Scoping rules in Javascript

What is scope?

- Scope is used to determine what code is accessible, where, during program execution
- The outmost boundary is the global scope, and is accessible everywhere throughout the program
- Local scope is walled off from other plots, but can access the global scope
- A variable might have local scoping or global scoping depending on where in the code it was defined
- Scopes can be nested, and follows the same rules

Lexical scoping

- Scope is defined by the source code at author-time, and does not change dynamically at execution time
- On the other hand, dynamic scoping means that scope is determined by the call stack, and changes depending on how a given function is executed
- Scope exists on the top level (global), and local level (inside individual functions, or blocks)
- Scope can nest, meaning code at lower levels of nested scope can access variables defined in higher levels, not vice versa

Function scope

- Each function has its own local scope, code inside the function block obey scoping rules: 1) code inside function may access var defined in global or higher-nested scope, not lower-nested scope, 2) var defined inside function may not be accessed from global or higher-nested scope, may be accessed by lower-nested scope

```
var computer = "hpDesktop";
function apartmentOne() {
 var computer = "iMac";

 function bedroomOne() {
   var computer = "macbookPro";
   console.log("apartmentOne, bedroomOne uses computer: " + computer);
   // macbookPro
 }

 function bedroomTwo() {
   var computer = "chromeBook";
   console.log("apartmentOne, bedroomTwo uses computer: " + computer);
   // chromebook
 }

 function bedroomThree() {
   console.log("apartmentOne, bedroomThree uses computer: " + computer);
   // iMac
 }

 bedroomOne();
 bedroomTwo();
 bedroomThree();
}
function apartmentTwo() {
 function bedroomOne() {
   console.log("apartmentTwo, bedroomOne uses computer: " + computer);
   //hpDesktop
 }
 bedroomOne();
}
apartmentOne();
apartmentTwo();
```

- In the above example, each function call checks for a `computer var` in its local scope before moving up to look in a higher scope
- There's also no conflict, each function has only one var called `computer` as far as it knows

Block scope

- Defined between `{...}`, within an `if...else` block
- Not all vars respect block scope
- Var defined with the var keyword are not block-scoped, accessible outside the block
- ES6 keywords `let` and `const` are block-scoped

```
{
  var fruit = "banana";
  let vegetable = "carrot";
  const spice = "paprika";

  console.log("Inside the block...");
  console.log("The fruit is: " + fruit);
  console.log("The vegetable is: " + vegetable);
  console.log("The spice is: " + spice);
}
console.log("Outside the block...");
console.log("The fruit is: " + fruit); //banana
console.log("The vegetable is: " + vegetable); //Uncaught ReferenceError: vegetable is not defined
console.log("The spice is: " + spice); // Uncaught ReferenceError: spice is not defined
```

- Note the difference in scoping between the `var` and `let` and `const`
- The `var` keyword is not block-scoped, `let` and `const` are

Closure

- Scoping closure behaviour means they close over vars that are within scope at the time a function is defined
- Closure allows functions to retain access to given var even if invoked from a different scope
- Functions retain access to vars defined by their lexical scope, even when invoked from outside the scope
- Scope used by a function is the scope at function definition, not the scope at function invocation

```
var sandwich = "ham and cheese";

function eatSandwich() {
  console.log("Now eating " + sandwich + "!");
}

function lunch() {
  var sandwich = "BLT";
  eatSandwich();
}

lunch(); //Now eating ham and cheese!
```

- In this example, `eatSandwich` only cares about the variable `sandwich` within its scope at definition time, it closes over this value at definition and retains access to it, even when called from another scope
- Thus the importance of defining closure at function definition versus function invocation
- With lexical scoping(versus dynamic scoping), the function would not look through the call-stack for the necessary var