Variable Scope

Not An Oral Hygeine Product

Levi Pearson

Introduction

Goals

- Understand How Variable Naming Works
- Learn What “Scope” Really Means
- See How Different Languages Handle Scope
- Identify Tricky Bits in Javascript Scope Rules

Variables, Names, and Scope

On the difficulty of names

There are only two hard things in Computer Science: ***cache invalidation*** *and* ***naming things****.*
– Phil Karlton

Some definitions

- A **variable** is a **name** bound to a **storage location** within an **environment**
- An **environment** holds all variable bindings visible at some point in a program.
- Sometimes an environment is called a **context**

What about scope?

- The **scope** of a variable refers to the region of the program where its binding is visible in the environment

or

- **Scope** is the *property of a variable* that defines what part of the program its name can be used in to identify the variable

Static vs. dynamic

- **Static** or **lexical**: Scope determined just by text of program
- **Dynamic**: Scope affected by bindings in the run-time stack

Early vs. late

- Lexical scope can be determined at compile-time; sometimes called **early binding**
- Dynamic scope relies on run-time information; sometimes called **late binding**

Shadowing

- Scopes of variables with the same name can *overlap*
- Rules determine which scope “wins” at any point
- The *losers* are said to be **shadowed** by the *winner*
- Rule is usually a variation on “most recent declaration wins”

Levels of scope

Global scope

- **Global** scope extends through the whole program
- Sometimes restricted to just after the declaration.

Function scope

- **Function** scope extends through the whole function body
- Sometimes restricted to just after the declaration, e.g. C
- When not restricted, it is sometimes called **hoisting** the declaration
- Hoisting affects scope, but not initialization point

```javascript
// Shadowing and Hoisting

var x = 1;

function foo() {
    console.log(x);
    var x = 2;
    console.log(x);
}

console.log(x);
```

```javascript
// Shadowing and Hoisting

var x = 1;

function foo() {
    console.log(x); // x is undefined!
    var x = 2;
    console.log(x); // prints 2
}

console.log(x);     // prints 1
```

Block scope

- **Block** scope extends through a compound statement block
- Variables declared in control statements (like `for`) are scoped to the block
- Javascript (pre-ES6) and Python don’t have block scopes!
- ES6 keeps function scope for `var`, adds `let` and `const` for block scope

```javascript
// Block Scope
var y = 2;
function foo() {
    let x = 5;
    for (let x = 0; x < 3; x++) {
        let y = x + 1;
        console.log(y);
    }
    console.log(x);
    console.log(y);
    let y = 3;
    console.log(y);
}
console.log(y);
```

```
// Block Scope
var y = 2;
function foo() {
    let x = 5;
    for (let x = 0; x < 3; x++) {
        let y = x + 1;
        console.log(y); // prints 1, 2, 3
    }
    console.log(x);     // prints 5
    console.log(y);     // ReferenceError
    let y = 3;
    console.log(y);     // prints 3
}
console.log(y);         // prints 2
```

Other scopes

- In C and C++, there is a **file** scope
- Functional languages often provide **expression** scope
- Languages with module systems provide **module** scope
- Python has a module system and module scope; ES6 does too

Dynamic scope

- Dynamic scope refers to time periods instead of text regions
- Dynamic global scope refers to the whole program execution
- Dynamic function scope
    - starts when execution enters the function body
    - extends through any function calls in the body
    - ends when the function returns
- Dynamic scope is default in Bourne-style shells, PowerShell, Emacs Lisp
- Dynamic scope is optional in Perl, Common Lisp, and others

```
// In a hypothetical dynamically-scoped Javascript
var x = 1;

function foo() {
    var x = 2;
    bar();
}

function bar() {
    console.log(x);
}

foo();
bar();
```

```javascript
// In a hypothetical dynamically-scoped Javascript
var x = 1;

function foo() {
    var x = 2;
    bar();
}

function bar() {
    console.log(x);
}

foo(); // prints 2
bar(); // prints 1
```

Tricky Bits

Assignment

- What should be the scope of a variable created by assignment?
- Local?
    - Can’t assign to a variable in enclosing environment
    - Will create a new variable shadowing the one you wanted to change
    - Python works this way
- Global?
    - Might accidentally change existing global instead of making a new one
    - Can work around it by declaring all your variables
    - Javascript works this way
- Just make declaration of variables mandatory!

Closures

- A variable is **free** within a function body if:
    - it is referenced in the body
    - it is not declared in the body
    - it is not a parameter to the function
- A function *closes over* free variables bound in an enclosing environment
- The variables are found in its **closure**
- We call a function returned from its enclosing environment a **closure** too
- Bindings in a closure keep the scope from their *definition* point, even if the function is invoked in a different environment

```
// Fun with closures!
function a() {
    var x = 0;
    return function() {
        x++; console.log(x);
    }
}
function b(g) {
    var x = 0;
    g(); console.log(x);
}
var c1 = a(), c2 = a();
b(c1);
b(c2);
b(c1);
```

```
// Fun with closures!
function a() {
    var x = 0;
    return function() {
        x++; console.log(x);
    }
}
function b(g) {
    var x = 0;
    g(); console.log(x);
}
var c1 = a(), c2 = a();
b(c1); // prints 1, 0
b(c2); // prints 1, 0
b(c1); // prints 2, 0
```

This

- It is dynamically scoped, late-bound
- Rules for `this`:
    1. invoked as a function, it is the global object
    2. invoked with `new`, it is the object the constructor will return
    3. invoked as a method, it is the object before the `.`
    4. invoked with `call` or `apply`, it's what you asked it to be
    5. wrapped by `Function.prototype.bind`, it's `bind`'s argument
    6. invoked as a DOM Event handler, it's the element the event fired from
- For closure creation, `this` is not a free variable
- Except for ES6 arrow functions!

Thanks for listening!