# 2. Creating a database

## Designing the database schema

As you have seen, once the data model for a system has been designed, you need to work out how to represent that model in a relational database. This representation is sometimes referred to as the **database schema**. In a relational database, the schema defines the **tables**, the **fields** in each table, and the **relationships** between fields and tables.

No matter how carefully the data model has been designed, it is almost always necessary to make modifications and refinements to turn it into a practical and efficient database.

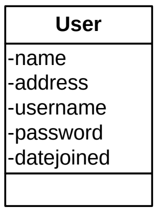
Once the schema has been designed, it can be implemented in the RDBMS.

**NOTE**

The process of designing a schema which can be implemented in a specific RDBMS is sometimes called **physical design**. This contrasts with **logical design**, which is the process which deals with gathering business requirements and converting those requirements into a model.

## Representing Classes, Attributes and Objects

The following class diagram shows the *User* entity in the GCUTours data model



So how do we represent these entities in a database? We need to design a **database table** for each class. A table is a set of data that is organized in horizontal **rows** and vertical **columns**. The columns are identified by names. Columns in a data table are very often called **fields**, and we will use that term from now on.

**NOTE**

Database designers often seem to have several words for the same thing, like *column* and *field*. Sometimes there are very slight differences between the exact meanings, but they are pretty much used interchangeably. Here is a list of some (nearly) equivalent database terms:

**Table** = **Relation** (in fact, a *relational* database is a database made up of *relations*)

**Column** = **Field**

**Row** = **Record** = **Tuple**

We need to design a table for the *User* class. Designing a table requires you to:

- provide a **name** for the table
- provide a **name** and **data type** for each field in the table

### Naming the table

It's quite common to give database tables plural names, so we'll call the table ***Users***. (You don't have to use this naming convention, but it is a useful way of making sure you know when you're talking about the class and when you mean the database table.)

## Defining the fields

Each attribute in the class becomes a field, and we usually match field names to attribute names. We also need to specify the type of data to be held in that field. The possible **data types** will depend on the RDBMS.

For example, the ***name*** field needs to hold a piece of text. We might also want to specify a maximum number of characters which can be held in the field. Similarly, ***address*** needs to hold text, and may need more characters than name.

In contrast, the ***datejoined*** field needs to represent a date.

Other common database data types include numbers (integers and numbers with fractional parts) and binary data (for example, images).

In Access, the fields and datatypes for the *Users* tables could be:

- ***name*** – Text, field size=25
- ***address*** – Text, field size= 100
- ***username*** – Text, field size=10
- ***password*** – Text, field size=10
- ***datejoined*** – Date/Time

The table design looks like this: the column headings are the field names, and there is a data row for each user in the database.

### Users

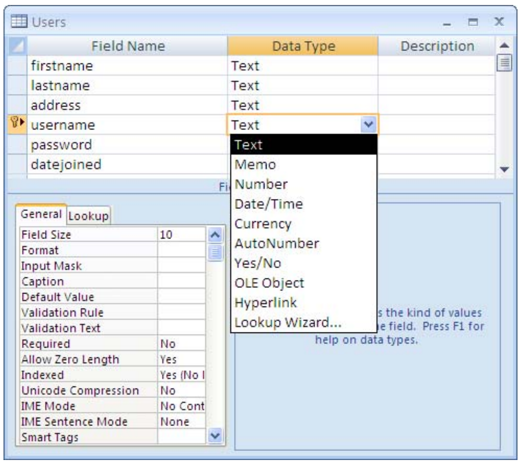
| name | address | username | password | datejoined |
|---|---|---|---|---|
| *Text data* | *Text data* | *Text data* | *Text data* | *Date/Time data* |
| *Text data* | *Text data* | *Text data* | *Text data* | *Date/Time data* |
| … | … | … | … | … |

## Table data

Each row of data in the table represents a single user. In the model of the system, each individual user is represented by an **object** which is an instance of the User class. A **database row** contains the data representing the **attribute values for an object**.

## Implementing the table

Access lets you create a new table and define the fields in **design view**, as shown below:



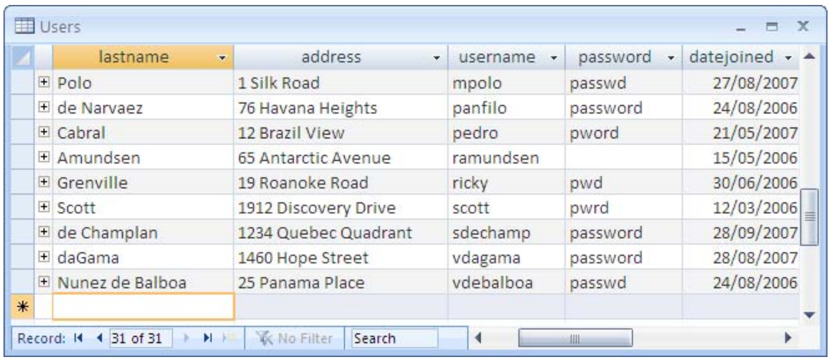
| Field Name | Data Type |
|---|---|
| firstname | Text |
| lastname | Text |
| address | Text |
| username | Text |
| password | |
| datejoined | |

Text
Memo
Number
Date/Time
Currency
AutoNumber
Yes/No
OLE Object
Hyperlink
Lookup Wizard...

| | |
|---|---|
| Field Size | 10 |
| Format | |
| Input Mask | |
| Caption | |
| Default Value | |
| Validation Rule | |
| Validation Text | |
| Required | No |
| Allow Zero Length | Yes |
| Indexed | Yes (No I |
| Unicode Compression | No |
| IME Mode | No Cont |
| IME Sentence Mode | None |
| Smart Tags | |

s the kind of values e field. Press F1 for help on data types.

You can then switch to **datasheet view** to see the data, and you can also enter new data:



| lastname | address | username | password | datejoined |
|---|---|---|---|---|
| Polo | 1 Silk Road | mpolo | passwd | 27/08/2007 |
| de Narvaez | 76 Havana Heights | panfilo | password | 24/08/2006 |
| Cabral | 12 Brazil View | pedro | pword | 21/05/2007 |
| Amundsen | 65 Antarctic Avenue | ramundsen | | 15/05/2006 |
| Grenville | 19 Roanoke Road | ricky | pwd | 30/06/2006 |
| Scott | 1912 Discovery Drive | scott | pwrd | 12/03/2006 |
| de Champlan | 1234 Quebec Quadrant | sdechamp | password | 28/09/2007 |
| daGama | 1460 Hope Street | vdagama | password | 28/08/2007 |
| Nunez de Balboa | 25 Panama Place | vdebalboa | passwd | 24/08/2006 |

**NOTE**

These notes will show you examples of tasks you can do in Access, but will not give detailed instructions on how to do these – you will find these details in your lab exercise sheets.

## Data types

The set of possible data types for a field will depend on the RDBMS you are using. The SQL standard specifies data types which are used by most RDBMSs, including JavaDB. Access data types are generally a bit different from the SQL standard ones. The table below shows some common data types (this is NOT a complete list):

| Type | Uses | Access type | SQL type |
|---|---|---|---|
| **Character** | Any characters, including numbers, letters and punctuation, Usually specify maximum length. | *Text*, can set Field Size (up to 255) *Memo* for longer text | ***VARCHAR*, e.g. VARCHAR(25)** |
| **Integer** | Numbers with no fractional part. Typically used for things that you can count. RDBMS often provides different sizes of integer | *Number*, can set Field Size to *Integer* or *Long Integer* (in Access SQL use *Number, Integer* or *Long)* | **INTEGER SMALLINT** |
| **Numbers with fractional part** | Typically used for things that you measure or calculate, or for amounts of money. | *Number*, can set Field Size to *Single*, *Double* or *Decimal* and specify Decimal Places *Currency* is used to handle large monetary amounts accurately | ***NUMERIC*, can set maximum number of digits and digits after the decimal point, e.g. NUMERIC(5,2)** |
| **Dates and times** | Special formats for storing date and time values. Can be used in calculations, e.g "how long since…" | *Date/Time* (in Access SQL use *DateTime*) | ***DATE*** ***TIME*** ***TIMESTAMP*** **(combines a DATE and a TIME)** |

### Why are data types important?

A character field can contain pretty much anything – words, numbers, even dates. So why use any other data types? Here are three advantages of using well-chosen data types:

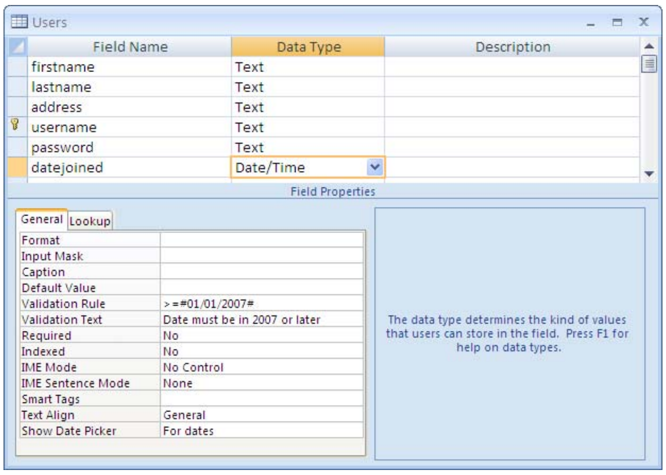
- **Constraining data** – for example, if you want to make sure that a field can **only** contain numbers, then using a numeric type means that the database will check all data entered in the field, and will prevent non-numeric data being stored. Similarly, date fields will only allow valid dates to be entered.

- **Ordering** – you might want to sort or order your data, and numbers or dates in character fields will probably not be sorted in the order you would want – can you think why?

- **Calculations** – you may want to do things like add up a field of numbers to get a total, or figure out how many days between two dates. These calculations will only work properly if the data type is correct.

## Additional constraints

Sometimes you might want to prevent data being entered that is valid as far as the field data type is concerned. For example, you might want to make sure that the value of *datejoined* in the *Users* table is in 2007 or later. You can do this by adding a specific constraint on the field. In Access design view you can specify a **Validation Rule**, and **Validation Text** to display if invalid date is entered:


Users

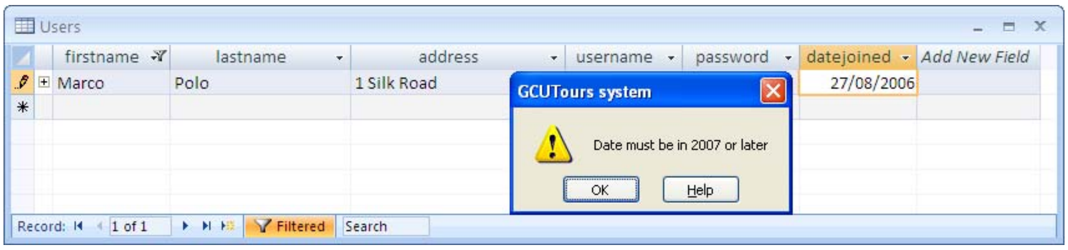
| Field Name | Data Type | Description |
|---|---|---|
| firstname | Text | |
| lastname | Text | |
| address | Text | |
| username | Text | |
| password | Text | |
| datejoined | Date/Time | |

Field Properties

General | Lookup

| | |
|---|---|
| Format | |
| Input Mask | |
| Caption | |
| Default Value | |
| Validation Rule | >=#01/01/2007# |
| Validation Text | Date must be in 2007 or later |
| Required | No |
| Indexed | No |
| IME Mode | No Control |
| IME Sentence Mode | None |
| Smart Tags | |
| Text Align | General |
| Show Date Picker | For dates |

The data type determines the kind of values that users can store in the field. Press F1 for help on data types.


Users

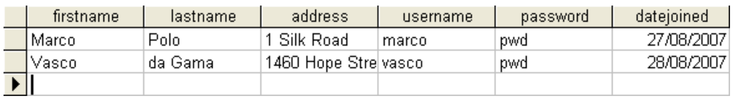
| firstname | lastname | address | username | password | datejoined | Add New Field |
|---|---|---|---|---|---|---|
| Marco | Polo | 1 Silk Road | | | 27/08/2006 | |

GCUTours system

Date must be in 2007 or later

## Choosing the right fields

Using a table can be made a lot easier with a bit of thought about the fields in your table. With the right choices, you can make it much easier to find and use the data. For example, data in character fields can easily be ordered alphabetically. So what happens if we order data in the *name* field of the *Users* table? We won't get a very sensible order, as the user's full name is in a single field. Usually we want to sort people by last name. Also, the current table is not very helpful if you want to search for a user by last name. A good rule of thumb is that any data that you're likely to want to sort or search by should be in a field by itself. We should really split *name* into two separate fields, *firstname* and *lastname*.



| firstname | lastname | address | username | password | datejoined |
|---|---|---|---|---|---|
| Marco | Polo | 1 Silk Road | marco | pwd | 27/08/2007 |
| Vasco | da Gama | 1460 Hope Stre | vasco | pwd | 28/08/2007 |
| | | | | | |

## Implementing a table in SQL

All RDBMSs allow you to create a table and define its fields using an SQL command. You can write SQL by creating a **query** and working with the query in **SQL View**. The Users table could be created with the following SQL in Access. Each field is specified with its name followed by its data type.

```
CREATE TABLE Users (
 firstname Text(10),
 lastname Text(15),
 address Text(50),
 username Text(10),
 password Text(10),
 datejoined DateTime
);
```

Here's the same example for a database which uses standard ANSI SQL types

```
CREATE TABLE Users (
 firstname VARCHAR(10),
 lastname VARCHAR(15),
 address VARCHAR(50),
 username VARCHAR(10),
 password VARCHAR(10),
 datejoined DATE
);
```

**NOTE**

We'll stick to Access SQL from now on, but do remember that to the examples you see might need to be changed slightly to be used with another database. Note also that you can use the **VARCHAR** data type in Access as it is recognised as a **synonym** for Text.

## Inserting data into a table

In Access you can insert data in datasheet view. You can also create user-friendly forms to allow non-expert users of your database to enter data. Another way of entering data into any RDBMS, including Access, is to use - guess what – SQL. The following SQL statement adds a new row of data into the *Users* table:

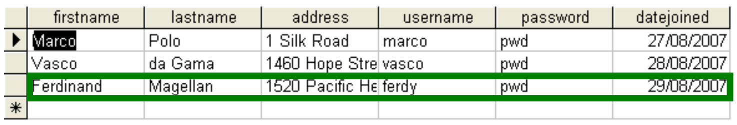
*list of field names*

```
INSERT INTO Users
(firstname,lastname,address,username,password,datejoined)
VALUES
('Ferdinand', 'Magellan', '1520 Pacific Heights', 'ferdy', 'pwd',
'29/8/2007');
```

*list of values –in same order as field names*

As long as the VALUES list has values for all the fields, in the same order as in the original CREATE TABLE statement, then you can miss out the list of fields.

```
INSERT INTO Users
VALUES
('Ferdinand', 'Magellan', '1520 Pacific Heights', 'ferdy', 'pwd',
'29/8/2007');
```



| firstname | lastname | address | username | password | datejoined |
|---|---|---|---|---|---|
| Marco | Polo | 1 Silk Road | marco | pwd | 27/08/2007 |
| Vasco | da Gama | 1460 Hope Stre | vasco | pwd | 28/08/2007 |
| Ferdinand | Magellan | 1520 Pacific He | ferdy | pwd | 29/08/2007 |
| | | | | | |

*new row added*

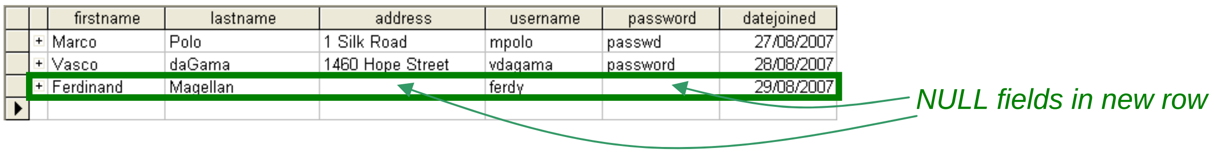
**NOTE**

You are probably wondering why on earth you would want to use SQL to insert data (or to query data) – surely it's much easier just to type data in an Access datasheet? It is worth learning the SQL way, though. If you know SQL you can **work the same way with any RDBMS.** Also, a database may be **part of an enterprise application**, and the data may be inserted through, for example, a web page. The application will typically use an SQL statements to get data from a web form into the database.

## NULL values

What if we miss out some of the fields when inserting a new row, like this (note that the list of fields and list of values still match):

```
INSERT INTO Users
(firstname,lastname,username,datejoined)
VALUES
('Ferdinand', 'Magellan', 'ferdy','29/8/2007');
```



| | firstname | lastname | address | username | password | datejoined |
|---|---|---|---|---|---|---|
| + | Marco | Polo | 1 Silk Road | mpolo | passwd | 27/08/2007 |
| + | Vasco | daGama | 1460 Hope Street | vdagama | password | 28/08/2007 |
| + | Ferdinand | Magellan | | ferdy | | 29/08/2007 |

The row is added, with the missing fields left **empty**. We say the value of these empty fields is **NULL**.

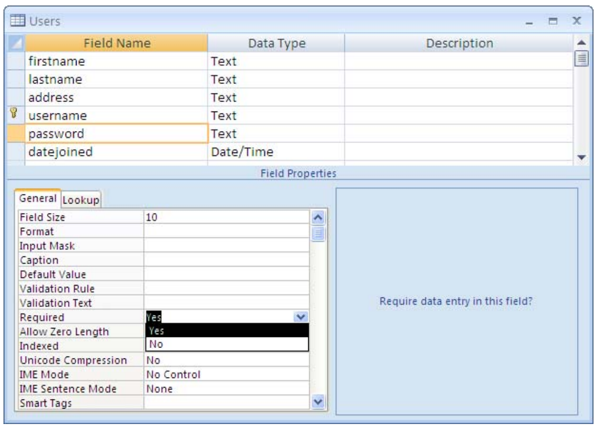
You can also set fields to have a NULL value explicitly:

```
INSERT INTO Users
(firstname,lastname,address,username,password,datejoined)
VALUES
('Ferdinand', 'Magellan', NULL, 'ferdy', NULL, '29/8/2007');
```

Often you **don't want to allow a field to ever be left empty**. You probably want to make sure, for example, that every user has a password. You make sure that any field in a table must contain a value by adding the words **NOT NULL** after the SQL field definition, for example:

```
password Text(10) NOT NULL,
```

You can also set the **Required** property to **Yes** in table design view:



| Field Name | Data Type | Description |
|---|---|---|
| firstname | Text | |
| lastname | Text | |
| address | Text | |
| username | Text | |
| password | Text | |
| datejoined | Date/Time | |

| | |
|---|---|
| Field Size | 10 |
| Format | |
| Input Mask | |
| Caption | |
| Default Value | |
| Validation Rule | |
| Validation Text | |
| Required | Yes |
| Allow Zero Length | Yes |
| Indexed | No |
| Unicode Compression | No |
| IME Mode | No Control |
| IME Sentence Mode | None |
| Smart Tags | |

Require data entry in this field?

Now the database will **not allow** the INSERT to add a new row to the table.

## Primary keys

What do you think will happen if two users with the same name are added to the Users table? We need to be able to tell them apart in some way, otherwise we may end up booking a holiday for the wrong person.

It is almost always necessary to ensure that every row in a table is **uniquely identified** – there shouldn't be two identical records in a table. This is done by defining a **primary key** for each table. A primary key is a **field**, or a **combination of fields**, that is guaranteed to have a unique value for every row in the table. The database will not allow new row to be added if the value or values of its primary key match a row which already exists in the table.

So what will we choose as the primary key for the Users table? What about *firstname*? No, it is likely that there will be users with the same first name. The same applies to *lastname*. We could use *(firstname, lastname)*, a combination of *firstname* and *lastname*, but there could easily be several users with the same full name. What about *(firstname, lastname, address)*? This looks more promising, but what if there are a father and son with the same name, living at the same address – we might think it's unlikely but we can't ignore the possibility.

For this table, two better possible choices would be:

- use the *username* field, so that every user needs to have a unique username
- add an **ID field** to the table specifically for the purpose of identifying each row

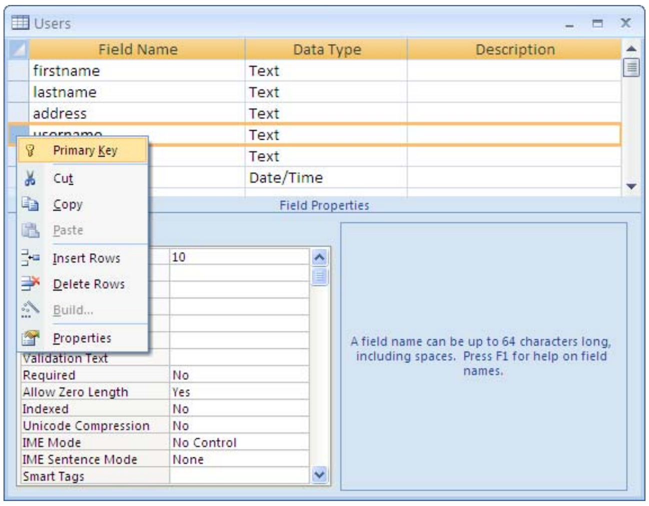
*username* is probably a good choice here as this particular table has an existing field which should contain a unique value in each row. However, many tables don't have an obvious candidate like this, and an ID field is the best choice. For example, there will be a table in this system to store bookings, and that table should probably have a *bookingID* field to give every booking a unique number.

**NOTE**

You are probably used to seeing numbers in everyday life which are the values of ID fields in database tables behind the scenes – for example customer numbers, order numbers, student numbers, and so on.

## Defining a primary key

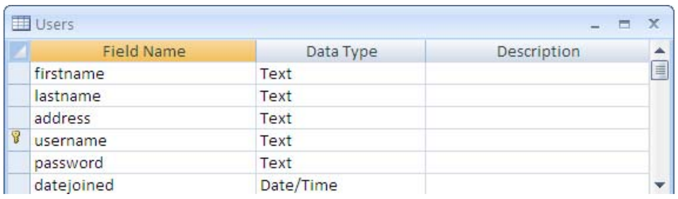
You can define a primary key in Access table design view. In the figures below, the username field is being defined as the primary key, and after doing so, a key symbol appears beside it. You can select more than one field for the primary key.



| Field Name | Data Type | Description |
|---|---|---|
| firstname | Text | |
| lastname | Text | |
| address | Text | |
| username | Text | |
| | Text | |
| | Date/Time | |

| | |
|---|---|
| | 10 |
| Validation Text | |
| Required | No |
| Allow Zero Length | Yes |
| Indexed | No |
| Unicode Compression | No |
| IME Mode | No Control |
| IME Sentence Mode | None |
| Smart Tags | |

A field name can be up to 64 characters long, including spaces. Press F1 for help on field names.



| Field Name | Data Type | Description |
|---|---|---|
| firstname | Text | |
| lastname | Text | |
| address | Text | |
| username | Text | |
| password | Text | |
| datejoined | Date/Time | |

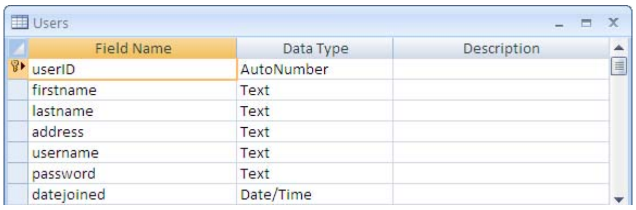
You can also define the primary key in SQL when you create the table, as shown in the example below. The brackets after PRIMARY KEY can contain one or more field names, separated by commas if there is more than one.

```
CREATE TABLE Users (
 firstname Text(10),
 lastname Text(15),
 address Text(50),
 username Text(10),
 password Text(10),
 datejoined DateTime,
 PRIMARY KEY(username)
);
```

## Data type for primary key

Fields of any data type can be used for the primary key. It is common to use either integer or text fields.

Most RDBMSs have a special data type intended for ID fields. In Access it is called the **AutoNumber** type – this field holds an integer value which is set automatically by the database when a new row of data is added to a table. The value set is always unique, and is one more than the largest value already in the database. The figure below shows the *Users* table with an ID field added as the primary key instead of *username*.



Users

| Field Name | Data Type | Description |
|---|---|---|
| userID | AutoNumber | |
| firstname | Text | |
| lastname | Text | |
| address | Text | |
| username | Text | |
| password | Text | |
| datejoined | Date/Time | |

**NOTE**

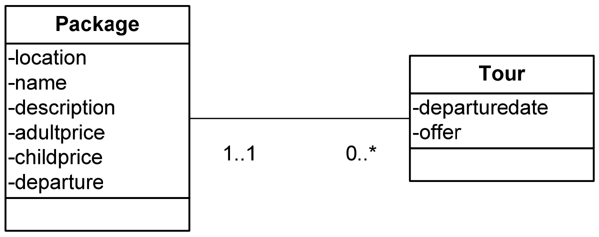
The data type for an Access AutoNumber field in an SQL CREATE TABLE statement is ***Counter***. This may seem strange – it's a hangover from very old versions of Access.

When you use an SQL **INSERT INTO** statement to add a row to a table with an AutoNumber primary key, you should **leave the AutoNumber field out of the list of fieldnames AND the list of values.** The database will automatically set the field value.

## Representing relationships

Entities in a data model rarely exist by themselves. The entities in the GCUTours class diagram are all related to other entities. For example bookings are related to tours and to users. It wouldn't make sense to create a booking for a tour that didn't exist, or for a user that didn't exist. Relationships in a database can prevent that sort of invalid data from being stored – this is known as enforcing **referential integrity**.

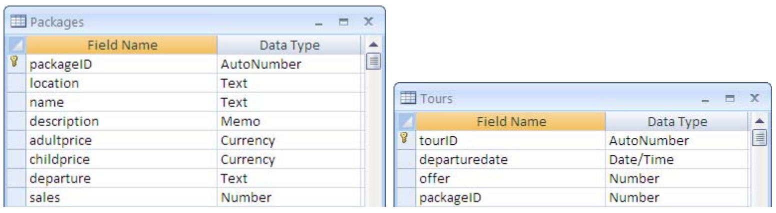
Relationships are defined in the data model as one-to-one, one-to-many or occasionally many-to-many associations. So how do we make these work in a database schema? Let's look at two related entities with a one-to-many association, shown in the class diagram below:



- **Package** – a holiday package, such as the “Western Adventure” holiday, including information about the destination and the activities on holiday

- **Tour** – a holiday tour, which is related to a specific package. A tour includes information on departure dates and any discounts offered on the standard package price. There can be many tours (with different dates and special offers) for the same package – for example there might be four separate “Western Adventure” tours in a year.

First, we need to design two tables to represent these classes, *Packages* and *Tours*. Each table should have a primary key, and we’ll use ID fields for these. The figure below shows these tables in Access design view:


Packages

| Field Name | Data Type |
|---|---|
| packageID | AutoNumber |
| location | Text |
| name | Text |
| description | Memo |
| adultprice | Currency |
| childprice | Currency |
| departure | Text |
| sales | Number |

Tours

| Field Name | Data Type |
|---|---|
| tourID | AutoNumber |
| departuredate | Date/Time |
| offer | Number |
| packageID | Number |

**THINK ABOUT IT...**

We could have used the *name* field as the primary key for Packages – can you think why that might not be a good choice?

## Foreign keys

There is nothing yet in either table to say which tours are connected to which packages. We need to add a field to one table which will make that connection. This field will be a **foreign key field**.

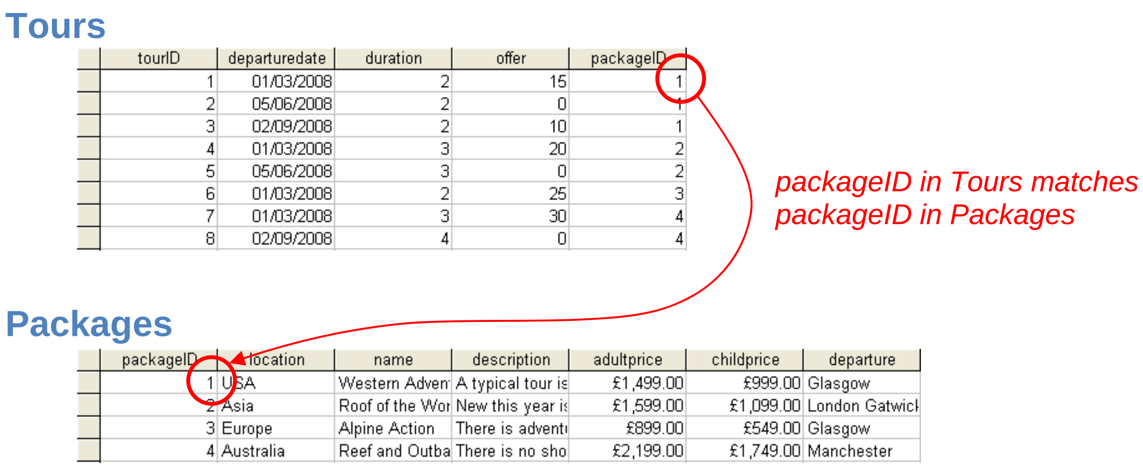
Which table should contain the foreign key field? Generally, in a one-to-many relationship, the foreign key field will be in the table at the ‘many’ end of the relationship. In this example, this is the *Tours* table.

What should the foreign key field in *Tours* contain? Well, it needs to be some value that **uniquely identifies a row** in the *Packages* table. As you have learned, that is exactly what the **primary key** of *Packages* does. As a general rule:

> **the *foreign key field* of a row in the table at the *many* end of the relationship should contain a value matching the value of the *primary key field* in one row of the related table**

In other words, *Tours* should have a field matching the *packageID* field in *Packages*. The figure below shows some data in both tables. The foreign key field has been called *packageID* and is of type *Number* with field size Long Integer – this matches the data in the AutoNumber field in *Packages*. Note that to create a Long Integer field in an SQL Create Table statement you should specify ***Long*** as the data type.

The figure below shows some data in these tables with the relationship defined in this way. Note that there are several rows in *Tours* which match the same row in *Packages*.



| tourID | departuredate | duration | offer | packageID |
|---|---|---|---|---|
| 1 | 01/03/2008 | 2 | 15 | 1 |
| 2 | 05/06/2008 | 2 | 0 | 1 |
| 3 | 02/09/2008 | 2 | 10 | 1 |
| 4 | 01/03/2008 | 3 | 20 | 2 |
| 5 | 05/06/2008 | 3 | 0 | 2 |
| 6 | 01/03/2008 | 2 | 25 | 3 |
| 7 | 01/03/2008 | 3 | 30 | 4 |
| 8 | 02/09/2008 | 4 | 0 | 4 |

| packageID | location | name | description | adultprice | childprice | departure |
|---|---|---|---|---|---|---|
| 1 | USA | Western Adven | A typical tour is | £1,499.00 | £999.00 | Glasgow |
| 2 | Asia | Roof of the Wor | New this year is | £1,599.00 | £1,099.00 | London Gatwick |
| 3 | Europe | Alpine Action | There is advent | £899.00 | £549.00 | Glasgow |
| 4 | Australia | Reef and Outba | There is no sho | £2,199.00 | £1,749.00 | Manchester |

### Referential integrity

The foreign key makes sure that we **can't create a tour for a package that doesn't exist**. If we tried to insert a new row in *Tours* with *packageID* = 20, for example, the database wouldn't allow it because there is no matching row in *Packages*.

What if we insert a row into *Tours* with a **NULL** in *packageID*? This will work, as the foreign key relationship only ensures that we can't have a value that doesn't exist in the other table. If we want to make sure that **any new tour MUST match an existing package**, then we can set the *packageID* field in *Tours* to be **NOT NULL**.

Should you make a foreign key field NOT NULL? What is the effect of doing do?

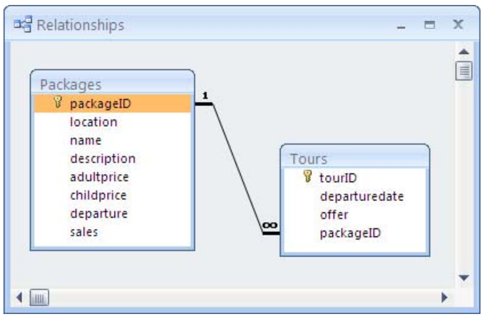
- **NULL allowed**: a Tour can be created and assigned to a package later
- **NOT NULL**: a tour must be assigned to a valid package at the time it is created

**THINK ABOUT IT…**

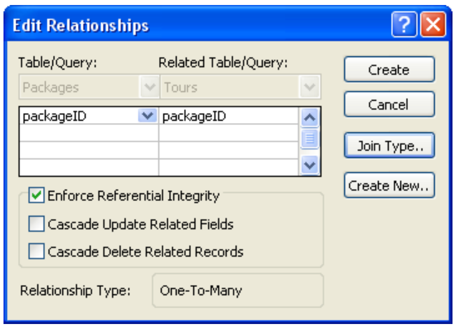
Which do you think is the right choice here? Can you think of another situation where you would choose differently?

## Defining relationships

Access has a graphical tool for defining relationships. The **Relationships** window displays tables of your choice. You can then create a relationship by dragging a connection from the field which you want to be the foreign key to the primary key field of the related table. The relationship is shown graphically – the figure below shows the relationship between *Packages* and *Tours*:



When you create the relationship you can set up some properties of it. You will usually want to choose the 'Enforce Referential Integrity' option. Can you guess what effects the other options shown in the figure below will have?



Table/Query: Packages
Related Table/Query: Tours

packageID — packageID

[✓] Enforce Referential Integrity
[ ] Cascade Update Related Fields
[ ] Cascade Delete Related Records

Relationship Type: One-To-Many

As always, we should look also at the SQL version. The same relationship would be defined by including the following lines in the CREATE TABLE statement for *Tours*:

```
packageID Long NOT NULL,
FOREIGN KEY(packageID) REFERENCES Packages(packageID)
```

Relationships defined using SQL will appear automatically in the Relationships window.

## Relationships with multiple fields

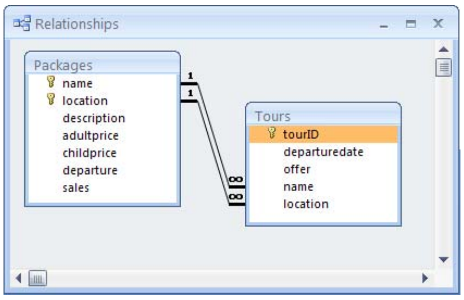
What if the primary key of the related table contains more than one field? For example, instead of creating the *packageID* field in *Packages*, we might have used *name* and *location* together as the primary key.

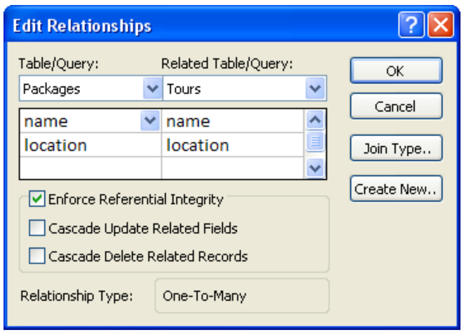
```
PRIMARY KEY(name, location)
```

The foreign key in *Tours* would **need to reference both fields**. Note that *Tours* would also need to have *name* and *location* fields defined:

```
name Text NOT NULL,
location Text NOT NULL,
FOREIGN KEY(name, location) REFERENCES Packages(name, location)
```

When you look at the tables in the Relationships window you see that both fields are linked, and the Edit Relationships dialogue shows that the relationship is based on two fields.





Table/Query: Packages
Related Table/Query: Tours

| Packages | Tours |
|---|---|
| name | name |
| location | location |

[✓] Enforce Referential Integrity
[ ] Cascade Update Related Fields
[ ] Cascade Delete Related Records

Relationship Type: One-To-Many

The effect of defining this relationship is that each row in *Tours* must have a value of *name* **and** a *value* of location matching those in a single row of *Packages*.

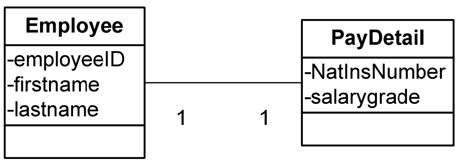
## Other types of relationship

The example we have looked at is a one-to-many relationship, which is probably the most common type. What about other kinds of relationships?

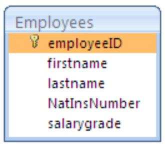
### *One-to-one*

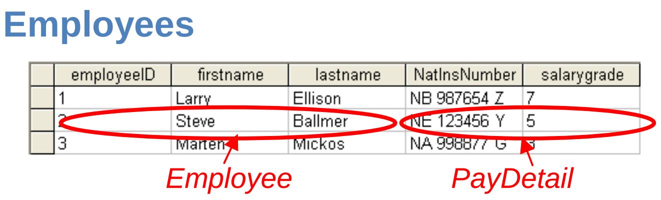
In this kind of relationship, pairs of rows in two tables are matched only to each other. Exactly one row in one table is matched exactly to one row in the other table.

Let's see an example. GCUTours might have another database to keep information about its staff. The data model has an *Employee* entity and another entity to represent the details of an employee's pay (NI number, salary grade). A *PayDetail* belongs exclusively to one *Employee*, who can only have one *PayDetail*.



A one-to-one relationship like this can usually be implemented with a single table with fields representing the attributes of both entities. Each row contains an employee **and** the related pay details. There is little need to have two separate tables here.





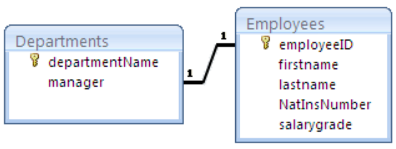
Employees

| employeeID | firstname | lastname | NatInsNumber | salarygrade |
|---|---|---|---|---|
| 1 | Larry | Ellison | NB 987654 Z | 7 |
| 2 | Steve | Ballmer | NE 123456 Y | 5 |
| 3 | Marten | Mickos | NA 998877 G | 3 |

### *A more complicated example of one-to-one*

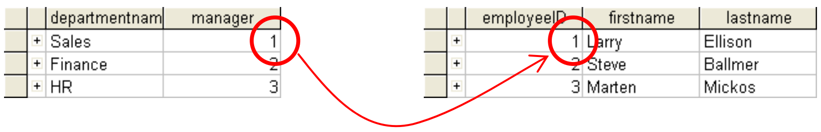
There are some situations where there is a one-to-one relationship between entities which really do belong in different tables. Let's say the staff database has tables for *Employees* and *Departments*. Each department has **one manager**, whose details will be in the *Employees* table. **One employee** can only be **manager of one department**, so this is a one-to-one relationship.

The relationship is implemented in the same way as a one-to-many relationship, with a foreign key. Here, the *manager* field in *Departments* matches the *employeeID* primary key in *Employees*.



## Departments

## Employees



| | departmentnam | manager |
|---|---|---|
| + | Sales | 1 |
| + | Finance | 2 |
| + | HR | 3 |

| | employeeID | firstname | lastname |
|---|---|---|---|
| + | 1 | Larry | Ellison |
| + | 2 | Steve | Ballmer |
| + | 3 | Marten | Mickos |

Note that as this relationship matches one employee to one department, **each *employeeID* value should appear once only** in the *manager* field in *Departments*. This can be ensured by either:

- Making *manager* the primary key of *Departments*, as well as being a foreign key

```
manager Long PRIMARY KEY,
FOREIGN KEY(manager) REFERENCES Employees(employeeID)
```

- or, making *manager* **unique**, without it being the primary key – a **unique constraint** is similar to a primary key, except that there can be more than one unique constraint in a table, but a table can have only one primary key (although the primary key can contain more than one field as you have seen)

```
manager Long UNIQUE,
FOREIGN KEY(manager) REFERENCES Employees(employeeID)
```

**NOTE**

Notice from this example that the name of a foreign key field does **not have to be the same** as the name of the referenced primary key field.

We could equally well have chosen to put the foreign key field in *Employees*, so that it had a field called *manager_of*? However, most employees would not be managers, so the *manager_of* field would be null for most rows of the table, wasting storage space.

**THINK ABOUT IT...**

1.There could be another relationship between these tables, because employees work for departments. What kind of relationship is this and how would you implement it?

2. The *Employee* and *PayDetail* example could be implemented with two tables and a foreign key in a similar way. How would you do this? Can you think of any advantage over a single table?

**You've had a couple of options thrown at you here – let's sum up:**

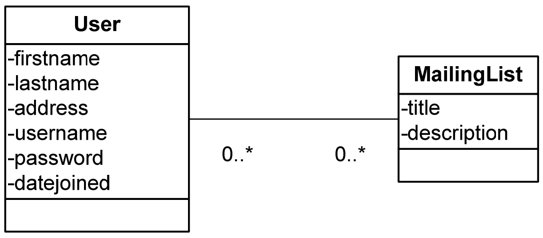
- You can **usually** represent a one-to-one relationship with **a single table** containing the attributes of both entities
- If there is a good reason to use **two tables**, then you represent the relationship with a **foreign key** field in any one of the tables, matching the primary key in the other. The foreign key field must itself be unique.

## *Many-to-many*

To understand this relationship, let's assume that GCUTours has decided to set up some email lists for its users.

- each user can be subscribed to **many** mailing lists
- each mailing list can have **many** users subscribed to it

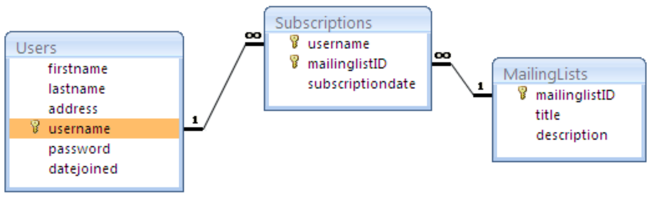
This is represented in a UML data model like this:



However, relational databases do not allow direct many-to-many relationships between tables. Using foreign keys would not work – a foreign key can only match **one** value in a related table.

We need to represent the relationship by creating an **additional table** with which both entities have a **one-to-many** relationship.

In this case the additional table could be called *Subscriptions*, and **each row** would represent the subscription of **one user** to **one mailing list**. Note that we can also include in this table further information about the subscription, for example the subscription date.



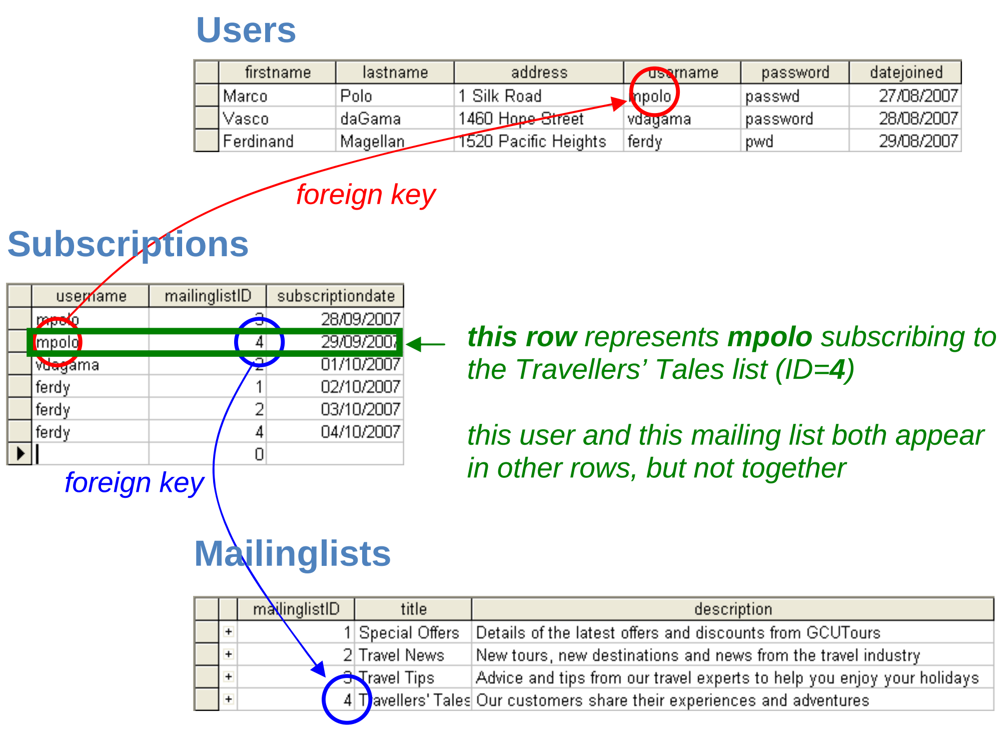
The relationships between the additional table and each other table are set up in the usual way for one-to-many relationships:

- ***Subscriptions*** has a foreign key field *username* which matches the *username* field in ***Users.*** The data type is text.
- ***Subscriptions*** has a foreign key field *mailinglistID* which matches the *mailinglistID* field which is the primary key of ***Mailinglists***. The data type is Long to match the Counter primary key of Mailinglists.

The primary key of the *Subscriptions* table should be (*username*, *mailinglistID*) as each user should be subscribed only once to a particular mailing list. As you can see from the data shown, each *username* can be in the table several times, and so can each *mailinglistID* – however, each **combination** is unique.

The SQL to create the Subscriptions table looks like this:

```
CREATE TABLE Subscriptions(
 username Text NOT NULL,
 mailinglistID Long NOT NULL,
 subscriptiondate DateTime NOT NULL,
 PRIMARY KEY(username,mailinglistID),
 FOREIGN KEY(username) REFERENCES Users(username),
 FOREIGN KEY(mailinglistID) REFERENCES Mailinglists(mailinglistID)
);
```



Users

| firstname | lastname | address | username | password | datejoined |
|---|---|---|---|---|---|
| Marco | Polo | 1 Silk Road | mpolo | passwd | 27/08/2007 |
| Vasco | daGama | 1460 Hope Street | vdagama | password | 28/08/2007 |
| Ferdinand | Magellan | 1520 Pacific Heights | ferdy | pwd | 29/08/2007 |

Subscriptions

| username | mailinglistID | subscriptiondate |
|---|---|---|
| mpolo | 3 | 28/09/2007 |
| mpolo | 4 | 29/09/2007 |
| vdagama | 2 | 01/10/2007 |
| ferdy | 1 | 02/10/2007 |
| ferdy | 2 | 03/10/2007 |
| ferdy | 4 | 04/10/2007 |
| | 0 | |

***this row** represents **mpolo** subscribing to the Travellers' Tales list (ID=**4**)*

*this user and this mailing list both appear in other rows, but not together*

Mailinglists

| | mailinglistID | title | description |
|---|---|---|---|
| + | 1 | Special Offers | Details of the latest offers and discounts from GCUTours |
| + | 2 | Travel News | New tours, new destinations and news from the travel industry |
| + | 3 | Travel Tips | Advice and tips from our travel experts to help you enjoy your holidays |
| + | 4 | Travellers' Tales | Our customers share their experiences and adventures |

**NOTE**

In this example there was an obvious name for the additional table. Sometimes there is no obvious meaningful name to call an additional table. In that case, it is usually named with a combination of the names of the two related tables. In this case, that would mean calling the table something like *UsersMailinglists*.

## Summary of GCUTours schema

The notation commonly used to summarise a database schema indicates table and field names, and which fields are primary keys and which are foreign keys. The notation is as follows:

**TableName**(<u>primary key field(s)</u>, non key field(s), *foreign key field(s)*)

Here's a summary of the main tables in the GCUTours schema:

**Packages**(<u>packageID</u>, location, name, description, adultprice, childprice, departure)
**Tours**(<u>tourID</u>, departuredate, offer, *packageID*)
**Users**(firstname, lastname, address, <u>username</u>, password, datejoined)
**Bookings**(<u>bookingID</u>, *tourID*, *username*, adults, children, status)
**MailingLists**(<u>mailinglistID</u>, title, description)
**Subscriptions**(<u>*username*</u>, <u>*mailinglistID*</u>, subscriptiondate)

Note that *Subscriptions* has a **compound primary key** whose fields are **both foreign keys** (relating to two different tables).

## A final thought on relationships

In an enterprise system, the **same data model** may be represented in **different ways** in different parts of the system, for example by a relational database schema for long-term storage and by Java classes for processing in memory. These representations deal with relationships in very different ways:

- **relational database** – uses matching foreign and primary key fields
- **Java classes** – use object references, with no need for matching fields

This can get confusing – you need to make sure you use the right techniques for the representation you are designing.

## Altering a table

Sometimes you need to **change the design** of a table after it has been created, or even after it has data in it. You can do this in table design view, or in SQL. The following SQL adds a new field to the *Users* table. Existing rows will simply have a NULL value in that field.

```
ALTER TABLE Users ADD email Text(25)
```

The next example adds a primary key, which could be useful if we forgot to define it when the table was created:

```
ALTER TABLE Users ADD PRIMARY KEY(username)
```

Finally, this example drops, or deletes, a field from the table:

```
ALTER TABLE Users DROP COLUMN datejoined
```

Dropping fields can be dangerous as the data in that field of every row will be deleted. Some RDBMSs don't allow alterations which may lead to loss of data – Access seems quite happy to drop fields without warning you of the consequences!