Client-Side JavaScript

JavaScript in the Browser

Can interact with the windows of the browser (open, close, status line, etc.), with the documents displayed by the browser (add material, delete material, change the presentation), react to events (clicking on a button, hovering over an element, etc.), manage cookies

Security limitations: to prevent scripts from stealing or modifying local files, to prevent script from cross-site attacks & privacy violations

Evolution of Client-Side JavaScript

Browser vendors implemented methods to access the browser & documents displayed by it

These legacy methods are known as DOM 0; essentially standardized, apply only to HTML

Standardization by W3C led to DOM 1 and DOM 2, which are API's to manipulate XML (and HTML) documents, including events

We focus on DOM 1 and 2, not DOM 0

The Document Object Model (DOM)

A tree; the root is a **Document** node

Element nodes (represent `html`, `p`, `head`, etc.)

Text nodes and **Comment** nodes are leaves

Attr nodes represent attributes of elements (but we can manage them through the element API)

DocumentFragement nodes (later)

Enumerating Nodes

```
var root = document.documentElement;
var children = root.childNodes;
var a        = children.item(0);
var b        = children[0];
```

Both a and b refer to the same child (convenience)

```
var x = document.forms.namedItem("f7");
var y = document.forms["f7"];
var z = document.forms.f7;
```

all refer to the same form (convenience again)

Also `firstChild`, `nextSibling`, `parentNode`

Node Types

```
Node.DOCUMENT_NODE                  == 9
Node.ELEMENT_NODE                   == 1
Node.TEXT_NODE                      == 3
Node.COMMENT_NODE                   == 8
Node.ATTRIBUTE_NODE                 == 2
Node.DOCUMENT_FRAGMENT_NODE         == 11

if ( n.nodeType == 3 /* text */ )
  alert( "Text node contains "
         + n.length + " characters");
```

Symbolic constants not always defined

Generic Properties of HTML Elements

```
id
style
lang
dir
className
```

These provide easy access to the common attributes of most HTML elements; can set and read these properties

Managing Attributes in General

```
var l = n.getAttribute("lang");
n.setAttribute("lang","en-US");
n.removeAttribute("dir");
var a = n.getAttributeNode("lang");
```

Attributes are not children of an element; the childNodes array does not include them

Finding Specific Elements

```
var pars =
 document.getElementsByTagName("p");
var par2 = pars[2];

var par821 =
 document.getElementById("para821");
```

Also `getElementByName` (returns an array; less useful)

```
var par821_strongs = // search a subtree
 para821.getElementsByTagName("strong");
```

Modifying the Document

```
n.removeChild(c);
```

```
n.appendChild(c);
```

removes `c` from its current parent if any

```
n.replaceChild(new_child,old_child);
```

The `childNodes` array changes dynamically after every modification; this can cause strange behavior in loops; pay attention

Adding Content: Factories Only

```
var p = document.getElementById("p33");
var ch0 = (p.childNodes)[ 0 ];

var s =
  document.createElement("strong");
var t =
  document.createTextNode(
    "Important: "
  );
s.appendChild(t);

p.insertBefore( s, ch0 );
```

Document Fragments

```
var f =
  document.createDocumentFragement();
... // add children to the fragment

p.appendChild( f );
```

The children of the fragment are moved from the fragment to the document; the fragment becomes empty again

An efficient way to add content; one redraw

A Few Ideas

To change the presentation of an element (color, background, hiding or un-hiding): change it to control the CSS rules that apply to it (by changing the class or id)

To associate program data with elements: add a child that contains that data at text, and hide the child using a CSS rule; when the program visits the element later it will find the hidden text

Event Handling

```
var b = document.getElementById(...);
b.addEventListener(
    "focus",                    // kind of event
    function(e) { ... },        // event handler
    false);                     // no capture
```

Handling of an event occurs in 3 phases:

- capturing: top down from the document down to the triggering element
- at the target
- bubbling: activate handlers up the tree

Stopping Event Propagation

In a handler, call

```
e.stopPropagation();
```

This captures the event on the way down, or stops bubbling on the way up

More on Registration & Deregistration

You can register more than one event listener for a particular event on a particular element

Does not make sense in a single piece of code, but makes sense if different parts of the code need to respond to a single event

```
b.removeEventListener(
  "focus",
  fn_name,
  false);
```

The Event Object

```
type                // string, e.g., "focus"
target              // original target
currentTarget       // current element
eventPhase          // Event.CAPTURING_PHASE
                    // Event.AT_TARGET
                    // Event.BUBBLING_PHASE
timestamp
bubbles             // boolean, not all event bubble
cancelable          // can the default action be
                    // canceled with
                    // preventDefault() ?
```

Events Module: HTML Events

The argument to the listener is an `Event`

abort (img, object)
focus, blur (selectable elements)
select (input, textarea), change (input elements),
reset, submit (form)
scroll (body)
error (body, frameset, img, object)
load, unload (body, frameset, iframe, img, object)
resize (body, frameset, iframe)

Events Module: Mouse Events

The argument to the listener is a `MouseEvent` properties: `screenX`, `screenY`, `clientX`, `clientY`, `altKey`, `ctrlKey`, `shiftKey`, `metaKey`, `button`, `detail`

`click`
`mousedown`, `mouseup`
`mousemove`
`mouseover`, `mouseout` [w/`relatedTarget` property]

`keypress` [w/`charCode` property]

Other Event Modules: UI & Mutation Events

Not commonly used

UIEvent is the parent interface of MouseEvent

Mutation events allow the program to respond to changes in the document (but in most cases, the program itself caused these changes...)

Introduction to Cookies

Key-value pairs that the server and browser exchange

Used by servers to store small amounts of information on the browser

Cookies can survive browser shutdowns etc., so they provide long-term session memory

The Structure of a Cookie

Name
Value (no semicolon, comma, white space; or use `escape()` to write and `unescape()` to read)

expires	optional, GMT String (use `Date.toGMTString()`), default=when the browser exits
domain	optional, generalization of server's name, e.g., `domain=.tau.ac.il` (must be a suffix of the server's name)

The Structure of a Cookie Continues

path	optional, generalization of the path within the domain, e.g., `/~stoledo` default is the same directory as the document that sets the cookie
secure	optional, send only over HTTPS or other secure protocols default=false

Cookies are Few and Small

Browsers are only required to store 300 cookies (total), 20 per web server, 4KB each (name+value)

Writing Cookies

```
document.cookie = "version=2.1beta" +
                  "; path=/~stoledo" +
                  "; domain=.ac.il"  +
                  "; secure"         +
                  "; expires="       +
                  nxtyear.toGMTString();
```

This adds the cookie to the set of cookies associated with the document

To change: set to a new value

To delete: set to some value with an expired date

Reading Cookies

When you read the value of `document.cookie`, you get a string with all the associated cookies,

```
"version=2.1beta;nameA=valueA;..."
```

Split using `String.split` or search for the cookie you need

`document.cookie` is **not** a normal JS property