

From Data Model to Relational Database Design

- We start with use cases to describe the basic requirements of a problem and develop an initial data model.
- By looking carefully at the details of the model, we are able to develop questions to help understand further subtleties and complexities of the real-world problem.
- We then look at a number of situations that occur in many models in the hope that these would be useful when difficult situations arose in other contexts.
- The goal is not to attempt to get a perfect or complete model.
- The outcome we are seeking is agreement on a model that accurately reflects the essential requirements of the real-world problem.
- This will involve numerous iterations as the use cases adjust to reflect the improved understanding and the changing scope.
- Having arrived at a set of use cases and a data model with which everyone is comfortable, we can now move on to the third phase of the development process.

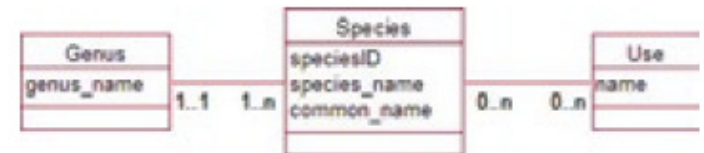
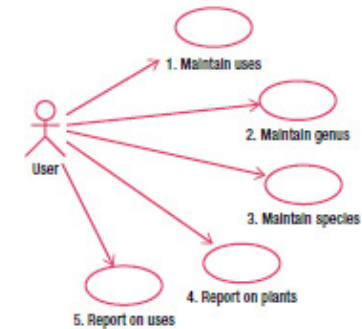
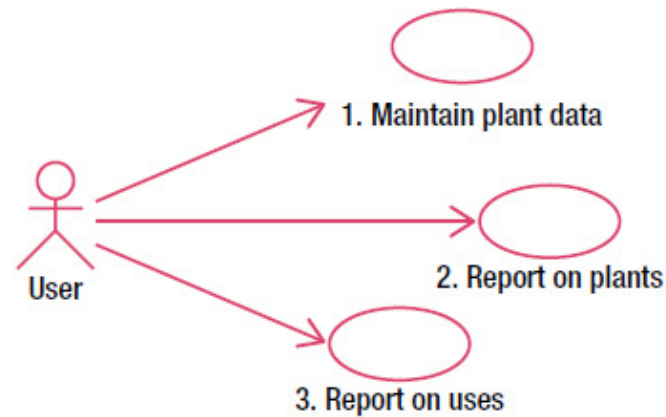
analysis



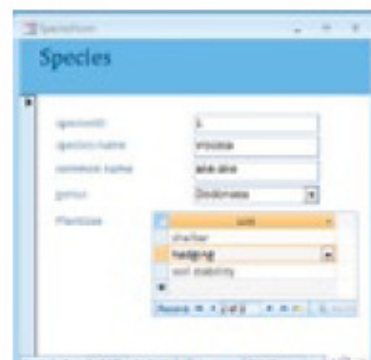
Real world

Abstract world

Problem



Solution



| PlantUse | | | |
|---------------------|----|-------|--------------|
| Use | ID | Genus | Species Name |
| 1. Maintain uses | 1 | 1 | 1 |
| 2. Maintain genus | 2 | 2 | 2 |
| 3. Maintain species | 3 | 3 | 3 |
| 4. Report on plants | 4 | 4 | 4 |
| 5. Report on uses | 5 | 5 | 5 |



Database development process

Representing the Model

- There is a great deal of trouble to capture as much detail as possible in the data model.
- Much of this detail can be represented and enforced by standard techniques built into relational database management software.
- A good model, implemented using the standard techniques, allows us to capture many of the constraints implied by the relationships between classes without recourse to programming or complex interface design.
- We will discuss how many of the aspects of the data model can be captured by standard database functionality.

Techniques to Represent Aspects of the Data Model

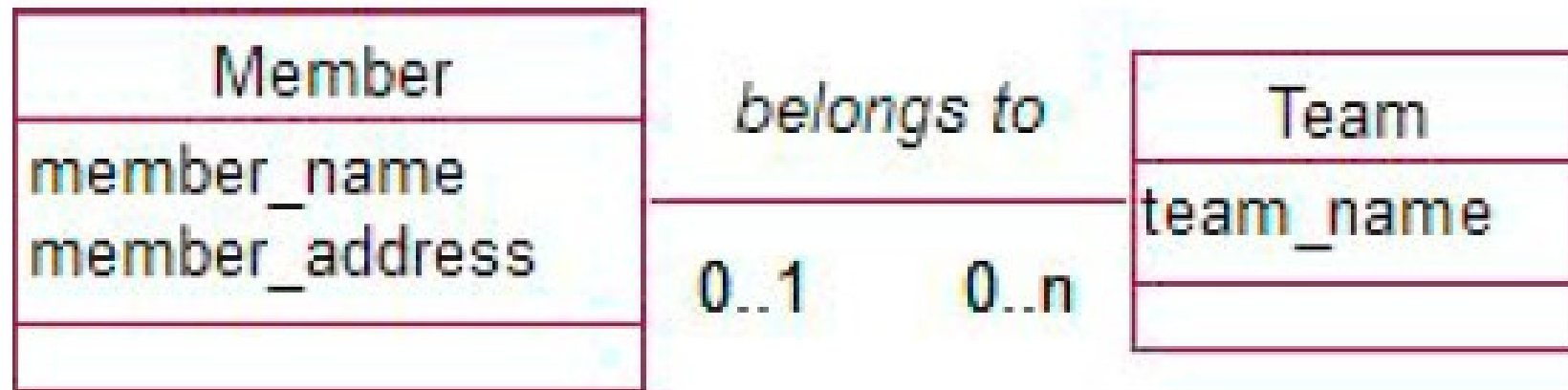
| Feature in Model | Technique Used in Relational Database |
|---|--|
| Class | Add a table with a primary key. |
| Attribute | Add a field with an appropriate data type to the table. |
| Object | Add a row of data to the table. |
| 1-Many relationship | Use a foreign key, i.e., a reference to a particular row (or object) in the table at the 1 end of the relationship. |
| Many-Many relationship | Add a new table with two foreign keys. |
| Optionality of 1 at the 1 end of a relationship | Make the value of the foreign key required. |
| Parent and child classes (inheritance) | Add a table for the parent class. Add tables for each child class with a primary key that is also a foreign key referencing the parent table (not an exact representation but OK). |

All of the techniques described above can be carried out in most database management products as part of the specification of the tables.

More complex constraints may require some additional procedures or checking at data input time, but with a good model this can be minimized.

By using the built-in facilities of the database product, the time required for implementation, maintenance, and expansion of the application is greatly reduced.

Representing Classes and Attributes



Part of data model for members and teams

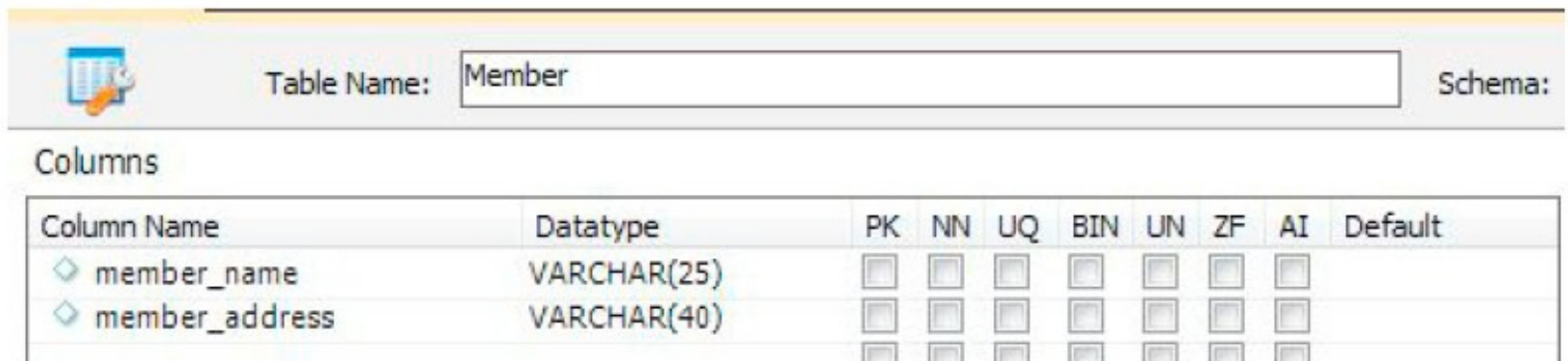
The first step is to design a database table for each class.

The attributes of the class will become the field or column names of the table, and when the data is added, each row, or record, in the table will represent an object.

Creating a Table

Standard SQL Command to Create a Customer Table with Two Fields

```
CREATE TABLE Member (  
  member_name VARCHAR(25),  
  member_address VARCHAR(40)  
)
```



The image shows the MySQL Workbench Table Creation Wizard. At the top, there is a 'Table Name' field containing 'Member' and a 'Schema' field. Below this, the 'Columns' section is visible, showing a table with two columns: 'member_name' and 'member_address'. Both columns are of type 'VARCHAR' with lengths of 25 and 40 respectively. The table has columns for various constraints: PK, NN, UQ, BIN, UN, ZF, AI, and Default.

| Column Name | Datatype | PK | NN | UQ | BIN | UN | ZF | AI | Default |
|----------------|-------------|--------------------------|--------------------------|--------------------------|--------------------------|--------------------------|--------------------------|--------------------------|---------|
| member_name | VARCHAR(25) | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | |
| member_address | VARCHAR(40) | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | |

Creating a Member table in MySQL Workbench

Once the table has been created, we can enter data

SQL Command for Entering a Record into the Member Table

```
INSERT INTO Member (name, address)  
VALUES ('Green, Ruth' '36 Some Street, Christchurch' )
```

| member_name ▾ | member_address ▾ |
|---------------|-----------------------------|
| Green, Ruth | 36 Some Street Christchurch |

Entering data through the MS Access interface

Choosing Data Types

- Each attribute in a class becomes a field or a column in a table.
- When we create the table, we need to provide a name for the field (e.g., name, address) and specify the type of data that will be stored in that field.
- Database products often offer a bewildering number of different data types, but they basically fall into the following groups:
 - ▣ Character types
 - ▣ Integer types
 - ▣ Numbers with a fractional part
 - ▣ Dates:

Character types:



- ❑ These allow you to enter any combination of characters-numbers, letters, and punctuation.
- ❑ They are used for names, addresses, descriptions, and so on.
- ❑ You usually need to provide a maximum length for the data going into the field.
- ❑ In SQL, a type of VARCHAR(60) would allow you to enter any number of characters up to 60.
- ❑ If you have very large amounts of text (notes, discussions, and so on), you might like to look at other types (e.g., Text in SQL)

Integer types:




- ❑ These types are for entering numbers with no fractional part. They are great for ID numbers such as customer numbers and for anything that you can count.
- ❑ Database systems often provide different-sized integer types (long, short, byte, etc.) that have different maximum numbers that can be entered.
- ❑ Unless you have particular performance problems or extremely large amounts of data, you will probably be fine if you use the ordinary integer type (INT in SQL).
- ❑ Just check that the biggest number it can handle is large enough for your data.

Numbers with a fractional part:

- These are used for things that you measure (heights, weights, etc.) and also for numbers that result from calculations such as averages.
- Most of the time you will be just fine with what is called a float or single (depending on the product you are using). Other types exist if you need particularly accurate measurements or calculations.
- One situation when a float may not be suitable is when you need to record rather large amounts of money accurately.
- Many products now provide money or currency types for this situation, or you may find the type is called something like fixed-length decimal.
- These types enable you to have many significant figures so that you can accurately keep track of your billions, down to a fraction of a cent!

Dates:

- 
- ❑ No prizes for guessing the type of data you can put in fields with these types.
 - ❑ If your product has different date types, some may allow you to include times and others may allow you to access dates further into the past or future.

Why is it important to get the correct data type?



- You could argue that since you can put anything in a character field, you can have character fields for everything.
- There are three main reasons why it is important to choose an appropriate data type for each of your fields.
 1. Constraints on the values
 2. Ordering
 3. Calculations

1. Constraints on the values



- ❑ A character field type has no constraints on what you can enter; however, most other fields do.
- ❑ Number fields won't allow you to accidentally mistype a number, say, by putting in an extra decimal point or a letter "O" instead of the number 0.
- ❑ Dates won't allow February 29 unless it is in a leap year, and so on.
- ❑ For this reason, phone numbers, which are likely to have extra symbols like () for area codes, need to be stored in character rather than number fields.

2. Ordering



- Different types of fields have different ways for comparing or ordering values.
- For example, character fields can be sorted alphabetically (A to Z), number fields numerically (small to large), and dates chronologically (older first).
- If you store numbers in character fields and then ask your product to sort them for you, you might get something like this: 10, 12, 123, 2, 200, 36.
- Dates in a character field might be sorted like this: August 1, 2012; February 1, 2012; May 4, 2012.

3. Calculations



- ❑ Your database product can do arithmetic and perform other functions on your data, but only if it is the correct type.
- ❑ For example, it will be able to add, multiply, and average numbers; figure out how many days fall between two dates; and look for particular characters in a piece of text.
- ❑ You need to have the correct types in order to take advantage of this functionality.
- ❑ And getting back to phone numbers, you never want to subtract them, average them, or order them numerically, so they can and should generally be stored in a character field type.

Domain and Constraints

- A domain is a set of values allowed for an attribute or field.
- For something like a product description it might be sufficient for the domain to be any set of characters up to some specified length.
- Other attributes might have more specific domains.
- For example,
 - ▣ a gender attribute might only allow the values “M” or “F”
 - ▣ a day of the week might be constrained to the characters “Mon,” “Tue,” “Wed,” “Thu,” “Fri,” “Sat,” and “Sun”
 - ▣ possible marks for an exam might be whole numbers between 0 and 100.

Constraint Vs. Domain




- ❑ Some database products (e.g., SQL Server) allow users to create their own domains or data types.
- ❑ For example, you might define the type ExamMark as an integer between 0 and 100. This user defined domain or type can then be used in all the tables in the database.
- ❑ Other products (e.g., Access) do not permit the creation of domains, but all products allow constraints to be declared on individual columns.
- ❑ For instance, we could declare gender as being a character type of length 1 with the constraint that it can only accept the values “M” or “F.”
- ❑ The difference between a constraint and a domain is that the former has to be specified in every table whereas the latter only needs to be declared once in the database.

Creating a table with a constraint on the values for a gender

```
CREATE TABLE Member (  
    member_name VARCHAR(25) ,  
    member_address VARCHAR(40) ,  
    gender VARCHAR(1) CHECK (gender='M' OR gender='F' )  
)
```

In MySQL, the **CHECK** clause is parsed but ignored by all storage engines.

One very important constraint is to specify whether a value is required or can be left empty



```
CREATE TABLE Member (  
    member_name VARCHAR(20) NOT NULL,  
    member_address VARCHAR(45),  
    gender VARCHAR(1) CHECK gender IN ('M', 'F')  
)
```

- It is reasonable to ask why we haven't insisted gender must always have a value as well.
- All members have a gender, after all. In general, there are two main reasons why we might need to put a null in a field:


why we may need to put a null in a field?

- either the field doesn't apply for a particular record (a person may or may not have a driver's license number)
- the field does apply, but at the moment we don't know the actual value.
- For the situation with gender then, clearly the value applies, but there could be situations where we do not know what it is.
- If we force a value to always be entered, we risk not being able to enter the record or having a distressed data entry operator taking a guess at a likely value.
- Consider a university administrator entering details from a stack of student applications, a couple of which have left the box for gender empty.
- The university would much rather have the student's information entered incompletely than not at all.
- At least then they can extract some fees and contact the person about the gender at a later time.

What about name—should that be allowed to be null?

- It is always a judgment call, but recording details about a nameless student is probably going to result in trouble somewhere down the line.
- Even if you think a value is going to be essential for the accuracy of your data, do not underestimate the likelihood that disallowing nulls might cause an incorrect value to be entered.

Checking Character Fields

- 
- Character fields are a bit different from other field types.
 - With a character field, we can enter anything we like (if there are no other constraints), and so it is possible to enter several values into a field.
 - Other fields such as numbers and dates only ever allow one value to be entered.

The member_name field as it stands could contain data about the first name, last name, possibly other names, initials, and titles.

To find the record shown below is going to be very difficult.

Will the user know whether to search for Mrs. Green, Mrs. Rose Green, Rose Green, Mrs. R. Green, or Ms. Green?

It is also going to be difficult to sort the records sensibly.

| member_name ▾ | member_address ▾ |
|---------------|-----------------------------|
| Green, Ruth | 36 Some Street Christchurch |

Separating the data into fields makes the data much more useful

- We usually want to order people by last names, and this is not going to be possible with the way we are recording the data in the previous slide.
- The way the address data is being recorded is going to make it difficult to select records by city or print nicely formatted address labels easily.
- A good rule of thumb is that any data that you are likely to want to search for, sort by, or extract in some way should be in a field all by itself.

| last_name ▾ | first_name ▾ | title ▾ | street_address ▾ | city ▾ | post_code ▾ |
|-------------|--------------|---------|----------------------|--------------|-------------|
| Turner | John | Mr | Flat 1, 6 Moa Street | Christchurch | 8033 |
| Green | Ruth | Mrs | 36 Some Street | Christchurch | 8041 |

Improved fields to describe a customer

Accuracy of values

- If the accuracy of values in a field is really crucial to the project, maybe that particular piece of information should actually be in a class of its own.
- In the previous slide, we might ask how important it is for the values in the city field to be accurately recorded.
- If accuracy is essential (e.g., we regularly want to target advertising to customers in a particular city), we may need two classes, City and Customer, with a 1–Many relationship between them.
- If we only want the address for sending general mail, accuracy isn't so important, as the postman can probably cope with the odd misspelling.

Primary Key

- We have thus far overlooked one constraint that is so important that it gets a section all to itself. This involves choosing a primary key for the table.
- It is imperative that we can always find a particular object (or row or record in a table).
- This means that all records must be unique; otherwise, you can't distinguish between two that are identical.
- Consider the consequences of two identical records: when a member pays his annual fee, we need to connect that payment to the member somehow. What if we have two identical rows in our Member table for Ruth
- Smith? How will we know which row is associated with a payment of fees?
- If we get it wrong, then one Mrs. Smith will be pretty upset when she receives another invoice.
- **Every member needs to be able to be uniquely identified.**
- **There must never be two identical records in any of our tables.**

Without a primary key, it is impossible to uniquely identify a row in a table

- This table clearly contains information about users and their respective ages.
- However, there are four users with the same.
- There is probably a way of distinguishing them somewhere else in the system—perhaps by an e-mail address or a user number.
- But if, for example, you want to know the ages of dminter with user ID 32, there is no way to obtain it from the table above.

| | |
|----------|----|
| dminter | 35 |
| dminter | 40 |
| dminter | 55 |
| dminter | 40 |
| jlinwood | 57 |

Updating an Ambiguous Table

```
UPDATE users SET Age=10 WHERE User='dcminter'
```

| User | Age |
|----------|-----|
| dcminter | 30 |
| dcminter | 42 |

Which of the following should be contained in the resulting table?

- A single row for the user dcminter, with the age set to 10
- Two rows for the user, with both ages set to 10
- Two rows for the user, with one age set to 10 and the other to 42
- Two rows for the user, with one age set to 10 and the other to 30
- Three rows for the user, with one age set to 10 and the others to 30 and 42

Determining a Primary Key


- A key is a field, or combination of fields, that is guaranteed to have a unique value for every record in the table.
- It is possible to learn quite a bit about a problem by considering which fields are likely to be possible keys.
- We will see later that there can be more than one set of fields that can have unique values in a table.
- We choose one of these to be the primary key and then use that to enable us to identify records uniquely.
- Consider which fields could be keys for the following table where the names of the fields are given in parentheses after the table name:
Member (name, address, phone, birth_date)


Primary Key



- How about name for a key?
 - ▣ No; it is entirely possible that we may have two members with the same name, and we will need to be able to distinguish them.
- What about the combination (name, address)?
 - ▣ This is more promising, but then dads have been known to name their sons after themselves, and it is not improbable that they may at various times of their lives share the same address and be members of the same club.

A potential key must be guaranteed to be unique for every possible record

- 
- In cases like our Member table, there is not much choice but to add a new attribute or field such as member_number and then assign all members their own unique numbers so we can distinguish them.
 - This is sometimes called a surrogate key.
 - In real life, we can always distinguish individuals, but when we look at the data we are storing about them, we may not be able to find a unique set of values.
 - In many cases, privacy laws prevent information such as social security or tax numbers being used to identify people, so each business or organization is often compelled to provide its own personal identification number.

- 
- When we create a table in our database, we can specify the field which is to be the primary key of the table. The SQL to do this is shown below.
 - Most database products also usually provide you with an interface to help create a table and select the field(s) that make up the primary key.

```
CREATE TABLE Member (  
    member_number INT PRIMARY KEY,  
    member_name VARCHAR (25)  
)
```

- With the primary key field specified, a constraint is put on the table that will ensure that every record must have a unique value for member_number.
- The user will never be able to put in two records with the same value for member_number, and so every member in our table can be uniquely distinguished.
- The constraint also ensures that the primary key field always has a value, so every record is certain to have a value for member_number.

Generating Unique Values



- It is possible to get the database to automatically generate unique values for fields like `member_number`.
- Depending on the product you are using, you will find a field type called `identity`, `auto_increment`, `autonumber`, or something similar.
- You can then specify some starting number and a step size, and every new row entered into the table will automatically be assigned the next available number.

Concatenated Keys

It isn't always necessary or even advisable to introduce a new automatically incrementing number field into a table to act as a primary key. With the case of members, there was no other way to ensure a unique field for every record, but often a unique field or combination of fields already exists in the table. When we have a combination of fields that can uniquely identify a record, this is referred to as a *concatenated* or *composite* key. Thinking about which combinations of fields are possible keys can help you discover and understand subtleties of the problem. Here is an example.

What is a possible key for the table in Figure 7-7, which is keeping information about students enrolling in courses?

| student ▼ | course ▼ | year ▼ | grade ▼ |
|-----------|----------|--------|---------|
| 13887 | COMP101 | 2011 | B |
| 17625 | COMP101 | 2011 | E |
| 17625 | COMP102 | 2012 | A |
| 18574 | COMP102 | 2012 | B |

Figure 7-7. *Enrollments table*

student will not be suitable as a key, as a student will have a record for each of the courses in which he enrolls (we can see that the value 17625 appears in at least two records). Similarly, course will not do, as a course will have many enrollments each with its own record (the value COMP102 is duplicated). In fact, every column has duplicated values, so no single field is suitable as a key.

What about the combination (student, course)? In the few records shown in Figure 7-7, this combination is always unique, but we have to be sure this will *always* be the case for every record we may need to enter. We need to find out a bit more about the problem.

Whenever we are checking the suitability of a combination of fields as a key, we need to find a question that checks that the combination will always be unique. In this case, we needed to ask questions such as the following:

Is a student ever likely to enroll in a course more than once?

Yes. (student, course) is not a suitable key.

Is it possible that a student will enroll in the same course more than once in a single year?

Yes. (student, course, year) is not a suitable key.

Is it possible that a student will enroll in the same course more than once in a single semester of a given year?

No. (student, course, year, semester) is a possible key.

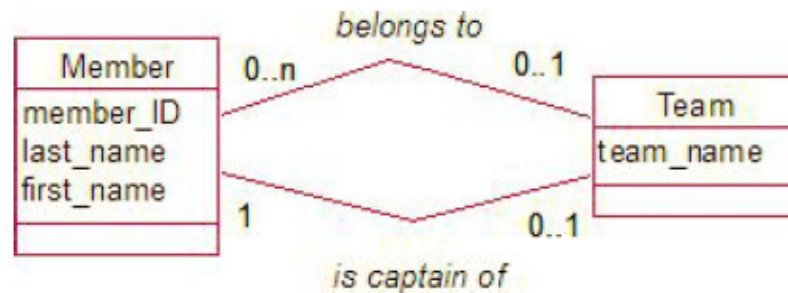
Now look at the other fields in the table:

Is it possible that a student might need to have more than one grade for a given enrollment (e.g., an initial grade and a revised grade)?

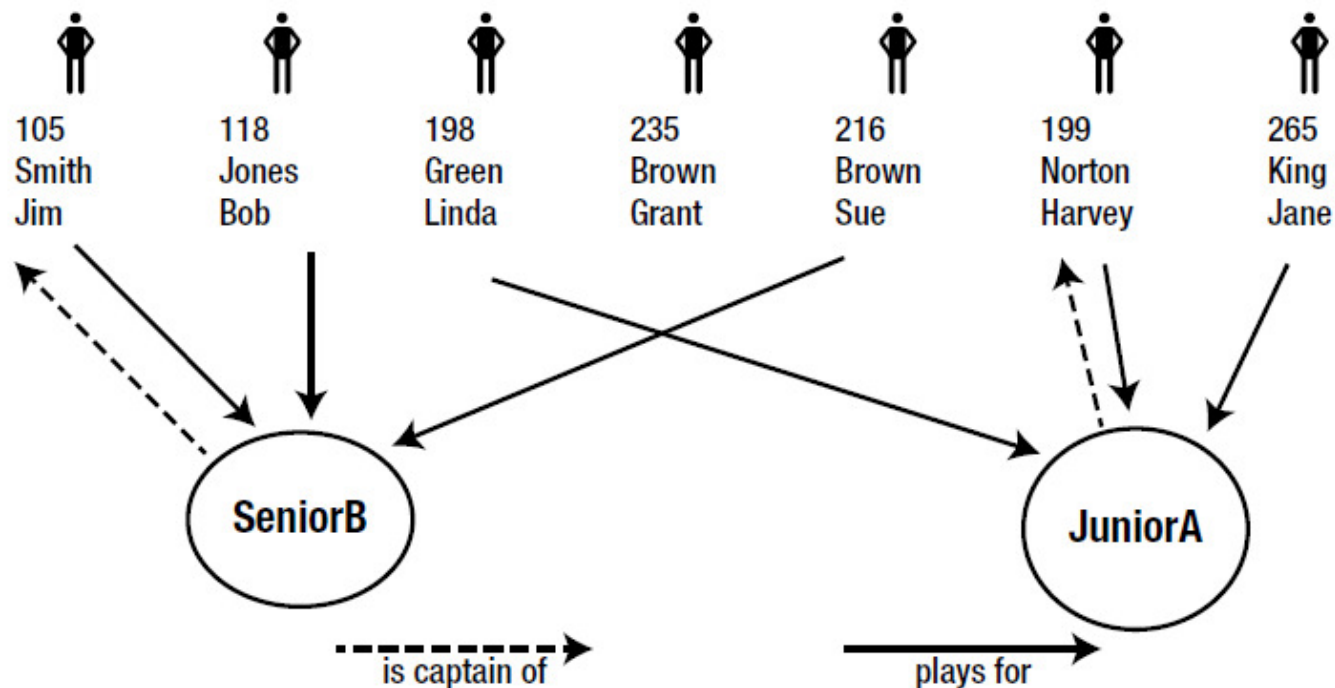
Maybe. (In that case, the problem is much more complicated than we thought.)

Would it all have been easier if we had just abandoned looking for a concatenated key and added an automatically generated enrollment_number field that could be guaranteed to be always unique?

Representing Relationships



Sports club data model



Members, teams, and instances of the relationships between them

Each of the objects in the figure will be a row in the appropriate table

- First we design two tables to represent the classes and choose a primary key for each.
 - ▣ Member: (member_ID, last_name, first_name)
 - ▣ Team: (team_name)
- To represent the relationships *plays for* and *is captain of*, we need a way of specifying each of the lines between the objects in the figure.
- For example, we need to show that Bob Jones plays for SeniorB, and the captain of JuniorA is Harvey Norton.
- As we have primary keys established, we can easily identify the row associated with each object (e.g., Harvey Norton is the row in the Member table where the primary key field member_ID has the value 199).
- To represent the relationship between the objects, we use these key values by way of a *foreign key*.

Foreign Keys

- The figure shows the two tables Member and Team again, but now we have added a field to show who is the captain of each team.
- What we have done is put a new field in the Team table (captain) that will contain the key value of the member who is its captain. This is a foreign key.
- A foreign key is a field(s) (in this case captain) that refers to the primary key field(s) in some other table (in this case it contains a value of the key field member_ID from the table Member).
- In this way, we establish the relationships between objects of different classes.

| member_ID | last_name | first_name |
|-----------|-----------|------------|
| 105 | Smith | Jim |
| 118 | Jones | Bob |
| 198 | Green | Linda |
| 199 | Norton | Harvey |
| 216 | Brown | Sue |
| 235 | Brown | Grant |
| 265 | King | Jane |

Member table

| team_name | captain |
|-----------|---------|
| JuniorA | 199 |
| SeniorB | 105 |

Team table



The SQL statement for creating the table `Team` with a foreign key referring to the `Member` table is shown in Listing 7-6. Many products also provide a diagrammatic interface for specifying foreign keys. The interface for setting up a foreign key in Access is shown in Figure 7-11.

Listing 7-6. *SQL to Create a Team Table with a Foreign Key*

```
CREATE TABLE Team (  
  team_name VARCHAR(10) PRIMARY KEY,  
  captain INT FOREIGN KEY REFERENCES Member  
)
```

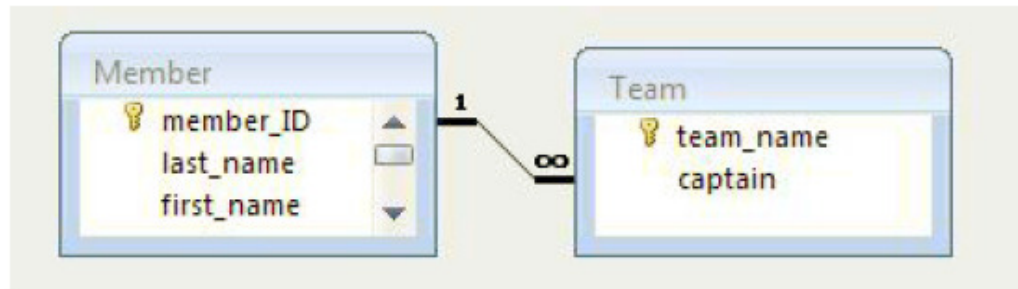


Figure 7-11. *Access interface for specifying captain is a foreign key referring to the Member table*

The two fields `member_ID` and `captain` will both have values from the same domain, that is, the set of MemberIDs. Formally, a foreign key and the primary key of the table it references must have the same domain. In most database products this is softened to requiring them to be the same data type or a compatible data type. The types of data that are regarded as compatible will depend on the database software being used (e.g., character fields of different lengths are compatible in some products, but not in others).

Referential Integrity

- Arm-in-arm with the idea of a foreign key is the concept of referential integrity.
- This is a constraint that says that each value in a foreign key field (i.e., 199 and 105 in the Team table in Figure) must exist as values in the primary key field of the table being referred to (i.e., 199 and 105 must exist as values in the member_ID field in Member).
- This prevents us putting a nonexistent member (say, 765) as the captain of a team.
- It also means that we cannot remove members 199 and 105 from our member table while they are captains of teams. As soon as you set up a foreign key, this referential integrity constraint is automatically taken care of for you.

Representing 1-Many Relationships

In the previous sections, you have seen how it is possible to represent instances of a relationship in the data model by using a foreign key. In general, the process for a 1-Many relationship is as follows:

For a 1-Many relationship, the key field from the table representing the class at the 1 end is added as a foreign key in the table representing the class at the Many end.

We have already represented the relationship *is captain of* in Figure 7-8. Let's now use our general guideline to do the same thing for the relationship *plays for* between Member and Team.

The class at the 1 end is Team, so we take the primary key field from the Team table and add it as a new foreign key attribute in the Member table. We can give the field any name we like, but it should clearly indicate the relationship it is representing, for example `current_team`. This is shown in Figure 7-12.

| member_ID | last_name | first_name | current_team |
|-----------|-----------|------------|--------------|
| 118 | Jones | Bob | SeniorB |
| 198 | Green | Linda | JuniorA |
| 199 | Norton | Harvey | JuniorA |
| 216 | Brown | Sue | SeniorB |
| 235 | Brown | Grant | |
| 265 | King | Jane | JuniorA |

Member table


`current_team` is a foreign key referencing Team

| team_name | captain |
|-----------|---------|
| JuniorA | 199 |
| SeniorB | 105 |


Team table

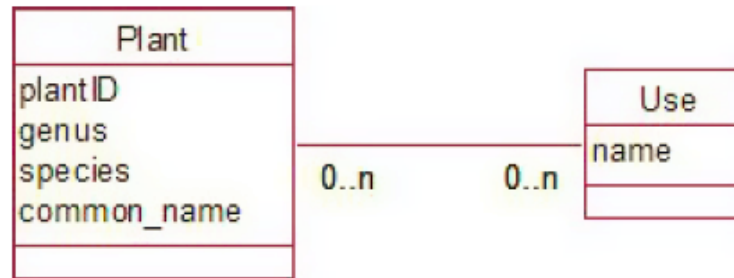
`captain` is a foreign key
referencing Member

Figure 7-12. Both relationships in sports club model represented by foreign keys

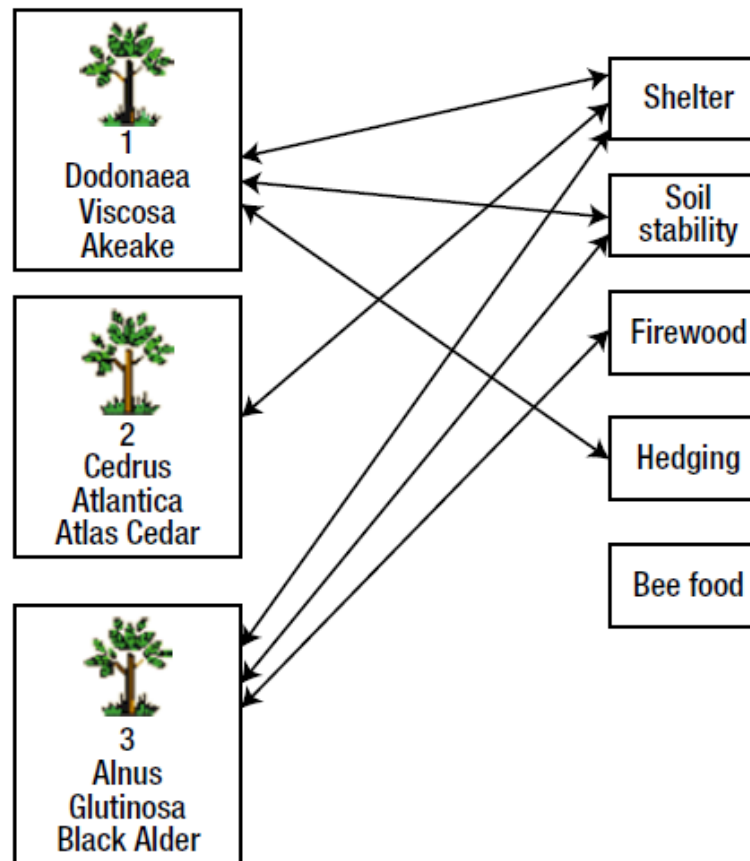
- 
- Referential integrity, which is a result of making the `current_team` field a foreign key, will ensure that the value entered in `current_team` can be found in the primary column (`team_name`) of the Team table.
 - This ensures that members can only play for teams that already exist in the Team table.
 - Note that a foreign key field can be null.
 - Grant Brown does not belong to a team, so there is no value in the foreign key field in his record.
 - This is consistent with the optionality of the *plays for* relationship.
 - If the relationship was not optional, we would have to impose an additional constraint on the field to say that nulls were not permitted.
 - If we wanted to ensure that every team had a captain (as the data model suggests), then as well as making the captain field in the Team table a foreign key, we would also specify that it cannot ever be null.

Representing Many–Many Relationships

- 
- You may remember that Many–Many relationships are not as common as you might at first expect.
 - Often they are a sign that some information about the problem has been initially overlooked, and an intermediate class is required to store that information.
 - They do, however, genuinely occur where we have objects that simultaneously belong in many categories.



Data model for plant database

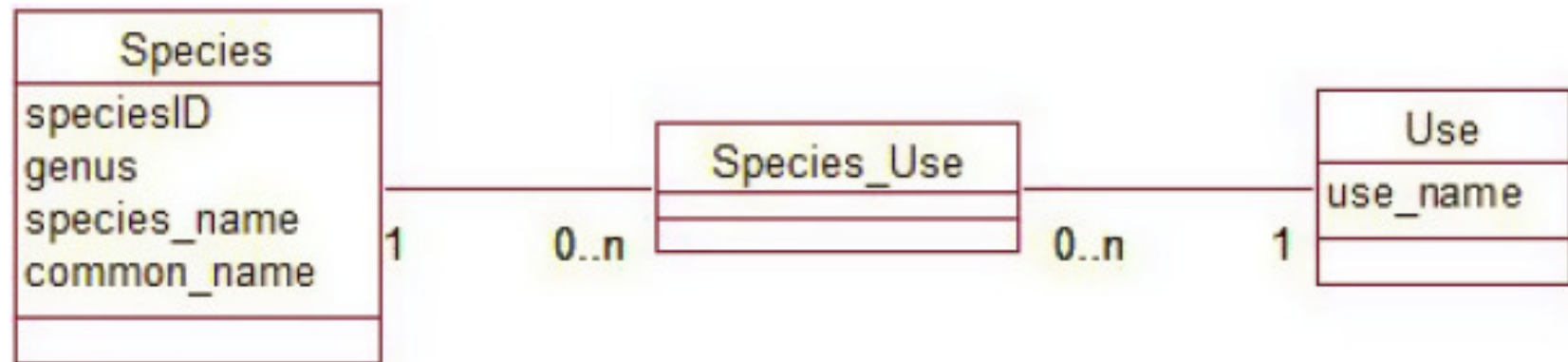


Some examples of species and their uses

How are we to represent all the instances of this relationship?

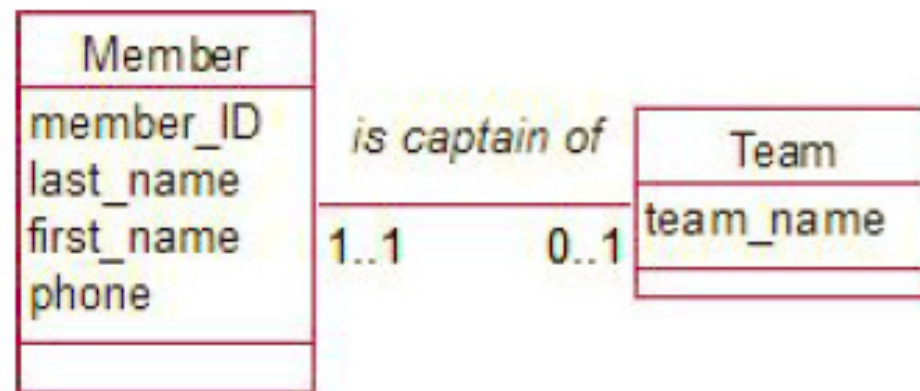
- Foreign keys will no longer do the trick, as we will never know how many uses a particular species will have, nor how many species will be related to a particular use.
- To deal with this in a relational database, we have to introduce a new intermediate class in our data model.
- In this Many–Many situation, the new intermediate class will not have any attributes, as there is nothing we wish to know about a particular combination of Species and Use.
- We use the new class (Species_Use) simply to store all the relevant pairings of Use and Species.

Adding another class to represent a Many–Many relationship in a relational DB



Representing 1–1 Relationships

- In all of the previous sections, we have always ended up taking the primary key field at the 1 end of the relationship and using it as a foreign key in the table at the other end.
- If both ends of the relationship have a cardinality of 1, which way around should we do this?
- Our example of members and teams had a 1–1 relationship: *is captain of*.
- That part of the data model is shown below.



The question is whether to put `member_ID` as a foreign key in the `Team` table or `team_name` as a foreign key in the `Member` table. The resulting tables for these alternatives are shown in Figure 7-21.

| member_ID | last_name | first_name | is_captain_of |
|-----------|-----------|------------|---------------|
| 105 | Smith | Jim | SeniorB |
| 118 | Jones | Bob | |
| 198 | Green | Linda | |
| 199 | Norton | Harvey | JuniorA |
| 216 | Brown | Sue | |
| 235 | Brown | Grant | |
| 265 | King | Jane | |

Foreign key in Member table

| team_name | captain |
|-----------|---------|
| JuniorA | 199 |
| SeniorB | 105 |

OR

Foreign key in Team table

Figure 7-21. Alternative ways to represent the 1-1 relationship

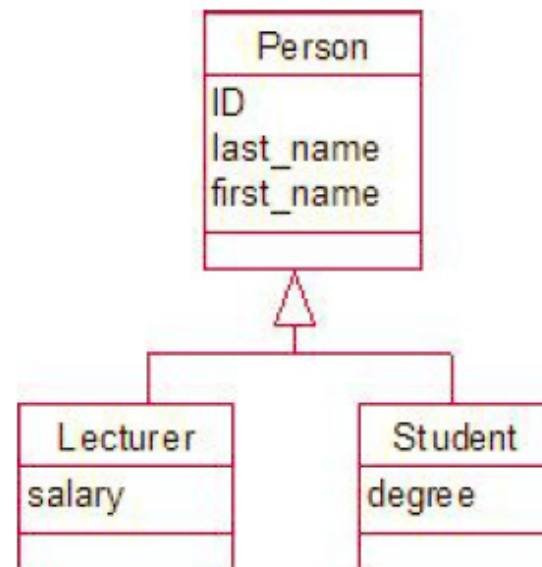
The same information is represented in both tables. We mustn't do both simultaneously, as we might end up with inconsistent data. For example, we could end up with Bob being captain of SeniorB according to the `Member` table, but Jim being the captain according to the `Team` table.

In the `Member` table in Figure 7-21, we have many empty values for the field `is_captain_of` because that end of the relationship is optional (a member doesn't have to be a captain and most members won't be). In general, you should put the foreign key in the table that has the compulsory association if there is one. A team *must* have a captain, so put the foreign key in the `Team` table.

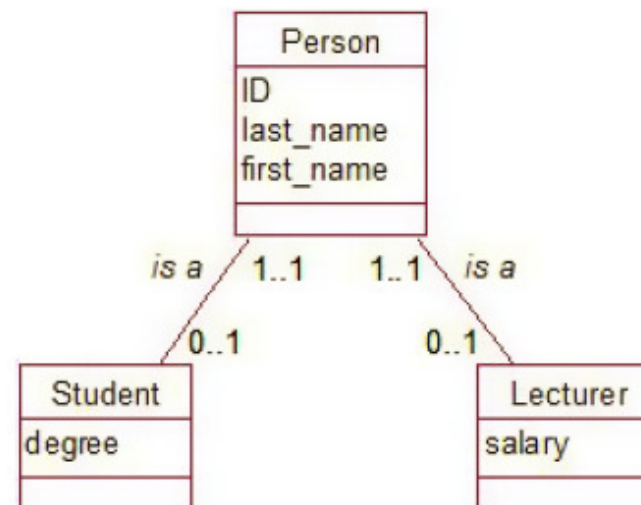
Putting the foreign key in the `Team` table ensures that each team only has one captain, however there is nothing to prevent a member being captain of more than one team (e.g., there is nothing currently in the design of the `Team` table to prevent us putting 199 on more than one row). With the foreign key in the `Member` table we have the opposite situation: a person can only captain one team but we could have JuniorA appearing on several rows, and so effectively having several captains.

Representing Inheritance

- Relational databases do not have the concept of inheritance built into them; however, it is possible to approximate the idea of inheritance.
- Following figure shows a simple case of inheritance in which lecturers and students inherit the attributes of a person and also have some specialized attributes of their own.
- One way to capture the main aspects of inheritance in a relational database is to set up classes for each parent class and subclass and include a 1–1 relationship between each subclass and its parent.
- The relationships (reading upward) say that a lecturer is a person and a student is a person, which is a natural way to think about the model.
- The relationship between Student and Person is compulsory at the top end because every student is a person, but optional at the bottom end because a person does not have to be a student.
- We can now set up tables as we did for the 1–1 relationship in the previous section.
- We choose to put the foreign key (personID) in the Student table (because a student has to be a person) and similarly for the Lecturer table.
- personID will also be the primary key of the Lecturer and the Student tables.



Simple model with inheritance



Inheritance approximated with 1-1 is a relationship

Tables representing the inheritance model

| ID | last_name | first_name |
|-----|-----------|------------|
| 101 | Jones | Sue |
| 108 | Brown | Lin |
| 110 | Li | Bo |
| 112 | Green | Mike |

Person table

| personID | salary |
|----------|--------|
| 101 | 75000 |
| 108 | 90000 |
| 110 | 12000 |

Lecturer table

| personID | degree |
|----------|---------|
| 108 | Arts |
| 110 | Science |
| 112 | Arts |

Student table