JavaScript: Coercion and Functions

Lecture 23

Conversion of Primitive Values

		string	number	boolean
numbers	`0`	`"0"`		`false`
	`-0`	`"0"`		`false`
	`1`	`"1"`		`true`
	`NaN`	`"NaN"`		`false`
	`Infinity`	`"Infinity"`		`true`
	`-Infinity`	`"-Infinity"`		`true`
	`6.022e23`	`"6.022e+24"`		`true`

Conversion of Primitive Values

		string	number	boolean
boolean	`true`	`"true"`	`1`	
	`false`	`"false"`	`0`	
strings	`""`		`0`	`false`
	`"   "`		`0`	`true`
	`"1.2"`		`1.2`	`true`
	`"0"`		`0`	`true`
	`"one"`		`NaN`	`true`

Conversion of Primitive Values

		string	number	boolean
undefined	`undefined`	`"undefined"`	`NaN`	`false`
null	`null`	`"null"`	`0`	`false`

Summary of (Simple?) Rules

- How do numbers convert to things?
  - Boolean: 0 is false, non-0 is true (exception: NaN)
- How do strings convert to things?
  - Numbers: non-valid syntax give NaN (exception: empty/blank give 0)
  - Boolean: true, only empty string is false
- How does undefined convert to things?
  - Number: NaN
- How does null convert to things?
  - Number: 0

Easier? Column-Major View

- How do things convert to boolean?
  - Empty string is **`false`**
  - Numbers **`(+/-)`**`0` and **`NaN`** are **`false`**
  - **`undefined`** and **`null`** are **`false`**
- Aka "falsy" (vs. "truthy")
- Importance: Boolean contexts

```
if (pet)… // evaluate pet as a boolean
```

- Pitfall: &&, || may not result in a boolean
  - **`x || y`** means **`x ? x : y`** (first **`x`** converted)

```
p = "cat" || "dog"    //=> p == "cat"
```

  - Idiom: **`!!x`** forces conversion to boolean

```
p = !!("cat" || "dog") //=> p == true
```

Easier? Column-Major View

- How do things convert to Numbers?
  - Empty (and whitespace) string is `0`
  - Non-numeric strings are **`NaN`**
  - **`undefined`** is **`NaN`**
  - **`null`** is 0
- Importance: Used in **`==`** evaluation

== Evaluation is… Different

- When types do not match, coerce:
  - **`null`** & **`undefined`** (only) equal each other
  - Strings & booleans converted to *numbers*

    `"1.0" == true && "" == false`
  - Pitfall: **`NaN`** is *not equal* to **`NaN`**
- When one operand is an object:
  - Convert via **`valueOf`** (or **`toString`**)
  - Result then compared with usual **==** rules
  - Note: no coercion when both operands are references (**==** is reference equality)
- Note:
  - === never coerces

Surprising Consequences

```
false == 'false'        //=>
false == '0'            //=>
!!'0'                   //=>
('0' == 0) && (0 == '') &&
              ('0' != '') //=>
(NaN == true) || (NaN == false)
                        //=>
!!NaN                   //=>
(NaN != 0) && (!!NaN == !!0)
                        //=>
```

- dorey.github.io/JavaScript-Equality-Table

Functions are People too

- Named functions: declaration & use

```
function foo(a, b) { … }
foo("hi", 3);
```

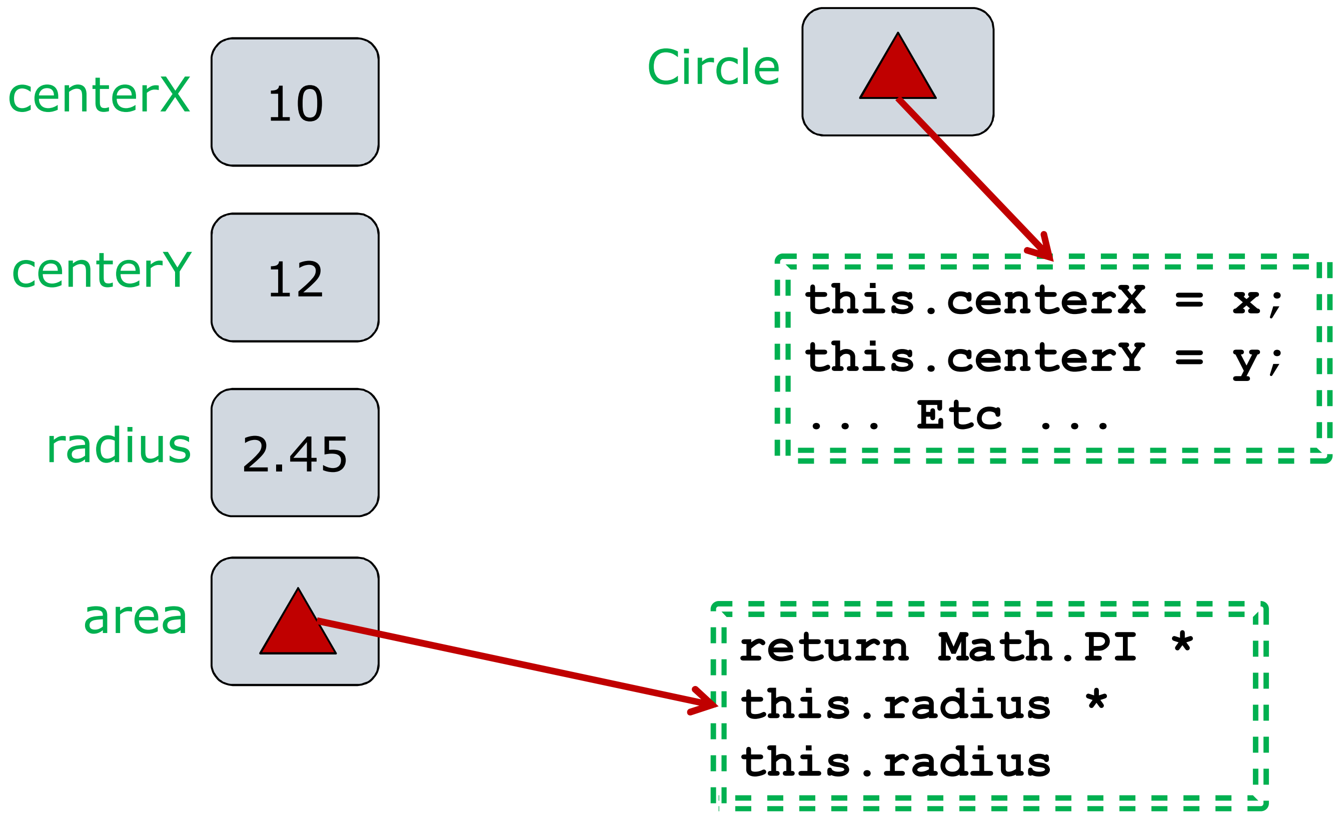
- Anonymous functions

```
function(a, b) { … }
// how do we invoke such a thing?
```

- Functions are objects (first-class citizens)
- They can be assigned to variables!

```
let foo = function(a, b) {…};
foo("hi", 3);
let bar = foo;   // cf. let bar = foo();
bar("world", 17);
```

Functions are Objects



```
this.centerX = x;
this.centerY = y;
... Etc ...
```

```
return Math.PI *
this.radius *
this.radius
```

Functions Can Be Arguments

```
function apply(x, a) {
  return x(a); // x is a function!
}

function square(i) {
  return i * i;
}

apply(square, 5) //=> 25
```

Functions Can Be Return Values

```
function grantDegree() {
  function addTitle(name) {
    return "Dr. " + name;
  }
  return addTitle; // a function!
}

let phd = grantDegree();
phd("Turing"); // phd is a function
phd(3/2);      //=> "Dr. 1.5"
```

Closures

```
function greaterThan(bound) {
  function compare (value) {
    return value > bound;
  }
  return compare; // 1-arg function
}

let testPos = greaterThan(0);
testPos(4)  //=> true
testPos(-3) //=> false
```

Closures + Anonymity

```
function greaterThan(bound) {
  function compare (value) {
    return value > bound;
  }
  return compare; // 1-arg function
}

let testPos = greaterThan(0);
testPos(4)  //=> true
testPos(-3) //=> false
```

Closures + Anonymity

```
function greaterThan(bound) {
  let compare = function(value) {
    return value > bound;
  }
  return compare; // 1-arg function
}

let testPos = greaterThan(0);
testPos(4)  //=> true
testPos(-3) //=> false
```

Closures + Anonymity

```
function greaterThan(bound) {
  return function(value) {
    return value > bound;
  }
}


let testPos = greaterThan(0);
testPos(4)  //=> true
testPos(-3) //=> false
```

An Old Idiom: IIFE

- Immediately Invoked Function Expression
  - Define *and* invoke function at the same time
- Basic forms:
  - `(function(){ /* code here */ })();`
  - `(function(){ /* code here */ }());`
- Work-around for weird JavaScript scoping
  - **`var`** scopes variables to the enclosing *function*
  - IIFE creates a lexical scope (with closures)
- Modern JavaScript has **`let`** (and **`const`**)
  - These scope variables to the enclosing *block*
  - General advice: prefer **`let`** to **`var`**
  - IIFEs are still encountered in the wild

Summary

- Truthy, falsey, and friends
  - Type coercion is everywhere
  - Coerce to boolean in conditionals
  - Coerce to number for ==
- Functions as first-class citizens
  - Can be passed as arguments
  - Can be returned as return values!
  - Closure: carry their context