Week 5, Lecture 2

Part 1:

ODMG and ODMG's object model

Part 2:

Introduction to Object Definition Language (ODL)

Part 3:

Introduction to Object-Relational Database Management Systems (OR-DBMS)

Part 1

ODMG and ODMG’s object model

PART 1, ODMG and OODBMS, OVERVIEW

- ODMG
- OO concepts and OO-DBMS properties
  - Object identity and object identifier (OID)
  - Objects and values
  - Extent (instances of a class)
  - Complex objects and type constructors
  - Operators
  - Programming language compatibility (match, seamlessness)
  - Encapsulation, information hiding
  - Type/class hierarchies, inheritance and polymorphism

ODMG

- ODMG (@ Jan. 2000, v3.0) – The Object Data Management Group
- See http://www.odmg.org/
  NOTE: ‘Standard Overview’ has a list of all ODMG standards.
- Offers standards for storing (and retrieving) objects.

- ODMG is a close relative of the Object Management Group (OMG).
  See: http://www.omg.org/.

- ODMG offers:
  - Object Management Architecture (OMA) and Object Data Model
  - Object Specification Languages:
    - ODL (Object Definition Language), based upon OMG’s Interface Definition Language (IDL)
    - OIF (Object Interchange Format)
    - OQL (Object Query Language), based upon SQL (as much as possible)
  - Language Bindings: ODL, OML and OQL for C++, Smalltalk and Java
  - Note that there is no other distinct Object Manipulation Language (OML). Manipulation of objects hgappen in the languages C++, Smalltalk and Java.

OO CONCEPTS #1

THE OBJECT-ORIENTED (OO) PARADIGM:

- Intended for modeling a mini-world (the world of interest, often called the Universe of Discourse or UoD) as a collection of communicating/co-operating entities called OBJECTS

OO CONCEPTS #2

- ABSTRACTION AND AUTONOMY:
  - OBJECT: <value, {operators}>
    where the operators are implemented as methods
    and the object is distinct and universally identifiable
  - VALUE: Data-structure
    where a value can be different form other values but not distinct or universally identifiable
  - ENCAPSULATION:
    whereby an object contains and hides information about its internals
  - Requires that other objects “behave” (can’t reach internals)
  - CONTRACT:
    Requires that all objects “behave” (communicate/cooperate) according to agreed upon rules

OO CONCEPTS #3

CLASSIFICATION:

- Common description for all objects belonging to the same class, like a template
- Also called INTENT

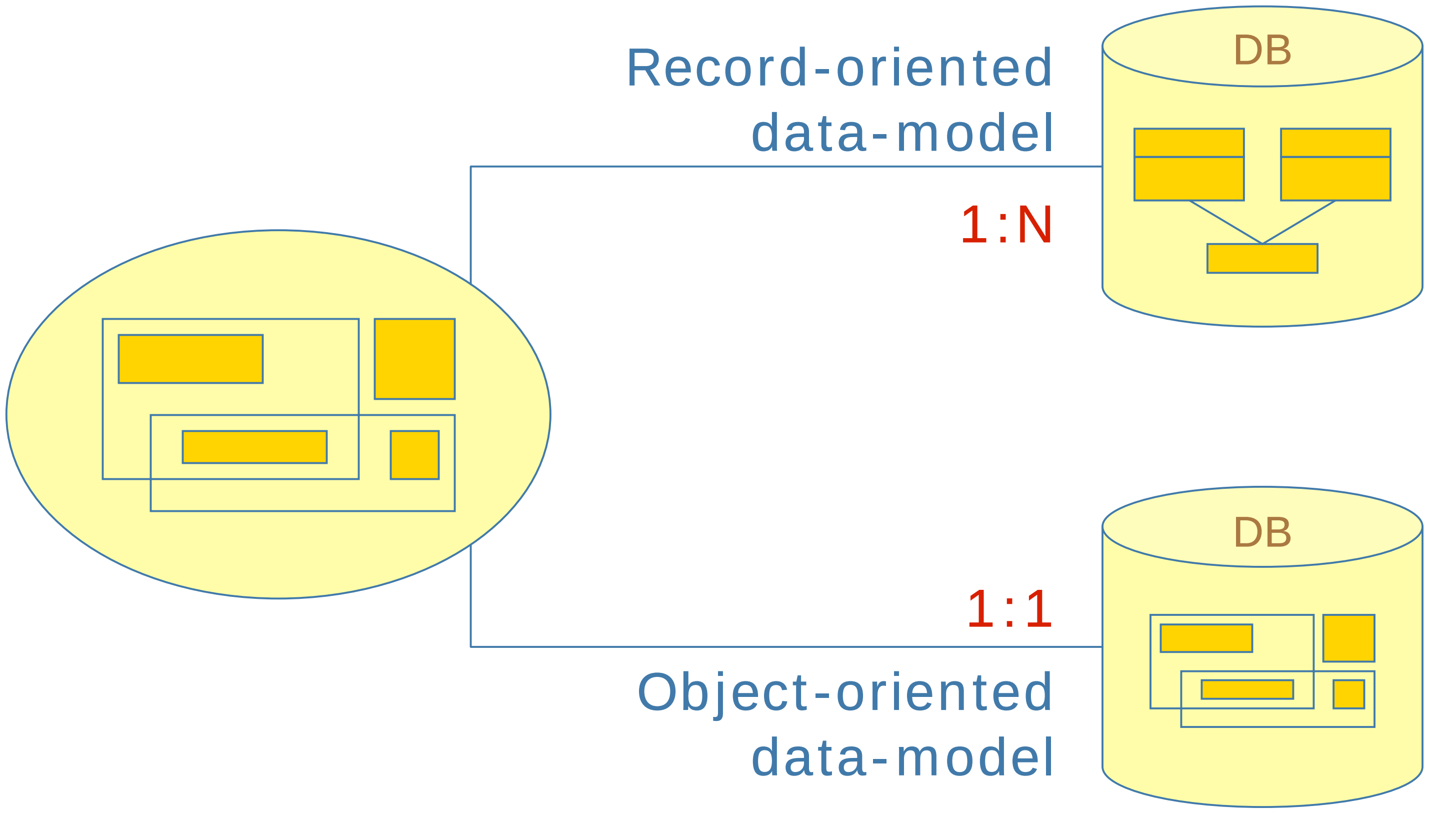
- Can also be though of as a collection of like objects (objects with same properties)
- Also called EXTENT

OO CONCEPTS #4

TAXONOMY

- Super and sub-classes
- Inheritance of properties
- Polymorphism

PURPOSE OF THE OO DATA-MODEL



ADVANTAGES OF OO DATA MODELING

- MORE NATURAL (as compared to traditional data models)
  - Meaningful abstraction, high modularity
  - Better control of complexity
  - Separation of interface and implementation
- EVOLUTIONARY SYSTEMS DESIGN
  - Incremental programming
  - Reuse

OO DBMS

- OO-DBMS:
  DBMS in accordance with the OO DATA MODEL

- OO-database:
  A collection of objects

- An OO-DB object:
  <OID, value, {operations}>

OO DBMS, OBJECTS and OBJECT VALUES

- Object: <OID, value, {operations}>

- Example class:

```
class athlete
{
   text name;
   integer salary;
}
```

- Exampel values:

V1 = tuple of (name: ”Pooh”,   salary: 4.000.000)
V2 = tuple of (name: ”Mowgli”, salary: 1.000.000)
V3 = tuple of (name: ”Mickey”, salary: 6.000.000)

- Example objects:

O1 = < □, V1, □ >
O2 = < □, V2, □ >
O3 = < □, V3, □ >

CHARACTERISTICS OF OO-DBMS

MUST HAVE:

- OID (object identity/identifier)
- Complex/composite objects
- Types/Classes
- User-defines types
- Computational (language) completeness
- Encapsulation
- Inheritance: type/class hierarchies
- Polymorphisms:
  overloading, re-definition, late binding

... all orthogonal properties

SHOULD HAVE:

- Object versions
- Support for distribution (client/server architectures etc.)
- New transaction mechanisms
- Support for (active/deductive) rule-based systems

... and much more.

OID

- Objects exist independently of their (current) values
  - No matter how the values in an object are changed, the object is the same object
  - Objects are identified uniquely through the object identifier (OID)
  - Thus, there can be no erroneous references to the object as long as it is referred to through its OID
- The concepts of being identity and being equal both exist, and they do not mean the same thing (identity ≠ equality)
- OID can not be (reliably) based upon changing object values, but are usually system generated and managed surrogate values…
  - They are unique (system-wide, global, universal)
    GUID: Globally Unique ID (property of the MS world)
    UUID: Universally Unique ID (property of the Unix world)
  - Immutable (unchanging throughout the life an object – and kept intact after the object's destruction as well)

OPERATIONS ON OID

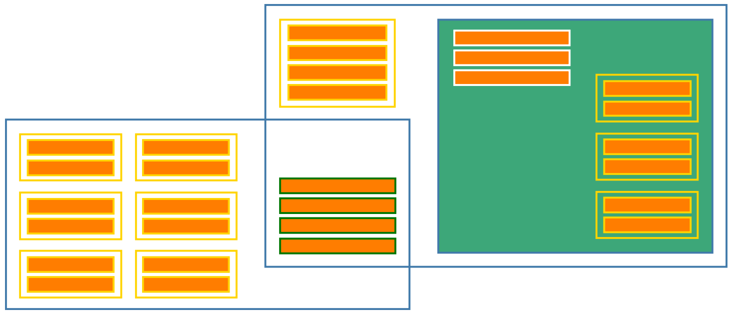
Generic Object-operations like...

- Comparing objects for equality, identity etc.
- Referencing objects
- Finding/fetching objects

... are all based upon the OID.

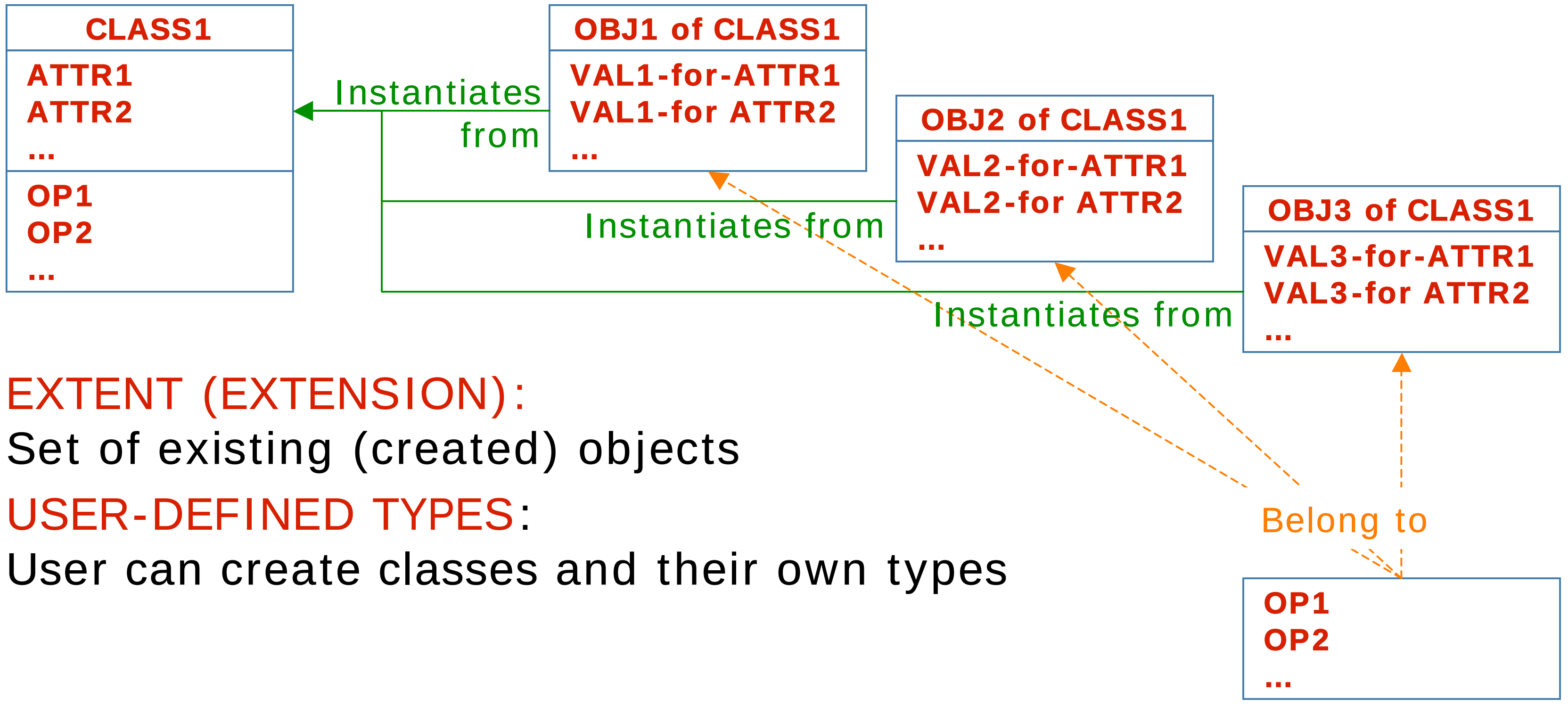
COMPLEX/COMPOSITE OBJECTS

- The OO paradigm supports objects that are complex in structure
- There are two kinds of complexity that the OO paradigm supports:
  - UNSTRUCTURED complex objects
    as in long time-series objects, media recording objects etc., collectively referred to as Binary Large Objects or BLOBS
  - STRUCTURED complex objects
    as in composite objects (objects that contain other objects or parts of other objects)



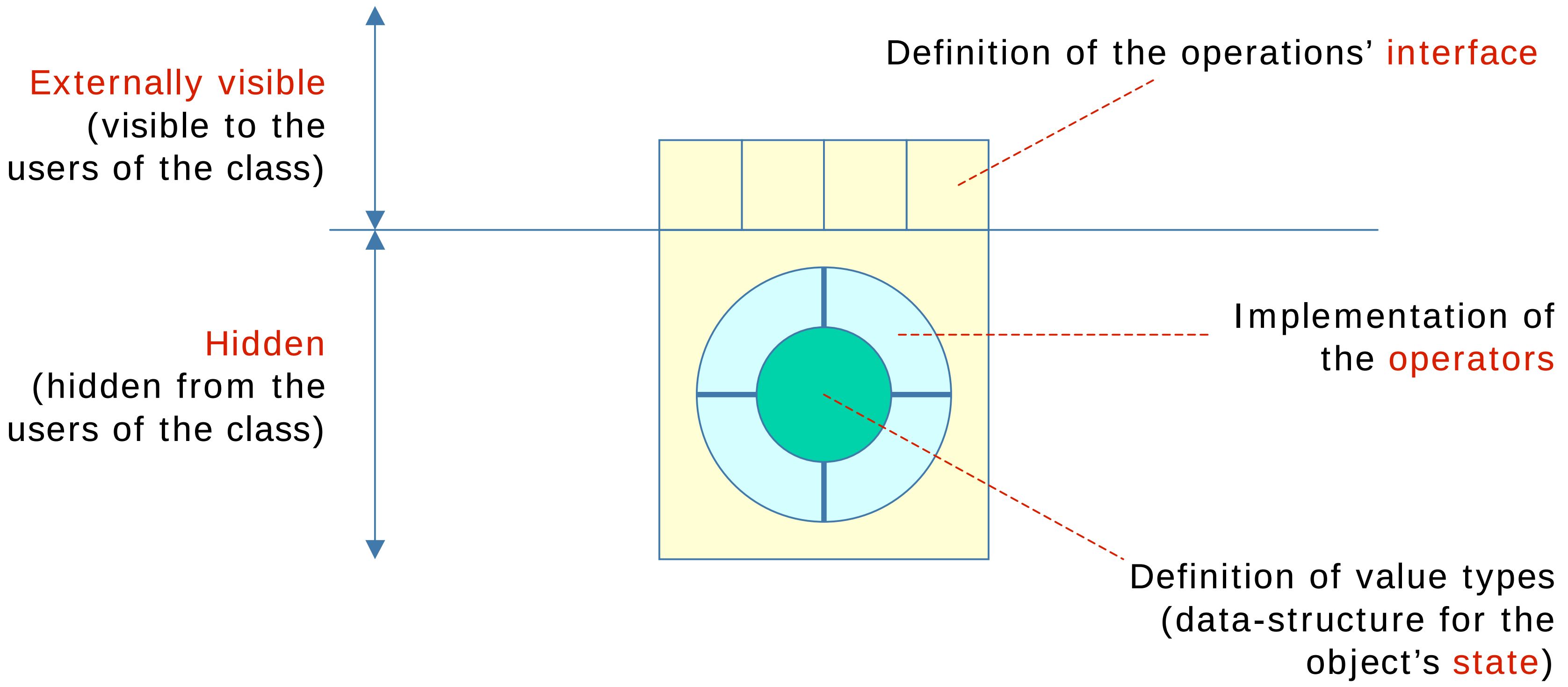
CLASSES and TYPES

- INTENT (INTENSION):
  Template for like objects (classes)
- INSTANTIATION:
  Creating new objects from the “template” (class)



- EXTENT (EXTENSION):
  Set of existing (created) objects
- USER-DEFINED TYPES:
  User can create classes and their own types

ENCAPSULATION


Externally visible
(visible to the users of the class)
Hidden
(hidden from the users of the class)
Definition of the operations' interface
Implementation of the operators
Definition of value types (data-structure for the object's state)

INHERITENCE, TYPE/CLASS HIERARCHIES

- Object-types(1) are not always independent of each other: GENERALIZATION/SPECIALIZATION ◊ SUB-TYPES/SUPER-TYPES

- Sub-types INHERIT properties (attributes and operations) from super-types

- There are two types of inheritance:
  - Single inheritance ◊ leads to a type hierarchy
  - Multiple inheritance (sub-type or sub-class inherits from more than one super-type or super-class) ◊ leads to type lattices

- Advantages of inheritance:
  - Re-use
  - Capability to extend semantics
  - Reinforcement of design discipline (stepwise refinement)

(1) Object-types also refer to classes in this context

POLYMORPHISM

- OVERLOADING
  Use of same name in different operators (in different classes/types)

- RE-DEFINITION
  Re-implementation of operators at a lower level in the class or type hierarchy

- LATE BINDING
  Bind an operator name to a specific implementation late in run-time (decided individually for each object)

- EXAMPLE:

```
      print-geometric-object(go: g-object)
```

  instead of

```
      print-circle(c: circle) and
      print-rectangle(r: rectangle) and
      print-triangle(t: triangle) etc.
```

Part 2

Introduction to the Object Definition Language (ODL)

PART 2, ODL, OVERVIEW

- Object Definition Language (ODL)
  - Classes
  - Attributes
  - Relationships
  - Methods
  - Type systems
  - Extensions
  - Keys
  - Inheritance

ODMG and STANDARDIZATION (REPETITION)

- OO-DBMS STANDARDIZATION
  - ODMG:
    Object Data Management Group
    Offers OO DBMS standard
    Offers (amongst others) two languages: ODL and OQL
  - ODL
    Object Definition Language
  - OQL
    Object Query Language

ODL CLASS DECLARATION

- ODL class declaration elements:
  - NAME of the class
  - KEY, as in other (relational) database systems, optional in ODL(1)
  - EXTENT, name of the set to contain all the instances (objects) of the class
  - ELEMENT declarations:
    - ATTRIBUTE
    - RELATIONSHIP
    - METHOD
- Syntax:

```
class <class-name>
{  <elements>
}
```

(1) Remember that a key is dependent upon (mutable) value. Remember also that an object has an OID.

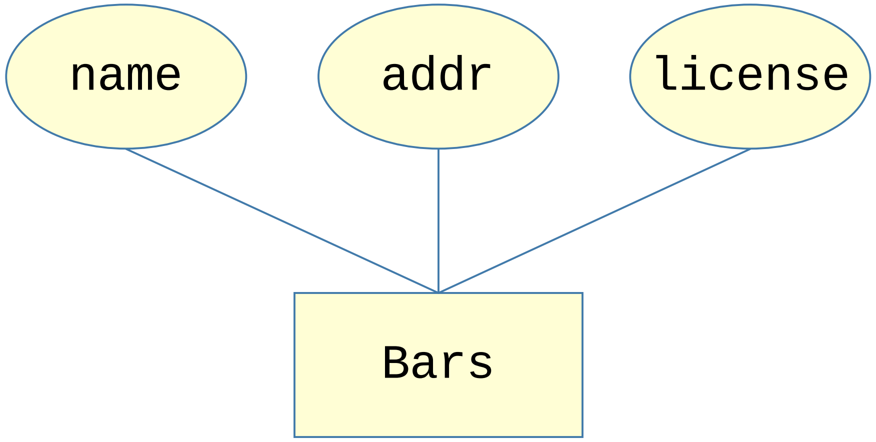
ODL CLASS DECLARATIONS – ATTRIBUTES #1

- Syntax:

```
attribute <attribute-type> <attribute-elements>
```

- Example:

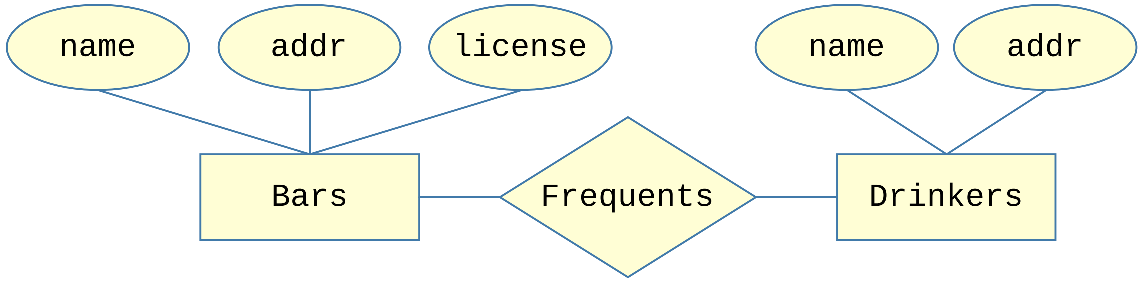
```
class Bars
{ attribute string name;
  attribute Struct addr
      {  string street,
         string city
      }  address;
  attribute Enum license
      {  full,
         beer,
         none
      }  licenseType;
}
```



ODL CLASS DECLARATIONS – ATTRIBUTES #2

- Q: Why do we name Structs and Enums?:
- A: Because we will need to refer to them.

- Example:



```
class Drinkers
{ attribute string name;
  attribute Struct Bars::addr address
}
```

- NOTE re-use of the Struct addr of Bars as type of the address attribute in Drinkers
- Elements in another class are represented by <class-name>::<element-name>

ODL CLASS DECLARATIONS – RELATIONSHIPS #1

- Relationships help relate (connect) objects to each other. They are references.

- Syntax:

  ```
      relationship <relationship-type> <relationship-name>
  ```

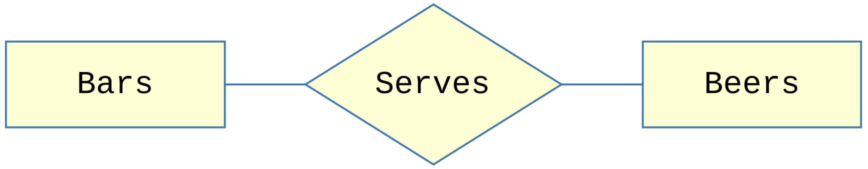
- Examples:

  ```
  relationship Set<Person> hasKids;
  relationship Person hasWife;
  relationship Set<Cars> hasCars;
  ```

ODL CLASS DECLARATIONS – RELATIONSHIPS #2

- Relationships come in pairs (with the relationship and its inverse)
- Examples:

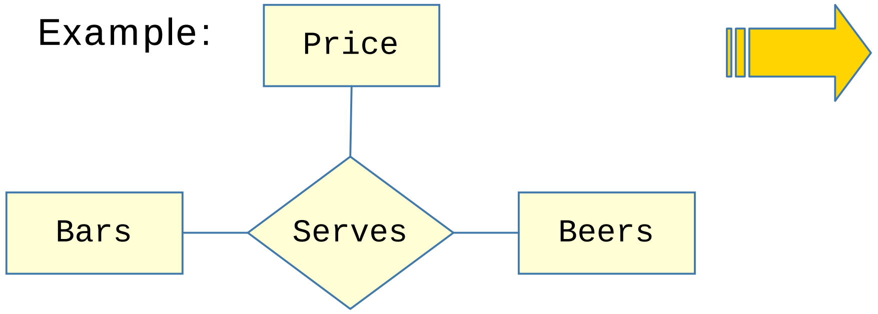
  The relationship Serves between Bars and Beers is represented through a relationship in Bars that indicates which Beers are sold, and another relationship in Beers indicated the Bars where the specific Beers are sold.

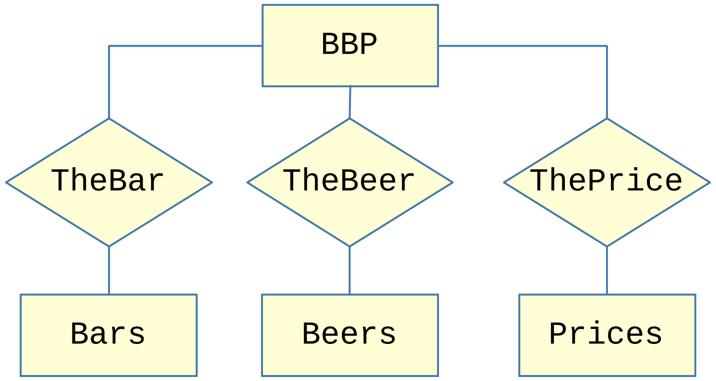


```
class Bars
{ relationship Set<Beers> Serves inverse Beers::ServedAt;
}

class Beers
{ relationship Set<Bars> ServedAt inverse Bars::Serves;
}
```

ODL CLASS DECLARATIONS – RELATIONSHIPS #3

- ODL supports only binary relationships, and not ternary (3-ways or tertiary) relationships and higher level relationships
- Ternary relationships and higher level relationships need own “cross-reference” class
- Example:



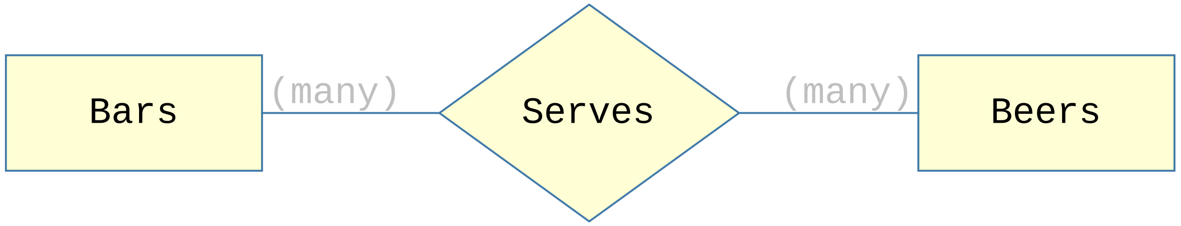


```
class Prices
{  attribute real price;
   relationship Set<BBP> toBBP inverse BBP::ThePrice
}
class BBP
{  relationship Bars TheBar inverse …;
   relationship Beers TheBeer inverse …;
   relationship Prices ThePrice inverse Prices::toBBP;
}
```

ODL CLASS DECLARATIONS – RELATIONSHIPS #4

- Many-to-many relationships use Set-types in both directions

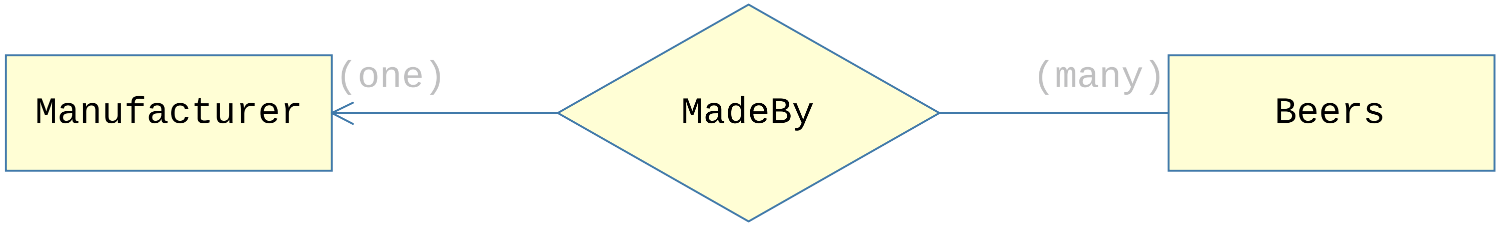
- Example:



```
class Bars
{ relationship Set<Beers> Serves inverse Beers::ServedAt;
}

class Beers
{ relationship Set<Bars> ServedAt inverse Bars::Serves;
}
```

ODL CLASS DECLARATIONS – RELATIONSHIPS #5

- One-to-many relationships use Set-type in one direction only
- Example:



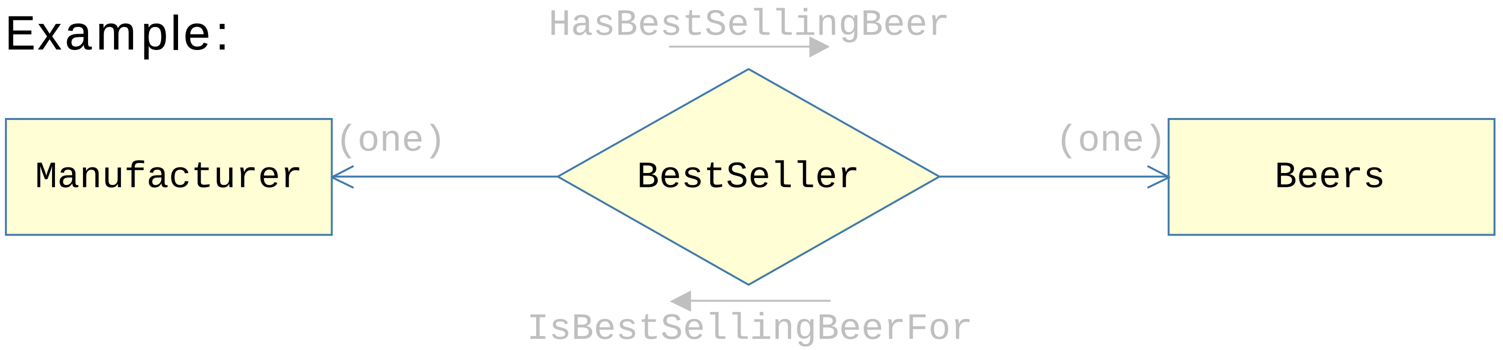
```
class Manufacturer
{ relationship Set<Beers> Makes inverse Beers::MadeBy;
}

class Beers
{ relationship Manufacturer MadeBy
       inverse Manufacturer::Makes;
}
```

NOTE:
Names of the relationships in the reading direction,
i.e., “Manufacturer **Makes** Beers” from Manufacturer to Beers, and
“Beers **MadeBy** Manufacturer” from Beers to Manufacturer.

ODL CLASS DECLARATIONS – RELATIONSHIPS #6

- One-to-one relationships are obvious (only the class-names of each other in both directions, no Set-type)

- Example:



```
class Manufacturer
{  relationship Beers HasBestSellingBeer
        inverse Beers::IsBestSellingBeerFor;
}

class Beers
{  relationship Manufacturer IsBestSellingBeerFor
        inverse Manufacturer::HasBestSellingBeer;
}
```

ODL CLASS DECLARATIONS – METHODS

- METHOD: Named and parameterized executable code (procedure, function) that functions as the operations of the class' objects.
- A method can return a value and raise exceptions.
- All method parameters and return value are typed.
- In addition to standard types for the parameters, parameters of a method can be tagged with in, out and inout:
  - in – for passing a copy of the value in the parameter (variable or object "value container") into the method
  - out – for passing value out from the method
  - inout – both of the above (like passing the value container or a reference to the value-containing object itself into the method instead of a copy of the value container's contents as in the case of in)
- Only the method signature is part of ODL. The code (implementation) is written in the host language (Java, C++, Smalltalk).
- EXAMPLE:

```
class Bars
{   . . .
    void availableBeers(out Set<Beers>);
    . . .
}
```

ODL – THE TYPE SYSTEM

BASE TYPES:

- **integer, real, float, character, string, enumerated types, boolean** and more.
- Type constructors:
  - **Struct** en (a structure composed of type and name pairs, like a record)
- Collection types:
  - **Set<T>** un-ordered set of (distinct) objects of type T
  - **Bag<T>** un-ordered set of objects of type T where duplicates are allowed
  - **List<T>** ordered collection of objects of type T where duplicates are allowed
  - **Array<T>** ordered and indexed collection of objects of type T where duplicates are allowed
  - **Dictionary<S,T>** set of object-pairs of type S and T respectivelyT

NOTE:

- Type of a relationship can only be a class or a collection of classes as we have seen.

ODL – KEYS

- In ODL, classes (and their objects) do not need keys. OID is fully capable of distinguishing between objects that have the same value-set in its elements (attributes, relationships etc).
- In ODL, a key is specified with the key-word key or keys and a list of the attributes that form the key[(1)]
- Several lists ca be specified to define several alternative keys
- Parentheses are used to group the members in multi-valued keys:
  - **key**(a_1, a_2 , … , $a_\mathbf{n}$) = “key with **n** attributes”
  - **keys** a_1, a_2 , … , $a_\mathbf{n}$ = “each $a_\mathbf{i}$ *is a key*, and each one of them can be a multi-valued key, i.e., at =($b1$,$b2$, … , $b_\mathbf{k}$)”

- EXAMPLE of a single valued key:

```
class Beers (key name)
{   attribute string name;
}
```

- EXAMPLE of two 2-valued keys:

```
class Courses (key (dept, number), (room, hours))
{   ...
}
```

(1) Note that use of the term “attributes” here is actually wrong. In addition to attributes, relationships and even methods cam be part of a key.

ODL – INSTANTIATABLE CLASSES and EXTENTS

- There is a difference between a class definition and the set of existing (created and not yet destroyed) instances (objects) of the class, called an **extent**
- In ODL the extent is expressed with the key-word `extent` followed by the name of the extent (i.e., the name of the set to contain the instances of the class)

- SYNTAX/EXAMPLE:

```
class Student (extent students key SSN)
{ ...
}
```

- Note that a class defined with the key-word `class` can be instantiated from, i.e., can be used to create objects from

ODL – INTERFACES

- ODL allows for INTERFACES, which are in essence “signature classes” without own objects (I.e., classes that one can not be used to instantiate objects from)
- Useful especially when we have several extents but with (some) common properties.

- EXAMPLE

```
interface Person
{   attribute integer SSN;
}

class Student : Person (extent students key SSN)
{   ...
}

class Teacher : Person (extent teachers key SSN)
{   ...
}
```

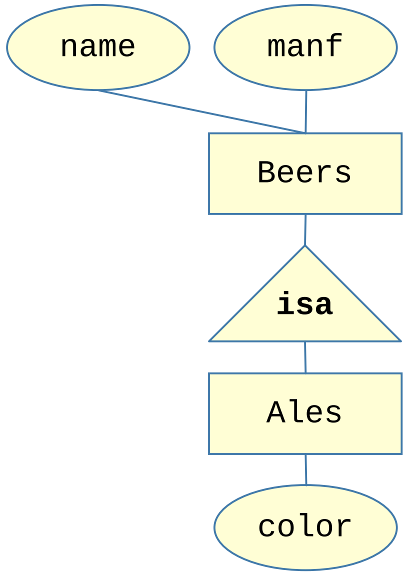
- Interfaces are defined using the key-word `interface` instead of `class`
- Interfaces can not be instantiated from but can be used to define other classes (as in the example, indicating that both `Students` and `Teachers` are `Persons`.
- Since they cannot be instantiated from, it is meaningless to use the keywords `extent` and `key` (or `keys`) in interfaces.

ODL – SUB-CLASSES, SUPER-CLASSES and INHERITANCE

- A sub-class inherits all properties of its super-class.

- EXAMPLE: `Ales` gets all the attributes, relationships and methods of the `Beers` Class

- Super-classes are denoted by prefixing them with:
  - colon (:) for interfaces
  - Keyword extends for instantiatable classes



- EXAMPLE: All `Ales` are `Beers` with `color`:

```
class Ales extends Beers
{   attribute string color;
}
```

- Interfaces can inherit only from other interfaces (but classes can inherit from interfaces as in our previous example of interfaces).

ODL – MULTIPLE INHERITANCE

- Multiple inheritance is denoted by the keyword **`extends`** and a colon-separated (“:”-separated) list of the classes being inherited from

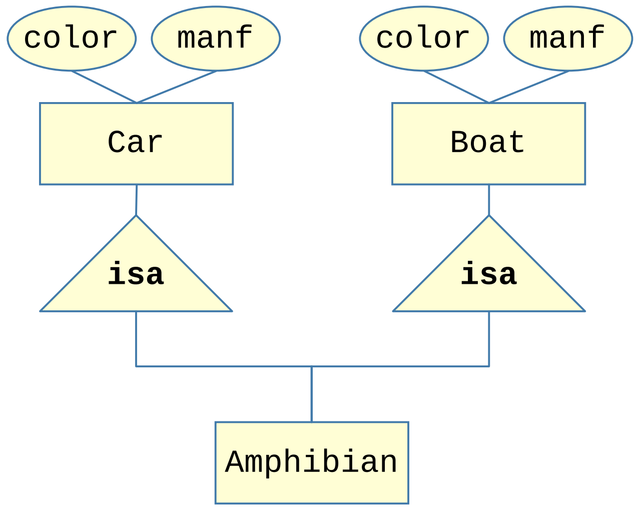
- EXAMPLE:
  `class Amphibian extends car:boat { ... }`

- Name conflicts are not allowed and its designer’s/developer’s responsibility to avoid such conflicts.

- All classes can inherit from (an arbitrary number of) other classes or interfaces, but:

- an interfaces can only inherit from other interfaces as we saw earlier

- and an instantiatable class can only inherit from another instantiatable class, so a class cannot be an extension of more than one class



Part 3

Introduction to Object-Relational Database Systems

(OR-DBMS)

(... over to old slides)

NEXT TIME

Next time is week 6 (22. Feb. 2005)

- A bit about XML
- The Object Query Language (OQL)