Object Database Standard ODMG

ODMG Object model

- Provides a standard model for object databases.
- Supports object definition via ODL.
- Supports object querying via OQL.
- Supports a variety of data types and type constructors.

ODMG Objects and Literals

- The basic building blocks of the object model are:
  - **Objects**
  - **Literals**
- An object has four characteristics:
  - **Identifier**: unique system-wide identifier.
  - **Name**: unique within a particular database and/or program; it is optional. Designated by programmers as convenient way to refer.
  - **Lifetime**: how the memory and storage allocated to objects are managed: transient or persistent.
  - **Structure**: specifies how object is constructed by the type constructor and whether it is an atomic object or collection object.

ODMG Objects

- Object has both an object identifier and a state (current value).
- The object state can change over a time by modifying the object value.
- In ODMG all objects inherit the basic interface of Object.

```
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();
};
```

ODMG Objects

- Collection_object
  - `Set<>`
  - `Bag<>`
  - `List<>`
  - `Array<>`
  - `Dictionary<>`
- Structured_object
  - `Date`
  - `Time`
  - `Timestamp`
  - `Interval`
- Atomic_object

ODMG Objects

- Collection_object

```
Set<>
Bag<>
List<>
Array<>
Dictionary<>
```

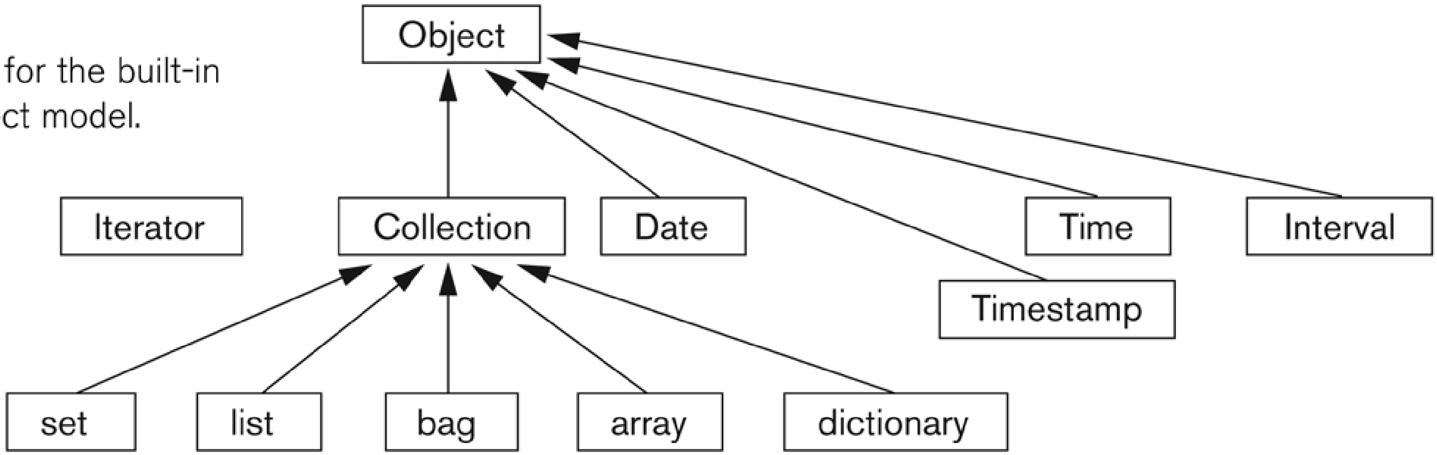


Figure 21.2
Inheritance hierarchy for the built-in interfaces of the object model.

Collection Object

- Collection object inherits the basic Collection interface.
- The instances of collection objects are composed of elements which can be an instance of atomic type, another collection or a literal type.
- Important characteristic of a collection is that *all* the elements of the collection must be of the *same* type.
- The collections supported by ODMG Object Model include:

  Set<t>

  Bag<t>

  List<t>

  Array<t>

  Dictionary<t,v>

Collection Object

```
interface Collection : Object {
exception InvalidCollectionType{};
exception ElementNotFound{Object element; };
unsigned long cardinality();
boolean is_empty();
boolean is_ordered();
boolean allows_duplicates();
boolean contains_element(in Object element);
void insert_element(in Object element);
void remove_element(in Object element)
               raises(ElementNotFound);
…....
Object select_element(in string OQL_predicate);
boolean query(in string OQL_predicate, inout Collection result);
boolean exists_element(in string OQL_predicate)
};
```

Collection Object

- An Iterator, which is a mechanism for accessing the elements of a Collection object, can be created to traverse a collection.

```
interface Iterator {
exception NoMoreElements{};
exception InvalidCollectionType{};
boolean is_stable();
boolean at_end();
void reset();
Object get_element() raises(NoMoreElements);
void next_position() raises(NoMoreElements);
void replace_element (in Object element)
       raises(InvalidCollectionType);
}
```

Set Object

- A Set object is an unordered collection of elements, with no duplicates allowed.
- The interface has the conventional mathematical set operations.

```
class Set : Collection {

attribute set<t> value;
Set        create_union(in Set other_set);
Set        create_intersection(in Set other_set);
Set        create_difference(in Set other_set);
boolean    is_subset_of(in Set other_set);
boolean    is_proper_subset_of(in Set other_set);
boolean    is_superset_of(in Set other_set);
boolean    is_proper_superset_of(in Set other_set);
};
```

Bag Object

- A Bag object is an unordered collection of elements that may contain duplicates.

```
class Bag : Collection {

attribute  bag<t>value;
unsigned   long occurrences_of(in Object element);
Bag        create_union(in Bag other_bag);
Bag        create_intersection(in Bag other_bag);
Bag        create_difference(in Bag other_bag);
};
```

List Object

- A List object is an ordered collection of elements.
- The operations defined in the List interface are positional in nature, in reference either to a given index or to the beginning or end of a List object.
- Indexing of a List object starts at zero.
- List interface defines operations for selecting, updating, and deleting elements from a list.

List Object

```
class List : Collection {
exception InvalidIndex{unsigned long index; };
attribute list<t>value;
void   remove_element_at(inunsigned long index)
       raises(InvalidIndex);
Object retrieve_element_at(in unsigned long index)
       raises(InvalidIndex);
void   replace_element_at(in Object element, in unsigned long index)
       raises(InvalidIndex);
void   insert_element_after(in Object element, in unsigned long index)
       raises(InvalidIndex);
void   insert_element_before(in Objectelement, in unsigned long index)
       raises(InvalidIndex);
void   insert_element_first (in Object element);
void   insert_element_last (in Object element);
void   remove_first_element() raises(ElementNotFound);
void   remove_last_element() raises(ElementNotFound);
Object retrieve_first_element() raises(ElementNotFound);
Object retrieve_last_element() raises(ElementNotFound);
…...
};
```

Array Object

- An Array object is a dynamically sized, ordered collection of elements that can be located by position.

```
class Array : Collection {

exception InvalidIndex{unsigned long index; };
exception InvalidSize{unsigned long size; };
attribute array<t> value;
void   replace_element_at(in unsigned long index, in Object element)
       raises(InvalidIndex);
void   remove_element_at(inunsigned long index)
       raises(InvalidIndex);
Object retrieve_element_at(in unsigned long index)
       raises(InvalidIndex);
void   resize(in unsigned long new_size)
       raises(InvalidSize);
};
```

Dictionary Object

- A Dictionary object is an unordered sequence of key-value pairs with no duplicate keys.
- Each key-value pairs is constructed as an instance of:

  Struct Association {Object key; Object value;};

```
class Dictionary: Collection {
exception  DuplicateName{string key; };
exception  KeyNotFound{Object key; };
attribute  dictionary<t,v>value;
void       bind(in Object key, in Object value)
           raises(DuplicateName);
void       unbind(in Object key) raises(KeyNotFound);
Object     lookup(in Object key) raises(KeyNotFound);
Boolean    contains_key(in Object key);
};
```

ODMG Objects

- Collection_object
    - `Set<>`
    - `Bag<>`
    - `List<>`
    - `Array<>`
    - `Dictionary<>`
- Structured_object
    - `Date`
    - `Time`
    - `Timestamp`
    - `Interval`
- Atomic_object

Structured Objects

- All structured objects support the Object ODL interface.
- Date
- Interval – represents a duration of time and are used to perform some operations on Time and Timestamp objects.
- Time – denote specific world times, which are internally stored in GMT.
- Timestamp – consist of an encapsulated Date and Time.

Literals

- A Literal has a current value but no object identifier..
- A literal is basically a constant value, possibly having a complex structure that does not change.
- Three types of literals:
  - Atomic
  - Collection
  - Structured

Atomic Literals

- Numbers and characters are examples of atomic literal types.
- Instances of these types are not explicitly created by applications, but rather implicitly exist.
    - long
    - long long
    - short
    - unsigned long
    - unsigned short
    - float
    - double
    - boolean
    - octet
    - char (character)
    - string
    - enum (enumeration)

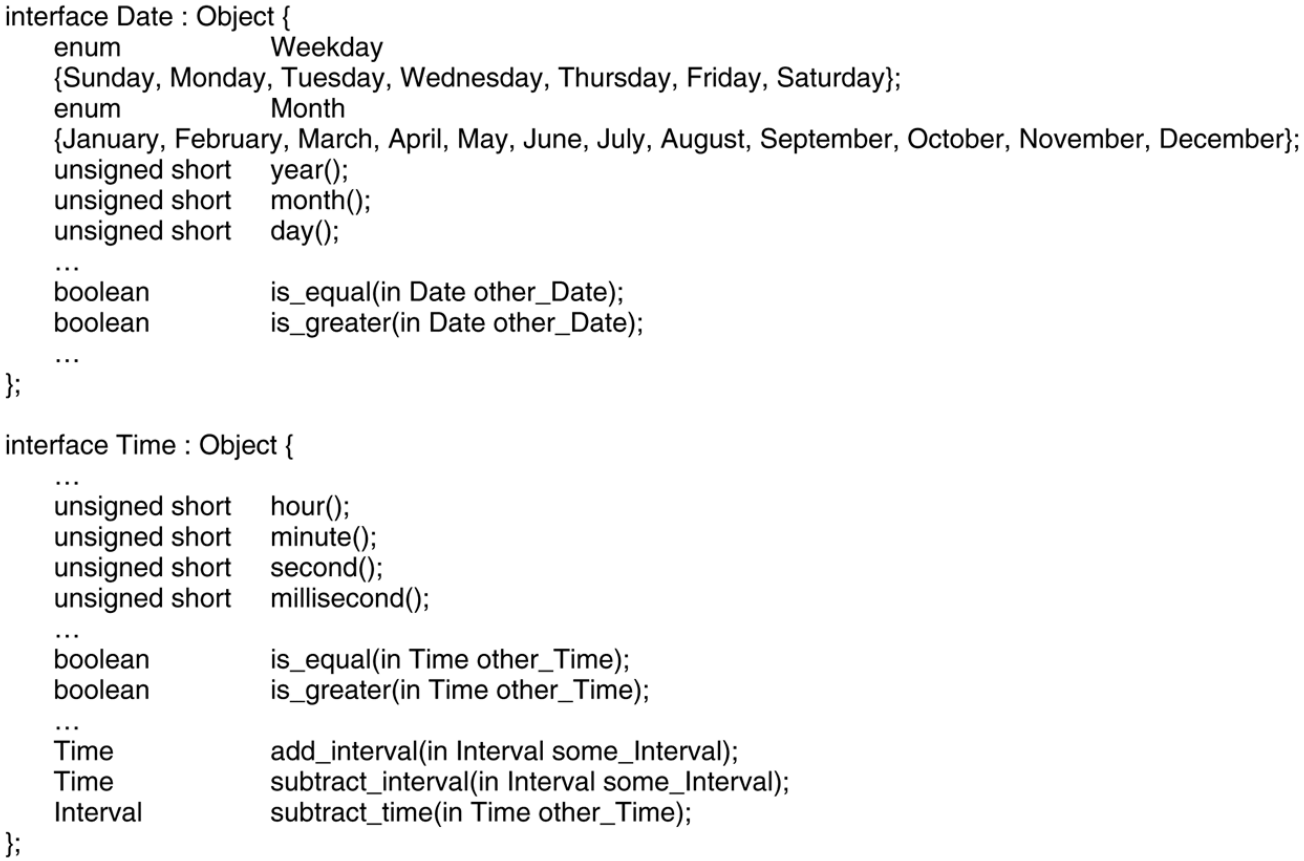
Collection Literals

- Collection literal specify a value that is collection of objects or values.
- These are analogous to those of collection objects, but these collections do not have object identifiers.
  - set<t>
  - bag<t>
  - list<t>
  - array<t>
  - dictionary<t, v>

Structured Literals

- A structured literal has a variable name and can contain whether a literal value or an object.
- They include built-in structures as well as any user-defined structures
  - date
  - interval
  - time
  - timestamp

Structured Literals



```
interface Date : Object {
        enum                    Weekday
        {Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday};
        enum                    Month
        {January, February, March, April, May, June, July, August, September, October, November, December};
        unsigned short          year();
        unsigned short          month();
        unsigned short          day();
        …
        boolean                 is_equal(in Date other_Date);
        boolean                 is_greater(in Date other_Date);
        …
};

interface Time : Object {
        …
        unsigned short          hour();
        unsigned short          minute();
        unsigned short          second();
        unsigned short          millisecond();
        …
        boolean                 is_equal(in Time other_Time);
        boolean                 is_greater(in Time other_Time);
        …
        Time                    add_interval(in Interval some_Interval);
        Time                    subtract_interval(in Interval some_Interval);
        Interval                subtract_time(in Time other_Time);
};
```

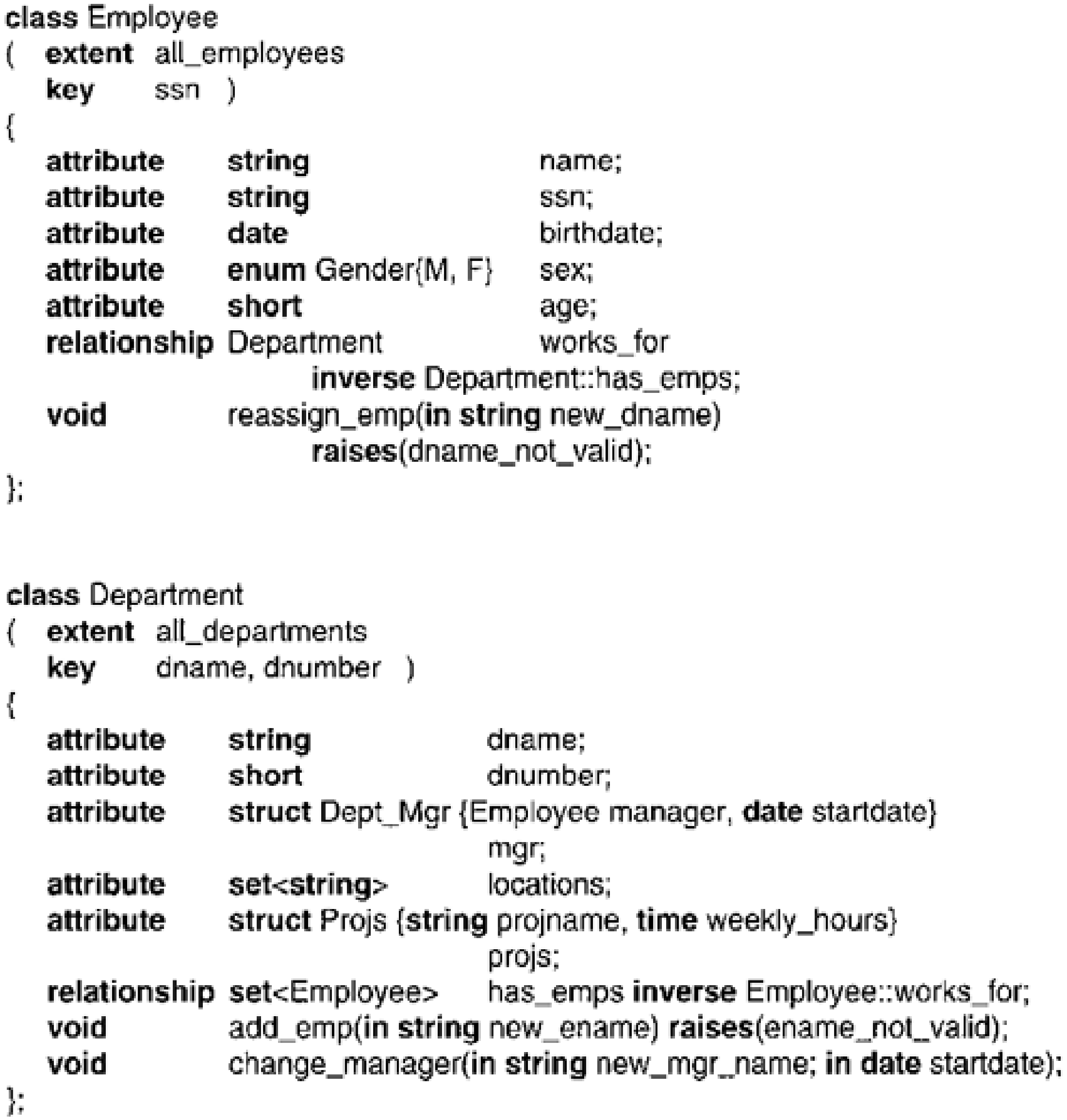
Atomic Objects

- A **Atomic objects** are user-defined objects and are defined via keyword **class.**
- Atomic object contains properties and operations.
- Properties define the state of the object that has:

  attributes and relationships
- Attribute
  - It is a property that describes some aspect of an object
  - Attributes have values (literals with simple or complex) that are stored within an object.
  - Can also be Object_id of other objects.

Atomic Objects

- Relationship specifies that two objects in the database are related.
- Only binary relationships are represented.
- Represented by a pair of *inverse references* specified via relationship.



```
class Employee
(   extent  all_employees
    key      ssn  )
{
    attribute      string                      name;
    attribute      string                      ssn;
    attribute      date                        birthdate;
    attribute      enum Gender{M, F}           sex;
    attribute      short                       age;
    relationship   Department                  works_for
                          inverse Department::has_emps;
    void           reassign_emp(in string new_dname)
                          raises(dname_not_valid);
};


class Department
(   extent  all_departments
    key      dname, dnumber  )
{
    attribute      string                 dname;
    attribute      short                  dnumber;
    attribute      struct Dept_Mgr {Employee manager, date startdate}
                                          mgr;
    attribute      set<string>            locations;
    attribute      struct Projs {string projname, time weekly_hours}
                                          projs;
    relationship   set<Employee>          has_emps inverse Employee::works_for;
    void           add_emp(in string new_ename) raises(ename_not_valid);
    void           change_manager(in string new_mgr_name; in date startdate);
};
```

Class Extent

- An ODMG object can have an extent defined via a class declaration.
- Each extent is given a name and will contain all persistent objects of that class.
- For Employee class, for example, the extent is called all_employees
- This is similar to creating an object of type Set<Employee> and making it persistent.

Class Key

- A class key consists of one or more unique attributes.
- For the Employee class, the key is ssn.
  - Thus each employee is expected to have a unique ssn.
- Keys can be composite, e.g.,(**key** dnumber, dname)

Object Factory

- An object factory is used to generate individual objects via its operations.
- An example:

  ```
  interface ObjectFactory {
     Object new ();
  };
  ```

  `new()` returns new objects with an `object_id`
- One can create their own factory interface by inheriting the above interface.

Interface and Class Definition

- ODMG supports two concepts for specifying object types:
  - **Interface**
  - **Class**
- There are similarities and differences between interfaces and classes
- Both have behaviors (operations) and state (attributes and relationships)

Interface and Class Definition

- An interface is a specification of the abstract behavior of an object type – which the specifies object signature.
- State properties of an interface (i.e., its attributes and relationships) cannot be inherited from.
- Interfaces are used to specify abstract operations.
- Objects cannot be instantiated from an interface – **noninstantiable**.
- A class is a specification of both abstract behavior and abstract **state** of an object type.
- A class is **Instantiable** – one can create object instances.

Interface and Class Definition

- Behavior inheritance
  - Interfaces can be inherited by Classes or by other interfaces.
  - Specified by colon (:) notation.
  - Supertype is interface, subtype could be a class / interface.
- Extends inheritance
  - Supports **“extends”** inheritance to allow both state and behavior inheritance among classes.
  - Both supertype and subtype must be classes.
  - **Multiple inheritance** via “extends” is not allowed.

References

- Fundamentals of Database Systems, by Ramez Elamsri, Navathe.
- The Object Data Standard : ODMG 3.0, by Catell, Douglas Barry