Intermediate Javascript

Chris Langtiw, Training Connection

Variables

- Container to store values
- Names
  - Can consist of letters, numbers, $ and _
  - Case sensitive
- Data types are strings, numbers, booleans (true and false), `null`, `undefined`, other objects (arrays, functions, user-defined, etc.)
- Loosely typed
- Values assigned via = operator
- Value is referenced in place of variable name

```
message = ‘<p>This space for rent</p>’;
document.write(message); // <p>This space for rent</p>
```

- Do NOT put quotes around variable name

- Common operators:

```
=       ++        ()
+       --
-       +=
*       -=
/       *=
%       /=
```

- Be careful with + as concatenation takes precedence

```
var sum = 10 + ‘3’; // ‘103’
```

- parseInt() and parseFloat() can correct this

```
var sum = 10 + parseInt(‘3’);
```

null, NaN and undefined

- undefined – a variable or member that does not exist or does not have a value

  ```
  typeof abc; // undefined
  var abc;
  typeof abc; // undefined
  ```

- null – a placeholder meaning ‘no value’

  ```
  var xyz = null; // null
  ```

- NaN – a number type meaning ‘not a number’

  ```
  var product = 13 * ‘orange’; // NaN
  isNaN(product); // true
  ```

Arrays

- Collection of values (think egg carton)
- Defined as a list of comma-separated values enclosed within brackets `[]`
- Elements (individual values) referenced using numeric key starting at 0

  ```
  var names = [‘John’, ‘Peter’, ‘Nancy’, ‘Betty’];
  document.write(names[0]); // John
  document.write(names[2]); // Nancy
  ```

- Elements can be reassigned using =
- Can contain mixed data types
- Internal methods to manipulate collection

Objects

- Containers of properties
- Self-contained entity
- Properties can be any data type
- Attributes (values) and methods (functions)
- Declared as `name:value` comma-separated pairs within braces `{}`
- Properties can be accessed via dot notation or array notation
- Use `new` to create new objects (instances) based on existing objects

```
var pillbox =
{
   Sun: ‘white’,
   Mon: ‘white’,
   Tue: ‘none’,
   Wed: ‘blue’,
   Thu: ‘orange’,
   Fri: ‘red’,
   Sat: ‘green’
};
document.write(pillbox.Mon);      // dot notation
document.write(pillbox[‘Fri’]);   // array notation
var pillbox2 = new pillbox(); // create new pillbox object
pillbox2[‘Bob’] = ‘rainbow’;      // original pillbox
   intact
```

Program Flow

- Script executes top-down unless flow is disrupted
- 2 types of disruption
  - Branching
  - Looping

Branching

- Statements executed conditionally
- 3 types of branching
  - Optional path
  - Either/or
  - Multiple choice
- Use `if`, `if`…`else` and `switch`

Optional Path

```
if (condition) {
  // statements to execute
}
```

- Comparison operators:

```
==    ===  !=      !==
<     >    >=      <=
&&    ||   !
```

- Don't confuse = and ==

Either/Or

```
if (condition) {
  // execute if condition is true
} else {
  // execute if condition is false
}
```

Multiple Choice

```
// link multiple if statements
if (door == 1) {
  // door 1 code
} else if (door == 2) {
   // door 2 code
} else {
  // door 3 code
}
```

```
// multiple choice using switch statement
switch(door) {
  case 1:
   // do stuff
  break;
  case 2:
   // do stuff
  case 3:
   // do other stuff
  break;
  default:
   // if no matching case label do this stuff
  break;
}
```

```
// alternate version of switch
switch (true) {
  case door == 1:
   // do stuff
  break;
  case door > 1 && door < 7:
   // make sure ranges do not overlap
   // do stuff
  break;
  case door == 3:
   // do stuff
  break;
}
```

Ternary Operator

- Used to do inline conditional assignment or output
- Generally faster than if…else
- Format: condition ? trueValue : falseValue;

```
var isDoor1 = door == 1 ? true : false;

document.write(
  ‘This ‘ +
  (door == 1 ? ‘is ‘ : ‘is not ‘) +
  ‘door 1’
); // ternary inside () makes it an expression
```

Looping (`for` and `while`)

```
for (var c = 0; c < 10; c++) {
  document.write(c);
}
```

```
var c = 0;
while (c < 10) {
  document.write(c);
  c++;
}
```

```
var c = 0;
do {
  document.write(c);
  c++;
} while (c < 10);
```

Looping

- Used to repeat one or more statements
- 2 basic types of loops
  - for used when number of iterations is known
  - while used when number of iterations is unknown or unimportant
  - while performs zero or more iterations
  - do…while performs one or more iterations

Functions

```
function greeting() {
  document.write(‘<p>Hello!</p>’);
}
greeting();

// using return value rather than direct output
function greeting2() {
  return ‘Hello!’;
}
document.write(‘<h1>’ + greeting2() + ‘</h2>’);
```

```
function foo() {
  // functions have their own scope
  var c = 100; // DON’T forget the var
  return c;
}
var c = 1;
document.write(c); // 1
document.write(foo()); // 100
document.write(c); // 1
```

```
function foo() {
  c = 100; // note lack of var keyword
  return c;
}
var c = 1;
document.write(c); // 1
document.write(foo()); // 100
document.write(c); // 100
```

```
// parameters can be passed into a function
function greeting(name) {
  return ‘Hello ‘ + name + ‘!’;
}
document.write(
‘<h1>’ + greeting(‘Hans’) + ‘</h1>’
);

function foo(a, b) {
  b = typeof b == ‘undefined’? 10: b;
  return a * b;
}
document.write(foo(10)); // 100
```

```
// assign an anonymous function
var foo = function() {
  return 100;
};
document.write(foo());

var Car = {
  running: false,
  startEngine: function() {
   // ‘this’ refers to current object
   this.running = true;
  }
}
var myCar = new Car();
myCar.startEngine(); // call startEngine method
```

```
function foo() {
  return ‘Hello’;
}

document.write(foo()); // Hello

var bar = foo(); // Hello

var bar2 = foo; // reference to function
document.write(bar2()); // Hello
```

- Extremely useful for making multiple references to the same function

Functions

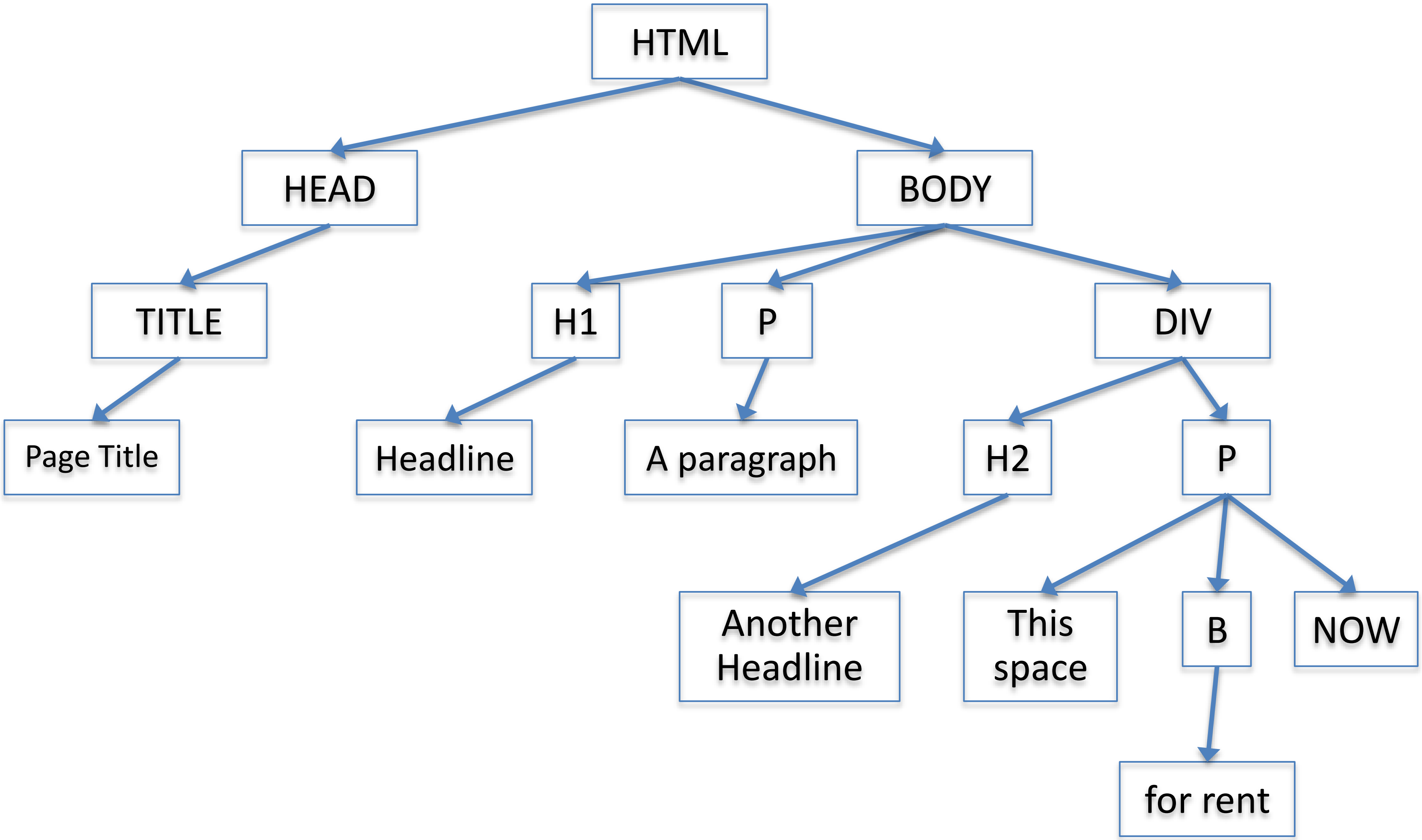
- Makes code reusable and modular
- Can be named or anonymous
- Has own variable scope
- Can return a value to be manipulated
- Have zero or more parameters
- Called a method when inside an object
- Referenced directly by omitting ()

The Document Object Model (DOM)

- The web browser parses the HTML and stores the content in memory as a tree
- EVERYTHING is represented in the tree as nodes (12 kinds of nodes total)
- The order of the nodes is important
- In Javascript, our concern is with element and text nodes

The Document Object Model (DOM)

```
<html>
<head>
    <title>Page Title</title>
</head>
<body>
    <h1>Headline</h1>
    <p>A paragraph</p>
    <div>
     <h2>Another Headline</h2>
     <p>This space <b>for rent</b> NOW</p>
    </div>
</body>
</html>
```



HTML

HEAD

BODY

TITLE

H1

P

DIV

Page Title

Headline

A paragraph

H2

P

Another Headline

This space

B

NOW

for rent

Javascript and the DOM

- Element nodes expose all HTML attributes and inline CSS styles as properties in the element object
- Javascript manipulates the DOM, NOT the markup or stylesheets
- Be mindful of browser-specific properties
- General work pattern:
  - Select part of DOM to manipulate
  - Create new nodes if necessary
  - Set node properties if necessary
  - Attach/remove/move nodes to or in DOM as needed

Common Javascript DOM manipulation methods

- document.getElementById(id)
- node.getElementsByTagName(TAG)
- node.getElementsByClassName(class)
- document.createElement(TAG)
- document.createTextNode(text)
- parentNode.appendChild(newNode)
- parentNode.insertBefore(newNode, refNode)
- node.innerHTML = HTMLString

```
<div id=“target”></div>
```

```
// get reference to target node
var targetElem =
  document.getElementById(‘target’);
// create new P node
var pElem = document.createElement(‘P’);
// add P to DIV
targetElem.appendChild(pElem);
// create text node
var text = document.createTextNode(‘This is a
  new paragraph.’);
// add text to P
pElem.appendChild(text);
```

```
<div id="target"></div>
```

```
// get reference to target node
var targetElem =
  document.getElementById('target');
// overwrite content
targetElem.innerHTML = '<p>This HTML string
  will replace <b>ALL</b> content inside the
  target div.</p>';
```

jQuery

- jQuery is
  - A library that lets you write LESS code
  - Fairly small (94k minified and compressed)
  - Designed to be easily extended
- jQuery is NOT
  - A replacement for Javascript
  - A framework or complete solution
  - Ubiquitous or omnipotent

What jQuery Does Well

- Element selector engine that fully supports CSS selectors
- Traverse and manipulate DOM nodes
- Normalizes event handling
- Basic animation
- Basic utility functions
- Highly leverages chaining

Selecting Elements

- Main interface is the $() function
- Accepts the following:
  - Selector as text (‘#main h2’)
  - DOM node
  - jQuery collection
  - HTML as string (‘<p>Text</p>’)
- Matching elements returned as a jQuery collection object

Common CSS Selectors

element	All elements in DOM
#id	Element with id #id
.class	Elements with class .class
element#id / element.class	Element with id #id / Elements with class .class
element1 element2	All element2s that are descendants of element1
element1 > element2	All element2s that are children of element1
[att=value]	All elements with attribute att equaling value

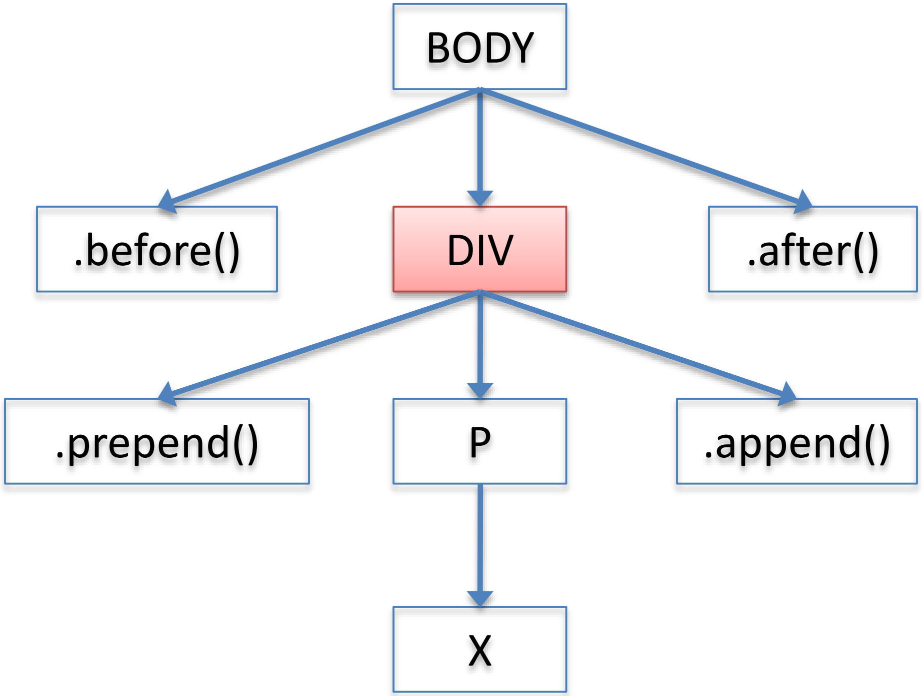
http://www.w3schools.com/cssref/css_selectors.asp
http://api.jquery.com/category/selectors/

Adding Nodes

A	B
.append()	.appendTo()
.prepend()	.prependTo()
.before()	.insertBefore()
.after()	.insertAfter()

```
$(target).A(content);
```

```
$(content).B(target);
```



$(‘div’)…

Demo/Discussion: Basic Manipulation

Other DOM Manipulation

- Removing nodes
  - detach()
  - remove()
- Copying nodes
  - clone()
- Manipulating attributes
  - attr()
  - prop()
  - css()
  - addClass()
  - removeClass()
  - toggleClass()

jQuery Collections

- The $() function returns a jQuery collection object that contains an array of matched elements
- Any changes get applied to EVERY element in the collection (implicit iteration)
- Most of the collection object methods return the modified collection object, allowing chaining
- Retrieving values via getters returns the value from the FIRST element in the collection

Filtering and Changing the Collection

- The collection can be pruned, added to, or changed completely
- Calling a traversal method will apply the traversal to EVERY element in the collection
- When the collection is modified the previous collection will be cached
- Use .end() to release the current collection and revert to the previous collection in the cache

Iterating Through The Collection

- Implicit iteration lessens the need to manually code our loops
- jQuery provides two methods for explicit iteration
  - `$.each()`
  - `.each()`
- Manages what kind of loop to perform automatically

Exercise: Building a Table

Build a table using data in an array.

```
var updates = [
      {
           "symbol": "MMM",
           "desc":"3M Co",
           "price": "117.92",
           "curShares": "50.0000",
           "curValue": "5896.00",
           "amInvested": "5800.00",
           "amtWithdrawn":"0.00"
      },
      {
           "symbol": "ESRX",
           "desc":"Express Scripts Holding Co",
           "price": "65.83",
           "curShares": "100.0000",
           "curValue": "6583.00",
           "amInvested": "3050.45",
           "amtWithdrawn":"0.00"
      },
      {
           "symbol": "PG",
           "desc":"Proctor & Gamble Co",
           "price": "81.40",
           "curShares": "50.0000",
           "curValue": "4070.00",
           "amInvested": "4000.05",
           "amtWithdrawn":"0.00"
      }
];
```

Storing Data in Elements

- Most Javascript objects are mutable – be careful with this!
- Don’t modify objects you don’t own
- Easiest approach is to store values as attributes using `.attr(‘data-var’,’value’)`
- Use `.data()` to associate complex/lots of data with elements
- Explicit iteration is required if each element’s data is unique

Events

- Events are generated based on actions taken by the user or user agent
  - Mouse activity
  - Keyboard activity
  - Window/browser state changes
- Code does not execute in real time, so callback functions are required
- Events are bound using `.on()`

```
<button id=“thebutton”>Click Me</button>

<script type=“text/javascript”>
$(‘#thebutton’)
  .on(
   ‘click’,
   function() {
      alert(‘The button was clicked.’);
   }
  );

</script>
```

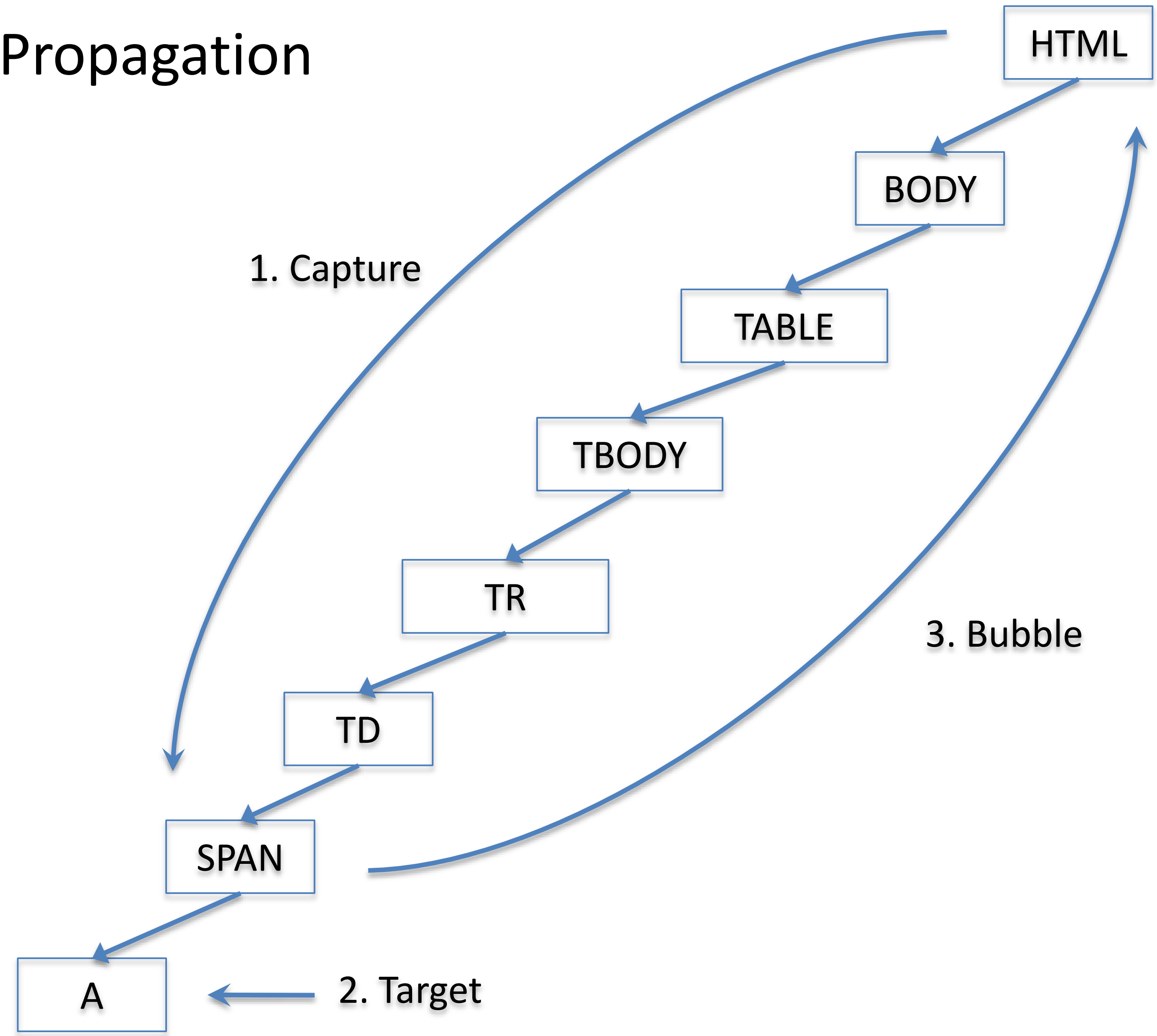
The Event Object

- Information about the event (whodunit, button/key state, mouse position, etc) are stored in the event object
- Most browsers pass the event object to the event handler (IE doesn’t)
- jQuery normalizes browser-specific implementation details into a custom object passed to the handler

Event Propagation

- Events are triggered in three phases
  - Capture phase
  - Target phase
  - Bubble phase
- Few browsers support capture
- Not all events bubble
- Use propagation to optimize event code

Event Propagation



HTML

BODY

TABLE

TBODY

TR

TD

SPAN

A

1. Capture

2. Target

3. Bubble

Deferring Script Execution

- Scripts usually must wait for the DOM to load before being executed
- window.onload is too slow
- Use jQuery’s $(document).ready() handler

Additional Event Handling

- Default actions can be stopped using Event.preventDefault()
- Event propagation can be interrupted using Event.stopPropagation()
- Event listening can be filtered using selectors
- Data can be passed to the event handler and is accessible via the event object

Exercise: Adding Interaction To The Table

Effects

- Effects are accomplished by changing CSS properties in real time or over time via Javascript
- CSS properties may be changed by modifying classes, applying inline styles, or via .animate()
- Timed events can be set using setTimeout() and setInterval(). Likewise they can be removed using clearTimeout() and clearInterval()

CSS Conflict Resolution Rules

Rules are hierarchal and flow from highest to lowest priority:

1. !important outranks all, including inline styles
2. Rank order: inline styles => ids => classes => elements
3. Rule with highest specificity wins
4. Last defined selector wins
5. Inheritance always loses

Inheritance Always Loses

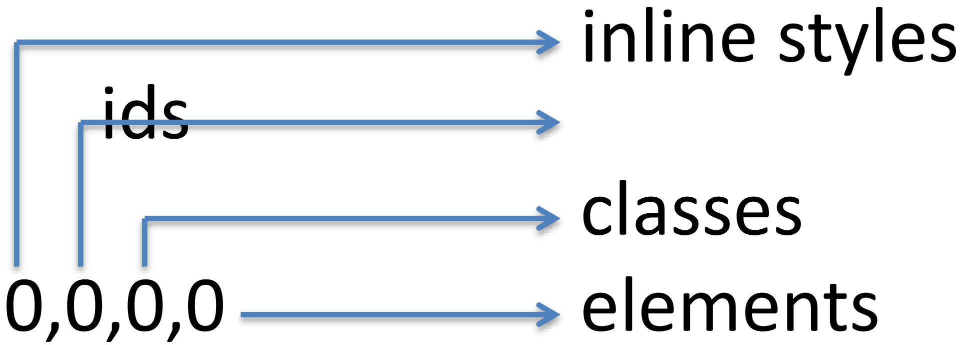
```
body {
    color: #000;
}

p {
    color: #7a7a7a;
}
```

Last Rule Defined Wins

```
p { color: black; }
…
p { color: blue; }
```

Rule With The Highest Specificity Wins

Specificity is expressed as a value 0,0,0,0 where:



inline styles

ids

classes

0,0,0,0

elements

To determine a selector’s specificity count the number of elements, classes, and ids contained in the selector.

```
HTML
    BODY
        DIV#main
            P
        DIV#sidebar
            P
```

```
body p { color: #000; }
p { color: #f00; }
```

Rank Order (a.k.a. “The Poker Rule”)

```
html body div div div table tbody tr td div a span
{ color: red; }
```

```
.highlight
{ color: yellow; }
```

Which rule has the highest rank?

0,0,1,0

0,1,0,0

0,0,0,102

0,1,1,0

0,0,2,15

0,1,0,2

0,0,4,3

!important sucks!

- !important overrides all other directives, including inline styles
- Can only be overridden with more !important directives
- Quickly gets out of hand
- Avoid using unless absolutely necessary

Forms

- jQuery has custom selectors to make selection of form elements simpler
- Form element values are retrieved using .val()
- Forms are made dynamic by using CSS to show/hide/change form content triggered by events

Exercise: Creating a Dynamic Form

Form Validation

- Basic approach is “innocent until proven guilty”
- Assume form data is valid
- Test data against a validation rule
- If the data fails the test, mark the data invalid
- Provide some sort of user feedback

Exercise: Form Validation

Asynchronous Javascript and XML (Ajax)

- Ajax is a means of creating a better user experience by removing the need to do complete page reloads
- Utilizes browser objects available since IE 5.5 and Mozilla 1
- Perceived performance is better
- Primary means of implementation is the XMLHttpRequest object

Making an Ajax Request using jQuery

- $.Ajax()
- Most important options
  - url
  - type
  - dataType
  - success

Side-loading Page Content

- Put your most important content in the main part of the page
- Secondary content can be loaded via Ajax request after main content loads and renders

Processing Data From The Server

- Main data transports in use today are JSON and XML
- JSON is natively understood by Javascript
- XML may be traversed as an XML Document Object just like the DOM

Additional Useful Request Parameters

- error – Specifies an error handler to use in case things go wrong
- cache – Browser caching workaround

Sending Data To The Server

- Data should be serialized into a query string and placed inside the data parameter

Application Concerns When Using Ajax

- Usability
- Navigation
- Security
  - Cross-site scripting
  - JSONP callback requests

Exercise: Adding AJAX

Populate the table with data retrieved via an Ajax request.

Closures

- Functions have access to the context (scope) in which they are created.
- A closure is created when a scope object persists (is not deallocated) when the context completes execution – usually because of an external reference to the scope object.
- Ideal for protecting and/or hiding data, or making data persistent without relying on global variables.

```
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8" />
<title>Demo</title>
</head>
<body>
<button>Click Me!</button>
<button>Click Me!</button>
<button>Click Me!</button>
<button>Click Me!</button>

<script type="text/javascript">
var butList = document.body.getElementsByTagName('BUTTON');

for (var c = 0; c < butList.length; c++) {
   butList[c].onclick = function() { alert(c); // will always return '4' -
     // the value of c AFTER loop completes
    }
}

</script>
</body>
</html>
```

```
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8" />
<title>Demo</title>
</head>
<body>
<button>Click Me!</button>
<button>Click Me!</button>
<button>Click Me!</button>
<button>Click Me!</button>

<script type="text/javascript">
var butList = document.body.getElementsByTagName('BUTTON');
function addClickHandler(button, index) {
    button.onclick = function() {
      alert(index);   // Returns the expected index value because
      // the parent function’s scope object persists
    }
}
for (var c = 0; c < butList.length; c++) {
    addClickHandler(butList[c], c);
}
</script>
</body>
</html>
```

```
<script type="text/javascript">
/*
A common pattern is to have a function return a function,
allowing us to encapsulate and protect data. (variation of factory pattern)
Note the use of an anonymous function in an assignment statement for positioning flexibility.
*/
var butList = document.body.getElementsByTagName('BUTTON');

var makeHandler = function(index) {
    // return the actual handler method
    return function(e) {
     alert(index); // index is persistent and protected
    }
};

for (var c = 0; c < butList.length; c++) {
    butList[c].onclick = makeHandler(c);
}
</script>
</body>
</html>
```

```
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8" />
<title>Demo</title>
</head>
<body>
<button>Click Me!</button>
<button>Click Me!</button>
<button>Click Me!</button>
<button>Click Me!</button>
<script src="/lib/jquery-1.9.1.js" type="text/javascript"></script>
<script type="text/javascript">
$('button').each(
    function(index, element) {
     $(element).click(
           function() {
                alert(index);
           }
     );
    }
);
</script>
</body>
</html>
```

```
<script src="/lib/jquery-1.9.1.js" type="text/javascript"></script>
<script type="text/javascript">
/*
Closures allow persistent data that can be changed while protecting
the data from outside tampering
*/
$('button').each(
    function(index, element) {
     var c = 0; // c remains persistent in each scope object
     // created at function invocation
     // c is private and not accessible elsewhere
     $(element).click(
           function() {
                alert('Button ' + index +
                ' has been clicked ' + (++c) + ' times!');
           }
     );
    }
);

</script>
```

Immediately-Invoked Function Expressions (IIFEs)

- An IIFE is a function declaration written as an expression and then executed at declaration time
- Useful for creating temporary objects or creating protected/hidden namespaces via closures

```
(function() {
   /*
   Items created within this function do not
   affect outer scope and are hidden/protected
   */
})();
```

```
<button id="button1">Click Me</button>

<script type="text/javascript">
// Conventional example using a named function closure
function init() {
    var c = 0;
    document.getElementById('button1').onclick = function() {
     alert('Click count: ' + (++c));
    };
}
init(); // we have to call the function to start the process
</script>

<script type="text/javascript">
// Same thing, only use an IIFE to avoid having to execute the function separately
(function() {
    var c = 0;
    document.getElementById('button1').onclick = function() {
     alert('Click count: ' + (++c));
    };
})(); // our function is run immediately after declaring it
</script>
```

Namespaces

- Used to avoid issues that arise from cluttering the global environment such as name collisions
- Helps to organize and modularize code

```
/*
    Simplest namespace - Single-Object
    Pros - easy to use
    Cons - multiple objects can still clutter global space in large projects,
    objects may still get overwritten without existence check
*/

var Application = {
    /*
    properties and methods go into this object instead of the global namespace
    */
};

// safer implementation in larger projects/multiple developers
var Application = Application || {}; // use existing object or create new one
/*
    extending the Application object separately allows us to extend the existing
    object without overwriting it
*/
Application.method = function() { … };
// object can also be further extended into subobject namespaces
Application.subObject = {};
Application.subObject.newProperty = … ;
```

```
// Alternate namespace pattern using IIFE

// src: http://addyosmani.com/blog/essential-js-namespacing/
// namespace (our namespace name) and undefined are passed here
// to ensure 1. namespace can be modified locally and isn't
// overwritten outside of our function context
// 2. the value of undefined is guaranteed as being truly
// undefined. This is to avoid issues with undefined being
// mutable pre-ES5.
(function ( namespace, undefined ) {
    // private properties
    var foo = "foo",
     bar = "bar";
    // public methods and properties
    namespace.foobar = "foobar";
    namespace.sayHello = function () {
     speak("hello world");
    };
    // private method function speak(msg) {
     console.log("You said: " + msg);
    };
    // check to evaluate whether 'namespace' exists in the
    // global namespace - if not, assign window.namespace an
    // object literal
})(window.namespace = window.namespace || {});
```

Object Oriented Programming in Javascript

- Javascript does not have classes – it uses prototypal inheritance
- Objects are based off of other objects (their ‘prototype’)
- Object constructors are functions
- An instance of an object is created by invoking the constructor using the `new` keyword
- Object literals are derived from the generic `Object` type and have no constructor
- Object properties are public

Object Constructor

- Constructor functions are used to create instances of objects via the new keyword
- Properties declared using this within the constructor are public and copied to the local instance
- Private variables within the constructor are NOT accessible to the object instances

```
// define a Car template object
var Car = function(make, model) {
   // ‘this’ refers to a specific instance of Car
   this.make = make || null;
   this.model = model || null;
   this.start = function() {
    console.log('started.');
   };
};

// use ‘new’ to create instances of the constructor object
// each instance is a separate object
var myCar = new Car('Chevy','Camaro');

var yourCar = new Car('Ford','Mustang');
```

Object Prototype

- An object ‘inherits’ the properties of objects within its prototype chain
- Local properties will override properties of the same name in the prototype chain
- An object’s prototype object is set at creation time and cannot be changed afterwards
- A constructor can have its prototype altered and the changes will affect existing and future instances
- If the prototype object itself is reset, only future instances will use the new prototype object

```javascript
// define a Car template object
var Car = function(make, model) {
   // ‘this’ refers to a specific instance of Car
   this.make = make || null;
   this.model = model || null;
   this.start = function() {
    console.log('started.');
   };
};

// each instance is a separate object
var myCar = new Car('Chevy','Camaro');

// ALL instances, existing and future, will inherit the stop method
Car.prototype.stop = function() {
   console.log('stopped.');
};

var yourCar = new Car('Ford','Mustang');

myCar.stop(); // works
yourCar.stop() // works
```

```
var Car = function(make, model) {
this.make = make || 'default Make';
this.model = model || 'default model’;
}

var SUV = function() {
   this.has4WD = true;
}

var yourCar = new SUV();

/*
Set the prototype of SUV to Car. Any instances of SUV
AFTER setting the prototype will also inherit from Car
*/
SUV.prototype = new Car();

var myCar = new SUV();

console.log(myCar.make); // 'default Make’
console.log(yourCar.make); // undefined

// All instances with Car in its prototype chain will inherit this
Car.prototype.stop = function() {
   console.log('stopped.');
};
```

Constructors and Prototype

- Best practice is to create empty constructor and assign default properties using prototype

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

Car.prototype.make = 'default Make';
Car.prototype.model = 'default Model';
Car.prototype.start = function() {
    console.log('started.');
};

var myCar = new Car();

console.log(myCar.make); // 'default Make'
console.log(myCar.hasOwnProperty('make')); // false

myCar.make = 'Chevy'; // local property overrides prototype property

console.log(myCar.make); // 'Chevy'
console.log(myCar.hasOwnProperty('make')); // false
```

Object Literals and Prototype

- An object literal’s prototype is `Object`.
- New instances cannot be created using `new`
- Use `Object.create()` to create new objects instead of `new`

```
/*
   Object.create is native in ECMA 5+. Most modern browsers support
   it.
*/
// polyfill solution
if (typeof Object.create !== 'function') {
   Object.create = function(parentObj) {
    var F = function() {};
    F.prototype = parentObj;
    return new F();
   };
}

var Car = {
   make: 'default Make',
   model: 'default Model',
   start: function() { console.log('started.'); }
};

var myCar = Object.create(Car); // make new object of type Car
myCar.make = 'Chevy';
var yourCar = Object.create(Car);
```

Best Practices

- ‘Best’ is sometimes subjective
- Balance between performance and maintainability
- Test on per-case basis
- Remember the saying: “Opinions are like…”

Best Practices

- Use a style guide and be consistent

  Google: http://google-styleguide.googlecode.com/svn/trunk/javascriptguide.xml

  jQuery: http://contribute.jquery.org/style-guide/js/

  Crockford: http://javascript.crockford.com/code.html

- Maintain loose coupling
- Avoid globals
- Use dependency injection

Best Practices

- Don’t modify objects you don’t own
  - -Native objects (String, Object, etc)
  - -DOM nodes
  - -BOM nodes (window, etc)
  - -Library objects
- Don’t override methods
- Don’t add new methods
- Don’t remove existing methods

Best Practices

- Pass parameters to functions as objects to avoid confusion dealing with multiple parameters
- Store globals and object properties as local variables when used repeatedly
- Take advantage of event propagation and filtering
- Minimize the amount of work done inside the catch block of a try…catch by calling an error handler from within the catch block

Additional References

- jQuery Documentation
  http://api.jquery.com/
- W3Schools CSS selector reference
  http://www.w3schools.com/cssref/css_selectors.asp
- Javascript: The Good Parts by Douglas Crockford
- Javascript Bible 7th Ed. Appendix A
  http://www.wiley.com/WileyCDA/WileyTitle/productCd-0470526912,descCd-DOWNLOAD.html

Additional References

- Maintainable Javascript, Nicholas C. Zakas, O'Reilly Media
- High Performance Javascript, Nicholas C. Zakas, O'Reilly Media
- Javascript: The Good Parts, Douglas Crockford, O'Reilly Media
- Essential Javascript Namespacing Patterns http://addyosmani.com/blog/essential-js-namespacing/
- Design Patterns, GoF, Addison--Wesley
- XMLHttpRequest W3C Specification http://www.w3.org/TR/XMLHttpRequest/