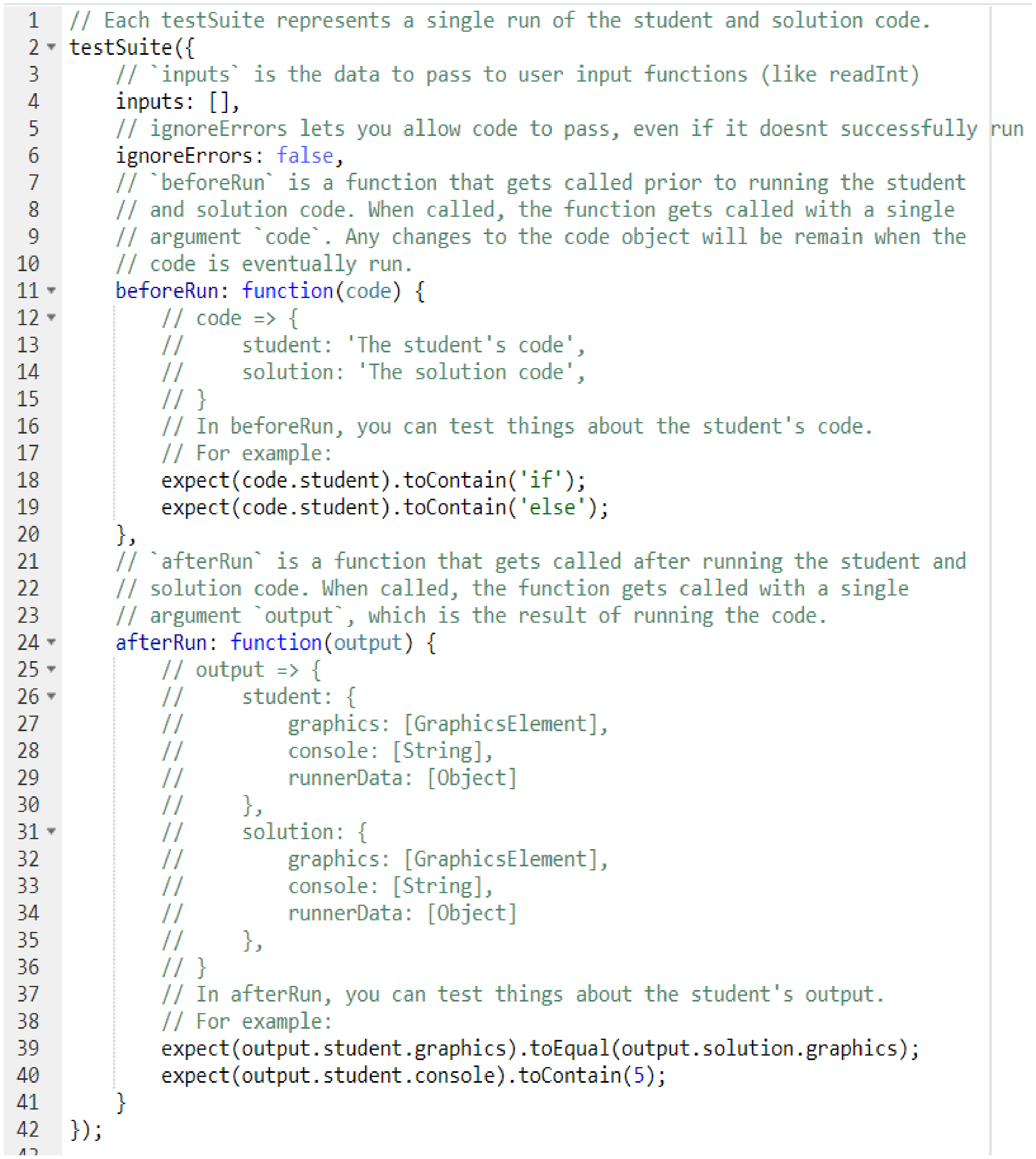
Programming JavaScript Autograders

This overview assumes you know how to add an autograder, what an autograder fundamentally is, etc. This is focusing on how to program the autograder.

Boiler Plate Autograder

This is what you see when you add the autograder. It's prefilled with examples of how to write very basic tests. The main parts are well commented.



```
// Each testSuite represents a single run of the student and solution code.
testSuite({
    // `inputs` is the data to pass to user input functions (like readInt)
    inputs: [],
    // ignoreErrors lets you allow code to pass, even if it doesnt successfully run
    ignoreErrors: false,
    // `beforeRun` is a function that gets called prior to running the student
    // and solution code. When called, the function gets called with a single
    // argument `code`. Any changes to the code object will be remain when the
    // code is eventually run.
    beforeRun: function(code) {
        // code => {
        //     student: 'The student's code',
        //     solution: 'The solution code',
        // }
        // In beforeRun, you can test things about the student's code.
        // For example:
        expect(code.student).toContain('if');
        expect(code.student).toContain('else');
    },
    // `afterRun` is a function that gets called after running the student and
    // solution code. When called, the function gets called with a single
    // argument `output`, which is the result of running the code.
    afterRun: function(output) {
        // output => {
        //     student: {
        //         graphics: [GraphicsElement],
        //         console: [String],
        //         runnerData: [Object]
        //     },
        //     solution: {
        //         graphics: [GraphicsElement],
        //         console: [String],
        //         runnerData: [Object]
        //     },
        // }
        // In afterRun, you can test things about the student's output.
        // For example:
        expect(output.student.graphics).toEqual(output.solution.graphics);
        expect(output.student.console).toContain(5);
    }
});
```

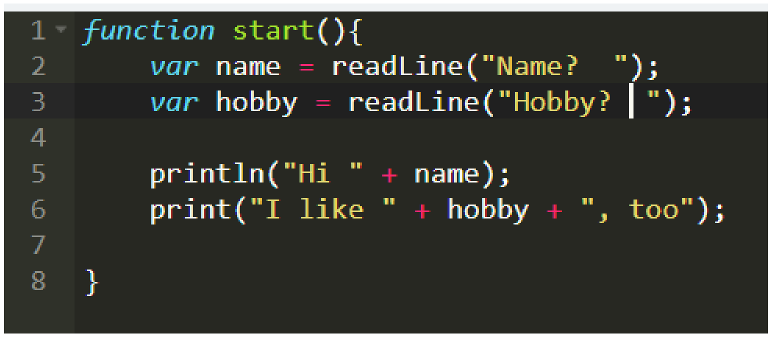
The Parameters

- **beforeRun**
  - ***code***: object with keys *student* and *solution*
  - *code.student, code.solution* are Strings
  - *code.student:* String containing all of the student's code
  - *code.solution:* String containing all of the solution's code
- **afterRun**
  - ***output****:* object mapping {string: object}
    - keys are *student, solution*
  - ***output.student****:* object mapping {String : array} with keys *graphics, console, runnerData*
    - *output.student.graphics:* array of all graphic objects created by student's program
    - *output.student.console:* array of Strings, each element is one line of output from the student's program
      - **VERY IMPORTANT**: this does not contain any text that was printed using *readInt, readLine, etc*
    - *output.student.runnerData:* information about the execution of the program
  - ***output.solution****:* object mapping {String:array} with keys *graphics, console, runnerData*
    - *output.solution.graphics:* array of all graphic objects created by solution program
    - *output.solution.console:* array of Strings, each element is one line of output from the solution program
      - **VERY IMPORTANT**: this does not contain any text that was printed using *readInt, readLine, etc*
    - *output.solution.runnerData:* information about the execution of the program

Parameter Value Examples

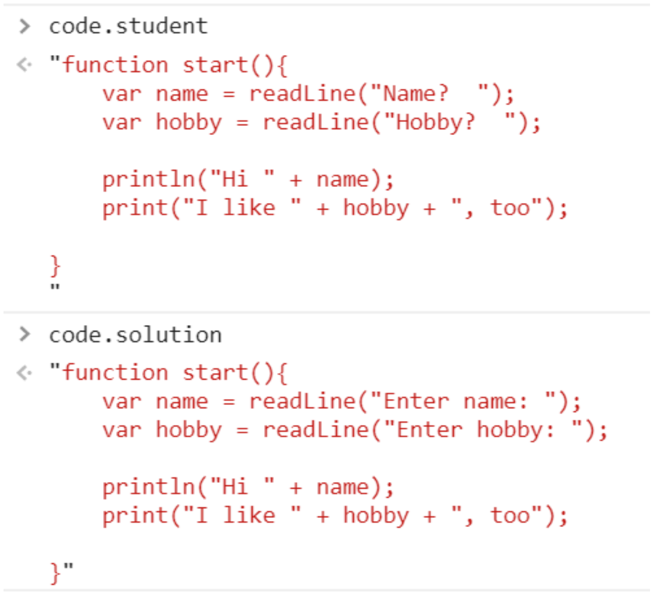
Example One: JS Console Program

Given the student's code:



```
function start(){
    var name = readLine("Name?  ");
    var hobby = readLine("Hobby? | ");

    println("Hi " + name);
    print("I like " + hobby + ", too");

}
```

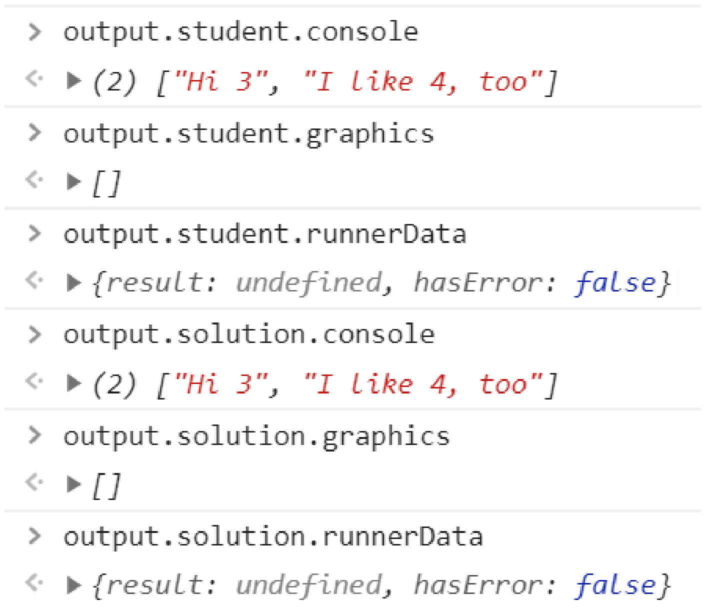
The arguments are as follows:
code:



```
> code.student
<· "function start(){
    var name = readLine("Name?  ");
    var hobby = readLine("Hobby?  ");

    println("Hi " + name);
    print("I like " + hobby + ", too");

}
"
> code.solution
<· "function start(){
    var name = readLine("Enter name: ");
    var hobby = readLine("Enter hobby: ");

    println("Hi " + name);
    print("I like " + hobby + ", too");

}"
```

Note: The solution code has been set in the CMS as the solution to this program.

output:

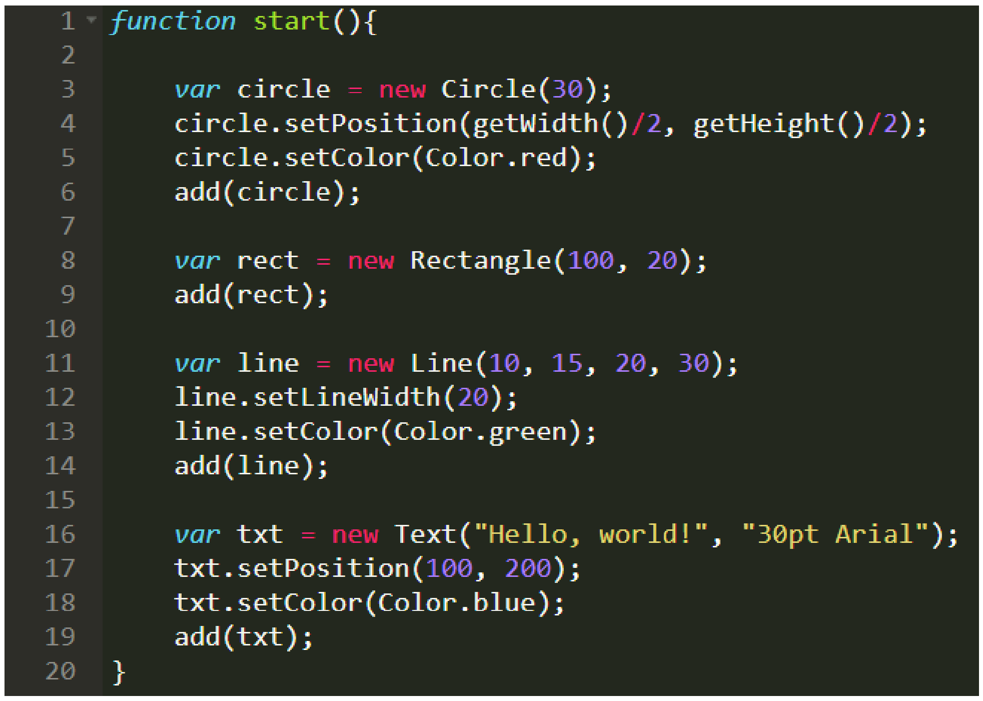


```
> output.student.console
<· ▶ (2) ["Hi 3", "I like 4, too"]
> output.student.graphics
<· ▶ []
> output.student.runnerData
<· ▶ {result: undefined, hasError: false}
> output.solution.console
<· ▶ (2) ["Hi 3", "I like 4, too"]
> output.solution.graphics
<· ▶ []
> output.solution.runnerData
<· ▶ {result: undefined, hasError: false}
```

Note: The console output only contains the output from *print* type statements, not input statements (like *readLine)*!

Example Two: JS Graphics Program

Given the student's code:

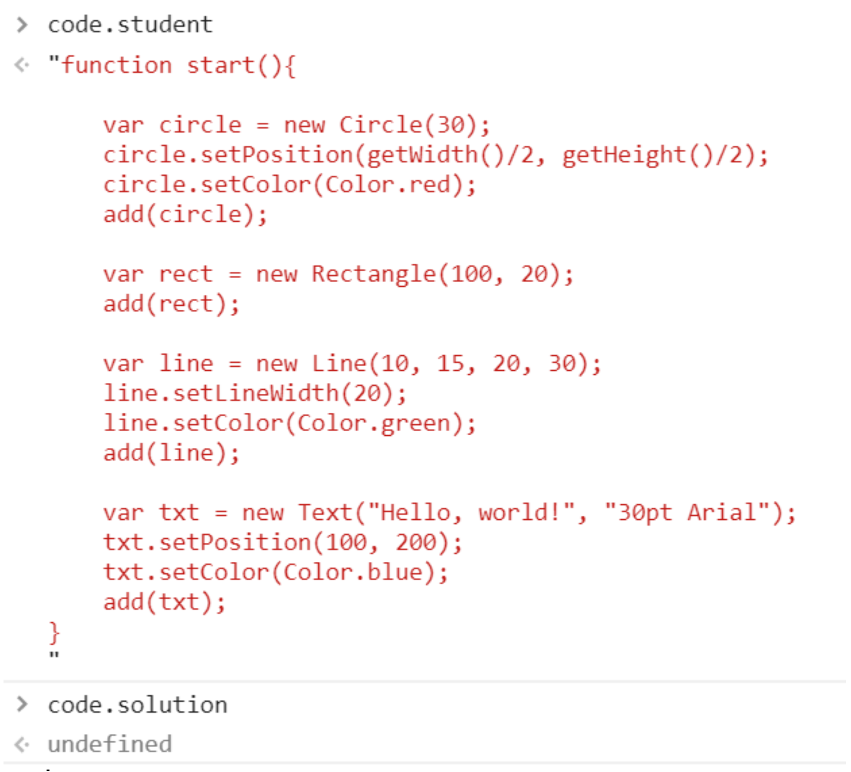


```javascript
function start(){

    var circle = new Circle(30);
    circle.setPosition(getWidth()/2, getHeight()/2);
    circle.setColor(Color.red);
    add(circle);

    var rect = new Rectangle(100, 20);
    add(rect);

    var line = new Line(10, 15, 20, 30);
    line.setLineWidth(20);
    line.setColor(Color.green);
    add(line);

    var txt = new Text("Hello, world!", "30pt Arial");
    txt.setPosition(100, 200);
    txt.setColor(Color.blue);
    add(txt);
}
```

Note: solution code has not been set for this program

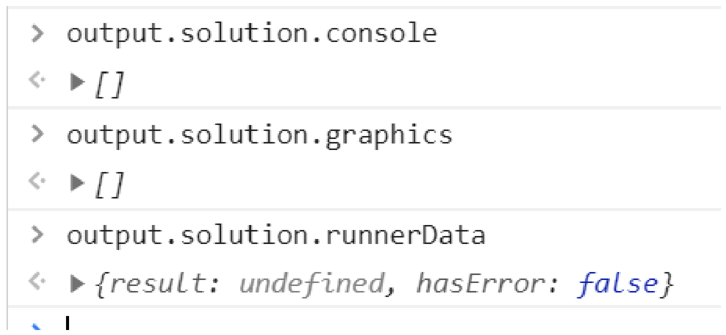
The arguments are as follows:

code:

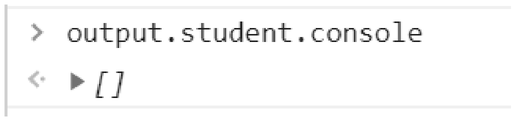


```
> code.student
< "function start(){

       var circle = new Circle(30);
       circle.setPosition(getWidth()/2, getHeight()/2);
       circle.setColor(Color.red);
       add(circle);

       var rect = new Rectangle(100, 20);
       add(rect);

       var line = new Line(10, 15, 20, 30);
       line.setLineWidth(20);
       line.setColor(Color.green);
       add(line);

       var txt = new Text("Hello, world!", "30pt Arial");
       txt.setPosition(100, 200);
       txt.setColor(Color.blue);
       add(txt);
   }
   "
> code.solution
< undefined
```

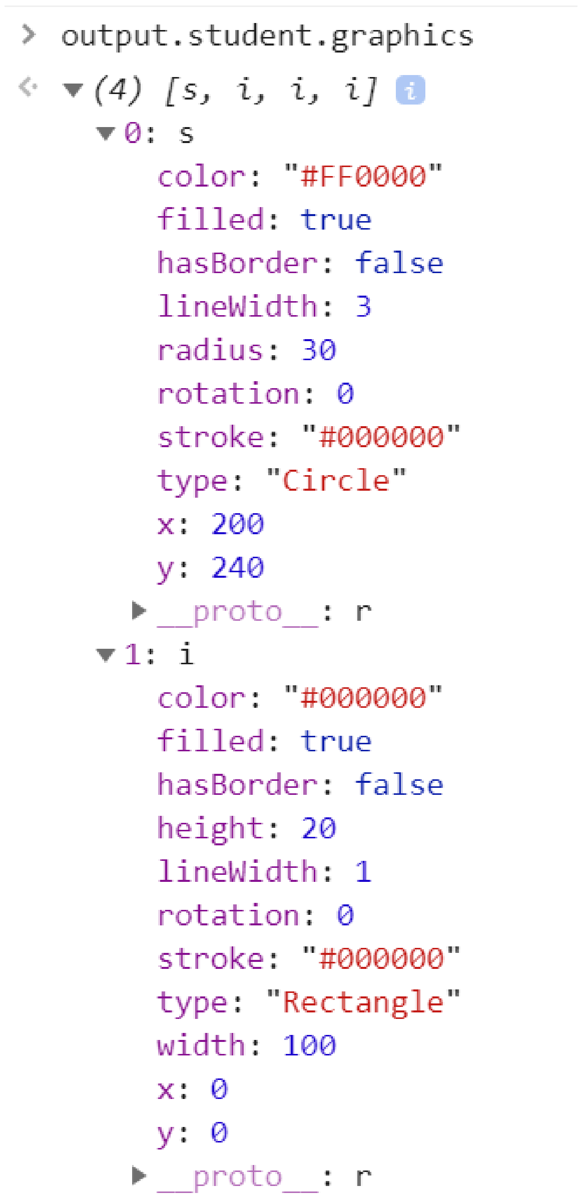
output:



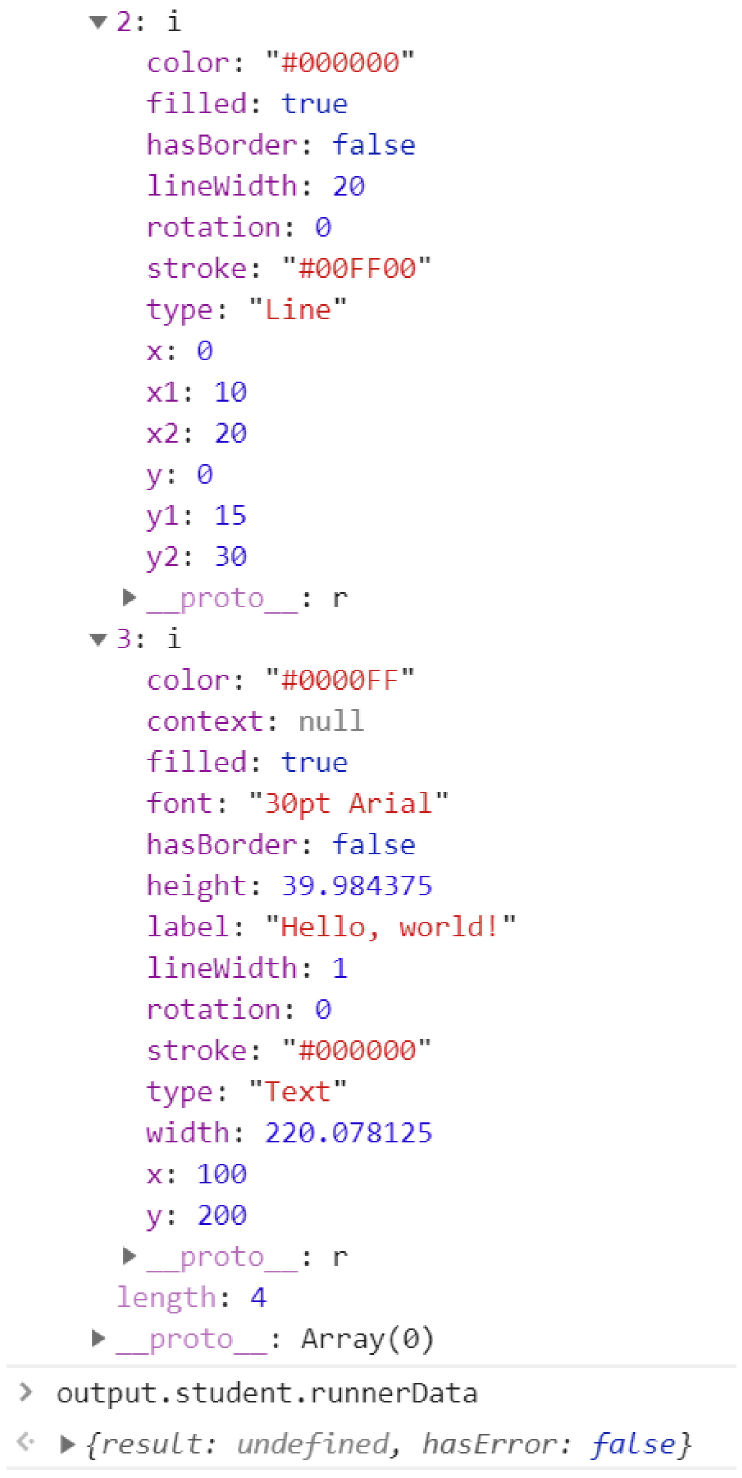
```
> output.solution.console
<· ▶ []
> output.solution.graphics
<· ▶ []
> output.solution.runnerData
<· ▶ {result: undefined, hasError: false}
```



```
> output.student.console
<· ▶ []
```



```
> output.student.graphics
<· ▼ (4) [s, i, i, i]
    ▼ 0: s
        color: "#FF0000"
        filled: true
        hasBorder: false
        lineWidth: 3
        radius: 30
        rotation: 0
        stroke: "#000000"
        type: "Circle"
        x: 200
        y: 240
      ▶ __proto__: r
    ▼ 1: i
        color: "#000000"
        filled: true
        hasBorder: false
        height: 20
        lineWidth: 1
        rotation: 0
        stroke: "#000000"
        type: "Rectangle"
        width: 100
        x: 0
        y: 0
      ▶ __proto__: r
```



```
▼2: i
   color: "#000000"
   filled: true
   hasBorder: false
   lineWidth: 20
   rotation: 0
   stroke: "#00FF00"
   type: "Line"
   x: 0
   x1: 10
   x2: 20
   y: 0
   y1: 15
   y2: 30
  ▶__proto__: r
▼3: i
   color: "#0000FF"
   context: null
   filled: true
   font: "30pt Arial"
   hasBorder: false
   height: 39.984375
   label: "Hello, world!"
   lineWidth: 1
   rotation: 0
   stroke: "#000000"
   type: "Text"
   width: 220.078125
   x: 100
   y: 200
  ▶__proto__: r
 length: 4
▶__proto__: Array(0)
> output.student.runnerData
< ▶{result: undefined, hasError: false}
```

Things to keep in mind:

- If you want to pass more than one set of inputs to the program, you need to create another testSuite.
- You can't access the inputs directly from the *beforeRun* and *afterRun* functions

Tests That Can Be Run

You can see the full test options here:
https://github.com/codehs/codehs/blob/master/codehs/editor/static/js/autograder/components/autograder-js/kombucha.js

To create a test, use ***expect(…)*** and then call one of the following test methods.

Let *res* be the student's output/code. Your options are:

- **expect(res).toBe(expected)**
  - Checks that *res* == *expected*
  - Example that checks the first line of output prints a title:
    - expect(output.solution.student[0]).toBe("Karel's To Do List");

- **expect(res).toEqual(expected)**
  - Checks *res.isEqual(expected)*
  - You usually want to use this one and not toBe(...)
  - Example that checks the first graphics object added to the Canvas is a circle:
    - expect(output.student.graphics[0].type).toEqual('Circle');

- **expect(res).toContain(expected)**
  - Checks that the string *expected* is contained in *res*
  - Example that checks the student has a while loop:
    - expect(code.student).toContain("while");

- **expect(res).toBeGreaterThan (expected)**
    - Checks res > expected
    - Example that checks the student printed at least 4 lines of output
        - expect(output.student.console.length).toBeGreaterThan(3);

- **expect(res).toBeGreaterThanOrEqualTo (expected)**
    - Checks res >= expected
    - Example that checks the student printed at least 4 lines of output
        - expect(output.student.console.length).toBeGreaterThanOrEqualTo(4);

- **expect(res).toBeLessThan(expected)**
    - Checks res < expected
    - Example that checks the student used no more than 2 for loops
        - var fors = code.student.match(/for/g); // match takes a regular expression
          var numFors = fors ? fors.length : 0;   // make sure *fors* isn't null
          expect(numFors).toBeLessThan(3);

- **expect(res).toBeLessThanOrEqualTo(expected)**
    - Checks res <= expected
    - Example that checks the student used no more than 2 rectangles
        - var rects = output.student.graphics.filter(function(elem){
                  return elem.type == "Rectangle"});
          // rects will be an array with only Rectangle objects
           expect(rects.length).toBeLessThanOrEqualTo(1);

- **expect(res).toBeUndefined()**
    - Checks res === 'undefined'
    - Example that checks the student did not use any for loops
        - str.match returns null if there are no matches
        - var fors = code.student.match(/for/g);
          expect(fors).toBeUndefined();

Summary of Expectation Functions

- toEqual(value)
- toBe(expected)
- toContain(expected)
- toBeGreaterThan(value)
- toBeGreaterThanOrEqualTo(value)
- toBeLessThan(value)
- toBeLessThanOrEqualTo(value)
- toBeUndefined()

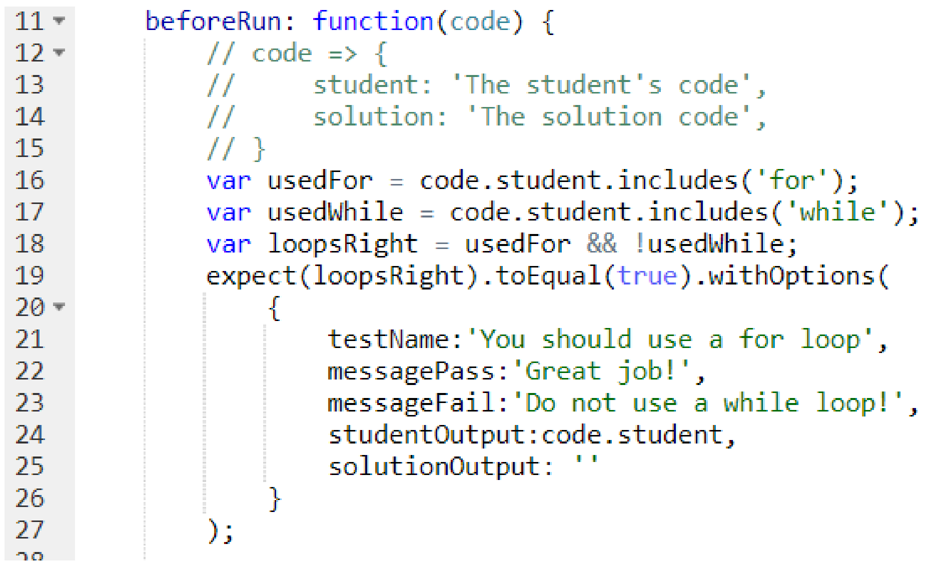
Test Options

With each of the previous methods, you can then call *withOptions()* on the expect object. The parameter is an object with the following possible keys:

- **testName**: the test name displayed to the student
    - defaults to a string representation of the test
        - for example, something like "Expected "print("hello world')" to contain "print""
- **messagePass**: Message displayed if the test passed
    - defaults to "Success"
- **messageFail**: Message displayed if the test failed
    - defaults to "Try Again"
- **studentOutput**: What is shown as the student output (labeled "your result")
    - defaults to what's passed to *expect*
- **solutionOutput**: What is shown as the solution output
    - defaults to the value passed to the expectation function (e.g *expected* or *value* from the examples above)
- **showDiff**: Shows the difference between the student's output and the solution output
    - defaults to False
    - Note: unless the program only passes with very specific formatting, students usually find this more confusing than helpful

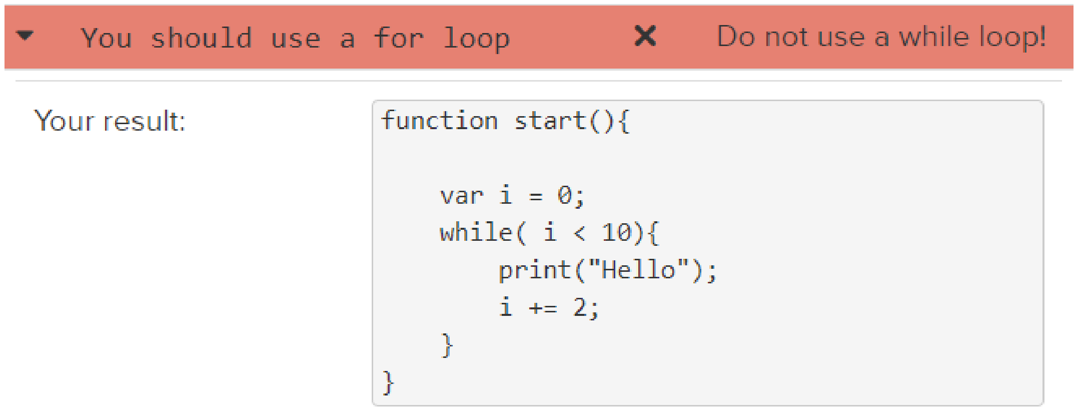
Test Options Examples

Example that checks the student used a for loop and did not use a while loop



```
    beforeRun: function(code) {
        // code => {
        //     student: 'The student's code',
        //     solution: 'The solution code',
        // }
        var usedFor = code.student.includes('for');
        var usedWhile = code.student.includes('while');
        var loopsRight = usedFor && !usedWhile;
        expect(loopsRight).toEqual(true).withOptions(
            {
                testName:'You should use a for loop',
                messagePass:'Great job!',
                messageFail:'Do not use a while loop!',
                studentOutput:code.student,
                solutionOutput: ''
            }
        );
```

This test produces the following test results. The code shown is the student's program. Note there is no expected output shown.

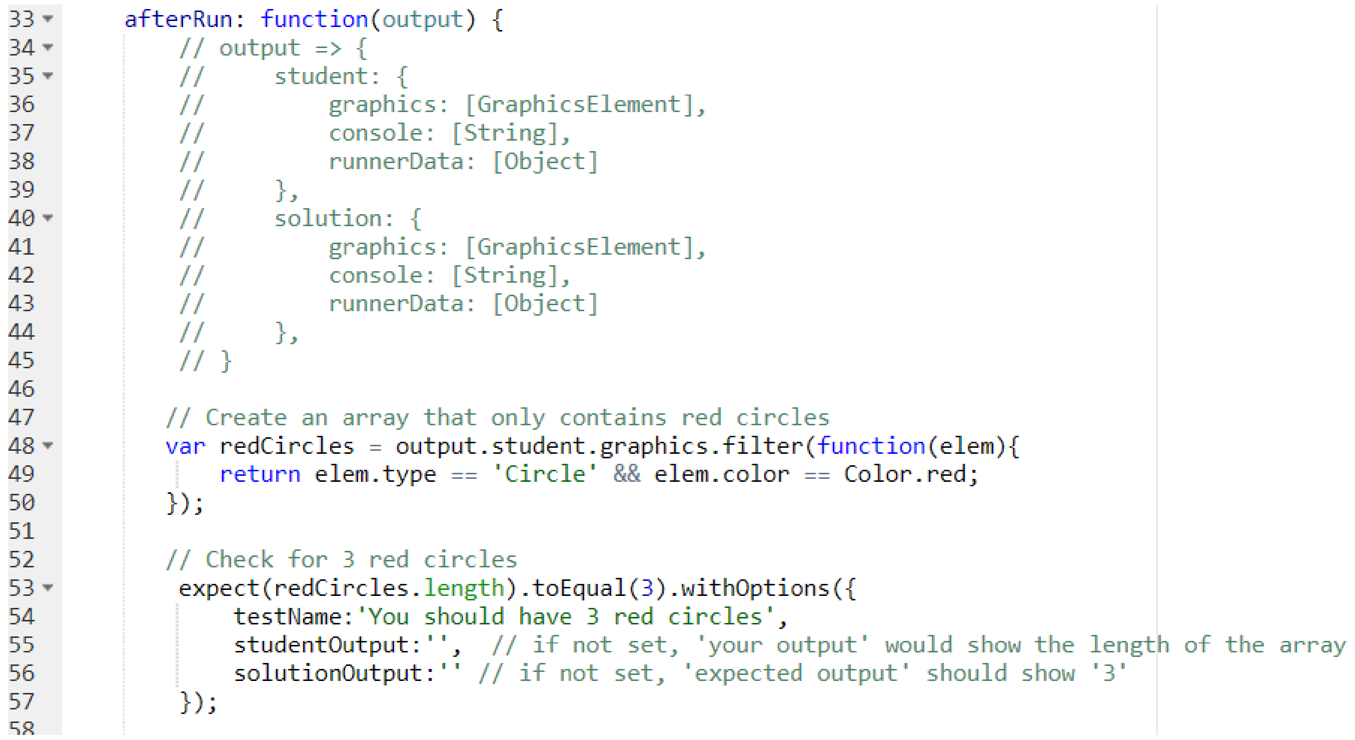


You should use a for loop ✗ Do not use a while loop!

Your result:

```
function start(){

    var i = 0;
    while( i < 10){
        print("Hello");
        i += 2;
    }
}
```

Example that checks for three red circles



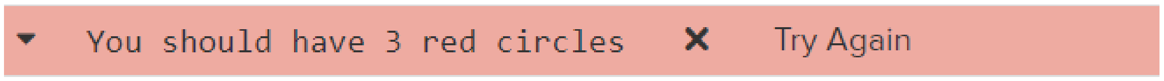
```
    afterRun: function(output) {
        // output => {
        //     student: {
        //         graphics: [GraphicsElement],
        //         console: [String],
        //         runnerData: [Object]
        //     },
        //     solution: {
        //         graphics: [GraphicsElement],
        //         console: [String],
        //         runnerData: [Object]
        //     },
        // }

       // Create an array that only contains red circles
       var redCircles = output.student.graphics.filter(function(elem){
           return elem.type == 'Circle' && elem.color == Color.red;
       });

       // Check for 3 red circles
        expect(redCircles.length).toEqual(3).withOptions({
            testName:'You should have 3 red circles',
            studentOutput:'',  // if not set, 'your output' would show the length of the array
            solutionOutput:'' // if not set, 'expected output' should show '3'
        });
```

Note the success and failure messages are not set; they will be given default values (shown below). Note also that to check the color of the graphics object, you should use the Color class. The colors in the object are given as hex color codes.

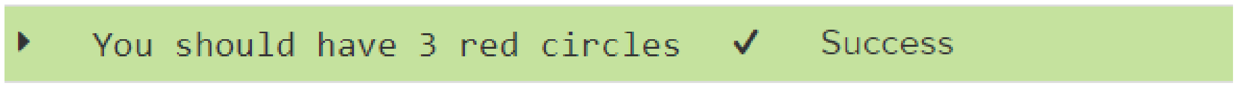
For example, here is one of the circle objects from the *redCircles* array:

```
{
    "x": 200,
    "y": 240,
    "color": "#FF0000",
    "stroke": "#000000",
    "type": "Circle",
    "lineWidth": 3,
    "filled": true,
    "hasBorder": false,
    "rotation": 0,
    "radius": 30
  }
```

Two separate runs of this test produces the two outputs below. Note there is no additional information, since both *studentOutput* and *solutionOutput* have been set to empty strings.



```
You should have 3 red circles    ✕    Try Again
```



```
You should have 3 red circles    ✓    Success
```

Tests With Input

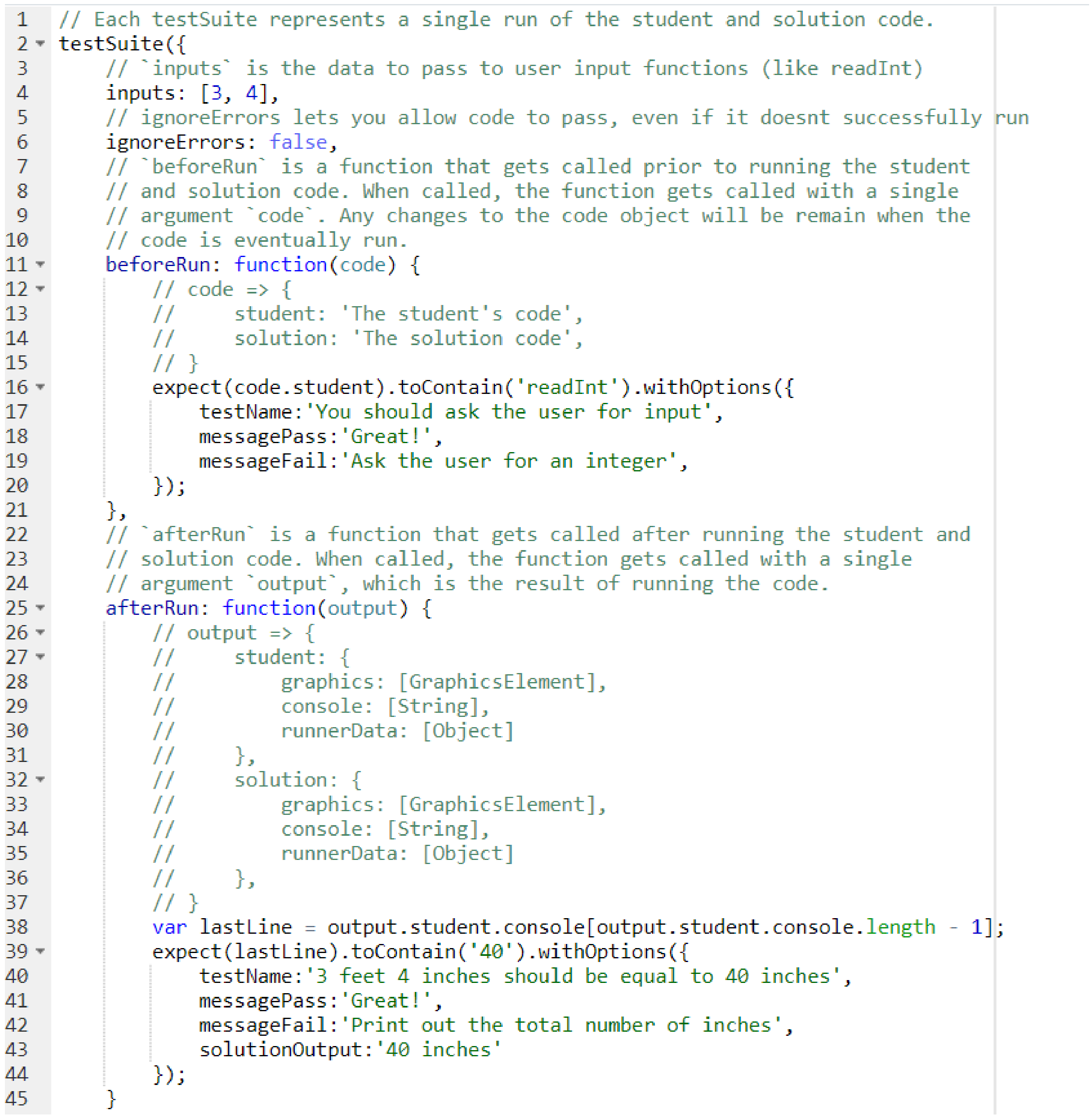
To set the input to a program, put the values in the *input* list.
The program will be run **once** with the given input. If you want to test a different set of input, you need to write another test suite(see examples below).
The input should be given as strings. This minimizes the number of weird errors that occur for the student.

Example

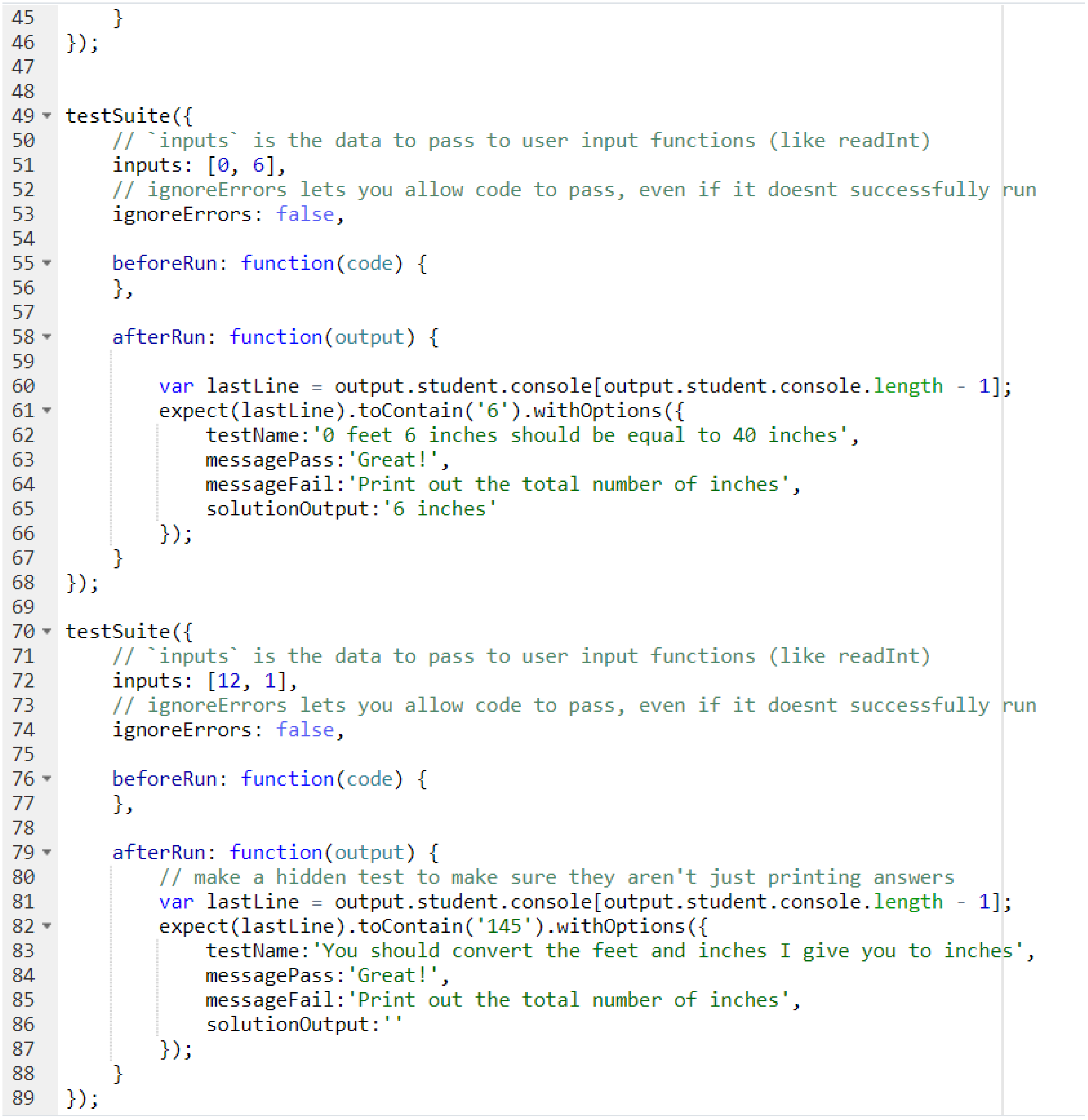
Problem: The student is supposed to ask the user for a number of feet and number of inches, then print out the number of inches.

Test Cases: 3 ft, 4 inches → 40 inches
0 ft, 6 inches → 6 inches
12 ft, 1 inches → 145 inches

The autograder:



```
// Each testSuite represents a single run of the student and solution code.
testSuite({
    // `inputs` is the data to pass to user input functions (like readInt)
    inputs: [3, 4],
    // ignoreErrors lets you allow code to pass, even if it doesnt successfully run
    ignoreErrors: false,
    // `beforeRun` is a function that gets called prior to running the student
    // and solution code. When called, the function gets called with a single
    // argument `code`. Any changes to the code object will be remain when the
    // code is eventually run.
    beforeRun: function(code) {
        // code => {
        //     student: 'The student's code',
        //     solution: 'The solution code',
        // }
        expect(code.student).toContain('readInt').withOptions({
            testName:'You should ask the user for input',
            messagePass:'Great!',
            messageFail:'Ask the user for an integer',
        });
    },
    // `afterRun` is a function that gets called after running the student and
    // solution code. When called, the function gets called with a single
    // argument `output`, which is the result of running the code.
    afterRun: function(output) {
        // output => {
        //     student: {
        //         graphics: [GraphicsElement],
        //         console: [String],
        //         runnerData: [Object]
        //     },
        //     solution: {
        //         graphics: [GraphicsElement],
        //         console: [String],
        //         runnerData: [Object]
        //     },
        // }
        var lastLine = output.student.console[output.student.console.length - 1];
        expect(lastLine).toContain('40').withOptions({
            testName:'3 feet 4 inches should be equal to 40 inches',
            messagePass:'Great!',
            messageFail:'Print out the total number of inches',
            solutionOutput:'40 inches'
        });
    }
```



```
    }
});


testSuite({
    // `inputs` is the data to pass to user input functions (like readInt)
    inputs: [0, 6],
    // ignoreErrors lets you allow code to pass, even if it doesnt successfully run
    ignoreErrors: false,

    beforeRun: function(code) {
    },

    afterRun: function(output) {

        var lastLine = output.student.console[output.student.console.length - 1];
        expect(lastLine).toContain('6').withOptions({
            testName:'0 feet 6 inches should be equal to 40 inches',
            messagePass:'Great!',
            messageFail:'Print out the total number of inches',
            solutionOutput:'6 inches'
        });
    }
});

testSuite({
    // `inputs` is the data to pass to user input functions (like readInt)
    inputs: [12, 1],
    // ignoreErrors lets you allow code to pass, even if it doesnt successfully run
    ignoreErrors: false,

    beforeRun: function(code) {
    },

    afterRun: function(output) {
        // make a hidden test to make sure they aren't just printing answers
        var lastLine = output.student.console[output.student.console.length - 1];
        expect(lastLine).toContain('145').withOptions({
            testName:'You should convert the feet and inches I give you to inches',
            messagePass:'Great!',
            messageFail:'Print out the total number of inches',
            solutionOutput:''
        });
    }
});
```

Some things to Note:

- Each set of input needs its own testSuite, hence there are three testSuite objects
- *output.student.console* is an array of Strings, which means if you expect the student's code to contain something, you either have to iterate through the array or convert the array into a string
    - Here, the output we're looking for is in the last line, so we just checked the last line
- Remember that *withOptions* takes an object!
- Be careful of what you do with *code.student* in *beforeRun*. Any changes you make to that code will persist when the code is run to produce their output!

Some General Hacks and Tips

- You very, very rarely want to use expect(output.student.console).toEqual(output.solution.console);
    - It’s better to just look for key pieces
- To help debug your tests, you can set *studentOutput* or *solutionOutput* to see the values of your variables or print them using *console.log(...)*
    - Remember *studentOutput* and *solutionOutput* have to be string values
- Remember the browser has a built in JavaScript console. You can use that to work out syntax errors or make sure your array functions are behaving like you think.
- String comparisons are case sensitive and whitespace sensitive
    - use str.toLowerCase() to make str all lowercase
    - use str.replace(‘ ‘, ‘’) to remove spaces (but not all whitespace)
        - first parameter is a single space, second is an empty string
    -

Additional Checks

Check for a number of keywords:

```
function countWords(str, regex){
    // an array of all the matches in str
    let matches = str.match(regex);

    if (matches != null){
        return matches.length;
    } else{
        return 0;
    }
}
```

Example that checks for definition and call a function:

```
// counts "calculateInterestGrowth("
let calculateInterestGrowthCount =
countWords(code.student.replaceAll(' ', ''), /calculateInterestGrowth+\(/g);


expect(calculateInterestGrowthCount).toEqual(2).withOptions({
     testName: 'You need to define and call calculateInterestGrowth',
     messagePass: 'Nicely done!',
     messageFail: 'Did you define and call the functions properly?',
     studentOutput: code.student,
     solutionOutput: 'calculateInterestGrowth()'
});
```

Check for graphic with specific characteristics

```
// checks for a square with length 1 at (0,0)
let starterSquare = output.student.graphics.filter(function(elem){
    return elem.type == 'Rectangle' && elem.x == 0 && elem.y == 0 && elem.width
== 1 && elem.height == 1; });
```

```
expect(starterSquare.length).toEqual(1).withOptions({
     testName: 'You should have a starting square with length 1 at (0,0)',
     studentOutput: '',
     solutionOutput: ''
});
```

Check for keywords inside of () - like parameters in a function definition

```
// finds things inside of the parameters definition of a function
// regex should be "functionNAME(...)" in regex form
// searchRegex could be anything you want to find in the parameters, like "="
for default values

function countThingsWithinParameters(str, regex, searchRegex){
   let matches = str.match(regex);
   let matches2 = matches[0].match(searchRegex);
   if (matches2 != null){
       return matches2.length;
   }else{
       return 0;
   }
}
```

Example

```
// counts how many = signs there are in drawHead parameter definition

let equalsInHead = countThingsWithinParameters(code.student.replaceAll(' ',
''), /functiondrawHead+\(([^{]+)\)/g, /\=/g);

expect(equalsInHead).toEqual(4).withOptions({
     testName: 'You need to have default values for all parameters in
drawHead()',
     messagePass: 'Nicely done!',
     messageFail: 'Are you setting default values properly?',
     studentOutput: equalsInHead,
     solutionOutput: 4
});
```