

COMP353 Databases

Database Design: Object Definition Language (ODL)

ODL

- **ODL** (Object Definition Language) is a standard text-based language for describing the structure of databases
- **ODL** is an extension of **IDL** (Interface Description Language), a component of **CORBA** (Common Object Request Broker Architecture)

Object Oriented World

- In an object oriented design, the **"world"** we want to model is thought of as being **composed of objects**
- Everything is an **object**
 - *people*
 - *bank accounts*
 - *airline flights*
- Every **object** has a **unique object id (OID)**
- Every **object** is an **instance of a class**
- A **class** simply represents a grouping of **similar objects**
- All **objects** that are instances of the same class have the same **properties and behaviors**

Class Declarations

- A declaration of a **class** in ODL consists of:
 - The keyword **class**
 - The **name** of the class
 - A bracketed { ... } **list of properties** of the class

```
class <name> {  
    <list of properties>  
};  
  
class Movie {  
    ...  
};
```

Properties of ODL classes

- **ODL classes** can have three kinds of properties:
 - **Attributes**
 - properties whose types are built from **primitive/basic types** such as integers, strings,...
 - **Relationships**
 - properties whose type is either a **reference** to an object or a **collection** of such references
 - **Methods**
 - **functions** that may be applied to objects of the class

Attributes in ODL

- **Attributes** are the **simplest kinds** of properties
- An **attribute** **describes some aspect of an object** by associating, with the object, a value of some **simple type**
- For example, attributes of a **Student** object
 - Student ID
 - Name
 - Address
 - E-mail

Keys in ODL

- In ODL, we declare keys using the keyword **key**
 - If a key has more than one attribute, we surround them by (...)
 - Example: (two attributes forming a key)

```
class Movie
    (extent Movies key (title, year) ) {
        attribute string title;
        ...
    };
```
 - If a class has > one key, we may list them all, separated by commas
 - Example: (A class with two keys)

```
class Employee
    (extent Employees key empID, SIN) {...};
```

Single-Value Constraints in ODL

- Often, we **should enforce** properties in the database saying that there is **at most one value** playing a particular role
 - For example:
 - that a movie object has a **unique** title, year, length, etc
 - that a movie is owned by a **unique** studio

Single-Value Constraints

- In ODL:
 - An **attribute is not of a collection type** (Set, Bag, Array, List, Dictionary are **collection types**.)
 - A relationship is either a class type or (a single use of) a collection type constructor applied to a class type.
- Recall that in the **E/R notation**:
 - **attributes are atomic**
 - an **arrow (→)** can be used to express the multiplicity of relationships (1:1), (1:M), and (N:M)

Type system

- A **type system** consists of
 - **Basic types**
 - **Type constructors**
 - recursive rules whereby **complex types** are built from simpler ones

Basis of types in ODL

- **Primitive types (atomic)**
 - Integer
 - Float
 - Char
 - Character String
 - Boolean
 - Date
 - Enumeration (a list of names declared to be **synonyms for integers**)
- **Class types**
 - Movie

Type constructors in ODL

- **Set**
 - Set <integer>
 - Set <Movie>
- **Bag**
 - Bag <integer>
 - Bag <Movie>
- **Array**
 - Array <integer, 10>
 - Array <Movie, 3>
- **Structure**
 - Struct Address {string street, string city}
- **List**
 - List <integer>
 - List <Student>
- **Dictionary <keyType, valueType>**
 - Dictionary<Student, string>
- **Note:**
 - Set, Bag, Array, List and Dictionary are called **collection types**
 - Collection type cannot be applied repeatedly (nested)
 - E.g., it is **illegal** to write Set<Array<integer,10>>

Example

```
class Movie {
    attribute string title;
    attribute integer year;
    attribute integer length;
    attribute enum Film {color, blackAndWhite} filmType;
};

("Gone with the Wind", 1939, 231, color) is a Movie object.
```

Example (non-atomic type)

```
class Star {
    attribute string name;
    attribute Struct Address {
        string street,
        Array<char, 10> city
    } homeAddress;
    attribute Address officeAddress;
};
```

Example

```
class Student {
    attribute string ID;
    attribute string lastName;
    attribute string firstName;
    attribute date dob; /* date is a basic type in ODL */
    attribute string program;
    attribute Struct Address {
        string street,
        string city
    } homeAddress;
};
```

Example

```
class Course {
    attribute string courseNumber;
    attribute string courseName;
    attribute integer noOfCredits;
    attribute string department;
};
```

Relationships in ODL

- If we are designing a database about **Movies** and **Stars**, what are we missing? The relationships....
- How are **Movies** and **Stars** related?
- Every movie has a star (or stars)

Example

- Can we write " **attribute Star starOf;** " ?
- ```
class Movie {
 attribute string title;
 attribute integer year;
 attribute integer length;
 attribute enum Film {color, blackAndWhite} filmType;
 attribute Star starOf;
};
```
- No, the attribute types must not be classes

## Example

- **starOf** is a relationship between **Movie** and **Star**

```
class Movie {
 attribute string title;
 attribute integer year;
 attribute integer length;
 attribute enum Film {color, blackAndWhite} filmType;
 relationship Star starOf;
};
```

## Inverse Relationships

- How are **Movies** and **Stars** related?
- Not only every movie has a star but also every star has a role in some movie(s)
- To fix this in the **Star** class, we add the line:  
**relationship Movie starredIn;**

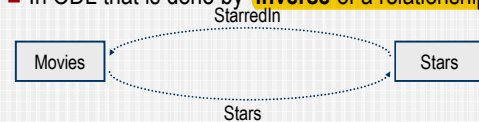
## Example

```
class Star {
 attribute string name;
 attribute Struct Address {
 string street,
 string city
 } address;
 relationship Movie starredIn;
};
```

- What is the problem here?

## Inverse Relationships

- We are **omitting a very important aspect of the relationship between movies and stars**
- We need a way to ensure that if a star **S** is connected to a movie **M** via **stars**, then conversely, **M** is connected to **S** via **starredIn**
- In ODL that is done by **inverse of a relationship**



## Example

```
class Movie {
 attribute string title;
 attribute integer year;
 attribute integer length;
 attribute enum Film {color, blackAndWhite} filmType;
 relationship Star stars
 inverse Star::starredIn;
};
```

## Example

```
class Star {
 attribute string name;
 attribute Struct Address {
 string street,
 string city
 } address;
 relationship Movie starredIn
 inverse Movie::stars;
};
```

## Relationships in ODL

- Our design is missing another important point!
- A movie typically has several stars
- A star usually plays in more than one movie
- To fix this, we write:

**relationship Set<Star> stars;**

## Example

```
class Movie {
 attribute string title;
 attribute integer year;
 attribute integer length;
 attribute enum Film {color, blackAndWhite} filmType;
 relationship Set<Star> stars
 inverse Star::starredIn;
};
```

## Example

```
class Star {
 attribute string name;
 attribute Struct Address {
 string street,
 string city
 } address;
 relationship Set<Movie> starredIn
 inverse Movie::stars;
};
```

## Example

- Suppose we introduce another class, **Studio**, representing the studios, i.e., companies that produce movies

```
class Studio {
 attribute string name;
 attribute string address;
};
```

## Example

- How are **Movies** and **Studios** related?
- Every **Studio** owns several **Movies**

```
class Studio {
 attribute string name;
 attribute string address;
 relationship Set<Movie> owns
 inverse Movie::ownedBy;
};
```

## Example

- What about inverse?
- Every **Movie** is owned by some **Studio**

```
class Movie {
 attribute string title;
 attribute integer year;
 attribute integer length;
 attribute enum Film {color, blackAndWhite} filmType;
 relationship Set<Star> stars inverse Star::starredIn;
 relationship Studio ownedBy inverse Studio::owns;
};
```



## Multiplicity of relationships

- In **general**, when we have a pair of inverse relationships, there are **four** cases:
  - The relationship is unique in both directions (1)
  - The relationship is unique in just one direction (2)
  - The relationship is not unique in any direction (1)
  - The **multiplicity** is thus referred to the kinds of these 4 relationships, also denoted as 1-1 (read as one-one), 1-M (one-many), M-1 (many-one), and M-N (many-many).

## Multiplicity of relationships

- A **many-many** relationship from a class **C** to a class **D** is one in which, for each **C** there is a set of **Ds** associated with **C**, and in the inverse relationship, associated with each **D** is a set of **Cs**

- For example, each student can take many courses and each course can be taken by more than one student

```
class Student {
 ...
 relationship Set<Course> takes inverse Course::takenBy;
};

class Course {
 ...
 relationship Set<Student> takenBy inverse Student:: takes;
};
```

## Multiplicity of relationships

- A **many-one** relationship from class **C** to a class **D**, is one where for each **C** there is a at most one **D**, but no such a constraint in the reverse direction (similarly for one-many)

- For example, many employees may work in the same department, but each employee works only in one department

```
class Employee {
 ...
 relationship Department worksIn inverse Department::workers;
};

class Department {
 ...
 relationship Set< Employee > workers inverse
 Employee::worksIn;
};
```

## Multiplicity of relationships

- A **one-one** relationship from class **C** to class **D** is one that for each **C** there is a at most one **D**, and conversely, for each **D** there is at most one **C**

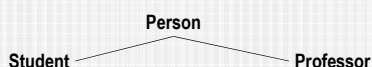
- For example, each department has at most one professor as its chairperson and each professor can be the chair of at most one department

```
class Professor {
 ...
 relationship Department chairOf inverse Department::chair;
};

class Department {
 ...
 relationship Professor chair inverse Professor:: chairOf;
};
```

## Inheritance in Object Oriented World

- Objects can be organized into a hierarchical inheritance/is structure
- A **child class** (or subclass) will inherit properties from a parent class (or all the superclasses) higher in the hierarchy.



## Subclasses in ODL

- Often, a **class** contains some objects that have **special properties** not associated with all members of the class
- If so, we find it useful to organize the class into **subclasses**, each subclass having its **own special** attributes and/or relationships

## Subclasses in ODL

- We define a class **C** to be a subclass of another class **D** by following the name **C** in its declaration with a keyword **extends** and the name **D**

```
class Cartoon extends Movie {
 relationship Set<Star> voices;
};
```

A subclass *inherits* all the properties of its superclasses

So, each cartoon object has *title*, *year*, *length*, *filmType*, and inherits relationships *stars* and *ownedBy* from *Movie*, in addition to its own relationship *voices*.

## Example

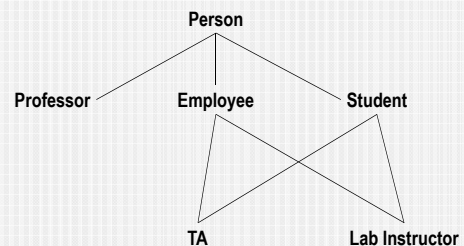
```
class Person {
 attribute string lastName;
 attribute string firstName;
 attribute integer age;
 attribute Struct Address {
 string street,
 string city
 } homeAddress;
};

class Student extends Person {
 attribute string ID;
 attribute string program;
};
```

## Inheritance in ODL

- A class may have more than one subclass.
- A class may have more than one class from which it inherits properties; those classes are its superclasses
- Subclasses may themselves have subclasses, yielding a **hierarchy** of classes where each class inherits the properties of its ancestors.

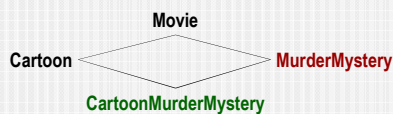
## Multiple Inheritance in ODL



## Example

```
class MurderMystery extends Movie {
 attribute string weapon;
};

class CartoonMurderMystery extends Cartoon : MurderMystery;
```



- Thus, a **CartoonMurderMystery** object is defined to have all the properties of both of its superclasses: **Cartoon** and **MurderMystery**.