Functions, function declaration, function expresssions, hoisting

- Function declaration -> Anonymous function expressions -> Named function expressions
- Variable declaration hoisting -> Function declaration hoisting -> Function expression hoisting ->

Functions

- When defining a function: parameters => the function `multiply()` takes two `parameters` a, and b
- When invoking a function: actual values passed to a function during execution are the `arguments`
- Functions can be nested

```
function one() {
  function two() {
    return x;
  }
  return y;
}
```

Functional Scopes and Lexical Scoping

- Code within an inner scope can access any variables in the same scope or any surrounding scope
- A nested function scope can always access a variable from the outer scope

Closure

- A function retains access to (closes over) the var scope currently in effect, this is creating a closure
- It retains references to everything in scope when closure is created, and retains those references for as long as the closure exists
- So the function can still access those references when invoking the function

```
var name = 'Julian';
function greet() {
  function say() {
    console.log(name);
  }
  say();
}
greet(); // Julian
```

- `greet()` can be called anywhere in program, can access `name` even if `name` if out of scope at invocation point
- Value of var can change after creating a closure that includes the var, the closure will see the new var (old no longer available)
- Another example, note the differences in output based on `return` and `console.log`

```
var name = 'Julian';
function greet() {
  function say() {
    var name = 'Aaron';
    console.log(name); // Aaron (2)
```

```
    return(name);
  }
  console.log(name); // Julian (1)
  console.log(say()); // Aaron (3)
  return(name);
}
console.log(greet()); // Julian (4)

var count = 1;
function logCount() {  // create a closure
  console.log(count);
}

logCount();            // logs: 1

count += 1;            // reassign count
logCount();            // closure sees new value for count; logs: 2
```

Lexical Scoping

- Lexical scoping (static scoping, instead of dynamic scoping) is used to resolve variables
- Source code defines the scope
- A function creates a scope regardless of whether it's execcuted or not
- At any point in a JS program, there's a hierarchy of scopes from local scope of code up to program's global scope
- JS searches from bottm to top when looking for a var, and stops and returns the first var it finds with matching name
- Vars in a lower scope can shadow/hide a var with same name in a higher scope

Adding vars to current scope

- 3 ways: using `var` keyword, use arguments passed to a function, function declaration itself

```
function lunch() {    // A function declaration creates a new variable scope
  var food = 'taco';  // 1. Add a new variable food within the current variable scope
}

function eat(food) {  // 2. Parameters create variables during function invocation
  console.log('I am eating ' + food);
}

function drink() {    // 3. Add a new variable drink within the global variable scope
  console.log('I am drinking a glass of water');
}
```

- Note the scope of `food` variable from parameter of `eat()`, its scope is the `eat` function because of way source code is written, not because function gets invoked. At runtime, scope implies `eat` can only be accessed from within body of `eat` function

Variable Assignment

- Var scoping rules also applied to referencing (in addition to assignment)

```
var country = 'Spain';
function update() {
  country = 'Liechtenstein';
}

console.log(country);  // logs: Spain

update();
console.log(country);  // logs: Liechtenstein
```

- It sets the first `country` variable it finds by checking current scope and then each higher scope, looking for var with name `country`
- If JS can't find a matching var, it creates a new global var instead

```
function assign() {
  var country1 = 'Liechtenstein';
  country2 = 'Spain';
}

assign();
console.log(country2);   // logs: Spain
console.log(country1);   // gets ReferenceError
```

- Above, `country2` is not declared elsewhere, and is assigned a value inside the function
- Since JS can't find a matching var, it creates a new `global` variable and logs its value
- Similar to earlier code in adding vars to current scope section, `country2` is in global scope because of the way source code is written, not because the `assign` function was executed

Variable shadowing

- Var declaration for `name` in `greet()` shadows the outer `name` variable
- Within `greet()`, can only access the inner `name`

```
var name = 'Julian'; //global scope

function greet() { //function scope
  var name = 'Logan';
  console.log(name);
}

greet(); // Logan
```

- If function definition has a parameter with same name as a var from outer scope, parameter shadows outer var

```
var name = 'Julian';

function greet(name) {
  console.log(name);
}
greet('Sam');  // logs: Sam
```

- Throws a `ReferenceError` if it can't find a var anywhere in scope hierarchy

- Var scoping rules:
  - Every function declaration creates a new variable scope
  - Lexical scope uses structure of the source code to determine variable's scope. The code doesn't have to be executed for the scope to exist
  - All variables in the same or surrounding scopes are available to your code

Function declaration

- Function declaration is the same as a function statement
- A function declaration defines a variable whose type of function
- It does not require assignment to a variable, since the value of the function variable is the function itself
- Function declaration starts with `function`, just as var declaration starts with `var`

```
function hello() {
  return 'hello world!';
}
console.log(typeof hello); //function
```

- This function variable (function itslef) obeys general scoping rules

```
function outer() {
  function hello() {
    return 'hello world!';
  }
  return hello();
}
console.log(typeof hello); // can't access local scope from here
var foo = outer; // assign function to another variable
foo(); // can be used to invoke function
```

- Recap, how is a variable created? 1. `var` keyword, 2. passing arguments to a function, 3. function declaration
- A function declaration defines a function, and it defines a variable with the same name as the function, then assigns function to variable (thus `typeof outer` would return function, and `var foo` is assigned to `outer` and not `outer()`)
- With every function declaration, a variable is initialized

```
var stringVar = 'string ref';
var numberVar = 42;

function functionVar() {
  return 'function reference';
}

console.log(typeof stringVar); //string
console.log(typeof numberVar); //number
console.log(typeof functionVar); //function

stringVar = functionVar;
functionVar = 'string reference';
```

```
console.log(typeof stringVar);  //function
console.log(typeof functionVar);// string
```

- Is this an example of dynamic typing? note the stringVar var is reassigned to the function and looks up the var from the bottom up and returns the first value it sees, it's not dynamically typed, meaning it's not looking at the rest of the stack to see what functionVar might be later on

Function expressions

- Defines a function as part of a larger expression syntax (variable assignment)
- It's basically a function declaration and assignment at the same time?
- An anonymous function is defined and assigned to var `hello`, variable is used to invoke the function

```
var hello = function() {
  return 'hello';
console.log(hello()); // hello
}
console.log(typeof hello); // function
```

- A function expression cannot return a value without an actual assignment invocation. If the inner function expression didn't exists, then calling the function variable `foo` would not work

```
var foo = function () {
  return function () {   // function expression as return value
    return 1;
  };
};
var bar = foo();         // bar is assigned to the returned function
bar();                   // 1
```

- In this case, `foo` returns an anonymous function, the returned function expression is later assigned to `bar`

Named function expressions

- Next, we name the expressions

```
var hello = function foo() {
  console.log(typeof foo);   // function
};
hello();
foo();                       // Uncaught ReferenceError: foo is not defined
```

- ?When does calling a function expression result in `undefined` versus `Uncaught ReferenceError`?
- In this case, the name `foo` is only available inside the function (local scope)
- Using named function expression is useful for debugging, can show function name in call stack
- What's the difference between a named function expression and function declarations? If a statement starts with `function`, it's a function declaration, otherwise a function expression
- Note that only the named expression is in scope, and not the function expression itself and this is a Uncaught ReferenceError

```
function foo() {
  console.log('function declaration');
}
(function bar()) {
  console.log('function expression');
}
foo(); // function declaration
bar(); // Uncaught ReferenceError: bar is not defined
```

- A function defined using a function declaration must always have a name (and not be anonymous)
- Function declaration creates a variable with the same name as function's name
- Below, both function definitions define a named function, and a variable with the same name as that function

```
var foo = function foo() {
  return 'a named function expression assigned to a variable';
};

function bar() {
  return 'a function declaration';
}
```

Variable declaration hoisting

- Var declarations are processed before any code is executed within a scope
- Hoisting: declaring a var anywhere in a scope == declaring it at top of scope
- The two examples below are equivalent.
- Note JS only hoists variable declarations, and not assignments

```
console.log(a);   // undefined
var a = 123;
var b = 456;
```

```
var a;
var b;
console.log(a); // undefined, since JS hoists only variable declarations
a = 123;
b = 456;
```

Hoisting for function declarations

- Function declarations are also hoisted to the top of scope
- Entire function declarations is hoisted, including body
- These two are equivalent

```
console.log(hello());
function hello() {
  return 'hello world';
}
```

```
function hello() {
  return 'hello world';
```

```
}
console.log(hello());
```

Hoisting for function expressions

- Since function expressions is merely the assignment of function to a declared variable, expressions are just variable declarations, they obey the hoisting rules for var declarations
- The examples below are equivalent

```
console.log(hello());
var hello = function() {
  return 'hello world';
};
```

- In the below case (same as above), `var hello` is declared but not assigned, so no type is attached to the var. When trying to log `hello()`, an uncaught typeerror is raised. If you try to log `hello` instead, the return value would be `undefined`.
- A function expression works the same way as a variable, where the declaration is hoisted but not the assignment. In this case, a var is declared with unknown type, and when `console.log` attempts to log the return value of a function, this is unknown. On the contrary, if it attempts to log the value of `hello` as a var, then the return value is `undefined`.

```
var hello;
console.log(hello()); // raises "Uncaught TypeError"
hello = function() {
  return 'hello world';
}
```

Hoisting var and function declarations

- Which one is hoisted first when both var and function declaration exist? Function declaration is hoisted first (above the var declaration)
- The two below are the same
- Function declaration > var declaration (assignment is not hoisted and is executed based on where it is in the program)

```
bar();  // logs undefined
var foo = 'hello';
function bar() {
  console.log(foo);
}
```

```
function bar() {
  console.log(foo);
}
var foo;
bar();  // logs undefined
foo = 'hello';
```

- Timing is relevant here: `bar` uses a var in global scope. Even though `bar` was declared below the assignment of `hello` (because of hoisting), when `bar` is invoked, the value logged will not be `hello` already

- Because of hoisting rules for variable and function declaration, `foo` is still undefined when `bar` is invoked
- What happens if the same name is used for variable and function?

```
// version 1
bar();  // logs "world
var bar = 'hello';
function bar() {
  console.log('world');
}

// version 2
var bar = 'hello';
bar();    // raises "Uncaught TypeError"
function bar() {
  console.log('world');
}
```

- Notice the change in code results when the first two lines are switched order
- Notice that function declarations are hoisted first, var declaration of the same name becomes redundant, and becomes a reassignment
- Here are the hoisted versions

```
// version 1 hoisted
function bar() {
  console.log('world');
}
bar();
bar = 'hello';

// version 2 hoisted
function bar() {
  console.log('world');
}
bar = 'hello';
bar();
```

- For version 2, the `function bar()` is indeed hoisted up to the top, but it's then reassigned to a string. So when a function `bar()` gets called, it doesn't exist anymore.
- Notice that in version 1 of the hoisted code, the function variable is hoisted to the top. And because the variable declaration now effectively becomes a variable reassignment, it's not hoisted, and needs to be processed in the order it's declared. At this point, where the variable invocation is positioned starts to matter. if `bar();` appears before the variable reassignment, then an error is raised.

Some general guidelines for hoisting

- Declare vars at top of scope
- Declare functions before calling them

Some more clarification on function declarations vs. function expressions

Function declaration

- Function declaration = named function variable without requiring variable assignment
- This is a standalone construct, and can't be nested within non-function blocks
- Siblings of variable declaration
- Must start with `function`

```
function bar() {
  return 3;
}
```

- Function name is visible within scope and scope of its parent

```
function bar() {
  return 3;
}
bar() //3
bar   //function
```

Function expression

- Defines a function as part of larger expression syntax (var assignment)
- Can be named or anonymous
- Does not start with `function`

```
//anonymous function expression
var a = function() {
  return 3;
}

//named function expression
var a = function bar() {
  return 3;
}

//self invoking function expression
(function sayHello() {
  console.log("hello!"); // hello!
})();
```

- A few examples to demonstrate the difference between function declaration and function expressions
- The last function with the function name gets returned. In the below case, both functions `bar` get hoisted, and `return bar()` returns the last function called `bar()`

```
// 1. This gets hoisted
function foo(){
  function bar() {
    return 3;
  }
  return bar();
  function bar() {
    return 8;
```

```
  }
}
console.log(foo());

// Processing sequence
function foo(){
  // define bar once
  function bar() {
      return 3;
  }
  // redefine it
  function bar() {
      return 8;
  }
  // return its invocation
  return bar(); //8
}
console.log(foo());
```

- Sidebar: do function expressions get hoisted too?

```
var bar = function() {
  return 3;
}
```

- In this case, the left hand side (var bar) is a var declaration. Variable declaration gets hoisted, but assignment expressions don't. When `bar` is hoisted, the interpreter initially sets `var bar = undefined`. So, the function definition itself is not hoisted.

// 2. Look at the sequence of execution here

- In this case, both var declarations are hoisted to the top, but only the first var declaration is assigned a value. When the `return` command is called, the first function expression is reached, and the second function expressions unreachable.

```
function foo(){
    var bar = function() {
        return 3;
    };
    return bar();
    var bar = function() {
        return 8;
    };
}
console.log(foo());

// 2. Processing sequence
function foo() {
  // a declaration for each function expression
  var bar = undefined;
  var bar = undefined;
  // first function expression is executed
```

```
  bar = function() {
    return 3;
  };
  // function created by first function expression
  return bar();
  // second function expression unreachable
}
console.log(foo()); //3
```

*** I'm here ***

```
// 3. Look at execution sequence
console.log(foo());
function foo(){
  var bar = function() {
    return 3;
  };
  return bar();
  var bar = function() {
    return 8;
  };
}
```

```
// 3. Execution sequence
function foo(){
  var bar = function() {
      return 3;
  };
  return bar(); // 3
  // program exists, next part is unreachable
  var bar = function() {
      return 8;
  };
}
console.log(foo());
```

- Below, function expression is declared but not assigned, `return bar()` is executed before the function expresson is reached.

```
// 4. Look at execution sequence
function foo(){
    return bar();
    var bar = function() {
        return 3;
    };
    var bar = function() {
        return 8;
    };
}
alert(foo());
```

```
// 5. Actual processing sequence
function foo() {
  // declaration for each function expression
  var bar = undefined;
  var bar = undefined;
  return bar(); // TypeError: 'bar not defined'
  // neither function expression is reached
}
alert(foo());
```

Function expression benefits

1. The function expression suggests we are creating an object

```
// Function declaration
function add(a, b) {
  return a + b;
}
// Function expression
var add = function(a, b) {
  return a + b;
}
```

2. Function expressions are more versatile.

- Function declaration can only exist as a statement in isolation, and can create an object variable parented by its current scope.
- A function expression is part of a larger construct
- Function expression is needed if you want to create an anonymous function, or assign function to a prototype, or as property of some other object

Function expression drawbacks

- Function expressions are typically unnamed, so debugging could be frustrating
- The workaround is to use NFE (named function expressions)

```
// Turn this into ...
var today = function() {return new Date()}
// Something easier to debug
var today = function today() {return new Date()}
```