# Changes To ECMA-262 Specification

### 8.6.2 Internal Properties and Methods

Internal properties and methods are not exposed in the language. For the purposes of this document, their names are enclosed in double square brackets [[ ]]. When an algorithm uses an internal property of an object and the object does not implement the indicated internal property, a runtime error is generated.

There are two types of access for exposed properties: *get* and *put*, corresponding to retrieval and assignment, respectively.

Native ECMAScript objects have an internal property called [[Prototype]]. The value of this property is either **`null`** or an object and is used for implementing inheritance. Properties of the [[Prototype]] object are exposed as properties of the child object for the purposes of get access, but not for put access.

The following table summarises the internal properties used by this specification. The description indicates their behaviour for native ECMAScript objects. Host objects may implement these internal methods with any implementation-dependent behaviour, or it may be that a host object implements only some internal methods and not others.

| ***Property*** | ***Parameters*** | ***Description*** |
|---|---|---|
| [[Prototype]] | none | The prototype of this object. |
| [[Class]] | none | A string value indicating the kind of this object. |
| [[Value]] | none | Internal state information associated with this object. |
| [[Get]] | (PropertyName) | Returns the value of the property. |
| [[Put]] | (PropertyName, Value) | Sets the specified property to Value. |
| [[CanPut]] | (PropertyName) | Returns a boolean value indicating whether a [[Put]] operation with the specified PropertyName will succeed. |
| [[HasProperty]] | (PropertyName) | Returns a boolean value indicating whether the object already has a member with the given name. |
| [[Delete]] | (PropertyName) | Removes the specified property from the object. |
| [[DefaultValue]] | (Hint) | Returns a default value for the object, which should be a primitive value (not an object or reference). |
| [[Construct]] | a list of argument values provided by the caller | Constructs an object. Invoked via the **`new`** operator. Objects that implement this internal method are called *constructors*. |
| [[Call]] | a list of argument values provided by the caller | Executes code associated with the object. Invoked via a function call expression. Objects that implement this internal method are called *functions*. |
| [[HasInstance]] | (Value) | Returns a boolean value indicating whether the Value delegates behaviour to this object. Of the native ECMAScript objects, only Function objects implement [[HasInstance]]. |
| [[Closure]] | none | A scope chain that defines the environment in which a Function object is executed. |

Every object must implement the [[Class]] property and the [[Get]], [[Put]], [[HasProperty]], [[Delete]], and [[DefaultValue]] methods, even host objects. (Note, however, that the [[DefaultValue]] method may, for some objects, simply generate a runtime error.)

The value of the [[Prototype]] property must be either an object or **null**, and every [[Prototype]] chain must have finite length (that is, starting from any object, recursively accessing the [[Prototype]] property must eventually lead to a **null** value). Whether or not a native object can have a host object as its [[Prototype]] depends on the implementation.

The value of the [[Class]] property is defined by this specification for every kind of built-in object. The value of the [[Class]] property of a host object may be any value, even a value used by a built-in object for its [[Class]] property. Note that this specification does not provide any means for a program to access the value of a [[Class]] property; that value is used internally to distinguish different kinds of built-in objects.

Every native object implements the [[Get]], [[Put]], [[CanPut]], [[HasProperty]], and [[Delete]] methods in the manner described in sections **Error! Reference source not found.**, **Error! Reference source not found.**, **Error! Reference source not found.**, **Error! Reference source not found.**, and **Error! Reference source not found.**, respectively, except that Array objects have a slightly different implementation of the [[Put]] method (section **Error! Reference source not found.**). Host objects may implement these methods in any manner; for example, one possibility is that [[Get]] and [[Put]] for a particular host object indeed fetch and store property values but [[HasProperty]] always generates **false**.

In the following algorithm descriptions, assume *O* is a native ECMAScript object and *P* is a string.

## 8.9 The Completion Type

***The internal Completion type is not a language data type****.* It is defined by this specification purely for expository purposes. An implementation of ECMAScript must behave as if it produced and operated upon Completion values in the manner described here. However, a value of the Completion type is used only as an intermediate result of statement evaluation and cannot be stored as the value of a variable or property.

The Completion type is used to explain the behaviour of statements (**`break`**, **`continue`**, **`return`** and **`throw`**) that perform nonlocal transfers of control. Values of the Completion type are triples of the form (*type*, *value*, *target*), where *type* is one of **normal**, **break**, **continue**, **return**, or **throw**, *value* is any ECMAScript value, or **empty**, and *target* is any ECMAScript identifier, or **empty**.

The term "abrupt completion" refers to any completion with a reason value other than **normal**.

### 10.1.2 Types of Executable Code

There are five types of executable ECMAScript source text:

- *Global code* is source text that is treated as an ECMAScript *Program*. The global code of a particular *Program* consists does not include any source text that is parsed as part of a nested *FunctionBody*.

- *Eval code* is the source text supplied to the built-in **`eval`** function. More precisely, if the parameter to the built-in **`eval`** function is a string, it is treated as an ECMAScript *Program*. The eval code for a particular invocation of **`eval`** is the global code portion of the string parameter.

- *Function code* is source text that is parsed as part of a *FunctionBody*. The *function code* of a particular *FunctionBody* does not include any source text that is parsed as part of a nested *FunctionBody*.

- *Anonymous code* is the source text supplied when instantiating **`Function`**. More precisely, the last parameter provided in an instantiation of **`Function`** is converted to a string and treated as the *FunctionBody*. If more than one parameter is provided in an instantiation of **`Function`**, all parameters except the last one are converted to strings and concatenated together, separated by commas. The resulting string is interpreted as the *FormalParameterList* for the *FunctionBody* defined by the last parameter. The *anonymous code* for a particular instantiation of a **`Function`** does not include any source text that is parsed as part of a nested *FunctionBody*.

- *Implementation-supplied code* is the source text supplied by the host when creating an implementation-supplied function. The source text is treated as a *FunctionBody*. Depending on the implementation, the host may also supply a *FormalParameterList*. The *implementation-supplied code* of a particular function does not include any source text that is parsed as part of a nested *FunctionBody*.

### 10.1.3 Variable Instantiation

Every execution context has associated with it a variable object. Variables and functions declared in the source text are added as properties of the variable object. For function, anonymous, and implementation-supplied code, parameters are added as properties of the variable object.

Which object is used as the variable object and what attributes are used for the properties depends on the type of code, but the remainder of the behaviour is generic. On entering an exection context, the properties are bound to the variable object in the following order:

- For function code, anonymous code, and implementation-supplied code, for each formal parameter as defined in the *FormalParameterList*, create a property of the variable object whose name is the *Identifier* and whose attributes are determined by the type of code. The values of the parameters are supplied by the caller as arguments to [[Call]]. If the caller supplies fewer parameter values than there are formal parameters, the extra formal parameters have value **`undefined`**. If two or more formal parameters share the same name, hence the same property, the corresponding property is given the value that was supplied for the last parameter with this name. If the value of this last parameter was not supplied by the caller, the value of the corresponding property is **`undefined`**.

- For each *FunctionDeclaration* in the code, in source text order, create a property of the variable object whose name is the *Identifier* in the *FunctionDeclaration*, whose value is the result returned by creating a Function object as described in section 13, and whose attributes are determined by the type of code. If the variable object already has a property with this name, replace its value and attributes. Semantically, this step must follow the creation of *FormalParameterList* properties.

- For each *VariableDeclaration* in the code, create a property of the variable object whose name is the *Identifier* in *VariableDeclaration*, whose value is **`undefined`** and whose attributes are determined by the type of code. If there is already a property of the variable object with the name of a declared variable, the value of the property and its attributes are not changed. Semantically, this step must follow the creation of the *FormalParameterList* and *FunctionDeclaration* properties. In particular, if a declared variable has the same name as a declared function or formal parameter, the variable declaration does not disturb the existing property.

### 10.1.6 Activation Object

When control enters an execution context for declared function code, anonymous code or implementation-supplied code, an object called the activation object is created and associated with the execution context. The activation object is initialised with a property with name **`arguments`** and property attributes { DontDelete }. The initial value of this property is the arguments object described below.

The activation object is then used as the variable object for the purposes of variable instantiation.

The activation object is purely a specification mechanism. It is impossible for an ECMAScript program to access the activation object. It can access members of the activation object, but not the activation object itself. When the call operation is applied to a Reference value whose base object is an activation object, **`null`** is used as the **`this`** value of the call.

#### 10.2.3 Function and Anonymous Code

- The scope chain is initialised to contain the activation object followed by the objects in the scope chain stored in the [[Closure]] property of the function object.

- Variable instantiation is performed using the activation object as the variable object and using property attributes { DontDelete }.

- The caller provides the **`this`** value. If the **`this`** value provided by the caller is not an object (including the case where it is **`null`**), then the **`this`** value is the global object.

### 11.2 Left-Hand-Side Expressions

#### Syntax

*MemberExpression* **:**
  *PrimaryExpression*
  *FunctionExpression*
  *MemberExpression* **`[`** *Expression* **`]`**
  *MemberExpression* **`.`** *Identifier*
  **`new`** *MemberExpression Arguments*

*NewExpression* **:**
  *MemberExpression*
  **`new`** *NewExpression*

*CallExpression* **:**
  *MemberExpression Arguments*
  *CallExpression Arguments*
  *CallExpression* **`[`** *Expression* **`]`**
  *CallExpression* **`.`** *Identifier*

*Arguments* **:**
  **`( )`**
  **`(`** *ArgumentList* **`)`**

*ArgumentList* **:**
  *AssignmentExpression*
  *ArgumentList* **`,`** *AssignmentExpression*

*LeftHandSideExpression* **:**
  *NewExpression*
  *CallExpression*

### 11.2.2 The `new` Operator

The production *NewExpression* **: new** *NewExpression* is evaluated as follows:

1. Evaluate *NewExpression*.
2. Call GetValue(Result(1)).
3. If Type(Result(2)) is not Object, generate a runtime error.
4. If Result(2) does not implement the internal [[Construct]] method, generate a runtime error.
5. Call the [[Construct]] method on Result(2), providing no arguments (that is, an empty list of arguments).
6. If Result(5).type is **throw** then generate a runtime error.
7. If Result(5).type is not **return** then generate a runtime error.
8. If Type(Result(5).value) is not Object, then generate a runtime error.
9. Return Result(5).value.

The production *MemberExpression* **: new** *MemberExpression Arguments* is evaluated as follows:

1. Evaluate *MemberExpression*.
2. Call GetValue(Result(1)).
3. Evaluate *Arguments*, producing an internal list of argument values (section **Error! Reference source not found.**).
4. If Type(Result(2)) is not Object, generate a runtime error.
5. If Result(2) does not implement the internal [[Construct]] method, generate a runtime error.
6. Call the [[Construct]] method on Result(2), providing the list Result(3) as the argument values.
7. If Result(6).type is **throw** then generate a runtime error.
8. If Result(6).type is not **return** then generate a runtime error.
9. If Type(Result(6).value) is not Object, then generate a runtime error.
10. Return Result(6).value.

### 11.2.3 Function Calls

The production *CallExpression* **:** *MemberExpression Arguments* is evaluated as follows:

1. Evaluate *MemberExpression*.
2. Evaluate *Arguments*, producing an internal list of argument values (section **Error! Reference source not found.**).
3. Call GetValue(Result(1)).
4. If Type(Result(3)) is not Object, generate a runtime error.
5. If Result(3) does not implement the internal [[Call]] method, generate a runtime error.
6. If Type(Result(1)) is Reference, Result(6) is GetBase(Result(1)). Otherwise, Result(6) is **null**.
7. If Result(6) is an activation object, Result(7) is **null**. Otherwise, Result(7) is the same as Result(6).
8. Call the [[Call]] method on Result(3), providing Result(7) as the **`this`** value and providing the list Result(2) as the argument values.
9. If Result(8).type is **throw** then generate a runtime error.
10. If Result(8).type is not **return** then return **undefined**.
11. Return Result(8).value.

The production *CallExpression* **:** *CallExpression Arguments* is evaluated in exactly the same manner, except that the contained *CallExpression* is evaluated in step 1.

**NOTE** Result(8).value will never be of type Reference if Result(3) is a native ECMAScript object. Whether calling a host object can return a value of type Reference is implementation-dependent.

### 11.2.5 Function Expressions

The production *MemberExpression* **:** *FunctionExpression* is evaluated as follows:

1. Evaluate *FunctionExpression*.
2. Return Result(1).

## 12.9 The `return` Statement

### Syntax

*ReturnStatement* **:**
    **`return`** [no *LineTerminator* here] *Expression*$_{opt}$ **`;`**

### Semantics

An ECMAScript program is considered syntactically incorrect if it contains a **`return`** statement that is not within a *FunctionBody*. It causes a function to cease execution and return a value to the caller. If *Expression* is omitted, the return value is the **`undefined`** value. Otherwise, the return value is the value of *Expression*.

The production *ReturnStatement* **::** **`return`** [no *LineTerminator* here] *Expression*$_{opt}$ **`;`** is evaluated as:

1. If the *Expression* is not present, return (**return**, **`undefined`**, **empty**).
2. Evaluate *Expression*.
3. Call GetValue(Result(2)).
4. Return (**return**, Result(3), **empty**).

# 13 Function Definition

### Syntax

*FunctionExpression* **:**
    **`function (`** *FormalParameterList*$_{opt}$ **`) {`** *FunctionBody* **`}`**

*FunctionDeclaration* **:**
    **`function`** *Identifier* **`(`** *FormalParameterList*$_{opt}$ **`) {`** *FunctionBody* **`}`**

*FormalParameterList* **:**
    *Identifier*
    *FormalParameterList* **`,`** *Identifier*

*FunctionBody* **:**
    *SourceElements*

### Semantics

The production *FunctionExpression* **:** **`function (`** *FormalParameterList*$_{opt}$ **`) {`** *FunctionBody* **`}`** is evaluated as follows:

1. Create a new Function object as specified in section 13.1 with parameters specified by *FormalParameterList*$_{opt}$ and body specified by *FunctionBody*. Pass in the scope chain of the running execution context as the closure.
2. Return Result(1).

The production *FunctionDeclaration* **: `function`** *Identifier* **(** *FormalParameterList*$_{opt}$ **) {** *FunctionBody* **}** is processed for function declarations as follows:

1. Create a new Function object as specified in section 13.1 with parameters specified by *FormalParameterList*$_{opt}$, and body specified by *FunctionBody*. Pass in the scope chain of the running execution context as the closure.
2. Create a property of the variable object as specified in section 10.1.3 with Result(1) as the Function object.

The production *FunctionBody* **:** *SourceElements* is evaluates as follows:

1. Process *SourceElements* for function declarations.
2. Evaluate *SourceElements*.
3. Return Result(2).

## 13.1 Creating Function Objects

Given an an optional parameter list specified by *FormalParametersList*$_{opt}$, a body specified by *FunctionBody*, and a closure specified by *Closure*, a Function object is constructed as follows:

1. Create a new native ECMAScript object.
2. Set the [[Class]] property of Result(1) to "Function".
3. Set the [[Prototype]] property of Result(1) to the original Function prototype object as specified in section 15.3.3.1.
4. Set the [[Call]] property of Result(1) to a value which when called establishes a new execution context as described in Section 10 and returns the result of evaluating *FunctionBody* in the new execution context.
5. Set the [[Construct]] property of Result(1) as described in section 15.3.5.4.
6. Set the [[Closure]] property of Result(1) to a copy of *Closure*.
7. Set the **`length`** property of Result(1) to the number of formal properties specified in *FormalParameterList*. If no parameters are specified, set the **`length`** property of Result(1) to 0. This property is given attributes as specified in section 15.3.5.1.
8. Create a new object as would be constructed by the expression **`new Object()`**.
9. Set the **`constructor`** property of Result(8) to Result(1). This property is given attributes { DontEnum }.
10. Set the **`prototype`** property of Result(1) to Result(8). This property is given attributes as specified in section 15.3.5.2.
11. Return Result(1).

A **`prototype`** property is automatically created for every function, against the possibility that the function will be used as a constructor.

### 15.3.2.1 new Function (p1, p2, . . . , pn, body)

The last argument specifies the body (executable code) of a function; any preceding arguments specify formal parameters.

When the **`Function`** constructor is called with some arguments *p1*, *p2*, . . . , *pn*, *body* (where *n* might be 0, that is, there are no "*p*" arguments, and where *body* might also not be provided), the following steps are taken:

1. Let *P* be the empty string.
2. If no arguments were given, let *body* be the empty string and go to step 13.
3. If one argument was given, let *body* be that argument and go to step 13.
4. Let Result(4) be the first argument.
5. Let *P* be ToString(Result(4)).

6. Let *k* be 2.
7. If *k* equals the number of arguments, let *body* be the *k*'th argument and go to step 13.
8. Let Result(8) be the *k*'th argument.
9. Call ToString(Result(8)).
10. Let *P* be the result of concatenating the previous value of *P*, the string **`","`** (a comma), and Result(9).
11. Increase *k* by 1.
12. Go to step 7.
13. Call ToString(*body*).
14. If *P* is not parsable as a $\mathit{FormalParameterList}_{opt}$ then generate a runtime error.
15. If body is not parsable as *FunctionBody* then generate a runtime error.
16. Create a new Function object as specified in section 13.1 with parameters specified by parsing *P* as a $\mathit{FormalParameterList}_{opt}$ and body specified by parsing *body* as a *FunctionBody*. Pass in a scope chain consisting of the global object as the closure.
17. Return Result(16).

Note that it is permissible but not necessary to have one argument for each formal parameter to be specified. For example, all three of the following expressions produce the same result:

```
new Function("a", "b", "c", "return a+b+c")

new Function("a, b, c", "return a+b+c")

new Function("a,b", "c", "return a+b+c")
```

### 15.3.5.4 [[Construct]]

Assume *F* is a function object.

When the [[Construct]] property for *F* is called, the following steps are taken:

1. Create a new native ECMAScript object.
2. Set the [[Class]] property of Result(1) to "Object".
3. Get the value of the **`prototype`** property of the *F*.
4. If Result(3) is an object, set the [[Prototype]] property of Result(1) to Result(3).
5. If Result(3) is not an object, set the [[Prototype]] property of Result(1) to the original Object prototype object as described in section 14.2.3.1.
6. Invoke the [[Call]] property of F, providing Result(1) as the **`this`** value and providing argument list passed into [[Construct]] as the argument values.
7. If the Result(6).type = throw then return Result(6).
8. If the Result(6).type = return and Type(Result(6).value) is Object then return Result(6).
9. Return (return, Result(1), empty).