MODULE 4

ADVANCED JAVASCRIPT

Functions

> Functions extend objects with an additional feature:

– It can be executed.

JavaScript Functions Are Objects

> The JavaScript text of a function is called the function's body

> Functions have an additional feature:

– The body is code that can be executed

```
function sayHello() {
  alert("Hello World");
}
// dialog appears with message
// “Hello World”
sayHello();
sayHello.top = 5;
// y is assigned value 5
y = sayHello.top;
```

Calls and Object References

> Parentheses appended to the function name indicate a call to the function.

> No parentheses appended to the function name, indicates a reference to the function object.

```
function one() {
  return 1; // returns the value 1
}
y = one(); // sets y to the value 1
z = one;
y = z(); // sets y to the value 1
one.two = 2;
alert( one() + one.two)
```

JavaScript Methods

> The ECMAScript specification states that a function stored in a property of another object is called a method.

```
function myArea( ) {
  return ( this.height * this.width);
}
function myRectangle( t, l, h, w) {
  this.top = t;
  this.left = l;
  this.height = h;
  this.width = w;
  this.area = myArea;
}
x = new myRectangle( 0, 0, 2, 3);
y = x.area(); // y is assigned value: 6
```

Function Parameters

> Parameters are the same as local variables.

```
function twiceAlert( msg1 , msg2 ){
  alert(msg1);
  alert(msg2);
}
```

> Parameters and local variables are completely separate from a Function object's properties.

– Parameters and local variables belong to an instance of a call to the code in the Function body.

– Properties belong to the Function object.

No Parameter Checking

> No compile-time or runtime argument checking:

```
function twiceAlert( msg1, msg2 ) {
  alert(msg1);
  alert(msg2);
}
twiceAlert('Hello World');
twiceAlert('Hello World', "Hello
Again",
"Ignored Greeting");
```

Constructors Are Functions

> A constructor is used to create new object instances.

> When a function is invoked with the new operator, that function is acting as a constructor in JavaScript.

```
function myPoint( x, y) {
  this.x = x;
  this.y = y;
}
x = new myPoint( 10, 27);
// x gets a new myPoint object
y = new myPoint( 17, 25);
// y gets a new myPoint object
alert( x.x + y.y);
```

Constructors Are Functions

> In a constructor, the keyword `this` refers to the new instance of the object.

```
function myPoint( x, y) {
  this.x = x;
  this.y = y;
  var z = 10;
  this.z = z;
}
```

Constructors Are Functions

> Parameters are local variables.

> Variables are not properties:

- this.x is not the same as the parameter x;
- this.y is not y.
- this.z is not z.

> The declaration of an function object instantiates an original instance of that object.

Constructor Inner-Workings

1. Create a new empty object.
2. Add the constructor's prototype property to the new object's prototype chain, to resolve property references.
3. Execute the constructor's function body using the new object for the value of this, and using the current call scope.

> If Step 3 returns an object, return that object; otherwise return the newly created object (Step 1).

Prototype Inheritance

> When a property is accessed, the value of first instance of a property with that name found in the prototype chain is used.

```
function myPoint( x, y ) {
  this.x = x;
  this.y = y;
}
x = new myPoint(50, 100);
alert( x.x + "," + x.y+ "," + x.z);
myPoint.prototype.z = 150;
alert(x.x + "," + x.y+ "," + x.z );
alert( myPoint.x + "," + myPoint.y +
"," + myPoint.z);
```

Prototype Inheritance

> Altering a property value shadows other same-named properties in the Prototype chain.

```
function myPoint( x, y ) {
  this.x = x;
  this.y = y;
}
x = new myPoint(50, 100);
myPoint.prototype.z = 150;
alert( x.z );
x.z = 33;
alert( x.z);
// dialog appears with message: 150
alert( myPoint.prototype.z);
```

Prototype Inheritance

> All of the properties of the constructor's prototype are available to the child objects

> Even methods that are added to the constructor's prototype after the children were instantiated.

```
function myPoint( x, y ) {
  this.x = x;
  this.y = y;
}
x = new myPoint(50, 100);
x.helloWorld() // Error
myPoint.prototype.helloWorld =
function(){ ("x=" + this.x +", y="
            + this.y))};
x.helloWorld();
```

Prototype Inheritance

> Prototypical inheritance even applies to Built-In Objects.

```
x = new String("ABCDEFG");
x.helloWorld() // Error
String.prototype.helloWorld =
function() {alert("Hello World")};
x.helloWorld(); // alerts: Hello World.
```

The Function Keywords and Objects

> **`function`** (lower case f) is a reserved keyword like new.

> The function keyword is not the same as the name of the **`Function`** (upper case F) built-In Object.

> The **`function`** keyword is used as part of a special syntax to declare a new instance of **`Function`** Objects.

> Typically, do not use **`Function`**

The Function Keywords and Objects

> **function** (lower case f) is a reserved keyword like new.

> The function keyword is not the same as the name of the **Function** (upper case F) built-In Object.

> The **function** keyword is used as part of a special syntax to declare a new instance of **Function** Objects.

> Typically, do not use **Function**

Creating Objects

> Creating an empty object:

```
var obj_1 = {};
var obj_2 = new Object();
var obj_3 = Object.create(null);
```

> Creating an object:

```
var person = {
    "full-name" : "John Doe",
    age: 35,
     address: {
        address_line1: "Clear Trace, Glaslyn, Arkansas",
        "postal code": "76588-89"
    }
};
```

Creating Objects

> Creating an object with `new`:

```
function Tree(type, height, age) {
        this.type = type;
        this.height = height;
        this.age = age;
}

var mapleTree =
    new Tree("Big Leaf Maple", 80, 50);
```

Accessing Object Properties

```
var myObject = {
   name: "luggage",
   length: 75,
   specs: {
    material: "leather",
    waterProof: true
   }
}
```

> Getting and setting properties:

```
myObject["name"];                    // "luggage"
myObject.name;                       // "luggage"
myObject.specs.material;             // "leather"
myObject["specs"]["material"];       // "leather"
myObject.width;                      // undefined
myObject.tags.number;                // TypeError thrown
myObject["name"] = "suitcase";       // name : "suitcase"
myObject.name = "suitcase";          // name : "suitcase"
myObject.width = 40;                 // creates a new property
myObject.tags.number = 6;            // TypeError thrown
```

Object Methods

> Object method example:

```
var myObj = {
        print: function() {
             console.log("Hello World!");
        }
};

myObj.print();
```

Creating Arrays

> Initializing an array:

```
var myNewEmptyArray = [];
var numbers = [1, 2, 3, 4, 'five'];
```

> Creating an array with `new`:

```
var myNewEmptyArray = new Array();
var myNewNonEmptyArray = new Array(15);
var numbers = new Array(1, 2, 3, 4, 'five');
```

Iterating Through Objects and Arrays

> Iterating object properties:

```
for(var key in obj) {
  if (obj.hasOwnProperty(key)) {
    // no parent properties
    console.log(obj[key]);
  }
}
```

> Iterating arrays:

```
for (var i = 0; i < c.length; i++) {
       console.log(c[i]);
}
```

JavaScript Arrays as Collections

> To modify arrays and treat them as collections, you can use several methods:

– `array.pop()` removes and returns the last element.

– `array.push(value)` adds the value at the end.

– `array.shift()` removes and returns the first element.

– `array.unshift(value)` adds the value at the start.

– `array.indexOf(value)` gets the index of value.

– `array.join(", ")` joins all items in a single string.

JavaScript Arrays as Collections

> To modify arrays and treat them as collections, you can use several methods:
  – `array.sort(function)` sorts all items by using the provided function. The function takes two arguments and returns less than 0 if the first argument comes first, 0 if the arguments are equal, and greater than 0 if the second comes first.

JavaScript RegExp Object

> A regular expression is an object that describes a pattern of characters.

> When you search in a text, you can use a pattern to describe what you are searching for.

> A simple pattern can be one single character.

> A more complicated pattern can consist of more characters, and can be used for parsing, format checking, substitution and more.

> Regular expressions are used to perform powerful pattern-matching and "search-and-replace" functions on text.

Example

```
var pat1=new RegExp("e");
document.write(pat1.test("The best things
in life are free"));
var pat2=new RegExp("e");
document.write(pat2.exec("The best things
in life are free"));
```

Modifiers

Modifier	Description
I	Perform case-insensitive matching
g	Perform a global match (find all matches rather than stopping after the first match)
m	Perform multiline matching

Brackets

Expression	Description
[abc]	Find any character between the brackets
[^abc]	Find any character NOT between the brackets
[0-9]	Find any digit between the brackets
[^0-9]	Find any digit NOT between the brackets
(x|y)	Find any of the alternatives specified

Metacharacters

Metacharacter	Description
.	Find a single character
\w	Find a word character
\W	Find a non-word character
\d	Find a digit
\D	Find a non-digit character
\s	Find a whitespace character
\S	Find a non-whitespace character
\b	Find a match at the beginning/end of a word
\B	Find a match not at the beginning/end of a word
\0	Find a NUL character
\n	Find a new line character
\f	Find a form feed character

Metacharacters

Metacharacter	Description
\r	Find a carriage return character
\t	Find a tab character
\v	Find a vertical tab character
\xxx	Find the character specified by an octal number xxx
\xdd	Find the character specified by a hexadecimal number dd
\uxxxx	Find the Unicode character specified by a hexadecimal number xxxx

Quantifiers

Quantifier	**Description**
n+	Matches any string that contains at least one n
n*	Matches any string that contains zero/more occurrences of n
n?	Matches any string that contains zero/one occurrences of n
n{X}	Matches any string that contains a sequence of X n's
n{X,Y}	Matches any string that contains a sequence of X to Y n's
n{X,}	Matches any string that contains a sequence of at least X n's
n$	Matches any string with n at the end of it
^n	Matches any string with n at the beginning of it
?=n	Matches any string that is followed by a specific string n
?!n	Matches any string that is not followed by a specific string n

RegExp Object Methods

Method	Description
compile()	Deprecated in version 1.5. Compiles a regular expression
exec()	Tests for a match in a string. Returns the first match
test()	Tests for a match in a string. Returns true or false
toString()	Returns the string value of the regular expression

Example

```
meyveler= [ "elma", "armut", "kiraz", "karpuz",
            "kavun", "muz"]
pattern= new RegExp("^k.*z$")
for (meyve in meyveler)
     if( pattern.test(meyveler[meyve]) )
        console.log(meyveler[meyve]);
pattern= new RegExp("^[a-z]{1,4}$")
for (meyve in meyveler)
     if( pattern.test(meyveler[meyve]) )
        console.log(meyveler[meyve]);
```

OBJECT CLONING

Object Cloning

```
function clone(obj) {
   var target = {};
   for (var i in obj) {
    if (obj.hasOwnProperty(i)) {
     target[i] = obj[i];
    }
   }
   return target;
  }
```

Object Cloning

```
var oldObject = {
  a: 1,
  b: 2,
  c: 3,
  d: 4,
  e: 5,
  f: function() {
   return 6;
  },
  g: [7, 8, 9]
 };
```

Object Cloning

```
var newObject= clone(oldObject);
```

Object Cloning

```
newObject= JSON.parse(
    JSON.stringify( oldObject)
);
```

Object Cloning

```
var newObject = {};
jQuery.extend(newObject, oldObject);
```

Object Cloning

```
newObject= {} ;
newObject= Object.create(oldObject);
```

How to remove a property

```
function Circle(x,y,radius){
    this.x= x;
    this.y= y;
    this.radius= radius;
    this.area= function(){
        return 3.1415 * radius * radius;
    }
}
```

How to remove a property

```
delete Circle.prototype.x
unitCircle= new Circle(0,0,1);
console.log(unitCircle.area());
delete unitCircle.area;
console.log(unitCircle.area());
undefined is not a function
```

UNDERSTAND THE DIFFERENCE BETWEEN FUNCTION, METHOD, AND CONSTRUCTOR CALLS

Function, Method, and Constructor Calls

> If you’re familiar with object-oriented programming, you’re likely accustomed to thinking of functions, methods, and class constructors as three separate things.

> In JavaScript, these are just three different usage patterns of one single construct: functions.

```
function hello(username) {
    return "hello, " + username;
}
hello("Keyser Söze"); // "hello, Keyser Söze"
```

Function, Method, and Constructor Calls

```
var obj = {
    hello: function() {
        return "hello, " + this.username;
    },
    username: "Hans Gruber"
};
obj.hello(); // "hello, Hans Gruber"
```

Function, Method, and Constructor Calls

```
var obj = {
    hello: function() {
        return "hello, " + this.username;
    },
    username: "Hans Gruber"
};
obj.hello(); // "hello, Hans Gruber"

var obj2 = {
    hello: obj.hello,
    username: "Boo Radley"
};
obj2.hello(); // "hello, Boo Radley"
```

Function, Method, and Constructor Calls

```
function hello() {
    return "hello, " + this.username;
}
var obj1 = {
    hello: hello,
    username: "Gordon Gekko"
};
obj1.hello(); // "hello, Gordon Gekko"

var obj2 = {
    hello: hello,
    username: "Biff Tannen"
};
obj2.hello(); // "hello, Biff Tannen"
```

Function, Method, and Constructor Calls

```
function hello() {
    return "hello, " + this.username;
}
hello();
```

Function, Method, and Constructor Calls

```
function hello() {
    return "hello, " + this.username;
}
hello(); // "hello, undefined"
```

Function, Method, and Constructor Calls

```
function User(name, passwordHash) {
    this.name = name;
    this.passwordHash = passwordHash;
}

var u = new User("sfalken",
                 "0ef33ae791068ec64b502d6cb0191387");
u.name; // "sfalken"
```

Things to Remember

> Method calls provide the object in which the method property is looked up as their receiver.

> Function calls provide the global object (or undefined for strict functions) as their receiver. Calling methods with function call syntax is rarely useful.

> Constructors are called with new and receive a fresh object as their receiver.

GET COMFORTABLE
USING HIGHER-ORDER FUNCTIONS

Functions

> Higher-order functions are nothing more than functions that take other functions as arguments or return functions as their result.

```
function compareNumbers(x, y) {
    if (x < y) {
        return -1;
    }
    if (x > y) {
        return 1;
    }
    return 0;
}
[3, 1, 4, 1, 5, 9].sort(compareNumbers);

// [1, 1, 3, 4, 5, 9]
```

Functions

```
[3, 1, 4, 1, 5, 9].sort(function(x, y) {
    if (x < y) {
        return -1;
    }
    if (x > y) {
        return 1;
    }
    return 0;
}); // [1, 1, 3, 4, 5, 9]
```

Functions

> Higher-order functions can often simplify your code and eliminate tedious boilerplate

```
var names = ["Fred", "Wilma", "Pebbles"];
var upper = [];
for (var i = 0, n = names.length; i < n; i++) {
    upper[i] = names[i].toUpperCase();
}
upper; // ["FRED", "WILMA", "PEBBLES"]
```

Functions

> Higher-order functions can often simplify your code and eliminate tedious boilerplate

```
var names = ["Fred", "Wilma", "Pebbles"];
var upper = names.map(function(name) {
    return name.toUpperCase();
});
upper; // ["FRED", "WILMA", "PEBBLES"]
```

Things to Remember

> Higher-order functions are functions that take other functions as arguments or return functions as their result.

> Familiarize yourself with higher-order functions in existing libraries.

> Learn to detect common coding patterns that can be replaced by higher-order functions.

USE CALL TO CALL METHODS WITH A CUSTOM RECEIVER

Use call to Call Methods with a Custom Receiver

- Ordinarily, the receiver of a function or method (i.e., the value bound to the special keyword this) is determined by the syntax of its caller.
- In particular, the method call syntax binds the object in which the method was looked up to this. However, it is sometimes necessary to call a function with a custom receiver, and the function may not already be a property of the desired receiver object.

Use call to Call Methods with a Custom Receiver

```
// what if obj.temporary already existed?
obj.temporary = f;
var result = obj.temporary(arg1, arg2, arg3);
// what if obj.temporary already existed?
delete obj.temporary;
```

Use call to Call Methods with a Custom Receiver

```
var table = {
    entries: [],
    addEntry: function(key, value) {
        this.entries.push({ key: key, value: value });
    },
    forEach: function(f, thisArg) {
        var entries = this.entries;
        for (var i = 0, n = entries.length; i < n; i++) {
            var entry = entries[i];
            f.call(thisArg, entry.key, entry.value, i);
        }
    }
};


table1.forEach(table2.addEntry, table2);
```

Things to Remember

> Use the call method to call a function with a custom receiver.

> Use the call method for calling methods that may not exist on a given object.

> Use the call method for defining higher-order functions that allow clients to provide a receiver for the callback.

USE APPLY TO CALL FUNCTIONS WITH DIFFERENT NUMBERS OF ARGUMENTS

Use apply to Call Functions with Different Numbers of Arguments

> Imagine that someone provides us with a function that calculates the average of any number of values:

```
average(1, 2, 3);                       // 2
average(1);                             // 1
average(3, 1, 4, 1, 5, 9, 2, 6, 5); // 4
average(2, 7, 1, 8, 2, 8, 1, 8);    // 4.625
```

> The average function is an example of what's known as a *variadic* or *variable-arity* function (the *arity* of a function is the number of arguments it expects): It can take any number of arguments.

Use apply to Call Functions with Different Numbers of Arguments

```
averageOfArray([1, 2, 3]);                       // 2
averageOfArray([1]);                             // 1
averageOfArray([3, 1, 4, 1, 5, 9, 2, 6, 5]); // 4
averageOfArray([2, 7, 1, 8, 2, 8, 1, 8]);    // 4.625
```

Use apply to Call Functions with Different Numbers of Arguments

```
var buffer = {
    state: [],
    append: function() {
       for (var i = 0, n = arguments.length; i < n; i++) {
           this.state.push(arguments[i]);
       }
    }
};

buffer.append("Hello, ");
buffer.append(firstName, " ", lastName, "!");
buffer.append(newline);

buffer.append.apply(buffer, getInputStrings());
```

Things to Remember

> Use the apply method to call variadic functions with a computed array of arguments.

> Use the first argument of apply to provide a receiver for variadic methods.

USE ARGUMENTS
TO CREATE VARIADIC FUNCTIONS

Use arguments to Create Variadic Functions

- A variadic average function, which can process an arbitrary number of arguments and produce their average value.
- How can we implement a variadic function of our own?

```
function averageOfArray(a) {
    for (var i = 0, sum = 0, n = a.length; i < n; i++) {
        sum += a[i];
    }
    return sum / n;
}
averageOfArray([2, 7, 1, 8, 2, 8, 1, 8]); // 4.625
```

Use arguments to Create Variadic Functions

```
function average() {
    for (var i = 0, sum = 0, n = arguments.length;
         i < n;
         i++) {
        sum += arguments[i];
    }
    return sum / n;
}
```

```
function average() {
    return averageOfArray(arguments);
}
```

Things to Remember

> Use the implicit arguments object to implement variable-arity functions.

> Consider providing additional fixed-arity versions of the variadic functions you provide so that your consumers don’t need to use the apply method.

NEVER MODIFY THE ARGUMENTS OBJECT

Never Modify the `arguments` Object

- The arguments object may look like an array, but sadly it does not always behave like one.
- Programmers familiar with Perl and UNIX shell scripting are accustomed to the technique of “shifting” elements off of the beginning of an array of arguments
- JavaScript’s arrays do in fact contain a shift method, which removes the first element of an array and shifts all the subsequent elements over by one.
- **`arguments`** object itself is not an instance of the standard Array type, so we cannot directly call **`arguments.shift()`**.

Never Modify the `arguments` Object

```
function callMethod(obj, method) {
    var shift = [].shift;
    shift.call(arguments);
    shift.call(arguments);
    return obj[method].apply(obj, arguments);
}

var obj = {
    add: function(x, y) { return x + y; }
};
callMethod(obj, "add", 17, 25);
// error: cannot read property "apply" of undefined
```

Never Modify the `arguments` Object

- As a consequence, it is much safer never to modify the arguments object.
- This is easy enough to avoid by first copying its elements to a real array.
- A simple idiom for implementing the copy is

```
var args = [].slice.call(arguments);
```

Never Modify the `arguments` Object

```
function callMethod(obj, method) {
    var args = [].slice.call(arguments, 2);
    return obj[method].apply(obj, args);
}

var obj = {
    add: function(x, y) { return x + y; }
};
callMethod(obj, "add", 17, 25); // 42
```

Things to Remember

> Never modify the arguments object.

> Copy the arguments object to a real array using **`[].slice.call(arguments)`** before modifying it.

USE A VARIABLE TO SAVE A REFERENCE TO ARGUMENTS

Use a Variable to Save a Reference to arguments

> An *iterator* is an object providing sequential access to a collection of data.

> A typical API provides a next method that provides the next value in the sequence.

```
var it = values(1, 4, 1, 4, 2, 1, 3, 5, 6);
it.next(); // 1
it.next(); // 4
it.next(); // 1
```

Use a Variable to Save a Reference to arguments

```
function values() {
    var i = 0, n = arguments.length;
    return {
        hasNext: function() {
            return i < n;
        },
        next: function() {
            if (i >= n) {
                throw new Error("end of iteration");
            }
            return arguments[i++]; // wrong arguments
        }
    };
}
          var it = values(1, 4, 1, 4, 2, 1, 3, 5, 6);
          it.next(); // undefined
          it.next(); // undefined
          it.next(); // undefined
```

Use a Variable to Save a Reference to arguments

```
function values() {
    var i = 0, n = arguments.length, a = arguments;
    return {
     hasNext: function() {
         return i < n;
     },
     next: function() {
         if (i >= n) {
             throw new Error("end of iteration");
         }
         return a[i++];
     }
};
var it = values(1, 4, 1, 4, 2, 1, 3, 5, 6);
it.next(); // 1
it.next(); // 4
it.next(); // 1
```

Things to Remember

- Be aware of the function nesting level when referring to arguments.
- Bind an explicitly scoped reference to arguments in order to refer to it from nested functions.

USE BIND TO EXTRACT METHODS WITH A FIXED RECEIVER

Receiver

> With no distinction between a method and a property whose value is a function, it's easy to extract a method of an object and pass the extracted function directly as a callback to a higher-order function.

> But it's also easy to forget that an extracted function's receiver is not bound to the object it was taken from.

Receiver

> Imagine a little string buffer object that stores strings in an array that can be concatenated later:

```
var buffer = {
    entries: [],
    add: function(s) {
        this.entries.push(s);
    },
    concat: function() {
        return this.entries.join("");
    }
};

var source = ["867", "-", "5309"];
// error: entries is undefined
source.forEach(buffer.add);
```

Receiver

```
var buffer = {
    entries: [],
    add: function(s) {
        this.entries.push(s);
    },
    concat: function() {
        return this.entries.join("");
    }
};
var source = ["867", "-", "5309"];
source.forEach(buffer.add, buffer);
buffer.join(); // "867-5309"
```

Receiver

```
var buffer = {
    entries: [],
    add: function(s) {
        this.entries.push(s);
    },
    concat: function() {
        return this.entries.join("");
    }
};
var source = ["867", "-", "5309"];
source.forEach(function(s) {
    buffer.add(s);
});
buffer.join(); // "867-5309"
```

Receiver

```
var buffer = {
    entries: [],
    add: function(s) {
        this.entries.push(s);
    },
    concat: function() {
        return this.entries.join("");
    }
};

var source = ["867", "-", "5309"];
source.forEach(buffer.add.bind(buffer));
buffer.join(); // "867-5309"
```

Things to Remember

> Beware that extracting a method does not bind the method’s receiver to its object.

> When passing an object’s method to a higher-order function, use an anonymous function to call the method on the appropriate receiver.

> Use bind as a shorthand for creating a function bound to the appropriate receiver.

USE BIND TO CURRY FUNCTIONS

Use bind to Curry Functions

> The bind method of functions is useful for more than just binding methods to receivers.

> Imagine a simple function for constructing URL strings from components:

```
function simpleURL(protocol, domain, path) {
    return protocol + "://" + domain + "/" + path;
}
```

> Frequently, a program may need to construct absolute URLs from site-specific path strings. A natural way to do this is with the ES5 map method on arrays:

```
var urls = paths.map(function(path) {
    return simpleURL("http", siteDomain, path);
});
```

Use bind to Curry Functions

> Notice how the anonymous function uses the same protocol string and the same site domain string on each iteration of map; the first two arguments to **`simpleURL`** are fixed for each iteration, and only the third argument is needed.

> We can use the bind method on **`simpleURL`** to construct this function automatically:

```
var urls = paths.map(
               simpleURL.bind(null, "http", siteDomain)
             );
```

Currying

> The technique of binding a function to a subset of its arguments is known as *currying,* named after the logician Haskell Curry, who popularized the technique in mathematics.

> Currying can be a succinct way to implement function delegation with less boilerplate than explicit wrapper functions.

Things to Remember

- Use bind to curry a function, that is, to create a delegating function with a fixed subset of the required arguments.
- Pass `null` or undefined as the receiver argument to curry a function that ignores its receiver.

PREFER CLOSURES TO STRINGS FOR ENCAPSULATING CODE

Prefer Closures to Strings for Encapsulating Code

> Functions are a convenient way to store code as a data structure that can be executed later.

> This enables expressive higher-order abstractions such as map and forEach, and it is at the heart of JavaScript’s asynchronous approach to I/O

> At the same time, it’s also possible to represent code as a string to pass to eval.

> Programmers are then confronted with a decision to make: Should code be represented as a function or as a string?

Prefer Closures to Strings for Encapsulating Code

```
function repeat(n, action) {
    for (var i = 0; i < n; i++) {
        eval(action);
    }
}

var start = [], end = [], timings = [];
repeat(1000,
       "start.push(Date.now()); f(); end.push(Date.now())");
for (var i = 0, n = start.length; i < n; i++) {
    timings[i] = end[i] - start[i];
}
```

Things to Remember

> Never include local references in strings when sending them to APIs that execute them with eval.

> Prefer APIs that accept functions to call rather than strings to eval.

ITEM 28

AVOID RELYING ON THE TOSTRING METHOD OF FUNCTIONS

Avoid Relying on the toString Method of Functions

```
(function(x) {
    return x + 1;
}).toString();
```

Avoid Relying on the toString Method of Functions

```
(function(x) {
    return x + 1;
}).toString();

// "function (x) {\n    return x + 1;\n}"
```

Avoid Relying on the toString Method of Functions

```
(function(x) {
    return x + 1;
}).bind(16).toString();

// "function (x) {\n    [native code]\n}"
```

Avoid Relying on the toString Method of Functions

```
(function(x) {
    return function(y) {
        return x + y;
    }
})(42).toString();
// "function (y) {\n    return x + y;\n}"
```

Notice how the resultant string still contains a variable reference to x, even though the function is actually a closure that binds x to 42.

Things to Remember

- JavaScript engines are not required to produce accurate reflections of function source code via toString.
- Never rely on precise details of function source, since different engines may produce different results from toString.
- The results of toString do not expose the values of local variables stored in a closure.
- In general, avoid using toString on functions.

AVOID NONSTANDARD STACK INSPECTION PROPERTIES

Properties

```
var factorial = (function(n) {
    return (n <= 1) ? 1 : (n * arguments.callee(n - 1));
});
```

Properties

```
var factorial = (function(n) {
    return (n <= 1) ? 1 : (n * arguments.callee(n - 1));
});

    function factorial(n) {
        return (n <= 1) ? 1 : (n * factorial(n - 1));
    }
```

Properties

```
function revealCaller() {
    return revealCaller.caller;
}

function start() {
    return revealCaller();
}

start() === start; // true
```

Properties

```
function getCallStack() {
    var stack = [];
    for (var f = getCallStack.caller; f; f = f.caller) {
        stack.push(f);
    }
    return stack;
}
```

Properties

```
function getCallStack() {
    var stack = [];
    for (var f = getCallStack.caller; f; f = f.caller) {
        stack.push(f);
    }
    return stack;
}
                function f1() {
                    return getCallStack();
                }

                function f2() {
                    return f1();
                }

                var trace = f2();
                trace; // [f1, f2]
```

Properties

```
function getCallStack() {
    var stack = [];
    for (var f = getCallStack.caller; f; f = f.caller) {
        stack.push(f);
    }
    return stack;
}

     function f(n) {
         return n === 0 ? getCallStack() : f(n - 1);
     }
     var trace = f(1);
```

Properties

```
function getCallStack() {
    var stack = [];
    for (var f = getCallStack.caller; f; f = f.caller) {
        stack.push(f);
    }
    return stack;
}
     function f(n) {
         return n === 0 ? getCallStack() : f(n - 1);
     }
     var trace = f(1); // infinite loop
```

Properties

```
function getCallStack() {
    var stack = [];
    for (var f = getCallStack.caller; f; f = f.caller) {
        stack.push(f);
    }
    return stack;
}
     function f(n) {
         return n === 0 ? getCallStack() : f(n - 1);
     }
     var trace = f(1); // infinite loop
```

The best policy is to avoid stack inspection altogether.
If your reason for inspecting the stack is solely for debugging, it’s much more reliable to use an interactive debugger.

Things to Remember

- Avoid the nonstandard arguments.caller and arguments.callee, because they are not reliably portable.
- Avoid the nonstandard caller property of functions, because it does not reliably contain complete information about the stack.