Functions

Objectives

- Define and use JavaScript functions
- Understand context
- Understand parameters

Overview

- Fundamental to understanding JavaScript
- 'First class' objects

Functions as first class objects

- Functions
  - Can be assigned to variables
  - Passed to other functions
  - Returned from functions

Assigning functions

- One of the first things we'll do as a JavaScript programmer

```
function onStart(){
}

window.onload = onStart;

// or

window.onload = function(){
};
```

Declaring functions

- Declared using a function literal
  - function keyword
  - optional name
  - comma separated parameter list
  - body

Examples of functions

```
function foo(){return true;}
assert(typeof foo === "function", "defined");
assert(foo.name === "foo", "named");

var bar = function(){return true;}
assert(typeof bar === "function", "defined");
assert(bar.name === "", "no name");

window.baz = function(){return true};
assert(typeof baz === "function", "defined");
assert(baz.name === "", "no name");
```

Non global functions

- Previous slide showed global functions
  - Scoped to 'window'
- Functions can be scoped

```
function outer(){
    assert(typeof outer === "function", "function");
    assert(typeof inner === "function", "nested");
    function inner(){}
}
outer();
assert(typeof inner === "undefined", "nested");
```

JavaScript scoping

- Scopes created by functions not blocks
  - i.e. {} does not create a scope
  - functions are hoisted to the top of the declaring block

```
function outer(){
    assert(typeof outer === "function", "function");
    assert(typeof inner === "function", "nested");
    function inner(){}
}
outer();
assert(typeof inner === "undefined", "nested");
```

Calling functions

- Four ways to call
  - As a function
  - As a 'method'
  - As a constructor
  - Using .call/.apply

Function parameters

- A list of arguments can be provided when calling a function
  - these are assigned to the function parameters
  - numbers of arguments and parameters do not have to match
- If fewer arguments than parameters
  - extra parameters are set to undefined
- If more arguments than parameters
  - excess arguments are not assigned

Implicit argument parameter

- 'this' and 'arguments' are also available inside the function
  - 'arguments' is collection of all arguments passed
  - has .length property
  - access using array syntax

'arguments' parameter

- Not an array
  - is 'array like'
- Often see this

```
function func() {
    var args = Array.prototype.slice.call(arguments);
}
```

Implicit 'this' parameter

- Reference to the *invoker* of the function
  - Also known as the *function context*
  - 'this' can vary depending on how the function is invoked

'function' invocation

- This is invocation as you would think of it

```
function createUser(){}
createUser();

var updateUser = function(){}
updateUser();
```

'method' invocation

- Function added as a property on an object
  - and called through that object
  - Inside the function 'this' is a reference to the calling object

```
var user = {};
user.createUser = function(){}
user.createUser();
```

```
function createUser(){ return this;}
createUser(); // this === window

var user = {};
user.createUser = createUser;
user.createUser(); // this === user
```

Functions as constructors

- Declared like other functions
  - invoked differently

```
function User(){
};

var user = new User();
```

Constructor invocation

- A new empty object is created
- New object is passed to the function as the 'this'
- New object is returned implicitly from the function
  - don't return anything else!
- Constructor functions start with uppercase first letter
  - By convention

Using constructors

```
function User(){
        this.create = function(){
                return this;
        };
        this.update = function(){};
};

var user1 = new User();
var user2 = new User();

// user1.create() != user2.create()
```

Invoking functions with 'call' and 'apply'

- Used to set caller's context (this) explicitly
- All functions have 'call' and 'apply' methods
  - functions are just objects
  - created with the Function() constructor
- 'apply'
  - two parameters, the context and array of args
- 'call'
  - similar but args passed individually

Call and Apply

- Useful for
  - changing the context of the function
  - split up parameters to one function to pass to another

Using call and apply

```
function createUser(count){
      this.count = count;
}

var user1 = {};
var user2 = {};

createUser.call(user1, 10);
createUser.apply(user2, [20]);

// user1.count == 10
// user2.count == 20
```

Useful for callbacks

```
function forEach(collection, fn){
    for(var n = 0; n < collection.length; n++){
        fn.call(collection[n], collection[n], n);
    }
};

var items = ['user', 'meeting', 'clock'];

forEach(items, function(){
    console.log(this.toString())
});
```

Anonymous functions

- Previous slide is an example of an anonymous function

```
window.onload = function(){};

var user = {
    create: function(){}
}

setInterval(function(){}, 500);
```

Storing functions

- Sometimes want to store related functions
  - e.g. event management

```
var store = {
    nextId: 1,
    cache: {},
    add: function(fn) {
        if (!fn.id) {
            fn.id = store.nextId++;
            return !!(store.cache[fn.id] = fn);
        }
    }
};
function create(){}
store.add(create);
store.add(create); // returns false – already stored
```

Memoizing

- Functions that remember the result of a previous called
    - Cache the result of the previous call
    - i.e. create a 'memo' of it
    - Makes calls more efficient

Self-memoizing

- Functions are objects
  - Can add properties to them
- Add a hash to the function to cache previous results
  - refer to this hash before executing function

Self memoizing functions

```
function fibonacci(value) {
    if (!fibonacci.answers) fibonacci.answers = {};
    if (fibonacci.answers[value] != null) {
        return fibonacci.answers[value];
    }
    var val = 0;
    var next = 0;
    var nextnext = 1;

    if(value == 1){
        return 1;
    }

    for (var i = 1; i < value; i++) {
        val = next + nextnext;
        next = nextnext;
        nextnext = val;
    }
    return fibonacci.answers[value] = val;
}
assert(fibonacci(6) == 8, fibonacci(6) + " == 8");
assert(fibonacci.answers[6] == 8, "fibonacci[6] is cached");
```

Summary

- Functions can be declared with or without a name
- Functions can be used as constructors
- Functions have an implicit this
- 'this' can be set by 'calling' or 'applying' functions
- Functions can be stored
- Functions set up a scope
- Functions can be memoized