Lecture 05

Practice: functions and while Loops
Introducing: Procedures

Go to poll.unc.edu

Sign-in via this website then go to pollev.com/compunc

VSCode: Open Project -> View Terminal -> npm run pull -> npm start

Announcements

- WS0 – Due Saturday at 11:59pm
  - Print and Handwrite!
  - Submission Instructions: Topics > Getting Started > Gradescope Submissions
  - Uploading your first scanned assignment will take 30 minutes – you should do it tomorrow before office hours close so you can get help, if needed!
- PS1 – Dwight's Beet Farm
  - Demo: http://apps.introcs.com/hank/ps01-beets/
  - Due Monday at 11:59pm
  - Submissions system will open over the weekend
- Q1 – Quiz 1 on Tuesday
  - boolean operators
  - if-then-else
  - while loops
  - function definition and calls

The **TypeScript** REPL

- For all lectures, worksheets, and problem sets, we will follow and grade based on the TypeScript language's rules.

- Your web browser's REPL follows the JavaScript language's rules.
  - JavaScript is TypeScript *without any data typing rules*
  - In JavaScript you can do **fully crazy** things like multiply numbers with booleans
    - What does it even mean? No point in knowing… it's never a good idea and is a bug or bad code.
  - We started in the JavaScript REPL because you didn't need VSCode installed to use it

- For a TypeScript REPL – open a new terminal in VSCode and run the command:

```
npm run repl
```

Increment Operator (++)

- Adding one to a variable is so common when looping there is a special operator for it…

- We often write: **`i = i + 1;`**

- We can instead write: **`i++;`**

- These two statements have the exact same impact of incrementing **`i`**'s value by **`1`**.

Decrement Operator (--)

- Subtracting one from a variable is *also* so common, there is a special operator for it…

- We often write: **`i = i - 1;`**

- We can instead write: **`i--;`**

- These two statements have the exact same impact of incrementing **`i`**'s value by **`1`**.

Challenge Question #0 - pollev.com/compunc

- What is the result of calling: **`michaelJackson(3)`**

```
let michaelJackson = (force: number): string => {
    let s = "";
    let i = 1;
    while (i < force) {
        s = s + "h";
        let h = 0;
        while (h < i) {
            s = s + "e";
            h++;
        }
        i++;
    }
    return s;
};
```

Notes on Nested Loops

- **General Rule:** When the closing curly brace of a loop is encountered, the loop jumps back to the start of **its matching condition**.

- An inner loop will jump back up to the inner loop's condition and an outer loop will jump back up to the outer loop's condition.

- Thus, an inner loop must complete all of its **iterations** for every individual iteration of an outer loop.

Built-in Math Functions

- There is a special, built-in "class" called Math with useful functions defined in it:
  - **Math.floor(n: number): number** – rounds a number down
  - **Math.ceil (roof: number): number** – rounds a number up
  - **Math.round(n: number): number** – rounds a number >= 0.5 up, < 0.5 down
  - **Math.random(): number** – generates a random decimal >= 0.0 and < 1.0
  - **Math.abs(n: number): number** – absolute value of a number
  - **Math.sqrt(n: number): number** – square root

- The way to **call** one of these functions is to write **Math.<function>(<arguments>)**
  - For example: **Math.floor(1.5)** will evaluate to a value of 1

- There are others as well, like trigonometric functions (sin, cos, tan).
  - Complete list found on official documentation:
  - https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Math

Challenge Question #1 - pollev.com/compunc

- What does **`mystery(15)`** evaluate to?

- Reminder: **`Math.floor`** is a function that rounds down.

```
let mystery = (n: number): number => {
   let d = Math.floor(n / 2);
   while (d > 1) {
      if (n % d === 0) {
         return d;
      }
      d = d - 1;
   }
   return d;
};
```

The **return** Statement – Important Rule

- When the processor reaches a **`return`** statement inside of a function…

THE FUNCTION CALL IS **COMPLETE**!

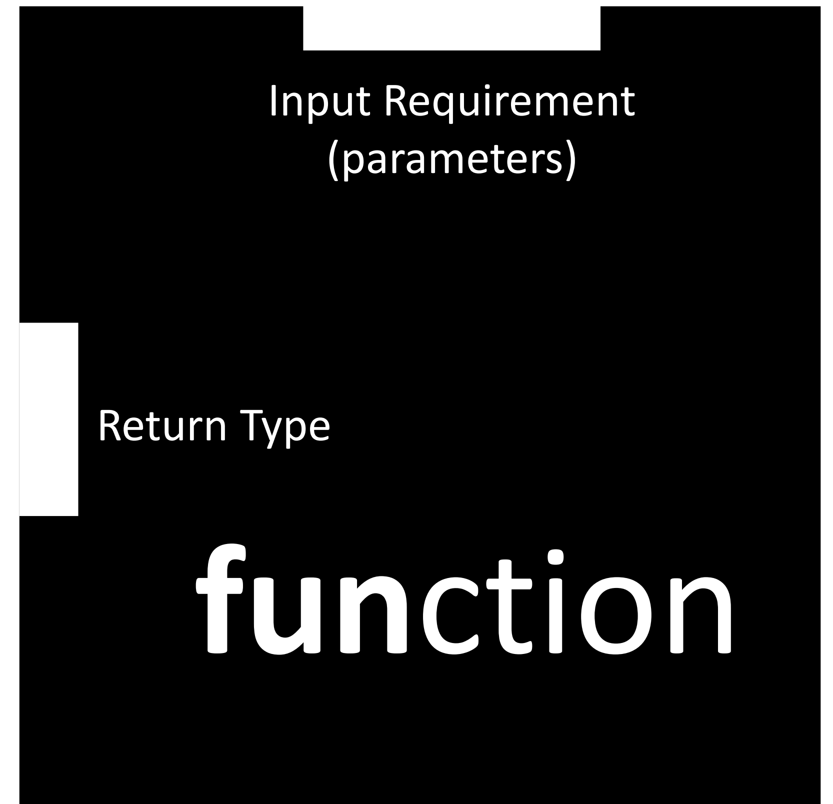
THAT FUNCTION'S JOB IS **DONE**!

THE RETURNED VALUE IS SENT BACK TO THE CALLER!

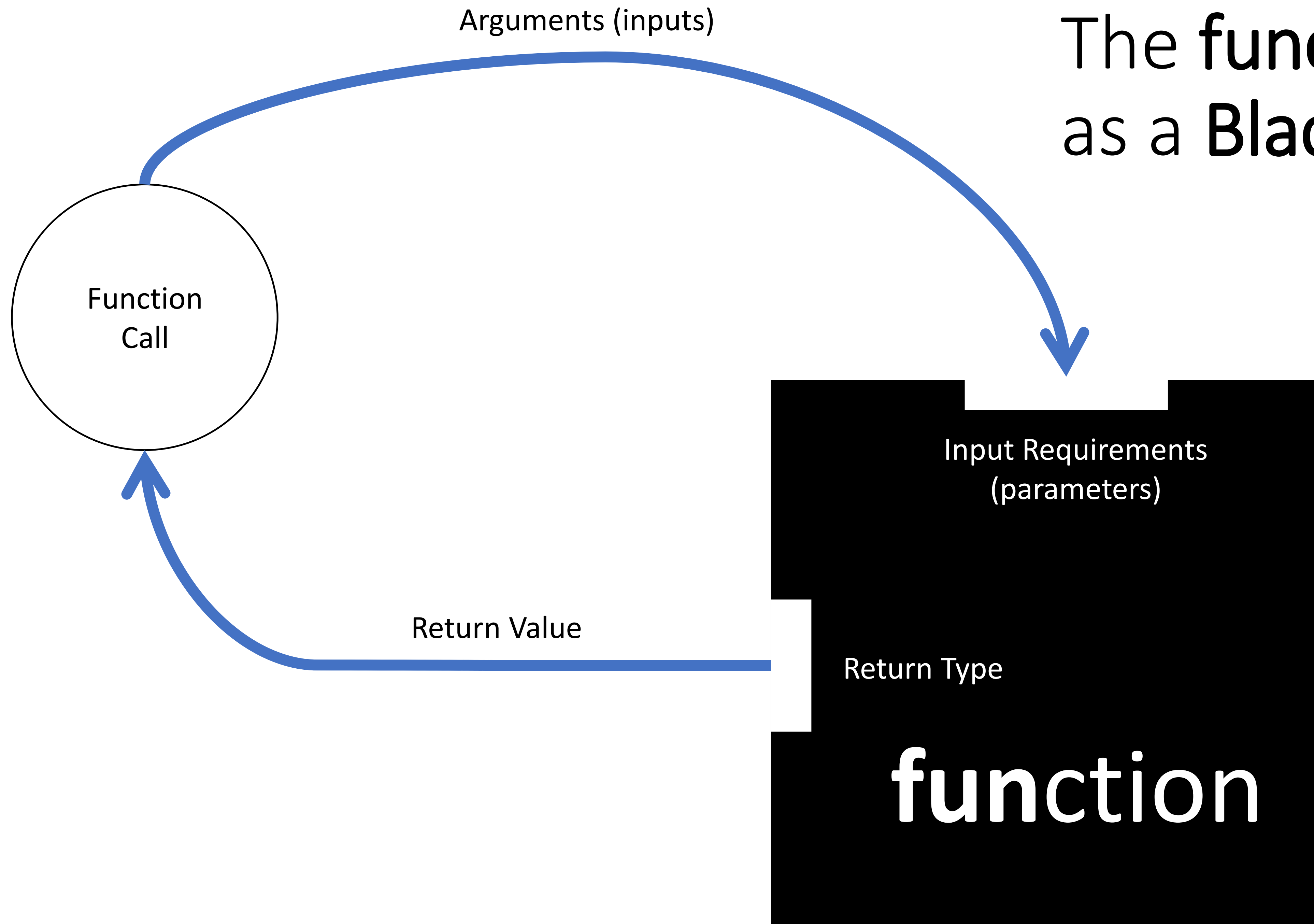
- A function will **never** complete more than one **`return`** statement in any given function call.

The **function** as a **Black Box**

- Once a function is correctly implemented, we can think of it as a **"black box"**
- We do not need to know or see what happens inside of the black box...
  that's magic
- All we need to know is:
  1. What inputs does it need?
  2. What does it return back to us?



The **function** as a **Black Box**



Challenge Question #2: What is **printed** when a() is called?

```
let a = (): string => {
   print("a");
   let x = b(2);
   print("a");
   print(x);
   return "a";
};

let c = (n: number): number => {
   print("c");
   return n * 2;
};

let b = (n: number): number => {
   print("b");
   let result = c(n + 1);
   print("b");
   return result;
};
```

Procedures – Functions that return nothing.

- There are times when it's useful to have a function that performs a set of steps but doesn't actually result back in a value in your program

- The **`print`** Function is the perfect example of a procedure
  - What does calling the **`print`** function return?
  - Nothing! It is a procedure the results in output to the screen.

- Procedures are commonly used to evoke effects *outside* itself
  - To make data or graphics appear on a screen
  - To save data to a file
  - To send data to another computer over the internet

- A procedure is another name for a function whose return type is **`void`**

Playing with Graphical Procedures

- Today we'll introduce a simple graphics library called Turtle Graphics
  - It's a style of teaching introductory computer science that dates back to 1967!

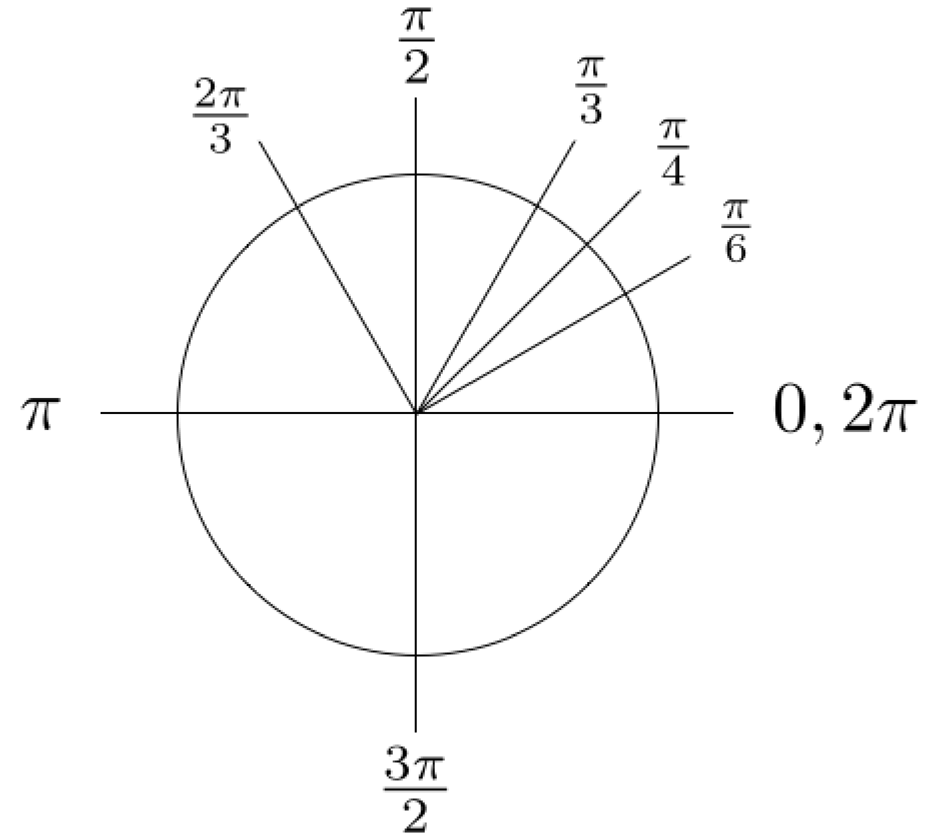
- We have a number of procedures available to us to guide an invisible "turtle" on the screen who is dragging around a marker…

`forward(n: number): void` – Moves the turtle forward by **`n`** pixels
`left(rad: number): void` – Turns the turtle left by **`rad`** in radians
`right(rad: number): void` – Turns the turtle right by **`rad`** in radians

- You can import these functions by:
  - **`import { forward, left, right } from "introcs/turtle";`**

Hands-on: Draw a Square

- Open 05 / 00-turtle-graphics-app.ts
- Your goal: draw a square whose side length is 50.
- You should do this by calling the imported procedures:
  - **`forward(n: number): void`**
  - **`left(radians: number): void`**
- You can write **`Math.PI`** to access the value of pi
- Check-in on PollEv.com/compunc
- Complete? Try doing this with a **`while`** loop instead!



```
let i = 0;
while (i < 4) {
    forward(50);
    left(Math.PI / 2);
    i++;
}
```

Hands-on: Draw a Parametric Star

- Open 01-star-app.ts

- At TODO #0 draw a parametric "star" by:
  - Writing a **`while`** loop that iterates **`points`** parameter number of times
    - Don't forget to increment your counter variable inside the loop!
  - Moves **`forward`** by the **`diameter`** parameter amount
  - Turns **`left`** by the **`angle`** parameter amount

- At TODO #1 call the star function ("procedure") with the following parameters:
  - 5 for the points argument
  - `100` for the diameter argument
  - `4 / 5 * Math.PI` for the angle argument

- Done? Check-in on PollEv.com/compunc. Try playing around with different arguments!

```
export let main = async () => {

    // TODO #1 - Call the star procedure
    star(5, 100, 4 / 5 * Math.PI);

};

let star = (points: number, diameter: number, angle: number): void => {
    // TODO #0 - Draw a star!
    let i = 0;
    while (i < points) {
        forward(diameter);
        left(angle);
        i++;
    }
};
```

Follow Along: Placing Stars

- Open 02-starry-night-app.ts

- Let's add a procedure to place a randomly generated star at some x, y coordinate

- The moveTo procedure jumps the turtle to some x, y coordinate on the screen without drawing a line

```
moveTo(x, y);
let points = 5;
let diameter = 100;
let angle = 4 / 5 * Math.PI;
star(points, diameter, angle);
```