JavaScript basics

In This Chapter

- Basic programming terminology
- Execution order
- Case sensitivity
  - Lowercase keywords
  - Camel casing (camel-back)
- Whitespace
  - Whitespace ignored
  - Whitespace not ignored (exception)
- Using semicolons
  - Single statement
  - Multiple statements on one line
  - Multiple statements on multiple lines
- Using curly braces
- Indenting
- Commenting
  - Single line comments
  - Multiple line comments
- Keywords
  - new
  - var
- JavaScript grammar
  - Variables
  - Introduction to declaring variables
  - Data types
    - Primitive data types
      - String
      - Numeric
      - Boolean
      - Undefined
      - NULL
    - Composite data types
      - Object
      - Array
      - Function
    - Weak typing
  - Operators
    - Arithmetic
    - Increment/Decrement
    - Logical
    - Tertiary statement
  - Expressions
  - Statements
  - Objects
  - Properties
  - Functions and methods
- Browser DOM vs. Servoy SOM
  - Object model
  - Document Object Model (DOM)
  - Solutions Object Model (SOM)
- Object references
  - Object
  - Constructor
  - Properties
  - Functions
  - Methods
  - Forms
  - Elements

Introduction to JavaScript

JavaScript was originally developed by Brendan Eichof Netscape under the name Mocha, which was later renamed to LiveScript and finally to JavaScript. JavaScript code, much like other programming languages, is made up of statements which serve to make assignments, compare values and execute other sections of code.

JavaScript:

- Is interpreted line-by-line
- Is case sensitive
- Ignores whitespaces
- Uses semicolon at the end of each statement
- Uses blocks (with curly braces)

- Uses identing in code
- Uses commenting symbols
- Uses keywords

Basic programming terminology

JavaScript has some basic programming terms:

Name	Definition	Examples
Token	The smallest indivisible lexical unit of the language; a contiguous sequence of characters whose meaning would change if the characters were separated by a space.	All identifiers; literals like *3.14* and *'Th is is a string'*.
Literal	A value found directly in the script.	*3.14*, *'This is a string'*
Identifier	The name of a variable, object, function or label.	*x*, *myValue*, *userName*
Operator	Tokens that perform built-in language operations like assignment, addition and substraction.	=, +, -, *
Expressions	A group of tokens; often literals or identifiers, combined with operators that can be evaluated to a specific value.	*2.0*, *'This is a string.'*, *(x + 2) * 4*
Iterations	Executing the same set of instructions a given number of times or until a specific result is attained.	*for (var i = 0; i < 10; i++) // set i = 0, if i < 10, add 1 to i*
Statement	An imperative command, usually causes the state of the execution environment to change.	*var x = x + 2;* *if(x == 4) {* *alert('It is x');* *}*
Keyword	A word that is part of the language, may not be used for identifiers.	*while*, *do*, *function*, *var*
Reserved word	A word that may become part of the language; may not be used as identifiers.	*break*, *import*, *public*

Note

For a complete list of reserved words, please check the TODO: reference to reserved words section.

Execution order

JavaScript is interpreted (executed) line-by-line as it is found in a piece of code (script)

Case sensitivity

JavaScript is case sensitive, meaning that capital letters are distinct from their lowercase counterparts.

For example, the following identifiers refer to separate, distinct values:

- result
- Result
- RESULT

Case sensitivity applies to all aspects of the JavaScript language:

- Keywords
- operators
- Variable names
- Event handlers
- Object properties

Lowercase keywords

All JavaScript keywords are lowercase. For example, when you use a feature like an *if* statement, make sure that you type *if* and not *If* or *IF*.

Camel casing (camel-back)

JavaScript uses the camel casing (mixed casing) naming convention for functions, methods and properties. If a name is one word, then the name is made of all lowercase characters. If more words are used to form a name, the first word is lowercase followed by the next word(s) which all start with a capital letter.

Here some examples:

- displayType
- getTimeZoneOffset
- round

- toFixed

> **Note**
>
> No spaces and punctuations are using with forming these names.

Whitespace

JavaScript applies these rules to the use of the whitespace in coding.

Whitespace ignored

JavaScript ignores those characters that take up space on the screen without visual representation or without necessary meaning.

For example:

```
x = x + 1
```

Can be written like:

```
x=x+1
```

or:

```
x = x + 1
```

Whitespace not ignored (exception)

However, most operations other than simple arithmetic functions require a space to make their meaning clear.

For example:

```
s = typeof x
```

and:

```
s = typeofx
```

do not have the same meaning. The first statement invokes the *typeof* operator on a variable *x* and places the result in *s*. The second statement copies the value of a variable called *typeofx* into *s*.

Whitespaces within a string, between single or double quotes, are preserved. For example:

```
var s = 'This spacing is      preserved.';
```

Using semicolons

A semicolon indicates the end of a JavaScript statement. You can use a semicolon for:

- A single statement
- Multiple statements on one line
- Multiple statements on multiple lines

Single statement

Example:

```
x = x + 1;
```

Multiple statements on one line

Example:

```
x = x + 1; y = y + 1; z = 0;
```

Multiple statements on multiple lines

Example:

```
x = x + 1;
y = y - 1;
```

> **Note**
>
> You do not need to use a semicolon for the last statement of a method (script).

Using curly braces

JavaScript uses curly braces *{}* to group a list of statements together into one larger statement, known as a block statement.

Here an example of statements that make up the body of a function:

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

Here is an example of more than one statement to be executed:

```
if (x > 10) {
        x = 0;
        y = 10;
}
```

Indenting

Here is an *if else* example to show standard identing in JavaScript code:

```
var x = 5;
if (x > 1) {
        if (x > 2) {
                alert('x > 2');
                alert('Yes, x > 2');
        }
} else {
        alert('x <= 1');
        alert('Yes, x <= 1');
}
alert('Moving on.');
```

Here is an explanation, step-by-step of the identing example:

Line	Step	Indenting	Semicolon	Purpose
1	var x = 5;	No	Yes	Declare variable.
2	if (x > 1) {	No	No	First *if* condition with open curly brace for block statement.
3	if (x > 2) {	Forward	No	Second *if* condition with opening curly brace for block statement.
4	alert('x > 2');	Forward	Yes	First statement for second *if* condition.
5	alert('Yes, x > 2');	No	No	Second statement for second *if* condition.
6	}	Back	No	Close curly brace for block statement of second *if* statement.
7	} else {	Back	No	Close curly brace for block statement of first *if* condition, *else* condition and open curly bracket for block statement.
8	alert('x <= 1');	Forward	Yes	First statement for *else* condition.
9	alert('Yes, x > 2');	No	Yes	Second statement for *else* condition.
10	}	Back	No	Close curly brace for block statement of *else* condition.
11	alert('Moving on.');	No	Yes	Final statement of the block.

Commenting

An important aspect of good JavaScript code (and actually all code) is commenting. Commenting means:

- You can insert remarks and commentary directly into your code.
- Included comments will be ignored by the JavaScript interpreter.

Single line comments

Single line comments begin with a double forward slash //. The interpreter will ignore everything from that point until the end of the line.

Example:

```
var count = 10; // holds a number of items
```

Multiple line comments

Multiple line comments (similar to C programming) are enclosed between forward slash and asterix / **at the beginning of the comment and an asterix and forward slash** / at the end of the comment. Everything in between is ignored by the interpreter.

Example:

```
/* Both x = 0 and y = 10
   statements will be executed
   when the if statement is true.
*/
if (x > 10) {
        x = 0;
        y = 10;
}
```

> **Note**
>
> It's not possible to nest comments and will result in an error.

Keywords

Keywords are part of the set of JavaScript reserved words.

> **Note**
>
> For a list of reserved words, please check the "JavaScipt reserved words" section.

new

the keyword *new* enables you to create your own object and is used for an object's constructor.

Example:

```
var myArray = new Array();
myArray[0] = "Thomas";
myArray[1] = "Bob";
```

TODO: reference to "Object references".

var

The keyword *var* is used to declare variables.

Example:

```
var x = 2;
```

JavaScript grammar

Here are the main elements of JavaScript grammar:

- Variables
- Data types
- Operators
- Expressions
- Statements
- Objects
- Properties
- Functions and methods

Variables

Variables are used to stored data. Each variable has a name, called its identifier. Variables are declared in JavaScript using the *var* keyword. This keyword allocates storage space for new data and indicates that a new identifier is in use.

> **Note**
>
> You should not use variables without first declaring them. Using a variable on the right-hand side of an assignment without first declaring it will result in an error.

Introduction to declaring variables

When you declare a variable you can do that with or without assigning a value to it.

This example shows how to declare a variable without assigning a value to it:

```
var x;
```

This example shows how to assign a value when declaring a variable:

```
var x = 0;
```

You can also declare multiple variables by using one *var* statement:

```
var x, y = 1, z;
```

Data types

JavaScript knows two categories of data types:

- Primitive (basic) data types
- Composite data types

Primitive data types

These are primitive or basic data types, which contain one kind of data, for variables:

- String
- Number
- Boolean
- Undefined
- NULL

String

A string is a list of characters. A string literal is indicated by enclosing characters in single quotes ' or double quotes ".

For example:

```
var myString = 'JavaScript has strings.";
```

Any alphabetic, numeric or punctuation characters can be placed in a string.

However, there are some exceptions known as escape codes. An escape code (also called escape sequence) is a small amount of characters preceded by a backslash that has a special meaning to the JavaScript interpreter. Escape codes allow you to enter special representative characters witout typing them directly into your code.

Let's say that you want to declare two lines of text in a variable:

```
var myString = 'This is the first line.
This is the second line';
```

The way this is done will result in a syntax error, because JavaScript interprets the second line as a separate statment. The correct way to do this is by using an escape character:

```
var myString = 'This is the first line.\nThis is the second line.";
```

Here a list of escape codes:

Escape code	Value
\b	Backspace
\t	Horizontal tab

\n	Line feed (new line)
\v	Vertical tab
\f	Form feed
\r	Carriage return
\'	Single quote
\"	Double quote
	Backslash
\OOO	Latin-1 character represented by the octal digits OOO. The valid range is 000 to 377.
\xHH	Latin-1 character represented by the hexidecimal digits HH. The valid range is 00 to FF.
\uHHHH	Unicode character represented by the hexidecimal digits HHHH.

Numeric

Numbers are integers or floating-point numeric values:

```
var myNumber = 3.14;
```

Boolean

Boolean data types take on one of two possible values: *true* or *false*. A boolean literal is indicated by using these values directly in the code:

```
var myBoolean = true;
```

Undefined

An undefined data type is used for variables and object properties that do not exist or have not been assigned a value. The only value an undefined type can have is *undefined*:

```
var x;
var x = String.noSuchProperty;
```

NULL

The NULL value indicates an empty or non-existent value, but the data type is defined. The only value a NULL data type can have is NULL:

```
var x = null;
```


Note

When undefined and NULL data types are being compared, the result will be true.

Composite data types

Composite data types are made up of strings, numbers, booleans, undefined values, null values and even other composite types. These are the three composite types:

- Object
- Array
- Function

Object

An object can hold any type of data and is the primary mechanism by wich tasks are carried out. Data contained in an object are properties for the object. Properties are accessed with the dot operator . followed by the property name:

```
objectName.propertyName
```

Array

The array data type is an ordered set of values grouped together under a single identifier. The easiest way to create an array is to define it as a standard variable and then group the set of values in brackets:

```
var myArray = [1, 5, 68, 3];
```

Arrays can contain data of different data types:

```
var myArray = ['Bob', 5, true, x];
```

To create an array with no length:

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

To create an array with a predetermined length, but no specific data:

```
var myArray = new Array(4);
```

The elements of an array are accessed by providing an index value within brackets. In the following example variable *x* equals *Bob* and variable *y* equals *tr ue*:

```
var myArray = new Array('Bob', 5, true, x);
var x = myArray[0];
var y = myArray[2];
```

Function

Functions are used to keep code that performs a particular job in one place, making the code reusable. In Servoy, built-in functions are accessible from the method editor.

Weak typing

One of the major differences between JavaScript and other programming languages is that JavaScript is weakly typed. This means that the type of variable is inferred from the variable's content. As a result, the data type can change the moment you put data into the variable of another type than it previously contained.

For example, first we assign a string value to the variable and after that we assign a numeric value to it:

```
var x = 'Hello';
x = 5.25;
```

Operators

Basic operators include:

- Arithmetic
- Increment and decrement
- Logical
- Comparison
- Tertiary statement

Arithmetic

Here are the common arithmetic operators:

- Assignment =
- Addition +
- Substraction or unary negation -
- Multiplication *
- Division /
- Modulus %

For example:

```
x = a + b;
x = a - b;
```

Increment/Decrement

This example shows how *a* is set to the value of *b* before *b* is incremented by 1.

```
a = b++;
```

This example shows how *a* is set to the value of *b* after *b* is incremented by 1.

```
a = ++b;
```

This example shows how *a* is set to the value of *b* before *b* is decremented by 1.

```
a = b--;
```

This example shows how *a* is set to the value of *b* after *b* is decremented by 1.

```
a = --b;
```

Logical

Here are some logical operators:

Operator	Example	Description
&&	if (x == 0 && y == 0)	Returns *true* if *x* equals *0* AND *y* equals *0*.
||	if (x == 0 || y == 0)	Returns *true* if *x* equals *0* OR *y* equals _0.

Here are some comparison operators:

Operator	Example	Description
==	a == b	Returns *true* if *a* and *b* are equal.
!=	a != b	Returns *true* if *a* and *b* are not equal.
>	a > b	Returns *true* if *a* is greater than *b*.
<	a < b	Returns *true* if *a* is less than *b*.
>=	a >= b	Returns *true* if *a* is greater than or equal to *b*.
<=	a <= b	Returns *true* if *a* is less than or equal to *b*.

Tertiary statement

Here is a tertiary statement:

```
x = if (test, result one, result two);
```

This example shows if *p* is greater than or is equal to *25* then *x* = *yes* and otherwise *x* = *no*.

```
x = (p >= 25) ? 'yes' : 'no'
```

Expressions

Expressions are the building blocks of many JavaScript statements. Any combination of variables, operators and statements which evaluate some result, like a sentence or a phrase. Literals and variables are the simplest kind of expressions and can be used to create more complex expressions.

For example:

```
var x = 3 + 3;
```

Statements

Statements are the essence of JavaScript. They are a set of instructions to carry our specific actions. JavaScript statements may take the form of conditionals, loop or object manipulations.

One of the most common statements is an assignment. This is a statement that uses the = operator and places the value on the right-hand side into the variable on the left. Here is an example of an assignment:

```
x = y + 10;
```

> **Note**
>
> The assignment operator = should not be confused with the "is equal to" comparison operator == which is used in conditional expressions.

Objects

Objects are a critical concept and feature of JavaScript. An object is a collection that can be primitive or composite data, inclusing functions and other objects.

Objects are created using constructors. These are mthods that create a fresh instance of an object for you to use. Objects can be created by using the *new* keyword, the name of the object and open and closed parentheses. The parentheses tell the interpreter that you want to invoke the constructor method for the given object.

This example shows the creation of a new string object:

```
var myString = new String();
```

> **Note**
>
> Servoy can reference a large number of built-in JavaScript objects.

Properties

The members of an object are called properties. Properties are accessed by placing a period . immediately following the name of the object.

For example, *length* is a property of the object *string*:

```
myString.length
```

Functions and methods

A JavaScript function is quite similar to a procedure or subroutine in other programming languages. A function is a discrete set of statements which perfom some actions. It may accept incoming values (parameters) and it may return an outgoing value.

a function is called from a JavaScript statement to perform its duty.

A method is simply a function which is contained in an object and is a member function of the object. Methods of objects are accessed the same way as properties, with trailing parentheses immediately following the name of the method. If the method takes arguments (parameters), the arguments are included in the parentheses.

For example:

```
application.output('Hello World.');
```

> ⚠ **Note**
>
> In Servoy, both methods and functions are accessible from the Servoy Solution Explorer.

Browser DOM vs. Servoy SOM

The Servoy Solutions Object Model (SOM) and the W3C Document Object Model (DOM) are both object models.

Object model

An object model:

- Defines the interface used by scripts to look at and manipulate structured information, like an HTML document.
- Determines the make-up and characteristics of an object's component parts.
- Defines how the parts work.

Document Object Model (DOM)

The DOM is primarily defined by the last two kinds of objects; browser objects and document objects.

In the past, different browsers have included or modified slightly different features for manipulating document objects. However, these objects now have been standardized by the World Wide Web Consortium (W3C) Document Object Model.

The Document Object Model (DOM) is a standardized Application Programming Interface (API) with a platform-independent language, whose intent is to:

- Enabled programmers to write applications that work with all browsers and servers on any platform.
- Allow programs to dynamically connect and update the content, structure and style of a document.
- Define a tree-like representation of the document (also referred to as the DOM tree) so the user can interact with any part of the hierarchy of elements.

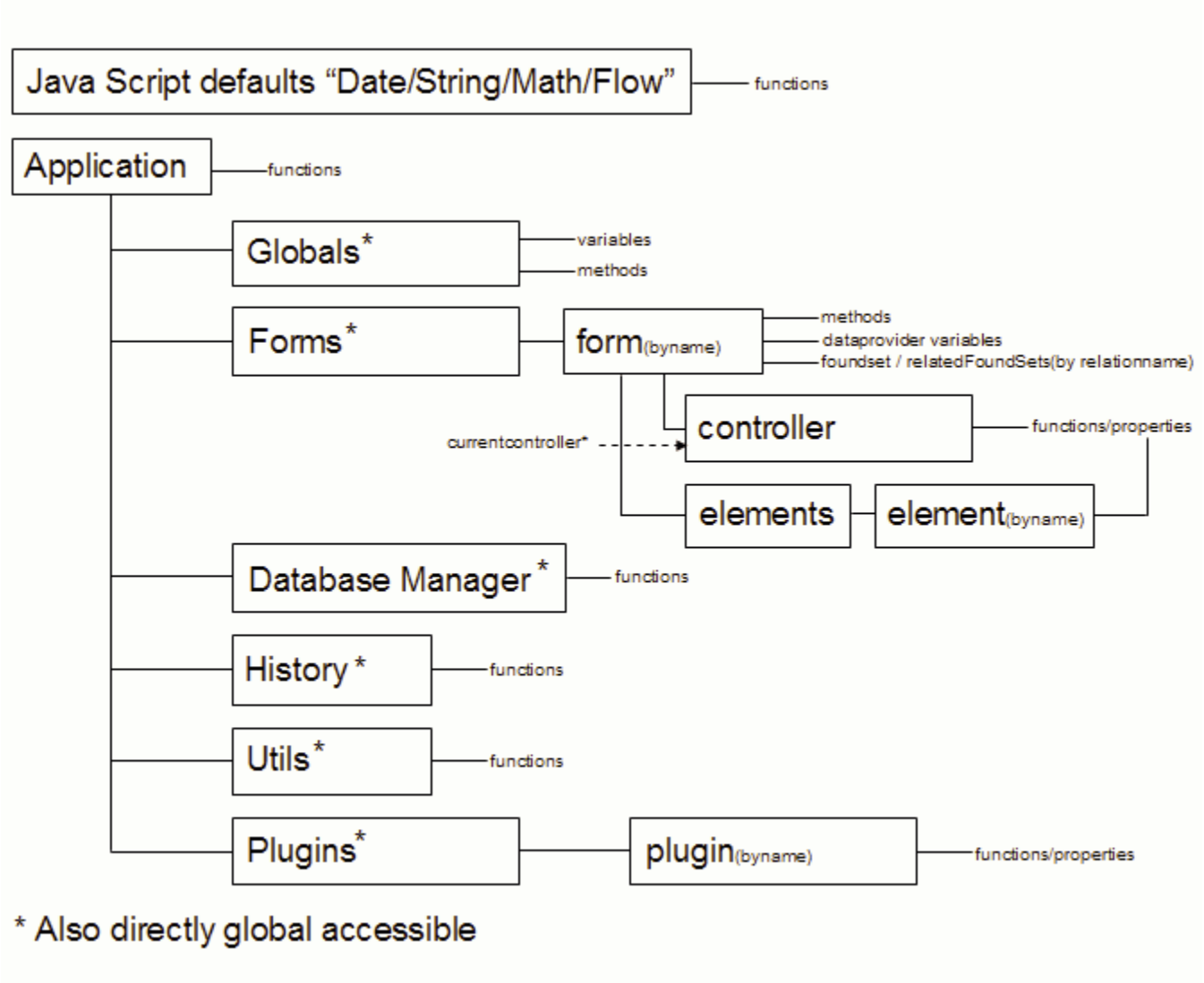
Solutions Object Model (SOM)

The Solutions Object Model (SOM) provides a JavaScript-based onject model for Servoy solutions, defined in large part by the first two kinds of JavaScript objects explained above (user-defined objects and buiilt-in objects). Similar to the DOM, the SOM core defines a tree-like representation of Servoy, also known as the SOM tree. All elements are related to each other in hierarchical way.

The DOM is limited to being defined by and acting on browsers and browser documents.

Servoy is a rapid application development (RAD) and deployment environment heavily focused on database applications.

Here an overview of the Servoy SOM:



Object references

Here is a list of the JavaScript object references and how they apply to Servoy:

- Object
- Constructor
- Properties
- Functions
- Methods
- Forms
- Elements

Object

An object is a collection that can hold any type of data, including functions and other objects and is the primary mechanism for carrying our useful tasks.

Constructor

Objects are created using constructors. These are methods that create a fresh instance of an object.

For example:

```
var myObject = new Object();
```

Properties

Data contained in an object are said to be properties of the object. Properties are accessed with a dot . operator (a period) followed by the property name.

Syntax:

```
objectName.propertName
```

Functions

Functions are a special type of JavaScript object that contains executable code. A function is called (invoked) by following the function name with parentheses *()*. The parenthesis can contain parameters (arguments) which are pieces of data that are passed to the function when it is invoked.

Syntax:

```
objectName.functionName()
```

> **Note**
>
> For a complete list of Servoy built-in functions, please refer to the TODO: Programming Reference Guide -> Runtime API section.

Methods

Functions that are part (a member) of objects are know as methods.

Syntax:

```
objectName.methodName
```

Forms

The forms collection contains objects that reference all forms in a solution.

If the form you are referencing is the currently active form, the syntax is:

```
controller.methodName
```

If the form you are referencing is not the currently active form, the syntax is:

```
forms.formName.controller.methodName
```

Elements

The elements collection contains objects contained in the form. You can access elments such as fields, buttons, labels, images, portals, tabpanels and JavaBeans.

If the form you are referencing is the currently active form, the syntax is:

```
elementName.propertyName
```

If the form you are referencing is notthe currently active form, the syntax is:
Syntax:

```
forms.formName.elementName.propertyName
```