Part III: On your marks... get set... go!

At this point we have a UML diagram of our database, complete with tables and connections; we have the corresponding table definitions lists, we have a design for our forms and a design for our reports. Now is time to turn on the computer. Go ahead! In fact, most of the text in this part will not make much sense unless you are following the "click on this, then click on that" instructions that we provide. Take your time and study the screens and dialog boxes with the aid of the descriptions made here.

1. General Overview:

The first thing that you will notice when you open Base is that the Database Wizard pops-up. The interface has two columns. The small one to the right describes the steps the wizard will walk you through. The bigger column offers the different options available to you. First the wizard asks you if you want to create a new database, open a previous one or connect to an existing database. By selecting "new database", Base will work with the embedded HSQL database engine.

In the second step (Save and proceed), Base asks us first if we want to register the database with OpenOffice.org. This doesn't mean that your information might be accessible to other folks using or developing the OpenOffice.org suite. What it does mean is that other applications of OpenOffice.org, like Writer or Calc, in your own computer, will be aware of the existence of this new database and, therefore, will be able to interact with it -for example, to produce and print labels. With all confidence, mark "yes". We will not be using the tables wizard, so leave the default and hit "finish". For the last step, chose a name for your database and save.

Now the database interface opens. Below the menu bar and the tool bar you will find the screen divided in 3 columns. The first one to the left is very narrow and has the heading: "Database". You will see 4 icons with the following text, correspondingly: Tables, Queries, Forms and Reports. The first one gives you access to the tables in your database and the tasks that you can perform on them. Click the "Tables" icon and you will see the available tasks appear in the second column. If you position your cursor on top of any of these tasks, a description of what they do appears in the third column. Pretty nice, eh?

One third down the page you can see a horizontal gray bar with the text "Table". There is where the name of each table you build will be stored, so that you can access any of them by clicking on it. Because we have not built any tables so far this section is now empty.

If you now click on "Queries" in the first column, you are taken to the queries page, and so on for Forms and Reports. Each page offers you the available tasks, a brief description of the task under the mouse cursor, and the bottom section that stores your work.

You will see that there is always more way than one to do the same task in Base. For example, "Create Table in Design View" and "Use Wizard to Create Table" (in the Tables page) both allow you to create tables for your database. In this case, the difference is that the wizard will make the task easier while Design View gives you somewhat more flexibility. You still have a third option for doing this: SQL commands, which gives you the most flexibility and control.

2. Creating tables and relationships with SQL commands.

We will start by creating the tables. In this tutorial we are going to use SQL commands to build them.

Base accepts SQL commands in two distinctive windows: The most common application of SQL is for creating queries, done in the “Queries” page, which is where you should be now. Let's analyze this for a moment. You can see that the third task is “Create Query in SQL View...” next to an icon that reads “SQL”. We will see later that queries start with the SQL instruction “SELECT”. This instruction does not modify (or create or delete) your data or data structure in any way, it just displays the available data in the fashion you specify with the “SELECT” command. For this reason, Base will report an error for any SQL query that does not start with “SELECT” made in the Queries page. Let's now move to the “Tables” page.

For creating tables w will need another window for SQL instructions. To reach it click “Tools” at the menu bar and then click on the option “SQL...”. The “Execute SQL Statement” window opens. You will find a cursor blinking at the “Command to Execute” box. This is where we will type the instructions to create the tables. SQL commands in this window do modify your data and do not need to start with the instruction SELECT.

You will have to type the complete following set of instructions. Pay attention to quote signs, capitalization and syntax. You can also copy and paste these instructions from here or from the link http://xxxxx. When you enter the instructions, the “Execute” box highlights. When you are done, click on it. This will run your commands and create the tables. After a few seconds, the window will inform you that the instructions have been executed. Other than that there will be no visible signs on your screen. If you now go to the “View” menu and click on “Refresh Tables”, a complete list of the tables you have created will appear in the lower section of your screen. (You can also try exiting and re-entering the “Tables” view). Try this now.

Before analyzing the SQL instructions that have just created your tables, I want to show you that the relationships between them have also been created. Click on “Tools” and then on “Relationships...”. You will see a long line of boxes and some lines connecting them. Each box is a table and their different attributes (columns) are listed inside. Some boxes have scroll bars. This is because they have more attributes than the number that can appear in the default size for the boxes. You can readjust their sizes (width and height). You can also reposition the boxes so that the tangle of lines connecting them becomes easier to read. You will also notice that these lines connect the primary keys (indicated with a small yellow key symbol) with the foreign keys and display the corresponding cardinalities, just like a UML diagram. Isn't this fine? In fact, by the time that you finish tidying the relationship view, it should appear just like the UML diagram in our example.

Let's analyze the SQL instructions that make all this possible. Take the time to read through this and see if you can derive any sense from it. We will analyze them together afterwards:

```
DROP TABLE "Patient Medication" IF EXISTS;
DROP TABLE "Medication" IF EXISTS;
DROP TABLE "Payment" IF EXISTS;

DROP TABLE "Account" IF EXISTS;
DROP TABLE "Transaction Type" IF EXISTS;

DROP TABLE "Schedule" IF EXISTS;
DROP TABLE "Assignment" IF EXISTS;
DROP TABLE "Therapists Number" IF EXISTS;
DROP TABLE "Phone Number" IF EXISTS;

DROP TABLE "Patient" if exists;
```

We have not respected name formation for physical tables that we have described!!

Not normalized. It will be considered caps by the SQL window, anyway

```
DROP TABLE "Psychiatrist" if exists;
DROP TABLE "Medical Doctor" if exists;
DROP TABLE "Therapist" IF EXISTS;

CREATE TABLE "Psychiatrist" (
"ID Number" INTEGER GENERATED BY DEFAULT AS IDENTITY(START WITH 0) NOT
NULL PRIMARY KEY,
"First Name" VARCHAR(50) NOT NULL,
"Surname" VARCHAR(50) NOT NULL,
"Gender" CHAR(6),
"Date of Birth" DATE,
"Street and number" VARCHAR(200),
"City" VARCHAR(100),
"Postal code" VARCHAR(50),
"State" CHAR(2),
"Phone Number" VARCHAR(10)
);

CREATE TABLE "Medical Doctor" (
"ID Number" INTEGER GENERATED BY DEFAULT AS IDENTITY(START WITH 0) NOT
NULL PRIMARY KEY,
"First Name" VARCHAR(50) NOT NULL,
"Surname" VARCHAR(50) NOT NULL,
"Gender" CHAR(6),
"Date of Birth" DATE,
"Street and number" VARCHAR(200),
"City" VARCHAR(100),
"Postal code" VARCHAR(50),
"State" CHAR(2),
"Phone Number" VARCHAR(10)
);

CREATE TABLE "Patient" (
"ID Number" INTEGER GENERATED BY DEFAULT AS IDENTITY(START WITH 0) NOT
NULL PRIMARY KEY,
"First Name" VARCHAR(50) NOT NULL,
"Surname" VARCHAR(50) NOT NULL,
"Gender" CHAR(6),
"Date of Birth" DATE,
"Street and number" VARCHAR(200),
"City" VARCHAR(100),
"Postal code" VARCHAR(50),
"State" CHAR(2),
"Diagnosis" VARCHAR(1024),
"Medical Doctor ID" INTEGER,
"Psychiatrist ID" INTEGER,
"Time of registry" TIMESTAMP,
CONSTRAINT "CK_PAT_GNDR" CHECK( "Gender" in ( 'Male', 'Female' ) ),
CONSTRAINT FK_PAT_PSY FOREIGN KEY ("Psychiatrist ID") REFERENCES
"Psychiatrist" ("ID Number"),
CONSTRAINT FK_PAT_DOC FOREIGN KEY ("Medical Doctor ID") REFERENCES
"Medical Doctor" ("ID Number")
);

CREATE TABLE "Phone Number" (
"Phone ID" INTEGER GENERATED BY DEFAULT AS IDENTITY(START WITH 0) NOT
NULL PRIMARY KEY,
"Patient ID" INTEGER NOT NULL,
"Number" VARCHAR(10),
```

Do I need such big vars?

```
"Description" VARCHAR(10),
CONSTRAINT FK_PAT_PHN FOREIGN KEY ("Patient ID") REFERENCES "Patient"
("ID Number")
);

CREATE TABLE "Therapist" (
"ID Number" INTEGER GENERATED BY DEFAULT AS IDENTITY(START WITH 0) NOT
NULL PRIMARY KEY,
"First Name" VARCHAR(50) NOT NULL,
"Surname" VARCHAR(50) NOT NULL,
"Gender" CHAR(6),
"Date of Birth" DATE,
"Street and number" VARCHAR(200),
"City" VARCHAR(100),
"Postal code" VARCHAR(50),
"State" CHAR(2),
"Tax number" VARCHAR(40),
"Academic degree" VARCHAR(10),
"License number" VARCHAR(40),
"Hiring date" DATE NOT NULL,
"Termination date" DATE,
CONSTRAINT "CK_THP_GNDR" CHECK( "Gender" in ( 'Male', 'Female' ) ),
CONSTRAINT "CK_TERM_DT" CHECK( "Termination date" > "Hiring date" )
);

CREATE TABLE "Therapists Number" (
"Phone ID" INTEGER GENERATED BY DEFAULT AS IDENTITY(START WITH 0) NOT
NULL PRIMARY KEY,
"Therapist ID" INTEGER,
"Number" VARCHAR(10),
"Description" VARCHAR(10),
CONSTRAINT FK_THP_PHN FOREIGN KEY ("Therapist ID") REFERENCES
"Therapist" ("ID Number")
);

CREATE TABLE "Assignment" (
"Assignment ID" INTEGER GENERATED BY DEFAULT AS IDENTITY(START WITH 0)
NOT NULL PRIMARY KEY,
"Patient ID" INTEGER NOT NULL,
"Therapist ID" INTEGER NOT NULL,
"Date assigned" DATE DEFAULT CURRENT_DATE NOT NULL,
"Date case closed" DATE,
CONSTRAINT FK_PAT_ASMT FOREIGN KEY ("Patient ID") REFERENCES "Patient"
("ID Number"),
CONSTRAINT FK_THP_ASMT FOREIGN KEY ("Therapist ID") REFERENCES
"Therapist" ("ID Number"),
CONSTRAINT "CK_CLOSE_DT" CHECK( "Date case closed" >= "Date assigned" )
);

CREATE TABLE "Schedule" (
"Schedule ID" INTEGER GENERATED BY DEFAULT AS IDENTITY(START WITH 0) NOT
NULL PRIMARY KEY,
"Assignment ID" INTEGER,
"Slot date" DATE NOT NULL,
"Slot hour" TIME NOT NULL,
"Status of the session" VARCHAR(10),
CONSTRAINT FK_SCH_ASMT FOREIGN KEY ("Assignment ID") REFERENCES
"Assignment" ("Assignment ID")
);
```

Default. Current_Date is a function

```
CREATE TABLE "Transaction Type" (
"Transaction Type ID" INTEGER GENERATED BY DEFAULT AS IDENTITY(START
WITH 0) NOT NULL PRIMARY KEY,
"Transaction Description" VARCHAR(50),
"Debit" CHAR(6) NOT NULL,
CONSTRAINT CK_DBT CHECK("Debit" IN ('DEBIT', 'CREDIT'))
);
```


```
CREATE TABLE "Account" (
"Account Entry ID" INTEGER GENERATED BY DEFAULT AS IDENTITY(START WITH
0) NOT NULL PRIMARY KEY,
"Patient ID" INTEGER NOT NULL,
"Transaction Type" INTEGER NOT NULL,
"Transaction comments" VARCHAR(1024),
"Amount" DECIMAL(10,2) NOT NULL,
"Date and time of transaction" TIMESTAMP(6) DEFAULT CURRENT_TIMESTAMP
NOT NULL,
CONSTRAINT FK_PAT_ACNT FOREIGN KEY ("Patient ID") REFERENCES "Patient"
("ID Number"),
CONSTRAINT FK_TRN_TYP FOREIGN KEY ("Transaction Type") REFERENCES
"Transaction Type" ("Transaction Type ID"),
CONSTRAINT IDX_PAT UNIQUE ( "Patient ID" )
);

CREATE TABLE "Payment" (
"Payment ID" INTEGER GENERATED BY DEFAULT AS IDENTITY(START WITH 0) NOT
NULL PRIMARY KEY,
"Patient ID" INTEGER NOT NULL,
"Date and time of payment" TIMESTAMP(6) DEFAULT CURRENT_TIME NOT NULL,
"Amount payed" DECIMAL(10, 2),
CONSTRAINT FK_PAT_PMT FOREIGN KEY ("Patient ID") REFERENCES "Account"
("Patient ID")
);

CREATE TABLE "Medication" (
"Medication ID" INTEGER GENERATED BY DEFAULT AS IDENTITY(START WITH 0)
NOT NULL PRIMARY KEY,
"Name" VARCHAR(100) NOT NULL,
"Description" VARCHAR(1024)
);

CREATE TABLE "Patient Medication" (
"Patient ID" INTEGER NOT NULL,
"Medication ID" INTEGER NOT NULL,
"Dosage" VARCHAR(50),
"Start date" DATE DEFAULT CURRENT_DATE NOT NULL,
"End date" DATE,
CONSTRAINT PK_PAT_MED PRIMARY KEY ("Patient ID", "Medication ID" ),
CONSTRAINT FK_MED_PAT FOREIGN KEY ("Medication ID") REFERENCES
"Medication" ("Medication ID"),
CONSTRAINT FK_PAT_MED FOREIGN KEY ("Patient ID") REFERENCES "Patient"
("ID Number"),
CONSTRAINT CK_END_DT CHECK( "End date" >= "Start date" )
);
```

The first thing one can notice is that this set has two types of instructions:

DROP TABLE

and

```
CREATE TABLE
```

in that order. The reason for this is that if you find the need to modify your database structure in any way (add a table, omit attributes, change connections, etc) and you run the modified SQL instructions, the commands will start by erasing the previous tables, in effect allowing you to start from scratch. The down side of this is that you will lose any data you might have entered into the tables.

Notice that the instructions use capitalized letters and that the name of the tables use caps and smalls and are enclosed within quotes. Without the quotes, Base will consider all letters as caps. This feature relieves the programmer from jumping from lower to upper case too often. Unfortunately, it also creates potential problems like lost tables or columns impossible to find. We explain this in more detail shortly.

Also note that the imperative tone: "DROP TABLE" is toned down by the inclusion of a conditional statement "IF EXISTS". This way, if the table didn't exist to begin with (like when you run these commands for the first time) then the application will not prompt an error message. For the same reason, be aware that it will also not prompt an error message if you mistype the name of the table, making you believe that you have erased a table that you really haven't, so type carefully or check this issue if things are not running the way you want.

Finally, note that all the instructions end with a semicolon: ";". This is the SQL equivalent of a period in the English language and its omission is bound the create confusion.

The CREATE TABLE instruction starts by providing the name of the table to be created, also within quotes and using caps and small letters, and then describes the particular characteristics this table will have: column names, type of variables used, relationships, the way to handle deletions, etc. These instructions are written within parentheses. It then finishes with a semicolon, more or less like this:

```
CREATE TABLE "Name of the Table" ("Name of Column" COLUMN ATTRIBUTES, etc.);
```

Inside the parentheses you specify first the name of each column, also within quote signs, and then the column attributes, like the kind of variable you are using (e.g.: integer, varchar, date variable types, etc.) and attributes like not null or auto increment, etc. Note that the instructions inside the parentheses are separated with a coma: "," and consequently the last instruction does not have one.

At this time, please note the close resemblance of the SQL instructions with the Variables Definition Lists we made earlier and how the former can be translated to the later with very little difficulty (so now you know why we insisted so much on them). In fact, SQL can be read very much like a formalized English. Go back and read some instructions creating tables and see if you can identify the components that we talk about here.

Maybe you are asking: Do I now need to learn SQL? To be exact, you should; and it can help you greatly in building good Base databases providing even greater flexibility than the wizard or design views. On the other hand, when you build tables, all your instructions will conform, more or less, to the same structures that appear in this example. Only the name of the tables or their attributes are bound to change (although there will be plenty of recurrences) and you will be selecting within a known set of column attributes, most of the which have been described so far. This means that if you understand the instruction set we use in this example, you should be able to handle most definitions for your own

databases without requiring a college degree. So take some time and analyze the way the different tables have been built, and start developing a sense of how SQL works and the syntax and grammar it uses.

Possibly the more puzzling instruction is CONSTRAINT found inside the CREATE TABLE command. This is a very useful instruction that allows you to, for example, set the connections between the tables, indicating which column is the primary key and which are the foreign keys. It can also allow you to set restrictions, for example, making sure that only M or F be accepted as gender entries, rejecting other input, or making sure that values in one column are always smaller, greater or equal than values in other columns, rejecting entries that do not conform to those rules.

The CONSTRAINT instruction is followed by the name of the constraint. This name is chosen by us, just like we gave a name to the tables or to their columns. In this case, all names are given in caps, with underscores separating components of the names. This way we don't need to use quote signs or worry about capitalization. For example, find the following name in the last table of the instruction set:

`FK_PAT_MED` that appears in:

```
CONSTRAINT FK_PAT_MED FOREIGN KEY ("Patient ID") REFERENCES "Patient" ("ID Number")
```

We came up with this name **`FK_PAT_MED`** to, somehow, mean that this constraint creates a foreign key in the patient-medication table. After the name comes the instruction that, in this case, defines that the "Patient ID" column in this table is a foreign key and holds the value of the "ID Number" column found in the "Patient" table. Now, that wasn't so difficult. All your foreign keys will be defined with this same pattern, just changing the names of your columns and the tables they reference. Of course, you will need to think of unique names for each declaration.

The CONSTRAINT names we chose can look a bit cryptic, but made sense to us because we wanted to capture a rather complex relationship with few characters. Of course you can define your own names, even using quotation marks and small and cap characters if you want to. For example, you could have written:

```
CONSTRAINT "Patient Foreign Key in Medication Table" FOREIGN KEY ("Patient ID")
REFERENCES "Patient" ("ID Number")
```

At this point you could be asking: Why do we need to name CONSTRAINTS just like we name Tables and Columns? Because in the future, as you optimize your database or adapt it to a new business practice, you might want to change the behavior of a particular constraint, maybe even drop it altogether (but without erasing the data you have entered in the tables so far). This would be impossible if you don't have a way to reference the constraint that you want to modify. The easiest way for doing this is to give a name to the constraint when you create it.

Instructions that shape the type of CONSTRAINT you are building include: PRIMARY KEY, FOREIGN KEY, REFERENCES and CHECK.

Let's see other examples (go and find the **`CREATE TABLE`** instructions that harbor these constraints):

- **`CONSTRAINT CK_END_DT CHECK( "End date" >= "Start date" )`**

This instruction checks that the “End date” column in this table always stores a value that is greater (newer) or at least equal to the value in the “Start date” column. This constraint received the name **`CK_END_DT`** which, in our mind, stands for “check end date”

- **`CONSTRAINT PK_PAT_MED PRIMARY KEY ("Patient ID", "Medication ID" )`**

This instruction establishes that the primary key of this table is formed by “Patient ID” and “Medication ID”, which is to say that this table uses a compound key formed by exactly these two attributes (columns).

- **`CONSTRAINT CK_DBT CHECK("Debit" IN ('DEBIT', 'CREDIT'))`**

This constraint is a bit more tricky to read. First we need to know that when using Base with the embedded HSQL engine, the names of tables, columns or constraints that are being referenced need to be enclosed in double quotation marks. Meanwhile, a string of characters (any string of characters) will be indicated by single quotation marks. With this in mind we can understand that the CK_DBT constraint checks that the column “Debit” can only receive the values 'DEBIT' or 'CREDIT' and will reject any other value (sequence of characters) that the user attempts to enter.

With this in mind now explain to me what this means:

```
CONSTRAINT CK_PAT_GNDR CHECK( "Gender" IN ( 'Male', 'Female' ) )
```

Don't forget to identify in what table this instructions appears in.

- **`CONSTRAINT IDX_PAT UNIQUE ( "Patient ID" )`**

This constraint, that we named IDX_PAT, makes sure that the “Patient ID” stores unique numbers. Should we want to eliminate this restriction in the future, we could go to the “Execute SQL Statement” window (found in the Tools menu under “SQL...”) and run the command: **`DROP IDX_PAT.`** However, don't do this. We want our primary keys to be unique.

If you happen to declare a constraint instruction that references a table that you have not yet created, Base will prompt an error message. This is relevant for constraints that reference other tables (than the one in which is being declared) like Foreign Keys. Review the SQL code again and note how the order on which tables are created avoids this problem, In short, when referencing other tables, you need to make sure that they already exist.

I recommend that you go back to the instruction set and read the CONSTRAINT instructions again with this new information. Don't they make more sense now?

Let's focus now on the way to define the columns inside the CREATE TABLE command. Let's review the following examples:

- **`"Transaction Description" VARCHAR(50)`**

This instruction creates a column called “Transaction Description” (note the use of double quotation marks for the name) which will accept **up to** 50 alphanumeric characters. Aha! Now is when that long and boring analysis of variable types we made before starts to make sense. Can you tell what happens if

I make an entry to “Transaction Description” that only uses 23 characters?

- **`"Debit" CHAR(6) NOT NULL`**

This instruction creates a column called “Debit” that accepts **exactly** 6 alphanumeric characters. It then adds the instruction “NOT NULL”, which means that Base will make sure that you enter a value here. Can you describe the other available conditions we can set to our columns?

- **`"Patient ID" INTEGER NOT NULL`**

This instruction creates a column called “Patient ID”, assigned to receive integer numbers. Can you tell me what is the biggest number I can enter here? Could I enter a negative number? Later I add the instruction “NOT NULL” to make sure that there is always a “Patient ID” value.

- **`"Amount" DECIMAL(10,2) NOT NULL,`**

This instruction could strike you as strange. You can easily read that we are creating a column called “Amount” that is to be not null and that is of the decimal type; but what are those numbers within parentheses? In alphanumeric variables you specify the number of characters you want inside parentheses but Decimal is a number type of variable, not alphanumeric. Well, the answer is that you can specify the size of the column for any type of variable, be it a number or alphanumeric. You can also define the precision, if you need to, with the number after the coma inside the parentheses. In this case, we have defined that the Decimal variable type (which can store very, very large numbers) used in “Amount” be organized in a column 10 digits wide and that it can store numbers with a precision of one hundredth of a point, that is, decimal places that range from 00 to 99 (which are two decimal places, hence the number two). In other words, “Amount” displays in a column of 10 numbers, where the last two numbers represents 1/100 numbers. If you were to enter the number 12.037, Base will store 12.04. This definition makes a lot of sense when you use a column to store money amounts. What happens when I try to store a number like: 987654321012 (other than that I become rich)?

- **`"Medication ID" INTEGER GENERATED BY DEFAULT AS IDENTITY(START WITH 0) NOT NULL PRIMARY KEY`**

This is an important pattern as it defines primary keys. This instruction generates the “Medication ID” column and assigns it an integer variable type. It then adds the instruction that it should be generated automatically, the first entry being number zero, it should be NOT NULL and is the Primary Key of this table. Although not explicit, this pattern also ensures that this column has unique numbers.

At this point you could be asking: Do I now need to create a CONSTRAINT that identifies “Medication ID” as a Primary Key? No you don't. This has already been done in the definition of this column. I am sure that you agree with me by now that, many times, there is more than one way to do things with Base.

- **`"Start date" DATE DEFAULT CURRENT_DATE`**

This is an interesting declaration and we will review it again later when talking about forms and reports. The first part is very easy to understand: create a column called “Start date” that will store a DATE variable. The DEFAULT setting makes sure that this variable is populated upon creation. But,

populated with what? With the content of the CURRENT_DATE function. CURRENT_DATE provides date information kept by your computer's internal clock. When we later create a new record (using forms), the default instruction will retrieve the value stored by CURRENT_DATE and store it in “Start Date”.

Let's wrap-up all what we have seen so far in a more formal definition of the **`CREATE TABLE`** instruction.

First, a “create table” instruction starts with **`CREATE TABLE`** followed by a name for the table. Then, within parentheses, you specify one or more column definitions and one or more constraint definitions, all separated by comas. Finally you indicate the end of the instruction with a semicolon.

If you were looking in the Wiki or other source for information on the **`CREATE TABLE`** command, you would come across a formal representation of the previous definition, which would look like this:

```
CREATE TABLE <name> ( <columnDefinition> [, ...][, <constraintDefinition>...]);
```

The first thing you will notice are the angle brackets (“<” and “>”). They indicate that “name” or “columndefinition”, etc., stand for names, or values, that you are going to chose, whatever those names or values will be. Then we have the square brackets (“[“ and “]”). They enclose instructions that are not mandatory, that is, that we can chose to include or not. So for instance, this formal definition implies that we need at least one column definition but having constraint definitions is optional. Lastly, the ellipsis (those three points together, “...”) mean that the last instruction is repeated, that is, that we can make more column definitions. Because it is enclosed between square brackets, in this case it also means that having more than one column definition is optative. Note the coma before the ellipsis. This reminds you that you must separate column definitions with comas.

Why are we going through this? Because after you finish this tutorial, you might want to know more about Base and the SQL instructions it uses. When you review the Wiki and other sources, this is the syntax that you are going to find.

Now, “**`columnDefinition`**” and “**`constraintDefinition`**” also have their formal representations. Let's see:

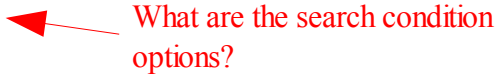
Column Definition:

```
<columnname> Datatype [(columnSize[,precision])]
    [{DEFAULT <defaultValue> |
    GENERATED BY DEFAULT AS IDENTITY
    (START WITH <n>[, INCREMENT BY <m>])}] |
    [[NOT] NULL] [IDENTITY] [PRIMARY KEY]
```

You should be able to read that after the column name, the only required parameter is the data type (integer, varchar, etc.). You can define column size and precision if you want to (and even precise column size without defining precision). What does that vertical line (also called pipe, “|”) stand for? That means that you have two or more options for a statement. For example, in defining a column you can write: **`GENERATED BY DEFAULT AS IDENTITY (START WITH 0)`** or you can write: **`NOT NULL IDENTITY PRIMARY KEY.`** Of course, if you wanted the primary key incremented by a value other than 1, then you would want to use the first option and include the increment that you want. So, in short, the pipe stands for a logical OR.

Let's now check the very powerful constraint definitions:

```
[CONSTRAINT <name>]
    UNIQUE ( <column> [,<column>...] ) |
    PRIMARY KEY ( <column> [,<column>...] ) |
    FOREIGN KEY ( <column> [,<column>...] )
    REFERENCES <refTable> ( <column> [,<column>...])
    [ON {DELETE | UPDATE} {CASCADE | SET DEFAULT | SET NULL}]
    CHECK(<search condition>)
```



Can you understand this formal representation? If you need help, you would find all you need in this tutorial!

3. What SQL window are you, anyway?

So far we have described two instances in Base where we can input SQL commands: the "Queries" page and the Command box. Throughout this chapter we have described some differences between them that I want to summarize here:

First, the "Queries" page is reached by clicking on "Queries" in the narrow column to the left under the heading "Database" while the Command Box is found in the "Execute SQL Statement" window called by the SQL icon in the Tools Menu.

Second, the instructions in the "Queries" page do not transform your data or data structure. This window only receives instructions on how to display the information that you already have in your database. It checks to see if your SQL statements start with the command "`Select`" and will prompt an error message if they don't. Meanwhile, the instructions through the Command box can transform your data or data structure, that is, you can create, delete or modify your tables, columns and constraints and, in the process, delete data. You can also input instructions that create, delete or modify records. The Command box also accepts "`SELECT`" commands.

The third important difference has to do with the way they handle names. As you know by now, tables, columns and constraints receive names that you assign in order to reference them later. The proper syntax requests that those names be enclosed in double quotes, particularly if you are using mixed case names and spaces (like "Patient ID"). Now, mixed case names defined in the command box that do not use quotes are silently converted to all uppercase but mixed case names used in the "Queries" page that are not enclosed in quotes are passed as they are typed, without a conversion.

This can create some confusion and you better be aware. Let's say that you define the following table through the Command box:

```
CREATE TABLE TransactionType (
Tran_Type_ID INTEGER GENERATED BY DEFAULT AS IDENTITY(START WITH 0) NOT NULL
PRIMARY KEY,
Tran_Description VARCHAR(50),
Debit CHAR(6) NOT NULL,
CONSTRAINT CK_DBT CHECK("Debit" IN ('DEBIT', 'CREDIT'))
);
```

and you later, through the same Command box, issue the following instruction:

```
SELECT * FROM TransactionType;
```

Both sets of instructions should execute without error. However, when you are later creating a report in the "Queries" page and issue the same instruction:

```
SELECT * FROM TransactionType;
```

You would receive the error message: "Table TransactionType does not exist". What happened here?!

If you notice, the **CREATE TABLE** command in this example does not use quotes when assigning a name to your table, so although you named your table "TransactionType" (without the quotes), Base really assigned it the name "TRANSACTIONTYPE" (all upper case). When you issue the **SELECT** command through the Command box, although you again used mixed case, Base silently converted the name to all caps and didn't have trouble finding the table. However, in the "Queries" page you repeated the name as you wrote it but this time Base did not convert it to all upper case, despite the fact that you didn't use quotes to encase the name. So now the application is really looking for the table "TransactionType" and not finding it, because you have really created the table "TRANSACTIONTYPE".

The same thing is bound to happen with the column names "Tran_Type_ID", "Tran_Description" and "Debit", which have also been defined without their double quotes.

Notice that the **CONSTRAINT** defined in the example uses a name with all uppercase and no quote signs. The silent conversion that will take place here is irrelevant because it leaves the name the same.

You can chose to name all your objects (tables, columns, constraints) with uppercase and forget about quotes but it is considered better practice to use quotes. This practice also gives you a bigger set of possible names. In this tutorial, constraints are the only objects that we will name using only all caps.

Recapitulating and to be more precise, any string in the Command box that is not enclosed in quotes will be transformed to upper case. For example:

```
DROP TABLE "Patient" if exists
```

will be converted by Base to:

```
DROP TABLE "Patient" IF EXISTS
```

which gives you a good hint on why the Command box behaves this way.

4. Producing our forms

At this moment we have a sound database design, which took a lot of thinking, and that we have encoded using SQL language to create the tables and their relationships with Base. Now we are going to focus on creating forms that will help us populate these tables swiftly and minimize errors in data entry.

If you have done your homework and studied other manuals and tutorials about forms with Base then you know that there are many options and tools available for producing these forms and also several ways to accomplish one particular task. We will not be able to review all options in this tutorial but we will make sure we come up with appropriate forms and that you understand *why* and *how* we are creating them.

The first thing will be to remember the considerations mentioned in the first part of this tutorial regarding the need to make forms that are unambiguous on what data are we supposed to provide for each field and to minimize errors in the input by offering pre-made options that are easy to indicate. In this tutorial we will focus on the following ways to simplify or automate data entry that are very useful: radio buttons, drop down lists, sub forms, default values and some tricks for time entry. We will further describe these options as we work with them.

Let's get to work.

On the Base document where you have the tables for our psychotherapy clinic, click on the icon to the left over the word “Forms”. You will see the following options appear in the *Task* panel (upper panel to the right): *Create forms in design view* and *Use wizard to create forms*. However there is still a third way not made explicit yet: Defining forms using SQL. At this moment it seems necessary to explain something:

Using SQL for creating tables or entering data requires that we know SQL. In contrast, using the Wizard just requires that we understand what are we being asked and chose the appropriate options. The result using SQL will be as good as the underlying engine that we are using, in this case the HSQL embedded database system. In all merit, HSQL is a very good implementation of the SQL standard. Using the wizard or design view, on the other hand, will only be as flexible as the software mediating our options and the resulting SQL code can be. In all honesty, the coders at OpenOffice.org have done a terrific job developing this easy to use graphic interface, but for its very nature, graphical interfaces can not always accommodate the complexities and intricacies that raw code can denote. Also, there can always be a bug here or there that could impair the ability of the software to achieve a particular goal (which is why it is so important to report bugs and help achieve better code). So we will be forced to use SQL code when the Graphical Interface does not provide us with an option that we need or when we suspect that there could be a bug affecting us. This means, again, that in order to develop robust databases, learning SQL is necessary. In short, SQL code gives us flexibility and power, design view and the wizard give us ease of use. It will not come as a surprise to you that in this tutorial we will be using these three options.

In general, we will use the following approach to building forms: First we create forms using the wizard, which will help us get up and going quite quickly. Later we use Design View to customize them, clean them and leave them easy to read and understand. Finally we add any SQL instructions required for more complex functionalities, if necessary.

The Form Wizard.

I want to start with the form for the information on the psychiatrists. First, because it is simple enough not to distract us from the process of building and cleaning the form, something we will cover here and later just assume that you are repeating with the other forms, and second because it allows us to explore how to use radio buttons for data entry.

So, let's recap: click on the icon over *Forms* in the left most column in your display, the one under the heading *Database.* Now choose *Use Wizard to Create Form...*

The first thing that will happen is that a Writer document pops up, with some extra controls on the bottom, and -over it- the Form Wizard.

So this is the big surprise, a form in Base is really a Writer document, the one you use for editing and printing documents. Of course, the elements that we will include in this document, like text boxes and radio buttons, need to somehow be connected to the database that we are working on. Don't worry about this as the Form Wizard will take care of this connection. But the fact that this is a writer document means that we can enjoy all the flexibility usually available for organizing the layout of our written documents. If you remember, the draft forms I made for part II in this tutorial were worked with Writer -although at that moment they were not functional and were made only to illustrate the layout- which means that, if we take the time we can replicate them completely if we want to, now fully functional and all.

The Form Wizard is organized in two columns: the one to the left names the different steps the wizard will walk us through. The panel to the right will ask us the questions that each step needs to know about.

In the first step we need to indicate where the fields that we will use are coming from, that is, what columns from what table will be associated to this form. In the drop down box you will find a list of all the tables available. Find the option “table: psychiatrist” and select it now. Note that on top of the drop down box the caption reads: *Table or Queries.* This is because the result of a Query is also composed of columns and you can use them with the wizard as if it were another table. (We will be using this later).

Upon selecting the psychiatrist table you will see that a box in the bottom to the left populates with all the column names that belong to the table. We are supposed to select all or some of these for which we want a field in our form. For the sake of simplicity let's just pick them all. You do this by clicking on the button with the sign “>>”. Do this now.

Now all the names for the columns have shifted to the box on the right, meaning that there will be an input field for each column.

If you had wanted input fields for only some of the columns, you would have had to select them with the mouse and then click the button with the “>” sign. Let's say that you see no reason for a field for the ID number -because it will be generated by Base automatically because the user does not need to know this value and because you don't want to leave it exposed to an involuntary (and fatal) change. It this case you would need to click every other name and then click on the “>” button, every time for each field. If this were the only attribute for which you don't want a field, you could also move all the column names to the right (with the “>>” button), select the ID number and then click on the “<” button, saving you some time.

However, for the time being, select all column names and then click on “Next”

The second step ask us if we want to set up a sub form. We will deal with sub forms later and the Psychiatrist form does not need one, so just click on “Next” without changing the content here.

Now the wizard jumps to step five and ask us what layout we want for the fields. The default uses a table configuration (third icon from the left), with the name of the columns in the header (called *Data sheet*). We will use option number one (called *Columnar -Labels Left*). Experiment clicking on the other options to get a feel of the alternatives.

Step number six creates restrictions about the use of the data by the form, either allowing the view and modification of all data or not allowing the data to be viewed, modified or some combination of these. The options are self explanatory and we will not be using this yet, so we will advance to the next step leaving the default: *This form is to display all data,* with no restrictions.

Step number seven allows you to chose the background color for the form and the style used by the text boxes that conform the entry fields. When you choose a color for the background, Base chooses colors for the fonts automatically. Your selection here will have no bearing on the functionality of the form other than make it easy to read. I believe that the offered combinations are well chosen and I like dark backgrounds with white characters. But that is me, you choose whatever you want.

The last step will ask us for a name for this form and what we want to do with it. The wizard will offer, as a default, the name of the table, in this case "Psychiatrist". Many times this is enough but I would recommend to use a more descriptive name. Some people would add the letters "frm" before the name, to make explicit the fact that this is a form. I invite you that, this time, you change the name of the form to: "Psychiatrist Data Entry Form" just to see what happens. Later, you can use your own naming conventions. Finally, Base wants to know if it should offer the table for data entry or if it should open it in Design View for further editing. We will take the second option. Click on it and then click on "Finish".

At this moment, the wizard closes and the form opens again, this time with text boxes for data entry and corresponding labels. The form has opened in Design View and is ready for our editing. You can also notice dotted lines forming a grid pattern. This is to make it easier to design and position elements on the page.

Design View and the Properties dialog box.

If you click on any of the text boxes or their accompanying labels, you will see small green boxes appear defining a rectangular limit. This limit includes both the label and the text box. If you try to move one, the other will drag along; if you try to modify one, chances are that you will somehow modify the other too.

At this moment I want you to double click with the left button of your mouse. A dialog box appears with the name *Properties: Multiselection.* This properties box allow us to modify the characteristics of the elements we place on our form, and will become very handy just now. Multiselection means that there are two or more elements alluded by the properties box because they have been grouped together. This is why the label and the text box are traveling together and changes on one can change the other too. If you now click elsewhere on the form, the group becomes un-selected and the Properties dialog box becomes empty, stating that no control has been selected. You can now click on another element and its properties will become displayed by the dialog box. If you close the box, you can call it again by left-double clicking the mouse while on top of an element.

Close the dialog box for now and then click (once) over an element with the mouse button to the right. Now we see the green squares again, indicating the active element, while the more familiar contextual

menu appears. Near the bottom of the menu you will find the item “Group” and within it the options ungroup and edit group. Choose ungroup.

If you click on an ungrouped element now, you will see that they display the green boxes but only in relation to themselves, not extending to include the other elements. If you call the properties dialog box, you will see that it offers options that are particular to the selected element, in this case, either a label or a text box.

Let us now analyze what we see on the screen: there are ten pairs of labels and text boxes. Most of the text boxes are of the same length except for two, the one after *ID Number* and *Date of Birth*. If you analyze the SQL code that created this table you will see that all the boxes with similar length are for some kind of text entry, while *ID Numbe*r and *Date of Birth* are integer and date variables respectively. So maybe Base chose the length of the boxes depending on the type of Variable they are. We might want to change these lengths in order to make the form less intimidating and easier to read.

The other thing you can notice is that the labels describing each box give a quite accurate description of the kind of data entry that we expect. How did Base know? If you look at the SQL code that created the table, again, you will see that Base actually copied the name of the column as given by us. Because we took the time to use double quotes, gave descriptive names and used spaces between words, now these descriptive names appear to the left of the boxes. If instead we had used more cryptic names for the columns, like PK_FST_NM, then the labels would be displaying just that. This would not be a problem however because we can easily change the caption in the label.

Select the pair with “Number ID” and ungroup them. Now select the text box only. You can position the cursor over it and, by dragging the borders, make it wider or higher. You can also reposition the box anywhere on the form by dragging it while the cursor looks like a cross with arrow heads. Nice! With these skills you can, for example, reduce the *Postal code* and *State* fields -they don't need more than five and two characters respectively- and place them side by side so the person confronted with this form finds it more familiar and less of a chore.

Now select the label element and call it's properties dialog box. This time, the box has two tabs: *General* and *Events*. Through the *Events* tab you can program behavior in response to the events named in the dialog box. This requires some macro programming and we will not be using this feature in this tutorial. With the *General* tab we can change other properties. The Name property is automatically provided by Base and it is used to refer it with internal code (connecting the form with the database). The second property corresponds to the caption that we actually read. Change “Number ID” for “Psychiatrist Primary Key” and then hit return. The cursor goes down to the next property in the dialog box and the value is immediately changed in the label. This also happens if you take focus from this field. Try it.

I feel that the default size for the font in the labels is somewhat small. Let's change this. Find the property *Font* in the dialog box and click on the “...” button. This will call a dedicated Fonts dialog box. The first tab, “Font” allows us to change the font, the typeface and the size of the text in the label. The next tab allows us to include other effects, including changing the color of the font. Feel free to experiment with this for a while.

Chose a font size of 12 and see what happens. The text in the label is now bigger but does not display completely. This is because the boundaries set by the green squares did not change after we increased the size. Drag the squares until you see the complete text of the label. As an exercise, change the font

size of every label in this form to 12 and adjust the length of the text boxes to something commensurate with the amount of data they will receive.

Now scroll to the bottom of the properties dialog box. The second to last property reads “Help Text”. By writing here you activate and populate the tool-tip. This is a yellow rectangle with helpful information that appears when you place the cursor over an element on a form. You find many examples of this in professional software or the Internet. Base automatically does this for us; we only need to insert the useful information. That is what you type here. For example, in the box for “Name” you could write: “Enter the first name of the psychiatrist”. You can now save the form edit, close it and call the form. Pace the cursor over the box and see the tool-tip appear.

One important way to make forms more gentle to the eyes of the user is to limit the amount of fields to the ones he will actually have to populate. As mentioned before, there are good reasons to eliminate the Number ID field and have this form less crowded. But wait! Would not this interfere with the ability of Base to include the auto-generated primary key number? It will not. Base will continue to generate unique primary keys and include them into the records. What we are doing here is to make explicit the fields that the user is shown or not. The automatic generation of primary key numbers was coded with the creation of the table itself and is not interfered by this.

Now that we are getting a hang on the usefulness of the Properties dialog box, lets use it to program our radio buttons.

Radio Buttons.

Radio buttons offer the user a set of mutually exclusive options. The user can chose only one of them. What we want to achieve is that the elected option is entered to the corresponding column in our table. Now, pay attention: The options for radio buttons better cover all possible alternatives. If not, then at least the user should find the option to leave the field unanswered. Gender is a good candidate for using radio buttons. Granted, male and female are not all the possible options (hermaphrodites, among other less common options) but will correspond to a quite high percentage of the users. In another context (a database for gender research, for example), however, this could be insufficient. So plan your radio buttons carefully. Another place where radio buttons can be used to an advantage in this example is in the table to register if the session took place or not (see the form layouts).

The first thing that we need to do is actually obtain the radio buttons. You will find them in the Form Controls Menu. This menu usually pops up automatically when you open a form in Design View. If not, go to “View” menu option in the writer document where we are editing our from and select “Toolbars” (third from the top) and click on “Form Controls”.

Now a floating menu appears, with the name “Form ...”. This is a small menu that offers several icons. The icon of a circle with a black spot is our radio button. If you leave the cursor on top of it, a message appears (a tool-tip, actually!) with the name: “option button” which is how Base calls the radio buttons.

If you left-click on this icon, the cursor changes to a cross with a small box. This allows you to place the radio button somewhere in the form page. You can actually place it wherever you like. Just for clarity with this example, try to place it next to the text box with the “Gender” label, by dragging the cursor and releasing the left mouse button.

If things happen like in my computer, you should see a radio button with the caption “option button”. If

you chose a dark background color (like I did), the caption in black will be difficult to read. To fix this, call the Properties dialog box for the radio button. In the general tab, change the *Label* to “Male”. Then click the *Font* button, select “bold” and change the font size to 12. In the “Font Effects” tab, change the font color to white. That should be much better.

Let's now program the behavior that we need. The Properties dialog box for a radio button offers a tab that we have not seen yet: The Data tab. If you select it you will see three fields: Data field, Reference value (on) and Reference value (off). Data field indicates what column this control will reference. Because the wizard built this form for the psychiatrist table, you will find the name of all the columns for such table. Click on the list box and see them appear. Of course, select “Gender”. The Reference value (on) holds the value that will be passed to the Gender column if this radio button is selected. Write “Male” there.

Beware!: if you check the SQL code that built the Psychiatrist table you will find a **`CONSTRAINT`** that only accepts “Male” and “Female” as possible entry strings for the Gender attribute. Note that the initials are capitalized. If you write “male” (without the capitalized initial) in the *Reference value (on)*, Base will report an invalid entry error every time you select this radio button.

Now on your own, create a second radio button. Change the label to “Female” and make it readable, relate the radio button to the “Gender” field and write “Female” in the *Reference value (on)* field.

Having created the second radio button, align both buttons so that they are at the same level (use the grid to guide you) and chose a harmonic spacing between them. Making it pretty makes data entry a bit more enjoyable.

At this point we no longer need the text box for entering the gender: the radio buttons will take care of this. But we are going to leave it for a while so that you can see the radio buttons in action. First, ungroup the label and the text box and make the box smaller (enough for 6 or 8 characters). Now take the box out of the way and place the radio buttons in its place (you can group them and move them as one element). For reference, leave the text box near by.

If you don't know how to group elements, here is how: Select one element by right clicking once on it. Next, press the *Shift* key and select another element. Now both elements appear under the same area defined by the green boxes. Repeat this to include as many elements as you need in the group. Right click the mouse while the cursor is a cross with arrow heads and in the context menu that appears, click on “Group...” and select “Group”. Now your elements are grouped.

This would be a good moment to save your work (lest a crash dissolve it into ether). Click the save button but then also close the form. When you go back to the Base interface we will find this document under “Forms”. Double click on it to use it.

Now the form appears again, but without the grid lines. This form is active and any data or alterations will end in the corresponding columns. You are actually in the position to start populating the Psychiatrist table! If you click on either radio button, you will see the corresponding text appear in the text box. This means that this record has accepted your gender input! If you change your mind and click the other radio button, the form automatically changes the value and also erases the dot in the radio button previously selected, placing it in the new option. This happens because both radio buttons have “Gender” in their *Data field* and radio buttons are programed to be mutually exclusive.

Most people that do data entry like to use the keyboard to navigate the form. If the cursor is in the first field, you can travel to the next by pressing the Tab key. They enter the value for the current field, press the Tab key and move to the next field. Try this.

I am sure that you noticed that every thing works fine except for the radio buttons. The navigation of the form just skips them. Currently a radio button can only be accessed by the Tab key when it has been selected. A group of radio buttons where none is selected cannot be accessed by the keyboard. I have found that even if you leave a radio button selected by default, that particular radio button is integrated into the tab order, but the others are left out, which does not help to change a selection. Also, the button is not necessarily left in a consecutive order, which is not elegant. This last thing, at least, we can fix!

Let's go back to Edit View: Close the active form and go back to the Base interface. Right click on the Psychiatrist's form and chose edit. The edit view of the form, grid pattern and all, should reappear now.

If you call the Properties dialog box of any ungrouped element you will see in the General section two properties: *Tabstop* and *Taborder*. The first one indicates if you want this element to be part of the tab order. For instance, we are using the Gender field placed by the wizard as a window to check the value assigned to Gender. We don't need the tab to stop here so we could, with all confidence, set the Tabstop property of this element to “No”. The *Taborder* indicates the sequence of tab stops in the form. You give each taborder property of every Properties dialog box of every element in your form a consecutive number starting with zero. Each press of the tab key will move the cursor in such order.

There are other ways to do this: For example, you can select all the elements in your form and then click on the Navigation icon (it looks like a form but with a compass in the front). This function organizes the tab order following an up-to-down and left-to-right order. Most times this arranges the sequence completely, other times some minor tweaking by hand is required.

Now, how do we make one of the radio buttons be selected by default, say, the “Female” option? For this to happen, you need to ungroup the radio buttons and call the Properties dialog box for the radio button with the female option. In the General tab you will find the *Default status* property. Check “Selected”. Immediately, a small black circle appears, indicating that this options is the current one.

As we said, we do not need the gender field box, and you should erase it. But before you do I want yo to study its Properties dialog box. Don't worry about the “Events” tab. Focus on the “General” and the “Data” tabs and study and play with the options. This teaches plenty.

Let's finish editing our Psychiatrist Form. By now you should have eliminated the “ID number” and “Gender” text boxes, made the labels bigger, resized the remaining text boxes to be more commensurate with the amount of data they will typically receive and organize them in a pleasant and easy to read manner. In the Toolbar select font size 26, font color white and center text. Then write: “Psychiatrist Data Entry Form” at the top of this page. Well, this is looking very nice.

At this moment you could produce the Medical Doctor Form because we are going to need it next and because it is very similar to the one we just made. This should be good practice.

Sub Form Fields.

Let's now develop the Patient form. This form imposes two new challenges: First, the need to record an unspecified amount of phone numbers for each patient and, second, the proper assignment of medical

doctor and psychiatrist. If we do not enter the data for medical doctor or psychiatrist exactly the same way we have them in their own tables, Base could never find the records and thus would provide wrong information when queried. To prevent this we are going to use a drop-down list that will automate the correct input.

The problem with the phone numbers for the patients is that each patient will have a different and unknown number of phone numbers that need to be recorded. True to first normal form we had decided to place these phone numbers in their own table, using the patient primary key as a foreign key in order to connect each phone number with its proper owner. The cardinality for this would be: one patient, one or more phone numbers, or 1..n.

Later, when we want to retrieve the phone numbers for a particular patient, we have to make a query that, more or less, reads: retrieve all phone numbers that have *this* primary key (belonging to a patient) as a foreign key. For this to work, we need to make sure to include the primary key of the current patient while recording his/her phone numbers.

In the draft we made for the patient form, we had devised a button that would call another form where we would record the phone numbers. This form would need to handle the recording of these relationships. The truth is that Base provides a solution that is by far more simple. This solution allows us to include a second form inside the primary form for tables that hold a 1..n relationship, where the sub form is for the table at the n side; and will automatically record the primary key of the current record at the principal form, in this case, the current patient. This is what sub forms do. Obviously, we are going to use this option.

Let's start by calling the form wizard and select the patient table. Next, select all column names to receive input fields.

In the second step we are asked if we want to set up a sub form. This time click the checkbox for “Add Subform”. Then select the option that reads: “Subform based on existing relation”. Base offers you the tables that have a defined connection with the patient table (as established by the SQL code creating the tables that defined relationships). In this case you are offered to choose between *Payments* and *Phone number*. Select *Phone number* and then click on “Next”.

Now the wizard offers you all the column names in the Phone Number table for you to decide which ones will appear on the form. In this case there is *Phone ID, Patient ID, Number* and *Description. Phone ID* is the primary key of the record and *Patient ID* is the foreign key we need to associate for this to work. It would be a bad idea to include them as fields and risk someone altering the numbers. We already know that not including them will not stop Base form recording the data properly, so my recommendation here is that we don't include them. We will just select Number and Description, which is the data the user really needs to work with. After you have passed them to the box in the right, click on “Next”.

Because Base is going to calculate the joins for itself, the wizard is going to skip step four and land us directly on step five. Here we will select the layout for the form and the sub-form. This time we are going to select the *Columnar -Labels left* for the principal form and *Data Sheet* for the sub-form (options one and three from the left, respectively).

The reason for choosing Data Sheet is that it allows us to see a good number of the recorded phone numbers for the particular patient without having to navigate the records. Anyway, feel free to

experiment.

Leave the default in step six (*The form is to display all data* -with no restrictions) and choose the style of your liking in step seven. In step eight chose a name for the form (I used Patient Data Entry) and ask the form to open it for editing.

[At this moment you could customize the form repeating the steps described before: Make the labels readable and adjust the size and layout of the text boxes. You can also take the field for the primary key out of tab order and make it to be read only, so no one inadvertently changes it (find the option in the Properties dialog box). Then, use font size 26 to write "Patient Data Entry Form" for a title and, hitting the <Enter> *key several times, place the cursor above the sub-form and type "Phone Numbers:"].*

To begin with let's arrange the sub-form. Left-double click on it. The green squares will appear, indicating that it has been selected and its Properties dialog box shows up. Analyze the General tab to have an idea of the options offered.

The first thing we can do is diminish the width of the sub-forms to make it less intimidating. You can do this by dragging one of the green squares on the sides, just like with the text boxes or the labels. Of course, changing the height will change the number of phone numbers in simultaneous display. The next thing that you can do is adjust the width of the columns for *Number* and *Description*. Remember that the phone number has been defined for a maximum of 10 characters and that if you include spaces while making an input and exceed the 10 number limit, Base will will reject it. Calculate a width accordingly so it gives the user a hint.

Close the design view and open the form. You will see that the sub-forms is not active at first. You can notice this because it does not offer you the chance to input a record. This is because the data in the sub-forms relates to data in the principal form but the principal form has no records yet. The principal form has the fields of First Name and Surname as required (i. e. NOT NULL. You can see this by reviewing the SQL code that created it). After you introduce the required data, the sub-form activates. You will see a green arrowhead indicating a yellow flash. You can now enter the number and its description, and navigate the form with the Tab key.

If you later decide to erase the record for the patient, Base will act depending on how you have handled deletions. If you had chosen Cascade Delete, for example, deleting the patient would delete all his/her phone numbers along with the record. If you had not made any provisions, Base will prompt an error dialog box instead. In that case you will need to manually erase the phone records and, only after that, erase the patient record. Give it a try. In order to manually delete a record press the icon with a red 'X' next to a green arrow head. If you haven't defined how to handle deletions yet, go to the "relationships" view (TOOLS > Relationships...) and find the line that connects the "Patient" table with the "Phone" table. Double click on it and a menu will appear. Select the "Delete cascade" and confirm.

Let's go back to design view and tackle our second challenge.

Drop Down lists

We have hinted that most times a unique number would be better than a surname as a primary key. When we associate a psychiatrist to a patient, what we are really doing is storing the primary key number -that identifies a psychiatrist- in the foreign key column of the patient table where psychiatrists

go. In other words, relating the appropriate psychiatrist requires recording its primary key in the field called *Psychiatrist ID* in the Patient Data Entry form. If you check the SQL code that created the Patient table you will see that the data type assigned to *Psychiatrist ID* is an INTEGER, which handles numbers. If you try to write "Dr. Jones" you will get an error. The form expects a value like "012", which should be the primary key for the record of Dr. Jones.

This means that we would need to keep track of the primary keys associated to each psychiatrist recorded in order to enter the right information. Who remembers this? This is not practical. What we really want is to have Base offer us the names of the psychiatrists but, once we have made our selection, really record only the key number. This is exactly what a drop down list can do.

At this point you have the Patient Data Entry Form in Design View. Select the Psychiatrist ID label and text box and un-group them. Now select the text box and delete it. Call the Form Controls menu and select the List Box icon, the one that looks like a rectangle with lines of text and *up* and *down* arrowheads to the right side. This icon is usually next to the radio button icon.

Place a List box to the right of the label. When you release the mouse button, the **List box wizard** appears. This wizard goes through three steps in order to identify the list box's source and destination values.

In the first step you are shown a list of all the tables in this database and are asked to select the source table, that is, the table from which the list box will gather its data. In this case you will need to choose the Psychiatrist's table. Then click next.

In the second step you are shown a list of the names of all the columns in the Psychiatrist's table (the chosen table) and are asked to indicate which one will provide the data to appear in the list box. In this case, chose the *Surname* attribute. This means that, when active, this list box will display all the *Surname* records found in the *Psychiatrist* table.

In the third step you indicate what data from the source table (the *Psychiatrist* table in this case) is going to what column in the destination table (the *Patient* table in this case). Please note that you are entitled to chose *any* column from the source table and any column from the destination table. They are all listed in their respective boxes. There is only one condition: the data type of the source must match the data type of the destination. This makes sense: it would be impossible to try to insert a VARVCHAR type of data into a TINYINT.

In this third step, the wizard shows you two boxes. Inside the boxes the wizard lists all the names of the columns in the corresponding tables. The box to the left, under the caption "Field from the Value Table" corresponds to the destination table (in this case, the *Patient* Table). When you chose a column here, this will be the receiving column. In this case you should select *Psychiatrist ID*.

The box to the right, under the caption "Field from the List Table" corresponds to the source table (in this case, the *Psychiatrist* table). The name you chose here will be the name of the column from which the values will be copied. In this case you should select *ID Number*.

After you have chosen a destination and a source, click on finish. This will close the wizard and leave you to edit your List box. Arrange position and size to your liking.

Call the List box's Property dialog box and study the attributes given to it by the wizard. The more

important ones are in the “Data” tab. Particularly, T*ype of lists contents* specifies the use of SQL and *List content* holds the SQL instruction to be executed.

For now, let's go and see how this table works. Close design view and call the psychiatrist table. Make sure to include some valid entries. Then, close this form and call the patient from.

When you click on the arrow head you will see the surnames of all the psychiatrists in your database. The beauty of this is that this list is prepared, by the SQL instruction that we saw, every time we use this control. This means that if you include more psychiatrists in your database, their names will also be included in the drop-down list. This list updates itself!

When you select a psychiatrist, it is the primary key number, not the surname, that is passed to the patient table. You can monitor this by creating a text box and associating it to the corresponding field. You will see it displaying the appropriate number.

However, you could express concern because the list box is displaying surnames only. With two psychiatrist with the same surname, which is the one I want to select? In order to decide, it could be good if the list box included the first name of the psychiatrist too. How do we do this?

Here is where we will need to use SQL again!

List Boxes with compound fields.

By now you know that the List Box wizard will ask you to chose a table and the name of one, and only one, column to populate the data. We have chosen “Surname” but now feel that it is not enough. We would like the List box to display both, the surname AND the first name, in order to make the selection clearer to the user.

The thing is, how do we produce a table -or something similar- that will have the first name and the surname of the psychiatrist table as one column?

This is where a *View* comes in handy. A View is the result of a query. As you know, the results of queries are displayed in columns and, consequently, can be thought of as tables. In fact, Base stores Views with the tables. So we need to make a query that will produce the result we want, transform that query into a view and then use the resulting view with the List Box wizard.

Queries are built with SQL code. Although Base allows us to create simple queries with the ease of point and click, this particular query uses a syntax available to the embedded HSQL database but not yet incorporated into Base. So we are going to produce it manually. Let's start.

First, in the Base interface, select the icon over the name “Queries” This is the second icon from top to bottom in the column under the name “Database”. Three options will appear in the top panel under the name “Tasks”: “Create Query in Design View...”, “Use Wizard to Create Query” and “Create Query in SQL view”. Select this last one.

A screen will open with a blinking cursor. Here is where you will enter the SQL code. Type the following:

```
SELECT "Surname" || ', ' || "First Name" AS "Name", "ID Number" FROM "Psychiatrist"
```

Actually, this code is quite simple except for those "|| ',' " or something like that between *surname* and *first name*. Let's analyze this statement:

This instruction commands the database to select *Surname*, *First Name* and *Number ID* from the *Psychiatrist* table. The double pipe is a concatenation instruction accepted by HSQL (but not yet accepted by Base). In this case, a double pipe is connecting the contents of Surname with a coma and a space and a second double pipe is connecting the previous with the contents of First Name, and all of these is to be placed inside one column called "Name". This is just what we needed.

Remember that with Base, single quotes are used for string literals, in this case, the characters for the coma and the space, and that double quotes are used for the names of tables, columns or constraints (unless they only use capitals and no spaces in those names, in which case the quotes are not needed).

So, if you ran this query you would see a table with two columns: The first one would have the surname and first names of all the psychiatrists in your table, separated by a coma (and a space, very important), and the second column would have the primary key number.

The editor in which you have typed this query has two buttons for running it: One that looks like two pages with a green tick where they overlap and another one that says SQL with a green tick over the letters. The first button runs the instruction by Base first, and because Base does not understand the double pipes, it will throw an error message of improper syntax. You must select the second button, which runs the SQL instruction directly. Press it and you will not see any changes in your screen, except that the button remains depressed and the other run query button becomes deactivated. Now save this query. I used the name "qryPsychiatrist" to remember that this is a query.

You can double click on the query and a table will come up. In my computer, and after previously inserting three fictional psychiatrists through the Psychiatrist Data Entry form, I have:

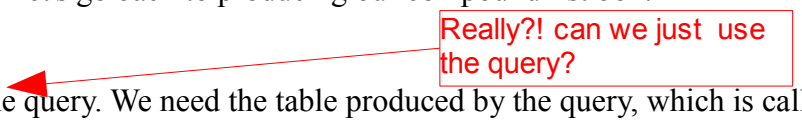
Name	ID Number
Smith, John	2
Perez, Adelaide	3
Lecter, Hannibal	5

My ID Numbers are not consecutive because I had made some deletions and HSQL does not recuperate numbers, it just keeps incrementing its counter.

Another important thing to know is that if any of the fields that were joined by the double pipe query command, in this case surname and first name, were empty, then the entire record in the "Name" column would appear empty too. That is, If I had not entered the first name for Dr. Perez, for example, my little table would look like this:

Name	ID Number
Smith, John	2
	3
Lecter, Hannibal	5

We have two options here: we include conditional clauses in the SQL command, looking for empty strings and instructing to include the non empty element, or we make sure that both, Surname and First Name are defined as NOT NULL so that they can not be empty when I need to use them. I will chose the second solution for simplicity. Let's go back to producing our compound list box.



For the next step we cannot use the query. We need the table produced by the query, which is called a View. You can think of a View as a virtual table. It is generated by the query when it's called, at which moment it becomes a table you can use. The good news is that this virtual table will update every time it's called, incorporating new records or deletions to the psychiatrist table. The bad news is that this calculation will add some time to the production of the form, making it slower the more records it needs to process.

To produce a View, right click the query we just created and, in the context menu that appears, select the option “Save as View”. A little box will ask you for a name for this view. I used “vPsyList”. After you confirm this, the box disappears and not much more happens. If you now go to the Tables section, you will see our new addition, with a particular icon showing that it is not a regular table but a view.

Now you can produce your list box as normally. Go to the Form section and select the “Patient Data Entry” form in edit mode. Create a new list box. When the Listbox wizard appears, it will include the “vPsyList” view along with your tables. Select it. Choose “Name” to populate the list box and chose “Number ID” as the record to be copied to “Psychiatrist ID” in the receiving table. TADAAAH!

Now, create a list box for the medical doctors that will also display both the surname and the first name. You will need to adapt the SQL code accordingly.

Default values.

The address component of the tables we have used in this tutorial all include a “State” attribute. One can expect that most of the clients for this psychotherapy clinic will reside in the same state. It is possible to have the forms display this state automatically and simplify the process of data entry for the user. Of course, if the user encounters a client that actually lives in another state and comes in for therapy, then he can change the value in the form.

Default values can be set by the Properties dialog box. Find the property “Default Value” and enter the string that you want. For this example I will use “NY”, which is where I live at the time of writing this tutorial. Now, the forms will include the “NY” value in the “State” field for each new record.

Default values can also be made possible by editing the tables. The big difference is that default values through the Properties box only affect the fields in the form. This means that if you decide to include records writing directly to the tables, there will be no default values in the assigned fields. However, if you assign default values through the design of the table, these values will appear in both, the forms and while using the tables to enter data. Let's see how to do this:

First, click on the Tables icon and right click on the “Patient” table. In the context menu that appears, click on “Edit”.

The table now opens in Design Mode. You can see a list of all column names and the data types each accept. In the lower part you can set some parameters for the variable, which are specific to each data

type and will change when you change the kind of variable you select.

Find the “State” column name and select it. The bottom of the page shows the parameters specific to Text (Char) data types. The third parameter from top to bottom is *Default value.* Here we will type “NY”. Now save the table and we are ready. If you open the form you will see that the next record already has NY in the field for State. Of course, if the user needs to change the value to “NJ”, for example, all he needs to do is overwrite on the field.

Entering time and date.

The patient table has two columns that store date information: the DATE for registering the birthday of the patient and a TIMESTAPM for registering the moment she/he is admitted as a patient. You might remember that DATE records year, month and day information while TIMESTAMP records year, month, day, hour, minute and second. To expand this example let me also reference the “Schedule” table, which is where we record the next session for a patient. This table records both DATE and TIME information. TIME records hour, minute and second.

Our goal here is to either set some date information automatically or help the user minimize error at entry.

Let's work the first approach. You tell Base to automatically set date or time information when you are building your table (with SQL code): first you create the type of date variable you need and then you specify that, as a default, it should receive the current date. HSQL has the following built-in functions for doing this:

`CURRENT_TIME` fetches the time at the moment this function is being used,
`CURRENT_DATE` fetches the date and
`CURRENT_TIMESTAMP` fetches both date and time information.

So your code would look something like this:

```
“Moment of registry” TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
```

In fact, if you analyze the SQL code that created the tables for this tutorial you will find some columns for date information that use this code.

Let's analyze this command: The string inside the double quotes is the name of the column. Then we assign a TIMESTAMP data type to it. Later, we specify that it will take a default value with the instruction “DEFAULT” and finally we call the “CURRENT_TIMESTAMP” function, which fetches date and time information from the internal clock of the computer. The final coma assumes that this is not the last instruction in the creation of the table. If it were, just omit it.

When you run the form wizard with a table that includes **`TIMESTAMP,`** it produces what seems like a box with a line inside separating it into two fields. The truth is that these are a time box and a date box which have been grouped.

While the user is filling the form, the **`TIMESTAMP`** box does not show any particular behavior. But when you come back to a record you have entered, you will see that the **`TIMESTAMP`** box displays date and time information. In theory, if you change any information in a form, this change should modify

the timestamp to reflect the time that the record has been updated. In practice I have seen that the **TIMESTAMP** box does not change if I modify the records and achieving this would require some Java programming, which is beyond the scope of this tutorial. The user can do it manually, however.

Now read and interpret the following:

```
"Time of registry" TIME DEFAULT CURRENT_TIME,
"Day of registry" DATE DEFAULT CURRENT_DATE,
```

These instructions would create columns that would automatically insert time and date information, respectively, when a record is created. In my experience, these instruction also do not update if you modify a record and the user will need to manually adjust them, if required.

However, there are some times when you do not want the date and/or time records modified. This is particularly true when your database stores historical information. For example, at what moment was one patient assigned to a therapist is such kind of historical information. When the patient was transferred to another therapist or when he stopped being a client of the clinic are all historical information which you would want to preserve, for administrative reasons. Not having them modified automatically would be a good thing. What you need to do now is prevent the user from modifying them manually.

In order to accomplish this you have several options. First, you could set Date or Timestamp by default and not display the corresponding field in the form. Remember that this automatic assignment is accomplished through the instructions that created the table, and not displaying the field does not stop the process, it just prevents the user from modifying the data. Later, when you are producing a report, you can retrieve this information. Or you could display the field anyway, but change its property to "Read only", and prevent anyone from changing the data.

To try this, build the form for the "Assignment" table. (While you are at this, make the labels bigger and easier to read, eliminate the "Assignment" primary key field, set the therapist and patient fields to drop-down lists that display both name and surname, provide some useful tooltips and give this form a descriptive title at the head of it). Now, set the "date assigned" to read only. With this form you can assign patients to a therapist with simple point and click and let Base automatically record the time of the assignment. This is really cool!

Date and Time boxes are customizable, allowing the display of widgets that aid the input of data . Lets see.

In the "Assignment" form, ungroup the Date box from it's label and call the Properties dialog box of the date box. Study the "General" tab to see the goodies that it offers. Find the *Spin* and *Repeat* properties (they are right next to each other) and set them to "yes". Further down you will find a *Dropdown* property. Also set it to yes. By now you will discover that the date box has changed: it offers two buttons, one on top of the other, with arrows in opposing directions. It also offers a bigger box with an arrow head pointing down. Save and run the form.

If you now click on the big arrow head you will see a small calendar for the month appear. You can swiftly change months by clicking on the arrows to the sides of the title where the name of the current month is in display. You can also select any day within the month just by clicking on it. At the bottom you will find two buttons: One for selecting the current day and one for erasing any date selected

previously. Feel free to play with this.

The nice thing about this small calendar is that the user can enter a date in the proximity of the current date checking with a simple glimpse if it falls on a weekend, in this week or the next, etcetera, minimizing errors while entering date information.

Some other circumstances in date entry could not need this widget: Imagine that you need to enter the birth day of a person who was born the 14th of December in 1936. How many times do I Have to click on the left arrowhead to reach such date? Well, about 12 times for each year separating us from 1936. For sure, just typing 1936/12/14 (or in the corresponding format) seems lightning fast by comparison. However, I could not think of a better tool for entering the "next session scheduled" information. So, again, the appropriateness of this tool depends on the context. Remember that simplicity and ease of use are our primary goals.

The smaller buttons with opposing arrow heads increment or decrement the unit of time selected. If you just press one of the buttons now, the cursor will appear in the first domain to the left, which holds the month information in the USA, and will increment or decrement it accordingly. If you move your cursor to the day or year section, the arrows will increment or decrement them, respectively.

You can do something similar for Time boxes, which you can try in the form for the "Schedule" table. If you set the *Spin* and *Repeat* properties to yes, the box will show the two opposing arrowheads, with which you can increment the time displayed by the box. Time boxes do not have a *Dropdown* property.

As said before, the TIMESTAMP box is really a grouped Date and Time boxes. If you ungroup them and call their respective Properties dialog box, you can activate these widgets.

To finish this exercise, set the small calendar as a property of the "Date case closed" date box in the Assignment" form.

Forms with two or more sub-forms.

We are now going to tackle a special but not uncommon case of forms: Where you need to link three or more tables. This usually happens when we confront intermediate tables that help with many to many relationships. Think about this: the intermediate table usually only holds combinations of primary keys from two different tables, sometimes some extra information. But these combinations make sense only when we can relate them to the tables they derive from. We need to handle information from -or for- three tables in one form.

Our example records the medication some patients are taking. The *Patient* table provides the data of a particular patient. The *Medication* table provides information about the medication. The intermediate *PatientMedication* table records one or more medications for each patient that is taking medication. The thing is: how can we we have all this information in the same form? By now we know about sub-forms and the help the wizard provides in setting them up. But the wizard only adds one sub-form per form, which allows us to enter data for a second table only. We need to add a third sub-form and we need to do it manually. This is what we will study next.

As we know, the forms that we have built with the wizard are really just writer (.odt) documents over which the wizard has added a form. The wizard has been kind enough to connect the form to a particular table in our database. In this sense, the .odt document is just like a canvas and we can add

more forms to it. We usually record information from one table to another in the form through the list box control. The wizard is also kind enough to make the proper connections by asking us which cells we want to connect. It will now be left to us to place the forms, connect them to the corresponding tables and later interconnect them using one or more list boxes.

Let's establish some generalities that can help us with this: Each form is associated with one table. In the case of forms with sub-forms, we have two forms in one .odt document: The original or principal form and the new or sub-form. The table for the principal form establishes a 1..n relationship with the table for the sub-form. While the principal form displays a particular record, the sub-forms displays all the records in the associated table that belong to the record in the principal form. So the principal form dictates what records the sub-form must display. This is why the sub-form is called “sub-form” or “Slave form”. The principal form is called the “Parent form” or the “Master form”.

The process of creating a .odt document with several forms is basically like this: We insert two or more forms in the writer document. Each form will be connected to a particular table. Then we will need to indicate which form will be the master form (or forms) and which will be the slave form (or forms). Each form will receive controls that display and allow the recording of information. Each control is usually connected to one particular field (column) except for list boxes that will connect two.

To follow the implementation we will do now, be sure to review the SQL code that created the *Patient*, *Medication* and *PatientMedication* tables. Pay attention to the variable data types of the columns. Then, make sure you study the way they interconnect by reviewing the UML diagram. Particularly, notice that the Patient table has a 1..n cardinality with the intermediate table and that the Medication table also has a 1..n relationship with it. The intermediate table will want to record the primary key of the patient and the primary key of the medication. This table also records data about the dosage assigned and the date such medication/dosage was started and ended. Because each patient could have more than one medication, it would be better to use a sub-form in the tabular format so we can read several entries at once, very much like the sub-form for telephones for the patient form. Finally, notice that the *Medication* table stores a description of what the medication does, info that would be useful to display in this form.

While following the coming instructions, take some time to study the properties dialog boxes for forms and controls, so you can readily identify the properties that we need to set. To learn more about them, don't forget to study the help files that describe them.

Let's start by using the form design wizard to create a new form. Select the Patient table but then only select the First Name and Surname columns. That's all the information we will need to display from this table. In the second step do not chose “Add a Subform” because we are going to create it manually. Finish the process by allowing the editing of all data, pick colors of your liking and open in edit mode.

You should see something like in figure xx. By now we have about one third of our form. For the next two thirds our new best friend will be the **Form Navigator**. Don't confuse the Form Navigator with the Navigator. The Navigator is called by clicking on the button that resembles a compass located in the toolbar. The Form Navigator, by contrast, is located in the Form Controls toolbar, which is usually found in the bottom of the form you are editing. The button for the Form Navigator looks like a compass over a form and is usually the fifth button from the left. If you can't see the Form Control toolbar, click on View> Toolbars> Form Controls.

The Form Navigator displays all the forms associated with the current .odt document. If we examine

the contents of the Form Navigator we will see a folder with the name “Forms” and then a folder with the name “Main Form”. This “Main Form” is associated with a label called “lblFirstName”, a text box called “txtFirstName”, a second label called “lblSurname” and a second box called “txtSurname”. You can see that this describes exactly our current form.

If you click once on “Main Form” you will see green handles surround the labels and text boxes in our form. The rest is just the writer (.odt) document.

Obviously, the content of this Form Navigator was written by the wizard after accepting our selections. Now, we are going to write directly to it.

Let's start by creating a sub-form. In the Form Navigator, right-click on “Main Form” and from the context menu that appears chose New>Form. Now a new form has appeared, with the name “Standard” that is as a dependency of (that is, with a line connected to) “Main Form”. You can actually drag this form and place it as an independent form, appearing connected to “Forms” instead. Or you could have created a new form (with controls and all) and only later drag it to become dependent of “Main Form”. It works either way. By creating this dependency, you establish that the form called “Standard” is a sub-form of the form called “Main Form”

Let's review the properties of this form. Right click on “Standard” and select “Properties”. Now, the Properties dialog box for this new form appears. You can see that a Form has three tabs: *General*, *Data* and *Events*. This last tab is reserved for Macro programming, so we will leave it unattended. The Data tab allows us to make all the connections that we need, so we will be spending some time here. The General tab is quite small and we won't need much of its functionality. However, you can change the name of the Form here. “Standard” is not very descriptive so change it now to “PatMed form”.

We have created a new form but this form is invisible to the user because it has no controls yet. Let's add the control “Table Control” which has the appearance of a grid. You can find a panel to your left with the Form Control elements. If you don't see it, go to View>Toolbars>Form Controls. Click on the box for: *More Controls* and then select “Table Control”. Now find a good position in your form and place it by dragging and releasing the mouse button. At this moment a Table Element Wizard appears, requesting that we connect this table to a data source. We could do this trough the *Data* tab in the Properties box (and we can change selections there later) but doing it here is fine too. First, we must select a table. In this case we need to connect to the *PatientMedication* table. Select it and then click on next. Now we are requested to chose the columns that will be displayed by this Table Control. We don't need the Patient ID. Just select Medication ID, Start Date, Dosage and End Date. If you now click on finish, you will see the table with column names for the data requested. You can also inspect the Form Navigator. If things have gone as expected, you should see the Table control associated to our new sub-form.

Now pay attention to this as things can get a bit confusing: The Table Control has a set of properties. If the Properties dialog box is open, you can see them by selecting the Table Control. Each of the fields inside the table control (the columns) also have properties that are unique to them. You can see them in the Properties dialog box when you click on the header of the columns. Finally, the form itself also has its own properties, which you can call by clicking on the name of the form in the Form Navigator. Do practice now calling for these properties and study their options until you become familiar with them. It feels terrible and is quite easy to be looking for properties that belong to the Form when you have really selected the Table Control, for example. So make sure that you can access them at will and can tell them apart.

Now is time to make our assignments. We have wanted this new form to be a sub-form of the form with the patient data. That is why we have placed it as a dependency of it. Because of this, Base assigns special properties to this form that allow us to link the corresponding primary and foreign keys so that the sub-form displays appropriate data. Let's see this: Click on the sub-form name (that is “PatMed form”) in the Form Navigator and call its properties dialog box. Select the Data tab.

You can read that the first option is “Content Type” and that it's assigned to Table. If you peek on the options you will discover that we can also chose a query or a SQL command. Table is selected here as a consequence of our earlier interaction with the wizard. For the same reason the *PatientMedication* table is selected in the second option: Content.

Further down we have the options: “Link master fields” and “Link slave fields”. The *Link master fields* option selects the data field of the table associated to the parent form that is responsible for the synchronization between parent and sub-form. This is usually the primary key and in our example it's the “ID Number” field from the *Patient* table. The slave field is the field of the table associated to the sub-form that holds the foreign key. In this example it's the “Patient ID” field from the *PatientMedication* table. You can write down these column names or you can click on “...” to the right. A dialog box will appear that allows you to find and select both fields simultaneously. Notice that the field for the sub-form is located to the right in the dialog box, under the name of the table it derives from; and that the field for the master from is offered to the right. Select them now.

How does the wizard know from which tables to offer options to link master and slave fields? Because we have them connected as shown by the Form Navigator. By the time you click OK, confirming your selections, “Main form” and “PatMed form” become form and sub-form, respectively. This means that when you select one particular patient, the sub-form “PatMed form” will show you any medications associated solely with the particular patient selected in “Main form”.

Let's now go to the third leg of this exercise: Including the “Medication” Table. What we want to achieve is that when we select a medication in the “PatMed form” we can read some information about that particular medication. This means that the “Pat Med form” will be the master in relation to a third form we will have to include although it is the slave form in relation to “Main form”.

Right click on “PatMed form” in the form navigator. In the context menu that appears select New... and then click on “Form”. A new form appears in the form navigator, with the name “Standard”. Call its properties dialog box. In the *General* tab, change the name to “Medication info”. Now go to the *Data* tab. For “Content type” select “Table” and for “Content” select the table *Medication*. Now skip to the part where we link master and slave fields and call the wizard (the “...” button). From the “Medication” option select “Medication ID”. Remember, this is going to be the slave field. From “PatientMedication” select “Medication ID”. This is going to be the master field. Don't get confused by the fact that both columns are called “Medication ID” and instead notice that what we are doing is joining foreign key with primary key. You might notice that it is the intermediate table, which provides the foreign key, the one linked to the master form and that the *Medication* table, which provides a primary key, is associated to the slave form. This is perfectly fine: It is not the primary key which defines who gets to be the master form but our need that the “Medication” table provide information selected in the “PatMed form”.

Our third form has no controls yet. Select it now and and add a label and a text box. Make them big so that they balance the big Table control if you place them side by side like I did. In the properties dialog

box name the label “Med Descrip” or something like this and make it display “Medication Description”. Also give it a background color that will make it stand out (I chose Grey 20%). In the *Data* tab of the Text box select “Description” for Data field. This way, every time you pick a medication in the PatMed form, a description of the chosen medication will be displayed in this box.

Our last task will be to have a list box display medication names (but really record primary key numbers in the “PatMed form”) so that we can assign medications to the chosen patient.

Now for a big surprise: The Table control is not really a control but a container of controls that are displayed in tabular form. If you select a column header you will see, upon calling the properties dialog box, that each column holds a control relevant to the column data type defined by the SQL code that created the table associated to the form. So, for example, if you select the “Dosage” column in the Table control, the header of the properties box calls it a text box, “End Date” is described as Date field and so on. If you examine their respective *Data* tabs, you will see that they were appropriately assigned by the wizard to the proper columns. Consequently, “Medication ID” corresponds to an Integer. We don't need this and we will change it to a list box.

Let's firs review what is it that we want to achieve: We want to record the primary key of the medication that is being assigned to the patient in display. Of course, we don't want to memorize which primary key belongs to which medication. Instead, we want the List box to display the names for each medication and assign the primary key of the one we click on. This also relives us from needing to type the name of the medication and making a mistake than can come and hunt us later. The selection we make will be recorded in the “Medication ID” field of the *PatientMedication* table, which we have associated with the “PatMed form”.

Let's get started: First, right click on the “Medication” column header in the Table control and select “Replace with...” and then chose “List Box”. With this column still selected, call the properties dialog box. In the *General* tab, change the Label to “Medication”. Now call the *Data* tab.

The *Data* tab of the List box has only four properties but all are relevant to the task at hand. These are: Data field, Type of list contents, List content and Bound field.

The Data field specifies the receiving cell of the table associated with the form. In this case, this would be “Medication ID” of *PatientMedication*. Because this control has already been assigned to such table, this property option has a drop down list that offers you all the available fields. Select it now.

Type of list contents is asking how the items to appear in the List box (the names of the medications) are to be generated. We are going to use some SQL code for this so you can select SQL. The SQL code that we are going to use is no more complex than the examples we have used so far. In any event, the next section explores the SELECT command in depth, if you want to have that information as background.

List content is going to be generated by the following type of SQL statement:

SELECT <field 0>, <field 1> FROM <selection table>

This instruction displays all the records in “field 0” and “field 1” of the table “selection table”, which is the table from which we are going to make our selections. In our example, it must look like this:

SELECT "Name", "Medication ID" FROM "Medication"

After we have entered it, Base is going to enclose the entire instruction in its own set of double quotes. Don't be alarmed by this.

Note the order in which the SELECT statement was designed: first we call for “Name” (field 0) and only secondly do we call for the primary key “Memeber ID” (field 1). This is relevant for our last property:

Bound field specifies how to bound the fields for the list box. When you select the option 1, field 0 will provide data to be displayed in the drop down list while the field 1 will provide the data to be entered in the field specified in Data field. In our example, then, field 0 (“Name” form the table *Medication*) will populate the drop down list while field 1 (“Medication ID” from the table *Medication*) will be entered in “Medication ID” of the table associated with this form (*PatientMedication).*

OK, we are ready! Let's test our form. First, populate the patient table with three or more fictitious patients. Then populate the medication table. You can use the following:

Name	Description
Librax	Treatment of peptic ulcer and irritable colon.
Loprox	Topical fungal skin conditions.
Maxaquin	Quinolone antibiotic for lower respiratory infections.

Now call our form. You should see that the “Medication” column has a box with an arrow head pointing down. If you click on it you should see the medications that you have entered for you to select from. After doing so, a description of the medication will appear under “Medication Description”.

Because we have set “Start Date” to be populated automatically, we should see the date displayed when we return to an entered record. “End Date” is supposed to be populated by us. To make this easier for the user, you can open the form in edit mode and call the properties dialog box for this column. Then assign the “Drop down” property to yes. This will bring out the calendar widget. While you are here, don't forget to give this form an appropriate title. Write a header with: “Patient Medication Assignment and History” or something like this.

5. Queries and Reports

Now we are ready to really take advantage of all the hard work we have done so far. Producing and reviewing queries can be truly exhilarating because this is the moment that all the *data* you have stored becomes *information* you can use.

Think about this: If you were the director of the psychological clinic we have been using for our example, wouldn't you find it informative to know that 74.6 % of all patients with a form of depression are female, that 12.7% of all the psychiatrists that handle the medication for your patients are in charge of actually 77.6% of all the patients that receive medication or that 26.8% of patients at your clinic have been diagnosed a form of post-traumatic stress disorder? Well, if you ask your database appropriately, this is the kind of information you can extract from it.

So, now you know that databases are much more than just a way to store names and addresses and later print labels for mailing. This kind of information can help you distinguish trends, calculate projections and help you make decisions.

For all this wonder to happen, you need to make sure that the information that you extract is valid information. This in turn rest on *Data Integrity*, the idea that you have stored the right information in the right place and that your system has not changed it for some obscure reason. You have been working on data integrity from the very beginning of this tutorial: The careful design of your tables and relationships -including column attributes and the handling of deletions, compliance to first, second and third normal form, making sure that the input of data minimizes entry errors... all this is aimed to ensure data integrity.

Now, the trick is to learn how to query your database in order to get this information out. If you have read other tutorials that describe reports and queries with Base, you know that a very friendly graphic user interface (GUI) has been developed to help you query your tables. In this tutorial, however, we are going to rely on the use of SQL code to create queries. First, because complex queries are really easier when formulated with SQL code (and sometimes, it's the only way to go). Second, because learning how to make SQL queries is not difficult and, third, because if you learn how to query with SQL, the GUI will be really easy to use.

Let's start with some generalities: A query always produces a table, that is, the result of your query will always be delivered as columns and rows. Even if it only has one column, it will still be a table. Reports are a fancier rendition of a query: You can include the logo of your company, include the day (date) or the person who designed the query and even organize the layout of your information. However, reports are based on queries. You should know that the Report Designer in Base can only use one table at the time as source for creating your report. This means that we have to create a query that integrates the information that we need into one table before we can give it a fancy look with a Report. We have already done something similar when creating compound list boxes for data entry.

Are you ready? Let's get started.

SQL queries with Base.

You might remember that Base offers two places to enter SQL instructions. This time we are going to use the Query section. Click on the icon over the text “Queries” in your Base interface, located in the column to the left under the heading “Database”. Three options will appear in the *Tasks* panel. Select the last one that reads: *Create Query in SQL view...*

The workhorse for creating queries is the SELECT command. This command is very versatile and powerful and will be the focus of attention for the rest of this section.

In its most basic form, the SELECT command needs only one parameter: the table from which you want to extract your information, like this:

```
SELECT * FROM “Psychiatrist”;
```

Let's analyze this statement:

First, the statement ends with a semicolon, which is true for all statements in SQL and, therefore, for

HSQL. We saw this when we analyzed the code for creating tables earlier in this chapter[1].

The '*' sign is a *wild card*, that is, it replaces the actual names of the columns in your table and represents them all. If you were to read this statement aloud, you should say something like: 'Select “all columns” from [the table:] Psychiatrist'.

This would produce the entire “Psychiatrist” table. Let's see if it is true. Type the statement and then click on the icon that has two papers and a green check where they intersect, which is the “Run Query” button. If you made no typos, then the “Psychiatrist” table should appear.

You see, this is all. Not that difficult. What we want to accomplish now is to be able to reduce the data displayed, by filtering it, so that we can extract information that is not immediately apparent.

Let's say that you want your assistant to contact all the psychiatrists by phone. She could review record by record and write down their phone numbers or you could make her life easier and swiftly produce a printed list for her. To do this, instead of selecting all columns in the table, you just want the name and phone number. Then, we should try this:

```
SELECT “First Name”, “Surname”, “Phone Number” FROM “Psychiatrist”;
```

Notice that I am indicating the names of the three columns that I want, inside double quotes and separated by comas, and the table from which I want them extracted. Run the query. Ah! This list is easier for the assistant to use.

Remember that if I had not used different case sizes and blank spaces for the names of columns and tables we would not need to use double quotation marks.

To make this list clearer to your assistant, we would like to make explicit that these are the psychiatrists and not patients or the medical doctors. In that case we can use an alias. You can see that every first row in the resulting table is appropriately labeled with the name of the columns they represent: “First Name”, “Surname” etc. But we can change this and request that the column be referenced with another name. We do this with the key word “AS”, like so:

```
SELECT “First Name” AS “Psychiatrist”,
        “Surname”,
        “Phone Number”
FROM “Psychiatrist”;
```

Note that the command has been fragmented. We have done this just to make it easier for humans to read. The SQL window will not pay attention to this and still understand, and execute, the instruction as if it were one long line. We should take advantage of this because it makes finding mistakes easier when we are composing complex instructions.

If you run this query now you will have almost the same output than before, except that the column with the first names will have the heading “Psychiatrist”. The alias command not only changes the label in the column, it also allows us to reference this column by the new name given.

If you are starting to get the hang of this you could be asking: Hey, can we concatenate the name and

1 However, HSQL is quite forgiving if you don't include it. This might not be true with other database systems.

surname into one column like we did for the list boxes? That would really make the resulting list a very cool one! And, of course, we can. You could just copy the instruction we used then, with the concatenation pipes and the strings within simple quotes. Just remember not to run the command by Base and use instead the “Run SQL command directly...” button (the one that reads SQL with a green tick on top).

But instead of going this way I want to introduce the use of functions.

Built in Functions in HSQL

Functions are statements in HSQL code that return a value. Most of these are understood and used by Base. We already know some:

CURRENT_TIME returns the time in the clock of your computer at the moment the function is called

Other functions take one or more parameters. Think of the COS(d) function: you give it the parameter 'd' (which is supposed to be and angle) and it will return the cosine of the angle. Let's imagine that you have a database that has collected some measured angles and you need their cosine. This would be as easy as:

```
SELECT “Angle Data” AS “Angle α”,
       COS(“Angle Data”) AS “Cos α”
FROM “Measurements”;
```

Note that we have given the name of a column as a parameter. Again, because of capitalization and spaces, this name is within double quotes. This instruction produces a result table with two columns: The angles you stored in the column “Angle Data” in the “Measurements” table, and the cosine of those very same angles, and all clearly labeled for ease of use.

There is a great deal of interesting functions available for HSQL, and you can find them in their website:

http://hsqldb.org/web/hsqlDocsFrame.html

If you check their documentation, you will find a function like this:

CONCAT (string1, string2)

This one would allow us to concatenate name and surname, like this:

```
SELECT CONCAT(“Surname”, “First Name”) AS “Psychiatrist”,
```

Let's replace our first line in the previous example with this instruction and run the query. What happens? This should really make your assistant very happy!

```
SELECT CONCAT(“Surname”, “First Name”) AS “Psychiatrist”,
       “Phone Number”
FROM “Psychiatrist”;
```

You will see that there is no space or coma between name and surname in the resulting set: they have

just been joined. In this respect, the method using the pipes is better. It also happens that this function accepts only two parameters: string1 and string2, while the pipes allow you to connect as many clauses as you want. For instance, we could be adding name, middle name and surname.

However, there is a workaround to solve this problem with **CONCAT** which also helps us understand the versatility of functions and how we can use them. Let's think about the end result that we want: We want the column that displays the names of the psychiatrists to have the following format: Surname + coma + space + first name. Basically this means that we want to join the surname with a string containing a coma and a space (remember that with HSQL strings are enclosed within single quotes) and then join this result with the first name. We can represent this idea like this:

CONCAT(α, "First Name") where

α means: **CONCAT("Surname", ', ')** so our statement should look like this:

```
CONCAT(CONCAT("Surname", ', '), "First Name")
```

This could be somewhat difficult to read, with so many quote signs and with comas that are both part of the end result and also part of the definition of the function. But after you analyze it for a while it makes perfect sense. It also hints us on that HSQL will not complain about applying any combination of functions as long as we don't default the syntax and the data type. This allows for a very powerful use of functions.

Can we use the CONCAT function instead of the double pipes in building compound list boxes? Yes, which will also allow us to run the instruction through Base and not skipping directly to HSQL.

This makes for a nice introduction to the versatility of functions. We will see later that our ability to synthesize data and produce global numbers (like percentages, averages and sums of columns) are all based on functions. We will come back to this when we finish exploring the SELECT command.

Saving and Calling a Query.

At this point you should save your query (File> Save, or click the icon with the diskette). You will be asked a name for this query (I used "qryPsyPhoneList"). If you close the query now and go back to the Base interface, you will see that this query has joined our list of available queries. If you double click on it, the resulting table will appear.

Because this query is executed when it is called, any additions or deletions to the Psychiatrist table will be reflected here, that is, the query updates itself every time it is called. Now, that is very useful.

Should you want to edit this query, right-click on the name of the query and select "Edit query SQL". If, after you amend your query, you want to keep both versions, simply save this new query with "Save as..."

Where?

So far you know how to select columns for a query, give them an alias if necessary and even run them through a function.

The SELECT command also allows you to pick a subset of your total data. You do this by imposing a condition with the WHERE clause. HSQL will go through all the records checking this condition. If the evaluation returns TRUE, that record will be displayed with the resulting set. If the evaluation of the condition for that record returns FALSE then the record will be skipped.

Let's imagine that you have an interest in listing all the patients of the clinic that are male only. In this case, you would use:

```
SELECT * FROM "Patients"
WHERE "Gender" = 'Male';
```

You can see that we are using the wild card, which means that all columns from the "Patient" table will be returned. However, only the records that have 'Male' in the "Gender" column will satisfy the condition imposed by the WHERE clause and, consequently, only those records will be displayed.

Notice the use of the equality sign "=". You have all comparison operators at your disposal: <, >, <=, >= and even <> (or !=). You can compare columns to strings, dates and numbers and even to other columns.

Using these operators with numbers is quite straightforward. When you compare dates you should know that an older date is considered "*less*" that a more recent one, so:

(Date1 < Date 2) will return true only if Date1 is an older date that Date2.

String comparison can also use these mathematical operators. In this case, they represent the position of the letter in the alphabet. 'A' is *less* than 'Z', 'C' is *bigger* than 'A'. In the comparison, if the first letters are equal then the second letters are compared, and so on.

Let's say that you want to break down your list of patients so several assistants can work on them simultaneously. Lets say that your first assistant will work with all patients where the surname starts with A, B or C. Then you can produce this list by requesting:

```
SELECT * FROM "Patients"
WHERE "Surname" < 'D';
```

Please note that in all these examples, the name of the tables are enclosed by double quotes while the string literals are enclosed by single quotes. This is how SQL tells them apart.

The WHERE condition accepts expressions comparing the value in a column with the NULL value, like this: WHERE <column_name> IS [NOT] NULL. The square brackets mean that you can decide to include or not the "Not" statement, depending on what you are looking for: the record to have a null value or the record NOT to have a null value in the particular attribute. In order to find a more comprehensive list of accepted expressions, check the documentation provided by the HSQL website at:

http://hsqldb.org/web/hsqlDocsFrame.html

Now, you can ask for exact matches (with the '=' sign) but also can request *similar* matches using the LIKE operator, which you use instead of the mathematical operators.

Because you, more or less, know the *kind* of matches you are looking for, SQL offers you two other wild cards to help you be more precise with your condition: The '%' (percent) sign and the '_' (underscore) sign. The '%' stands for any character, any number of times (including *zero* times). The '_' sign stands for any character, exactly once.

Let's see this in action:

```
SELECT * FROM "Patients"
WHERE "Surname" LIKE 'B%';
```

This will select all names that start with the letter "B", mo matter what characters -or how many characters- appear after the 'B'. If you had a record that only had a 'B' for surname, it would also be included in the result.

```
SELECT * FROM "Patients"
WHERE "Address" LIKE '%Main St.%';
```

This query will select all records that have 'Main St.' somewhere in the address. Notice that I used the '%' sign not only at the end of the string literal to match but also at the beginning. If I had omitted the first '%', it would have meant that I am looking for records whose address start with 'Main St.'. If I had omitted the last one, it would have meant that I am looking for records that end with 'Main St.'.

Believe it or not, these two wild cards are enough for many complex combinations you could want to match.

Compound queries

You can include more conditions to the WHERE clause with AND, OR and NOT to make more complex queries.

Lets say that you are planning a surprise party for Belinda, one of the psychiatrists, and wish to invite all female psychiatrists that practice in your same ZIP code (which, for this example, will be 13044). Of course, Belinda should not get an invitation or the surprise would be spoiled. The list your assistant needs could be composed with the following:

```
SELECT * FROM "Psychiatrist"
WHERE "Gender" = 'Female'
AND "Zip" = '13044'
AND NOT "First Name" = 'Belinda';
```

Note that this query will exclude all psychiatrists that have 'Belinda' as a first name, not only *our* Belinda. To avoid this you might want to include the surname as part of your query. But you see where I am going: you can make SELECT commands as complex as your search might require.

Order in the room, please!

Now, the output table of your query might not show any particular order. The database went about collecting the records that conform to your conditions and then displayed them in the order they were found. Many times you need to sort the result based on some parameter. In order to achieve this, the

SELECT command accepts an ORDER BY clause.

Let's say that you want the list for the party ordered alphabetically according to the surname. Then you should ad the following clause:

```
SELECT * FROM "Psychiatrist"
WHERE "Gender" = 'Female'
AND "Zip" = '13044'
AND NOT "First Name" = 'Belinda'
ORDER BY "Surname" ASC;
```

ASC means that you want the list in ascending order (remember that 'A' is *less* than 'Z'). Optionally, ORDER BY also accepts DESC for descending, that is, from *more* to less (from 'Z' to 'A')..

For example, if you wanted a list with all your therapists, ordered according to years of service, with the more recent additions to the top of the list, you should request:

```
SELECT * FROM "Therapist"
ORDER BY "Hiring date" DESC;
```

User defined parameters.

Let's say that as the director of the clinic you need to find a particular patient whose name was Sanders or Sandoz or something like that, that was admitted sometime between 2004 and 2006. Instead of going through 2500 records one by one, you can create a query:

```
SELECT * FROM "Patient"
WHERE "Surname" LIKE 'San%'
AND "Time of Registry" <#2007/01/01#
AND "Time of Registry" >#12/31/2003#
ORDER BY "Surname" ASC;
```

Note that the dates are enclosed by "#" signs, so Base can tell that they are dates. Also note that I have used two formats for entering date: the ISO format that uses yyyy/mm/dd and the ***Local*** format used in the US that uses mm/dd/yyyy. Of course, I should stick to using one or the other for consistency but I want to show that I can use either. If you live in Europe or Latin America, your Local format would be dd/mm/yyyy. The local format is set when you chose the Language settings for OpenOffice.org.

This could be a very useful query for finding patients, except that you will have to write it again, changing name and dates, every time you are looking for patients with other parameters.

Actually, Base allows you to leave variables in place of your parameters, and ask you to fill them at the time you run the query, without having to change the code.

This is how it works: instead of defining the parameters (in this case, the name and the date limits) you write the operator followed by a colon (the ":" sign) and then a variable name. It will look like this:

```
SELECT * FROM "Patient"
WHERE "Surname" LIKE: patientName
AND "Time of Registry" <: topDate
AND "Time of Registry" >: bottomDate
```

```
ORDER BY "Surname" ASC;
```

This is what it's going to happen when you run the query:

Base will display a small window called "Parameter Input". At the top of the window you will see a label that reads "Parameter" and immediately below it a box with the three variables present in this query: patientName, topDate and bottomDate. Under the box you will see another label the reads: "Values" and then a single text entry box. You then enter the parameters for your search in the order they are asked for, pressing the ENTER key after each one. The window also displays the buttons: *OK*, *Cancel* and *Next*. You can also enter the values by clicking on *Next* after each value and *OK* after you enter the last one.

After that, the matches will appear in table format ordered by surname from A to Z.

Because we built our query with the LIKE operator for patient name, you are entitled to use the wild cards "%" and "_" just like we have been doing so far. The dates you enter need to be enclosed by "#" signs, so Base can tell that they are dates. For example, you could enter this:

Smi%
#01/01/2005#
#12/31/2003#

Now, the query will look for all Surnames of patients that begin with Smi (like Smith or Smithers) that were admitted in 2004. You can run the query again and again, making new searches, without having to amend your SQL code.

Let's put all this together:

SELECT <column names> AS <alias> [more column names, separated by comas, with their own alias]
FROM <table> [more table names, separated by comas]
WHERE <condition1> [including LIKE and connection clause: AND, NOT, OR <condition 2> etc.]
ORDERED BY <column name> ASC or DESC [end query with semicolon]

Querying more than one table at a time

The things that you can do by now with the SELECT statement are already pretty impressive. We take this to the next level when we think about the ability to use the data in more than one table to create a query.

Imagine the following circumstance: You need a list of of all your patients and their psychiatrists. The first and last name of patients is in one table and the first and last names of the psychiatrists is in another table. How do we do this?

First, we need to reference both tables being used in the FROM clause of the SELECT command. Also, we need a convention to make explicit what tables are we extracting our columns from. Notice that both, the "Patient" table and the "Psychiatrist" table have a columns called "First Name" and "Surname". In order to tell them apart we are going to use the following syntax in our query:

<table name>.<column name>

This is to say that we are going to use the name of the table and the name of the column, separated by a period, to unequivocally reference the columns that we need. Following th example, we should write:

```
SELECT "Patient"."First Name",
          "Patient"."Surname",
          "Psychiatrist"."First Name",
          "Psychiatrist"."Surname"
FROM "Patient", Psychiatrist";
```

Please note that, because our names for columns and tables use upper and lower case and spaces, we need to enclose them in their own set of double quotes.

Let's assume that you have exactly the following three patients in your database: Alex Smith, John Serrato and Frank Thrung; and exactly the following two psychiatrists: Amanda Perez, Debbie Dobson. Also, we know beforehand that Alex and John have Amanda as their psychiatrist and Frank has Debbie.

However, if we run the above query we get this:

First Name	Surname	First Name	Surname
Alex	Smith	Amanda	Perez
Alex	Smith	Debbie	Dobson
John	Serrato	Amanda	Perez
John	Serrato	Debbie	Dobson
Frank	Thrung	Amanda	Perez
Frank	Thrung	Debbie	Dobson

This is clearly not what we expected. First of all, and if we look carefully, what we really have is a result where every patient is combined with every psychiatrist. We have 3 patients and 2 psychiatrists and a resulting set of 2 x 3 = 6 combinations. What we really want is the information of which patient has which psychiatrist. Second, both columns read “First Name” and “Surname” and, with no other clue, we can't tell which is a patient and which is a psychiatrist.

Well, this last thing is easy to correct. All we need to do is add an alias for each column. In general, when we are working with more than one table, it is a very good practice to use an alias for each column to avoid these uncertainties. It is also more elegant.

Now, the first problem is also easy to solve. If you remember, each psychiatrist has a primary key that appears as a foreign key in the patient's table. If you don't remember, review the SQL code that created the tables. It is this foreign key that tells us which psychiatrist sees which particular patient.

So what we want to do is produce a result set where the foreign key in the “Patient” table is identical to the primary key in the “Psychiatrist” table. Of course, we will use a WHERE clause to include this condition.

Consequently, our SQL code will look like this:

```
SELECT "Patient"."First Name" AS "Patient Name",
```

```
        "Patient"."Surname" AS "Patient Surname",
        "Psychiatrist"."First Name" AS " Psychiatrist Name",
        "Psychiatrist"."Surname" AS " Psychiatrist Surname"
FROM "Patient", Psychiatrist"
WHERE "Patient"."Psychiatrist ID" = "Psychiatrist"."ID Number";
```

Every time we are joining tables, we will need to match foreign and primary keys. Otherwise, we will get results like the one on the top. Such result, that shows all possible combinations between patient and psychiatrist, is known as a *Cartesian Join*. The WHERE clause that help us match key numbers transforms this into an *inner join*, which gives us the information we want.

Let's imagine that we also wanted to include the patient's phone number as a result of the query. If you remember, this data had been placed in its own table because we wanted to record an unspecified amount of phone numbers for the patients.

What we will need to do is to include the Patient Phone table to the query (with the FROM clause), use the proper <*table*>.<*column*> syntax and include an AND condition to the WHERE clause that links the primary key of the patient with the patient ID foreign key in the Phone table. Try writing such query now. If you notice, joining two tables requires one WHERE statement. Joining three tables requires two WHERE statements, connected by an AND. You can start seeing a pattern here: Joining four tables will require three statements. In general, if you are joining n tables, you will require n-1 joins.

As an exercise, do write this query that asks for a psychiatrist and lists all his/her patients with their respective phone numbers.

Intermediate tables are no more difficult that what we have done so far. If you remember, we use intermediate tables between two tables that have a Many to Many relationship (m..n). The use of an intermediate table transforms that cardinality to a manageable 1..n. However, an intermediate table basically just stores several combinations of foreign keys. What we want is to extract the *meaning* from those combinations.

In our example we have an intermediate table between patient and medication. One patient can use one or several medications. One medication can be used by one or more patients.

Let's say that we want to list all the medications used by a patient. This is stored in the intermediate table, which holds the patient's key number associated with the key numbers of the medications he/she uses. But this table only lists such combination of key numbers. In order to know what they mean, we also need to consult the "Patient" table and the "Medication" table. To avoid a Cartesian join, we need to include the appropriate conditions. Because we are joining three tables, we know that we will need two joins. To make the result of this query easy to read, we will also include informative aliases. Can you write this query before looking further? You should!

```
SELECT "Patient"."First Name" AS "Patient Name",
       "Patient"."Surname" AS "Patient Surname",
       "Medication"."Name" AS "Medication",
       "Medication"."Description"
FROM "Patient", "Medication", "Patient Medication"
WHERE "Patient Medication"."Patient ID" = "Patient"."ID Number"
AND "Patient Medication"."Medication ID" = "Medication"."ID Number"
ORDER BY "Patient"."Surname" ASC;
```

Let's dress up this query by including other elements reviewed in this part: Let's make the query ask us for the patient for which we want the report and let's have the output of the name be displayed in one column:

```
SELECT CONCAT(CONCAT ("Patient"."Surname", ', '), "Patient"."First Name") AS
       "Patient Name",
       "Medication"."Name" AS "Medication",
       "Medication"."Description"
FROM "Patient", "Medication", "Patient Medication"
WHERE "Patient"."First Name" LIKE: PatientName
AND "Patient"."Surname" LIKE: PatientSurname
AND "Patient Medication"."Patient ID" = "Patient"."ID Number"
AND "Patient Medication"."Medication ID" = "Medication"."ID Number"
ORDER BY "Patient Name" ASC;
```

Note that the order clause uses the name given to the column by the alias. Remember that aliases not only change the heading of the column but also allow us to reference the column with that name in the SQL code.

This is as complex as our SELECT statements will ever get, but by now you should be able to read them, and produce them, with no problem.

Aggregate Functions:

Aggregate functions also return a value, like regular functions, but instead of returning one value for each record evaluated, they return one value that represents the entire collection of evaluated records.

The most clear example of this is the COUNT function. COUNT will return the number of records in a specified table. Let's say that you need to know how many patients have been recorded in the database. Then you can run the following instruction:

```
SELECT COUNT(*) FROM "Patients";
```

which, in my computer, returns a result like this:

COUNT(*)
8

(Because I only entered data for eight patients)

Now, believe it or not, this result is still a table, albeit with only one column and only one row (plus the header row with the name of the column).

Compare this result with the COS(d) function that we reviewed earlier. In that example the parameter 'd' stood for a column name and the function returned the cosine for each and every angle recorded in that column. Aggregate functions, on the other hand, return a summary number that represents an aspect of the entire set.

We will briefly review the following aggregate functions:

COUNT, MIN, MAX, SUM, AVG, VAR_POP, VAR_SAMP, STDDEV_POP, STDDEV_SAMP.

In general, these functions use the following syntax:

SELECT <*function*> **("Column Name") FROM "Table Name" [...]**

where <*function*> stands for one of the 9 aggregate functions named above.

Let's check them out:

● As sated, COUNT returns the number of rows in a table. Of course, we can filter this number to obtain a count of subsets. For example, say that you need to know the number of male patients in the patient table. I am sure that you can already anticipate the needed code:

```
SELECT COUNT(*) FROM "Patients"
      WHERE "Gender"='Male';
```

You can also change the header row for a more descriptive name of your result with an alias.

```
SELECT COUNT(*) AS "Number of Male Patients" FROM "Patients"
      WHERE "Gender"='Male';
```

Let's say that you need to know the number of *unique* phone numbers in the Phone Number table. We can imagine that two or more family members that live together share at least the home number, so a simple count instruction will not do: there will be *repeats* that are counted. We can use the DISTINCT qualifier to eliminate repeats from a query, like this:

```
SELECT DISTINCT COUNT("Number") FROM "Phone Number";
```

In short, if the DISTINCT qualifier is included, only one instance of several equivalent values is used in the aggregate function. If we wanted the contrary, that is, to make sure that we include all instances in our query, we can use the qualifier ALL.

You should know that the data type of COUNT is Integer, in case that you want to do some kind of arithmetic operations with it or wish to use this result with a drop-down list. We will come back to this.

Also note that COUNT will include rows even if they have null values. For example, if you request the query:

```
      SELECT COUNT("Surname") FROM "Patient";
```

even if you have some records that do not yet have a surname, they will be counted anyway.

Being able to obtain the total number of records in a table and the count for subsets of it allow us to determine all kinds of percentages that can describe the entries in our database.

You can do this calculation by hand or have Base produce it for you. The truth is that getting such a number requires some work but, once you have it all in place, you can get updated results with a simple

double click of a mouse. Let's say that we want to calculate the percentage of patients in the database that have been diagnosed with post traumatic stress disorder (PTSD for short).

From a conceptual point of view, we first want to calculate the total number of patients in the database. We can use a simple SELECT COUNT clause for this. Next we need to calculate the number of patients that have been assigned a diagnosis of PTSD, for which we include a WHERE clause to match the condition. Now we divide the number of PTSD patients by the total number of patients, which gives us a decimal number between zero and one, and then multiply this number by one hundred to get the percentage, something like this:

$$\% \text{ PTSD Patients} = \left(\frac{n_{ptsd}}{n_{total}}\right) x\ 100$$

Notice that in calculating a percentage, the subset always goes in the numerator (above the division line) and the total always goes in the denominator (below the division line).

One difficulty of this procedure is that we need two SELECT clauses to produce this result, one with a WHERE clause and one without it. We can't have both SELECT instructions in the same query. How do we solve this?

Well, we produce two queries for each SELECT and then we produce a third query that combines the results of the previous queries. Here is when you could say: *Nah! leave it. I just calculate the two COUNTS and then divide their results with my calculator.* Fair enough. But with the three queries in place, in the future, after the database has seen new additions, you just call for the third query and you will have an updated percentage of patients diagnosed with PTSD.

The complete procedure is as follows: You create the first query for the total number of patients with code like:

```
SELECT COUNT (*) AS "Total Patients" FROM "Patients";
```

Then save the query with a name like "qryTotal Patients".

Next you create the query for the subset, patients with PTSD, with code like:

```
SELECT COUNT (*) AS "Total PTSD Patients" FROM "Patients" WHERE "Diagnosis"='PTSD';
```

and save it with a name like: "qryTotal PTSD".

Now, at the Base interface in the Queries view, we right click on these queries and select "Save as view". If you remember, this will create virtual tables with the result of the queries. Virtual, because the tables will be updated reflecting any relevant changes to the records in the database every time they are called. For the names of the views, leave the same name but delete the 'qry' component of the name. You find these views in the Tables section of the Base interface.

For the third query we will call the results from the two previous queries, with code like the following:

```
SELECT ("Total PTSD"."Total PTSD Patients"/"Total Patients"."Total Patients")*100
AS "Percentage of PTSD Patients"
FROM "Total PTSD", "Total Patients";
```

Notice that this SQL code not only selects the columns that we want but also includes arithmetic operations (the '/' sign for division, the parentheses and the "*" sign for multiplication). This is completely valid code and, in fact, most database engines like HSQL allow SELECT to modify the output with mathematical operations like these.

But alas, if we run this query we get a result of zero! What happened?!

Well, we know that the division produces a number between zero and one (because the subset is a fraction of the total) but, if you recall, the COUNT function produces an Integer data type, which means that the result is always rounded down -in this case to zero. Zero by one hundred is zero. What can we do?

One very clever solution is to change the formula, multiplying by one hundred before making the division, something like this:

```
...("Total PTSD"."Total PTSD Patients"*100)/"Total Patients"."Total Patients"
```

From a mathematical point of view both formulas are equivalent and from a practical point of view we avoid being completely rounded down to zero. But this solution is not entirely satisfactory because we still lose all decimal numbers, which can add to produce important error in our calculations[2]. The problem is the Integer data type of the COUNT result.

The functions will save us again. If you check the HSQL documentation that we have cited you will discover two functions that allow us to change the data type of our result. These functions are:

`CAST (term AS type)`	converts *term* to *type* data type
`CONVERT (term, type)`	converts *term* to *type* data type

Their descriptions in the documentation suggest that they are equivalent in their result and only change in the syntax they use. Maybe this is not complete and there are differences in the way they behave. We will only know this by trying them in different situations and comparing them. For now, I will just use the CAST function to achieve my goal, and will assign my results a Real data type, like this:

```
CAST ("Column 1" AS REAL) / CAST ("Column 2" AS REAL) * 100
```

Where: Column 1 = **`"Total PTSD"."Total PTSD Patients"`** and
Column 2 = **`"Total Patients"."Total Patients"`**

so the total code of our last query would look like this:

```
SELECT( CAST ("Total PTSD"."Total PTSD Patients" AS REAL ) / CAST ( "Total
Patients"."Total Patients" AS REAL )* 100) AS "Percentage of PTSD Patients"
FROM "Total PTSD", "Total Patients";
```

2 You could multiply by ten thousand to keep two decimal numbers, but because we do not have a decimal point, this makes reading the result ambiguous.

This way we will get enough decimal numbers and the proper format in the calculation of the percentages.

Remember that we can change the data types in the last query, like in the example above, but could also change them in the queries that produce the partial results that we need.

> *For the fun of it, create a query that compares the percentages of PTSD patients that were active three, two and one year ago with the patients active today.*

● MIN and MAX will browse the specified column and will identify the smallest and the biggest number, respectively, of your set. Obviously, these functions apply to numeric data types only. The resulting number will be of the same data type as the column over which the function operated.

● SUM will do exactly that: it will return the sum of all the values in the specified column. This comes in very handy, for example, when we need to know the total of sales, earnings or losses that we have tabulated. Again, SUM applies to numeric data types. HSQL will assign a variable type to this result in a way to ensure lossless results, that is, with a level of precision commensurate to the result of the sum.

We can describe SUM mathematically. Let's assume that you have n records (of numerical data) in a column, then, the SUM of that column returns:

$$\text{SUM (X)} = \sum_{i=1}^{n} x_i$$

Note that SUM, MIN and MAX will exclude null values from the calculation.

● Statistical functions include: AVG, STDDEV_POP, STDDEV_SAMP, VAR_POP and VAR_SAMP.

AVG calculates the average of the numbers in the specified column by calculating the sum of all the records and then dividing that by the total number of records. To be precise, this function performs the following calculation:

$$\text{AVG} = \overline{X} = \frac{1}{n}\sum_{i=1}^{n} X_i$$

The average is a tool of statistics that allows us to describe a collection of quantitative measurements. For example, we cite the average income of the people living in one neighborhood, or the average time of reaction to a specific environmental signal. To fully appreciate how representative that average is of the set it describes, we will also want to know how close of far apart each particular measurement is from that average. The standard deviation is *an average* of the deviation of all the particular measurements from the set's average[3]. A smaller number means that the individual measurements are close to the average, a bigger number means the opposite.

Not surprisingly, STDDEV_POP calculates the standard deviation of the numbers in the specified

3 Computed using the sum of squares of the distance between each measurement and the set's average.

column. Conceptually, STDDEV_POP represents the following formula:

$$\text{STDDEV\_POP} = \sigma = \sqrt{\frac{\sum (X_i - \overline{X})^2}{n}}$$

However, it is more likely that the standard deviation is calculated by HSQL using the following formula, which mathematical science has shown to be equivalent to the previous one but does not require the computation of the average:

$$\sigma = \sqrt{\frac{\sum x^2 - \frac{(\sum x)^2}{n}}{n}}$$

The average is of the same data type as the column from which it's calculated. Meanwhile, the standard deviation is always assigned a Double data type, to ensure its precision.

STDDEV_POP assumes that the values in the column belong to all the members of the population that we are trying to describe. If this is not the case, we say that the measurements belong to a *sample* (a subset) from the population. In this case, statistical theory requires that we adjust the formula to account for this:

$$\text{STDDEV\_SAMP} = S = \sqrt{\frac{\sum (X_i - \overline{X})^2}{(n-1)}}$$

Conceptually, that is the value that STDDEV_SAMP will produce, although it is more likely that it will compute it using the equivalent formula:

$$S = \sqrt{\frac{\sum x^2 - \frac{(\sum x)^2}{n}}{(n-1)}}$$

The Variance is another measure of variability used in statistics. From a mathematical point of view it corresponds to the square of the standard deviation. VAR_POP assumes that it describes the entire population, and is defined by:

$$\text{VAR\_POP} = \sigma^2$$

while VAR_SAMP assumes that it is describing a sample from it and, consequently, corresponds to:

$$\text{VAR\_SAMP} = S^2$$

However, it is more likely that these values are calculated by formulas like the ones used by

STDDEV_POP and STDDEV_SAMP, correspondingly, but without extracting the square root.

Variance is also assigned a Double data type by HSQL.

Note that none of these statistical functions allow for ALL or DISTINCT qualifiers and that they will exclude null values in their calculations.

Let's now tackle a problem that uses statistical functions: let's produce a report that calculates the average age (and the standard deviation, to make sense of this number) of all women with a diagnosis of depression in our database. Can you imagine the excitement of a researcher that is hinted on the idea that most women attending for depression significantly belong to the same phase in life?

I am sure that you have spotted the need to use the WHERE clause to extract the group we are interested in: Women with a diagnosis of depression. More subtle is the fact that we have not recorded the age of any woman but instead we have their dates of birth. What do we do now?

[to be completed...]

6. Creating reports with BASE

If you notice, there is currently no way to print on paper the tables you have created in the "Queries" view. For this to happen you need to create a report.

The report will provide the same information found in the table created by your query. Actually, you can even chose to omit certain items before printing it. The nice thing about a report is that you can customize the layout, add a company logo, add info about the date or the person who created the query and things like that, so that when you print it, these elements frame the validity of the information provided, it will be easy to read and will also look quite professional.

For our master exercise I want to create the Clinical Summary Report. This is the report where we provide contact and clinical information about a particular patient, his/her medical doctor, psychiatrist if there is one, medication and other data.

If you now click on "Report" in the left column of the Base interface, the task bar will offer you the option "Use Wizard to Create Report". This option is quite straightforward: the wizard will make a series of questions about what table you want to use, what fields you want to write, how do you want to group them and what template for layout you want to use. Actually, by now you should be able to understand most of the options offered and I recommend that you play with the wizard and become familiar with the way it works. For most things, the wizard will be just the right tool.

The only thing is that the Report Wizard will always display the result of your query as a table. This way, the result of the query for the Clinical Summary will be displayed as one long row. Because we are only interested in one particular patient, there will be no other rows to fill the page. You will end with one (quite long) row, spanning several pages, but leaving most of the printed page empty. This is not a satisfactory solution and departs greatly from the layout we did in Part II.

Don't despair. Sun Microsystems has made available a "Report Builder" that allows you the flexibility of design view in re-organizing the layout of your reports. There is no reason why you should not download it. At the time if this writing I am using OO.o 3.0 and the Report Builder 1.0.6 has integrated

nicely to the application and has worked fine so far.

After you download and install the extension you will find that the Tasks menu in the Report section offers you a second option: “Create Report in Design View”.

Just to get acquainted, click on this option to see what we get.

Wow, I bet that this was more than what you bargained for! Don't worry, all these options are here to help us.

Below all the menus and toolbars there is a page divided in three. To the left you can see that the top section is the Header, the middle section is the body of the report, called “Detail” and to the bottom we have the footer. To the right we have a column with two tabs: *General* and *Data*. They work just like the Properties Dialog Box that appears while working with the Forms in Design View, providing the characteristics of any objects we select and allowing us to customize them.

Here is an important trick: we connect the report we are building with the table that results from our query in the Data tab (which is typically the first step). However, if you select any other object in the form, this tab disappears. Why? Dunno, and I still have to learn of a way to call it back. If you miss it, you will have to close the builder and open it again. Anyway, don't worry about it yet as we are only browsing around.

Let's focus on the three sections in the page. You can click on any to activate it. The tab to the right shows options to customize each section. You should read them now to become familiar with them. If you don't see a tab to the right, go to View>Properties, or click on the *Properties box* button (usually second from the left in the lowest row of the toolbars).

The header section will receive data that you want displayed on the top of every page in the report (good for titles, dates, company logos and things like that). The footer is similar, displaying the info at the bottom of every page (good for page numbers, contact info and the like). The central part of the page, labeled “Detail” will receive the information from the tables and will repeat itself for every record collected by the query. This info will appear in Text Boxes, which we will tag, if needed, with Labels.

You will notice the property: “Height” followed by some unit in the properties panel. You can change the height of any section by either changing the units here or by dragging the horizontal line that separates two sections.

Let's assume that you want to insert a label. First you select the Label button (just to the right of the Properties button). Now you can click and drag the label box anywhere in your form. Immediately after, you will see this object's properties appear in the column to the right. Find the “Label” option and write there “Name:” to see the text in the label change in your form.

Let's now place a Text Box next to our label. You will find the Text Box button next to the Label button. Click, place and drag. You will see that in the column to the right there are two tabs this time: *General* and *Data*. Study them. If you examine the *Field* property in the *Data* tab you will see that it is empty. This is where the connection to a particular field in our table is defined and can be set automatically by the report builder or by you, as we will see later.

It goes without saying that you should also read the manual for the Report Builder in the help files. You can also find a very good reference here:

http://wiki.services.openoffice.org/wiki/SUN_Report_Builder/Documentation

Our first Report with Base:

Before attempting our more ambitious project of developing a Clinical Summary report, let's do something simpler, just to get the hang of it. For this exercise we are going to use both, the Wizard Report and the Sun Report Builder. Later on, when we tackle the clinical summary, we will only use the Sun Report Builder from beginning to end. However, there is nothing wrong with using this dual approach: the report wizard will have us up and running quite quickly and then we can use the Report Builder to fine tune our work.

Our report should produce a list of all patients in our database with their corresponding phone numbers. This exercise should be interesting for a number of reasons: First, it uses data from two different tables that need to be properly linked; second, we have an unknown number of phone numbers for each patient: some patients will have one but others can have two, three or more phone numbers. This gives our report a degree of unpredictability. Lastly, we are going to use the CONCAT function to create a column that we will name with an alias and then we are going to reference that column with the alias in another part of the SQL for our query. It is going to be very interesting and informative to see how do the report wizard and the report builder respond under these circumstances. Before advancing further, make sure that you have some entries in the Patient Table and that you give them differing amounts of phone numbers.

We should start by coming up with a desirable layout for the report. I would like it to have a heading that identifies the report clearly, something like “Patient's Phone Numbers”. We should also include the date in which we generated the report so that we know up to what moment in time this report is valid. I also want to number the pages so that I know if one has gone missing. Then we should have the name of the patients, surname first and separated from the first name by a coma and a space, and ordered alphabetically. I want to have the phone numbers underneath the names and in line with a reference to which number this is: home, office or mobile phone, etc.. Now that I am scribbling the layout of this report on a napkin, I realize that I would also want to include some kind of graphic, like an horizontal line, that would help me separate each record, making it easier to scan.

We will obviously need the “Phone Number” table, which holds the number and descriptor. But the names are not located here. For this I need the “Patient” table. We will need to use two CONCAT functions to produce the output for the name that we requested in the previous paragraph and we will have to name this new column with an alias so we can reference it in our ORDER BY instruction. We are going to need other columns so that we can properly join these two tables -mainly primary and foreign keys, although those columns will not be part of the output. Because we are joining two tables we know that we need only one join.

Al right then, can you produce the SQL code for the query that we need before reading further? You should!

```
SELECT CONCAT( CONCAT( “Patient”.“Surname”, ', ' ), “Patient”.“First Name” ) AS
“Patient Name”,
      “Phone Number”.“Number”,
```

```
      "Phone Number"."Description"
FROM "Patient", "Phone Number"
WHERE "Patient"."ID Number" = "Phone Number"."Patient ID"
ORDER BY "Patient Name" ASC;
```

Let's produce our query now. Click on the *Queries* button in the Base interface and then click on *Create Query in SQL view...*. Now type the SQL code and, just to make sure it works, run it (by clicking on the box with the green tick on top of the two overlapping papers). If every thing goes fine, you should see three columns, one with the header "Patient Name" and the other with the names: "Number" and "Description". The names should be in the "Surname, FirstName" format and ordered alphabetically. The names should be repeated for every different phone number the patient has. I have noticed that, sometimes I get an Error notice when I find no clear syntax error. I discover that, by replacing the double quotes, I fix this problem. You can see that the code in this page uses opening and closing double quotes (which are slanted), which are obviously two different characters. Yet, the code in the SQL window only shows straight double quotes. These differences can make you believe that you are inserting a character that the SQL window is actually not expecting.

At this moment we need to save the query. You do this by clicking on File>Save or by clicking on the diskette symbol. I used the name *qryPatientPhoneNumber* but hey, that is just me and you can use any name you feel comfortable with.

We are ready to build the report now. Click on the Report button on the Base interface and then select *Use Wizard to Create Report...*

At this moment you will see the Report Builder pop up and, over it, the Report Wizard. This is fine. The Report Builder provides the ground for our work. You will see that all our selections -while using the wizard- will be placed, or otherwise affect, the page on the report builder.

The wizard is composed of two columns. The one to the left names six steps that the wizard will walk us through and highlights the current step. The column to the right is wider and holds the options and stores our selections that correspond to each step. The first thing the wizard wants to know is which table will provide the information and which columns will constitute our report. The drop-down list under "Tables or Queries" displays all the available tables and queries in this database. Find the query we just built (that I named *qryPatientPhoneNumber)* and select it. Quite immediately you will see that the boxes below are populated with the columns that belong to this table. You can select now which columns from this table you want in your report. We want them all so click on the ">>" button. (Notice that the other columns part of the query do not appear as options here. This is because they were not really selected. They were *used* by the WHERE clause to match the info on the tables but were not called by the SELECT command). Click on "Next".

At this moment you might have noticed how the Report Builder, in the background behind the Wizard, has been automatically populated with the selections we have made so far.

In the second level, the Wizard wants to know what labels to use to name each field. The wizard uses the names of columns provided by the SQL code that either created the tables used or are defined in the code of our query. Because we took the time to enter informative names that use caps and spaces (and the double quotes they need), the names are actually quite appropriate. However, and just to get a feeling of how this works, change the "Description" label to "Phone type". Then click on "Next". Again, the Report Builder is updated by the Wizard.

In the third step, the Wizard needs to know if we want to use any grouping levels and offers us the names of the columns in our query. Let's explain what this means. Imagine that you have a patient, Stan Smith, that has a home phone number and an office number and a mobile phone and even a pager. When we display this information, we get something like this:

Patient Name	Number	Phone type
Smith, Stan	1234567890	Home
Smith, Stan	4567801239	Office
Smith, Stan	8529630147	Mobile
Smith, Stan	8945612307	Pager
Etc. ...	...	...

... repeating the name of the patient for each phone number she/he has. This is not very satisfactory. Instead we would like to group this information by name. This means that the name is displayed only once, followed by all the phone numbers that belong to him. That is what this step is offering us. To request for this we should select *Patient Name* from the box to the left and transfer it to the box to the right by clicking on the “>” button. You can see that the wizard offers us to transfer more columns if we wanted. Imagine that you have a database of movie and video producers from all over the world. If you were making a report of them you could group them by country, for example, and then add a second layer grouping them by specialty. You can notice some buttons to the right of the right box that display “^” and “v”. These allow you to change the order of precedence. If you place the specialty on top of the country then the report would organize your producers first by specialty and then by country, and this is not a minor difference!

However, for our example it is enough that we group our results by patient name only. Select this and click on “Next”. If you can peek at the Report Builder in the background you will see that, automatically and following our instructions, it placed the patient name on top of the other two fields. It also created a new section on the structure of the page, indicated by a blue shade at the left margin called “Patient Name Header”. This is a header because it will head a section of multiple results. This structure is repeated for every record in the patient table. Later, when we want to create groupings without the help of the wizard, we will be looking to place headers like this. Because now we know what they mean, it will be as simple and more flexible.

Now the Wizard wants to know if we want to give any particular order to the display of our data. Consistent with the SQL code in the query, the “Patient Name” column appears selected in ascending order, that is, alphabetically. This is enough to conform the requirements of the specification we did earlier. But now that we are here we can have a little fun and request that the “Description” field be also ordered. Go to the drop-down box under the “Then by” label and select “Description”. Now click on *Descending order*. This means that Stan Smith's number would appear in the order: Pager, Office, Mobile and Home. Notice that the Wizard gives us up to 4 levels of sorting. The levels also denote a hierarchy. In our example, the data will be ordered by name first and then by description (phone type).

In the fifth step the wizard gives us some options to organize the layout of the information in our report. Of course, the options are not many and in this task the Report Builder will excel, giving us enormous flexibility. At this moment I will select the “In blocks, levels above” option. Notice that the wizard has no options for the footers and headers. That is fine: the Report Builder will help us with

this. Below, the wizard asks us what orientation we want to give to the page. Because most of the time the information appears as tables with long rows the default option is “Landscape”, which makes plenty of sense. But in this case we only have three columns and one of them is being used as a header, so the “Portrait” orientation results in a better use of our paper. Select this option. The report builder adjusts immediately and changes the layout of the graphic interface. Don't get startled by this. Now click on “Next”.

In the last step the wizard needs to know three things: i) the name we want to give to this report, ii) whether we want to create a dynamic or a static report and iii) whether we want to keep on working on the layout of the report or if we want to produce it now. For the name of the report, the Wizard will offer us the name of the query used to build it. We could change it to something like: Patient Phone List or whatever you find informative. In any event, if you used my suggestion for the name of the query, erase the “qry” prefix so you don't get confused. A *Static Report* is built with the data in the database at the time the report is created and is not updated when the data is edited. This could be necessary for information that is time sensitive and that you are interested in tracking, allowing you something like taking a photo of the data at a moment in time. In contrast, a *Dynamic Report* acts as a template of a report, rebuilding itself every time it's called and reflecting any editions to the database. This is necessary when you need the reports to always be up to date. At this time select *Dynamic Report*.

We could go to fine tune the report and select the “Modify report layout” option but instead I am curious about the result so far so I am going to ask for the Wizard to create the report now. Do the same.

TADAAAH! Here we are: Our first report! This appears in a viewer as a .rpt document which is Read-Only. Now we have icons that allow us to print it directly or export it as a PDF document. This is very cool!

The information is quite clear. I see that the wizard included a horizontal line to separate the patient name header from the numbers and this helps to make the report easier to browse. I like this. The font chosen is a Sans Serif, which looks modern and is easy to read in a tabular context. Fine too. After initial happiness, however, I begin to notice several things I would like to change. First, we need a header with urgency. Only because I just created this piece of paper I understand what it is but give me a couple of weeks and I will have to scan it for several seconds before I remember what is this report about.

Then, I see than the label “Patient Name” is completely redundant. Actually, all labels are redundant in this report. This is because we don't have many categories and the information is quite self explanatory. Also the boldness of the labels attracts a lot of attention. In reality I would like the patient names to be in bold, so it can help me structure the layout of the page.

Finally, I would also like the distance between phone numbers to be less, as I feel that it makes it more difficult to perceive the structure of the page and also makes me feel that we are wasting too much paper. The font size could also be smaller, allowing more records per page and making the page easier to read. However, I do admit that the actual font size also fills the page quite nicely, providing with an harmonious balance of ink and white.

Of course, this fine-tuning is done with the Report Builder. Let's go. Start by closing the viewer and then go to the Report section of the Base interface. Find your report and right click on it. Then select the option “Edit”. The Report builder opens.

First of all, make sure that you have the Properties panel on (usually found to the right of the screen). Most of the times it appears automatically. If not, you can click on View>Properties or you can click on the second button at the lower level of the toolbar and to the left; the one whose tooltip indicates as “Properties” and that displays several form controls.

Let's start by providing a title to the report. First select the page header and then click on “Label Field” (the button to the right of the Properties button). While on the header, your cursor changes to the shape of a cross and a small box to the right. Use this to click and drag a rectangle. Immediately, the Properties panel displays its properties. This label object only has a *General* tab, but here we find all we need. First, enter into the *Label* attribute the text that you want. I used “Patient Phone Contact Information”. The default font size is not very impressive for a title. Click on the “...” button next to “Font”; this calls the Font dialog box. Leave the defaults but change the size to 22. You will notice that the text is now bigger than the boundaries of the box that holds the text. Left-click on it until you see the green boxes delimiting the boundaries of the text box and drag them until the entire caption can be read. Actually, drag the left boundary of the box all the way to the left margin of the page and drag the right boundary to the right margin of the page. Now look in the Properties panel for the “Alignment” attribute and select “Center. This ensures that the title will be centered. Now, just to show you the possibilities, call again for the “Font” dialog box and select the “Font Effects” tab. Click on the “Underlining” drop-down box and click on “Double”.

Now select the “Patient Name” label and delete it. Actually, delete all the labels. The rectangles that remain are Text Boxes. You can notice that the text box for the patient name field is somewhat to the right of the page. Select it by left-clicking on it and positioning your cursor over the box. The cursor will change to a black cross with arrow heads. You can now drag the box to the left. Use the guide lines that appear to align it with the box underneath it. The Text boxes don't really need to be so long. You can make them smaller by positioning the cursor on top of the green boxes. At that moment, your cursor changes to a white double-headed arrow with which you can change the size of the boxes in any direction. Reorganize them to your own taste.

Let's make sure that the patient name will be displayed in bold. Select this text box and click on the “...” button of the font properties. You can see that you can add any effect that you could like, even color the field. For this exercise, click on the “Font” tab and select “Bold” under typeface. Exit by selecting “OK”.

To arrange the distance between the phone numbers we need to select the *Detail* section under the Patient Name heading. If you check the General tab of the Properties panel after making the selection you will see the Height property, which, in my computer, is set to 0.50”. You can change the height here, by changing the number in the box, or you can drag the gray line that separates the detail from the page footer. Try this now and see how the height data in the Properties box is automatically updated. (make sure that no other object is selected).

To make the detail slimmer I found that I also needed to adjust the position of the boxes for number and phone type, moving them closer to the header (I also gave them some indentation by moving them to the right) and finally left the detail with a height of 0.40”.

Let's finish this by including the date the report is created and folios to number the page. Remember that we made this a dynamic form, so we need to update this data each time the report is called. If we used a label to write down the date, we would need to change it every time, which is not helpful. In

contrast, the Report Builder offers us an automatic option: For this, activate the page header by clicking on it. Then select on *Insert* and click on *Date and Time...* . At this moment the Date and Time dialog box appears. The dialog box shows two check boxes so you can decide if you want to include the time, the date or both. Drop-down boxes allow you to chose the format in which you want them displayed. I don't want the time in the report, only the date, so I unchecked the time box. Then I selected the format that I fancied best. Do the same.

After clicking on "OK" a box appeared on the page header at the upper-left margin. Now you can drag it, resize it and, in general, change any attributes permitted by the Properties dialog box. I moved mine to the right and under the report title. I also added a label to the left of this box wit the caption: "Report created on:". This way I make obvious the meaning of the date. Although not completely necessary, it provides a good excuse to practice your new skills. Go ahead and do the same.

Finally, let's add folios to our page. Select *Insert* and then click on *Page Numbers...* . The corresponding dialog box appears. First, select a format that you like (I picked *n of m*). For the position, indicate that you want this on the footer. Finally, select a left alignment and click on "OK". A box will appear on the footer. With this, I believe that our work matches the specifications we previously did for this report quite well!

Let's see our masterpiece in action. Lets first save it, just in case we have a crash (it could happen) by clicking on the blue diskette or clicking on File>Save. Now find a button in the first row of the toolbar with a page and a green arrow pointing to it from the left. This is the Execute Report button. Alternatively, you can find it under Edit>Execute Report or you can activate this function directly with Ctrl+E.

What do you think?

Fell free to experiment and analyze the attributes in the Properties dialog panel for the different objects. Be curious about the menus and the functions of the buttons. Play with them. This is the very best way to learn!

Now that we have the basics. Let's produce our Clinic Summary report.

Steps in designing a report:

In general, when you are producing a report, you should follow more or less the following sequence:

1. Analyze the report's requirements and decide on the overall layout.
2. Select needed tables and columns.
3. Compose query (include functions).
4. Build report.

If you notice, this is exactly the same procedure we did while developing our first example. Not surprisingly, the first two steps can be carried away with paper and a pencil. In composing the query you need to make sure you are using the SQL language properly, for which the earlier part of this tutorial should come in handy. Only the fourth step is done sitting in front of the Report Builder. So you see: plan before executing.

The first thing that we need for our Clinical Summary is to design the layout for our report. I am going

to base this report on the draft presented in part II.

The header of the report should include the name of the clinic and maybe a logo. Because this header will appear for every page in the report and because this report displays data related to only one entry (the particular patient) I find that it makes sense that I also include the name of the patient in the header, just in case the report requires more than one page. The footer could have the page number, the date the report was created and maybe a disclaimer about confidentiality and contact data for the clinic.

The next thing is to identify the tables where our data is stored. This table will include info from the Patient table, the Psychiatrist table, the MD table, the Patient Phone table and the Medication table. Because we don't want to re-write the code every time we need a particular report, we will use the "Like" operator with the parameters name AND surname OR medical record number.

You should be able to produce the SQL code for this by now, and it should look like this:

[to be continued...]

Maintenance of a database:

1. Defragmenting your database.
2. Dumping tables as .txt files.
3. Importing .txt files to conform tables.

Some final words

End of Tutorial!!