CSE 341
Lecture 25

More about JavaScript functions

slides created by Marty Stepp
http://www.cs.washington.edu/341/
First-class functions

• JS functions are *first-class* objects. You can:
  ▪ store a (reference to a) function in a variable
  ▪ create an array of functions
  ▪ use a function as a property of an object (a method)
  ▪ pass a function as a parameter; return a function
  ▪ write functions that take varying numbers of parameters
  ▪ write higher-order functions (that take others as params)
  ▪ define functions inside functions (nested functions)
  ▪ define anonymous functions (lambdas)
  ▪ store properties inside functions
function name(paramName, ..., paramName) {
  statements;
}

• example:

  function sumTo(n) {
    var sum = 0;
    for (var i = 1; i <= n; i++) {
      sum += i;
    }
    return sum;
  }
function maybeReturn(n) {
    if (n % 2 == 0) {
        return "even";
    }
    // else return undefined
}

• parameter and return types are not declared
  ▪ the function can return anything it wants
  ▪ if a function returns nothing, it returns undefined
  ▪ a function can sometimes return values and sometimes not
**Calling a function**

```javascript
functionName(expr, ..., expr)
```

- **example:**
  - `sumTo(6)` // returns 21

- **extra parameters passed are ignored:**
  - `sumTo(3, "hello", null, 42)` // returns 6

- **expected parameters not passed are undefined:**
  - `sumTo()` // returns 0
Optional parameters

function greet(name, msg) {
    if (typeof(msg) === "undefined") {
        msg = "Hello";
    }
    print(msg + " to you, " + name);
}

> greet("Bob", "Good day");
Good day to you, Bob
> greet("Sue");
Hello to you, Sue

- to have an optional parameter, check whether it is defined
Object as argument specifier

```javascript
function mealCost(argObj) {
    var amt = argObj["subtotal"]; 
    if (argObj["tax"]) { amt *= 1 + argObj["tax"]; }
    if (argObj["tip"]) { amt *= 1 + argObj["tip"]; }
    if (argObj["donation"]){ amt += argObj["donation"]; }
    return amt;
}

> mealCost({subtotal: 50.0, tip: .15})
57.5
> mealCost({subtotal: 10.0, tax: .08, donation: true})
11.8
```

- specify many parameters as properties of a single object
  - can pass many args in any order; optional args; clear naming
  - this style is seen in JavaScript libraries (jQuery, Prototype)
Variadic functions (var-args)

```javascript
function addAll() {
    var sum = 0;
    for (var i = 0; i < arguments.length; i++) {
        sum += arguments[i];
    }
    return sum;
}

- addAll(1, 7, 4, 3) returns 15
- addAll(1, 2, "3", 4, 5) returns "3345"

- each function has an array property* named arguments that stores all parameter values passed to it
  - can be used to create variadic (var-args) functions

* actually a duck-typed array-like object with a Length field
Anonymous functions (lambdas)

function(paramName, ..., paramName) {
    statements;
}

• anonymous functions can be stored, passed, returned

  function foo(x, f) { return f(x) + 1; }
  foo(5, function(n) { return n * n; })
  26

• Exercise: Sort an array of strings case-insensitively.
• Exercise: Sort an array of names by last, then first, name.
Two ways to declare a function

• The following are equivalent:

```javascript
function name(params) {
    statements;
}

var name = function(params) {
    statements;
}

var squared = function(x) {
    return x*x;
};
```
## Array higher-order functions *

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>every(function)</code></td>
<td>accepts a function that returns a boolean value and calls it on each element until it returns <code>false</code></td>
</tr>
<tr>
<td><code>filter(function)</code></td>
<td>accepts a function that returns a boolean; calls it on each element, returning a new array of the elements for which the function returned <code>true</code></td>
</tr>
<tr>
<td><code>forEach(function)</code></td>
<td>applies a &quot;void&quot; function to each element</td>
</tr>
<tr>
<td><code>map(function)</code></td>
<td>applies function to each element; returns new array</td>
</tr>
<tr>
<td><code>reduce(function)</code></td>
<td>accepts a function that accepts pairs of values and combines them into a single value; calls it on each element starting from the front, using the given <code>initialValue</code> (or element [0] if not passed)</td>
</tr>
<tr>
<td><code>reduce(function, initialValue)</code></td>
<td>reduceRight starts from the end of the array</td>
</tr>
<tr>
<td><code>reduceRight(function)</code></td>
<td>accepts a function that returns a boolean value and applies it to each element until it returns <code>true</code></td>
</tr>
<tr>
<td><code>reduceRight(function, initialValue)</code></td>
<td></td>
</tr>
<tr>
<td><code>some(function)</code></td>
<td></td>
</tr>
</tbody>
</table>

*most web browsers are missing some/all of these methods*
Higher-order functions in action

```javascript
> var a = [1, 2, 3, 4, 5];
> a.map(function(x) { return x*x; })
1,4,9,16,25

> a.filter(function(x) { return x % 2 == 0; })
2,4
```

- **Exercise:** Given an array of strings, produce a new array that contains only the capitalized versions of the strings that contained 5 or more letters.
// adds 1 to each element of an array of numbers
function incrementAll(a) {
    function increment(n) { return n + 1; }
    var result = a.map(increment);
    return result;
}

• functions can be declared inside of other functions
  ▪ the scope of the inner function is only within the outer one
Invocation patterns

- functions can be invoked in four ways in JavaScript:
  - as a normal function
  - as a method of an object
  - as a constructor
  - through their apply property
Functions as methods

• an object's **methods** are just properties that are functions
  - the function uses the **this** keyword to refer to the object

```javascript
> var teacher = {
    name: "Tyler Durden",
    salary: 0.25,
    greet: function(you) {
        print("Hi " + you + ", I'm " + this.name);
    },
    toString: function() {
        return "Prof. " + this.name;
    }
};
> teacher.greet("kids");
Hi kids, I'm Tyler Durden
```


Function binding

```javascript
{ ...
  propertyName: function, // bind at declaration
  ...
} // declaration

object.propertyName = function; // bind later
```

- when a function is stored as a property of an object, a copy of that function is *bound* to the object
  - calling the function through that object will cause that object to be used as `this` during that particular call
  - if you don't call the function through the object, that object won't be used as `this`
function printMe()
{
    print("I am " + this);
}

> teacher.print = printMe;
> teacher.print();
I am Prof. Tyler Durden
> printMe();
I am [object global]
> ({p: printMe}).p()
I am [object Object]
> var temp = teacher.print;
> temp();
I am [object global]
Aside: Web event handlers

`<button id="b1">Click Me</button>`  

```
var b1 = document.getElementById("b1");
b1.onclick = function() { ... };  
```

- most JavaScript code in web pages is *event-driven*
  - elements in the HTML have events that can be handled
  - you specify a JS function to run when the event occurs
  - the function can access/modify the page's appearance
Invoking with apply

```
func.apply(thisObj, arguments);
```

- You can call a function using its apply property
  - allows you to set this to be anything you want
  - allows you to pass a function its arguments as an array

```javascript
var o = ({toString: function(){return "!"}});
> apply(printMe, o, []);
I am !
```

**Exercise:** Write a function `callBoth` that takes two functions and an array of parameters and calls both, passing them those parameters, and printing both results.
function compose(f, g) {
    return function() {
        return f(g.apply(this, arguments));
    };
}

• JavaScript has no built-in syntax for composing functions
  ▪ but you can use apply to write a helper for composition
function **toArray**(a, i) {       // converts a
    var result = [], i = i || 0;  // duck-typed obj
    while (i < a.length) {        // into an array
        result.push(a[i++]);
    }
    return result;
}

function **curry**(f) {       // Usage: curry(f, arg1, ...)
    var args = toArray(arguments, 1);  // remove f
    return function() {
        return f.apply(this,
                        args.concat(toArray(arguments)));
    }
}

• JavaScript doesn't include syntax for currying functions
  ▪ but we can add such functionality ourselves
## Methods of Function objects

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>.toString()</code></td>
<td>string representation of function (its code, usually)</td>
</tr>
<tr>
<td><code>.apply(this, args)</code></td>
<td>calls a function using the given object as <code>this</code> and passing the given array of values as its parameters</td>
</tr>
<tr>
<td><code>.call(this, arg1, ...)</code></td>
<td>var-args version of apply; pass args without array</td>
</tr>
<tr>
<td><code>.bind(this)</code></td>
<td>attaches the function's <code>this</code> reference to given obj</td>
</tr>
</tbody>
</table>
A JS library: Underscore

http://documentcloud.github.com/underscore/

- Adds functional language methods to JavaScript.
  - **Collections**: each, map, reduce, reduceRight, detect, select, reject, all, any, include, invoke, pluck, max, min, sortBy, sortedIndex, toArray, size
  - **Arrays**: first, rest, last, compact, flatten, without, uniq, intersect, zip, indexOf, lastIndexOf, range
  - **Functions**: bind, bindAll, memoize, delay, defer, wrap, compose
  - **Objects**: keys, values, functions, extend, clone, tap, isEqual, isEmpty, isElement, isArray, isArguments, isFunction, isString, isNumber, isBoolean, isDate, isRegExp, isNaN, isNull, isUndefined
  - **Utility**: noConflict, identity, times, breakLoop, mixin, uniqueld, template
  - **Chaining**: chain, value

```javascript
> _( [1, 4, 2, 7, 3, 5] ).max() 
7
```