WEB TECHNOLOGIES
A COMPUTER SCIENCE PERSPECTIVE

JEFFREY C. JACKSON

Chapter 4
Client-Side Programming: the JavaScript Language

JavaScript History and Versions

- JavaScript was introduced as part of the Netscape 2.0 browser
- Microsoft soon released its own version called JScript
- ECMA developed a standard language known as ECMAScript
- ECMAScript Edition 3 is widely supported and is what we will call “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:

JavaScript Introduction

- Prompt window example:

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

JavaScript Properties

- Note that JavaScript code did not need to be compiled
  - JavaScript is an interpreted language
  - Portion of browser software that reads and executes JavaScript is an interpreter
- Interpreted vs. compiled languages:
  - Advantage: simplicity
  - Disadvantage: efficiency

JavaScript Properties

- JavaScript is a scripting language: designed to be executed within a larger software environment
- JavaScript can be run within a variety of environments:
  - Web browsers (our focus in next chapter)
  - Web servers
  - Application containers (general-purpose programming)

JavaScript Properties

- Components of a JavaScript implementation:
  - Scripting engine: interpreter plus required ECMAScript functionality (core library)
  - Hosting environment: functionality specific to environment
    - Example: browsers provide `alert` and `prompt`
    - All hosting environment functionality provided via objects

JavaScript Properties

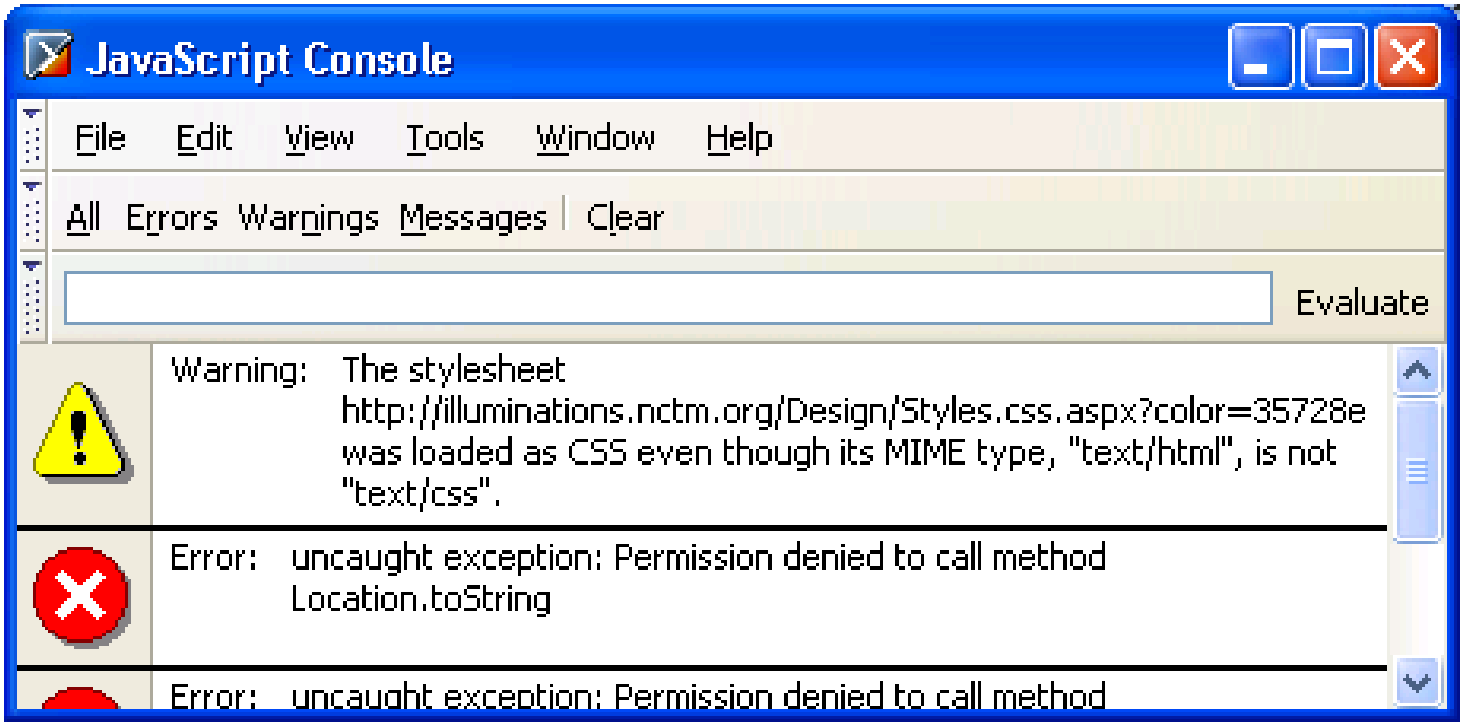
- All data in JavaScript is an object or a property of an object
- Types of JavaScript objects
  - Native: provided by scripting engine
    - If automatically constructed before program execution, known as a built-in object (ex: `window`)
  - Host: provided by host environment
    - `alert` and `prompt` are host objects

Developing JavaScript Software

- Writing JavaScript code
  - Any text editor (*e.g.*, Notepad, Emacs)
  - Specialized software (*e.g.*, MS Visual InterDev)
- Executing JavaScript
  - Load into browser (need HTML document)
  - Browser detects syntax and run-time errors
    - Mozilla: JavaScript console lists errors
    - IE6: Exclamation icon and pop-up window

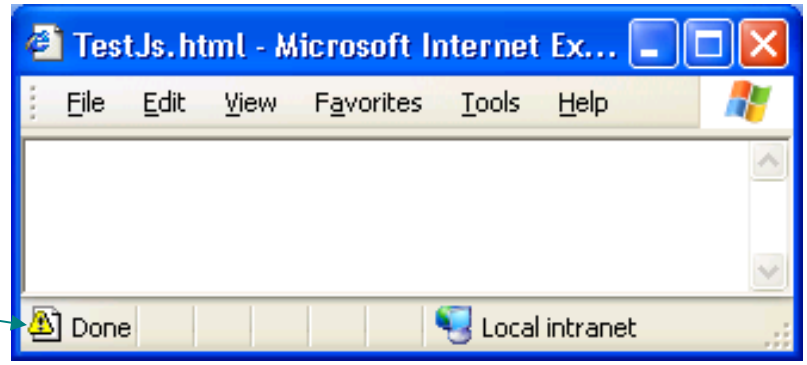
Developing JavaScript Software

- Mozilla JavaScript console (Tools | Web Development | JavaScript Console):

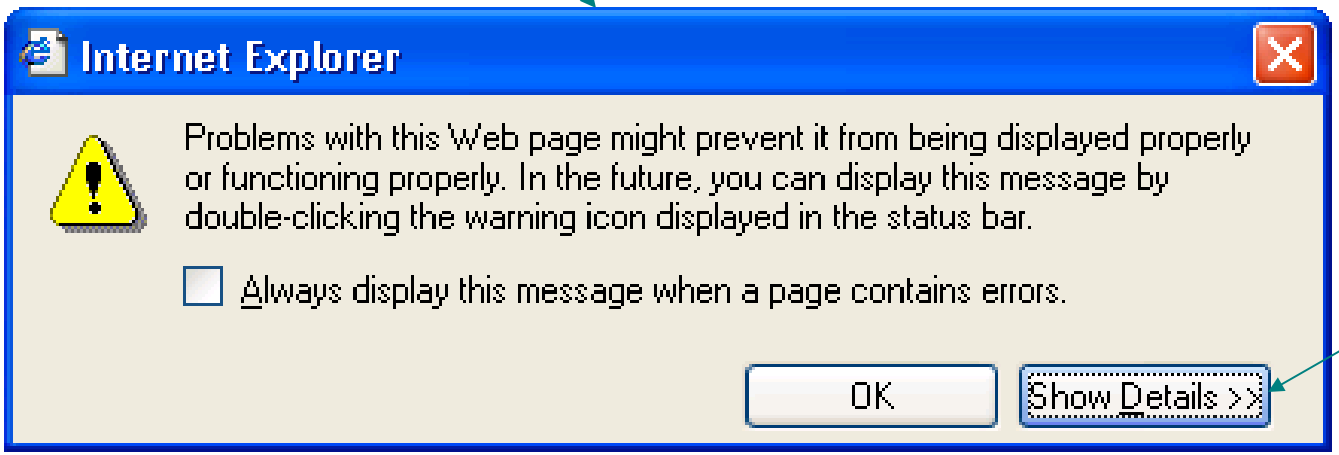


Developing JavaScript Software

- IE6 error window:



Error indicator; double-clicking icon opens error window



Click to see error messages

Developing JavaScript Software

- Debugging
  - Apply generic techniques: desk check, add debug output (alert's)
  - Use specialized JavaScript debuggers: later
- Re-executing
  - Overwrite .js file
  - Reload (Mozilla)/Refresh (IE) HTML document that loads the file

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

```
// HighLow.js
```

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

```
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?", "");
```

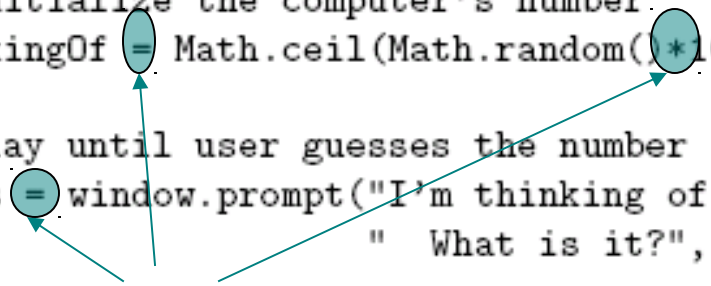
Basic JavaScript Syntax

```
// 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

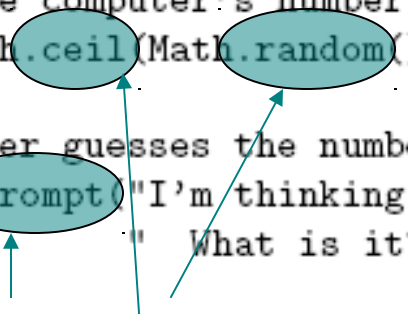
```
// HighLow.js

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

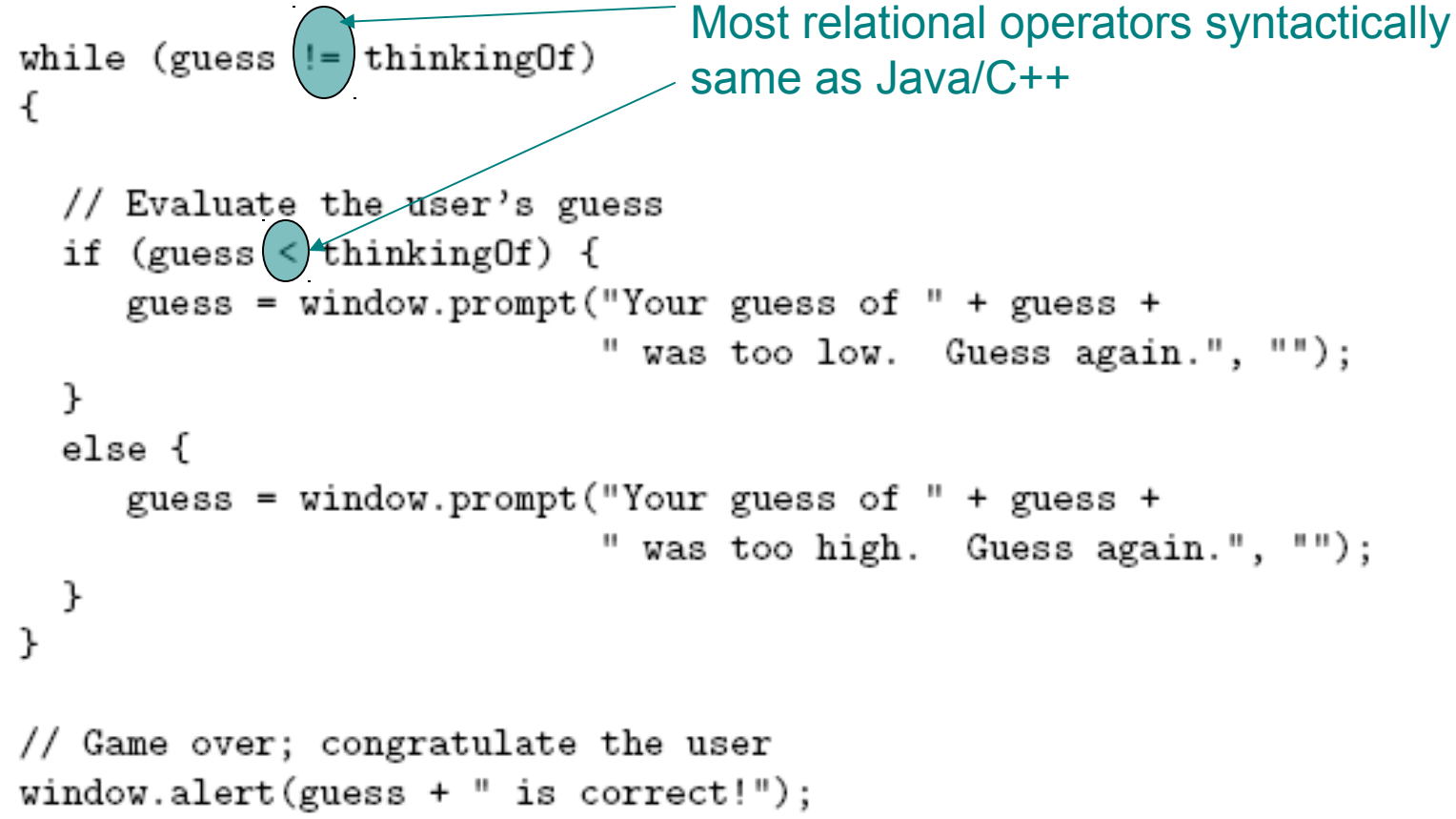
```
while (guess != thinkingOf)
{

  // 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

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

```
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



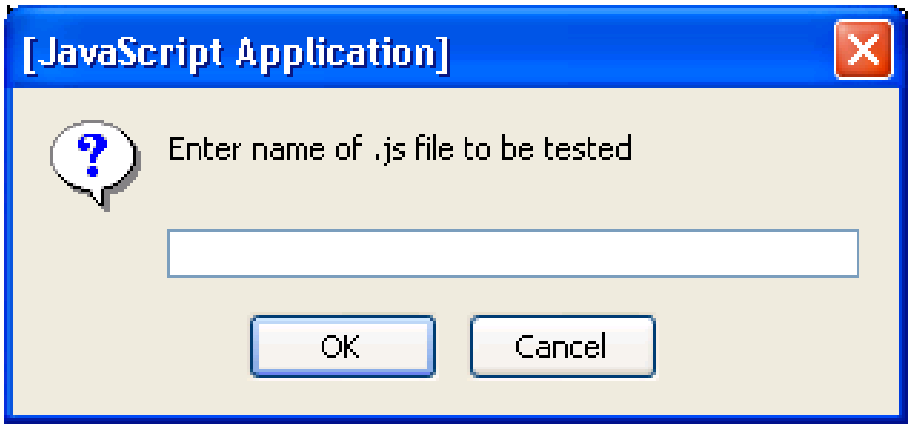
Basic JavaScript Syntax

```
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:



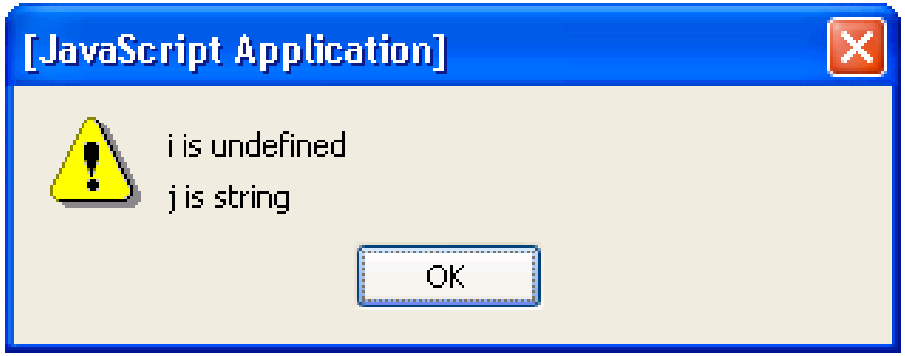
Variables and Data Types

- Type of a variable is dynamic: depends on the type of data it contains
- JavaScript has six data types:
  - Number
  - String
  - Boolean (values `true` and `false`)
  - Object
  - Null (only value of this type is `null`)
  - Undefined (value of newly created variable)
- Primitive data types: all but Object

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));
```



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

- Common automatic type conversions:
  - Compare String and Number: String value converted to Number
  - Condition of `if` or `while` converted to Boolean
  - Array accessor (*e.g.*, `3` in `records[3]`) converted to String

Variables and Data Types

TABLE 4.2: Data type conversions to Boolean.

Original Value	Value as Boolean
undefined	`false`
`null`	`false`
`0`	`false`
NaN	`false`
`""` (empty string)	`false`
any other value	`true`

Variables and Data Types

TABLE 4.3: Data type conversions to String.

Original Value	Value as String
undefined	`"undefined"`
`null`	`"null"`
`true`, `false`	`"true"`, `"false"`
NaN	`"NaN"`
Infinity, -Infinity	`"Infinity"`, `"-Infinity"`
other Number up to ≈20 digits	integer or decimal representation
Number over ≈20 digits	scientific notation
Object	call to `toString()` method on the object

Variables and Data Types

TABLE 4.3: Data type conversions to String.

Original Value	Value as String
undefined	`"undefined"`
`null`	`"null"`
`true`, `false`	`"true"`, `"false"`
NaN	`"NaN"`
Infinity, -Infinity	`"Infinity"`, `"-Infinity"`
other Number up to ≈20 digits	integer or decimal representation
Number over ≈20 digits	scientific notation
Object	call to `toString()` method on the object

Special Number values ("Not a Number" and number too large to represent)

Variables and Data Types

TABLE 4.4: Data type conversions to Number.

Original Value	**Value as Number**
undefined	`NaN`
`null`, `false`, `""` (empty string)	0
`true`	1
String representing number	represented number
other String	`NaN`
Object	call to `valueOf()` method on the object

Variables and Data Types

- Syntax rules for names (identifiers):
  - Must begin with letter or underscore (_)
  - Must contain only letters, underscores, and digits (or certain other characters)
  - Must not be a reserved word

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

- A variable will automatically be created if a value is assigned to an undeclared identifier:

var is not required

```
testing = "Does this work?";
window.alert(testing);
```

- Recommendation: declare all variables
  - Facilitates maintenance
  - Avoids certain exceptions

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*

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 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 ==, != convert both operands to Number
  - Exception: If both operands are String, no conversion is performed (lex. comparison)
  - Exception: values of Undefined and Null are equal(!)
  - Exception: instance of Date built-in “class” is converted to String (and host object conversion is implementation dependent)
  - Exception: two Objects are equal only if they are references to the same object

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 Operators

- Example bit operators:

```
// ComplementOne.js
window.alert(~1);
```

→ -2

```
// ComplementOne.js (part 2)
window.alert((~1) >>> 0);
```

→ 4294967294 ($2^{32} - 2$)

JavaScript Numbers

- Syntactic representations of Number
  - Integer (42) and decimal (42.0)
  - Scientific notation (-12.4e12)
  - Hexadecimal (0xfa0)
- Internal representation
  - Approximately 16 digits of precision
  - Approximate range of magnitudes
    - Smallest: 10^{-323}
    - Largest: 10^{308} (`Infinity` if literal is larger)

JavaScript Strings

- String literals can be single- or double-quoted
- Common escape characters within Strings
  - `\n` newline
  - `\"` escaped double quote (also `\'` for single)
  - `\\` escaped backslash
  - `\uxxxx` arbitrary Unicode 16-bit code point (x's are four hex digits)

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*

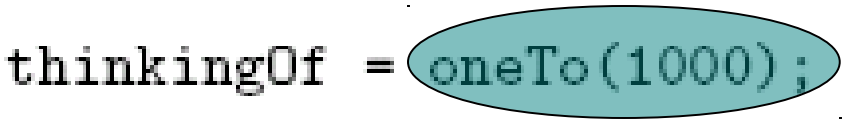
JavaScript Functions

- Function call syntax

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

JavaScript Functions

- Function call syntax



Function call is an expression, can
be used on right-hand side of assignments,
as expression statement, etc.

JavaScript Functions

- Function call syntax

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

Function name

JavaScript Functions

- Function call syntax

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

Argument list

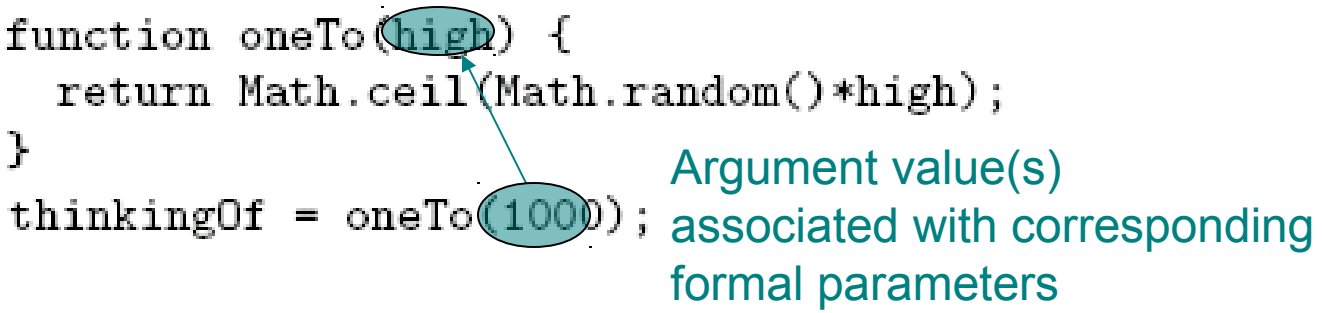
JavaScript Functions

- Function call semantics:

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

JavaScript Functions

- Function call semantics:



JavaScript Functions

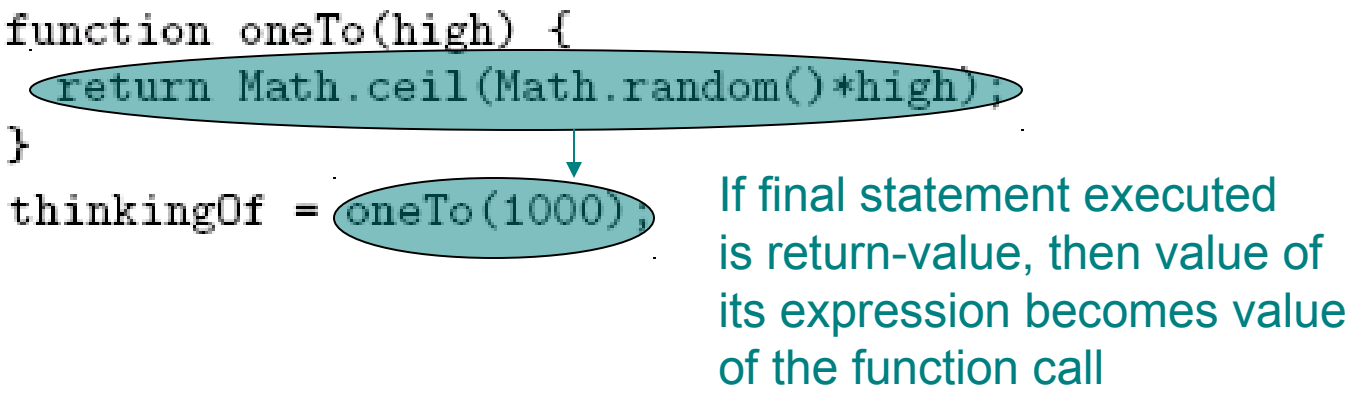
- Function call semantics:

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

Expression(s) in body evaluated as if formal parameters are variables initialized by argument values

JavaScript Functions

- Function call semantics:



JavaScript Functions

- Function call semantics:

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

Value of function call is then used in larger expression containing function call.

JavaScript Functions

- Function call semantics details:
  - Arguments:
    - May be expressions: `oneTo(999+1)`
    - Object's effectively passed by reference (more later)
  - Formal parameters:
    - May be assigned values, argument is not affected
  - Return value:
    - If last statement executed is not return-value, then returned value is of type Undefined

JavaScript Functions

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

JavaScript Functions

- Local vs. global variables

Global variable: declared outside any function

```
var j=6;   // global variable declaration and initialization
function test()
{
  var j;   // local variable declaration
  j=7;     // Which variable(s) does this change?
  return;
}
test();
window.alert(j);
```

JavaScript Functions

- Local vs. global variables

Local variable declared within a function

```
var j=6;   // global variable declaration and initialization
function test()
{
  var j;   // local variable declaration
  j=7;     // Which variable(s) does this change?
  return;
}
test();
window.alert(j);
```

JavaScript Functions

- Local vs. global variables

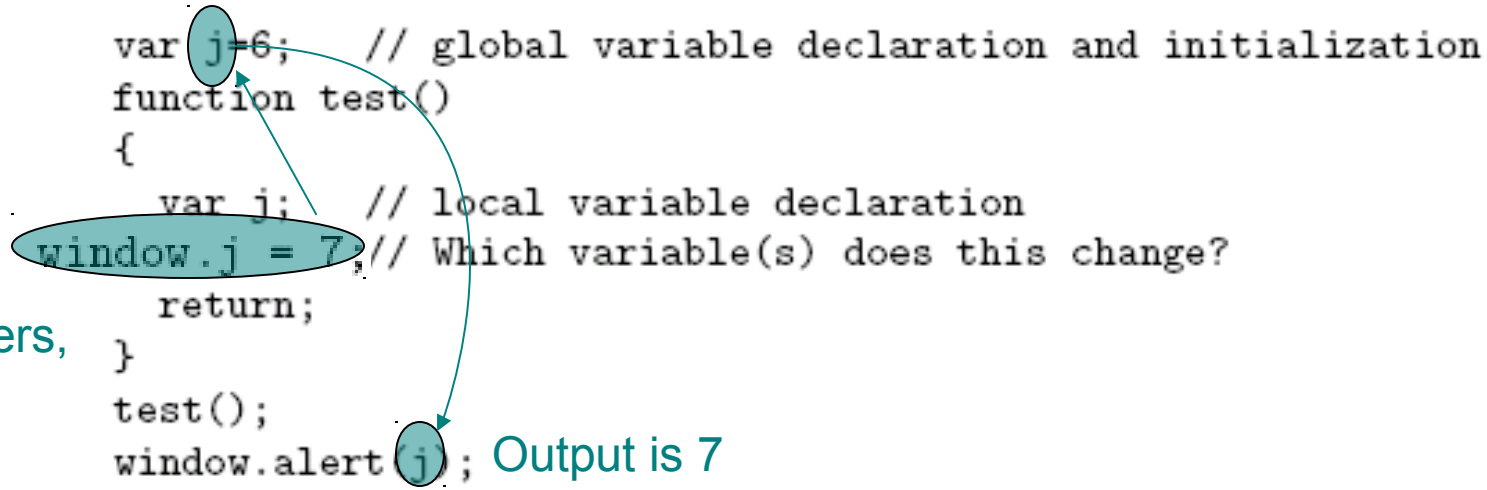
Local declaration shadows corresponding global declaration

```
var j=6;   // global variable declaration and initialization
function test()
{
  var j;   // local variable declaration
  j=7;     // Which variable(s) does this change?
  return;
}
test();
window.alert(j);
```

Output is 6

JavaScript Functions

- Local vs. global variables



In browsers, global variables (and functions) are stored as properties of the `window` built-in object.

JavaScript Functions

- Recursive functions
  - Recursion (function calling itself, either directly or indirectly) is supported
  - C++ static variables are not supported
  - Order of declaration of mutually recursive functions is unimportant (no need for prototypes as in C++)

JavaScript Functions

- Explicit type conversion supplied by built-in functions
  - Boolean(), String(), Number()
  - Each takes a single argument, returns value representing argument converted according to type-conversion rules given earlier

Object Introduction

- An object is a set of properties
- A property consists of a unique (within an object) name with an associated value
- The type of a property depends on the type of its value and can vary dynamically

```
o.prop = true;
o.prop = "true";
o.prop = 1;
```

prop is Boolean

prop is now String

prop is now Number

Object Introduction

- There are no classes in JavaScript
- Instead, properties can be created and deleted dynamically

```
var o1 = new Object();
o1.testing = "This is a test";
delete o1.testing;
```

Create an object `o1`
Create property `testing`
Delete `testing` property

Object Creation

- Objects are created using new expression

Constructor and argument list

- A constructor is a function
  - When called via new expression, a new empty Object is created and passed to the constructor along with the argument values
  - Constructor performs initialization on object
    - Can add properties and methods to object
    - Can add object to an inheritance hierarchy

Object Creation

- The `Object()` built-in constructor
  - Does not add any properties or methods directly to the object
  - Adds object to hierarchy that defines default `toString()` and `valueOf()` methods (used for conversions to String and Number, resp.)

Property Creation

- Assignment to a non-existent (even if inherited) property name creates the property: `o1.testing = "This is a test";`
- Object initializer notation can be used to create an object (using `Object()` constructor) and one or more properties in a single statement:

```
var o2 = { p1:5+9, p2:null, testing:"This is a test" };
```

Enumerating Properties

- Special form of `for` statement used to iterate through all properties of an object:

```
var hash = new Object();
hash.kim = "85";
hash.sam = "92";
hash.lynn = "78";
for (var aName in hash) {
  window.alert(aName + " is a property of hash.");
}
```

Produces three alert boxes; order of names is implementation-dependent.

Accessing Property Values

- The JavaScript object dot notation is actually shorthand for a more general associative array notation in which Strings are array indices:

```
hash.kim ⟶ hash["kim"]
```

- Expressions can supply property names:

```
window.alert(aName + " scored " + hash[aName]);
```

Converted to String if necessary

Object Values

- Value of Object is reference to object:

```
var o1 = new Object();
o1.data = "Hello";
var o2 = o1;
o2.data += " World!";
window.alert(o1.data);
```

Object Values

- Value of Object is reference to object:

o2 is another name for o1

```
var o1 = new Object();
o1.data = "Hello";
var o2 = o1;
o2.data += " World!";
window.alert(o1.data);
```

Object Values

- Value of Object is reference to object:

o1 is changed

```
var o1 = new Object();
o1.data = "Hello";
var o2 = o1;
o2.data += " World!";
window.alert(o1.data);
```

Object Values

- Value of Object is reference to object:

```
var o1 = new Object();
o1.data = "Hello";
var o2 = o1;
o2.data += " World!";
window.alert(o1.data);
```

Output is `Hello World!`

Object Values

- Object argument values are references

```
// Create two different objects with identical data
var o1 = new Object();
o1.data = "original";
var o2 = new Object();
o2.data = "original";

// Call the function on these objects and display the results
objArgs(o1, o2);

function objArgs(param1, param2) {...}
```

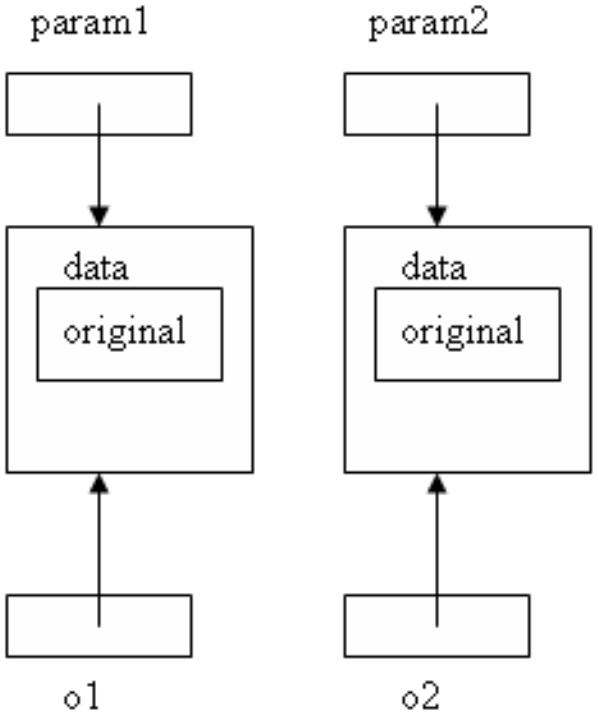
Object Values

- Object argument values are references

```
// Create two different objects with identical data
var o1 = new Object();
o1.data = "original";
var o2 = new Object();
o2.data = "original";

// Call the function on these objects and display the results
objArgs(o1, o2);

function objArgs(param1, param2) {...}
```

Object Values

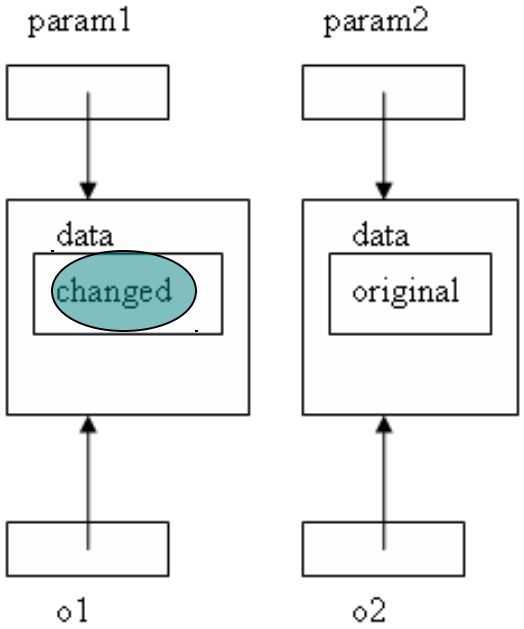
- Object argument values are references



Object Values

- Object argument values are references

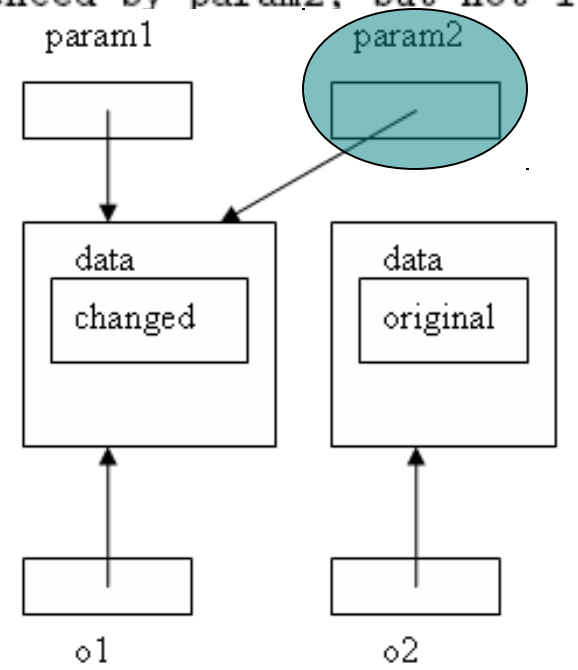
```
function objArgs(param1, param2) {
  // Change the data in param1 and its argument
  param1.data = "changed";
```



Object Values

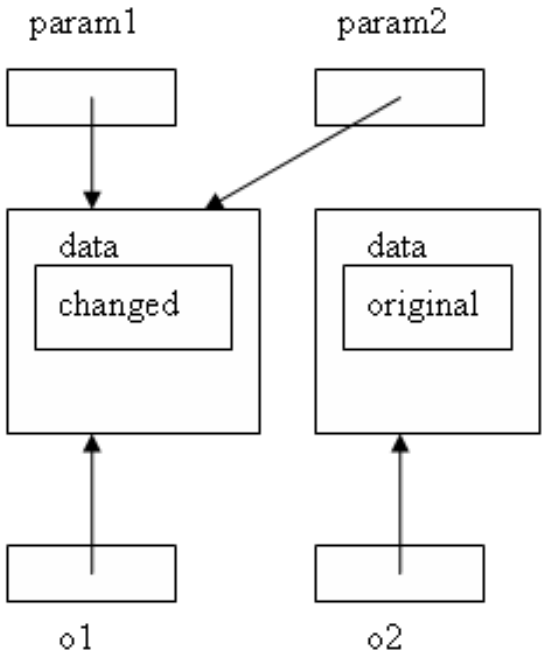
- Object argument values are references

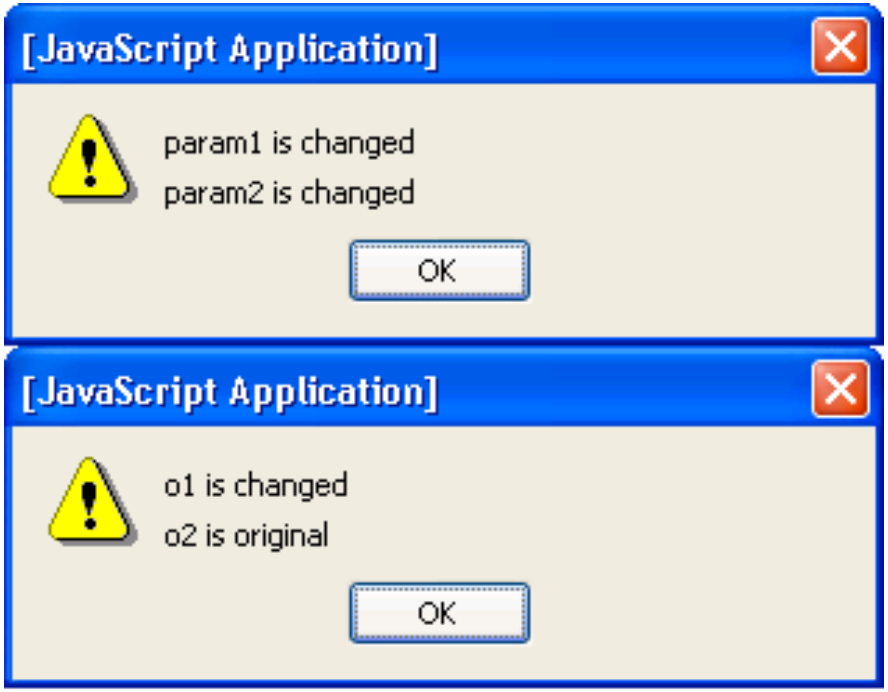
```
function objArgs(param1, param2) {
  // Change the data in param1 and its argument
  param1.data = "changed";
  // Change the object referenced by param2, but not its argument
  param2 = param1;
```



Object Values

- Object argument values are references





Object Methods

- JavaScript functions are stored as values of type Object
- A function declaration creates a function value and stores it in a variable (property of `window`) having the same name as the function
- A method is an object property for which the value is a function

Object Methods

```
function leaf() {
  return this.left == null && this.right == null;
}

function makeBTNode(value) {
  var node = new Object();
  node.left = node.right = null;
  node.value = value;
  node.isLeaf = leaf;
  return node;
}
```

Object Methods

Creates global variable named `leaf` with function value

```
function leaf() {
  return this.left == null && this.right == null;
}

function makeBTNode(value) {
  var node = new Object();
  node.left = node.right = null;
  node.value = value;
  node.isLeaf = leaf;
  return node;
}
```

Object Methods

```
function leaf() {
  return this.left == null && this.right == null;
}

function makeBTNode(value) {
  var node = new Object();
  node.left = node.right = null;
  node.value = value;
  node.isLeaf = leaf;
  return node;
}
```

Creates `isLeaf()` method that is defined by `leaf()` function

Object Methods

Refers to object that "owns" method when leaf() is called as a method

```
function leaf() {
  return this.left == null && this.right == null;
}

function makeBTNode(value) {
  var node = new Object();
  node.left = node.right = null;
  node.value = value;
  node.isLeaf = leaf;
  return node;
}
```

Object Methods

```
var node1 = makeBTNode(3);
var node2 = makeBTNode(7);
node1.right = node2;

// Output the value of isLeaf() on each node
window.alert("node1 is a leaf: " + node1.isLeaf());
window.alert("node2 is a leaf: " + node2.isLeaf());
```

Object Methods

```
var node1 = makeBTNode(3);
var node2 = makeBTNode(7);
node1.right = node2;

// Output the value of isLeaf() on each node
window.alert("node1 is a leaf: " + node1.isLeaf());
window.alert("node2 is a leaf: " + node2.isLeaf());
```

Creates two objects each with method `isLeaf()`

Object Methods

```
var node1 = makeBTNode(3);
var node2 = makeBTNode(7);
node1.right = node2;

// Output the value of isLeaf() on each node
window.alert("node1 is a leaf: " + node1.isLeaf());
window.alert("node2 is a leaf: " + node2.isLeaf());
```

Calls to `isLeaf()` method

Object Methods

- Original version: `leaf()` can be called as function, but we only want a method

```
function leaf() {
  return this.left == null && this.right == null;
}

function makeBTNode(value) {
  var node = new Object();
  node.left = node.right = null;
  node.value = value;
  node.isLeaf = leaf;
  return node;
}
```

Object Methods

- Alternative:

```
function leaf() {
  return this.left == null && this.right == null;
}

function makeBTNode(value) {
  var node = new Object();
  node.left = node.right = null;
  node.value = value;
  node.isLeaf = leaf;
  return node;
}
```

Function expression syntactically the same as function declaration but does not produce a global variable.

Object Methods

- Alternative

```
function makeBTNode(value) {
  var node = new Object();
  node.left = node.right = null;
  node.value = value;
  node.isLeaf =
    function leaf() {
      return this.left == null && this.right == null;
    };
  return node;
}
```

Object Constructors

- User-defined constructor is just a function called using new expression:

```
var node1 = new BTNode(3);
var node2 = new BTNode(7);
```

Constructor

- Object created using a constructor is known as an instance of the constructor

Object Constructors

Original function

```
function makeBTNode(value) {
  var node = new Object();
  node.left = node.right = null;
  node.value = value;
  node.isLeaf =
    function leaf() {
      return this.left == null && this.right == null;
    };
  return node;
}
```

Function intended to be used as constructor

```
function BTNode(value) {
  this.left = this.right = null;
  this.value = value;
  this.isLeaf =
    function leaf() {
      return this.left == null && this.right == null;
    };
 }
```

Object Constructors

```
function makeBTNode(value) {
  var node = new Object();
  node.left = node.right = null;
  node.value = value;
  node.isLeaf =
    function leaf() {
      return this.left == null && this.right == null;
    };
  return node;
}

function BTNode(value) {
  this.left = this.right = null;
  this.value = value;
  this.isLeaf =
    function leaf() {
      return this.left == null && this.right == null;
    };
}
```

Object is constructed automatically by new expression

Object Constructors

```
function makeBTNode(value) {
  var node = new Object();
  node.left = node.right = null;
  node.value = value;
  node.isLeaf =
    function leaf() {
      return this.left == null && this.right == null;
    };
  return node;
}

function BTNode(value) {
  this.left = this.right = null;
  this.value = value;
  this.isLeaf =
    function leaf() {
      return this.left == null && this.right == null;
    };
}
```

Object referenced using `this` keyword

Object Constructors

```
function makeBTNode(value) {
  var node = new Object();
  node.left = node.right = null;
  node.value = value;
  node.isLeaf =
    function leaf() {
      return this.left == null && this.right == null;
    };
  return node;
}

function BTNode(value) {
  this.left = this.right = null;
  this.value = value;
  this.isLeaf =
    function leaf() {
      return this.left == null && this.right == null;
    };
}
```

No need to return initialized object

Object Constructors

- Object created using a constructor is known as an instance of the constructor

```
var node1 = new BTNode(3);
var node2 = new BTNode(7);
```

Instances of BTNode

- `instanceof` operator can be used to test this relationship:

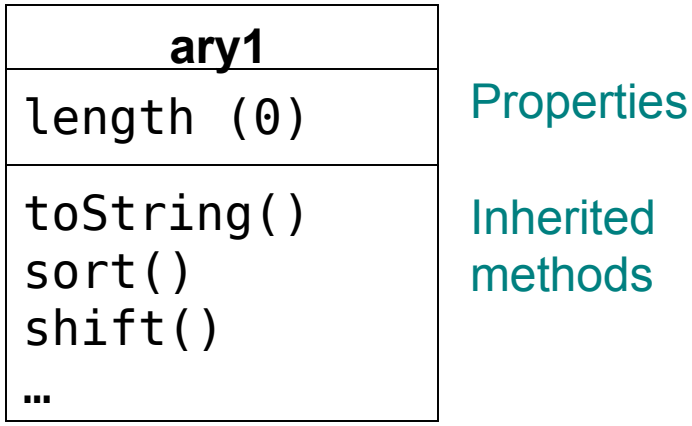
```
window.alert("node1 is instance of BTNode: " +
             (node1 instanceof BTNode));
```

Evaluates to `true`

JavaScript Arrays

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

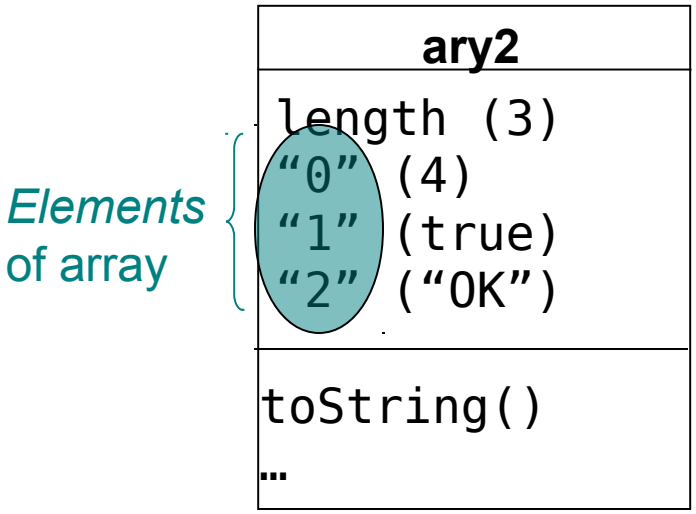
```
var ary1 = new Array();
```



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");
```



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" ] ];
```

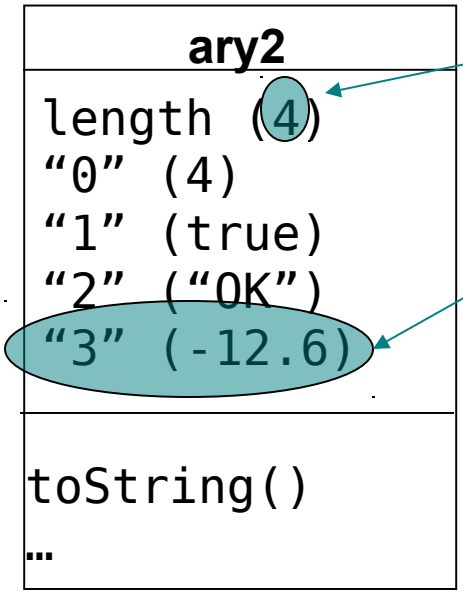
ttt[1][2]

JavaScript Arrays

- Changing the number of elements:

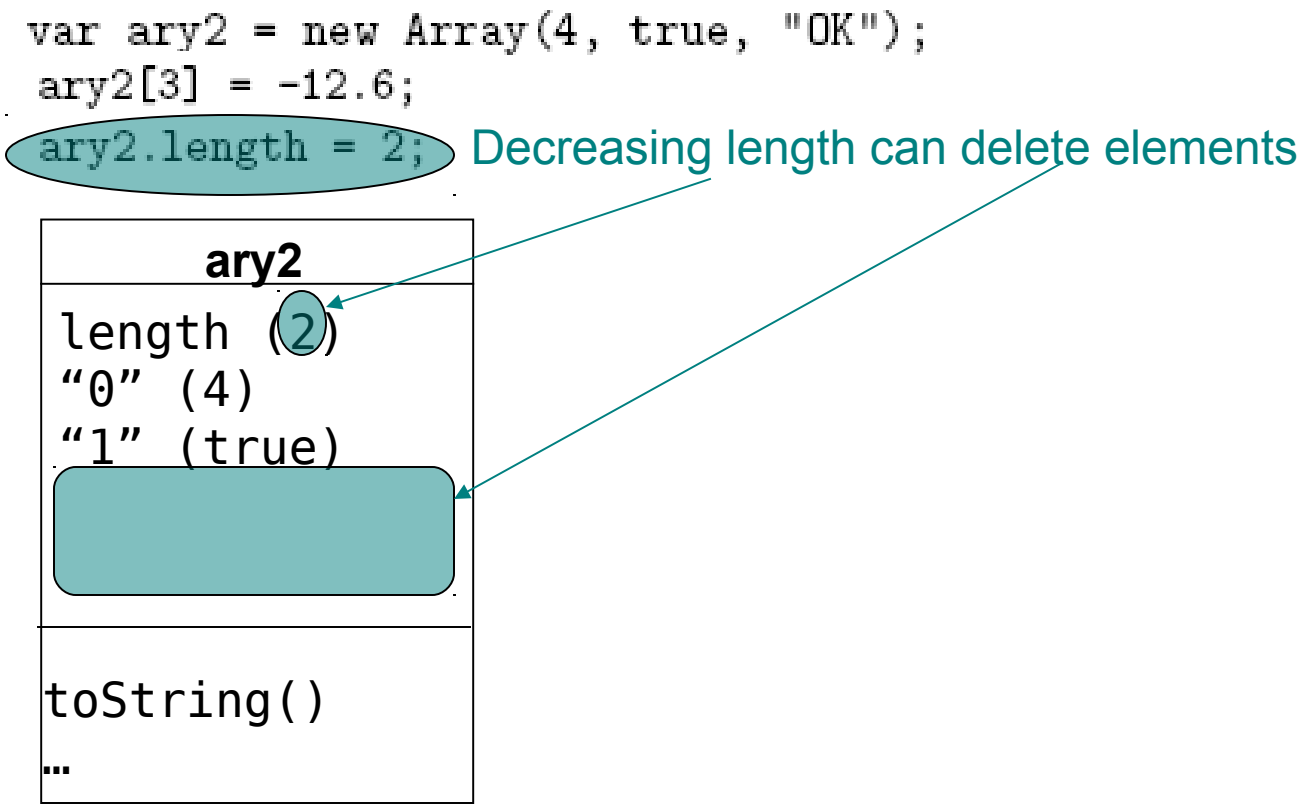
```
var ary2 = new Array(4, true, "OK");
ary2[3] = -12.6;
```

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



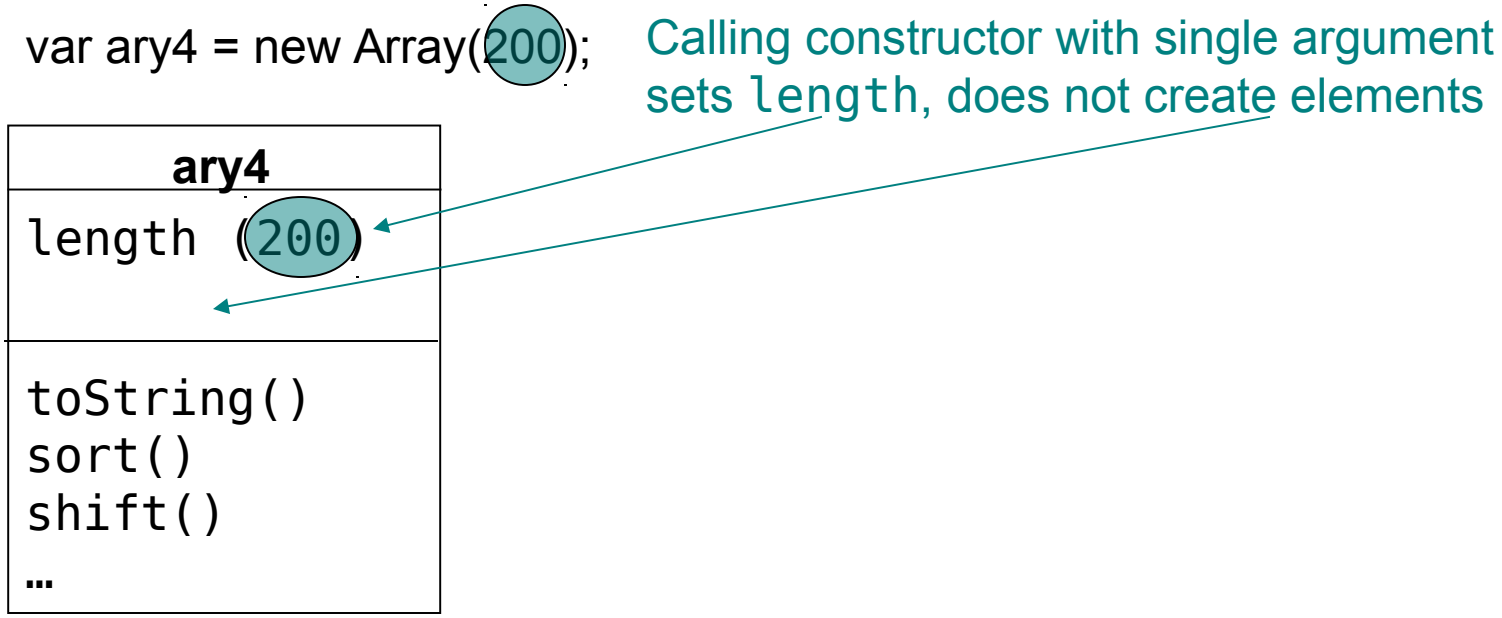
JavaScript Arrays

- Changing the number of elements:



JavaScript Arrays

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

JavaScript Arrays

```
// stack.js
var stack = new Array();
stack.push('H');
stack.push('i');
stack.push('!');

var c3 = stack.pop(); // pops '!'
var c2 = stack.pop(); // pops 'i'
var c1 = stack.pop(); // pops 'H'
window.alert(c1 + c2 + c3);  // displays "Hi!"
```

JavaScript Arrays

```
// stack.js
var stack = new Array();
stack.push('H');
stack.push('i');
stack.push('!');

var c3 = stack.pop(); // pops '!'
var c2 = stack.pop(); // pops 'i'
var c1 = stack.pop(); // pops 'H'
window.alert(c1 + c2 + c3);  // displays "Hi!"
```

push( ) adds an element to the end of the array

JavaScript Arrays

```
// stack.js
var stack = new Array();
stack.push('H');
stack.push('i');
stack.push('!');

var c3 = stack.pop(); // pops '!'
var c2 = stack.pop(); // pops 'i'
var c1 = stack.pop(); // pops 'H'
window.alert(c1 + c2 + c3);  // displays "Hi!"
```

pop( ) deletes and returns last element of the array

JavaScript Arrays

```
// stack.js
var stack = new Array();
stack.push('H');
stack.push('i');
stack.push('!');

var c3 = stack.pop(); // pops '!'
var c2 = stack.pop(); // pops 'i'
var c1 = stack.pop(); // pops 'H'
window.alert(c1 + c2 + c3);  // displays "Hi!"
```

Use `shift()` instead to implement queue

Built-in Objects

- The global object
  - Named `window` in browsers
  - Has properties representing all global variables
  - Other built-in objects are also properties of the global object
    - Ex: initial value of `window.Array` is `Array` object
  - Has some other useful properties
    - Ex: `window.Infinity` represents Number value

Built-in Objects

- The global object and variable resolution:


```
i = 42;
```

What does `i` refer to?

1. Search for local variable or formal parameter named `i`
2. If none found, see if global object (`window`) has property named `i`

- This is why we can refer to built-in objects (`Object`, `Array`, etc.) without prefixing with `window.`

Built-in Objects

- `String()`, `Boolean()`, and `Number()` built-in functions can be called as constructors, created "wrapped" Objects:

```
var wrappedNumber = new Number(5.625);
```

- Instances inherit `valueOf()` method that returns wrapped value of specified type:

```
window.alert(typeof wrappedNumber.valueOf());
```

Output is "number"

Built-in Objects

- Other methods inherited by `Number` instances:

```
var wrappedNumber = new Number(5.625);
window.alert(wrappedNumber.toFixed(2));
window.alert(wrappedNumber.toExponential(2));
window.alert(wrappedNumber.toString(2));
```

Outputs

5.63

5.63e+0

101.101

Base 2

Built-in Objects

- Properties provided by `Number` built-in object:
  - `Number.MIN_VALUE`: smallest (absolute value) possible JavaScript Number value
  - `Number.MAX_VALUE`: largest possible JavaScript Number value

Built-in Objects

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

- Instances of `String` have a `length` property (number of characters)
- JavaScript automatically wraps a primitive value of type Number or String if the value is used as an object:

```
window.alert("a String value".slice(2,5));
```

Output is “Str”

Built-in Objects

- 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

- `valueOf()` method inherited by `Date` instances returns integer representing number of milliseconds since midnight 1/1/1970
- Automatic type conversion allows `Date` instances to be treated as Numbers:

```
var startTime = new Date();
// Perform some processing
...
var endTime = new Date();
window.alert("Processing required " +
             (endTime - startTime)/1000 +
             " seconds.");
```

Built-in Objects

- `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 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)