# Functions

- Traditional way of packaging up parameterized code for subsequent execution.
- Functions are **first-class**: need not have a name ("anonymous"), can be passed as parameters, returned as results, stored in data structure.
- Functions can be nested within one another.
- **Closures** preserve the referencing environment of a function.

# Current Object Context

- During execution of a function, there is always an implicit object, referred to using **this**.
- Using the word "this" while speaking can cause confusion; hence when speaking, I will usually pronounce **this** as **self**.
- **this** cannot be assigned to.
- Usually, **this** depends on how the function was called, it can be different for the same function during different calls.
- In global contexts (outside any function), **this** refers to the global object.
- When `strict` mode is not in effect, in a simple function call `fn()` without any receiver, **this** will refer to the global object.
- When `strict` mode is in effect, in a simple function call `fn()` without any receiver, **this** will refer to the value within the calling context.

# Current Object Context Continued

- When called with a receiver using the dot notation, `this` refers to the receiver. So when `f()` is called using `o.f()`, the use of `this` within the call refers to `o`.
- It is possible to set the context dynamically using `apply()`, `call()` and `bind()`.
- When a function call is preceeded by the `new` operator, the call is treated as a call to a constructor function and `this` refers to the newly created object.
- `globalThis` provides a standard way to access the "global" object across platforms; i.e. references `window` on a browser platform and `global` on nodejs.

# Using call()

Allows control of this when calling a function with a fixed number of arguments known at program writing time.

- All functions have a call property which allows the function to be called. Hence

```
let f = function(...) { ... };
let o = ...;
f.call(o, a1, a2); //like o.f(a1, a2)
                    //but o may not contain a f()
```

- Within function body, this will refer to o.
- call allows changing this only for functions defined using function, not for fat-arrow functions.

# Example of Using call to control this

```
> obj1 = {
    x: 22,
    f: function(a, b) { return this.x*a + b; }
  }
> obj2 = { x: 42 }
{ x: 42 }
> obj1.f(2, 1)
45
> obj1.f.call(obj1, 2, 1) //obj1 as this.
45
> obj1.f.call(obj2, 2, 1) //obj2 as this
85
```

# Legacy JavaScript: Using Array Methods with arguments

arguments is not a real array:

```
> function f() { return arguments.map(v => v*2); }
undefined
> f(1, 2, 3)
Uncaught TypeError: arguments.map is not a function
    at f (REPL94:1:33)
```

Legacy workaround:

```
> function f() {
    return Array.prototype.slice
      .call(arguments).map(v => v*2);
  }
> f(1, 2, 3)
[ 2, 4, 6 ]
```

# Using apply()

Allows control of `this` when calling a function with a number of arguments not known at program writing time.

- All functions have a `apply` property which allows the function to be called. Hence

```
    let f = function(...) { ... };
    let o = ...;
    f.apply(o, [a1, a2]); //like o.f(a1, a2)
                          //but o may not contain a f()
```

- Within function body, `this` will refer to `o`.
- `apply` equivalent to `call` using spread operator; i.e. `f.apply(o, args)` is the same as `f.call(o, ...args)`.

# Playing with this Within a Module

Modules always have an implicit 'using strict'; declaration. All assertions in this-play.mjs pass.

```
//strict mode is on
import assert from 'assert';

//top-level this in nodejs
assert(this === undefined);

//plain function call using strict
function f1() {  return this; }
assert(f1() === undefined);
```

# Playing with this Within a Module Continued

```
//plain function call with explicit strict
function f2() {
  'use strict';
  return this;
}
assert(f2() === undefined);

const obj1 = { a: 22, f: function() { return this; } }

//like plain function call
const g = obj1.f;
assert(g() === undefined);

//normal object call
assert(obj1.f() === obj1);
```

# Playing with this Within a Module Continued

```
//change this using call
assert(obj1.f.call(obj1) === obj1);
assert(obj1.f.call(Array) === Array);
```

# Playing with this Within a Script

Unlike modules, scripts always start in non-strict mode.
All assertions in this-play.js pass:

```
//strict mode is off
const assert = require('assert');

//top-level this in nodejs
assert(this === module.exports);

//plain function call without strict
function f1() {  return this; }
assert(f1() === global);
```

# Playing with this Within a Script Continued

```
//plain function call with strict
function f2() {
  'use strict';
  return this;
}
assert(f2() === undefined);

const obj1 = { a: 22, f: function() { return this; } }

//like plain function call
const g = obj1.f;
assert(g() === global);

//normal object call
assert(obj1.f() === obj1);
```

# Playing with this Within a Script Continued

```
//change this using call
assert(obj1.f.call(obj1) === obj1);
assert(obj1.f.call(Array) === Array);
```

# Using bind()

bind() fixes this for a particular function.

```
> x = 44
44
> a = { x: 2, getX: function() { return this.x; } }
> a.getX()
2
> f = a.getX() //a.x
2
> f = a.getX
> f() //global x
44
> b = { x: 42 }
> f = a.getX.bind(b)
> f() //b.x
42
```

# Using bind() Continued

Can also be used to specify fixed values for some initial sequence of arguments. Can be used to implement currying.

```
> function sum(...args) {
    return args.reduce((acc, v) => acc + v);
  }
... ... undefined
> sum(1, 2, 3, 4, 5)
15
> add12 = sum.bind(null, 5, 7) //passing this as null
[Function: bound sum]
> add12(1, 2, 3, 4, 5)
27
>
```

# Difference in `this` between `function` and Fat-Arrow

- Within a nested function defined using `function`, `this` refers to global object.
- Within a nested function defined using the fat-arrow notation, `this` refers to that in the containing function.

# Difference in this between function and Fat-Arrow Example

```
> x = 22
22
> function Obj() { this.x = 42; }
> Obj.prototype.f = function() {
    return function() { return this.x; }
  }
> Obj.prototype.g = function() {
    return () => this.x;
  }
> obj = new Obj()
> obj.f()() //this refers to global obj
22
> obj.g()() //this refers to defn obj
42
>
```

# Common Idiom Used for Workaround for function this

```
> Obj.prototype.h = function() {
    const that = this;
    return function() { return that.x; }
  }
[Function]
> obj.h()() //access enclosing this via that
42
> obj.f()() //unchanged
22
>
```

# Nested Functions and Closures

- A function can include nested function definitions.
- A nested function can include references to variables declared not within itself but in its enclosing function; i.e. it has a referencing environment.
- A **closure** captures both the code of a function and its referencing environment.
- In general, JS functions are always closures which capture their referencing environment.
- Can use closures to get stronger information hiding than that provided by objects.

# Hiding Instance Variables: Bank Account

```
function Account(balance) {
  return {
    deposit: amount => balance += amount,
    withdraw: amount => balance -= amount,
    inquire: () => balance,
  };
}

a1 = new Account(100);
a2 = new Account(100);
a1.deposit(20);
a2.withdraw(20);
console.log(‘a1: ${a1.inquire()}‘);
console.log(‘a2: ${a2.inquire()}‘);
```

# Bank Account Log

```
$ nodejs account.js
a1: 120
a2: 80
$
```

# Functions Are Objects

```
> function add(a, b) { return a + b; }
undefined
> typeof add
’function’
> add.constructor
[Function: Function]
> add.x = 22
22
> add[42] = ’life’
’life’
> add(3, 5)
8
> add.x
22
> add[42]
’life’
```

# Function Properties for Memoization: Fibonacci

Memoize function by caching return values as a property of the function.

```
function fib(n) {
  return (n <= 1) ? n : fib(n - 1) + fib(n - 2);
}

//memoizing fibonacci caches results
//in function property
function memo_fib(n) {
  memo_fib.memo = memo_fib.memo || {};
  if (memo_fib.memo[n] === undefined) {
    memo_fib.memo[n] =
      (n <= 1) ? n
      : memo_fib(n - 1) + memo_fib(n - 2);
  }
```

# Function Properties for Memoization: Fibonacci Continued

```
const N = 45;
[fib, memo_fib].forEach(function(f) {
  console.time(f.name);
  console.log(‘${f.name}(${N}) = ${f(N)}‘);
  console.timeEnd(f.name);
});
```

# Fibonacci Log

```
$ ./fib.js
fib(45) = 1134903170
fib: 10080.337ms
memo_fib(45) = 1134903170
memo_fib: 0.216ms
$
```

# Defining Function Using Function() Constructor

Many dynamic languages allow converting strings into code. JavaScript supports this using `eval()` as well as via a `Function` constructor. `Function()` somewhat less problematic than `eval()`.

```
> x = max3 = new Function(’a’, ’b’,
                          ’return a > b ? a : b’)
[Function: anonymous]
> max3(4, 3)
4
> x(4, 3)
4
> x.name
’anonymous’
> x.length
2
>
```