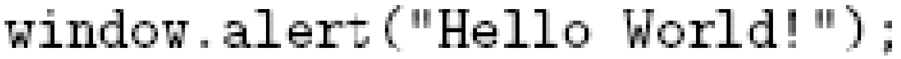
JavaScript

JavaScript Introduction

- Let’s write a “Hello World!” JavaScript program
- **Problem**: the JavaScript language itself has no input/output statements(!)
- **Solution**: Most browsers provide *de facto* standard I/O methods
  - `alert`: pops up alert box containing text
  - `prompt`: pops up window where user can enter text

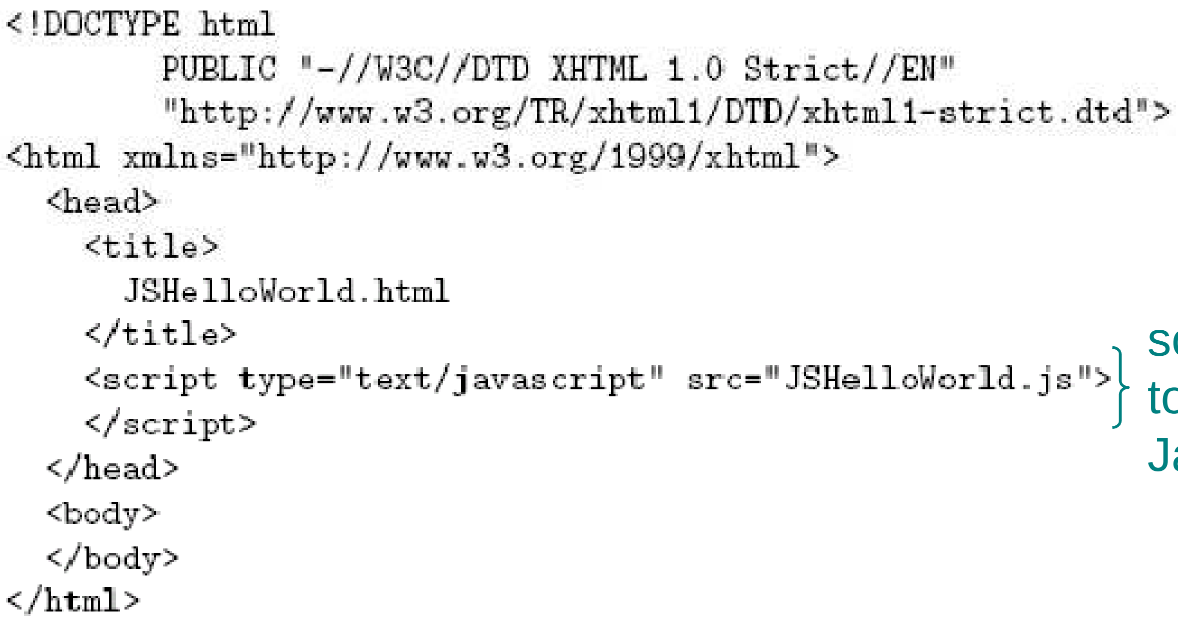
JavaScript Introduction

- File `JSHelloWorld.js`:



```
window.alert("Hello World!");
```

- HTML document executing this code:

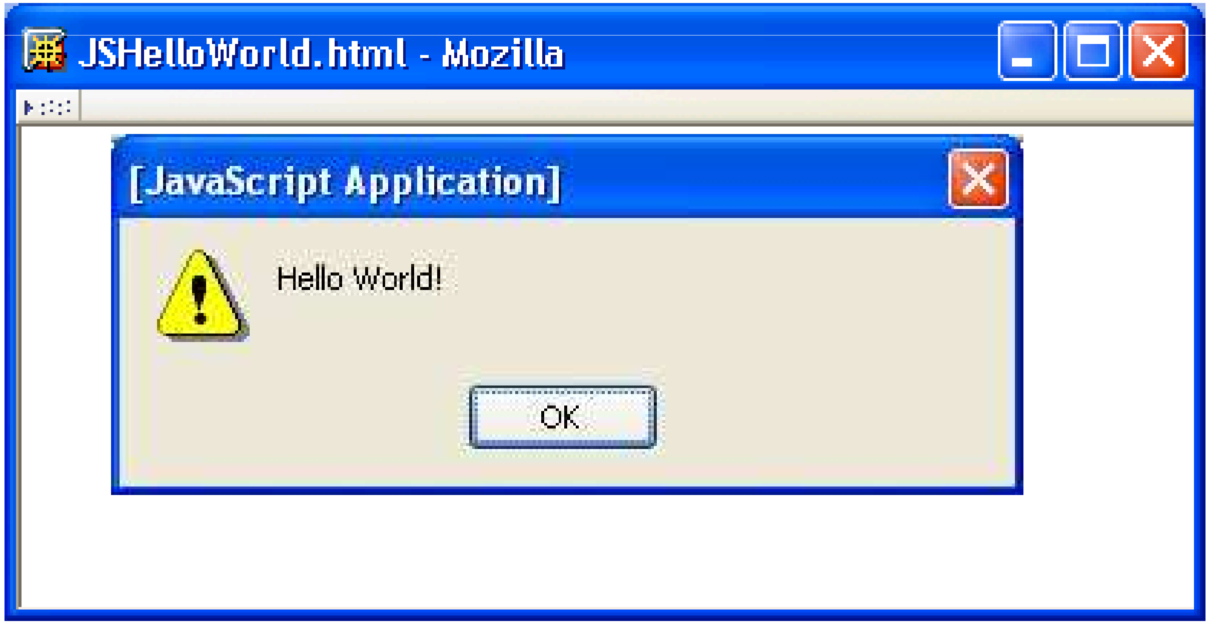


```
<!DOCTYPE html
        PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
        "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <title>
      JSHelloWorld.html
    </title>
    <script type="text/javascript" src="JSHelloWorld.js">
    </script>
  </head>
  <body>
  </body>
</html>
```

script element used to load and execute JavaScript code

JavaScript Introduction

- Web page and alert box generated by `JSHelloWorld.html` document and `JSHelloWorld.js` code:

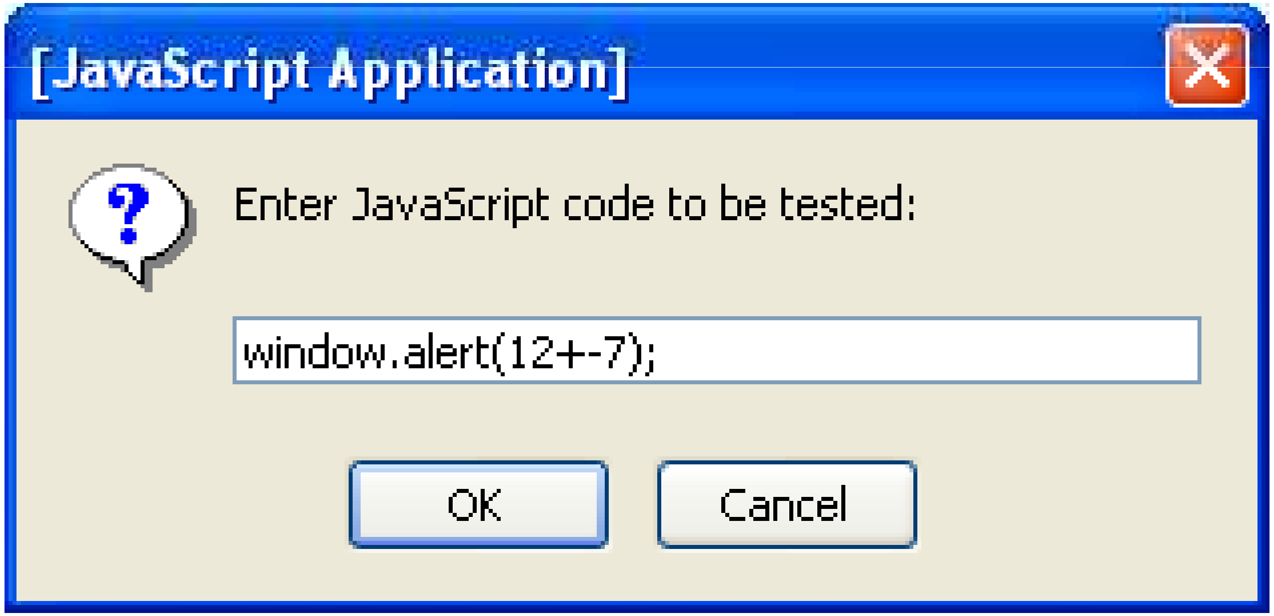


Hello World!

JavaScript Introduction

- Prompt window example:

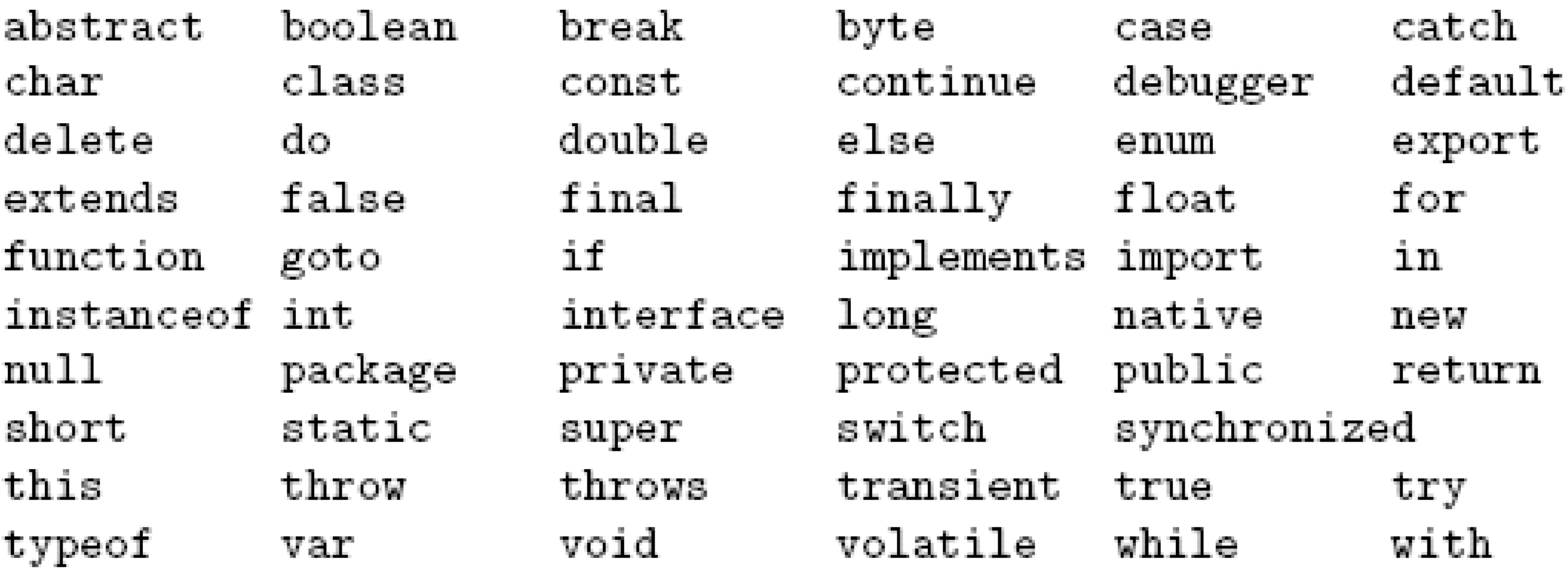
```
var inString = window.prompt("Enter JavaScript code to be tested:",
                             "");
```



Enter JavaScript code to be tested:

window.alert(12+-7);

Variables and Data Types



abstract	boolean	break	byte	case	catch
char	class	const	continue	debugger	default
delete	do	double	else	enum	export
extends	false	final	finally	float	for
function	goto	if	implements	import	in
instanceof	int	interface	long	native	new
null	package	private	protected	public	return
short	static	super	switch	synchronized	
this	throw	throws	transient	true	try
typeof	var	void	volatile	while	with

FIGURE 4.6: JavaScript reserved words.

Variables and Data Types

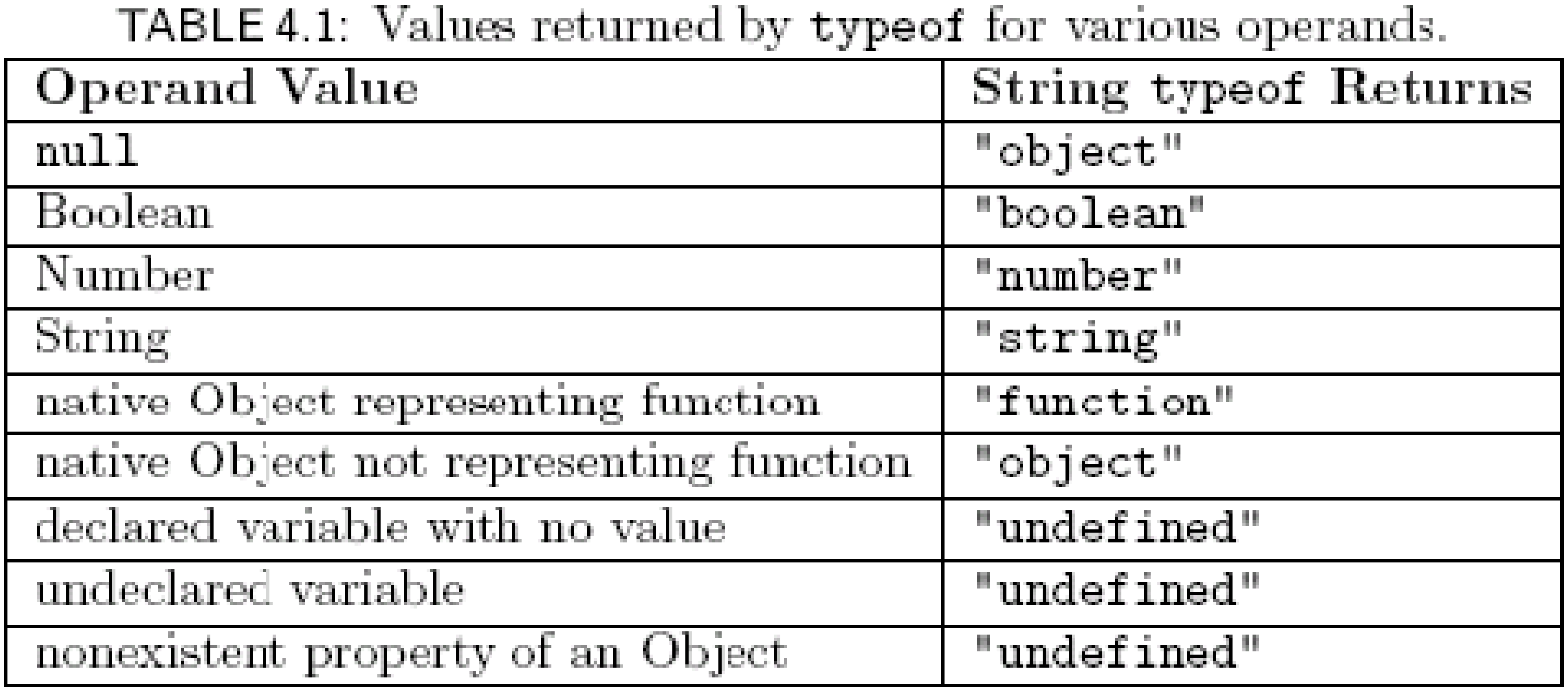
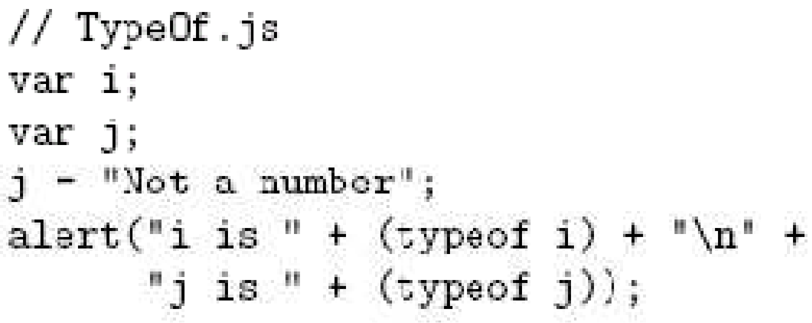


TABLE 4.1: Values returned by `typeof` for various operands.

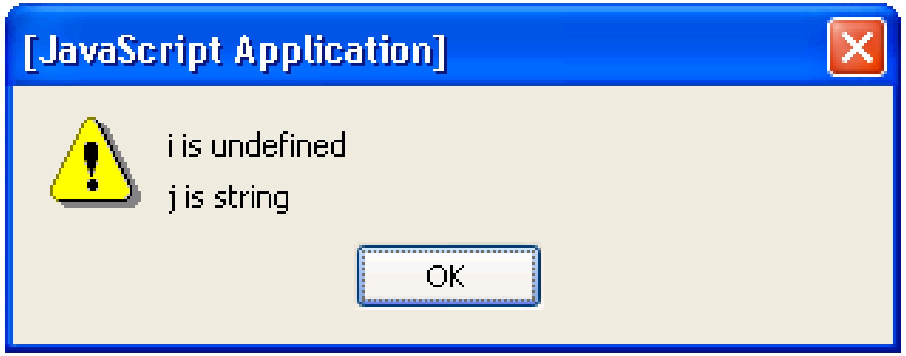
Operand Value	String typeof Returns
`null`	`"object"`
Boolean	`"boolean"`
Number	`"number"`
String	`"string"`
native Object representing function	`"function"`
native Object not representing function	`"object"`
declared variable with no value	`"undefined"`
undeclared variable	`"undefined"`
nonexistent property of an Object	`"undefined"`

Variables and Data Types

- `typeof` operator returns string related to data type
  - Syntax: `typeof` *expression*
- Example:



```
// TypeOf.js
var i;
var j;
j = "Not a number";
alert("i is " + (typeof i) + "\n" +
      "j is " + (typeof j));
```

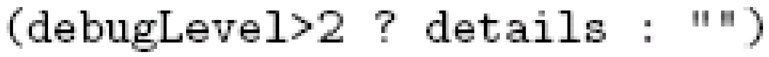


i is undefined

j is string

JavaScript Operators

- Operators are used to create compound expressions from simpler expressions
- Operators can be classified according to the number of operands involved:
  - Unary: one operand (*e.g.*, `typeof i`)
    - Prefix or postfix (*e.g.*, `++i` or `i++`)
  - Binary: two operands (*e.g.*, `x + y`)
  - Ternary: three operands (conditional operator)



```
(debugLevel>2 ? details : "")
```

JavaScript Operators

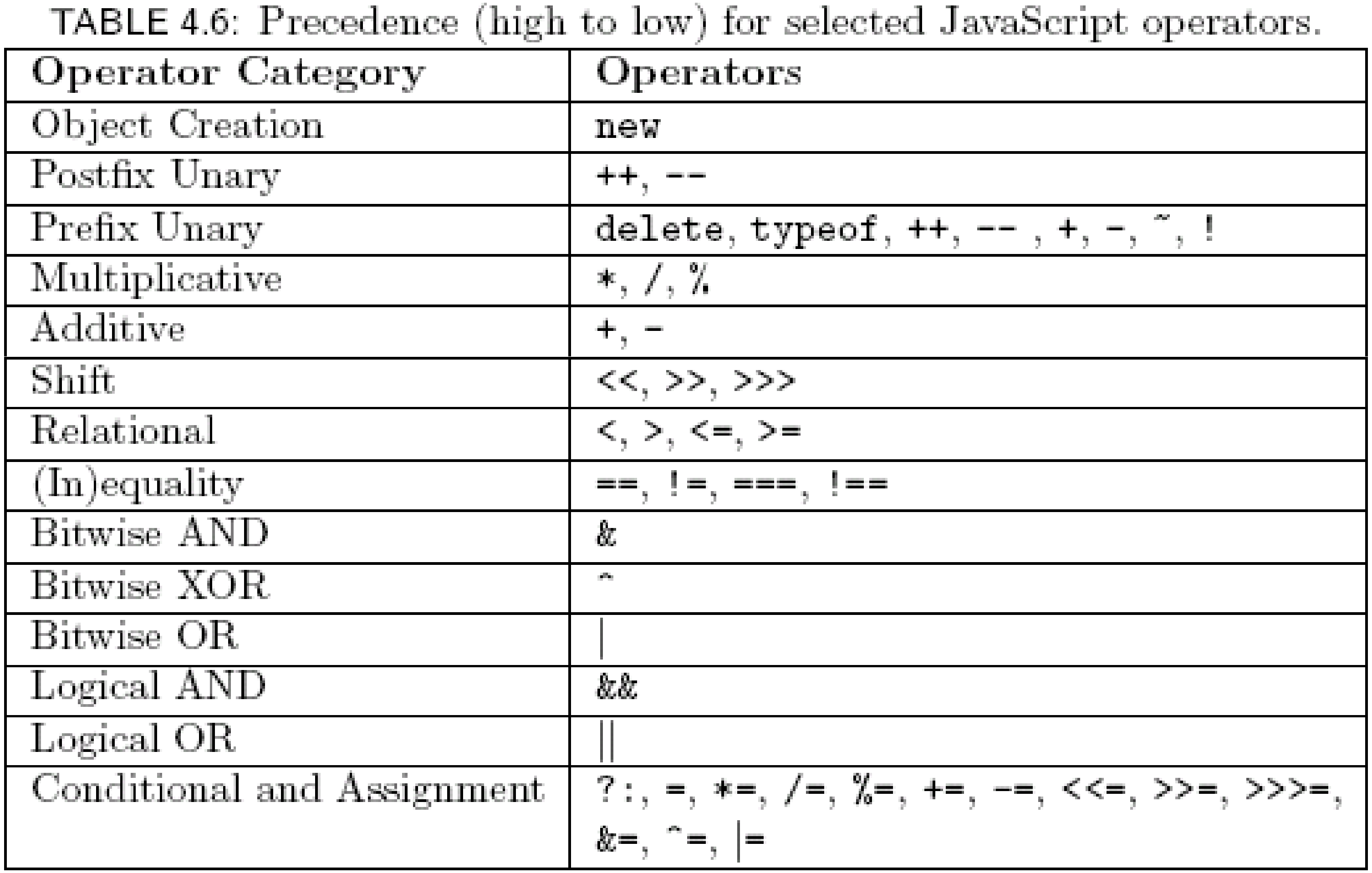
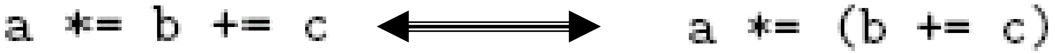


TABLE 4.6: Precedence (high to low) for selected JavaScript operators.

Operator Category	Operators
Object Creation	new
Postfix Unary	++, --
Prefix Unary	delete, typeof, ++, -- , +, -, ~, !
Multiplicative	*, /, %
Additive	+, -
Shift	<<, >>, >>>
Relational	<, >, <=, >=
(In)equality	==, !=, ===, !==
Bitwise AND	&
Bitwise XOR	^
Bitwise OR	|
Logical AND	&&
Logical OR	||
Conditional and Assignment	?:, =, *=, /=, %=, +=, -=, <<=, >>=, >>>=, &=, ^=, |=

JavaScript Operators

- Associativity:
  - Assignment, conditional, and prefix unary operators are right associative: equal-precedence operators are evaluated right-to-left:



```
a *= b += c        a *= (b += c)
```

  - Other operators are left associative: equal-precedence operators are evaluated left-to-right

JavaScript Operators: Automatic Type Conversion

- Binary operators +, -, *, /, % convert both operands to Number
  - Exception: If one of operands of + is String then the other is converted to String
- Relational operators <, >, <=, >= convert both operands to Number
  - Exception: If both operands are String, no conversion is performed and lexicographic string comparison is performed

JavaScript Operators: Automatic Type Conversion

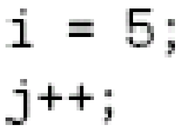
- Operators ===, !== are strict:
  - Two operands are === only if they are of the same type and have the same value
  - “Same value” for objects means that the operands are references to the same object
- Unary +, - convert their operand to Number
- Logical &&, ||, ! convert their operands to Boolean (normally)

JavaScript Operators

- Bit operators
  - Same set as Java:
    - Bitwise NOT, AND, OR, XOR (~, &, |, ^)
    - Shift operators (<<, >>, >>>)
  - Semantics:
    - Operands converted to Number, truncated to integer if float, treated as if two's complement, truncated to low-order 32 bits
    - Operators then applied as if in 32-bit registers
    - Result of >>> treated as unsigned, others signed

JavaScript Statements

- Expression statement: any statement that consists entirely of an expression
  - Expression: code that represents a value



```
i = 5;
j++;
```

- Block statement: one or more statements enclosed in { } braces
- Keyword statement: statement beginning with a keyword, *e.g.*, `var` or `if`

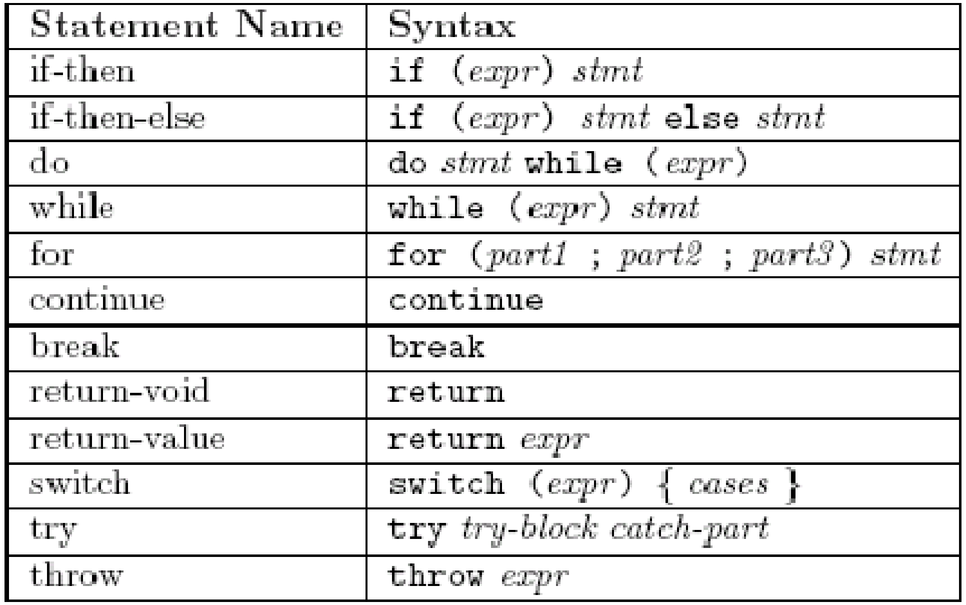
JavaScript Statements

- `var` syntax: `var i, msg="hi", o=null;`

  Comma-separated declaration list with optional initializers

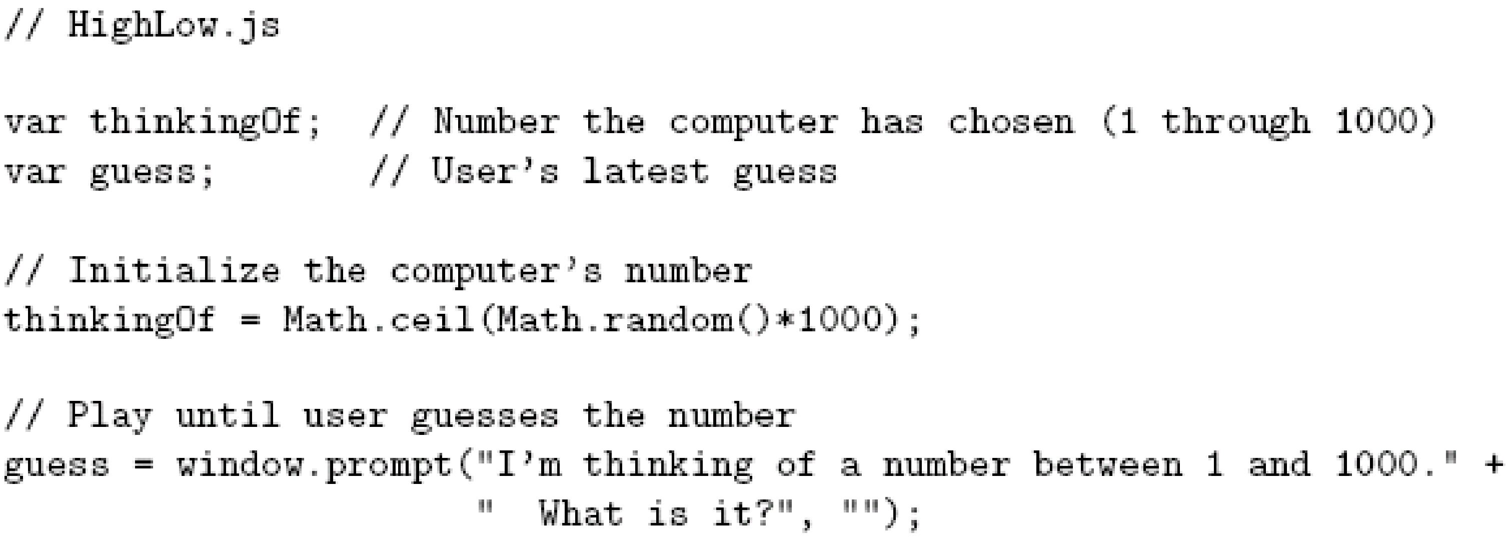
- Java-like keyword statements:

TABLE 4.5: JavaScript keyword statements.



Statement Name	Syntax
if-then	`if (`*expr*`)` *stmt*
if-then-else	`if (`*expr*`)` *stmt* `else` *stmt*
do	`do` *stmt* `while (`*expr*`)`
while	`while (`*expr*`)` *stmt*
for	`for (`*part1* `;` *part2* `;` *part3*`)` *stmt*
continue	`continue`
break	`break`
return-void	`return`
return-value	`return` *expr*
switch	`switch (`*expr*`) {` *cases* `}`
try	`try` *try-block catch-part*
throw	`throw` *expr*

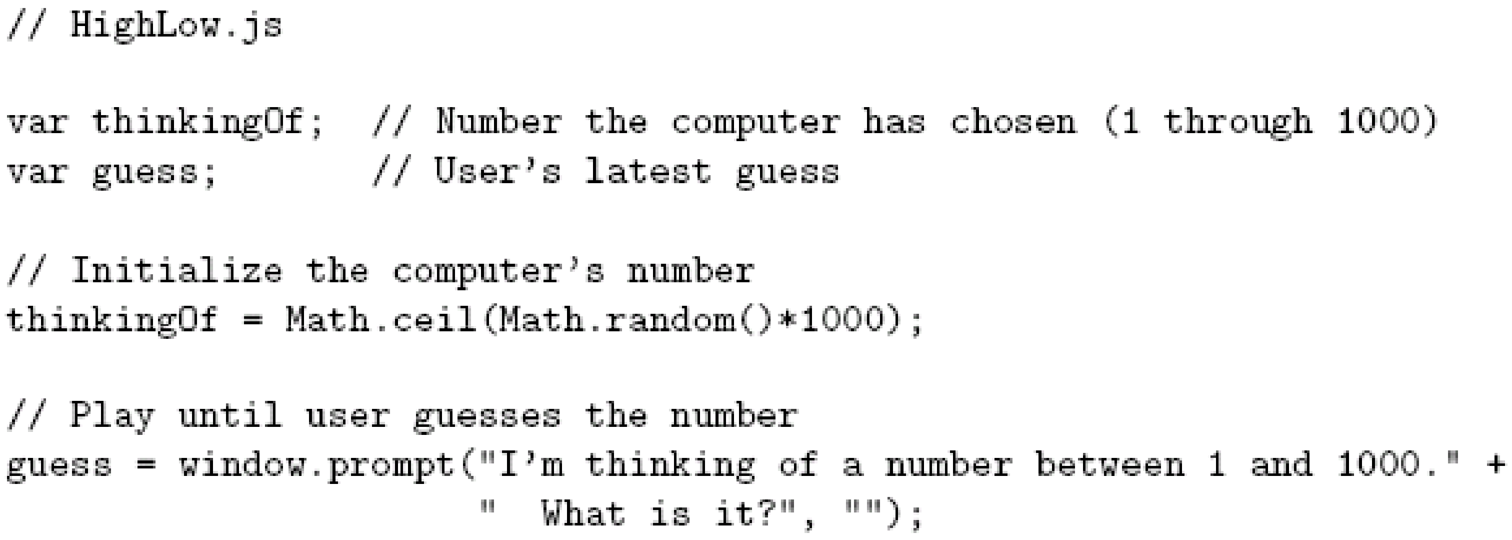
Basic JavaScript Syntax



```
// HighLow.js

var thinkingOf;  // Number the computer has chosen (1 through 1000)
var guess;       // User's latest guess

// Initialize the computer's number
thinkingOf = Math.ceil(Math.random()*1000);

// Play until user guesses the number
guess = window.prompt("I'm thinking of a number between 1 and 1000." +
                      "  What is it?", "");
```

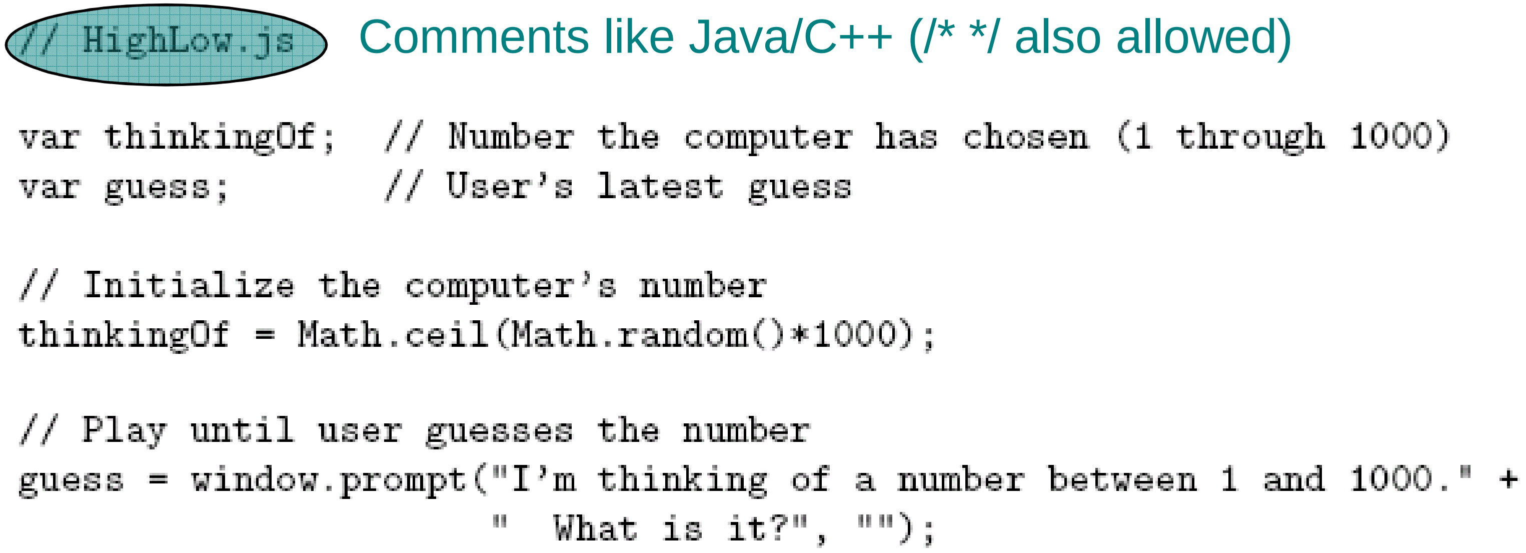
Basic JavaScript Syntax

Notice that there is no main() function/method



```
// HighLow.js

var thinkingOf;  // Number the computer has chosen (1 through 1000)
var guess;       // User’s latest guess

// Initialize the computer’s number
thinkingOf = Math.ceil(Math.random()*1000);

// Play until user guesses the number
guess = window.prompt("I’m thinking of a number between 1 and 1000." +
                      "  What is it?", "");
```

Basic JavaScript Syntax

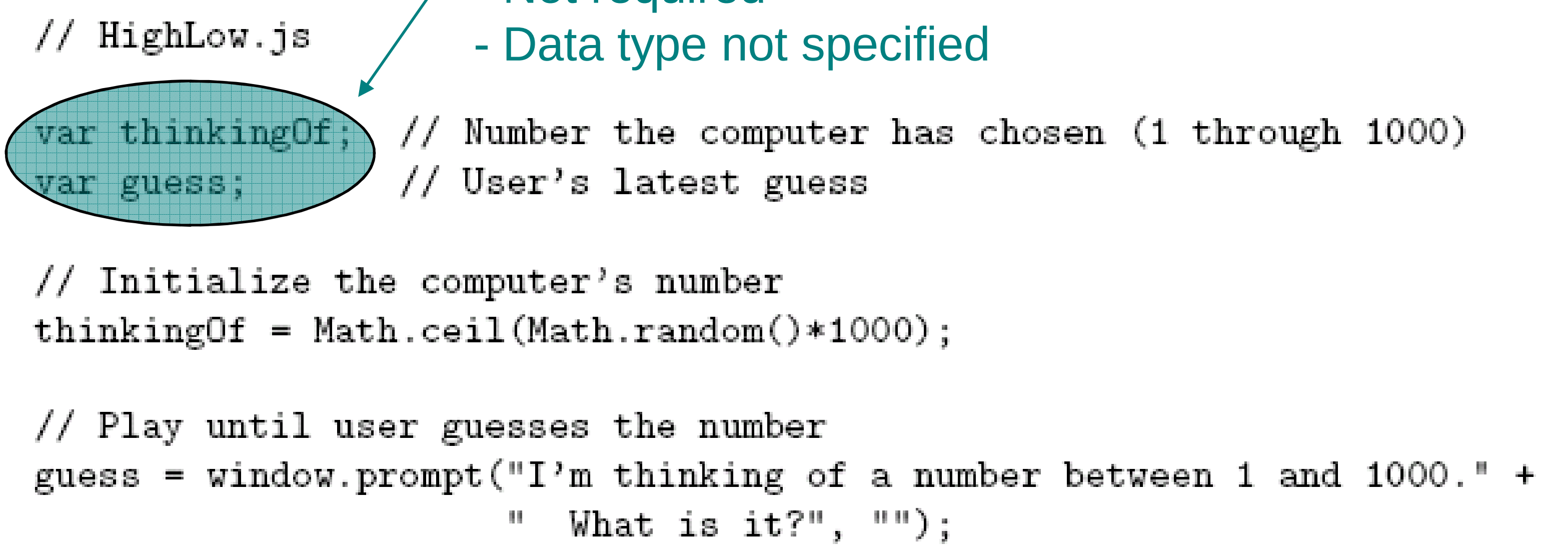


Comments like Java/C++ (/* */ also allowed)

```
// HighLow.js

var thinkingOf;  // Number the computer has chosen (1 through 1000)
var guess;       // User's latest guess

// Initialize the computer's number
thinkingOf = Math.ceil(Math.random()*1000);

// Play until user guesses the number
guess = window.prompt("I'm thinking of a number between 1 and 1000." +
                      "  What is it?", "");
```

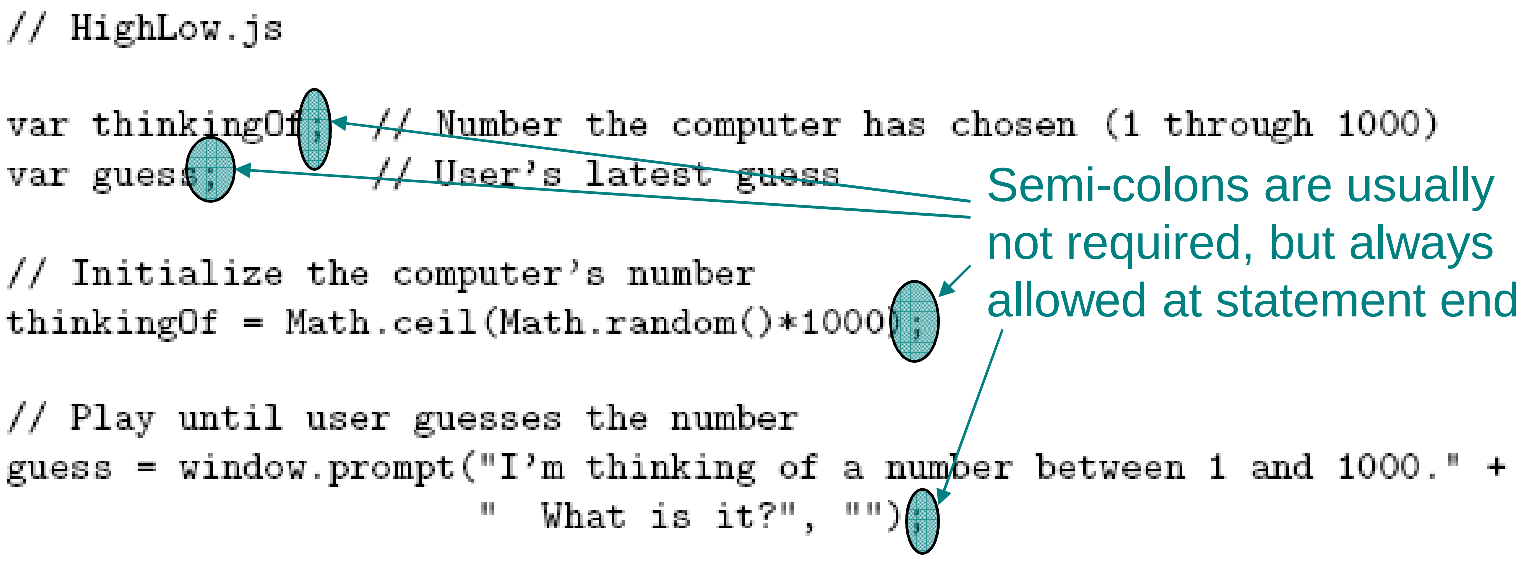
Basic JavaScript Syntax

Variable declarations:
- Not required
- Data type not specified



```
// HighLow.js

var thinkingOf;   // Number the computer has chosen (1 through 1000)
var guess;        // User's latest guess

// Initialize the computer's number
thinkingOf = Math.ceil(Math.random()*1000);

// Play until user guesses the number
guess = window.prompt("I'm thinking of a number between 1 and 1000." +
                      "  What is it?", "");
```

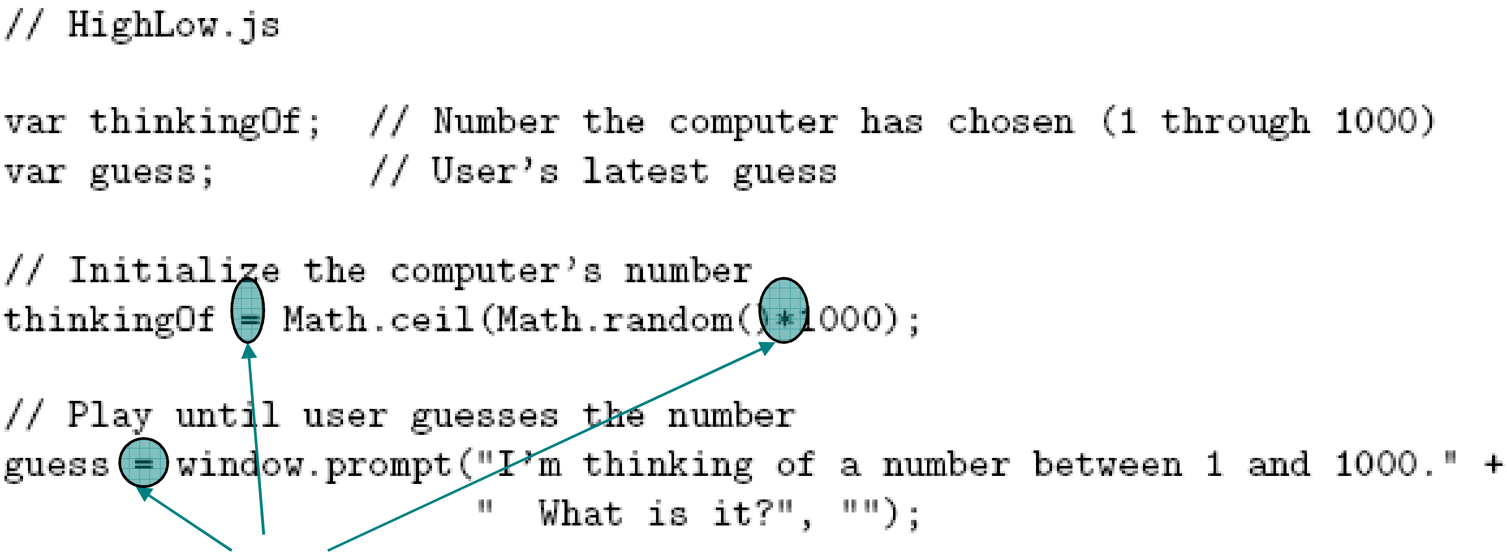
Basic JavaScript Syntax



```
// HighLow.js

var thinkingOf;  // Number the computer has chosen (1 through 1000)
var guess;       // User's latest guess

// Initialize the computer's number
thinkingOf = Math.ceil(Math.random()*1000);

// Play until user guesses the number
guess = window.prompt("I'm thinking of a number between 1 and 1000." +
                      "  What is it?", "");
```

Semi-colons are usually not required, but always allowed at statement end

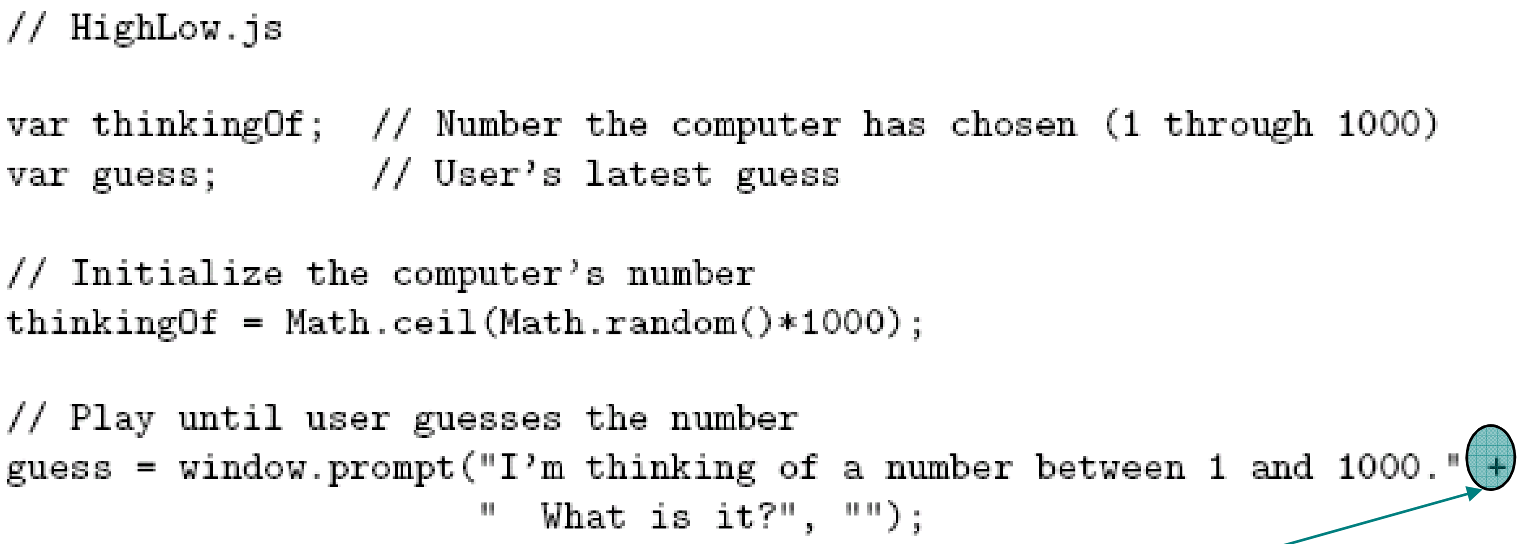
Basic JavaScript Syntax



```
// HighLow.js

var thinkingOf;  // Number the computer has chosen (1 through 1000)
var guess;       // User’s latest guess

// Initialize the computer’s number
thinkingOf = Math.ceil(Math.random()*1000);

// Play until user guesses the number
guess = window.prompt("I’m thinking of a number between 1 and 1000." +
                      "  What is it?", "");
```

Arithmetic operators same as Java/C++

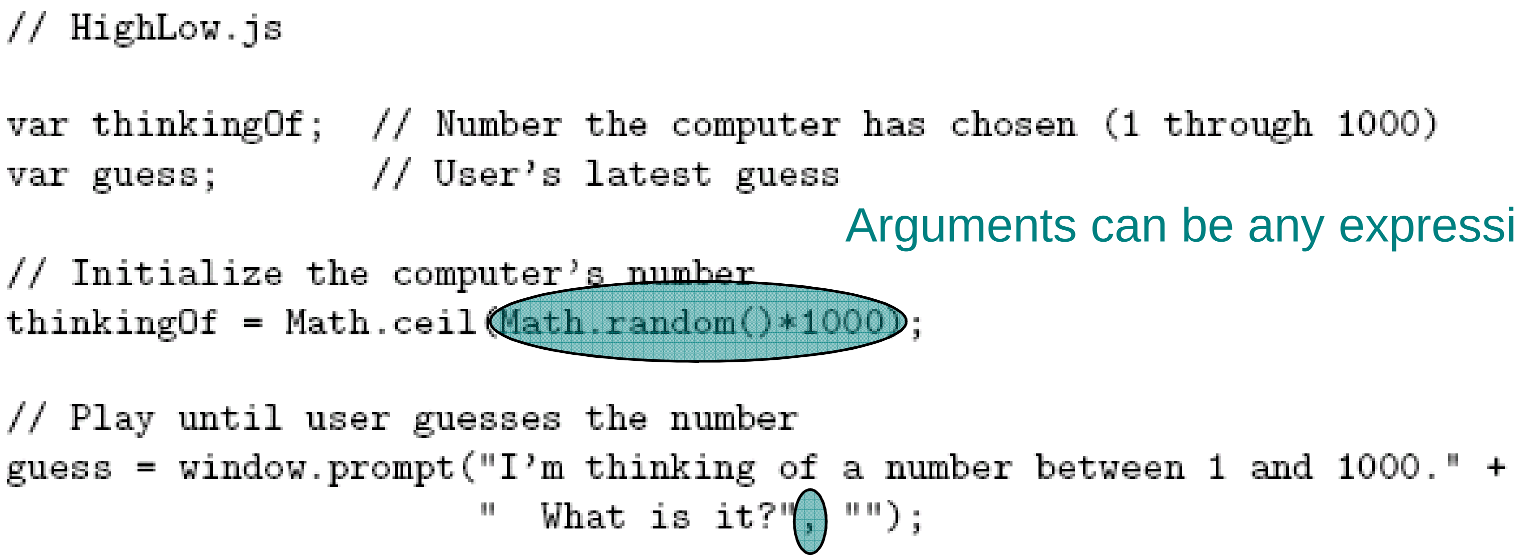
Basic JavaScript Syntax



```
// HighLow.js

var thinkingOf;  // Number the computer has chosen (1 through 1000)
var guess;       // User's latest guess

// Initialize the computer's number
thinkingOf = Math.ceil(Math.random()*1000);

// Play until user guesses the number
guess = window.prompt("I'm thinking of a number between 1 and 1000." +
                      "  What is it?", "");
```

String concatenation operator
as well as addition

Basic JavaScript Syntax

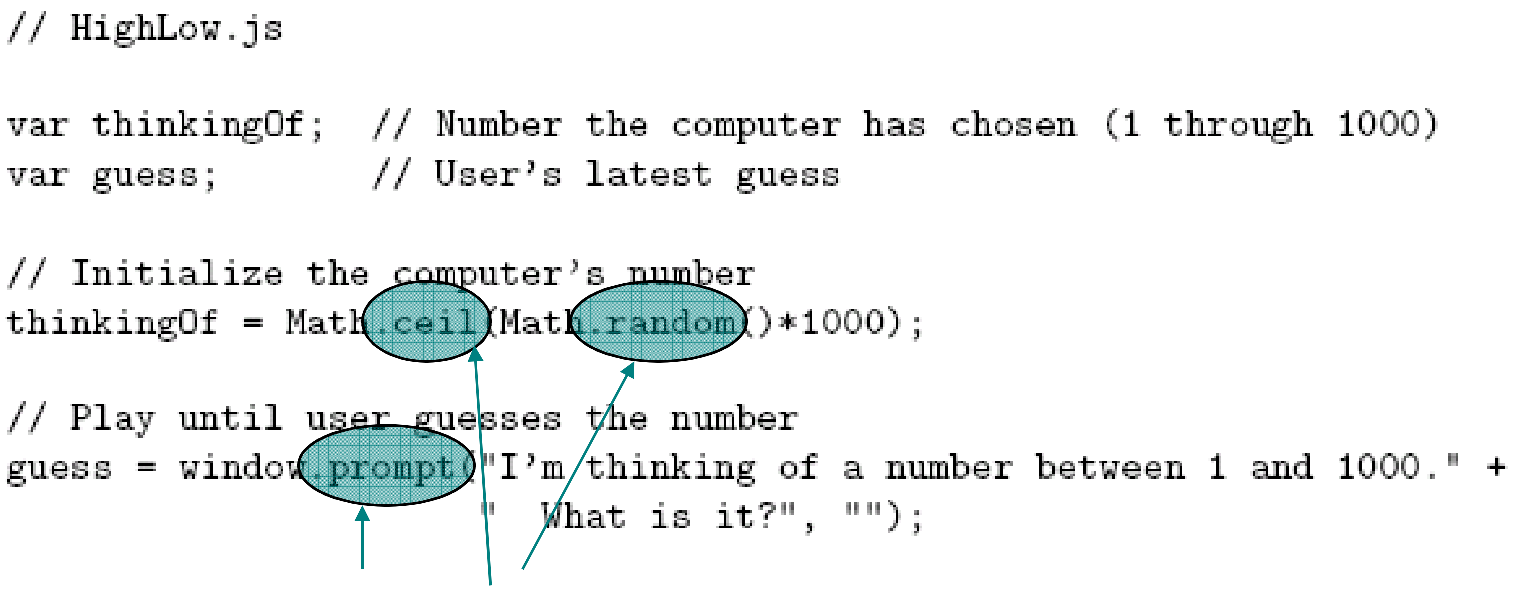


```
// HighLow.js

var thinkingOf;  // Number the computer has chosen (1 through 1000)
var guess;       // User’s latest guess

// Initialize the computer’s number
thinkingOf = Math.ceil(Math.random()*1000);

// Play until user guesses the number
guess = window.prompt("I’m thinking of a number between 1 and 1000." +
                      "  What is it?", "");
```

Arguments can be any expressions

Argument lists are comma-separated

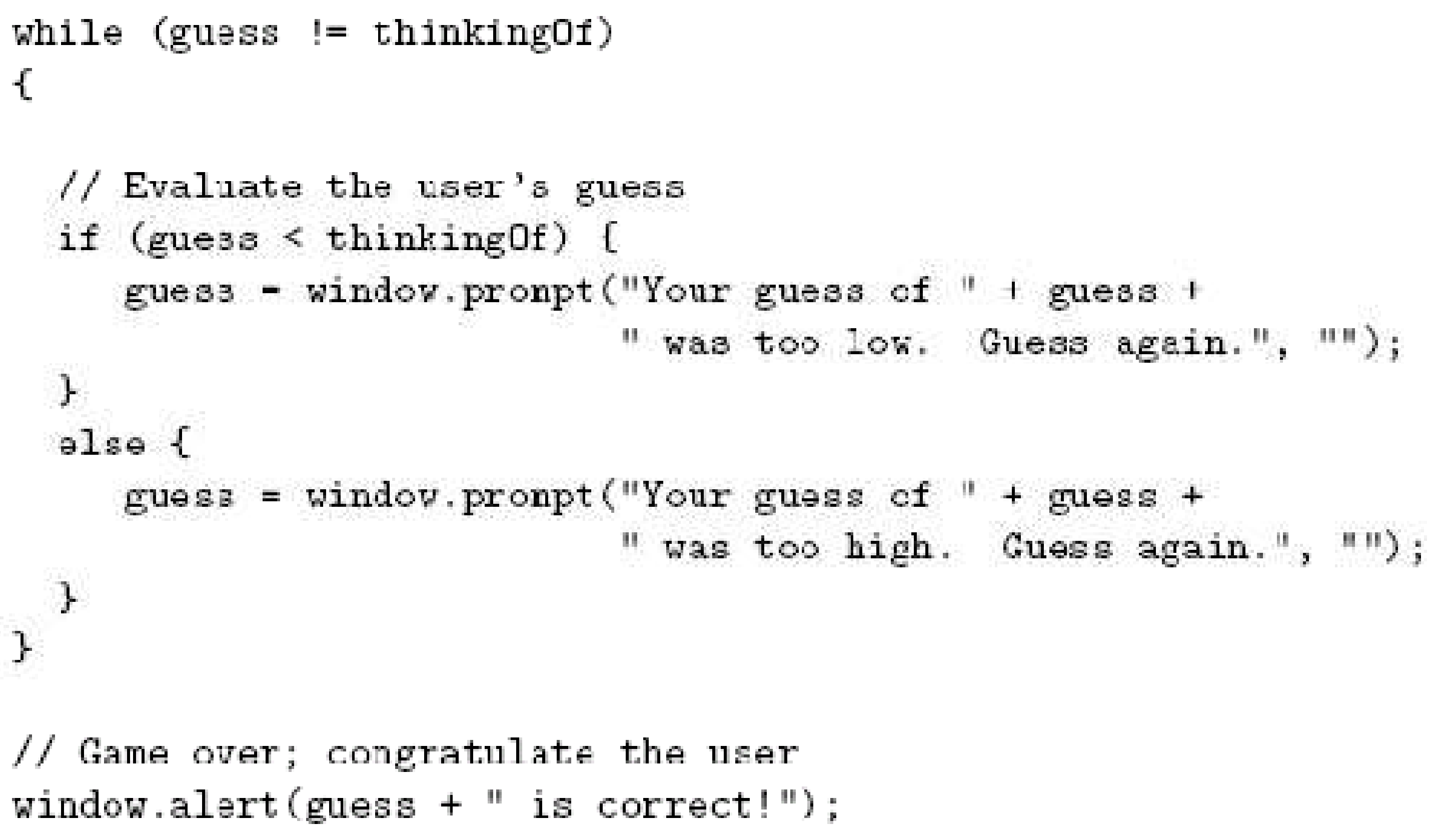
Basic JavaScript Syntax



```
// HighLow.js

var thinkingOf;  // Number the computer has chosen (1 through 1000)
var guess;       // User’s latest guess

// Initialize the computer’s number
thinkingOf = Math.ceil(Math.random()*1000);

// Play until user guesses the number
guess = window.prompt("I’m thinking of a number between 1 and 1000." +
                      "  What is it?", "");
```

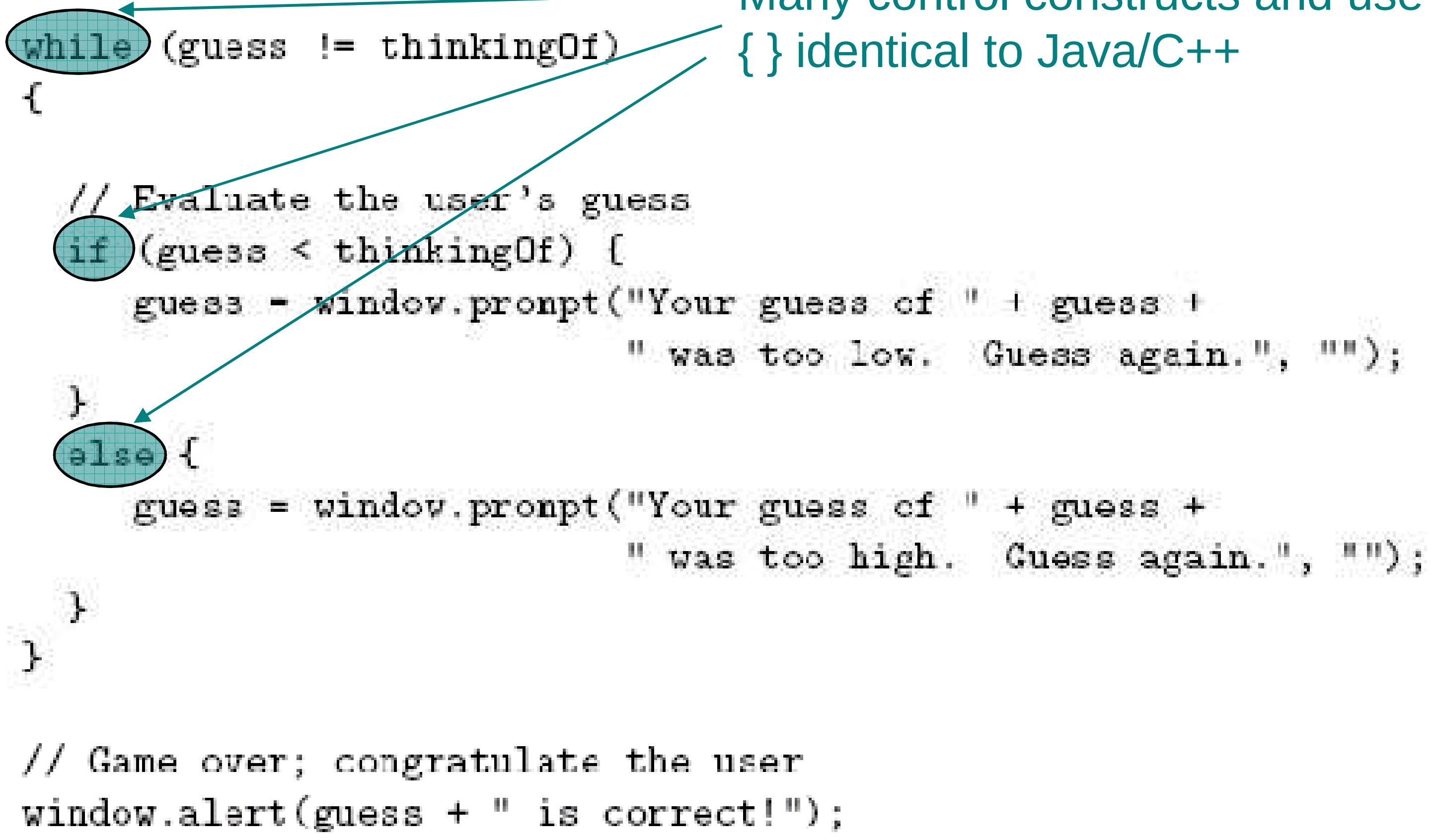
Object dot notation for method calls as in Java/C++

Basic JavaScript Syntax



```
while (guess != thinkingOf)
{

  // Evaluate the user's guess
  if (guess < thinkingOf) {
     guess = window.prompt("Your guess of " + guess +
                           " was too low.  Guess again.", "");
  }
  else {
     guess = window.prompt("Your guess of " + guess +
                           " was too high.  Guess again.", "");
  }
}

// Game over; congratulate the user
window.alert(guess + " is correct!");
```

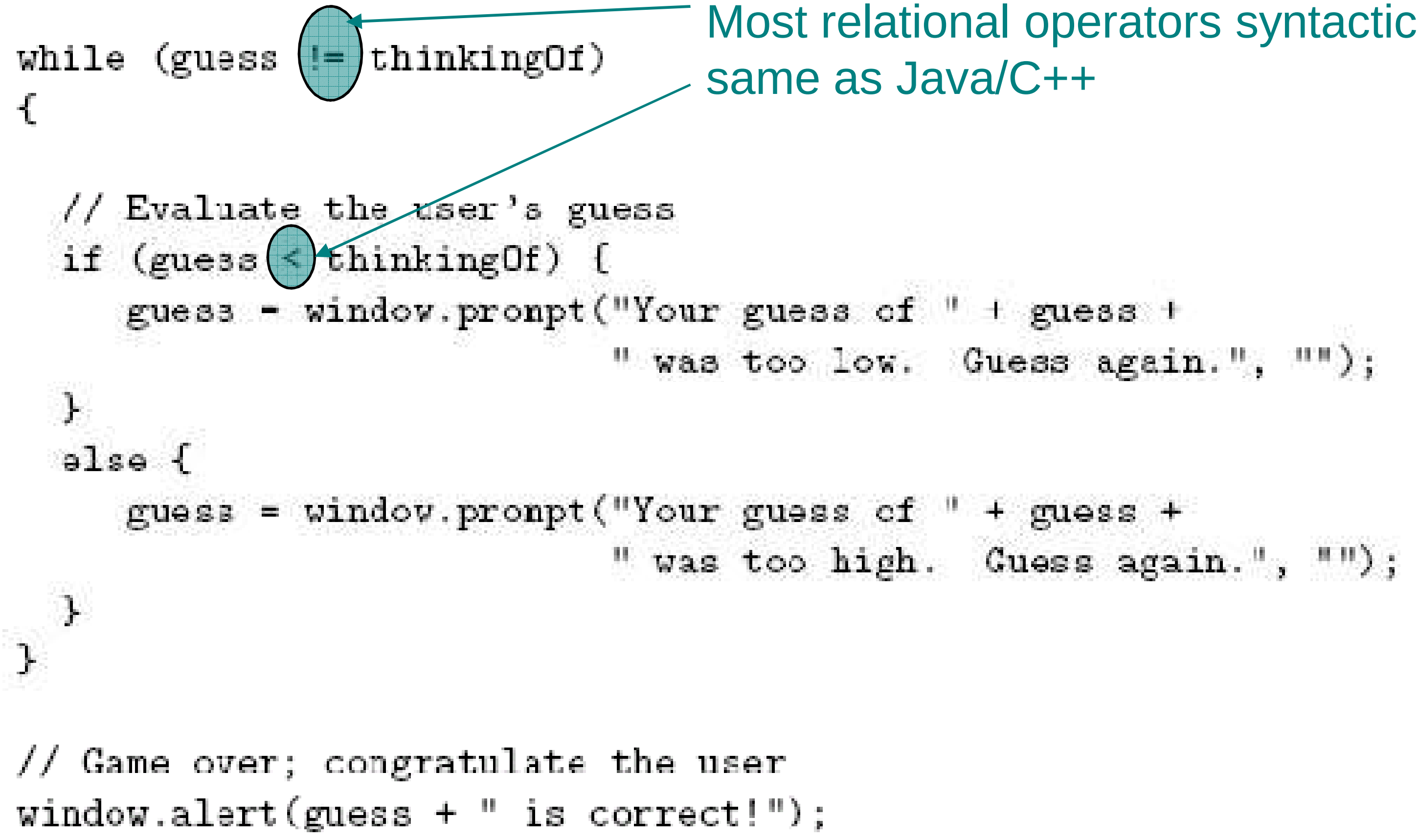
Basic JavaScript Syntax



```
while (guess != thinkingOf)
{

  // Evaluate the user's guess
  if (guess < thinkingOf) {
     guess = window.prompt("Your guess of " + guess +
                          " was too low.  Guess again.", "");
  }
  else {
     guess = window.prompt("Your guess of " + guess +
                          " was too high.  Guess again.", "");
  }
}

// Game over; congratulate the user
window.alert(guess + " is correct!");
```

Many control constructs and use of { } identical to Java/C++

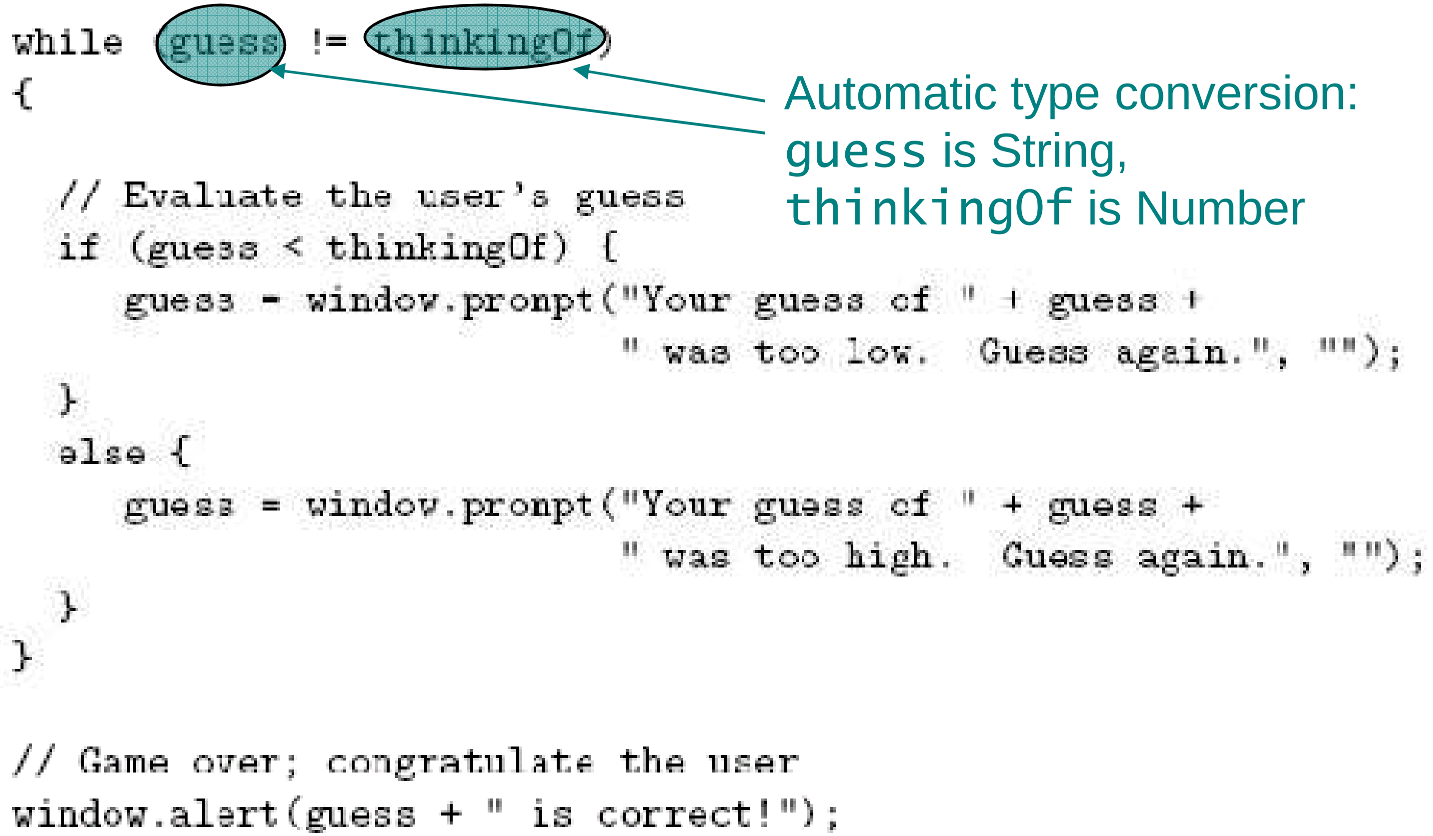
Basic JavaScript Syntax



```
while (guess != thinkingOf)
{

  // Evaluate the user's guess
  if (guess < thinkingOf) {
     guess = window.prompt("Your guess of " + guess +
                           " was too low.  Guess again.", "");
  }
  else {
     guess = window.prompt("Your guess of " + guess +
                           " was too high.  Guess again.", "");
  }
}

// Game over; congratulate the user
window.alert(guess + " is correct!");
```

Most relational operators syntactically same as Java/C++

Basic JavaScript Syntax

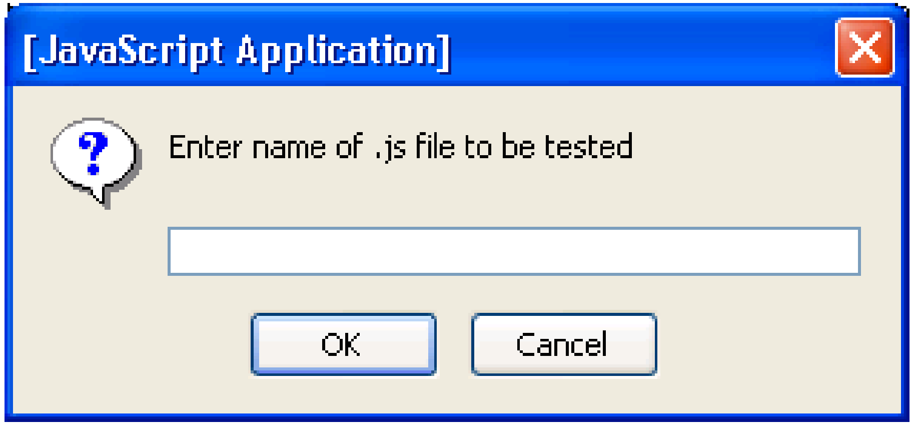


```
while (guess != thinkingOf)
{

  // Evaluate the user's guess
  if (guess < thinkingOf) {
     guess = window.prompt("Your guess of " + guess +
                           " was too low.  Guess again.", "");
  }
  else {
     guess = window.prompt("Your guess of " + guess +
                           " was too high.  Guess again.", "");
  }
}

// Game over; congratulate the user
window.alert(guess + " is correct!");
```

Automatic type conversion:
`guess` is String,
`thinkingOf` is Number

Running Examples

- Browse to `TestJs.html` in examples download package
- Enter name of .js file (*e.g.*, `HighLow.js`) in prompt box:

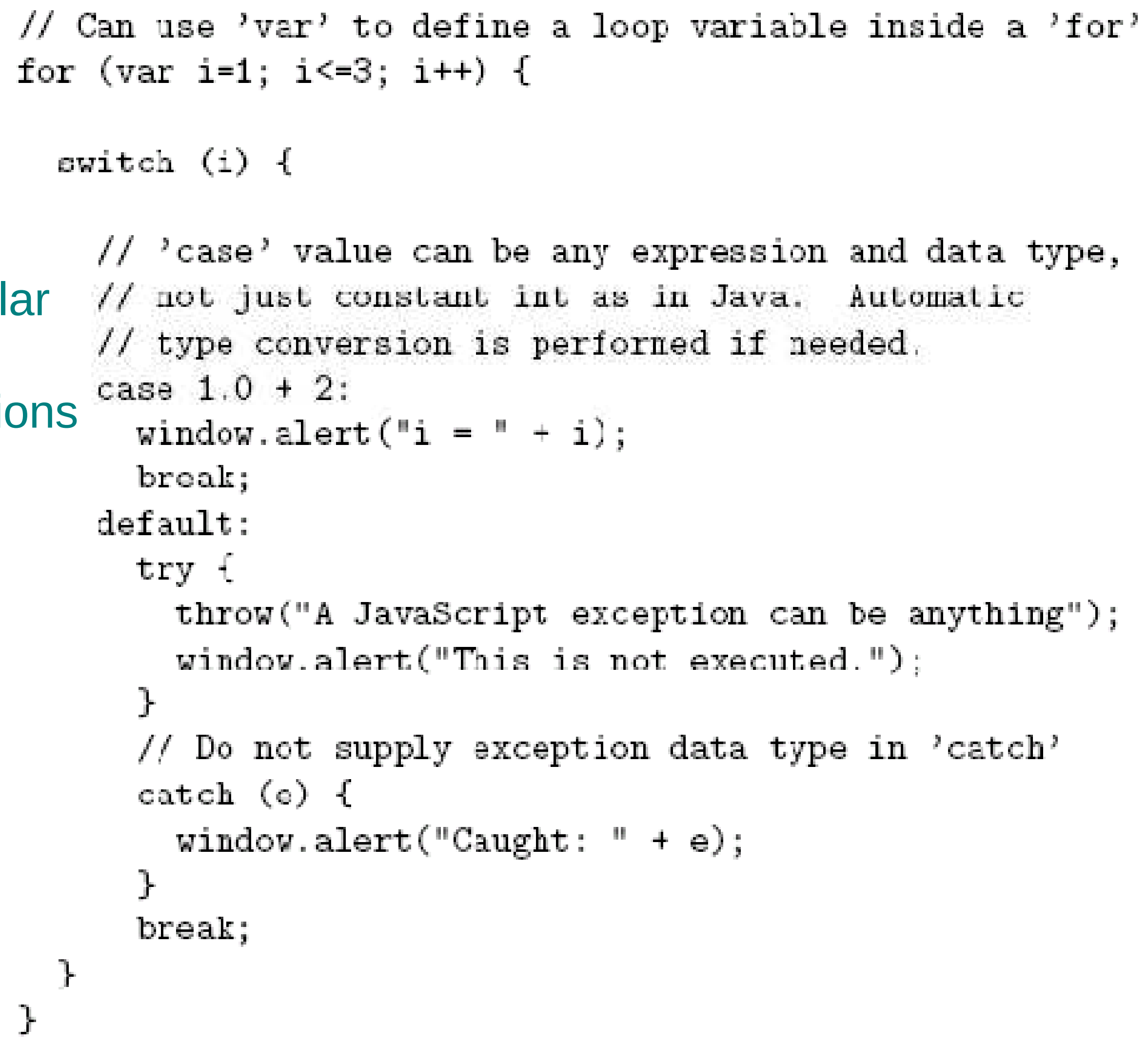


[JavaScript Application]

Enter name of .js file to be tested

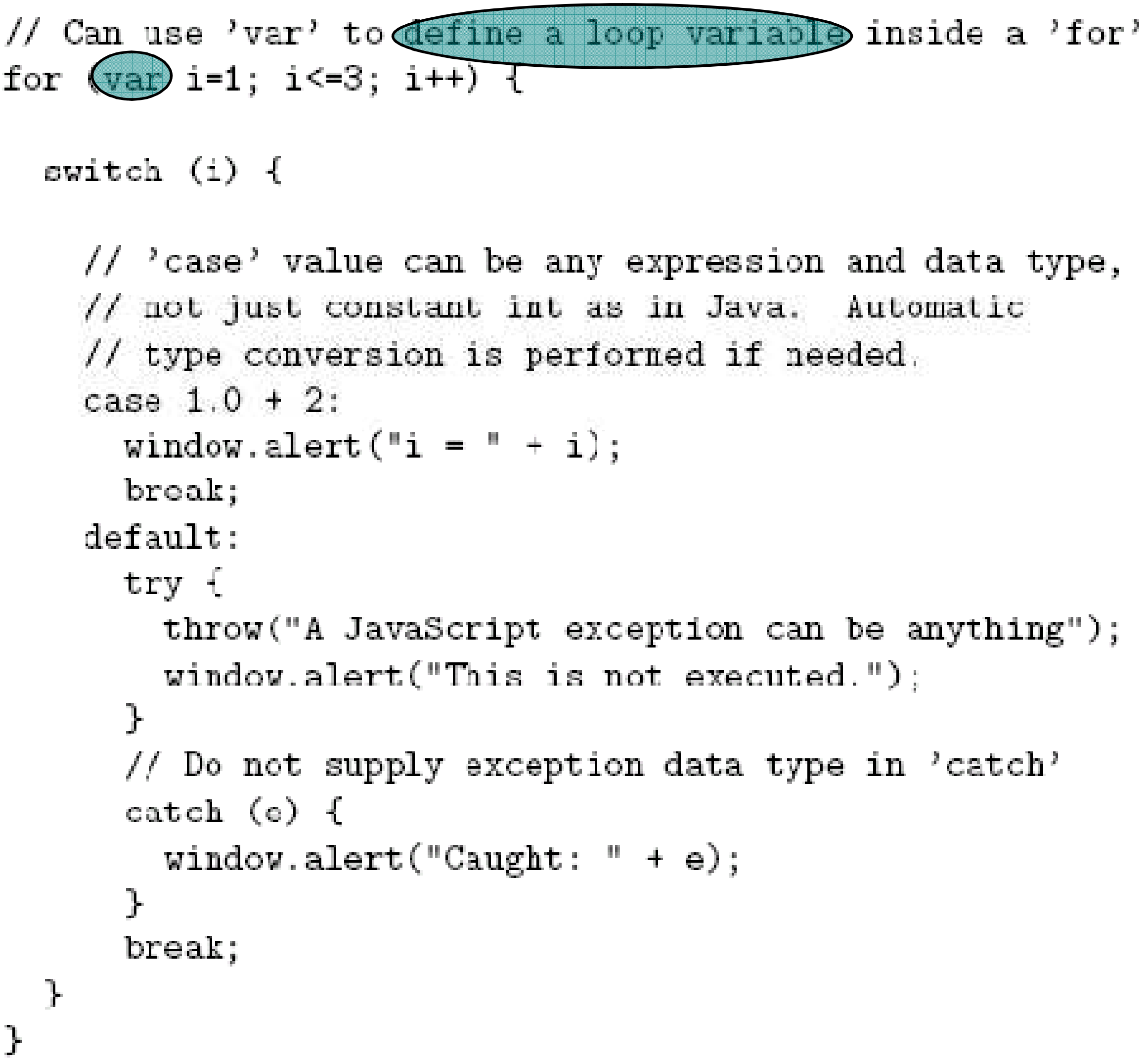
JavaScript Statements

JavaScript keyword statements are very similar to Java with small exceptions



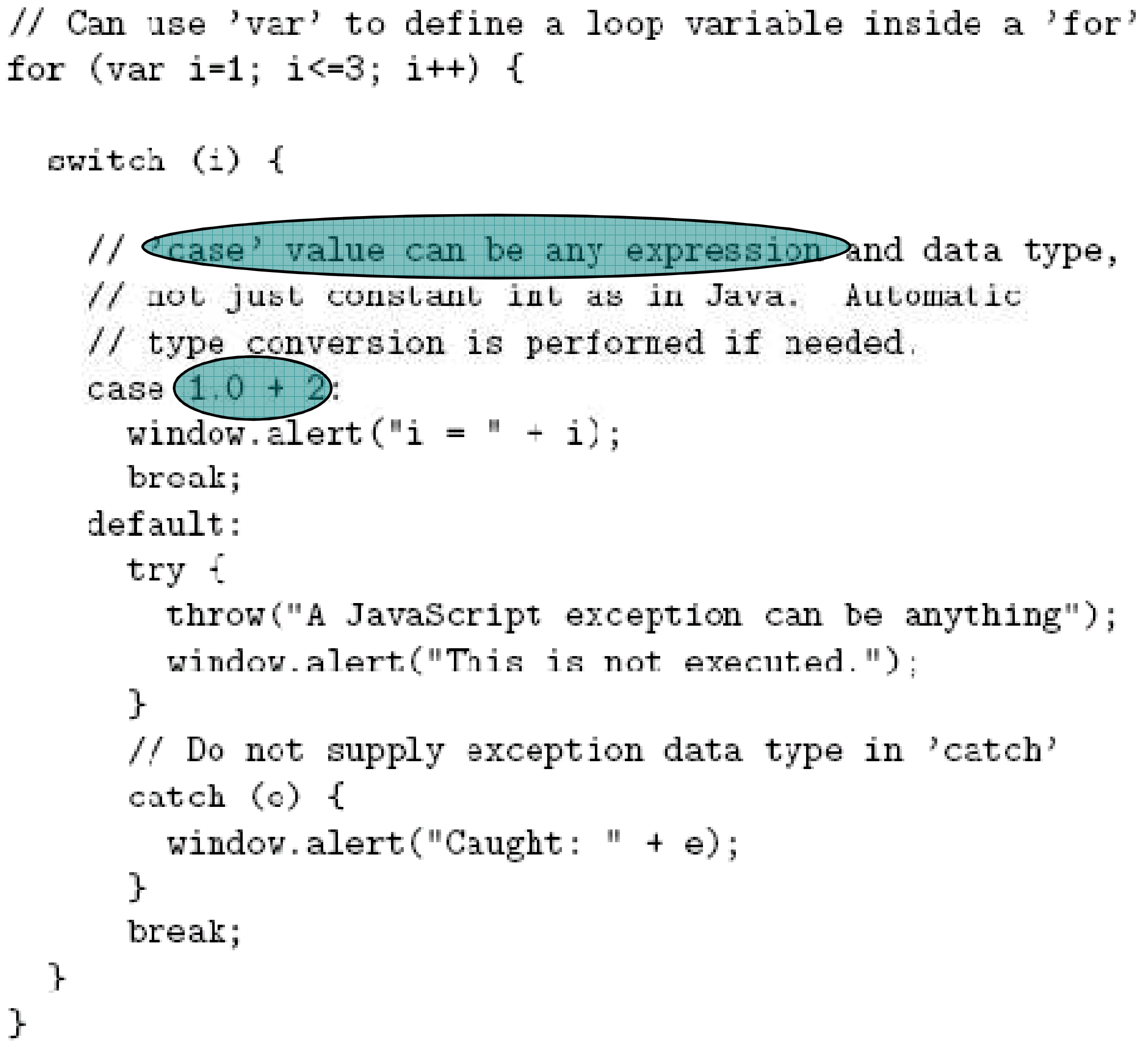
```
// Can use 'var' to define a loop variable inside a 'for'
for (var i=1; i<=3; i++) {

  switch (i) {

    // 'case' value can be any expression and data type,
    // not just constant int as in Java.  Automatic
    // type conversion is performed if needed.
    case 1.0 + 2:
      window.alert("i = " + i);
      break;
    default:
      try {
        throw("A JavaScript exception can be anything");
        window.alert("This is not executed.");
      }
      // Do not supply exception data type in 'catch'
      catch (e) {
        window.alert("Caught: " + e);
      }
      break;
  }
}
```

JavaScript Statements



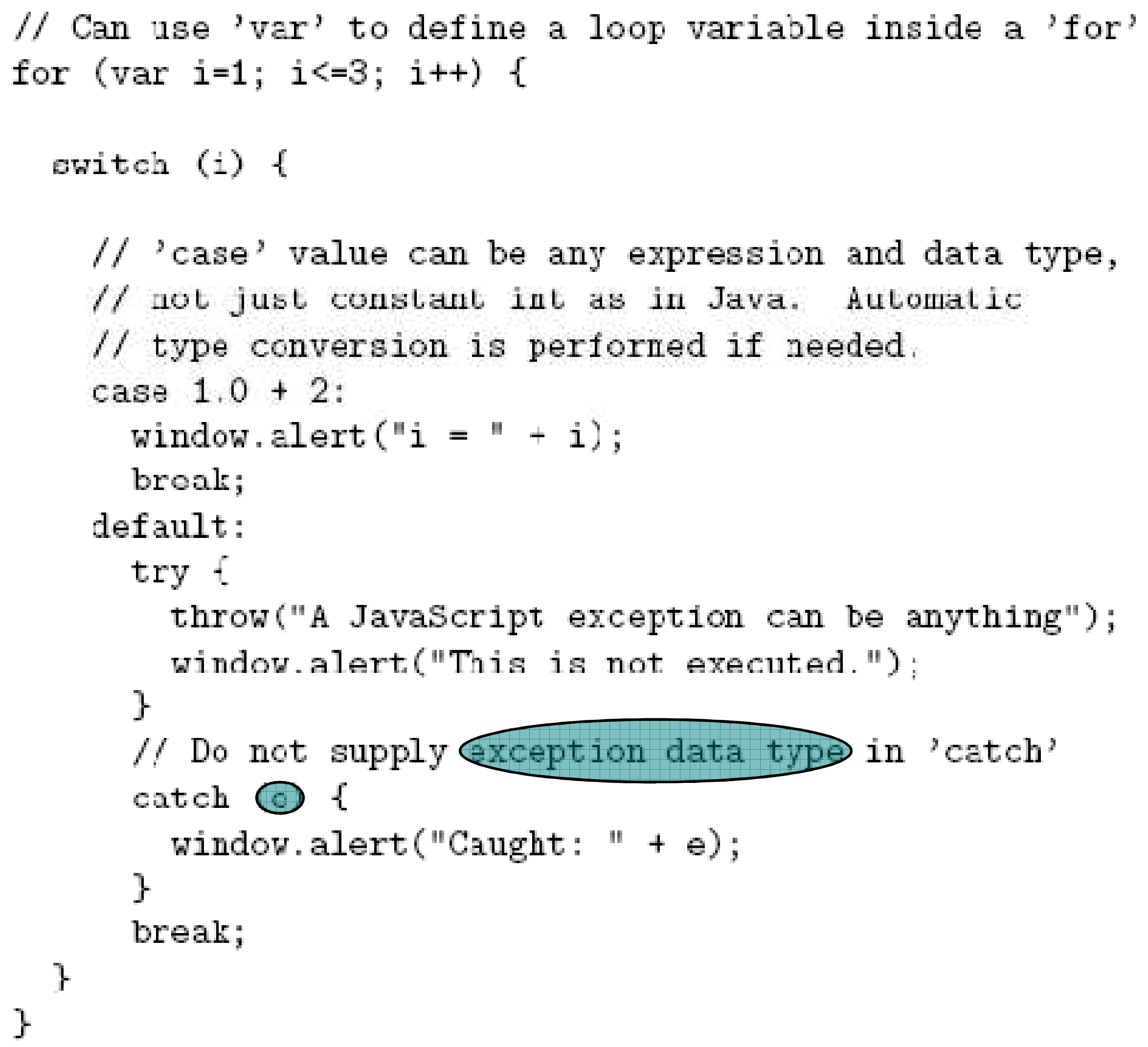
```
// Can use ’var’ to define a loop variable inside a ’for’
for (var i=1; i<=3; i++) {

  switch (i) {

    // ’case’ value can be any expression and data type,
    // not just constant int as in Java.  Automatic
    // type conversion is performed if needed.
    case 1.0 + 2:
      window.alert("i = " + i);
      break;
    default:
      try {
        throw("A JavaScript exception can be anything");
        window.alert("This is not executed.");
      }
      // Do not supply exception data type in ’catch’
      catch (e) {
        window.alert("Caught: " + e);
      }
      break;
  }
}
```

JavaScript Statements



```
// Can use 'var' to define a loop variable inside a 'for'
for (var i=1; i<=3; i++) {

  switch (i) {

    // 'case' value can be any expression and data type,
    // not just constant int as in Java.  Automatic
    // type conversion is performed if needed.
    case 1.0 + 2:
      window.alert("i = " + i);
      break;
    default:
      try {
        throw("A JavaScript exception can be anything");
        window.alert("This is not executed.");
      }
      // Do not supply exception data type in 'catch'
      catch (e) {
        window.alert("Caught: " + e);
      }
      break;
  }
}
```

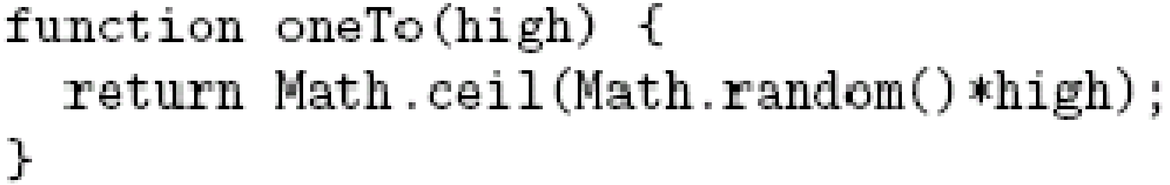
JavaScript Statements



```
// Can use 'var' to define a loop variable inside a 'for'
for (var i=1; i<=3; i++) {

  switch (i) {

    // 'case' value can be any expression and data type,
    // not just constant int as in Java.  Automatic
    // type conversion is performed if needed.
    case 1.0 + 2:
      window.alert("i = " + i);
      break;
    default:
      try {
        throw("A JavaScript exception can be anything");
        window.alert("This is not executed.");
      }
      // Do not supply exception data type in 'catch'
      catch (e) {
        window.alert("Caught: " + e);
      }
      break;
  }
}
```

JavaScript Functions

- Function declaration syntax

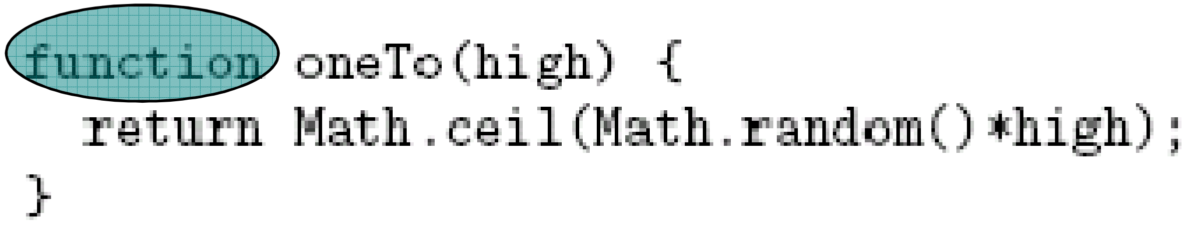


```
function oneTo(high) {
  return Math.ceil(Math.random()*high);
}
```

JavaScript Functions

- Function declaration syntax

Declaration always begins with keyword `function`, no return type

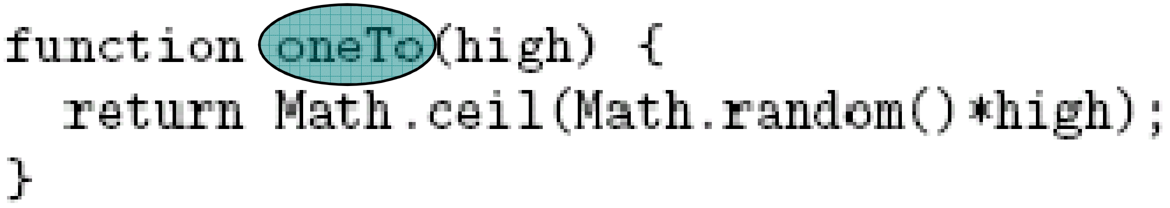


```
function oneTo(high) {
  return Math.ceil(Math.random()*high);
}
```

JavaScript Functions

- Function declaration syntax

Identifier representing function’s *name*

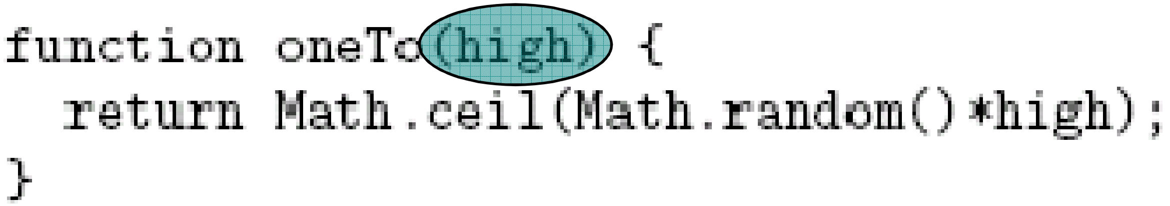


```
function oneTo(high) {
  return Math.ceil(Math.random()*high);
}
```

JavaScript Functions

- Function declaration syntax

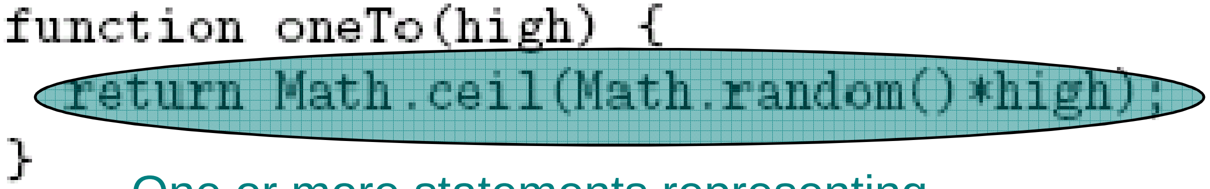
Formal parameter list



```
function oneTo(high) {
  return Math.ceil(Math.random()*high);
}
```

JavaScript Functions

- Function declaration syntax

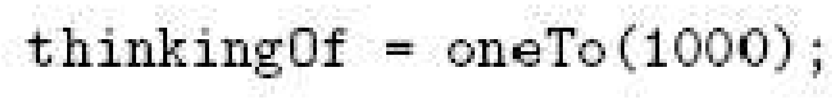


```
function oneTo(high) {
  return Math.ceil(Math.random()*high);
}
```

One or more statements representing *function body*

JavaScript Functions

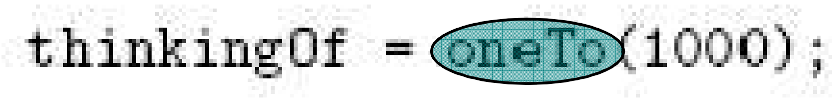
- Function call syntax



```
thinkingOf = oneTo(1000);
```

JavaScript Functions

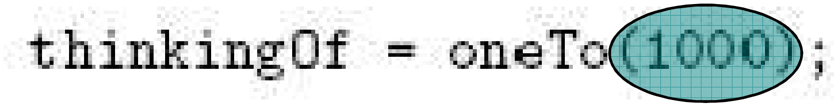
- Function call syntax



```
thinkingOf = oneTo(1000);
```

JavaScript Functions

- Function call syntax

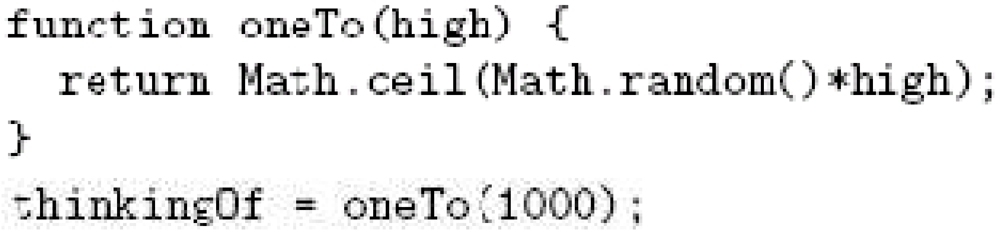


```
thinkingOf = oneTo(1000);
```

Argument list

JavaScript Functions

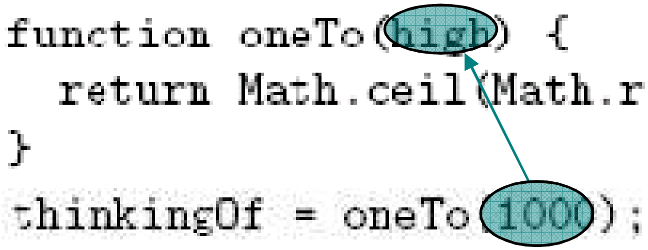
- Function call semantics:



```
function oneTo(high) {
  return Math.ceil(Math.random()*high);
}
thinkingOf = oneTo(1000);
```

JavaScript Functions

- Function call semantics:



```
function oneTo(high) {
  return Math.ceil(Math.random()*high);
}
thinkingOf = oneTo(1000);
```

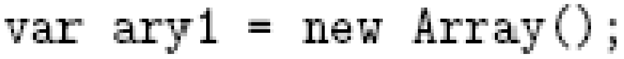
Argument value(s)
associated with corresponding
formal parameters

JavaScript Functions

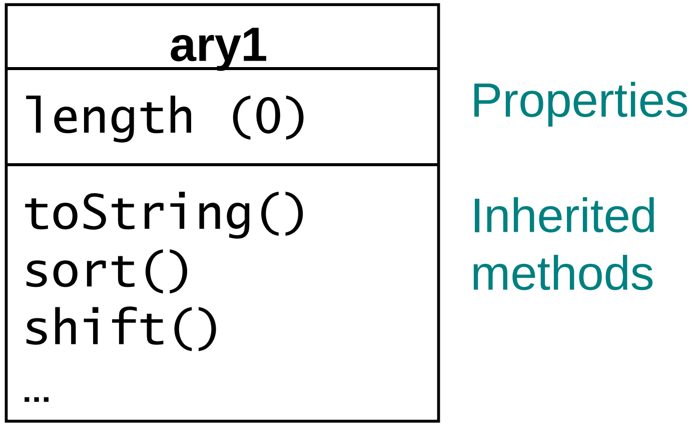
- Number mismatch between argument list and formal parameter list:
  - More arguments: excess ignored
  - Fewer arguments: remaining parameters are Undefined

JavaScript Arrays

- The `Array` built-in object can be used to construct objects with special properties and that inherit various methods

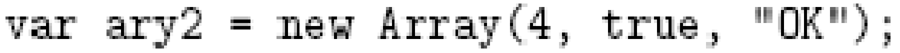


```
var ary1 = new Array();
```

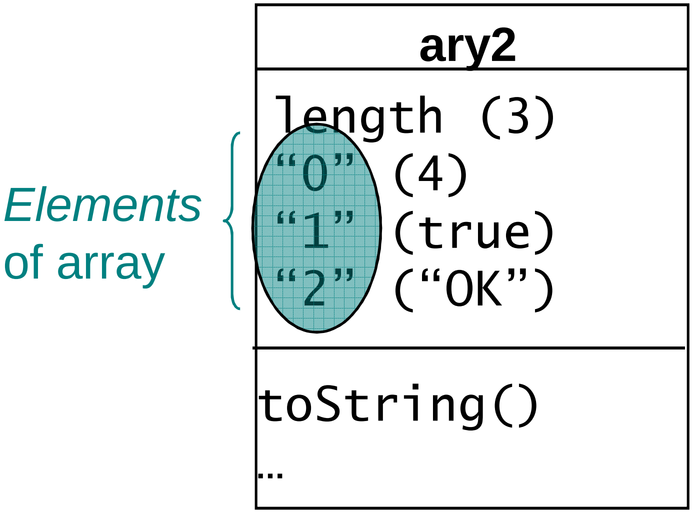


JavaScript Arrays

- The `Array` built-in object can be used to construct objects with special properties and that inherit various methods



```
var ary2 = new Array(4, true, "OK");
```



ary2

```
length (3)
"0" (4)
"1" (true)
"2" ("OK")
toString()
...
```

Elements of array

Accessing array elements:

✓ `ary2[1]`

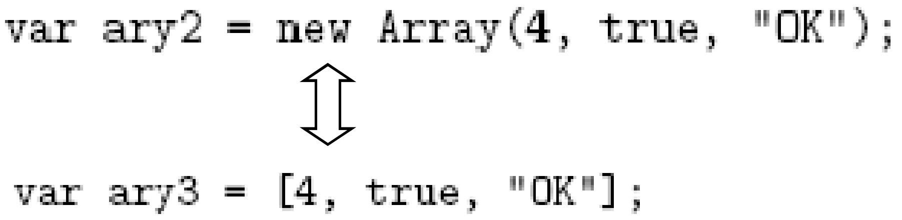
✓ `ary2["1"]`

✗ `ary2.1`

Must follow identifier syntax rules

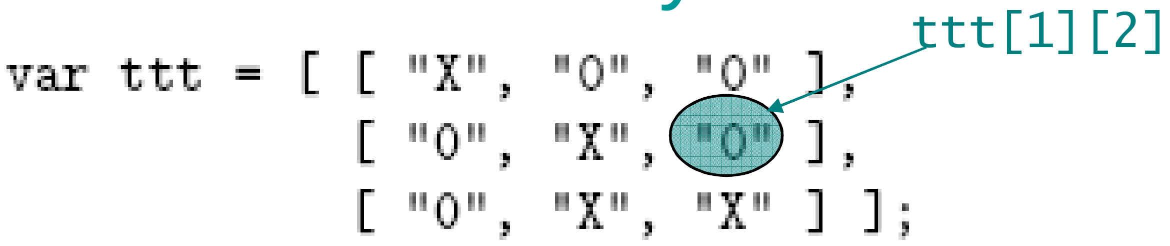
JavaScript Arrays

- The `Array` constructor is indirectly called if an array initializer is used



```
var ary2 = new Array(4, true, "OK");
var ary3 = [4, true, "OK"];
```

- Array initializiers can be used to create multidimensional arrays

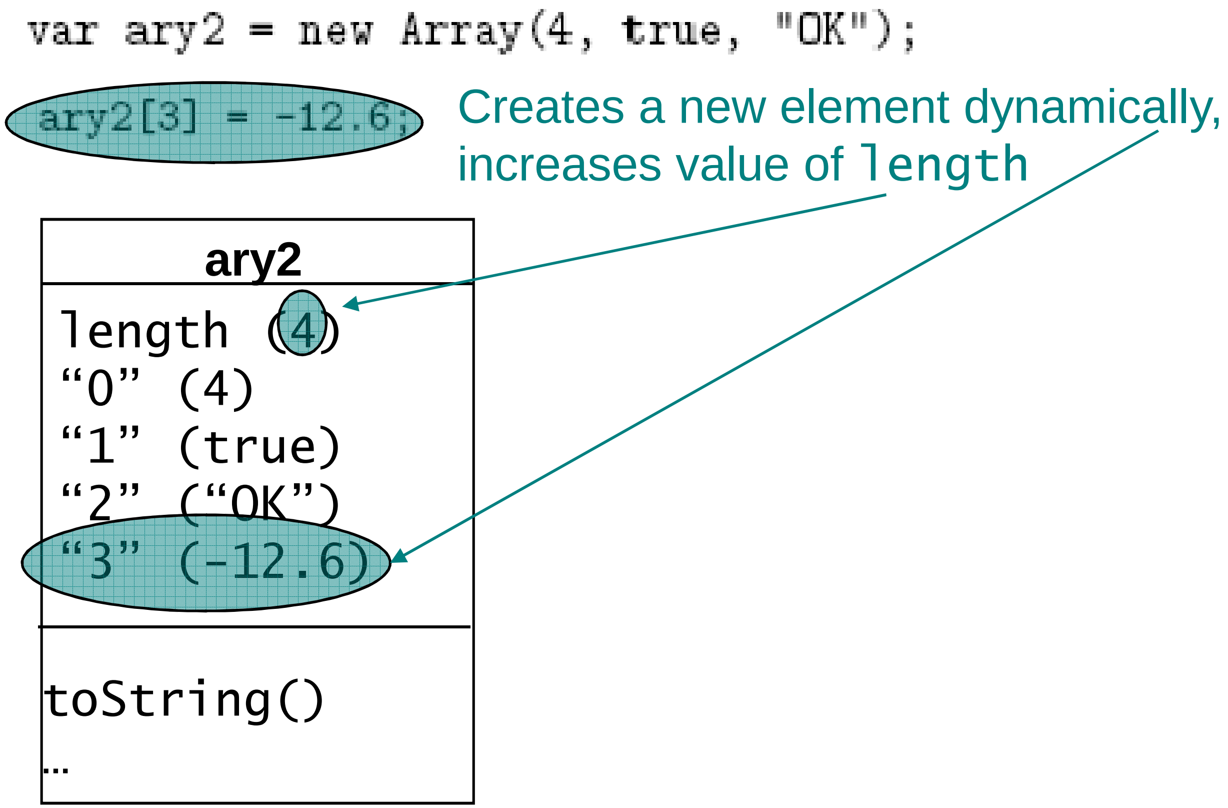


```
var ttt = [ [ "X", "O", "O" ],
            [ "O", "X", "O" ],
            [ "O", "X", "X" ] ];
```

ttt[1][2]

JavaScript Arrays

- Changing the number of elements:



```
var ary2 = new Array(4, true, "OK");
ary2[3] = -12.6;
```

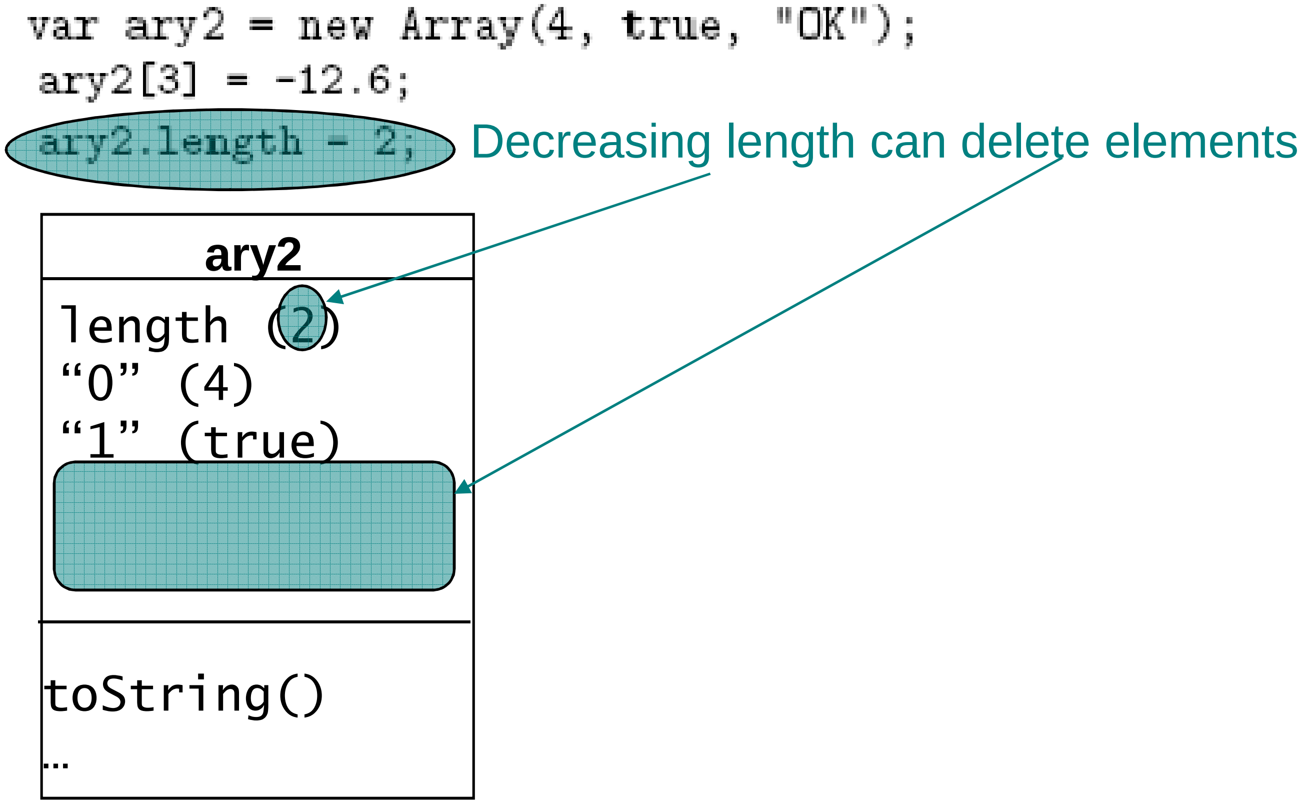
Creates a new element dynamically, increases value of `length`

ary2

```
length (4)
“0” (4)
“1” (true)
“2” (“OK”)
“3” (-12.6)

toString()
…
```

JavaScript Arrays

- Changing the number of elements:

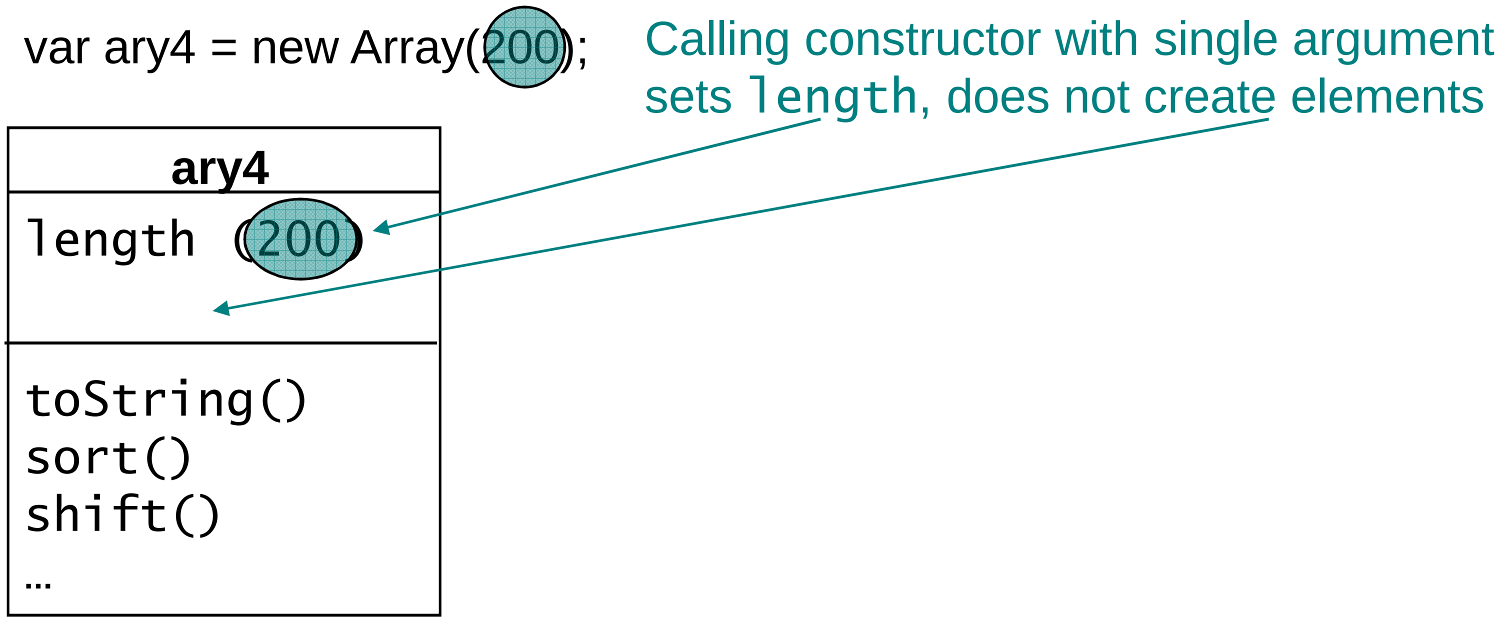


```
var ary2 = new Array(4, true, "OK");
ary2[3] = -12.6;
ary2.length = 2;
```

Decreasing length can delete elements

JavaScript Arrays

- Value of `length` is not necessarily the same as the actual number of elements



```
var ary4 = new Array(200);
```

Calling constructor with single argument sets `length`, does not create elements

JavaScript Arrays

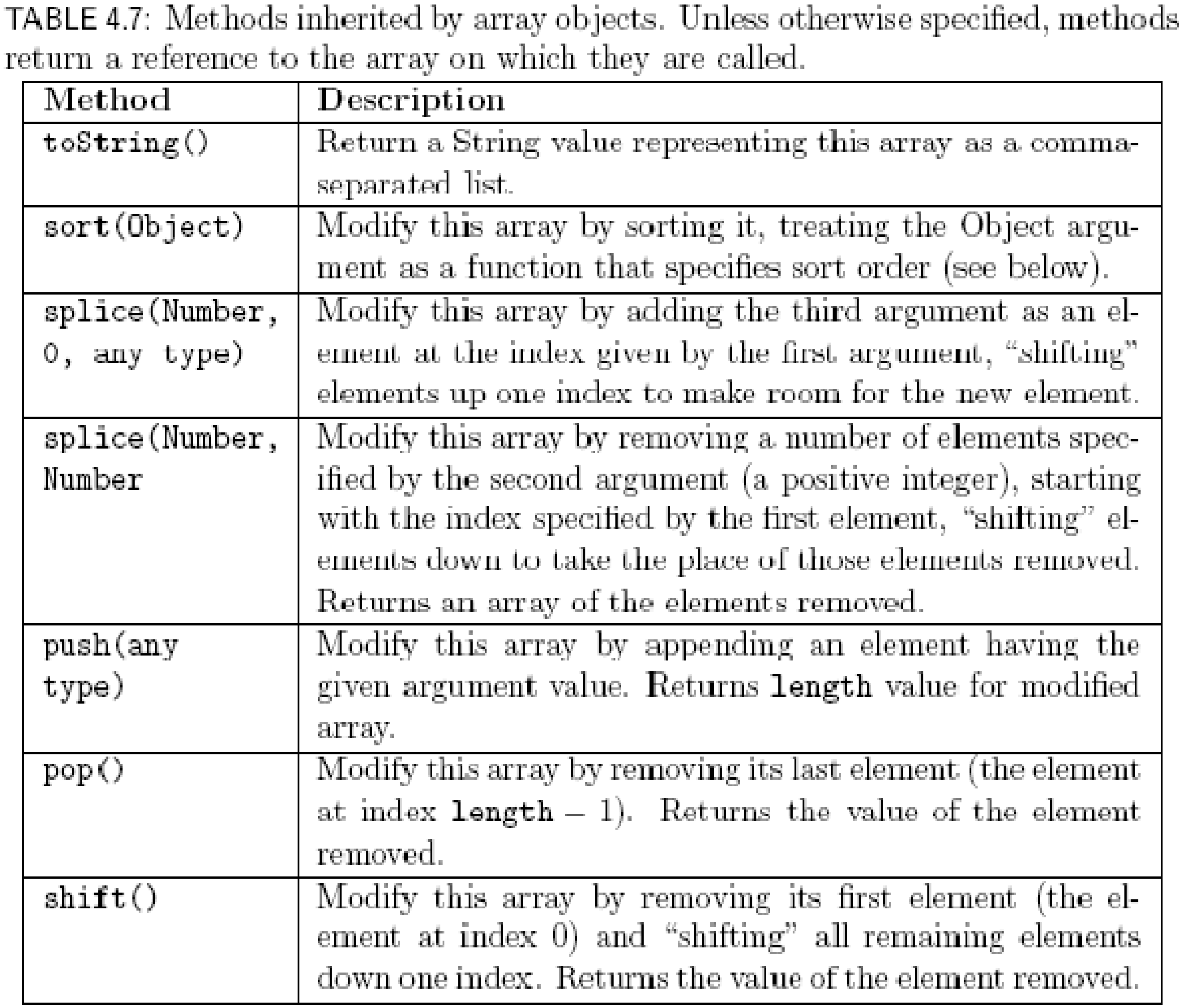
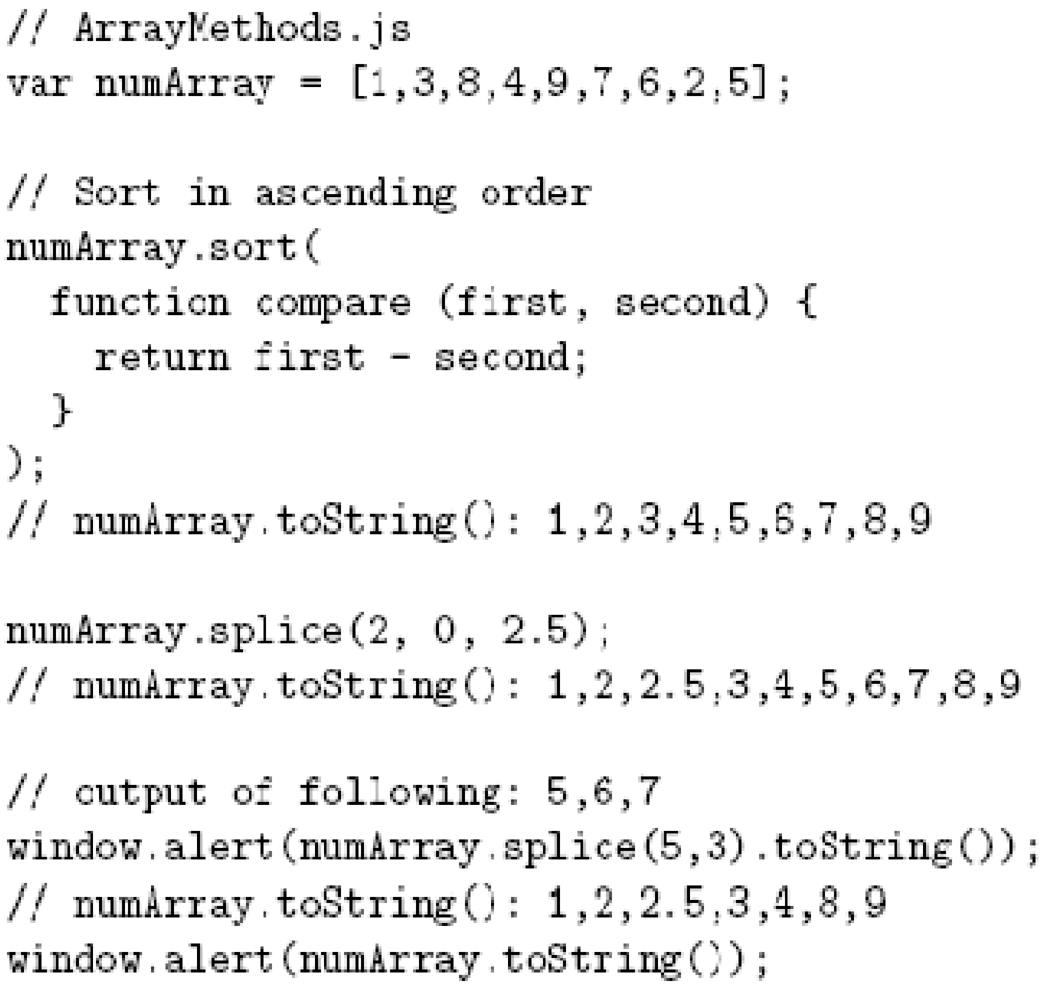


TABLE 4.7: Methods inherited by array objects. Unless otherwise specified, methods return a reference to the array on which they are called.

Method	Description
`toString()`	Return a String value representing this array as a comma-separated list.
`sort(Object)`	Modify this array by sorting it, treating the Object argument as a function that specifies sort order (see below).
`splice(Number, 0, any type)`	Modify this array by adding the third argument as an element at the index given by the first argument, "shifting" elements up one index to make room for the new element.
`splice(Number, Number`	Modify this array by removing a number of elements specified by the second argument (a positive integer), starting with the index specified by the first element, "shifting" elements down to take the place of those elements removed. Returns an array of the elements removed.
`push(any type)`	Modify this array by appending an element having the given argument value. Returns `length` value for modified array.
`pop()`	Modify this array by removing its last element (the element at index `length` – 1). Returns the value of the element removed.
`shift()`	Modify this array by removing its first element (the element at index 0) and "shifting" all remaining elements down one index. Returns the value of the element removed.

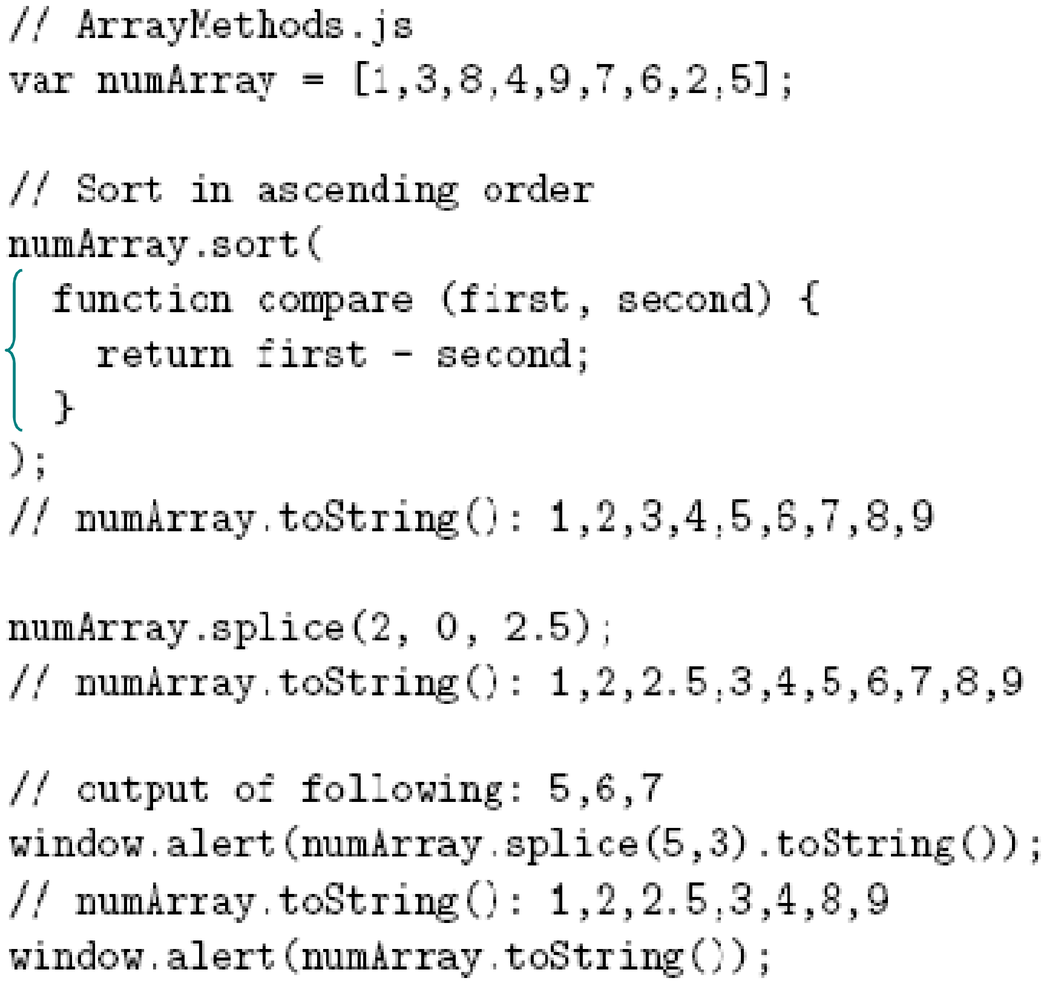
JavaScript Arrays



```
// ArrayMethods.js
var numArray = [1,3,8,4,9,7,6,2,5];

// Sort in ascending order
numArray.sort(
  function compare (first, second) {
    return first - second;
  }
);
// numArray.toString(): 1,2,3,4,5,6,7,8,9

numArray.splice(2, 0, 2.5);
// numArray.toString(): 1,2,2.5,3,4,5,6,7,8,9

// output of following: 5,6,7
window.alert(numArray.splice(5,3).toString());
// numArray.toString(): 1,2,2.5,3,4,8,9
window.alert(numArray.toString());
```

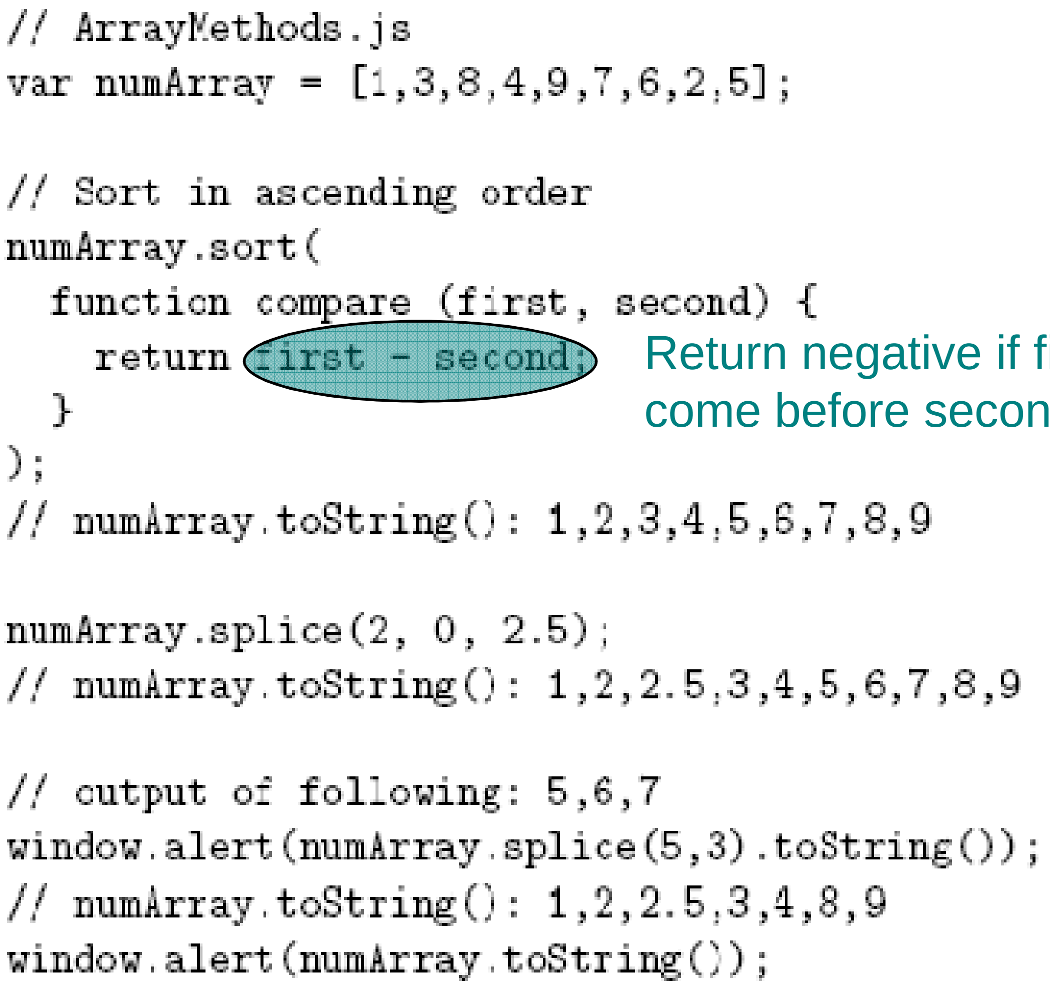
JavaScript Arrays

Argument to sort is a function



```
// ArrayMethods.js
var numArray = [1,3,8,4,9,7,6,2,5];

// Sort in ascending order
numArray.sort(
  function compare (first, second) {
    return first - second;
  }
);
// numArray.toString(): 1,2,3,4,5,6,7,8,9

numArray.splice(2, 0, 2.5);
// numArray.toString(): 1,2,2.5,3,4,5,6,7,8,9

// output of following: 5,6,7
window.alert(numArray.splice(5,3).toString());
// numArray.toString(): 1,2,2.5,3,4,8,9
window.alert(numArray.toString());
```

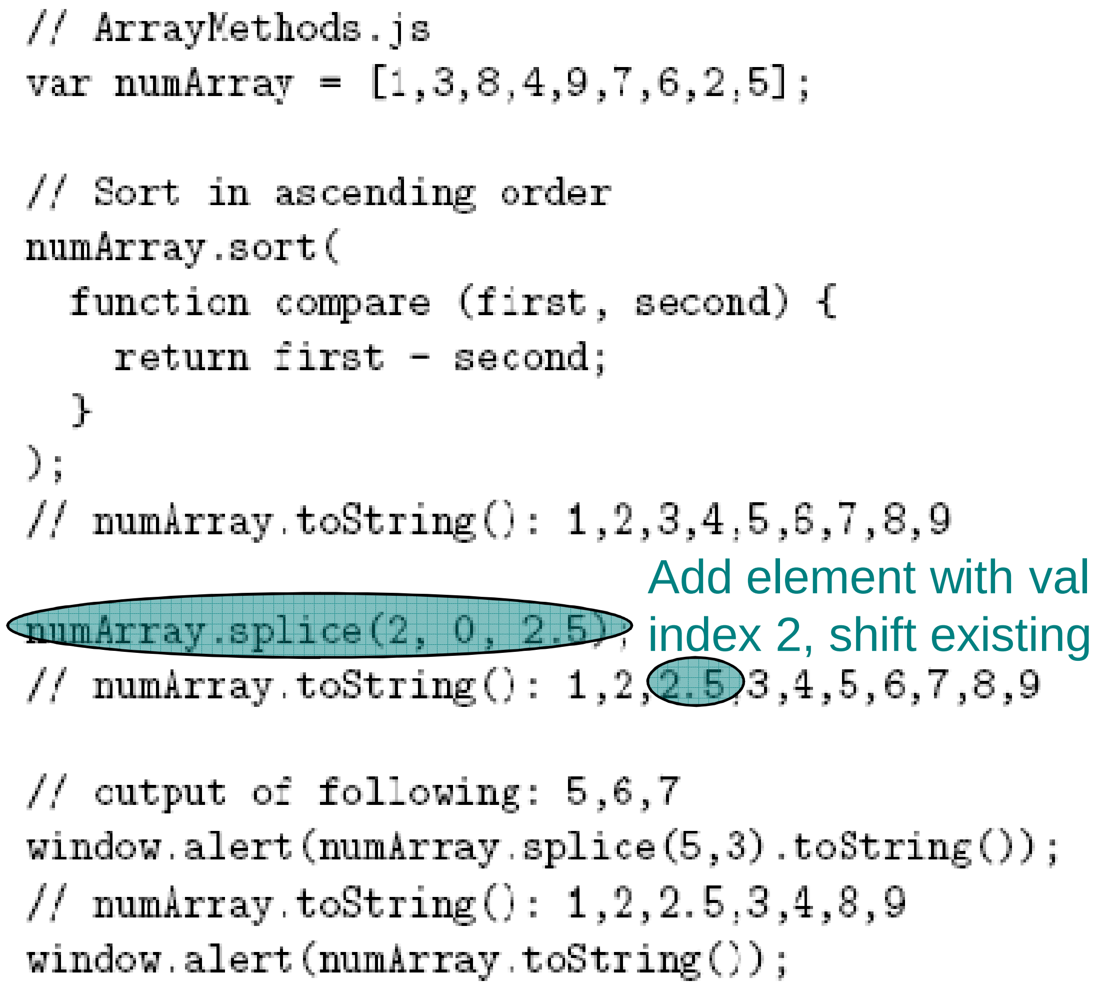
JavaScript Arrays



```
// ArrayMethods.js
var numArray = [1,3,8,4,9,7,6,2,5];

// Sort in ascending order
numArray.sort(
  function compare (first, second) {
    return first - second;
  }
);
// numArray.toString(): 1,2,3,4,5,6,7,8,9

numArray.splice(2, 0, 2.5);
// numArray.toString(): 1,2,2.5,3,4,5,6,7,8,9

// output of following: 5,6,7
window.alert(numArray.splice(5,3).toString());
// numArray.toString(): 1,2,2.5,3,4,8,9
window.alert(numArray.toString());
```

Return negative if first value should come before second after sorting

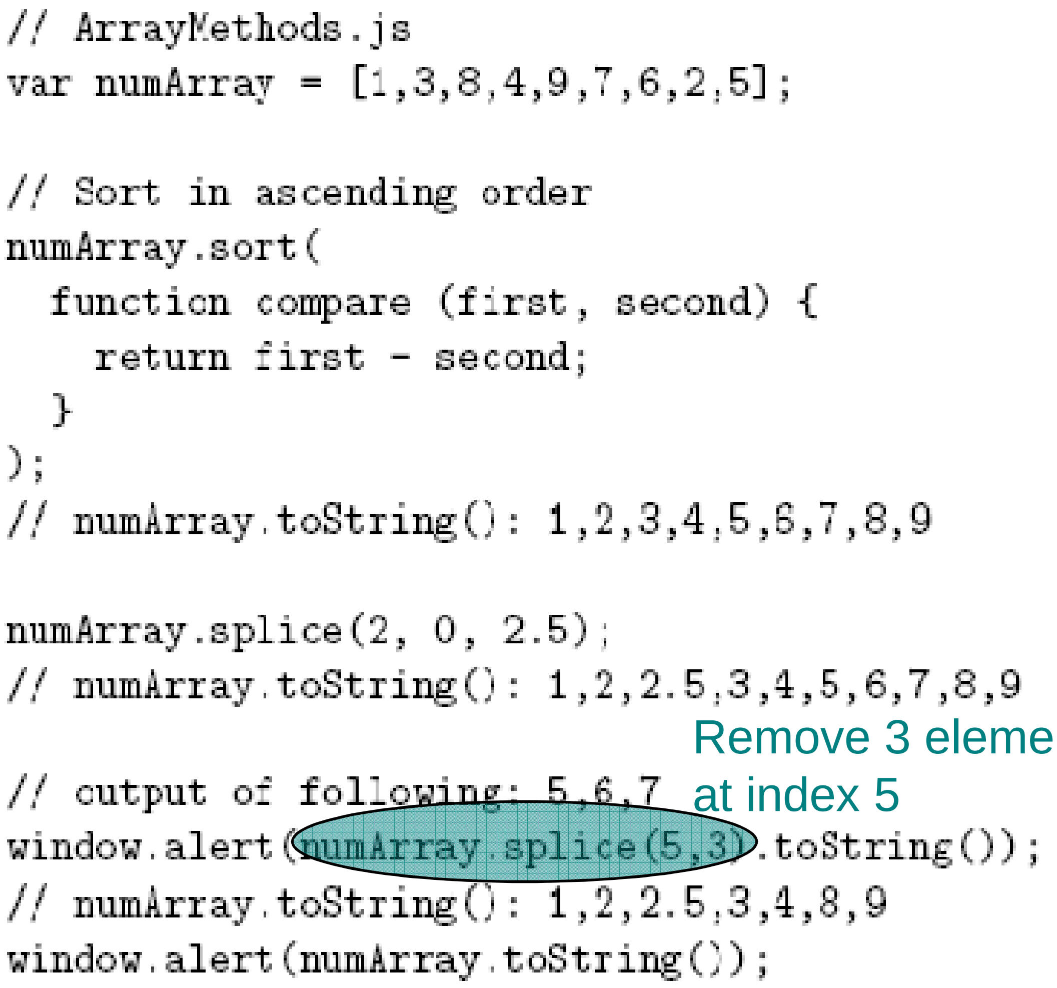
JavaScript Arrays



```
// ArrayMethods.js
var numArray = [1,3,8,4,9,7,6,2,5];

// Sort in ascending order
numArray.sort(
  function compare (first, second) {
    return first - second;
  }
);
// numArray.toString(): 1,2,3,4,5,6,7,8,9

numArray.splice(2, 0, 2.5);
// numArray.toString(): 1,2,2.5,3,4,5,6,7,8,9

// output of following: 5,6,7
window.alert(numArray.splice(5,3).toString());
// numArray.toString(): 1,2,2.5,3,4,8,9
window.alert(numArray.toString());
```

Add element with value 2.5 at index 2, shift existing elements

JavaScript Arrays



```
// ArrayMethods.js
var numArray = [1,3,8,4,9,7,6,2,5];

// Sort in ascending order
numArray.sort(
  function compare (first, second) {
    return first - second;
  }
);
// numArray.toString(): 1,2,3,4,5,6,7,8,9

numArray.splice(2, 0, 2.5);
// numArray.toString(): 1,2,2.5,3,4,5,6,7,8,9

// output of following: 5,6,7
window.alert(numArray.splice(5,3).toString());
// numArray.toString(): 1,2,2.5,3,4,8,9
window.alert(numArray.toString());
```

Remove 3 elements starting at index 5

Built-in Objects - String

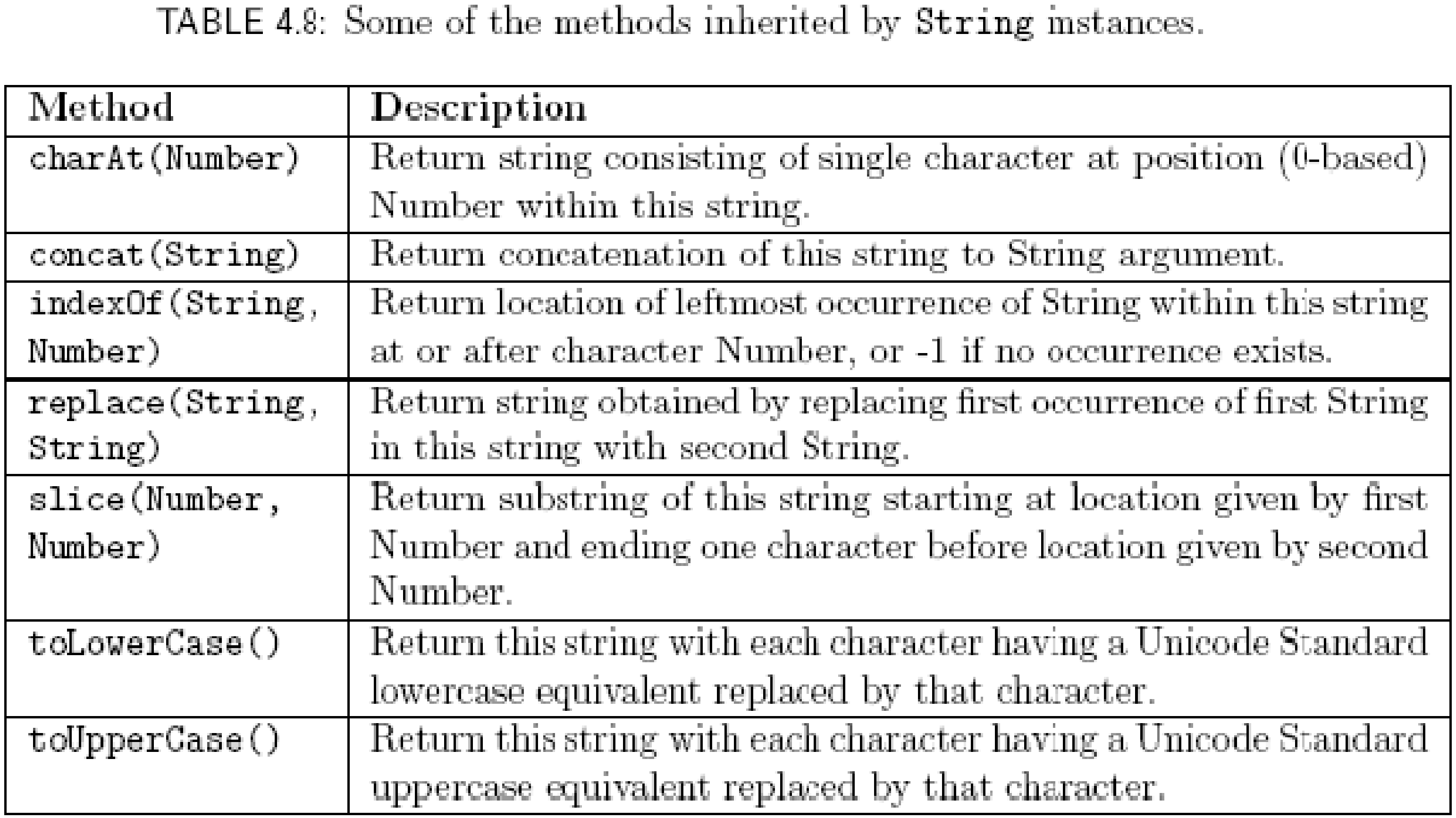


TABLE 4.8: Some of the methods inherited by `String` instances.

Method	Description
`charAt(Number)`	Return string consisting of single character at position (0-based) Number within this string.
`concat(String)`	Return concatenation of this string to String argument.
`indexOf(String, Number)`	Return location of leftmost occurrence of String within this string at or after character Number, or -1 if no occurrence exists.
`replace(String, String)`	Return string obtained by replacing first occurrence of first String in this string with second String.
`slice(Number, Number)`	Return substring of this string starting at location given by first Number and ending one character before location given by second Number.
`toLowerCase()`	Return this string with each character having a Unicode Standard lowercase equivalent replaced by that character.
`toUpperCase()`	Return this string with each character having a Unicode Standard uppercase equivalent replaced by that character.

Built-in Objects - Date

- The `Date()` built-in constructor can be used to create `Date` instances that represent the current date and time

  var now = new Date();

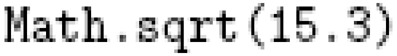
- Often used to display local date and/or time in Web pages

```
 window.alert(“Current date and time: “
              + now.toLocaleString());
```

- Other methods: `toLocaleDateString()` , `toLocaleTimeString()`, *etc*.

Built-in Objects - Math

- `Math` object has methods for performing standard mathematical calculations:

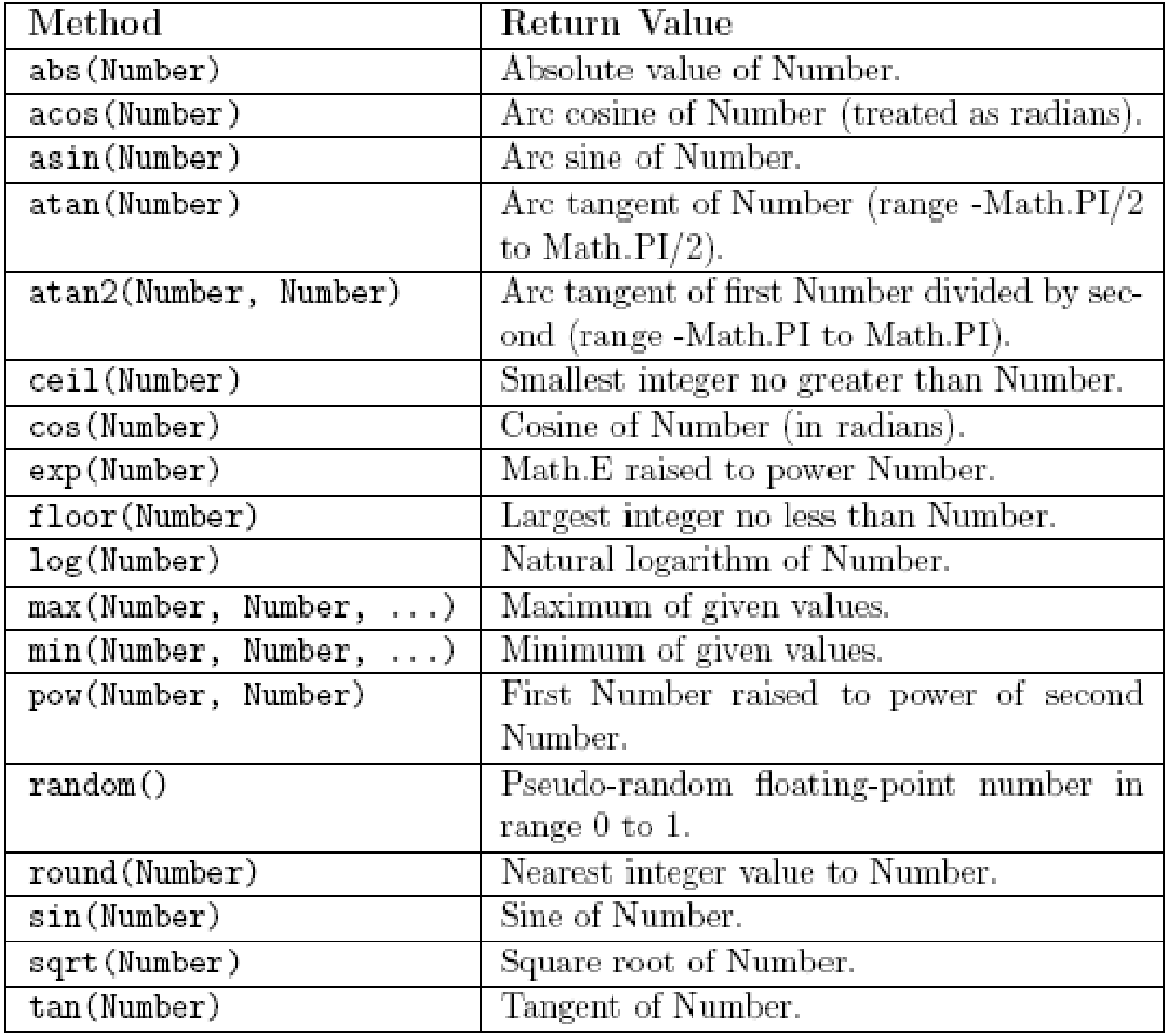


```
Math.sqrt(15.3)
```

- Also has properties with approximate values for standard mathematical quantities, *e.g.*, e (`Math.E`) and π (`Math.PI`)

Built-in Objects

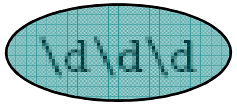
TABLE 4.9: Methods of the `Math` built-in object.



Method	Return Value
`abs(Number)`	Absolute value of Number.
`acos(Number)`	Arc cosine of Number (treated as radians).
`asin(Number)`	Arc sine of Number.
`atan(Number)`	Arc tangent of Number (range -Math.PI/2 to Math.PI/2).
`atan2(Number, Number)`	Arc tangent of first Number divided by second (range -Math.PI to Math.PI).
`ceil(Number)`	Smallest integer no greater than Number.
`cos(Number)`	Cosine of Number (in radians).
`exp(Number)`	Math.E raised to power Number.
`floor(Number)`	Largest integer no less than Number.
`log(Number)`	Natural logarithm of Number.
`max(Number, Number, ...)`	Maximum of given values.
`min(Number, Number, ...)`	Minimum of given values.
`pow(Number, Number)`	First Number raised to power of second Number.
`random()`	Pseudo-random floating-point number in range 0 to 1.
`round(Number)`	Nearest integer value to Number.
`sin(Number)`	Sine of Number.
`sqrt(Number)`	Square root of Number.
`tan(Number)`	Tangent of Number.

JavaScript Regular Expressions

- A regular expression is a particular representation of a set of strings
  - Ex: JavaScript regular expression representing the set of syntactically-valid US telephone area codes (three-digit numbers):

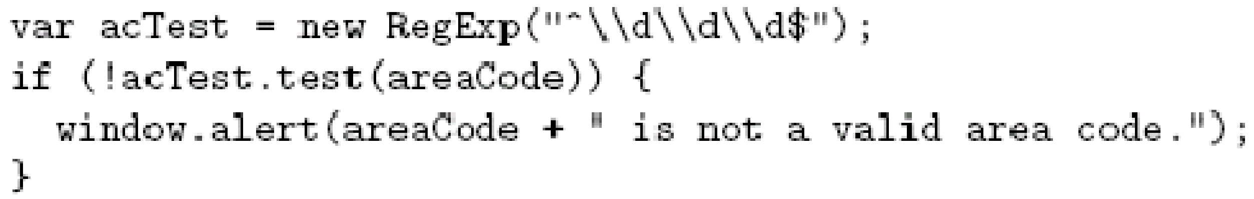


```
\d\d\d
```

- \d represents the set {“0”, “1”, …, “9”}
- Concatenated regular expressions represent the “concatenation” (Cartesian product) of their sets

JavaScript Regular Expressions

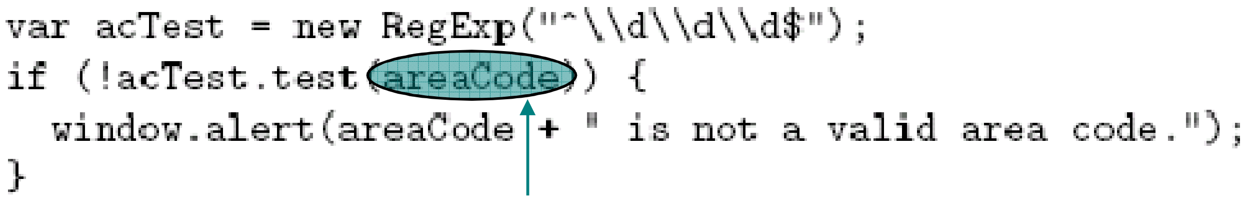
- Using regular expressions in JavaScript



```
var acTest = new RegExp("^\\d\\d\\d$");
if (!acTest.test(areaCode)) {
  window.alert(areaCode + " is not a valid area code.");
}
```

JavaScript Regular Expressions

- Using regular expressions in JavaScript



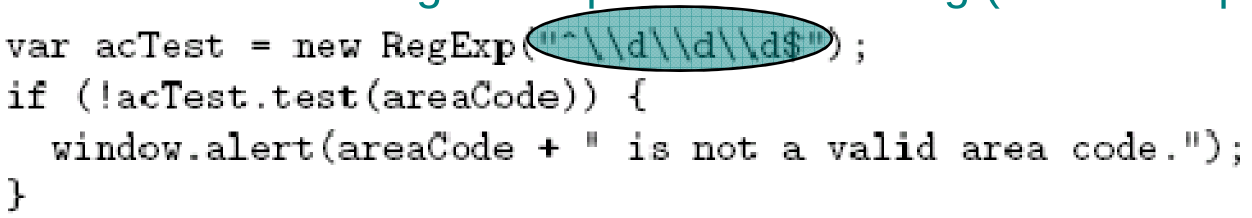
```
var acTest = new RegExp("^\\d\\d\\d$");
if (!acTest.test(areaCode)) {
  window.alert(areaCode + " is not a valid area code.");
}
```

Variable containing string to be tested

JavaScript Regular Expressions

- Using regular expressions in JavaScript

Regular expression as String (must escape \)

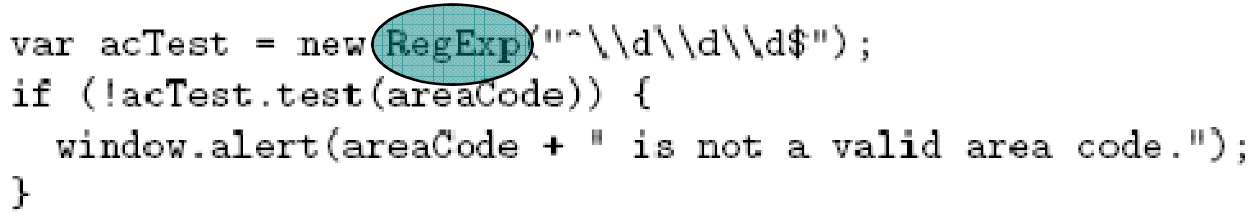


```
var acTest = new RegExp("^\\d\\d\\d$");
if (!acTest.test(areaCode)) {
  window.alert(areaCode + " is not a valid area code.");
}
```

JavaScript Regular Expressions

- Using regular expressions in JavaScript

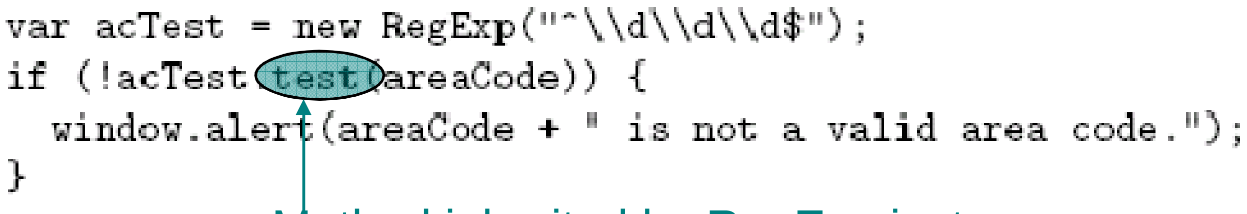
Built-in constructor



```
var acTest = new RegExp("^\\d\\d\\d$");
if (!acTest.test(areaCode)) {
  window.alert(areaCode + " is not a valid area code.");
}
```

JavaScript Regular Expressions

- Using regular expressions in JavaScript



```
var acTest = new RegExp("^\\d\\d\\d$");
if (!acTest.test(areaCode)) {
  window.alert(areaCode + " is not a valid area code.");
}
```

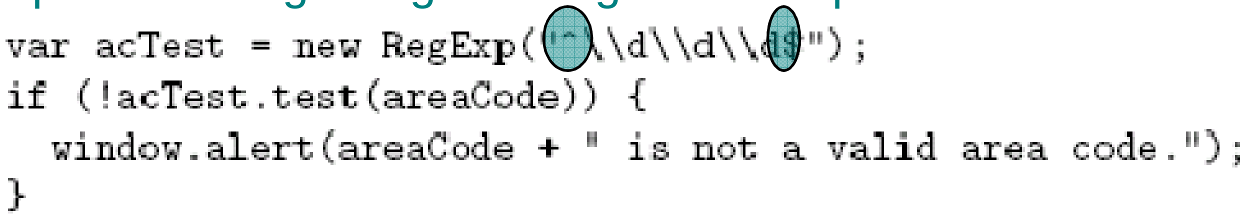
Method inherited by RegExp instances: returns true if the argument *contains* a substring in the set of strings represented by the regular expression

JavaScript Regular Expressions

- Using regular expressions in JavaScript

Represents beginning of string

Represents end of string

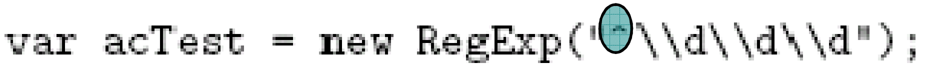


```
var acTest = new RegExp("^\\d\\d\\d$");
if (!acTest.test(areaCode)) {
  window.alert(areaCode + " is not a valid area code.");
}
```

This expression matches only strings with exactly three digits (no other characters, even white space)

JavaScript Regular Expressions

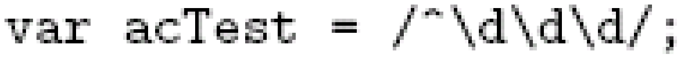
- Using regular expressions in JavaScript



```
var acTest = new RegExp("^\\d\\d\\d");
```

Represents all strings that *begin* with three digits

- Alternate syntax:

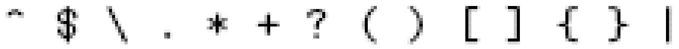


```
var acTest = /^\d\d\d/;
```

Regular expression literal.
Do *not* escape \.

JavaScript Regular Expressions

- Simplest regular expression is any character that is not a special character:



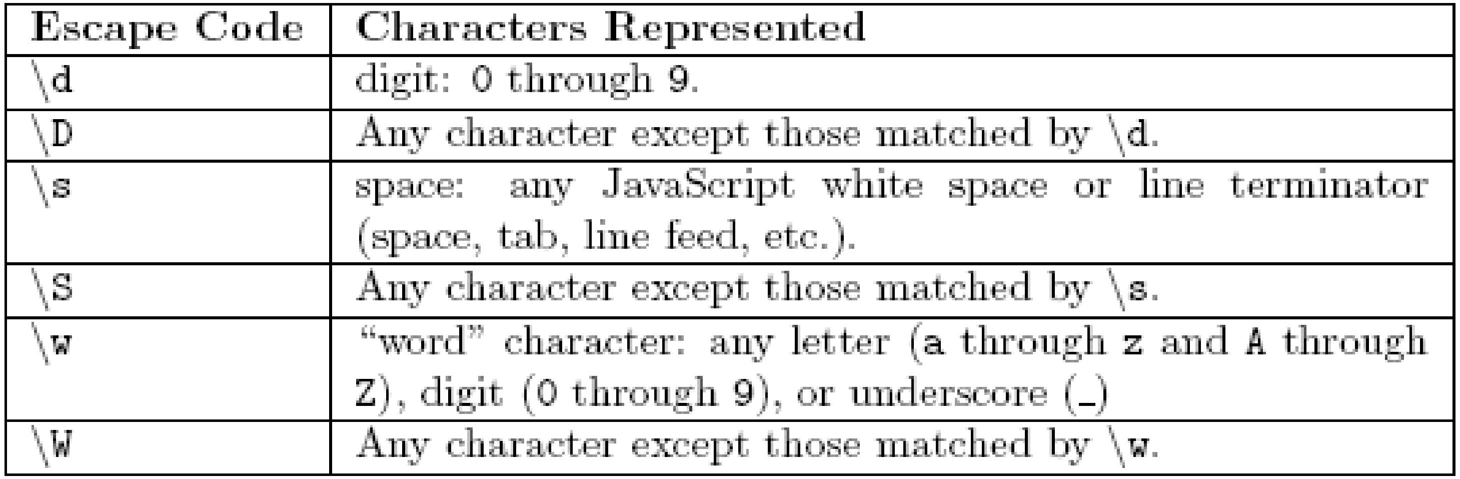
^ $ \ . * + ? ( ) [ ] { } |

  - Ex: _ is a regular expression representing {“_”}
- Backslash-escaped special character is also a regular expression
  - Ex: \$ represents {“$”}

JavaScript Regular Expressions

- Special character **.** (dot) represents any character except a line terminator
- Several escape codes are regular expressions representing sets of chars:

TABLE 4.10: JavaScript multi-character escape codes.



Escape Code	Characters Represented
`\d`	digit: `0` through `9`.
`\D`	Any character except those matched by `\d`.
`\s`	space: any JavaScript white space or line terminator (space, tab, line feed, etc.).
`\S`	Any character except those matched by `\s`.
`\w`	"word" character: any letter (`a` through `z` and `A` through `Z`), digit (`0` through `9`), or underscore (`_`)
`\W`	Any character except those matched by `\w`.

JavaScript Regular Expressions

- Three types of operations can be used to combine simple regular expressions into more complex expressions:
  - Concatenation
  - Union (|)
  - Kleene star (*)
- XML DTD content specification syntax based in part on regular expressions

JavaScript Regular Expressions

- Concatenation
  - Example: `^\d\. \w$`
    - String consisting entirely of four characters:
    - Digit followed by
    - A . followed by
    - A single space followed by
    - Any “word” character
  - Quantifier shorthand syntax for concatenation:

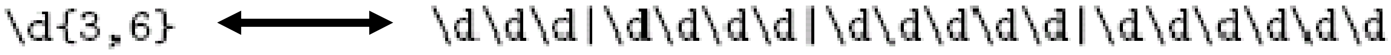
`\d{3}` ⟷ `\d\d\d`

JavaScript Regular Expressions

- Union
  - Ex: `\d|\s`
  - Union of set of strings represented by regular expressions
    - Set of single-character strings that are either a digit or a space character
- Character class: shorthand for union of one or more ranges of characters
  - Ex: `[a-z]` set of lower case letters
  - Ex: `[a-zA-Z0-9]|_` the \w escape code class

JavaScript Regular Expressions

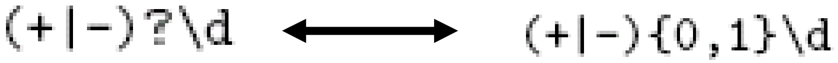
- Unions of concatenations



```
\d{3,6}  ↔  \d\d\d|\d\d\d\d|\d\d\d\d\d|\d\d\d\d\d\d
```

  - Note that concatenation has higher precedence than union

- Optional regular expression



```
(+|-)?\d  ↔  (+|-){0,1}\d
```

JavaScript Regular Expressions

- Kleene star
  - Ex: `\d*` any number of digits (including none)
  - Ex: `\w*(\d\w*[a-zA-Z]|[a-zA-Z]\w*\d)\w*`
    - Strings consisting of only “word” characters
    - String must contain both a digit and a letter (in either order)