Remembering numbers (and other stuff)...

Let's talk about one of the most important things in any programming language. It's called a "variable". Don't let the name scare you. What it does is really simple. A variable is like a post-it note or a scrap of paper you can use to remember a number.

Imagine you have a task where you need to remember how many birds, squirrels, and dinosaurs you see outside your window during the week. I track my own count of birds, squirrels, and dinosaurs using post-it notes stuck to my window. Each note is labeled: "birds", "squirrels", and "dinosaurs". Then I keep track of each count on the corresponding post-it note.

Computers keep track of numbers in a similar way, except instead of using post-it notes, they use spaces in their memory. A variable is just a way to name a certain spot in memory so you can store a number in that memory space, then use that number again later.

You can name a variable almost anything you want. Normally you'll pick a name that makes sense and relates to whatever you plan to store in the variable. In this case, "birds" makes sense if you are counting how many birds you've seen this week. You could keep track of birds using a variable named "lostSocks", but that wouldn't make much sense, though "lostSocks" would still work.

Lets consider this example...

```
birds = 10;
```

} This puts the number 10 inside the variable named "birds".

This line of code will remove whatever was previously stored in the memory space named "birds" and replace it with the number 10. This is also sometimes called an "assignment", as in, the number 10 is assigned to the "birds" variable. Experienced programmers would call the equals sign an "assignment operator", but you don't need to remember that for now.

You can also store the results of math in a variable. Let's consider this example...

```
birds = 10;       //birds now holds 10
birds = 2 + 3;    //birds now holds 5
```

} The second line adds 2 and 3, then takes the result of that addition and puts it back in "birds". The previous value of 10 is erased and replaced by the new value of 5.

You can also use the variable itself in the math if you want. This is often very useful, for example, if you're counting something. In the next example, we give "birds" a starting value. In the second line, we use whatever is already in the variable and do some math on it, then put the result back in the variable.

```
birds = 10;               //birds now holds 10
birds = birds + 1;        //birds now holds 11
```

} The second line takes what ever value is already present in "birds", then adds 1 to it, then takes the result and puts it back in birds.

2
SKILL LEVEL

"Declaring" variables...

Before you can actually use a variable for the first time, you have to tell the program ahead of time that you plan to use it. This tells the processor to automatically reserve a space in memory to hold the value you will eventually place inside the variable.

This process of telling the program that you plan to use a variable is called "declaring" the variable. Kind of like, if you were going to count jelly beans, you may get a scrap of paper and label it "jellyBeans" before you actually use it to count your beans. You only declare each variable one time.

```
int jellyBeans;  //this "declares" the variable jellyBeans
```

This is how you "declare" a variable. You need to do this for every variable your program uses.

A variable declaration needs at least two things. The first part is the "type" of variable. There are many different types of variables that can be declared depending on what you plan to store in the variable. The first part of the declaration ("int" in this case) is the type of variable you are declaring, and the second part ("jellyBeans" in this case) is the name you are giving to the variable.

The "int" variable type stands for "integer". This is a very flexible variable type and is commonly used within Arduino code projects. The "int" variable type can store any whole number between -32,768 and 32,767. You can NOT store decimal numbers in an "int" type. (More on that later).

You can name a variable almost anything you want as long as you follow a few simple rules.

1. Variable names can include upper case letters, lower case letters, digits (0 to 9), and the underscore (_) character. No other characters or symbols are allowed.

2. Variables must begin with a letter or the underscore _ character. You can not begin a variable name with a number. If you really need to begin a variable name with a number, it is customary to place an underscore first followed by the number.

3. No spaces are allowed in the variable name. All characters and numbers must be run together with no spaces. To create a variable out of several words, it is customary to run the words together and capitalize the first letter of each word, with the first word always using all lower-case letters.

4. Variable names can be up to 31 characters long. (They can be longer in some cases, but as a general rule, try to keep it to 31 or less to guarantee your code can be understood by the compiler that turns it into machine language).

5. The variable cannot be named any "reserved words". These are special words that mean something special to Arduino such as "break". If you type the name of a variable and it changes color on the screen, that is a good clue that it is probably a reserved word and cannot be used. You can still incorporate reserved words in the name as long as it's not an exact match to a reserved word. For example you could name a variable "breakPoint", but you can not name it "break" by itself because the word "break" means something special.

Initial value of a variable...

Every variable has a value stored in it, even if you have just declared it. Normally this will be zero, but there's no guarantee of this. In many cases, you will want your program to set a variable to some meaningful value before it is used. If you plan to use your variable to store the reading from a light sensor - the initial value doesn't really matter because you're going to read the sensor before using the value, which will erase whatever value it started with.

But in other cases, like counting how many dinosaurs you've seen this week, you'll probably want it to begin with some specific known value, like 0. You can give the variable an initial value at the same time you declare it if you want. This is optional, but considered good practice.

```
int loopCount = 0;  //declare loopCount and initialize it to 0
```

} This statement declares the variable and also specifically tells the program what its initial value should be.

A working example...

Lets have a look at working example. This is similar to the examples you've already seen. It declares the variable loopCount and makes sure it is set to a starting value of 0. Then the loop() function begins. Each time the program goes through loop, "loopCount" is increased by 1. It makes sense that we named it "loopCount" because that is exactly what we are storing in the variable. (That is, a "count" of how many times the loop() function runs).

```
# include WinkStuff.h

void setup(){
  hardwareBegin();
}

int loopCount = 0;

void loop(){
  loopCount = loopCount + 1;  //add 1 to loopCount
} //end of loop()
```

} Declares the variable "loopCount", sets it equal to 0.

} Add 1 to loopCount every time the program goes through the loop() function.

Wink_Ch06Vars_Ex01

If you actually load this onto Wink, it won't look like he's doing anything, but he is really using all his brain power to count loops as fast as he can. In the next lesson we'll show you how to print these values to your computer screen so you can see what he's thinking. This is just a simple example to show you how a variable can be used.

2
SKILL
LEVEL

A few factoids about variables...

Here are a few other random tidbits about variables.

Roll Over: A variable will “roll over” if you try to give it a value outside the range it can store. For example, if you populate a variable with the highest number it can store, then add one to it, the value will “roll over” and become the lowest value it can store. This works the same in the negative direction.

If you are ever working with a program and the variables seem to be giving you very strange numbers, this could be the cause. The C language will not prevent you from rolling over a variable. It is happy to let you make the mistake.

```
int x;
x = -32768;
x = x - 1;    // x now contains 32,767 - rolls over in neg. direction

x = 32767;
x = x + 1;    // x now contains -32,768 - rolls over
```

Source: Official Arduino Website
https://www.arduino.cc/en/Reference/Int

Memory Limits: You may already be wondering how many variables you can declare. This really depends on how much memory you have available on the processor you are using. Wink has 2 KB of RAM memory (that is, about 2000 bytes). Each “int” variable requires 2 bytes, so you could theoretically use 1000 different “int” variables in your program. However, Arduino, as well as Wink’s background code (inside WinkStuff.h) claim about 300 bytes of the available 2000 bytes, but that still leaves you about 1700 bytes (about 850 “int” variables) that you can declare and use.

It is unlikely that you will ever run out of memory on Wink. The only time you may press these memory limits is if you’re using large arrays (that’s a Skill Level 3 topic, so don’t worry about it now).

Memory Re-Use: Keep in mind that every time you load a program onto your Wink, all the memory is erased and re-used. (Except for some special memory called EEPROM that keeps values forever - we’ll discuss that later in Skill Level 3). For now, just know that you can keep re-writing the memory over and over. If you use lots of variables in a given sketch, you won’t eventually “run out of space” later on.

Other variable Types...

We mentioned before that there are several different “types” of variables. We’ll briefly discuss them here. There are even more types than we’ll cover, but these are the common ones that will get you through almost anything you’ll ever need to do.

Variable Type	Values Stored	Bits	Notes
`byte`	0 to 255	8	Number from 0 to 255, no negative numbers allowed.
`char`	-128 to 127	8	Some compilers will try to interpret this variable type as a Character (like a letter typed on your keyboard)
`word`	0 to 65535	16	This can count higher than “int” except you can’t use it to store negative numbers.
`int`	-32768 to 32767	16	This is the most common general purpose variable type. It allows storage of fairly large numbers as well as negative numbers.
`unsigned long`	0 to 4,294,967,295	32	This variable type is required to store the result of the millis() function which returns the number of milliseconds the current code has been running.
`long`	-2,147,483,648 to 2,147,483,647	32	Useful for storing very large numbers that could be positive or negative.
`float`	-3.4028235 e38 to 3.4028235 e38	32	The “Floating Point” data type can store very large numbers as well as decimal numbers. It is very flexible, though the processor does need more time to process it. Use float data types only when you really need the decimal function.

Note about variable Bit-Length: You’ll notice the “Bits” column in this chart. We’ll go deeper into bits in Skill Level 3, but here’s a quick intro. Computers store values as sequences of 1’s and 0’s. The “byte” variable type can be stored using a series of eight 1’s and 0’s, so we say it is “8 bits long”. In order to store bigger numbers like the “word” variable type, eight bits isn’t enough. We need a sequence of sixteen 1’s and 0’s to store a “word”. A “float” variable needs a sequence of 32 ones and zeros.

This doesn’t really matter now in our simple examples, but know that Wink’s brain (and all other 8-bit processors) can only “think” about 8 bits of information at once. Usually Wink can add a pair of 8-bit numbers together in a single cycle. He can add 16 bit numbers together, but he has to break them into several 8-bit cycles, and to do math on 32 bit numbers, he has to perform quite a few 8-bit cycles. He performs 8 million cycles each second, so even handling 32 bit numbers is extremely fast.

As a general rule it is a good idea to stick to 8 bit numbers if possible, with 16 bit as a second choice, and leave 32 bit variable types only for when you really need them.