar to the previous cases but the link information is stored only in one class:
  - There is a special container in the movie class,
  - There is no link container in the actor class.

Implementation of the recursive association

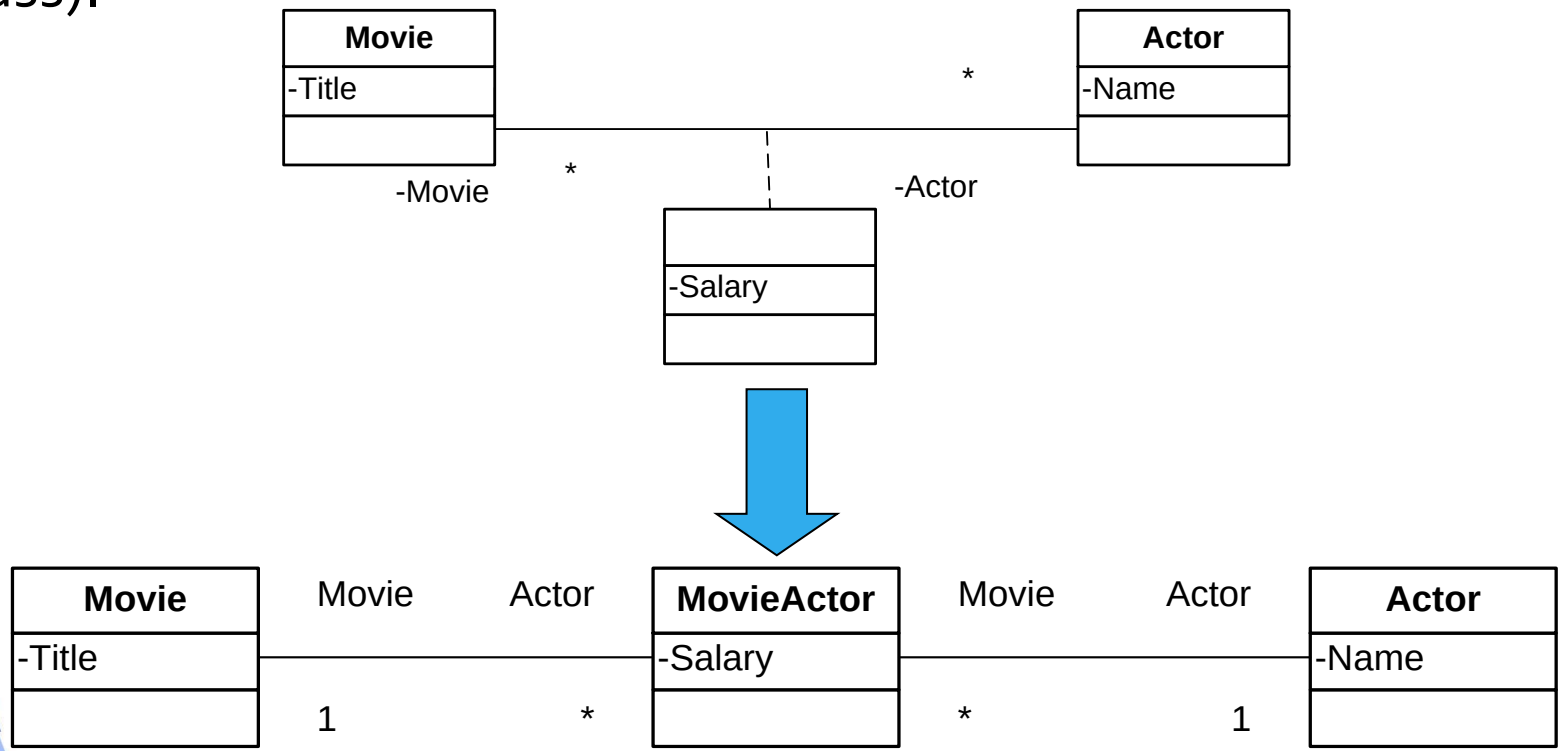


- The implementation is based on the same rules.
- However there are two containers in in the class (rather then one), storing information about each association role.

```
public class Actor {

   // [...]

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

   // [...]
}
```

Implementation of the class association

- At the beginning we need to transform:
  - One UML construct (an association with an attribute)
  - Into another UML construct (an association with the middle-class).

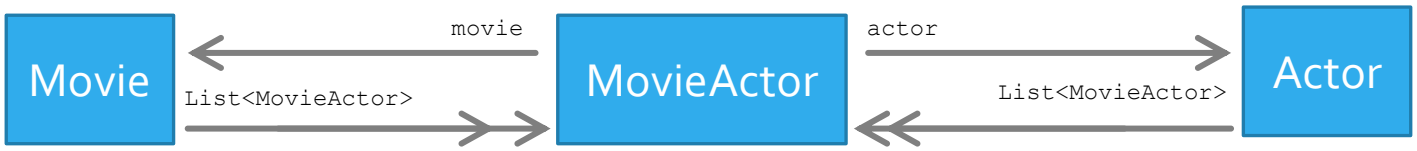


Implementation of the class association (2)

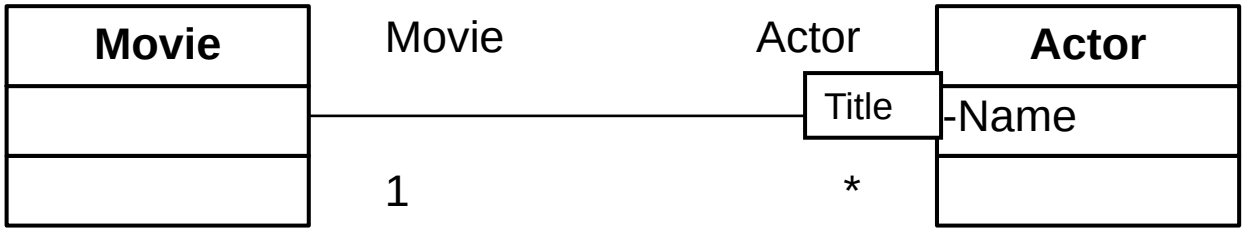
- Thanks to the transformation we got two „ordinary” associations.
- The „new” associations can be implemented in one of the described way.
- Possible problems with the semantics of the new middle class:
  - Name?
  - Roles' names: „old” and „new”
- More difficult access to the target objects (through the middle-class instance).

Implementation of the class association (3)

- Should the middle class have its own extent?
- It's a good idea to create a **special constructor** for the middle class:
  - parameters: references to external objects and business data,
  - local storage of information,
  - creating the required connections.



Implementation of the Qualified Association



- The simplest implementation:
  - Using the existing approach,
  - Adding a method, which using the Title gets an instance of the class Movie,
  - Poor performance.
- Better solution:
  - Do not use the `ArrayList` (or other „ordinary collection”),
  - Use a `Map`, where a *key* will be the title and a *value* will be an object describing connected movie.
  - The reverse information using „ordinary” approach.

Implementation of the Qualified Association (2)

```
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);
   }

   // [...]
}
```

Implementation of the Qualified Association (3)

```
public static void testQualifiedAssociations() throws Exception {
   // 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
   actor1.addMovieQualif(movie1);
   actor1.addMovieQualif(movie2);
   actor2.addMovieQualif(movie1);
   actor3.addMovieQualif(movie2);

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

   // Get the info about the "T1" movie for the actor1
   var movie = actor1.findMovieQualif("T1");
   System.out.println(movie);
}
```

```
Actor: AS
  T1
  T3

Actor: MB
  T1

Actor: KL
  T3

Movie: T1
  AS
  MB
```

To be continued…

Source files

Download source files for all MAS lectures

http://www.mtrzaska.com/plik/mas/mas-source-files-lectures