PROGRAMMING LANGUAGES PARADIGMS

JavaScript Scope and Closures

BINDINGS

- A binding is an association of a name with an entity. The scope of a binding is the region of code where a particular binding is active.
- Ways we can introduce bindings in Javascript:
 - var, let, and const
 - var and let are used to declare variables
 - const is used to declare constant variables (constants)
 - Function declarations
 - Class declarations
 - Function parameters

SCOPE

- The scope of a binding is the region of code where a particular binding is active.
- Bindings introduced with var are scoped to the innermost function (function-scoped).
- Bindings introduced with let and const are scoped to the nearest block (block-scoped).

```
const v1 = 1
const v2 = 2;
function foo(){
    console.log(v1);
    console.log(v2);
    if (4<5){
        var v1=100;
        let v2=200;
        const v3=300;
        console.log(v1);
        console.log(v2);
        console.log(v3);
    console.log(v1);
    console.log(v2);
    console.log(v3);
foo();
```

- Reading a var-declared variable in its scope but before its declaration produces *undefined*.
- Reading a let-declared variable in its scope but before the let throws a ReferenceError.

SCOPE

- The scope of a binding is the region of code where a particular binding is active.
- Bindings introduced with var are scoped to the innermost function (function-scoped).
- Bindings introduced with let and const are scoped to the nearest block (block-scoped).

```
const v1 = 1
const v2 = 2:
function foo(){
    console.log(v1);
    console.log(v2);
    if (4<5){
        var v1=100;
        let v2=200;
        const v3=300;
        console.log(v1);
        console.log(v2);
        console.log(v3);
    console.log(v1);
    console.log(v2);
    console.log(v3);
foo();
```

- Reading a var-declared variable in its scope but before its declaration produces undefined.
- Reading a let-declared variable in its scope but before the let throws a ReferenceError.

```
hwks - bash - 80×24
Zavala-MEC-MacBook-Air:hwks Admin$ node labs.js
undefined
100
200
300
100
/Users/Admin/Desktop/hwks/labs.js:125
    console.log(v3);
ReferenceError: v3 is not defined
    at foo (/Users/Admin/Desktop/hwks/labs.js:125:17)
    at Object.<anonymous> (/Users/Admin/Desktop/hwks/labs.js:127:1)
    at Module._compile (module.js:635:30)
    at Object.Module._extensions..js (module.js:646:10)
    at Module.load (module.js:554:32)
    at tryModuleLoad (module.js:497:12)
    at Function.Module._load (module.js:489:3)
    at Function.Module.runMain (module.js:676:10)
    at startup (bootstrap_node.js:187:16)
    at bootstrap_node.js:608:3
Zavala-MEC-MacBook-Air:hwks Admin$
```

FREE VARIABLES

Free variables – variables used but not declared inside a function.

FREE VARIABLES

Free variables – variables used but not declared inside a function.

```
const x= 'OUTER';
function second() {
    console.log(x); \leftarrow
                                             Variable x is free within function second
                                                     Where does x take its value from?
function first() {
                                                     From its caller, function first or from
    const x= 'FIRST';
                                                     the outer x?
    console.log(x);
    second();
                                                     FIRST
first();
                                                     OUTER
```

JAVASCRIPT IS STATICALLY SCOPED

Free variables – variables used but not declared inside a function.

```
const x= 'OUTER';
function second() {
    console.log(x);
                                            Variable x is free within function second
                                                   Where does x take its value from?
function first() {
                                                   From its caller, function first or from
    const x= 'FIRST';
                                                   the outer x?
    console.log(x);
    second();
                                                   FIRST
first();
                                                   OUTER
```

- Where do the free variables take their value from?
 - Languages that use the caller's values for free values are dynamically scoped
 - Languages that look outward to textually enclosing regions are <u>statically scoped</u>
 (JavaScript)

JAVASCRIPT CLOSURES

A closure is the combination of a function bundled together with an environment. The
environment is a mapping associating each free variable of the function with the value or
reference to which the name was bound when the closure was created.

```
function second(f) {
    const name="new";
    f();
}
function first() {
    const name = "OLD";
    const printName= ()=>console.log(name);
    second(printName);
}
first();
```

Variable *name* is free within the <u>anonymous</u> function assigned to *printName*

- At the time the printName function is created, it sees name defined within first with a value of "OLD".
- The variable name is bound to the value "OLD".
- The printName function carries the binding of the variable name to the value "OLD" with it.
- Within the scope of the printName function, the variable name will always hold the value "OLD".

JAVASCRIPT CLOSURES

A closure is the combination of a function bundled together with an environment. The
environment is a mapping associating each free variable of the function with the value or
reference to which the name was bound when the closure was created.

```
function second(f) {
    const name="new";
    f();
}
function first() {
    const name = "OLD";
    const printName= ()=>console.log(name);
    second(printName);
}
first();
```

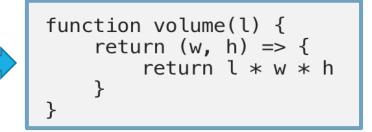
Variable *name* is free within the <u>anonymous</u> function assigned to *printName*

- At the time the *printName* function is created, it sees *name* defined within *first* with a value of "OLD".
- The variable name is bound to the value "OLD".
- The printName function carries the binding of the variable name to the value "OLD" with it.
- Within the scope of the printName function, the variable name will always hold the value "OLD".

USE OF CLOSURES

REMINDER: Currying and Partial Applications

 Partial application fixes the value of some of a function's arguments without fully evaluating the function.



 Currying is the process of turning a function with multiple arguments into a sequence/series of functions each taking a single argument.

Currying and partial application are possible because of closures: the arguments in currying and partial application are kept "alive" via closure.

USE OF CLOSURES

- Closures can also be used to make generators
 - A generator function produces a "next" value each time it is called.
 (we will learn more about generators later)
 - Example:

```
const nextSquare = (() => {
  let previous = -1;
  return () => {
    previous++;
    return previous * previous;
  }
})();

const assert = require('assert');
assert(nextSquare() === 0);
assert(nextSquare() === 1);
assert(nextSquare() === 4);
```