

Welcome to Processing

Are you sure you want to do this?

ok.

I'm going to give an introduction to programming in general and Processing specifically. Lets start with *kinds* programming.

For starters, there's *markup* (like HTML) and then there's *programming*.

Markup simply involves tagging, or marking, information (usually text). It's not really programming, It's more of a format for data. Programming languages also represent data, but more importantly, they contain instructions for *manipulating* that data.

Programming languages that you'll encounter can be split into two gross categories: *Interpreted* and *Compiled*. Compiled languages make up most of programming. Interpreted languages are usually for managing the more complicated compiled programs.

All computers compute in binary. No one writes binary though 'cause that would be A PAIN. Not to mention that doing binary math is what computers are for. Instead, at the most fundamental level programmers use larger sets of numbers to tell a computer chip what binary operations should be carried out on what binary data. This is called machine language or first generation programming. The resulting programs are super efficient but are Rambo tough to write and understand.

The first step away from this is assembly languages. At this level people write in a more abstracted form of code which a special program, an "assembler" turns it into machine language. The final translated code is a self sufficient program separate from the actual written code, or source. Assembly actually has some glimmer of human readability, but is still pretty obtuse.

Lastly (kind of) you have compiled, or third generation languages. These languages are far more abstract and human readable. Like assembly, they too need to get squished into something a computer chip and understand. In actuality they're first turned into assembly code, and *that* is assembled into machine language. This translation usually takes a few moments to a few minutes for small programs (the kind we'll build) to hours or days for very large ones (like photoshop).

Here's some examples in different types of programming languages of a small program that calculates Fibonacci (golden section) numbers.

1st generation: x86 machine code (similar to... other machine code)

```
8B542408 83FA0077 06B80000 0000C383
FA027706 B8010000 00C353BB 01000000
B9010000 008D0419 83FA0376 078BD98B
C84AEBF1 5BC3
```

2nd generation: x86 assembly (similar to COLBOL, FORTRAN, etc.)

```

fib:
    mov edx, [esp+8]
    cmp edx, 0
    ja @f
    mov eax, 0
    ret

@@:
    cmp edx, 2
    ja @f
    mov eax, 1
    ret

@@:
    push ebx
    mov ebx, 1
    mov ecx, 1

@@:
    lea eax, [ebx+ecx]
    cmp edx, 3
    jbe @f
    mov ebx, ecx
    mov ecx, eax
    dec edx
    jmp @b

@@:
    pop ebx
    ret

```

3rd generation: C (similar to C++, JAVA, etc..)

```

#include <stdio.h>

int fib(int n)
{
    if(n > 2)
        return fib(n-1) + fib(n-2);
    return n;
}

int main(int argc, char** argv)
{
    int res;
    printf("Calculating fib(30)...");
    fflush(stdout);
    res = fib(30);
    printf("done.\n");
    printf("fib(30) = %i\n", res);
}

```

}

There's also forth generation languages, but those are mostly 3rd generation languages made for solving specific problems (like perl, which was *originally* made *just* to manipulate strings of characters). Forth generation languages though can also include interpreted or scripting languages.

Interpreted languages are typically those used to talk to compiled programs. They can be used to manage processes. These languages are also known as scripts, or scripting languages. They're less complex and less powerful. Less powerful because A) the computer is interpreting them on the fly, opposed to having commands pre-digested (compiled) and B) because their commands are often made to be even easier to understand for humans, *even more* work is done translating these commands into something the computer can use. These kinds of languages are what you use when creating a macro or a batch process.

Ok, on to Processing!

Processing and Arduino (the language) are both compiled, they're also both built on 3rd generation languages (JAVA and C respectively). If you've tinkered around with processing and pushed the play button and waited a few seconds for something to happen, you were waiting on the program to compile!

If you don't know, Processing was created by two smart guys Ben Fry and Casey Reas at the MIT media lab in an ongoing attempt to make programming easier for artists and designers. Processing is built on JAVA, which you might have heard of. What this means is that advanced processing programmers can use JAVA to do more complex stuff, or even to extend what processing itself can do.

Arduino uses a separate language that *looks* a lot like processing, so if you learn a little of one, you know a little bit of the other. Each language needs to be compiled to run, but Processing is compiled into programs for your computer whereas arduino is compiled into programs for the arduino micro-controller. Think of like one makes a program for mac and the other makes a program for PC. The neat thing is you can write processing programs that talk to arduino programs and vice versa. Basically this allows you to use the horsepower of an intel (or whatever) chip to do heavy lifting while arduino handles custom input and output (if you so desire). This is particularly useful when using physical input to drive screen based or projected visualizations.

Before we move onto the basics, you should learn one *very* important line of code:

```
print( "Lamp" );
```

This will put the text, `Lamp`, in the output area. Getting your computer to talk to you is the *first* thing you need to learn in *any programming language*. You can use this (or similar) line anywhere in a program. It is VERY useful in trying to see whats going on under the hood. It is your keyhole into the black box of the computer. For now you should only change the text in between the quotes, nothing else. ie. TOUCH NOTHING BUT THE LAMP

Ok, onto the meat (or tofu).

While programming languages might have different spellings for things, they're all going to have a few of the same basic building blocks.

- **Expressions**
- **Variables**
- **Control Statements**
- **Functions**

Learning these can be a little dry, but if you *really truly* understand at least this much you have a *lot* to work with, whether in Processing, Arduino, or any other high-level language.

On a separate note, some of the code I include in the beginning of each section is for example purposes and won't do anything (or might through errors) if you attempt to use it by itself. The code towards the end of each section though is a-ok.

Expressions are essentially *mathematical* expressions

```
1+2
10*100
16016264/2
```

if you write something that doesn't make sense mathematically You'll get an error

```
1+(1*100/74)- 42           // ok!
1+(((*---6                // ERROR: WTF is this?
```

Expressions always need to be ended with a *semi-colon*:

```
1+2;
10*100;
16016264/2;
```

With a semi-colon you can put multiple expressions on the same line if you like, but it's hard to read that way...

```
1+2; 10*100; 16016264/2;
```

Expressions *won't work by themselves though*. The results need to be stuffed in (assigned to) a **variable**.

```
a = 1 + 2;
b = 10*100;
c = 16016264/2;
```

BUT, before you do this, because computers are dumb, they need to know two things:

1. *What words or letters are your variables*
2. *What kind of data are you going to put in those variables*

So before you use a variable *you must declare it*.
This is done by writing the **variable** with the **type** of data in front of it.

```
int a;  
float b;  
char c;  
string d;
```

Here are some important types of data:

```
int          // is an integer, a counting number like 1, 1001, or -10.  
  
float        // is a decimal number.  
  
char         // is a single character like "a" or "&". Characters must have  
            // quotes around them.  
  
string       // is a group of characters "Hello". They also need quotes.  
  
color        // is a hexadecimal number like #FFFFFF (white)  
  
boolean      // values are either true or false. An expression like 10<100  
            // turns into the value true.
```

These and other words are *reserved*, meaning you *can not* use them for variable names.

```
boolean int = true;          //yeaaaaaaaa no.
```

You can also declare a **variable** *and* assign a value to it at the same time.

```
int    a = 1;  
float  b = 3.14596;  
char   c = "1";  
string d = "Hello, how are you?"
```

If you try and put one kind of data into a variable that is declared as something else, the computer will try and convert it. Depending on the type of data and the type of variable this might not work and you'll get an error:

```
char   a = 1.1           // ERROR! cannot convert from float to char  
float  b = 1             // This is ok, integers can be converted to floats
```

Variables can be used in **expressions** just like numbers

```
float a = 1;  
float b = 2;  
float c = a + 10 + b;
```

Variables can have long names too, *but can not have spaces (or funky characters)!*

```
int tacosInMyBelly = 10           // Ok!  
int tacos I will eat = 100        // Uh Oh!  
int t@c0sIwillE@t = 100          // Uh Oh!
```

Finally, **Variables** are also *case sensitive*. These are two different variables:

```
int myValue = 1;           // a variable
int myvalue = 1;          // a different variable
```

Processing comes with some pre-made variables (also reserved keywords) that have data you can use, but that you can't replace. These kinds of variables are known as *read-only*. Here are two important ones

```
mouseX
mouseY
```

They can be used in your **expressions**

```
float a = mouseX / 100;
```

OK.

Control Statements

Computers typically march through your expressions one at a time from top to bottom. This is called *program flow*. Using control statements you can tell it to do other things, thus *controlling* program flow. There are two general kinds of control statements: “looping” and “branching”

Loops are the most powerful so we’ll start with them.

Computers seem like they do a lot. But really they only do a few things. They just do those things REALLY FAST and will do them FOREVER if you tell them to.

First, you can group a collection of expressions with brackets.

```
int a = 1;
int b = 2;

{ a = a + b; b = a + b; }
```

Uuuuuusually it’s spaced out so its easier to read.

```
int a = 1;
int b = 2;

{
    a = a + b;
    b = a + b;
}
```

You can then have the computer do (or execute) this collection over and over and over and over and over by using the built in control statement `while`. This statement is followed by a set of parenthesis that contain an **expression**. If the expression is true, the computer will execute the code in the brackets then return to the expression to check again. It will keep doing this so long as the expression evaluates to `true`. The program below will keep adding forEVER.

```
int a = 1;

while(true){
    a = a + 1;
}
```

The next loop will only execute three times.

```
int a = 1;

while(a < 4){
    a = a + 1;
}
```

The next type of statements are *branching statements* and look very similar. The keyword is `if`. Like the while loop, if the expression in the parentheses is true, the computer will execute the code inside the brackets. *But only once*. If false, it will ignore the code.

```
if(mouseX < 100){
    print("cursor is close to the left side");
}
```

Using the keyword `else`, you can have the computer execute a different chunk of code

```
if(mouseX < 100){
    print("cursor is close to the left side");
} else {
    print("cursor is NOT close to the left side");
}
```

You can combine `if` and `else` to create sophisticated decision trees

```
if( mouseX < 100){
    print("cursor is close to the left side");
} else if( mouseX > 400) {
    print("cursor is close to the right side");
} else {
    print("cursor is in the middle");
}
```

If (and else) statements can be nested for more complex decision making.

```
if( mouseX > 100){
    if( mouseX < 200){
        if( mouseY > 100){
            if( mouseY < 200 ){
                print("The cursor is within a 100 x 100 box");
            }
        }
    }
}
```

This is very useful but gets cumbersome. Sometimes you can save yourself trouble by using **logical operators** to write one **expression** that checks several things

```
true && false    // 'true AND false' equals false
true || false    // 'true OR false' equals true
!true            // 'NOT true' equals false
```

ex.

```
if( mouseX > 100 && mouseX < 200 && mouseY > 200 && mouseY < 200){
    print("The cursor is within a 100 x 100 box");
}
```

Fun fun **functions**.

Functions are collections of code you can re-use. Processing calls them **methods**. A method is actually a *specific kind of function*, but we won't cover that here. For now we'll just call them **functions**.

In addition to values, entire collections of code can be stuffed into a variable. A *special* variable; called a **function**. A function is like a variable that contains instructions instead of data.

A basic function declaration looks like the control statements from earlier, except it has a **type**.

```
void nameOfFunction() {
    int a = 1;
    int b = 2;
    int c = a + b;
}
```

Functions, when used, are transformed by the computer into the result of their expressions. So like variables, the computer needs to know what kind of data the function will turn into when executed. The most basic function has the type **void**. This means the function will not turn into anything. For example, `Print()` is a function (built into processing) that does something but does not transform into a value. The example above is a custom function that does not transform into a value.

For the function to transform into something when used, the code inside of it must include the command `return` followed by a value. Make sure the type of the function matches the type of the return value.

```
int nameOfFunction() {
    int a = 1;
    int b = 2;
    int c = a + b;
    return c;
}
```

This above is a **function declaration** but does nothing but take up space. To use it we must **call** the function somewhere else in our program by using it's name followed by `()`.

```
int a = 1 + nameOfFunction();
```

When the computer *evaluates* the above line of code, it turns the "nameofFunction()" into the number 3 (the value the function calculated and returns).

```
a = 1 + nameOfFunction();
```

In the computer's mind turns into:

```
a = 1 + 3;
```

Processing has a few built in functions that return values almost like they were variables, most notably the ones for getting the time.

```
second();  
minute();  
hour();  
day();
```

Some functions accept input data. Input is put in the parenthesis of the **function call**. The function then does something with it and returns the result. Processing has these kinds of functions too. Here's a couple of them.

abs() returns the absolute value of the input

```
abs( -100 ); // returns 100
```

floor() takes a decimal number and returns that value rounded down

```
floor( 3.14 ); // returns 3
```

random() accepts two numbers (separated by a comma). It then returns a random integer (or **int**) between the two numbers

```
random(-100,100); // returns... 10.... this time...  
random(-100,100); // returns -42... maybe...
```

To *write* a function that accepts input, we declare variables in the prothesis

```
int squareANumber(float value){  
    return value * value;  
}
```

Then we can use it!

```
int twoSquared = squareANumber(2);
```

Ok, that's all for now. If you're curious about something you come across in your tinkering that isn't covered here, take a moment to look through processing.org/reference/. There's all kinds of useful information there. Afterwards feel free to email me at ian@ianbellomy.com.

Potato