

THE NOT-SO-WIDE WORLD OF

FUNCTIONS

NON-RHETORICAL:

WHY DO WE GIVE THINGS NAMES?

FUNCTIONS ARE THE PRIMARY GROUPING MECHANISM FOR CODE.

- ▶ “What should happen when a button is clicked?”
- ▶ “What does it mean to ‘send a chat message’?”
- ▶ All functions can have:
 - ▶ **A name:** how we refer to the function
 - ▶ **Inputs:** parameters
 - ▶ **A body:** what the function actually does
 - ▶ **Side effects:** changes in your application perceivable outside of the function
 - ▶ **Outputs:** return value (or undefined)

ALL FUNCTIONS FOLLOW THE SAME RULES.

- ▶ Named and anonymous functions
- ▶ Pure and impure functions
- ▶ Callbacks
- ▶ Closures
- ▶ Constructors*
- ▶ Function properties
- ▶ IIFE's



THESE ARE WAYS WE
CAN USE FUNCTIONS,
NOT DIFFERENT
TYPES OF
FUNCTIONS.

** actually there are small differences*

FUNCTION NAMES

- ▶ **We name things so that we can refer to them later. That is the only reason.**
- ▶ If we do not need to refer to them later, we do not need to name them. Anonymous functions are unnamed functions.
- ▶ Anonymous functions are **never** required, but having a bunch of names floating around can be confusing for human-folk.

```
// Named so we can call it later
function showPerson(person) {

}

showPerson(personObject);

// Anonymous function; we'll never need to
// refer to it again.
window.addEventListener('load', function () {
    // do things...
});
```

FUNCTION INPUTS (PARAMETERS)

- ▶ There are only two ways to access information inside of a function: parameters (direct inputs) and variables available through the parent scope.
- ▶ A function can have zero or more inputs.
Parameters exist to make functions **general**: if we design functions to accept *types* of inputs instead of *specific* inputs, we can use the function in more than one place.
- ▶ If you want to make a function reusable, parameters are extremely convenient for abstracting the specific data into general types.

```
/* No params */
function init() {
  // set up the app on page load
}

const init = () => {
  // same as above, es6 syntax
};

/* Three params */
let findOldest = function (first, second,
third) {
  // some code to find the oldest
}
```

FUNCTION BODY

- ▶ All functions have bodies; the purpose of functions is to define processes, and a process with zero steps is not a process.
- ▶ Variables that are not required outside of a function should be defined in the function body so that the values can be cleared when the function completes.

```
/* Every function worth its rent has a body */  
function render(todos) {  
  // start of body  
  for (let i = 0; i < todos.length; i++) {  
    let title = todos[i].title.toUpperCase();  
    let from = todos[i].from;  
  
    // render to the DOM  
  }  
  
  // end of body  
}
```

SIDE EFFECTS

- ▶ Functions are either called because **you want a return value** or **you want a side effect**. Side effects are changes that are perceivable outside of the function.
- ▶ Examples:
 - ▶ Modifying the DOM
 - ▶ Sending AJAX requests
 - ▶ Changing global variables

```
/* Changing a global variable = side effect */  
let count = 0;  
  
function increment() {  
    count++;  
}  
  
/* Updating the DOM */  
function updateTheButton(num) {  
    document.querySelector('#the-  
btn').textContent = num;  
}
```


OUTPUTS (RETURN VALUES)

- ▶ Functions are either called because **you want a return value** or **you want a side effect**. A return value is the 'result' provided by running a function.
- ▶ A return value allows us to take something from a function's scope and 'return' it to the parent that called the function.
- ▶ **Return values are not required.** You only need to return a value if you want to use the returned value outside of the function.

```
/* No return value means 'undefined' is
returned */
function render() {
  // ...dom stuff...
  // no return
}

let out = render();
console.log(out); // undefined

/* Returning gives us access to local
information outside of this scope */
function tallestMountain(montanas) {
  // ... tallest mountain algo ...
  return tallest;
}

let mtn = tallestMountain([1, 6, 91, 44]);
console.log(mtn); // 91
```

THINKING ABOUT

APPLICATIONS OF FUNCTIONS

(VERY WORTH THINKING HARD ABOUT, BTW)

An **anonymous function** is a function with no name.

WHEN WOULD YOU USE AN ANONYMOUS FUNCTION?

A **pure function** is a function with no side effects.

WHY WOULD YOU EVER PREFER TO WRITE PURE FUNCTIONS?

A **callback function** is a function passed as an argument to another function intended to be 'called back' at another time.

WHEN WOULD YOU WANT TO USE A CALLBACK FUNCTION?

BONUS: CALLBACKS ARE OFTEN ANONYMOUS. WHY?

A **constructor** is a function whose sole purpose is to create a specific kind of object.

WHEN WOULD YOU USE A CONSTRUCTOR?

DEEP DIVE: CLOSURES

- ▶ A closure is a normal function in every way.
- ▶ What's different about closures is that certain **variables** within the functions scope are referenced in the returned object, so they *can't be deleted or the returned object would be incomplete*.
- ▶ You create a closure by *returning a reference to a local variable*.
- ▶ We use closures whenever it's helpful to retain a known value from the current scope.

```
function pokemon() {  
  // Closure because 'all' is required  
  // after function ends.  
  let all = [];  
  
  // Pretend: make ajax request to get all  
  // pokemon. Populate 'all' array.  
  
  return {  
    /* Pikachu, Bulbasaur, etc */  
    findByName: function (name) {  
      return all.find(poke => poke.name ===  
name);  
    },  
  
    /* Fire, water, rock, etc */  
    findByElement: function (element) {  
      return all.filter(poke => poke.element  
=== element);  
    }  
  };  
}  
  
let pokedex = pokemon();  
// one object  
let fav = pokedex.findByName('Pidgy');  
// array of objects  
let scum = pokedex.findByElement('Rock');
```

PROJECT

PERSON FINDER

APPENDIX: CONSTRUCTORS

- ▶ Constructors actually are a little bit different than all other functions, but they're different in very predictable ways.
- ▶ The sole purpose of a constructor is to ***create objects of a particular type, and all of their differences are designed to make this easier.***
- ▶ Differences:
 1. They have to be called with the `new` keyword.
 2. `this` is automatically set to a new object. It is an empty object except for that fact that it shares the constructors prototype.
 3. It will automatically `return this` at the end, even if you don't say to.

APPENDIX: CONSTRUCTORS VS CLOSURES

- ▶ Constructors and closures can be used to achieve very similar things: storing similar information together. Sometimes you can choose either.
- ▶ **You might choose to use a constructor** because constructors make it easy to take advantage of prototypes.
- ▶ **You might choose to use a closure** because closures make it impossible to access internal information directly ('private' properties).