MICROSOFT DESIGN PROPOSALS FOR THE ECMASCRIPT 2.0 LANGUAGE SPECIFICATION

ECMA COMMITTEE #39
VERSION 0.1
JULY 10, 1997
1 INTRODUCTION

Vendors are continuing to innovate in areas that we have decided not to deal with for ECMAScript 1.0, and will ship these features before we will have an ECMAScript 2.0 standard. Microsoft seeks to avoid a repetition of the behaviors that caused the current unfortunate situation -- multiple incompatible implementations from multiple vendors, and no sensible language design in the end. If all vendors are open about their plans, and solicit and use feedback from other committee members, then this will not be a problem. We do not need to complete a formal standard before people ship functionality, but we do need to have open discussion. If we do not do this, both the language and our customers will suffer.

Microsoft encourages all vendors to work together in this manner. The first step is to contribute detailed design proposals for ECMAScript 2.0 features. Our proposals are found in this document.
2 DETAILS

2.1 CONVENTIONS AND SEMANTIC BUILDING BLOCKS
This document follows the same conventions and semantic building blocks as the ECMAScript 1.0 specification, which is still in development.

2.2 FEEDBACK
Please send feedback regarding this document to Scott Wiltamuth (scottwil@microsoft.com).
3 NEW OPERATORS

3.1 EXTENSION OF PUNCTUATORS

Punctuators is extended to include === and !==.

Syntax

\[
Punctuator :: \text{one of } \\
= \quad > \quad < \quad == \quad <= \quad >= \\
!= \quad , \quad ! \quad \sim \quad ? \quad :: \\
. \quad && \quad || \quad ++ \quad -- \quad + \\
- \quad * \quad / \quad \& \quad ^ \quad ^\uparrow \\
\% \quad << \quad >> \quad >>> \quad += \quad -= \\
*= \quad /= \quad &= \quad \| \quad ^= \quad %= \\
\langle= \quad \rangle= \quad \rangle\rangle= \quad ( \quad ) \quad \{ \quad \} \quad \{ \quad \} \quad === \quad !==
\]

3.2 EXTENSION OF EqualityExpression

EqualityExpression is extended to include === and !==.

Syntax

\[
\text{EqualityExpression} : \\
\text{RelationalExpression} \\
\text{EqualityExpression} == \text{RelationalExpression} \\
\text{EqualityExpression} === \text{RelationalExpression} \\
\text{EqualityExpression} != \text{RelationalExpression} \\
\text{EqualityExpression} !== \text{RelationalExpression}
\]

The production \( \text{EqualityExpression} : \text{EqualityExpression} === \text{RelationalExpression} \) is evaluated as follows:
1. Evaluate \( \text{EqualityExpression} \).
2. Call \( \text{GetValue(Result(1))} \).
3. Evaluate \( \text{RelationalExpression} \).
4. Call \( \text{GetValue(Result(3))} \).
5. Perform the comparison \( \text{Result(4)} === \text{Result(2)} \). (See below.)
6. Return Result(5).

The production \( \text{EqualityExpression} : \text{EqualityExpression} !== \text{RelationalExpression} \) is evaluated as follows:
1. Evaluate \( \text{EqualityExpression} \).
2. Call \( \text{GetValue(Result(1))} \).
3. Evaluate \( \text{RelationalExpression} \).
4. Call \( \text{GetValue(Result(3))} \).
5. Perform the comparison \( \text{Result(4)} === \text{Result(2)} \). (See below.)
6. If Result(5) is \text{true}, return \text{false}. Otherwise, return \text{true}. 
The comparison \( x === y \), where \( x \) and \( y \) are values, produces \textbf{true} or \textbf{false}. Such a comparison is performed as follows:

1. If \( \text{Type}(x) \) is different from \( \text{Type}(y) \), return \textbf{false}.
2. If \( \text{Type}(x) \) is not Number, go to step 9.
3. If \( x \) is NaN, return \textbf{false}.
4. If \( y \) is NaN, return \textbf{false}.
5. If \( x \) is the same number value as \( y \), return \textbf{true}.
6. If \( x \) is \(+0\) and \( y \) is \(−0\), return \textbf{true}.
7. If \( x \) is \(−0\) and \( y \) is \(+0\), return \textbf{true}.
8. Return \textbf{false}.
9. If \( \text{Type}(x) \) is String, then return \textbf{true} if \( x \) and \( y \) are exactly the same sequence of characters (same length and same characters in corresponding positions). Otherwise, return \textbf{false}.
10. If \( \text{Type}(x) \) is Boolean, return \textbf{true} if \( x \) and \( y \) are both \textbf{true} or both \textbf{false}. Otherwise, return \textbf{false}.
11. Return \textbf{true} if \( x \) and \( y \) refer to the same object. Otherwise, return \textbf{false}.
12. Return \textbf{false}.
4 NEW CONTROL FLOW CONSTRUCTS

4.1 EXTENSION OF STATEMENTS

Statement is extended to include SwitchStatement and LabeledStatement.

Statement:
  Block
  VariableStatement
  EmptyStatement
  ExpressionStatement
  IfStatement
  IterationStatement
  ContinueStatement
  BreakStatement
  ReturnStatement
  WithStatement
  LabeledStatement
  SwitchStatement

4.2 EXTENSION OF THE COMPLETION TYPE

The Completion type is extended in order to define labeled break and continue.

The original Completion type values, as specified in the first version of the standard, are:

- “normal completion”
- “normal completion after value V”
- “abrupt completion because of break”
- “abrupt completion after value V because of break”
- “abrupt completion because of continue”
- “abrupt completion after value V because of continue”
- “abrupt completion because of return V” where V is a value

The new Completion type values are:

- “abrupt completion because of break with label I” where I is an identifier.
- “abrupt completion after value V because of break with label I” where I is an identifier.
- “abrupt completion because of continue with label I” where I is an identifier.
- “abrupt completion after value V because of continue with label I” where I is an identifier.

4.3 THE switch STATEMENT

Syntax

SwitchStatement:
  switch (Expression) CaseBlock

CaseBlock:
  { CaseClauses }
{ CaseClauses opt DefaultClause CaseClauses opt }

CaseClauses :
   CaseClause
   CaseClauses CaseClause

CaseClause :
   case Expression : StatementList opt

DefaultClause :
   default : StatementList opt

Semantics
The production SwitchStatement : switch ( Expression ) CaseBlock is evaluated as follows:
1. Evaluate Expression.
2. Call GetValue(Result(1)).
3. Evaluate CaseBlock, passing it Result(2) as a parameter.
4. If Result(3) is “abrupt completion because of break”, return “normal completion”.
5. If Result(3) is “abrupt completion after value V because of break”, return “normal completion after value V”.
6. Return Result(3).

The production CaseBlock : { CaseClauses, DefaultClause CaseClauses } is given an input parameter, input, and is evaluated as follows:
1. Let A be the list of CaseClause items in CaseClauses1, in source text order.
2. For the next CaseClause in A, evaluate CaseClause. If there is no such CaseClause, go to step 7.
3. If input is not equal to Result(2), as defined by the !== operator, go to step 2.
4. Evaluate the StatementList of this CaseClause.
5. If Result(4) is an abrupt completion then return Result(4).
7. Let B be the list of CaseClause items in CaseClauses2, in source text order.
8. For the next CaseClause in B, evaluate CaseClause. If there is no such CaseClause, go to step 15.
9. If input is not equal to Result(8), as defined by the !== operator, go to step 8.
10. Evaluate the StatementList of this CaseClause.
11. If Result(10) is an abrupt completion then return Result(10).
13. For the next CaseClause in A, evaluate the StatementList of this CaseClause. If there is no such CaseClause, go to step 15.
14. If Result(13) is an abrupt completion then return Result(13).
15. Execute the StatementList of DefaultClause.
16. If Result(15) is an abrupt completion then return Result(15).
17. Let B be the list of CaseClause items in CaseClauses2, in source text order.
18. For the next CaseClause in B, evaluate the StatementList of this CaseClause. If there is no such CaseClause, return “normal completion”.
19. If Result(18) is an abrupt completion then return Result(18).
20. Go to step 18.

The production CaseClause : case Expression : StatementList opt is evaluated as follows:
1. Evaluate Expression.
2. Call GetValue(Result(1)).
3. Return Result(2).

Note that evaluating CaseClause does not execute the associated StatementList. It simply evaluates the Expression and the value, which the CaseBlock algorithm uses to determine which StatementList to start executing.
4.4 **Labeled Statements**

**Syntax**

LabeledStatement:

  Identifier : Statement

**Semantics**

A Statement may be prefixed by a label. Labeled statements are only used in conjunction with labeled break and continue. ECMAScript has no goto statement. An ECMAScript program is considered syntactically incorrect, and may not be executed at all, if it contains a LabeledStatement that is enclosed by a LabeledStatement with the same Identifier label. LabeledStatement is evaluated as:

1. Evaluate Statement.
2. If Result(1) is “abrupt completion because of break with label I returns” and I is the same as Identifier, then return “normal completion”.
3. If Result(1) is “abrupt completion after value V because of break with label I” and I is the same as Identifier, then return “normal completion after value V”.
4. If Result(1) is “abrupt completion because of continue with label I” and I is the same as Identifier, then go to step 7.
5. If Result(1) is “abrupt completion after value V because of continue with label I” and I is the same as Identifier, then go to step 7.
6. Return Result(1).
7. Apply the continue operation to Statement.
8. Return Result(7).

**Issue:** The “Apply the continue operation to Statement.” needs to be defined in detail. It is not possible to simple re-execute Statement, since a statement may carry context forward from the execution that resulted in a continue. The context is different for each type of statement, so it will be necessary to do work for each statement type with which continue can be employed.

4.5 **The break Statement**

**Syntax**

BreakStatement:

  break [no LineTerminator here] Identifieropt ;

**Semantics**

The BreakStatement without the optional Identifier is as defined in the ECMAScript 1.0 standard. When the identifier is present, it specifies a label that indicates the LabeledStatement to which the break applies. This LabeledStatement is known as the break target. A program is considered syntactically incorrect, and may not be executed at all, if any of the following are true:

- It contains a break statement without the optional identifier, and the break statement is not contained within at least one IterationStatement or SwitchStatement.
- It contains a break statement with the optional Identifier, where Identifier does not match the Identifier label of an enclosing LabeledStatement.

The break statement with the optional Identifier is evaluated as:

1. Return “abrupt completion because of break with label Identifier”.

**Issue:** The ECMAScript 1.0 specification states “An ECMAScript program is considered syntactically incorrect and may not be executed at all if it contains a break statement that is not within at least one while or for statement.” This needs to be altered for several reasons. First, because the statement as written applies only to a break statement without the optional Identifier. Second, because we have added new IterationStatement types that should be included along with while and for statements. Third, because we added SwitchStatement, and break can be used within a SwitchStatement.
4.6 The continue Statement

Syntax

```
ContinueStatement:
  continue [no LineTerminator here] Identifier_opt ;
```

Semantics

The `ContinueStatement` without the optional `Identifier` is as defined in the ECMAScript 1.0 standard. When the identifier is present, it specifies a label indicating the `LabeledStatement` to which the `continue` applies. This `LabeledStatement` is known as the `continue target`.

A program is considered syntactically incorrect, and may not be executed at all, if either of the following are true:

- The program contains a `continue` statement with the optional `Identifier`, where `Identifier` does not match the `Identifier` label of an enclosing `LabeledStatement`.
- The `Statement` within the `continue target` is not an `IterationStatement`.

The `continue` statement with the optional `Identifier` is evaluated as:

1. Return “abrupt completion because of `continue` with label `Identifier`”.

Issue: The ECMAScript 1.0 specification says explicitly “An ECMAScript program is considered syntactically incorrect and may not be executed at all if it contains a `continue` statement that is not within at least one `while` or `for` statement.” This is a bit too specific, as it does not include the `IterationStatement` types that are being added as part of the 2.0 specification. This sentence should be changed to “An ECMAScript program is considered syntactically incorrect and may not be executed at all if it contains a `continue` statement that is not within at least one `IterationStatement`.”

4.7 Extension of Iteration Statements

`IterationStatement` is extended to include the `do...while` statement.

Syntax

```
IterationStatement:
  do Statement while ( Expression )
  while ( Expression ) Statement
  for ( Expression_opt ; Expression_opt ; Expression_opt ) Statement
  for ( var VariableDeclarationList ; Expression_opt ; Expression_opt ) Statement
  for ( var Identifier Initializer_opt in Expression ) Statement
```

4.8 The do...while Statement

The production `do Statement while ( Expression )` is evaluated as follows:

1. Let `C` be “normal completion”.
2. Evaluate `Statement`.
3. If `Result(2)` is a value completion, change `C` to be “normal completion after value `V`” where `V` is the value carried by `Result(2)`.
4. If `Result(2)` is a `return` completion, return `Result(2)`.
5. If `Result(2)` is a `break` completion, go to step 10.
7. Call `GetValue(Result(6))`.
8. Call `ToBoolean(Result(7))`.
9. If `Result(8)` is `true`, go to step 2.
10. Return `C`
5 NEW EXECUTION CONTEXT FEATURES

5.1 CALLER

Section 10.1.8 ("Arguments object") defines what the arguments object. When control enters an execution context for declared function code, anonymous code, or host code, an arguments object is created and initialized according to the steps specified. The “caller” property is added by appending the following step to the already-existing list:

- A property is created with name “caller” and property attributes { DontEnum }. The initial value of this property is the caller’s activation. In the case that the current function was called from global code, the caller property is given an initial value of null.

Issue: Need to allow the “foo.caller” usage as well, not just “arguments.caller”.
6 REFERENCES
