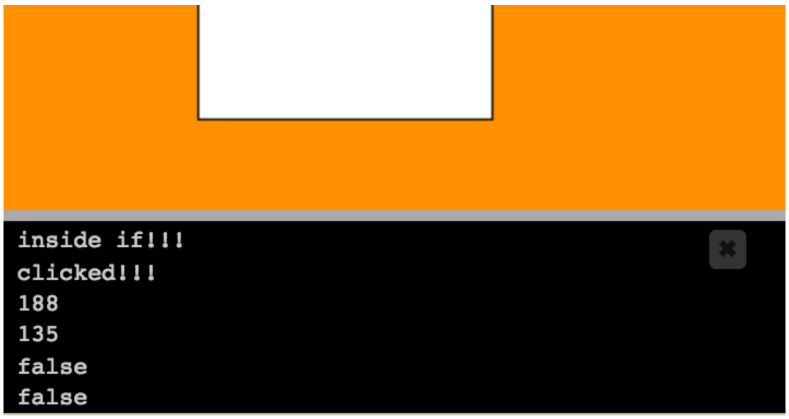
More debugging tips

Hide tutorial navigation

There are many ways to debug your programs! Here's a list to get you started:

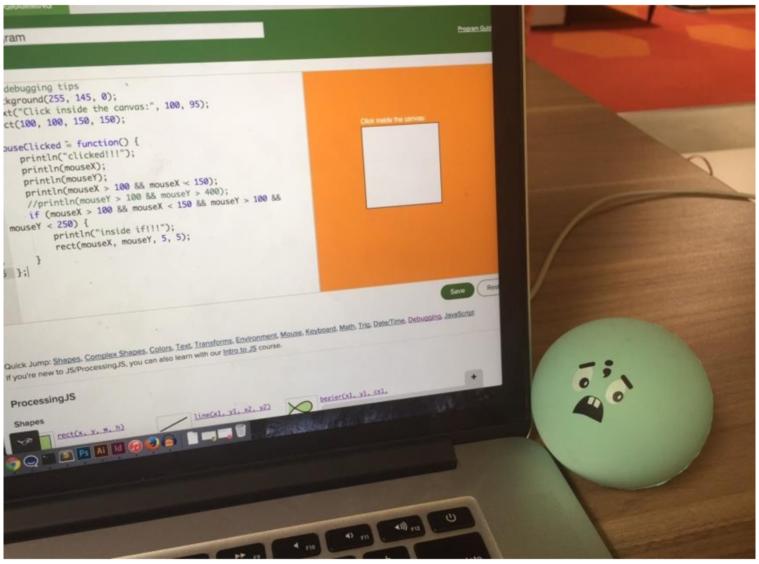
"Print" debugging: As we just showed, you can insert `print()`s or `println()`s into your code, to help you figure out which code is being called and with what values. Those functions both output values in a console that pops up over the canvas. You can also use `debug()` to send the output to your browser's JavaScript console, if you know how to use that.



```
inside if!!!
clicked!!!
188
135
false
false
```

"Rubber duck" debugging: Sit a rubber duck next to your computer (or whatever duck-like object you have handy), and explain your program and problem to it, line-by-line. Many programmers find that just the process of putting the problem in words helps their brain realize what's wrong. You can also ask a friend or teacher to "be your rubber duck" - sitting and listening to you explain it. Sometimes they might even think of a solution for you, but regardless, they're doing you a service by just listening to the explanation.

Here's the closest thing we have to a rubber duck in the KA office:

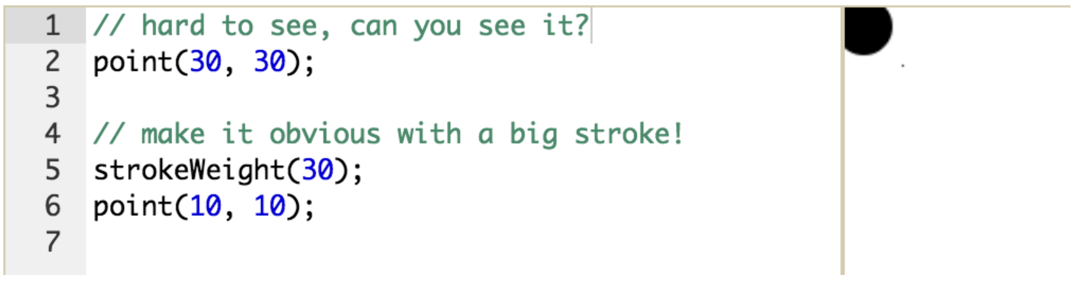


```
debugging tips
kground(255, 145, 0);
xt("Click inside the canvas:", 100, 95);
ct(100, 100, 150, 150);

ouseClicked = function() {
    println("clicked!!!");
    println(mouseX);
    println(mouseY);
    println(mouseX > 100 && mouseX < 150);
    //println(mouseY > 100 && mouseY > 400);
    if (mouseX > 100 && mouseX < 150 && mouseY > 100 &&
mouseY < 250) {
        println("inside if!!!");
        rect(mouseX, mouseY, 5, 5);
    }
};
```

Click inside the canvas:
Save
Quick Jump: Shapes, Complex Shapes, Colors, Text, Transforms, Environment, Mouse, Keyboard, Math, Trig, Date/Time, Debugging, JavaScript
If you're new to JS/ProcessingJS, you can also learn with our Intro to JS course.
ProcessingJS
Shapes
rect(x, y, w, h)
line(x1, y1, x2, y2)
bezier(x1, y1, cx1,

Exaggerate your output: Since you're making programs in ProcessingJS, you're dealing with lots of fill colors and strokes. When I'm not seeing the visual output I expect, sometimes it helps to use really big or extreme values for the fill and strokes - like `strokeWeight(30)`. Since our environment is real-time and includes the number scrubbers, it's really easy to change up the numbers in your program to see what effect it has on the output. For example, it might help you figure out where a missing shape went to.



```
// hard to see, can you see it?
point(30, 30);

// make it obvious with a big stroke!
strokeWeight(30);
point(10, 10);

```

It's a good idea to get comfortable with all your options for debugging programs, so you can use whichever one works the best in a particular situation.