12.6.1 **The do..while Statement**

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

1. Let $V = \text{empty}$.
2. Evaluate `Statement`.
3. If `Result(2).value` is not `empty`, let $V = \text{Result(2).value}$.
4. If `Result(2).type = \text{continue}` and `Result(2).target` is in the current label set, go to 2-step 7.
5. If `Result(2).type = \text{break}` and `Result(2).target` is in the current label set, return `(normal, V, empty)`.
6. If `Result(2)` is an abrupt completion, return `Result(2)`.
7. Evaluate `Expression`.
8. Call `GetValue(\text{Result(7)})`.
9. Call `ToBoolean(\text{Result(8)})`.
10. If `Result(9)` is true, go to step 2.
11. Return `(normal, V, empty)`.

12.7 **The continue statement**

**Syntax**

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

**Semantics**

A program is considered syntactically incorrect if either of the following are true:

∑ The program contains a `continue` statement without the optional `Identifier`, which is not nested, directly or indirectly (but not crossing function boundaries), within an `IterationStatement`.

∑ The program contains a `continue` statement with the optional `Identifier`, where `Identifier` does not appear in the label set of an enclosing (but not crossing function boundaries) `IterationStatement`.

A `ContinueStatement` without an `Identifier` is evaluated as follows:

1. Return `(continue, empty, empty)`.

A `continue` statement with the optional `Identifier` is evaluated as follows:

1. Return `(continue, empty, Identifier)`.

12.8 **The break statement**

**Syntax**

```
BreakStatement : break [no LineTerminator here] Identifier_opt ;
```

**Semantics**

A program is considered syntactically incorrect if either of the following are true:

∑ The program contains a `break` statement without the optional `Identifier`, which is not nested, directly or indirectly (but not crossing function boundaries), within an `IterationStatement` or a `SwitchStatement`.

∑ The program contains a `break` statement with the optional `Identifier`, where `Identifier` does not appear in the label set of an enclosing (but not crossing function boundaries) `Statement`.

A `BreakStatement` without an `Identifier` is evaluated as follows:
1. Return \((\text{break, empty, empty})\).

A \text{break} statement with an \text{Identifier} is evaluated as follows:
1. Return \((\text{break, empty, Identifier})\).

\textbf{12.10 The with statement}

**Syntax**

\text{WithStatement} :
\[ \text{with ( Expression ) Statement} \]

**Description**

The \text{with} statement adds a computed object to the front of the scope chain of the current execution context, then executes a statement with this augmented scope chain, then restores the scope chain.

**Semantics**

The production \text{WithStatement} : \text{with ( Expression ) Statement} is evaluated as follows:
1. Evaluate Expression.
2. Call GetValue(Result(1)).
3. Call ToObject(Result(2)).
4. Add Result(3) to the front of the scope chain.
5. Evaluate \text{Statement} using the augmented scope chain from step 4.
6. Remove Result(3) from the front of the scope chain.
7. Return Result(5).

**Discussion**

Note that no matter how control leaves the embedded \text{Statement}, whether normally or by some form of abrupt completion, the start of the scope chain is always restored to its former state.

\textbf{12.14 The try statement}

**Syntax**

\text{TryStatement} :
\[ \text{try Block CatchList} \]
\[ \text{try Block Finally} \]
\[ \text{try Block CatchList Finally} \]

\text{CatchList} :
\[ \text{Catch} \]
\text{CatchList Catch}

\text{Catch} :
\[ \text{catch (Identifier CatchGuard{opt}) Block} \]
**CatchGuard**: ```Expression```

**Finally**: ```finally Block```

**Description**

The `try` statement encloses a block of code in which an exceptional condition can occur, such as a runtime error or a `throw` statement. The `catch` clauses provide the exception-handling code. Entering a catch clause is similar to calling a function: there is a new execution context and the binding of a value to a formal parameter. The `finally` clause is executed just before control `finally` leaves a try block (that is, after any exception-handling code has been executed).

**Semantics**

The production `TryStatement : try Block CatchList` is evaluated as follows:

1. Evaluate `Block`.
2. If Result(1).type is not `throw`, return Result(1).
3. Evaluate `CatchList` with parameter Result(1).
4. If Result(3) = `(throw, empty, empty)`, return Result(1)
5. Return Result(3).

The production `TryStatement : try Block Finally` is evaluated as follows:

1. Evaluate `Block`.
2. Evaluate `Finally`.
3. If Result(2).type is `normal`, return Result(1).
4. Return Result(2).

The production `TryStatement : try Block CatchList Finally` is evaluated as follows:

1. Evaluate `Block`.
2. Let `C` = Result(1).
3. If Result(1).type is not `throw`, go to step 6.
4. Evaluate `CatchList` with parameter Result(1).
5. Let `C` = Result(4).
6. If Result(4) = `(throw, empty, empty)`, let `C` = Result(1).
7. Evaluate `Finally`.
8. If Result(7).type is `normal`, return `C`.
9. Return Result(7).

The production `CatchList : Catch` is evaluated as follows:

1. Evaluate `Catch` passing it the parameter passed to this production.
2. Return Result(1).

The production `CatchList : CatchList Catch` is evaluated as follows:

1. Evaluate `CatchList` passing it the parameter passed to this production.
2. If Result(1) is not `(throw, empty, empty)`, return Result(1).
3. Evaluate `Catch` passing it the parameter passed to this production.
4. Return Result(2).

The production `Catch : catch (Identifier CatchGuard)opt Block` is evaluated as follows:

1. Let `C` = `(throw, empty, empty)`.
2. Create a new Object object.
3. Call the `[Put]` method of Result(2) with parameters `Identifier` and `C.value`.
4. Add Result(2) to the front of the scope chain.
5. If there is no `CatchGuard`, go to step 10.
6. Evaluate \textit{CatchGuard}.
7. If an exception value $W$ was thrown during the evaluation of \textit{CatchGuard}, go to step 13.
8. If a runtime error occurred during the evaluation of \textit{CatchGuard}, go to step 15.
9. If \texttt{ToBoolean(Result(6))} is not true, go to step 17.
10. Evaluate \textit{Block}.
11. If an exception value $W$ was thrown during the evaluation of \textit{Block}, go to step 15.
12. If a runtime error occurred during the evaluation of \textit{Block}, go to step 17.
13. Let $C = \text{Result(10)}$.
14. Go to step 17.
15. Let $C = (\text{throw}, W, \text{empty})$ where $W$ is the exception value thrown during the evaluation of \textit{CatchGuard}.
16. Go to step 17.
17. Construct an appropriate Error object.
18. Let $C = (\text{throw}, \text{Result(15)}, \text{empty})$.
19. Remove \text{Result(2)} from the front of the scope chain.
20. Return $C$.

The production \textit{CatchGuard} : if Expression is evaluated as follows:
1. Evaluate Expression.
2. Return Result(1).

The production Finally : finally Block is evaluated as follows:
1. Evaluate Finally.
2. Return Result(1).

\textbf{Discussion}

An implementation of ECMAScript is permitted to evaluate the \textit{CatchGuards} early, at the time step 3 of the \texttt{throw} statement's algorithm (section 12.13) is evaluated. In this case the \textit{CatchGuards} would be evaluated with the same scope and environment that they would have normally (as invoked via the \textit{Catch} production at the regular time). On the other hand, if the \textit{CatchGuards} are evaluated early, they might be evaluated before finally blocks nested between the \texttt{throw} and the \textit{CatchGuards}' catch, so the \textit{CatchGuards} would not see the finally blocks' side effects, if any. If a \textit{CatchGuard} throws an exception or causes an error when being evaluated early, that exception or error is delayed until the time when that \textit{CatchGuard} would be evaluated normally; if that \textit{CatchGuard} would never be evaluated normally (because, say, an intervening finally block returns abnormally), the \textit{CatchGuard}'s error or exception is ignored altogether. If a \textit{CatchGuard} indicates that it would handle a particular exception, enclosing \textit{CatchGuards} are not called for that exception.

In addition to evaluating \textit{CatchGuards} early, an implementation is allowed to evaluate the same \textit{CatchGuard} multiple times -- for example, once at the time of the \texttt{throw}, and once when the \textit{Catch} production evaluates the \textit{CatchGuard}. Because of this possibility, ECMAScript users should not write programs that cause side effects from within \textit{CatchGuards}. If the same \textit{CatchGuard} returns contradictory results when called more than once, an implementation is allowed to use any of the results, at its discretion. If the same \textit{CatchGuard} throws an exception or error only some but not all of the time when called more than once, an implementation is allowed to either propagate the \textit{CatchGuard}'s error or exception or ignore it, at its discretion.

The above flexibility also applies if an error is generated and converted into an implicitly thrown exception.

\textbf{Note}: The above flexibility is provided to allow ECMAScript debuggers to intercept uncaught exceptions at the point of the \texttt{throw} instead of at the top level of a program. One possible conforming implementation of ECMAScript would be to evaluate dynamically enclosing \textit{CatchGuards} when an exception is about to be thrown; if any of them indicates that it would catch the exception (or if any of them itself throws an exception or error), the results (and exception or error) of the \textit{CatchGuards} are dropped and the unwinding process begins normally.
(which will evaluate the CatchGuards again to determine where the exception should be caught). If none of the CatchGuards indicates that the exception should be caught, the debugger is entered at the point of the throw. This technique erroneously enters the debugger in the case where a finally block would intercept the propagating exception, but this is poor programming style and likely to be rare in practice.