INFS1200 — Week 5 Notes

Table of contents

Applied Class 4: Entity Relationship Mapping **1**
- Section A — Basic mapping . . . 1
- Section B — Advanced mapping . . . 4
- Section C — EER diagram mapping . . . 6

Reference material **7**
- ER Diagram Notation . . . 7
- ER Mapping Pitfalls . . . 10
- ER to Relational Mapping . . . 13
- Relational Mapping Notation . . . 23

Applied Class 4: Entity Relationship Mapping

Practice for 2026-03-16-er-to-relational-mapping[1]. See relational-mapping-notation[2] and er-diagram-notation[3] for reference.

Section A — Basic mapping

Question 1 — Select the correct mapping

A diagram combining several mapping rules at once: `H` (strong entity, key `i`, attribute `j`) and `A` (strong entity, key `b`, attributes `c`, `d`) participate in a ternary relationship `K` (roles: `N` for `H`, `M` for `A`, `1` for `P`, own attribute `l`) together with `P` (strong entity, key `q`, attribute `r`). `A` —1:N

[1] courses/infs1200/2026-03-16-er-to-relational-mapping.pdf
[2] courses/infs1200/relational-mapping-notation.pdf
[3] courses/infs1200/er-diagram-notation.pdf

R1→ E (weak entity, partial key f, attribute g). A —1:1 R2— P. P —1:N S→ M (weak entity, partial key o).

i Working

```
H [i, j]

A [b, c, d]

E [b, f, g]
E.b references A.b

P [q, b, r]
P.b references A.b

K [b, i, q, l]
K.b references A.b
K.i references H.i
K.q references P.q

M [q, o]
M.q references P.q
```

- `E` is a **weak entity** owned by `A` — its key is `(b, f)` (`A`'s key
  - its own partial key `f`).
- `R2` is a plain 1:1 relationship between `A` and `P` — merge it into one side rather than creating a separate relation for it; `P` is extended with `A`'s key as a foreign key here.
- `K` is a **ternary** relationship — its own relation gets a foreign key to *all three* participants, but only the foreign keys from the **many**-cardinality sides (`H`'s `N` and `A`'s `M`) form the primary key; `P`'s foreign key (`q`, the `1` side) is a plain attribute, not part of the key — same pattern as the N-ary "1:1:1 & N:1:1" worked cases.
- `M` is a second **weak entity**, this time owned by `P` — key `(q, o)`.

Question 2 — Subclasses and weak entities

F (key a, composite attribute b → c/d) has a recursive 1:N relationship R (attribute r), and disjointly (d) specialises into G (attributes i, j) and H (attribute k). H —1:N S→ X (weak entity, partial key x, attributes y, z).

(a) What's the relationship between F, G, H, and how are G/H mapped? (b) Interpret S/X with respect to H.

Working

(a) `G` and `H` are **disjoint subclasses** of `F` (an instance is at most one of `G`/`H`, never both). Standard subclass mapping — each subclass's primary key *is* the superclass's primary key, plus a foreign key back to it:

```
F [a, c, d, superF, r]
F.superF references F.a

G [a, i, j]
G.a references F.a

H [a, k]
H.a references F.a
```

(`F.superF`/`r` come from `R`, `F`'s own recursive 1:N relationship — mapped the same way as any binary 1:N relationship, just back onto `F` itself.)
(b) `X` is a **weak entity** owned by `H` via the identifying relationship `S` — each `X` belongs to exactly one `H`, but one `H` can own many `X`s. `X`'s real key is `(a, x)` (`H`'s key, inherited from `F`, plus `X`'s own partial key `x`):

```
X [a, x, y, z]
X.a references H.a
```

Question 3 — Olympics database

Map the ER diagram: *Athlete (id, sex, name, age, country)* —*M:N* *Participates (placement)*→ *Event (id, name, category)* —*N:1* *Venue (id, name, address).*

Working

```
Athlete [id, sex, name, age, country]

Venue [id, name, address]

Event [id, name, category, venue]
Event.venue references Venue.id

Participates [athleteID, eventID, placement]
Participates.athleteID references Athlete.id
Participates.eventID references Event.id
```

`Event`—`Venue` is 1:N, so `Event` (the N side) gets a plain foreign key. `Athlete`—`Event` is M:N, so `Participates` is its own new relation, carrying both foreign keys plus the relationship's own attribute `placement`.

Section B — Advanced mapping

Question 1 — Cinema

Map: `Movie` (`name`, `description`, `runningTime`) —1:N `Viewing` (`date`, `time`, multivalued `Snack`)←1:N `Customer` (`id`, `name`, `email`), each `Viewing` overseen by exactly one `Attendant` (`staffId`, `name`, `phone`).

i Working

```
Movie [name, description, runningTime]

Customer [id, name, email]

Attendant [staffID, name, phone]

Viewing [name, id, date, time, attendant]
Viewing.name references Movie.name
Viewing.id references Customer.id
Viewing.attendant references Attendant.staffID

ViewingSnack [name, id, date, time, snack]
ViewingSnack.{name, id, date, time} references Viewing.{name, id, date, time}
```

`Viewing`'s own key is the composite `(name, id, date, time)` — the movie + customer + timestamp jointly identify one viewing. `Snack` is multivalued, so it gets its own relation (`ViewingSnack`) with a composite foreign key back to the full `Viewing` key.

Question 2 — Student peer evaluation

Map: `Student` (`sid`, `firstName`, `secondName`) submits `Assessments` (`sid`+`number`); students peer-evaluate each other's assessment submissions, recording a `mark`.

Working

```
Student [sid, firstName, secondName]

Assessment [sid, number]
Assessment.sid references Student.sid

PeerEvaluation [studentSid, assessmentSid, number, evaluationId, sidGrades, mark]
PeerEvaluation.studentSid references Student.sid
PeerEvaluation.sidGrades references Student.sid
PeerEvaluation.{assessmentSid, number} references Assessment.{sid, number}
```

`PeerEvaluation` has **two separate foreign keys to `Student`** playing different roles — `studentSid` (the evaluator submitting the mark) and `sidGrades` (whose assessment is being marked) — plus a composite foreign key to the specific `Assessment` being evaluated.

Question 3 — Spot the mistakes

A junior developer mapped question 1's Cinema diagram as:

```
Movie [name, description, runningTime]

Customer [id, name, email]

Attendant [staffID, name, phone]

Viewing [name, id, date, time, attendant]
Viewing.name references Movie.name
Viewing.id references Customer.id
Viewing.attendant references Attendant.staffID

ViewingSnack [name, id, date, time, snack]
ViewingSnack.name references Movie.name
ViewingSnack.id references Viewing.id
ViewingSnack.date references Viewing.date
ViewingSnack.time references Viewing.time
```

Find the mistakes.

Working

`ViewingSnack` should reference `Viewing`'s composite key as **one** foreign key constraint (`ViewingSnack.{name, id, date, time} references Viewing.{name, id, date, time}`), not four *separate*, independently-named foreign key lines pointing at different tables (`Movie` for `name`, `Viewing` for `id`, and `Viewing` again for `date`/`time`, inconsistently) — splitting a composite foreign key up like this breaks the link between a `ViewingSnack` and *the specific* ***`Viewing`*** it belongs to (nothing actually forces all four columns to jointly match one real `Viewing` row), and pointing `name` back at `Movie` instead of `Viewing` is simply the wrong target relation for a foreign key that's supposed to identify *which viewing* the snack belongs to.

Section C — EER diagram mapping

Question 1 — Medical clinic

Map: ***`Doctor`*** *(****`licenceNo`****) splits into* ***`GP`****/****`Specialist`*** *(****`specialisationArea`****);* ***`Doctor`*** *—M:N* ***`WorksIn`***→ ***`Clinic`*** *(****`regNo`****);* ***`Clinic`*** *has a subclass* ***`SpecialistClinic`****, run by exactly one* ***`Specialist`*** *(****`Runs`****, 1:N);* ***`Patient`*** *(****`patientId`****) —M:N* ***`Appointment`*** *(****`dateTime`****,* ***`fee`****)→* ***`Doctor`****.*

Working

```
Doctor [licenceNo]

Patient [patientId]

Clinic [regNo]

GP [licenceNo]
GP.licenceNo references Doctor.licenceNo

Specialist [licenceNo, specialisationArea]
Specialist.licenceNo references Doctor.licenceNo

SpecialistClinic [regNo]
SpecialistClinic.regNo references Clinic.regNo

Appointment [licenceNo, dateTime, patientId, fee]
Appointment.licenceNo references Doctor.licenceNo
Appointment.patientId references Patient.patientId
```

```
WorksIn [licenceNo, regNo]
WorksIn.licenceNo references Doctor.licenceNo
WorksIn.regNo references Clinic.regNo

Runs [licenceNo, regNo]
Runs.licenceNo references Specialist.licenceNo
Runs.regNo references SpecialistClinic.regNo
```

`GP`/`Specialist` are standard disjoint-subclass mappings of `Doctor`. `SpecialistClinic` is a subclass of `Clinic` with no extra attributes of its own — just a foreign key back to `Clinic`. `Runs` is a 1:N relationship between `Specialist` and `SpecialistClinic` specifically (not the general `Doctor`/`Clinic` pair), so its foreign keys target the *subclass* relations, not the superclasses.

Reference material

ER Diagram Notation

Reference legend for the symbols used across every ER/EER diagram in this course — introduced across 2026-02-23-conceptual-database-design-and-the-er-model[4] and 2026-03-02-weak-entities-the-eer-model-and-design-choices[5].

The notation below is the **mandatory** standard: it is what the lectures' own notation guide specifies, and it is what assignments and the exam are marked against. (The separately published *INFS1200/7900 EER & ER Notation Guide* handout only standardises *naming* — see the style guide at the bottom of this note — and explicitly defers the symbols themselves to the lecture material.)

Core symbols

Symbol	Meaning
Rectangle	**Entity type**
Double-line rectangle	**Weak entity type** (no key attribute of its own)
Oval	**Attribute**
Oval, name underlined	**Key attribute**

[4] courses/infs1200/2026-02-23-conceptual-database-design-and-the-er-model.pdf
[5] courses/infs1200/2026-03-02-weak-entities-the-eer-model-and-design-choices.pdf

Symbol	Meaning
Oval, name underlined with a dotted line	**Partial key attribute** (weak entity's own distinguishing attribute)
Double-line oval	**Multivalued attribute** (can hold more than one value)
Dashed oval, attached by a dashed line	**Derived attribute** (computed from another stored attribute, e.g. `Age` from `BirthDate`)
Diamond	**Relationship type**
Double-line diamond	**Identifying relationship type** (connects a weak entity to its owner entity)

- **Composite attribute**: an attribute with its own sub-attributes attached (e.g. `Name` splitting into `FirstName`/`MiddleName`/`LastName`) — the parent attribute oval isn't underlined/dashed/doubled itself unless it's also a key/derived/multivalued attribute.
- **Value sets** (the domain of legal values for an attribute, e.g. "integers 21-65") are never shown on the diagram itself.

Relationship constraints

Symbol	Meaning
`1`, `N`, `M` on the lines into a diamond	**Cardinality ratio** — `E1 -1- R -N- E2` reads as ratio 1:N for E1:E2 in R
Double line between an entity type and a diamond	**Total participation** of that entity type in that relationship
Single line	**Partial participation**

See relationship-constraints[6] for what each ratio and each participation constraint actually asserts.

Subclasses / superclasses (EER)

A subclass is drawn as an ordinary rectangle. What makes it a subclass is the symbol on the line joining it to its superclass, not the shape of its box — a subclass box is a plain single-line rectangle, never the double-line rectangle that means *weak entity*. The two are entirely different ideas (a subclass inherits its superclass's key; a weak entity has no key of its own at all) and the notation keeps them apart.

[6] courses/infs1200/relationship-constraints.pdf

Symbol	Meaning
on the line between two entity types	The entity type on the *closed* (narrow) side of the is a **subclass** of the one on the *open* (wide) side — read it as the set-inclusion symbol, `subclass superclass`
A circle joining several subclass lines to one superclass, each line carrying its own opening towards the circle	Several entity types are all **subclasses of the same superclass**
`d` inside that circle	**Disjoint** specialisation — an entity instance can belong to at most one of the subclasses
`o` inside that circle	**Overlapping** specialisation — an entity instance can belong to more than one subclass at once
Single line from superclass to the subclass circle	**Partial** specialisation — not every superclass instance needs to belong to a subclass
Double line from superclass to the subclass circle	**Total** specialisation — every superclass instance must belong to at least one subclass

Naming conventions (INFS1200/7900 style guide)

Course-specific naming standard for entities/relationships/attributes on ER and EER diagrams (distinct from — and in addition to — the mandatory notation above):

- **Entity names**: capitalised, no spaces, ideally one word (e.g. `STUDENT`, `POLICEOFFICER` — not `PoliceOfficer`).
- **Relationship names**: capitalised, no spaces, ideally one word, preferably a verb (e.g. `CREATES`, `ACTSIN` — not `LeadActor`).
- **Attribute names**: UpperCamelCase — first letter of each word capitalised, no spaces, acronym letters stay capitalised (e.g. `ComputerIP`, `DateOfBirth` — not `Date of Birth`).

How diagrams are drawn in these notes

Diagrams in this course's notes are drawn as Mermaid `flowchart` graphs (renders in both PDF and HTML), using shapes chosen to match the Chen notation above as closely as Mermaid's flowchart shapes allow:

Mermaid shape	Used for
`[Entity]` (rectangle)	Entity type — **and subclasses**, which are ordinary rectangles

Mermaid shape	Used for
`[[WeakEntity]]` (subroutine, double vertical bars)	Weak entity type, and nothing else
`{Relationship}` (diamond/rhombus)	Relationship type
`([Attribute])` (stadium)	Attribute
`((d))` / `((o))` (small circle)	Disjoint/overlapping specialisation marker

The `[[X]]` shape is **reserved for weak entity types**. Drawing an EER subclass with it is a notation error, not a stylistic one: it renders the double-line box that the official notation uses for a weak entity, and so asserts the opposite of what a subclass is. Subclasses are plain `[X]` rectangles; their subclass-ness is carried by the edge into the `((d))` or `((o))` circle, exactly as the carries it in the hand-drawn notation.

Since Mermaid can't reliably underline/dash node text across renderers, attribute qualifiers are written as a plain-text suffix instead: `(key)`, `(partial key)`, `(derived)`, `(multi)`. Composite attributes are drawn as a parent attribute node connected to its component attribute nodes. Cardinality is a text label on the edge (`"1"`, `"N"`, `"M"`); **total** participation/identifying relationships use a thick edge (`===`/`==>`), **partial** participation uses a thin edge (`---`/`-->`).

Any diagram with more than ~6-8 nodes needs an explicit `%%| fig-width:` cell option (in inches) or it runs off the right edge of the PDF page - mermaid sizes itself from the diagram's own SVG bounding box, and there's no project-wide default that reaches it (a document/project `fig-width` only applies to executed-code-cell figures, e.g. matplotlib). `5.5` fits this vault's PDF page width (`scrartcl`, `DIV=11`, letter) comfortably:

````
```{mermaid}
%%| fig-width: 5.5
flowchart TD
    ...
```
````

ER Mapping Pitfalls

A checklist for reviewing a finished relational schema — every item here is a mistake actually made and corrected in week5-tutorial-applied-class-4-entity-relationship-mapping[7], week4-tutorial-case-study-3-dirt-road-driving[8] or the worked questions in 2026-03-16-er-to-relational-mapping[9]. The mapping rules themselves are in er-to-relational-mapping[10]; this is what to check *after* applying them.

[7] courses/infs1200/week5-tutorial-applied-class-4-entity-relationship-mapping.pdf
[8] courses/infs1200/week4-tutorial-case-study-3-dirt-road-driving.pdf
[9] courses/infs1200/2026-03-16-er-to-relational-mapping.pdf
[10] courses/infs1200/er-to-relational-mapping.pdf

Keys

Most of this section is one question asked five ways: *what exactly is this relation's primary key?* (relational-keys[11]).

A weak entity's partial key is not a foreign key. For a weak `CITY` owned by `PROVINCE`, the correct mapping is `Cities [cName, pName, mayor]` with only `Cities.pName references Provinces.pName`. `cName` is the city's own partial key — it references nothing. Declaring both columns as foreign keys is wrong.

A weak entity's key includes its owner's key. Mapping `Cities [cName, mayor]`, keyed on `cName` alone, silently asserts that city names are globally unique — which is precisely the assumption that made `CITY` weak in the first place.

An ordinary 1:N foreign key is not part of the primary key. For `A -N R 1→ B`, the answer is `A [a, b, d]` keyed on `a` alone, with `A.b references B.b` as a plain column. Promoting `b` into `A`'s key is weak-entity treatment applied to a strong entity. (The genuine exception is an entity with no key of its own, identified through its owners — see foreign-keys[12].)

A composite foreign key is one constraint, not several. The cinema schema must say

```
ViewingSnack.{name, id, date, time} references Viewing.{name, id, date, time}
```

and not four separate single-column constraints. Splitting it breaks the link to *the specific* `Viewing` row: nothing then forces all four columns to match one real viewing rather than four unrelated ones. In the same botched attempt, `name` was pointed back at `Movie` rather than `Viewing` — a foreign key meant to identify *which viewing* a snack belongs to must target `Viewing`.

When the owner's key is composite, the inherited foreign key is composite. `Branch`'s key is `(bankCode, branchNo)`, so accounts and loans at a branch carry both columns: `Account.{bankCode, branchNo} references Branch.{bankCode, branchNo}`.

Relations that shouldn't exist

An identifying relationship never becomes a relation. A weak entity is connected to its owner purely through the inherited key. Inventing an `In [cName, pName]` relation alongside the weak entity's own is a distinctive wrong answer.

1:1 and 1:N relationships don't get their own relation by default. They are foreign keys on an existing relation. Creating an `S [b, c]` relation for a 1:N relationship `S` treats a

[11] courses/infs1200/relational-keys.pdf
[12] courses/infs1200/foreign-keys.pdf

relationship type as an entity type. (Doing this *deliberately* for a sparse relationship, to avoid nulls, is a different matter — a considered choice, not the default rule.)

Two entity types don't merge into one relation. Collapsing `S` and `T` into `ST [a1, b1]` because they happen to be in a 1:1 relationship discards a conceptual modelling decision that was made on purpose.

Nor does a relationship simply vanish. Mapping `S -1:1 R- T` to `S [a1]`, `T [b1]` with no foreign key anywhere loses all information about `R`.

Choosing sides

For 1:1, put the foreign key on the total-participation side. If every department has a manager but only some employees manage, extend `Department`. Choosing the other side doesn't lose information, but it fills the column with nulls, and the course's rule is to prefer the total-participation side. Between a schema putting `b1` into `S` and one putting `a1` into `T`, the first is correct for exactly one reason: `S` is preferred over `T` because it has full participation.

For 1:N, the foreign key goes on the N side — no judgement involved. In a chain `A -N R 1→ B -N S 1→ C`, that means `A` references `B` and `B` references `C`; each arrow is decided independently by where the N is.

For N-ary, only the many-sides make the key. All participants get foreign keys, but a `1`-side participant's foreign key is a plain attribute — and if the relationship carries a key attribute of its own, that attribute joins the key alongside the many-side foreign keys.

A ternary relationship with no many side gets no relation. `1:1:1` and `N:1:1` are explicitly excluded from step 7: creating an `R` relation for one of them is a wrong answer, because the relationship is absorbed into a participating relation as two foreign keys — into the `N` side if there is one, otherwise into whichever side is chosen. See step 7 of er-to-relational-mapping[13].

Attributes

Derived attributes are never columns. `NumberOfEmployees` is computed from the data, so storing it just creates something that can disagree with the data.

Composite attributes contribute their leaves, not themselves. `Name` becomes `fName`, `mIt`, `lName` — there is no `name` column.

Multivalued attributes always become their own relation, keyed on the owner's key plus the value. A column can only hold one atomic value (see relational-model[14]).

[13] courses/infs1200/er-to-relational-mapping.pdf
[14] courses/infs1200/relational-model.pdf

EER

Relationships drawn to a subclass reference the subclass relation. If only a `Specialist` can run a `SpecialistClinic`, then `Runs.licenceNo references Specialist.licenceNo` — not `Doctor`. Pointing at the superclass silently widens the rule to every doctor and every clinic, losing the constraint the specialisation was drawn to express (specialisation-and-generalisation[15]) — and the same widening is what er-design-choices[16] calls the subclass-vs-superclass relationship fork.

A subclass with no attributes of its own still gets a relation. `SpecialistClinic [regNo]` with a foreign key back to `Clinic` records *membership*, which is information even when there are no extra columns.

Map subclasses before the relationships that use them. The `+1` step runs right after step 1 or 2 for a reason — a later step that needs to reference `Specialist` can't if `Specialist` doesn't exist yet.

Style

Schemas are marked against the course style guide as well as the mapping rules: UpperCamelCase table names, lowerCamelCase attribute names, each relation's `references` lines directly beneath it, and a single continuous underline for a composite primary key. See relational-mapping-notation[17].

ER to Relational Mapping

The (7+1)-step algorithm that converts a conceptual ER/EER diagram into a logical relational schema — input: an ER model; output: relations with primary and foreign key constraints. Introduced in 2026-03-16-er-to-relational-mapping[18] and practised in week5-tutorial-applied-class-4-entity-relationship-mapping[19] and week4-tutorial-case-study-3-dirt-road-driving[20]. See relational-mapping-notation[21] for the notation and naming style used throughout, er-diagram-notation[22] for the diagram being mapped *from*, and er-mapping-pitfalls[23] for the mistakes to check for afterwards.

[15] courses/infs1200/specialisation-and-generalisation.pdf
[16] courses/infs1200/er-design-choices.pdf
[17] courses/infs1200/relational-mapping-notation.pdf
[18] courses/infs1200/2026-03-16-er-to-relational-mapping.pdf
[19] courses/infs1200/week5-tutorial-applied-class-4-entity-relationship-mapping.pdf
[20] courses/infs1200/week4-tutorial-case-study-3-dirt-road-driving.pdf
[21] courses/infs1200/relational-mapping-notation.pdf
[22] courses/infs1200/er-diagram-notation.pdf
[23] courses/infs1200/er-mapping-pitfalls.pdf

Why map at all

The ER model is used for **conceptual design** — the user's perspective, good for talking to stakeholders about entities and relationships. The relational model is what commercial DBMSs actually implement — the storage perspective. Mapping is the bridge: every construct that the ER model can draw has to be re-expressed using only relations, primary keys[24] and foreign-keys[25], because those are the only tools the relational model has (see relational-model[26]).

The steps at a glance

#	Construct	Produces
1	Strong entity type	A new relation
2	Weak entity type	A new relation, keyed on owner's key + partial key
3	Binary 1:1 relationship	A foreign key on one side
4	Binary 1:N relationship	A foreign key on the N side
5	Binary M:N relationship	A new relation
6	Multivalued attribute	A new relation
7	N-ary relationship	A new relation — except 1:1:1 and N:1:1, which take foreign keys instead
+1	Subclass (EER)	A new relation per subclass

Only steps 1, 2, 5, 6, 7 and +1 create relations; steps 3 and 4 only add foreign keys to relations that already exist, and step 7 does the same in its two excluded cases. Working in order matters, because every later step needs the primary keys that earlier steps established.

The running example below is the **Company database**: `EMPLOYEE` (key `Ssn`, composite `Name` → `Fname`/`Mit`/`Lname`, `Sex`, `Salary`, `Address`, `DOB`), `DEPARTMENT` (keys `Dnumber`/`Dname`, derived `NumberOfEmployees`, multivalued `Locations`), `PROJECT` (keys `Pno`/`Pname`, `Plocation`), and the weak entity `DEPENDENT` (partial key `DepName`), with relationships `WORKSFOR` (`EMPLOYEE` N:1 `DEPARTMENT`), `MANAGES` (`EMPLOYEE` 1:1 `DEPARTMENT`, attribute `StartDate`), `CONTROLS` (`DEPARTMENT` 1:N `PROJECT`), `WORKSON` (`EMPLOYEE` M:N `PROJECT`, attribute `Hours`), `SUPERVISION` (`EMPLOYEE` recursive 1:N), `DEPENDENTSOF` (identifying), and `EMPLOYEE` disjointly specialising into `SECRETARY`/`ENGINEER`.

[24] courses/infs1200/relational-keys.pdf
[25] courses/infs1200/foreign-keys.pdf
[26] courses/infs1200/relational-model.pdf

Step 1 — Strong entity types

For each strong (non-weak) entity type: create a relation, choose one of its keys as the primary key, and include **all simple attributes**.

```
Employee [ssn, fName, mIt, lName, dob, address, sex, salary]

Department [dNumber, dName]

Project [pNo, pName, pLocation]
```

Three attribute kinds are deliberately not copied across as columns:

- **Composite** attributes contribute their simple components instead — `Name` never appears, `fName`/`mIt`/`lName` do.
- **Derived** attributes are never stored at all — `NumberOfEmployees` is computed on demand, so storing it would only create an opportunity for it to go stale.
- **Multivalued** attributes can't be stored in an atomic column and are deferred to step 6 — `Locations` is missing from `Department` for now.

Where an entity has several keys (`DEPARTMENT` has both `Dnumber` and `Dname`), one becomes the primary key and the others are simply ordinary attributes in the relational schema — see relational-keys[27].

Step 2 — Weak entity types

For each weak entity type: create a relation whose primary key is **the owner's primary key plus the weak entity's partial key**; add a foreign key from the inherited part back to the owner; include all simple attributes.

```
Dependent [ssn, depName, sex, dob, relationship]
Dependent.ssn references Employee.ssn
```

`Dependent`'s full key is `(ssn, depName)`: `ssn` is `Employee`'s key inherited through the identifying relationship, `depName` is the partial key (see relationship-constraints[28]). Note what does *not* happen — the identifying relationship `DEPENDENTSOF` gets no relation of its own; it is entirely represented by the inherited key.

If a weak entity has **more than one owner**, the rule generalises rather than changes: include as foreign keys the primary key attributes of *every* owner entity type, and make the primary

[27] courses/infs1200/relational-keys.pdf
[28] courses/infs1200/relationship-constraints.pdf

key the combination of all of them plus the partial key. For a weak `W` (partial key `a1`) owned by both `E1` (key `b1`) and `E2` (key `c1`):

```
E1 [b1]

E2 [c1]

W [a1, b1, c1]
W.b1 references E1.b1
W.c1 references E2.c1
```

All three columns are underlined as one key; both inherited keys are foreign keys in their own right.

When the owner's own key is composite, the foreign key is composite too. In the Bank example, `BRANCH` is a weak entity owned by `BANK` (`branchNo` is only unique within a bank):

```
Bank [code, name, hoAddr]

Branch [bankCode, branchNo, addr]
Branch.bankCode references Bank.code
```

Step 3 — Binary 1:1 relationships

For each binary 1:1 relationship: pick one participating entity type, add a foreign key there referencing the other's primary key, and put the relationship's own simple attributes on that same chosen side.

Preference is given to entity types with total participation (relationship-constraints[29]) — it is part of the rule, not a stylistic tip. In the generic case, with `T` the totally-participating side (double line) and `c1` the relationship's own attribute:

```
S [a1, a2]

T [b1, b2, a1, c1]
T.a1 references S.a1
```

[29] courses/infs1200/relationship-constraints.pdf

If every department must have a manager but not every employee manages, then putting the foreign key on `Department` gives a value in every row, whereas putting it on `Employee` would leave it null for almost everyone (sql-nulls[30]). `MANAGES` is drawn with the double line on the `DEPARTMENT` side, so `DEPARTMENT` is the chosen side:

```
Department [dNumber, dName, mgrSSN, mgrStartDate]
Department.mgrSSN references Employee.ssn
```

`mgrStartDate` is `MANAGES`'s own attribute, and travels with the foreign key. The foreign key is a plain attribute here, not part of `Department`'s key.

Both landed columns are **renamed** on arrival — `Employee.ssn` becomes `mgrSSN`, `StartDate` becomes `mgrStartDate` — and the same happens in step 4 (`Dnumber` → `dNumber` in `Employee`, `Ssn` → `superSSN` for `SUPERVISION`). Renaming an inherited attribute to say what role it plays in its new relation is expected, and is the standard fix when the imported name would collide with one already there.

Step 4 — Binary 1:N relationships

For each binary 1:N relationship that isn't an identifying relationship: find the entity type on the **N side**, add a foreign key there referencing the primary key of the entity type on the **1 side**, and put the relationship's own simple attributes on the N side too.

The direction is forced, not chosen: one row on the N side relates to exactly one row on the 1 side, so a single foreign key column can hold it. The other direction would need many values in one cell.

```
Employee [ssn, fName, mIt, lName, dob, address, sex, salary, dNumber, superSSN]
Employee.dNumber references Department.dNumber
Employee.superSSN references Employee.ssn

Project [pNo, pName, pLocation, dNumber]
Project.dNumber references Department.dNumber
```

`SUPERVISION` is recursive, and is handled exactly like any other 1:N relationship — the N-side role (supervisee) carries a foreign key back to the 1-side role (supervisor) in the same relation.

[30] courses/infs1200/sql-nulls.pdf

Step 5 — Binary M:N relationships

For each binary M:N relationship: create a **new** relation; give it foreign keys to both participants' primary keys; **the combination of those foreign keys is its primary key**; add the relationship's own simple attributes.

```
WorksOn [ssn, pNo, hours]
WorksOn.ssn references Employee.ssn
WorksOn.pNo references Project.pNo
```

A new relation is unavoidable here: neither side can hold a single foreign key value, since both sides relate to many rows of the other.

One wrinkle the Company example doesn't show: the relationship may bring a **key attribute of its own**, and if it does, that attribute joins the primary key rather than sitting outside it. For an M:N `R` between `S` and `T` carrying an underlined `c1` and a plain `c2`:

```
S [a1, a2]

T [b1, b2]

R [a1, b1, c1, c2]
R.a1 references S.a1
R.b1 references T.b1
```

`R`'s key is `(a1, b1, c1)` — the pair of foreign keys *plus* `c1`. This is how a relationship that can repeat between the same two entities (the same employee working on the same project in several different years, say) gets modelled without a duplicate-key clash.

1:1 and 1:N relationships *may* also be mapped this way, and it is worth doing when the relationship is **sparse** — a separate relation stores only the rows that actually participate, instead of leaving a null foreign key in every row that doesn't. In the standard implementation the foreign key column is mostly nulls, while the M:N implementation holds one short relation of just the participating pairs, and no nulls at all.

Step 6 — Multivalued attributes

For each multivalued attribute: create a new relation containing a foreign key to the owning entity's primary key plus the attribute itself; **the primary key is the combination of the two**. If the multivalued attribute is composite, include its simple components instead of the attribute itself.

```
DeptLocs [dNumber, location]
DeptLocs.dNumber references Department.dNumber
```

Keying on both columns is what allows one department to have many locations while forbidding the same location twice. If the owning relation's key is itself composite, the foreign key here is composite — in the cinema example the snack relation carries all four columns of `Viewing`'s key:

```
ViewingSnack [name, id, date, time, snack]
ViewingSnack.{name, id, date, time} references Viewing.{name, id, date, time}
```

Step 7 — N-ary relationships

For each N-ary relationship type — **excluding the 1:1:1 and N:1:1 cases**, which are handled separately below: create a new relation; give it foreign keys to the primary key of **every** participating entity type; the foreign keys from the participants with **many cardinality** form the primary key; the relationship's own attributes may also contribute to that key; add its remaining simple attributes.

One relation body covers every non-excluded case, and only the underline moves. For a ternary `R` over `E1`, `E2` and `E3`, with own attributes `d1` and `d2`:

```
E1 [a1]

E2 [b1]

E3 [c1]

R [a1, b1, c1, d1, d2]
R.a1 references E1.a1
R.b1 references E2.b1
R.c1 references E3.c1
```

Cardinalities	Primary key of `R`
`N:M:P` — every participant many	All three foreign keys, `(a1, b1, c1)`
`N:M:1`	The two many-side foreign keys `(a1, b1)`; `c1` is a plain attribute
`N:M:1` with `d1` an underlined attribute of `R`	`(a1, b1, d1)` — the relationship's own key attribute joins them

Every participant is always represented, whatever its cardinality: a `1`-side foreign key is present but *outside* the key. For a ternary `K` over `H` (role `N`), `A` (role `M`) and `P` (role `1`), with own attribute `l`, that gives

```
K [b, i, q, l]
K.b references A.b
K.i references H.i
K.q references P.q
```

with only `b` and `i` underlined. The Company database contains no N-ary relationship — `WORKSON` is binary.

The excluded cases — 1:1:1 and N:1:1

These get **no new relation at all**. With at most one participant on a many side, there is no combination of foreign keys that would make a key, and the relationship can be absorbed exactly as in steps 3 and 4: one participating relation takes a foreign key to *each* of the others, plus the relationship's own attributes.

- **N:1:1** — the entity type on the `N` side is the one that takes them, and the choice is forced, since only its rows relate to exactly one row of each other participant:

```
E1 [a1, b1, c1, d1]
E1.b1 references E2.b1
E1.c1 references E3.c1

E2 [b1]

E3 [c1]
```

- **1:1:1** — nothing forces the direction, so *choose* a participant the same way step 3 does, preferring total participation, and put both foreign keys and the relationship's attributes there. The schema looks identical to the N:1:1 one above; only the reason for picking `E1` differs.

The giveaway that a question is testing this exclusion is a ternary diamond with at most one `N` on its edges: count the many sides before reaching for a new relation, because with fewer than two of them there is nothing to key it on.

Step +1 — Subclasses (EER)

For each subclass (specialisation-and-generalisation[31]): create a relation whose primary key **is** the superclass's primary key; add a foreign key from it back to the superclass relation; include the subclass's own simple attributes. The same rule works for every combination of total/partial and disjoint/overlapping specialisation.

```
Secretary [ssn, typingSpeed]
Secretary.ssn references Employee.ssn

Engineer [ssn, engineerType]
Engineer.ssn references Employee.ssn
```

A subclass with no attributes of its own still gets a relation — it is the row's *membership* in that subclass that is being recorded.

This step has no fixed position in the sequence. It is done straight after step 1 or 2 (whichever created the superclass), specifically so the subclass relations exist by the time a later relationship step needs to reference one — and relationships drawn to a subclass must reference the *subclass* relation, not the superclass.

Final schema — Company database

```
Employee [ssn, fName, mIt, lName, dob, address, sex, salary, dNumber, superSSN]
Employee.dNumber references Department.dNumber
Employee.superSSN references Employee.ssn

Department [dNumber, dName, mgrSSN, mgrStartDate]
Department.mgrSSN references Employee.ssn

Project [pNo, pName, pLocation, dNumber]
Project.dNumber references Department.dNumber

Dependent [ssn, depName, sex, dob, relationship]
Dependent.ssn references Employee.ssn

Secretary [ssn, typingSpeed]
Secretary.ssn references Employee.ssn

Engineer [ssn, engineerType]
```

[31] courses/infs1200/specialisation-and-generalisation.pdf

```
Engineer.ssn references Employee.ssn

WorksOn [ssn, pNo, hours]
WorksOn.ssn references Employee.ssn
WorksOn.pNo references Project.pNo

DeptLocs [dNumber, location]
DeptLocs.dNumber references Department.dNumber
```

Eight relations: the three strong entity types became three, the weak entity one, the two subclasses two, the M:N relationship one, and the multivalued attribute one — while all four of the 1:1 and 1:N relationships (`MANAGES`, `WORKSFOR`, `CONTROLS`, `SUPERVISION`) became nothing more than foreign key columns on relations that already existed.

Second worked example — Bank database

The other complete run-through is the bank: a `BANK` (unique `Code`, `Name`, `HOAddr`) has several branches; a `BANK-BRANCH` has a `BranchNo` unique only within its bank, so it is **weak**. One branch has many `ACCOUNT`s (`ACNo`, `Type`, `Balance`) and many `LOAN`s (`LoanNo`, `Type`, `Amount`), and every account and loan has one or more holders drawn from `CUSTOMER` (`SSN`, `Name`, `Addr`) — two M:N relationships, `AHOLDER` and `LHOLDER`.

```
Bank [code, name, hoAddr]

Branch [bankCode, branchNo, addr]
Branch.bankCode references Bank.code

Account [acNo, type, balance, bankCode, branchNo]
Account.{bankCode, branchNo} references Branch.{bankCode, branchNo}

Loan [loanNo, type, amount, bankCode, branchNo]
Loan.{bankCode, branchNo} references Branch.{bankCode, branchNo}

Customer [ssn, name, address]

AccountHolder [acNo, ssn]
AccountHolder.acNo references Account.acNo
AccountHolder.ssn references Customer.ssn

LoanHolder [loanNo, ssn]
LoanHolder.loanNo references Loan.loanNo
LoanHolder.ssn references Customer.ssn
```

Two things are worth reading off this schema. First, a weak entity's composite key propagates: because `Branch` is keyed on `(bankCode, branchNo)`, every 1:N foreign key drawn *from* `Branch` is composite too. Second, it propagates as a plain column — `Account` has its own key `acNo`, so `bankCode` and `branchNo` sit outside `Account`'s primary key, exactly as step 4 requires. `AccountHolder` and `LoanHolder`, by contrast, have no key of their own and are keyed on their pair of foreign keys.

The result is a logical schema, ready to be implemented with sql-ddl[32] and checked for redundancy with normal-forms[33]. Whether each inherited foreign key is *part of* its relation's primary key — the question that decides most of the answers above — is tabulated in foreign-keys[34].

Relational Mapping Notation

Reference legend + course style guide for writing relational schemas — introduced across 2026-03-09-the-relational-model-and-integrity-constraints[35] and 2026-03-16-er-to-relational-mapping[36]. These notations are specific to INFS1200/7900 and may differ from textbooks and online resources, so they are what to follow in an assignment or exam. The guide covers *naming and layout only*: it recommends nothing about which relations to produce. The mapping steps themselves are in er-to-relational-mapping[37].

Schema notation

- `Relation [attr1, attr2, ...]` — a relation schema; the (primary) key attribute(s) are underlined.
- A composite primary key gets a single continuous underline spanning all of its attributes, e.g. `Enrolment [studentId, courseCode, sem, year]`.
- `Relation.fk references OtherRelation.pk` — a foreign key constraint, listed directly under the relation it belongs to.
- `Relation.{fk1, fk2} references OtherRelation.{pk1, pk2}` — a composite foreign key referencing a composite primary key.

Naming convention (INFS1200/7900 style guide)

- **Table names**: UpperCamelCase (first letter of each word capitalised, no spaces) — e.g. `Flight`, not `FLIGHT`.

[32] courses/infs1200/sql-ddl.pdf
[33] courses/infs1200/normal-forms.pdf
[34] courses/infs1200/foreign-keys.pdf
[35] courses/infs1200/2026-03-09-the-relational-model-and-integrity-constraints.pdf
[36] courses/infs1200/2026-03-16-er-to-relational-mapping.pdf
[37] courses/infs1200/er-to-relational-mapping.pdf

- **Attribute names**: lowerCamelCase (first letter of each word from the *second* word onwards capitalised) — e.g. `departureTime`, not `Departure Time`. Acronym attribute names stay entirely lowercase (e.g. `eta`, not `ETA`).
- A space separates the table name from the opening bracket: `Flight [planeNumber, ...]`, not `Flight[planeNumber, ...]`.

The guide's compliant and non-compliant pair puts every rule in one line each:

```
Flight [planeNumber, departureTime, duration, eta]

FLIGHT[planeNumber, departure time, Duration, ETA]
```

The second breaks five rules at once — shouted table name, no space before the bracket, a space inside an attribute name, a capitalised attribute, and an upper-case acronym.

Layout convention

A table's foreign key constraint lines go directly underneath that table's own definition, with a blank line before the next table starts — not all grouped together under a separate "Foreign Keys:" heading at the end:

```
Employee [ssn, firstName, lastName, dob, manager]
Employee.manager references Employee.ssn

Dependant [ssn, name, dob]
Dependant.ssn references Employee.ssn

DependantPhoneNumber [ssn, name, phoneNumber]
DependantPhoneNumber.{ssn, name} references Dependant.{ssn, name}
```