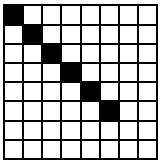
Drawing Reference

Introduction: Drawing in Processo is achieved by issuing shape command functions. The shapes will be drawn on the coordinate system of your drawing canvas, which starts from the *upper-left corner pixel* which is (0, 0). X values increase as you go to the right and Y values increase as you go down. You won't be able to see anything that you draw outside of your canvas. Shapes will assume the *last specified properties*.

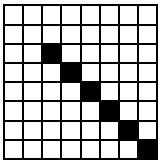
Shapes:

Command: `line(startX, startY, endX, endY);`
Description: Draws a line segment from pixel (***startX***, ***startY***) to pixel (***endX***, ***endY***), inclusive. Only uses stroke (no fill). The representations below are shown with a stroke weight (line thickness) of one pixel.

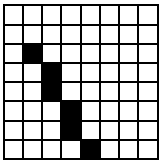
```
line(0, 0, 5, 5);
```



```
line(2, 2, 7, 7);
```

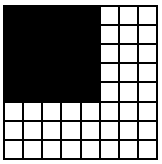


```
line(1, 2, 4, 7);
```

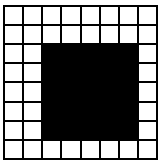


Command: `rect(startX, startY, width, height);`
Description: Draws a rectangle of size ***width*** × ***height*** (not including stroke) with upper left corner at (***startX***, ***startY***). The representations below are shown with no stroke and a black fill.

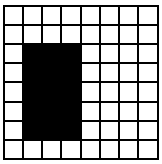
```
rect(0, 0, 5, 5);
```



```
rect(2, 2, 5, 5);
```

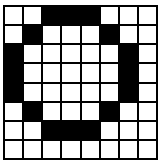


```
rect(1, 2, 3, 5);
```

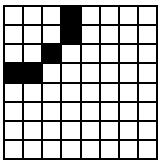


Command: `ellipse(centerX, centerY, width, height);`
Description: Draws an ellipse of size (***width***+1)×(***height***+1) (not including stroke) with center at (***centerX***, ***centerY***). **(The extra pixel represents the center from which the ellipse's radius extends.)**

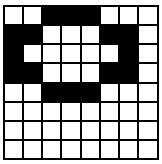
```
ellipse(3, 3, 6, 6);
```



```
ellipse(0, 0, 6, 6);
```

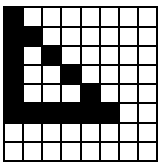


```
ellipse(3, 2, 6, 4);
```

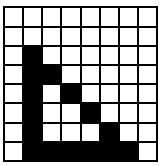


Command: `triangle(x1, y1, x2, y2, x3, y3);`
Description: Draws a triangle between points (***x1***, ***y1***), (***x2***, ***y2***), and (***x3***, ***y3***), inclusive.

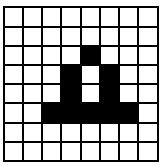
```
triangle(0,0, 0,5, 5,5);
```



```
triangle(1,2, 6,7, 1,7);
```



```
triangle(4,2, 6,5, 2,5);
```



Color: Colors are represented by a triplet (i.e. 3 numbers) that specifies the amount of red, green, and blue – always in that order – to mix together. Each number ranges from 0 (none of that color) to 255 (bright), inclusive. Colors can be applied to your drawing canvas background as well as the fill and stroke of shapes.

In a computer, colors mix like light (i.e. additive color). You can find color triplets by clicking the [Color Selector] button, choosing a color, and observing the values in the R, G, and B fields. You can form a color by giving a triplet to the `rgb` function. The `rgb` function will then return a color which can be given to the background, fill, or stroke commands. Here are a few example color triplets:

black:	`0, 0, 0`	white:	`255, 255, 255`	dark-ish gray:	`100, 100, 100`
yellow:	`255, 255, 0`	cyan:	`0, 255, 255`	magenta:	`255, 0, 255`
purple:	`128, 0, 128`	brown:	`150, 75, 0`	orange:	`255, 165, 0`

Alternatively, colors may be specified using the HSV color model using the `hsv` function which also takes 3 numbers. The first number is hue, which is a color of the rainbow starting with 0 for red, 30 for orange, 60 for yellow, 120 for green, 240 for blue, 270 for purple, 300 for magenta, 360 back to red. The second number is for saturation – saturation of 0 is completely washed out (white) while 1 is for completely saturated color. The third number is for brightness value – value of 0 is completely dark (black) while 1 is for completely bright.

Command: `background(rgb(color triplet));`
Description: Covers the entire drawing canvas with the specified ***color triplet***. When your Processo program launches, the canvas begins as a light gray shade with the RGB color triplet 204, 204, 204.

Command: `fill(rgb(color triplet));`
Description: Changes the fill (inside) ***color*** for all future shapes. The fill begins as white.

Command: `stroke(rgb(color triplet));`
Description: Changes the stroke (outline) ***color*** for all future shapes. The stroke begins as black.

Other Drawing Commands: These may come in handy.

Command: `size(width, height);`
Description: Sets the size of your drawing canvas in pixels. Can only be used once (i.e. no resizing). If you don't set it, the default size will be 100 by 100 pixels.

Command: `noFill();`
Description: All future shapes will be drawn with an empty inside. Overridden by future calls to `fill()`.

Command: `noStroke();`
Description: All future shapes will be drawn without an outline. Overridden by future calls to `stroke()`.

Command: `strokeWeight(weight);`
Description: All future lines and outlines will be drawn with the specified pixel thickness. Weight begins as 1.

Command: `quad(x1, y1, x2, y2, x3, y3, x4, y4);`
Description: Draws a quadrilateral (four sided shape) between points (***x1***, ***y1***), (***x2***, ***y2***), (***x3***, ***y3***), and (***x4***, ***y4***), inclusive. The points should be in clockwise or counterclockwise order around the edge.

Command: `arc(centerX, centerY, width, height, startAngle, endAngle, arcType);`
Description: Draws an arc based on an `ellipse`. Angles are in radians (0 to 2*PI) and ***arcType*** is `OPEN` (like the letter C), `CHORD` (line drawn between start and end like the letter D), or `PIE` (like a pie chart segment).

Variables Reference

Introduction: A variable is a storage location that is referenced by name and contains a value of a particular data type. A variable is like a container of a specific shape: the name lets you refer to a container and each container can only hold contents/values that fit that particular shape.

Variables are most often used for (1) storing and manipulating a value that will change over the course of your program and/or (2) giving a meaningful name to a particular value used in your program.

Syntax is the way we write a computer program, consisting of letters, numbers, symbols, punctuation, and spacing. When we give syntax below, we will use *underlined italics* for a word (usually a phrase written as a word, *likeThis*) to indicate that you don't type the word as written, instead filling it in as appropriate. For example, here is syntax to identify yourself in English: "My name is *myName*." When writing the sentence, you would fill in *myName* with your actual name. I don't know what your name is so I can't fill it in for you!

We write Processo syntax `using the typewriter font`. Processo uses syntax highlighting which we try to follow: types are `orange` and built in functions are `blue`, as we saw on the Drawing Reference sheet.

Data Types: The following is a list of the most commonly used data types in Processo.

`int` – Stores an integer: a positive or negative number with no decimal point (e.g. -7, 0, 120).
`double` – Stores a positive or negative number which can include a decimal point (e.g. 3.14, -1.0).
`color` – Stores a color triplet, which can be defined using `rgb(`*redValue*`, `*greenValue*`, `*blueValue*`)`.
`char` – Stores a character: a single symbol that you can find in text files (e.g. `'a'`, `'C'`, `'5'`, `'$'`, `' '`).
`boolean` – Stores a logical value, which can only be `true` or `false`.

Variable Declaration: Before you can use a variable, you must declare it (i.e. ask Processo to create the container). Because programs are executed in sequence, the variable declaration must occur *before* you try to use the variable. A variable declaration is of the form: *varType* ***varName***;

- *varType* is one of the data types listed above and indicates the shape of your container (and therefore determines what values it can store). I wrote it in bold because
- ***varName*** is the name associated with this variable. You should try to use intuitive names that indicate what the variable is used for. Variable names can include upper and lower case letters, numbers, and underscores, but *cannot start* with a number. Names are case sensitive, so the name Hello and hello refer to totally different variables in Processo which could each hold different values. You cannot use a data type as a variable name (e.g. you can't declare `int color;` because `color` is a type).

When Processo creates a new variable/container, it starts "empty." Processo will complain if you try to use a variable when it is empty. To fill a variable while you are declaring it, you may initialize it like so:

varType ***varName*** = *initialValue*;

Examples:
```
int len = 50;
color purple = rgb(128, 0, 128);
```

On the next page, we'll see more about using variables as well as some advanced math expression material which you can come back to later if Processo's math ever mystifies you!

Using variables: You can freely use variable names in your program wherever you are performing a command and they will automatically be substituted with the *current* value stored in the variable.

Examples:
```
ellipse(40, len+20, len, len); // circle of diameter 50 at (40, 70)
background(purple);            // set background to (128, 0, 128)
```

To *change* the value stored in a variable, we use an assignment statement: ***varName*** `=` *`newValue;`*
Although we use an equals sign, you should think of the effect/behavior as ***varName*** ← *newValue*.

Examples:
```
len = len + 10; // len becomes 60, which is current value (50) plus 10
purple = rgb(75, 42, 133); // a darker shade of purple
```

Advanced Math Expressions: Processo can do math with `int` and `double` values using the familiar addition +, subtraction -, and division / symbols as well as * for multiplication and % to calculate remainder. It uses mathematical order of operations (multiplication and division occur before addition and subtraction). You can use parentheses to evaluate an expression separately, just as in math.

Examples:
```
len = 1 - 2 * 3;   // len becomes 1 – 6 which is -5
len = (1 - 2) * 3; // len becomes (-1) times 3 which is -3
```

Processo remembers what kind of number you work with, so when you add two `int` values, the result will also be an `int`. What may be surprising is that **when you divide two `int` numbers, you will ALSO get an `int`!** Processo always **rounds toward zero** in this case. Processo isn't smart enough to know if you are saving the result into a `double` variable, so even if it has room for to calculate a `double`, it will still round.

Processo won't let you try to store a `double` value in an `int` variable (you will see a red underline in the editor). If you do math with an `int` value and a `double` value, it will promote the `int` value to a `double` value for you. Therefore, **if you want to avoid rounding, make sure that at least one side of any division operation is a `double` value!**

Exampless:
```
len = 19/10;           // len becomes 1, which is 1.9 rounded down
double area = len / 2; // area stores 0 which is 0.5 rounded down
area = len / 2.0;      // area stores 0.5 since 2.0 is a double
```

The % operator to calculate remainder helps make up for the rounding behavior. This is the same as the remainder that you would calculate when doing long division. The % operator has the same order of operations as division.

Example:
```
len = 19 % 10; // len becomes 9, which is the remainder of dividing 19 by 10
```

If you want to force a `double` value to fit into an `int` variable, you can use the cast operator. Put the desired type in parentheses next to the value. Casting has a high order of operations so use parentheses around the expression you want to cast. When casting a `double` value to an `int`, it will round toward zero.

Example:
```
len = (int) (area + 1); // len becomes 1, which is (0.5 + 1) rounded down
```

Functions Reference

Introduction: A function is a subroutine that can be referenced by name. You may liken this to the chorus of a song: between verses you sing or play the same chorus without having to write out the same thing all over again. In a computer program, a function is a set of instructions that execute when you call a function. After the function is done, it will return to the point in the program that it was called from. Optionally, a function may pass a return value back to its caller.

The main benefits of using functions are (1) decomposition of larger problems and (2) code reuse from different places in your program. Code reuse can be *identical*, or *similar* if you use parameters.

Function Definition: Because you are creating a subroutine from scratch, you need to tell Processo (1) what it does and (2) how you plan to use it. The first line of a function definition is the function interface and tells Processo how you will use the function: *returnType* ***functionName***(*type1 param1, …*)

- *returnType* is the data type of the return value. If there's no return value, use `void` here instead.
- ***functionName*** is the name of your function. The naming rules follow those of variables.
- The parameter list is contained within the parentheses. This is a comma-separated list of variable types and names. This will declare a set of variables that you can use only within your function. If you are not using any parameters, you may leave the parentheses empty (but they are still required!).

After the function interface, the function body (what it does) is found within curly braces ({}). Note that a function definition does not require a semicolon! It is good practice to indent your function body so it is easy to tell which statements are part of that function.

Examples:

```
void owl(int s) {
    // s is short for "scale"
    strokeWeight(s/2);
    ellipse(7*s, 7*s, 9*s, 12*s);
    ellipse(5*s, 5*s, 4*s, 4*s);
    ellipse(9*s, 5*s, 4*s, 4*s);
    ellipse(5*s, 5*s, 1*s, 1*s);
    ellipse(9*s, 5*s, 1*s, 1*s);
    triangle(6*s,7*s, 8*s,7*s, 7*s,9*s);
}
```

```
int sum(int x, int y) {
    return x+y;
}
```

Calling Functions: Functions are called by name, followed by an argument list in parentheses. The argument list corresponds to the parameter list from the function definition and provides the values that the parameters get initialized to when your function executes.

- Argument values get assigned to parameters *in order*
- Argument values can be written explicitly or taken from variables or calculations.

`min`(*a*, *b*) returns the lesser number (minimum) of *a* and *b* `max`(*a*, *b*) returns the greater of *a* and *b*

Examples:

```
int z;
z = sum(5,10); // run sum() with x=5, y=10; assignment puts return value in z
owl(z);        // draw an owl with s=15
owl(max(z, z+z));            // draw an owl with s=30 (larger of 15 and 30)
owl(min(sum(5,7), sum(8,2))) // draw an owl with s=10 (smaller of 12 and 10)
```

Setup and draw: The commands you have been writing so far in Processo outside of any function have been *setup* commands. Those commands are run as soon as you click the Run button for your program. After that, the program will automatically run the `draw` function over and over again, 60 times a second. If you define any functions at all, you must define a `void draw()` function at the bottom of your program. Function calls can be in your setup code or inside other functions, but you can only call functions defined before you call them (or ones that are built into Processo, like the drawing commands and the math functions below). You can alternate between setup code and function definitions.

Math Function Reference: Here is a list of helpful math functions. All the math functions have `double` parameters and calculate and return a `double` value based on the parameter values, but as we saw in the Variables Advanced Math Expressions Reference, we can pass an `int` value in place of a `double` parameter and it will be promoted to a double. To convert the output of the math function to an `int` value, you must cast it by writing (`int`) to the left of the function call. **Don't worry if you don't need to use most of these functions!**

Exponentiation and logarithm:

`pow(a, b)`	Calculate *a* to the *b* power
`sqrt(a)`	Calculate the square root of *a*
`log(a)`	Calculate the natural logarithm of *a* (use `log(a) / log(b)` to get logarithm base *b* of *a*)

Rounding:

`ceil(a)`	Calculate *a* rounded up to an integer (**note that you still must cast to get an `int`**)
`floor(a)`	Calculate *a* rounded down to an integer (**see above note**)
`round(a)`	Calculate *a* rounded to the nearest integer (**see above note**)

Trigonometry (you probably won't need these for this class!):

`sin(a)`	Calculate the sine of *a* in radians (conversion: `2 * PI` radians = 360 degrees)
`cos(a)`	Calculate the cosine of *a* in radians
`tan(a)`	Calculate the tangent of *a* in radians
`asin(a)`	Calculate the inverse sine of *a* (vertical coordinate on unit circle) to radians
`acos(a)`	Calculate the inverse cosine of *a* (horizontal coordinate on unit circle) in radians
`atan(a)`	Calculate the inverse tangent of *a* in radians
`atan2(y, x)`	Calculate the angle in radians of the coordinate (*x* , *y*) **(NOTE: *y* is passed first)**

Random numbers:

`random(a, b)`	Calculate a random number uniformly between *a* and *b* but ***not including*** *b*

The `random` function can be very useful for games and simulations! If you need a random integer, make sure you remember that the upper end of the range is not included.

For example, if you want a random integer that has roughly equal chances of being 1, 2, or 3, use (`int`) `random(1, 4)`. The random number has an equal likelihood of being generated in each of the following three ranges (because each of the three ranges is the same size and covers the whole range specified by the parameters):

- 1 to 1.999… which will round down to 1,
- 2 to 2.999… which will round down to 2, or
- 3 to 3.999… which will round down to 3.

Input and Output Reference

Introduction: Here we will cover some basic ways to allow a user to interact with your program!

Output: Ways to display something for the user to see.

1) Drawing: You can draw shapes on the canvas and even change what you draw based on conditionals.
2) Printing: Using the commands `print()` and `println()` allows us to print text to the Console, which is the window at the bottom of the Processo IDE. `println()` adds a newline.
3) Text: The command `text("your text",x,y);` will draw "your text" onto the drawing canvas with its *lower-left* corner at position `(x,y)`. The color of the text is set by `fill()`. The size of the text can be adjusted using `textSize()`.

Keyboard Input: Processing always detects when a key on your keyboard is pressed, as long as the drawing canvas window is active. You can use these keys in your program with the following special system variables and functions:

`key` – System variable of type `char` that contains the most recently used key (Unicode).

`keyPressed()` – Function (return type void) that *you define*. Runs once every time a key is **pressed**.

`keyReleased()` – Function (return type void) that *you define*. Runs once every time a key is **released**.

`keyTyped()` – Function (return type void) that *you define*. Runs once every time a key is **pressed**, but ignores special (non-ASCII) keys. Holding it down may repeatedly call `keyTyped`.

Mouse Input: Processing also detects mouse clicks, as long as the click occurred within the bounds of the drawing canvas. You can use these clicks in your program with the following special system variables and functions:

`mouseX` – System variable of type `int` that contains the horizontal coordinate of the mouse in the current frame. There is a similar variable called `mouseY`.

`pmouseX` – System variable of type int that contains the horizontal coordinate of the mouse in the previous frame. There is a similar variable called `pmouseY`.

`mousePressed()` – Function (return type `void`) that *you define*. Runs once every time a mouse button is **pressed**, or your screen is touched if you have a touch screen.

`mouseReleased()` – Function (return type `void`) that *you define*. Runs once every time a mouse button is **released**, or you lift your finger off your touchscreen.

Examples:

```
void keyPressed() {
   if ( key == 'j' ) {
      println("j pressed!");
   } else if (key == 'J') {
      println("J pressed.");
   }
}
```

```
void mousePressed() {
   print("horizontal mouse position = ");
   println(mouseX);
   print("vertical mouse position = ");
   println(mouseY);
}
```

Conditionals Reference

Introduction: A conditional is an expression, often a comparison, that evaluates to `true` or `false`. This value can be stored in a `boolean` variable or used in your program's control flow.

Comparison Operators: These operators are placed in-between two values and return true or false after comparing them. The values can come from evaluating expressions. Parentheses are useful, and sometimes even necessary, to clarify your comparison.

equal to: `==`	greater than: `>`	greater than or equal to: `>=`
not equal to: `!=`	less than: `<`	less than or equal to: `<=`

Examples:
```
boolean b = (2*3) <= max(1,4);            // false
boolean b = ( color(0) != color(1) );     // true
```

Compound Conditionals: We can combine multiple conditionals using special boolean operators. These operators are placed next to boolean values and return true or false.

A	B	A && B (AND)
false	false	false
false	true	false
true	false	false
true	true	true

A	B	A || B (OR)
false	false	false
false	true	true
true	false	true
true	true	true

A	!A (NOT)
false	true
true	false

Examples:
```
boolean b = (x1==x2) && (y1==y2); // is (x1,y1) same pt as (x2,y2)?
boolean b = (z < 1) || (z > 10);  // is z outside the range 1-10?
```

If-Statements: Normally, programs are executed sequentially from top to bottom, also jumping to and from functions as they are called. If-statements give us the ability to choose between statements to execute based on the value returned by a conditional. Code before and after the if-statement executes as usual.

```
if (cond) {
   // this code block executes if cond is true, skipped if cond is false
}
// this code executes after the above (whether the block executed or not)
```

The most general form of an if-statement can optionally include any number of else-if statements and an else statement. Like functions, if-statements don't need semicolons.

```
if (cond1) {
   // this code block executes if cond1 is true
} else if (cond2) {
   // this code block executes if cond1 is false AND cond2 is true
} else {
   // this code block executes if all conditions before this are false
}
```

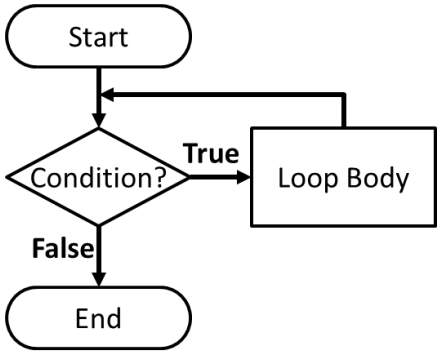
Example:
```
if ( page == 0) { // assume page was declared earlier as an int
   println("This is page 0.");
} else if ( page == 1 ) {
   println("This is page 1.");
}
```

Loops Reference

Introduction: A loop allows us to execute the same block of code multiple times until a specified conditional expression becomes false (i.e. "do something until condition fails"). Similar to multiple function calls, loops tend to be most useful when they are used to execute *similar* (not *identical*) sequences of instructions. You may find it helpful to think of a loop as a *condensed* form of repeated, similar code.

while-loops: This type of loop repeatedly runs the code inside of it as long as a conditional expression is true:

```
while ( condition ) {
    body-statement-1; // while-loop body
    …                 // more body if needed
} // jump back to the top of while loop
```



Notice how the code inside of the loop is contained within curly braces, just like the code in a function! In general, curly braces denote a "block" of code.

Example:
```
int x = 1;
while ( x < 10 ) {
    x = x * 2;
}
```

Iteration	x	Condition (x < 10)	Result
1	1	true	Execute x = 1 * 2;
2	2	true	Execute x = 2 * 2;
3	4	true	Execute x = 4 * 2;
4	8	true	Execute x = 8 * 2;
5	16	false	Exit loop

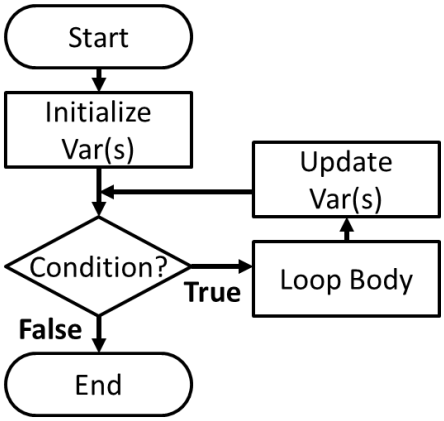
The above loop will execute the statement `x = x * 2` four times, with the final value of x = 16, as seen in the table to the right.

For-loops: These are very similar to while-loops, but they allow you to specify additional initialization and update statements, which are separated by semicolons ;
The following side-by-side code segments are equivalent:

```
for (init; condition; update) {
  body;
}
```

⇔

```
init;
while (condition) {
  body;
  update;
}
```



Examples:
```
int x;
for ( x = 1; x < 10; x = x * 2 ) { }
```

This (empty body!) for-loop is equivalent to the while-loop example: we're able to eliminate the entire body of the loop because it's now executed as the update statement!

Note that if you *declare* a variable in the init statement, then that variable is *local* to the body of the for-loop, similar to parameters being local to a function body. (For int, the variable name i is often used.)

```
int sum = 0;
for ( int i = 1; i <= 6; i = i + 1 ) {
  sum = sum + i;
}
```

Iteration	i	sum	Condition (i <= 6)	Result
1	1	0	true	Execute sum = 0 + 1;
2	2	1	true	Execute sum = 1 + 2;
3	3	3	true	Execute sum = 3 + 3;
4	4	6	true	Execute sum = 6 + 4;
5	5	10	true	Execute sum = 10 + 5;
6	6	15	true	Execute sum = 15 + 6;
7	7	21	false	Exit loop

The above loop will sum the numbers from 1 to 6, with the final value of sum = 21, as seen in the table to the right.

Arrays Reference

Introduction: An array is a common data structure: a way of efficiently organizing and storing related data. An array groups a set of pieces of data of the same variable type under a common name, with individual elements referenced by a numbered index. In lecture, we used buckets to represent the indices of the array and colored ping pong balls to represent the values stored in each index.

Arrays are useful in place of declaring many individual variables that will be used for similar purposes (e.g. the position variables from Lego Family). Arrays can be used and manipulated easily in conjunction with loops.

Array Declaration: Just like with variables, you must declare an array before you can use it. But unlike variables, array declarations come in two separate parts:

(1) Declaring the array variable is of the form: `var_type[] array_name;`

- `var_type` declares that your array will contain values of the specified data type.
- `array_name` is the name associated with the array. You should try to use an intuitive name that indicates what the array is used for. Has the same naming restrictions as other variables.

(2) Creating the array itself is of the form: `new var_type[len];`

- `var_type` sets the data type/container mold of every index and must match that of your array variable.
- `len` is the number of indices that Processo will create in your array (i.e. its length). Note that we always start counting indices from 0, so for `len` indices, they will be referenced as `0, 1, ..., len-1`.

Examples:

```
color[] myColors;            // declares array variable; no array created
int[] familyX = new int[5];  // creates array for x-positions of the
                             // 5 Simpsons characters
```

Array Initialization: Arrays differ from variables in that Processo will automatically initialize array values when you create a new array. The default value in an array depends on the data type, but roughly equates to “zero” in the various contexts: 0 (`int`), 0.0 (`double`), black (`color`), `false` (`boolean`).

Alternatively, you can initialize an array to chosen values using curly bracket ({ }) notation, with values separated by commas. Every value must match the variable type of your array variable and the size of the array will depend on how many commas you use.

Examples:

```
color[] myColors = new color[3];         // all indices contain black
int[] familyX = {1, 100, 5, 200, 400};   // inferred array size of 5
```

Using Arrays: To use the values in an array, we use the array variable name along with `[brackets]` to indicate which index. The index can be specified using a hard-coded int value or from the evaluation of an int expression. As a reminder, array indices start at `0` and go to `len-1` (from left to right). The length of an array can be found using the special notation: `array_name.length`

Examples:

```
background( myColors[0] );          // set to the color in index 0
familyX[familyX[0]] = familyX[4];   // set familyX[1] to 400
int len = familyX.length;           // 5 based on initialization above
familyX = new int[]{5, -10, 0};     // create another new array with values
```

Strings Reference

Introduction: The `String` data type is used to store sequences of characters, which we commonly think of as "text." Strings share some similarities to arrays of `char`s but with special properties and restrictions.

String Literals: Arbitrary strings created by placing text between double-quotes. We've already been using these in the course! Notice that for a known string that you are only going to use once (e.g. as an argument to `println()` or `text()`), you can use a string literal without creating a variable!

Examples:
```
String s = "stored text!";   // string literal stored in variable s
text(s,10,50);               // draws string on canvas
text("stored text!",10,50);  // same result as the statement above
```

Using Strings: You are provided with a number of useful ways to use and manipulate strings:

`s.length()` – Returns the length of the string (i.e. the number of characters).

`s.charAt(i)` – Returns the `char` at the index i, where the indices start from the left with index 0.

`s.substring(i,j)` – Returns a new String including the characters between index i and j (**not including j**)

`s1 + s2` – The plus (+) operator concatenates the strings `s1` and `s2` together. That is, it returns a new `String` that is the combination of `s1` and `s2`, with the characters in `s1` coming *before* (to the left of) the characters in `s2`. If one of the two isn't a string, it converts the other one into a string representation, so `"1"+2` evaluates to `"12"`.

`s1.equals(s2)` – Returns true if the contents of the two strings match (i.e. they are the same length and have matching characters in every index) and false otherwise. `s2.equals(s1)` is equivalent. *Characters with different case are not equal* (e.g. `"a".equals("A")` is false). **Using == is not allowed!** (This is because other programming languages use == to check to see if they are the same memory storage object. Very tricky!)

Examples:
```
String s1 = "ate", s2 = "con", s3 = "nation";  // create 3 strings
int i = s3.length();                           // stores 6
char c = s2.charAt(0);                         // stores 'c'
boolean bl = sl.equals(s2);                    // stores false
boolean b2 = s2.equals("con");                 // stores true
println( s2 + s2.charAt(0) + sl + s3 );        // prints "concatenation"
println((s2.charAt(0) + "at").equals("cat"));  // prints true
```

String and Array Comparison: These are conceptually similar, so it is easy to confuse them. We hope this comparison will help you recognize the differences in syntaxes between the two.

Command	String Syntax	(Character) Array Syntax
Declaration	`String s;`	`char[] ar;`
Assignment	`s = "call";`	`ar = new char[]{'c','a','l','l'};`
Get length (`int`)	`s.length()`	`ar.length`
Get character i (`char`)	`s.charAt(i)`	`ar[i]`
Change character i	Not allowed	`ar[i] = 'w';`
Concatenation	`s = s + " me, maybe?";`	Requires creating a new array and using loops

Files Reference

Introduction: Files are useful for reading and writing data because they exist *outside* of your programs. This allows for lots of different possibilities: (1) you can store the result of your program execution somewhere more permanent, (2) you can edit the data values between program executions, (3) you can pass data in files between *different* programs, or (4) you can change the *amount* of data your program reads by modifying the file contents and length.

Importing a File: There are many file formats but, in this course, we will use data files in comma-separated values (CSV) format. The simplest way to import these kinds of files is to call the special function `loadStrings(String filename)` and store its return value into a `String` array. Each *line* of the file will be stored in a different index of the array (i.e. the 1st line will be in index 0, the 2nd line in index 1, etc.) as a String.

- It is easiest if you put your CSV file into your Processo project folder and then you can just use the filename as the argument.
- Files should be imported *once* at the beginning of your program in the setup code.
- The `filename` must be a URL on the Internet, a Canvas course file, or internal browser storage (below).

As the name implies, each row/line of a CSV file contains values with columns separated by commas. So we will want to *split* each row into its values using the function `split(String s, char delim)`. This function breaks `s` into pieces (returns a `String[]`) using `delim` as the delimiter, a boundary marker between values.

- Note that `split()` takes a `String`, not a `String[]`, so it should be used on a *row* of imported data, not the whole imported file.

Example:

```
String[] importedData, header;
void setup() {
  importedData = loadStrings("data.csv");
  header = split(importedData[0], ",");    // split header/1st row
}
```

Converting Data: `loadStrings()` imports your CSV file as a `String` array and `split()` returns the values in a row in a `String[]` as well. However, if the file was not intended to be text, you will need to first convert the data before you use it. Luckily, Processo has a handy set of *conversion* functions that will do this for you! These conversion functions are intuitively named: `toInt()`, `toDouble()`, and `toString()`.

Example:

```
String row = "120,3.14,hi";
String[] vals = split(row,",");  // split into array of Strings
int i = toInt(vals[0]);          // stores 120
double f = toDouble(vals[1]);    // stores 3.14
String s = vals[2];              // stores "hi" - no conversion needed
```

Exporting to a File: To save data to a file, we can use `saveStrings(String filename, String[] data)`. This will put the file in internal browser storage. For CSV files, `filename` should end in `.csv` and data should be an array of Strings, each using commas as delimiters. Each index of `data` is written into the file as a separate line/row. Note that we **separate** values with commas below.

Example:

```
int[] row1 = {1, 20, 120, -5};
double[] row2 = {0.33, 1.41, 1.62, 2.71, 3.14};
String[] data = {toString(row1[0]), toString(row2[0])}; // initial data
for (int i = 1; i < row1.length; i = i+1 ) {   // skip 1st entry
  data[0] = data[0] + "," + toString(row1[i]); // add commas and cols
}
for (int i = 1; i < row2.length; i = i+1 ) {   // skip 1st entry
  data[1] = data[1] + "," + toString(row2[i]); // add commas and cols
}
saveStrings("myData.csv", data);
```