{2, 5, 7}"s -> {2, 5, 7}

or variables whose value is of that type

called immutable objects (e.g. {2, 5, 7})

objects are either values or the type

class consists of a type and possibly one or more methods.

Classes and Objects:

memory addresses, disk addresses (for persistent objects), etc.

reference types

decre, list, set, bag of T

collection types

T, F, T, T, F, ... T

record structures (struct)

type constructors:

base types: integers, real numbers, booleans, character strings

Type System:

Objective Orientation: Types, Classes, Objects
Key concept in reliable software development:
modify the state of an object in a method.
Classes are also "abstract data types" because the only way to
Abstract Data Types:
associated with a class.
Methods:
old must be valid at all time for persistent objects.
oId is unique (i.e., no two objects can have the same old)
Object Identity:
Object Orientation: Object Identity, Methods, ADTs
CLASS SavingAccount: Account

{  
  withdraw(m: Real) {  
    deposit(m: Real) {  
      owner: REF Customer;  
      balance: Real;  
      accountNumber: Integer;  
    }  
  }  
}

Example:

- C may also extend the type of D by adding new fields.
- C may also extend the type of D by adding new methods in addition or in place of super-class methods.
- However, C may have additional properties
- C inherits all properties of D, including:
- C inherits all properties of D, including:
- C inherits all properties of D, including:

Subject: Class Hierarchy

Object Orientation: Class Hierarchy
ODM: Object Data Management (a standards group)
Each movie is produced by one studio.
Studios produce one or more movies.
Stars may act in one or more movies.
Movies have stars acting in them.

Consider database about movies, stars and studios.

- attributes
- relationships
- methods

ODL classes (called interfaces): contain definitions for
ODL Schema for Movie database

{ relationship SetMovie omits inverse Movie::owned?
 attribute string address;
 attribute string name;
 } Interface Studio::

{ relationship SetMovie star::Inverse Movie::stars;
 attribute string addr {string street, string city; address;
 attribute string name;
 } Interface Star;

{ relationship Studio omits inverse Studio::owns;
 relationship SetStar stars::Inverse Star::stars;
 attribute enum Film {color, blackandwhite}; film
 attribute integer length;
 attribute integer year;
 attribute string title;
 } Interface Movie;
Struct N {T1, T2, ..., Tn}.

Non-collection Types:

Array<T, Tn> array of objects of type T
List<T>
Bag<T>
Collection Types: Set<T>

Type Constructors:

Interface Types: such as Movie, Star, Studio etc defined by users.
Atomic Types: Integer, Float, Character, String, Boolean, Enumeration.

Basic Types:
Movie
Set<
Movie>

Integral attribute types:

{Struct N {string field, integer field2}}
{Struct N {string field, integer field2}}
{Struct N {string field, integer field2}}

Integer attribute type examples:

Atomic type or structure. Then we may apply a collection type to the initial

an atomic type or a structure of atomic types

The type of an attribute is built starting with

Restrictions on Attribute Types
atomic types are not allowed in relationship types.

Note: interface types are not allowed in attribute types and

\[
\text{Set<\text{Array<Star}>}
\]
\[
\text{Set<integer>}
\]

\{\text{Struct } N \text{ Movie Field1, Start Field2}\}

\text{Integral relationship types:}

\[
\text{Bag<Star>}
\]
\[
\text{Movie}
\]

- relationship type examples:

  - an interface type.
  - either an interface type or a collection type applied to
  - the type of a relationship is

\text{Restrictions on Relationship Types}
Subclasses in ODL

interface Movie {
    string key (title, year);
    ... // defining keys in ODL
    interface MurderMystery: Movie {
        // attribute string weapon;
    }
    interface Cartoon: Movie {
        attribute Set<Star> voices;
    }
}

Subclasses, Keys
Move(title, year, length, type)

translates into

{
  attribute enumeration(color, blackandwhite) type;
  attribute integer length;
  attribute integer year;
  attribute string title;
  interface Movie
}

Example 1: (simple atomic properties)

properties -> attributes
classes -> relations

ODT Designs to Relational Designs
Example 2: (non-atomic property: struct)

ODL Designs to Relational Designs – continued

Star(name, street, city)

translates into

{ attribute struct Addr {string street, string city, address;
  attribute string name;
 } interface Star

Star(name, street, city)

translates into

{ attribute struct Addr {string street, string city, address;
  attribute string name;
 } interface Star

Example 2: (non-atomic property: struct)
Each "multi-valued attribute" results in a separate relation.

```java
Star (name, street, city)
Star (name, birthdate)
```

translates into

```java
{ 
  attribute Date birthdate;
  attribute Set<String Address {string street, string city}} address;
  attribute String name;
} 
```

Example 3: (set constructor)

ODL Designs to Relational Designs – continued
To convert to relational designs

ODL Designs to Relational Designs – continued
Studio class

where studiNAME is the primary key of studio relation derived from

Move (... studiNAME, ...)

corresponding relation:

{ ...

relationship studying indirectly inverse studio: comes;
...

} Move Interface

Example 4: (single-valued relationship)

ODL Design to Relational Designs – continued
In case of one-to-many relationship, represent the single-valued side.
In the relational model, only one direction need be represented.

```java
int title, year; key for Movie
name: key for Star

stars(title, year, name)

Corresponding relation for stars relationship:

关系 Relation for stars relationship:

{ ... 

relationship relationship

set< Star, Set< Star >>

stars inverse stars: star

Example 4: (multi-valued relationship)

ODL Designs to Relational Designs - continued
Converting Subclasses to relations:

Consider the Movie $\rightarrow$ Cartoon Movie $\rightarrow$ MurderMystery

interface MurderMystery: Movie {
  interface Cartoon: Movie {
    relationship Set<Star> voices;
  }
  attribute string weapon;
}
search all 3 relations.

However to query common attributes such as length, we have to

All information in one place for a particular movie:

MurderMysteryStars(title, year, starName)

Mystery(titre, year, length, titleType, studiName, weapon)

Vociis(title, year, voicename)

CartoonStars(title, year, starName)

tinosaur(title, year, length, titleType, studiName)

MovieStarts(title, year, length, titleType, studiName)

(inherited ones)

(1) one relation per subclass; includes all attributes (including

3 approaches:

ODL Designs to Relational Designs – continued
Querying common attributes done on one relation.
Information for a particular movie scattered around:

MurderMystery(title, year, weapon)

Voice(title, year, voiceName)

Cartoon(title, year)

MovieStars(title, year, starName)

Movie(title, year, theme, filmType, studiOname)

(2) one relation per subclass;
Movietesters(title,year,starName)

Movie(title,year,genre,libtype,studioName,weapon)

Voices(title,year,voiceName)

(3) one relation with all attributes with lots of null values.

ODL designs to Relational Designs - continued
there is no guarantee function implements what their names suggest!

- functions may raise exceptions
- in, out, inout parameters in methods
- all getters refer to the extent of a class, not to the class name
- extent of the class: name for the current set of objects in the class

{%
  otherPrints(in Star, out Set<Movie> raises(NoSuchFile))
  StarNames(out Set<String>)
  Float LentthHours() raises (NoSuchFileDialog)

  } }

interface Movie (extent: Movie key (title, year))

Declaring method signatures in OD: (code is not part of OD)

Query-related features of OD:

Querying OO Databases
are comparable (values easily transferred between the two)

- Much easier to embed OQL queries in host language since both

or many be embedded in a host language such as C++, Java.

- OQL queries may be interpreted (as in SQL*Plus or Oracle)

OQL (Object Query Language -- OMG standard)
struct (title: "Laws", year: 1981),
struct (title: "ET", year: 1985),
set (struct (title: "My Fair Lady", year: 1965),
  struct (foo: bar (2, 1), bar: "baz")
examples: baz (2, 1, 2)

struct (...) array (...) list (...) bag (...) set ...
- Complex Types: built using enumerations declared in ADL!
  strings, and booleans (surrounded by "")
- Basic Types: atomic types; integers, floats, characters,
  constants are constructed as follows:
  ODL Type System:

OCL - continued
result of applying p to a
- if p is a method (perhaps with parameters) then a.p is the
  objects returned to a by relationship p
- if p is a relationship then a.p is the object or collection of
  in object a
- if p is an attribute then a.p is the value of that attribute

a.p is a path expression interpreted as follows:

    p: some property of the class (attribute/relationship/method)
    a: an object belonging to class C

Path Expressions: OQL - continued
Examples of path expressions:

OQL - continued
where m.title = "Gone With the Wind"
from movies m
select m.year

(1) Find the year of movie "Gone With the Wind"

QL Questions:

QL - continued
The OQL query produces a bag of objects:

```
WHERE keyword followed by a boolean valued expression (can use only
  - followed by the name of the variable
  - followed by an optional AS keyword
  - an extent? could be another select-from-where
  - giving an expression whose value is a collection type (typically
    variable is declared by
  FROM keyword followed by a list of variable declarations

  SELECT keyword followed by a list of expressions (using constants and
  SELECT from-where statement in OQL is constructed as follows:
```

OQL - continued
where m.ownedBy.name = "Disney"
from Movies m, m.stars
select distinct s.name

Find the names of stars of "Disney" movies

(2) Find the names of the stars of "Casablanca"

(3) Eliminating duplicates (distinct keyword)

where m.title = "Casablanca"
from Movies m, m.stars
select s.name

OCT - continued
The result type of this query is Set<Struct N {star1: Star, star2: Star}>

Note: Such a type cannot appear in an ODL declaration.

(4) Get set of pairs of stars living at the same address

select distinct Struct(star1: s1, star2: s2)
from Stars s1, Stars s2
where s1.addr = s2.addr
and s1.name < s2.name

Complex output type:

\[
OQL - continued
\]
where w.ownedBy.name = "Disney"
from Movies m
from (select m
select distinct s.name

follows:
Gives us the Disney Movies. This can be used in the from clause as

where w.ownedBy.name = "Disney"
from Movies m
select m

(5) Get the stars in movies made by Disney.

Suggestions:
asc or desc may be specified after order by (default is asc)

order by m. length, m. title
where m. owned by. name = "Disney"
from movies m
select m

(6) get Disney movies ordered by length (ties broken by title)

Ordering the Result:

QL continued...
WHERE FOR ALL m in s.starrelden : m.ownedBy.name = "Disney"

from Stars s
select s

(6) Get stars who appear only in Disney movies

WHERE EXISTS m in s.starrelden : m.ownedBy.name = "Disney"

from Stars s
select s

(7) Get stars acting in Disney movies

EXISTS x in S : c(x)

for all x in S : c(x)

Quantifier Expressions:
(note: set of movie lengths would be incorrect!)
a bag of movie lengths is created; then the avg operator is applied.

avg(select m. length from movies m)

(6) Find the average length of all movies.

min, max apply to any collection in which the members can be compared.
sum, avg apply to any collection of numbers
count applies to any collection

These apply to collections whose members are of a suitable type.
same 5 operations as in SQL: avg, min, max, sum, count

Aggregate Expressions:

OQL - continued
(select distinct m)

where s.name = "Harrison Ford"

from Movies m, m.stars

except

("

where m.ownedBy.name = "Disney"

from Movies m

select distinct m

)
The result of the query is a set if both operands are sets otherwise it is a bag.

\[ x \text{ appears } \min(n_1, n_2) \text{ times in } (B_1 \cap \text{intersection } B_2)\]

\[ x \text{ appears } n_2 \text{ times in } (B_1 \cup \text{union } B_2)\]

\[ x \text{ appears } 0\text{ times in } (B_1 \text{ difference } B_2) \text{ if } n_1 \geq n_2 \text{ (n}_1-n_2) \text{ times otherwise}\]

Note: If the one of both operands of these set operations is a bag, the "bag" meaning is used. Say \( x \) appears \( n_1 \) times in \( B_1 \) and \( n_2 \) times in \( B_2 \).
element function extracts single element from bag of one element.

\[
\text{GETPM} = \text{element \{select m from MOVIES where m.title = "Gone With the Wind"\}}.
\]

MOVIE \text{GETPM};

Extracting Elements of Collections:

\[
\text{SELECT movies = select distinct m from MOVIES where m.year > 1920;}.
\]

\[
\text{SET MOVIE=movies};
\]

Assigning Values to host variables:

Object Assignment and Creation in OQL. OQL and host Language (Good Fit?).

OQL - continued
```java
{
    System.out.println("Movie: " + movie + ", Year: " + year);
    for (int i = 0; i < numMovies; i++) {
        int num_movie_titles = countMovieTitles();
        movie_titles = (select m from Movie m order by m.title, m.year);
        if (movie_titles.size() > i) {
            movie = movie_titles.get(i);
            System.out.println("Movie: " + movie + ", Year: " + year);
        }
    }
}
```

A small program fragment to display movie titles, years and genres:

Instead of bag/set.

Note: order by clause automatically converts the result type of query to a list.

Extracting each element from a collection:
\texttt{ms = \texttt{oldmovies except \texttt{set(\texttt{GTMW})};}}

\texttt{GTMW = MOVIE(title="Gone With the Wind", year:1949, length:239,\texttt{ownedBy:mem});}

\texttt{y = \texttt{bag(x,struct(a:3, b:4));}}

\texttt{x = \texttt{struct(a:1, b:2);}}

\texttt{Creating New Objects:}

\texttt{OCL - continued}