

# Expressions and Functions that Return

Lecture 04

Step 1: Open **VSCode** and its Integrated Terminal

Step 2: **npm run pull**

Step 3: **npm run start**

Step 4: Open another tab to **[pollev.com/comp110](https://pollev.com/comp110)**

# UTA Teaching Teams Assigned

- You can contact your teaching team from My110
  - Please only contact via the My110 page
- Your first line of defense for questions is your "Teaching Team"
- If a question takes more than 5 minutes to answer, or will require back-and-forth, UTAs are instructed to guide you to office hours
  - Questions specific to code you are writing on a problem set should be addressed in office hours
- For all COMP110 e-mail
  - 24h response goal
  - E-mails sent after 8pm will not be responded to until next day...

**start and submit early!**

# Worksheet

- Due **TONIGHT** at 11:59pm!
- Instructions for registering for Gradescope and printing/scanning your worksheet are on the home page of [COMP110.com](https://COMP110.com)
- The purpose of worksheets is to help you prepare for exams
  - Future worksheets we will require handwritten submissions
- Next worksheet and problem set will be posted by tomorrow.

# HACK110 - Hackathon

- Mark your calendars for **November 17th** from **7pm until 7am**
- OPTIONAL Hackathon
  - Continuing to COMP401? Highly encouraged!

Warm-up #1) What is the output of these programs?

```
let x: number = 13;
if (x < 18) {
    print("A");
}

if (x === 13) {
    print("B");
} else {
    print("C");
}
```

```
let x: number = 13;
if (x < 18) {
    print("A");
} else {
    if (x === 13) {
        print("B");
    } else {
        print("C");
    }
}
```

Warm-up #1) What is the output of these programs?

```
let x: number = 13;
if (x < 18) {
    print("A");
}

if (x === 13) {
    print("B");
} else {
    print("C");
}
```

A, B

```
let x: number = 13;
if (x < 18) {
    print("A");
} else {
    if (x === 13) {
        print("B");
    } else {
        print("C");
    }
}
```

A

# Expressions

- Expressions are a fundamental building block in programs
- Analogous to the idea of clauses in English
  - Single clause sentence:  
*"I am a student."*
  - Multiple clause sentence:  
*"I am a student and I am currently sitting in COMP110."*
- Just like ***sentences*** can become *more expressive* through the creative use of clauses...
- The ***statements*** we make in programming languages become *more expressive* through the creative use of clauses...
- This is what gives us so many (*unlimited!*) ways to write programs which all achieve the same goal.

# Expressions

There are two **big ideas** behind expressions

1. *Every expression simplifies to a single value*
  - Thus, every expression has a *single result type*.
  - This occurs *only* when the program runs and the computer reaches that line of code in the program.
2. Anywhere you can write an expression, you can choose any expression you'd like as long as their types match



# Expressions – Some examples we've seen...

Expression	Resulting Type	Resulting Value	Expression Name
"Hello, World"	string	"Hello, World"	string Literal
length	number	?	Variable Reference
length * length	number	?	2x Variable Reference Arithmetic Operation
"Area " + area	string	?	Variable Reference Concatenation Operation

# Where have we *used* expressions?

- Assignment operator:

```
let <name>: <type> = <expression of same type>;
```

- We are able to assign *any* of the expressions below because each results in a single *number* value:

```
let x: number = 1;  
let y: number = x + 1;  
let cubeY: number = y * y * y;
```

- Notice that we are combining *multiple* expressions in the same line.
- After each line completes, the declared variable has a *single* value.

# Singular Expressions

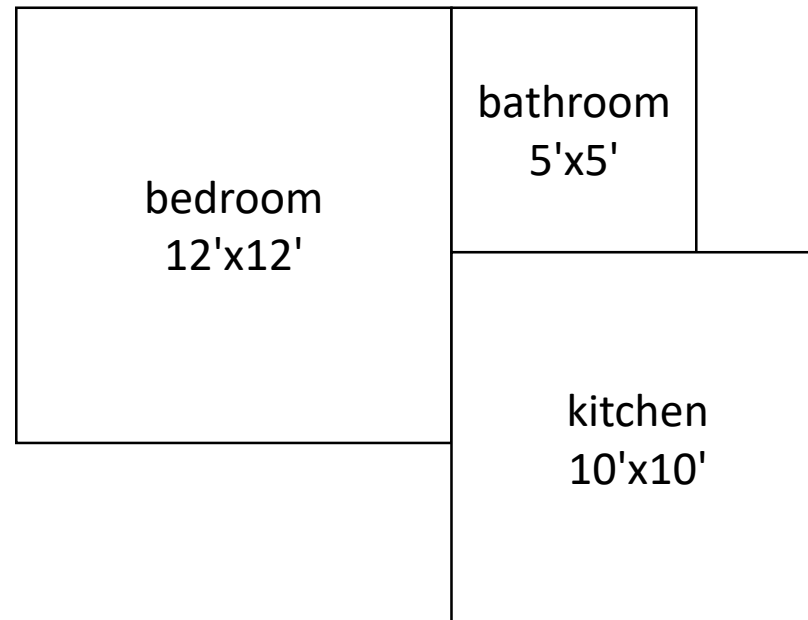
- Literal Values
  - 1, 3.14, true, "hi"
- Variable Access
  - x, compCourseNumber
- "Unary" operators (-)
  - -x

# Compound Expressions

- Operators
  - Arithmetic
  - Concatenation
  - Equality
    - ==
    - !=
  - Relational
    - >
    - >=
- Function Calls
  - Functions that **return!**

# What if we wanted to use a single function's computation as part of a larger computation?

- Last week we wrote a function to calculate the area of a square
- What if we wanted to calculate the **total** area of a "house" with three square-shaped rooms?



We can't reuse our old **void** function...

```
function squareArea(length: number): void {  
    let area: number = length * length;  
    print("The area is " + area);  
}
```

// The following line has an ERROR:

```
let sum: number = squareArea(5) + squareArea(10) + squareArea(12);  
// "Operator + cannot be applied to types void and void"
```

Calling a **void** function does *return a value* the program can later make use of.

# Follow-along:

Let's write a function that *returns* a number

```
1. function area(length: number): number {  
2.   return length * length;  
}
```

1. Notice the function's return type is specified as **number**, *not void*!
  2. The **return statement** inside of the function computes a **number**.
- Notice: this function does not *print anything*, but it "returns" or "gives back" a value we can use later in our program:

```
let sum: number = area(5) + area(10) + area(12);
```

# Function Syntax – Return Type

```
function <name>(<parameters>): <returnType> {  
    <function body statements>  
}
```

- A function's **return type** is like a variable declaration's type.
  - It can be a *string*, *number*, *boolean*, and complex types we'll see soon.
- It means *calling this function will give you back a value of this type which you can later use*
- Functions with a non-void return type **require** a matching **return statement** inside of their function body statements.

# The **return** Statement

- General form:

**return** <expression>;

- Expression's type ***must* match the return type** of its function
- Every function that returns a value must have at least one **return** statement
- IMPORTANT: As soon as *any* **return** statement is reached the function call is complete.
  - The computer evaluates the expression and returns the value immediately to its bookmark.
  - ***This is true even if the return statement is inside of an if-then-else statement!***



# Functions with Multiple Inputs

- What if we wanted a function that, given 2 numbers, will return the largest?
- Let's name the function **max2**
- Our desired *usage* via function calls is:

```
print(max2(3, 5));  
// Prints: 5
```

```
print(max2(5, 3));  
// Prints: 5
```

# Function Syntax – Multiple Parameters

```
function <name>(<parameters>): <returnT> {  
    <function body statements>  
}
```

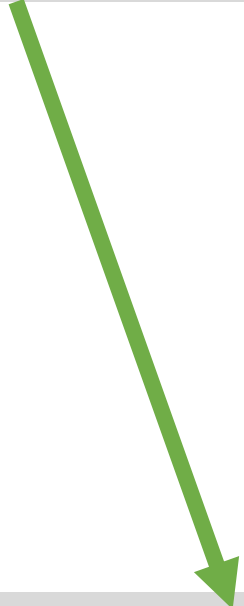
- A function can declare **multiple parameters** when it needs more than one piece of input information
- Each parameter has a name and a type separated by a comma:  
    <name>: <type>, <name>: <type>
- For example:  

```
function max2(a: number, b: number): number {  
        // Elided  
    }
```

# Function Calls – Multiple Arguments & Parameters (1/4)

When a function ***call*** is encountered...

**max2(3, 5)**



1. **Is there a function definition with this name?**
2. Do the number and types of arguments match the number and types of parameters 1-to-1?
3. Drop bookmark. Assign arguments to parameters in the same order each is given.
4. Jump into function body block.

```
function max2(a: number, b: number): number {  
    // Elided  
}
```

# Function Calls – Multiple Arguments & Parameters (2/4)

When a function *call* is encountered...

`max2(3, 5)`

1. Is there a function definition with this name?
2. **Do the number and types of arguments match the number and types of parameters 1-to-1?**
3. Drop bookmark. Assign arguments to parameters in the same order each is given.
4. Jump into function body block.

```
function max2(a: number, b: number): number {  
    // Elided  
}
```

# Function Calls – Multiple Arguments & Parameters (3/4)

When a function *call* is encountered...

`max2(3, 5)`

1. Is there a function definition with this name?
2. Do the number and types of arguments match the number and types of parameters 1-to-1?
3. **Drop bookmark. Assign arguments to parameters in the same order each is given.**
4. Jump into function body block.


```
function max2(a: number, b: number): number {  
  a = 3; // Reminder: these assignments happen invisibly  
  b = 5; // when the program is running.  
  // Elided  
}
```

# Function Calls – Multiple Arguments & Parameters (4/4)

When  function **call** is encountered...

`max2(3, 5)`

1. Is there a function definition with this name?
2. Do the number and types of arguments match the number and types of parameters 1-to-1?
3. Drop bookmark. Assign arguments to parameters in the same order each is given.
4. **Jump into function body block.**

```
function max2(a: number, b: number): number {  
    a = 3;    // Reminder: these assignments happen invisibly  
    b = 5;    // when the program is running.  
     // Program continues here  
}
```

# Hands-on #1) Implement the max2 function

- We're trying to calculate the price of 2 sushi rolls at SPICY 9
  - They have a BOGO deal where you pay the price of the more expensive roll and the other is free
- Your objective:
  - Write an if-then-else statement in the max2 function with the following logic

```
IF parameter a IS GREATER THAN parameter b
    THEN return a
    OTHERWISE return b
```

- Test by changing the prices of the two rolls
- Check-in on [PollEv.com/comp110](https://pollev.com/comp110) when complete

# Return Semantics: Consider the following **function**

- Consider the **max2** function to the right
- Its purpose is to return the greater value of the parameters **a** and **b**
- *Does it?* What happens when **a** is greater?

```
function max2(a: number, b: number): number {  
    if (a > b) {  
        return a;  
    }  
    return b;  
}
```



# Returning from a function

```
let result 1number;  
result = max(10, 5);
```

1. The **max2** function is called with arguments: **10, 5**
2. The processor jumps to max2 function.
  - if (a > b) evaluates to true, enters **then block**
3. **return** Statement encountered.  
Expression **a** evaluates to **10**. The function call is complete and this value is returned to step 4.
4. Processor jumps back to bookmark it left at #1 and "max2(10, 5)" evaluates to **10**.

```
function max2(a: number, b: number): number {  
  2if (a > b) {  
    return a; 3  
  }  
  
  return b;  
}
```

## Parameters

a	10
b	5

# Every function *call* can *return only once*

- Every function call is an *expression*. By definition, an expression is something that *evaluates to a single value*.
- A function *may* contain many return statements
- A function *may* contain a return statement inside of a loop (coming next)
- As soon as the computer reaches *any* return statement *once* within a function, that function call is completed and the value is returned.

# A function call is an expression

- Because the function **area** returns a *number*, we can ***call*** the area function *anywhere* a number expression is used.

```
print(area(3));
```

```
let foo: number = area(3);
```

```
print(area(area(3)));
```

- This generalizes: A function's **return type** defines the type of **calls** to it.

# Hands-on #2: max3

- In the file **lec 04 / 02-expression-composition-app.ts**
- Try to implement the max3 function WITHOUT using an if-then-else
- Hint: Follow the strategy in the comments and remember you can use the max2 function defined above it!
- Test by changing around variable values in the main function
- Check-in on [PollEv.com/comp110](https://pollev.com/comp110)

```
function max3(a: number, b: number, c: number): number {  
    // 1. Declare and initialize a variable that holds the max of a, b  
    // 2. Return the max of the variable from step 1 and c  
    let maxAB: number = max2(a, b);  
    return max2(maxAB, c);  
}
```

# Composing Functions

- General Principle of Programming:

**Writing many small functions, each with a single purpose, is better than writing few large functions, each with complex purposes.**

- Why? Because we can more easily reuse small, simple functions just like we did in `max3`.
- Smaller, single-purposed functions are easier to compose to solve novel problems.

# Review & Upcoming Assignments