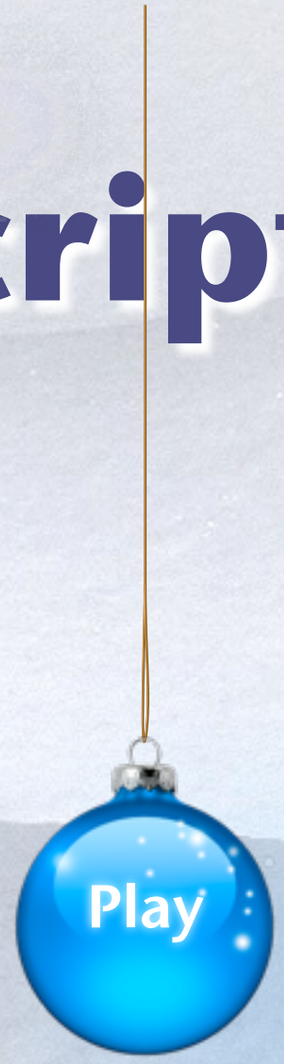
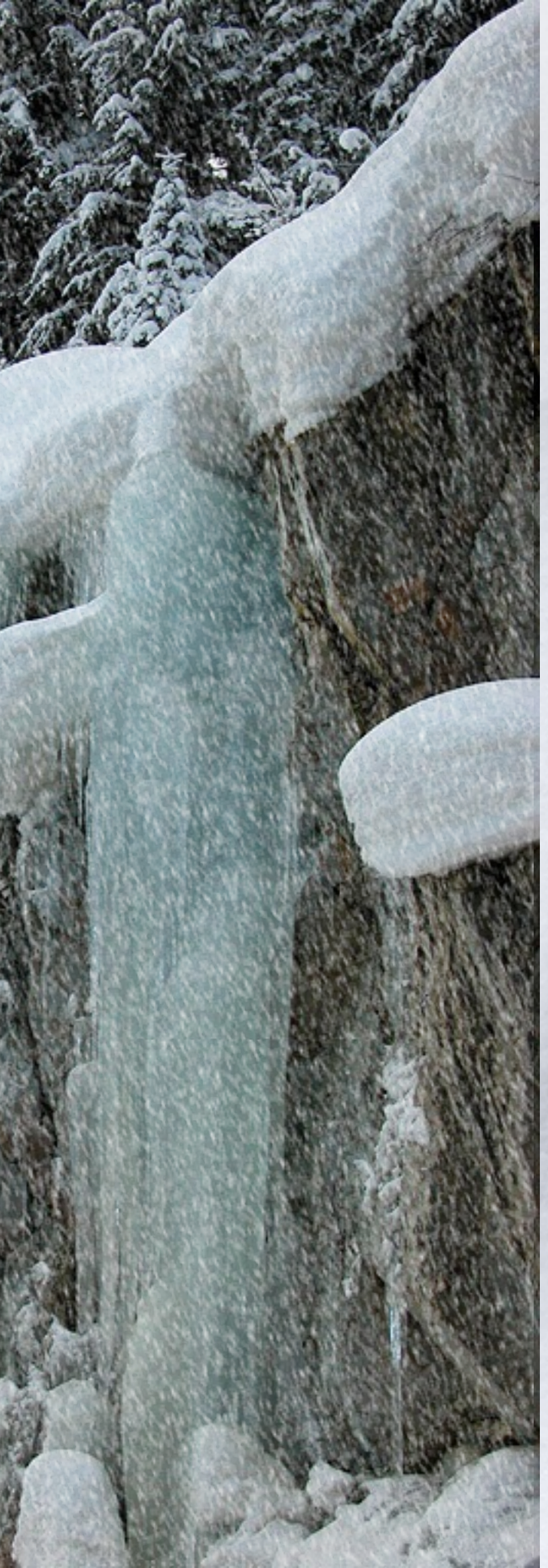
Lecture 6
Advanced JavaScript and DOM





Outline

- Javascript Events
- DOM 2 Event Types
- Event handling models
- Scope and Closure
- Client-side validation
- JavaScript Best Practices

Events

- Events and event handling
  - make web applications more responsive, dynamic and interactive
  - programming by callbacks

- JavaScript events
  - allow scripts to respond to user's interactions with elements on a web page
  - can initiate a modification of the page

Event-driven programming

- In computer programming, event-driven programming is a programming paradigm in which the flow of the program is determined by events such as user actions (mouse clicks, key presses), sensor outputs, or messages from other programs/ threads. Event-driven programming is the dominant paradigm used in graphical user interfaces and other applications (e.g. JavaScript web applications) that are centered on performing certain actions in response to user input.
- In an event-driven application, there is generally a main loop that listens for events, and then triggers a callback function when one of those events is detected. In embedded systems the same may be achieved using hardware interrupts instead of a constantly running main loop. Event-driven programs can be written in any programming language, although the task is easier in languages that provide high-level abstractions, such as closures.

Event Handlers

Event handler

- a callback subroutine that handles inputs received in a program (called a listener in Java and JavaScript)
- function that is called in when an event occurs
- typically associated with an XHTML element
- must be registered
  - i.e., the association must be specified

Event handler registration

- inline:
  - <a href="somewhere.html" onClick="myFunction()">
- traditional:
  - element.onclick = myFunction;
- DOM 2:
  - element.addEventListener(“click", myFunction);
- IE: (evil enough!)
  - element.attachEvent('onclick', myFunction);
- Prototype...and so on: (prefered)
  - Event.observe('target', 'click', myFunction);
  - document.observe('dom:loaded', myFunction);

Syntax

- element.addEventListener(event, function, useCapture);
  - The first parameter is the type of the event (like "click" or "mousedown").
  - The second parameter is the function we want to call when the event occurs.
  - The third parameter is a boolean value specifying whether to use event bubbling or event capturing. This parameter is optional.
- The removeEventListener() method removes event handlers that have been attached with the addEventListener() method

```
element.addEventListener("click", myFunction);
element.addEventListener("click", mySecondFunction);
document.getElementById("myDiv").addEventListener("click", myFunction, true);
element.removeEventListener("mousemove", myFunction);
```

Browser Support

- The numbers in the table specifies the first browser version that fully supports these methods.

Method	Chrome	IE	Firefox	Safari	Opera
addEventListener()	1.0	9.0	1.0	1.0	7.0
removeEventListener()	1.0	9.0	1.0	1.0	7.0

Cross-browser solution

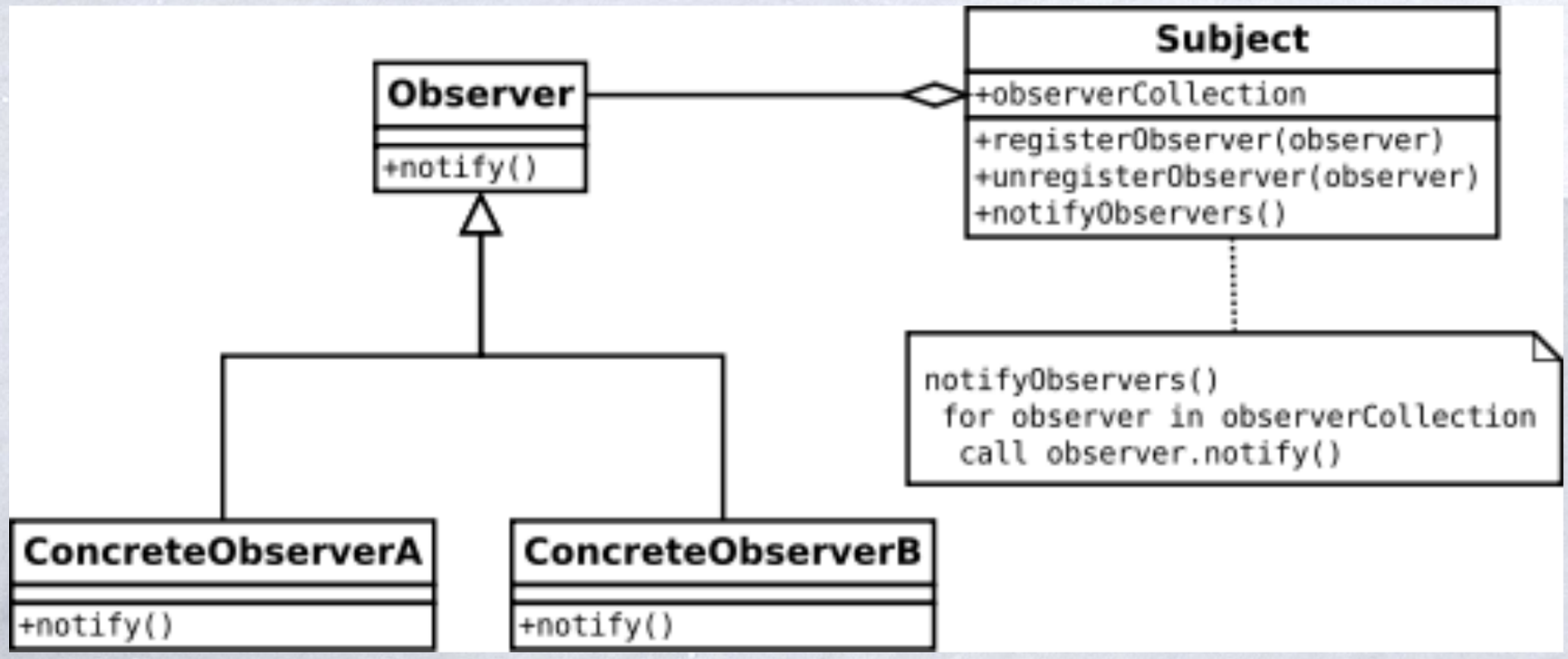
```
var x = document.getElementById("myBtn");
if (x.addEventListener) {
// For all major browsers, except IE 8 and earlier
    x.addEventListener("click", myFunction);
} else if (x.attachEvent) {// For IE 8 and earlier versions
    x.attachEvent("onclick", myFunction);
}
```

Observer pattern

- The observer pattern is a software design pattern in which an object, called the subject, maintains a list of its dependents, called observers, and notifies them automatically of any state changes, usually by calling one of their methods.
- It is mainly used to implement distributed event handling systems.
- The Observer pattern is also a key part in the familiar model–view–controller (MVC) architectural pattern.The observer pattern is implemented in numerous programming libraries and systems, including almost all GUI toolkits.
- a subset of the publish/subscribe pattern

Observer pattern

- Events makes a subject may have multiple observer queues



Attaching event handlers the Prototype way

- to use Prototype's event features, you must attach the handler using the DOM element object's observe method (added by Prototype)
- pass the event of interest and the function to use as the handler
- handlers must be attached this way for Prototype's event features to work
- observe substitutes for addEventListener and attachEvent (IE)

```
element.onevent = function;
element.observe("event", "function");
```
JS

```
// call the playNewGame function when the Play button is clicked
$("play").observe("click", playNewGame);
```
JS

Attaching multiple event handlers with $$

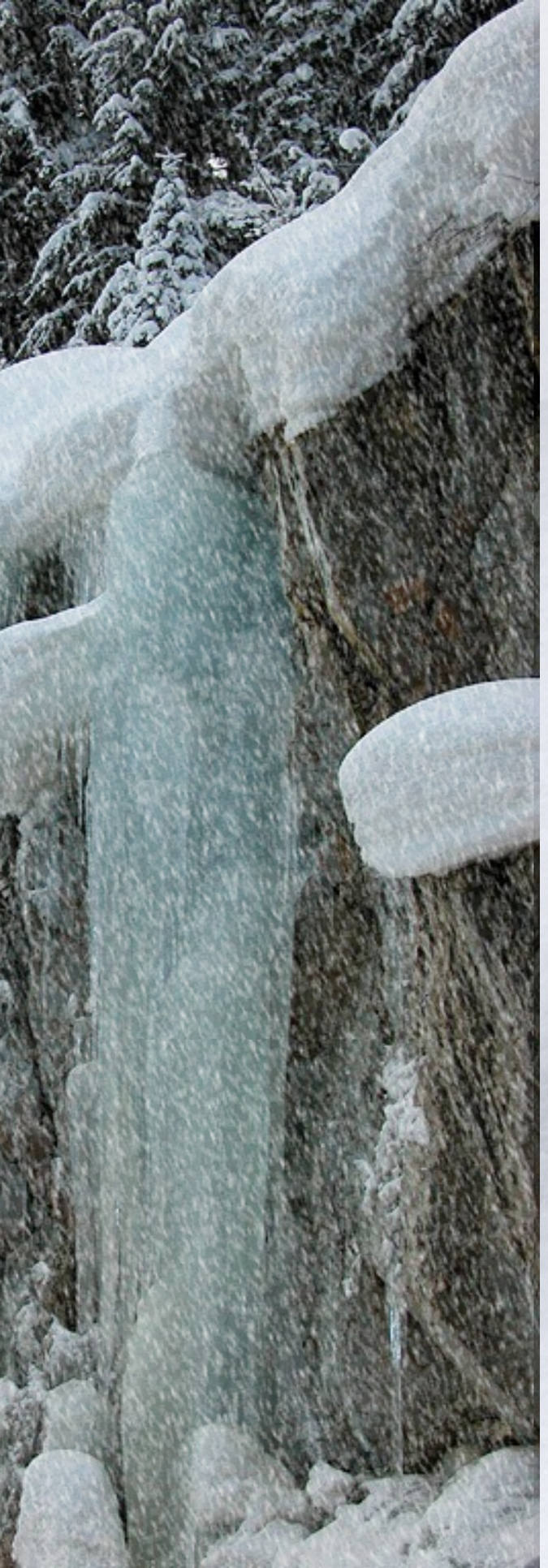
- you can use $$ and other DOM walking methods to unobtrusively attach event handlers to a group of related elements in your window.onload code

```
// listen to clicks on all buttons with class "control" that
// are directly inside the section with ID "game"
window.onload = function() {
  var gameButtons = $$("#game > button.control");
  for (var i = 0; i < gameButtons.length; i++) {
    gameButtons[i].observe("click", gameButtonClick);
  }
};

function gameButtonClick() { ... }
```
JS

What can JavaScript Do?

- Event handlers can be used to handle, and verify, user input, user actions, and browser actions:
  - Things that should be done every time a page loads
  - Things that should be done when the page is closed
  - Action that should be performed when a user clicks a button
  - Content that should be verified when a user inputs data
  - And more ...
- Many different methods can be used to let JavaScript work with events:
  - HTML event attributes can execute JavaScript code directly
  - HTML event attributes can call JavaScript functions
  - You can assign your own event handler functions to HTML elements
  - You can prevent events from being sent or being handled
  - And more ...



Outline

- Javascript Events
- DOM 2 Event Types
- Event handling models
- Scope and Closure
- Client-side validation
- JavaScript Best Practices

DOM 2 Event Types

- UI event types:
  - DOMFocusIn, DOMFocusOut, DOMActivate
- Mouse event types:
  - click, mousedown, mouseup, mouseover, mousemove, mouseout
- Key event types: (not in DOM 2, but will in DOM 3)
- Mutation events:
  - DOMSubtreeModified, DOMNodeInserted, ...
- HTML event types:
  - load, unload, abort, error, select, change, submit, reset, focus, blur, resize, scroll

Useful event types

- problem: events are tricky and have incompatibilities across browsers reasons:
  - fuzzy W3C event specs;
  - IE disobeying web standards;
  - etc.
- solution: Prototype includes many event-related features and fixes

abort	blur	change	click	dbclick	error
keydown	keypress	keyup	load	mousedown	mousemove
mouseover	mouseup	reset	resize	select	submit
focus	mouseout	unload			

The Event object

- Event handlers can accept an optional parameter to represent the event that is occurring. Event objects have the following properties / methods:

```
function name(event) {
  // an event handler function ...
}
```
JS

method/property name	description
type	Returns the name of the event, such as "click" or "mousedown"
currentTarget	Returns the element whose event listeners triggered the event
preventDefault()	Cancels the event if it is cancelable, meaning that the default action that belongs to the event will not occur
stopPropagation()	Prevents further propagation of an event during event flow

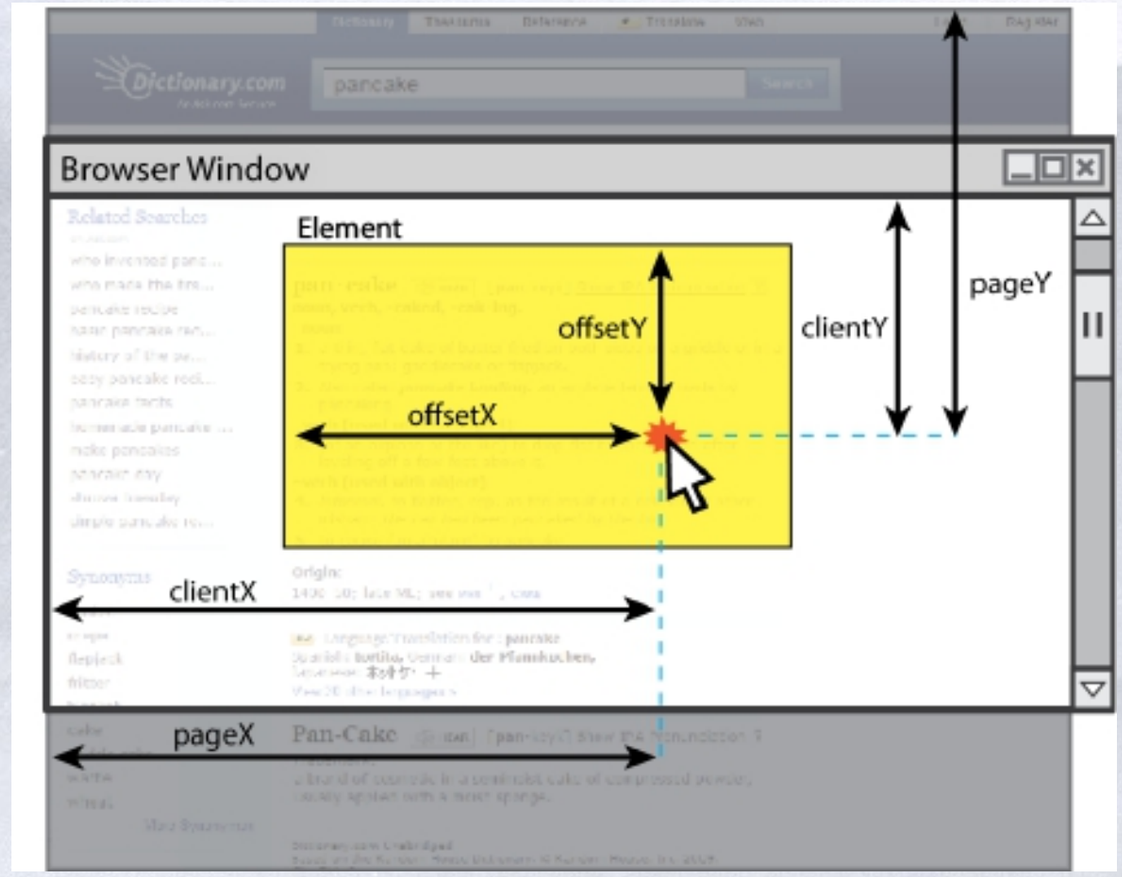
Mouse events

Event	Description	DOM
onclick	The event occurs when the user clicks on an element	2
ondblclick	The event occurs when the user double-clicks on an element	2
onmousedown	The event occurs when the user presses a mouse button over an element	2
onmouseenter	The event occurs when the pointer is moved onto an element	2
onmouseleave	The event occurs when the pointer is moved out of an element	2
onmousemove	The event occurs when the pointer is moving while it is over an element	2
onmouseover	The event occurs when the pointer is moved onto an element, or onto one of its children	2
onmouseout	The event occurs when a user moves the mouse pointer out of an element, or out of one of its children	2
onmouseup	The event occurs when a user releases a mouse button over an element	2

Mouse event objects

- The event parameter passed to a mouse event handler has the following properties:

property/ method	description
clientX, clientY	Returns the horizontal and vertical coordinate of the mouse pointer, relative to the current window, when the mouse event was triggered
screenX, screenY	Returns the horizontal and vertical coordinate of the mouse pointer, relative to the screen, when an event was triggered
offsetX, offsetY	coordinates in element



Frame/Object Events

Event	Description	DOM
onabort	The event occurs when the loading of a resource has been aborted	2
onerror	The event occurs when an error occurs while loading an external file	2
onload	The event occurs when an object has loaded	2
onresize	The event occurs when the document view is resized	2
onscroll	The event occurs when an element's scrollbar is being scrolled	2
onunload	The event occurs once a page has unloaded (for <body>)	2

Form events

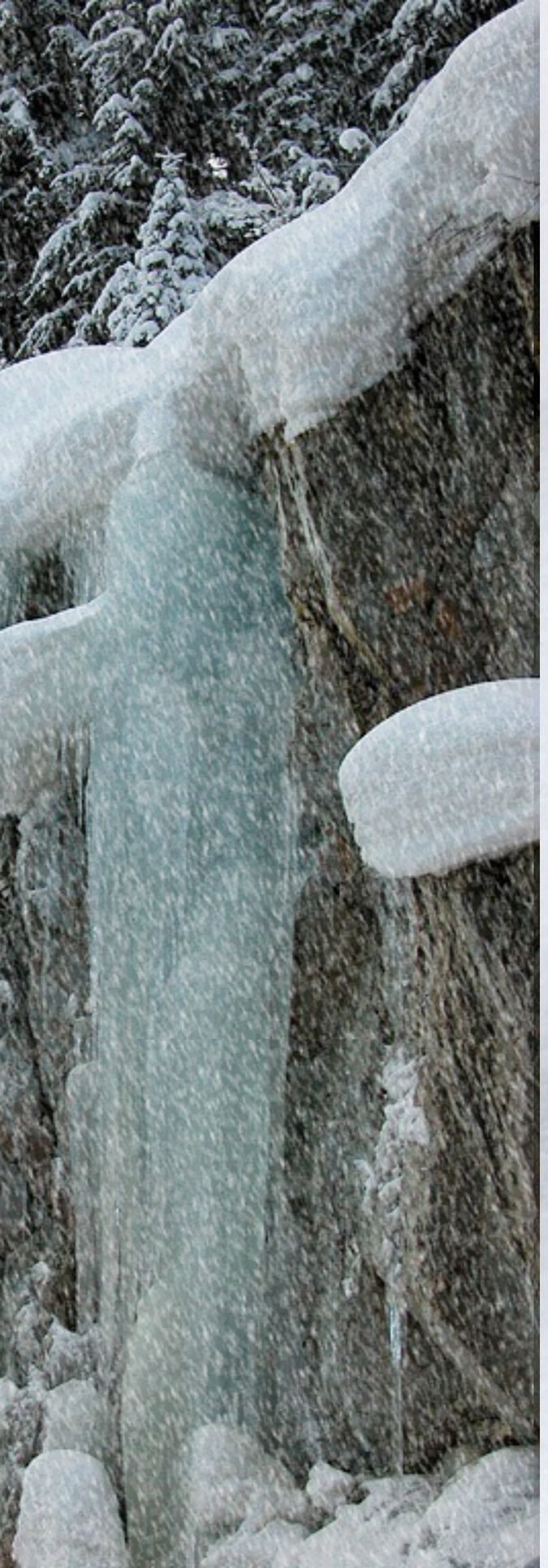
Event	Description	DOM
onblur	The event occurs when an element loses focus	2
onchange	The event occurs when the content of a form element, the selection, or the checked state have changed (for <input>, <keygen>, <select>, and <textarea>)	2
onfocus	The event occurs when an element gets focus	2
onreset	The event occurs when a form is reset	2
onselect	The event occurs after the user selects some text (for <input> and <textarea>)	2
onsubmit	The event occurs when a form is submitted	2

Keyboard Events

Event	Description	DOM
onkeydown	The event occurs when the user is pressing a key	2
onkeypress	The event occurs when the user presses a key	2
onkeyup	The event occurs when the user releases a key	2

KeyboardEvent Object

Property	Description	DOM
altKey	Returns whether the "ALT" key was pressed when the key event was triggered	2
ctrlKey	Returns whether the "CTRL" key was pressed when the key event was triggered	2
charCode	Returns the Unicode character code of the key that triggered the onkeypress event	2
key	Returns the key value of the key represented by the event	3
keyCode	Returns the Unicode character code of the key that triggered the onkeypress event, or the Unicode key code of the key	2
location	Returns the location of a key on the keyboard or device	3
metaKey	Returns whether the "meta" key was pressed when the key event was triggered	2
shiftKey	Returns whether the "SHIFT" key was pressed when the key event was triggered	2
which	Returns the Unicode character code of the key that triggered the onkeypress event, or the Unicode key code of the key	2



Outline

- Javascript Events
- DOM 2 Event Types
- Event handling models
- Scope and Closure
- Client-side validation
- JavaScript Best Practices

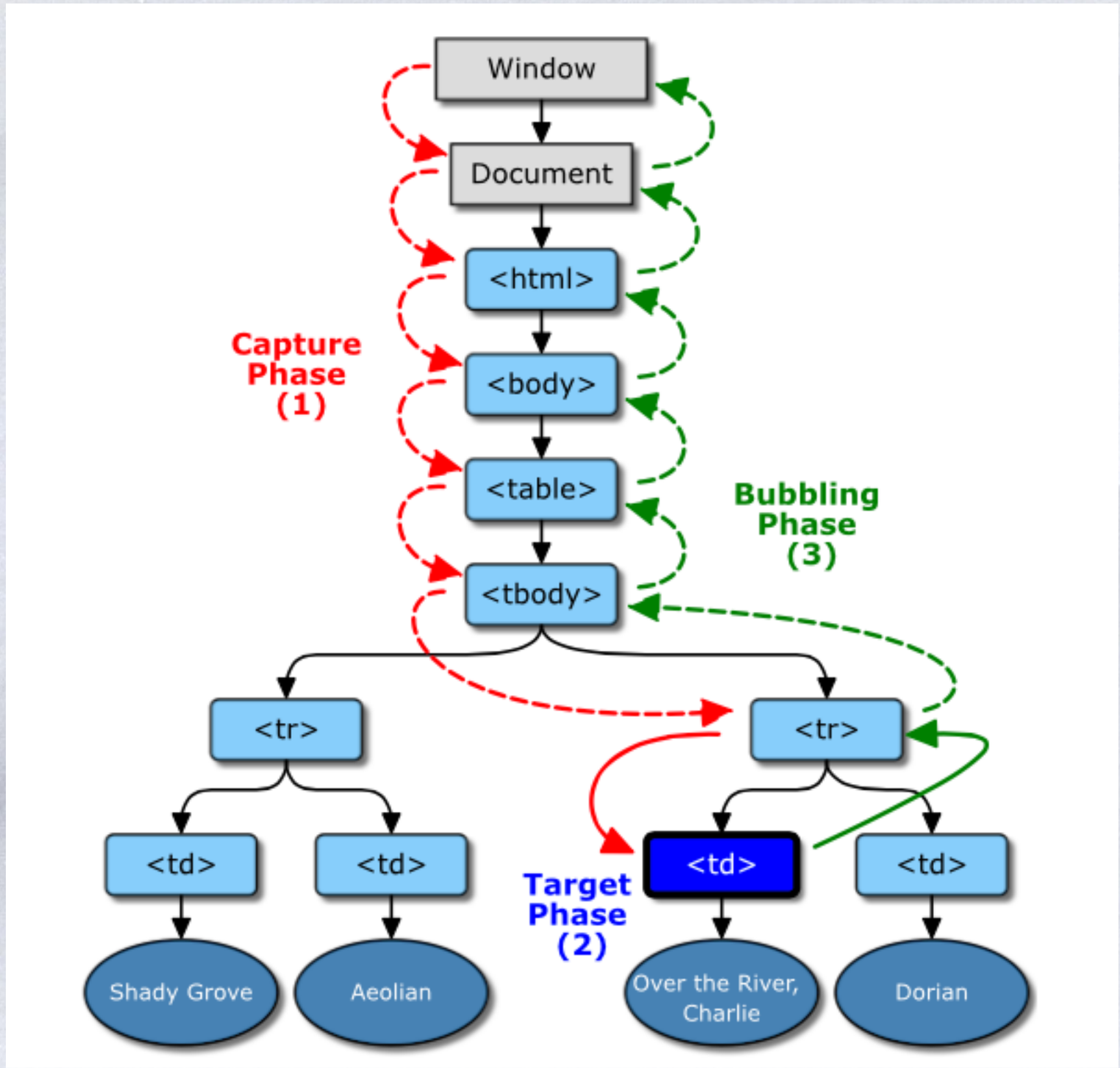
DOM Level 0

- This event handling model was introduced by Netscape Navigator, and remains the most cross-browser model as of 2005.There are two model types: inline model and traditional model.
  - Inline model: event handlers are added as attributes of elements.
  - Traditional model: event handlers can be added/removed by scripts. Like the inline model, each event can only have one event handler registered. The event is added by assigning the handler name to the event property of the element object. To remove an event handler, simply set the property to null:

```
<p>Hey <a href="http://
www.example.com"
onclick="triggerAlert('Joe'); return
false;">Joe</a>!</p>
<script>
        function triggerAlert(name) {
           window.alert("Hey " + name);
        }
</script>
```

```
<script>
        var triggerAlert = function () {
              window.alert("Hey Joe");
        };
        // Assign an event handler
        document.onclick = triggerAlert;
        // Assign another event handler
        window.onload = triggerAlert;
        // Remove the event handler that was just assigned
        window.onload = null;
</script>
```

DOM event flow



Event Phases

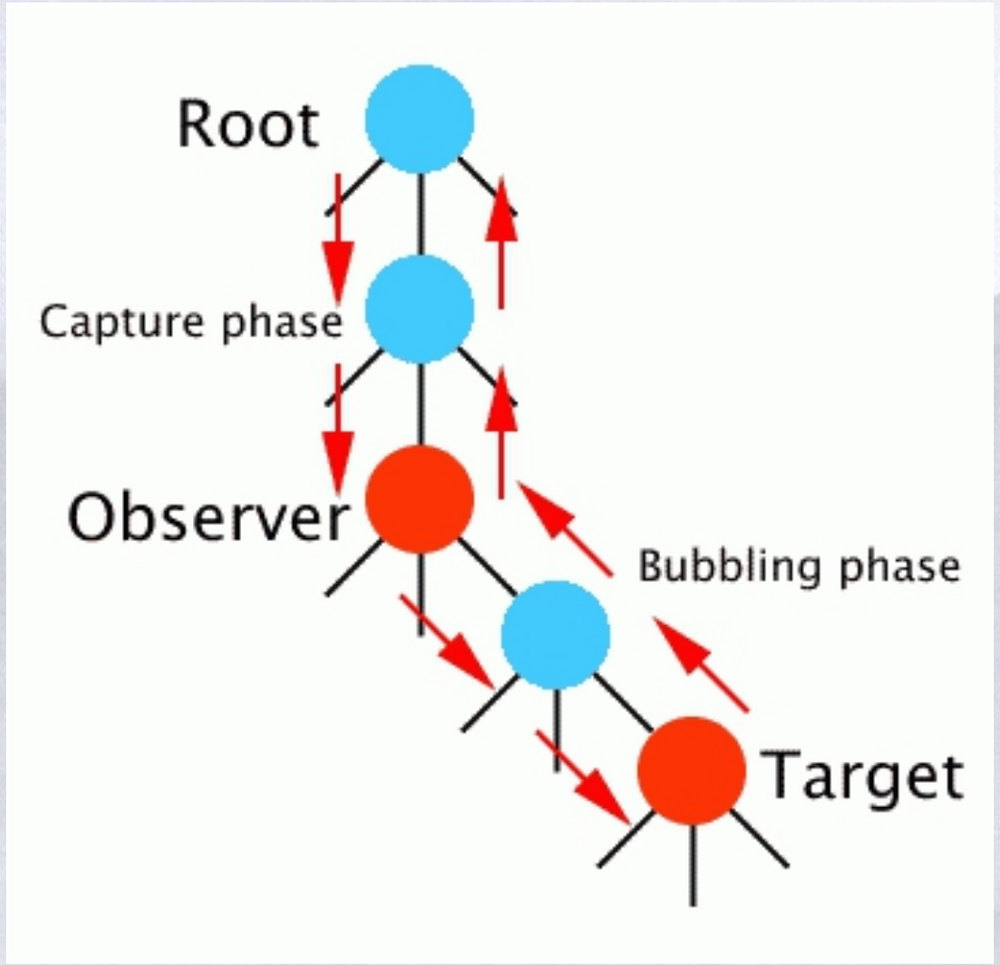
- The **capture** phase: The event object must propagate through the target's ancestors from the Window to the target's parent. This phase is also known as the capturing phase. Event listeners registered for this phase must handle the event before it reaches its target.
- The **target** phase: The event object must arrive at the event object's event target. This phase is also known as the at-target phase. Event listeners registered for this phase must handle the event once it has reached its target. If the event type indicates that the event must not bubble, the event object must halt after completion of this phase.
- The **bubble** phase: The event object propagates through the target's ancestors in reverse order, starting with the target's parent and ending with the Window. This phase is also known as the bubbling phase. Event listeners registered for this phase must handle the event after it has reached its target.

Event flow

- each event has a target, which can be accessed via event

```
element.onclick = handler(e);
function handler(e){
        if(!e) var e = window.event;
        // e refers to the event
        // see  detail of event
        var original = e.eventTarget;
}
```

- each event originates from the browser, and is passed to the DOM



DOM Level 2

Name	Description	Argument type	Argument name
addEventListener	Allows the registration of event listeners on the event target.	DOMString	type
		EventListener	listener
		boolean	useCapture
removeEventListener	Allows the removal of event listeners from the event target.	DOMString	type
		EventListener	listener
		boolean	useCapture
dispatchEvent	Allows sending the event to the subscribed event	Event	evt

Some useful things to know

- To prevent an event from bubbling, developers must call the "stopPropagation()" method of the event object.
- To prevent the default action of the event to be called, developers must call the "preventDefault" method of the event object.
  - canceling default action (e.g. navigating to a new page when clicking on a hyperlink)

A rewrite of the example used in the traditional model

```
<script>
var heyJoe = function () {
   window.alert( "Hey Joe!" );
}

// Add an event handler
document.addEventListener( "click", heyJoe, true );  // capture
phase

      // Add another event handler
      window.addEventListener( "load", heyJoe, false );  // bubbling
phase

      // Remove the event handler just added
       window.removeEventListener( "load", heyJoe, false );
      </script>
```

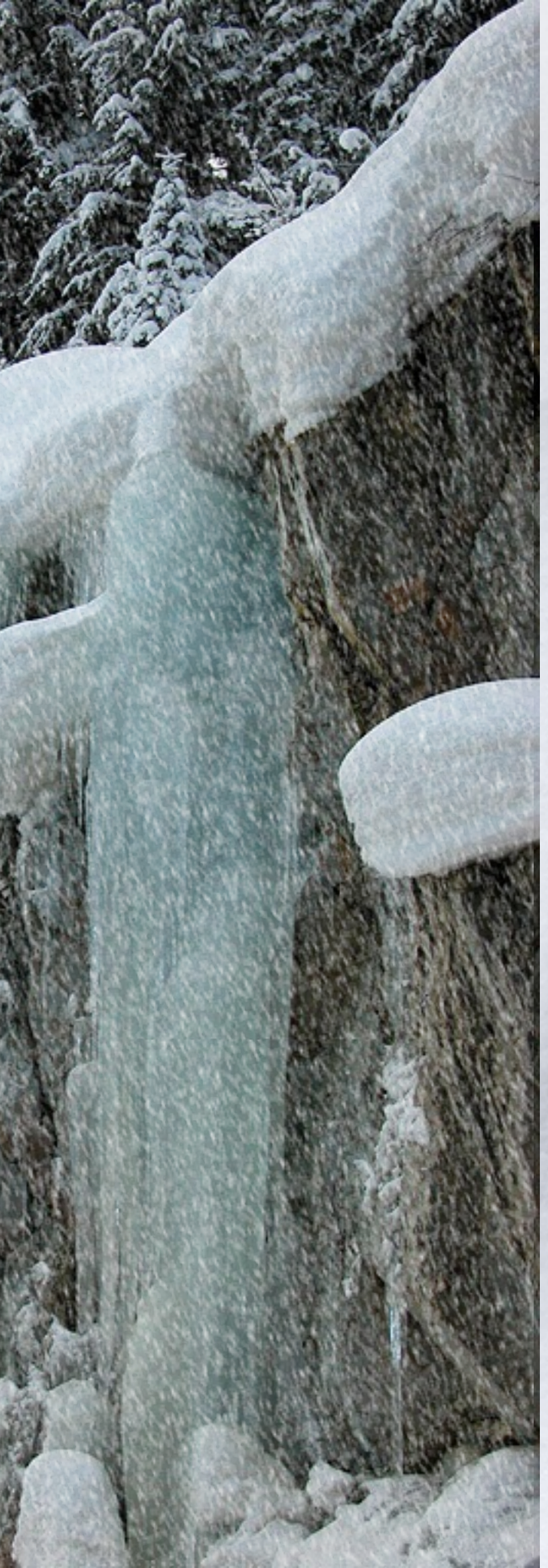
Microsoft-specific model

- Microsoft does not follow the W3C model up until Internet Explorer 8, as its own model was created prior to the ratification of the W3C standard. Internet Explorer 9 follows DOM level 3 events, and Internet Explorer 11 deletes its support for Microsoft-specific model.

Name	Description	Argument type	Argument name
attachEvent	Similar to W3C's addEventListener method.	String	sEvent
		Pointer	fpNotify
detachEvent	Similar to W3C's removeEventListener method.	String	sEvent
		Pointer	fpNotify
fireEvent	Similar to W3C's dispatchEvent method.	String	sEvent
		Event	oEventObject

Some useful things to know……

- To prevent an event bubbling, developers must set the event's cancelBubble property.
- To prevent the default action of the event to be called, developers must set the event's "returnValue" property.
- The this keyword refers to the global window object.



Outline

- Javascript Events
- DOM 2 Event Types
- Event handling models
- Scope and Closure
- Client-side validation
- JavaScript Best Practices

JavaScript Scope

- Scope is the set of variables you have access to.

- JavaScript Scope
  - In JavaScript, objects and functions are also variables.
  - In JavaScript, scope is the set of variables, objects, and functions you have access to.
  - JavaScript has function scope: The scope changes inside functions.

Local JavaScript Variables

- Variables declared within a JavaScript function, become LOCAL to the function.
- Local variables have local scope: They can only be accessed within the function.
- Since local variables are only recognized inside their functions, variables with the same name can be used in different functions.
- Local variables are created when a function starts, and deleted when the function is completed.

Example

```
// code here can not use carName

function myFunction() {
    var carName = "Volvo";

    // code here can use carName

}
```

Global JavaScript Variables

- A variable declared outside a function, becomes GLOBAL.
- A global variable has global scope: All scripts and functions on a web page can access it.

Example

```
var carName = " Volvo";

// code here can use carName

function myFunction() {
    // code here can use carName
}
```

Automatically Global

- If you assign a value to a variable that has not been declared, it will automatically become a GLOBAL variable.
- This code example will declare carName as a global variable, even if it is executed inside a function.

Example

```
// code here can use carName

function myFunction() {
    carName = "Volvo";

    // code here can use carName

}
```

The Lifetime of JavaScript Variables

- The lifetime of a JavaScript variable starts when it is declared.
- Local variables have short lives. They are created when the function is invoked, and deleted when the function is finished.

Function Arguments

- Function arguments (parameters) work as local variables inside functions.

Global Variables in HTML

- Global variables live as long as your application (your window / your web page) lives.
- Global variables are deleted when you close the page.
- In HTML, the global scope is the window object: All global variables belong to the window object.

A Counter Dilemma

- Suppose you want to use a variable for counting something, and you want this counter to be available to all functions.
- You could use a global variable, and a function to increase the counter:

```
Example
var counter = 0;

function add() {
    counter += 1;
}

add();
add();
add();
```

But

- If I declare the counter inside the function, nobody will be able to change it without calling add():

Example

```
function add() {
    var counter = 0;
    counter += 1;
}

add();
add();
add();

// the counter should now be 3, but it does not work !
```

JavaScript Nested Functions

- All functions have access to the global scope.
- In fact, in JavaScript, all functions have access to the scope "above" them.
- JavaScript supports nested functions. Nested functions have access to the scope "above" them.
- In this example, the inner function plus() has access to the counter variable in the parent function:

Example

```
function add() {
    var counter = 0;
    function plus() {counter += 1;}
    plus();
    return counter;
}
```

JavaScript Closures

- The variable add is assigned the return value of a self-invoking function.
- The self-invoking function only runs once. It sets the counter to zero (0), and returns a function expression.
- This way add becomes a function. The "wonderful" part is that it can access the counter in the parent scope.
- This is called a JavaScript closure. It makes it possible for a function to have "private" variables.
- The counter is protected by the scope of the anonymous function, and can only be changed using the add function.

- A closure is a function having access to the parent scope, even after the parent function has closed.

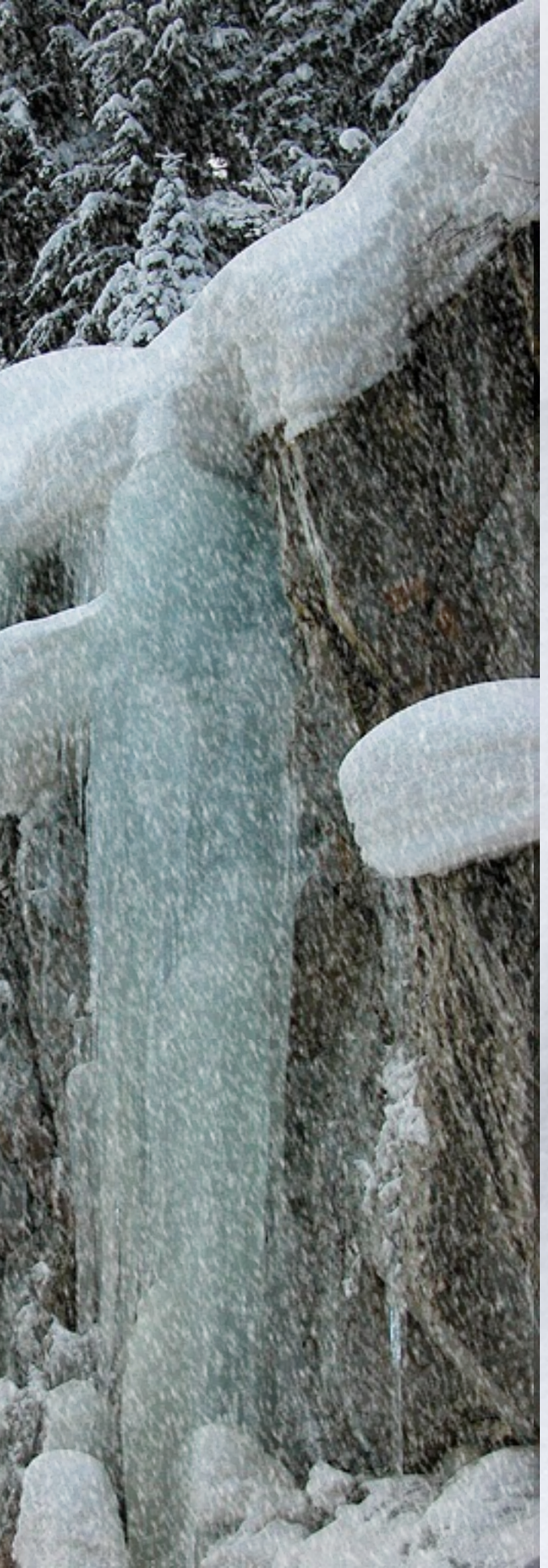
Example

```
var add = (function () {
    var counter = 0;
    return function () {return counter += 1;}
})();
add();
add();
add();
```

闭包应用场景

- 实现私有成员
- 保护命名空间
- 避免污染全局变量
- 变量需要长期驻留在内存

```
function a() {
    var i = 0;
    function b() {
        alert(++i);
    }
    return b;
}
var c = a();
c();
```



Outline

- Javascript Events
- DOM 2 Event Types
- Event handling models
- Scope and Closure
- Client-side validation
- JavaScript Best Practices

Client-side validation

- forms expose onsubmit and onreset events
- to abort a form submission, call Prototype's Event.stop on the event

```
<form id="exampleform" action="http://foo.com/foo.php">
```
HTML

```
window.onload = function() {
  $("exampleform").onsubmit = checkData;
};

function checkData(event) {
  if ($("city").value == "" || $("state").value.length != 2) {
    Event.stop(event);
    alert("Error, invalid city/state.");  // show error message
  }
}
```
JS

Replacing text with regular expressions

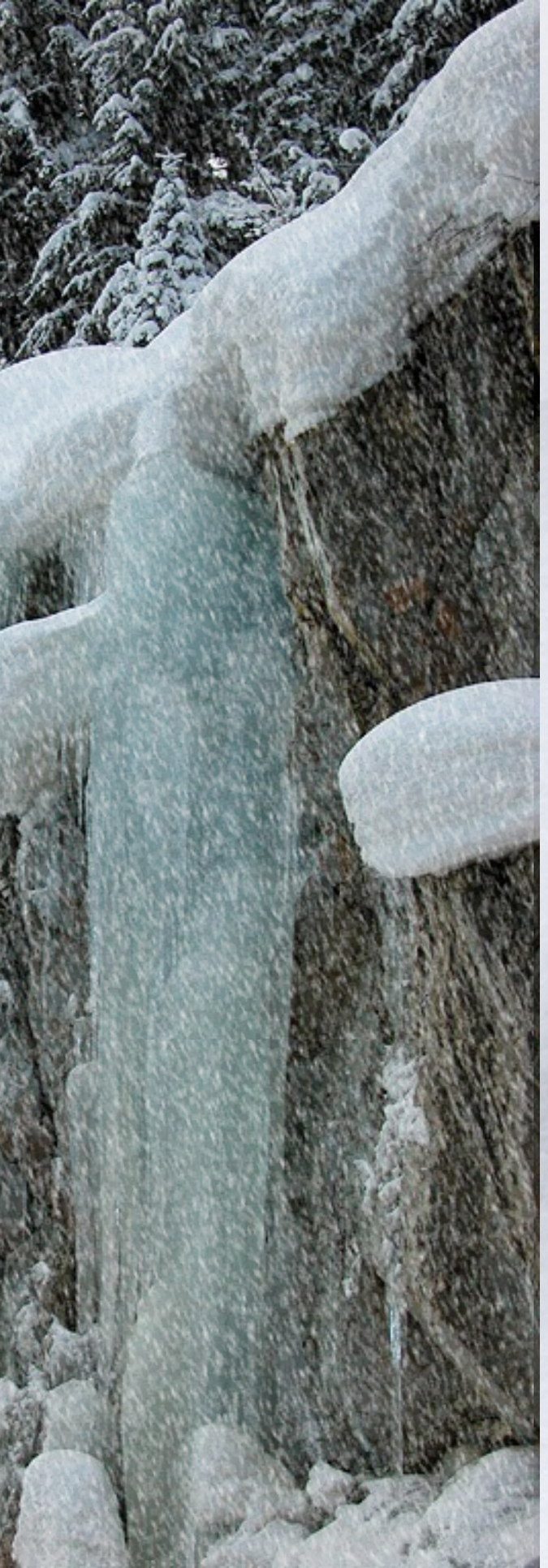
- string.replace(regex, "text")
  - replaces the first occurrence of given pattern with the given text
  - var str = "Qing Zang"; str.replace(/[a-z]/, "x"); //returns “Qxng Zang"
  - returns the modified string as its result; must be stored str = str.replace(/[a-z]/, "x")
- a g can be placed after the regex for a global match (replace all occurrences)
  - str.replace(/[a-z]/g, "x"); //returns “Qxxx Zxxx"
- using a regex as a filter
  - str = str.replace(/[^A-Z]+/g, ""); // turns str into "QZ"

What Is a Regular Expression?

- A regular expression is a sequence of characters that forms a search pattern.
- When you search for data in a text, you can use this search pattern to describe what you are searching for.
- A regular expression can be a single character, or a more complicated pattern.
- Regular expressions can be used to perform all types of text search and text replace operations.
- Syntax
  - /pattern/modifiers;

Regular expression in JavaScript

- string.match(regex)
  - if string fits the pattern, returns the matching text; else returns null
  - can be used as a Boolean truthy/falsey test: var name = $("name").value; if (name.match(/[a-z]+/)) { ... }
- an i can be placed after the regex for a case-insensitive match
  - name.match(/Eric/i) will match “eric", “ERic", ...



Outline

- Javascript Events
- DOM 2 Event Types
- Event handling models
- Scope and Closure
- Client-side validation
- JavaScript Best Practices

Avoid Global Variables

- Minimize the use of global variables.
- This includes all data types, objects, and functions.
- Global variables and functions can be overwritten by other scripts.
- Use local variables instead, and learn how to use closures.

Always Declare Local Variables

- All variables used in a function should be declared as local variables.
- Local variables must be declared with the var keyword, otherwise they will become global variables.
- Strict mode does not allow undeclared variables.
  - “use strict”;

Declarations on Top

- It is a good coding practice to put all declarations at the top of each script or function.
  - Give cleaner code
  - Provide a single place to look for local variables
  - Make it easier to avoid unwanted (implied) global variables
  - Reduce the possibility of unwanted re-declarations

```
// Declare at the beginning
var firstName, lastName, price, discount, fullPrice;
// Use later
firstName = "John";
lastName = "Doe";
price = 19.90;
discount = 0.10;
fullPrice = price * 100 / discount;
This also goes for loop variables:
// Declare at the beginning
var i;
// Use later
for (i = 0; i < 5; i++) {
```

Initialize Variables

- It is a good coding practice to initialize variables when you declare them.
  - Give cleaner code
  - Provide a single place to initialize variables
  - Avoid undefined values

```
// Declare and initiate at the beginning
var firstName = "",
    lastName = "",
    price = 0,
    discount = 0,
    fullPrice = 0,
    myArray = [],
    myObject = {};
```

Never Declare Number, String, or Boolean Objects

- Always treat numbers, strings, or booleans as primitive values. Not as objects.
- Declaring these types as objects, slows down execution speed, and produces nasty side effects:

Example
var x = "John";
var y = new String("John");
(x === y) // is false because x is a string and y is an object.

Or even worse:
Example
var x = new String("John");
var y = new String("John");
(x == y) // is false because you cannot compare objects.

Don't Use new Object()

- Use {} instead of new Object()
- Use "" instead of new String()
- Use 0 instead of new Number()
- Use false instead of new Boolean()
- Use [] instead of new Array()
- Use /()/ instead of new RegExp()
- Use function (){} instead of new function()

Example

```
var x1 = {};           // new object
var x2 = "";           // new primitive string
var x3 = 0;            // new primitive number
var x4 = false;        // new primitive boolean
var x5 = [];           // new array object
var x6 = /()/;         // new regexp object
var x7 = function(){}; // new function object
```

Beware of Automatic Type Conversions

- Beware that numbers can accidentally be converted to strings or NaN (Not a Number).
- JavaScript is loosely typed. A variable can contain different data types, and a variable can change its data type:

Example

```
var x = "Hello";     // typeof x is a string
x = 5;               // changes typeof x to a number
```

When doing mathematical operations, JavaScript can convert numbers to strings, Example:

```
var x = 5 + 7;       // x.valueOf() is 12,  typeof x is a number
var x = 5 + "7";     // x.valueOf() is 57,  typeof x is a string
var x = 5 - 7;       // x.valueOf() is -2,  typeof x is a number
var x = 5 - "7";     // x.valueOf() is -2,  typeof x is a number
var x = 5 - "x";     // x.valueOf() is NaN, typeof x is a number
```

Subtracting a string from a string, does not generate an error but returns NaN (Not a Number), Example

```
"Hello" - "Dolly"    // returns NaN
```

Use === Comparison

- The == comparison operator always converts (to matching types) before comparison.
- The === operator forces comparison of values and type:

```
Example
0 == "";       // true
1 == "1";      // true
1 == true;     // true

0 === "";      // false
1 === "1";     // false
1 === true;    // false
```

Use Parameter Defaults

- If a function is called with a missing argument, the value of the missing argument is set to undefined.
- Undefined values can break your code. It is a good habit to assign default values to arguments.

Example

```
function myFunction(x, y) {
    if (y === undefined) {
        y = 0;
    }
}
```

End Your Switches with Defaults

- Always end your switch statements with a default. Even if you think there is no need for it.

Example

```
switch (new Date().getDay()) {
    case 0:
        day = "Sunday";
        break;
    ......
    case 6:
        day = "Saturday";
        break;
    default:
        day = "Unknown";
}
```

Avoid Using eval()

- The eval() function is used to run text as code. In almost all cases, it should not be necessary to use it.
- Because it allows arbitrary code to be run, it also represents a security problem.

推荐阅读

- JAVASCRIPT设计模式，Ross Harmes & Dustin Diaz. 人民邮电出版社.
- wikipedia: scope closure
- 阮一峰：学习Javascript闭包(Closure)
- 深入理解JavaScript闭包(Closure)
- http://w3techs.com
- http://kangax.github.io/es5-compat-table/es6/
- https://babeljs.io/docs/learn-es2015/
- https://github.com/lukehoban/es6features

Thanks!!!