

Binding.

Dr. Tom Johnston.
MindfulData.com

Binding Values to Variables.

In its most universal sense, binding is the assignment of content to form. In the context of a language, it is the assignment of semantics to syntax. But IT professionals usually think of binding in the more specific sense of the assignment of values to variables in a computer program.

To illustrate, let's say that you have a cursor-controlled loop through a table of customers. The schema for that table is shown in Figure A1-1. A schema, in the specific sense in which I will use the expression, is a named and typed template of variables. A schema for a relational table is created with a CREATE TABLE statement, and it defines the *syntax* of a table.

Customer

cust- nbr	cust-nm	cust-status	q1- purch
char(7)	char(30)	char(3)

Figure A1-1. The Customer Table Schema.

A brief note on how to read these graphical representations of schemas. The top row contains the column names for the table. The left-to-right sequence of columns corresponds to the top-to-bottom sequence of columns in a CREATE TABLE statement. Following standard convention, I place the primary key column or columns first. In addition, primary keys columns will always be shown in boldface. Foreign keys are shown in italics.

The second row contains the data types and lengths (henceforth, “data types” by itself will be taken to include length, when length can be specified) of the columns.

Binding.

© Copyright 2008, Dr. Tom Johnston. Only personal, non-commercial copies, are permitted.

Page 1.

Neither of these rows, obviously, is an instance of a relational table. They are just table headings. However, the third row can be thought of as a “viewer”, in which “real” rows, i.e. instances of the table, appear. If I can illustrate my points one sample row at a time, I will use this “viewer” format. If it takes multiple rows, then the concept of a viewer goes away. In that case, the third row of this graphic of a relational table is then the first real row of some result set; the next row of this graphic is the second row of the result set, and so on.

As our cursor scrolls through each row of the table, a different set of values appears in the viewer. Let’s say that we have just moved the cursor to some row, and the following set of values pops up:

Customer

cust- nbr	cust-nm	cust-status	q1- purch
char(7)	char(30)	char(3)
CHI- 305	Jones	vip	\$386.04

Figure A1-2. A row of the Customer Table.

At this point – *but not before* – the value “CHI-305” is bound to the variable cust-nbr. The value “Jones” is bound to the variable cust-nm. And so on. At this point, if code were to display the current value of the variable cust-nm on the screen, it would display “Jones”.

When it’s time to move on, we bump the cursor, and get the next customer. Let’s say what pops up in the viewer this time is:

Customer

cust- nbr	cust-nm	cust-status	q1- purch
----------------------	---------	-------------	--------------	-------	-------

char(7)	char(30)	char(3)
GA-338	Smith	occ	\$705.85

Figure A1-3. Another Row of the Customer Table.

Now it is the value “GA-338” which is bound to the variable cust-nbr, and the value “Smith” which is bound to the variable cust-nm.

These variables are said to be “late-bound” to their values. The specific form of late-binding we see here is often called “run-time binding” because values are assigned to these variables only when the program is actually running.

The important question about binding is cost. (Indeed, the important question about *any* aspect of IT development and maintenance is cost.¹) And a critically important question that we should always ask ourselves is: how much does it cost to *change* the binding of variables?

Well, in this example, how much does it cost to execute the program which bumps a cursor through the rows of a Customer table? For all practical purposes, it costs *nothing*. As the variables are unbound from one set of values and bound to the next set, unbound from one customer row and bound to the next, the programmer doesn’t have to do anything.

Now let’s consider a program which calculates a gift certificate to be offered to each customer, based on how much the customer has purchased in the most recently-completed quarter. That total, for the first quarter, is contained in the q1-purch column. Management decides that the discount will be five percent of total purchases for the quarter.

Inside this program, there is a variable called cust-gift-pct-rate, type Decimal (3,2), and the variable is assigned the value “1.05”. Note that this value is assigned in the program itself. It is written in the source code, and compiled

¹ More precisely, of course, the important thing, the bottom-line thing, is the difference between cost and value provided, the “margin” so to speak. But if the objective has already been decided on, i.e. if management has already decided that a new system will be built, the focus then shifts to cost. This is because, with the decision to build having been made, the only way in which IT can improve the margin is to lower the cost.

into the object code. We often say in such cases that the value is *hardcoded* into the program.

This value could have been read from an external data source. If it had been, it would have been *run-time-bound* to the program. But having been written into the source code, it is said to be “early-bound”, and in particular *compile-time-bound*, to that program. It is compile-time bound because in order to change it, we would have to change the source code and recompile the program.

In this case, hardcoding, or compile-time-binding, is not a bad idea. Gift certificates are offered quarterly, but the discount rate – in this case, five percent – applies to all customers, and (let us suppose) is good for all four quarters. So this program won’t have to be re-written and re-compiled until next year. And the re-write, if needed, will involve only a one-line change (the line which assigns the value to `cust-gift-pct-rate`).

Good programmers instinctively (or sometimes consciously) try to minimize binding costs. They do this by hardcoding only those variables which will not change values very often and which, when they do, will be easy and inexpensive to change. Variables like `cust-gift-pct-rate`.

But suppose that the discount offered to customers was computed by a different program which took a percentage associated with each customer, calculated total sales for the quarter, and then computed the gift discount. In this case, it would be absurd to hardcode these discount percentage values in the program that calculates the discounts, because that would mean hardcoding an internal table that had one row for each customer and two columns – `cust-nbr` and `cust-gift-pct-rate`.

Hardcoding is so obviously wrong in this case that it is unlikely that it would ever occur to an experienced developer to do it. But that doesn’t mean that there isn’t an important principle at work. That principle is shown in Figure A1-4.

Early-bind only what changes infrequently, and can be

changed at little cost.

Figure A1-4. Principle: The Early-Binding Principle.

The reason behind this principle is simple. It is that the earlier the binding, the more expensive it is to change.

So far, we have thought of binding in terms of binding values to variables. Before we go on, we need to recognize that this is only a special case of the more general issue of binding.

Binding Semantics to Syntax.

Binding is more than what happens when variables are assigned a value. Binding, in its most general sense, is the assignment of *content* to *form*, of *semantics* to *syntax*.² In the example just considered, the values were the content/semantics, while the variables (with their names and data types, and in their specific sequence) were the form/syntax. The rows scrolling through the viewer were the semantics. The “viewer” itself was the syntax.³ As each new row popped up, the viewer changed its content. Semantics is content; syntax is structure.

If there are other cases of semantics, syntax and binding which are relevant to our work as data modelers, what are they? Are there early and late binding issues that affect data modelers, and not just programmers? Yes, there are.

There are two binding time issues that directly affect data modelers. They are:

² These different formulations of binding exist in decreasing order of generality. Binding, in the most general sense, is *the assignment of content to form*. Since the kind of content we in IT are interested in is structured information, we are familiar with a more specific way of thinking about binding, which is as *the assignment of information to the structures that contain it*. But this is just *the assignment of semantics to syntax*, in the specific form of database structures and processes. Finally, from the programmer/developer’s point of view, binding is *the assignment of values to variables*.

³ In my graphical convention for illustrating result sets, the top two rows provide the syntax for the viewer, which is the third row.

1. Business data used as primary and foreign keys.
2. Associative tables and embedded foreign keys as two different syntactical forms that express relationships.

In both cases, relational theory and relational DBMSs *early bind* semantics to syntax. And in both cases, this has caused an almost unimaginably great amount of unnecessary cost in the maintenance of information systems and their databases.

But before I explain how this is so, and what we can do about it, I need to warn you that the situation is somewhat like that of the emperor and his clothes. For it is generally accepted, among both data modelers and academic computer scientists, that one of the strengths of relational theory is that it does *not* early bind semantics to syntax. The point is seldom couched in this language, however. Instead, practitioners and computer scientists talk about *logical/physical data independence*.

And let me add here that this reference to a fairy tale by Hans Christian Andersen intends no disrespect to Dr. Codd. The mathematicization of data structures (as relations), permitting the manipulation of data by a well-defined formal language (predicate logic plus a few other operators, dressed up as SQL), was a Copernican Revolution in the management of data – and will prove itself, over time, to be just as significant, and just as enduring, as that 16th century revolution.

But as for “logical data independence”, we are dealing with an *unfinished* revolution. As significant an advance as relational theory was, in this respect, over its predecessors, it did not go nearly far enough. That is the principal theme of Part II of this book.

Appendix 2. A Refinement of the Rule That All Tables Must Contain Business Keys.

Thus far, I have made a strong case for using system-generated surrogate keys in all tables, for also providing a business key for every table, but also for keeping the two completely distinct. So it may come as a surprise that in Figure 38 (for the first time), I have tables which do not have business keys. So why do some have them and others not?

Directly and Indirectly Instanciable Tables.

To begin with, notice that these types form a hierarchy, from the root node Thing, to its immediate subtypes Group and Person, and finally to the two subtypes of Group and the two subtypes of Person. Those last-mentioned four tables are called “leaf-node” tables, because they are subtype but not supertype tables. The root node is a supertype but not a subtype table, and all other nodes (often called “branch” nodes) are both.

The example in Figure 38 suggests that all leaf-node tables have business keys, and this is correct. But why do some supertype tables (which are all and only those tables which are not leaf-node tables) have business keys while others do not?

To understand when to properly use and not use business keys in supertype tables, we must distinguish two kinds of supertype tables. This distinction is not made in data model diagrams, nor is it supported by relational DBMSs, nor is it a distinction made by relational theory itself. It is an example of semantics that are important, but that are not supported by relational theory or products.

Consider the Person table in Figure 38. Its two subtypes are Employee and Customer. One represents a person who produces goods or services while working for our company, and the other a person who consumes goods or services, as a customer of our company. But let us suppose that our company also receives data on employee dependents. Now although it would be unusual not to have a Dependents table as a third subtype of Person, it's both possible and quite legitimate. And, for the sake of example, that's what we

have here. But with no Dependents table, where are the rows for dependents stored?

In this case, they are stored in the Person table, along with rows for employees and customers. We may presume, that among the “other-data” shown for Person, there is a “subtype discriminator” with three values: “cust”, “emp” and “dpndt”. Every row in the Person table has one of those three values. If the discriminator is “cust” or “emp”, we know that additional information about that person – additional attributes and/or relationships – will be found in the indicated subtype table.

But the interesting case is the dependents. They have no subtype table, and so all the information we have on them comes from the Person table itself (and from its supertype, Thing). This makes the Person table a special kind of supertype table. It is a table some of whose rows have corresponding subtype rows, and others of whose rows do not. Because of the latter situation, we say that the Person table can be *directly instantiated*.

This situation arises because the supertype table does not contain an exhaustive (complete) set of subtype tables. In fact, a supertype table can be directly instantiated if and only if it does not contain an exhaustive set of subtypes.

Another important property of supertypes and subtypes that neither relational products nor relational theory support is whether subtypes are mutually exclusive or not. For example, in the model shown in Figure 38, we should assume that a person can be both an employee and a customer, but we can assume that a group cannot be both a household and an organization (since the latter term is obviously a shorthand for something like “business organization”). In fact, I have encountered many cases in which it would be helpful to declaratively express, not just whether a

set of relationships for a table were inclusive or exclusive, but rather what boolean combinations of relationships were exclusive or inclusive.

So both “exhaustive” and “exclusive” refer to properties of the supertype/subtype relationship that are not supported by relational theory or products. How, then, do these properties get enforced? What guarantees that a person can be both an employee and a customer, but a group must be either an organization or a household and never both? By the same token, what guarantees that we can add rows to the Person table for which there is no corresponding subtype, while that is not the case with the Group table?

The answer, of course, is: code. Most likely, this code will be written in SQL and executed as a stored procedure, a pre-insert trigger. So what’s wrong with that?

There is nothing wrong with it, except that code like this is being written over and over again, both within a single company and across companies. It would be far more cost-effective for DBMS vendors to write it once, and then provide a declarative interface (a new SQL DDL clause) so that DBAs could specify it rather than code it.

Figure A2-1. Aside: Two Properties of Type Hierarchies That Relational Theory Ignores.

We can now formulate the fully-explicit rule about business keys in tables.

**If a table is directly instanciable,
it *must* have a business key.**

Otherwise, it *cannot* have a business key.

Figure A2-2. Principle: Business Keys Belong in and only in Directly Instanciable Tables.

All leaf-node tables are directly instanciable. That's what it means to be a leaf-node table. And, as we have just seen, some supertype tables are also directly instanciable.⁴

But if you are satisfied with this explanation, I suggest that you shouldn't be. You should be looking for the deep patterns, in this case for the reason behind the rule. Why do business keys go with and only with directly instanciable tables? In fact, since it is business users who create and use business keys, why isn't our rule "Tables have business keys if and only if business users say they do."?

Of course, it is indeed business users who ultimately define the business keys for every table that has them. They define them when the data modeler asks them "What should we use as the primary key of this table?" (or, better, "What uniquely identifies each instance of this type of thing?"). For unless surrogates are used as primary keys, business keys *are* the primary keys of tables.

A typical conversational snippet might be:

Susan (the data modeler): "Mike, what uniquely identifies a line item on an invoice?"

⁴ The word "instanciable" is not in any dictionary I have consulted. But it is very useful. Consider it a cognate of "instantiate". For example, "Table X is directly instanciable" means exactly the same thing as "Table X can be directly instantiated".

Mike (the Accounts Receivable manager): “Well, it’s the line number, of course.”

Susan: “No, I mean what would distinguish a line on an invoice from any invoice line, including lines on other invoices. A lot of invoices have a line 3, for example, so what distinguishes all those line 3s from one another?”

Mike: “OK. I see what you mean. It’s the combination of the invoice number and the line number.”

That’s the business key – “the combination of the invoice number and the line number”. Susan may choose to make it the primary key of her Invoice Line table. Of course, the argument of Part II, Chapter 1 has been that she shouldn’t, but that’s not what we’re interested in right now. What we want to know is why an Invoice Line Item table, for example, has a business key whereas a Group table does not.

Remember that we started with the rule that business keys belong in and only in directly instanciable tables. By saying that we can directly create instances of these tables, we mean that it is the existence (and recognition) of something “out there” in the real world that causes us to create an instance of that table, i.e. to insert a row into the table. In this case, obviously, it’s the existence of real line items on real invoices.

Semantics and Ontology: the Referential Component of Meaning.

Both line items and invoices are things in the real world, things that can be distinguished from one another, and things that our company is interested in. Because we are interested in them, we want to represent them with data in a database – in this case with rows in an Invoice Line Item table. One line item, one row in the table.

But what distinguishes one row from another? It is the data which corresponds to what distinguishes one line item from another. The former is a question about data, a semantic question. The latter is a question about what’s “out there”, an ontological question.

So invoice and the line number on the invoice are what distinguish invoice line items from one another. Assuming that we already know that invoices themselves are distinguished by a business-assigned invoice number, we now know that rows in the Invoice Line Item table are distinguished by the combination of invoice number and line number.

Most data modelers are familiar with documents that have a header and detail line structure, and so this example of invoices and invoice lines will seem very straightforward to them. But we should not let a familiar surface pattern obscure the deep pattern underneath. What this example illustrates is *the way that data acquires meaning, and how ontology and semantics are related*. If we can see beyond the example, to this deep pattern underneath it, we can reach a conceptual *paradigm* (in Thomas Kuhn’s sense) the internalization of which will help us better model anything we are asked to model, and especially to better model unusual and puzzling cases – the ones at the opposite end of the spectrum from the “straightforward” invoices and line items types of cases.⁵ So let’s try to see the patterns underneath our invoices and invoice lines.

There are distinct things in the world. Call them “individuals”. There are also types of things. Call them “types”. The third line on invoice 44395 is an individual (as is the invoice itself). This is a fact about the world. Given that we are interested in individuals of this type, we need some way to distinguish them. This is a fact about our knowledge of the world.

If our databases are to reflect the world we are interested in, they need to be “isomorphic” with that world. The way we do that, in databases, is to make one table for each *type* we are interested in, and one row for each *individual* of that type that we are interested in. Being individuals, i.e. distinguishable

⁵ I apologize for the vagueness of the point I am trying to make in this sentence. It is difficult to make clearly, but I think it is so important that it’s worth the effort, however imperfect the results. To move forward in acquiring this paradigm, my recommendation would be to read about Kuhn’s concept of paradigm, and best of all to read *The Structure of Scientific Revolutions* itself. Then, as you continue your work of data modeling, try to use the concepts of semantics and ontology as a “lens” through which to view your work. Just as a lens focuses a blurry image and makes it clear, a good conceptual paradigm focuses a wide range of situations encountered, and makes them clear. From personal experience, I can tell you that semantics and ontology – albeit more in the classical than the comp sci sense – have been that lens for me.

things, there is some way to distinguish them. Wanting to know about those individuals, we identify what it is that makes them distinguishable to us.

In order to keep our databases isomorphic with the world and with our knowledge of it, type-level and individual-level isomorphism is not enough. We must also have an isomorphism between what (as we know them) distinguishes individuals from one another, and something in the data that corresponds to it. That is the business key. Business keys are isomorphic with what (as we know them) distinguishes one individual in the world from any other individual.

Tables which are not directly instanciable have a derivative kind of existence. Their rows do not directly correspond to individuals in the world. So tables which are not directly instanciable are best thought of as generalizations from tables that are, not as tables whose instances are concepts, or types, or universals.⁶

Tables, as I said, correspond to types. But types can be specific instances of more general types; they can have a supertype. In this way, type hierarchies are formed. But why would we be interested in any types or hierarchies of types which did not eventually end up being used to categorize individuals, specific things that exist in the world? I can't think of any reason.

So we have learned two things. First, that we need some way to distinguish individuals from one another. Whatever it is, it is what the business key for rows representing individuals of that type corresponds to.

Second, that for tables which have instances not directly, but only through their subtypes, what distinguishes each of their rows from one another is their direct or indirect link to a row in a directly instanciable table, a row which has a business key which corresponds to features of the corresponding individual that make it distinct within its type.

⁶ As described in Part 1, this is a nominalistic position on the issue of particulars and universals. A realistic position would say that non-directly-instanciable type tables are best thought of as tables whose instances are universals that "apply to" (another difficult concept) the individuals that "embody" them. For example, that red is a color that red things embody, and also is an instance of the universal color.

This is the answer to the original question that we have been seeking, because it is the answer that *empowers* you, as a data modeler. “Tables have business keys if users say so, and otherwise not” is true enough. But it provides no insight. “Tables have business keys when their rows correspond to distinguishable individuals in the real world” is not only true. It provides insight. With this insight, if a business user tells you “Well, there really is no unique identifier for this table”, and yet the table is directly instanciable, you can push back. You can say “There’s got to be one.” And sure enough, there will be one. The misunderstanding, for that is what it was, might have been, for example, that the business user was confused by your term “unique identifier”.

In this appendix, more than in many other places in this book, we have been *doing* ontology; we have been *doing* semantics. We have talked about types and individuals. We have made an important distinction between two “types of types”, the directly instanciable kind and the other kind. So much is ontology.

We have also talked about three kinds of isomorphism between the world and our data – tables corresponding to types, rows corresponding to individuals, business keys corresponding to what it is about individuals that we use to distinguish them. So much is semantics, for it is this isomorphism, and nothing else, that makes our data *meaningful*.