SNS COLLEGE OF TECHNOLOGY

Coimbatore-35
An Autonomous Institution

Accredited by NBA – AICTE and Accredited by NAAC – UGC with 'A+' Grade
Approved by AICTE, New Delhi & Affiliated to Anna University, Chennai

DEPARTMENT OF INFORMATION TECHNOLOGY

WEB TECHNOLOGY

III YEAR - V SEM

UNIT 2 – CSS AND CLIENT-SIDE SCRIPTING

Topic 6 - Syntax-Variables and Data Types

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

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

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

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:

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

Argument value(s) associated with corresponding formal parameters

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:

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

If final statement executed is return-value, then value of its expression becomes value of the function call

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:
    - Object's effectively passed by reference

```
oneTo(999+1);
```

  - 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

```
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

Local declaration shadows corresponding global declaration

JavaScript Functions

- Local vs. global variables

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

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