From https://happycoding.io/tutorials/processing/debugging

Debugging

tutorial processing

As you write code, you'll inevitably encounter errors. This can be frustrating, but it's a huge part of being a programmer. Every programmer encounters errors, every single time they sit down to write code. Errors can happen for several different reasons, and this tutorial shows you how to **debug** your code to fix any errors in it.

Compiler Errors

The first type of error you'll probably see is a **compiler** error. Basically, a compiler error happens when you write code that the computer doesn't understand. In this case, Processing can't even run your code, because it doesn't know what you want it to do.

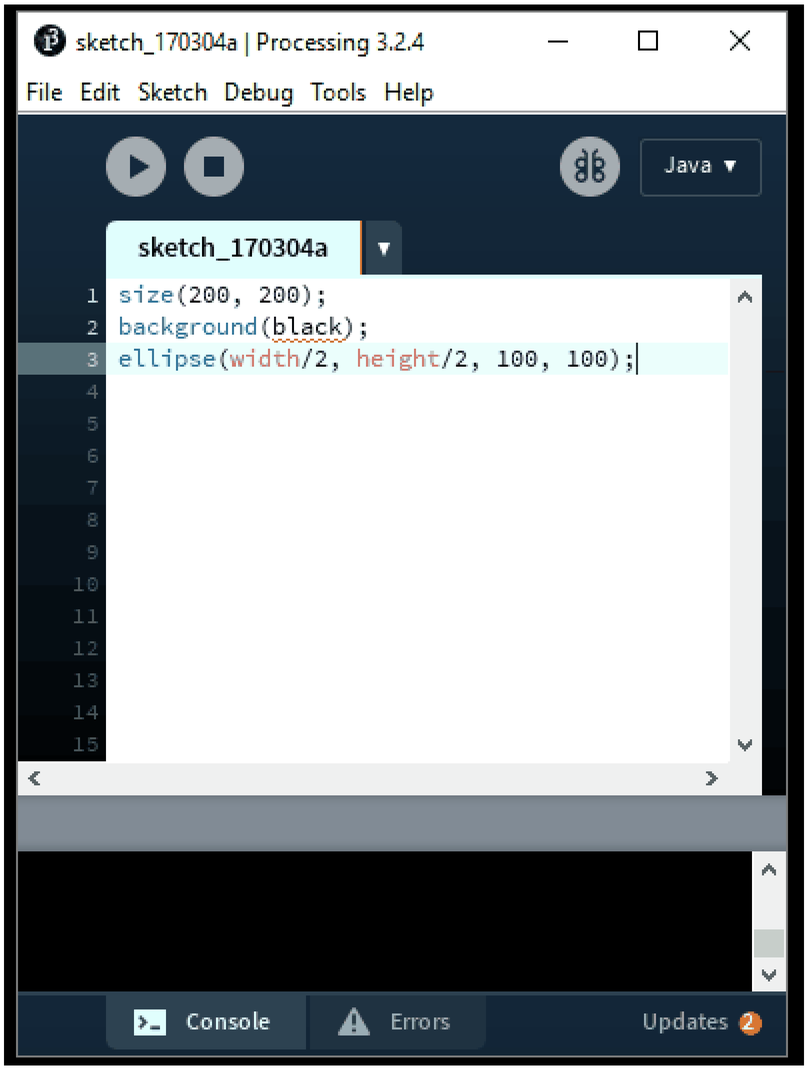
The rules of a language are called its **syntax**. Compiler errors are also called **syntax errors**, because they mean that your code broke the rules of the langauge. Compiler errors can be for things like forgotten semicolons or misspelled variables, but they can also be for violating the rules of Processing, like trying to store a `String` value in an `int` variable.

Here’s an example that contains a syntax error. Can you spot it?

```
size(200, 200);
background(black);
ellipse(width/2, height/2, 100, 100);
```

Do you see the syntax error? Type this code into your Processing editor, and notice a couple things:

- The Processing editor shows a red line under the `background(black);` line.
- If you mouse over the red line, a tooltip saying `The variable "black" does not exist` appears.
- If you try to run the code by pressing the play button, the editor shows a red message saying `black cannot be resolved to a variable`.


sketch_170304a | Processing 3.2.4
File Edit Sketch Debug Tools Help
Java
sketch_170304a
Console
Errors
Updates 2

```
size(200, 200);
background(black);
ellipse(width/2, height/2, 100, 100);
```

Notice that you can't run the code, because a compiler error prevents the computer from understanding what you want it to do.

In this case, the error is caused because we're using the word `black`, which Processing doesn't understand. We either need to create a variable named black:

```
size(200, 200);
float black = 0;
background(black);
ellipse(width/2, height/2, 100, 100);
```

Or we could pass the value into the `background()` function directly:

```
size(200, 200);
background(0);
ellipse(width/2, height/2, 100, 100);
```

Either way, this gets rid of the error, so now we can run the code. So when you see a compiler error, you should do the following:

- Look at the line being underlined.
- Look at the tool tip text for more info.
- Look at the message when you try to run the code for even more info.

If you can't figure out what the problem is, try going back and reading the tutorial for what your code is doing on that line. Compare your code to what's in the tutorial and look for differences. If you still can't figure it out, post a question in the forum! (Please read the "work in small pieces" section below!)

Runtime Errors

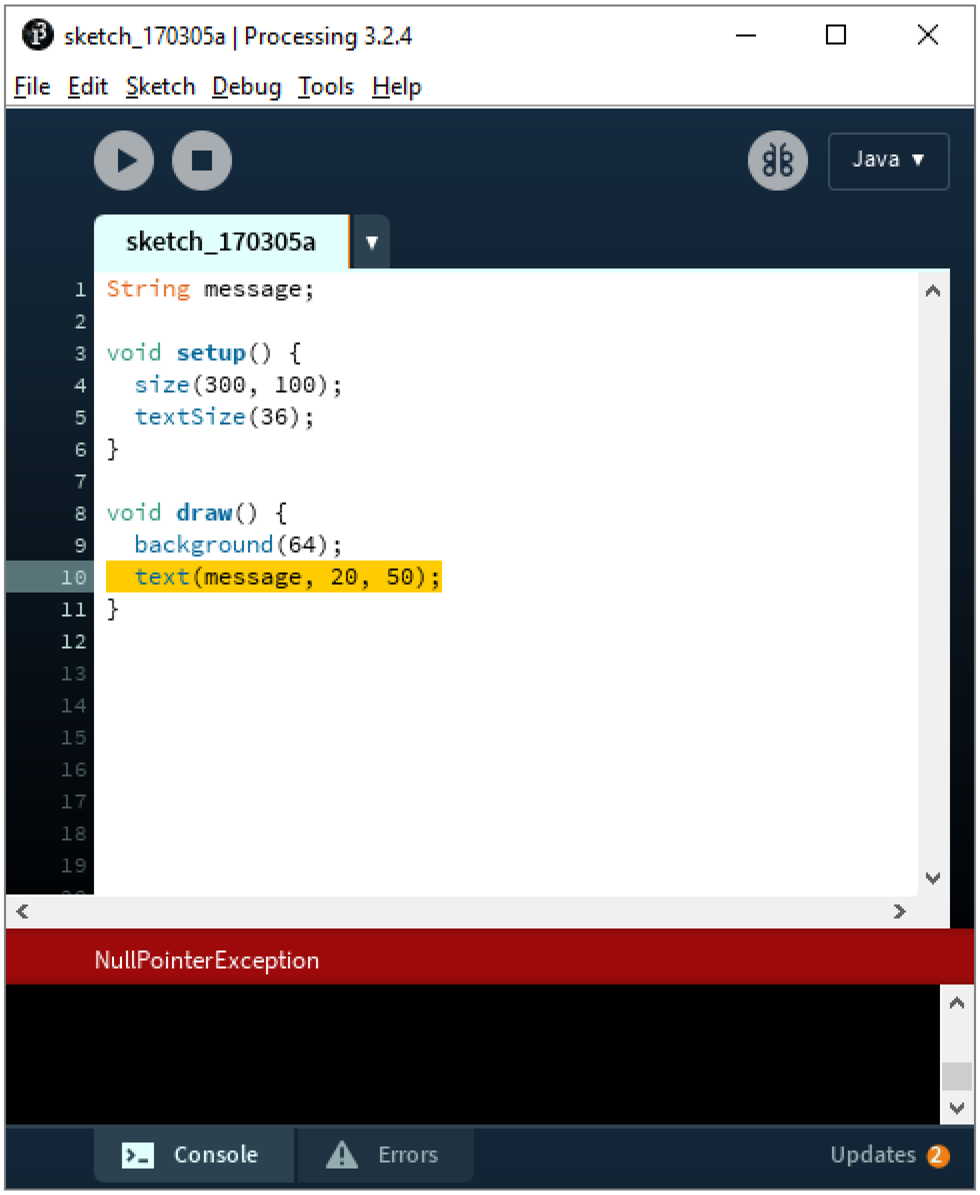
You can write code that the computer knows how to run, but causes an error when the computer tries to run it. This type of error is called a **runtime error**, and it usually results in your program crashing with an error message. Here's an example:

```
String message;

void setup() {
  size(300, 100);
  textSize(36);
}

void draw() {
  background(64);
  text(message, 20, 50);
}
```

This program defines a `message` variable, and uses the `text()` function to draw that message to the screen. Type this into the Processing editor, and notice that you don’t see any compiler errors. When you run the program, a window pops up, but no text is displayed! And you get an error:


sketch_170305a | Processing 3.2.4
File Edit Sketch Debug Tools Help
Java
sketch_170305a

```
String message;

void setup() {
  size(300, 100);
  textSize(36);
}

void draw() {
  background(64);
  text(message, 20, 50);
}
```

NullPointerException

The Processing editor highlights the `text(message, 20, 50);` line and shows `NullPointerException` in the error bar.

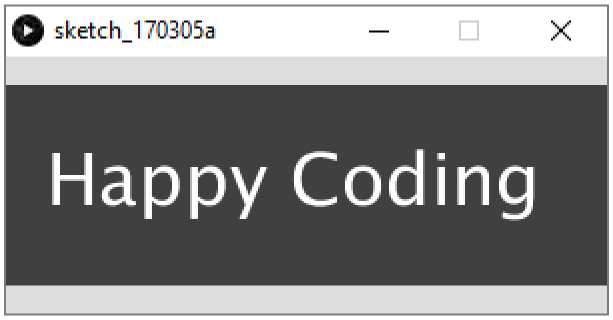
`NullPointerException` is one of the most common runtime errors, and it’s caused by using a variable that you haven’t given a value. In this case, we forgot to give the `message` variable a value, which is pretty easy to fix:

```
String message;

void setup() {
  size(300, 100);
  textSize(36);
  message = "Happy Coding";
}

void draw() {
  background(64);
  text(message, 20, 60);
```

Now the code gives the `message` variable a value inside the `setup()` function. This code runs fine!



There are many other types of errors, but the general process for debugging them is the same: look at the line highlighted in the editor, and look in the error bar to see what error was triggered.

If you still can't figure it out, check out the debugging sections below.

Logic Errors

Sometimes you'll write code that doesn't generate any errors, but still doesn't work how you expected it to. This is almost always caused by a **logic error**, and it's usually a result of the programmer (that's you!) making an invalid assumption or a typo.

Let's say we wanted to create a program that showed a direction being rotated: it starts out as up, then right, then down, then left, and finally up again. We might write this code:

```
String direction = "UP";

void setup(){
  frameRate(2);
  textSize(24);
  textAlign(CENTER);
}

void draw(){
  background(32);
  text(direction, 50, 60);
  rotateDirection();
}

void rotateDirection() {
  if (direction.equals("UP")) {
    direction = "RIGHT";
  }
  if (direction.equals("RIGHT")) {
    direction = "DOWN";
  }
  if (direction.equals("DOWN")) {
    direction = "LEFT";
  }
  if (direction.equals("LEFT")) {
    direction = "UP";
  }
}
```

This program defines a `direction` variable, which it draws to the screen. At the end of every frame, it calls the `rotateDirection()` funtion. The `rotateDirection()` function uses `if` statements to change the value of the `direction` variable to be the next direction in the sequence.

But if you run this program, you'll see that only `UP` is ever drawn to the screen. You don't get any errors, but the code isn't doing what we thought it would.

Debugging with Your Brain

When you get a compiler or logic error, the first thing you need to do is run through your code in your head. Read the code line by line, and double check that the code you wrote is the code you meant to write- just like you'd proofread anything you were writing.

Use a piece of paper and a pencil to write down any variable values or to draw stuff to the "screen" as you read through your code. Try reading through your code and imagining what it will do when it gets different values. Step through individual functions with imaginary parameters to make sure it will work in every case.

In the above example, read through the `rotateDirection()` function and think about exactly what each line does. Make sure you read the code correctly! Try not to assume you know what the code does, and just take it one line at a time. Reading it out loud can help.

Debugging with the `println()` Function

If you can't spot the error from reading through your code, then you need to figure out what the code is doing, and exactly where that differs from what you expected it to do.

A simple way to figure that out is by adding `println()` statements to your code. You can use `println()` statements to figure out the values of variables, which functions are being called in what order, how many times a loop is iterating, etc.

We might add `println()` statements to our above program to help figure out what it's doing:

```
String direction = "UP";

void setup() {
  frameRate(2);
  textSize(24);
  textAlign(CENTER);
  println("in setup, direction: " + direction);
}

void draw() {
  println("in draw, direction: " + direction);
  background(32);
  text(direction, 50, 60);
  rotateDirection();
  println("after rotateDirection, direction: " + direction);
}
```

```
void rotateDirection() {
  println("in rotateDirection, direction: " + direction);
  if (direction.equals("UP")) {
    println("Entered first if statement.");
    direction = "RIGHT";
    println("Direction is now: " + direction);
  }
  if (direction.equals("RIGHT")) {
    println("Entered second if statement.");
    direction = "DOWN";
    println("Direction is now: " + direction);
  }
  if (direction.equals("DOWN")) {
    println("Entered third if statement.");
    direction = "LEFT";
    println("Direction is now: " + direction);
  }
  if (direction.equals("LEFT")) {
    println("Entered fourth if statement.");
    direction = "UP";
    println("Direction is now: " + direction);
  }
}
```

Now when we run the program, we get this printed to the console:

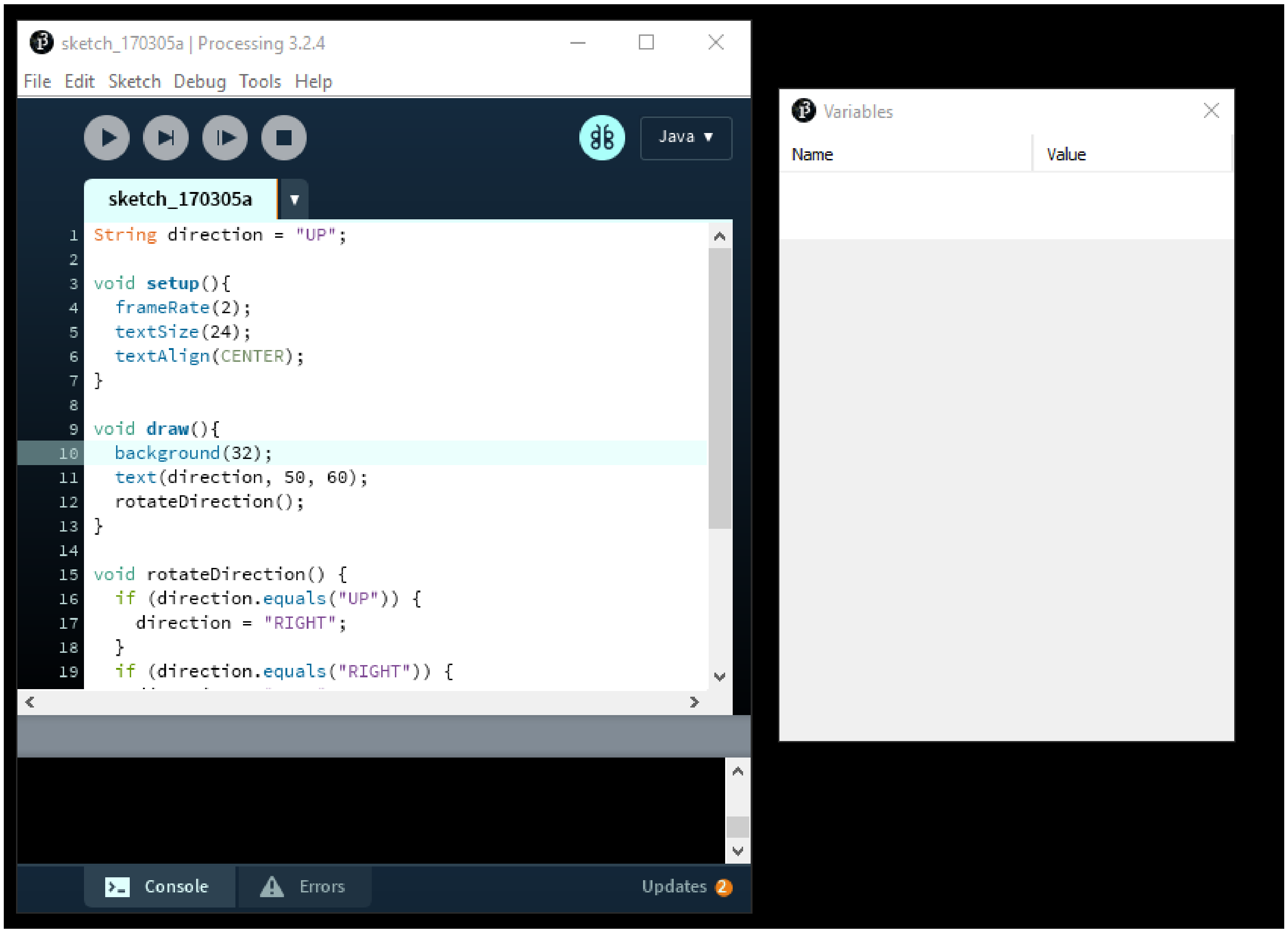
```
in setup, direction: UP
in draw, direction: UP
in rotateDirection, direction: UP
Entered first if statement.
Direction is now: RIGHT
Entered second if statement.
Direction is now: DOWN
Entered third if statement.
Direction is now: LEFT
Entered fourth if statement.
Direction is now: UP
after rotateDirection, direction: UP
```

We can use this output to read through our code again, this time comparing what we expect to be printed to what's actually printed. Do you see where the difference is?

Debugging with the Debugger

If all else fails, you can use the debugger that comes with the Processing editor. The debugger is a tool that allows you to watch your code run one line at a time.

To activate the debugger, press the Debug button- the butterfly-looking circle in the upper-right corner of the Processing editor. That brings up a new `Variables` window, which is blank for now.


sketch_170305a | Processing 3.2.4
File Edit Sketch Debug Tools Help
Java
sketch_170305a
Console
Errors
Updates 2
Variables
Name
Value

```
String direction = "UP";

void setup(){
  frameRate(2);
  textSize(24);
  textAlign(CENTER);
}

void draw(){
  background(32);
  text(direction, 50, 60);
  rotateDirection();
}

void rotateDirection() {
  if (direction.equals("UP")) {
    direction = "RIGHT";
  }
  if (direction.equals("RIGHT")) {
```

Now that the debugger is activated, you can use it by following these steps:

- First set a **break point** by clicking a line number. The line number should turn into a diamond symbol.

- Now press the play button to run your code using the debugger.
- The debugger will pause the execution of your code when it reaches your break point.
- Then you can use the **step** button to walk through the code one line at a time.
- The `Variables` window will show the value of every variable as you step through your code.

(To see this process in action visit https://happycoding.io/tutorials/processing/debugging)

The debugger allows us to run the code in "slow motion" and watch the values of variables change as the code runs. Again, the goal is to figure out exactly where the code does something different from what you expected.

In this case, the `println()` statements and the debugger tells us that the code is entering every `if` statement. So the `rotateDirection()` function reaches the first `if` statement, which it enters because `direction` starts out as `"UP"`. It reassigns `direction` to be `"RIGHT"`", which is correct so far. But then it looks at the next `if` statement, which checks whether `direction` is `"RIGHT"`! It is, so it reassigns it to be `"DOWN"`. This process repeats for all of the `if` statements, until the last one reassigns `direction` to be `"UP"`.

To fix this, we need to use `else if` statements:

```
if (direction.equals("UP")) {
  direction = "RIGHT";
}
else if (direction.equals("RIGHT")) {
  direction = "DOWN";
}
else if (direction.equals("DOWN")) {
  direction = "LEFT";
}
else if (direction.equals("LEFT")) {
  direction = "UP";
}
```

Now when one of the `if` statements is entered, it doesn't evaluate any of the other `if` statements. This causes the direction to be rotated only once, which was our original goal!

Work in Small Pieces

Programming is a process of breaking a problem down into smaller pieces, and approaching each of those pieces one at a time.

You should **not** be writing your entire sketch and then running it only after you've written all the code. Instead, test each piece as you write it. Make sure each line works how you expect, that way you can find errors as soon as they happen. It's very hard to debug a hundred lines of code that aren't doing what you expected. It's much easier to debug just a few lines of code.

Similarly, when you do find an error, you can use the above approaches (stepping through the code, using `println()` statements, and running the debugger) to narrow it down to just a few lines that aren't doing what you expect. If you still can't figure it out, then try creating a smaller example sketch that **only** contains enough code to see the same behavior.

For example, if your code loads 100 images, gives them random positions and speeds, and has them all bouncing off each other, but you can't get the bouncing to work right, then you need to start with a simpler sketch:

- Instead of images, use basic rectangles.
- Instead of showing 100 of them, only show 2 of them.
- Instead of giving them random positions and speeds, use hard-coded values.

But the small example program should still show the same problem as your main sketch. But now it's much easier to debug, because you don't have to look at any of the code that has nothing to do with the problem.

This type of program is called a MCVE (Minimal Complete Verifiable Example), and it also makes it easier for you to ask questions on the forum! Half the time you'll figure out your problem while trying to narrow your problem down to a smaller example sketch.

If all else fails, sometimes the best thing you can do is take a break. Go on a walk, pet your cat, and try to clear your head. You'll be amazed at how many solutions you end up thinking of in the shower.