Intro to Processing

GIRLSCRIPT

Week 1 - Fundamentals

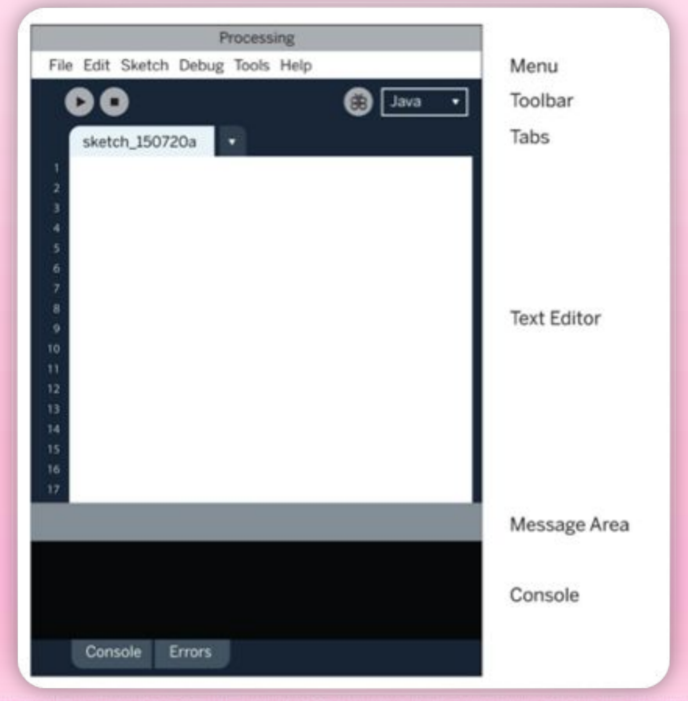
By Nicole McCabe

What will we Learn?

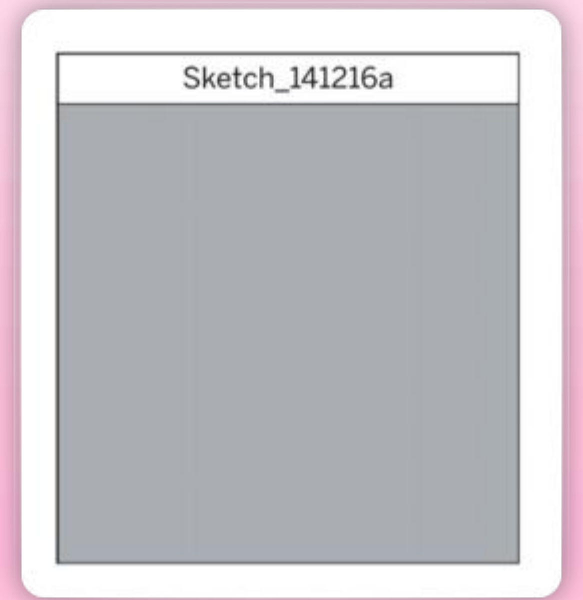
- The Development Environment (IDE)
- Functions
- Variables
- Math
- Comments
- Conditional Statements (If Else)
- Loops (Iteration)
- Images
- Arrays

The Development Environment

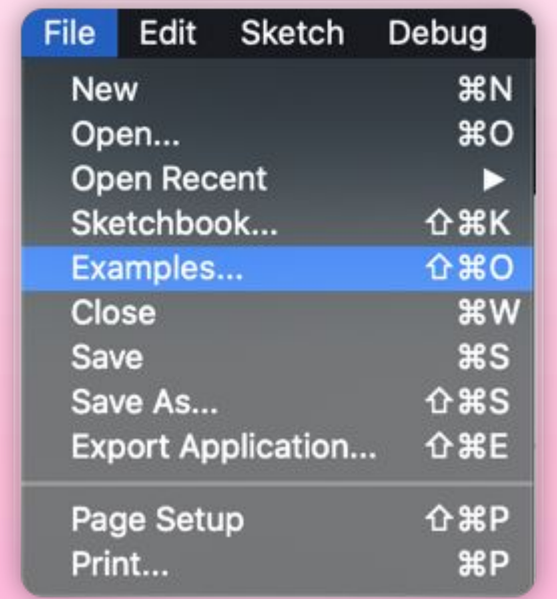
The Development Environment

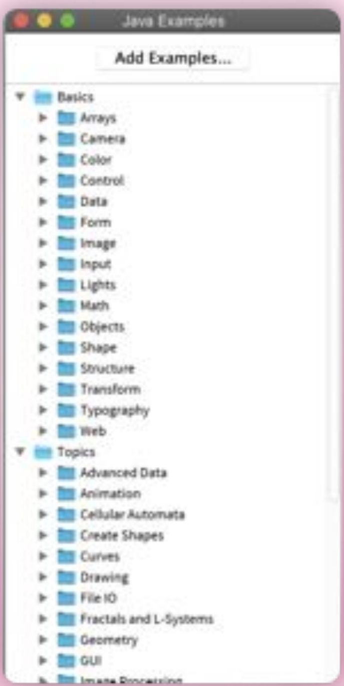


The window that pops up is called a sketch



Processing also comes with Examples!





Processing Coordinate System

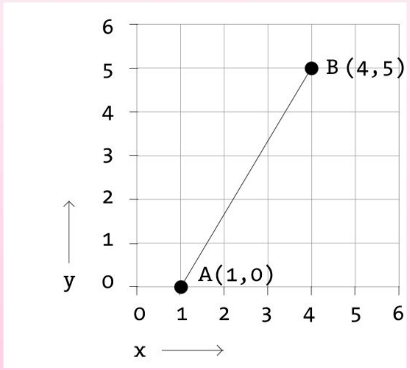
———

The computer screen is a grid of light elements called **pixels.** Each pixel has a position within the grid defined by coordinates.

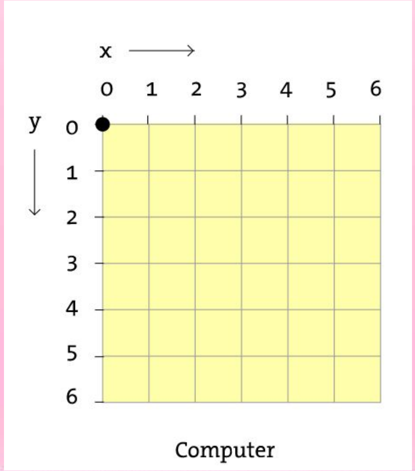
In Processing, the x coordinate is the distance from the **left edge** of the Display Window and the y coordinate is the distance from the top edge.

We write coordinates of a pixel like this: (x, y).

So, if the screen is 200×200 pixels, the upper-left is (0, 0), the center is at (100, 100), and the lower-right is (199, 199). Reference



cartesian coordinates



Processing

Functions

What is a Function?

Functions are the basic building blocks for Processing programs.

A computer runs a program one line at a time. But when a function is run, the computer jumps to where the function is defined and runs the code there, then jumps back to where it left off.

The power of functions is **modularity.** Functions are independent software units that are used to build more complex programs — like LEGO bricks, where each type of brick serves a specific purpose, and making a complex model requires using the different parts together. As with functions, the true power of these bricks is the ability to build many different forms from the same set of elements. The same group of LEGOs that makes a spaceship can be reused to construct a truck, a skyscraper, and many other objects.

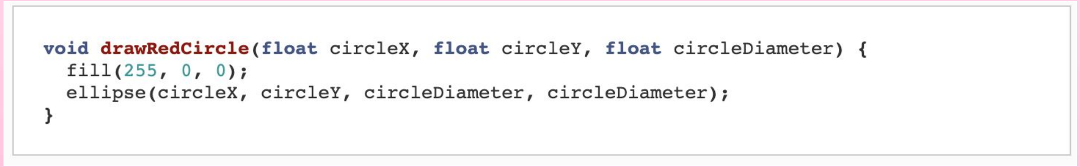
Another purpose of functions is r**eusability.** Once a function is defined, the code inside the function need not be repeated again.

Creating Your Own Function

To create your own function, you need to do four things:

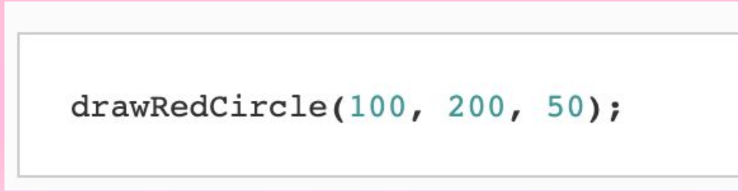
- Write the **return type** of the function (void, int, float etc.)
- Write the **name** of the function.
- Inside parenthesis (), list any **parameters** (input) the function takes.
- Inside curly brackets {}, write the code that will run whenever the function is called. This is called the **body** of the function.

This function has a **void** return type (which means it does something instead of giving you a value), and takes three parameters: circleX, circleY, and circleDiameter. The **body** of the function changes the fill color to red and then uses the parameters to draw a circle.



```
void drawRedCircle(float circleX, float circleY, float circleDiameter) {
  fill(255, 0, 0);
  ellipse(circleX, circleY, circleDiameter, circleDiameter);
}
```

To call this function, you'd use its name and give it parameters, exactly like you've been calling other functions:



```
drawRedCircle(100, 200, 50);
```

size() - Change Size of Display Window

size() is an example of a **built in function,** it is already created in Processing so you don't have to create it.

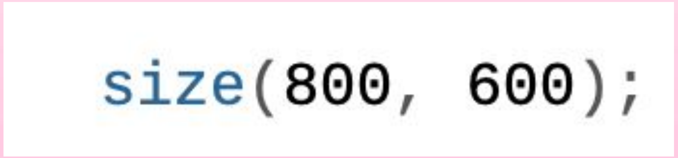
The **size()** function is to set the width and height of the Display Window.

If your program doesn't have a **size()** function, the dimension is set to 100×100 pixels.

It has two **parameters** (input):

- the first sets the width of the window
- the second sets the height.

To draw a window that is 800 pixels wide and 600 high, type:



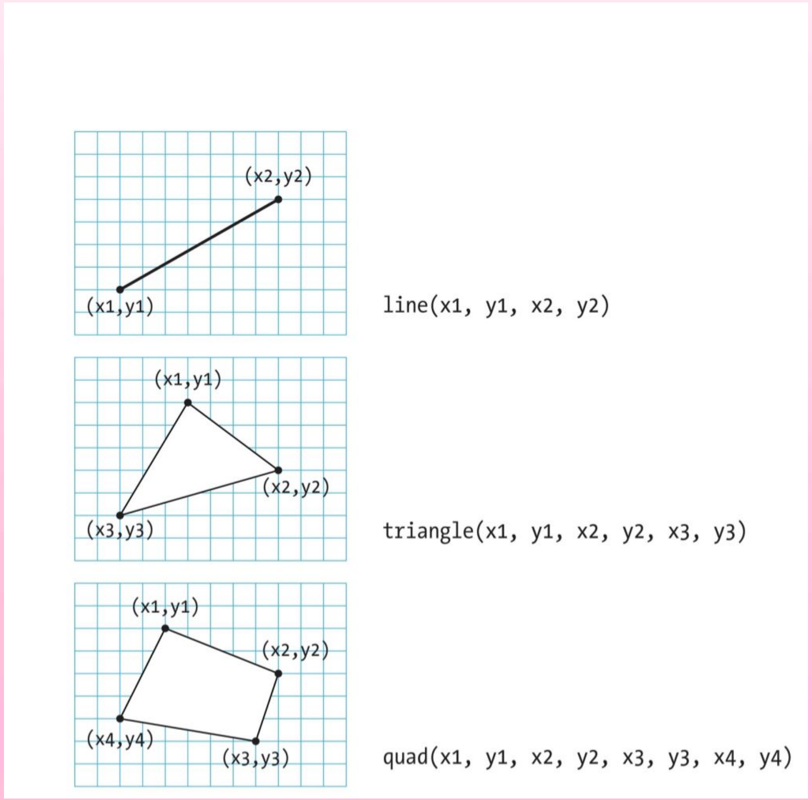
```
size(800, 600);
```

Basics Shapes

Processing includes a group of in built functions to draw basic shapes.

Simple shapes like lines can be combined to create more complex forms like a leaf or a face.

These shapes can also be drawn partially (or entirely) out of the window without an error.

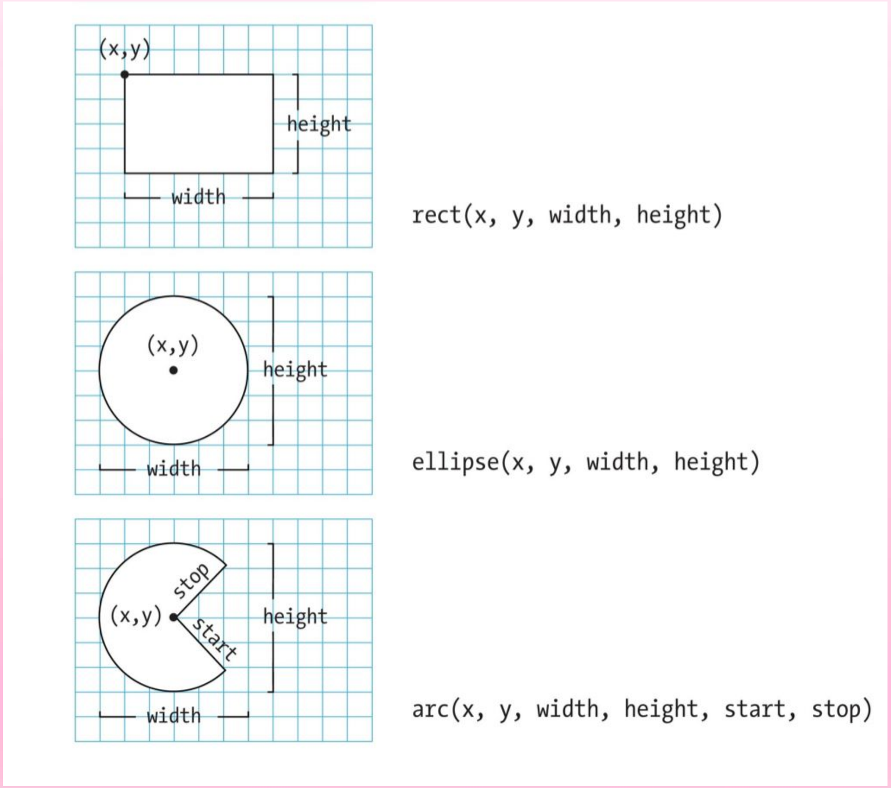


```
line(x1, y1, x2, y2)
triangle(x1, y1, x2, y2, x3, y3)
quad(x1, y1, x2, y2, x3, y3, x4, y4)
```

More Basic Shapes

Processing doesn't have separate functions to make squares and circles. To make these shapes, use the same value for the width and the height parameters to **ellipse()** and **rect().**

When a program runs, the computer starts at the top and reads each line of code until it reaches the last line and then stops. If you want a shape to be drawn on top of all other shapes, it needs to follow the others in the code.


(x,y)
height
width
rect(x, y, width, height)
(x,y)
height
width
ellipse(x, y, width, height)
stop
(x,y)
start
height
width
arc(x, y, width, height, start, stop)

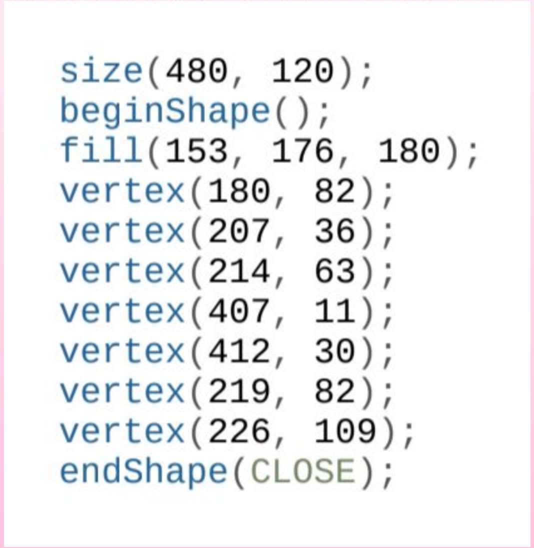
Custom Shapes

You're not limited to using the previous basic shapes — you can also define new shapes by connecting a series of points.

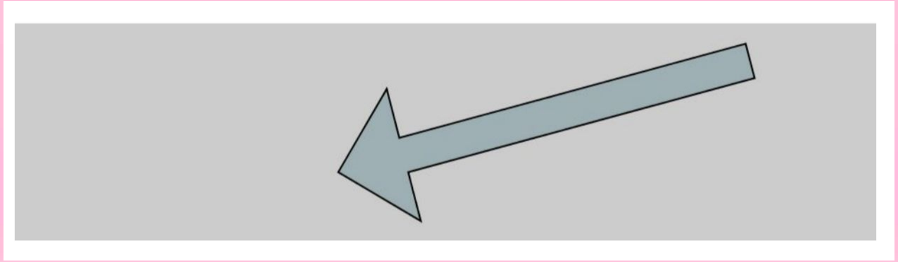
The **beginShape()** function signals the start of a new shape.

The **vertex()** function is used to define each pair of x and y coordinates for the shape.

Finally, **endShape()** is called to signal that the shape is finished.



```
size(480, 120);
beginShape();
fill(153, 176, 180);
vertex(180, 82);
vertex(207, 36);
vertex(214, 63);
vertex(407, 11);
vertex(412, 30);
vertex(219, 82);
vertex(226, 109);
endShape(CLOSE);
```

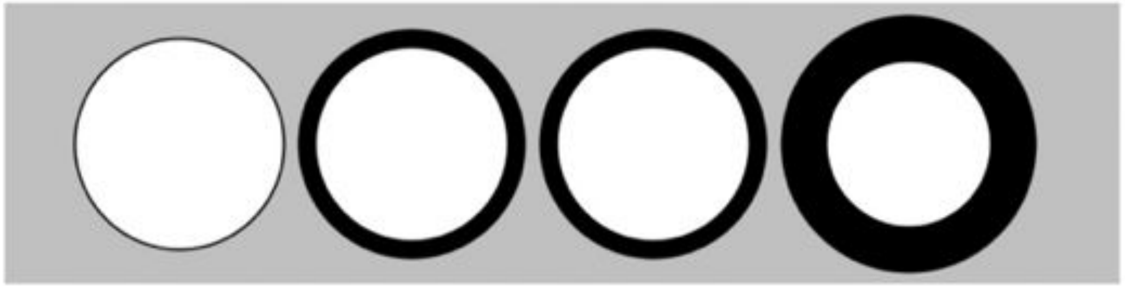


strokeWeight()

The stroke is the outline of the shape.

The default stroke weight is a single pixel, but this can be changed with the **strokeWeight()** function.

The single parameter to **strokeWeight()** sets the width of drawn lines.

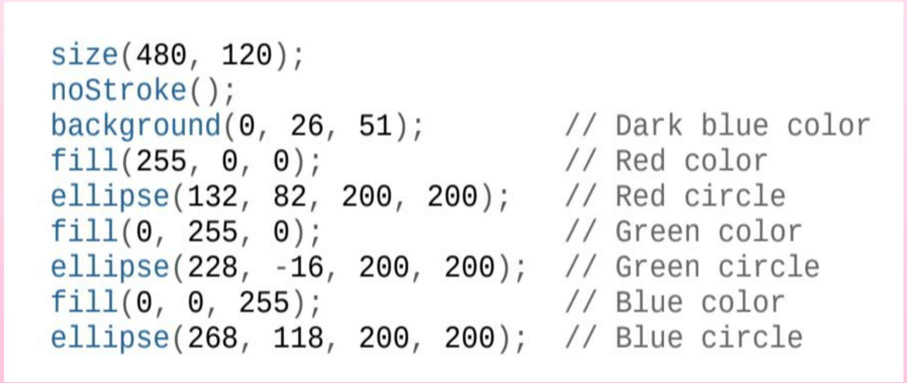


```
size(480, 120);
ellipse(75, 60, 90, 90);
strokeWeight(8);  // Stroke weight to 8 pixels
ellipse(175, 60, 90, 90);
ellipse(279, 60, 90, 90);
strokeWeight(20);  // Stroke weight to 20 pixels
ellipse(389, 60, 90, 90);
```

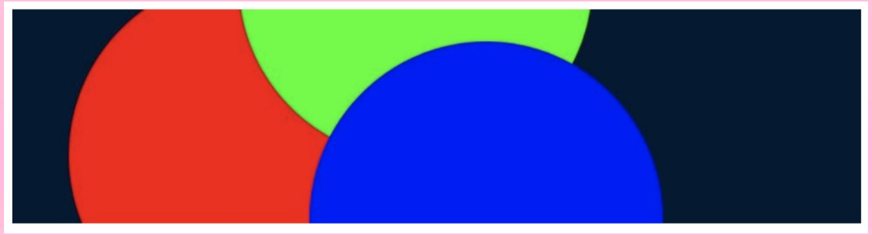
Adding Color: stroke(), fill(), background()

- **stroke()** - Sets the color used to draw lines and borders around shapes.
- **fill()** - Sets the color used to fill shapes
- **background()** - Sets the color used for the background for the Processing window.

The default color space is RGB, with each value in the range from 0 to 255. RGB Reference



```
size(480, 120);
noStroke();
background(0, 26, 51);          // Dark blue color
fill(255, 0, 0);                // Red color
ellipse(132, 82, 200, 200);     // Red circle
fill(0, 255, 0);                // Green color
ellipse(228, -16, 200, 200);    // Green circle
fill(0, 0, 255);                // Blue color
ellipse(268, 118, 200, 200);    // Blue circle
```

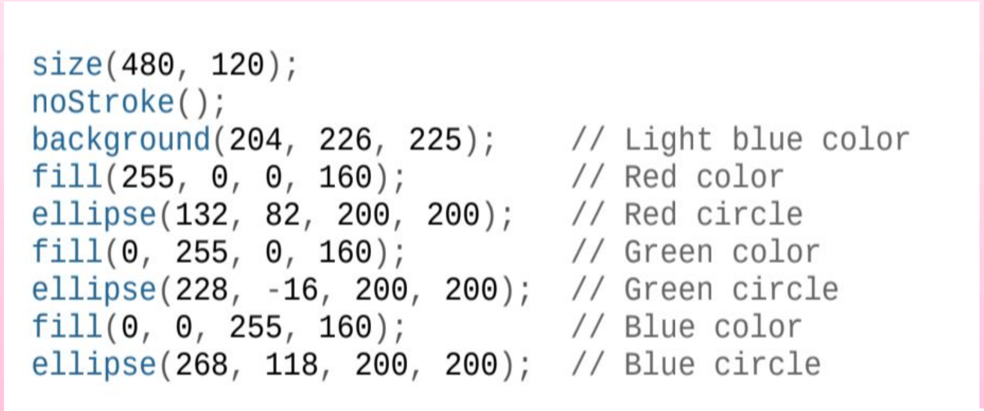


Transparency

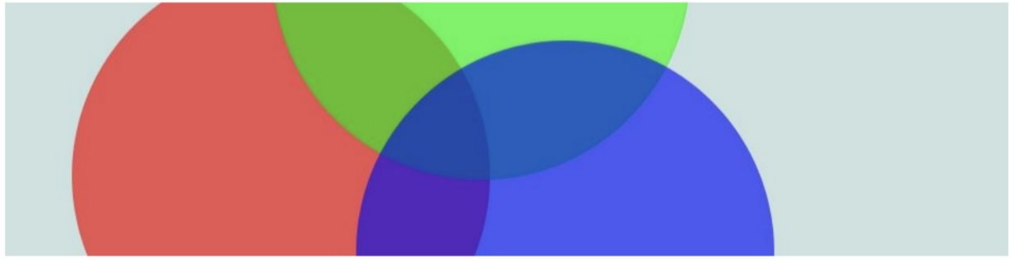
By adding an optional fourth parameter to **fill()** or **stroke()**, you can control the transparency.

This fourth parameter is known as the alpha value, and also uses the range 0 to 255 to set the amount of transparency.

The value 0 defines the color as entirely transparent (it won't display), the value 255 is entirely opaque, and the values between these extremes cause the colors to mix on screen



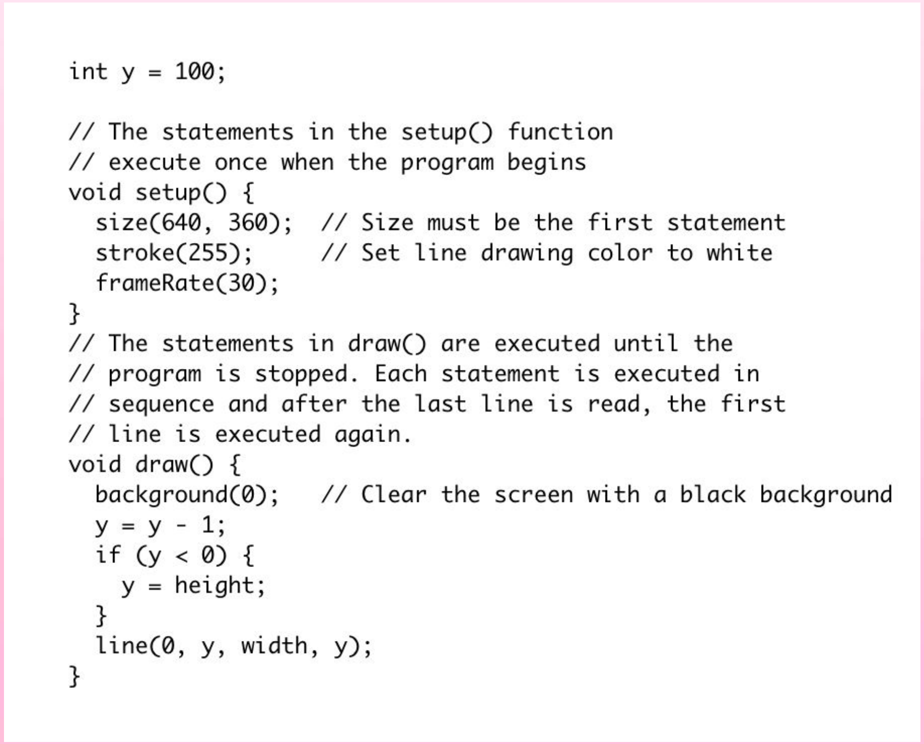
```
size(480, 120);
noStroke();
background(204, 226, 225);    // Light blue color
fill(255, 0, 0, 160);         // Red color
ellipse(132, 82, 200, 200);   // Red circle
fill(0, 255, 0, 160);         // Green color
ellipse(228, -16, 200, 200);  // Green circle
fill(0, 0, 255, 160);         // Blue color
ellipse(268, 118, 200, 200);  // Blue circle
```



setup() and draw()

The **setup()** function is run once, when the program starts. It's used to define initial environment properties such as screen size and to load media such as images and fonts as the program starts. There can only be one **setup()** function for each program and it shouldn't be called again after its initial execution.

Called directly after **setup()**, the **draw()** function continuously executes the lines of code contained inside its block until the program is stopped. If you want animation, use the **draw()** function.



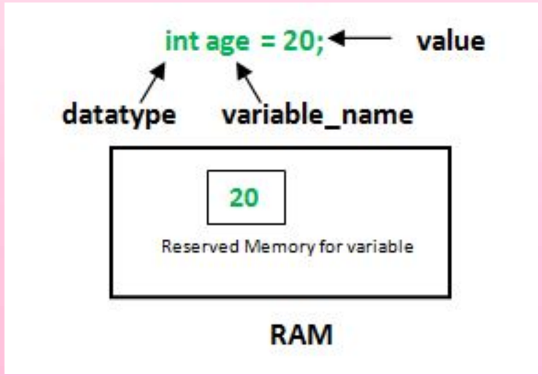
```
int y = 100;

// The statements in the setup() function
// execute once when the program begins
void setup() {
  size(640, 360);  // Size must be the first statement
  stroke(255);     // Set line drawing color to white
  frameRate(30);
}
// The statements in draw() are executed until the
// program is stopped. Each statement is executed in
// sequence and after the last line is read, the first
// line is executed again.
void draw() {
  background(0);   // Clear the screen with a black background
  y = y - 1;
  if (y < 0) {
    y = height;
  }
  line(0, y, width, y);
}
```

Variables

What is a Variable?

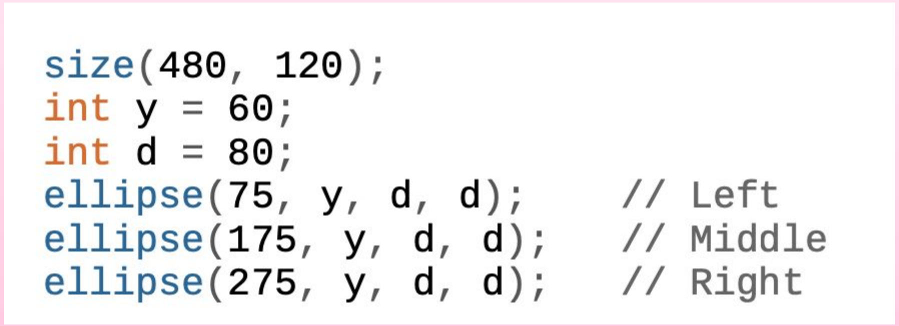
- A variable stores a **value in memory (RAM)** so that it can be used later in a program.
- The variable can be **used many times** within a single program, and the value is easily changed while the program is running.
- One of the reasons we use variables is to avoid repeating ourselves in the code.
- If you are typing the same number more than once, consider using a variable instead so that your code is more general and easier to update.



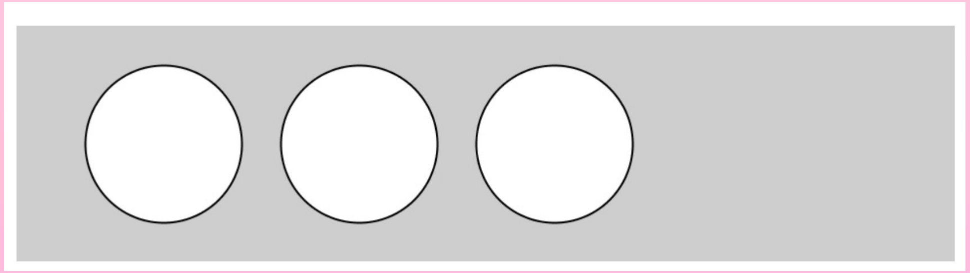
Variable Example

For instance, when you make the *y* coordinate and diameter for the three circles in this example into variables, the same values are used for each ellipse.

Let's say that you wanted to change the y coordinate and diameter, without the variables, you'd need to change the *y* coordinate used in the code three times and the diameter six times.



```
size(480, 120);
int y = 60;
int d = 80;
ellipse(75, y, d, d);     // Left
ellipse(175, y, d, d);    // Middle
ellipse(275, y, d, d);    // Right
```

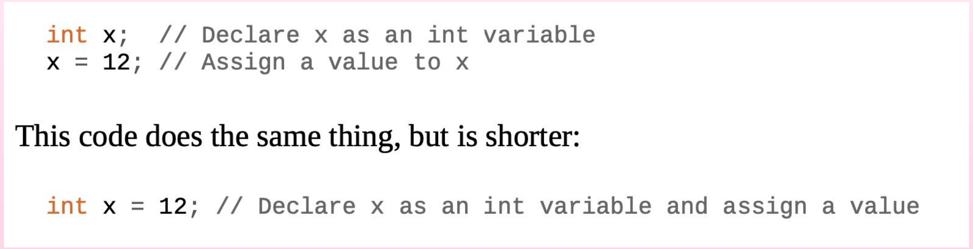


Creating a Variable

When you make your own variables, you determine the *name*, the *data type*, and the *value*. The name is what you decide to call the variable.

The range of values that can be stored within a variable is defined by its *data type*. For instance, the *integer* data type can store numbers without decimal places (whole numbers).

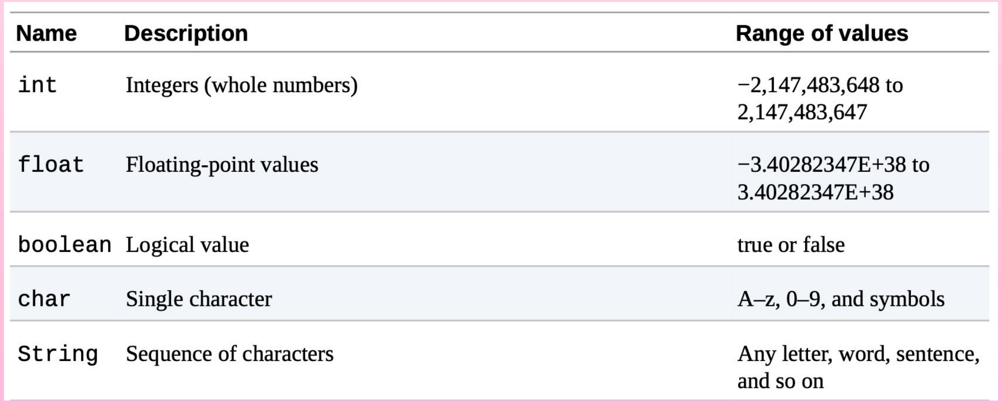
After the data type and name are set, a value can be assigned to the variable:



```
int x;  // Declare x as an int variable
x = 12; // Assign a value to x
```

This code does the same thing, but is shorter:

```
int x = 12; // Declare x as an int variable and assign a value
```



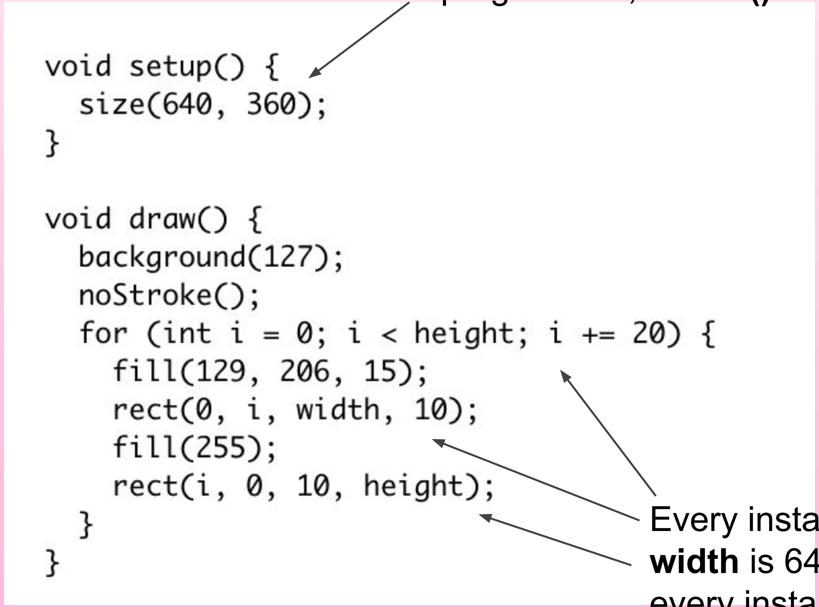
Name	Description	Range of values
`int`	Integers (whole numbers)	−2,147,483,648 to 2,147,483,647
`float`	Floating-point values	−3.40282347E+38 to 3.40282347E+38
`boolean`	Logical value	true or false
`char`	Single character	A–z, 0–9, and symbols
`String`	Sequence of characters	Any letter, word, sentence, and so on

width and height Variables

The '**width**' and '**height**' variables contain the width and height of the display window as defined in the **size()** function.

These variables are already built into Processing and are examples of **built in variables.**

The width and height of the window is defined by the programmer, in **size()**



```
void setup() {
  size(640, 360);
}

void draw() {
  background(127);
  noStroke();
  for (int i = 0; i < height; i += 20) {
    fill(129, 206, 15);
    rect(0, i, width, 10);
    fill(255);
    rect(i, 0, 10, height);
  }
}
```

Every instance of **width** is 640 and every instance if height is **350.**

mouseX and mouseY Variables: Track Your Mouse Movements

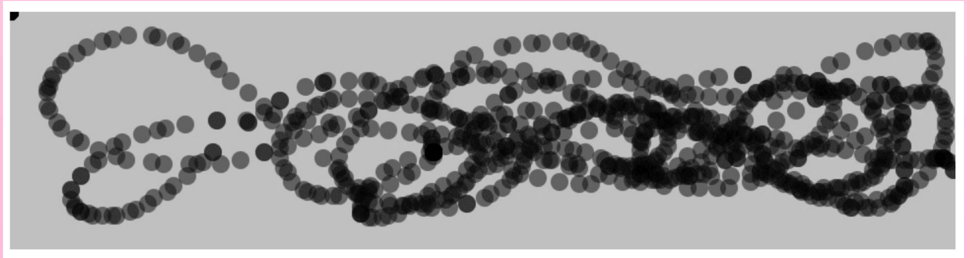
The **mouseX** variable stores the *x* coordinate, and the **mouseY** variable stores the *y* coordinate.

These are built in variables that can be used to track the mouse position and use those numbers to move elements on screen.

In this example, each time the code in the **draw()** block is run, a new circle is drawn to the window. This image was made by moving the mouse around to control the circle's location.

```
void setup() {
  size(480, 120);
  fill(0, 102);
  noStroke();
}

void draw() {
  ellipse(mouseX, mouseY, 9, 9);
}
```



Math

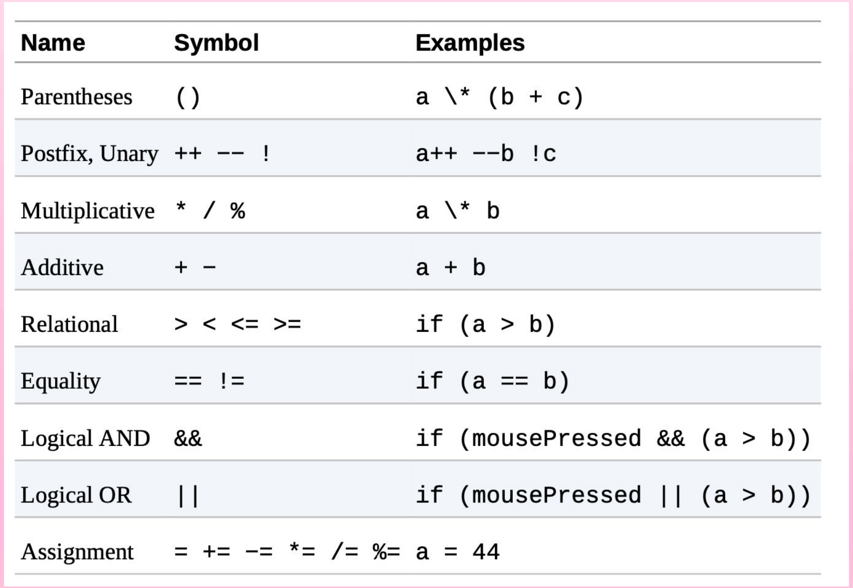
Order of Operations

When mathematical calculations are performed in a program, each operation takes place according to a pre-specified order.

This **order of operations** ensures that the code is run the same way every time.

This is no different from arithmetic or algebra, but programming has other operators that are less familiar.

In the following table, the operators on the top are run before those on the bottom — therefore, an operation inside parentheses will run first and an assignment will run last:



Name	Symbol	Examples
Parentheses	`()`	`a \* (b + c)`
Postfix, Unary	`++ -- !`	`a++ --b !c`
Multiplicative	`* / %`	`a \* b`
Additive	`+ -`	`a + b`
Relational	`> < <= >=`	`if (a > b)`
Equality	`== !=`	`if (a == b)`
Logical AND	`&&`	`if (mousePressed && (a > b))`
Logical OR	`||`	`if (mousePressed || (a > b))`
Assignment	`= += -= *= /= %=`	`a = 44`

Comments

What are Comments?

- Comments are notes that you write to yourself (or other people) inside the code.
- They are ignored when the program is run.
- If others are reading your code, comments are especially important to help them understand your thought process.

A comment starts with two forward slashes (//) and continues until the end of the line:

```
// This is a one-line comment
```

You can make a multiple-line comment by starting with /* and ending with */. For instance:

```
/* This comment
   continues for more
   than one line
*/
```

Conditional Statements (If Else)

What are Conditional Statements?

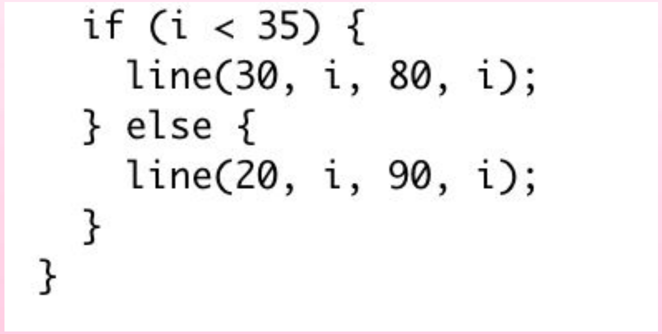
Conditions are like questions. They allow a program to decide to take one action if the answer to a question is "true" or to do another action if the answer to the question is "false."

A **conditional statement** is a set of rules performed if a certain condition is met. It is sometimes referred to as an **If-Else** statement, because IF a condition is met then an action is performed, ELSE if the condition is false then this action is performed.

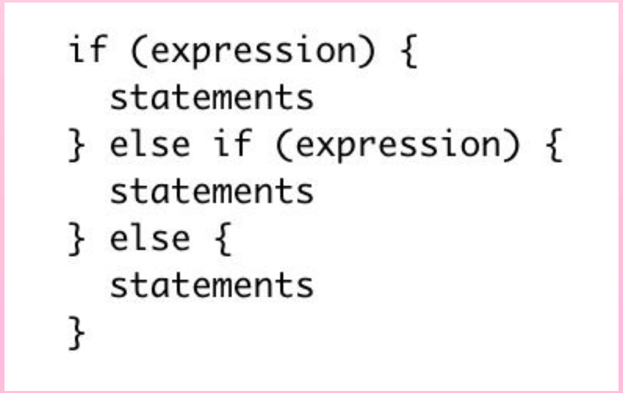
Creating an If Else Statement

If the test evaluates to true, the statements enclosed within the if block are executed and if the test evaluates to false the statements are not executed.

Else extends the if structure allowing the program to choose between two or more blocks of code. It specifies a block of code to execute when the expression in if is false.

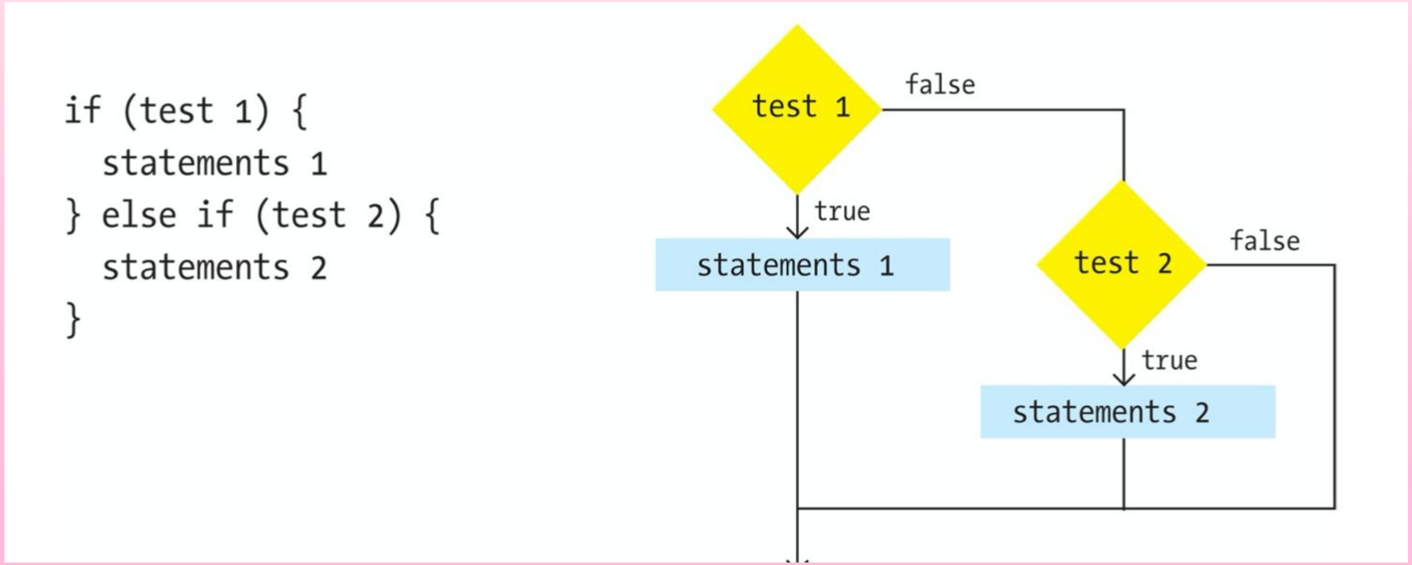


```
  if (i < 35) {
    line(30, i, 80, i);
  } else {
    line(20, i, 90, i);
  }
}
```



```
if (expression) {
  statements
} else if (expression) {
  statements
} else {
  statements
}
```

If Else Statement Diagram

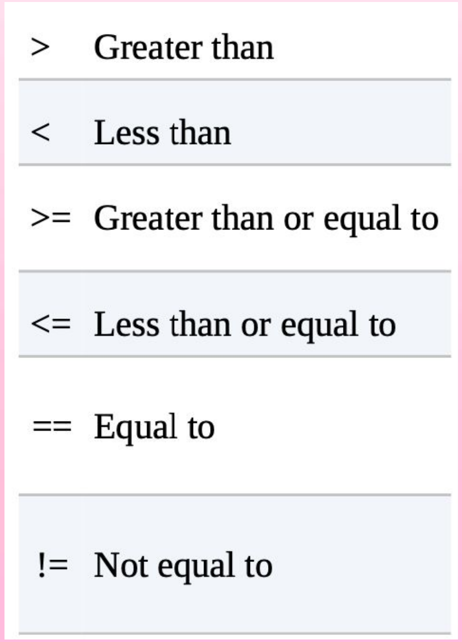


```
if (test 1) {
  statements 1
} else if (test 2) {
  statements 2
}
```

Loops (Iteration)

What is a Loop?

- As you write more programs, you'll notice that patterns occur when lines of code are repeated, but with slight variations.
- A code structure called a *loop* makes it possible to run a line of code more than once to condense this type of repetition into fewer lines.
- This makes your programs more modular and easier to change.



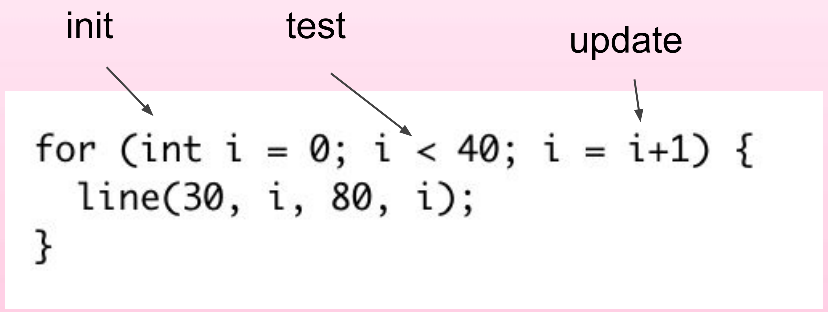
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to
==	Equal to
!=	Not equal to

There are two types of loops: **For** and **While**

For Loop

A basic for structure has three parts: **init, test,** and **update**. Each part must be separated by a **semicolon (;)**. The loop continues until the test evaluates to **false.** When a for structure is executed, the following sequence of events occurs:

1. The init statement is run.
2. The test is evaluated to be true or false.
3. If the test is true, jump to step 4. If the test is false, jump to step 6.
4. Run the statements within the block.
5. Run the update statement and jump to step 2.
6. Exit the loop



```
for (int i = 0; i < 40; i = i+1) {
  line(30, i, 80, i);
}
```

In the example above, the for structure is executed 40 times. In the init statement, the value i is created and set to zero. i is less than 40, so the test evaluates as true. At the end of each loop, i is incremented by one. On the 41st execution, the test is evaluated as false, because i is then equal to 40, so i < 40 is no longer true. Thus, the loop exits.

Example of For Loop

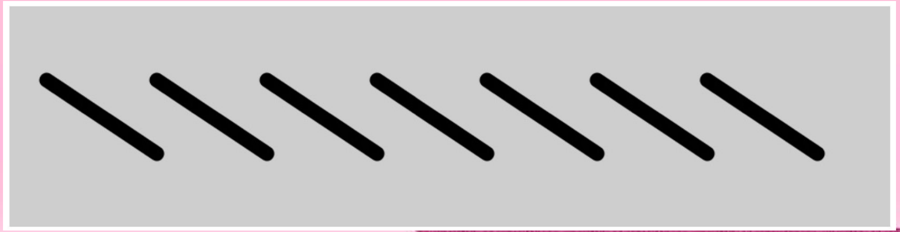
Example with For Loop

```
size(480, 120);
strokeWeight(8);
for (int i = 20; i < 400; i += 60) {
  line(i, 40, i + 60, 80);
}
```

Example with no loop

```
size(480, 120);
strokeWeight(8);
line(20, 40, 80, 80);
line(80, 40, 140, 80);
line(140, 40, 200, 80);
line(200, 40, 260, 80);
line(260, 40, 320, 80);
line(320, 40, 380, 80);
line(380, 40, 440, 80);
```

Both codes produce the same output but the example with the loop is less tedious to write

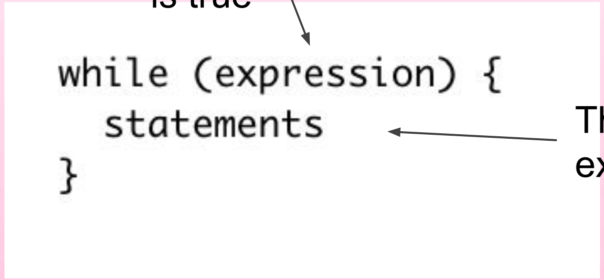


While Loop

The while structure executes a series of statements continuously while the expression is **true.**

The expression must be updated during the repetitions or the program will never "break out" of while.

As long as this is true

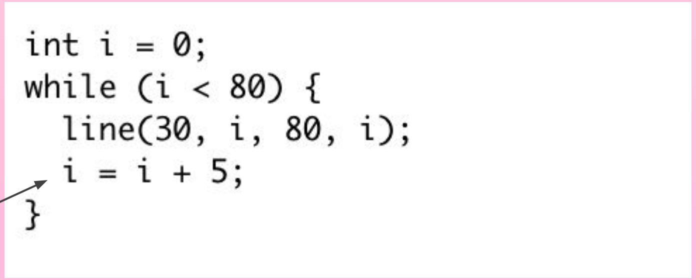


```
while (expression) {
  statements
}
```

This code is executed

As long as the variable i is less than 80, the line is drawn

the variable is incremented by 5 each time a line is drawn until eventually i becomes 80 and the statement becomes false



```
int i = 0;
while (i < 80) {
  line(30, i, 80, i);
  i = i + 5;
}
```

Images

How to Add an Image

There are three steps to follow before you can draw an image to the screen:

1. Add the image to the the same folder as your code.

2. Create a **PImage** variable to store the image.

3. Load the image into the variable with **loadImage().**

4. After all three steps are done, you can draw the image to the screen with the **image()** function.

The first parameter to **image()** specifies the image to draw; the second and third set the *x* and *y* coordinates.

```
PImage img;

void setup() {
  size(480, 120);
  img = loadImage("lunar.jpg");
}

void draw() {
  image(img, 0, 0);
}
```

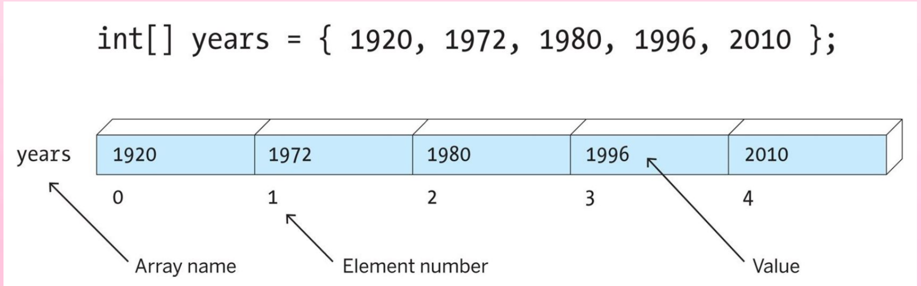
Arrays

What is an Array?

Each item in an array is called an **element**, and each has an **index** value to mark its position within the array.

Imagine what would happen if you wanted to have 3,000 circles. This would mean creating 3,000 individual variables, then updating each one separately. Could you keep track of that many variables? Would you want to? Instead, we use an array.

An *array* is a list of variables that share a common name. Arrays are useful because they make it possible to work with more variables without creating a new name for each. This makes the code shorter, easier to read, and more convenient to update.



```
int[] years = { 1920, 1972, 1980, 1996, 2010 };
```

Figure 11-1. An array is a list of one or more variables that share the same name

Just like coordinates on the screen, index values for an array start counting from 0. For instance, the first element in the array has the index value 0, the second element in the array has the index value 1, and so on. If there are 20 values in the array, the index value of the last element is 19.

Example of an Array

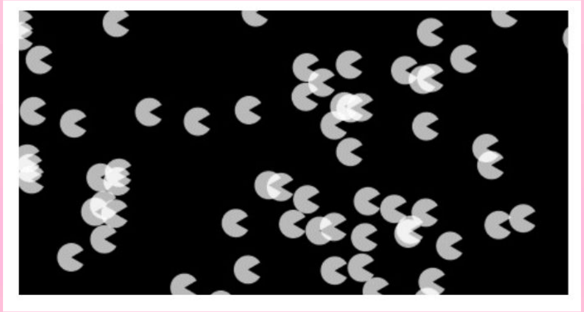
With Array

```
float[] x = new float[3000];

void setup() {
  size(240, 120);
  noStroke();
  fill(255, 200);
  for (int i = 0; i < x.length; i++) {
    x[i] = random(-1000, 200);
  }
}

void draw() {
  background(0);
  for (int i = 0; i < x.length; i++) {
    x[i] += 0.5;
    float y = i * 0.4;
    arc(x[i], y, 12, 12, 0.52, 5.76);
  }
}
```

Without an array. There are too many variables and the code is out of control.



```
float x1 = -10;
float x2 = 10;
float x3 = 35;
float x4 = 18;
float x5 = 30;

void setup() {
  size(240, 120);
  noStroke();
}

void draw() {
  background(0);
  x1 += 0.5;
  x2 += 0.5;
  x3 += 0.5;
  x4 += 0.5;
  x5 += 0.5;
  arc(x1, 20, 20, 20, 0.52, 5.76);
  arc(x2, 40, 20, 20, 0.52, 5.76);
  arc(x3, 60, 20, 20, 0.52, 5.76);
  arc(x4, 80, 20, 20, 0.52, 5.76);
  arc(x5, 100, 20, 20, 0.52, 5.76);
}
```

Creating an Array

Example 1. First, we'll declare the array outside of setup() and then create and assign the values within. The syntax x[0] refers to the first element in the array and x[1] is the second:

```
int[] x = new int[2];  // Declare and create the array

void setup() {
  size(200, 200);
  x[0] = 12;           // Assign the first value
  x[1] = 2;            // Assign the second value
}
```

Example 2. You can also assign values to the array when it's created, if it's all part of a single statement:

```
int[] x = { 12, 2 };  // Declare, create, and assign

void setup() {
  size(200, 200);
}
```

The End

Video Explanations

- Functions https://www.youtube.com/watch?v=zBo2D3Myo6Q&list=PLRqwX-V7Uu6ajGB2OI3hl5DZsD1Fw1WzR&index=2
- Variables https://www.youtube.com/watch?v=B-ycSR3ntik&list=PLRqwX-V7Uu6aFNOgoIMSbSYOkKNTo89uf
- Conditional Statements (If Else) https://www.youtube.com/watch?v=mVq7Ms01RjA&list=PLRqwX-V7Uu6YqykuLs00261JCqnL_NNZ_&index=2
- Loops (Iteration) https://www.youtube.com/watch?v=h4ApLHe8tbk&list=PLRqwX-V7Uu6bm-3M4Wntd4yYZGKwiKfrQ&index=3
- Arrays https://www.youtube.com/watch?v=NptnmWvkbTw&list=PLRqwX-V7Uu6bO9RKxHObluh-aPgrrvb4a&t=163s