Topic 4-2 Core JavaScript  
(FUNCTIONS AND OBJECTS)
CITS3403 Web & Internet Technologies

Reference: Sebesta, Chapter 4
Numeric and String Literals

• Number values are represented internally as *double-precision floating-point* values
  – Number literals can be either integer or float
  – Float values may have a decimal and/or an exponent
• A String literal is delimited by either single or double quotes
  – There is no difference between single and double quotes
  – Certain characters may be *escaped* in strings
    • \’ or \” to use a quote in a string delimited by the same quotes
    • \\ to use a literal backslash
    • \n new line
    • \t tab
    etc
  – The empty string ‘’ or “” has no characters
Type 1 – Numbers

• Numbers in Javascript are all floats.
  – Number literals can be either integer or float
  – Float values may have a decimal and/or an exponent

• There is no integer type in here. According to the spec., they are all double-precision 64-bit format IEEE 754 values. So you might have to be a little careful with arithmetic. Watch out for stuffs like these –
  – 0.1 + 0.2 = 0.30000000000000004

• For advanced mathematical operations you can use the built-in Math object
  – var value = Math.sin(3.5); // gets the sine of 3.5
  – var d = Math.PI * r * r;
Type 1 – Numbers(Contd.)

- You can convert a string to an integer by using the `parseInt()` function –
  
  ```javascript
  var i = parseInt("124", 10); // i now contains 124
  ```

  This function takes the base of the conversion as an optional second argument which you should always provide. Otherwise something like this might happen –

  ```javascript
  var value = parseInt("010"); // value now contains 8!
  ```

  Similarly, you can use `parseFloat()` to convert strings to float, which always uses base 10.
Type 1 – Numbers (Contd.)

- A special value called **NaN (Not a number)** is returned if the argument string to `parseInt()` is non-numeric
  
  ```javascript
  var value = parseInt("hello", 10); // value now contains NaN
  // Question - what will be the type of value?
  ```

- **NaN** is toxic – if you provide it as an input to any mathematical operation the result will also be **NaN**
  
  ```javascript
  var value = NaN + 5; // value is now NaN
  ```

  You can check for **NaN** by using the built-in `isNaN()` function –
  
  ```javascript
  isNaN(value); // will return true if value is NaN
  ```
Type 1 – Numbers (Contd.)

- Javascript also has some special values denoting *Infinity* and –Infinity

```javascript
var infinity = 1 / 0;  // infinity now contains Infinity
var negativeInfinity = -1 / 0;  // as described above
```
Type 2 – Strings

• Strings in Javascript are sequence of Unicode characters, where each character is represented by a 16-bit number. This is a very good news to anyone who has to deal with internationalization.

• A String literal is delimited by either single or double quotes
  – There is no difference between single and double quotes
  – Certain characters may be escaped in strings
    • ‘\’ or “\” to use a quote in a string delimited by the same quotes
    • ‘\’ to use a literal backslash
    • ‘\n’ new line
    • ‘\t’ tab
      etc
  – The empty string ‘” or “” has no characters

• They have some useful properties and methods for manipulation like `length`, `charAt()`, `replace()`, `toUpperCase()`, `toLowerCase()` etc.

• Javascript doesn’t have any Character data-type. So if you want to represent a single character, you need to use a string of length 1.
Other Primitive Types

• Null
  • `null` is a reserved word
    – A variable that is intentionally not assigned a value has a null value
    – Using a null value usually causes an error

• Undefined
  – The value of a variable that is not declared or not assigned a value

• Javascript distinguishes between null, which is a special type of object that indicates a deliberate non-value, and undefined, which is an object of type undefined that indicates an uninitialized value.
Other Types

- Boolean
  - Two values: true and false
- Javascript has a boolean type, with possible values of true and false. Any value can be converted to a boolean according to the following rules –
  - false, 0, the empty string, NaN, null, and undefined all become false
  - all other values become true.

Javascript will apply these rules whenever it expects a boolean, but you can coerce these type conversion by using the Boolean() function.
Declaring Variables

• JavaScript is *dynamically typed*, that is, variables do not have declared types
  – A variable can hold different types of values at different times during program execution

• A variable is declared using the keyword `var`
  ```javascript
  var counter,
  index,
  pi = 3.14159265,
  rover = "Palmer",
  stop_flag = true;
  ```
Variables – some odd moments

- If a variable remains uninitialized, then its type is undefined.

- An important difference from other languages like Java is that in Javascript, you don’t get block-level scope, only functions have scope. So if a variable is defined using `var` inside an `if` or `for` block, it will be visible to the entire function.

- In Javascript, there is no strong type-checking like Java. You can declare a variable to hold an integer and then you can assign a string to that same variable (much like PHP...argh....) –
  
  - var value = 5; value = “Hello”; // No error
Assignment Statements

- Plain assignment indicated by =

- Compound assignment with
  \[ - += \quad -= \quad /= \quad *= \quad %= \quad \ldots \]

- \( a += 7 \) means the same as \( a = a + 7 \)
### Precedence & Associativity of Operators

<table>
<thead>
<tr>
<th>Operators</th>
<th>Associativity</th>
</tr>
</thead>
<tbody>
<tr>
<td>++, --, unary -</td>
<td>Right</td>
</tr>
<tr>
<td>*, /, %</td>
<td>Left</td>
</tr>
<tr>
<td>+, -</td>
<td>Left</td>
</tr>
<tr>
<td>&gt;, &lt;, &gt;=, &lt;=</td>
<td>Left</td>
</tr>
<tr>
<td>==, !=</td>
<td>Left</td>
</tr>
<tr>
<td>===, !==</td>
<td>Left</td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td>Left</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>=, +=, -=, *=, /=, &amp;=,</td>
<td>=, %=</td>
</tr>
</tbody>
</table>

Highest-precedence operators are listed first.
Operators

• Numeric
  – Standard arithmetic
    +  *  –  /  %
  – Increment and decrement
    --  ++

• String
  – Concatenation
    +
Operators – the overloaded + operator

- Operators in Javascript behave the same way as Java, except the plus (+) operator which has some extra features/usage.

- Like Java, you can use it to concatenate two different strings. You can also use it to convert a string to a number

```javascript
var value = + "123"; // value contains 123, as Number

// What could be the difference with parse functions?

- If you add a string to a number, everything is converted into a string first

```javascript
var value = "3" + 4 + 5; // value now contains the string "345";
var test = 3 + 4 + "5"; // test now contains "75"
var test = "August " + 1977 // test now contains "August 1997"
```
Implicit Type Conversion

- JavaScript attempts to convert values in order to be able to perform operations

- Numeric Context
  - $7 \times \text{"3"}$
  - null is converted to 0 in a numeric context, undefined to NaN

- Logical/Boolean Context
  - 0 is interpreted as a Boolean false, all other numbers are interpreted a true
  - The empty string is interpreted as a Boolean false, all other strings (including “0”!) as Boolean true
  - undefined, NaN and null are all interpreted as Boolean false
Comparisons

• Comparisons in Javascript can be made using >, <, >=, <=, ==, ===, != and !== operators. These works for both strings and numbers.

• The == operator performs type coercion if you give it two different types

```
"dog" == "dog"    // true
1 == true        // true!
```
Automatic Type Coercion in comparison context

- If either operand is a number or a boolean, the operands are converted to number if possible.

- Else if either operand is a string, the other operand is converted to a string if possible.

- If both operands are objects, then Javascript compares internal references which are equal when operands refer to the same object in memory.

- For the detailed Abstract Equality Comparison Algorithm, you can check out section 11.9.3 of *ECMAScript Language Specification*.

```
"0" == false  //??
```
Automatic Type Coercion – consequences

• Given the above definition
  – String comparison can be forced by – ( "" + a == "" + b )
  – Numeric comparison can be forced by – ( +a == +b )
  – Boolean comparison can be forced by – ( !a == !b )

• Some interesting comparisons
  – null == null.................................................// true
  – undefined == undefined............................// true
  – NaN == NaN..............................................// false
  – null == undefined......................................// true
  – new String(“a”) == “a”.................................// true
  – new String(“a”) == new String(“a”)...............// false
Explicit Type Conversion

- Explicit conversion of string to number
  - `Number(aString)`
    - `aString` – 0
    - Number must begin the string and be followed by space or end of string
  - `parseInt` and `parseFloat` convert the beginning of a string but do not cause an error if a non-space follows the numeric part
The `typeof` Operator

- Returns “number” or “string” or “boolean” for primitive types
- Returns “object” for an object or null
- Two syntactic forms
  - `typeof x`
  - `typeof(x)`
The Number Object

• Properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>MAX_VALUE</td>
<td>Largest representable number</td>
</tr>
<tr>
<td>MIN_VALUE</td>
<td>Smallest representable number</td>
</tr>
<tr>
<td>NaN</td>
<td>Not a number</td>
</tr>
<tr>
<td>POSITIVE_INFINITY</td>
<td>Special value to represent infinity</td>
</tr>
<tr>
<td>NEGATIVE_INFINITY</td>
<td>Special value to represent negative infinity</td>
</tr>
<tr>
<td>PI</td>
<td>The value of π</td>
</tr>
</tbody>
</table>

Operations resulting in errors return NaN

  – Use isNaN(a) to test if a is NaN

• toString method converts a number to string
String Properties and Methods

- One property: length
  - Note to Java programmers, this is not a method!
- Character positions in strings begin at index 0

<table>
<thead>
<tr>
<th>Method</th>
<th>Parameters</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>charAt</td>
<td>A number</td>
<td>Returns the character in the String object that is at the specified position</td>
</tr>
<tr>
<td>indexOf</td>
<td>One-character string</td>
<td>Returns the position in the String object of the parameter</td>
</tr>
<tr>
<td>substring</td>
<td>Two numbers</td>
<td>Returns the substring of the String object from the first parameter position to the second</td>
</tr>
<tr>
<td>toLowerCase</td>
<td>None</td>
<td>Converts any uppercase letters in the string to lowercase</td>
</tr>
<tr>
<td>toUpperCase</td>
<td>None</td>
<td>Converts any lowercase letters in the string to uppercase</td>
</tr>
</tbody>
</table>
Regular Expressions

- Regular expressions are used to specify patterns in strings
- JavaScript provides two methods to use regular expressions in pattern matching
  - String methods
  - RegExp objects (not covered in the text)
- A regular expression pattern is indicated by enclosing the pattern in slashes
- The `search` method returns the position of a match, if found, or -1 if no match was found
Example Using search

```javascript
var str = "Rabbits are furry";
var position = str.search(/bits/);
if (position > 0)
    console.log("'bits' appears in position",
                position, "<br />");
else
    console.log(
             "'bits' does not appear in str <br />");
```

- This uses a pattern that matches the string ‘bits’

- The output of this code is as follows:
  - 'bits' appears in position 3
Characters and Character-Classes

- **Metacharacters** have special meaning in regular expressions
  - \ | ( ) [ ] { } ^ $ * + ? .
    - May be used literally by escaping them with \n- Other characters represent themselves
- A period matches any single character
  - /f.r/ matches for and far and fir but not fr
Characters and Character-Classes

- A character class matches one of a specified set of characters
  - `[character set]`
  - List characters individually: [abcdef]
  - Give a range of characters: [a-z]
  - Beware of [A-z]
  - `^` at the beginning negates the class
## Predefined character classes

<table>
<thead>
<tr>
<th>Name</th>
<th>Equivalent Pattern</th>
<th>Matches</th>
</tr>
</thead>
<tbody>
<tr>
<td>\d</td>
<td>[0–9]</td>
<td>A digit</td>
</tr>
<tr>
<td>\D</td>
<td>[^0–9]</td>
<td>Not a digit</td>
</tr>
<tr>
<td>\w</td>
<td>[A–Za–z_0–9]</td>
<td>A word character (alphanumeric)</td>
</tr>
<tr>
<td>\W</td>
<td>[^A–Za–z_0–9]</td>
<td>Not a word character</td>
</tr>
<tr>
<td>\s</td>
<td>[\r\t\n\f]</td>
<td>A whitespace character</td>
</tr>
<tr>
<td>\S</td>
<td>[^\r\t\n\f]</td>
<td>Not a whitespace character</td>
</tr>
</tbody>
</table>
Repeated Matches

• A pattern can be repeated a fixed number of times by following it with a pair of curly braces enclosing a count

• A pattern can be repeated by following it with one of the following special characters
  – * indicates zero or more repetitions of the previous pattern
  – + indicates one or more of the previous pattern
  – ? indicates zero or one of the previous pattern

• Examples
  – /\(\backslash\{3\}\) *\d\{4\} *\d\{4\}/ might represent a telephone number
  – /\[$_a-zA-Z][$_a-zA-Z0-9]*/ matches identifiers
Anchors

• Match positions rather than characters
  – Anchors are 0 width and may not take multiplicity modifiers

• Anchoring to the ends of a string
  – ^ at the beginning of a pattern matches the beginning of a string
  – $ at the end of a pattern matches the end of a string
    • The $ in /a$b/ matches a $ character

• Anchoring at a word boundary
  – \b matches the position between a word character and a non-word character or the beginning or the end of a string
  – /\bthe\b/ will match ‘the’ but not ‘theatre’ and will also match ‘the’ in the string ‘one of the best’
Pattern Modifiers

• Pattern modifiers are specified by characters that follow the closing / of a pattern

• Modifiers modify the way a pattern is interpreted or used

• The $x$ modifier causes whitespace in the pattern to be ignored
  – This allows better formatting of the pattern
  – $\backslash s$ still retains its meaning

• The $g$ modifier is explained in the following
Other Pattern Matching Methods

- The `replace` method takes a pattern parameter and a string parameter
  - replaces a match of the pattern in the target string with the second parameter
  - A `g` modifier on the pattern causes multiple replacements

```javascript
var str = "low, lower, lowest";
str.replace(/low/g, "high");
```
Other Pattern Matching Methods

• Parentheses can be used in patterns to mark sub-patterns
  – The pattern matching machinery will remember the parts of a matched string that correspond to sub-patterns

• The `match` method takes one pattern parameter
  – With a `g` modifier, the return is an array of all matches
  – Without a `g` modifier, the return is an array of the match and parameterized (parenthesised) sub-matches

```javascript
var str = "The Dockers will be 1st in 2009";
var matches = str.match(/\d/g); // [1,2,0,0,9]
matches = str.match(/(\d+)([^\d+])(\d+)/);
// ["1st in 2009","1","st in ","2009"]
```
Other Pattern Matching Methods

- The `split` method splits the object string using the pattern to specify the split points
  - useful for parsing input..

```javascript
var str = "The Dockers will be 1st in 2009";
var words = str.split(" ");
// ["The", "Dockers", "will", "be", "1st",...]
```
An Example

• Using javascript to check the validity of input data

• forms_check.js

• (Note, a server program may need to check the data sent to it since the validation can be bypassed in a number of ways)
Control Statements

- A *compound statement* in JavaScript is a sequence of 0 or more statements enclosed in curly braces.

- A *control construct* is a control statement including the statements or compound statements that it contains.
Control Structures

• Javascript has for, while, do-while loops just like Java. It also has if-else, switch statements and ternary operator. Switch statements can compare string values. You can also use an expression in the case statement.

• The && and || operators use short-circuit logic, which means whether they will execute their second operand depends on the first. This is useful for checking for null objects before accessing their attributes –

    // && will return Object if it’s null
    var property = Object && Object.getProperty();

• Or for setting their default values –

    var name = otherName || “default”;

Copyright © 2013 Slideshare. Introduction to JavaScript by Sayem Ahmed. Edited by Wei Liu.
Control Expressions

• A control expression has a Boolean value
  – An expression with a non-Boolean value will have its value converted to Boolean automatically

• Comparison operators
  
  ==  !=  <  <=  >  >=
  
  ===  and  !== compares identity of values or objects
  
  3 == ‘3’ is true due to automatic conversion
  
  3 !== ‘3’ is false

• Boolean operators
  
  –  &&  ||  !

• Warning! A Boolean object evaluates as true
  
  – Unless the object is null or undefined
Selection Statements

• The if-then and if-then-else are similar to that in other programming languages, especially C/C++/Java
**switch Statement Syntax**

```java
switch (expression) {
    case value_1:
        // statement(s)
    case value_2:
        // statement(s)
        ...
    [default:
        // statement(s)]
}
```

- Execution continues until either the end of the switch is encountered or a `break` statement is executed.
switch Statement Semantics

- The expression is evaluated
- The value of the expressions is compared to the value in each case in turn
- If no case matches, execution begins at the default case
- Otherwise, execution continues with the statement following the case
- Execution continues until either the end of the switch is encountered or a `break` statement is executed
Example borders2.js

User Input Prompt

Results

2006 NFL Divisional Winners

<table>
<thead>
<tr>
<th>American Conference</th>
<th>National Conference</th>
</tr>
</thead>
<tbody>
<tr>
<td>East New England Patriots</td>
<td>Philadelphia Eagles</td>
</tr>
<tr>
<td>North Baltimore Ravens</td>
<td>Chicago Bears</td>
</tr>
<tr>
<td>West San Diego Chargers</td>
<td>Seattle Seahawks</td>
</tr>
<tr>
<td>South Indianapolis Colts</td>
<td>New Orleans Saints</td>
</tr>
</tbody>
</table>
Loop Statements

• Loop statements in JavaScript are similar to those in C/C++/Java.

• For

```javascript
for (initial expression; control expression; increment expression)  
    statement or compound statement
```
Loop Statements

• while

  while \( (control\ expression) \)
  \( statement\ or\ compound\ statement \)

• do/while

  do  \( statement\ or\ compound\ statement \)
  while \( (control\ expression) \)
Arrays

- Arrays are lists of elements indexed by a numerical value.
- Array indexes in JavaScript begin at 0.
- Arrays can be modified in size even after they have been created.
### Array Object Creation

- Arrays can be created using the **new Array** method
  - new Array with one parameter creates an empty array of the specified number of elements
    ```javascript
    new Array(10);
    ```
  - new Array with no parameter creates an empty array
    ```javascript
    var a = new Array();
    a[0] = "dog"; a[1] = "cat"; a[2] = "hen";
    console.log(a.length);  // outputs 3
    ```
  - new Array with two or more parameters creates an array with the specified parameters as elements
    ```javascript
    new Array(1, 2, "three", "four");
    ```
- Literal arrays can be specified using square brackets to include a list of elements
  ```javascript
  var alist = [1, "ii", "gamma", "4"];
  ```
- Elements of an array **do not have to be of the same type**
Characteristics of Array Objects

• The length of an array is one more than the highest index

• You can iterate over an array using this length property, or you can use `for...in` construct

```
for(var i in a)
    console.log( a[i] );
```

• Assignment to an index greater than or equal to the current length simply increases the length of the array 😨

  - `a[100] = “lion”;`  
  - `console.log(a.length);`  
  - (Note: errors may go unnoticed.)

• Only assigned elements of an array occupy space

  – Suppose an array were created using `new Array(200)`
  – Suppose only elements 150 through 174 were assigned values
  – Only the 25 assigned elements would be allocated storage, the other 175 would not be allocated storage

• If you query a non-existent array index, you get undefined –

```
console.log(a[90])  // outputs undefined
```
Example

- *insert_names.js*
  - This example shows the dynamic nature of arrays in JavaScript
Array Methods

- **join**  
  `array.join(separator)`

- **reverse**  
  `array.reverse()`

- **sort**  
  `array.sort(sortfunction)`

- **concat**  
  `array1.concat(array2,array3,...,arrayX)`

- **slice**  
  `array.slice(start, end)`
Dynamic List Operations

- **push**
  - Add to the end
- **pop**
  - Remove from the end
- **shift**
  - Remove from the front
- **unshift**
  - Add to the front

Like [Perl](https://en.wikipedia.org/wiki/Perl) and [Ruby](https://en.wikipedia.org/wiki/Ruby), JavaScript already have operations for pushing and popping an array from both ends, so one can use `push` and `shift` functions to enqueue and dequeue a list (or, in reverse, one can use `unshift` and `pop`).


Diagram from Wikipedia.
Two-dimensional Arrays

- A two-dimensional array in JavaScript is an array of arrays
  - This need not even be rectangular shaped: different rows could have different length
- Example [nested_arrays.js](nested_arrays.js) illustrates two-dimensional arrays
Function Fundamentals

• Function definition syntax
  – A function definition consists of a header followed by a compound statement
  – A function header:
    • \textit{function}\ function-name\( (\text{optional-formal-parameters}) \)

• Function call syntax
  – Function name followed by parentheses and any actual parameters
  – Function call may be used as an expression or part of an expression

• Functions must be defined before use in the page header (or linked in an external file)
Function Fundamentals

- return statements
  - A return statement causes a function to cease execution and control to pass to the caller
  - A return statement may include a value which is sent back to the caller
  - If the function doesn’t have any return statement, or uses an empty return with no value, then undefined is returned.
Functions

• Along with the objects, functions are the core components in understanding Javascript. The most basic function is as follows

```javascript
function add(x, y){
    var total = x+y;
    return total;
}
```

• You can call the above function with no parameter as well. In such case, they will be set to `undefined`. 
Calling Functions from XHTML

- **JavaScript file:**
  ```javascript
  function myfunction (myparameter1, myparameter 2, ...) {
      // do something
      console.log("My answer is...",answer); // or maybe...
      return answer;
  }
  ```

- **XHTML file:**
  ```html
  <head>
  <script type="text/javascript" src="somefile.js"></script>
  ...
  </head>
  <body>
  <script type="text/javascript">
  myfunction(7,6);
  </script>
  ```
Functions are Objects

- Functions are objects in JavaScript

- Functions may, therefore, be assigned to variables and to object properties
  - Object properties that have function name as values are methods of the object
Functions are Objects

- Example

```javascript
function fun() {
    console.log("This surely is fun!");
}
ref_fun = fun; // Now, ref_fun refers to // the fun object
fun(); // A call to fun
ref_fun(); // Also a call to fun
```
Local Variables

• “The *scope* of a variable is the range of statements over which it is visible”

• A *variable not declared using* `var` *has global scope*, visible throughout the page, even if used inside a function definition

• A variable declared with `var` *outside a function definition* has global scope

• A variable declared with `var` *inside a function definition* has local scope, visible only inside the function definition
  
  – If a global variable has the same name, it is hidden inside the function definition
Parameters

• Parameters named in a function header are called *formal parameters*

• Parameters used in a function call are called *actual parameters*

• Parameters are *passed by value*
  
  – For an object parameter, the reference is passed, so the function body can actually change the object (effectively *pass by reference*)
  
  – However, an assignment to the formal parameter will not change the actual parameter
Parameter Passing Example

```javascript
function fun1(my_list) {
    var list2 = new Array(1, 3, 5);
    my_list[3] = 14; //changes actual parameter

    my_list = list2; //no effect on actual parameter

    return my_list;
}
```

```javascript
var list = new Array(2, 4, 6, 8)
fun1(list);
```

- The first assignment changes list in the caller
- The second assignment has no effect on the list object in the caller
Parameter Checking

• JavaScript checks neither the type nor number of parameters in a function call
  – Formal parameters have no type specified
  – Extra actual parameters are ignored (however, see below)
  – If there are fewer actual parameters than formal parameters, the extra formal parameters remain undefined

• This flexibility is typical of many scripting languages
  – different numbers of parameters may be appropriate for different uses of the function

• A property array named arguments holds all of the actual parameters, whether or not there are more of them than there are formal parameters
Functions

• You can pass in more arguments than the function is expecting
  
  ```javascript
  console.log( add(2, 3, 4) ); // outputs 5
  ```

• Functions have access to an additional variable inside their body called **arguments**, which is an array-like objects holding all of the values passed to that function. Let’s write a function which takes as many arguments values as we want

  ```javascript
  function avg(){
      var sum = 0;
      for (var i=0; i<arguments.length; i++)
          sum += arguments[i];
      return sum / arguments.length;
  }
  ```
Calling/Applying Functions

What if we want to calculate the average value of an array? We can re-use the above function for arrays in the following way:

```javascript
console.log( avg.apply(null, [2, 3, 4, 5]) ); // outputs 3.5

// apply() has a sister function called `call`. The fundamental difference is that
// `call()` accepts an argument list, while `apply()` accepts a single array of arguments.

var x = 10;
var o = { x: 15 };
function f(message) {
    console.log(message, this.x);
}

f("invoking f");
f.call(o, "invoking f via call");
```
Anonymous Functions

• In Javascript, you can create anonymous functions

  var avg = function() { // the rest of the body……… } 

This is extremely powerful as it lets you put a function definition anywhere that you would normally put an expression.
Function Passing Example - The sort Method

• A parameter can be passed to the sort method to specify how to sort elements in an array
  – The parameter is a function that takes two parameters
  – The function returns a negative value to indicate the first parameter should come before the second
  – The function returns a positive value to indicate the first parameter should come after the second
  – The function returns 0 to indicate the first parameter and the second parameter are equivalent as far as the ordering is concerned

Example: median.js illustrates the sort method.

```
Median of [1,3,4,7,8,9] is: 6
Median of [-2,0,1,3,5,7,10] is: 3
```
Functions (Recursive)

- Like any other languages, you can write recursive functions in Javascript. However, this creates a problem if the function is anonymous. How would you call a function without its name? The solution is using named anonymous functions -

```javascript
var ninja = {
    yell: function cry(n) {
        return n > 0 ? cry(n-1) + "a" : "hiy";
    }
};

console.log( ninja.yell(5) ); // outputs hiyaaaaaa
```
Constructors

• Constructors are functions that create and initialize properties for new objects

• A constructor uses the keyword `this` in the body to reference the object being initialized

• Object methods are properties that refer to functions
  – A function to be used as a method may use the keyword `this` to refer to the object for which it is acting

• Example `car_constructor.js`
Example: Car Constructor

```javascript
function display_car () {
    console.log ("Car make: ", this.make, ",<br/>";?></p>
    console.log ("Car model: ", this.model, ",<br/>";?></p>
    console.log ("Car make: ", this.year, ",<br/>";?></p>
}
// Constructor
function Car (new_make, new_model, new_year) {
    this.make = new_make;
    this.model = new_model;
    this.year = new_year;
    // add a method...
    this.display = display_car;
}

// create an object
my_car = new Car ("Ford", "Fusion", "2012");
my_car.display();
```
Objects

• Javascript objects are simply collections of name-value pairs. As such, they are similar to HashMaps in Java. An object may be thought of as a Map/Dictionary/Associative-Storage.

• If a variable is not a primitive (undefined, null, boolean, number or string), its an object.

• The name part is a string, while the value can be any Javascript value – including more objects.
Object Creation and Modification

- There are two basic ways to create an empty object –
  - The `new` expression is used to create an object
    ```javascript
    var obj = new Object();
    ```
    - This includes a call to a `constructor`
    - The new operator creates a blank object, the constructor creates and initializes all properties of the object
  - The second is called object literal syntax. It’s also the core of JSON format and should be preferred at all times.
    ```javascript
    // sets the objects prototype to Object.prototype
    var obj = {};
    // sets null as object prototype
    var obj = Object.create(null);
    ```
Object literal

- Object literal syntax can be used to initialize an object in its entirety –
  ```javascript
  var obj = {
    name: "Carrot",
    for: "Max",
    detail: { color: "Orange", size: 12 }
  };
  ```

- Attribute access can be chained together –
  ```javascript
  console.log(obj.detail.color);
  ```
Accessing Object Properties

• Just like Java, an object’s properties can be accessed using the dot operator -
  – Obj.name = “Issa Vai the Guru”
• And using the array-like index –
  – Obj[“name”] = “Anwar the Boss”;
• Both of these methods are semantically equivalent.

• The second method has the advantage that the name of the property is provided as a string, which means it can be calculated at run-time, though using this method prevents some Javascript engine and minifier optimizations being applied. It can also be used to set and get properties with names that are reserved words.
Dynamic Properties

• Create my_car and add some properties

   // Create an Object object
   var my_car = new Object();
   // Create and initialize the make property
   my_car.make = "Ford";
   // Create and initialize model
   my_car.model = "Contour SVT";

• The delete operator can be used to delete a property from an object

   • delete my_car.model
The \textit{for-in} Loop

- **Syntax**

\begin{verbatim}
for (\texttt{identifier} in \texttt{object})
  \texttt{statement} or \texttt{compound statement}
\end{verbatim}

- The loop lets the identifier take on each property in turn in the object

\begin{verbatim}
for (\texttt{var prop in my\_car})
  \texttt{console.log("Key: ", prop, "; Value:",my\_car[prop]);}
\end{verbatim}

- **Result:**

- Name: make; Value: Ford
- Name: model; Value: Contour SVT
Creating Object Properties

```javascript
var person = Object.create(null);

Object.defineProperty(person, 'firstName', {
    value: "Yehuda", writable: true, enumerable: true,
    configurable: true
});

Object.defineProperty(person, 'lastName', {
    value: "Katz", writable: true, enumerable: true,
    configurable: true
});
```
Object-orientation in Javascript

• Javascript doesn’t have classes, so its object-oriented approach doesn’t match that of other popular OOP languages like Java, C# etc. Instead, it supports a variation of Object-oriented programming known as **Prototype-based** Programming.

• In prototype-based programming, classes are not present, and behavior reuse (equivalent to *inheritance* in Java) is accomplished through a process of decorating existing objects which serves as prototypes. This model is also known as **class-less, prototype-oriented** or **instance-based programming**.

• Just like Java, every object in Javascript is an instance of the object **Object** and therefore inherits all its properties and methods.
Creating Custom Objects

- As we have discussed earlier, there is no class statement in Javascript like C++ and Java. Instead, Javascript uses functions as classes. Let’s consider a Person object with first and last name fields. There are two ways in which the name might be displayed: as “first last” or as “last, first”. Using the functions and objects, here is one way of doing it –

  ```javascript
  function makePerson(first, last)
  {
    return { first: first, last: last }; 
  }
  
  function personFullName(person) {
    return person.first + " " + person.last;
  }
  
  function personFullNameReversed(person) {
    return person.last + " , " + person.first;
  }
  
  s = makePerson("Arif Vai", "the Handsome Guy");
  console.log(personFullName(s)); // Arif Vai the Handsome Guy
  console.log(personFullNameReversed(s)); // the Handsome Guy, Arif Vai
  ```
Creating Custom Objects

- This works, but it’s pretty ugly. You end up with dozens of functions in your global namespace. What we really need is a way to attach a function to an object. Since functions are treated as objects, this is pretty easy.

```javascript
function makePerson(first, last){
    return {
        first : first,
        last : last,
        fullName : function() {
            return this.first + " " + this.last;
        },
        fullNameReversed : function() {
            return this.last + " " + this.first;
        }
    }
}

s = makePerson("Arif Vai", "the Handsome Guy");
console.log( s.fullName() );
console.log( s.fullNameReversed() );
```
The *this* keyword

- When used inside a function, this refers to the current object. What that actually means is specified by the way in which you called that function.

- If you called it using the dot notation or bracket notation on an object, that object becomes this. If any of these notations wasn’t used for the call, then this refers to the global object (the window object). For example

```
s = makePerson("Simon", "Willison")
var fullName = s.fullName;
console.log( fullName() ); // will output undefined undefined
```
• We can take advantage of this keyword to improve our function in the following way

```javascript
function Person(first, last) {
    this.first = first;
    this.last = last;
    this.fullName = function() {
        return this.first + ' ' + this.last;
    }
    this.fullNameReversed = function() {
        return this.last + ', ' + this.first;
    }
}

var s = new Person("Kowser Vai", "the Ice-cream Guy");
```
The *new* keyword

- *new* is strongly related to *this*. What it does is it creates a brand new empty object, and then calls the function specified, with this set to that new object. Functions that are designed to be called by *new* are called constructor functions.

- When the code `new Person(…)` is executed, the following things happen –
  1. A new object is created, inheriting from `Person.prototype`.
  2. The constructor function `Person` is called with the specified arguments and this bound to the newly created object. `new Person` is equivalent to `new Person ()`, i.e. if no argument list is specified, `Person` is called without arguments.
  3. The object returned by the constructor function becomes the result of the whole new expression. If the constructor function doesn't explicitly return an object, the object created in step 1 is used instead. (Normally constructors don't return a value, but they can choose to do so if they want to override the normal object creation process.)
Function objects reuse (method 1)

- Every time we are creating a *person* object, we are creating two new brand new function objects within it. Wouldn’t it be better if this code was shared? There are two ways in which code can be shared. The first way is the following

```javascript
function personFullName() {
    return this.first + ' ' + this.last;
}

function personFullNameReversed() {
    return this.last + ', ' + this.first;
}

function Person(first, last) {
    this.first = first;
    this.last = last;
    this.fullName = personFullName;
    this.fullNameReversed = personFullNameReversed;
}
```
Function objects reuse (method 2)

• The second way is to use the *prototype*

```javascript
function Person(first, last) {
    this.first = first;
    this.last = last;
}
Person.prototype.fullName = function() {
    return this.first + ' ' + this.last;
}
Person.prototype.fullNameReversed = function() {
    return this.last + ', ' + this.first;
}
```
The prototype

- `Person.prototype` is an object shared by all instances of `Person`. It forms a part of a lookup chain (or, *prototype chain*): any time you attempt to access a property of `Person` that isn’t set, Javascript will check `Person.prototype` to see if that property exists there instead. As a result, anything assigned to `Person.prototype` becomes available to all instances of that constructor via the `this` object. The root of the prototype chain is `Object.prototype`.

- This is an incredibly powerful tool. Javascript lets you modify something’s prototype at anytime in your program, which means you can add extra methods to existing objects at runtime.
Adding methods at run time using prototype

```javascript
var s = "Issa"
String.prototype.reversed = function(){
    var r = ""
    for (var i = this.length - 1; i >= 0; i--)
    {
        r += this[i];
    }
    return r;
}
s.reversed(); // will output assi

“This can now be reversed".reversed()  // outputs deresver eb won nac sihT
```
// define the Person Class
function Person() {}

Person.prototype.walk = function(){
    console.log('I am walking!');
};

Person.prototype.sayHello = function(){
    console.log('hello');
};

// define the Student class
function Student() {}

// inherit Person
Student.prototype = new Person();

// modify the Person prototype
Person.prototype.sing = function(){
    console.log("Rock and roll");
};

// replace the sayHello method
Student.prototype.sayHello = function(){
    console.log('hi, I am a student');
}

// add sayGoodBye method
Student.prototype.sayGoodBye = function(){
    console.log('goodBye');
}

var student1 = new Student();
student1.sayHello();
student1.walk();
student1.sayGoodBye();
student1.sing();

// check inheritance
console.log(student1 instanceof Person); // true
console.log(student1 instanceof Student); // true
So, what exactly is a prototype?

- A prototype is an object from which other objects inherit properties. Any object can be a prototype.

- Every object has a prototype by default. Since prototype are themselves objects, every prototype has a prototype too (There is only one exception, the default Object prototype at the top of every prototype chain).

- If you try to look up a key on an object and it is not found, JavaScript will look for it in the prototype. It will follow the prototype chain until it sees a null value. In that case, it returns undefined.
Setting object prototype

```javascript
var man = Object.create(null);
defineProperty(man, 'sex', "male");

var yehuda = Object.create(man);
defineProperty(yehuda, 'firstName', "Yehuda");
defineProperty(yehuda, 'lastName', "Katz");

yehuda.sex // "male"
yehuda.firstName // "Yehuda"
yehuda.lastName // "Katz"

Object.getPrototypeOf(yehuda) // returns the man object
```
Inner functions

- JavaScript function declarations are allowed inside other functions

```javascript
function betterExampleNeeded()
{
    var a = 1;
    function oneMoreThanA()
    {
        return a + 1;
    }

    return oneMoreThanA();
}
```

- A closure is the local variables for a function – kept alive after the function has returned.
Why inner functions?

• This provides a great deal of utility in writing more maintainable code. If a function relies on one or two other functions that are not useful to any other part of your code, you can nest those utility functions inside the function that will be called from elsewhere. This keeps the number of functions that are in the global scope down, which is always a good thing.

• This is also a great counter to the lure of global variables. When writing complex code it is often tempting to use global variables to share values between multiple functions — which leads to code that is hard to maintain. Nested functions can share variables in their parent, so you can use that mechanism to couple functions together when it makes sense without polluting your global namespace — 'local globals' if you like. This technique should be used with caution, but it's a useful ability to have.
Javascript Closure

- Using inner functions we can use one of the most powerful abstractions Javascript has to offer – closure. A quick quiz, what does this do –

```javascript
function makeAdder(a) {
    return function(b) {
        return a + b;
    }
}

x = makeAdder(5);
y = makeAdder(20);

console.log(x(6)); // ?
console.log(y(7)); // ?
```
Javascript closure (cont.)

- Here, the outer function (makeAdder) has returned, and hence common sense would seem to dictate that its local variable no longer exist. But they do still exist, otherwise the adder function would be unable to work.

- In actuality, whenever Javascript executes a function, a scope object is created to hold the local variables created within that function. It is initialized with any variables passed in as function parameters.

- This is similar to the global object that all global variables and functions live in, but with a couple of important differences: firstly, a brand new scope object is created every time a function starts executing, and secondly, unlike the global object these scope objects cannot be directly accessed from your code.
Javascript closure (cont.)

• So when makeAdder is called, a scope object is created with one property: a, which is the argument passed to the function. It then returns a newly created function.

• Normally JavaScript's garbage collector would clean up the scope object created for makeAdder at this point, but the returned function maintains a reference back to that scope object. As a result, the scope object will not be garbage collected until there are no more references to the function object that makeAdder returned.

• Scope objects form a chain called the scope chain, similar to the prototype chain used by JavaScript's object system. A closure is the combination of a function and the scope object in which it was created. Closures let you save state — as such, they can often be used in place of objects.
The Math Object

• Provides a collection of properties and methods useful for Number values

• This includes the trigonometric functions such as \( \text{sin} \) and \( \text{cos} \)

• When used, the methods must be qualified, as in \( \text{Math.sin(x)} \)
The Date Object

• A Date object represents a *time stamp*, that is, a point in time

• A Date object is created with the new operator
  
  − `var d = new Date();`
  − creates a Date object for the time at which it was created

  *d.setFullYear(2003, 10, 5);*
  − resets to 5th November 2003
# The Date Object: Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Returns</th>
</tr>
</thead>
<tbody>
<tr>
<td>toLocaleString</td>
<td>A string of the Date information</td>
</tr>
<tr>
<td>getDate</td>
<td>The day of the month</td>
</tr>
<tr>
<td>getMonth</td>
<td>The month of the year, as a number in the range of 0 to 11</td>
</tr>
<tr>
<td>getDay</td>
<td>The day of the week, as a number in the range of 0 to 6</td>
</tr>
<tr>
<td>getFullYear</td>
<td>The year</td>
</tr>
<tr>
<td>getTime</td>
<td>The number of milliseconds since January 1, 1970</td>
</tr>
<tr>
<td>getHours</td>
<td>The number of the hour, as a number in the range of 0 to 23</td>
</tr>
<tr>
<td>getMinutes</td>
<td>The number of the minute, as a number in the range of 0 to 59</td>
</tr>
<tr>
<td>getSeconds</td>
<td>The number of the second, as a number in the range of 0 to 59</td>
</tr>
<tr>
<td>getMilliseconds</td>
<td>The number of the millisecond, as a number in the range of 0 to 999</td>
</tr>
</tbody>
</table>
Example

- *date.js*
  - Uses Date objects to time a calculation
  - Displays the components of a Date object
  - Illustrates a for loop
Window and Document

• The Window object represents the window in which the document containing the script is being displayed

• The Document object represents the document being displayed using DOM (more on this later...)

• Window has two properties
  - `window` refers to the Window object itself
  - `document` refers to the Document object

• The Window object is the default object for JavaScript, so properties and methods of the Window object may be used without qualifying with the class name
Screen Output and Keyboard Input

- Standard output for JavaScript embedded in a browser is the window displaying the page in which the JavaScript is embedded.

- The `write` method of the Document object writes its parameters to the browser window.

- *The output is interpreted as HTML by the browser*

  ```javascript
  document.write("The result is: ", result, "<br />");
  ```

- Writing to the document object is now considered bad practice. For simple debugging use

  ```javascript
  console.log("The result is: ", result, "<br />");
  ```

  The result is: 42
The alert Method

• The alert method opens a dialog box with a message

• The output of the alert is *not* XHTML, so use new lines rather than <br/>

    alert("The sum is:" + sum + "\n");
The confirm Method

• The confirm methods displays a message provided as a parameter
  – The confirm dialog has two buttons: OK and Cancel

• `true` or `false` is returned
  ```javascript
  var question = confirm("Do you want to continue this download?");
  ```
The prompt Method

- Displays its string argument in a dialog box
  - A second argument provides a default content for the user entry area
- The dialog box has an area for the user to enter text

```javascript
name = prompt("What is your name?", "");
```
Example of Input and Output

- roots.html
Errors in Scripts

- JavaScript errors are detected by the browser
- Different browsers report this differently
  - Firefox uses a special console
- Support for debugging is provided
  - IE, the debugger is part of the browser
  - Firefox, plug-ins are available
    - These include Venkman and Firebug
  - Safari: Develop | Show Error Console
    - First use: Choose Preferences | Advanced | Show Develop menu in menu bar
  - Note: Reopen error console after reloading page (bug?)
  - example: forms_check.html