JavaScript Objects, Properties, And Methods of the DOM

Today's Goals

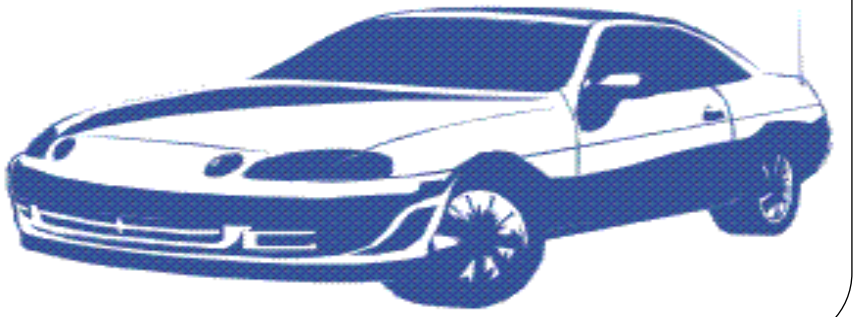
- Objects, Properties, and Methods;
- Document Object Model (DOM);
- Browser Object Model;
- Accessing a Specific Element Value in HTML Form
- The getElementById() method;
- Introduction to form validation

Introduction

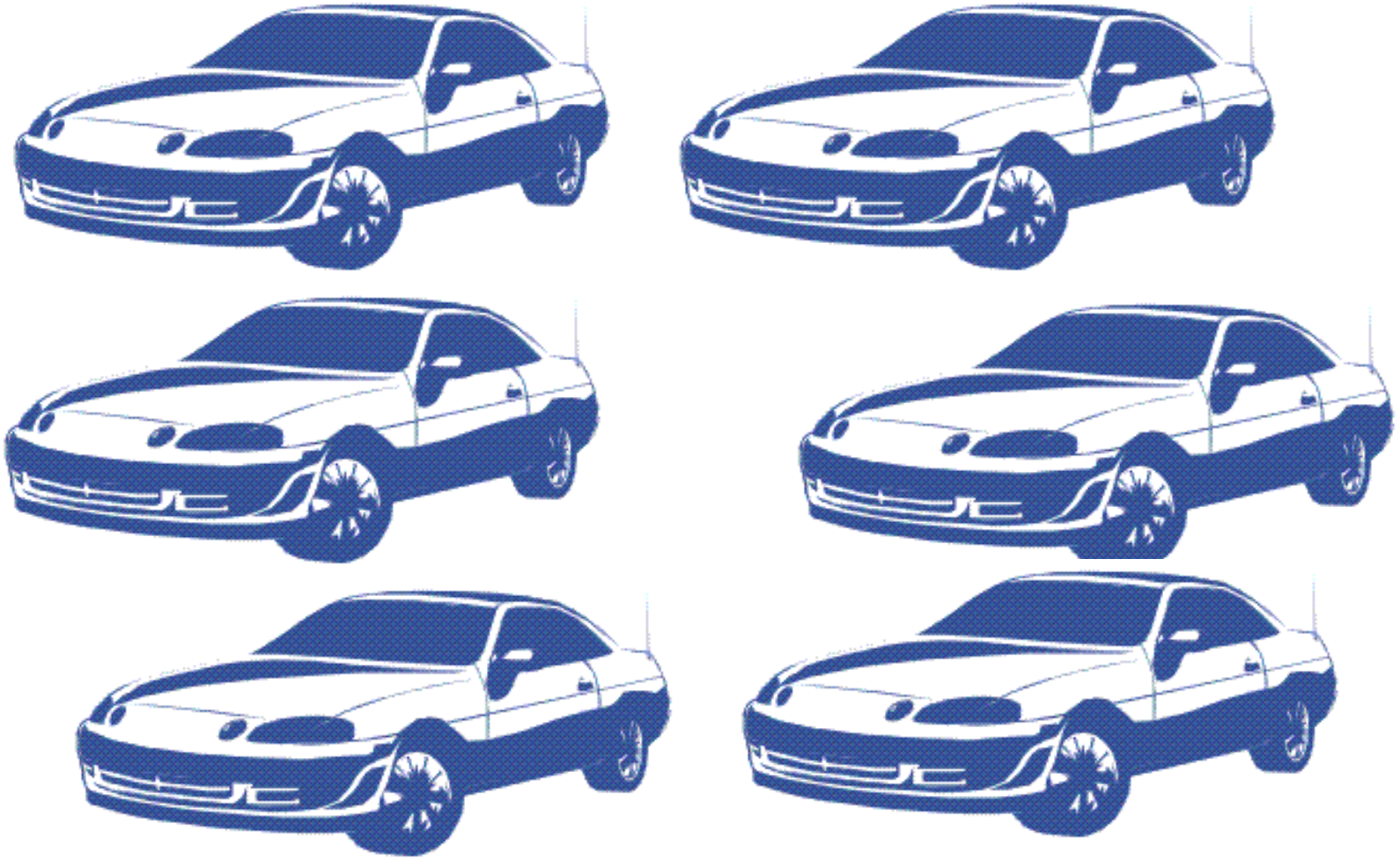
- In addition to objects that are predefined in the browser, you can define your own objects.
- This chapter describes:
  - how to use objects, properties, methods, and
  - how to create your own objects.

Objects

- Objects in JavaScript are a collection of properties, each of which can contain a value.
- Each value stored in the properties can be a value, another object, or even a function.
- You can define your own objects with JavaScript, or use the several built-in objects.

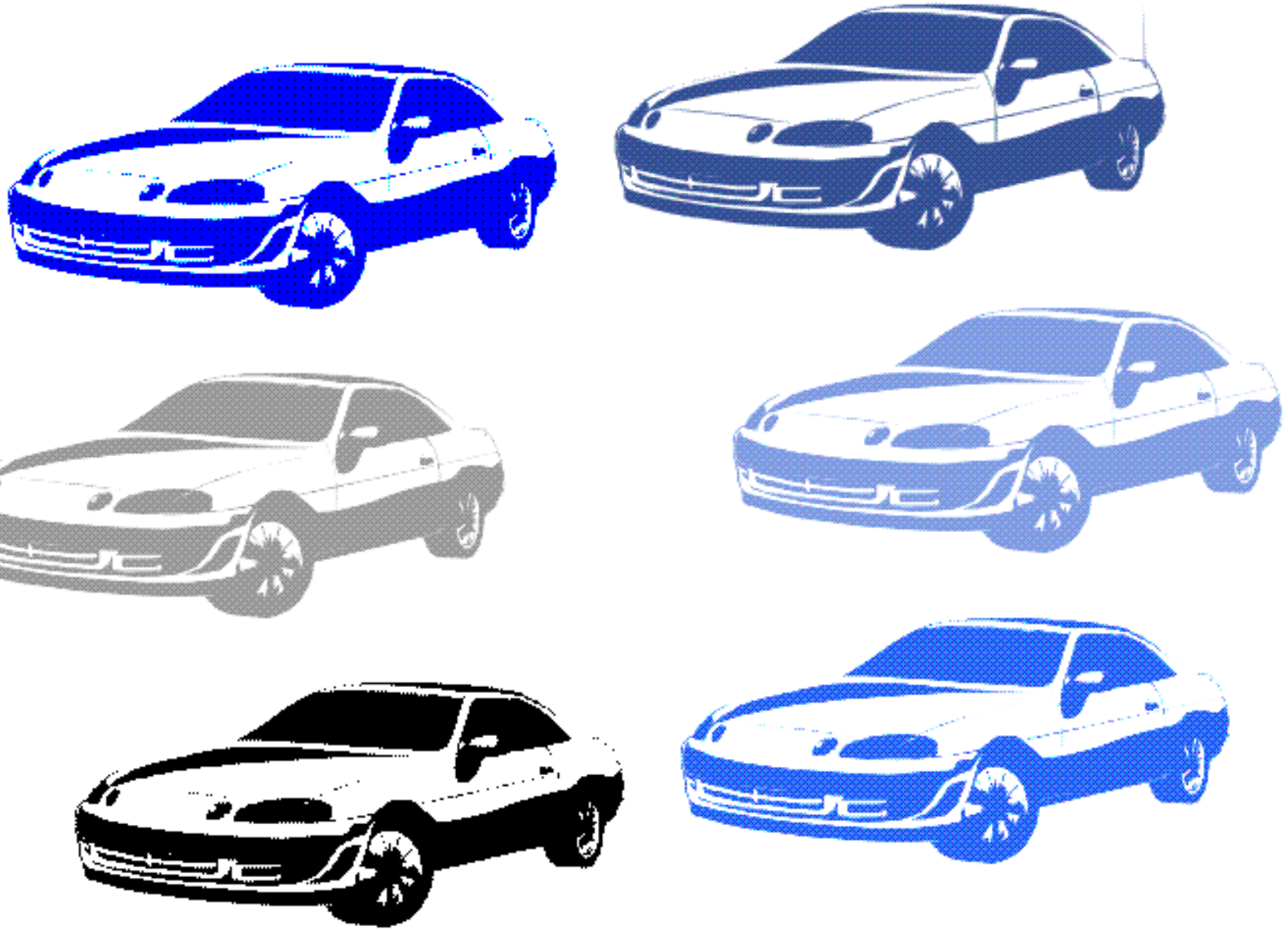


‘Instances’ of an Object



All *instances*
of an object are objects themselves!

‘Property’ Values of the Instances May Differ



Working with objects

- JavaScript is designed on a simple **object-based** paradigm.
- An **object** is a collection of properties.
- A **property** is an association between a name (or key) and a value. Defines the object's characteristics.
- A **property's value** can be a function, in which case the property is known as a **method**.
- A JavaScript object has properties associated with it.
- Object properties are basically the same as variables,
- The properties of an object define the characteristics of the object.

Working with objects

- JavaScript Objects are **Mutable.**
  - They are addressed by reference, not by value.
- If person is an object, the following statement will not create a copy of person:

```
const x = person;  // Will not create a copy of person.
```

- The object x is not a copy of person. It is person. Both x and person are the same object.
- Any changes to x will also change person, because x and person are the same object.

Example:

```
const person = {
  firstName:"John",
  lastName:"Doe",
  age:50, eyeColor:"blue"
}
const x = person;
x.age = 10;     // Will change both x.age and person.age
```

How to Create new objects

1. Create a single object, using an object literal.
2. Using an object initializer.
3. Creating a constructor function and then instantiate an object by invoking that function with the new operator.
   - Create an object with these two steps:
     i. Define the object type by writing a constructor function. There is a strong convention to use a **capital initial letter**.
     ii. Create an instance/single object with keyword **new** Object().
4. Using the **Object.create()** method

These are Explained Next

How to Create new objects – (Cont.)

1. **Using an Object Literal**

- This is the easiest way to create a JavaScript Object.
- Using an object literal, you both define and create an object in one statement.
- An object literal is a list of name:value pairs (like age:50) inside curly braces {}.
- Example

```
const person = {firstName:"John",
lastName:"Doe", age:50,
eyeColor:"blue"};
```

```
const person = {
  firstName: "John",
  lastName: "Doe",
  age: 50,
  eyeColor: "blue"
};
```

This example creates an empty JavaScript object, and then adds 4 properties:

```
const person = { };
person.firstName = "John";
person.lastName = "Doe";
person.age = 50;
person.eyeColor = "blue";
```

How to Create new objects – (Cont.)

2. **Using the JavaScript Keyword, new**

- The following example create a new JavaScript object using new Object(), and then adds 4 properties:

Example

```
const person = new Object();
person.firstName = "John";
person.lastName = "Doe";
person.age = 50;
person.eyeColor = "blue";
```

How to Create new objects – (Cont.)

3. **Using an object initializer**

- The **syntax**:

```
const obj = {
  property1: value1, // property name may be an identifier
  2: value2, // or a number
  "property n": value3, // or a string
};
```

- **Example:**

```
const myHonda = {
  color: "red",
  wheels: 4,
  engine: { cylinders: 4, size: 2.2 },
};
```

The following statement creates an object and assigns it to the variable x if and only if the expression cond is true:

```
let x;
if (cond) {
  x = { greeting: "hi there" };
}
```

How to Create New Objects – (Cont.)

4. **Creating a constructor function** and then instantiate an object by invoking that function with the new operator.

The **syntax**:

- Create an object with these two steps:
  - Define the object type by writing a constructor function. use a capital initial letter.
  - Create an instance of the object with new.

Example:

```
function Car(make, model, year) {
  this.make = make;
  this.model = model;
  this.year = year; }
//Now you can create an object called myCar as follows:
const myCar = new Car("Eagle", "Talon TSi", 1993);
//You can create any number of Car objects by calls to new
const kenscar = new Car("Nissan", "300ZX", 1992);
const vpgscar = new Car("Mazda", "Miata", 1990);
```

- You can rewrite the definition of Car to include an owner property that takes a Person object, as follows:

```
function Car(make, model, year, owner) {
  this.make = make;
  this.model = model;
  this.year = year;
  this.owner = owner;
}
```

- Instantiating the new objects

```
const car1 = new Car("Eagle", "Talon TSi",
1993, rand);
const car2 = new Car("Nissan", "300ZX",
1992, ken);
```

How to Create New Objects – (Cont.)

- An object can have a property that is itself another object. For example, suppose you define an object called **Person** as follows:

```
function Person(name, age, sex) {
  this.name = name;
  this.age = age;
  this.sex = sex;
}
```

and then instantiate two new Person objects as follows:

```
const rand = new Person("Rand Mark", 33, "M");
const ken = new Person("Ken Jones", 39, "M");
```

How to Create New Objects – (Cont.)

- Using the **class syntax** instead of the **function syntax** to define a constructor function.
- Define class in two ways: a **class expression** or a **class declaration**.

```
// Declaration
class Rectangle {
  constructor(height, width) {
    this.height = height;
    this.width = width;
  }
}
```

```
// Expression2; the class has its own name

const Rectangle = class Rectangle2 {
  constructor(height, width) {
    this.height = height;
    this.width = width;
  }
};
```

```
// Expression1; the class is anonymous but assigned to a variable
const Rectangle = class {
  constructor(height, width) {
    this.height = height;
    this.width = width;
  } };
```

How to Create New Objects – (Cont.)

- A class element can be characterized by **three aspects**:

i. Kind: Getter, setter, method, or field
ii. Location: Static or instance
iii. Visibility: Public or private

- Together, they add up to **16 possible combinations**.

- Read More at: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes

How to Create New Objects – (Cont.)

5. **Using the Object.create() method.**

- Very useful, because it allows you to choose the **prototype** object for the object you want to create, without having to define a constructor function.

```
// Animal properties and method encapsulation
const Animal = {
  type: "Invertebrates", // Default value of properties
  displayType() {
    // Method which will display type of Animal
    console.log(this.type);
  },};

// Create new animal type called animal1
const animal1 = Object.create(Animal);
animal1.displayType(); // Logs: Invertebrates

// Create new animal type called fish
const fish = Object.create(Animal);
fish.type = "Fishes";
fish.displayType(); // Logs: Fishes
```

Accessing Objects Properties

- An object property name can be any JavaScript string or symbol, including an empty string. However, you cannot use dot notation to access a property whose name is not a valid JavaScript identifier.

```
const myObj = {};
const str = "myString";
const rand = Math.random();
const anotherObj = {};

// Create additional properties on myObj
myObj.type = "Dot syntax for a key named type";
myObj["date created"] = "This key has a space";
myObj[str] = "This key is in variable str";
myObj[rand] = "A random number is the key here";
myObj[anotherObj] = "This key is object anotherObj";
myObj[""] = "This key is an empty string";

console.log(myObj);
```

```
const myCar = {
  make: "Ford",
  model: "Mustang",
  year: 1969,
};

//Accessing object properties

// 1. Dot notation
myCar.make = "Ford";
myCar.model = "Mustang";
myCar.year = 1969;

// 2. Bracket notation
myCar["make"] = "Ford";
myCar["model"] = "Mustang";
myCar["year"] = 1969;
```

Access Object Properties with Square Brackets

- Accessing object properties with square bracket notation
- It enables you to access a property created by a special key in JavaScript.

Example:

```
let person = {
  'last name':'kamal',
  age:30,
  friends:[ 'Shola','Ade','Ibraheem' ],
}

// console.log(person.last name); is not valid in JavaScript
```

You cannot access the key name 'last name' using the dot notation method, but you can use the square bracket notation, which is available for any object.

```
console.log(person['last name']);
```

Method Shorthand Syntax

The method is created with a function keyword after a colon

```
let person = {
  name:'jamaldeen',
  age:30,
  hobbies:[
'reading','playing','sleeping'
  ],
  speek:function(){
    return this.hobbies
  }
}
console.log(person.speak());
```

shorthand syntax

```
let person = {
  name:'jamaldeen',
  age:30,
  hobbies:[
  'reading','playing','sleeping'
  ],
  speek(){
    return this.hobbies
  }
}
console.log(person.speak());
```

The above Two Codes will produce the same result

Inheritance

- All objects in JavaScript inherit from at least one other object.
- The object being inherited from is known as the **prototype**,
- The inherited properties can be found in the prototype object of the constructor.

The following code adds a color property to all objects of type Car, and then reads the property's value from an instance car1.

```
Car.prototype.color = "red";
console.log(car1.color); // "red"
```

Defining methods

- A method is a function associated with an object,
- A method is a property of an object that is a function.
- A method is an action that can be performed on objects.
- Methods are defined the way normal functions are defined, except that they have to be assigned as the property of an object.

Syntax:

```
objectName.methodName = functionName;
const myObj = {
  myMethod: function (params) {
    // do something
  },
  // this works too!
  myOtherMethod(params) {
    // do something else
  }, };
```

- You can then call the method as follows:

```
objectName.methodName(params);
```

Example:

```
Car.prototype.displayCar = function () {
  const result = `A Beautiful ${this.year} ${this.make} ${this.model}`;
  console.log(result);
};
```

- **this** refer to the object to which the method belongs.
- you can use **this** within a method to refer to the current object.

Then you can call the **displayCar** method for each of the objects as follows:

```
car1.displayCar();
car2.displayCar();
```

In Class Activity 2

```
const Manager = {
  name: "Karina",
  age: 27,
  job: "Software Engineer",
};
const Intern = {
  name: "Tyrone",
  age: 21,
  job: "Software Engineer Intern",
};

function sayHi() {
  console.log(`Hello, my name is ${this.name}`);
}
```

```
// add sayHi function to both objects
Manager.sayHi = sayHi;
Intern.sayHi = sayHi;

Manager.sayHi(); // Hello, my name is Karina
Intern.sayHi(); // Hello, my name is Tyrone
```

Defining getters and setters

- A **getter** is a function associated with a property that gets the value of a specific property.
- A **setter** is a function associated with a property that sets the value of a specific property.
- Together, they can indirectly represent the value of a property.

- **Getters and setters can be either**:
  - defined within object initializers, or
  - added later to any existing object.

Defining getters and setters

- Within object initializers, **getters** and **setters** are defined like regular methods, but prefixed with the keywords **get** or **set**.
- The getter method must not expect a parameter,
- The setter method expects exactly one parameter (new value to set). Example:

```
const myObj = {
  a: 7,
  get b() {
    return this.a + 1;
  },
  set c(x) {
    this.a = x / 2;
  },
};
console.log(myObj.a); // 7
console.log(myObj.b); // 8, returned from the get b() method
myObj.c = 50; // Calls the set c(x) method
console.log(myObj.a); // 25
```

Defining getters and setters

- Getters and setters can also be added to an object at any time after creation using the **Object.defineProperties()** method.
- This method's first parameter is the object on which you want to define the getter or setter.
- The second parameter is an object whose property names are the getter or setter names, and whose property values are objects for defining the getter or setter functions.
- Example that defines the same getter and setter used in the previous example:

```
const myObj = { a: 0 };
Object.defineProperties(myObj, {
  b: {
    get() {
      return this.a + 1;
    },
  },
  c: {
    set(x) {
      this.a = x / 2;
    },
  },
});
myObj.c = 10; // Runs the setter, which assigns 10 / 2 (5) to the 'a' property
console.log(myObj.b); // Runs the getter, which yields a + 1 or 6
```

Comparing objects

In JavaScript, objects are a reference type. Two distinct objects are never equal, even if they have the same properties. Only comparing the same object reference with itself yields true.

```
// Two variables, two distinct objects with the same properties
const fruit = { name: "apple" };
const fruitbear = { name: "apple" };

fruit == fruitbear; // return false
fruit === fruitbear; // return false
```

```
// Two variables, a single object
const fruit = { name: "apple" };
const fruitbear = fruit; // Assign fruit object reference to fruitbear

// Here fruit and fruitbear are pointing to same object
fruit == fruitbear; // return true
fruit === fruitbear; // return true

fruit.name = "grape";
console.log(fruitbear); // { name: "grape" }; not { name: "apple" }
```

Activity 1

```
<!DOCTYPE html>
<html>
<body>
<h2>JavaScript Object Properties</h2>
<p>Access a property with ["property"]:</p>
<p id="demo"></p>
<script>
const person = {
  firstname: "John",
  lastname: "Doe",
  age: 50,
  eyecolor: "blue"
};
document.getElementById("demo").innerHTML = person["firstname"] + " is " +
person["age"] + " years old.";
</script>
</body> </html>
```

Understanding Objects

- Objects in JavaScript are a collection of properties, each of which can contain a value.
  - Each value stored in the properties can be a value, another object, or even a function.
- You can define your own objects with JavaScript, or use the several built-in objects.

- Strings and Arrays in JavaScript are just special types of objects.

- Objects can also have methods, or functions associated with the object. Example: The **document.write ();**

Understanding Objects

- **Object Properties and Methods**
  - JavaScript supports a number of built-in objects, including those in the DOM.
  - Objects can have properties, or variables associated with the object.
  - Object and property names are separated by periods. E.g.:

    location.href property contains the current document's URL.
  - You can change this property to force the browser to go to another Web page. E.g.:

    location.href="http://www.mcp.com/";

Creating/Defining Objects (continued)

- For example, the following function defines the object instructor:

```
function instructor(name, phone, email)
{
    this.name = name;
    this.phone = phone;
    this.email = email;
}
```

- The keyword ***this*** is used to identify the current instance being referenced by the function.

- To create an instance of instructor, simply initialize a var with the constructor containing the appropriate arguments.

```
var tarnoff = new instructor("David Tarnoff",
  "423.439.6404", "tarnoff@etsu.edu");
```

Creating/Defining Objects (continued)

- Remember that objects can be embedded into hierarchies, i.e., an object can become the property of an object.
  - For example, the instructor object defined above could become a property of a course_section object.

Creating/Defining Objects (continued)

```
function course_section(course_title,
  section_number, assigned_instructor)
{
    this.title = course_title;
    this.section_number = section_number;
    this.instructor = assigned_instructor;
}
```

An instance could then be created:

```
var INF314 = new course_section("Web
  Technologies II", "001", "Dr. Alimatu");
```

Creating/Defining Objects (continued)

- To create a function for an object, we used the keyword "prototype".
- Within the constructor function, insert the code:

```
this.prototype.myfunction = function(args)
{
    // insert myfunction code here
}
```

- Can also define outside constructor function:

```
obj_name.prototype.myfunction =
function(args)
{
    // insert myfunction code here
}
```

In Class Exercise

Study the code below and correct it:

```
const person = [];
person["firstName"] = "John";
person["lastName"] = "Doe";
person["age"] = 46;
person.length;
person[0];
```

Built-in Objects in JavaScript

Image Object

- There are a number of pre-defined JavaScript objects such as the Image object
- Properties of the Image object include:
  - ***border*** – Contains the width of the border in pixels (read only)
  - ***complete*** – Boolean value indicating whether the browser has finished loading the image. (read only)
  - ***height*** – The height of the image in pixels (read only)
  - ***lowsrc*** – Specifies the URL of a low-resolution replacement of the image which is loaded and displayed before the high-resolution image is loaded and displayed
  - ***name*** – This is the name/id property of the image
  - ***src*** – Specifies the URL of the image
  - ***width*** – The width of the image in pixels (read only)

String Object

- The constructor for a new String object takes as its argument the initial string:

```
myString = new String("This is great!");
```

- The property length returns the length of the string. For the example below, mylength would equal 14.

```
mylength = myString.length;
```

String Object Methods

- ***charAt(index)*** – returns the character at the position in the string referred to by index.
- ***charCodeAt(index)*** – returns the Unicode value of the character at the position in the string referred to by index.
- ***fromCharCode(num1,...,numN)*** – creates a string from the sequence of Unicode values passed to it as arguments.
- ***toLowerCase( )*** – converts all of the characters in the string to lower case.
- ***toUpperCase( )*** – converts all of the characters in the string to upper case.
- ***indexOf(character [, start_index])*** – returns the index of the first occurrence of the specified character. If start_index is used, search begins from that point in the string.
- ***lastIndexOf(character [, start_index])*** – returns the index of the first occurrence of the specified character. If start_index is used, search begins from that point in the string.
- ***split(delimiter)*** – splits a string into substrings and returns an array that contains the resulting substrings.

Formatting Methods of String

- There are some methods of the object String that when used in conjunction with an output method will create HTML like formatting.
  - For example, the method sub() will cause the text to be outputted as a subscript:

```
var subscript = "24";
document.write("A" + subscript.sub());
```

- outputs the following to the HTML screen:

A_{24}

Formatting Methods of String (continued)

- ***anchor("name")*** – creates an HTML anchor.
- ***blink()*** – makes the displayed string blink. (Of course you know the warnings about blink in HTML, right?)
- ***fixed()*** – makes the displayed string appear as if contained within <tt>…</tt> tags.
- ***strike()*** – makes the displayed string appear as if contained within <strike>…</strike> tags. (strike through)
- ***sub()*** – makes the displayed string appear as if contained within _… tags. (subscript)
- ***sup()*** – makes the displayed string appear as if contained within […] tags. (superscript)
- ***link("URL")*** – creates an HTML link pointing to URL.

Date Object

- There are a number of constructors that can be used to create a new Date object.
  - new Date()
  - new Date(milliseconds)
  - new Date(dateString)
  - new Date(yr_num, mo_num, day_num [, hr_num, min_num, sec_num, ms_num])
- Sometimes, the arguments to these constructors may be confusing
- milliseconds – an integer that represents the number of milliseconds since 01/01/70 00:00:00.
- dateString – a string that represents the date in a format that is recognized by the Date.parse method.
- yr_num, mo_num, day_num – a set of integers that represent the year, month, and day of the date
- hr_num, min_num, sec_num, ms_num – a set of integers that represent the hours, minutes, seconds, and milliseconds.

Date Object Methods

- ***getDate( )*** – returns an integer (between 1 and 31) representing the day of the month for the specified (local time) date.
- ***getDay( )*** – returns an integer (0 for Sunday thru 6 for Saturday) representing the day of the week.
- ***getFullYear( )*** – returns an integer representing the year of a specified date. The integer returned is a four digit number, 1999, for example, and this method is to be preferred over getYear.
- ***getHours( )*** – returns an integer between 0 and 23 that represents the hour (local time) for the specified date.
- ***getMilliseconds( )*** – returns an integer between 0 and 999 that represents the milliseconds (local time) for the specified date.
- ***getMinutes( )*** – returns an integer between 0 and 59 that represents the minutes (local time) for the specified date.

Date Object Methods (continued)

- ***getMonth( )*** – returns an integer (0 for January thru 11 for December) that represents the month for the specified date.
- ***getSeconds( )*** – returns an integer between 0 and 59 that represents the seconds (local time) for the specified date.
- ***getTime( )*** – returns a numeric value representing the number of milliseconds since midnight 01/01/1970 for the specified date.
- ***getTimezoneOffset( )*** – returns the difference in minutes between local time and Greenwich Mean Time. This value is not a constant, as you might think, because of the practice of using Daylight Saving Time.
- ***getUTCDate( )*** – returns an integer between 1 and 31 that represents the day of the month, according to universal time, for the specified date.
- ***getUTCDay( )*** – returns an integer (0 for Sunday thru 6 for Saturday) that represents the day of the week, according to universal time, for the specified date.
- ***getUTCFullYear( )*** – returns a four-digit absolute number that represents the year, according to universal time, for the supplied date.

Date Object Methods (continued)

- ***getUTCHours( )*** – returns an integer between 0 and 23 that represents the hours, according to universal time, in the supplied date.
- ***getUTCMilliseconds( )*** – returns an integer between 0 and 999 that represents the milliseconds, according to universal time, in the specified date.
- ***getUTCMinutes( )*** – returns an integer between 0 and 59 that represents the minutes, in universal time, for the supplied date.
- ***getUTCMonth( )*** – returns an integer, 0 for January thru 11 for December, according to universal time, for the specified date.
- ***getUTCSeconds( )*** – returns an integer between 0 and 59 that represents the seconds, according to universal time, for the specified date.
- ***parse(dateString)*** – takes a date string and returns the number of milliseconds since January 01 1970 00:00:00.

Date Object Methods (continued)

- ***setDate(dateVal)*** – used to set the day of the month using an integer for the supplied date according to local time. (1 to 31)
- ***setFullYear(yearVal [, monthVal, dayVal])*** – used to set the full year for the supplied date according to local time.
- ***setHours(hoursVal [, minutesVal, secondsVal, msVal])*** – used to set the hours for the supplied date according to local time.
- ***setMilliseconds(millisecondsVal)*** – used to set the milliseconds for the supplied date according to local time. (0 to 999)
- ***setMinutes(minutesVal [, secondsVal, msVal])*** – used to set the minutes for the supplied date according to local time.
- ***setMonth(monthVal [, dayVal])*** – used to set the month for the supplied date according to local time.
- ***setSeconds(secondsVal [, msVal])*** – used to set the seconds for the specified date according to local time.

Date Object Methods (continued)

- ***setTime(timeVal)*** – used to set the time of a **Date** object according to local time. The **timeVal** argument is an integer that represents the number of milliseconds elapsed since 1 January 1970 00:00:00.
- ***setUTC?????( )*** – there are similar functions for setting UTC date
- ***toGMTString( )*** – converts a local date to Greenwich Mean Time.
- ***toLocaleString( )*** – uses the relevant locale's date conventions when converting a date to a string.
- ***toString()*** – returns a string representing a specified object.
- ***toUTCString( )*** – uses the universal time convention when converting a date to a string.
- ***UTC(year, month, day [, hours, minutes, seconds, ms])*** – returns the number of milliseconds from the date in a **Date** object since January 1, 1970 00:00:00 according to universal time. This is a static method of **Date** so the format is always **Date.UTC()** as opposed to **objectName.UTC()**.

Boolean Object

- A number of methods such as isNaN() return true/false values
- Programmers can create their own true/false values using Boolean elements.
- Can create objects explicitly using new along with constructor (constructor takes as argument initial value – default is "false")

```
var b_val = new Boolean("true");
```

- Supports toString() method.

Number Object

- There is an object allowing programmers to create variables to hold numeric constants.
- Primarily used to access Number methods.

```
const_val = new Number(24);
```

- Number properties:
  - MAX_VALUE – property that represents the largest possible JavaScript value (approx. 1.79769e+308)
  - MIN_VALUE – property that represents the smallest possible positive JavaScript value. (5e-324)

Number Object Methods

- ***toExponential(num_digits)*** – returns a string containing the number in exponential form with the number of digits following the decimal point defined by num_digits..
- ***toFixed(num_digits)*** – returns a string containing the number represented in fixed-point notation with the number of digits following the decimal point defined by num_digits.
- ***toString([radix])*** – returns a string representing the Number object. If used, "radix" indicates the base to be used for representation. "radix" can be between 2 and 36.

Document Object Model (DOM)

The set of all events which may occur and the page elements on which they can occur is part of the Document Object Model(DOM) not JavaScript.

Browsers don't necessarily share the same set of events

Object Models (continued)

- As stated earlier, JavaScript is an object-based language.
  - Can you name some objects related to the client viewing a page?
- To support standard implementation across all browsers and applications, a set of object models has been created for use with JavaScript.
  - Browser Object Model (BOM)
  - Document Object Model (DOM)

The Document Object Model

- When a document is loaded in the web browser, a number of objects are created. Most commonly used are window and document
- **Window**
  - **open(), close(), alert(), confirm(), prompt()**
- **Document**
  - Contains arrays which store all the components of your page
  - You can access and call methods on the components using the arrays
  - An object may also be accessed by its name
    - **document.myform.address.value = "123 Main"**
    - **document.myform.reset()**
  - Can also search for element by name or id
    - **document.getElementById("myelementid")**
    - **document.getElementByName("myelementname")**

Object Models

JavaScript provides access to a number of different components on the client's side:

- HTML elements
- Browser information
- JavaScript-Specific Objects

Browser Object Model (BOM)

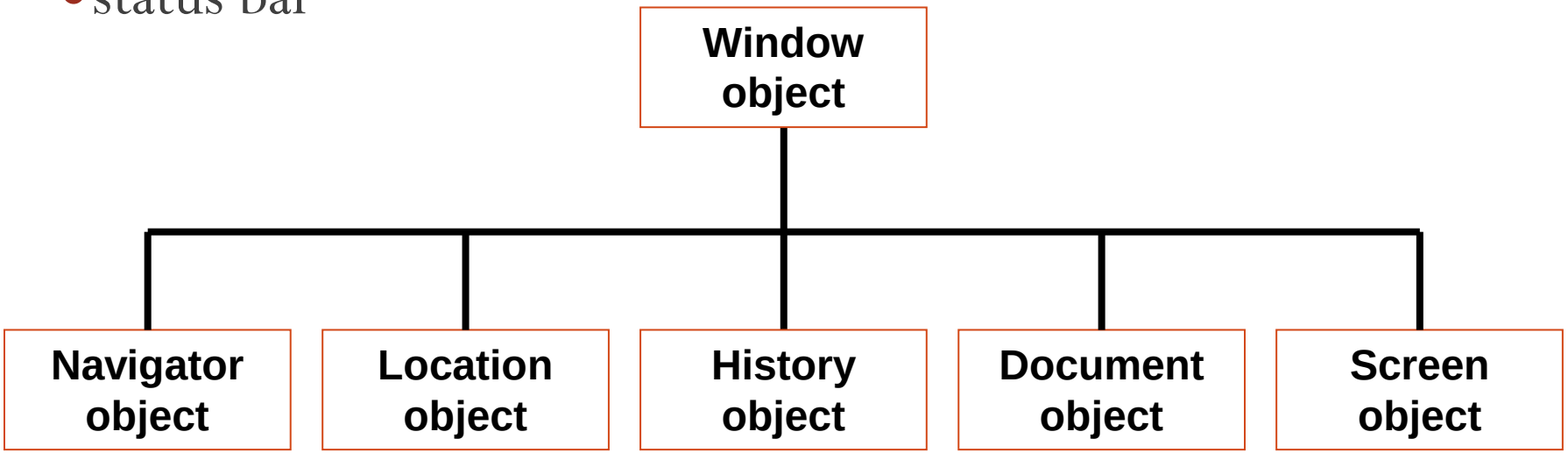
- The BOM defines the components and hierarchy of the collection of objects that define the browser window.
- For the most part, we will only be working with the following components of the BOM.

- window object
- location object
- history object

- document object
- navigator object
- screen object

Window Object

- Top level in the BOM
- Allows access to properties and method of:
  - display window
  - browser itself
  - methods thing such as error message and alert boxes
  - status bar



Document Object

- Top of the Document Object Model (DOM).
  - This is probably the one you'll use most.
- Allows access to elements of the displayed document such as images and form inputs.
- The root that leads to the arrays of the document: forms[ ] array, links[ ] array, and images[ ] array.
- Least compliance to standard here – Netscape 6, Opera 6, and Mozilla 1.0 are the best.

Navigator Object

- Provides access to information and methods regarding the client's browser and operating system.
- Commonly used to determine client's browser capabilities so page can be modified in real time for best viewing.
  - Example: A script may check the browser type in order to modify CSS styles.

History Object

- Provides access to the pages the client has visited during the current browser session.
- Methods such as back() and forward() can be used to move through the history.
- Can also be used to jump to any point in the history.
- As with any browser history, it only allows for a single path.

Other BOM Objects

- **Location Object** – Provides access to and manipulation of the URL of the loaded page.
- **Screen Object** – Provides access to information about the client's display properties such as screen resolution and color depth.
- More information can be found at:

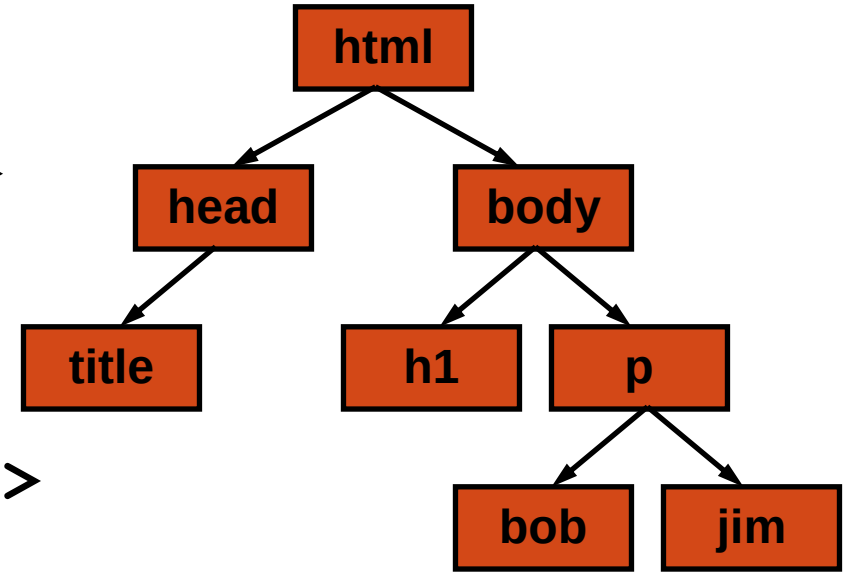
  http://javascript.about.com/library/bltut22.htm

  http://www-128.ibm.com/developerworks/web/library/wa-jsdom/

Document Object Model (DOM)

- Document is modeled as a tree.
- DOM Changes based on page displayed. E.g.:

```
<html>
<head>
  <title>My Page</title>
</head>
<body>
  <h1>My Page</h1>
  <p name="bob" id="bob">
    Here's the first
    paragraph.</p>
  <p name="jim" id="jim">
    Here's the second
paragraph.</p>
</body>
</html>
```



Math Object

- a stand alone object for use with mathematical operations. E.g.
- ***Math.E*** **– returns the base of natural logarithms, i.e., e ≈ 2.7183**
- ***Math.LN10*** **– returns the natural logarithm of 10, i.e., ln(10) ≈ 2.3026**
- ***Math.LN2*** **– returns the natural logarithm of 2, i.e., ln(2) ≈ 0.6931**
- ***Math.LOG10E*** **– returns the base 10 logarithm of e, i.e., $\log_{10}(e) \approx 0.4343$**
- ***Math.LOG2E*** **– returns the base 2 logarithm of e, i.e., $\log_2(e) \approx 1.4427$**
- ***Math.PI*** **– returns the ratio of the circuference of a circle to its diameter, i.e., pi ≈ 3.1416**
- ***Math.SQRT1_2*** **– returns the value of 1 divided by the square root of 2, i.e., $1/(\sqrt{2}) \approx 0.7071$**
- ***Math.SQRT2*** **– returns the square root of 2, i.e., $\sqrt{2} \approx 1.4142$**

Math Object Methods

- **Math.abs(x) – returns the absolute value of x**
- **Math.acos(x) – returns the arccosine of x as a numeric value between 0 and PI radians**
- **Math.asin(x) – returns the arcsine of x as a numeric value between -PI/2 and PI/2 radians**
- **Math.atan(x) – returns the arctangent of x as a numeric value between -PI/2 and PI/2 radians**
- **Math.atan2(y, x) – returns the arctangent of the quotient of its arguments**
- ***Math.ceil(x)* – returns the smallest integer greater than or equal to x**
- ***Math.cos(x)* – returns the cosine of x where x is in radians**
- ***Math.exp(x)* – returns the value of e^x where e is Euler's constant**

Math Object Methods (continued)

- *Math.floor(x)* – returns the largest integer less than or equal to x
- *Math.log(x)* – returns the natural logarithm of x
- *Math.max(x, y)* – returns the greater of x and y
- *Math.min(x, y)* – returns the lesser of x and y
- *Math.pow(x, y)* – returns the value of x^y
- *Math.random()* – returns a pseudo-random number between 0 and 1
- *Math.round(x)* – rounds x to the nearest integer
- *Math.sin(x)* – returns the sine of x where x is in radians
- *Math.sqrt(x)* – returns the square root x
- *Math.tan(x)* – returns the tangent of x where x is in radians

Accessing HTML Elements as Objects

Accessing HTML Elements as Objects

- In order to have access to the object properties and methods inherent to an HTML element, we have to declare an object instance to refer to the HTML element.

- This is done with the ***getElementById()*** method.

```
var html_obj =  document.getElementById("test");
```

- This code will create the object **html_obj** that points to the tag that used the name/id "test".

Accessing Data from Forms

- Using the document object **forms array**.

document.forms[n]

- The most reliable way to reference a form object is to consistently identify everything with the ***name* and *id* attributes.**

document.*formname*

document.forms["*formname*"]

Form Object Properties

- ***action*** – Returns the URL address to which the form's data will be submitted.
- ***length*** – Returns the number of elements in the form.
- ***method*** – Returns a string specifying data submission method, i.e., either 'get' or 'post'.
- ***target*** – Returns the target window where the form's response will appear.
  - Values include: self, parent, etc.
- ***reset( )*** – Resets the form to its default values. (Same result as clicking the reset button.)
- ***submit( )*** – Submits the form's data. (Same result as clicking the submit button.)

Form Element Object Properties

- ***defaultValue*** – sets or returns a string representing the default value of the element.
- ***name*** – sets or returns the element's name or id attribute.
- ***type*** – returns the element's type property.
- ***value*** – sets or returns the element's value attribute. Works differently for different elements.

Form Element Object Methods

- ***blur( )*** – removes the focus from the specified element
- ***click( )*** – simulates a mouse-click for some elements
- ***focus( )*** – returns focus to the specified element

Accessing a Specific Element Value in HTML Form

- To access the element array under the form object.

document.formname.elements[n]

where n equals the position the element holds in the order that the elements were added to the form.

- The most reliable way to reference an element of a form is to consistently identify everything with the *name* and *id* attributes.

document.*formname.elementname*

document.forms["*formname*"].*elementname*

The Use of value

```
<form name="userinput" id="userinput">
<input type="checkbox" name="gen_check" id="gen_check" checked="checked" />
   Checkbox<br /><br />
<input type="radio" name="gen_radiobutton" id="gen_radiobutton" value="1" /> Firs
<input type="radio" name="gen_radiobutton" id="gen_radiobutton" value="2" />
   Second
<input type="radio" name="gen_radiobutton" id="gen_radiobutton" value="3" />
   Third
   <br /><br />
<input type="text" name="gen_text" id="gen_text" value="Type name here"><br
   /><br />
<select name="gen_select" id="gen_select" size="1">
  <option value="one">One</option>
  <option value="2">Two</option>
  <option value="3">Three</option>
  <option value="four">Four</option>
</select><br /><br />
<input type="file" name="gen_file" id="gen_file" size="20" /><br /><br />
<input type="button" onClick = "printVals()" name="gen_button" id="gen_button"
   value="Click here" />
<input type="reset" value="Reset" />
<br /><br /><br />
<textarea cols="40" rows="6" name="output" id="output" /></textarea>
</form>
```

The Use of *value* (continued)

```
var output_string;
function printVals()
{
        output_string="Checkbox value = " +
                        document.userinput.gen_check.checked + "\n"
                + "Radio button value = " +
                        document.userinput.gen_radiobutton.value + "\n"
                + "Text value = " +
                        document.userinput.gen_text.value + "\n"
                + "Select value = " +
                        document.userinput.gen_select.value + "\n"
                + "File selection value = " +
                        document.userinput.gen_file.value + "\n"
                + "Button value = " +
                        document.userinput.gen_button.value + "\n";
        document.userinput.output.value=output_string;
}
```

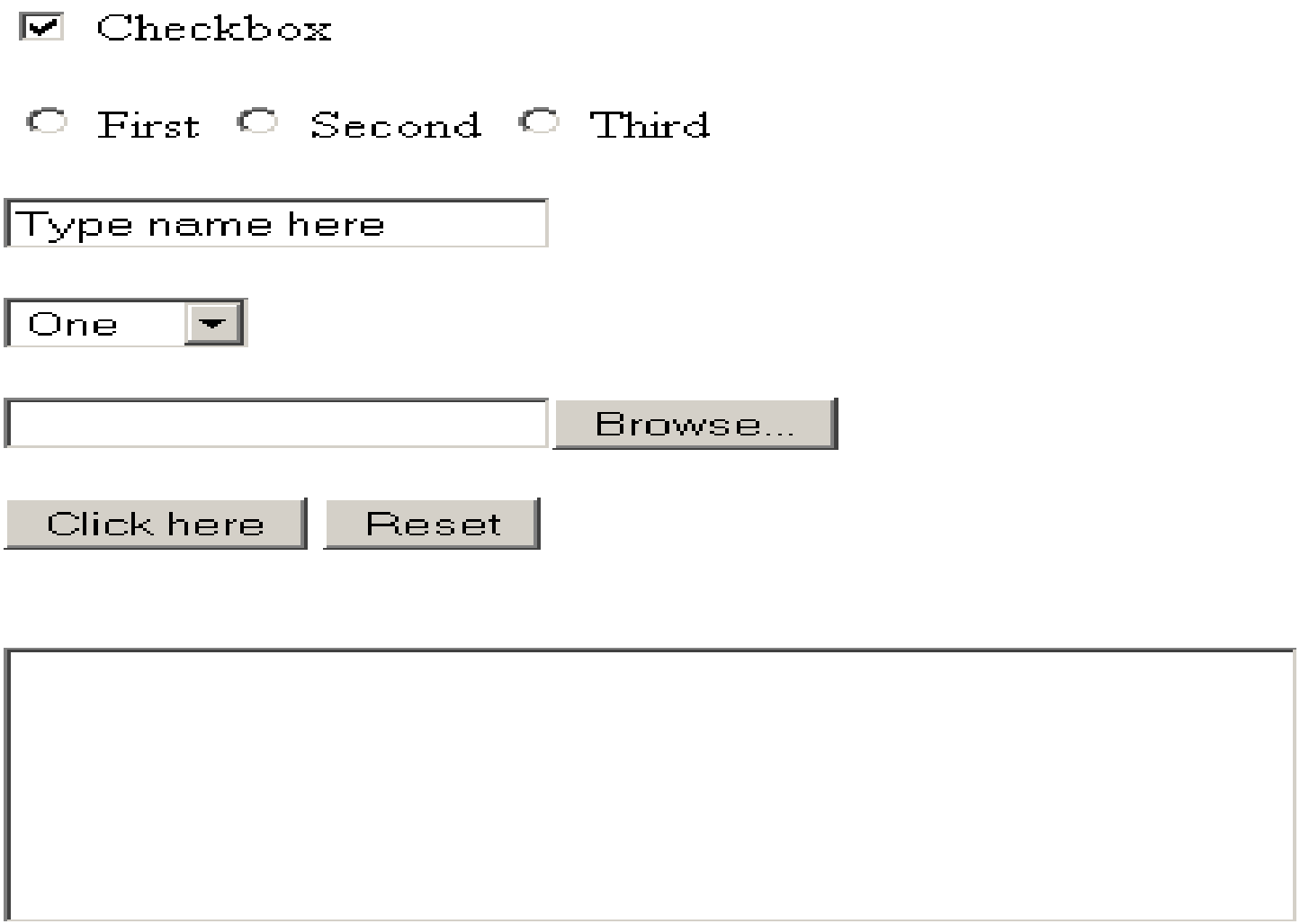
The Use of *value* (continued)

- Some of the values make sense, e.g., the text value equaled the text in the box.
- Some of the values did not make sense
  - Checkbox value always equals "on"
  - Radio buttons always equal "undefined"
  - Button value equals the text on the button

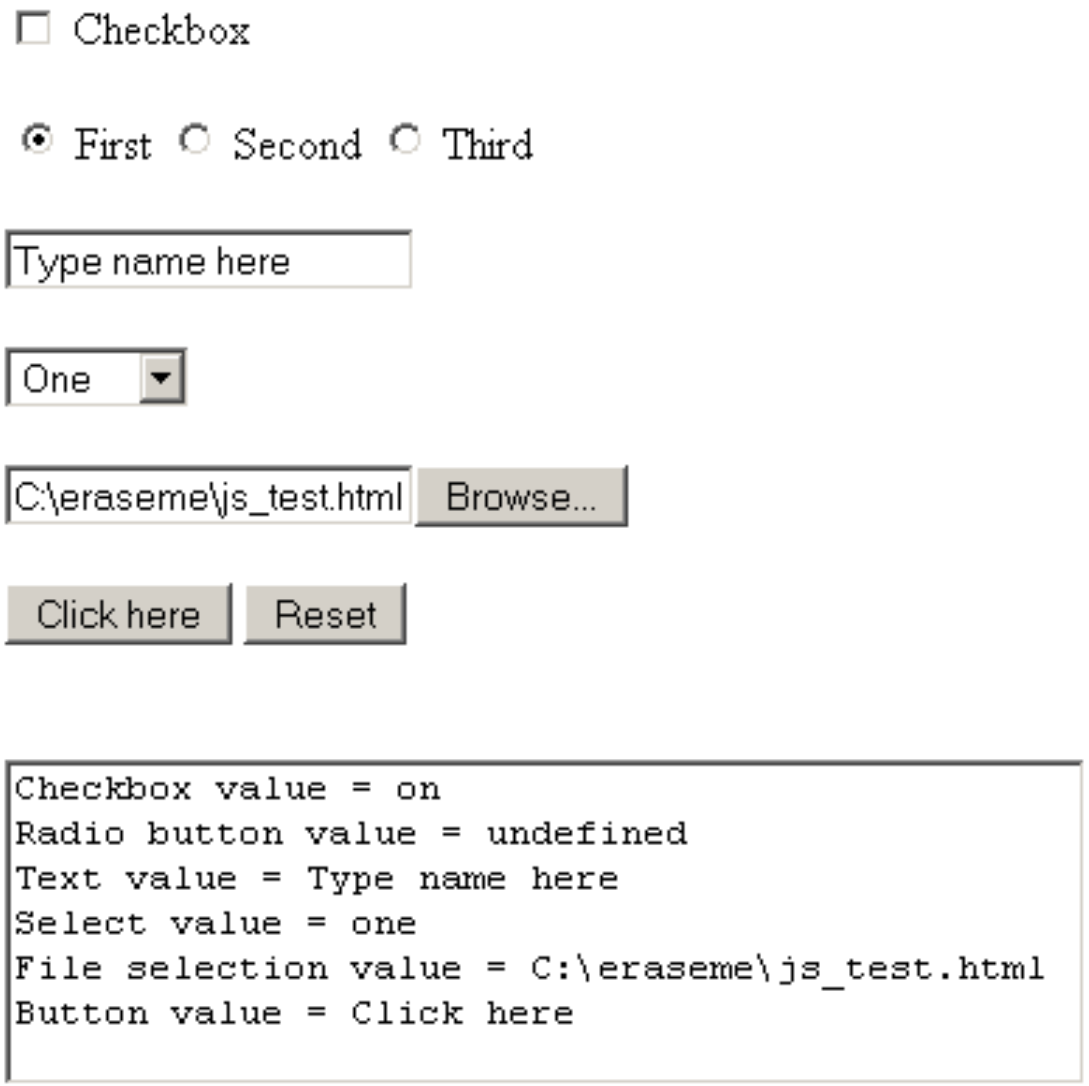
The Use of *value* (continued)

- Solutions
  - To read the checkbox values, use the property *checked* – returns "true" or "false"
  - Associate an *onClick* event for each radio button that modifies a variable
  - Associate an *onClick* event for the button to indicate when it is pressed.

The Use of *value* (continued)



The Use of *value* (continued)



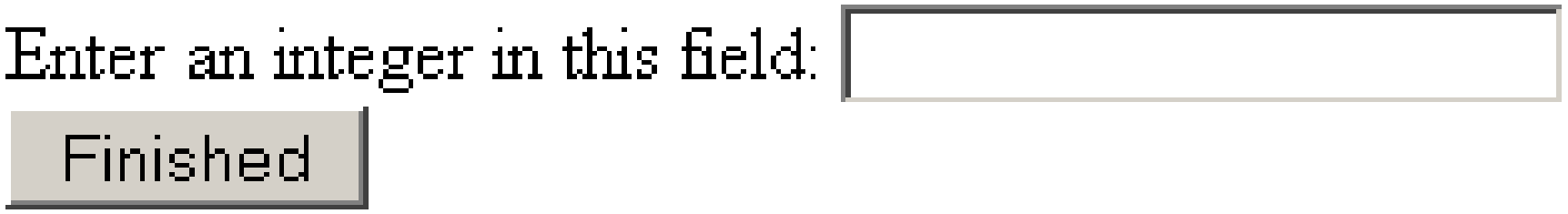
Form Validation

- A very common application of client-side scripts is for validating the data users have entered in a form.
  - For example, we would like to make sure that the user has not done something like entered the word “string" where the form asked for an age.
- The functions covered over the past two lectures will allow us to access form data and verify that it is correct.

Example 2: Simple Number Verification

The form below creates a text box and a button.

```
<form name="sample" id="sample">
Enter an integer in this field:
<input type="text" size="20"
 name="justanumber" id="justanumber"
 onblur="integercheck()" /><br />
<input type="button" value="Finished" />
</form>
```



Example 2: Simple Number Verification (cont)

```
<script language="JavaScript"
  type="text/javascript">
function integercheck()
{
  if (isNaN(document.sample.justanumber))
  {
     window.alert("This field requires an
  integer!");
  document.sample.justanumber.focus();
  }}  </script>
```

Example 3: Name and Password Verification

```
<script>
function validateform(){
var name=document.myform.name.value;
var password=document.myform.password.value;

if (name==null || name==""){
  alert("Name can't be blank");
  return false;
}
else if(password.length<6){
alert("Password must be at least 6 characters
  long.");
return false;
  }
}
```

Example 2: Checking for '@' in E-mail

```
function emailcheck(email_string)
{
    if(email_string.indexOf("@")==-1)
        window.alert("Not a valid
            email address!");
}
```

Example 2: Checking for '@' in E-mail

```
function validateEmail (emailAdress)
{
  let regexEmail = /^\w+([\.-]?\w+)*@\w+([\.-
  ]?\w+)*(\.\w{2,3})+$/;
  if (emailAdress.match(regexEmail)) {
    return true;
  } else {
    return false;
  }
}

let emailAdress = "test@gmail.com";
console.log(validateEmail(emailAdress));
```

Example3: Form Validation

```
<html>
<head><title>Form example </title>
<script language="JavaScript">
function verify(f)
{

if (f.lname.value == null || f.address.value == null)
{   alert("Form needs a last name and an address");
    return false;
}
if (f.lname.value == "" || f.address.value == "")
{   alert("Form needs a last name and an address");
    return false;
}
return true;

}
</script>
</head>
```

Example3: Form Validation (Continue)

```
<body>
<h1> Address Information </h1> <br>
<form method=post enctype="text/plain" action=""
onSubmit="return verify(this);">
```

JavaScript Event

```
First Name: <input type="text" name="fname"> <br />
Last Name: <input type="text" name="lname"> <br />
Street Address: <input type="text" name="address" size=30> <br />
Town/City: <input type="text" name="city">  <br />
State: <select name="state" size=1> <br />
<option value="NY" selected> Utah
<option value="NY" selected> Idaho
<option value="NY" selected> New York
<option value="NJ"> New Jersey
<option value="CT"> Connecticut
<option value="PA"> Pennsylvania
</select> <br />
Status: <input type="radio" name="status" value="R"> Returning client
<input type="radio" name="status" value="N"> New client
<hr /> Thank you <p>
<input type="submit" value="Send information">
</form> </body> </html>
```