Tools and techniques - we get into the weeds here

How computers work

It's time to dive into the *guts* of computers and talk just a little about how they actually work. Everyone, interested or not, needs to follow this link and read up on how computers work, which will illustrate the fact that computers are systems in the exact same sense that we have been describing here. Note especially the difference between hardware and software. The distinction between system software (operating systems such as Widows, Android and Apple OS, etc.) and application software (things that you actually use on the computer such as Excel, Word or a browser) is also important. Finally, this short link provides a gentle introduction to *the digital divide*, to which we will return later. [Move this figure to that discussion.]

We humans are most familiar with the Base 10 number system because we have 10 fingers and 10 toes. We refer to this as the *decimal system*. In this system, the columns of numbers correspond to ones, tens, hundreds, thousands, and so on. And it's also *digital* because it's based on our digits (fingers). Fun eh?

Computers, however, are *binary* creatures, run by 0s and 1s. They understand only black or white, up or down, in or out, off or on -- like a collection of light switches. They operate under the *binary* (or *Boolean* see below) system, which operates on two states. In a computer, a vast array of *transistors* capable of holding a tiny electrical charge are *loaded* with a meaningful and constantly changing pattern of such charges. The pattern represents the *real world* that we are modelling. The contents of these transistors are referred to as *bits*, short for *binary digits*. In this system, the columns represent powers of two (two, four, eight, sixteen, thirty two, sixty four, and so on). So the decimal number 55 becomes 110111 in binary, which is 32+16+4+2+1. With eight bits (also called a byte), you can store any decimal number from 0 to 255 (00000000 to 11111111 in binary). It's no coincidence that the RGB system (which we will tackle in Chapter 2 (take me there) represents colour in a range of 0-255. It should come together for you.

Interested?

Wikipedia offers this: "Another mathematician and philosopher by the name of George Boole published a paper in 1847 called 'The Mathematical Analysis of Logic' that describes an algebraic system of logic, now known as Boolean algebra. Boole's system was based on binary, a yes-no, on-off approach that consisted the three most basic operations: AND, OR, and NOT. This system was not put into use until a graduate student from Massachusetts Institute of Technology by the name of Claude Shannon (yes, the one-and-the-same Claude Shannon whom we will discuss when we consider *entropy* in Chapter 2 (take me there)) noticed that the Boolean algebra he learned was similar to an electric circuit. Shannon wrote his thesis in 1937, which implemented his findings. Shannon's thesis became a starting point for the use of the binary code in practical applications such as computers, electric circuits, and more."

Interested in Binary Code?

Interested in Boolean?

An exhaustive list of devices

Binary [Binary: So simple a computer can do it]

ASCII codes

Check out Figure NF below, representing the encoding of a simple series of characters.

Let's unpack Figure NF above. ASCII stands for *American Standard Code for Information Interchange*, and was developed from telegraph codes to represent some basic English-language characters and certain *control codes* (non-printable codes which did things like ring a bell on a teletype machine to get the operator's attention or to signal a certain state, such as 'end of message transmission'). ASCII is a 7-bit encoding scheme (note the first bit of each character in Figure NF is never on) and can encode 128 characters and codes. (Remember binary? Using 7 bits, we can encode 64+32+16+8+4+2+1 = 127 and also remember that the range is 0 to 127 and this equals 128 states.) Expanding to an 8-bit standard adds 128 states, allowing 256 things (0-255) to be represented. This 8-bit schema is the basis for the currently-standard (on the web) UTF-8 encoding scheme (which is *backwards compatible* with ASCII-7 or USASCII as some call it).

Figure NF shows the ASCII encoding of the characters 'BUSI 2400'. The characters are encoded in a vertical fashion from bottom to top. Also included are the binary values for each bit position. (Many of you experienced your eyes glazing over as soon as you saw this thing. But bear with

me. I'll be quick and gentle as we make sense of this stuff. And if no one has ever used that 'quick and gentle' line on you before, please be warned. It's neither.)

The binary values make their way up, doubling with each bit position, from 1 to 128. This give us an 8-bit scheme. If we sum the decimal equivalent represented by the binary positions for the letter 'B', for example, we see that the '2' bit and the '64' bit are on (circle darkened), giving us 64+2=66. Thus the decimal equivalent of this binary pattern is 66. Taking a look at the table of ASCII values on the right (only printable characters are shown here), and locating decimal value 66 (it's highlighted so you can find it) we see that our bit pattern translates to decimal 66, representing the letter 'B'. (Note that a lowercase 'b' is decimal 98, an altogether different character and, to a computer, as different as the number 6 and the % character.) Summing the locations for the letter 'U' gives 85. Same process: look for the decimal 85 in the ASCII chart and note that it's the letter 'U'. You can continue the process with the remainder of the BUSI 2400 representation and see that it makes sense. Look at the decimal value for the character 'S' and verify that the binary sum of positions equals 83.

Note especially that the ASCII for the space between BUSI and 2400 (here denoted by a '␣') is a real printable control code character, having decimal value 32. A space is as real to a computer as a letter or number and it's not at all the equivalent of a 'null' (nothing; absent;) character which has the ASCII code of 0000000 and the decimal value of 0 (zero)).

But that's going too far. Let's get back to what's important here.

Note again that the first (topmost or 128) bit is never on in this 8-bit representation of a 7-bit standard. Seven bits are sufficient to represent a range of control codes, all the letters of the alphabet in both *UPPERCASE* and *lowercase* format, the numbers from 0-9, along with some common typographic glyphs Interested in Glyphs? in the English language (such as swearing, often denoted by '!@#$%^&*) and the mathematical operators such as +-/*, etc. Other implementations of ASCII provide mappings to the Euro glyph (€) or the British Pound Sterling (£) for example.

Interested in ASCII?

Fonts

On top of ASCII is layered a mapping onto a specific font. One the specific character is known, computers can draw that character according the style characteristics of particular font sets. This online publishing service doesn't allow one to specify particular fonts in the text. I had to choose one font for the whole book. Below is an image of this book when it was in draft form in an MS Word doc with several fonts shown. Important to note is that the ASCII codes for each letter are identical. Only the mapping of those codes to the particular font style are different, and that happens on a different layer altogether.

Figure XYZ. Fonts…

The phrase "The quick brown fox jumps over the lazy dog." Is often seen in connection with the discussion of fonts as it contains all 26 letters of the English alphabet. Other aspects of fonts include whether the individual characters have little *feet* on them (called *serifs* and rhymes with *sheriffs* - fonts without feet are called *sans serif sans* being French for *without* - there's no rocket surgery here) whether they are bold, italics, underlined, strikethrough or otherwise *decorated*. Some fonts are created with only UPPERCASE. In addition, fonts have a *size* attribute, normally represented in *points*. Some fonts are *monospaced* meaning that all characters occupy the exact same *real estate* on the screen or in print, regardless of the difference in the actual size of the character. Note the difference in width between a lowercase L "l" and an UPPERCASE "X" for example. The OCR font above is monospace. Finally, some fonts are considered *decorative* meaning that they are for titles or limited use for emphasis on posters or adverts and not meant for reading reams of text (as you are doing here). Algerian, Brush Script and Old English above are decorative.

It is interesting to note that font *decoration* characteristics such as **bold** and *italics* require an entirely separate set of fonts for each. So a font such as Baskerville above would have one set of glyphs for the normal set of letters and characters, a different set with the same style characteristics but drawn heavier for the bold font, one for the slanted italics set of characters, one for underlined (the underline is part of the font and not a separate element of typesetting), etc. It's a major undertaking to create a font. Maybe you'd like to try your hand?

Interested in creating a font?

All this is beyond our scope here but in general you might be Interested in fonts?

Interface design and programming

Let's set the stage with a quick backgrounder.

Figure HH. Terminology 101

Figure HH shows a very high level look at some basic architecture. Beginning on the far left, we see the *frontend*. A frontend is anything we use to interact with a system. This can be an iPod, an Android tablet, a Blackberry, a laptop, a desktop or a dedicated terminal *frontending* for a huge mainframe computer. The frontend acts as both an input device for us to *talk to* the system, as well as the system's voice when it communicated output back to us. Input, process, output, feedback. It's all about the system.

The next need to be clear on from *where* the system operates. Is the program code that drives the processing located on the frontend device itself, or is that code resident on a remote device somewhere in the cloud? *The cloud* is huge these days. Remote processing is all the rage as devices (the frontend displays we use) get smaller and smaller and less and less *willing* to perform their own work locally. There are compelling reasons to compute in the cloud (see our discussion in the Systems Development chapter). So it's not really that our handheld devices are not *capable*. Not at all. Here's a quote attributed to Dr. Michio Kaku, who researches and reports on future trends in computing: "Today, your cell phone has more computer power than all of NASA back in 1969, when it placed two astronauts on the moon." The review authors of the go on to state that it "Seems hard to believe, we know, but it is actually true – a hand-held apparatus on which we fling birds at pigs has greater computational capabilities than the arsenal of machines used for guiding crafts through outer space some 45 years ago." So it's rather that cloud computing is much more efficient.

Interested?

So processing can be done either locally or remotely. But to users, it really doesn't matter much. We ask for something, it gets processed, and the results get returned to our frontend. When we browse the web, for example, our browser (Chrome or Safari or Internet Explorer) simply renders the information sent from the web server in the cloud. No local computing is done. But there are some web applications that use *client-side scripting* and do some processing locally. Some applications combine server-side with client-side scripting to give a richer environment. Dropdown menus on websites, for example, are most often animated by client-side script, while the heavy lifting is done on remote, often specialised, servers.

So processing can be effected either locally or remotely, and *permanent* storage can equally be either local or remote. Temporary (volatile) memory (sometimes referred to as *Random Access Memory* or RAM) on the other hand, is always local to the machine doing the processing. So if the code is executed on your phone, for example,. then it uses RAM on your phone. If it's crunching in the cloud (at some remote server farm, for example) then that's where the RAM gets used. RAM is simply local memory that is used in working on a computing problem and, when execution ends, RAM gets overwritten by the data for the next problem in code. This is why it's called *volatile*. It goes away. (The accompanying icon I chose in Figure HHH for RAM is a paper shredder. Get it...?)

We end up finally with the *backend*, which not only stores the details of transactions we perform, but also the parameters that form the context of our interactions, such as preferences (language, colour schemes), credit card info, shipping addresses etc. Normally a backend is synonymous with a database. We consider databases later in this chapter. The backend performs many crucial tasks in computing, but the most important of which is storing all necessary transaction detail so that organisations can account for themselves. No firm of larger than one person (and precious few of them in the developed world) use a pen and paper to log transactions. But you never know... back to the land could become back to the pen.

And now a little something about interfaces, including our friend the frontend. First some levity.

[Source: http://www.gocomics.com/chucklebros]

There are two *interfaces* to which are alluded in the cartoon above (which is brilliant from an entrepreneurial perspective). An interface is the place where two systems with different *immediate* goals interact and fulfil the goals of a *higher-order* system. It's where systems talk to each other and exchange stuff.

So an *interface* is a *meeting place*. In the case of the above cartoon,it's *people meets beer* and *people meets... well, urinal*. It's commerce. To sell beer efficiently at this client-server interface, there needs to be two systems simultaneously available and functioning. Input and output. Even if you've never been to a bar, surely you've been to a McDonald's. Imaging those taps in the cartoon full of pop or tea or coffee or even water. There would need to be a way to deal with the output in order make room for new input. *Ergo* there are two interfaces here. The interface to the input and the interface to the output. It is to the *design* of these spaces where humans meet machines, the *user interface*, that we now turn our attention.

Interested?

I fancy myself an interface designer. If there was anything about creating systems that I enjoyed (other than a really tight algorithm that churned out accurate results in the blink of an eye), it was the art of the beautiful interface. And beauty, in an interface, has less to do with nice fonts and complementary colour schemes (although that's important too, don't get me wrong) than to the raw simplicity that allows people to just attack and use it and feel as if they were smart because it was so obvious to them how to make it sing. They could make music with the interface. It was a *Stradivarius*... something that in the right hands could create beauty itself. OK that's just a bit overstated, but the feeling is genuine. It's nice to see

your interface being used by people who just *knew what to do with it* and could use it in the way it was intended and it just worked seamlessly. It's so nice watching people feel smart. It's a bit like designing a piece of clothing and seeing people wearing it in the wild. So satisfying.

Interface design and algorithm design go hand in hand. From the interface perspective, people need to be able to address it and instinctively know what to do with it. It has to be what we call *intuitive*. It must suggest to the user how it can be used. Much of this usability, which the Business Dictionary (on June 20, 2015) defined as: "Ease, speed, and intuitiveness in operating or using a device, service, or facility. Usability arises from a combination of well thought-out architectural and design factors, and translates into user's ability to successfully perform tasks and solve problems with customary effort." Pithy, and right on the money. So how to get there?

Much of what we consider to be the interface in a piece of software has been standardised by the operating systems we have grown accustomed to using. The various widgets and icons we use have come to accepted ways of communicating choice and receiving output. The general term for our ability to understand what these tools is encapsulated in the term *affordance*, defined by dictionary.com (on June 20, 2015) as: "A visual clue to the function of an object."

Take a button as an example. A everyday, garden-variety interface button, as below.

Figure POL. An interface button

As savvy and experienced computer users, we know what to do with buttons. When we click them with a mouse of tap them with our finger we effect some change. Our *input* by clicking will set some *process* in motion, resulting in some *output*. It all goes back to the simple system we've talked about since Chapter 1. But what is it about the button that cues us to what to do with it? Why does a button *afford* clicking?

First and foremost, it affords clicking because we have learned that objects that look like buttons operate by clicking. Elevator buttons. Microwaves. Telephones. Keyboards. Radios (at least in the old days anyway). So *graphical user interface* (GUI) designers leveraged everyday things that people already knew how to operate and built them into the human-computer interface. The early designers of what we now know as Windows and Mac interfaces pioneered these techniques (and the use of a mouse as an input device) in the 1970s at Xerox (the photocopier people) Palo Alto Research Center (or PARC). Xerox PARC is legendary in computer geek history. They are responsible for some incredible, ground-breaking advances in computing including, but not limited to, laser printing (obvious when you think about their signature copier machines), ubiquitous computing (computing everywhere with multiple devices talking to multiple devices through middleware - this is the origin of the *internet of things* which we will discuss at length elsewhere in this text), Ethernet (the wired foundation of local area networks), object-oriented programming (which also makes sense in terms of their pioneering work in GUI), the concept of the computer desktop and to top it all off, no less than the origin of the personal computer that we all know and love.

Interested?

So the nice folks at PARC used everyday objects and then created what we refer to as *idioms* for other functionality. Things that have real-world correspondence like *check boxes* for multiple item selection, *radio buttons* for single selection (like an old-time radio where only one button could be depressed at a time, the already selected one popping back out when you pressed a new one to select a different channel), an *image container* like a picture frame and a matched set of *scroll bars* (one horizontal and one vertical) and a brand-new idiom, the *dropdown* menu. Here are some examples:

Button with visual cue	Check boxes for multiple selection
	
Series of combo boxes	Image container and scroll bar
	
Radio buttons for single selection	Label and text box for data input
	

But what really got the ball rolling was the idea of the desktop, and virtual windows onto that desktop. Again, Xerox PARC were the innovators who brought these idioms to the world. (Many Apple devotees erroneously believe that it was Apple which 'invented' Windows and the GUI and icons and mouse input etc. Not so. Everyone followed PARC's lead. They were the first.)

To go with these idioms and metaphors that allow us to instinctively understand how to interface with a machine (my 93 yer old mother is a whiz with an iPad and the envy of her nursing home residents and the much younger staff), designers have used all manner of both traditional and new research and best practice to build intuitive interfaces. A large part of the miniaturisation we have experienced is the result of clever work with

icons, which have largely been standardised across vendors and platforms and have allowed for complicated symbols to be communicated in 15 X 15 pixel format. Some examples are below. Give yourself a little test. If you can't associate an icon with an activity or command or an idea, punish yourself with two hours of aimlessly bashing your favourite device:

Combined with traditional knowledge about visual design (an enormously interesting and exciting field) such as Gestalt (for closure and similarity and proximity Interested? and cognitive psychology (another course you should take at least one of in your career), rich and effective interfaces have evolved into use and are accepted means of communicating with machines. In this I am reminded of French aviator and author *Antoine de Saint-Exupéry* (1900-1944 – author of The Little Prince and only 44 when he died), who characterised engineering elegance by writing: "A designer knows when he has achieved perfection not when there is nothing left to add, but when there is nothing left to take away." So like any other piece of engineering, it's not about a *tour de force* but rather about simple elegance. Edward Tufte (the godfather of modern information visualisation) included the following line in his dog and pony show (which I attended in 2010 in San Francisco) "No matter how beautiful your interface, it would be better if there was less of it." Wise words. Simple elegance is the designer's goal.

Here's something from the Nielson/Norman Group, the *de facto* experts in everything UI lately. It's a bit long for an [Interested?] but it's a really good read if you are interested in minimalist interface design. And you should be as it translates very well to creating compelling visual images for presentations.

The desktop metaphor and the idea of iconic representation of tasks and the manipulation of things in the interface (such as files and folders and removable drives such as a USB) would not be workable in the absence of another of PARC's great initiatives: *Object-oriented Programming* (OOP). OOP describes *objects* (everything is an object including command buttons, file folders, printer icons, and all the way down to a single cell in a spreadsheet application) which can have *properties* (characteristics that describe them - for example a simple OK button has a width, height, position in the window in which it appears, a caption (what it says - such as 'OK already!' or 'Print') and a background colour, font, etc.), *events* (things that can happen to it over its lifespan - for a button, the most important is the *click* event) and finally and most importantly, *methods* or things it can accomplish given the occurrence of an event. Designers design the interface, set the properties and scope the events to which the object should respond. Programmers write the code that executes when an event occurs to the object. This genre of programming is referred to as object-oriented, event-driven programming. Interested in OOP?

Figure RS. Object-oriented, event-driven programming example

Here we have a little window with a purpose: to display and sort the names of TAs for a course. At the very top, we find a window label indicating what we are looking at. "My TAs". How nice (and they were all great people who worked very hard for me). The blue strip at the top also includes other *objects* owned by the window, such as the *minimise, maximise* and *close* functions on the right of the strip. Below that, the meat of the interface - the actual list of TAs (not yet sorted at all, just dumped into the list box, which is an object that holds lists of things). Below that, we see two command buttons, one to sort the list in ascending order (from A to Z) and one to sort in the opposite direction. Below that, we have two buttons to deal with disposal of the window; one to cancel the operation (likely the list would be returned to it's sort order on entry to the window, but that would have to be coded) and one to close the window and return control to the calling window (the greyed-out window to the right with the little "Show TAs" button that actually asked for this TA window to be shown).

Let's further unpack this figure. Each of the objects, including the window itself, has Properties that can be set by either the interface designer or the coder. These properties can be set once and left alone, or can be controlled by code. Objects can be made to appear and disappear, for example, depending on the context of the application. Each object here has properties such as 'Top' indicating the location of its horizontal position (or top, left corner) in the window container. Analogous properties include 'Left', indicating its vertical position (how far left in the window), its width and height and caption (what the button's label says). Note the 'Cancel' button has a red caption (and is a different size) to set it off from the other exit button.

There are plenty of design principles on display here. The first thing your eye would catch on displaying this window is the reasonably discrete 'My TAs' window caption at the top, serving as a reminder of what the window displays. The actual list itself is the most prominent object on the interface (through the use of a coloured background) as this is the real purpose of the window, and our eye is drawn there next. Once we are satisfied that we are looking at what we intended, we next see the actions or work we might want to accomplish with our list. Finally, we see the disposal actions, with the only *emergency* action indicated in red, and the most likely disposal action *Close* at the very bottom, right of the window. This is where people look for actions when they are done.

There are also other more subtle things going on. Gestalt things (remember Gestalt?). Specifically, grouping through proximity, size and colour. Note the space between the two sort buttons is smaller than the space between the list box and the first command button. Note that the space between the list box and the first command button is the same as between the bottom sort button and the top of the disposal buttons. So we have three functional groupings here, defined by proximity, to represent the three function types on this interface. It's a subtle confirmation (and our brains are aware of this whether we know it or not) that there are three things going on here. Finally, size (and colour) is used to set off the

'Emergency' possibility. Our eye is drawn to the differences between this and the other objects in the window, and that's a good thing. We want our user to have a good exit strategy. Note that *Emergency Exit* signs in public buildings are always lit and always red. We borrow that motif here on the interface.

While this example barely scratches the outer shell of the discipline of design, it does illustrate some things of which users are not necessarily consciously aware, but which are enforced for our (user's) good. When an interface is designed with simplicity in mind, and *from the perspective of the user* and not that of the designer, we, the users, use it and feel smart about it. That's an important goal of design. Make the user feel smart which, by the way, makes for a smart designer.

Let's finally look at the code, which is why we're *really* here. I have written the code in an odd, hybrid pseudo-code / real code format in an effort to make it at least a bit understandable to a non-programmer. If it's not, I'm sure you will let me know. But let's walk through the logic of the program flow here.

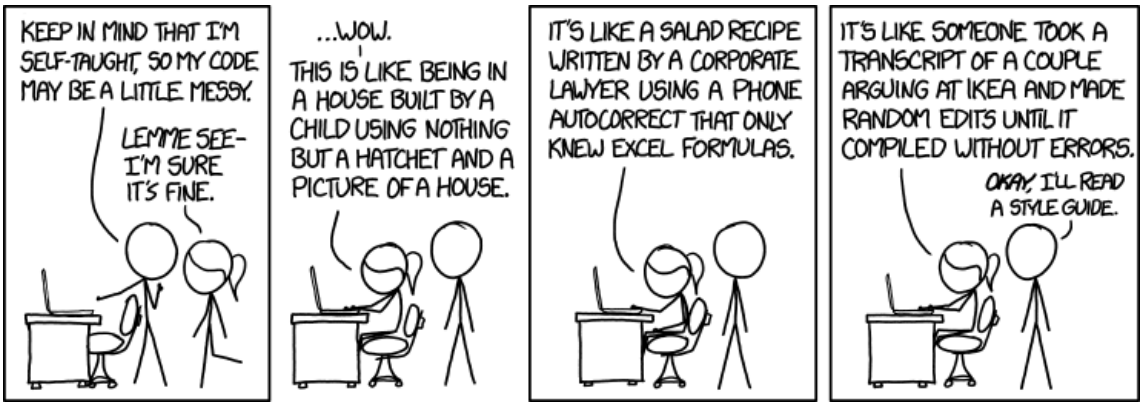
First, this window is *called* from another window, through the user clicking the "Show TAs" button on that interface. The TA window will then appear (pop up) over the calling window and will have *the focus* - meaning that all the user's actions will be directed to this window because it is *in focus*. We might argue that this is an overly simplistic example, and in fact the button on the calling interface should be "Manage TAs" and there should be functions on our TA interface to add, delete, edit, pay, schedule and all the things a professor would need to do with their TA objects over the life of delivering a course. And this is true. But let's just digest this simple example here, with a few simple functions, before being critical of the overall design.

So our TA window now has *the focus.* We can observe the list of TAs and exit (through either Cancel or Close), or we can choose to sort our list. Knowing that the sorting of the list in either ascending or descending order is a *requirement* of the system, the designer and coder (programmer) would work in tandem to ensure that this functionality exists in the system. The designer would have placed the buttons in the appropriate places on the interface, with the appropriate properties set, and then handed it over to the coder, whose job it is to write the code to do the work. Since the most obvious thing to do with a button, as a user, is to click it (buttons *afford* clicking after all), the coder would write the code to do the sorting in the *Click* event of the button. The interface just sits there being all *interfacey* until the user *does something* such as click a button. Those buttons have other events to which they can respond as well. For example, the 'MouseOver' event, which is fired when the user passes the mouse over the real estate occupied by the button. The programmer might stick some code in the 'MouseOver' event to pop up a little message such as "Click me!' when the mouse in over the button. Likely not, but you get the idea. Here, the most important thing is to react to the *Click* event of the button and do some work since that's what the user, who knows all about how to make an application sing, expects.. So in the space in the program (we are skipping over massive detail here by the way) reserved for reacting to the *Click event* of the *Sort ascending* button, we have the code in green in the Figure.

So let's take a deep breath and dive in. When the button is clicked, the code associated with the event is executed sequentially, from top to bottom. We first ask the machine to count the number of items (TAs) that are in our list box. We have nine. We then stick that number into a *variable* - an area in RAM (volatile memory) that can hold onto a value for us and to which we can later refer when we need to use it. So we've got a variable named 'z' which now holds the number 9. We are using a (simplified) Bubble Sort algorithm here to order the items in our list. We next launch into that algorithm and tell the machine to iterate (step) through each item in our list box and compare it with the item *following* it in our list to determine which is *greater*. This is effected implicitly by the operating system of the machine by examining the ASCII codes of each letter in the TA name beginning with the first letter of the last name. If you forget what ASCII codes are, please go back and review (take me there). So to begin, the machine would compare 'Jamieson' with 'Shapransky', starting by comparing 'J' (ASCII code 74) with 'S' (ASCII code 83). Since 74 is not greater than 83, these two items are already properly sorted and the algorithm moves on to compare 'Jamieson' with 'Platt'. Same deal. Next comes the comparison with 'Ahmed'. Oh oh. Trouble in Toyland! 'J' is greater than 'A' so we need to swap and move Ms. Ahmed up a notch (or demote Ms. Jamieson, whichever way you look at it, it's the same). The process continues until all items are sorted in ascending order, the sorted list is displayed again (which occurs automatically thanks to the operating system). The interface then just interfaces along, waiting for the user to do something else.

It is interesting to note that in order to change the algorithm from sorting in an ascending order to one that will sort in a descending order, all that's necessary is to change one *relational operator*, the '>' sign. Copy the code, switch to a '<' sign *et volia*! Done. Ahhh were all programming so simple... Note that appropriate code would need to be written in each of the other button's *Click* events as well in order for them to function as expected.

Time for an XKCD.



And that's the thousand kilometre high view of interface design, objects, programming and systems. There's an incredible world of exciting work out there in design and programming. Don't discount it until you consider how satisfying it can be to create systems that make people feel smart. I still love getting my hands dirty in low-level design and coding. I can get lost for days in it.

Interested in User Experience Design?

How about User-centred Design?

Or simply User Experience?

Or the underlying aspects of Human-computer Interaction?

Database design

A *database* is a collection of related data, organized in a logical order and used in many everyday activities. They are called *databases* because they store *data* and not *information*. Examples include a dictionary, a catalogue, TV listings and even the directory in a building, listing all the names of the people, perhaps their title and/or function along with the floor they occupy and their office number. Dictionaries, like phone books and building directories, are typically organized alphabetically. These are all non-electronic databases.

Computerized databases are stored on electronic media either local to the user or in any of the various manners in which we store data remotely. They serve the same purpose as non-electronic databases. Examples include an MP3 music library, the contact list on your cell phone or in your email client.

Why use a database?

Databases allow large (even massive) scale data storage. They also reduce possible errors (because they allow rules to be enforced about how things are stored and what *type* of data can be stored in the base). And if properly designed, they reduce *storage redundancy*, or the necessity to store the same piece of data in multiple locations in the database. The software provides fast, easy and efficient retrieval, viewing, searching, sorting, adding, deleting and editing of the stored data. Databases are used by almost all organizations in this information age.

Why not just use Excel?

What's the difference between a spreadsheet and a database anyway? Aren't we just splitting hairs here? They both do the same things... right? Nope. Spreadsheets are optimised for a different purpose, and provide a simple flat file data storage system. They are optimised for doing calculations, for *visually* arranging data and providing *visualisations* (charts, graphs, etc.). Databases, on the other hand, are optimised for efficient and effective data storage and retrieval. We are not even given the courtesy of seeing exactly how the data are stored by the database. We see a representation of how the data are stored, but we needn't worry about those low-level implementation details. The *relational* functionality of databases is the key to reducing error and redundancy. We will return to this shortly.

What is MS Access?

Microsoft Access is perhaps the most prevalent Database Management System (DBMS) on the planet, owing to the ubiquity of Microsoft Office. Access (and other databases) allows users to:

- Easily create and manage a database
- View information stored in the DB
- Search/Sort/Filter the information
- Insert/Delete/Update new and existing information

- Allow for a full *CRUD* (Create, Read, Update ans Delete capability for all data in the database - more detail later (take me there)

Access is widely used by office and home users due to its user friendliness and the fact that it requires minimal technical skill. YOu can be up and working in a database in a few hours.

How does MS Access work?

Access is a collection of objects (which we discussed above in terms of OOP). These objects include:

- Tables
- Queries
- Forms
- Reports

Data is stored within table objects, and queries and forms are used to retrieve data from those tables.

Key term summary

- Database: is a collection of related data
- Object: is a table, query, form, etc. within a database
- Table: is a collection of related data organized in fields (columns) and records (rows) on a datasheet

A table in a database corresponds to an *entity*. An entity is a person, place, thing or event of interest. Entities become tables, and store only things that are of the same nature. So a *person entity* would store only attributes (characteristics) of people. You would find nothing in a *person entity* describing a movie or a school, for example. Choosing the entities you will store in your database for a particular system is an important early step in creating a defensible *data model*. So the first question you need to ask when creating a database is "What are the entities about which I will need to store information?" What are the *things* that form my problem space? Do I have people? Do I have locations? Am I tracking events? Are things like trucks or books or coffee cups involved? It's important to understand that a single database solution can have all these things in the same database, in different tables, if they interact with each other. For example a *people table* can store information about persons and those persons might attend a series of events and have parking spaces and own a bunch of pieces of art. If the problem space includes all these things and they interact with each other then a database can store and manage all of them. But let's not get too far ahead of ourselves here.

Tables are the basic structure in which data is stored within a database. They consist of a collection of records (one per instance of the entity - like one per *person* or one per *geological specimen* or per *real estate transaction*) and the records themselves are a collection of fields (which are characteristics of the entity and can vary from one instance of the entity to the next). So in a *person entity*, fields (characteristics) might be the person's name and age and perhaps email address or Twitter handle. So each record, in this case, would be unique because while people might have the same name and age, they cannot have the same email address or Twitter handle. This introduces an important concept - the Primary Key (PK) for a table.

Every table in a database needs a PK. A *Primary Key* is a unique identifier for each record. Databases need a way to be able to retrieve a unique record for each entity, otherwise, we would never know whose record we were looking at if there were 15 people named Jane Smith or Mohamed Mohamed in our database. You are already aware of several unique identifiers in your life... such as your student ID, driver's licence number, and if you are a citizen, a newcomer to or a temporary resident of Canada, then you have a SIN card with a unique 9-digit number. There are others I'm sure you know (Health Card for example). This is enforced so that there is never any doubt about to whom particular records refer. A database is no different. It must be able to unequivocally pinpoint the correct instance of an entity. Get the right person every time.

So, one of the fields in every table is dedicated to a Primary Key. We will come back to Primary Keys and its very attractive cousin, the *Foreign Key*, just below.

Figure STT. Tables (Datasheet View)

Terminology

Field is a column on a datasheet; it allows data of a particular type to be stored in records in a table. In Microsoft Access, a field has a Name, an associated data type and a Description. This data, associated with the fields in a table, are referred to as *metadata*, literally *data about data*. In Figure STT above, we see that the table named *tStudent* has four fields, *sCUID*, *sTerm*, *sYear* and *sSection.* You can see their data types and the description of the field's contents in the Figure. Fields are sometimes referred to as 'attributes', which is a synonym for *characteristics.* Note also the little gold *key* beside the sCUID field. This indicates that this field is the *Primary Key* or PK for the table. We will see why this is important later when we examine the *WHERE* clause in SQL statements.

Figure STP. Tables (Design View)

Record is a row on a datasheet and is a collection of values defined by fields. In our auction application, the four fields are shown here *populated* (data entered) for three records, corresponding to three students. The first record captures the information for a student with the CUID of '100999997' (in a field names sCUID - the 's' prepended indicating that it's in the Student table), a Term indicating 'Fall', in the Year 2015, and who is, finally,in Section 'A'. Two other students have been entered below. This is from a real, live, database. This is what it looks like.

Primary Keys (PKs) and Foreign Keys (FKs)

Now that we have a basic understanding of tables (entities), fields (attributes) and records, we can move on to the more sophisticated consideration of what makes *relational* database management systems so powerful. Let's imagine trying to implement the auction application in Excel, for example. We will just consider (model) the Student/Bid part of the application. Some background is necessary here.

The *auction application* is a tool that I use in my Intro to ICT class. It pits student against student in a real-time race to buy a required number of some commodity item in a restricted amount of time from a dynamic system that accepts *bids* in the form of a commitment to purchase. A *bid* in this game is simply a *sale*. In *real speak*, what might happen in a hypothetical game is that all students in the class are required to virtually purchase at least seven and at most 10 items (it matters not what the items are) in a five-minute period. There are not enough items in inventory for every student to purchase 10 items. The group of students (half the class, or 25% of the class, it's up to the instructor) with the lowest *average* purchase price for their purchased commodities is awarded bonus marks. The trick is that as the *velocity* of purchasing by the group increases, the higher the price rises. In order to fulfil the requirements of buying a minimum number of items at the *lowest* average price, it's necessary to buy when few others are buying, because the price is dropping as fewer and fewer are bidding. But the clock is ticking... anyway, it's a fun simulation and it's meant to stimulate thinking around disrupting how commodity items are sold in public venues (such as concerts or sporting events). That's enough detail for our purposes here.

Let's now consider how much data needs to be tracked in this simulation if our goal were *only* to award bonus marks. We'd need the student ID and the exact price at which the article was bought. And seriously, that's it. But if we wanted to control the *context* in which the bids happened and to be safe and secure and to be able to enforce the rules around the process so that if challenged, we could say with near certainty that Student X bought Y articles at this array of prices and that the competition was still open at the time of final purchase and that any particular *sale* occurred while the simulation was running and that there was no data entry error, and that the student was actually in the classroom and participating etc. then we'd need a lot more information than just those two pieces of data. This is an important aspect of systems development. Often systems are created in order to provide *accountability* for processes. Systems, with auditable rules and archived inputs and outputs, are seen to be impartial appraisers of process.

Imagine a spreadsheet storing the necessary data to allow for the distribution of bonus marks - marks which might mean the difference between a pass and a fail in the class. The stakes are high. So how to gather the data? One solution might be to insist that each student keep accurate records of their own transactions. There are *at least* two problems here: 1) Record keeping in the heat of competition might not be accurate, and, perhaps more importantly, doing so would almost certainly distract the student from their primary task - to buy stuff cheap. So that's not a great solution. We might also consider assigning a non-partisan, independent recorder to each student. Not feasible at all - who would it be? 2) Perhaps even more importantly, and here's where the audit issue comes into play, people are not always honest. Sorry to be blunt but after 35 or so years in the teaching game, I can state unequivocally that I don't expect *all* people to be objective assessors of their own performance, especially when something as critical as grades are concerned. Be honest with yourself. Would you trust *everyone* in the class to be capable of not only protecting their own interests, but yours as well? So we are well advised to create a system which can *prove* it is impartial because, well, it's a machine and if the rules are properly encoded, there's no reason to suspect that bias is creeping into the system. It might be *systematically* biased (see precision in Chapter 2) but the bias will be equally distributed across players and not in favour of, or detrimental to, any particular player. That's how we *level the playing field*. Machines are impartial, don't care about your previous record or your current trajectory, don't lie, cheat, fall in love or get a mad on, and can't be easily fooled on the rules. And they provide some security if properly implemented.

We often defer to systems when challenged on whether or not rules or procedure were properly followed. And that's not such a bad thing. In fact, it's the essence of audit; a set of articulated, transparent and defensible rules, accompanied by evidence that the rules were followed. In the case of the system under consideration here, it's not feasible to collect data in any other way than centrally, through an application and a database backend. (A *backend* is a system component that is not visible to a user but which performs vital services for the process piece and for the *frontend* - the part which the user sees.) So all the transactions in this auction system are recorded in a database. But how to accomplish this efficiently?

No surprise here but we need to create the *context* around the system in order for it to operate in a defensible way. We need all the rich soup of data that surrounds the simple system transactions (student bought something at this price) in order for the rules to be unequivocally interpreted and for the decision to be make with certainty. Thus we arrive at the data model in Figure EZ below.

Figure EZ. The auction system ERD (Entity Relationship Diagram)

* While it seems daunting at first glance, it's not really that bad. Each box is a table. Recall that tables store *entities* which are *things*. So we have a *Bonus* entity (my convention is to prepend a 't' to each table name in order to keep objects clearly identified), a *Student* entity, a *Bids* entity and so forth. With the exception of the *Defaults* entity, each of the other tables has at least one odd-looking and more oddly-labelled line attaching it to another entity. These lines represent *relationships*, which provide some of the real value in database management systems (DBMS). These relationships between tables allow us to model several different kinds of relations between entities. The most widely modelled type is the *one-to-many*. This relationship is easily understood as *one* instance of an entity in a table can have *many* related instances in another table. So *one* student in the Student table can have *many* bids in the Bids table *without having to duplicate all the student data in the bids table*. This is the critical piece. Let's dig in.

In the auction system modelled in Figure EZ, take note of the relationship between the *student* and the *bids* entities. At the tStudent end of the joining line, we find a tiny number '1'. At the tBids end, we find the symbol for *infinity*. This represents a one-to-many relationship between student and bid; literally, any *one* student can make infinitely *many* bids. While not exactly obvious, the relationship is forged between the student ID (in this case named sCUID - the prepended 's' referring to the fact that it is in the Student table) and a field in the bids table named 'bCUID' - the 'b' because it's in the Bids table. Let's take this slow... this is the heart of the system. Recall we said that, in reality, all we need to administer this bonus marks scam is a record of the student bids. There would be plenty of other administrative tasks to complete around tallying the number of bids and making sure the rules were followed and creating the distributions to determine who would be awarded bonus marks, but the *core* was just those two things: student ID and bid price. The relationship here between student and bid automates that requirement, but significantly reduces the data collection burden to achieve the same result. And this, we know, satisfies our input/output maxim for progress. We get a better outcome with less input, which equals *double progress!* So we have only one record per student, with one of the fields being CUID and, for each purchase (bid) made by that student, we replicate the CUID in the bid table. *Don't leave yet!* We're almost home... So one student can make many purchases but we only need to record that student's *Term, Year* and *Section* once, in the Student table.

Note that each *entity* (table) has a Primary Key (denoted by the little gold key). The PK in one table can become the *Foreign Key* in another table to facilitate the relational functionality. Note especially that the FK (for example bgID in the tBids table is a FK holding the value of a record in the tGame table's gID field. By convention, I prepend both the first letter of the table itself (in this case b for Bids table) and the first letter of the table from which the foreign key comes (in this case the PK field g (for Game) ID to the FK field name in the host table. That's how the bgID field in tBids gets its name. So what I want you to see is that even though the gID field in table tGame is a Primary Key, when we duplicate it as a Foreign Key in the tBids table and name it bgID, it does not enjoy any special status. It's simply a Foreign Key with no special identifier (no little coloured key at all). Still, it's critical. it allows us the store *many* bids for a single game simply by replicating the gID in the tBids table. Clear as mud?

This needn't get too technical, and indeed there are plenty of reasons to use a database *backend* on this system that we don't have need to explain. But let's just imagine this system being run using a spreadsheet as the backend. And make no mistake, it could well be done using a spreadsheet, but everything would need to be specifically designed and automated and each instance of a game in class would need to be on its own *datasheet* and eventually there would need to be a new spreadsheet because we would exceed the 255 (open to interpretations and available memory) datasheet capacity of a single spreadsheet and there would be much manual thrashing about looking for a particular student in a particular game in a particular term and heaven forbid that student should drop the course and come back in a subsequent term or even switch sections mid-term. This would cause no end of headaches. A new auction would require the duplication of an existing simulation onto a new sheet and any number of manual adjustments including deleting all the bid data from the previous game in order to run a new one. A database can handle all such *exceptions* with ease, can store all the data for an infinite number of games in the same database and can quickly and efficiently retrieve any set of records according to any criteria in an instant using *Structured Query Language* (SQL - see below). Databases make my job, as administrator of the system, infidelity less complicated. And isn't that what we are after? Same (or better) output from same (or less) input. Indeed.

Finally, much of he data collected by the system is dedicated to making certain that we are accountable for the awarding of bonus marks. I need to be 100% certain that everyone who deserves a bonus receives it, and equally that no one who did not earn it, gets it. Tall order, but handled nicely by system design, coding and database design.

ACID: https://en.wikipedia.org/wiki/ACID

Structured Query Language (SQL - pronounced *see-quill*) - the 1,000km view

SQL is referred to as a *fourth generation language*. As the name implies, SQL is a rules-based language (Structured) that, despite many slight variations, is the engine behind all relational database management systems. SQL is how everything gets created, updated, manipulated, deleted, copied, archived and reported upon. It has a very simple structure (which can get quite *logically* complex in use) that references all the objects in a database. Objects such as *entities* (tables), *attributes* (fields) and *relationships* (relations). SQL will even create a database for you. Here are a few SQL statements with explanations to get you familiar with its syntax.

Let's stick with the online auction system design we've been looking at so far.

Imagine a database to hold the tables in the auction. We can name it anything we like, so let's go with something... I don't know, novel? How about *auction*? Outrageous, I know. So the database name will refer to the database object, which is a container for other objects like tables. It's also how

the operating system in your computer will keep track of it. If you are using Access on a Windows machine (as am I), then creating a database with the name *auction* will result in a file being created with the name *auction.accdb*. The *acc* part is shorthand for *Access* and the *db* part is, well, guess. Sadly I can't show you this in action, as Access doesn't support creating databases *programmatically* (through SQL code). When you open Access, it gives you a chance to either open an existing database or create a new one, and it will name it whatever you like. You can then store it in the computer's file structure or in the cloud, as you prefer.

If Access *did* support the creation of a database via code, the statement might look something like this:

CREATE DATABASE auction

Boom. That's it. Once the database container is created, you can start filling it up with tables and specifying columns (attributes) and data types. To create a new TABLE names *tStudent* in our *auction* database, the SQL command might look something like this:

CREATE TABLE tStudent (sCUID varchar(9), sTerm varchar(1), sYear int, sSection varchar(1));

Let's unpack that on **bolded** piece at a time:

CREATE TABLE tStudent is just as it appears. It asks the database to create a table in the current database (the SQL system *lives* inside a database) and name it -*tStudent*. Simple. The rest of the statement concerns creating the fields (attributes) inside that table. Let's look at each in sequence:

CREATE TABLE tStudent **(sCUID varchar(9),** sTerm varchar(1), sYear int, sSection varchar(1));

This part has several pieces. First, the *(* parenthesis separates the CREATE part of the statement from the field list part. Note at the end there is a matching *)* closing paren. Simple stuff. Then inside the paren, we specify the fist field we would like to create by naming it, each new field delimited by a comma. Ours first field is *sCUID.* Additionally, we need to specify the *data type* of this field. We examined data type extensively without really specifying it as such in Chapter 1 when we introduced *measurement*. This variable we are creating will be a *varchar* (in SQL speak, this is a *variable-length character field*, meaning that it will hold *alphanumeric* data (text basically and if there are numbers in the data, then they are to be treated as text and not numbers) of variable size from 0 to 9 characters. So we could store *any* string of text from 1 to 9 characters in length in that field. Confusing I know. Stick with me. The sCUID field is an ID number. If you are at my school, the CU stands for *Carleton University* and the ID part is ID of course. This is a 9-digit number that begins with *100* and increments as each new student enters the school. It begins with 100 because we purchased and implemented an enterprise system (and ERP for schools) named *Banner*. Recall we said that ERP systems make you do business the way *they* say? Well in Banner, ID numbers are 9-digits long. Ours were 6 long. So we changed. We prepended a *100* arbitrarily to the existing CUIDs and forever after, have been merrily creating CUIDs that begin with 100. No other reason.

OK, long explanation for little value, except that it explains why this *varchar* ID field is 9 characters long (as indicated by the *(9)* following the name in the SQL statement). But why *character* data and not *numeric* data? It is, after all, a number like *100999666* isn't it? Yup, it is. So let's get back into the weeds again.

I decided to make it a character field type because I'm old. I was trained and building systems when storage was *very* expensive. I recall paying over $700 for a 20 MB (yes *megabyte*) external hard drive for my first PC (an IBM clone made by Dynalogic and sold all over the world by Bytek Comterm of Ottawa - a gorgeous little *luggable* called the *Hyperion* and retailing for ~$5,000 USD!) back in 1985. And even $700 in 1985 was a *lot* of dough. I thought I'd died and gone to heaven. People in the neighbourhood would drop in randomly just to look at it. I still have one in my backyard shed. Maybe I'll open a computer museum one day.

Back to the rationale. Every little trick to save memory (both RAM and external storage) was important. The size of the CUID, as a number, is large, it's a hundred million and some. In order to get the capacity to store that big a number in Access, you must specify a *long integer* field - long integers are capable of storing a number in the range of -9,223,372,036,854,775,808 through 9,223,372,036,854,775,807 (that's 9.2^E+18). Big, big number. Here's the thing. Even though our CUID is only 100 million and some, specifying the field here as *long data type* when we create it causes the database to *reserve space* for a number in that huge range *even though we don't need it*. So wasteful. So I tell the database to just set me aside 9 little spaces to store a number as text, and save all the overhead. I know what you're thinking (and it has nothing to do with what we're talking about here) but maybe some day you'll say to yourself "Then why store it as a *long* integer? If there's a *long* integer there must be a *short* integer!" And you'd be right. But short integers (referred to simply as *integers*) can store numbers only in the range of -32,767 through 32,768. Snookered. Our number is too large for that.

Figure HC. Oh yes, and here's that Hyperion computer I loved so much...

Photo credit: http://s7.computerhistory.org/is/image/CHM/102662819p-03-01?$re-zoomed$

And besides, we're never going to do any *math* on an ID number. Like add two of them together or calculate an *average student ID*. At best, that metric *might* give us a very rough indication of the average length of time a particular group of students (such as those in a class) have been

students - also a rough proxy for their average age. The higher the average ID, the more recently the group of students had enrolled and therefore the younger they might be. But really... is there any value in that? And it would be pretty rough and only of value when comparing one group of student to another. So it's fine if sCUID is stored as text. It's efficient use of memory and storage space.

Onwards.

CREATE TABLE tStudent (sCUID varchar(9), **sTerm varchar(1)**, sYear int, sSection varchar(1));

This is an easy one. at my school there are three terms *F*all, *W*inter and *S*ummer. I need a variable that will store up to one character, and let's call it *sTerm*.

CREATE TABLE tStudent (sCUID varchar(9), sTerm varchar(1), **sYear int**, sSection varchar(1));

Here's our *integer* variable. Normally I'd store this as text too because it's year and who does math on years? But I wanted to show you how to specify an integer variable (note it's shortened to *int* and there's no *field size* because Access knows it needs a number in the -32,767 to 32,768 range so it manages that storage allocation for you. Simple.

CREATE TABLE tStudent (sCUID varchar(9), sTerm varchar(1), sYear int, **sSection varchar(1)**);

Finally, I need a place to store which *section* of the course our players are in. At my school, there are several concurrent instances of this intro class running in any one term. They are designated by an UPPERCASE letter ranging from *A* to whatever is the last section offered in any academic year. So this one is easy. Just as we've spec'd above. A field named sSection that will store one character.

Et voila! We end with the closing paren ")" and a semicolon ";" by convention and we're done creating a table.

Fortunately for you, as savvy and well-trained users of database software, you don't need to actually *use* SQL in this way to do anything. In Access, there is a WYSIWYG (what you see is what you get) way to do everything. To create a table, you simply select *CREATE* then *Table* from the interface and then specify the fields and their properties one by one. Easy.

Figure RK. Create a table in-] Access

But let's press on shall we? There's some important stuff to cover here. We will finish with three different types of SQL statements. The *INSERT INTO*, *UPDATE* and the *SELECT* statements.

Given that we have created a table. we can now set about filling (*populating*) that table with data (this word is pronounced *day-tah*). The *INSERT INTO* SQL command allows us to do so, programmatically. There are two variations on this statement, but each accomplishes the same thing. Type 1 assumes that you will input values for each field in the table, from beginning to end in the same order in which that the fields were created. So it might look like this (note that the data for all Alphanumeric fields in a SQL statement is enclosed in 'single parens') :

UPDATE INTO tStudent ('100999666', 'F', 2015, 'A')

This SQL statement puts values into the four fields in our table, and assumes that the data in the comma-separated list correspond to the order of the fields in our table. Recalling that the order of our fields from the *CREATE TABLE* statement was: sCUID, sTerm, sYear and sSection, the UPDATE INTO statement would create a *record* in the tStudent table and insert '100999666' into the sCUID field, 'F' into the sTerm field and so on. This statement would then cause the tStudent table to appear as below in *Table View* in Access:

Figure RK. INSERT INTO version 1

The second variation of the *INSERT INTO* statement allows the programmer to pick and choose which field(s) to update and also allows to add out of order. This statement simply requires that the field(s) to be updated are *named* before their values are specified. Check this out:

INSERT INTO tStudent (sTerm, sCUID, sYear) VALUES ('F', '100666999', 2015)

This INSERT INTO statement, where we are missing the valuue for the sSection field, would insert a record into tStudent, causing the table to appear as below:

The resultant record is incomplete but doesn't violate any *integrity* rules in the database which would require that every field in a record must have a non-blank value. We could very swell set up one of these rules. But I haven't because I now want to demonstrate the *UPDATE* statement.

Let's assume that for one reason or another, the sSection value was unavailable at the time the record was created in tStudent. We INSERTed the record anyway, and made a note to come back later and UPDATE the record with the sSection data. In the meantime, the student corresponding to the sCUID '100666999' had unenrolled from the 'F' term and instead enrolled in Section 'G' in the 'W' (or Winter) term of 2016. So we have some work to do in order to bring our database up to date. The following SQL statement would do it all. Take a look:

*UPDATE tStudent SET sTerm = 'W', sYear = 2016, sSection = 'G' WHERE sCUID = '100666999' *

Bingo. All is now up to date.

But note that we have introduced a new concept (or SQL function) here, namely *WHERE*. The WHERE clause restricts the actions of the SQL statement to *only the record WHERE* sCUID has the value specified. If we did not do this, *ALL* records in tStudent would be updated with those values. No matter which student we looked at, they would all be in Section G or the Winter term of the year 2016. *Disaster!* So SQL is very powerful indeed, and not very *forgiving* shall we say.

The WHERE clause implicitly makes use of the *PRIMARY KEY* in the table. recall from Figure EZ above that the *Primary Key* in the tStudent table is sCUID? When we use the WHERE clause, it is *normally* implemented using the value of the PK, as the PK, as we recall, must be unique and must identify one and only one record in a table. Above we are informing the database engine that these changes in the UPDATE statement must *only* apply to the record uniquely identified with the sCUID of '100666999'.

The same *WHERE* clause is an integral part of selecting *subsets* of data for viewing. The WHERE clause can become quite complex using *boolean* logic (remember boolean? (take me there)). Let's start simple. Suppose we have a table full of students (hundreds or even more) and we are looking for one particular student in our database table. We *could* simply open the table view in Access, sort the students by sCUID and scroll down to find the target sCUID. While this is an effective way to search, it's certainly not efficient (way too much input effort for a little bit of output - so if this were how searches had to be done in a database, no one would use them and databases wouldn't exist as a tool). Imagine that we're looking for 10 different students from among 8,500. An arduous task you would agree. Thankfully, the SQL WHERE clause comes to the rescue. All we need to do in Access is to open up the Query Design window, click the SQL View option in the View menu and write our SQL statement. Let's say we're looking for our friend sCUID = '100666999'. The statement to pull (filter) the data to show only that student in table view (or *datasheet view* as Access calls it) would be:

*SELECT * FROM tStudent WHERE sCUID = '100666999'*

When we execute (run) this query, only the record for the student with that sCUID would be returned to us. the asterisk (*) in the SELECT statement is a shorthand way to say to the database that you ant to see *all fields* in the record. You could instead simply list, in any order and including duplicates (asking for the same field to show twice of more times - for what reason I don't know but I'm just saying you *could*) by simply listing them, comma-delimited, following the SELECT clause. An example:

SELECT sTerm, sYear, sCUID FROM tStudent WHERE sCUID = '100666999'

But let's get a little more complex, shall we? Let's go wild and ask for the data for *two* students! Shocking, I know, but easily doable. Just like this:

*SELECT * FROM tStudent WHERE sCUID = '100666999' OR sCUID = '100999666'*

Or how about this?

*SELECT * FROM tStudent WHERE sTerm = 'W' AND (sSection = 'D' OR sSection = 'E')*

Think back to your high school algebra for order of execution and the use of parentheses. We've bow combined *AND* and *OR* in our statement. The last remaining refinement in this simple SQL statement is the *NOT* conjunction. Decipher this example:

*SELECT * FROM tStudent WHERE sTerm = 'F' AND NOT sSection = 'B'*

This would give us all students from tStudent table from the Fall term (of ANY year in the database because we ahve not set a filter for Year in our statement) *except* for those enrolled in Section 'B'. Sneaky. Nobody likes those Bs anyway...

Ok now we get into the weeds again. Imagine we're looking not just for the information about a particular student, but also for information about all the *bids* they have made in our auction simulation game. Complex stuff. We need to isolate the student using a WHERE clause, but then take that student and query the bids table for all instances of that student's CUID in that table. This is where the *relational* aspect of databases really shine. Lt's build a query to show what we're looking for. We'll introduce some slight variations in the syntax here but the logic is follow-able.

First the *SELECT* part:

SELECT tBids., tStudent.sCUID*

So the logic here is "show me the data in all the fields in tBids, but only the sCUID from tStudent. Ok, simple enough. Now we'll skip over the fun

part in the middle and show the WHERE part:

WHERE tStudent.sCUID ='100666999'

That we've seen before. But the complex part is in-between these clauses, where we specify how to get the Bid data from the tBids table related to the sCUID we've specified. We do this using *JOINS*. Recall from Figure EZ (our ERD diagram) that there we specified a one-to-many relationship between tStudent and tBids? We used sCUID from tStudent as a Foreign Key in tBids so that we could do precisely what we're trying to do here, and that is *specify particular students and see what bids (purchases) they made.* This relationship is accessed through asking the database to *JOIN* the tables and return *JOINT* data, based on certain criteria. And that criteria here is the common CUID the tables share through the PK/FK relationship. Totally clear as mud I know. But just watch. We use the FROM clause to effect the JOIN (in this case an *INNER JOIN* but let's not worry about *types* of joins for now (or maybe even *ever*). Here's how it's done:

FROM tStudent INNER JOIN tBids ON tStudent.sCUID = tBids.bCUID

Let's unpack that. We want data from both the tStudent and the tBids tables, so we JOIN them using the *INNER JOIN* clause. But what's the criteria? The criteria specifies to show me the bid data where the bCUID in tBids is equal to sCUID that we selected from tStudent with the WHERE clause. Her is is in all its glory:

SELECT tBids.*, tStudent.sCUID FROM tStudent INNER JOIN tBids ON tStudent.sCUID = tBids.bCUID WHERE tStudent.sCUID = '100666999'

The result might look like this Figure RR below:

Figure RR. Datasheet view of SELECT with INNER JOIN

But listen. It's all *much* easier than this. These days, you really don't need to know how to code SQL (but you *do* need to know the logic!). Desktop databases such as Access are so user-friendly that they provide functionality that either fully automate tasks (such as CREATE DATABASE) or allow you to do easily specify complex tasks using wizards or WYSIWYG, drag-and-drop interfaces. But make no mistake. The database is still driven by SQL. The tools simply automate the creation of the SQL statements out of your direct sight and then submit them on your behalf to the database engine for execution. Just click the *View* menu item and then *SQL View* after creating a drag-and-drop query to see the SQL generated by Access. All databases and all applications using a database as a backend must allow for what is called a *complete CRUD.* CRUD stands for *Create, Read, Update, Delete*. The application must allow you to create records, read those records once created, update the data in the records and finally delete data when it is no longer of value. SQL allows for all of this, and more. Here is a good *quick reference* to basic SQL (just the syntax elements but there are links on the page for tutorials in all aspects of SQL).

What makes SQL and databases so useful is that SQL/database functionality can be *embedded* in applications to provide the full functionality of the database at the fingertips of the app developer. Once the API (Application Program Interface) has facilitated a database connection between a program and a database, SQL statements can be built on-the-fly in a program and the full functionality is available. This is exactly what is happening in the auction game we have been dissecting. Below is an early prototype of the user interface for the game.

Figure BG. Prototype interface

Let's just look at the elements in the interface and specify their relationship to the database ERD in Figure EZ.

1. This field comes from the database table tStudent (and early approximation of sCUID). It simply identifies the user and allows them to determine whether they are correctly identified by the program following login.
2. This is a link provided by the application to stop executing the program for this player.
3. This is a static title provided by the programmer. This is not related to a database field.
4. Another link to allow the user to go back and select a different simulation if one is available. No database link.
5. This is an output that is a hybrid of some static text provided by the programmer in the application (BETA 1001 A Week) and the gWeek field from tGame.
6. An indicator of the status of the game, written in the application code and controlled by an administrative interface (not shown) run by the instructor whereby games can be stated, paused, stopped and all other parameters set.
7. The start time of the game from gStart in tGame. This value was written to the database when the instructor clicked a *Start Game* button on the administrative interface (not shown).
8. Time remaining is calculated by the application by taking the difference between gStart + gDuration from tGame and subtracting a running count of seconds the game has been active, determined by the system clock.
9. This is from gMin_Buy from tGame.
10. This is from gMax_Buy from tGame. This and #9 were written to tGame when the instructor creted the game and set the parameters.

11. Another field that is dynamically updated by the running game application code and stored for retrieval in gTotal_Bids in gGame. This field represents the total number of purchases (bids) made thus far in the game.
12. Another calculated field determined by the difference between gAvailable and gTotal_Bids from gGame. gTotal_Bids is updated by the program code as each purchase in the game is made.
13. This is calculated by the application code and represents the average price paid by all players on all bids so far in this game. This data ceoms from tBids.
14. More from tBids. This is a COUNT (a SQL *function* we didn't introduce above) of all purchases made by Player 10 so far. It uses some complex SQL to isolate (filter) only the current player (using a WHERE clause) and then counts the number of records which, in this case, is equal to the number of purchases.
15. Here's a great example of the SELECT WHERE clauses, representing the average price paid so far on all purchases (bids) by Player 10 in this game.
16. This is a value calculated in code by the application according to a complex algorithm. When the player clicks the BUY NOW button (#17), the current price is written to the bBid field of a new record in the tBids table, along with all the other important data such as the bCUID of the bidder.
17. A button coded in the app to react to the click event, creating the record of a sale in the tBid table and writing all the pertinent data to the table.

Et volia! The game, explained.

To wrap up databases, let's talk about ACID:

Figure TL. Psychedelic trip, man...

Image credit: http://community.ebay.com/t5/image/serverpage/image-id/114045i37E0A83D33C5BCF8/image-size/original?v=mpbl-1&px=-1

Wait. Not *that* kind of acid. The kind where we're talking about *Atomicity, Consistency, Isolation and Durability.* What do these terms mean in the context of our discussion? Let's unpack.

Spreadsheets