First-class functions

First-class functions

Functions in JavaScript are objects.

- They can be saved in variables
- They can be passed as parameters
- They have properties, like other objects
- They can be defined without an identifier

(This is also called having **first-class functions**, i.e. functions in JavaScript are "first-class" because they are treated like any other variable/object.)

Back to the veeeeery basics

What is code?

- A list of instructions your computer can execute
- Each line of code is a **statement**

What is a function?

- A labeled group of **statements**
- The statements in a function are executed when the function is invoked

What is a variable?

- A labeled piece of **data**

Recall: Objects in JS

Objects in JavaScript are sets of property-value pairs:

```
const bear = {
  name: 'Ice Bear',
  hobbies: ['knitting', 'cooking', 'dancing']
};
```

- Like any other value, Objects can be saved in **variables.**
- Objects can be passed as parameters to functions

Back to the veeeeery basics

What is code?

- A list of instructions your computer can execute
- Each line of code is a **statement**

What is a function?

- A labeled group of **statements**
- The statements in a function are executed when the function is invoked

What is a variable?

- A labeled piece of **data**

What could it mean for a function to be an object, i.e. a kind of data?

Function variables

You can declare a function in several ways:

```
function myFunction(params) {
}

const myFunction = function(params) {
};

const myFunction = (params) => {
};
```

Function variables

```
function myFunction(params) {
}

const myFunction = function(params) {
};

const myFunction = (params) => {
};
```

Functions are invoked in the same way, regardless of how they were declared:

```
myFunction();
```

```
const x = 15;
let y = true;

const greeting = function() {
  console.log('hello, world');
}
```

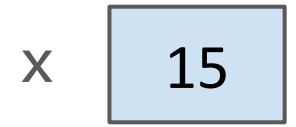
"A function in JavaScript is an object of type `Function`"

In the interpreter's memory:

```
const x = 15;
let y = true;

const greeting = function() {
  console.log('hello, world');
}
```

"A function in JavaScript is an object of type `Function`"

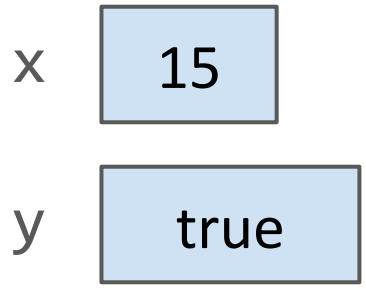
In the interpreter's memory:




```
const x = 15;
let y = true;

const greeting = function() {
  console.log('hello, world');
}
```

"A function in JavaScript is an object of type `Function`"

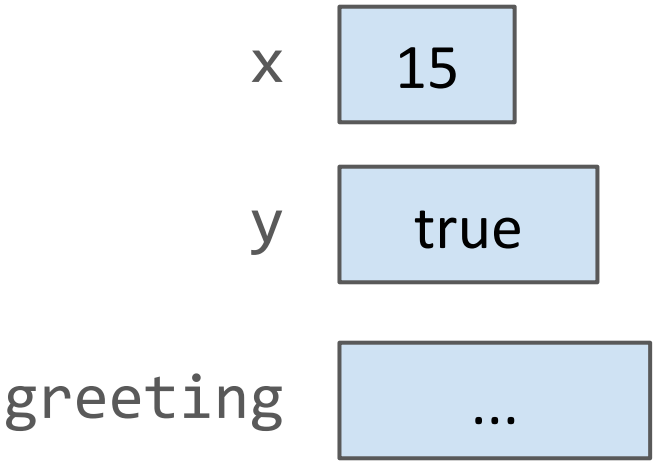
In the interpreter's memory:




```
const x = 15;
let y = true;

const greeting = function() {
  console.log('hello, world');
}
```

"A function in JavaScript is an object of type `Function`"

In the interpreter's memory:




```
const x = 15;
let y = true;

const greeting = function() {
  console.log('hello, world');
}
```

"A function in JavaScript is an object of type `Function`"

What this really means:

- When you declare a function, there is an object of type `Function` that gets created alongside the labeled block of executable code.

Function properties

```
const greeting = function() {
  console.log('hello, world');
}

console.log(greeting.name);
console.log(greeting.toString());
```

When you declare a function, you create an object of type `Function`, which has properties like:

- `name`
- `toString`

CodePen

`Function` properties

```
const greeting = function() {
  console.log('hello, world');
}

greeting.call();
```

`Function` objects also have a `call` method, which invokes the underlying executable code associated with this function object.

CodePen

Function properties

```
const greeting = function() {
  console.log('hello, world');
}

greeting.call();
greeting();
```

() is an operation on the Function object (spec)

- When you use the () operator on a Function object, it is calling the object's call() method, which in turn executes the function's underlying code

Code vs `Functions`

Important distinction:

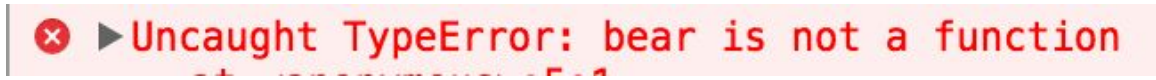
- Function, the executable code
  - A group of instructions to the computer
- `Function`, the object
  - A JavaScript object, i.e. a set of property-value pairs
  - `Function` objects have executable code associated with them
  - This executable code can be invoked by
    - *functionName*`();` *or*
    - *functionName*`.call();`

Note: `Function` is special

Only `Function` objects have executable code associated with them.

- Regular JS objects **cannot** be invoked
- Regular JS objects **cannot** be given executable code
  - I.e. you can't make a regular JS object into a callable function

```
const bear = {
  name: 'Ice Bear',
  hobbies: ['knitting', 'cooking', 'dancing']
};
bear(); // error!
```



Function Objects vs Objects

```
function sayHello() {
  console.log('Ice Bear says hello');
}

const bear = {
  name: 'Ice Bear',
  hobbies: ['knitting', 'cooking', 'dancing'],
  greeting: sayHello
};
bear.greeting();
```

CodePen

But you can give your object `Function` properties and then invoke those properties.

Function Objects vs Objects

```
function sayHello() {
  console.log('Ice Bear says hello');
}

const bear = {
  name: 'Ice Bear',
  hobbies: ['knitting', 'cooking', 'dancing'],
  greeting: sayHello
};
bear.greeting();
```

CodePen

The `greeting` property is an object of `Function` type.

Why do we have `Function` objects?!

Callbacks

`Function` objects **really** come in handy for event-driven programming!

```
function onDragStart(event) {
    ...
}
dragon.addEventListener('pointerdown', onDragStart);
```

Because every function declaration creates a `Function` object, we can pass `Functions` as parameters to other functions.

Creating functions within functions

Functions that create functions

In JavaScript, we can **create** functions from within functions (CodePen).

```
function printMessage(birthYear) {
  function getLabel(age) {
    if (age < 2) {
      return "baby";
    }
    if (age < 4) {
      return "toddler";
    }
    if (age < 13) {
      return "kid";
    }
    if (age < 20) {
      return "teenager";
    }
    return "grown-up";
  }

  const ageThisYear = 2017 - birthYear;
  const label = getLabel(ageThisYear);
  console.log('You are a ' + label + ' this year.');
}

printMessage(2005);
```

Functions that create functions

In JavaScript, we can **create** functions from within functions (CodePen).

```
function printMessage(birthYear) {
  function getLabel(age) {
    if (age < 2) {
      return "baby";
    }
    if (age < 4) {
      return "toddler";
    }
    if (age < 13) {
      return "kid";
    }
    if (age < 20) {
      return "teenager";
    }
    return "grown-up";
  }
```

A function declared within a function is also known as a **closure**.

Scope of closures

```
function printMessage(birthYear) {
  if (true) {
    function getLabel(age) {
      if (age < 2) {
        return "baby";
      }
      if (age < 4) {
        return "toddler";
      }
      if (age < 13) {
        return "kid";
      }
      if (age < 20) {
        return "teenager";
      }
      return "grown-up";
    }
  }

  const ageThisYear = 2017 - birthYear;
  const label = getLabel(ageThisYear);
  console.log('You are a ' + label + ' this year.');
}
```

Functions declared with `function` (or `var`) have function scope.

- Can be referenced anywhere in the function after declaration

This example works:

Console

```
"You are a kid this year."
```

Scope of closures

```
function printMessage(birthYear) {
  function getLabel(age) {
    if (age < 2) {
      return "baby";
    }
    if (age < 4) {
      return "toddler";
    }
    if (age < 13) {
      return "kid";
    }
    if (age < 20) {
      return "teenager";
    }
    return "grown-up";
  }

  const ageThisYear = 2017 - birthYear;
  const label = getLabel(ageThisYear);
  console.log('You are a ' + label + ' this year.');
}

printMessage(2005);
const label = getLabel(8);
```

Functions declared with `function` (or `var`) have function scope.

- Cannot be referenced outside the function

This example doesn't work:

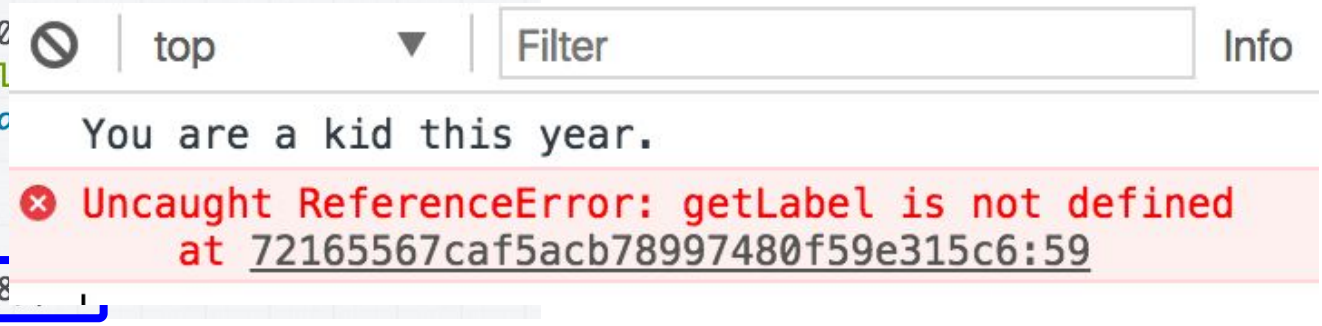
Scope of closures

```
function printMessage(birthYear) {
  function getLabel(age) {
    if (age < 2) {
      return "baby";
    }
    if (age < 4) {
      return "toddler";
    }
    if (age < 13) {
      return "kid";
    }
    if (age < 20) {
      return "teenager";
    }
    return "grown-up";
  }

  const ageThisYear = 20
  const label = getLabel
  console.log('You are a
}

printMessage(2005);
const label = getLabel(8
```

Functions declared with `function` (or `var`) have function scope.

- Cannot be referenced outside the function

This example doesn't work:



Scope of closures

```
function printMessage(birthYear) {
  if (true) {
    const getLabel = function(age) {
      if (age < 2) {
        return "baby";
      }
      if (age < 4) {
        return "toddler";
      }
      if (age < 13) {
        return "kid";
      }
      if (age < 20) {
        return "teenager";
      }
      return "grown-up";
    }
  }

  const ageThisYear = 2017 - birthYear;
  const label = getLabel(ageThisYear);
  console.log('You are a ' + label + ' this year.');
}
```

```
▶Uncaught ReferenceError: getLabel is not defined
    at printMessage (pen.js:22)
    at pen.js:26
```

Functions declared with `const` or `let` have block scope

- Cannot be referenced outside of the block.

This example doesn't work:

Functions that return functions

In JavaScript, we can **return** new functions as well.
(We kind of knew this already because `bind` returns a new function.)

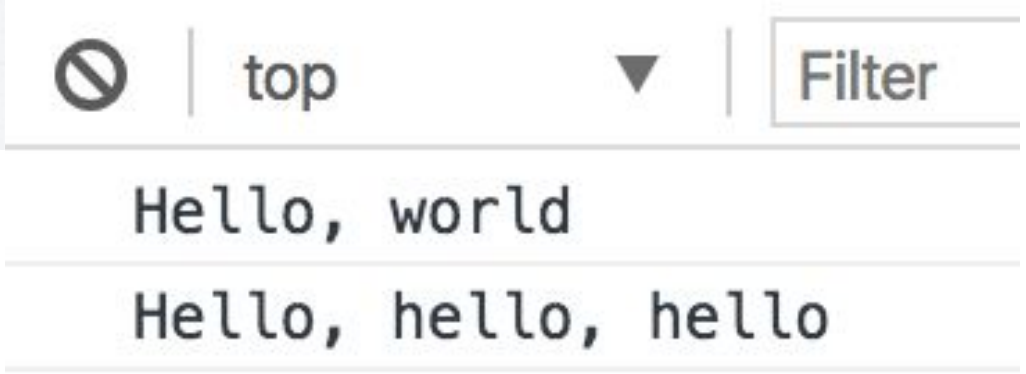
```
function makeHelloFunction(name) {
  const greeting = function() {
    console.log('Hello, ' + name);
  };
  return greeting;
}

const helloWorld = makeHelloFunction('world');
const hello3 = makeHelloFunction('hello, hello');

helloWorld();
hello3();
```

CodePen

Functions that create functions

```
function makeHelloFunction(name) {
  const greeting = function() {
    console.log('Hello, ' + name);
  };
  return greeting;
}

const helloWorld = makeHelloFunction('world');
const hello3 = makeHelloFunction('hello, hello');

helloWorld();
hello3();
```

CodePen



Closure: an inner function

```
function makeHelloFunction(name) {
  const greeting = function() {
    console.log('Hello, ' + name);
  };
  return greeting;
}
```

- When you declare a function inside another function, the inner function is called a **closure**.

Closure: an inner function

```
function makeHelloFunction(name) {
  const greeting = function() {
    console.log('Hello, ' + name);
  };
  return greeting;
}
```

- Within a closure, you can reference variables that were declared in the outer function, and those variables **will not go away** after the outer function returns.

Functions that create functions

```
function makeHelloFunction(name) {
  const greeting = function() {
    console.log('Hello, ' + name);
  };
  return greeting;
}

const helloWorld = makeHelloFunction('world');
const hello3 = makeHelloFunction('hello, hello');

helloWorld();
hello3();
```

The scope of `greeting` is only in the `makeHelloFunction` function, as well as the scope of `name`...

Functions that create functions

```
function makeHelloFunction(name) {
  const greeting = function() {
    console.log('Hello, ' + name);
  };
  return greeting;
}

const helloWorld = makeHelloFunction('world');
const hello3 = makeHelloFunction('hello, hello');

helloWorld();
hello3();
```

But the `makeHelloFunction` function returns a reference to the function, which is an object, so the function object doesn't go away

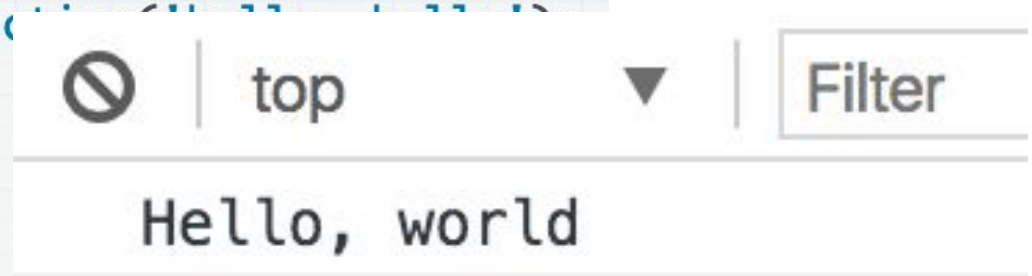
Functions that create functions

```
function makeHelloFunction(name) {
  const greeting = function() {
    console.log('Hello, ' + name);
  };
  return greeting;
}

const helloWorld = makeHelloFunction('world');
const hello3 = makeHelloFunction('hello, hello');

helloWorld();
hello3();
```

And the function object keeps a reference to the name parameter, so that when the created function is called...

Functions that create functions

```
function makeHelloFunction(name) {
  const greeting = function() {
    console.log('Hello, ' + name);
  };
  return greeting;
}

const helloWorld = makeHelloFunction('world');
const hello3 = makeHelloFunc

helloWorld();
hello3();
```



… we see that the new function returned from `makeHelloFunction` still has access to the `name` variable.

Functions that create functions

```
function makeHelloFunction(name) {
  const greeting = function() {
    console.log('Hello, ' + name);
  };
  return greeting;
}

const helloWorld = makeHelloFunction('world');
const hello3 = makeHelloFunction('hello, hello');

helloWorld();
hello3();
```

The idea of constructing a new function that is "partially instantiated" with arguments is called **currying**. (article)

Anonymous functions

We do not need to give an identifier to functions.

When we define a function without an identifier, we call it an **anonymous function**

- Also known as a **function literal**, or a **lambda function**

```
function makeHelloFunction(name) {
  const greeting = function() {
    console.log('Hello, ' + name);
  };
  return greeting;
}
```

Anonymous functions

We do not need to give an identifier to functions.

When we define a function without an identifier, we call it an **anonymous function**

- Also known as a **function literal**, or a **lambda function**

```
function makeHelloFunction(name) {
  return function() {
    console.log('Hello, ' + name);
  };
}
```

CodePen

Gotchas and style notes

Recall: Present example

```
class Present {
  constructor(containerElement, giftSrc) {
    this.containerElement = containerElement;
    this.giftSrc = giftSrc;

    this._openPresent = this._openPresent.bind(this);

    const image = document.createElement('img');
    image.src = OUTSIDE_IMAGE_URL;
    image.addEventListener('click', this._openPresent);
    this.containerElement.append(image);
  }

  _openPresent(event) {
    const image = event.currentTarget;
    image.src = this.giftSrc;
  }
}
```

[CodePen]

We implemented a Present class that had a separate **`openPresent`** method.

```
class Present {
  constructor(containerElement, giftSrc) {
    this.containerElement = containerElement;
    this.giftSrc = giftSrc;

    const image = document.createElement('img');
    image.src = OUTSIDE_IMAGE_URL;
    image.addEventListener('click', function(event) {
      const image = event.currentTarget;
      image.src = this.giftSrc;
    });
    this.containerElement.append(image);
  }
}
```

What would happen if we defined the click event handler directly in the call to `addEventListener` (CodePen)?

```
class Present {
  constructor(containerElement, giftSrc) {
    this.containerElement = containerElement;
    this.giftSrc = giftSrc;

    const image = document.createElement('img');
    image.src = OUTSIDE_IMAGE_URL;
    image.addEventListener('click', function(event) {
      const image = event.currentTarget;
      image.src = this.giftSrc;
    });
    this.containerElement.append(image);
  }
}
```

We didn't bind `this`, so we have a bug:

`this` is the `img` instead of the `Present` object.

```
class Present {
  constructor(containerElement, giftSrc) {
    this.containerElement = containerElement;
    this.giftSrc = giftSrc;

    const image = document.createElement('img');
    image.src = OUTSIDE_IMAGE_URL;
    image.addEventListener('click', (function(event) {
      const image = event.currentTarget;
      image.src = this.giftSrc;
    }).bind(this));
    this.containerElement.append(image);
  }
}
```

Fixed CodePen

```
class Present {
  constructor(containerElement, giftSrc) {
    this.containerElement = containerElement;
    this.giftSrc = giftSrc;

    const image = document.createElement('img');
    image.src = OUTSIDE_IMAGE_URL;
    image.addEventListener('click', (function(event) {
      const image = event.currentTarget;
      image.src = this.giftSrc;
    }).bind(this));
    this.containerElement.append(image);
  }
}
```

Fixed CodePen

```
class Present {
  constructor(containerElement, giftSrc) {
    this.containerElement = containerElement;
    this.giftSrc = giftSrc;

    const image = document.createElement('img');
    image.src = OUTSIDE_IMAGE_URL;
    image.addEventListener('click', event => {
      const image = event.currentTarget;
      image.src = this.giftSrc;
    });
    this.containerElement.append(image);
  }
}
```

What would happen if we defined the click event handler like this, with the arrow function instead (CodePen)?

This works! **Why?!**

([CodePen])

```
image.addEventListener('click', event => {
  const image = event.currentTarget;
  image.src = this.giftSrc;
});
```

=> versus function

When you define a function using **function syntax**:

```
const onClick = function() {
    const image = event.currentTarget;
    image.src = this.giftSrc;
};
```

this is will be dynamically assigned to a different value depending on how the function is called, like we've seen before (unless explicitly bound with bind)

=> versus function

When you define a function using **arrow syntax**:

```
const onClick = event => {
    const image = event.currentTarget;
    image.src = this.giftSrc;
};
```

this is **bound** to the value of this in its enclosing context

```
class Present {
  constructor(containerElement, giftSrc) {
    this.containerElement = containerElement;
    this.giftSrc = giftSrc;

    const image = document.createElement('img');
    image.src = OUTSIDE_IMAGE_URL;
    image.addEventListener('click', event => {
      const image = event.currentTarget;
      image.src = this.giftSrc;
    });
    this.containerElement.append(image);
  }
}
```

Since we've used the arrow function in the constructor, the `this` in the enclosing context is the new Present object.

Which is better style?

```
class Present {
  constructor(containerElement, giftSrc) {
    this.containerElement = containerElement;
    this.giftSrc = giftSrc;

    this._openPresent = this._openPresent.bind(this);

    const image = document.createElement('img');
    image.src = OUTSIDE_IMAGE_URL;
    image.addEventListener('click', this._openPresent);
    this.containerElement.append(image);
  }

  _openPresent(event) {
    const image = event.currentTarget;
    image.src = this.giftSrc;
  }
}
```

(A) Explicit event handler

```
class Present {
  constructor(containerElement, giftSrc) {
    this.containerElement = containerElement;
    this.giftSrc = giftSrc;

    const image = document.createElement('img');
    image.src = OUTSIDE_IMAGE_URL;
    image.addEventListener('click', event => {
      const image = event.currentTarget;
      image.src = this.giftSrc;
    });
    this.containerElement.append(image);
  }
}
```

(B) Inline event handler

Callback style

```
image.addEventListener('click', this._openPresent);
```

Version A: Explicit event handler

- Pros:
  - Easier to read
  - More modular
  - Scales better to long functions, several event handlers
- Cons:
  - Because all class methods are public, it exposes the onClick function (which should be private)

Callback style

```
image.addEventListener('click', this._openPresent);
```

Version A: Explicit event handler

- Pros:
  - Easier to read
  - More modular
  - Scales better to long functions, several event handlers
- Cons:
  - Because all class methods are public, it exposes the onClick function (which should be private)
  - Need to `bind` explicitly

Callback style

```
image.addEventListener('click', this._openPresent);
```

Version A: Explicit event handler

- Pros:
  - Easier to read
  - More modular
  - Scales better to long functions, several event handlers
- Cons:
  - Because all class methods are public, it exposes the onClick function (which should be private)
  - Need to `bind` explicitly

Callback style

```
image.addEventListener('click', event => {
  const image = event.currentTarget;
  image.src = this.giftSrc;
});
```

Version B: Inline event handler

- Pros:
  - Does not expose the event handler: function is privately encapsulated
- Cons:
  - Constructor logic has unrelated logic inside of it
  - Will get messy with lots of event handlers, long event handlers

Callback style

```
image.addEventListener('click', event => {
  const image = event.currentTarget;
  image.src = this.giftSrc;
});
```

Version B: Inline event handler

- Pros:
  - Does not expose the event handler: function is privately encapsulated
- Cons:
  - Constructor logic has unrelated logic inside of it
  - Will get messy with lots of event handlers, long event handlers

Advanced closures

```
function createFunction() {
  let x = 0;

  function inner() {
    x++;
    let y = 0;
    y++;

    console.log('x is: ' + x + ', ' + 'y is: ' + y);
  }
  return inner;
}

const functionOne = createFunction();
functionOne();
functionOne();
functionOne();
```

What's the output of this program? (CodePen)

Advanced closures

```
function createFunction() {
  let x = 0;

  function inner() {
    x++;
    let y = 0;
    y++;

    console.log('x is: ' + x + ', ' + 'y is: ' + y);
  }
  return inner;
}

const functionOne = createFunction();
functionOne();
functionOne();
functionOne();
```

Console

```
"x is: 1, y is: 1"
"x is: 2, y is: 1"
"x is: 3, y is: 1"
```

Closures

```
function createFunction() {
  let x = 0;

  function inner() {
    x++;
    let y = 0;
    y++;

    console.log('x is: ' + x + ', ' + 'y is: ' + y);
  }
  return inner;
}
```

Within a closure, you can reference variables that were declared in the outer function, and those variables **will not go away** after the outer function returns.

Closures

```
function createFunction() {
  let x = 0;

  function inner() {
    x++;
    let y = 0;
    y++;

    console.log('x is: ' + x + ', ' + 'y is: ' + y);
  }
  return inner;
}
```

The variable is not copied to the inner function; the inner function has a **reference** to the variable in the outer scope.

- See this iconic StackOverflow post to learn more

Closures

```
function createFunction() {
  let x = 0;

  function inner() {
    x++;
    let y = 0;
    y++;

    console.log('x is: ' + x + ', ' + 'y is: ' + y);
  }
  return inner;
}
```

tl;dr: Be careful with closures! For now, we are not going to be modifying outer function variables in the closure.