Relational Table design from EER diagrams
(Mapping from EER to Relational Model)

- Goal of table design
- Principles for table design
  1. **From information capture pt. of view**
  2. **From systems (space,time) and programming effort point of view**
- Rules for table design start from 1 and then are refined using 2.
- Some worked examples

Minimal needs for declaring tables:

- list of columns (and their datatype)
- primary key
- foreign key references
- non-null constraints
- other constraints

A simple way to start:

Create a separate table for every entity and relationship appearing in the EER diagram.

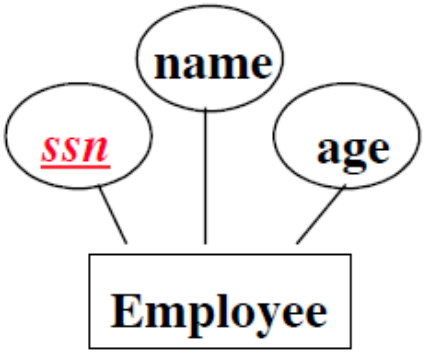
We will go through the various kinds of entities and relationships, showing an example design

(BUT we will have to come back and revisit some of these, in light of other principles of table design, to be discussed a bit later)

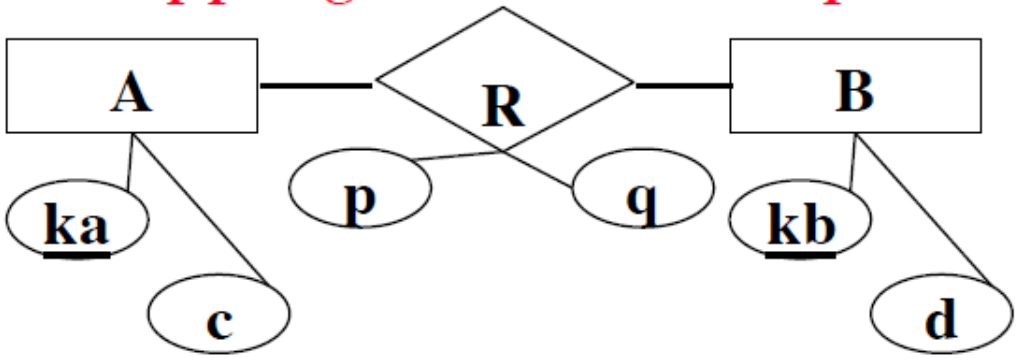
Mapping an Entity Set:

- Relation T_A for every entity A. Attributes of relation are all attributes of the entity.
- Constraints
  - » **key of ER entity becomes primary key**

```
table T_Employee(
        ssn integer primary key,
        name varchar(30),
        age integer
)
```



Mapping a Relationship:

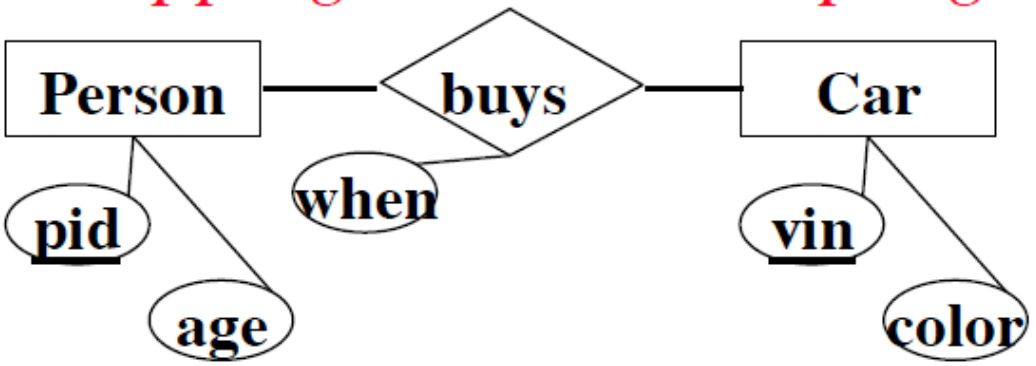


e.g., A=Person, B=Car
R=buys

```
table T_R
        ( ka integer  references T_A   on delete cascade,
          kb char(5) references T_B    on detete cascade,
          p date,
          q char(3),
        primary key ( ka , kb ) )
```

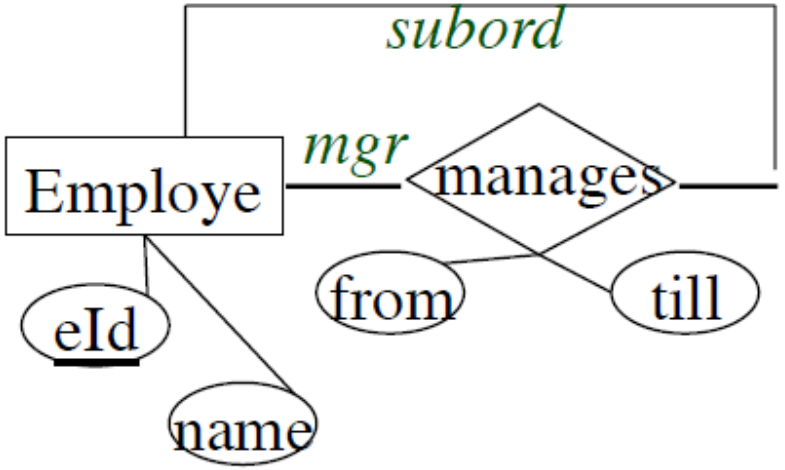
- same idea for n-ary relationships, which have *n* foreign keys then

Mapping a Relationship e.g.:




```
table T_buys
  ( pid integer  references T_Person      on delete cascade,
    vin char(25) references T_Car         on detete cascade,
    when date,
          primary key ( pid,  vin)  )
```

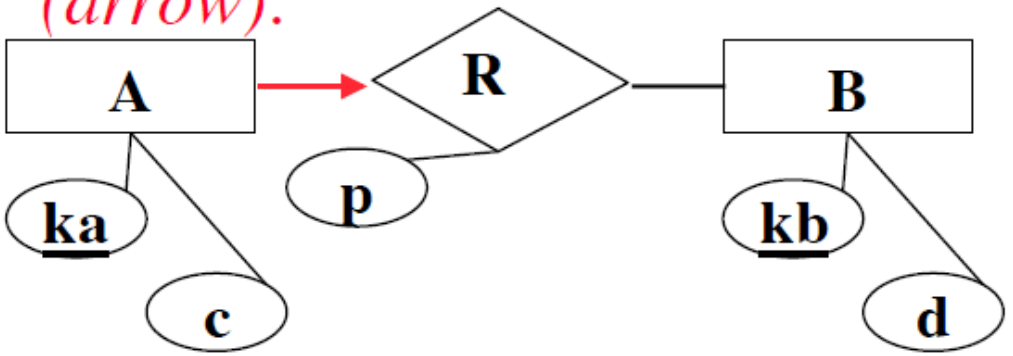
Mapping a "reflexive" relationship (when A = B)




```
T_Employee(eId,  name)
```

```
T_manages(mgr_eId, subord_eId, from, till)
```

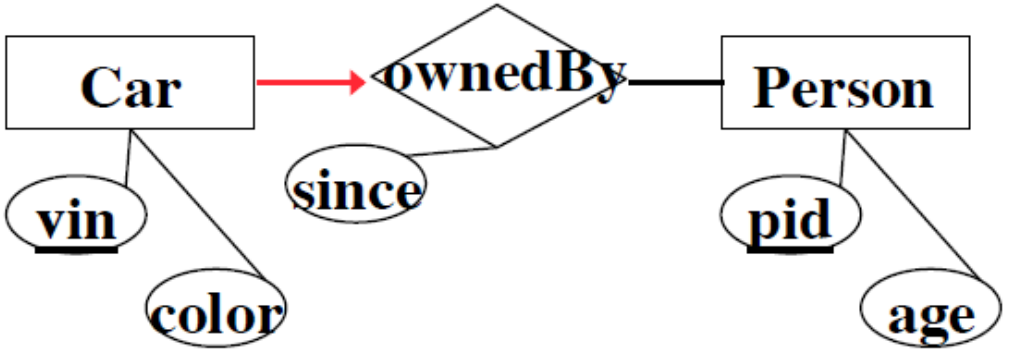
Mapping Relationships with functional constraints (arrow):




```
table T_R
        ( ka char(25) references T_A,
          kb int references T_B,
          p date,
        primary key ( ka ) )
```

- **The key had to be made smaller, since for each A object there is at most one R-related B object, so (ka,kb) is a super-key but not a key.**
- **Note: this way the functional constraint is checked by the schema!**

Mapping Relationships with functional constraints (eg):




```
table T_ownedBy
        ( vin char(25) references T_Car,
          pid int references T_Person,
          since date,
         primary key ( vin ) )
```

Mapping 1-N Relationships vs N:M recap

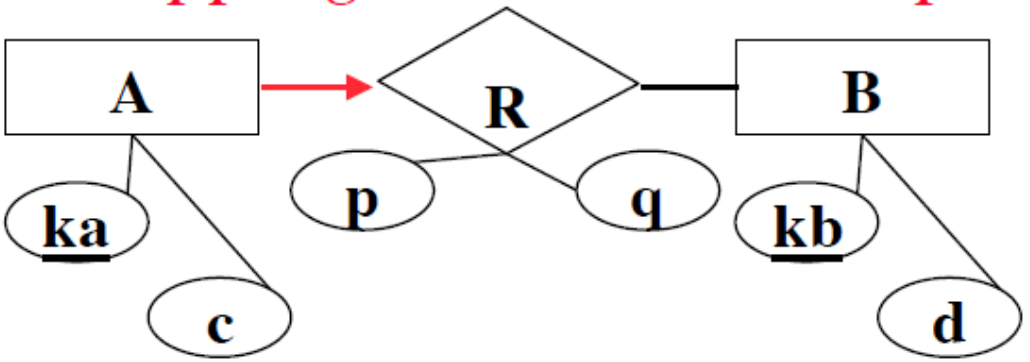


table T_R(<u>ka</u>,kb,p,q)
has ***primary key*** (**ka**)

In general the key of the entity with the "arrow" to the binary relationship is the key to the relationship table itself.

(Suppose B or A's participation in R is 'total' (thick line) Note that there is no constraint on table T_R alone that can enforce this.)

Mapping 1-1 Relationships:

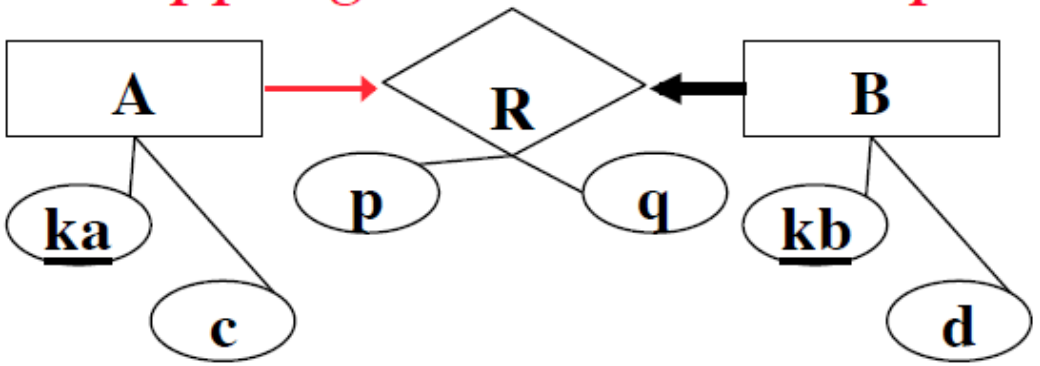
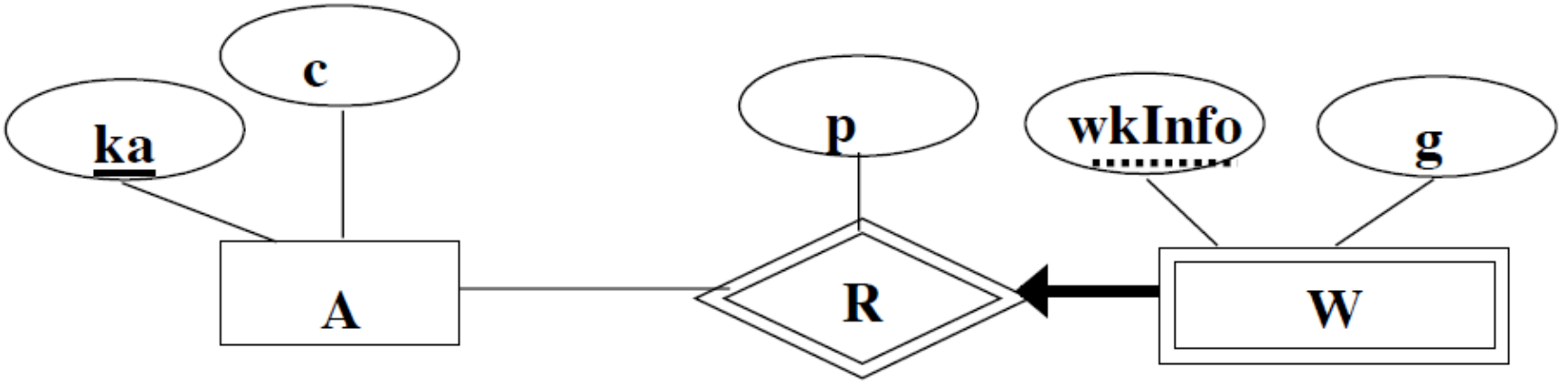


table T_R(<u>ka</u>,kb,p,q) *or* T_R(ka,<u>kb</u>,p,q)
are acceptable alternatives.

If one entity's participation is total (say B), it is better to choose that one to be the primary key (<u>kb</u>) . (See later how NOT NULL can then enforce this constraint in a merged table.)

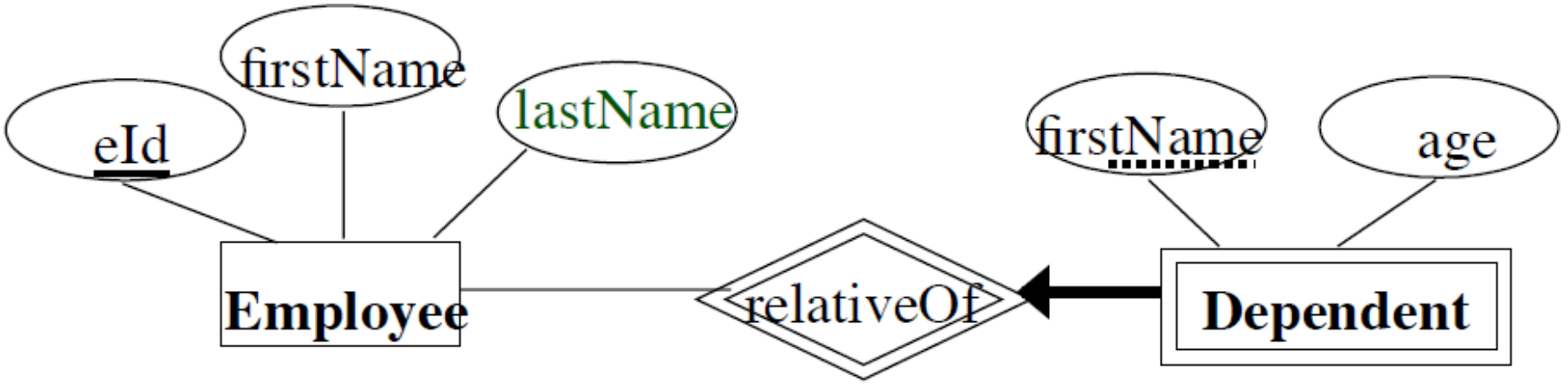
Mapping Weak Entities




```
table T_W(
    wkInfo
    g
    p
    ka  references T_A
    primary key (wkInfo, ka)
```

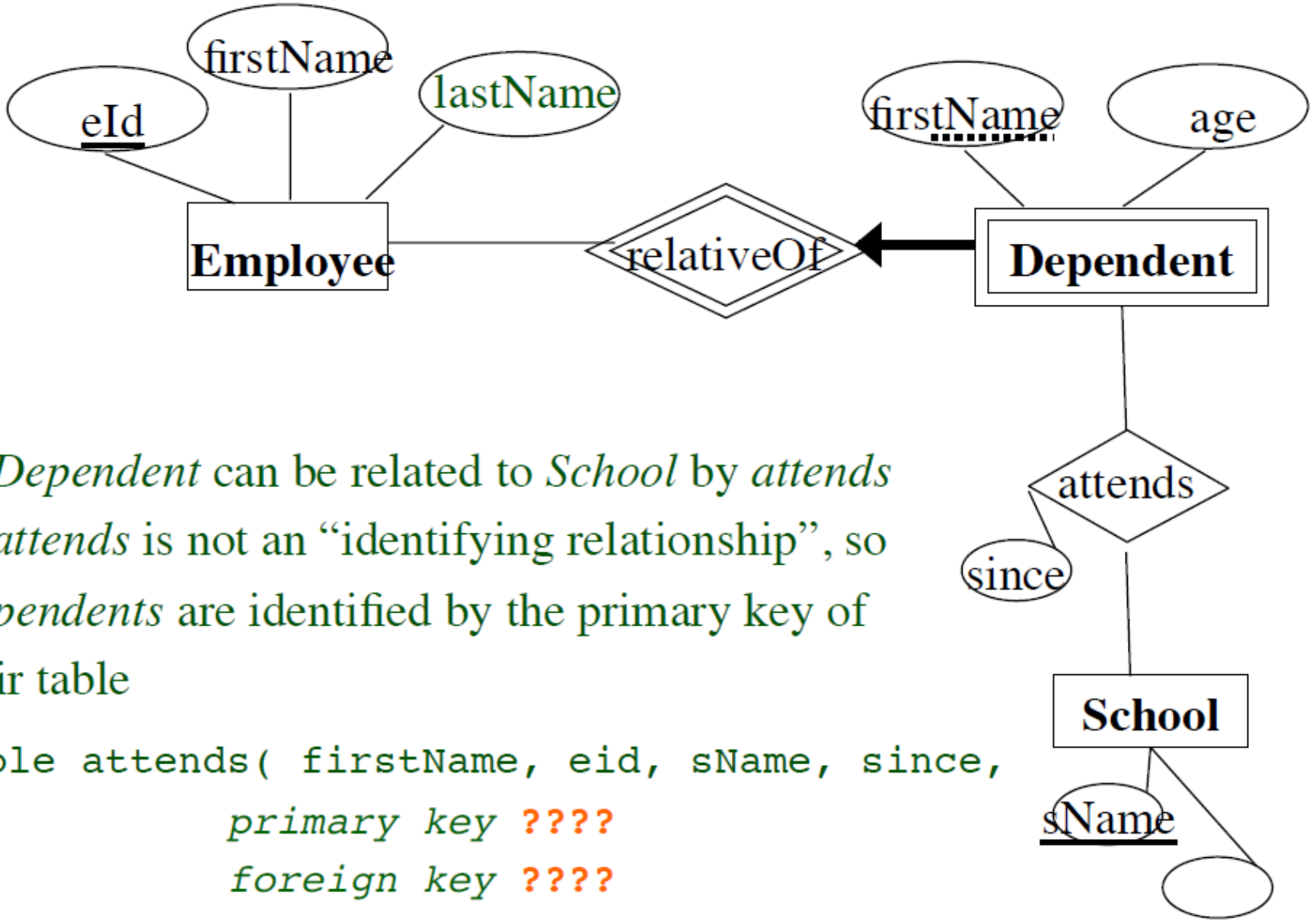
- Cannot have *wkInfo* be a key alone since it does not identify objects in W on its own. (That is the whole point of weak entities)
- Throw in attribute p of the relationship, so T_W is really T_WR

Weak entity example:




```
table T_Dependent(
    firstName,
    eId,                    } Describes a Dependent
   foreign key (eId)  references T_Employee,
    primary key (firstName, eId)  )
```

Weak entities participating in other relationships:

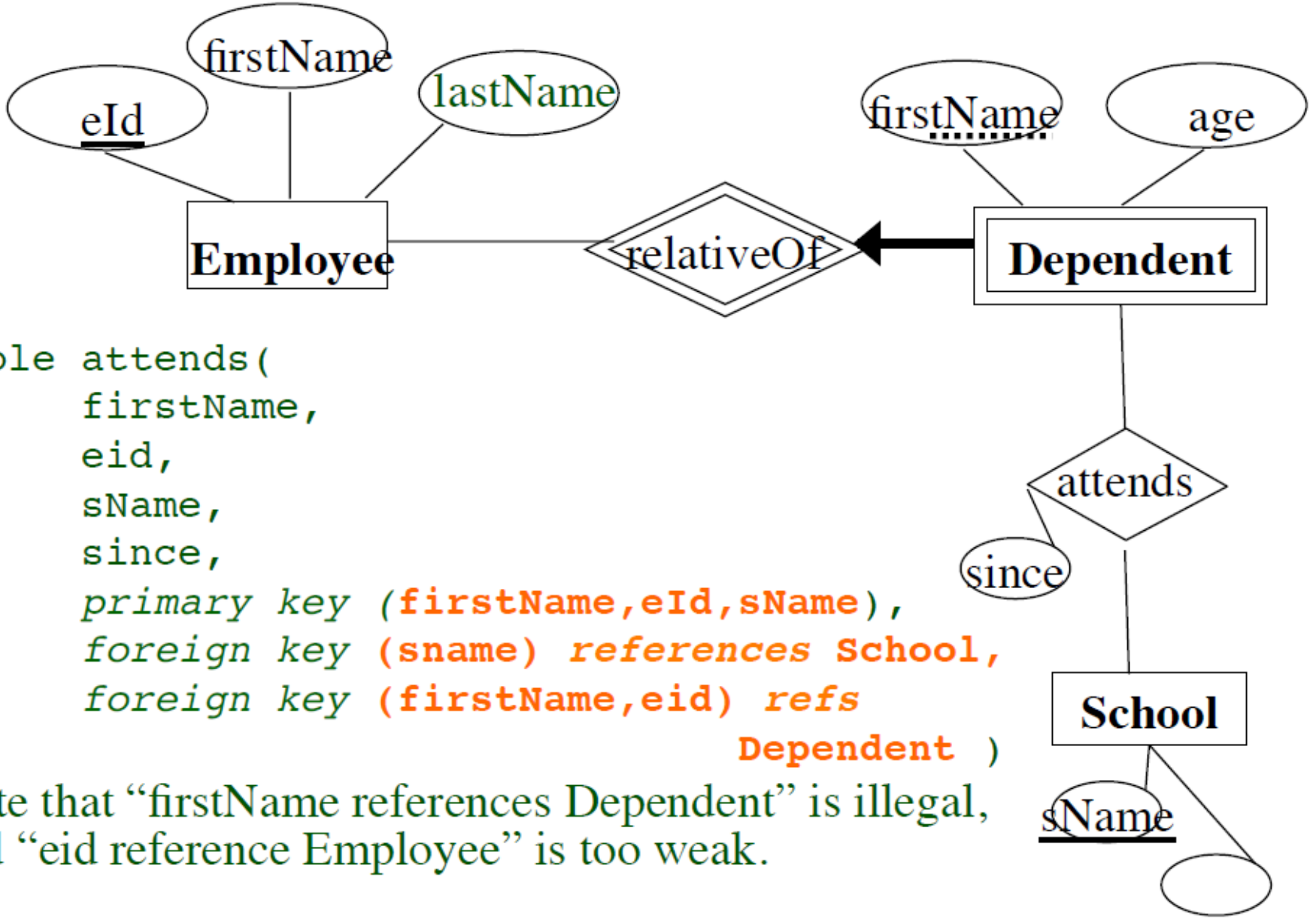


- *Dependent* can be related to *School* by *attends*
- *attends* is not an "identifying relationship", so

Dependents are identified by the primary key of their table

```
table attends( firstName, eid, sName, since,
               primary key ????
               foreign key ????
```

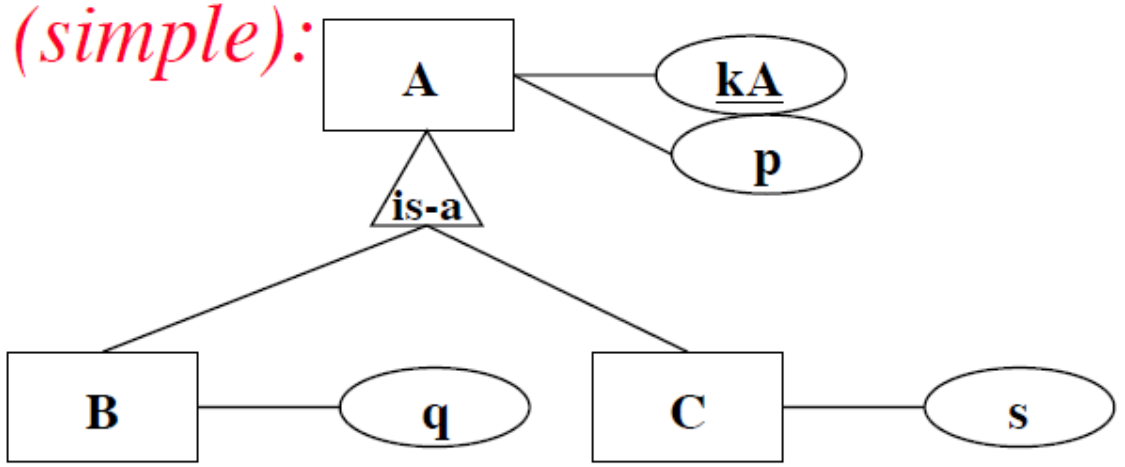
Weak entities participating in other relationships:




```
table attends(
      firstName,
      eid,
      sName,
      since,
      primary key (firstName,eId,sName),
      foreign key (sname) references School,
      foreign key (firstName,eid) refs
                                  Dependent )
```

Note that "firstName references Dependent" is illegal, and "eid reference Employee" is too weak.

Mapping Subclasses (simple):




```
table T_A(
        kA primary key
        p  )
```

```
table T_B(
        kA primary key
        q)
```

foreign key(kA) *references* T_A

```
table T_C(
        kA primary key
        s)
```

foreign key(kA) *references* T_A

These capture the subclass inclusion constraints "every B,C is an A"

- One table per entity, just like before
- For B and C, just inherit the same key as you did for the root class (A here), since every B and C is also an A
- Disjointness or covering by subclasses need to be stated as *check* constraints *Excercise: try writing this!!*

Mapping Subclasses (example):

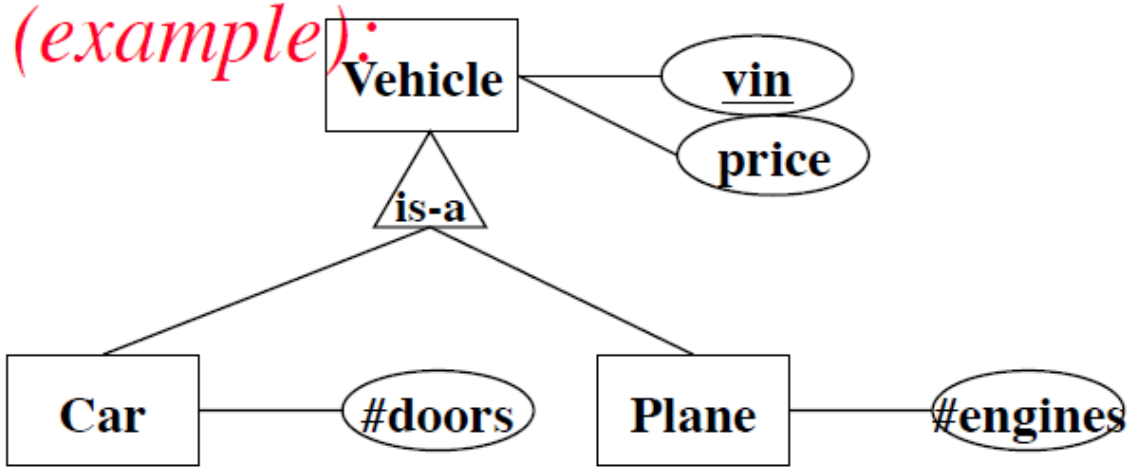


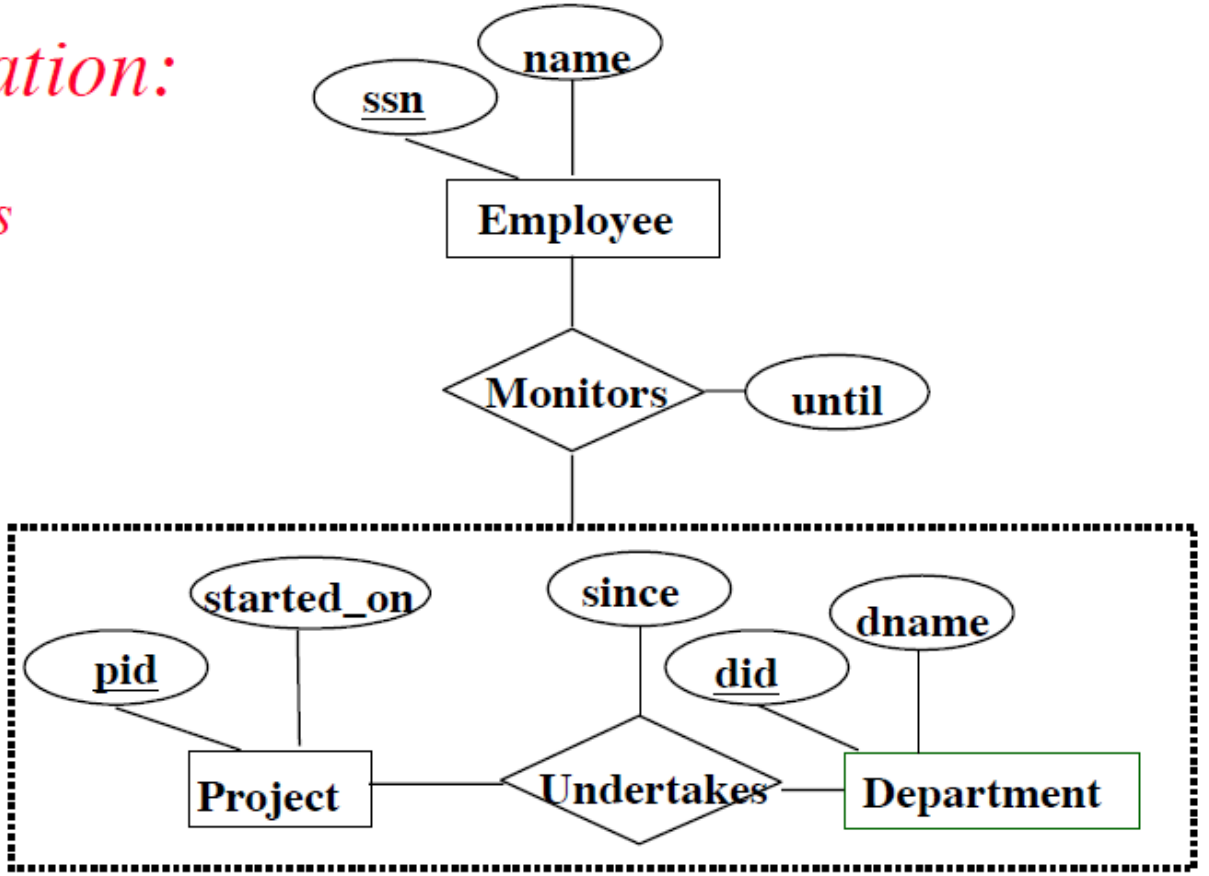
table T_Vehicle(
vin *primary key*
price)

table T_Car(
vin *primary key*
#doors)
foreign key(vin) *references* T_Vehicle

table T_Plane(
vin *primary key*
#engines)
foreign key(vin) *references* T_Vehicle

- One table per entity, just like before
- For B and C, just inherit the same key as you did for the root class (A here), since every B and C is also an A
- Disjointness or covering by subclasses need to be stated as *check* constraints *Excercise: try writing this!!*

Tables for Aggregation:
nothing new:
just apply previous ideas

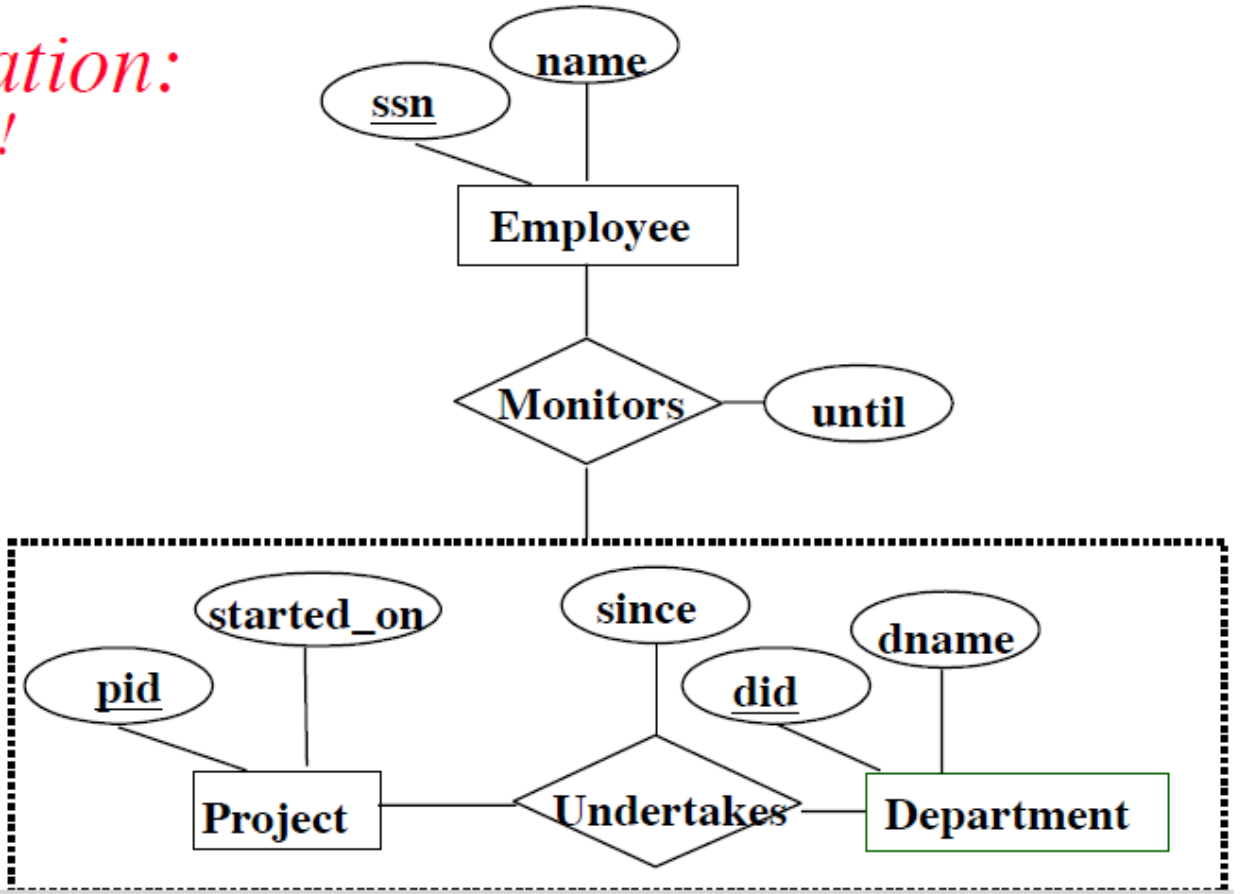


Tables for Aggregation:
just apply previous ideas!

Tables for nested reln's

T_proj(**pid**. started_On)
T_dept(**did**, dname)
T_undertakes(
pid,**did**, since)

T_employee(**ssn**, name)



DRAW >>

- ***Table for "aggregate" relationship treats Udertakes as entity***

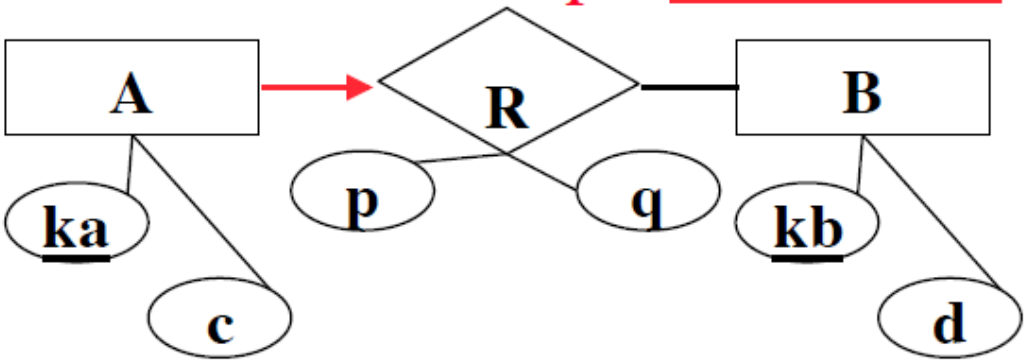
table T_monitors(
ssn integer *references* T_emp
did integer
pid integer
foreign key (did,pid) *references* T_undertakes
until date
primary key **(ssn,did,pid)**)

Merge rule:

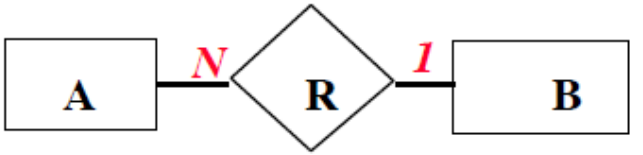
If you have two tables of the form T(K,X) and S(K2,Y) , *Y not null*, where K2 is a foreign key referencing T(K) then you can merge S into T to get instead a single table TS(K,X,Y). Column Y will have NULLs for all keys in T but not S.

- We'll talk later why this is good
- The reason we need Y to be not null, is to be able to distingish the case when a tuple was in T only, but not in S. (Y will be null in table TS in exactly this case.)
- If no such attribute Y is present, a boolean attribute *inS?* can be introduced to distinguish the tuples in TS which were not in S.
- One can think of TS as being the "left outer join" of T and S (the plain join would lose the tuples in T that are not in S)

1-N Relationships **Revisited**



Alternative notation:



- **Merge** T_R(ka,kb,p,q) **into** T_A(ka, c)

```
table T_A
        ( ka                          }
          c
          R_kb  references T_B        }
          p
          q
        primary key ( ka )
```

Describes an A

Describes the B related to A via R