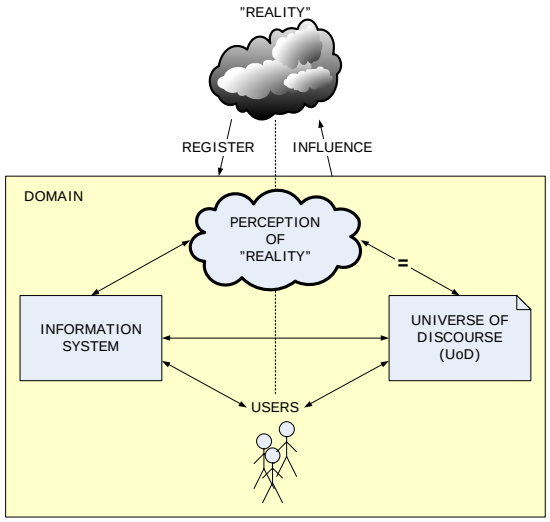
INF5100

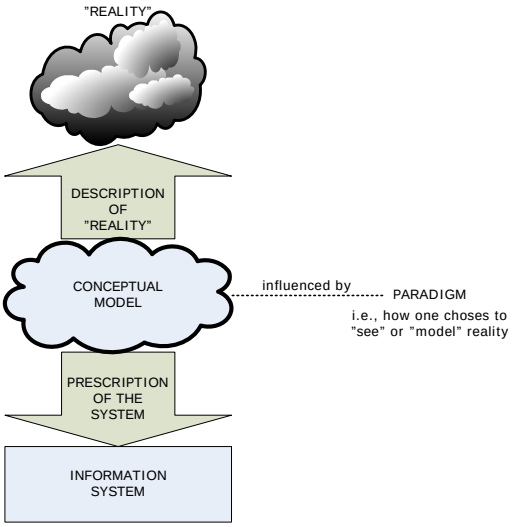
Advanced Database Systems

(Previously INF3180, also based upon earlier INF312, IN-MDS and UNIKI 330)

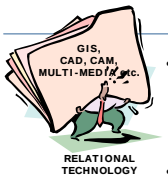
Reference:
These foils are based upon foils by
Ragnar Normann, Gerhard Skagestein and Vera Goebel



INFORMATION MODELS – *Purpose of a model*



INFORMATION MODELS – *Beyond the relational model*



- **SINCE...**
  The newer requirements are becoming too heavy to carry for the relational model,
- **WE ALSO CHOOSE TO LOOK AT OTHER MODELS / DATABASES LIKE:**

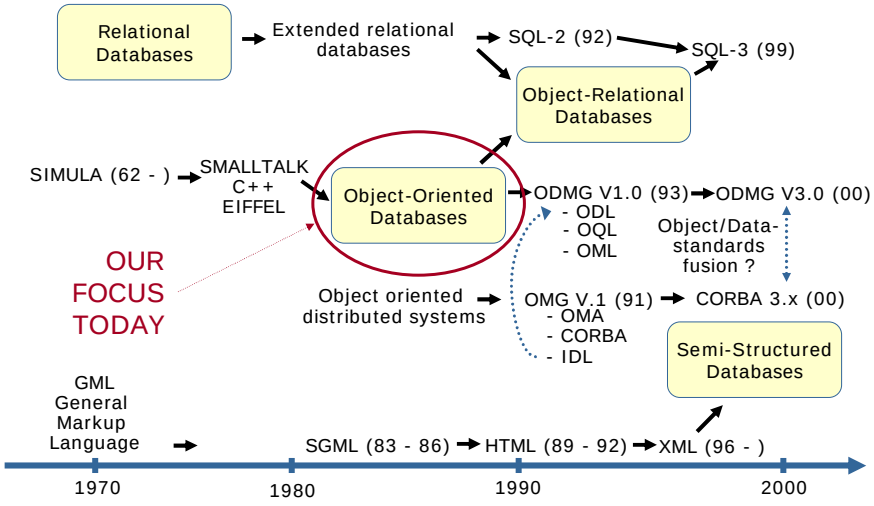
We'll look at these

- Object-Oriented (OO) Databases
  - To exploit the OO paradigm and to match the OO languages
- Extended relational (ER) or object-relational (OR) databases
  - To allow for a smooth passage to the OO world by adding OO functionality to relational databases

This comes later

- XML and XML-databases
  - For document databases, semi-structured data storage/retrieval, data-integration

A PARTIAL HISTORY – *Data models and standards*



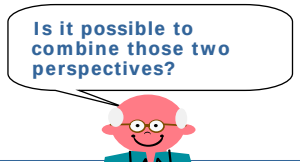
A CENTRAL CONCEPT – *Persistence*

- **From the perspective of the program:**
  Objects die at program termination.

  THEY ARE TRANSIENT!

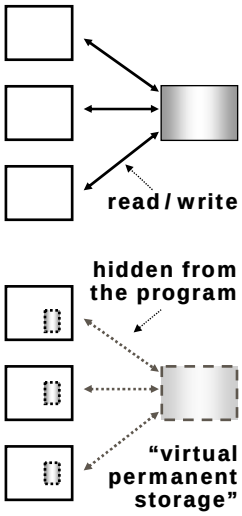
- **From the perspective of the database (also user and the world external to the program):**
  Objects live after program termination – forever, until they are explicitly removed.

  THEY ARE **PERSISTENT**!



HOW TO ACHIEVE PERSISTENCE

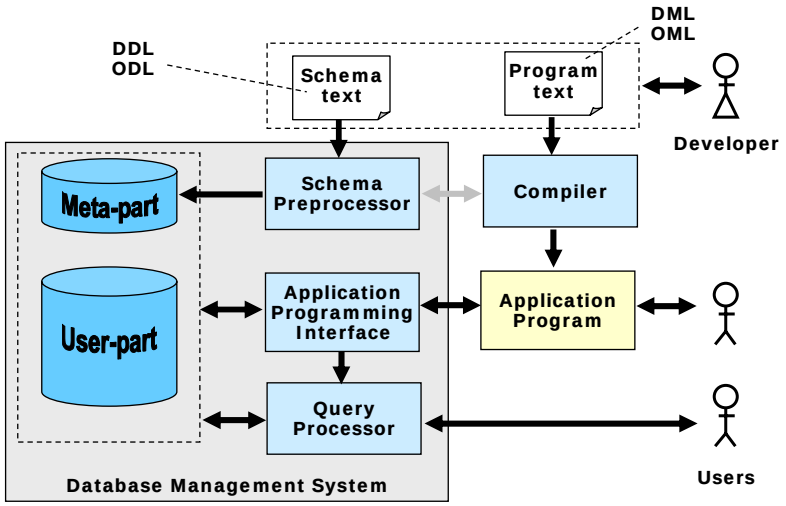
- Explicitly transfer “objects” to and from a permanent storage by read/write - commands
  - Database separated from the program
- Make “objects” persistent by the program
  (SQL Create, bind object to database)
  - Database integrated with the program
    (The Single Storage Illusion)



DATABASE INTEGRATED WITH THE PROGRAM

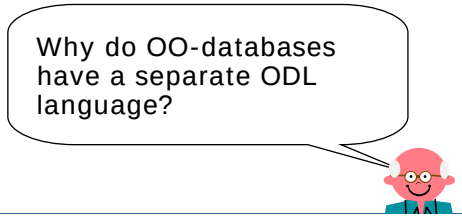
- The usual programming language (Java, C++, Smalltalk) should also be a Object Manipulation Language
- Transient and persistent objects should be handled the same way
  – simple programming
- All types of objects should be able to be persistent
  (Type and persistence are orthogonal)

SCHEMAS AND PROGRAMS

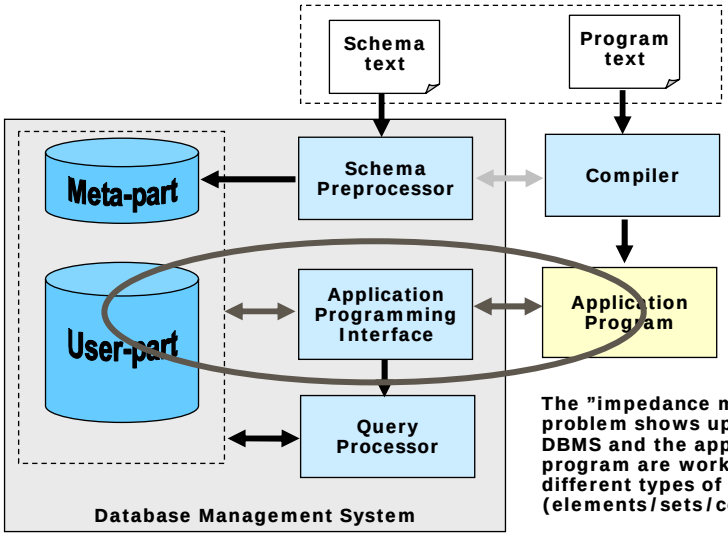


SCHEMA AND MANIPULATION LANGUAGES

- Relational databases:
  The schema language (DDL) and the manipulation language (DML) are integrated in the same language (SQL)
- OO-databases:
  The schema language (ODL, IDL) is a separate language



THE IMPEDANCE MISMATCH

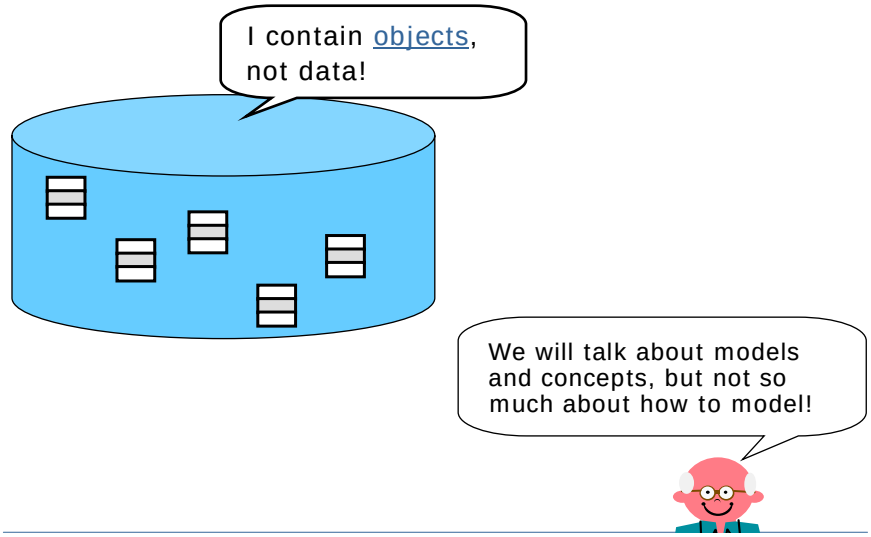


The "impedance mismatch" problem shows up when the DBMS and the application program are working with different types of operands (elements/sets/containers)

SOLVING THE IMPEDANCE MISMATCH PROBLEM

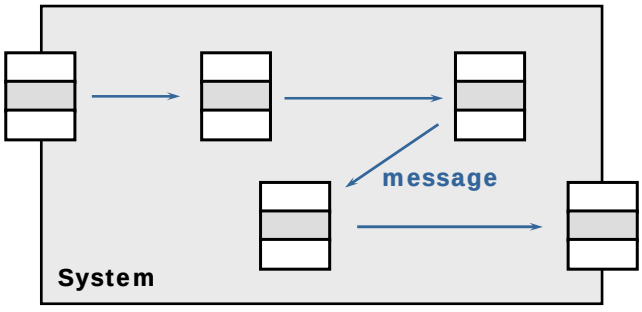
- Let the API mimic a navigational view into the database (the SQL-solution – "cursors")
- Give the database navigational capabilities
- Give the application programming languages container capabilities

OBJECT-ORIENTED DATABASES



THE BASIC PRICIPLE OF OBJECT-ORIENTATION

Model the mini-world (Universe of Discourse, UoD) as a collection of cooperating, related units, called objects



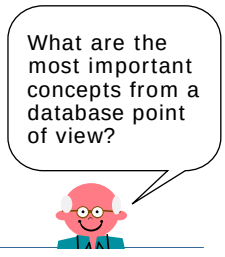
THIRTEEN OODBMS COMMANDMENTS

Atkinson et al.:
The Object Oriented Database System Manifesto (1990)

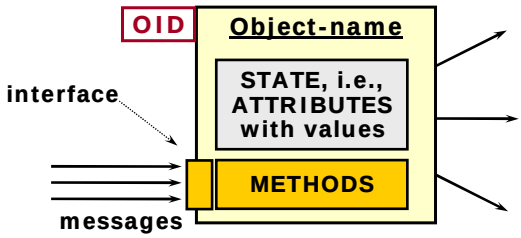
- Rules that make it an **OO system**:
  - Thou shalt support complex objects
  - Thou shalt support object identity
  - Thou shalt encapsulate thine objects
  - support types or classes
  - Thine classes or types shalt inherit from their ancestors
  - Thou shalt not bind prematurely
  - Thou shalt be computationally complete
  - Thou shalt be extensible
- Rules that make it a **DBMS** :
  - Thou shalt remember thy data
  - Thou shalt manage very large databases
  - Thou shalt accept concurrent users
  - Thou shalt recover from hardware and software failures
  - Thou shalt have a simple way of querying data

OO CONCEPTS (GENERAL)

- Abstraction and autonomy
  - object: <value, {operators}>
  - value: data structure
  - encapsulation (information hiding)
  - request of performance from other objects
- Classification
  - common description (intension)
  - collection of similar objects (extension)
- Taxonomy
  - super-/sub-classes
  - inheritance of properties
  - polymorphism



CHARACTERISTICS OF OBJECTS



- Objects have a permanent, immutable, not reusable identity – the Object identifier (OID)
- Objects remember (they have a state)
- Objects have a behavior (they have methods)

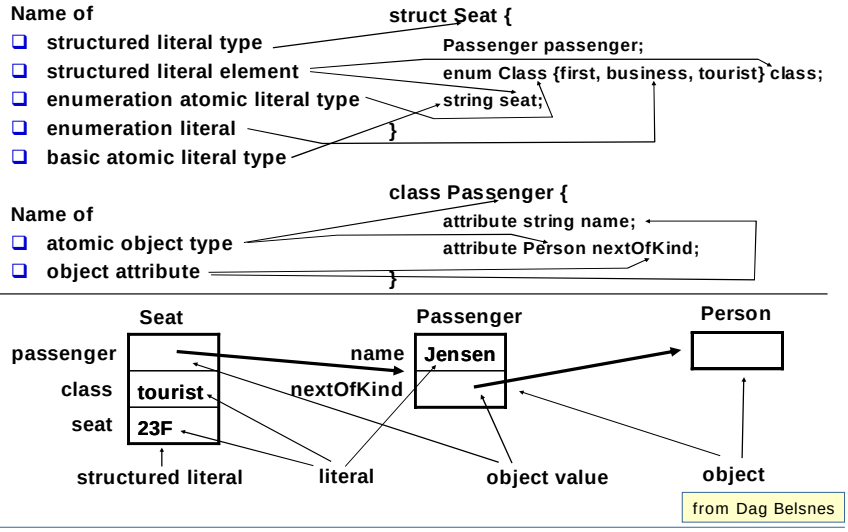
OBJECT IDENTITY ⇨ OBJECT IDENTIFIER ⇨ OID

- Objects exist independently of their (current) values
  - modifications of any kind result in "same" object
  - no misleading references to objects
  - "identity" ≠ "equality" (both needs to be expressible)
- Object identity cannot (reliably) be based on ordinary values provided by application (value orientation)
  … but on surrogates: object identifiers being
  - (system-wide) unique
  - unchanged during object lifetime
  - not reused after object deletion
  - generally system-managed

"generic" object operators:
- object comparison
- object retrieval
- …

based on OID

OBJECT - LITERAL



NON-ATOMIC OBJECT

Collections

- Set<T> Unordered set of different objects of type T
- Bag<T> Unordered collection of objects of type T, duplicates allowed.
- List<T> Ordered collection of objects of type T.
- Array<T> Ordered, indexed collection of objects of type T.
- Dictionary<T> Set of object pairs of type T (struct Association {Object key, Object value; } ;)

NOTE: An iterator can be created to traverse a collection.

Structured (predefined)

- Date, Time, Timestamp, Interval

NON-ATOMIC LITERALS

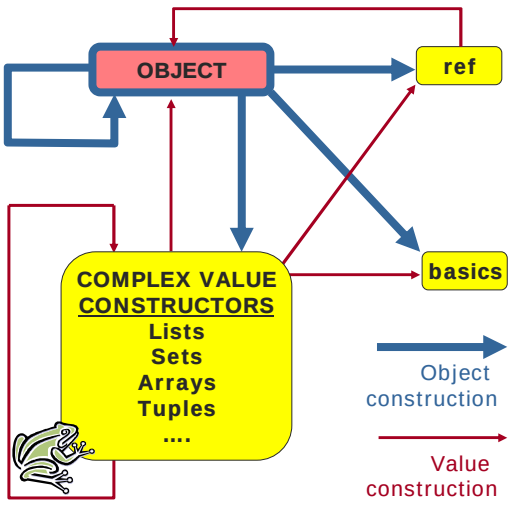
- **Collections**
  - as for objects:
    set<t>, bag<t>, list<t>, array<t>, dictionary<t>
- **Structured**
  - predefined: date, time, timestamp, interval
  - user defined: struct { … }

```
struct Address
{     string   street;
      unsigned short number;
      unsigned short postNo;
      string   postArea;
}
```


CONSTRUCTION OF COMPLEX OBJECTS

Degrees of freedom:

- Which constructors?
- Which base types?
- References, subobjects, explicit relationships



from Vera Goebel

SUB-OBJECTS VERSUS REFERENCES TO OBJECTS

	Independent subobjects (own existence)	Dependent subobjects (no own existence)
Sharable subobjects (logical)	e.g. module in software system	e.g. chapter in book
Not sharable subobjects (physical)	e.g. disk drive in PC	e.g. path in design of VLSI cell

from Vera Goebel

CONSTRUCTION OF COMPLEX VALUES AND OBJECTS

V1 = tuple of (name: "Solskjær", salary: 4000)
V2 = tuple of (name: "Berg", salary: 2000)
V3 = tuple of (name: "Dæhli", salary: 1000)

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

V4 = tuple of (name: "Hermansen",
address: tuple of (zipcode: N-0157,
city: "Oslo",
street: "Tollbudgata",
phone: set of (22 93 54 32, 977 54 36)),
salary: 2000)

V5 = tuple of (depname: "finance" employees: set of (V1, V2, V3)

V5 = tuple of (depname: "finance" employees: set of (O1, O2, O3)
O4 = < •, V6, •>

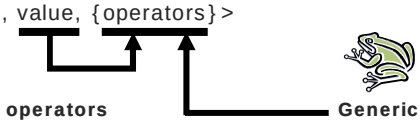
V5 = tuple of (depname: "finance" employees: set of (ref O1, ref O2, ref O3)
O4 = < •, V7, •>

from Vera Goebel

OPERATORS FOR COMPLEX OBJECTS

For composite objects:
- one-level operators
  affect topmost level only, NOT subobjects
- multilevel ("transitive") operators
  potentially affect ALL direct and indirect subobjects
  ("propagation effect")

Object := <OID, value, {operators}>



Value operators | **Generic object operators**

Processing of entire objects (transitively; depending on actual structure)	Processing of individual object levels (nontransitively; structure irrelevant)
retrieve/delete/copy object (parts) modify values	retrieve/delete/copy object

Structure related operators (insert/remove subobjects; navigation in object structures

(also) included: value-based selection of desired objects

from Vera Goebel

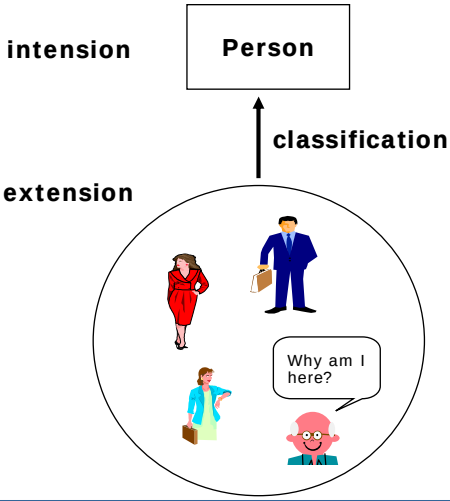
NOTIONS OF EQUALITY

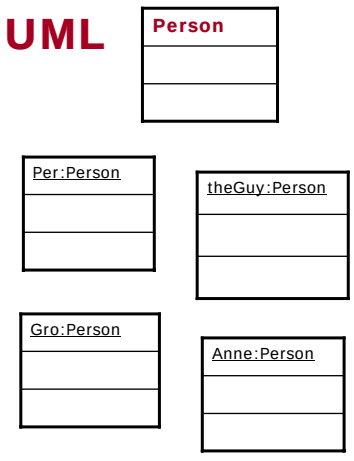
- **Shallow equality**
  - References are equal (same OID)
- **Deep equality**
  - the objects are of atomic type and have the same value,
  - or -
  - the objects are of reference type, and the deep equals operator is true for the two referenced objects,
  - or -
  - the objects are of structured type, and the deep equals operator is true for all the corresponding subparts of the two objects

CLASSIFICATION

- Identification and description of a concept as a type
- A type has an external specification and one or more implementations
- The specification defines the external aspect, visible to the user of the type:
  - operations that can be invoked on the instances
  - properties (or state variables), whose values can be accessed
  - exceptions that can be raised by the operations

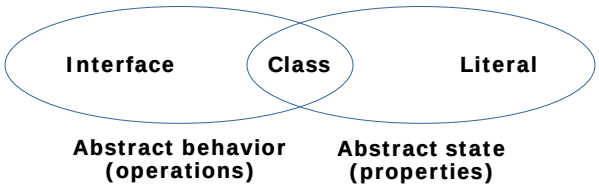
INTESION and EXTENSION





SPECIFICATIONS

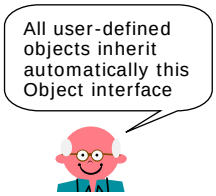
- Interface definition
  Specification of the abstract behavior of an object type
- Class definition (abstract class)
  Specification of the abstract behavior and abstract state of an object type
- Literal definition
  specification of the abstract state of a literal type



Cattell et al.: The Object Data Standard: ODMG 3.0, Figure 2-1

INTERFACE DEFINITION EXAMPLE

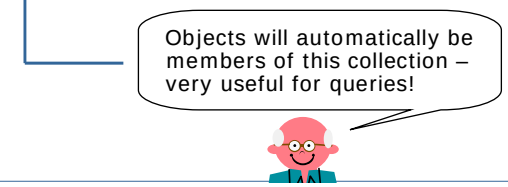
```
interface Object
{
   Enum  Lock_Type{read, write, upgrade};
   Void  lock(in Lock_Type mode) raises
     (LockNotGranted);
   Boolean  try_lock(in Lock_Type mode);
   boolean  same_as(in Object anObject);
   Object   copy( );
   void     delete( );
}
```



- If there is an attribute in the interface definition, this just says that it should be possible to read/write that attribute – it does not belong to the state

CLASS DEFINITION EXAMPLE

```
class Person {
      (extent persons key ssn) {
      exception NoSuchPerson { } ;
      attribute string name;
      readonly attribute string ssn;
      attribute Address address;
      relationship <Person> spouse inverse Person::spouse;
      relationship list<Person> children inverse Person::parents;
      relationship set>Person>parents inverse Person::children;
      void marriage (in string ssn) raises(NoSuchPerson);
      unsigned short descendants(out set<Person> inheritors);
}
```



LITERAL DEFINITION EXAMPLE

```
struct Address
{
    string street;
    unsigned short number;
    unsigned short postNo;
    string postArea;
}
```

TYPE/CLASS HIERARCHIES, INHERITANCE

- Object types not always independent of each other
- TAXONOMY: generalization/specification
  ⇒ subtypes/supertypes, is_a-relationship
- Considerable variation in details:
  - interface hierarchy (with regard to operators)
  - implementation hierarchy (with regard to operators, representation)
  - extension hierarchy (with regard to membership of instances)
- Instances of subtypes inherit properties from supertypes

Advantages of the inheritance principle:
- code reusability (when operators are inherited)
- representation of additional semantics
- design discipline (stepwise refinement)

VARIATIONS IN THE INTERPRETATION OF SUBTYPES

B is_a A

- Taxonomy
  - where an A-object is required, a B-object may be used
  - implementation of B uses implementation of A
  - sets (extensions) of instances: {B} ⊆ {A}
- Inheritance of
  - value types
  - external interfaces (signatures of operator set)
  - code
  - simple polymorfism (cncrete operation inherits definition)

SIMPLE OR MULTIPLE INHERTIANCE?

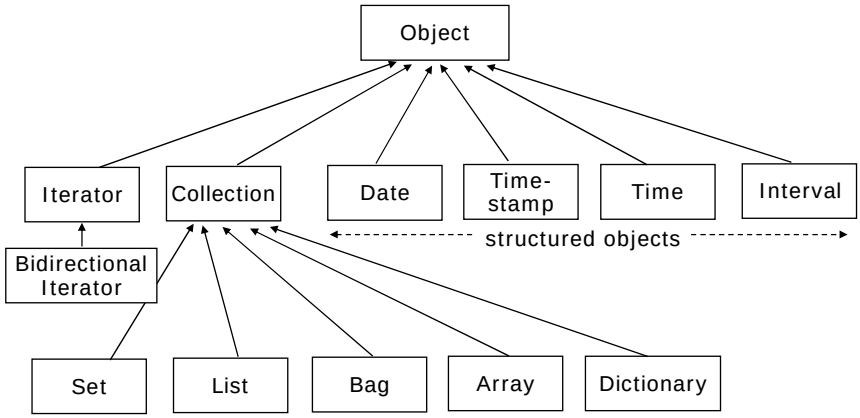
- Rules for inheritance
  - simple inheritance (type hierarchy)
  - multiple inheritance (type lattice)

- The ODMG object model:
  - interfaces and classes may inherit from multiple interfaces (denoted by :)
  - classes can inherit only from a single class (denoted by extends)

  A pragmatic decision!

  Example:
  class TeacherAssistant extends Employee:StudentIF {…}

BUILT-IN INTERFACES OF THE OBJECT MODEL

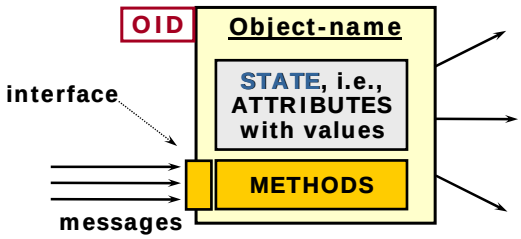


Elmasri & Navathe Figure 12.2 (page 394)
For an overview of the definition of the interfaces, see Elmasri & Navathe Figure 12.1a & 12.1b

PROPERTIES OF OBJECTS

- **State-properties**
  - Attributes
  - Relationships
- **Operator-properties**
  - Operations
  - … with exceptions

THE STATE



The state of an object is the union of the current values of its
- attributes - literals, “collections” and OIDs
- relationships

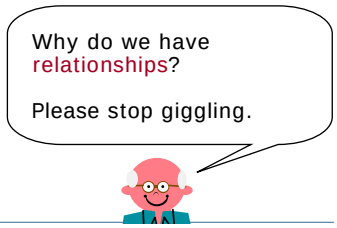
MODELING STATE PROPERTIES

- **Attributes**
  Defines the abstract state of the instances of the class

```
class Person {
    attribute short age;
    attribute string name
    attribute enum gender {male, female};
    attribute Address home_address;
    attribute set<Phone_no> phones;
    attribute Department dept;
}
```

- **Relationships**
  Defines relationships between two types

```
class Professor {
    …
    relationship set<Course> teaches
       inverse Course::is_taught_by;
}
class Course {
    …
    relationship Professor is_taught_by
       inverse Professor::teaches;
}
```



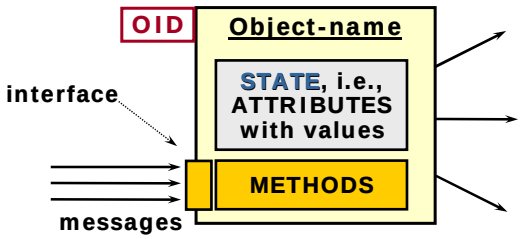
MODELING BEHAVIOR: OPERATIONS

- **Operation signatures** – the **interface** of an operation
  - Name of operator
  - List of parameters
  - Type of result
  - Exceptions that may be raised
- **Overloading**
  - Operator name may be reused under the condition that the parameter list is different

OPERATORS

- **System defined (predefined)**
  - Type-specific
    - (at least) for atomic values
    - overloading possible
  - Generic
    - for composite values only
    - uniform applicability for all values built by a given constructor
- **User-defined**
  - building upon predefined or other user-defined operators
  - appropriate mechanism needed

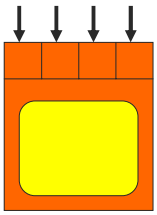
ENCAPSULATION



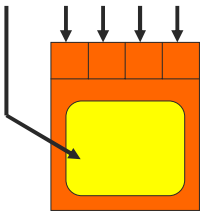
- The state is not directly accessible from outside – it can only be inspected or changed by calling methods (i.e. sending messages)
- The implementation of the operator bodies is hidden – only the signatures are visible from outside

DEGREES OF ENCAPSULATION

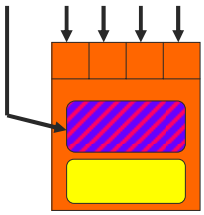
a) complete encapsulation (all accesses exclusively by calling defined operations)



b) write-encapsulation ("direct" read-access allowed)



c) partial encapsulation arbitrary "direct" access to public data allowed)



from Vera Goebel

POLYMORPHISM

- **Overloading:** Use of same name for different operators (in different types)
- **Overriding:** Reimplementation of operator bodies on lower level of type hierarchy

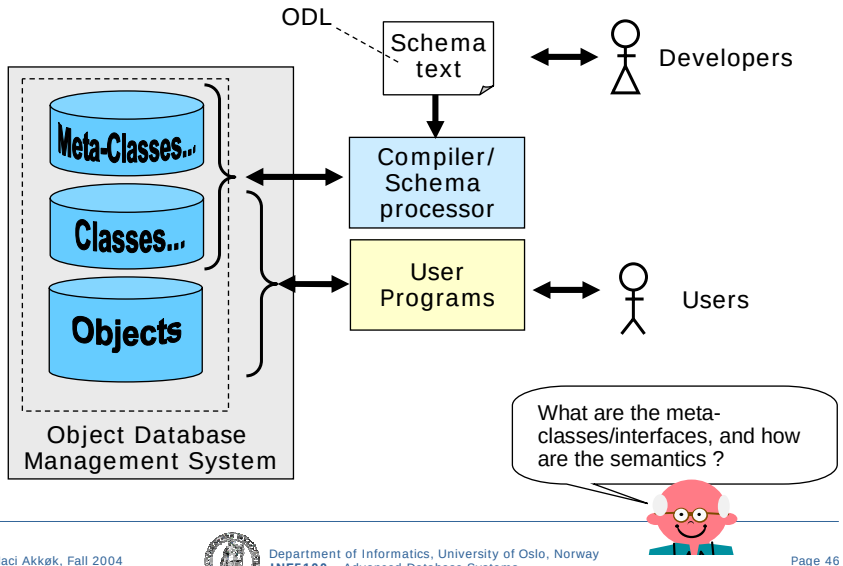
```
print_geometric_object (o: g_obj)
(implemented for circles, rectangels, triangles etc)

for all x in M do print_geometric_object(x);

- versus -

for all x in M do case      x is circle:      print_circle(x);
                            x is rectangle:   print_rectangle(x);
                            x is triangle:    print_triangle(x);
                            otherwise (exception handling);
```

- Requires "late" binding
  ... of an operator name to an associated implementation (... to a type)
- Operators: generic ↔ overloaded ↔ individual

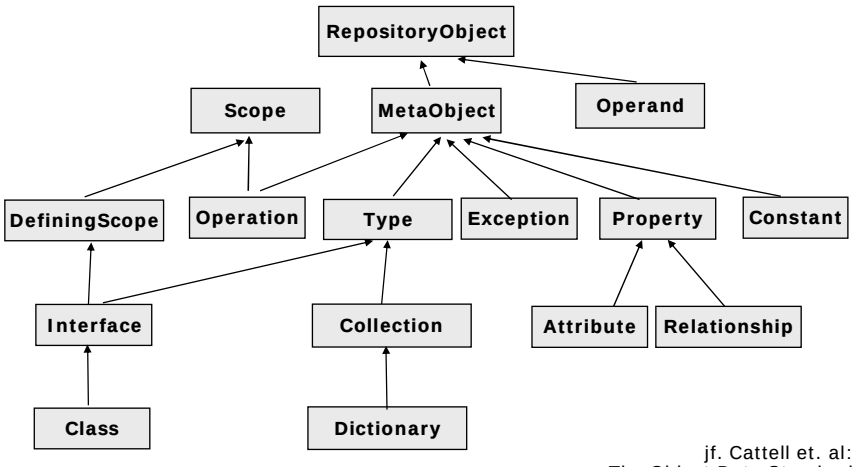
(INTEGRITY) CONSTRAINTS ON OBJECTS

- Mostly implemented in methods (per type/class)
- Inherited along type/class hierarchy
- Explicit relationships / consistency constraints
- Key constraints

THE META LEVEL



THE META LEVEL



jf. Cattell et. al:
The Object Data Standard
ODMG 3.0 , page 42-53

ONCE MORE: HOW TO MAKE OBJECTS PERSISTENT

- Seamless integration of DBS and programming language
  ➔ two types of objects: transient and persistent objects
- Persistence specified explicitly by
  (1) Naming
  - or -
  (2) Reachability
- Via entry points into the database
  (1) persistent collections
  - or -
  (2) root of network of connected objects (by references)

THE ODMG OBJECT MODEL

- **ODMG** = **O**bject **D**ata **M**anagement **G**roup
- Basic building blocks:
  - Object, each object has a unique identifier (OID)
  - Literal, no OID, represents a value (possible with a complex structure)
- Objects and literals are classified by their types
- An object has a set of state-properties
  - The attributes of the object
  - The relationships between the object and other objects
- The state of an object is the value of its state-properties
- An object has a set of operation-properties. These operations can be executed by the object and make up the behavior of the object

THE CHOICES MADE IN THE ODMG OBJECT MODEL #1

Types, instances, interfaces, and implementations:

- Objects are instances of types
- A type defines the behavior and state of its instances
- Behavior is specified as a set of operations
- An object can be an immediate instance of only one type
- The type of an object is determined statically at the time the object is created; objects do not dynamically acquire and lose types
- Types are organized into a subtype-supertype graph
- A type may have multiple supertypes
- Supertypes are explicitly specified; subtype-supertype relationships between types are not deduced from signature compatibility of the types

THE CHOICES MADE IN THE ODMG OBJECT MODEL #2

Operations:

- Operations have signatures that specify the operation name, arguments, and return values
- Operations are defined on a single type – the type of thier distinguished first argument – rather than on two types
- Operations may take either literals or objects as their arguments. Semantics of argument passing is pass by reference
- Operations are invoked
- Operations may have side effects
- Operations are implemented by methods in the implementation portion of the type definition

PROPERTIES OF THE OO DATA MODEL (OODM)

- Object identity
- Complex (composite) objects
- Types / classes
- User-definable types
- Language completeness
- Encapsulation
- Type/class hierarchies
- Overloading / overriding / late binding (polymorphisms)

... ALL ORTHOGONAL!

NOT PART OF THE OODM DEFINITION (ON PURPOSE)

Also needed/provided in “new” DBS:

- Object versions
- Specific realization (implementation) of concepts
- Distribution (client/server architectures)
- Specific processing aspects, e.g., new transaction mechanisms
- Rule-based mechanisms (active / deductive features)

… and much more