

Document: Mid Level Tutorial for Base  
Working Title: Using Base for Database Applications  
Level: Beginner to intermediate

Written by: Mariano Casanova  
Technical advisor: Drew Jensen

Revision release: 2008-07-27  
Status: Draft

Content overview:

### Part I:

Where we introduce this tutorial and its scope. We start by analyzing what a database is and describe its different components: Tables, relationships, primary and foreign keys, columns, column attributes, deletion attributes and relationship cardinalities and finally we provide a definition of database and Base database. We later comment on forms and reports and on modeling data and goals of proper design, after which we provide an overview of UML diagrams to use as a conceptual vocabulary. We then summarily review phases in database design and the importance of Normal form. We also review First, Second and Third normal forms and how they aid in class extraction. After this we review the way that Base records attributes and the nature of the variables it uses, analyzing text variables, numeric variables, date and time variables and object variables and how choosing them properly will affect the performance of a Base application.

### Part II

Where we offer a real case example and, using the elements presented in part I, we start with a problem and end with a database design. We cover class formation, class extraction, the importance of atomization and descriptors. We apply normalization and other tools to decide on data structure and finally come up with a complete UML diagram of our database. We also stress the importance of using auxiliary elements like a Variables Definition List for the aid they provide and take some time to discuss the difference between physical names and logic names as they apply to coding the application. We even offer a standard way to name variables and constraints (**although we have not applied them to our own design yet!**) After this we analyze the problem of duplicity of roles in order to introduce the concept of super-class formation. We then analyze the forms and reports that we will use in conjunction with the design we have produced and describe the kind of features we will want them to have. Now that we understand how to design our tables, its connections, forms and reports, and what do the options that Base offers mean, we are ready to sit down in front of Base and start coding our database.

### Part III

Where we describe how to create-using Base- the database it took us so long to design, complete with forms and reports. We start by analyzing the set up and then explain how to use SQL commands for queries and the creation of tables and relationships. We then provide and analyze in depth the SQL code that produces our database, particularly, the "Create Table" and "Constraint" instructions. We also review how to read their descriptions when consulting the Wiki or other sources. We then describe OOBasic code for creating dynamic forms and the SQL and formatting instructions for producing the reports.

## Mid level tutorial for Base

### Introduction:

Databases are very useful programs that allow us to store information, organize it, retrieve it and even extract new information from it. OpenOffice.org offers a very powerful database system with Base. But because Base is a powerful and flexible application, you need to be able to make some informed decisions while working with it and this, in turn, requires some preparation.

This tutorial will try to help you better understand the options offered by Base while attempting to develop a functional application of a medium level of complexity. For this reason, the first part of the tutorial will review some important concepts in the design of databases, that is, on how to organize the information that you need to collect. The later part of this tutorial will show how to implement those decisions while developing an actual application with Base.

For the previous reason, you will see that part I is of a general nature that applies to any attempt to design a database, including Base. Part II is somewhat more specific to Base as it describes attributes that are particular to it. Part III is solely focused on Base and the way to implement your decision with it. However, the three parts are necessary to understand how to effectively design database applications with Base.

If you are reading this, chances are that you are a non-expert looking for answers to concrete questions and with very little time to spare. By following along these lines you will find concepts and tips that will return the time invested as you will be able to design more useful, flexible and reliable databases. As with everything, practice does perfect and mistakes are a problem only if you are planning not to learn from them.

This tutorial can not cover all aspects of Base. Base has many features, like the ability to be a front end for other database systems. We will focus on Base working with its own embedded database engine called HSQL. Even in this more narrow topic we can not cover all of its functionality. We can only try to provide some elements here that can help you build a working database model with which to keep track of some resources and processes. However, we hope that this tutorial will provide a foundation from which to continue your exploration of this application.

Please note that this text has been arranged as a tutorial and not as a reference manual. This means that the information here has been organized thinking more on aiding the generation of meaning. I strongly suggest that you make notes in the margins and summaries from this text and form later your own reference manual.

Base is a great tool at your disposal and, the more you know about it, the more functionality and flexibility you can get from it. Don't hesitate to read everything you can about Base.

## Part I: Elements in Database Design

### 1. Conceptualizing a database.

In the later part of this tutorial we are going to assemble a database of medium complexity with Base. That will be the *'how-to'*. But in order to understand the options we will exercise then, we need to review some of the concepts in database design, the *'why'* behind our actions. Instead of just providing arid definitions, we will work a plausible example to illustrate the concepts.

#### 1. a. Anatomy of a Database: Tables, attributes and relationships.

Let's start with some general statements that we will later attempt to explain: A database consists of a group of tables that are interrelated. A table records objects that have the same characteristics. These characteristics are called attributes and are decided by you depending on the purpose of your database and the information you find necessary to collect and store. One attribute in a table can reference a record in another table, allowing tables to relate information in ways that are very flexible and powerful when you later want to retrieve information from your database. We will examine this in depth. When we say “database structure” we mean the collection of attributes you have chosen to build your tables with and the way these tables connect with one another. The process of deciding on a database structure is called “data modeling”. **The heart of database design, then, consists on forming** tables and deciding how to interconnect them. This topic will occupy the rest of part I and part II of this tutorial.

Let's imagine that you have collected over 10.000 books (I know someone who has done this!). Maybe you don't read them all but like to know that you have them, who wrote them, when and things like that. Memory will fail with a big number like this and keeping and consulting a written log can be very cumbersome as well. Enter Base and the help of databases.

A database is a way of storing information that can be easily retrieved later. You can retrieve a particular record (e. g. Who is the author of “Around the World in 80 days”?) or a summary of information (e. g. list all books by Jules Verne).

The information in a database is organized in tables. We will see later why it makes sense to use many tables in one database instead of just one big table. If you can find an example where just one table would suffice, then you might find it more straightforward to work it with Calc instead.

Tables are organized in columns (from top to bottom) and rows (from left to right). The columns represent different attributes that you want to store. For example, you can create the table “Authors” with the columns to store the attributes: First Name, Surname, Date of Birth, Country of Birth and Date of Death. Each row will hold one particular author, like: Jules Verne, Alexander Dumas or Pablo Neruda. Each row is called a 'record'. Each cell -the intersection of a row with an attribute- can also be called a 'record', just to confuse you. Rows are said to store “objects” (see figure 1).

This row holds the names of the attributes for the 'Authors' table

Authors				
First Name	Surname	Date of Birth	Country of Birth	Date of Death
Alexander	Dumas	07/24/1802	France	12/05/1870
Pablo	Neruda	07/12/1904	Chile	09/29/1973
Jules	Verne	02/08/1828	France	03/24/1905

Row: Object or Record

Column: Attribute

Cell: Particular record

*Figure 1: The 'Authors' Table*

Notice this: some of the authors could not be dead but your table will not become obsolete if (or rather when) this happens. This shows that in designing your tables, which information you decide to include and which not, will have an impact on the overall usefulness of the database. Don't panic just yet, as we will be describing some systematic ways to decide on what data is relevant to collect. Besides, I am counting on your imagination and intelligence to sort this out.

Now that we can collect information about the authors, we want to record the books they wrote as well. Most of us would think about just adding more columns to record the books. For this you would need at least one column for each title assuming that you only record the name of the book. If you also want to record the year of publication and the country of first publication, for example, you would need three columns per title (see figure 2).

Authors										
FirstName	Surname	DOBirth	Country	DODeath	Book1	Publication1	Country1	Book2	Publication2	Country2

Biographic info of the authors

Books written by the authors

Notice 3 columns per book

and etc.

*Figure 2: Unpopulated table for authors and their books*

Let's say that each author has four or five titles, except for one that has twenty. You will need to create 20 columns (or 60!) if you want to store your data faithfully, most of which would not be used, wasting

computer memory and speed. And what if this author later writes a 21<sup>st</sup> book? You will need to re-structure your database- adding new columns- and face unforeseen consequences that can come to hunt you later.

Instead of taking this route we could decide to focus on the books instead and create a “Titles” table, with the attributes we need for each book, and later add the columns for Author's name, surname and the rest. This way it would not matter how many books one author writes and we would not waste space with cells that would never be populated (see figure 3).

Titles						
Book Name	Published	In Country	Author	Date of Birth	Country of Birth	Date of Death
The Three Musketeers	1844	France	A. Dumas	07/24/1802	France	12/05/1870
The Count of Monte Cristo	1846	France	A. Dumas	07/24/1802	France	12/05/1870
20 Love Poems and a song of despair	1924	Chile	P. Neruda	07/12/1904	Chile	09/29/1973
One hundred love sonnets	1960	Argentina	P. Neruda	07/12/1904	Chile	09/29/1973
Around the world in 80 days	1873	France	J. Verne	02/08/1828	France	03/24/1905
Invasion of the sea	1904	France	J. Verne	02/08/1828	France	03/24/1905

*Figure 3: A 'Titles' table with author information*

Still, this would create new problems: Note that we need to repeat the author's data for every book the author wrote or writes in the future. This complicates maintenance of the data. For example, if one author dies, you need to locate every book she wrote in order to add that data. You also open the door for inconsistencies. What if some books have one date of birth for this author while others have a different day? One (or both dates) is wrong. Now you need to know which one is the correct day and find and modify each wrong record.

Instead of trying to merge both tables into one, we will keep them separated, but we will find a way to **relate** the info in one table to the info in the other table. Modern computer databases like Base can do this easily, which is why they are called “Relational Databases”.

This mistake, trying to create one big comprehensive table, is very common and arises because beginners want to make sure that all the relevant data is connected and forget that there are tools that can link them in more flexible and powerful ways.

This is your first lesson in database design: each table needs to cover one topic and one topic only. Creating tables that hold the attributes of both authors and titles is a bad idea. Creating tables that mix employee and department attributes is a bad idea. Creating tables that mix customers and services fields is a bad idea. Instead, you need to create one table for books, one table for authors, one table for employees, one table for departments, one table for customers, one table for services and so on. If you are taking notes, write this down because I will ask you later.

1.b. Establishing relationships within tables.

How exactly do we link the 'Authors' table with the 'Titles' table? How do we make sure that, for example, the object 'Jules Verne' in the 'Authors' tables can reference 'Around the world in 80 days' in the 'Titles' table AND all the other related books? To accomplish this, you need to chose a field in the 'Authors' table that uniquely identifies each record. The chosen field that uniquely identifies each record is called a '*Primary Key*'. Now you create an extra column in the 'Titles' table that is going to hold the value of the primary key. This column, that stores information on the primary key of the linked table, is said to store a '*Foreign Key*'.

For instance, you could decide that the 'Surname' field in the 'Authors' table will be the primary key. You then create a new column in the 'Titles' table -that you can call 'author' if you like. You then write the surname of the author at each book record, correspondingly, in the column that holds the foreign key: Verne, Dumas, Neruda, etc. This 'author' column in the 'Titles' table unequivocally links each book with one author (see figure 4). Base will use that index to link and retrieve the information that you will be asking for later.

Titles			
Book Name	Year published	Country of Publication	Author (FK)
The Three Musketeers	1844	France	Dumas
The Count of Monte Cristo	1846	France	Dumas
20 Love Poems and a song of despair	1924	Chile	Neruda
One hundred love sonnets	1960	Argentina	Neruda
Around the world in 80 days	1873	France	Verne
Invasion of the sea	1904	France	Verne

**Figure 4: 'Titles' table linked to the 'Authors' table through the 'Author' foreign key**

This way, for example, you could ask your database to list all “Book Name” where “Author” equals “Neruda”. Although you had not explicitly recorded such a list, the database can create it for you. Imagine how useful this is if, instead of six records, you really have ten thousand!

Using a surname as a primary key works fine many times, but has one drawback: what happens if you have two authors with the same surname?

One solution is to use more than just one field to form the primary key. In this case it is said that you have a *compound primary key*. For example, you could use the first name and the surname, together, to form the primary key. This solution is possible and perhaps in many cases the right one. Of course, we

can also think of cases when two authors have the same name and surname (like Alexander Dumas, father and son, for example). We could extend the notion of a compound key and work with a combination of three and even four fields to form a unequivocal primary key, but think of all the calculations the computer would have to do just to handle rare exceptions.

To simplify this, Base can automatically generate a unique number for each record in a table that you specify. This way each author would have a unique number (Dumas father would have one number, Dumas son would have a different number) that you can record in the column for the foreign key in the 'Titles' table. Instead of writing 'Verne', 'Dumas' 'Neruda', the computer will write 003, 004, 005 or something like this. You, as a user of the database, might not even be aware of the existence of these numbers or what numbers they are exactly. The relevant thing, however, is that these automatically generated numbers allow for a unequivocal connection between one object in the 'Authors' table with one or more objects in the 'Titles' table.

At other times, however, you could be using primary keys that are not numbers or primary keys that are compound. Keep this in mind.

Our example uses only one foreign key in the 'Titles' table, but don't be surprised if you find yourself needing to record two or more foreign keys in a table. This is how tables are related!

The primary key is important in more ways than one, and we will check this when we review the process of Normalization.

1.c. Cardinality of the relationship:

There is an aspect of the relationship that is very important that we analyze now: Note that one author will have at least one and possibly many titles. On the other hand, one title can have exactly one author only (let's leave collaborations aside for now. I promise to include them by the end of this section). When you say : “one author, many titles” you are talking about the cardinality of the relationship.

Note again that the relationship is not the same for authors and titles: one author can have many titles, titles have exactly one author. So, when we analyze the relationship from author to title, the cardinals are: 1..n (one to many). When we analyze the relationship from title to author, the cardinals are: 1..1 (one to one). Strictly speaking then, relationships always have two sets of cardinals.

The options for cardinality include:

- Zero to one (0..1)
- Zero to many (0..n)
- One to one (1..1)
- One to many (1..n)
- Many to many(n..n)

Zero to one and zero to many imply the possibility that one object in the first table is associated to no object in the second table, to exactly one or to many. The Zero cardinal opens the possibility for the existence of an object even if is not associated to other objects in other tables. If the cardinality from title to author had been 0..1, that would have meant that you can register a book even if you don't know who the author is. Here you have something extra to think about: whether your database needs, or needs to avoid, objects in one table that are not associated to objects in the other table.

One to one cardinality connotes properties of objects in one table that are expanded in the second table. Let's say that some of your books have beautiful illustrations -paintings, engravings, photos, etc- and others don't. You would want to register who was the artist, the name of the art piece and other data. It would be wasteful to include these attributes in the 'Titles' table, because most books would leave these records unpopulated. Notice the Set-subset relationship here. You have a set: books, and a subset:books with illustrations. You record the attributes common to all in the 'Titles' table and then create a new table - "Art info" for example- where you record the extra attributes of the subset. Oh! And of course, the foreign key. Every time you have a set-subset situation , 1..1 cardinalities come in handy. Note that if this cardinality were the same at both sides of the relationship (in the direction subset-set) then maybe both classes are really one big class.

Many to many relationships can not be performed without the use of an intermediate table. For example, one author could write one or more books; at the same time, one book could be written by several authors (in collaboration). In order to keep track of this, you will need a simple table -maybe with only two columns- between the 'Author' and 'Titles' tables that can record all the combinations of book and author. Every time you encounter a n..n cardinality, you now that you will be using an intermediate table.

#### 1.d. Managing relationships with column and delete options.

In order to enforce and make tidy the inclusion of primary keys, foreign keys and the cardinalities of the relationships, Base allows you to specify certain options for the columns (attributes) in your tables. In this tutorial we will consider the following ones:

Key: This option tells Base that this column will hold the primary key of the table. Base will prepare then to see it associated with other tables.

Unique: When you specify this option, Base will make sure that records in this column are not repeated. If, for example, you specify that the 'surname' column be unique, then the second time you try to enter 'Dumas', Base will reject it as an invalid entry. It makes a lot of sense to make sure that a column set to KEY is also set to UNIQUE.

Not null: This option means that records can not be left with this attribute empty. Base will display an error message if you try to enter a record that leaves a NOT NULL column empty. This forces whomever is using your database to at least have the information requested in the NOT NULL columns if they want to enter the record. For example, if you set the 'surname' and 'date of birth' as NOT NULL, then a user can not input a new author if he doesn't have at least the surname and the date of birth. Again, it makes sense that Key columns are also set to NOT NULL. This option can also affect cardinalities. If you have decided that your database should be able to accept a book entry even if it is not associated to an author, you can not set the foreign key to NOT NULL. On the other hand, if you will not accept a book entry unless it is associated to an author, the foreign key should be set to NOT NULL in order to enforce this.

Because being able to relate the object in one table to another object in another table is so important, special care must be placed on the subject of deleting records. Think about this: Let's say that there is an author that wrote only one book and you discover that you no longer have that book in your collection. As you update your database and erase that title, what will happen to the 'author' information? Should it be deleted too? Should it be kept as an historical record or in case you find and

buy one of his books again?

Actually, both options are valid and you can choose the one that reflects the purpose of your database better. But your application will not know what to do unless you make explicit what your preference is. For this reason, when you are developing your application and defining relationships, Base will need instructions on how to handle the deletion of records and will offer you the following options. This is what they mean:

No action: Base will delete the record you want but will not delete the records associated with it. This way, you can delete the missing book and keep the record of the author that wrote it.

Delete Cascade: With this option, Base will delete the record you are requesting to delete and will also delete all other records that have this record's key as a foreign key. This option is called cascade because it elicits the image of a deletion creating further deletions. Following the example, if you delete the author, any book associated to him will be deleted too.

Set Null: With this option, Base will delete the record you are requesting but will not delete the other records related to it. Instead it will erase their foreign keys to reflect that they are no longer associated to other objects. Note that this requires that NOT NULL is not a condition of the foreign key column. If you decide to erase the author in the example, the book record would not be deleted but you would find that it no longer has data for the foreign key, i. e. It would be set to null.

Set Default: When deleting an object, the foreign key column of the associated tables will be populated with a default parameter that you previously specify. This way, instead of just leaving an empty foreign key in the book record, Base could write, for example (and this is completely arbitrary) "000", which you know means "I have no author info for this book in this database".

## 2. A definition of Database and database design:

It took some time, but now that we have described a database and its properties, we are ready to offer a definition of Database and Base Database.

We have seen that database design uses many tables that are all related one way or another. Each table will only cover one particular topic or unity (authors, titles, places, events, etc.). These topics are called classes. A class is a collection of objects that have the same attributes. In database theory, each class translates to a table.

With that said, we can attempt to define a Relational Database as a collection of objects -organized in classes- and their relationships.

This definition didn't take long to write, but by now you know that 'object' has a rather precise meaning, that 'class' is a very important concept, and that there is a big chunk of knowledge around the concept of 'relationship'. You won't be deceived by this illusory simplicity!

Database design then is about deciding on class structure (which classes to work with and which attributes to record in each) and the structure of connections they establish (which table connects with which and with what cardinalities).

Of course, you also need a method to add records to your tables. And you need a method to retrieve

useful information -like particular records or summaries- to produce reports. Base offers *forms* and *queries* for this. For this reason, when we say a 'Base database', we mean not only the data and their relationships but also the forms and queries to use it.

### 3. Talking about Forms and Reports...

Forms allow you to enter data to populate your tables. You can build them by using the Form Wizard, which simplifies the task, or in design mode, which gives you maximum flexibility. There are several excellent tutorials that teach you how to build forms with Base and, if you haven't checked them yet, you should before we move to the third part of this tutorial.

In any event, you will need to think about what you want to achieve from your forms before you build them and what do the different options offered in building them mean. The time you spend thinking about the design of your forms will be paid back with better performance and efficiency.

Among these considerations, you need to make sure that the user of your form understands exactly what she/he is being asked (e.g. To the entry "Sex:" do you input "Male" or "Scarcely"?). It is not uncommon that we use words believing that they have a very precise meaning but later discover that they can be ambiguous. That is because when we use a word, it correlates strongly with an image in our mind, but our readers could have other associations. The context in which they appear can also suggest meanings that were not intended. To avoid traps like these you need to test and test and test your forms, asking friends to read them and give you feedback about what they understand and how they think they should answer. Even if you think that you will be the only one using your forms, you never know when someone will show up to help you with data entry or when someone will ask you for a copy of your application to use it herself.

The other important aspect of forms is that you want to ensure that data entry is normalized, that is, that the same element is entered in the same way every time. In general, Base will consider as different entities two spellings of the same word. Also, if you are not consistent with the use of capitals, Base can consider 'Verne' and 'verne', for example, as two different objects. This could lead to wrong summaries of data or records impossible to find. Base offers a type of variable that treats words with the same letters as the same word, no matter how you capitalize them. This can be very helpful. However, if you are keeping customer data, you would not want to have them receive mail with their names carelessly treated, as in "Dear Mr. SMith", so you still need to monitor input.

Base offers several ways to handle this. One of my favorites is the drop-down menu, where you point-and-click your entry. You can also use check lists and circular buttons. What they have in common is that they automatize the entry of data. Base also offers functions with which you can uniform data entry, for example, changing all letters to small caps (as in "proCEduRe" to "procedure"). Of course, this means more work while you are developing your form, but Base makes it really easy and it's worth the extra effort.

For data output you have the Reports. They are usually built by queries and you also have the options of either getting help from the Wizard or using the full flexibility of Design Mode. Again, you should read the tutorials that explain how to build them if you haven't read them by now. What I want to point out is that reports need to be easy to read, unambiguous and provide the necessary information. This means that spending some time in their design is also bound to repay you later with added efficiency. For example, I always recommend to include time data in a report so we can precise the validity of the information.

So get as many people as you can to test your forms and reports. At the same time, test your database design. See if someone can fool your input options or find exceptions that are possible but that your design can't handle. Test, Test, Test. It is easier to make changes at earlier stages than trying to repair a live version.

## 5. Modeling data and goals of proper design

In designing your database, you will need to define what are the classes that you will be working with (books, authors, etc.) and what relationships within them you need. Then you need to decide on the structure of each table, that is, what attributes you need to record for each class (name, surname, etc. or title, year of publication, etc.) and what properties you will give to the columns (not null, unique, etc.). You can't attempt to collect ALL information about your classes and you will need to edit your selections according to the **purpose** of your database. You will then need to refine the structure of connections, deciding not only what table connects with which, but also the cardinality they use, the type of primary keys that you choose and how to treat deletions. These decisions might also influence, or be influenced by, the order in which your forms ask for the information and the reports that will make sense to produce. When you are deciding all this and fine tuning your decisions, it is said that you are modeling your data.

Don't feel dismayed by all this. The subject of database design is vast and complex enough to comprise several years of college education. So it is going to take some time and patience for you to feel comfortable with these concepts and the ways to implement them. But you don't need to be a rocket scientist to become quite proficient at working with Base. The description offered in the previous paragraph is quite a good summary. Read it and translate to a numbered list and then try to visualize your actions at each step.

One of the most important aspects in data modeling is defining (also said 'extracting') your classes. We will review two ways for doing this. The first is a consequence of your activities during the different phases of database design, which we will describe shortly<sup>1</sup>. The second is a formal method called Normalization. Normalization is a formal method because it does not analyze the content of your tables (it does not care if the classes are books or authors or if the attributes are names or dates, etc.) but focuses on certain mechanical connections or characteristics of the primary key. There are several normal forms and, in this tutorial, we will review three: first, second and third normal form. In your work you will be doing both, content analysis and formal analysis.

There is no one way to design a relational database. Your design will reflect your level of preparation and experience. The techniques we will review here are only tools to help you make better decisions. In general, however, you will want your database to avoid repeating information as much as possible, while still recording all the data you need; that your database can grow, as your collection of data grows, without a breakdown in functionality, and that your database has some flexibility to handle entries that are not common but possible. Principally, you want your database to provide the information you need and that such information is valid for the sample it was extracted from.

## 6. Visual vocabulary

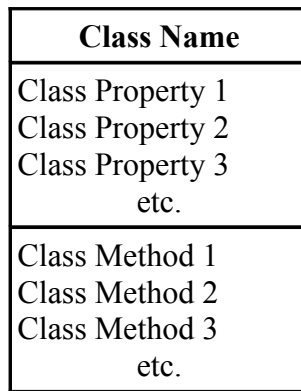
Database design is aided by a diagram protocol called UML, acronym for Unified Modeling Language.

---

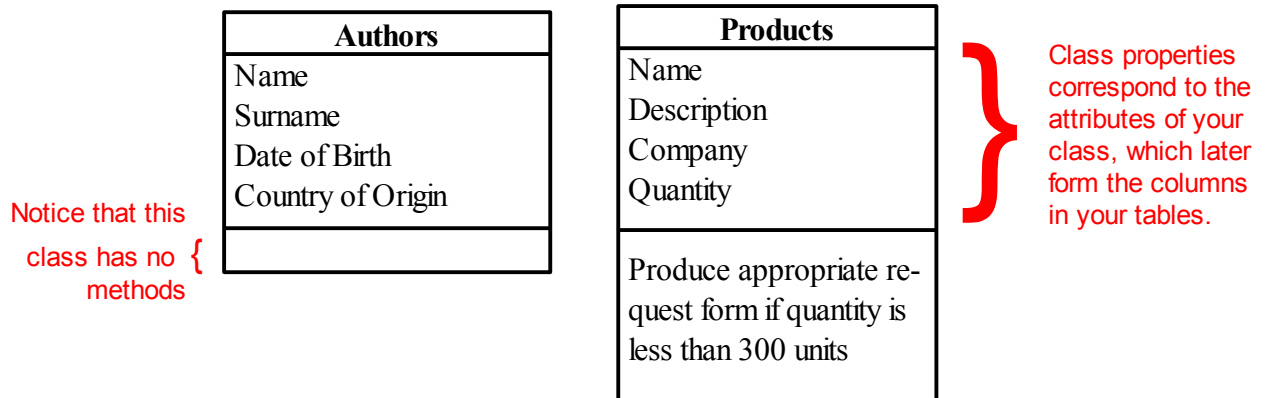
<sup>1</sup>Yes, let's use numbered paragraphs!

UML provides a standardized way to signify components or aspects of the Database. Let's see this:

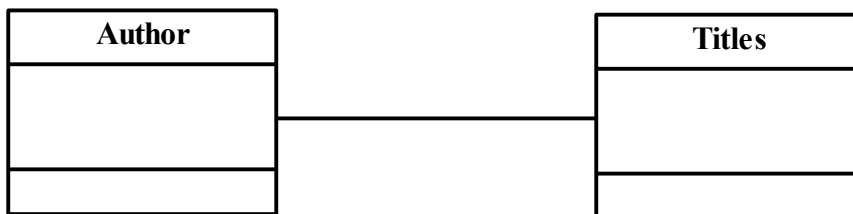
Classes are often notated this way:



See the examples:



The relationships are commonly notated by a line or arrow between the classes:



Because in the design of a database you are mostly concerned with the classes and their relationships, these two elements will suffice for now to describe, think about and communicate with others the structure we give to our data (That is, the structure of our classes -which attributes we will want to record- and the structure of connections -what table connects with what table and with what cardinality). UML offers other elements but we will not review them here.

You can imagine that, in the process of fleshing out the design of your database, you are prone to spend a lot of time drawing boxes, connecting them with lines, deciding on cardinalities and deciding which attributes to include in each class. It's funny to notice that this very important phase of database design

is best aided by a simple pencil and paper. However, this is very powerful. We have already hinted at the fact that you will be working with several tables and the more thought you invest in the design, the more flexible, reliable and functional it might become. Changes in design now are just a matter of erasing a line and drawing a new one, or deleting or adding attributes; instead of trying to undo part of your work in front of the computer while trying to retain other decisions at the same time. With the level of abstraction offered by UML, you can focus on deciding the general structure of your database and deal with the details later.

Let me note that Draw offers a flexible set of tools for drawing boxes and connecting them with arrows. Draw can even let you easily adjust the arrows, for example, if you have several boxes and need to surf the connections between them. You can also find other freeware for doing this, like Dia (a GNU license project) that comes with a complete set of UML tools.

Later, when we discuss the database we will work with in the *hands-on* component of this tutorial, you will see how our design decisions are represented in the diagrams, how this simplifies the communication about the design and how this translates into the development of the application.

A more formal rendition of your design should include the names of your attributes in the corresponding classes and the identification of the primary and foreign keys. The relationships should indicate the cardinalities and the arrows should go from the primary key to the foreign keys (figure 5).

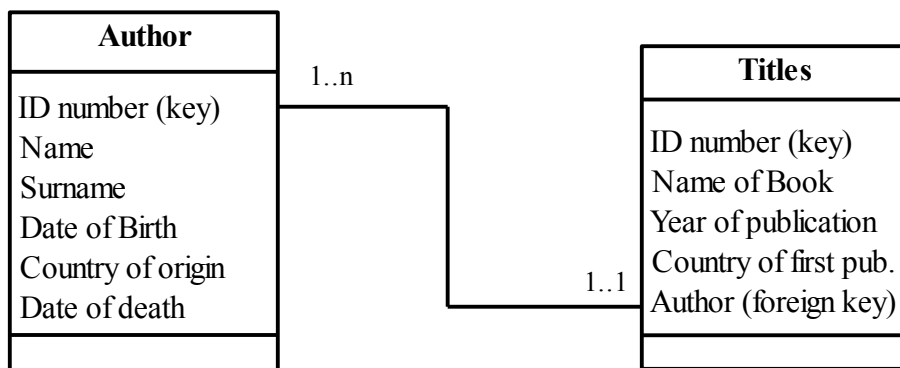


Figure 5

## 7. Phases in database design

In general, the effort to design a database will start with someone finding a problem in the real world. It can be a lawyer trying to organize his clients and cases in court or a company trying to keep inventory of on-line sells. This person will need the help from someone with knowledge in database design and in implementing it with Base. Let's call the person a 'client' and the expert the 'developer'.

Of course, the client and the developer can be the same person -and if you are reading this, then this is most probably the case. But I make the distinction because a dialog between the client and the developer is always necessary, even if it only takes place inside someone's head. In general, the client has a coarse idea of what he wants and the developer will ask questions in order to be able to translate the need into a well developed database application.

Broadly, the phases of database development can be described in the following way:

1. Presenting problem is identified
2. Possible solution is defined
3. Data modeling
4. Model testing
5. Database running and maintenance

During the first phase, the client tries to explain what his needs are -which are often multiple- and the expert tries to help him narrow it down to the kind of tasks a database can really perform. In the second phase, the expert develops a formal statement of the goals and scope of the database application. This is important because a small change in a goal can produce important changes in the design of the database. Besides, you will discover yourself becoming more ambitious with what you want your database to do. If you don't set a line somewhere, you can find yourself making your model progressively more and more complex and never actually develop your application or, worse, have it perform erratically as you keep modifying data structure.

Now that, as the expert, you have defined the scope and goals of the database, you are in the privileged position to decide which information is relevant and which not. You can make better decisions that can help define classes and attributes, primary keys and relationships with zero or one cardinalities. This is the extraction of classes we earlier identified with **content analysis**. You can also sketch and consider the forms and reports that your application will need and how they integrate with the design. Of course, you also have to test your logic and grammar by asking friends to find holes in your design and meaning in your reports and forms. This two phases do not have a clear boundary because the testing will impose changes in your design and the new design will need to be tested. Don't mind to spend some time here and test, test test!

Now that you feel confident of your database design (class structure and structure of connections) and the forms and reports you want to use, it is time to code them with Base. Now you are sitting in front of the computer for the first time since the project started! Of course, if you are in the process of learning then I strongly encourage you to sit and play around with Base, the more the better, so that you can become acquainted with its different screens, dialog boxes and options. But if you are in the task of designing a database, this is the first time that you really need to interact with the computer.

During maintenance, you test that your database remains consistent and able to handle the growing number of data it receives. Now is when you can also try to introduce the features that you didn't think of before that can make your application run faster or more efficiently. Of course, these changes need to be permitted by the structure of your database. When your structure can no longer accommodate the new functionalities that you want, then maybe it's time to go for a new design.

The more that you, as the expert, question the client and become familiar with the business process, the better your decisions for class extraction and table structure. Usually, the actors involved in the productive process (customers, providers of service, suppliers, investors, etc.) become classes. Different phases in the production processes (recorded as schedules or logs), places (e.g. the different stores of the chain) and even events (e.g. rendition of services) can also become classes. Some data can be better recorded as tables; some other times you will find that the same data is better recorded as attributes. The difference is set by the purpose (functionality) of the database. Here is where imagination and experience come in very handy. Try out your ideas and see what happens. Read about database design and don't fear to ask other people (especially if they know something about database design!).

The other tool for deciding on class extraction is Normalization, which deserves a chapter of its own.

## 8. Getting into normal form.

Normalization is a formal way to ensure that our tables have a good structure, that is, that each table has the appropriate attributes.

Typical problems avoided with proper normalization include:

- Unable to create a record without information from a foreign object.
- Deleting one record will result in other data being lost.
- Data being repeated in other fields (or tables) resulting in possible inconsistent records of data.

Many of these problems have to do with deciding if certain data should be organized by itself -in a table of its own- or as fields in a host table.

Because this is a formal method, we will not deal with the actual content of our tables (which is what we are supposed to do the rest of the time anyway) but will analyze certain mechanical connections of the primary key.

A key concept of normalization is that of 'Functional Dependency'. This concept means that there is one field (the primary key) or more fields (for a compound primary key) of a record that is/are enough to help me identify any other field (attribute) of that record, that is, knowing the values of any field **depends** on knowing the value of the primary key.

Clearly, true primary keys help me identify -or determine- all other attributes in a record (row).

This also means that, if we have a compound key and one of the fields in it turns out to be superfluous or redundant, then this primary key is not a true primary key, just a key. Therefore, a primary key has no subset of fields that is also a primary key.

Many times, the vastness and complexity of the data or our database obscure this leanness, potentially creating problems like those described above. By analyzing functional dependency -at several levels- we can identify if our data structure is compromised or not and avoid those problems. There are several normal forms and in this tutorial we will cover three: first, second and third normal form.

### 8.a. First normal form.

According to this rule, *a table must not try to keep multiple values for a piece of information, whether in one field or in multiple columns.*

We would have this situation if we tried to include in the 'Authors' table all the books written by them. Let's say that we create a column called 'Books' and then we cram there 'The Count of Monte Cristo', 'The Three Musketeers', 'The Man in the Iron Mask' and etc., all separated by comas, for the record of A. Dumas. Even if we use several columns (Book1, Book2, Book3, etc.) we are still trying to cram multiple values (the name of the different titles) for one type of information (Books written by author).

We already saw other reasons why this is a bad idea. First normal form helps us identify this problem. It also offers a generic solution:

*If a table is not in first normal form, remove the multivalued information from the table and create a new table with that information. Include the primary key of the original table as a foreign key.*

In our example, this would result in the creation of the 'Titles' table, with info on the books, and a relationship to the 'Authors' table.

Although not immediately evident, first normal form solves a problem of functional dependency, as the authors' primary key would have not helped to identify a particular value for the multivalued field. If I ask you to provide a concrete title for a particular author, you need a second criteria to chose among the options.

Thinking in terms of extracting multivalued pieces of information is very simple and completely equivalent to the analysis we did in the earlier chapter with this same example.

#### 8.b. Second normal form

Second normal form requires that a table be in first normal form AND that *all fields in the table (that are not primary keys) can be identified only by the primary key and not by a subset of it.*

By definition, this problem can arise only when we have a compound primary key to identify a record.

To explain this, let's analyze the following table:

Teacher ID	Project Name	Department	Contact	Hours
mackormac032	Science Fair	Science Dept	344-2713	10
Phillipi72	Soccer coach	Athletic Dept	225-5902	18
mackormac032	Art Contest	Art Department	639-6798	7
Phillipi72	Science Fair	Science Dept	344-2713	15

Is this table in first normal form? Yes, If I tell you the primary key, you can identify unique values for the other fields.

In this case, the primary key is a compound key that requires the Teacher ID and the Project Name. I need both bits of information to know, for example, how many hours have been devoted by a certain teacher to a certain project.

But it is also true that I only need the Project name if I want to know the involved department or the contact information. Oops!

So, while I am thinking that Teacher ID and Project Name conform the compound primary key, it turns out that a subset of it, the Project Name, is a primary key for certain fields in this table. To solve this, we need to extract the info related to the subset primary key (Department and Contact) and form a new table with them, including the subset key they depend on (Project Name).

This process is called 'Decomposition'. Decomposing the table of the example we get:

Teacher ID	Project Name	Hours

and

Project Name	Department	Contact

Note that the original table retains the entire compound key (teacher Id and Project Name) and the info that functionally depends on it (Hours).

Lets re-sate this in formal parlance:

*A table is in second normal form if it is in first normal form and we need all the fields in the key to determine the value of non-key fields.*

*If a table is not in second normal form, remove the non-key fields that do not depend on the whole of the compound primary key, then create a table with these fields and the part of the primary key they do depend on.*

### 8.c. Third normal form

*Third normal form requires a table to be in second normal form AND that you can not use regular fields (that is, that are not key fields) to determine other regular fields.*

In other words, you need to make sure that only key fields determine non-key fields.

To make sense of this, let's analyze to following example:

Teacher ID	Surname	Name	Department ID	Department Name
fsmith089	Smith	Fred	17	Science
miling619	Ling	Mei	17	Science
syork233	York	Susan	18	History

If each teacher works for only one department, then this table is in first and second normal form. Let us analyze this:

1. If I know the teacher's primary key (the Teacher ID) I can tell you a unique value for all the other fields.
2. The primary key is not compound, so I have no possibility of subset conflicts.

However, notice that there is information about the department that is repeated (department ID and Department Name) and could become inconsistent (e.g. It seems that the Science Dept. has been assigned the ID number 17. If we later find a record that assigns the number 23 instead, we would have an inconsistency). Further, it is possible to determine the Department Name by knowing the Department ID, which is not a primary key, and *vice-versa*.

In this case, we *remove the regular fields that are dependent on the non-key field* (In this case, we

remove the Department Name) and create a table with it, in which we also include the field they depend on (the Department ID), left as their primary key.

This way we would have:

Teacher ID	Surname	Name	Department ID

and

Department ID	Department Name

In formal parlance we have:

*A table is in third normal form if it is in second normal form and we can not use non-key fields to determine other non-key fields.*

*If a table is not in third normal form, remove the regular fields that are dependent on the non-key field and create a table with them, including the field they depend on as their primary key.*

#### 8.d. Aiming for simplicity

An analysis of functional dependency can easily identify decisions in table structure that will create problems sooner or later. Notice that some of the decisions made with Normalization had already been implemented with content analysis. This shows that both tools have an area of overlap. This is fine, normalization can help make decisions when we are very in love with the table designs we have done and it hurts us to chop them further.

However, in the process of designing your database, you will be using both tools: Normalization and content analysis, i. e., the extraction of classes by understanding the production processes, the key roles involved, the required reports and the overall purpose of the database. Which one you use might depend on the nature of the problem at hand and your personal preferences.

#### 9. Recording Attributes: Are you my variable?

The data that you will store in your tables will actually be stored as variables in your computer. Computers have different ways of storing data. In general, they trade memory or speed for accuracy: Computations that require more accuracy tend to be slower and use more memory. When you are building your tables with Base, you are asked what kind of variables you want to store and are presented with a drop-down menu of options. The decisions you make will affect the performance of your database. To better understand what do these options mean and how they affect the way Base handles your variables, it is necessary to review the way computers handle data.

In general, computers handle numbers and characters differently. Some people are confused by the fact that numbers can be treated as characters also. e. g. '7' is a sign that represents the concept of seven. In that sense, it is stored in the same way as the signs '@' or '#' or the letter 'A'. As a character, the sign can not be operated mathematically, just as it would not make sense to calculate the sum of '@' and '#'. Phone numbers are good candidates for being stored as characters. Of course, any information that uses text is also stored as characters. Addresses need both, numbers and text, but we store them all as characters. To make obvious the fact that we are only storing the signs (e.g. '7') as opposed to the meaning (seven) whether they are letters or numbers, we call these 'Alphanumeric characters'.

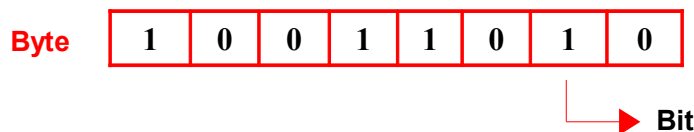
Computers have different ways of storing alphanumeric characters. For example there is the ASCII code, that needs only one byte to store a character. Unfortunately, this limits the number of possible characters that you can use to only 256. Although enough for many applications, it falls short if you want to have access to expanded character sets: Chinese or Arabic characters, accented vowels (diacritics) and things like that. Protocols that allow for larger numbers of characters have been developed, like Unicode, that use more bytes per character.

Base will store alphanumeric characters using UTF-8, which is a code that is compatible with both ASCII and Unicode. Base will use one or more bytes for each character according to internal calculations. When Base asks you the length for a particular record, e.g. Surname of Author, it is not asking the number of Bytes you want to allocate but for the number of characters you want to receive. How many bytes are actually used is fixed by the software.

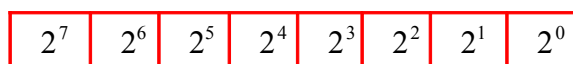
This is not the case when you store the value for a number. Different ways of storing numbers will require more or less bytes. Let's see why:

Computers store information in binary form. The unit is called a 'bit' and can have one of two values (thus binary): either zero or one (or On and Off, or 3.5 volts and 5 Volts, or any of two possible states). Bits are organized in larger groups -usually eight bits- to form a Byte.

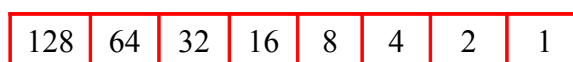
In this conceptual representation, a byte is drawn as a box with eight spaces. Each space can hold a zero or a one:



Being this a binary numeral system, each box represents a power of two, in the same way our decimal number system assigns successive powers of ten from right to left:



Which is to say:



To represent the number five, you would need to 'activate' or indicate the numbers for four and one

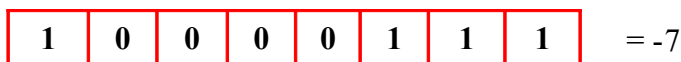
(because four plus one is five). So your byte representing five would look like this:



With a little bit of math you can figure out that you can represent all numbers from 0 to 255 (that is 256 numbers in total) using one byte.

If you need to store numbers bigger than 255 then you need to add extra bytes. Base offers different amounts of byte aggregates options to store numbers, depending on how potentially big the values that you need to record are. Now, what if you need to store negative numbers, say -7?

The solution for this has been to use the last bit to the left of your byte to represent the sign and use the other seven bits to represent the number, in this way:



Because the last bit no longer represents the value 128 but instead works as a flag to indicate if the number is positive or negative, we can represent numbers from -127 to 127 with this arrangement .

This type of bytes are called 'signed' bytes, while the bytes that only store positive numbers are called 'unsigned bytes'. The chart below indicates which options are signed or unsigned and helps you know the range of values you can store with them.

Again, if you need to store values beyond these boundaries, you need to add more bytes, which uses more memory per record. In any event, only the leftmost bit, also called 'Most Significant bit' (MSb) will act as a flag, that is, if you are using two bytes, the MSb will act as a flag and the other 15 bits will store numbers, including  $2^8$  ,  $2^9$  and all the way up to  $2^{14}$  . With this information, you can calculate the range of such a variable.

In general, numeric data variables are described by the number of bytes they use and whether they are signed or unsigned. These two factors determine the range of possible values they can hold. Base offers several types of numeric data variables, both signed and unsigned and using different amount of bytes.

At the least memory consuming side of number storage we have the Boolean numbers. A Boolean number is in fact just a bit, and we use it to store YES/NO type of data, like and answer to the question 'have you ever been to Hawaii? At the other end there are variables called 'float point numbers' (just 'Float' to family and friends) that allow us to store numbers that have decimal places like 1,618033. They are the most memory consuming numbers but the only ones that can perform divisions with good accuracy.

***Numeric Type Variables: Used for storing numeric values:***

Name	Data type	No. of Bytes	Signed	Range
Boolean	yes/no	1 Bit	----	0 - 1
Tinyint	Tiny Integer	1 Byte	No	0 – 255
Smallint	Small Integer	2 Bytes	Yes	-32768 to 32768
Integer	Integer	4 Bytes	Yes	-2.14* 10 <sup>9</sup> to 2.14* 10 <sup>9</sup>
Bigint	Big integer	8 Bytes	Yes	-2.3 * 10 <sup>18</sup> to 2.3 * 10 <sup>18</sup>
Numeric	Number	No limit	Yes	Unlimited
Decimal	Decimal	No limit	Yes	Unlimited
Real	Real	4 Bytes	Yes	5* 10 <sup>(-324)</sup> to 1.79* 10 <sup>308</sup>
Float	Float	4 Bytes	Yes	5* 10 <sup>(-324)</sup> to 1.79* 10 <sup>308</sup>
Double	Double	4 Bytes	Yes	5* 10 <sup>(-324)</sup> to 1.79* 10 <sup>308</sup>

You might have noticed that many variables in Base using HSQL have the same parameters. This is because those variables might behave differently when Base acts as a front end for other database systems. When you use the HSQL embedded engine -like in this tutorial-, you can consider these variables as interchangeable.

To be more precise, Base using HSQL understands NUMERIC and DECIMAL as one set of interchangeable types of numeric data. Their characteristic is that they can hold very large numbers. On the other hand, REAL, FLOAT and DOUBLE, also interchangeable, handle better the divisions. For example, if you store 8 and 10 as NUMERIC and then ask for 10/8, Base will return 1,2. If you store them as REAL, it will return: 1,25.

Don't be afraid to use all variable types at your disposal. Just make sure you assign the data type that best describes the values you need to store. If your database records the number of children or previous spouses of a person, and unsigned byte should be enough and a signed double would be a waste. Not only does this planning save memory but it also ensures that the application will run as fast as it can and will be less prone to breakdowns.

The same care should be exercised with alphanumeric characters. Although the number of bytes per character might depend on the code system used (e. g. ASCII or Unicode), Base allows you to limit the maximum number of characters it will accept for an entry. Here you have some options: Lets say that you have the attribute 'Surname'. You assign it a CHAR (fixed) data type and give it the value of 50. This means that Base will assign a space of **exactly** 50 characters to store each 'Surname' entry. What if you enter the name 'Verne', that only uses 5 characters? The computer will store 'Verne' and 45 blank characters. Instead, if you assign the variable type VAR CHAR (Var short for 'Variable') and the value 50 then the computer will store **up to** 50 characters and not more BUT would only store 5 characters for the entry 'Verne', saving you memory. If you try to store more than 50 characters, Base will only record the first 50 characters and ignore the rest.

You will have to do some research before assigning length values. 50 characters could seem like a waste for surnames, but five or then could be dangerously low if you think, for instance, in Indian or

Greek names. Also check address formats and the length of street names that could show up. You will notice in the chart below that the different alphanumeric data types can store, as of the time of this writing, up to 2 gigabytes of characters, which is more than plenty. You will also note that the parameters are the same for several data types. Again, in HSQL they can be considered equivalent.

***Alphanumeric Type Variables: Used for storing alphanumeric characters:***

Name	Data type	Max length	Description
Memo	Long Var Char	2 GB for 32 bit OS	Stores up to the max length or number indicated by user. It accepts any UTF 8 Character
Text (fix)	Char	2GB for 32 bit OS	Stores exactly the length specified by user. Pads with trailing spaces for shorter strings. Accepts any UTF 8 Character.
Text	Var Char	2GB for 32 bit OS	Stores up to the specified length. No padding (Same as long var char)
Text	Var Char Ignore Case	2GB for 32 bit OS	Stores up the the specified length. Comparisons are not case sensitive but stores capitals as you type them.

Another important type of variable is the DATE type. They are used to store calendar information like year, month, day, hour, minute, second and fraction of a second. There are several types, designed to be the most efficient in storing some or all of this information. Date allows you to store year, month and day as it is stored in your computer (yes, your computer keeps track of this. Other database systems use the time stored in servers in the Internet -which could be more precise, but then they require an Internet connection). The same is true for the Time type variable, which stores the time of the day: hour, minute and second. Be sure to understand the format in which this information is given. The US uses the month before the day. Other countries use the day before the month. Some countries use the am/pm format while others use the military format (e.g 19:30 hrs. for 7:30 in the evening). Finally, some procedures might need you to record both the time and day of an event. Timestamp has been designed for this, recording all information at once.

***Calendar Type Variables: used for storing dates and hours:***

Name	Description	Format
Date	Stores month, day and year information	1/1/99 or 1/1/9999
Time	Stores hour, minute and second info	Seconds since 1/1/1970
Timestamp	Stores date and time information	

The Binary type variables allow you to store any information that comes as a long string of zeros and ones. Digitized images use this format. Digitized sound uses this format too. In general, anything digitized is stored as a long sequence of zeros and ones. They are told apart by the computer because the first zeros and ones identify the kind of file they represent (a JPEG image or an MP3 file, etc.).

However, Base will make no attempt to identify the kind of file you have stored. This is to say that it won't care if the file is an MP3 or a TIFF and it will happily store it, regardless. The only limitation is the amount of memory the file uses. At the time of the writing of this tutorial, the maximum amount of memory Base will use to store Binary variables is 2 gigabytes.

This in effect means that you could use a Base database to store, for example, photos of the members of a project or the staff, or sound snippets or voice messages. However, be warned that images and sounds tend to use a lot of memory and limit the functionality of your database.

Again, the different Binary types can be assumed as interchangeable when using the embedded HSQL database engine.

***Binary Type variables: Used for storing files like JPEGs, Mp3s, etc.:***

<b>Name</b>	<b>Data type</b>	<b>Max length</b>	<b>Description</b>
Image	Long Var Binary	2GB for 32 bit OS	Stores any array of bytes (images, sounds, etc). No validation required.
Binary	Var Binary	2GB for 32 bit OS	Stores any array of bytes. No validation required.
Binary (fix)	Binary	2GB for 32 bit OS	Stores any array of bytes. No validation required.

Finally, there are two variable types offered by Base called OTHER and OBJECT. They are used to store small programs that really advanced developers of databases can use in their applications. Essentially they store binary data just like Binary but this time Base pays attention to the beginning of the code so it knows what kind of program it is and how to use it. We will not be using this type of variables in this tutorial.

***Other Variable types: For storing small computer programs in the Java language:***

<b>Name</b>	<b>Description</b>
Other	Stores serialized Java objects – user application must supply serialization routines
Object	Same

**Why is all this information relevant?**

Base, working with the embedded HSQL database engine, requires that all your database ( data, data structure and forms and reports) be in RAM memory while you work with it. This means that the number of records that you can keep will be affected by the size of your variables and the amount of RAM in your computer. The number of your reports and the speed in which they are calculated will also be affected by this.

In theory, HSQL limits your tables to 2 gigabytes (which is a lot of memory) and the overall size of your database to 8 gigabytes (remember, this includes the forms and reports -but still it's plenty). If you

had a table with an image that is roughly 2 gigabytes, you would not be able to record more information in that table. You would have one table with only one record in it.

In practice, however, and at the time of this writing, computers have 512 megabytes, 1 gigabyte or at most 2 gigabytes of memory, drastically reducing the resources available to your Base database. Although this is still plenty for a functional application, you can maximize your resources by spending some time assigning variable types and memory allocations (with CHARs) commensurate to the variables you need to record. Now you know!

## Part II: Let's get real

Now we need a problem... (case example)

Let's apply the phases of database design to a concrete and possible problem. We will consider the needs of a small psychotherapy center. The director will be the client and you and I will be the experts.

Our first step as experts will be to try and identify the presenting problem. Because this is a stylized version of the work, the most important questions -and answers- will be identified quite quickly. In real life, this can easily take several meetings or a good chunk of time devoted to understanding the productive process.

In our example we start by asking the client and learning what is the business about and who are the principal players. The clinic receives people looking for psychotherapy services. They are usually referred by their medical doctor, but not always. Once evaluated and admitted, they are assigned to a therapists according to relevant training and time availability. Then they show up for a number of sessions they or their insurances need to pay for. The director will then pay the therapists according to the number of patients they saw in a month.

Now that we have a broad understanding of the process, we need to know what the needs are: our client needs to record data from his clients and decide if they should be admitted there. He needs to organize the admitted clients, matching them to a therapists and finding an available time slot. He needs to keep track of the performed sessions, by therapists so he can pay them, and by client so he can charge them. He also needs to keep info about the therapists, their training and other data for tax purposes.

So we need to record info about the clients, the therapists, about their meetings and about the payments. After our first (and very stylized and lean) approach we can start to identify some classes we are going to need: The Client class, the Therapists class, the Schedule class and the Payment class.

We go back to the client for more information. We know by now that he needs to collect info about the potential client but that he also needs to perform an evaluation of several factors and later make a decision to admit them or not. Now we can see that there is contact info, evaluation info and resolution info. The dialog between Client and Expert continues: At what moment is the clinic's client registered into the database? When he calls seeking services or when it is resolved that he be admitted? Will all the information requested be available at the time that the record is entered? If not, what information is essential to decide to register a record?

Now, let's...