In The Name Of Allah

Intro to ODBMSs

A Background for Relational DBs

> Relational DBMSs support a **small** , **fixed** collection of data types (e.g. integer, dates, string, etc.) which has proven adequate (مناسب) for traditional application domains such as administrative and business data processing.

> RDBMSs support very **high-level queries** , query optimization, transactions, backup and crash recovery, etc.

- However, many other application domains need **complex kinds** of data such as `CAD/CAM`, multimedia repositories, and document management.
- To support such applications, DBMSs must *support complex data types* .
- Object-oriented strongly influenced efforts to enhance database support for complex data and led to the development of object-database systems.

Object-Oriented Database Systems.

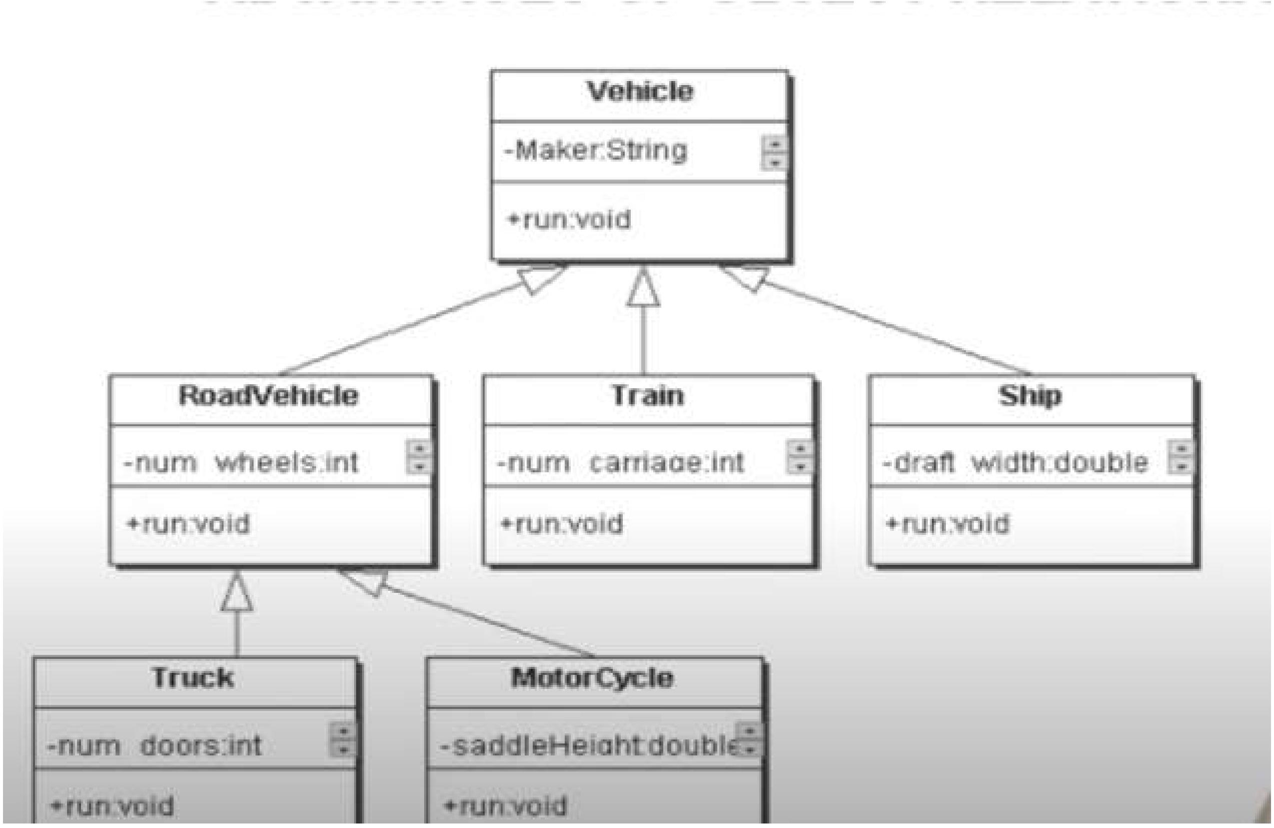
- The approach is heavily influenced by **OO programming** languages and can be understood as an attempt (محاولة) to add DBMS functionality to a programming language environment.
- The **Object Database Management Group (ODMG)** has developed a standard **Object Data Model (ODM)** and **Object Query Language (OQL)** , which are the **equivalent of the SQL standard** for relational database systems.

Object-Relational Database Systems.

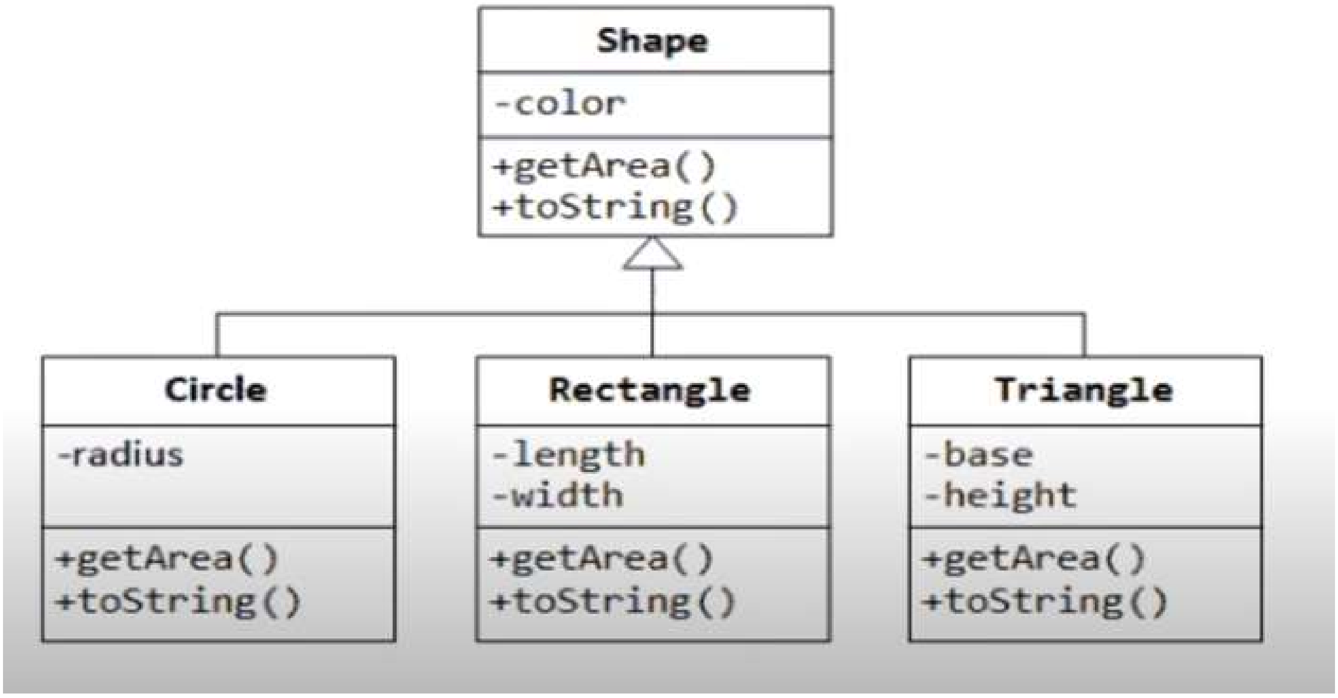
- **ORDB** systems can be thought of as an attempt to extend relational database systems with the functionality necessary to support a broader class of application domains, provide a **bridge** between the relational and object-oriented paradigms.
- This approach attempts to get the best of both.

What are the advantages of Object Relational DataBases?

- Allow the use of inheritance, you can develop classes for your data types.

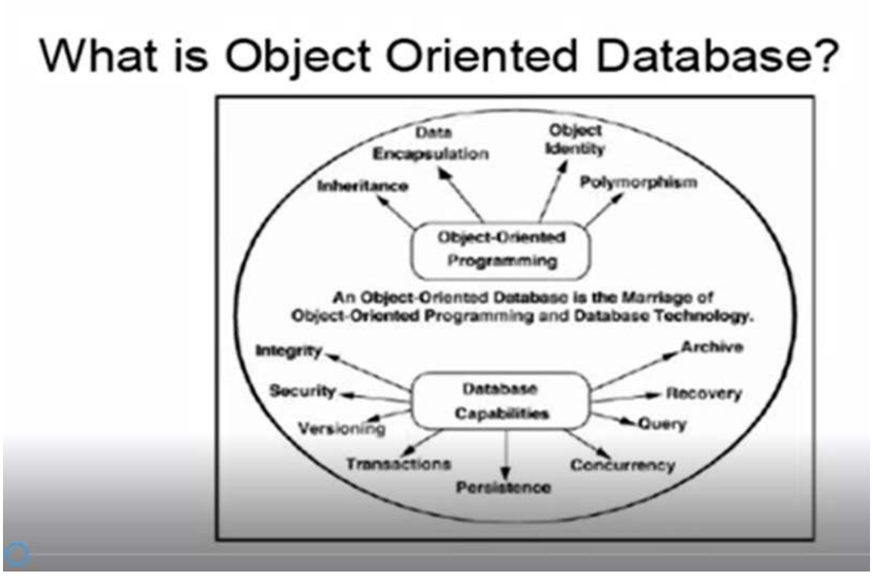


- Allow Polymorphism which involves allowing one operator to have different meanings within the same databases.



What is Object Oriented (OO) Data Base

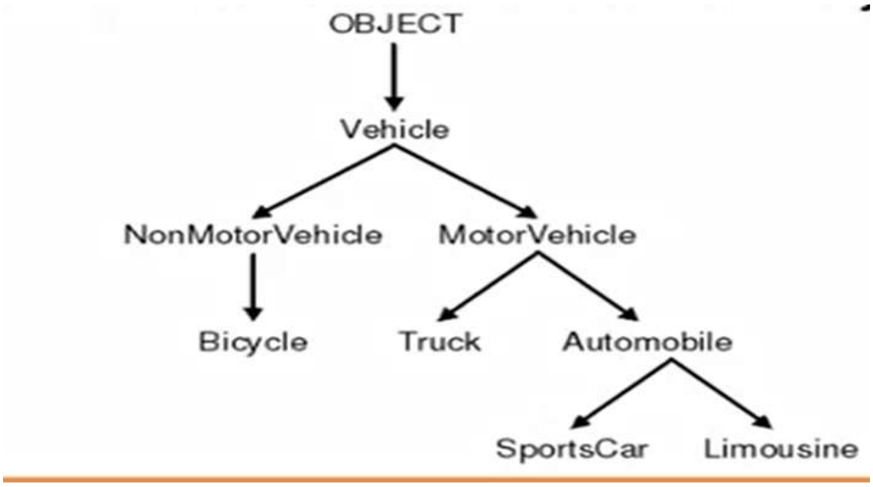
- Object Oriented Data Base (OODB) is all facilities associated with object oriented paradiam .
- It enabled thus creats **Classes , Objects, Structure, Inheeritance hierarchy** and all methods other classes.
- Beside these it also provide facilities with standard data base systems.
- However, OODB have not yet replace RDBMS in commercial business applications .



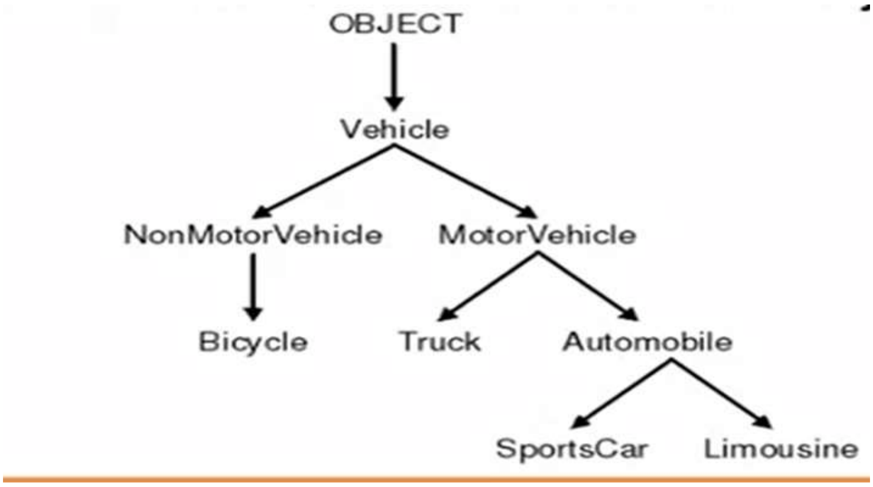
What is Object Oriented Database?

An Object-Oriented Database is the Marriage of Object-Oriented Programming and Database Technology.

- Object oriented databse enables to represent information in the form of **objects** .



- Object oriented data base present **data modeling and programming** in an object oriented environment.

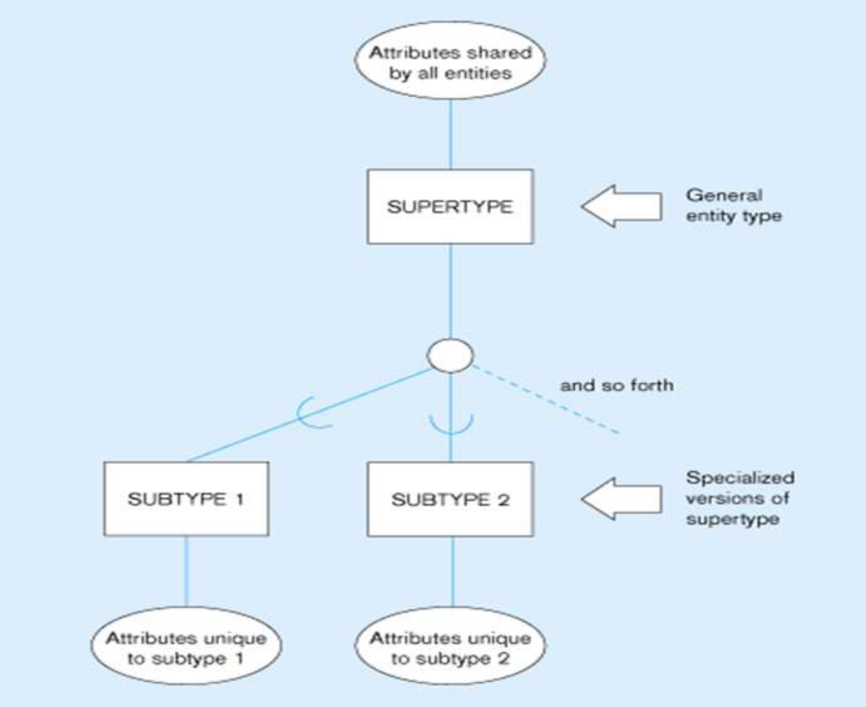


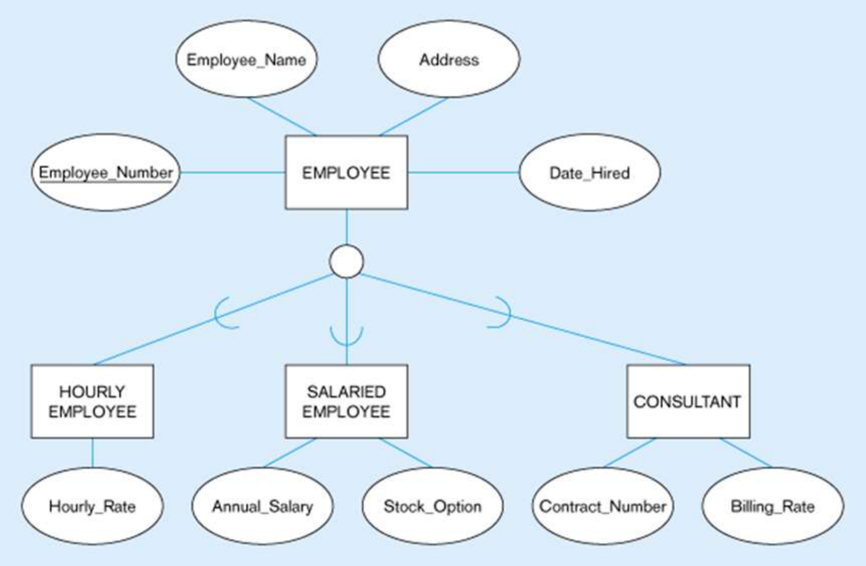
The Enhanced E-R Model and {Business Rules}

- ≫ Supertype & Subtype

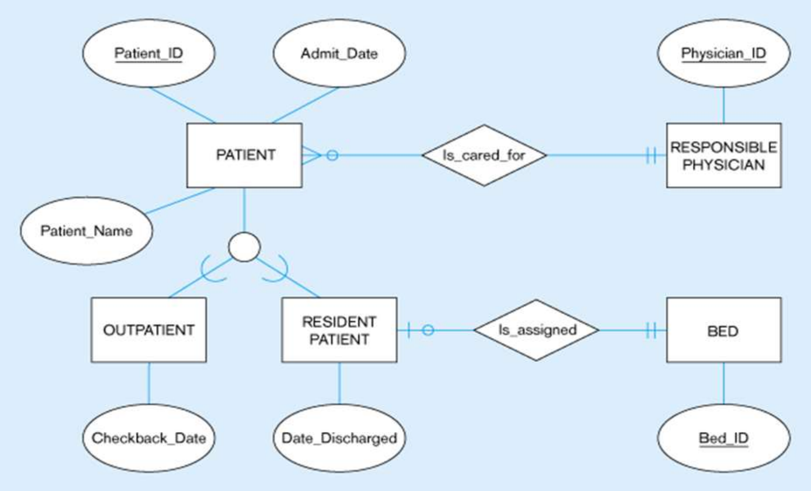
- ≫ Specialization & Generalization
- ≫ Constraints in Supertype
- ≫ Business rules

- Subtype: A **subgrouping** of the entities in an entity type which has attributes that are distinct from those in other subgroupings.
- Supertype: An **generic entity type** that has a relationship with one or more subtypes.
- Inheritance
  - Subtype entities **inherit** values of all attributes of the supertype
  - An instance of a subtype is also an instance of the supertype.


Attributes shared by all entities
SUPERTYPE
General entity type
and so forth
SUBTYPE 1
SUBTYPE 2
Specialized versions of supertype
Attributes unique to subtype 1
Attributes unique to subtype 2



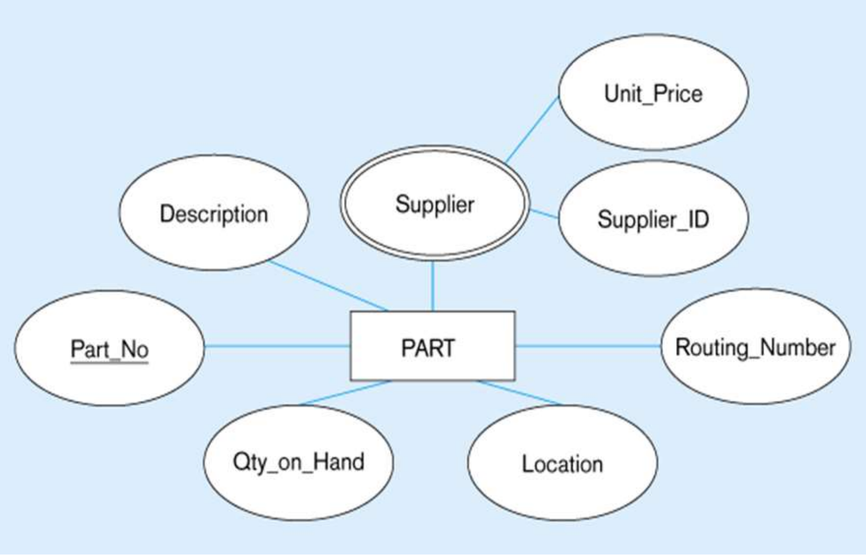
- **Relationships** at the supertype level indicate that all subtypes will participate in the relationship
- The instances of a subtype may participate in a relationship unique to that subtype. In this situation, the relationship is shown at the **subtype level**

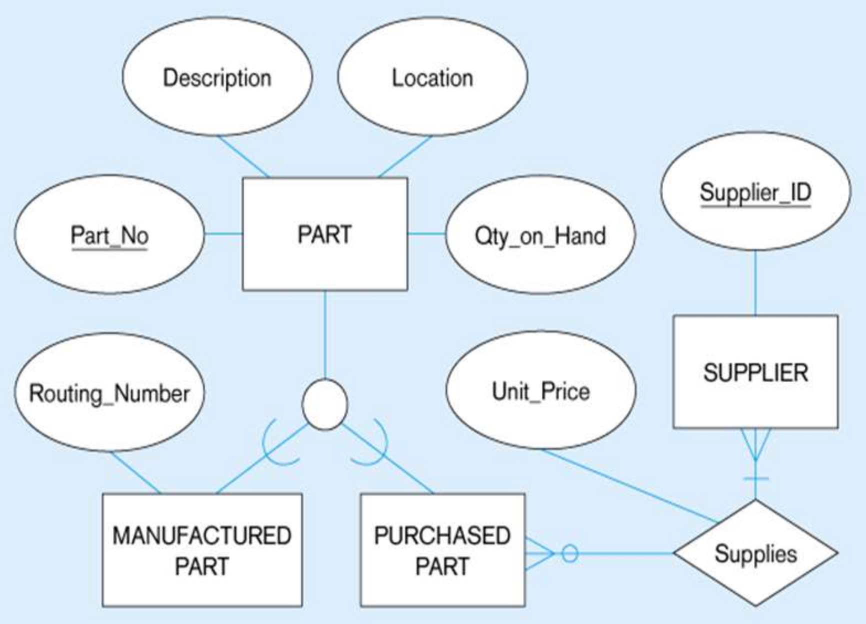


Specialization & Generalization

- **Specialization** : The process of defining one or more subtypes of the supertype, and forming supertype/subtype relationships. **TOP-DOWN**
- **Generalization** : The process of defining a more general entity type from a set of more specialized entity types. **BOTTOM-UP**

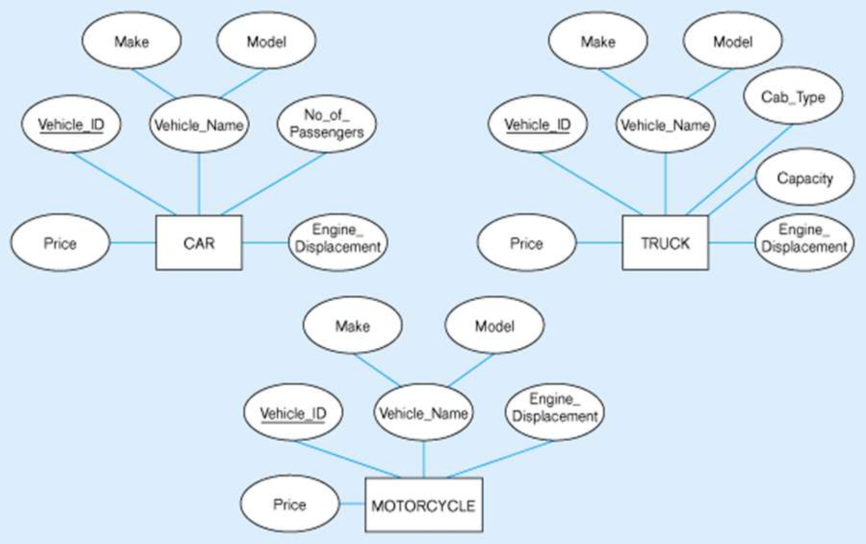
- **Specialization** Example multivalued attribute was replaced by a relationship to another entity!

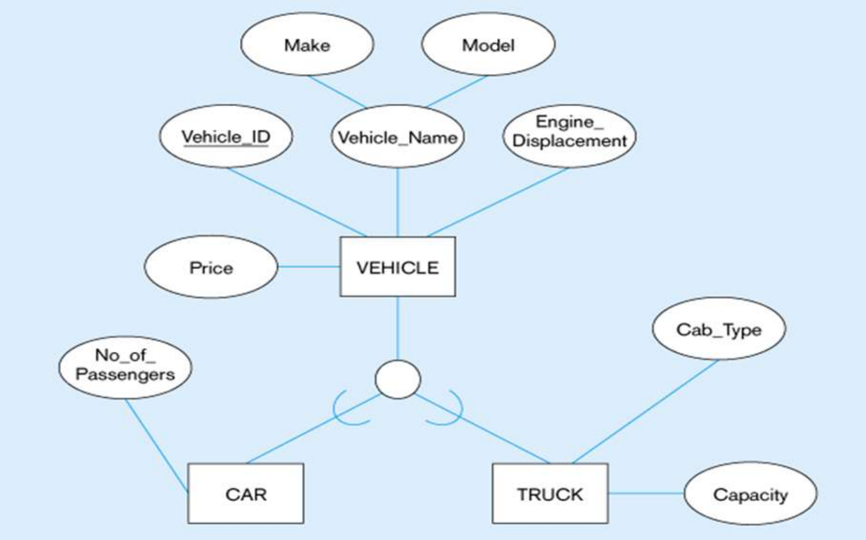




- Generalization Example
- Three entity types: CAR, TRUCK, and MOTORCYCLE

> Note: no subtype for motorcycle, since it has no unique attributes

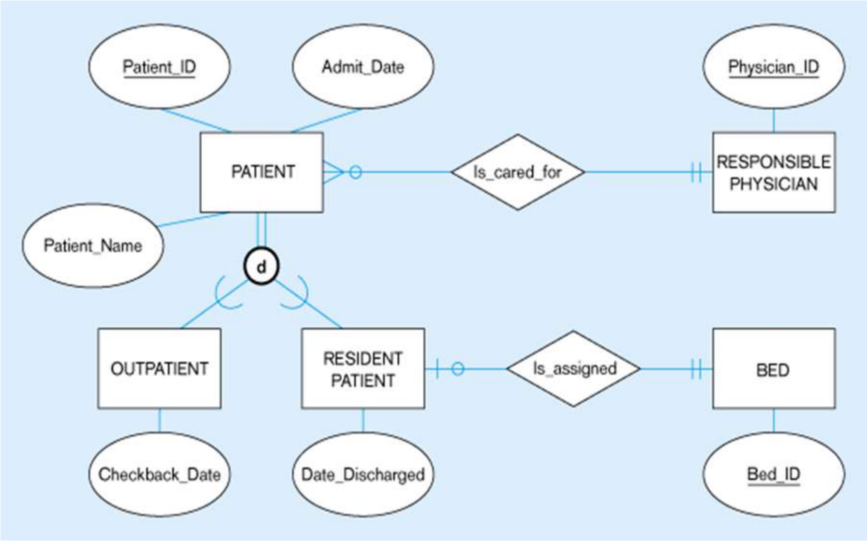


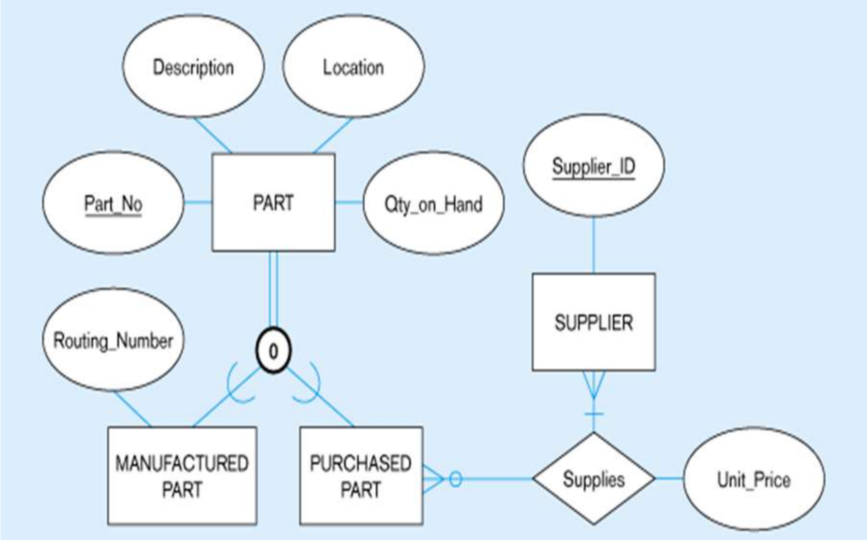


Constraints of Supertype

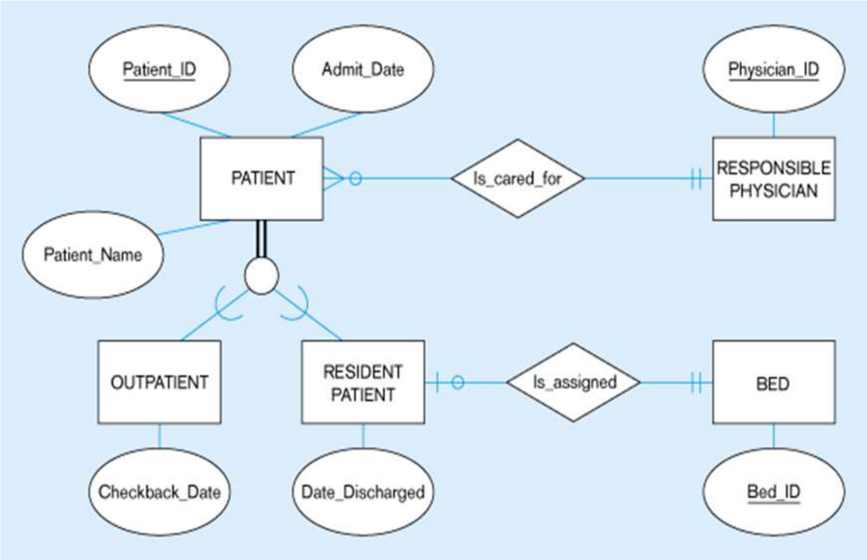
- 1) Disjointness Constraints
- 2) Completeness Constraints
- **Disjointness Constraints** : Whether an instance of a supertype may simultaneously be a member of two (or more) subtypes.
- **Disjoint Rule** : An instance of the supertype can be only ONE of the subtypes.
- **Overlap Rule** : An instance of the supertype could be more than one of the subtypes

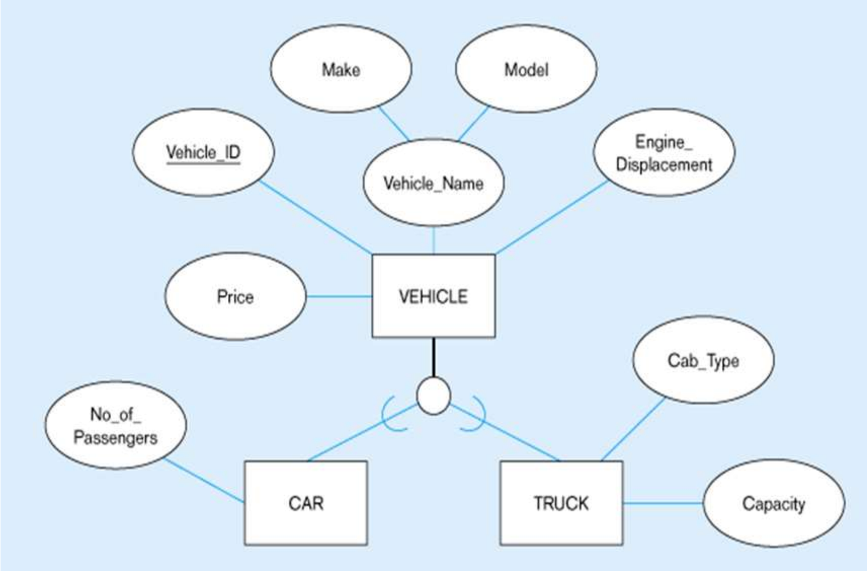
---



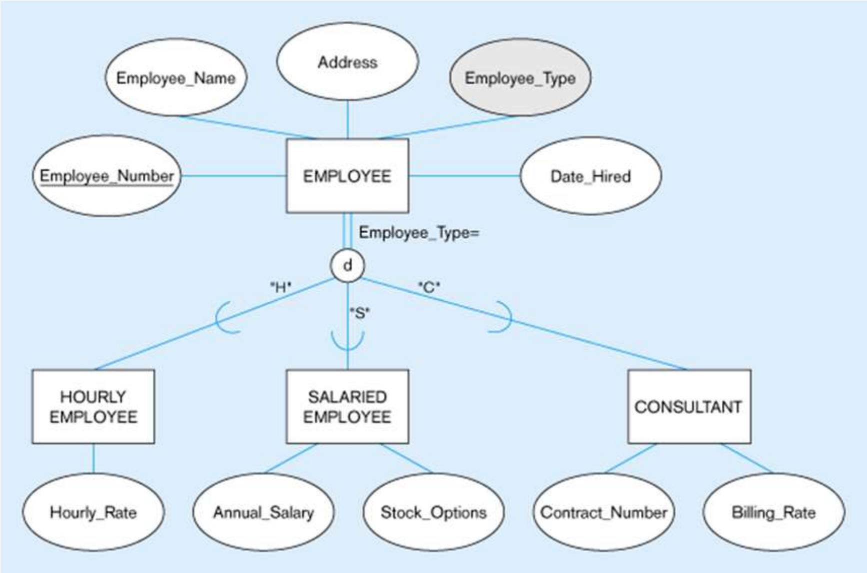


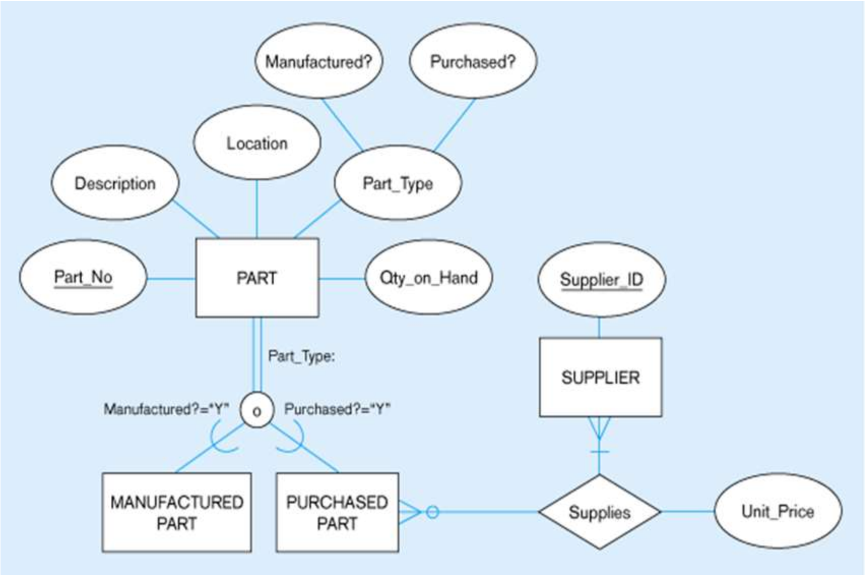
- **Completeness Constraints**: Whether an instance of a supertype must also be a member of at least one subtype
  - Total Specialization Rule: Yes (double line)
  - Partial Specialization Rule: No (single line)





- **Subtype Discriminator** An attribute of the supertype whose values determine the target subtype(s)
  - Disjoint: a simple attribute with alternative values to indicate the possible subtypes
  - Overlapping: a composite attribute whose subparts pertain to different subtypes. Each subpart contains a boolean value to indicate whether or not the instance belongs to the associated subtype





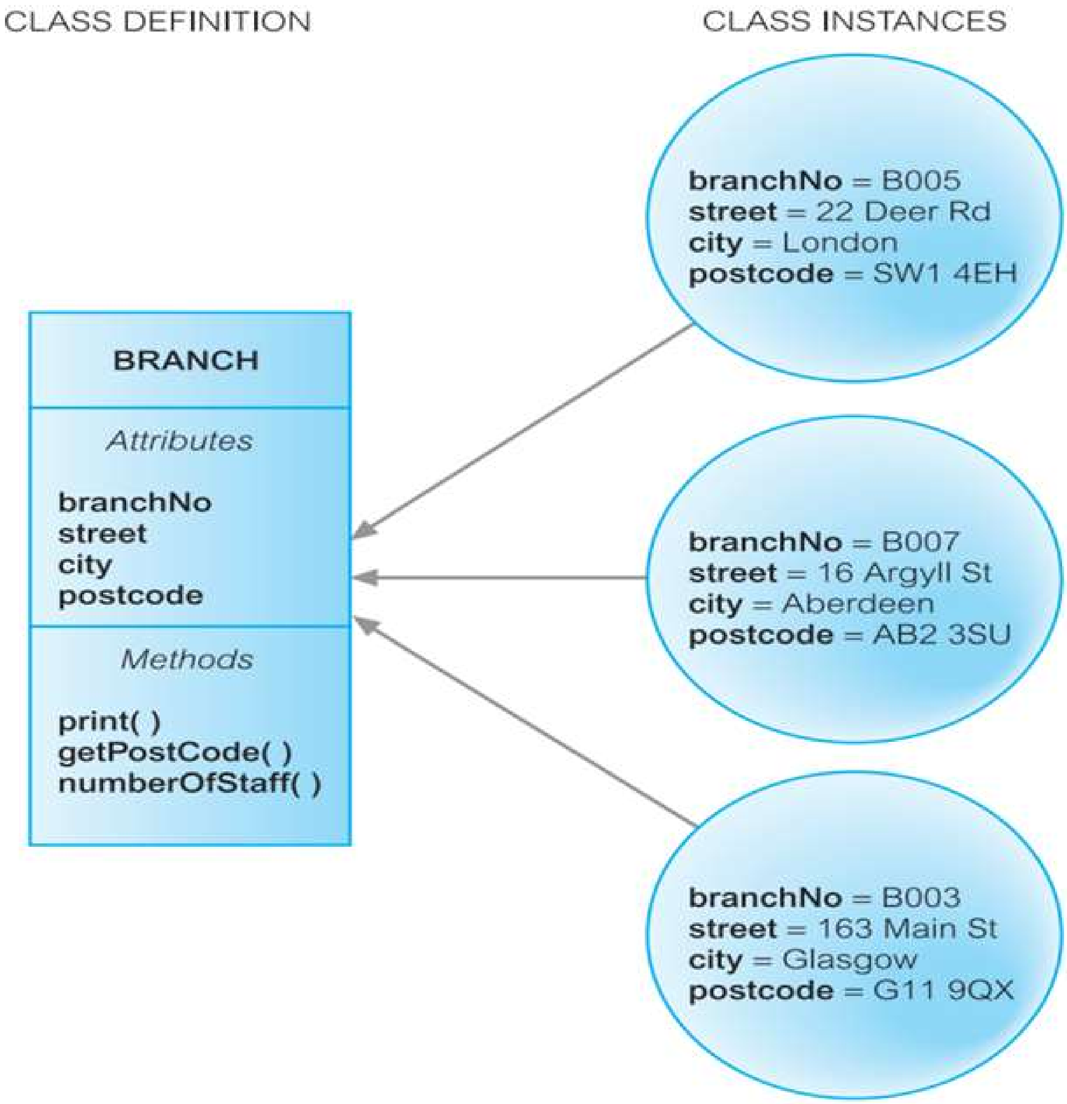
- **Business rules** Statements that define or constrain some aspect of the business.
- Constraints can impact:
  - Structure (definition, domain, relationship)
  - Behavior (operational constraints)
- Classification of business rules:
  - Derivation : rule derived from other knowledge
  - Structural assertion : rule expressing static structure
  - Action assertion : rule expressing constraints/control of organizational actions

Introduction to ODBMS

OOP

> **Object-oriented programming (OOP)** is a programming paradigm based on the concept of "objects", which can contain data and code: data in the form of fields (often known as attributes or properties), and code, in the form of procedures (often known as methods).

- **Objects** – instances of classes
- **Classes** – template for the object ; contain data and procedures (known as class methods).
- classes contain the **data members** and **member functions**.



Methods and Messages

- An object encapsulates both **data and functions** into a **self-contained** package. In object technology, functions are usually called **methods**.
- Messages are the means by which **objects communicate**.
- A **message** is simply a **request** from one object (the sender) to another object (the receiver) asking the second object to execute one of its **methods**.
- The sender and receiver **may be the same object**.

Encapsulation

- **Encapsulation** is an object-oriented programming concept that **binds together the data and functions** that manipulate the **data**, and that keeps both *safe from outside interference and misuse*.
- Data encapsulation led to the important OOP concept of **data hiding**.
- Encapsulation prevents **external code** from being concerned with the internal workings of an object.
- The concept of information hiding means that we **separate** the external aspects of an object from its **internal details**, which are hidden from the outside world.

Abstraction

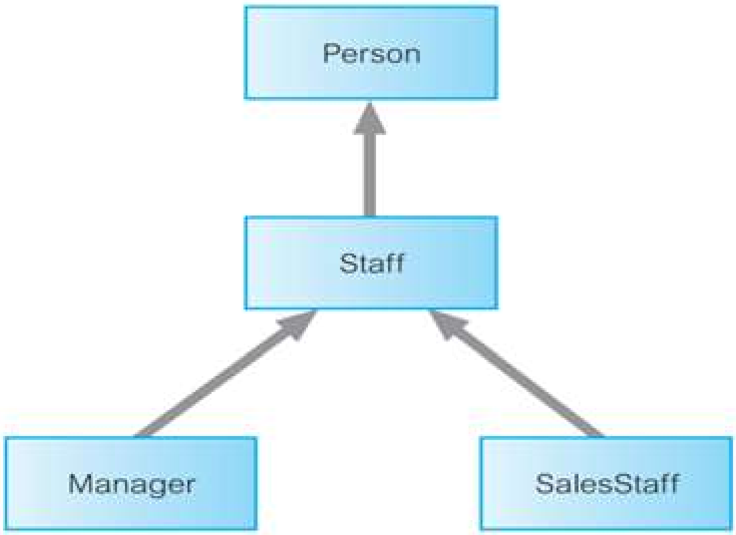
> If a class does not allow calling code to access internal object data and permits access through **methods only** , this is a strong form of abstraction or information hiding known as encapsulation.

Inheritance

- Class Employee might inherit from class Person.
- All the data and methods available to the parent class also appear in the child class with the same names.
- class Person might define variables "first_name" and "last_name" with method "make_full_name()".
- These will also be available in class Employee, which might add the variables "position" and "salary".
- There are several forms of inheritance:
  - single inheritance
  - multiple inheritance
  - repeated inheritance
  - selective inheritance

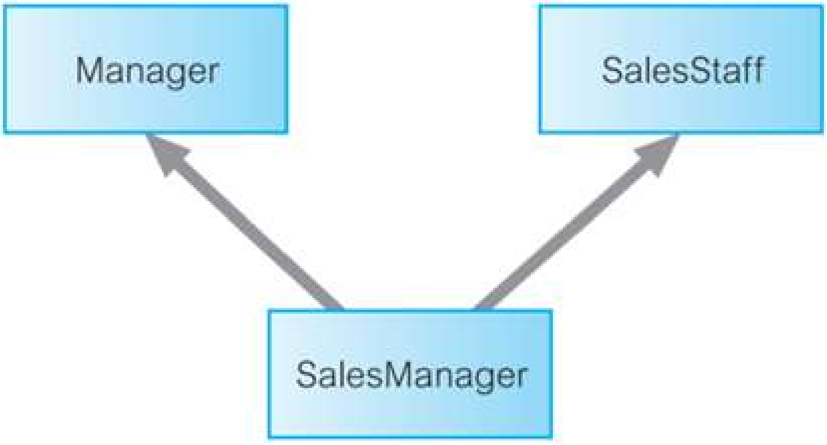
Single Inheritance

- The term *‘single inheritance’* refers to the fact that the subclasses inherit from **no more than one superclass**.
- The superclass Staff could itself be a subclass of a superclass, Person, thus forming a class hierarchy.



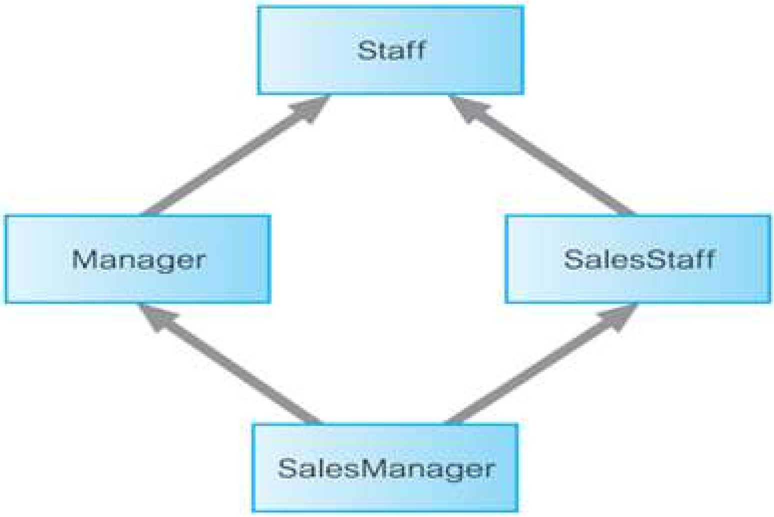
Multible Inheritance

- where the subclass SalesManager inherits properties from both the superclasses Manager and SalesStaff.



Repeated Inheritance

- is a special case of **multiple inheritance** where the superclasses inherit from a **common superclass** .
- the classes Manager and SalesStaff may both inherit properties from a **common superclass Staff** .



Polymorphism

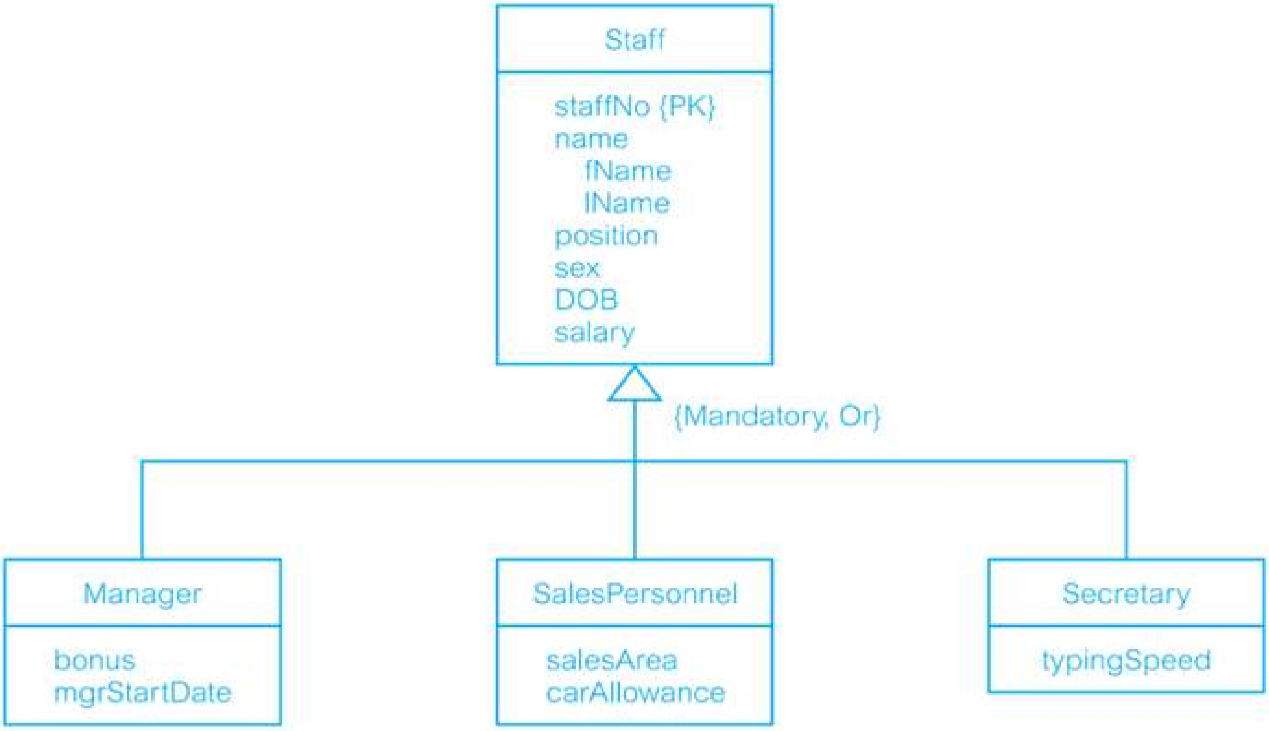
- Subclasses can **override** the methods defined by super classes (Override).
- The class may contain the **same function more than once** with the same name, but with different attributes(Overloading).

ORM (Object Relational Mapping)

> Is a technique for **storing, retrieving, updating, and deleting** from an Object-Oriented program in a relational database.

- This requires mapping class instances (that is, objects) to one or more tuples distributed over one or more relations.

- which has a Staff superclass and three subclasses: Manager, SalesPersonnel, and Secretary.



- To handle this type of class hierarchy, we have two basics tasks to perform:
  - Design the relations to represent the class hierarchy?.
  - Design how objects will be accessed?.
- which means:
  - writing code to decompose the objects into tuples and store the decomposed objects in relations; – writing code to read tuples from the relations and reconstruct the objects.

Types of implementaions

1- Map each class or subclass to a relation.

- this would give the following four relations (with the primary key underlined):
  - Staff (staffNo, fName, lName, position, sex, DOB, salary)
  - Manager (staffNo, bonus, mgrStartDate)
  - SalesPersonnel (staffNo, salesArea, carAllowance)
  - Secretary (staffNo, typingSpeed)

2- Map each subclass to a relation.

- this would give the following three relations:
  - Manager (staffNo, fName, lName, position, sex, DOB, salary, bonus, mgrStartDate)
  - SalesPersonnel (staffNo, fName, lName, position, sex, DOB, salary, salesArea, carAllowance)
  - Secretary (staffNo, fName, lName, position, sex, DOB, salary, typingSpeed)

> to produce a list of all staff we would have to select the tuples from each relation and then union the results together.

3- Map the hierarchy to a single relation.

- Staff (staffNo, fName, lName, position, sex, DOB, salary, bonus, mgrStartDate, salesArea, carAllowance, typingSpeed, typeFlag)

> The attribute typeFlag is a discriminator to distinguish which type each tuple is (for example, it may contain the value 1 for a Manager tuple, 2 for a SalesPersonnel tuple, and 3 for a Secretary tuple).

> Again, we have lost semantic information in this mapping. Further, this mapping will produce an **unwanted number of nulls** for attributes that do not apply to that tuple. For example, for a Manager tuple, the attributes salesArea, carAllowance, and typingSpeed will be null.

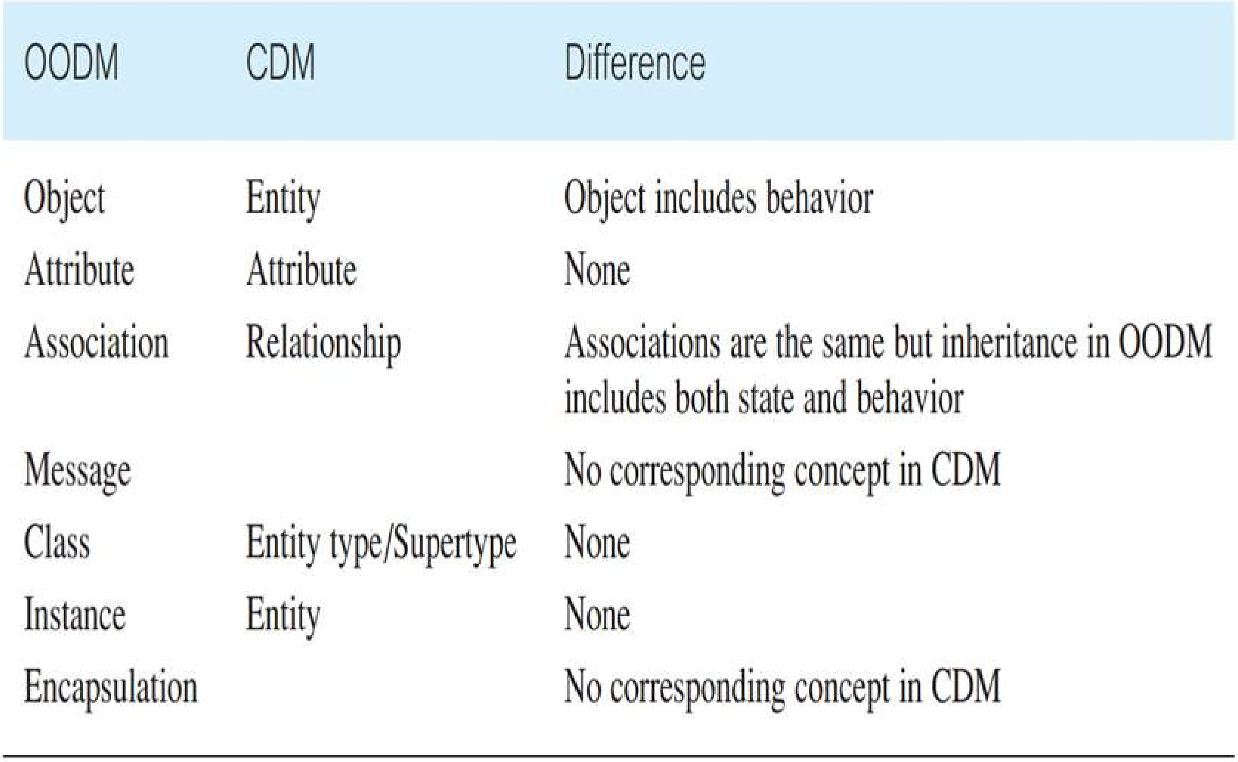
Accessing Objects in the Relational Database

- we now need to insert objects into the database and then provide a mechanism to read, update, and delete the objects.
  - For example, to insert an object into the first relational schema in the previous section (that is, where we have created a relation for each class), the code may look something like the following using programmatic SQL (see Appendix E):

```
Manager* pManager = new Manager; // create a new Manager object called pManager
EXEC SQL INSERT INTO Staff VALUES (:pManager->staffNo, :pManager->fName, :pManager->lName, :pManager->position, :pManager->sex, :pManager->DOB, :pManager->salary);
EXEC SQL INSERT INTO Manager VALUES (:pManager->bonus, :pManager->mgrStartDate);
![image-3.png](attachment:image-3.png)
```

Comparison of Object-Oriented Data Modeling and Conceptual Data Modeling?

- The main difference is the **encapsulation of both state and behavior** in an object,
  - whereas CDM captures only state and has no knowledge of behavior.
  - Thus, CDM has no concept of messages and consequently no provision for encapsulation.



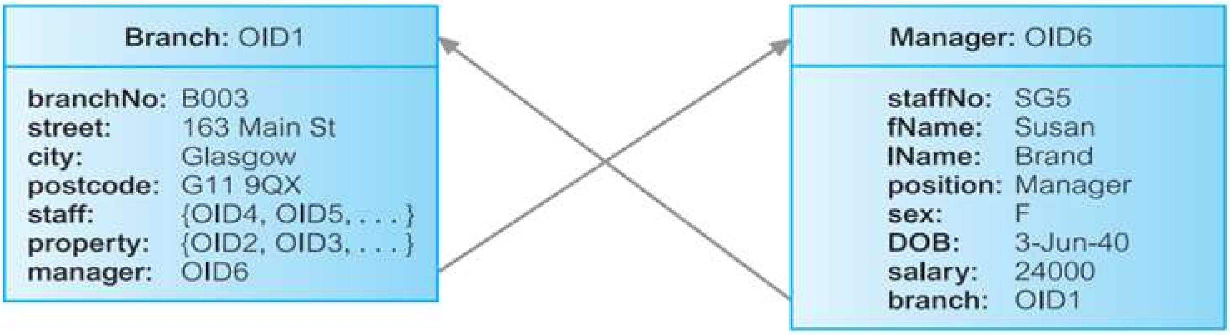
OODM	CDM	Difference
Object	Entity	Object includes behavior
Attribute	Attribute	None
Association	Relationship	Associations are the same but inheritance in OODM includes both state and behavior
Message		No corresponding concept in CDM
Class	Entity type/Supertype	None
Instance	Entity	None
Encapsulation		No corresponding concept in CDM

Relatioships

1:1 relationships

> A 1:1 relationship between objects A and B is represented by adding a reference attribute to object A and, to maintain referential integrity, a reference attribute to object B.

- **Object Identity(OID)**: system-generated; unique to that object;



Branch: OID1

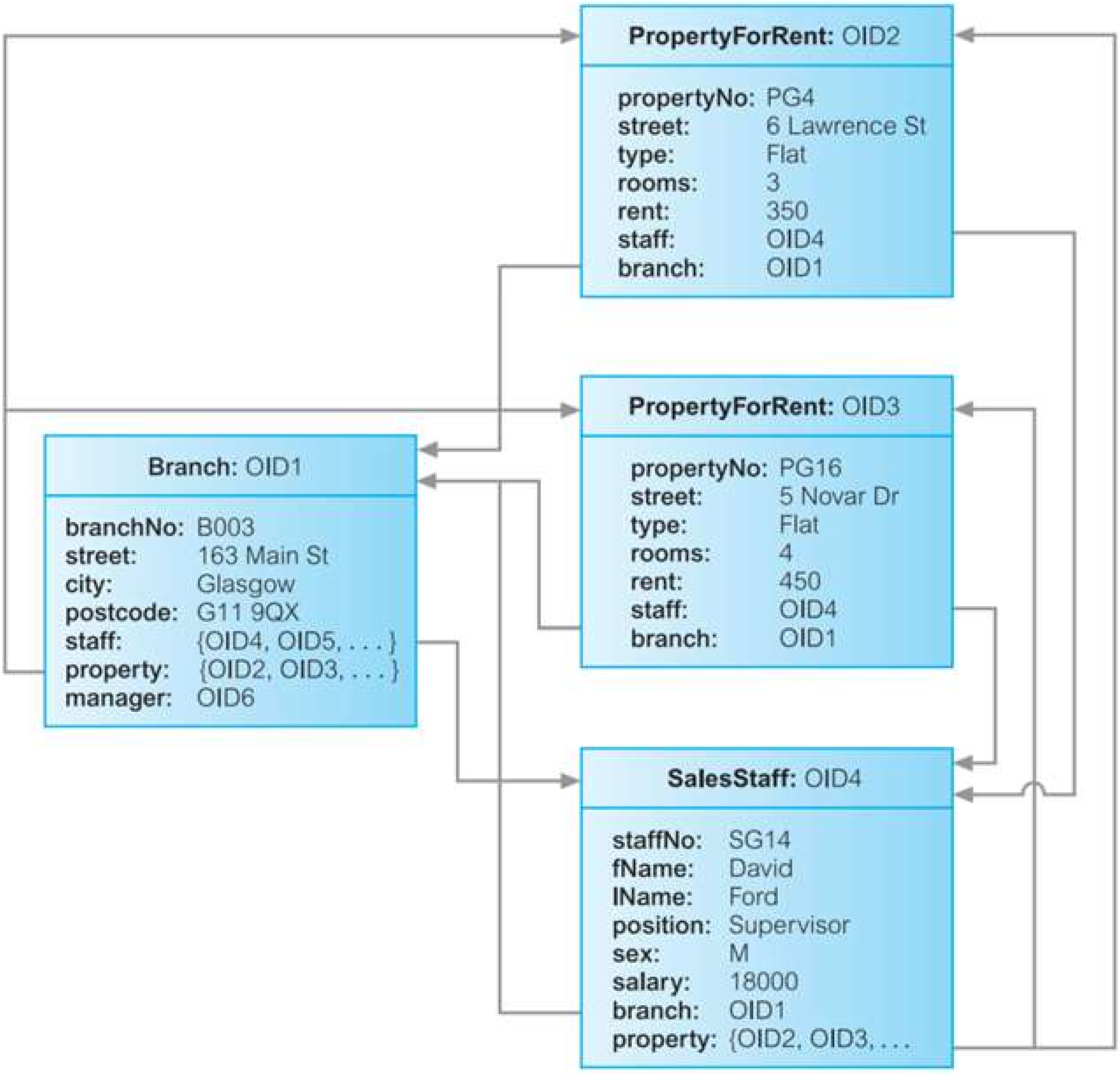
branchNo:	B003
street:	163 Main St
city:	Glasgow
postcode:	G11 9QX
staff:	{OID4, OID5, . . . }
property:	{OID2, OID3, . . . }
manager:	OID6

Manager: OID6

staffNo:	SG5
fName:	Susan
lName:	Brand
position:	Manager
sex:	F
DOB:	3-Jun-40
salary:	24000
branch:	OID1

1:* relationships

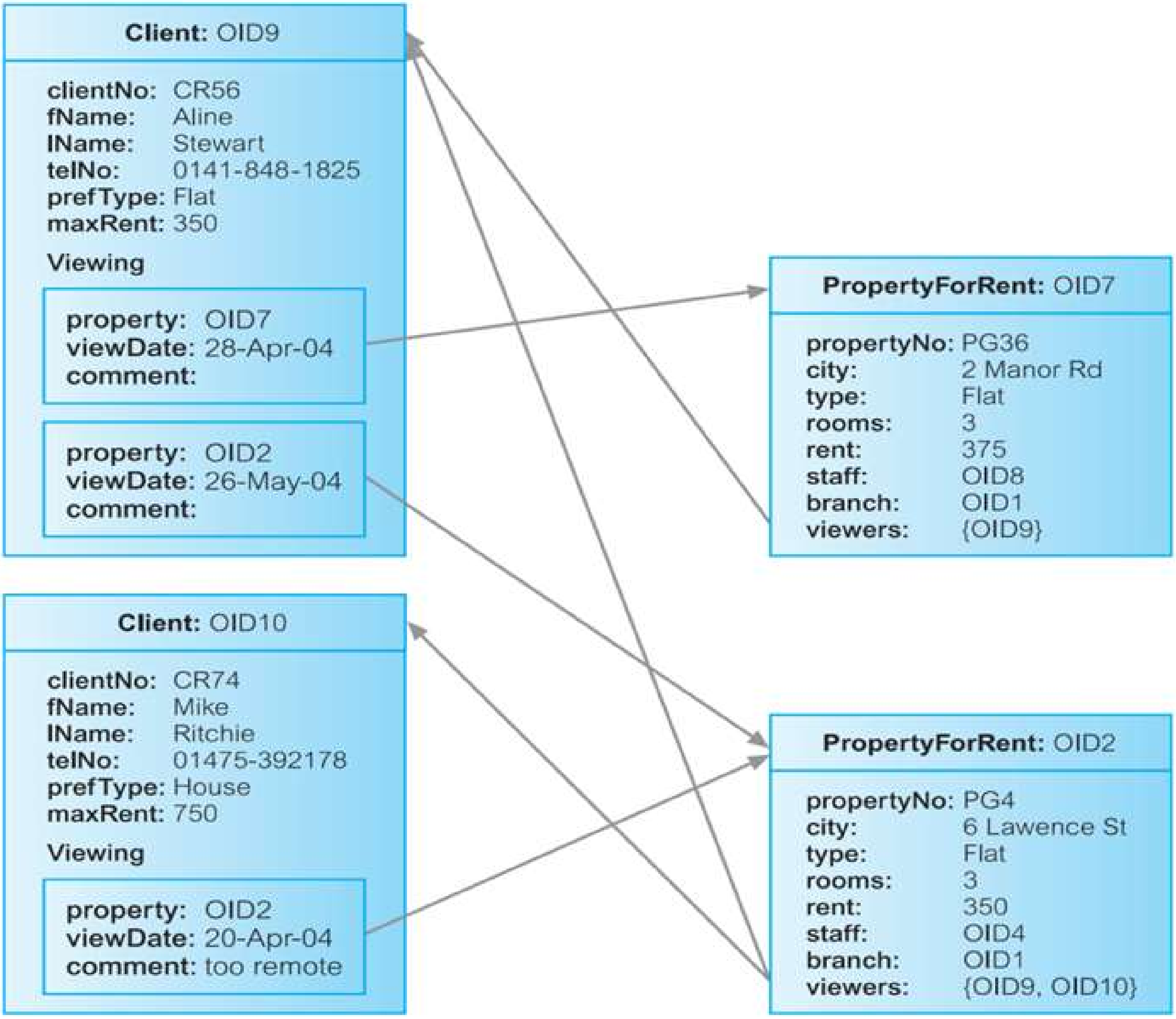
> A 1:* relationship between objects A and B is represented by adding a reference attribute to object B and an attribute containing a set of references to object A.



: relationships

> A *:* relationship between objects A and B is represented by adding an attribute containing a set of references to each object.

- For example, there is a *:* relationship between Client and PropertyForRent,
- For relational database design, we would decompose the *:* relationship into two 1:* relationships linked by an intermediate entity. It is also possible to represent this model in an OODBMS,



Object-Oriented Analysis and Design with UML

> UML (Unified Modeling Language) for _*ER modeling* and conceptual database design.

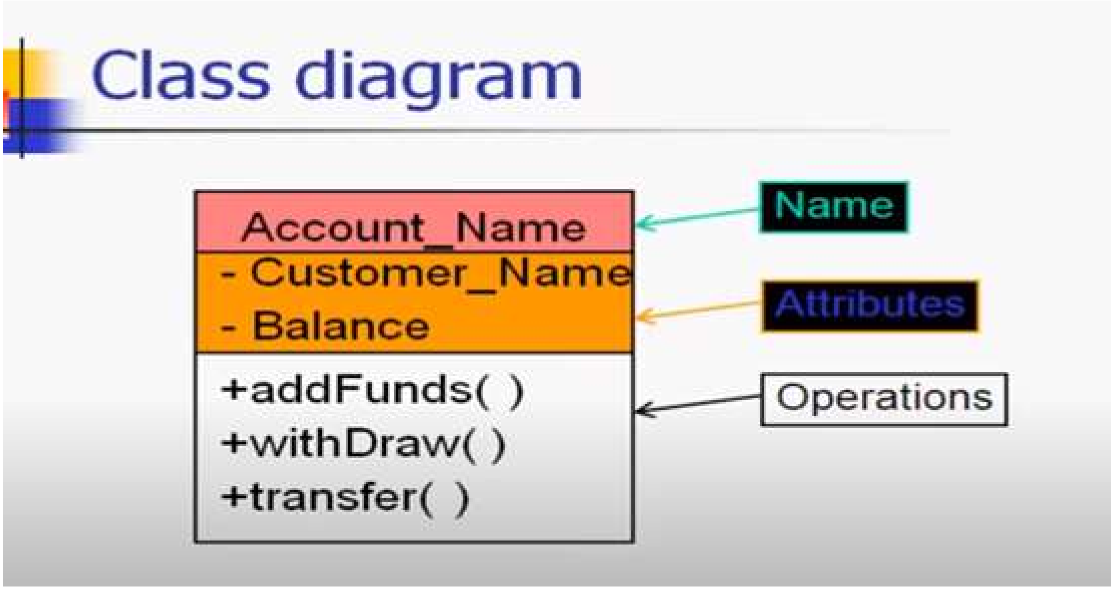
- The primary goals in the design of the UML were to?:
  - Provide users with a ready-to-use, expressive visual modeling language so they can develop and exchange meaningful models
  - Provide extensibility and specialization mechanisms to extend the core concepts.
  - Be independent of particular programming languages and development processes.
  - Provide a formal basis for understanding the modeling language.
  - Encourage the growth of the object-oriented tools market.
  - Support higher-level development concepts such as collaborations, frameworks, patterns, and components.
  - Integrate best practices.

UML Diagrams

UML Diagrams

> Structural diagrams, which describe the static relationships between components.

- class diagrams,



Class diagram

Account_Name

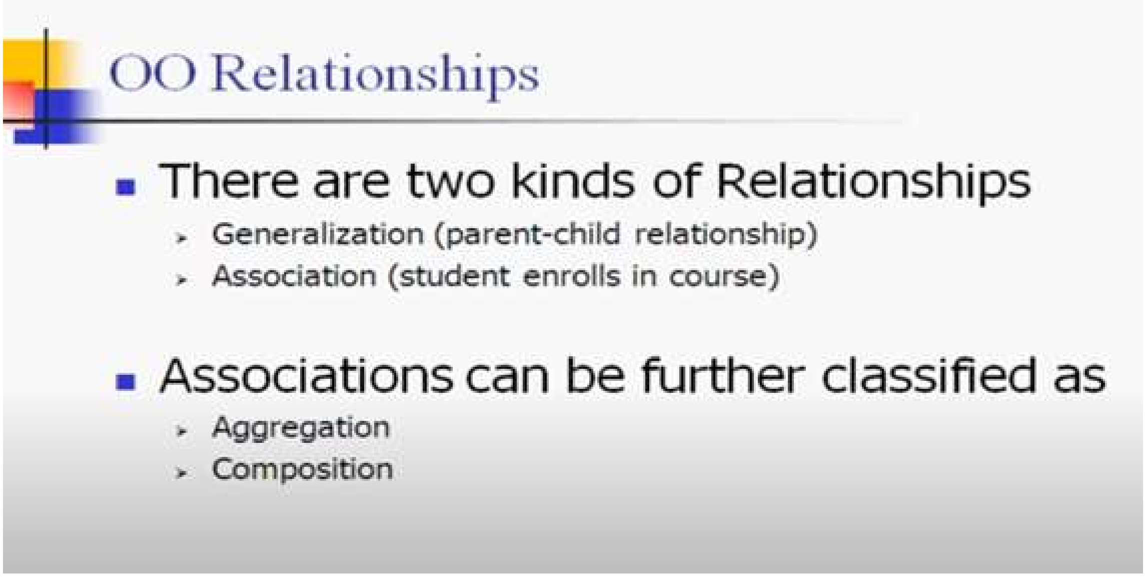
- Customer_Name
- Balance

+addFunds( )
+withDraw( )
+transfer( )

Name

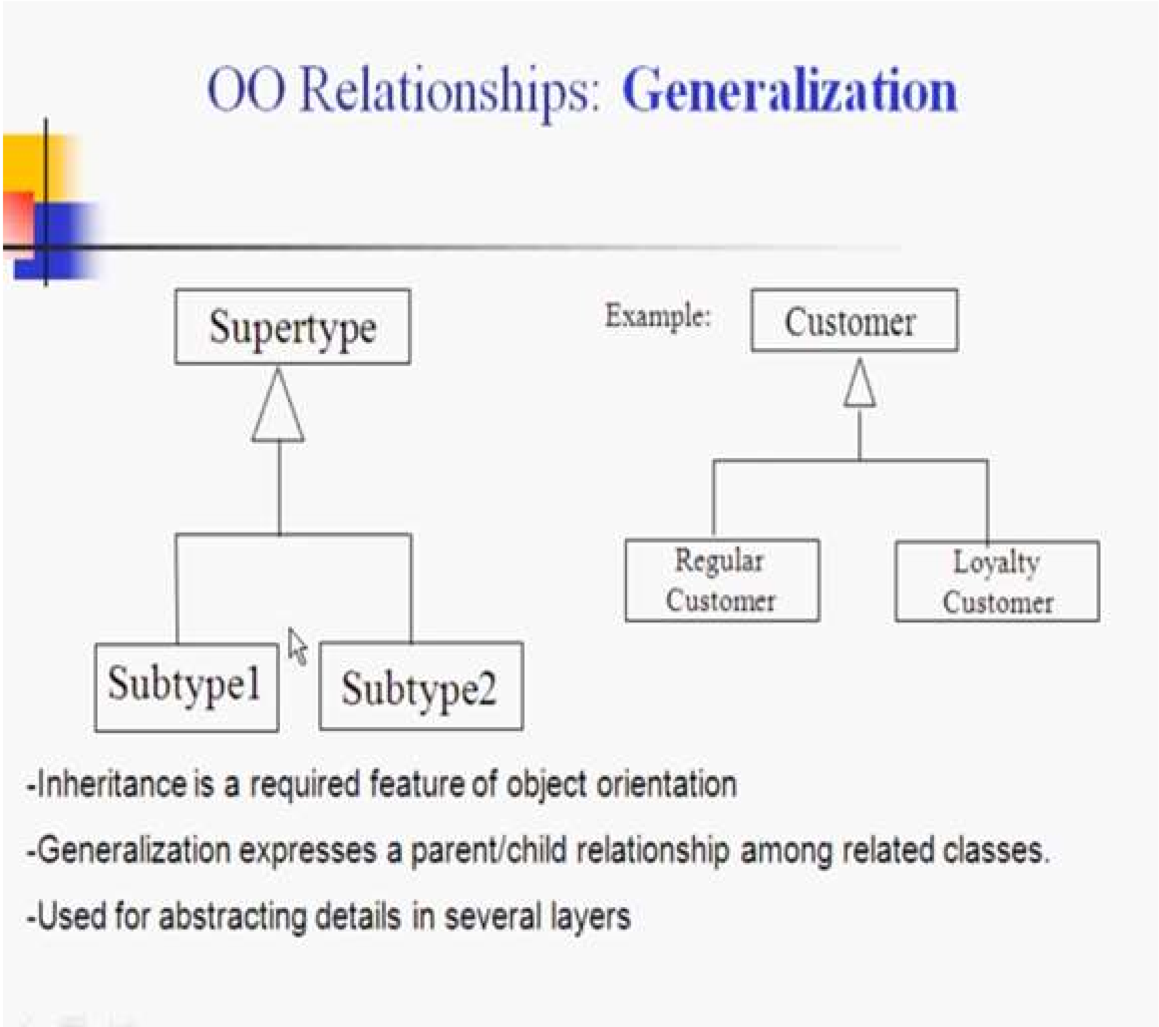
Attributes

Operations



OO Relationships

- There are two kinds of Relationships
  - Generalization (parent-child relationship)
  - Association (student enrolls in course)
- Associations can be further classified as
  - Aggregation
  - Composition

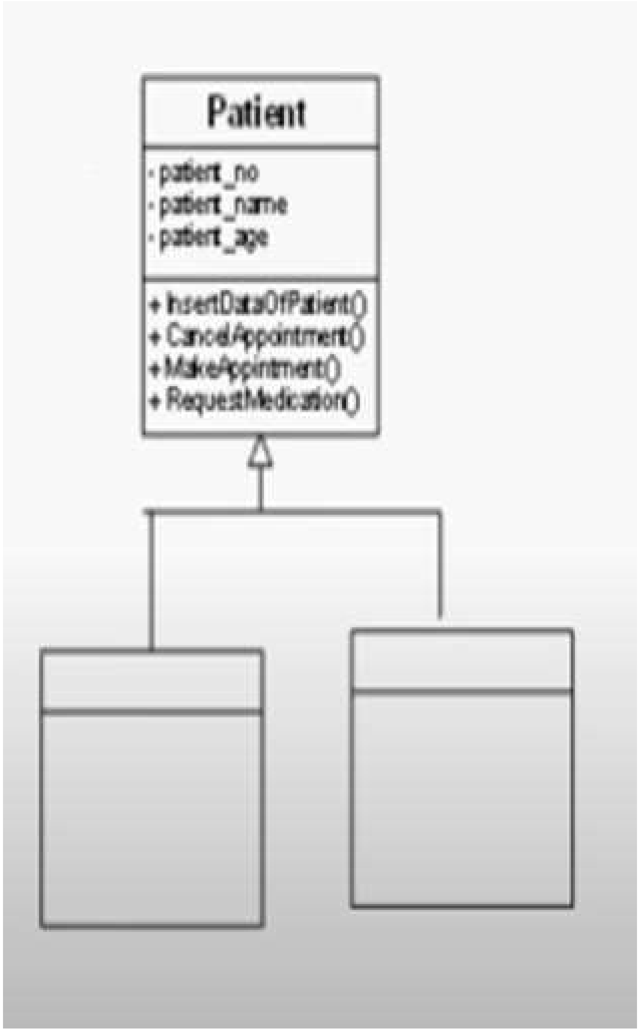


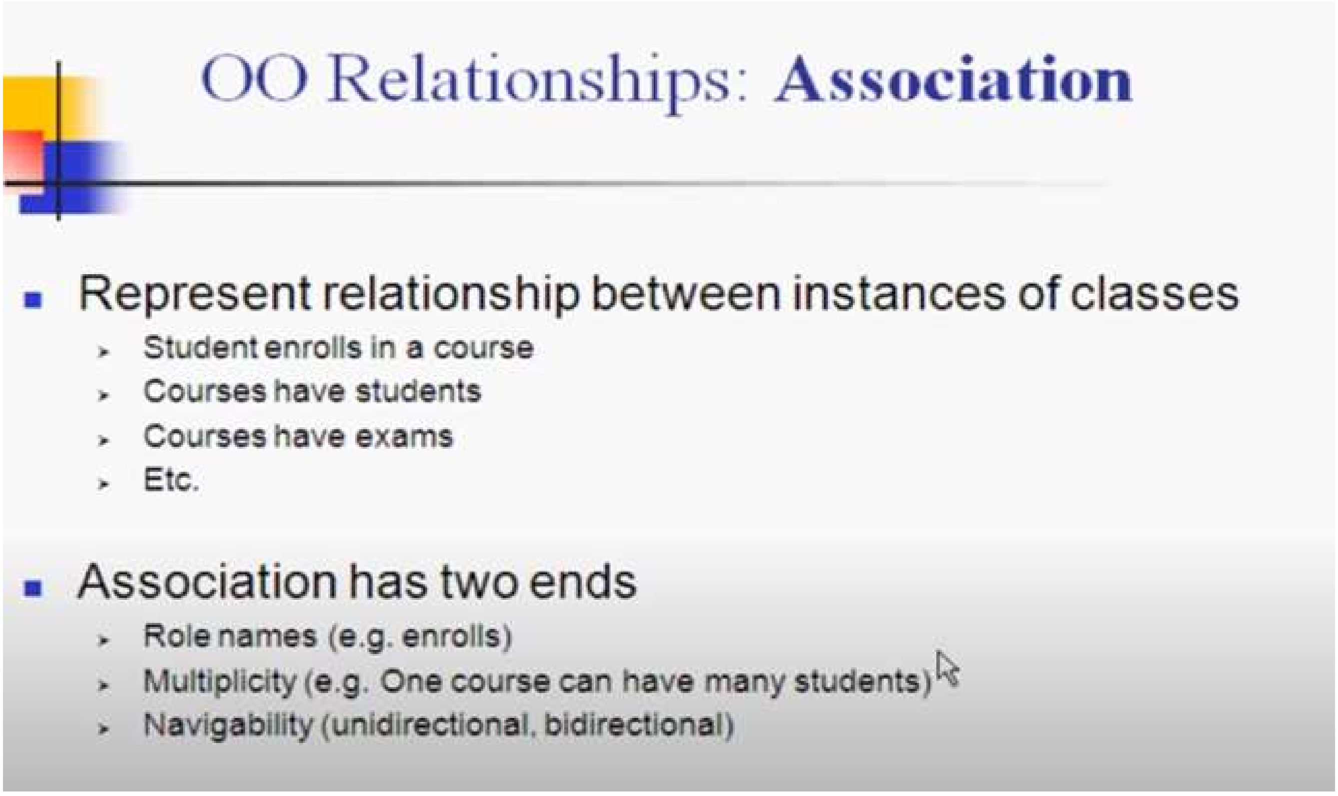
OO Relationships: **Generalization**

-Inheritance is a required feature of object orientation

-Generalization expresses a parent/child relationship among related classes.

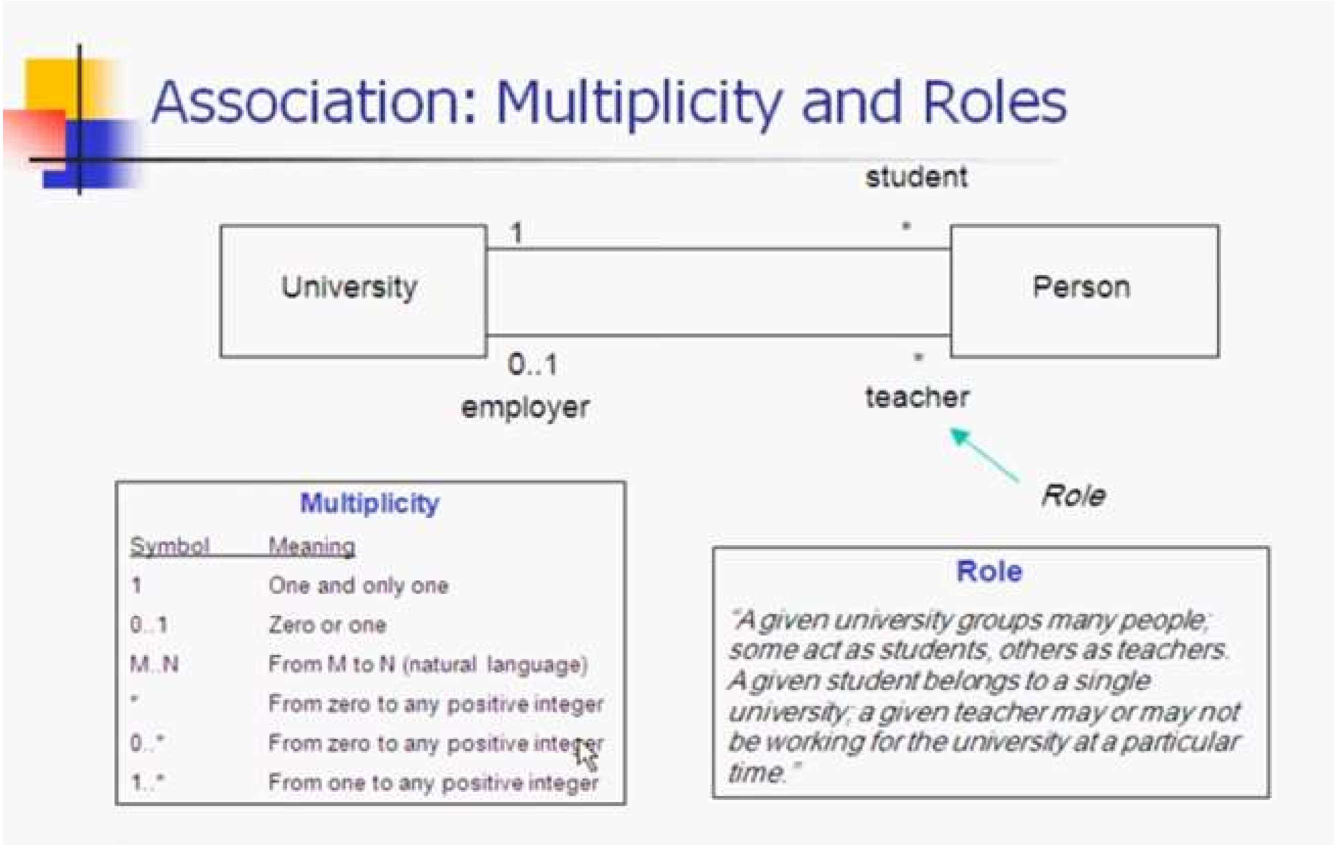
-Used for abstracting details in several layers





OO Relationships: **Association**

- Represent relationship between instances of classes
  - Student enrolls in a course
  - Courses have students
  - Courses have exams
  - Etc.
- Association has two ends
  - Role names (e.g. enrolls)
  - Multiplicity (e.g. One course can have many students)
  - Navigability (unidirectional, bidirectional)



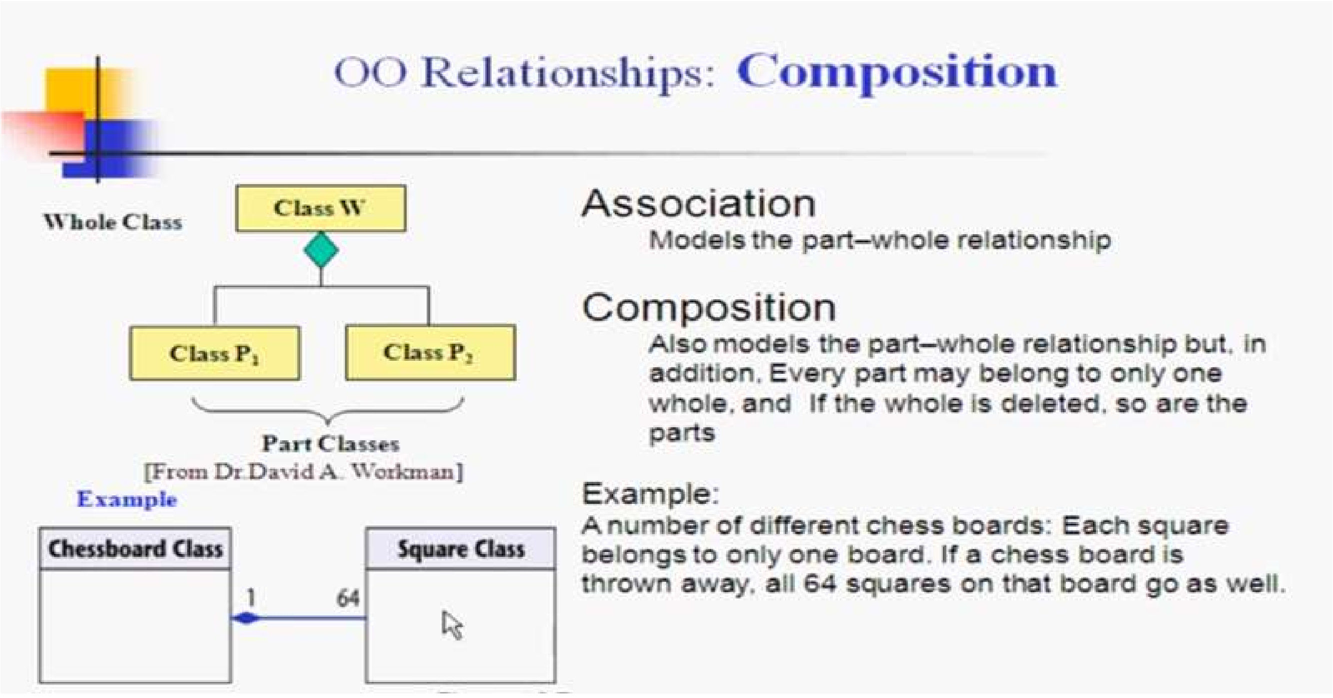
Association: Multiplicity and Roles

Multiplicity

Symbol	Meaning
1	One and only one
0..1	Zero or one
M..N	From M to N (natural language)
*	From zero to any positive integer
0..*	From zero to any positive integer
1..*	From one to any positive integer

Role

"A given university groups many people; some act as students, others as teachers. A given student belongs to a single university; a given teacher may or may not be working for the university at a particular time."



OO Relationships: Composition

Association

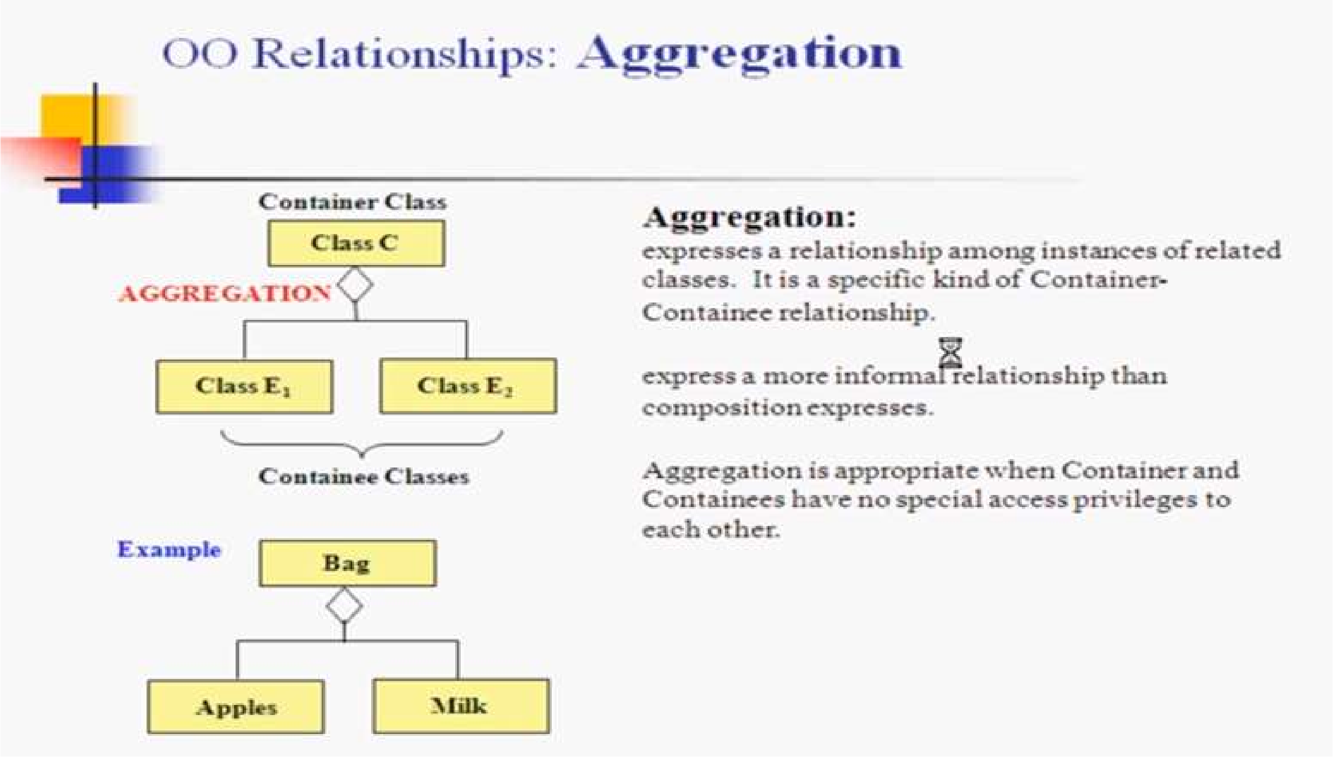
Models the part–whole relationship

Composition

Also models the part–whole relationship but, in addition, Every part may belong to only one whole, and If the whole is deleted, so are the parts

Example:

A number of different chess boards: Each square belongs to only one board. If a chess board is thrown away, all 64 squares on that board go as well.



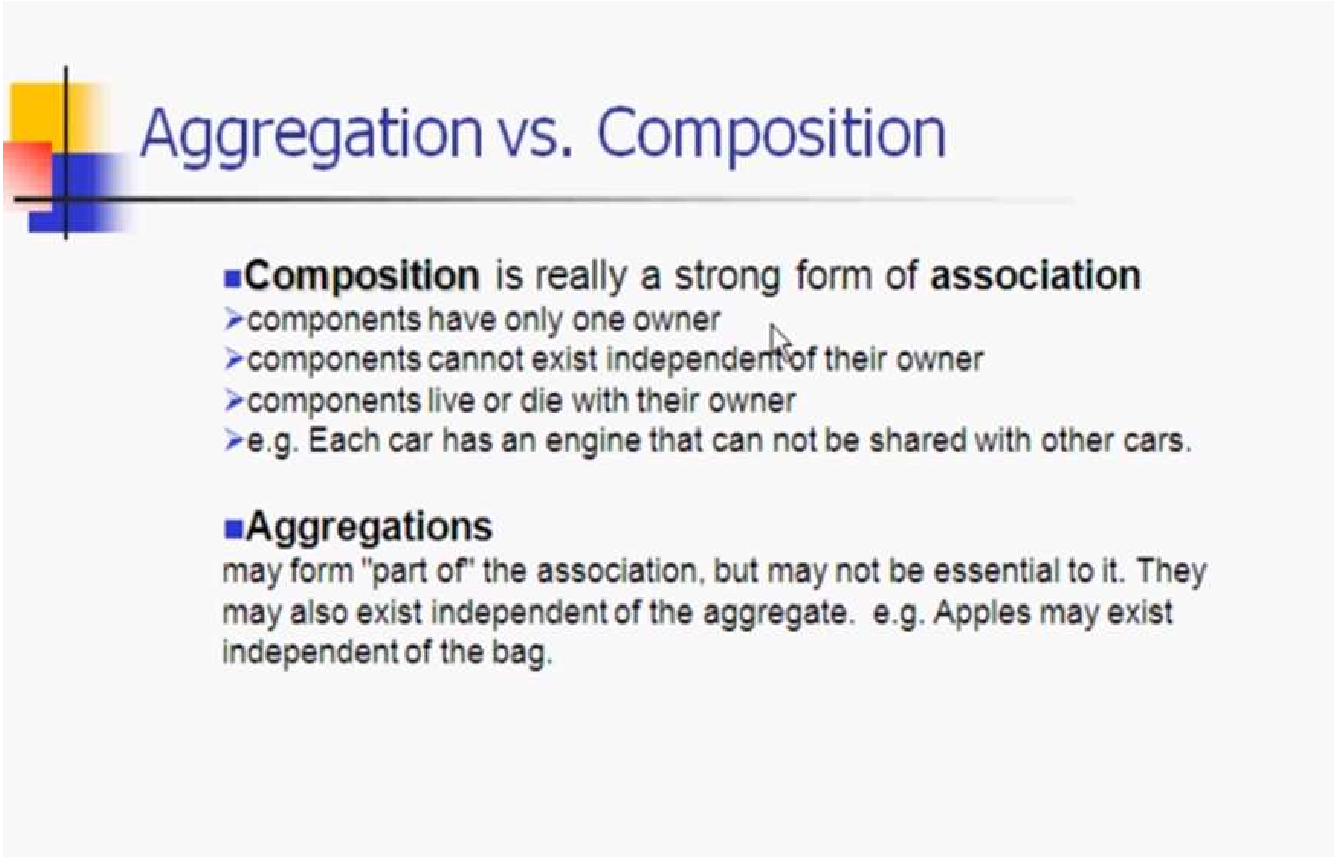
OO Relationships: Aggregation

Aggregation:

expresses a relationship among instances of related classes. It is a specific kind of Container-Containee relationship.

express a more informal relationship than composition expresses.

Aggregation is appropriate when Container and Containees have no special access privileges to each other.



Aggregation vs. Composition

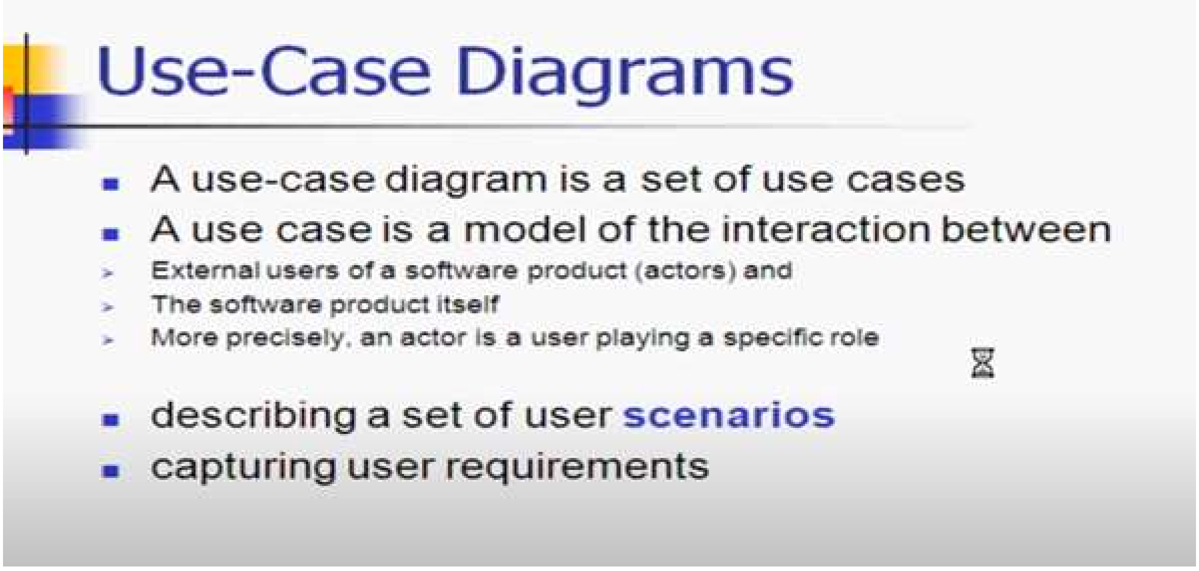
- **Composition** is really a strong form of **association**
  - components have only one owner
  - components cannot exist independent of their owner
  - components live or die with their owner
  - e.g. Each car has an engine that can not be shared with other cars.
- **Aggregations**

may form "part of" the association, but may not be essential to it. They may also exist independent of the aggregate. e.g. Apples may exist independent of the bag.

- object diagrams,
- component diagrams,
- deployment diagrams.

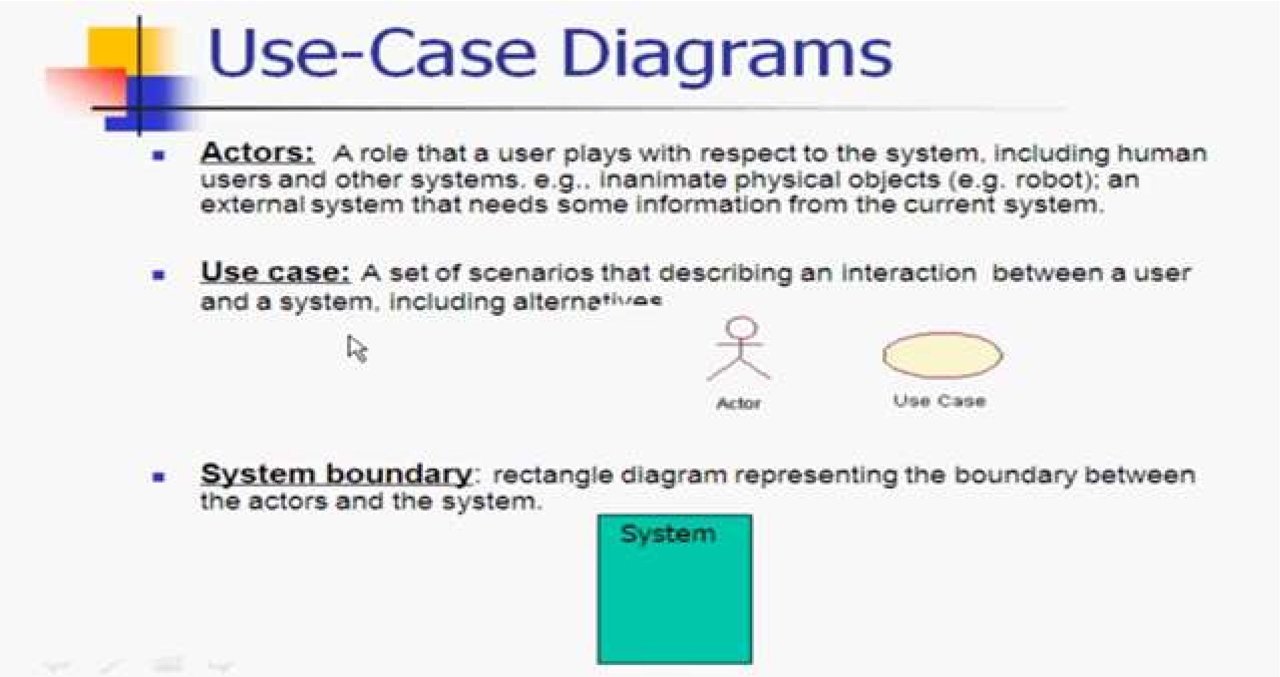
> Behavioral diagrams, which describe the dynamic relationships between components.

- use case diagrams,



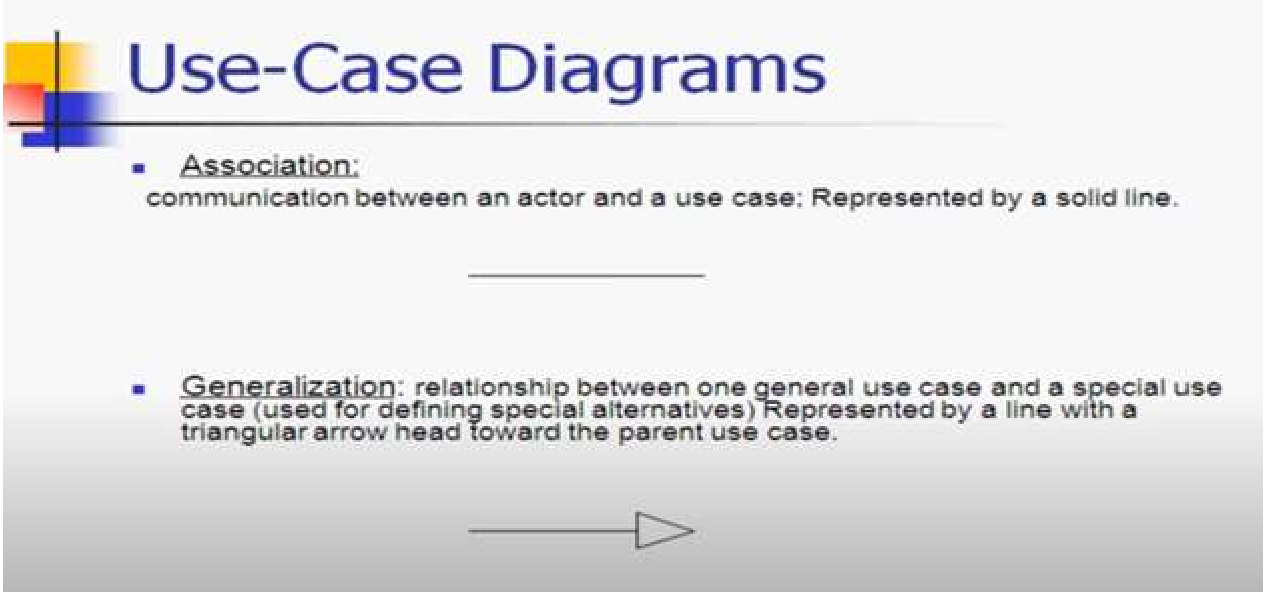
Use-Case Diagrams

- A use-case diagram is a set of use cases
- A use case is a model of the interaction between
  - External users of a software product (actors) and
  - The software product itself
  - More precisely, an actor is a user playing a specific role
- describing a set of user scenarios
- capturing user requirements



Use-Case Diagrams

- **Actors:** A role that a user plays with respect to the system, including human users and other systems. e.g., inanimate physical objects (e.g. robot); an external system that needs some information from the current system.
- **Use case:** A set of scenarios that describing an interaction between a user and a system, including alternatives
- **System boundary**: rectangle diagram representing the boundary between the actors and the system.

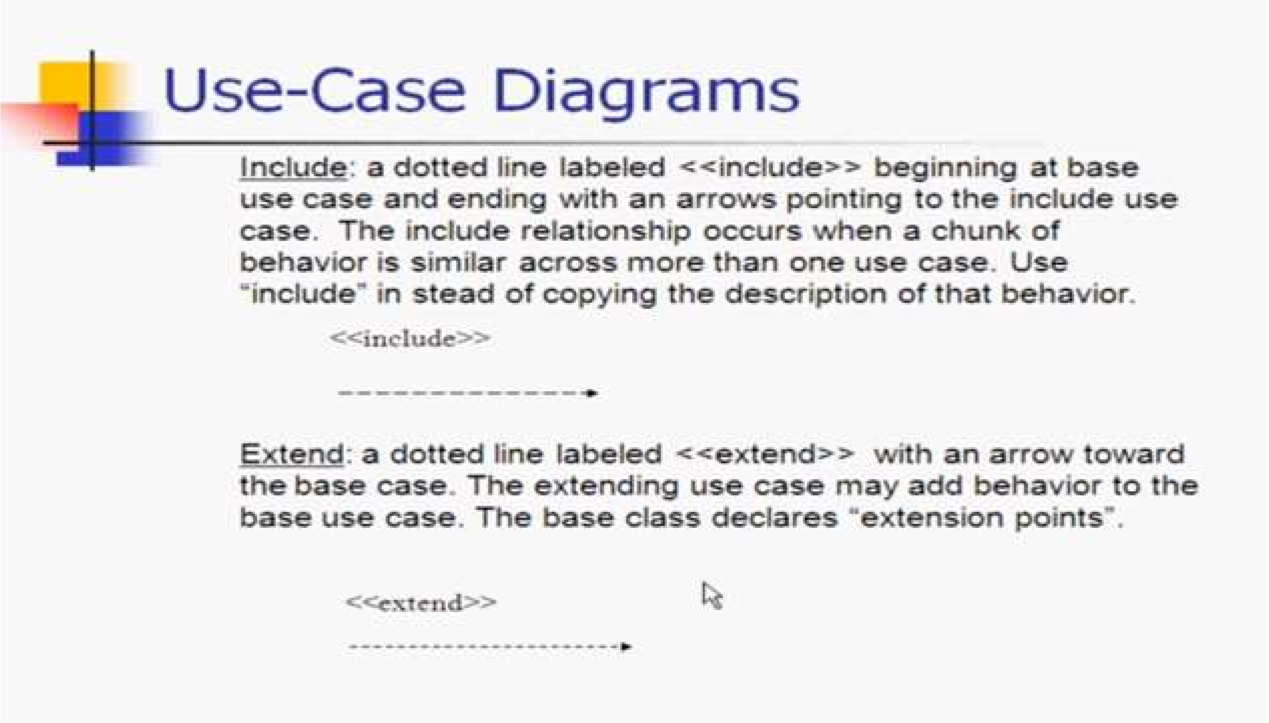


Use-Case Diagrams

- Association:

communication between an actor and a use case; Represented by a solid line.

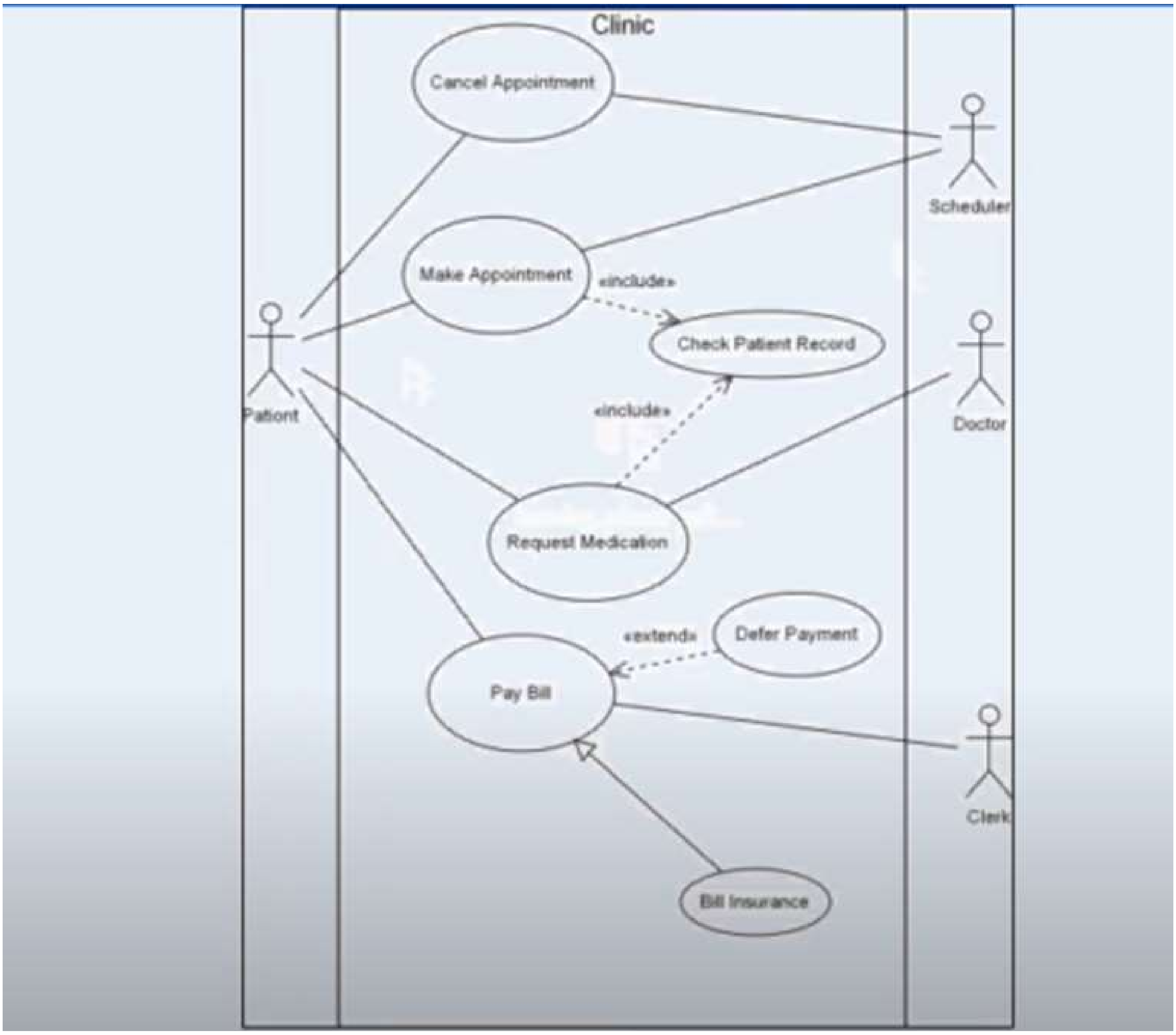
- Generalization: relationship between one general use case and a special use case (used for defining special alternatives) Represented by a line with a triangular arrow head toward the parent use case.



Use-Case Diagrams

Include: a dotted line labeled <<include>> beginning at base use case and ending with an arrows pointing to the include use case. The include relationship occurs when a chunk of behavior is similar across more than one use case. Use "include" in stead of copying the description of that behavior.

Extend: a dotted line labeled <<extend>> with an arrow toward the base case. The extending use case may add behavior to the base use case. The base class declares "extension points".



- sequence diagrams,
- collaboration diagrams,
- statechart diagrams,
- activity diagrams.

In [ ]: