

Data modelling and OO/ER data models

- Data Modelling
- Data Modelling
- Data Models
- Object-based vs. Record-based
- Database Design
- Some Design Ideas
- Case Study: Banking System
- Correctness of Designs
- Object-Oriented Data Modelling
- Object-oriented Data Modelling
- Object Definition Language (ODL)
- ODL Definitions
- Classes
- Attributes
- ODL Types
- Relationships
- Relationship Multiplicity
- ODL Model for Bank
- Entity-Relationship (ER) Model
- Entity-Relationship Data Modelling
- Entity-Relationship (ER) Diagrams
- Attributes

- Entity Sets
 - Keys
 - Relationship Sets
 - Weak Entity Sets
 - Subclasses and Inheritance
 - Design Using the ER Model
 - ER Model for Bank
 - Limitations of ER Model
 - Summary
-

§ Data Modelling

Data Modelling

Data modelling: an important early stage of database application development (aka "database engineering").

1. requirements analysis (identify data and operations)
2. **data modelling** (high-level, abstract)
3. database schema design (detailed, relational/tables)

4. database physical design (based on expected workload)
 5. database implementation (create instance of schema)
 6. build operations/interface (SQL, stored procedures, GUI)
 7. performance tuning (physical re-design)
 8. schema evolution (logical schema re-design)
-

Data Modelling (cont)

Aim of data modelling:

- describe what **information** is contained in the database
(e.g. entities: students, courses, accounts, branches, patients, ...)
- describe **relationships** between data items
(e.g. John is enrolled in COMP3311, Paul's account is held at Coogee)
- describe **constraints** on data
(e.g. 7-digit IDs, students can enrol in no more than 36UC per semester)

Data modelling is analogous to the design phase of software engineering, but deals only with data structures.

Data Modelling (cont)

Inputs to data modelling:

- enterprise to be modelled, user requirements

Outputs from data modelling:

- (semi) formal description of the database structure

Many languages/methodologies have been developed to assist, e.g.

- entity–relationship (ER) diagrams
 - ODL = object design language
 - UML = unified modelling language
-

Data Models

Data models are either:

- **logical models** ... deal with conceptual modelling of information
- **physical models** ... deal with physical layout of data in storage

Two main groups of logical data models:

- **object-based models**
 - e.g. entity-relationship, object-oriented, semantic, ...
 - **record based models**
 - e.g. relational, network, hierarchical, ...
-

Object-based vs. Record-based

Object-based data models

- treat database as a collection of entities of various kinds
- provide very flexible/natural data structuring facilities

- may also allow description of code for actions on objects

Record-based data models

- treat database as a collection of fixed-size records
 - less flexible data structures than with object-based models
 - closer to physical level so easier to implement efficiently
-

Object-based vs. Record-based (cont)

Formal mappings exist between the different kinds of models.

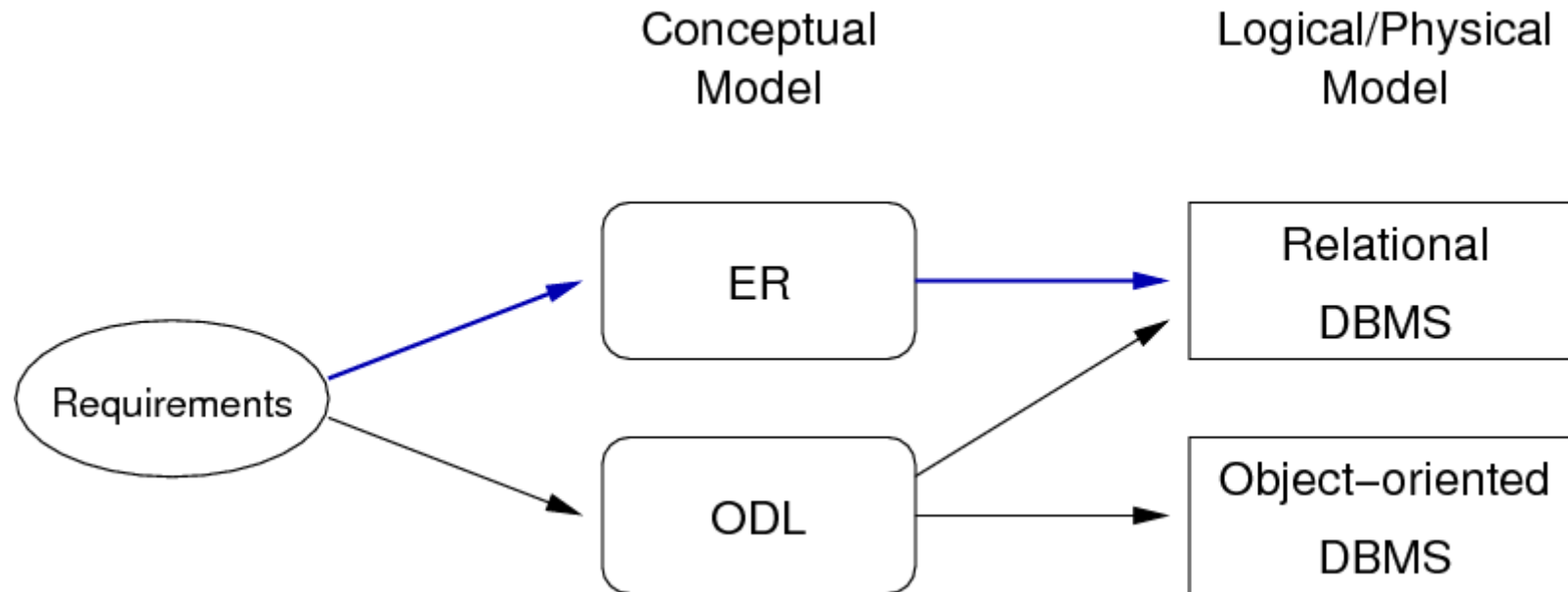
A useful strategy:

- design in an object-based model (for clarity)
- then convert to a record-based model (for efficiency)

We adopt this (very common) strategy in this course.

Database Design

The initial elements of the database design process:



Some Design Ideas

Consider the following while doing data modelling:

- ought to establish all requirements before designing
- designs evolve as we consider more aspects of the problem

- most designs involve various kinds (classes) of people
 - keywords in requirements give clues for data/relationships
(simplistic rule-of-thumb: nouns suggest data, verbs suggest relationships)
 - don't confuse operations with relationships
(operation: he **buys** a book; relationship: the book **is owned** by him)
 - consider all potential scenarios, not just actual data (if given)
-

Case Study: Banking System

Consider the database system for a savings/loan bank ...

What kind of information does it need to maintain?

- e.g. customers, accounts, branches, staff, transactions, ...

How are all of these data items related?

- e.g. customers hold accounts at specific branches, ...

We illustrate data models by:

- describing, in abstract terms, the capabilities they offer
 - showing how they could be used to model this application
-

Correctness of Designs

In general, there is no single "best" design for a given application.

Also, limitations of modelling framework may force compromises.

However, we may describe a proposed design as "inadequate" if it:

- omits some information that is supposed to be modelled
(e.g. no customer name or account balance)
- contains redundant information (but may be unavoidable)
(e.g. customer name stored in both **Customer** and **Account**)
- leads to an inefficient implementation
(e.g. required to traverse large amount of data for most common operation)

- violates syntactic or semantic rules of data model
(e.g. in ER, line connecting two entities)
-

Correctness of Designs (cont)

Initial requirements statements are typically vague

- elicit more detailed requirements before design
- ask for clarification during design

Most important aspects of design documents:

- completeness (all assumptions are explicit)
 - consistency (no contradictory statements)
-

§ Object–Oriented Data Modelling

Object–oriented Data Modelling

The world is viewed as a collection of inter-related **objects**.

Examples: a person, a car, a bank account, ...

Each object is described by:

- a unique object identifier (OID) (to distinguish it from other objects)
 - a collection of attributes that model "properties of interest"
 - operations (**methods**) that can be performed on the object
-

Object-oriented Data Modelling (cont)

A **class** is a set of objects with similar properties and methods.

Examples: all students (at UNSW), all cars (in Australia), ...

All student objects have the same data and operations,

e.g. **student.name**, **student.graduates(2003)**

An alternative view of classes

- a template for generating objects of a certain kind
-

Object Definition Language (ODL)

ODL is a proposed standard for object-oriented data modelling:

- developed by Object Data Management Group (ODMG)
- an extension of the Interface Definition Language (IDL)
- IDL defined in CORBA (model for distributed object systems)

For object-oriented database management systems:

- ODL is the proposed standard data definition language
 - OQL (Object Query Language) is the proposed standard query language
-

ODL Definitions

An ODL definition consists of a collection of classes.

ODL describes classes using three kinds of properties:

- **attributes** – data values associated with objects
- **relationships** – connections between objects
- **methods** – actions on objects

Since we are concerned with *data* modelling here, we ignore methods.

Classes

Class declarations have the following syntax:

```
interface ClassName : ParentClass { List of Properties }
```

Example:

```
interface Person {  
    attribute string  name;  
    attribute integer height;  
}
```

```
    attribute Enum Gender gender;  
    attribute Struct Date birthday;  
};  
  
interface Employee : Person {  
    attribute float salary;  
    relationship Company WorksFor  
        inverse Company::Employs;  
};
```

Attributes

Attributes describe the data values associated with an object.

Each attribute is defined by a **type** and a **name**.

ODL defines a variety of atomic, user-defined, and structured data types.

Attributes can have any type **except** types involving class types.

Names have global scope, and are used in context of object (e.g. **p.name**).

Attributes are inherited by sub-classes.

ODL Types

Atomic built-in data types:

- integer, float, character, string, boolean

Atomic user-defined data types:

- enumerated types (cf. C's **enum**)

Examples:

```
Enum Gender    {male, female}
Enum AcctType  {savings, cheque, credit}
Enum Colour    {red, orange, yellow, green, blue}
```

ODL Types (cont)

Structured data types are built using a set of type constructors:

Set $\langle T \rangle$ unordered collections of distinct elements of type T

Bag $\langle T \rangle$ unordered collections of elements of type T

List $\langle T \rangle$ sequences of elements of type T

Array $\langle T, i \rangle$ fixed-length vectors of i elements of type T

The base types T 's can be arbitrary types, including class types.

A structured type for an attribute may not contain any class types.

These structured types are ODL's **collection types** (important for relationships).

ODL Types (cont)

Differences between Sets, Bags, Lists:

$\{1,2,3\}=\{1,2,3\}$ $\{1,2,3\}=\{3,2,1\}$ $\{1,1,2\}=\{1,2,1\}$ $\{1,1,2\}=\{1,2\}$

Set	Yes	Yes	Yes	Yes
Bag	Yes	Yes	Yes	No
List	Yes	No	No	No

Arrays and Lists are similar (e.g. could view Array as fixed-size List)

ODL Types (cont)

Examples of structured type definitions:

```
// Can be used as types for attributes
```

```
List<real>
```

```
Array<string, 20>
```

```
Set<integer>
```

```
Bag<Enum Colour>
```

```
List<Array<Set<integer>, 10>>>
```

```
// Cannot be used as types for attributes
```

```
List<People>
```

```
Set<Company>
```

```
List<Set<People>>>
```

ODL Types (cont)

One other structured data type builds tuples (cf. C's **struct**).

```
Struct Name {  
    Type1 FieldName1,  
    Type2 FieldName2,  
    ...  
    Typen FieldNamen,  
}
```

Examples:

```
Struct Date {integer day, integer month, integer year}  
Struct Name {string family, string given}  
Struct Address {integer number, string street, ...}
```

Relationships

Relationships describe connections between objects (classes).

A relationship is defined in a class C by:

- naming the class D to which it is related
- describing the kind of relationship between C and D (details to follow)
- giving a name to the relationship
- specifying the inverse relationship (defined in D)

Syntax for relationship declarations:

```
relationship Collection(D) RelName inverse D::InverseRelName
```

Relationships (cont)

Example:

- employees work in a company; may work on many projects
- each project requires several employees to get it completed

```
interface Employee { ...
    relationship Company WorksFor;
    relationship Set<Project> WorksOn;
};
interface Company { ...
    relationship Set<Employee> Employs;
};
interface Project { ...
    relationship Set<Employee> Involves;
};
```

Relationships (cont)

Example (with inverses included):

```
interface Employee { ...
    relationship Company WorksFor
        inverse Company::Employs;
    relationship Set<Project> WorksOn
        inverse Project::Involves;
};

interface Company { ...
    relationship Set<Employee> Employs
        inverse Employee::WorksFor;
};

interface Project { ...
    relationship Set<Employee> Involves
        inverse Employee::WorksOn;
};
```

Relationships (cont)

Example (family trees):

```
interface Person {
    attribute string name;
    attribute Date    birthday;
```

```
...
relationship Person FatherIs
    inverse Person::FatherOf;
relationship Set<Person> FatherOf
    inverse Person::FatherIs;
...
relationship Set<Person> Children
    inverse Person::BiologicalParent;
relationship Array<Person,2> BiologicalParents
    inverse Person::Children;
...
relationship Set<Person> Raised
    inverse Person::RaisedBy;
relationship Set<Person> RaisedBy
    inverse Person::Raised;
...
relationship Person BestFriend
    inverse Person::BestFriend;
};
```

Notes:

- an object in class *C* can be related other objects in class *C*

- relationships have different **multiplicities**
-

Relationship Multiplicity

An important property of relationships:

- how many objects is a given object *Obj* related to under relationship *R* ?

Possibilities:

- related to exactly one other (e.g. **FatherIs**)
- related to exactly *n* other (e.g. **BiologicalParents**)
- related to many others (e.g. **Children**)

Similarly for *R*'s inverse relationship R^{-1} .

Relationship Multiplicity (cont)

Consider classes C , D , and relationships R , R^{-1} between them.

The most common **relationship multiplicities**:

- many-to-many** $C \rightarrow R \rightarrow D$ is a set of objects
 $D \rightarrow R^{-1} \rightarrow C$ is a set of objects
- many-to-one** $C \rightarrow R \rightarrow D$ is a single object
 $D \rightarrow R^{-1} \rightarrow C$ is a set of objects
- one-to-one** $C \rightarrow R \rightarrow D$ is a single object
 $D \rightarrow R^{-1} \rightarrow C$ is a single object
-

ODL Model for Bank

Kinds of objects:

- name, address, date, ...
- customer, account, branch, transaction, ...

Some we can model via user-defined types:

```
Struct Name { string family, string given };
Struct Address {
    integer number;
    string street;
    string suburb;
    integer postcode;
};
Enum Activity { deposit, withdrawal };
Enum AcctType { savings, cheque, credit };
```

ODL Model for Bank (cont)

Modelling people:

```
interface Person {
    attribute Struct Name name;
    attribute Struct Address address;
};
```

```
interface Employee : Person {
```

```
attribute int employeeNo;  
attribute float salary;  
relationship Branch WorskAt  
    inverse Branch::Staff;  
};
```

```
interface Manager : Employee {  
    relationship Branch Manages  
        inverse Branch::ManagedBy;  
};
```

ODL Model for Bank (cont)

Modelling branches:

```
interface Branch {  
    attribute integer branchNo;  
    attribute string name;  
    attribute Struct Address address;  
    attribute float assets;  
    ...  
    relationship Set<Customer> Customers
```

```
    inverse Customer::HomeBranch;  
    relationship Set<Account> Accounts  
    inverse Account::HeldAt;  
};
```

ODL Model for Bank (cont)

Modelling accounts:

```
interface Account {  
    attribute integer accountNo;  
    attribute float balance;  
    ...  
    relationship Branch HeldAt  
        inverse Branch:Accounts;  
    relationship Set<Customer> Owners  
        inverse Customers::Accounts;  
    relationship List<Transaction> Transactions  
        inverse Transaction::TransAcct;  
};
```

ODL Model for Bank (cont)

Modelling transactions:

```
interface Transaction {  
    attribute Enum Activity type;  
    attribute Date when;  
    attribute float amount;  
    ...  
    relationship Account TransAcct  
        inverse Account::Transactions;  
    relationship Customer MadeBy  
        inverse Customer::Transactions;  
};
```

ODL Model for Bank (cont)

Modelling people (cont):

```
interface Customer : Person {  
    attribute int customerNo;
```

```
attribute Date memberFrom;  
...  
relationship Branch HomeBranch  
    inverse Branch::Customers;  
relationship Set<Account> Holds  
    inverse Account::HeldBy;  
relationship List<Transaction> Transactions  
    inverse Transaction::MadeBy;  
};
```

§ Entity–Relationship (ER) Model

Entity–Relationship Data Modelling

The world is viewed as a collection of **inter–related entities**.

Goal of ER modelling:

- a notation for describing entities and their relationships

- abstract/graphical \Rightarrow used between developers/clients

Examples of entities: John Shepherd, his car, K17, ...

Example of relation: John Shepherd **owns** his car

Entity–Relationship Data Modelling (cont)

Each entity is described by:

- a collection of attributes that model "properties of interest"

Examples:

- Person = (Name, TaxFileNum, DateOfBirth, Citizenship)
e.g. (John, 2233456789, 12–Feb–1978, Australian)
- Car = (Make, Model, Year, Colour, Registration)
e.g. (Ford, Laser, 1990, Red, JAS–007)

An **entity set** is a collection of entities with similar properties.

Entity–Relationship Data Modelling (cont)

Analogy between ER and OO models:

- an **entity** is like an **object**
- an **entity set** is like a **class**

Differences between ER and OO models:

- entities don't have OIDs to distinguish them
- ER modelling doesn't consider operations (methods)

Entity–Relationship Data Modelling (cont)

The entity–relationship model has existed for almost 30 years.

(Original description: Chen, *ACM Transactions on Database Systems*, 1(1), 1976)

It was never standardised, but has been well–used ...

- so, unfortunately, many variations exist
(major variations involve relationship cardinalities and OO extensions)

In lectures, we use the notation from the KSS book.

Other texts (EN, GUW, RG) use slightly different notation.

Choose whichever **one** you like, and use it consistently.

Entity–Relationship (ER) Diagrams

ER diagrams are a graphical tool for data modelling.

An ER diagram consists of:

- a collection of **entity set** definitions
- a collection of **relationship set** definitions
- **attributes** associated with entity and relationship sets
- connections between entity and relationship sets

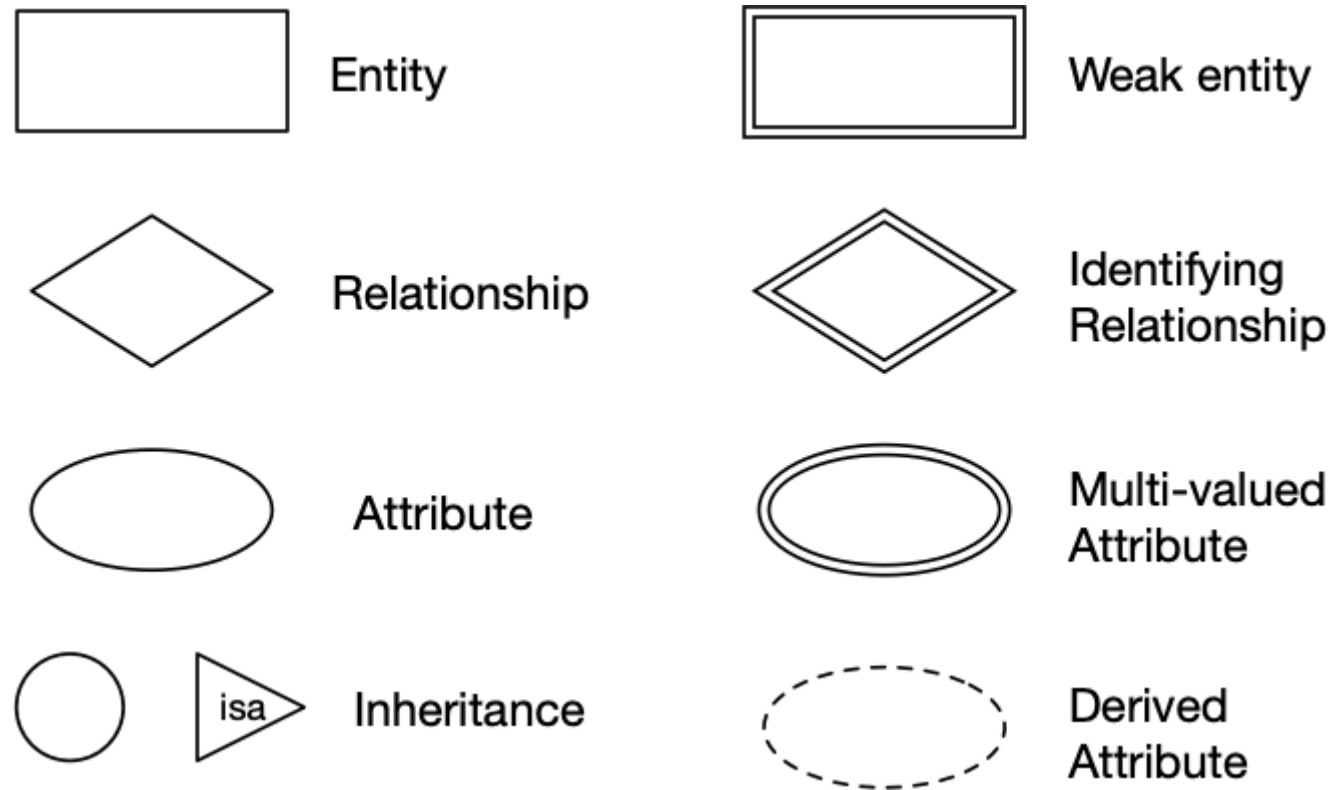
Warning: 99% of the time ...

- we say "entity" when we mean "entity set"
- we say "relationship" when we mean "relationship set"

If we want to refer to a specific entity, we generally say "entity instance".

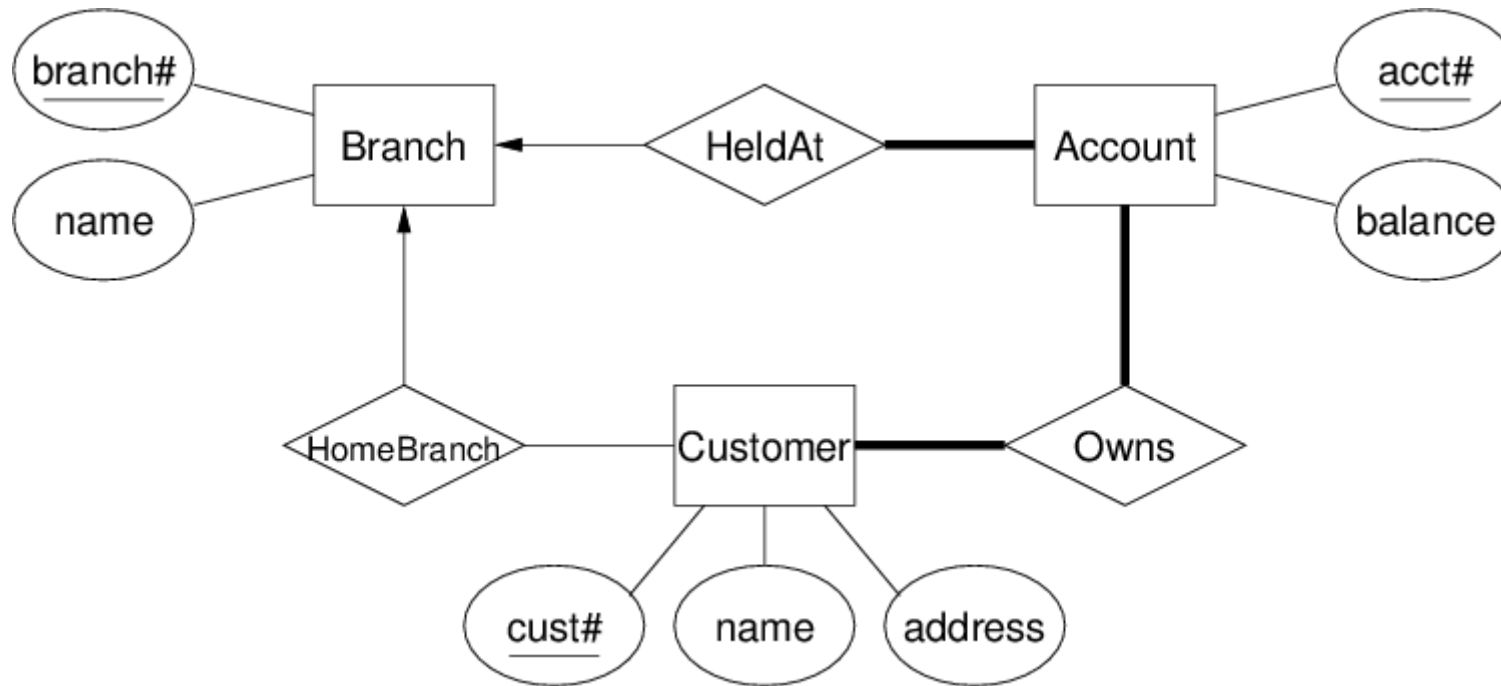
Entity–Relationship (ER) Diagrams (cont)

Specific visual symbols indicate different ER design elements:



Entity–Relationship (ER) Diagrams (cont)

Example ER diagram (details explained below):



Attributes

Each **attribute** in an ER diagram:

- has a **name** (which appears on the ER diagram)
- is associated with an entity or relationship set
- has an underlying **domain** (type)

Information about attribute domains

- is not presented on the ER diagram
 - supplied separately via some other formalism (e.g. SQL)
-

Attributes (cont)

Simple attributes are drawn from atomic value domains.

(cf. ODL's atomic data types)

Composite attributes consist of a hierarchy of attributes.

(cf. ODL's **Struct** types)

Single-valued attributes have one value for each entity.

(cf. ODL's atomic or **Struct** types)

Multivalued attributes have a set of values for each entity.

(cf. ODL's collection types)

Attributes (cont)

Null attributes may contain a distinguished **null** value to indicate:

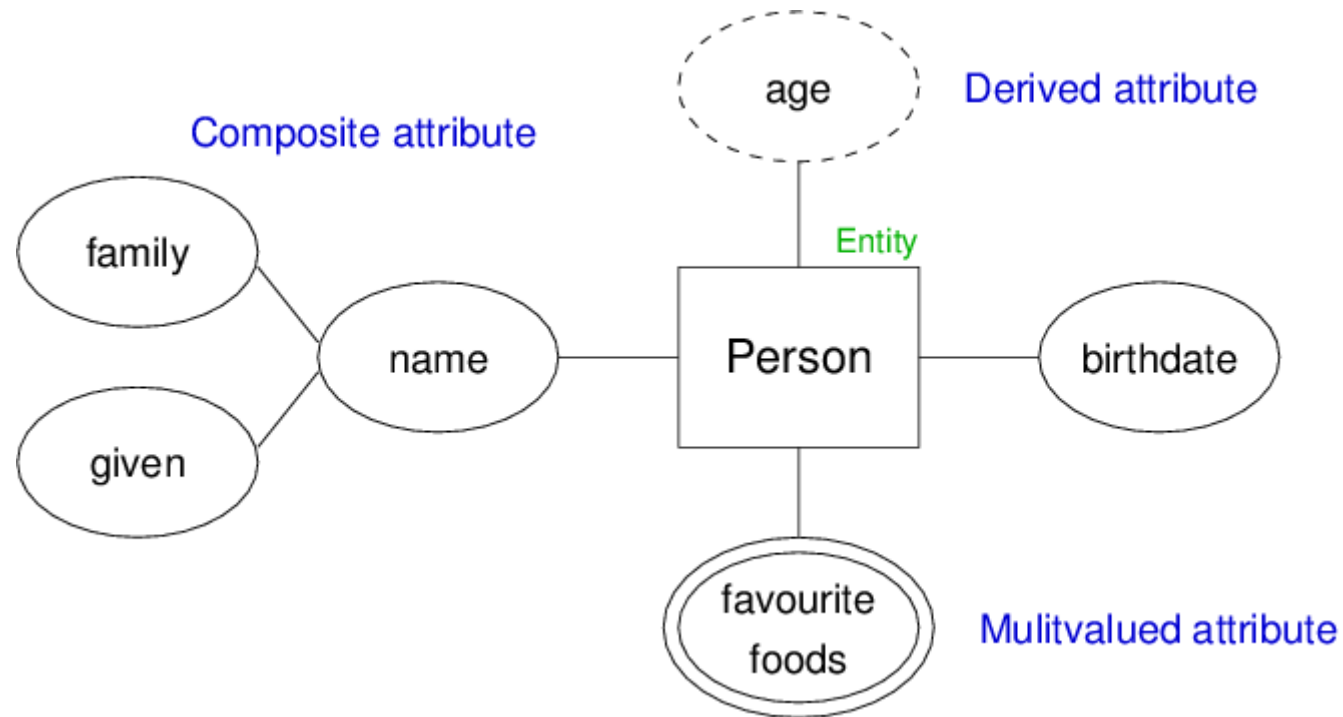
- the attribute is not relevant for a particular entity
(e.g. a customer under 18 would not have a credit card number)
- the value of the attribute is unknown for a particular entity
(e.g. don't know someone's address or phone number)

Derived attributes contain values that are calculated from other attributes.

(e.g. a person's age may be computed given their date of birth)

Attributes (cont)

Example of attribute notations:



Entity Sets

An **entity set** can be viewed as either:

- a set of entities with the same set of attributes
(**extensional view** of entity set)
- an abstract description of a class of entities
(**intensional view** of entity set)

Entity sets are not necessarily disjoint.

(e.g. a person may be both a **Customer** and **Employee** of a bank)

The "raw data" in a database may be viewed as a collection of entity sets.

Keys

ER model does not have a notion of OIDs, so how to distinguish entities?

What if two entities have the same set of attribute values?

They're regarded as the same entity.

So, each entity must have a distinct set of attribute values.

Implications:

- it is not possible to distinguish two separate real-world entities if their set of attribute values is identical

- since it is necessary to distinguish entities, we need to be careful about how we choose attributes
-

Keys (cont)

One approach to ensuring that attribute value sets are distinct:

- add a new attribute to each entity
- containing a unique value for each entity in the entity set

(like re-introducing OIDs, but they only need to be distinct within an entity set)

This approach is used commonly in practice e.g.

- social security number (SSN) (distinct for all residents of USA)
- tax file number (TFN) (distinct for all Australian taxpayers)
- UNSW student id (distinct for all students at UNSW)

This is a specific example of the more general notion of a **key**.

Keys (cont)

A **superkey** is

- a set of one or more attributes for an entity set
- whose values are distinct for all entities in that set

(i.e. a set of values for superkey attributes identifies at most one entity in the set)

The notion of superkey is a property of the whole entity set (extension).

During design, we typically consider keys relative to all possible extensions.

Keys (cont)

Example (bank customer entities):

`Customer = (custNo, name, address, taxFileNo)`

Definite superkeys:

- any set of attributes involving **custNo** or **taxFileNo**

Possible superkeys:

- **(name, address)**

Unlikely superkeys:

- **(name), (address)**
-

Keys (cont)

The entire set of attributes is always a superkey.

However, most entity sets have several superkeys.

E.g. **(custNo, name)**, **(custNo, address)**, ...

It would be convenient to have just one key to identify entities.

Can we identify a minimal set of attributes to be **the** key?

In examples with artificial identifiers (e.g. SSN), use identifier.

In other examples ... ?

Keys (cont)

A **candidate key** is any superkey such that

- no proper subset of its attributes is also a superkey

E.g. (**custNo**), (**taxFileNo**), (**name**, **address**), ...

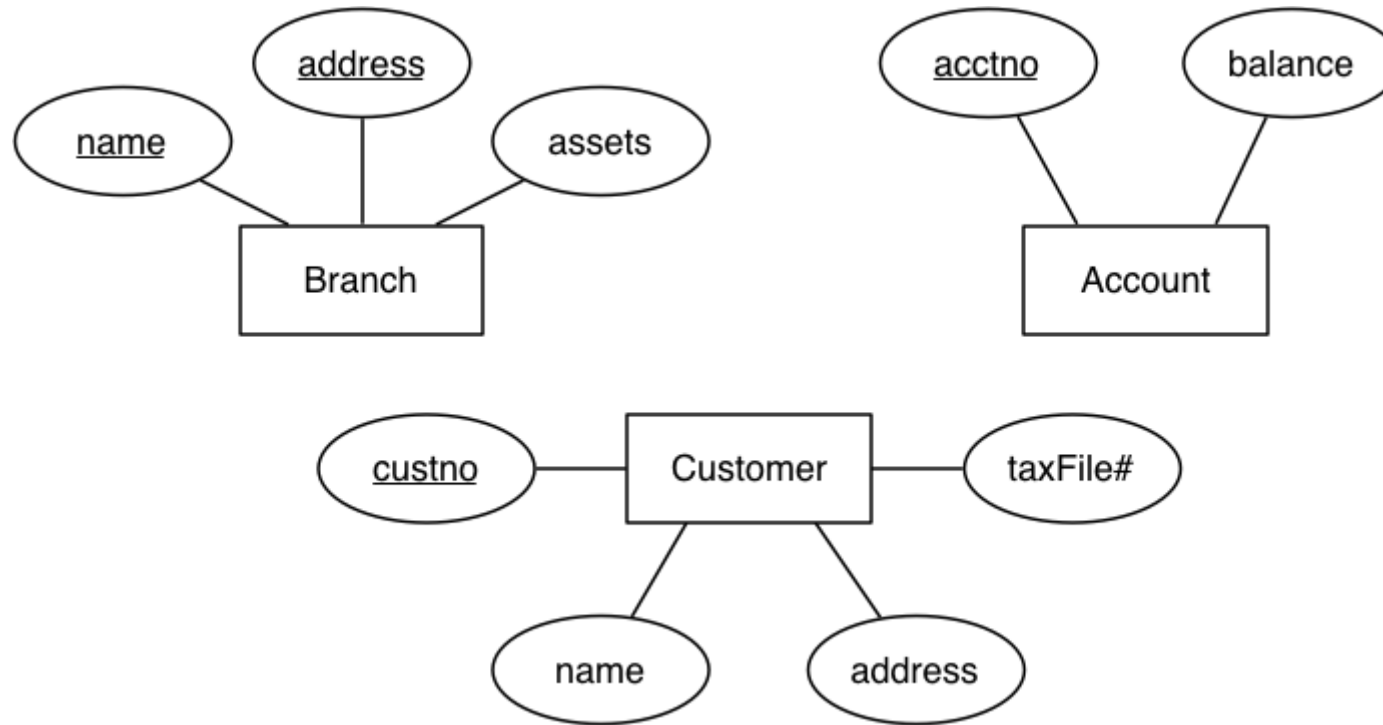
A **primary key**:

- is a candidate key chosen by the database designer
- to be used as the identifying mechanism for the entity set

A **composite key** is a key with two or more attributes.

Keys (cont)

Keys are indicated in ER diagrams by underlining the key attributes.



Relationship Sets

A **relationship** is an association among several entities.

E.g. Customer(9876) is the owner of Account(12345)

A relationship set is a collection of relationships of the same type.

E.g. the "is the owner of" relationship set

Customer(9876) is the owner of Account(12345)

Customer(9426) is the owner of Account(54321)

Customer(9511) is the owner of Account(88888)

Customer(9303) is the owner of Account(78787)

etc. etc.

Relationship Sets (cont)

Stated more formally:

- consider $n \geq 2$ entity sets E_1, E_2, \dots, E_n (possibly non-distinct)
- a relationship is a tuple $(e_1, e_2, \dots, e_n) \in E_1 \times E_2 \times \dots \times E_n$
- a relationship set R is a subset of $E_1 \times E_2 \times \dots \times E_n$

- entity sets E_1, E_2, \dots, E_n participate in R , R has degree n

ER relationships' degree ≥ 2

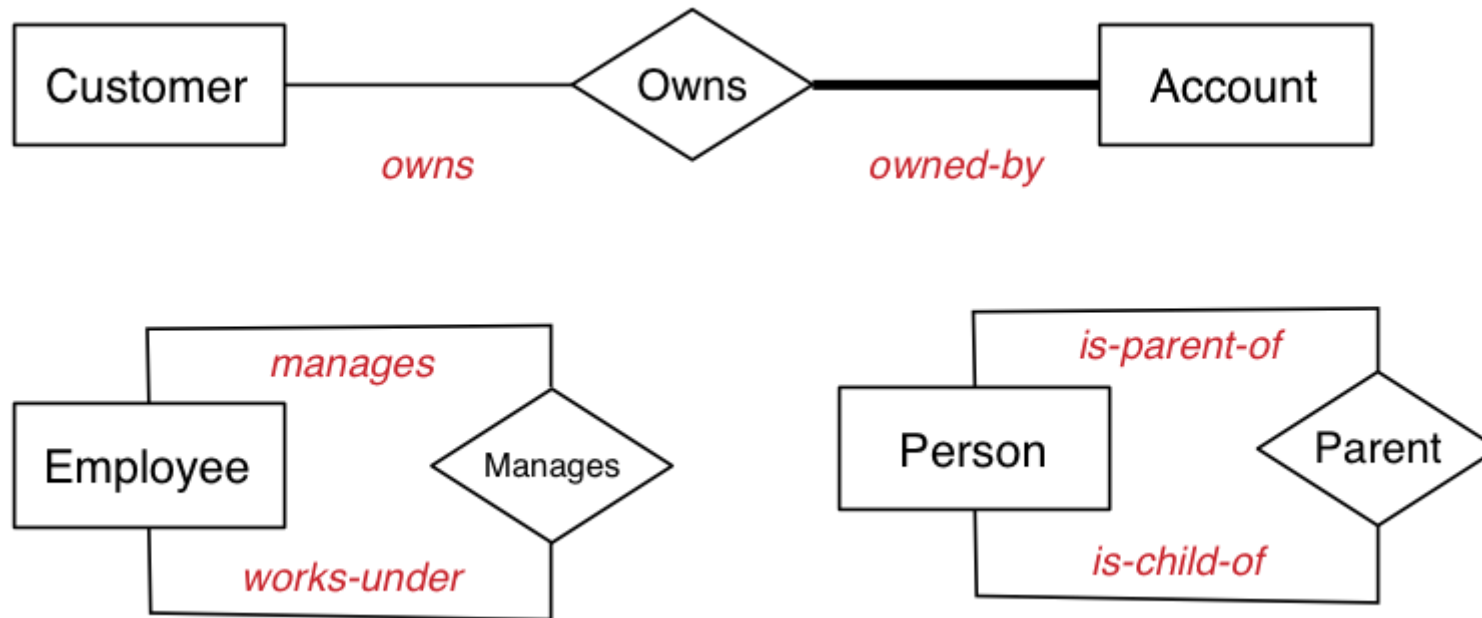
Relationship Sets (cont)

In an ER diagram, relationship sets are assigned names.

The role of each entity in the relationship is usually implicit.

Roles can be explicitly named if needed (in red below)

Examples:



Relationship Sets (cont)

Mapping cardinalities describe the number of entities that a given entity can be associated with via a relationship.

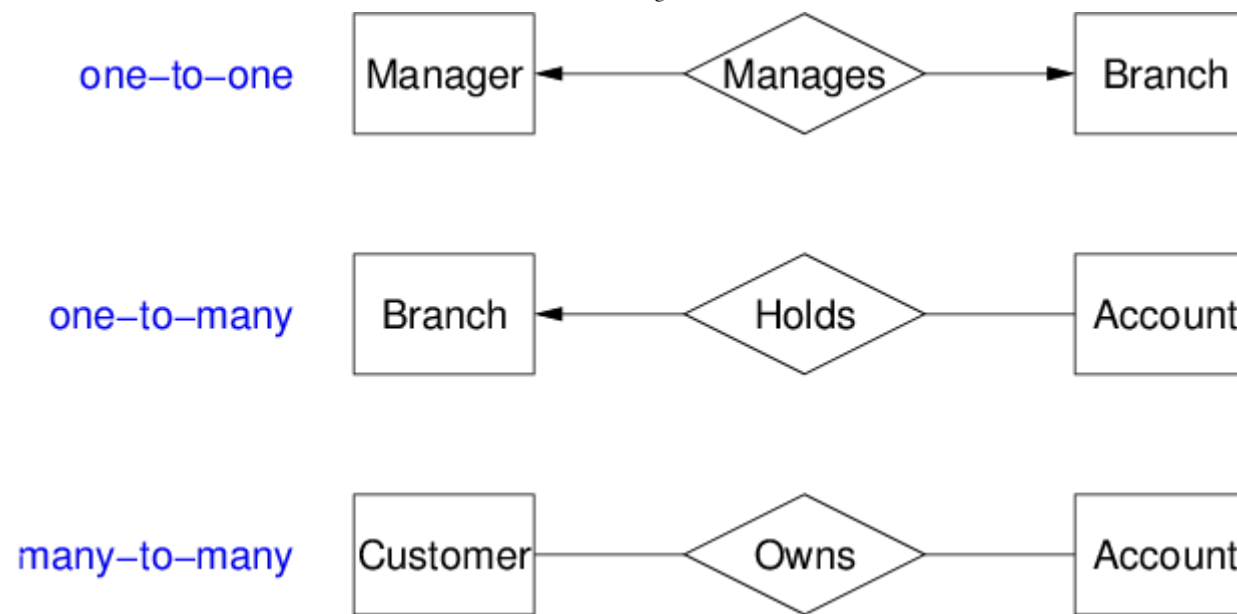
A binary relationship set R between entity sets A and B may be (assuming $a \in A$, $b \in B$):

one-to-one each a is associated with at most one b

	each b is associated with at most one a
one-to-many	each a is associated with zero or more b each b is associated with at most one a
many-to-one	each a is associated with at most one b each b is associated with zero or more a
many-to-many	each a is associated with zero or more b each b is associated with zero or more a

Relationship Sets (cont)

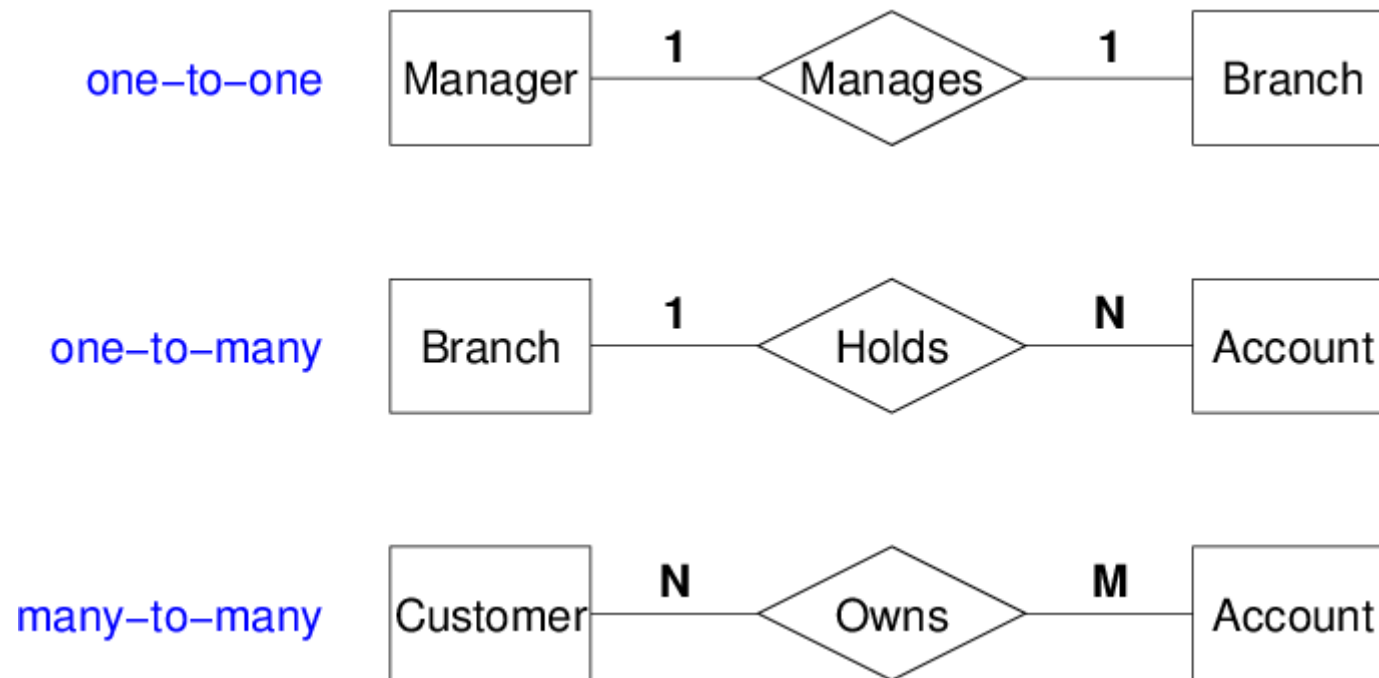
Examples:



An arrow from A to B indicates that there is at most one $b \in B$ for each $a \in A$.

Relationship Sets (cont)

An alternative notation makes cardinalities explicit:



Relationship Sets (cont)

Level of participation is another type of relationship constraint.

Participation in relationship set R by entity set A may be:

total every $a \in A$ participates in ≥ 1 relationship in R

partial only some $a \in A$ participate in relationships in R

Example:

- every bank loan is associated with at least one customer
 - not every customer in a bank has a loan
-

Relationship Sets (cont)

In ER diagrams:

- partial participation is denoted by single lines
- total participation is denoted by double lines (or **thick** lines)

Example:



Relationship Sets (cont)

In the x-to-one and one-to-x relationships above, we noted that e.g.

"Each $a \in A$ is associated with **at most one** $b \in B$ "

Sometimes, we require that there **must** be **exactly one** associated entity.

E.g. a manager must have a branch to manage, but a branch may (temporarily) have no manager.

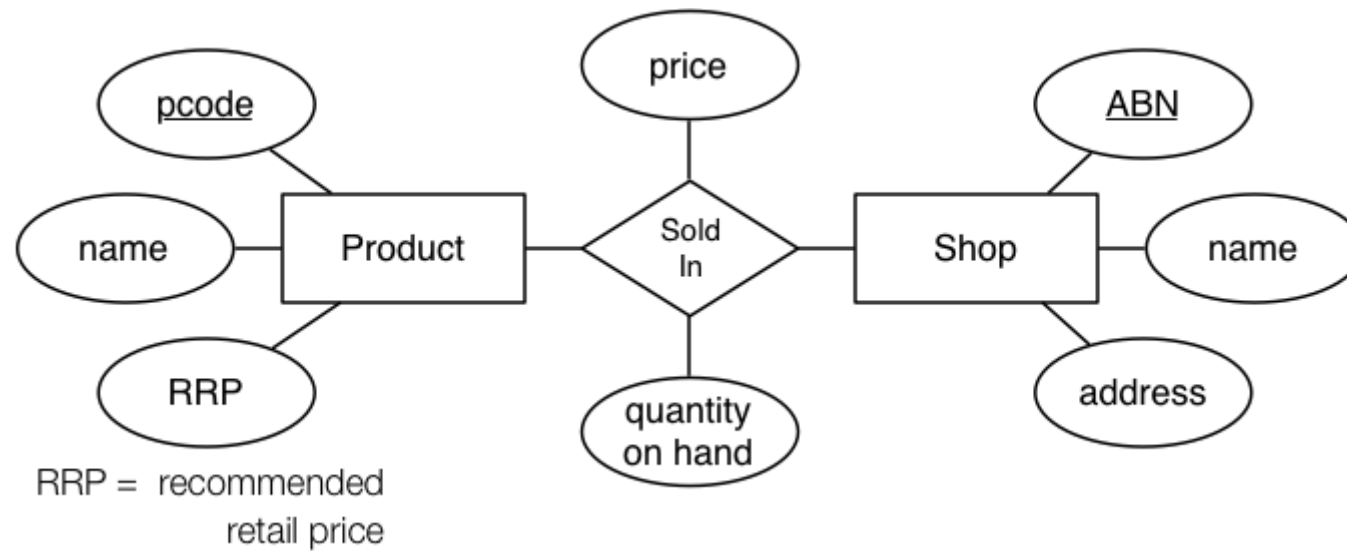


This is known as a **referential integrity constraint**.

Relationship Sets (cont)

In some cases, a relationship needs associated attributes.

Example:



(Price and quantity are related to products in a particular store)

Weak Entity Sets

Weak entities exist only because of association with other entities.

Examples:

- family of employees in a company
(would not be interested in the family once the employee leaves)

- payments on bank loans
(if there were no loans, no need to keep information about payments)

Weak entities

- do not have a primary key (or any superkey)
 - have a subset of attributes that form a **discriminator**
 - need to be considered in conjunction with strong entities
-

Weak Entity Sets (cont)

While weak entities do not have a primary key

- they have a subset of attributes that form a **discriminator**

We can form a primary key by taking a combination of

- the set of values for the discriminator
- the primary key of the associated strong entity

Example:

- Ian's son called Tim is different to Paul's son called Tim
-

Weak Entity Sets (cont)

E.g. employees/family, with name as a "key" ...

```
Employees = {Anne, John, Liam}
Families  = {Alice, Bob, David, Jane, Mary, Tim, Tim}
John's family = {Alice, Bob, Mary, Tim}
Anne's family = {David, Tim}
Liam's family = {Jane}
```

E.g. loans/payments, with loan ids and payment sequence ...

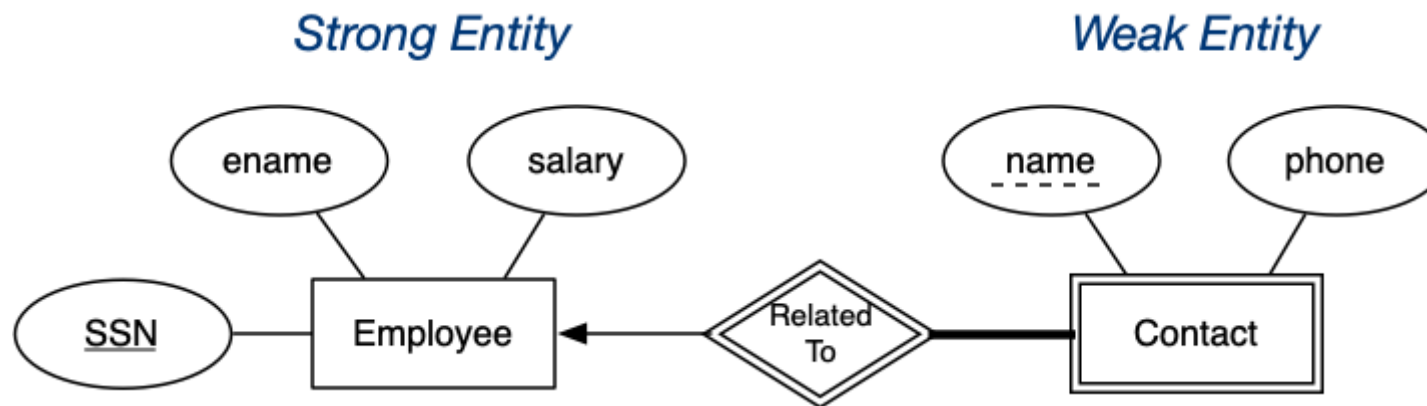
```
Loan          = {(1234, John, 100000), (4321, Arun, 70000)}
Payments      = {(1, 1500), (2, 1500), (3, 1500), (1, 750), (2, 1500)}
John's payments = {(1, 1500), (2, 1500), (3, 1500)}
Arun's payments = {(1, 750), (2, 1500)}
```

Weak Entity Sets (cont)

In ER diagrams:

- weak entities are denoted by double-boxes
- strong/weak entity relationships are denoted by double-diamonds
- discriminators are denoted by dotted underline

Example:



Subclasses and Inheritance

Extensions to the "standard" ER model include inheritance.

A **subclass** of an entity set A is a set of entities:

- with all of the attributes found in entities of A
- with (generally) additional attributes of its own
- that is involved in all of the relationships involving A
- may be involved in additional relationships on its own

In other words, the subclass **inherits** the attributes and relationships of A .

Some texts use the term **lower level entity set** as synonym for "subclass".

Subclasses and Inheritance (cont)

If an entity set has multiple subclasses, they may be:

- **disjoint** – an entity belongs to at most one subclass
- **overlapping** – an entity may belong to several subclasses

An orthogonal property is the **completeness constraint**:

- **total** – all entities must belong to at least one subclass
- **partial** – some entities may belong to no subclass

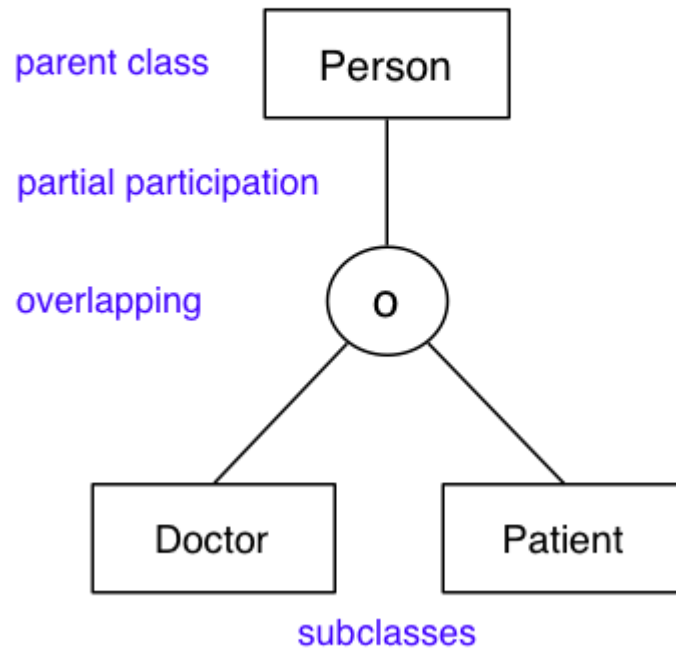
ER diagrams use the following notation:

- subclass denoted by **ISA** or subset symbol on line
- disjoint/overlapping = the letter '**d**' / '**o**' in a circle
- total/partial completeness = double(thick)/normal line

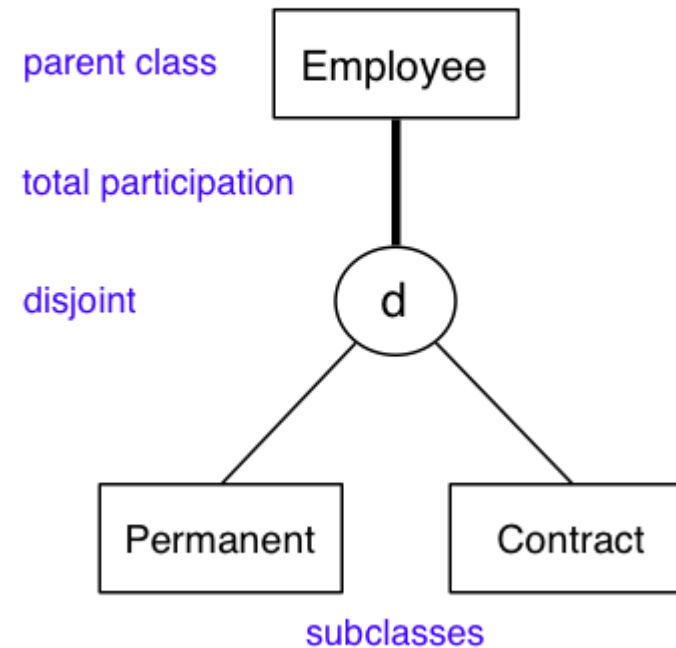
Subclasses and Inheritance (cont)

Example of subclasses:

*A person may be a doctor and/or
may be a patient or may be neither*



*Every employee is either a permanent
employee or works under a contract*



Design Using the ER Model

The ER model provides a powerful, general set of data modelling tools.


Some considerations in using these tools to create a design:

- should an "object" be represented by an attribute or entity?

- is a "concept" best expressed as an entity or relationship?
 - should we use n -way relationship or several 2-way relationships?
 - is an "object" a strong or weak entity?
 - are there subclasses/superclasses within the entities?
-

Design Using the ER Model (cont)


Attribute vs Entity Example (v1)

 [Diagram:Pic/er-rel/attr-ent1.png]

Employees can work for several departments, but cannot work for the same department over two different time periods.

Design Using the ER Model (cont)

Attribute vs Entity Example (v2)

 [Diagram:Pic/er-rel/attr-ent2.png]

Employees can work for the same department over two different time periods.

Design Using the ER Model (cont)

Design is initially somewhat fluid; no single "correct" answer.

Some design refinements:

- an attribute might be used in several entities
→ attribute becomes new entity+relationships
- entity used via simple relationship to one other entity
→ entity becomes attribute of other entity

The texts discuss design issues relatively briefly.

Other texts discuss design in more detail (e.g. Conolly/Begg)

To some extent, it's a case of "learn by practising"

ER Model for Bank

Kinds of information in model:

- attributes: name, address, date, dollar amount, ...
- entities: customer, account, branch, transaction, ...
- relationships: holds, worksAt, manages, ...

How to give a model?

- detailed description of attribute domains
- detailed description of entities (maybe SQL notation)
- detailed description of relationships (especially constraints)
- ER diagram showing how it all "fits together"

ER Model for Bank (cont)

ER diagrams are typically too large to fit on a single screen.
(or a single sheet of paper, if printing)

One commonly used strategy:

- define entity sets separately, showing attributes
- if relationships have attributes, treat similarly
- combine entities and relationships on a single diagram
(without attributes)
- if very large design, may use several linked diagrams

ER Model for Bank (cont)

Modelling people:

 [Diagram:Pic/er-rel/bank1.png]

ER Model for Bank (cont)

Modelling people (cont):

 [Diagram:Pic/er-rel/bank2.png]

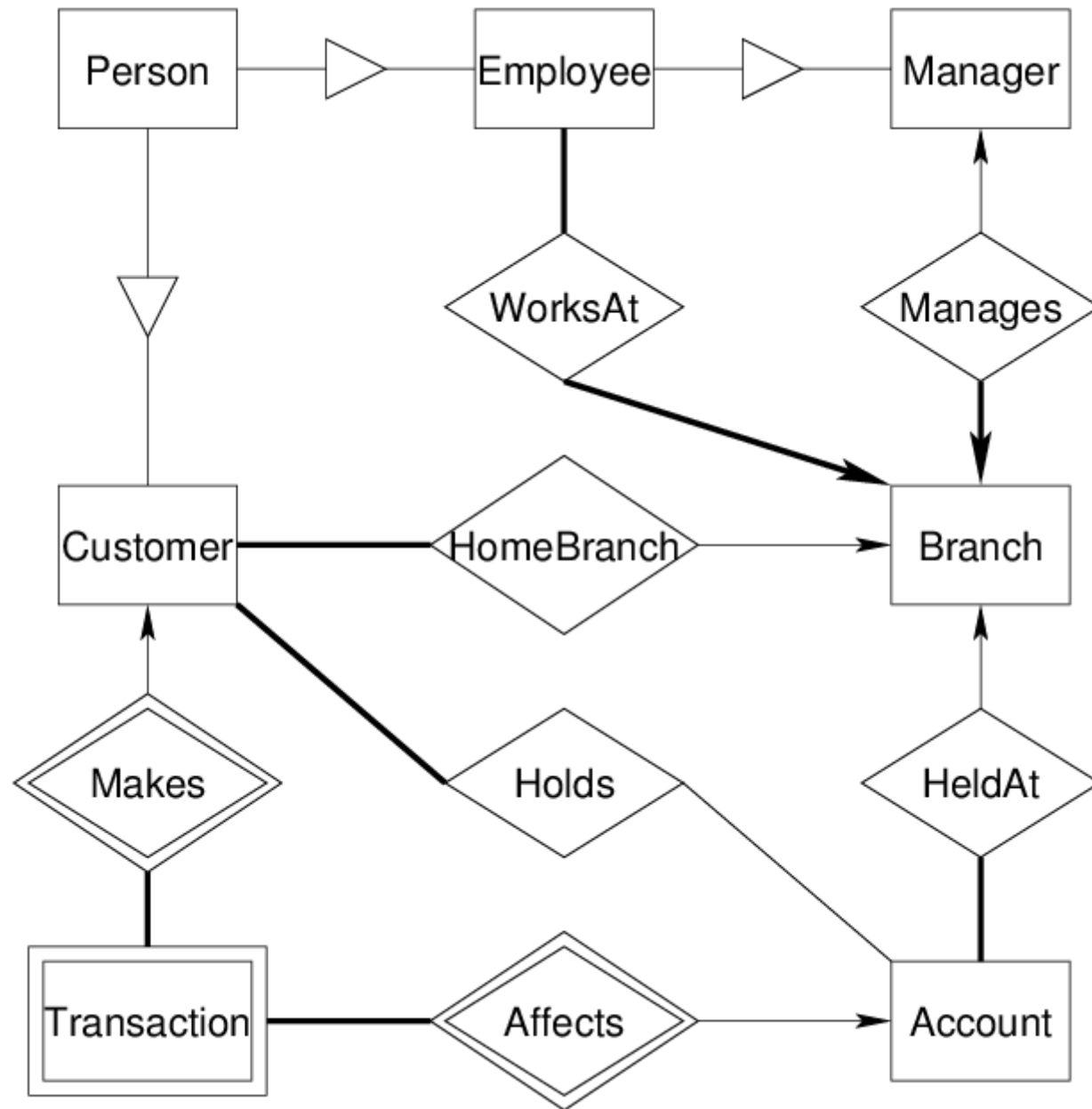
ER Model for Bank (cont)

Modelling branches, accounts, transactions:

 [Diagram:Pic/er-rel/bank3.png]

ER Model for Bank (cont)

Putting it all together with relationships ...



Limitations of ER Model

There are some design aspects that ER does not deal with:

- attribute domains
e.g. should phone "number" be represented by number or string?
- computational dependencies
e.g. employee's salary is determined by department and level
- general constraints
e.g. manager's budget is less than 10% of the combined budget of all departments they manage

Some of these are handled later in the relational model.

Summary

Conceptual design (data modelling) follows requirements analysis.

ER model is popular for doing conceptual design

- has good expressive power, close to how we think

Basic constructs: entities, relationships, attributes

Additional constructs: weak entities, ISA hierarchies

Many notational variants of ER exist
(especially in the expression of constraints on relationships)

Produced: 13 Sep 2020