15 ECMAScript Language: Modules and Scripts

This section is incomplete

15.1 Modules

Module :
ModuleBody$_{opt}$

ModuleBody :
ModuleItemList

ModuleItemList :
ModuleItem
ModuleItemList *ModuleItem*

ModuleItem :
ImportDeclaration
ExportDeclaration
StatementListItem

15.1.0 Module Semantics

15.1.0.1 Static Semantics: Early Errors

ModuleBody : *ModuleItemList*

- It is a Syntax Error if the LexicallyDeclaredNames of *ModuleItemList* contains any duplicate entries.
- It is a Syntax Error if the ExportedBindings of *ModuleItemList* contains any duplicate entries.
- It is a Syntax Error if any element of the LexicallyDeclaredNames of *ModuleItemList* also occurs in the VarDeclaredNames of *ModuleItemList*.
- It is a Syntax Error if *ModuleItemList* Contains **`super`**.

NOTE Additional error conditions relating to conflicting or duplicate declarations are checked during module linking prior to evaluation of a *Module*. If any such errors are detected the *Module* is not evaluated.

15.1.0.2 Static Semantics: ExportedBindings

See also:15.1.2.2.

ModuleItemList : [empty]

1. Return a new empty List.

ModuleItemList : *ModuleItemList* *ModuleItem*

1. Let *names* be ExportedBindings of *ModuleItemList*.
2. Append to *names* the elements of the ExportedBindings of *ModuleItem*.
3. Return *names*.

ModuleItem :
ImportDeclaration
StatementListItem

1. Return a new empty List.

15.1.0.3 Static Semantics: ExportEntries

See also:15.1.2.3.

ModuleItemList : [empty]

1. Return a new empty List.

ModuleItemList **:** *ModuleItemList ModuleItem*

1. Let *entries* be ExportEntries of *ModuleItemList*.
2. Append to *entries* the elements of the ExportEntries of *ModuleItem*.
3. Return *entries*.

ModuleItem **:**
ImportDeclaration
StatementListItem

1. Return a new empty List.

15.1.0.4 Static Semantics: ImportedBindings

ModuleItemList **:** [empty]

1. Return a new empty List.

ModuleItemList **:** *ModuleItemList ModuleItem*

1. Let *names* be ImportedBindings of *ModuleItemList*.
2. Append to *names* the elements of the ImportedBindings of *ModuleItem*.
3. Return *names*.

*ModuleItem***:** *ImportDeclaration*

1. Return the BoundNames of *ImportDeclaration*.

ModuleItem **:**
ExportDeclaration
StatementListItem

1. Return a new empty List.

15.1.0.5 Static Semantics: ImportEntries

See also:15.1.1.3.

ModuleItemList **:** [empty]

1. Return a new empty List.

ModuleItemList **:** *ModuleItemList ModuleItem*

1. Let *entries* be ImportEntries of *ModuleItemList*.
2. Append to *entries* the elements of the ImportEntries of *ModuleItem*.
3. Return *entries*.

ModuleItem **:**
EzportDeclaration
StatementListItem

1. Return a new empty List.

15.1.0.6 Static Semantics: IsStrict

See also: 14.1.9, 15.2.2.

ModuleBody **:** *ModuleItemList*

1. Return **true**.

15.1.0.7 Static Semantics: KnownExportEntries

ModuleBody : *ModuleItemList*

1. Let *allExports* be ExportEntries of *ModuleItemList*.
2. Return a new List containing all the entries of *allEntries* whose [[ImportName]] field is not all.

15.1.0.8 Static Semantics: ModuleRequests

See also: 15.1.1.5, 15.1.2.5.

ModuleItemList : [empty]

1. Return a new empty List.

ModuleItemList : *ModuleItem*

1. Return ModuleRequests of *ModuleItemList*.

ModuleItemList : *ModuleItemList ModuleItem*

1. Let *moduleNames* be ModuleRequests of *ModuleItemList*.
2. Let *additionalNames* be ModuleRequests of *ModuleItem*.
3. Append to *moduleNames* each elements of *additionalNames* that is not already an element of *moduleNames*.
4. Return *moduleNames*.

ModuleItem : StatementListItem

1. Return a new empty List.

15.1.0.9 Static Semantics: LexicallyDeclaredNames

See also: 13.1.3, 13.11.3, 14.1.10, 14.2.7, 14.4.7, 14.5.8, 15.2.3.

ModuleItemList : [empty]

1. Return a new empty List.

ModuleItemList : *ModuleItemList ModuleItem*

1. Let *names* be LexicallyDeclaredNames of *ModuleItemList*.
2. Append to *names* the elements of the LexicallyDeclaredNames of *ModuleItem*.
3. Return *names*.

ModuleItem: *ImportDeclaration*

1. Return the BoundNames of *ImportDeclaration*.

ModuleItem: *ExportDeclaration*

1. Return the BoundNames of *ExportDeclaration*.

ModuleItem: *StatementListItem*

1. Return LexicallyDeclaredNames of *StatementListItem*.

NOTE At the top level of a *Module*, function declarations are treated like lexical declarations rather than like var declarations.

15.1.0.10 Static Semantics: LexicalDeclarations

See also: 13.1.2, 13.11.2.

ModuleItemList **:** [empty]

1. Return a new empty List.

ModuleItemList **:** *ModuleItemList ModuleItem*

1. Let *declarations* be LexicalDeclarations of *ModuleItemList*.
2. Append to *declarations* the elements of the LexicalDeclarations of *ModuleItem*.
3. Return *declarations*.

ModuleItem **:** *ImportDeclaration*

1. If the BoundNames of *ImportDeclarations* is empty, then return an empty List.
2. Return a new List containing *ImportDeclaration*.

ModuleItem **:** *ExportDeclaration*

1. If *ExportDeclaration* is **`export`** *Declaration* **`;`** then return a new List containing *Declaration*.
2. Return a new empty List.

15.1.0.11 Static Semantics: UnknownExportEntries

ModuleBody **:** *ModuleItemList*

1. Let *allExports* be ExportEntries of *ModuleItemList*.
2. Return a new List containing all the entries of *allEntries* whose [[ImportName]] field is all.

15.1.0.12 Static Semantics: VarDeclaredNames

See also: 13.0.1, 13.1.8, 13.5.1, 13.6.1.1, 13.6.2.1, 13.6.3.1, 13.6.4.3, 13.10.2, 13.11.4, 13.12.2, 13.14.2, 14.1.11, 14.4.10, 14.5.14, 15.2.5.

ModuleItemList **:** *ModuleItemList ModuleItem*

1. Let *names* be VarDeclaredNames of *ModuleItemList*.
2. Append to *names* the elements of the VarDeclaredNames of *ModuleItem*.
3. Return *names*.

ModuleItem **:** *ImportDeclaration*

1. Return an empty List.

ModuleItem **:** *ExportDeclaration*

1. If *ExportDeclaration* is **`export`** *VariableStatement* **`;`** then return BoundNames of *ExportDeclaration*.
2. Return a new empty List.

15.1.0.13 Static Semantics: VarScopedDeclarations

See also: 13.1.9, 15.2.6.

ModuleItemList **:** [empty]

1. Return a new empty List.

ModuleItemList **:** *ModuleItemList ModuleItem*

1. Let *declarations* be VarScopedDeclarations of *ModuleItemList*.
2. Append to *declarations* the elements of the VarScopedDeclarations of *ModuleItem*.
3. Return *declarations*.

ModuleItem **:** *ImportDeclaration*

1. Return a new empty List.

ModuleItem : *ExportDeclaration*

1. If *ExportDeclaration* is **`export`** *VariableStatement* **`;`** then return a new List containing *VariableStatement*.
2. Return a new empty List.

15.1.1 Imports

ImportDeclaration :
ModuleImport
`import` *ImportClause FromClause* **`;`**
`import` *ModuleSpecifier* **`;`**

ModuleImport :
`module` [no *LineTerminator* here] *ImportedBinding FromClause* **`;`**

FromClause :
`from` *ModuleSpecifier*

ImportClause :
ImportedBinding
ImportedBinding **`,`** *NamedImports*
NamedImports

NamedImports :
`{ }`
`{` *ImportsList* **`}`**
`{` *ImportsList* **`,}`**

ImportsList :
ImportSpecifier
ImportsList **`,`** *ImportSpecifier*

ImportSpecifier :
ImportedBinding
IdentifierName **`as`** *ImportedBinding*

ModuleSpecifier :
StringLiteral

ImportedBinding :
BindingIdentifier

15.1.1.1 Static Semantics: Early Errors

ModuleItem : *ImportDeclaration*

- It is a Syntax Error if the BoundNames of *ImportDeclaration* contains any duplicate entries.

15.1.1.2 Static Semantics: BoundNames

See also: 13.2.1.2, 13.2.2.1, 13.2.3.2, 13.6.4.2, 14.1.2, 14.2.2, 14.4.2, 14.5.2, 0.

ImportDeclaration : **`import`** *ImportClause FromClause* **`;`**

1. Return the BoundNames of *ImportClause*.

ImportDeclaration : **`import`** *ModuleSpecifier* **`;`**

1. Return a new empty List.

ModuleImport : **module** *ImportedBinding FromClause* **;**

1. Return the BoundNames of *ImportedBinding*.

ImportClause : *ImportedBinding* **,** *NamedImports*

1. Let *names* be the BoundNames of *ImportedBinding*.
2. Append to *names* the elements of the BoundNames of *NamedImports*.
3. Return names.

ImportsList : *ImportsList* **,** *ImportSpecifier*

1. Let *names* be the BoundNames of *ImportsList*.
2. Append to *names* the elements of the BoundNames of *ImportSpecifier*.
3. Return names.

ImportSpecifier : *IdentifierName* **as** *ImportedBinding*

1. Return the BoundNames of *ImportedBinding*.

15.1.1.3 Static Semantics: ImportEntries

See also:15.1.0.5.

ImportDeclaration : **import** *ImportClause FromClause* **;**

1. Let *module* be the sole element of *ModuleRequests* of *FromClause*.
2. Return ImportEntriesForModule of *ImportClause* with argument *module*.

ImportDeclaration : **import** *ModuleSpecifier* **;**

1. Return a new empty List.

ModuleImport : **module** *ImportedBinding FromClause* **;**

1. Let *module* be ModuleRequests of *FromClause*.
2. Let *localName* be the StringValue of *ImportedBinding*.
3. Let *entry* be the Record {[[ModuleRequest]]: *module*, [[ImportName]]: **"default"**, [[LocalName]]: *localName* }.
4. Return a new List containing *entry*.

15.1.1.4 Static Semantics: ImportEntriesForModule

With parameter *module*.

ImportClause : *ImportedBinding* **,** *NamedImports*

1. Let *localName* be the StringValue of *ImportedBinding*.
2. Let *defaultEntry* be the Record {[[ModuleRequest]]: *module*, [[ImportName]]: **"default"**, [[LocalName]]: *localName* }.
3. List *entries* be a new List containing *defaultEntry*.
4. Append to *entries* the elements of the ImportEntitiesForModule of *NamedImports* with argument *module*.
5. Return *entries*.

NamedImports : **{ }**

1. Return a new empty List.

ImportsList : *ImportsList* **,** *ImportSpecifier*

1. Let *specs* be the ImportEntitiesForModule of *ImportsList* with argument *module*.
2. Append to *specs* the elements of the ImportEntitiesForModule of *ImportSpecifier* with argument *module*.
3. Return *specs*.

ImportSpecifier : *ImportedBinding*

1. Let *localName* be the StringValue of *ImportedBinding*.
2. Let *entry* be the Record {[[ModuleRequest]]: *module*, [[ImportName]]: *localName* , [[LocalName]]: *localName* }.
3. Return a new List containing *entry*.

ImportSpecifier : *IdentifierName* **as** *ImportedBinding*

1. Let *importName* be the StringValue of *IdentifierName*.
2. Let *localName* be the StringValue of *ImportedBinding*.
3. Let *entry* be the Record {[[ModuleRequest]]: *module*, [[ImportName]]: *importName*, [[LocalName]]: *localName* }.
4. Return a new List containing *entry*.

15.1.1.5 Static Semantics: ModuleRequests

See also: 15.1.0.8, 15.1.2.5.

ImportDeclaration : **import** *ImportClause FromClause* **;**

1. Return ModuleRequests of *FromClause*.

ModuleImport : **module** *ImportedBinding FromClause* **;**

1. Return ModuleRequests of *FromClause*.

ModuleSpecifier : *StringLiteral*

1. Return a List containing the SV of *StringLiteral*.

15.1.2 Exports

ExportDeclaration :
export * *FromClause* **;**
export *ExportClause*[NoReference] *FromClause* **;**
export *ExportClause* **;**
export *VariableStatement*
export *Declaration*[Default]
export default *AssignmentExpression* **;**

ExportClause[NoReference] :
{ }
{ *ExportsList*[?NoReference] **}**
{ *ExportsList*[?NoReference] **, }**

ExportsList[NoReference] :
ExportSpecifier[?NoReference]
ExportsList[?NoReference] **,** *ExportSpecifier*[?NoReference]

ExportSpecifier[NoReference] :
[~NoReference] *IdentifierReference*
[~NoReference] *IdentifierReference* **as** *IdentifierName*
[+NoReference] *IdentifierName*
[+NoReference] *IdentifierName* **as** *IdentifierName*

NOTE *ExportSpecifier* is used to export bindings from the enclosing module *Module*. *ExportSpecifier*[NoReference] is used to export bindings from a referenced *Module*. In that case *IdentifierReference* restrictions are not applied to the naming of the items too be exported because they are not used to create local bindings.

15.1.2.1 Static Semantics: BoundNames

See also: 13.2.1.2, 13.2.2.1, 13.2.3.2, 13.6.4.2, 14.1.2, 14.2.2, 14.4.2, 14.5.2, 15.1.1.2.

ExportDeclaration **:**
export ***** *FromClause* **;**
export *ExportClause FromClause* **;**
export *ExportClause* **;**

1. Return a new empty List.

ExportDeclaration : **export** *VariableStatement* **;**

1. Return the BoundNames of *VariableStatement*.

ExportDeclaration : **export** *Declaration* **;**

1. Return the BoundNames of *Declaration*.

ExportDeclaration : **export default** *AssignmentExpression* **;**

1. Return a List containing **"default"**.

15.1.2.2 Static Semantics: ExportedBindings

See also:15.1.0.2.

ExportDeclaration **: export** ***** *FromClause* **;**

1. Return a new empty List.

ExportDeclaration **:**
export *ExportClause FromClause* **;**
export *ExportClause* **;**

1. Return the ExportedBindings of this *ExportClause*.

ExportDeclaration **:**
export *VariableStatement*
export *Declaration*$_{[Default]}$

1. Return the BoundNames of this *ExportDeclaration*.

ExportDeclaration : **export default** *AssignmentExpression* **;**

1. Return a List containing **"default"**.

ExportClause **: { }**

1. Return a new empty List.

ExportsList : *ExportsList* **,** *ExportSpecifier*

1. Let *names* be the ExportedBindings of *ExportsList*.
2. Append to *names* the elements of the ExportedBindings of *ExportSpecifier*.
3. Return *names*.

ExportDeclaration **: export** *ExportClause FromClause*$_{opt}$ **;**

1. Return the ExportedBindings of *ExportClause*.

ExportSpecifier **:** *IdentifierReference*

1. Return a List containing the StringValue of *IdentifierReference*.

ExportSpecifier **:** *IdentifierReference* **as** *IdentifierName*

1. Return a List containing the StringValue of *IdentifierName*.

ExportSpecifier **:** *IdentifierName*

1. Return a List containing the StringValue of *IdentifierName*.

ExportSpecifier **:** *IdentifierName* **as** *IdentifierName*

1. Return a List containing the StringValue of the second *IdentifierName*.

15.1.2.3 Static Semantics: ExportEntries

See also:15.1.0.3.

ExportDeclaration **: export *** *FromClause* **;**

1. Let *module* be the sole element of *ModuleRequests* of *FromClause*.
2. Let *entry* be the Record {[[ModuleRequest]]: *module*, [[ImportName]]: all, [[LocalName]]: **null**, [[ExportName]]: **null** }.
3. Return a new List containing *entry*.

ExportDeclaration **: export** *ExportClause FromClause* **;**

1. Let *module* be the sole element of *ModuleRequests* of *FromClause*.
2. Return ExportEntriesForModule of *ExportClause* with argument *module*.

ExportDeclaration **: export** *ExportClause* **;**

1. Let *module* be the sole element of *ModuleRequests* of *FromClause*.
2. Return ExportEntriesForModule of *ExportClause* with argument **null**.

ExportDeclaration **: export** *VariableStatement***;**

1. Let *entries* be a new empty List.
2. Let *names* be the BoundNames of *VariableStatement*.
3. For each *name* in *names, do*
    a. Append to *entries* the Record {[[ModuleRequest]]: **null**, [[ImportName]]: **null**, [[LocalName]]: *name*, [[ExportName]]: *name* }.
4. Return *entries*.

ExportDeclaration **: export** *Declaration***;**

1. Let *entries* be a new empty List.
2. Let *names* be the BoundNames of *Declaration*.
3. For each *name* in *names, do*
    a. Append to *entries* the Record {[[ModuleRequest]]: **null**, [[ImportName]]: **null**, [[LocalName]]: *name*, [[ExportName]]: *name* }.
4. Return *entries*.

ExportDeclaration : **export default** *AssignmentExpression***;**

1. Let *entry* be the Record {[[ModuleRequest]]: **null**, [[ImportName]]: **null**, [[LocalName]]: **"default"**, [[ExportName]]: **"default"**}.
2. Return a new List containing *entry*.

15.1.2.4 Static Semantics: ExportEntriesForModule

With parameter *module*.

ExportClause **: { }**

1. Return a new empty List.

ExportsList : *ExportsList* **,** *ExportSpecifier*

1. Let *specs* be the ExportEntitiesForModule of *ExportsList* with argument *module*.
2. Append to *specs* the elements of the ExportEntitiesForModule of *ExportSpecifier* with argument *module*.
3. Return *specs*.

ExportSpecifier : *IdentifierReference*

1. Let *localName* be the StringValue of *IdentifierReference*.
2. Return a new List containing the Record {[[ModuleRequest]]: *module*, [[ImportName]]: **null**, [[LocalName]]: *localName*, [[ExportName]]: *localName* }.

ExportSpecifier : *IdentifierReference* **as** *IdentifierName*

1. Let *localName* be the StringValue of *IdentifierReference*.
2. Let *exportName* be the StringValue of *IdentifierName*.
3. Return a new List containing the Record {[[ModuleRequest]]: *module*, [[ImportName]]: **null**, {[[LocalName]]: *localName*, [[ExportName]]: *exportName* }.

ExportSpecifier : *IdentifierName*

1. Let *sourceName* be the StringValue of *IdentifierName*.
2. Return a new List containing the Record {[[ModuleRequest]]: *module*, [[ImportName]]: *sourceName*, [[LocalName]]: **null**, [[ExportName]]: *sourceName* }.

ExportSpecifier : *IdentifierReference* **as** *IdentifierName*

1. Let *sourceName* be the StringValue of the first *IdentifierName*.
2. Let *exportName* be the StringValue of the second *IdentifierName*.
3. Return a new List containing the Record {[[ModuleRequest]]: *module*, [[ImportName]]: *sourceName*, [[LocalName]]: **null**, [[ExportName]]: *exportName* }.

15.1.2.5 Static Semantics: ModuleRequests

See also: 15.1.0.8, 15.1.1.5.

ExportDeclaration **: export** *ExportClause FromClause* **;**

1. Return the ModuleRequests of *FromClause*.

ExportDeclaration **:**
export *ExportClause* **;**
export *VariableStatement*
export *Declaration*
export default *AssignmentExpression* **;**

1. Return a new empty List.

1 Modules: Semantics

1.1 Module Loading

1.1.1 Load Records

The Load Record type represents an attempt to locate, fetch, translate, and parse a single module.

Each Load Record has the following fields:

load.[[Status]]	One of: **`"loading"`**, **`"loaded"`**, **`"linked"`**, or **`"failed"`**.
load.[[Name]]	The normalized name of the module being loaded, or **undefined** if loading an anonymous module.
load.[[LinkSets]]	A List of all LinkSets that require this load to succeed. There is a many-to-many relation between Loads and LinkSets. A single **`import()`** call can have a large dependency tree, involving many Loads. Many **`import()`** calls can be waiting for a single Load, if they depend on the same module.
load.[[Metadata]]	An object which loader hooks may use for any purpose. See Loader.prototype.locate.
load.[[Address]]	The result of the locate hook.
load.[[Source]]	The result of the translate hook.
load.[[Kind]]	Once the Load reaches the **`"loaded"`** state, either **declarative** or **dynamic**. If the **`instantiate`** hook returned undefined, the module is declarative, and load.[[Body]] contains a Module parse. Otherwise, the **`instantiate`** hook returned a ModuleFactory object; load.[[Execute]] contains the **`.execute`** callable object.
load.[[Body]]	A Module parse, if load.[[Kind]] is **declarative**. Otherwise undefined.
load.[[Execute]]	The value of **`factory.execute`**, if load.[[Kind]] is **dynamic**. Otherwise undefined.
load.[[Dependencies]]	Once the Load reaches the **`"loaded"`** state, a List of pairs. Each pair consists of two strings: a module name as it appears in a **`module`**, **`import`**, or **`export from`** declaration in

	load.[[Body]], and the corresponding normalized module name.
load.[[Exception]]	If load.[[Status]] is **`"failed"`**, the exception value that was thrown, causing the load to fail. Otherwise, **null**.
load.[[Module]]	The Module object produced by this load, or undefined. If the **`instantiate`** hook returns undefined, load.[[Module]] is populated at that point, if parsing succeeds and there are no early errors. Otherwise the **`instantiate`** hook returns a factory object, and load.[[Module]] is set during the link phase, when the **`factory.execute()`** method returns a Module.

1.1.1.1 CreateLoad(name) Abstract Operation

The abstract operation CreateLoad creates and returns a new Load Record. The argument *name* is either **`undefined`**, indicating an anonymous module, or a normalized module *name*.

The following steps are taken:

1. Let *load* be a new Load Record.
2. Set the [[Status]] field of *load* to **`"loading"`**.
3. Set the [[Name]] field of *load* to *name*.
4. Set the [[LinkSets]] field of *load* to a new empty List.
5. Let *metadata* be the result of ObjectCreate(%ObjectPrototype%).
6. Set the [[Metadata]] field of *load* to *metadata*.
7. Set the [[Address]] field of *load* to undefined.
8. Set the [[Source]] field of *load* to undefined.
9. Set the [[Kind]] field of *load* to undefined.
10. Set the [[Body]] field of *load* to undefined.
11. Set the [[Execute]] field of *load* to undefined.
12. Set the [[Exception]] field of *load* to undefined.
13. Set the [[Module]] field of *load* to undefined.
14. Return *load*.

1.1.1.2 LoadFailed Functions

A LoadFailed function is an anonymous function that marks a Load Record as having failed. All LinkSets that depend on the Load also fail.

Each LoadFailed function has a [[Load]] internal slot.

When a LoadFailed function *F* is called with argument *exc*, the following steps are taken:

1. Let *load* be *F*.[[Load]].
2. Assert: *load*.[[Status]] is **`"loading"`**.

3. Set *load*.[[Status]] to `"failed".
4. Set *load*.[[Exception]] to *exc*.
5. Let *linkSets* be a copy of the List *load*.[[LinkSets]].
6. For each *linkSet* in *linkSets*, in the order in which the LinkSet Records were created,
    a. Call LinkSetFailed(*linkSet*, *exc*).
7. Assert: *load*.[[LinkSets]] is empty.

1.1.1.3 RequestLoad(loader, request, refererName, refererAddress) Abstract Operation

The RequestLoad abstract operation normalizes the given module name, *request*, and returns a promise that resolves to the value of a Load object for the given module.

The *loader* argument is a Loader object.

request is the (non-normalized) name of the module to be imported, as it appears in the import-declaration or as the argument to **`loader.load()`** or **`loader.import()`**.

refererName and *refererAddress* provide information about the context of the **`import()`** call or import-declaration. This information is passed to all the *loader* hooks.

If the requested module is already in the *loader*'s module registry, RequestLoad returns a promise for a Load with the [[Status]] field set to **`"linked"`**. If the requested module is loading or loaded but not yet linked, RequestLoad returns a promise for an existing Load object from *loader*.[[Loads]]. Otherwise, RequestLoad starts loading the module and returns a promise for a new Load Record.

The following steps are taken:

1. Let *F* be a new anonymous function as defined by CallNormalize.
2. Set the [[Loader]] internal slot of *F* to *loader*.
3. Set the [[Request]] internal slot of *F* to *request*.
4. Set the [[RefererName]] internal slot of *F* to *refererName*.
5. Set the [[RefererAddress]] internal slot of *F* to *refererAddress*.
6. Let *p* be the result of calling OrdinaryConstruct(%Promise%, (*F*)).
7. Let *G* be a new anonymous function as defined by GetOrCreateLoad.
8. Set the [[Loader]] internal slot of *G* to *loader*.
9. Let *p* be the result of calling PromiseThen(*p*, *G*).
10. Return *p*.

1.1.1.4 CallNormalize Functions

A CallNormalize function is an anonymous function that calls a *loader*'s normalize hook.

Each CallNormalize function has internal slots [[Loader]], [[Request]], [[RefererName]], and [[RefererAddress]].

When a CallNormalize function *F* is called with arguments *resolve* and *reject*, the following steps are taken.

1. Let *loader* be *F*.[[Loader]].
2. Let *request* be *F*.[[Request]].
3. Let *refererName* be *F*.[[RefererName]].
4. Let *refererAddress* be *F*.[[RefererAddress]].
5. Let *normalizeHook* be the result of Get(*loader*, **`"normalize"`**).

6. Let *name* be the result of calling the [[Call]] internal method of *normalizeHook* passing *loader* and (*request*, *refererName*, *refererAddress*) as arguments.
7. ReturnIfAbrupt(*name*).
8. Call the [[Call]] internal method of *resolve* passing undefined and (*name*) as arguments.

1.1.1.5 GetOrCreateLoad Functions

A GetOrCreateLoad function is an anonymous function that gets or creates a Load Record for a given module *name*.

Each GetOrCreateLoad function has a [[Loader]] internal slot.

When a GetOrCreateLoad function *F* is called with argument *name*, the following steps are taken:

1. Let *loader* be *F*.[[Loader]].
2. Let *name* be ToString(*name*).
3. ReturnIfAbrupt(*name*).
4. If there is a Record in *loader*.[[Modules]] whose [[key]] field is equal to *name*, then
    a. Let *existingModule* be the [[value]] field of that Record.
    b. Let *load* be the result of CreateLoad(*name*).
    c. Set the [[Status]] field of *load* to **`"linked"`**.
    d. Set the [[Module]] field of *load* to *existingModule*.
    e. Return *load*.
5. Else, if there is a Load Record in the List *loader*.[[Loads]] whose [[Name]] field is equal to *name*, then
    a. Let *load* be that Load Record.
    b. Assert: *load*.status is either **`"loading"`** or **`"loaded"`**.
    c. Return *load*.
6. Let *load* be the result of CreateLoad(*name*).
7. Add *load* to the List *loader*.[[Loads]].
8. Call ProceedToLocate(*loader*, *load*).
9. Return *load*.

1.1.1.6 ProceedToLocate(loader, load, p) Abstract Operation

The ProceedToLocate abstract operation continues the asynchronous loading process at the **`locate`** hook.

ProceedToLocate performs the following steps:

1. Let *p* be the result of PromiseResolve(undefined).
2. Let *F* be a new anonymous function object as defined in CallLocate.
3. Set *F*.[[Loader]] to *loader*.
4. Set *F*.[[Load]] to *load*.
5. Let *p* be the result of calling PromiseThen(*p*, *F*).
6. Return ProceedToFetch(*loader*, *load*, *p*).

1.1.1.7 ProceedToFetch(loader, load, p) Abstract Operation

The ProceedToFetch abstract operation continues the asynchronous loading process at the **`fetch`** hook.

ProceedToFetch performs the following steps:

1. Let *F* be a new anonymous function object as defined in CallFetch.

2. Set *F*.[[Loader]] to *loader*.
3. Set *F*.[[Load]] to *load*.
4. Set *F*.[[AddressPromise]] to *p*.
5. Let *p* be the result of calling PromiseThen(*p*, *F*).
6. Return ProceedToTranslate(*loader*, *load*, *p*).

1.1.1.8 ProceedToTranslate(loader, load, p) Abstract Operation

The ProceedToTranslate abstract operation continues the asynchronous loading process at the **`translate`** hook.

ProceedToTranslate performs the following steps:

1. Let *F* be a new anonymous function object as defined in CallTranslate.
2. Set *F*.[[Loader]] to *loader*.
3. Set *F*.[[Load]] to *load*.
4. Let *p* be the result of calling PromiseThen(*p*, *F*).
5. Let *F* be a new anonymous function object as defined in CallInstantiate.
6. Set *F*.[[Loader]] to *loader*.
7. Set *F*.[[Load]] to *load*.
8. Let *p* be the result of calling PromiseThen(*p*, *F*).
9. Let *F* be a new anonymous function object as defined in InstantiateSucceeded.
10. Set *F*.[[Loader]] to *loader*.
11. Set *F*.[[Load]] to *load*.
12. Let *p* be the result of calling PromiseThen(*p*, *F*).
13. Let *F* be a new anonymous function object as defined in LoadFailed.
14. Set *F*.[[Load]] to *load*.
15. Let *p* be the result of calling PromiseCatch(*p*, *F*).

1.1.1.9 SimpleDefine(obj, name, value) Abstract Operation

The SimpleDefine operation defines a writable, configurable, enumerable data property on an ordinary object by taking the following steps:

1. Return the result of calling OrdinaryDefineOwnProperty with arguments *obj*, *name*, and PropertyDescriptor{[[Value]]: *value*, [[Writable]]: true, [[Enumerable]]: true, [[Configurable]]: true}.

1.1.1.10 CallLocate Functions

A CallLocate function is an anonymous function that calls the **`locate`** *loader hook*.

Each CallLocate function has [[Loader]] and [[Load]] internal slots.

When a CallLocate function *F* is called, the following steps are taken:

1. Let *loader* be *F*.[[Loader]].
2. Let *load* be *F*.[[Load]].
3. Let *hook* be the result of Get(*loader*, **`"locate"`**).
4. ReturnIfAbrupt(*hook*).
5. If IsCallable(*hook*) is false, throw a TypeError exception.
6. Let *obj* be the result of calling ObjectCreate(%ObjectPrototype%, ()).
7. Call SimpleDefine(*obj*, **`"name"`**, *load*.[[Name]]).
8. Call SimpleDefine(*obj*, **`"metadata"`**, *load*.[[Metadata]]).

9. Return the result of calling the [[Call]] internal method of *hook* with *loader* and (*obj*) as arguments.

1.1.1.11 CallFetch Functions

A CallFetch function is an anonymous function that calls the **`fetch`** *loader hook*.

Each CallFetch function has [[Loader]] and [[Load]] internal slots.

When a CallFetch function *F* is called with argument *address*, the following steps are taken:

1. Let *loader* be *F*.[[Loader]].
2. Let *load* be *F*.[[Load]].
3. If *load*.[[LinkSets]] is an empty List, return undefined.
4. Set the [[Address]] field of *load* to *address*.
5. Let *hook* be the result of Get(*loader*, **`"fetch"`**).
6. ReturnIfAbrupt(*hook*).
7. If IsCallable(*hook*) is false, throw a TypeError exception.
8. Let *obj* be the result of calling ObjectCreate(%ObjectPrototype%, ()).
9. Call SimpleDefine(*obj*, **`"name"`**, *load*.[[Name]]).
10. Call SimpleDefine(*obj*, **`"metadata"`**, *load*.[[Metadata]]).
11. Call SimpleDefine(*obj*, **`"address"`**, *address*).
12. Return the result of calling the [[Call]] internal method of *hook* with *loader* and (*obj*) as arguments.

1.1.1.12 CallTranslate Functions

A CallTranslate function is an anonymous function that calls the **`translate`** *loader hook*.

Each CallTranslate function has [[Loader]] and [[Load]] internal slots.

When a CallTranslate function *F* is called with argument *source*, the following steps are taken:

1. Let *loader* be *F*.[[Loader]].
2. Let *load* be *F*.[[Load]].
3. If *load*.[[LinkSets]] is an empty List, return undefined.
4. Let *hook* be the result of Get(*loader*, **`"translate"`**).
5. ReturnIfAbrupt(*hook*).
6. If IsCallable(*hook*) is false, throw a TypeError exception.
7. Let *obj* be the result of calling ObjectCreate(%ObjectPrototype%, ()).
8. Call SimpleDefine(*obj*, **`"name"`**, *load*.[[Name]]).
9. Call SimpleDefine(*obj*, **`"metadata"`**, *load*.[[Metadata]]).
10. Call SimpleDefine(*obj*, **`"address"`**, *load*.[[Address]]).
11. Call SimpleDefine(*obj*, **`"source"`**, *source*).
12. Return the result of calling the [[Call]] internal method of *hook* with *loader* and (*obj*) as arguments.

1.1.1.13 CallInstantiate Functions

A CallInstantiate function is an anonymous function that calls the **`instantiate`** *loader hook*.

Each CallInstantiate function has [[Loader]] and [[Load]] internal slots.

When a CallInstantiate function *F* is called with argument *source*, the following steps are taken:

1. Let *loader* be *F*.[[Loader]].
2. Let *load* be *F*.[[Load]].
3. If *load*.[[LinkSets]] is an empty List, return undefined.
4. Set the [[Source]] internal slot of *load* to *source*.
5. Let *hook* be the result of Get(*loader*, **"instantiate"**).
6. ReturnIfAbrupt(*hook*).
7. If IsCallable(*hook*) is false, throw a TypeError exception.
8. Let *obj* be the result of calling ObjectCreate(%ObjectPrototype%, ()).
9. Call SimpleDefine(*obj*, **"name"**, *load*.[[Name]]).
10. Call SimpleDefine(*obj*, **"metadata"**, *load*.[[Metadata]]).
11. Call SimpleDefine(*obj*, **"address"**, *load*.[[Address]]).
12. Call SimpleDefine(*obj*, **"source"**, *source*).
13. Return the result of calling the [[Call]] internal method of *hook* with *loader* and (*obj*) as arguments.

1.1.1.14 InstantiateSucceeded Functions

An InstantiateSucceeded function is an anonymous function that handles the result of the **instantiate** hook.

Each InstantiateSucceeded function has [[Loader]] and [[Load]] internal slots.

When an InstantiateSucceeded function *F* is called with argument *instantiateResult*, the following steps are taken:

1. Let *loader* be *F*.[[Loader]].
2. Let *load* be *F*.[[Load]].
3. If *load*.[[LinkSets]] is an empty List, return undefined.
4. If *instantiateResult* is undefined, then
    a. Let *body* be the result of parsing *load*.[[Source]], interpreted as UTF-16 encoded Unicode text as described in clause 10.1.1, using Module as the goal symbol. Throw a SyntaxError exception if the parse fails or if any static semantics errors are detected.
    b. Set the [[Body]] field of *load* to *body*.
    c. Set the [[Kind]] field of *load* to **declarative**.
    d. Let *depsList* be the ModuleRequests of *body*.
5. Else if Type(*instantiateResult*) is Object, then
    a. Let *deps* be the result of Get(*instantiateResult*, **"deps"**).
    b. ReturnIfAbrupt(*deps*).
    c. If *deps* is undefined, then let *depsList* be a new empty List.
    d. Else:
        i. Let *depsList* be the result of calling the IterableToArray abstract operation passing *deps* as the single argument.
        ii. ReturnIfAbrupt(*depsList*).
    e. Let *execute* be the result of Get(*instantiateResult*, **"execute"**).
    f. ReturnIfAbrupt(*execute*).
    g. Set the [[Execute]] field of *load* to *execute*.
    h. Set the [[Kind]] field of *load* to **dynamic**.
6. Else,
    a. Throw a TypeError exception.
7. Return the result of calling ProcessLoadDependencies(*load*, *loader*, *depsList*).

1.1.1.15 ProcessLoadDependencies(load, loader, depsList) Abstract Operation

The ProcessLoadDependencies abstract operation is called after one module has nearly finished loading. It starts new loads as needed to *load* the module's dependencies.

ProcessLoadDependencies also arranges for LoadSucceeded to be called.

The following steps are taken:

1. Let *refererName* be *load*.[[Name]].
2. Set the [[Dependencies]] field of *load* to a new empty List.
3. Let *loadPromises* be a new empty List.
4. For each *request* in *depsList*, do
    a. Let *p* be the result of RequestLoad(*loader*, *request*, *refererName*, *load*.[[Address]]).
    b. Let *F* be a new anonymous function as defined by AddDependencyLoad.
    c. Set the [[Load]] internal slot of *F* to *load*.
    d. Set the [[Request]] internal slot of *F* to *request*.
    e. Let *p* be the result of PromiseThen(*p*, *F*).
    f. Append *p* as the last element of *loadPromises*.
5. Let *p* be PromiseAll(*loadPromises*).
6. Let *F* be a new anonymous function as defined by LoadSucceeded.
7. Set the [[Load]] internal slot of *F* to *load*.
8. Let *p* be the result of PromiseThen(*p*, *F*).
9. Return *p*.

1.1.1.16 **AddDependencyLoad Functions**

An AddDependencyLoad function is an anonymous function that adds a Load Record for a dependency to any LinkSets associated with the parent Load.

Each AddDependencyLoad function has [[ParentLoad]] and [[Request]] internal slots.

When an AddDependencyLoad function *F* is called with argument *depLoad*, the following steps are taken:

1. Let *parentLoad* be *F*.[[ParentLoad]].
2. Let *request* be *F*.[[Request]].
3. Assert: There is no Record in the List *parentLoad*.[[Dependencies]] whose [[key]] field is equal to *request*.
4. Add the Record {[[key]]: *request*, [[value]]: *depLoad*.[[Name]]} to the List *parentLoad*.[[Dependencies]].
5. If *depLoad*.[[Status]] is not **`"linked"`**, then
    a. Let *linkSets* be a copy of the List *parentLoad*.[[LinkSets]].
    b. For each *linkSet* in *linkSets*, do
        i. Call AddLoadToLinkSet(*linkSet*, *depLoad*).

1.1.1.17 **LoadSucceeded Functions**

A LoadSucceeded function is an anonymous function that transitions a Load Record from **`"loading"`** to **`"loaded"`** and notifies all associated LinkSet Records of the change. This function concludes the loader pipeline. It is called after all a newly loaded module's dependencies are successfully processed.

Each LoadSucceeded function has a [[Load]] internal slot.

When a LoadSucceeded function *F* is called, the following steps are taken:

1. Let *load* be *F*.[[Load]].
2. Assert: *load*.[[Status]] is **`"loading"`**.
3. Set the [[Status]] field of *load* to **`"loaded"`**.

4. Let *linkSets* be a copy of *load*.[[LinkSets]].
5. For each *linkSet* in *linkSets*, in the order in which the LinkSet Records were created,
    a. Call UpdateLinkSetOnLoad(*linkSet*, *load*).

1.1.2 LinkSet Records

A LinkSet Record represents a call to **`loader.define()`**, **`.load()`**, **`.module()`**, or **`.import()`**.

Each LinkSet Record has the following fields:

linkSet.[[Loader]]	The Loader object that created this LinkSet.
linkSet.[[Loads]]	A List of the Load Records that must finish loading before the modules can be linked and evaluated.
linkSet.[[Done]]	A Promise that becomes fulfilled when all dependencies are loaded and linked together.
linkSet.[[Resolve]]	Function used to resolve linkSet.[[Done]].
linkSet.[[Reject]]	Function used to reject linkSet.[[Done]].

1.1.2.1 **CreateLinkSet(loader, startingLoad) Abstract Operation**

The CreateLinkSet abstract operation creates a new LinkSet record by performing the following steps:

1. If Type(*loader*) is not Object, throw a TypeError exception.
2. If *loader* does not have all of the internal properties of a Loader Instance, throw a TypeError exception.
3. Let *deferred* be the result of calling GetDeferred(%Promise%).
4. ReturnIfAbrupt(*deferred*).
5. Let *linkSet* be a new LinkSet Record.
6. Set the [[Loader]] field of *linkSet* to *loader*.
7. Set the [[Loads]] field of *linkSet* to a new empty List.
8. Set the [[Done]] field of *linkSet* to *deferred*.[[Promise]].
9. Set the [[Resolve]] field of *linkSet* to *deferred*.[[Resolve]].
10. Set the [[Reject]] field of *linkSet* to *deferred*.[[Reject]].
11. Call AddLoadToLinkSet(*linkSet*, *startingLoad*).
12. Return *linkSet*.

1.1.2.2 **AddLoadToLinkSet(linkSet, load) Abstract Operation**

The AddLoadToLinkSet abstract operation associates a LinkSet Record with a Load Record and each of its currently known dependencies, indicating that the LinkSet cannot be linked until those Loads have finished successfully.

The following steps are taken:

1. Assert: *load*.[[Status]] is either **`"loading"`** or **`"loaded"`**.
2. Let *loader* be *linkSet*.[[Loader]].

3. If *load* is not already an element of the List *linkSet*.[[Loads]],
    a. Add *load* to the List *linkSet*.[[Loads]].
    b. Add *linkSet* to the List *load*.[[LinkSets]].
    c. If *load*.[[Status]] is **`"loaded"`**, then
        i. For each *name* in the List *load*.[[Dependencies]], do
            1. If there is no element of *loader*.[[Modules]] whose [[key]] field is equal to *name*,
                a If there is an element of *loader*.[[Loads]] whose [[Name]] field is equal to *name*,
                    i. Let *depLoad* be that Load Record.
                    ii. Call AddLoadToLinkSet(*linkSet*, *depLoad*).

1.1.2.3 UpdateLinkSetOnLoad(linkSet, load) Abstract Operation

The UpdateLinkSetOnLoad abstract operation is called immediately after a Load successfully finishes, after starting Loads for any dependencies that were not already loading, loaded, or in the module registry.

This operation determines whether *linkSet* is ready to link, and if so, calls Link.

The following steps are taken:

1. Assert: *load* is an element of *linkSet*.[[Loads]].
2. Assert: *load*.[[Status]] is either **`"loaded"`** or **`"linked"`**.
3. Repeat for each *load* in *linkSet*.[[Loads]],
    a. If *load*.[[Status]] is **`"loading"`**, then return.
4. Let *startingLoad* be the first element of the List *linkSet*.[[Loads]].
5. Let *status* be the result of Link(*linkSet*.[[Loads]], *linkSet*.[[Loader]]).
6. If *status* is an abrupt completion, then
    a. Call LinkSetFailed(*linkSet*, *status*.[[value]]).
    b. Return.
7. Assert: *linkSet*.[[Loads]] is an empty List.
8. Call the [[Call]] internal method of *linkSet*.[[Resolve]] passing undefined and (*startingLoad*) as arguments.
9. Assert: The call performed by step 8 completed normally.

1.1.2.4 LinkSetFailed(linkSet, exc) Abstract Operation

The LinkSetFailed abstract operation is called when a LinkSet fails. It detaches the given LinkSet Record from all Load Records and rejects the *linkSet*.[[Done]] Promise.

The following steps are taken:

1. Let *loader* be *linkSet*.[[Loader]].
2. Let *loads* be a copy of the List *linkSet*.[[Loads]].
3. For each *load* in *loads*,
    a. Assert: *linkSet* is an element of the List *load*.[[LinkSets]].
    b. Remove *linkSet* from the List *load*.[[LinkSets]].
    c. If *load*.[[LinkSets]] is empty and *load* is an element of the List *loader*.[[Loads]], then
        i. Remove *load* from the List *loader*.[[Loads]].
4. Call the [[Call]] internal method of *linkSet*.[[Reject]] passing undefined and (*exc*) as arguments.
5. Assert: The call performed by step 4 completed normally.

1.1.2.5 FinishLoad(loader, load) Abstract Operation

The FinishLoad Abstract Operation removes a completed Load Record from all LinkSets and commits the newly loaded Module to the registry. It performs the following steps:

1. Let *name* be *load*.[[Name]].
2. If *name* is not undefined, then
    a. Assert: There is no Record {[[key]], [[value]]} p that is an element of *loader*.[[Modules]], such that p.[[key]] is equal to *load*.[[Name]].
    b. Append the Record {[[key]]: *load*.[[Name]], [[value]]: *load*.[[Module]]} as the last element of *loader*.[[Modules]].
3. If *load* is an element of the List *loader*.[[Loads]], then
    a. Remove *load* from the List *loader*.[[Loads]].
4. For each *linkSet* in *load*.[[LinkSets]],
    a. Remove *load* from *linkSet*.[[Loads]].
5. Remove all elements from the List *load*.[[LinkSets]].

1.1.2.6 LoadModule(loader, name, options) Abstract Operation

The following steps are taken:

1. Let *name* be ToString(*name*).
2. ReturnIfAbrupt(*name*).
3. Let *address* be GetOption(*options*, **`"address"`**).
4. ReturnIfAbrupt(*address*).
5. Let *F* be a new anonymous function object as defined in AsyncStartLoadPartwayThrough.
6. Set *F*.[[Loader]] to *loader*.
7. Set *F*.[[ModuleName]] to *name*.
8. If *address* is undefined, set *F*.[[Step]] to **`"locate"`**.
9. Else, set *F*.[[Step]] to **`"fetch"`**.
10. Let *metadata* be the result of ObjectCreate(%ObjectPrototype%, ()).
11. Set *F*.[[ModuleMetadata]] to *metadata*.
12. Set *F*.[[ModuleSource]] to source.
13. Set *F*.[[ModuleAddress]] to *address*.
14. Return the result of calling OrdinaryConstruct(%Promise%, (*F*)).

1.1.3 AsyncStartLoadPartwayThrough Functions

An AsyncStartLoadPartwayThrough function is an anonymous function that creates a new Load Record and populates it with some information provided by the caller, so that loading can proceed from either the **`locate`** hook, the **`fetch`** hook, or the **`translate`** hook. This functionality is used to implement builtin methods like **`Loader.prototype.load`**, which permits the user to specify both the normalized module *name* and the *address*.

Each LoadSucceeded function has internal slots [[Loader]], [[ModuleName]], [[Step]], [[ModuleMetadata]], [[ModuleAddress]], and [[ModuleSource]].

When an AsyncStartLoadPartwayThrough function *F* is called with arguments *resolve* and *reject*, the following steps are taken:

1. Let *loader* be *F*.[[Loader]].
2. Let *name* be *F*.[[ModuleName]].
3. Let *step* be *F*.[[Step]].
4. Let *metadata* be *F*.[[ModuleMetadata]].
5. Let *address* be *F*.[[ModuleAddress]].

6. Let *source* be *F*.[[ModuleSource]].
7. If *loader*.[[Modules]] contains an entry whose [[key]] is equal to *name*, throw a TypeError exception.
8. If *loader*.[[Loads]] contains a Load Record whose [[Name]] field is equal to *name*, throw a TypeError exception.
9. Let *load* be the result of calling the CreateLoad abstract operation passing *name* as the single argument.
10. Set *load*.[[Metadata]] to *metadata*.
11. Let *linkSet* be the result of calling the CreateLinkSet abstract operation passing *loader* and *load* as arguments.
12. Add *load* to the List *loader*.[[Loads]].
13. Call the [[Call]] internal method of *resolve* with arguments null and (*linkSet*.[[Done]]).
14. If *step* is **"locate"**,
    a. Call ProceedToLocate(*loader*, *load*).
15. Else if *step* is **"fetch"**,
    a. Let *addressPromise* be PromiseOf(*address*).
    b. Call ProceedToFetch(*loader*, *load*, *addressPromise*).
16. Else,
    a. Assert: *step* is **"translate"**.
    b. Set *load*.[[Address]] to *address*.
    c. Let *sourcePromise* be PromiseOf(*source*).
    d. Call ProceedToTranslate(*loader*, *load*, *sourcePromise*).

1.1.4 EvaluateLoadedModule Functions

An EvaluateLoadedModule function is an anonymous function that is used by Loader.prototype.*module* and Loader.prototype.import to ensure that a *module* has been evaluated before it is passed to script code.

Each EvaluateLoadedModule function has a [[Loader]] internal slot.

When a EvaluateLoadedModule function *F* is called, the following steps are taken:

1. Let *loader* be *F*.[[Loader]].
2. Assert: load.[[Status]] is **"linked"**.
3. Let *module* be load.[[Module]].
4. Let *result* be the result of EnsureEvaluated(*module*, (), *loader*).
5. ReturnIfAbrupt(*result*).
6. Return *module*.

1.2 Module Linking

1.3 Module Evaluation

Module bodies are evaluated on demand, as late as possible. The loader uses the function **EnsureEvaluated**, defined below, to run scripts. The loader always calls **EnsureEvaluated** before returning a Module object to user code.

There is one way a module can be exposed to script before its body has been evaluated. In the case of an import cycle, whichever module is evaluated first can observe the others before they are evaluated. Simply put, we have to start somewhere: one of the modules in the cycle must run before the others.

1.3.1 EnsureEvaluated(mod, seen, loader) Abstract Operation

The abstract operation EnsureEvaluated walks the dependency graph of the module *mod*, evaluating any module bodies that have not already been evaluated (including, finally, *mod* itself). Modules are evaluated in depth-first, left-to-right, post order, stopping at cycles.

mod and its dependencies must already be linked.

The List *seen* is used to detect cycles. *mod* must not already be in the List *seen*.

On success, *mod* and all its dependencies, transitively, will have started to evaluate exactly once.

EnsureEvaluated performs the following steps:

1. Append *mod* as the last element of *seen*.
2. Let *deps* be *mod*.[[Dependencies]].
3. For each *pair* in *deps*, in List order,
    a. Let *dep* be *pair*.[[value]].
    b. If *dep* is not an element of *seen*, then
        i. Call EnsureEvaluated with the arguments *dep*, *seen*, and *loader*.
4. If *mod*.[[Body]] is not undefined and *mod*.[[Evaluated]] is false,
    a. Set *mod*.[[Evaluated]] to true.
    b. Let *initContext* be a new ECMAScript code execution context.
    c. Set *initContext*'s Realm to *loader*.[[Realm]].
    d. Set *initContext*'s VariableEnvironment to *mod*.[[Environment]].
    e. Set *initContext*'s LexicalEnvironment to *mod*.[[Environment]].
    f. If there is a currently running execution context, suspend it.
    g. Push *initContext* on to the execution context stack; *initContext* is now the running execution context.
    h. Let *r* be the result of evaluating *mod*.[[Body]].
    i. Suspend *initContext* and remove it from the execution context stack.
    j. Resume the context, if any, that is now on the top of the execution context stack as the running execution context.
    k. ReturnIfAbrupt(*r*).

1.4 Module Objects

A Module object has the following internal slots:

module.[[Environment]]	a Declarative Environment Record consisting of all bindings declared at toplevel in the module. The outerEnvironment of this environment record is a Global Environment Record.
module.[[Exports]]	a List of Export Records, {[[ExportName]]: a String, [[SourceModule]]: a Module, [[BindingName]]: a String}, such that the [[ExportName]]s of the records in the List are each unique.
module.[[Dependencies]]	a List of Modules or undefined. This is populated at link time by the loader and used by EnsureEvaluated.

The [[Prototype]] of a Module object is always null.

A Module object has accessor properties that correspond exactly to its [[Exports]], and no other properties. It is always non-extensible by the time it is exposed to ECMAScript code.

1.4.1 The Module Factory Function

The **`Module`** factory function reflectively creates module instance objects.

1.4.1.1 Constant Functions

A Constant function is a function that always returns the same value.

Each Constant function has a [[ConstantValue]] internal slot.

When a Constant function *F* is called, the following steps are taken:

1. Return *F*.[[ConstantValue]].

1.4.1.2 CreateConstantGetter(key, value) Abstract Operation

The CreateConstantGetter abstract operation creates and returns a new Function object that takes no arguments and returns *value*. It performs the following steps:

1. Let *getter* be a new Constant function.
2. Set the [[ConstantValue]] internal slot of *getter* to *value*.
3. Call SetFunctionName(*getter*, *key*, **`"get"`**).
4. Return *getter*.

1.4.1.3 Module (obj)

When the **`Module`** function is called with optional argument *obj*, the following steps are taken:

1. If Type(*obj*) is not Object, throw a TypeError exception.
2. Let *mod* be the result of calling the CreateLinkedModuleInstance abstract operation.
3. Let *keys* be the result of calling the ObjectKeys abstract operation passing *obj* as the argument.
4. ReturnIfAbrupt(*keys*).
5. For each *key* in *keys*, do
    a. Let *value* be the result of Get(*obj*, *key*).
    b. ReturnIfAbrupt(*value*).
    c. Let *F* be the result of calling CreateConstantGetter(*key*, *value*).
    d. Let *desc* be the PropertyDescriptor {[[Configurable]]: false, [[Enumerable]]: true, [[Get]]: *F*, [[Set]]: undefined}.
    e. Let *status* be the result of calling the DefinePropertyOrThrow abstract operation passing *mod*, *key*, and *desc* as arguments.
    f. ReturnIfAbrupt(*status*).
6. Call the [[PreventExtensions]] internal method of *mod*.
7. Return *mod*.

1.4.1.4 Module.prototype

The initial value of **`Module.prototype`** is null.

1.5 Realm Objects

1.5.1 The Realm Constructor

1.5.1.1 new Realm (options, initializer)

1. Let *realmObject* be the **this** value.
2. If Type(*realmObject*) is not Object, throw a TypeError exception.
3. If *realmObject* does not have all of the internal properties of a Realm object, throw a TypeError exception.
4. If *realmObject*.[[Realm]] is not undefined, throw a TypeError exception.
5. If *options* is undefined, then let *options* be the result of calling ObjectCreate(null, ()).
6. Else, if Type(*options*) is not Object, throw a TypeError exception.
7. Let *realm* be the result of CreateRealm(*realmObject*).
8. Let *evalHooks* be the result of Get(*options*, **"eval"**).
9. ReturnIfAbrupt(*evalHooks*).
10. If *evalHooks* is undefined then let *evalHooks* be the result of calling ObjectCreate(%ObjectPrototype%, ()).
11. Else, if Type(*evalHooks*) is not Object, throw a TypeError exception.
12. Let *directEval* be the result of Get(*evalHooks*, **"direct"**).
13. ReturnIfAbrupt(*directEval*).
14. If *directEval* is undefined then let *directEval* be the result of calling ObjectCreate(%ObjectPrototype%, ()).
15. Else, if Type(*directEval*) is not Object, throw a TypeError exception.
16. Let *translate* be the result of Get(*directEval*, **"translate"**).
17. ReturnIfAbrupt(*translate*).
18. If *translate* is not undefined and IsCallable(*translate*) is false, throw a TypeError exception.
19. Set *realm*.[[translateDirectEvalHook]] to *translate*.
20. Let *fallback* be the result of Get(*directEval*, **"fallback"**).
21. ReturnIfAbrupt(*fallback*).
22. If *fallback* is not undefined and IsCallable(*fallback*) is false, throw a TypeError exception.
23. Set *realm*.[[fallbackDirectEvalHook]] to *fallback*.
24. Let *indirectEval* be the result of Get(*options*, **"indirect"**).
25. ReturnIfAbrupt(*indirectEval*).
26. If *indirectEval* is not undefined and IsCallable(*indirectEval*) is false, throw a TypeError exception.
27. Set *realm*.[[indirectEvalHook]] to *indirectEval*.
28. Let *Function* be the result of Get(*options*, **"Function"**).
29. ReturnIfAbrupt(*Function*).
30. If *Function* is not undefined and IsCallable(*Function*) is false, throw a TypeError exception.
31. Set *realm*.[[FunctionHook]] to *Function*.
32. Set *realmObject*.[[Realm]] to *realm*.
33. If *initializer* is not undefined, then
    a. If IsCallable(*initializer*) is false, throw a TypeError exception.
    b. Let *builtins* be the result of calling ObjectCreate(%ObjectPrototype%, ()).
    c. Call the DefineBuiltinProperties abstract operation passing *realm* and *builtins* as arguments.
    d. Let *status* be the result of calling the [[Call]] internal method of the *initializer* function, passing *realmObject* as the **this** value and *builtins* as the single argument.
    e. ReturnIfAbrupt(*status*).
34. Return *realmObject*.

1.5.2 Properties of the Realm Prototype Object

1.5.2.1 Realm.prototype.global

`Realm.prototype.global` is an accessor property whose set accessor function is undefined. Its get accessor function performs the following steps:

1. Let *realmObject* be this Realm object.
2. If Type(*realmObject*) is not Object or *realmObject* does not have all the internal properties of a Realm object, throw a TypeError exception.
3. Return *realmObject*.[[Realm]].[[globalThis]].

1.5.2.2 Realm.prototype.eval (source)

The following steps are taken:

1. Let *realmObject* be this Realm object.
2. If Type(*realmObject*) is not Object or *realmObject* does not have all the internal properties of a Realm object, throw a TypeError exception.
3. Return the result of calling the IndirectEval abstract operation passing *realmObject*.[[Realm]] and *source* as arguments.

1.5.2.3 Realm [@@create] ( )

The @@create method of the builtin Realm constructor performs the following steps:

1. Let *F* be the **this** value.
2. Let *realmObject* be the result of calling OrdinaryCreateFromConstructor(*F*, "%RealmPrototype%", ([[Realm]])).
3. Return realm.

1.6 Loader Objects

Each Loader object has the following internal slots:

loader.[[Realm]]	The Realm associated with the loader. All scripts and modules evaluated by the loader run in the scope of the global object associated with this Realm.
loader.[[Modules]]	A List of Module Records: the module registry. This List only ever contains Module objects that are fully linked. However it can contain modules whose code has not yet been evaluated. Except in the case of cyclic imports, such modules are not exposed to user code. See **`EnsureEvaluated()`**.
loader.[[Loads]]	A List of Load Records. These represent ongoing asynchronous module loads. This List is stored in the loader so that multiple calls to **`loader.define()/.load()/.module()/.import()`** can cooperate to fetch what they need only once.

1.6.1 GetOption(options, name) Abstract Operation

The GetOption abstract operation is used to extract a property from an optional *options* argument.

The following steps are taken:

1. If *options* is undefined, then return undefined.
2. If Type(*options*) is not Object, then throw a TypeError exception.
3. Return the result of Get(*options*, *name*).

1.6.2 The Loader Constructor

1.6.2.1 Loader (options)

When the **`Loader`** function is called with optional argument *options* the following steps are taken:

1. Let *loader* be the **this** value.
2. If Type(*loader*) is not Object, throw a TypeError exception.
3. If *loader* does not have all of the internal properties of a Loader Instance, throw a TypeError exception.
4. If *loader*.[[Modules]] is not undefined, throw a TypeError exception.
5. If Type(*options*) is not Object, throw a TypeError exception.
6. Let *realmObject* be the result of Get(*options*, **`"realm"`**).
7. ReturnIfAbrupt(*realmObject*).
8. If *realmObject* is undefined, let realm be the Realm of the running execution context.
9. Else if Type(*realmObject*) is not Object or *realmObject* does not have all the internal properties of a Realm object, throw a TypeError exception.
10. Else let realm be *realmObject*.[[Realm]].
11. For each *name* in the List (**`"normalize"`**, **`"locate"`**, **`"fetch"`**, **`"translate"`**, **`"instantiate"`**),
    a. Let *hook* be the result of Get(*options*, *name*).
    b. ReturnIfAbrupt(*hook*).
    c. If *hook* is not undefined,
        i. Let *result* be the result of calling the [[DefineOwnProperty]] internal method of *loader* passing *name* and the Property Descriptor {[[Value]]: *hook*, [[Writable]]: true, [[Enumerable]]: true, [[Configurable]]: true} as arguments.
        ii. ReturnIfAbrupt(*result*).
12. Set *loader*.[[Modules]] to a new empty List.
13. Set *loader*.[[Loads]] to a new empty List.
14. Set *loader*.[[Realm]] to realm.
15. Return *loader*.

1.6.2.2 Loader [@@create] ( )

The @@create method of the builtin Loader constructor performs the following steps:

1. Let *F* be the **this** value.
2. Let *loader* be the result of calling OrdinaryCreateFromConstructor(*F*, "%LoaderPrototype%", ([[Modules]], [[Loads]], [[Realm]])).
3. Return *loader*.

1.6.3 Properties of the Loader Prototype Object

The abstract operation thisLoader(*value*) performs the following steps:

1. If Type(*value*) is Object and value has a [[Modules]] internal slot, then
    a. Let *m* be *value*.[[Modules]].
    b. If *m* is not **undefined**, then return *value*.
2. Throw a **TypeError** exception.

The phrase "this Loader" within the specification of a method refers to the result returned by calling the abstract operation thisLoader with the **this** value of the method invocation passed as the argument.

1.6.3.1 Loader.prototype.realm

`Loader.prototype.realm` is an accessor property whose set accessor function is undefined. Its get accessor function performs the following steps:

1. Let *loader* be this Loader.
2. If Type(*loader*) is not Object or *loader* does not have all the internal properties of a Loader object, throw a TypeError exception.
3. Return *loader*.[[Realm]].[[realmObject]].

1.6.3.2 Loader.prototype.global

`Loader.prototype.global` is an accessor property whose set accessor function is undefined. Its get accessor function performs the following steps:

1. Let *loader* be this Loader.
2. If Type(*loader*) is not Object or *loader* does not have all the internal properties of a Loader object, throw a TypeError exception.
3. Return *loader*.[[Realm]].[[globalThis]].

1.6.3.3 Loader.prototype.define (name, source, options = undefined)

The **`define`** method installs a module in the registry from *source*. The module is not immediately available. The **`translate`** and **`instantiate`** hooks are called asynchronously, and dependencies are loaded asynchronously.

`define` returns a Promise object that resolves to *undefined* when the new module and its dependencies are installed in the registry.

NOTE This is the dynamic equivalent of the proposed **`<module name=>`** element in HTML.

1. Let *loader* be this Loader.
2. ReturnIfAbrupt(*loader*).
3. Let *name* be ToString(*name*).
4. ReturnIfAbrupt(*name*).
5. Let *address* be GetOption(*options*, **`"address"`**).
6. ReturnIfAbrupt(*address*).
7. Let *metadata* be GetOption(*options*, **`"metadata"`**).
8. ReturnIfAbrupt(*metadata*).
9. If *metadata* is *undefined* then let *metadata* be the result of calling ObjectCreate(%ObjectPrototype%, ()).
10. Let *F* be a new anonymous function object as defined in AsyncStartLoadPartwayThrough.
11. Set *F*.[[Loader]] to *loader*.
12. Set *F*.[[ModuleName]] to *name*.
13. Set *F*.[[Step]] to the String **`"translate"`**.
14. Set *F*.[[ModuleMetadata]] to *metadata*.

15. Set *F*.[[ModuleSource]] to *source*.
16. Set *F*.[[ModuleAddress]] to *address*.
17. Let *p* be the result of calling OrdinaryConstruct(%Promise%, (*F*)).
18. Let *G* be a new anonymous function as defined by ReturnUndefined.
19. Let *p* be the result of calling PromiseThen(*p*, *G*).
20. Return *p*.

The **`length`** property of the **`define`** method is **2**.

1.6.3.4 ReturnUndefined Functions

A ReturnUndefined function is an anonymous function.

When a ReturnUndefined function is called, the following steps are taken:

1. Return undefined.

1.6.3.5 Loader.prototype.load (request, options = undefined)

The **`load`** method installs a module into the registry by name.

NOTE Combined with the **`normalize`** hook and **`Loader.prototype.get`**, this provides a close dynamic approximation of an ImportDeclaration.

1. Let *loader* be this Loader.
2. ReturnIfAbrupt(*loader*).
3. Let *p* be the result of LoadModule(*loader*, name, *options*).
4. ReturnIfAbrupt(*p*).
5. Let *f* be an anonymous function as described by ReturnUndefined.
6. Let *p* be the result of PromiseThen(*p*, *f*).
7. Return *p*.

The **`length`** property of the **`load`** method is **1**.

1.6.3.6 Loader.prototype.module (source, options)

The **`module`** method asynchronously evaluates a top-level, anonymous module from *source*.

The module's dependencies, if any, are loaded and committed to the registry. The anonymous module itself is not added to the registry.

`module` returns a Promise object that resolves to a new Module instance object once the given module body has been evaluated.

NOTE This is the dynamic equivalent of an anonymous **`<module>`** in HTML.

1. Let *loader* be this Loader.
2. ReturnIfAbrupt(*loader*).
3. Let *address* be GetOption(*options*, **`"address"`**).
4. ReturnIfAbrupt(*address*).
5. Let *load* be CreateLoad(undefined).
6. Set the [[Address]] field of *load* to addres.
7. Let *linkSet* be CreateLinkSet(*loader*, *load*).
8. Let *successCallback* be a new anonymous function object as defined by EvaluateLoadedModule.

9. Set *successCallback*.[[Loader]] to *loader*.
10. Set *successCallback*.[[Load]] to *load*.
11. Let *p* be the result of calling PromiseThen(*linkSet*.[[Done]], *successCallback*).
12. Let *sourcePromise* be PromiseOf(*source*).
13. Call the ProceedToTranslate abstract operation passing *loader*, *load*, and *sourcePromise* as arguments.
14. Return *p*.

The **`length`** property of the **`module`** method is **1**.

1.6.3.7 Loader.prototype.import (name, options)

The **`import`** method asynchronously loads, links, and evaluates a module and all its dependencies.

`import` returns a Promise that resolves to the requested **`Module`** object once it has been committed to the registry and evaluated.

NOTE This is the dynamic equivalent (when combined with normalization) of an ImportDeclaration.

1. Let *loader* be this Loader.
2. ReturnIfAbrupt(*loader*).
3. Let *p* be the result of calling LoadModule(*loader*, *name*, *options*).
4. ReturnIfAbrupt(*p*).
5. Let *F* be an anonymous function object as defined by EvaluateLoadedModule.
6. Set the [[Loader]] field of *F* to *loader*.
7. Let *p* be the result of calling PromiseThen(*p*, *F*).
8. Return *p*.

The **`length`** property of the **`import`** method is **1**.

1.6.3.8 Loader.prototype.eval (source)

The following steps are taken:

1. Let *loader* be this Loader.
2. ReturnIfAbrupt(*loader*).
3. Return the result of calling the IndirectEval abstract operation passing *loader*.[[Realm]] and *source* as arguments.

1.6.3.9 Loader.prototype.get (name)

If this Loader's *module* registry contains a Module with the given normalized *name*, return it. Otherwise, return undefined.

If the *module* is in the registry but has never been evaluated, first synchronously evaluate the bodies of the *module* and any dependencies that have not evaluated yet.

When the **`get`** method is called with one argument, the following steps are taken:

1. Let *loader* be this Loader.
2. ReturnIfAbrupt(*loader*).
3. Let *name* be ToString(*name*).
4. ReturnIfAbrupt(*name*).
5. Repeat for each Record {[[key]], [[value]]} p that is an element of *loader*.[[Modules]],

    a. If p.[[key]] is equal to *name*, then
        i. Let *module* be p.[[value]].
        ii. Let *result* be the result of EnsureEvaluated(*module*, (), *loader*).
        iii. ReturnIfAbrupt(*result*).
        iv. Return p.[[value]].
6. Return undefined.

1.6.3.10 **Loader.prototype.has (name)**

Return true if this Loader's module registry contains a Module with the given *name*. This method does not call any hooks or run any module code.

The following steps are taken:

1. Let *loader* be this Loader.
2. ReturnIfAbrupt(*loader*).
3. Let *name* be ToString(*name*).
4. ReturnIfAbrupt(*name*).
5. Repeat for each Record {[[*name*]], [[value]]} p that is an element of *loader*.[[Modules]],
    a. If p.[[key]] is equal to *name*, then return true.
6. Return false.

1.6.3.11 **Loader.prototype.set (name, module)**

Store a *module* in this Loader's *module* registry, overwriting any existing entry with the same *name*.

The following steps are taken:

1. Let *loader* be this Loader.
2. ReturnIfAbrupt(*loader*).
3. Let *name* be ToString(*name*).
4. ReturnIfAbrupt(*name*).
5. If *module* does not have all the internal slots of a Module instance, throw a TypeError exception.
6. Repeat for each Record {[[*name*]], [[value]]} *p* that is an element of *loader*.[[Modules]],
    a. If *p*.[[key]] is equal to *name*,
        i. Set *p*.[[value]] to *module*.
        ii. Return *loader*.
7. Let *p* be the Record {[[key]]: *name*, [[value]]: *module*}.
8. Append *p* as the last record of *loader*.[[Modules]].
9. Return *loader*.

1.6.3.12 **Loader.prototype.delete (name)**

Remove an entry from this *loader*'s module registry.

The following steps are taken:

1. Let *loader* be this Loader.
2. ReturnIfAbrupt(*loader*).
3. Let *name* be ToString(*name*).
4. ReturnIfAbrupt(*name*).
5. Repeat for each Record {[[*name*]], [[value]]} p that is an element of *loader*.[[Modules]],
    a. If p.[[key]] is equal to *name*,
        i. Set p.[[key]] to empty.

ii. Set p.[[value]] to empty.
iii. Return true.

6. Return false.

1.6.3.13 Loader.prototype.entries ( )

The following steps are taken.

1. Let *loader* be this Loader.
2. ReturnIfAbrupt(*loader*).
3. Return the result of CreateLoaderIterator(*loader*, **"key+value"**).

1.6.3.14 Loader.prototype.keys ( )

The following steps are taken.

1. Let *loader* be this Loader.
2. ReturnIfAbrupt(*loader*).
3. Return the result of CreateLoaderIterator(*loader*, **"key"**).

1.6.3.15 Loader.prototype.values ( )

The following steps are taken.

1. Let *loader* be this Loader.
2. ReturnIfAbrupt(*loader*).
3. Return the result of CreateLoaderIterator(*loader*, **"value"**).

1.6.3.16 Loader.prototype.normalize (name, refererName, refererAddress)

This hook receives the module *name* as written in the import declaration. It returns a string or a thenable for a string, the full module *name*, which is used for the rest of the import process. In particular, loader.[[Loads]] and loader.[[Modules]] are both keyed by normalized module names. Only a single load can be in progress for a given normalized module *name* at a time. The module registry can contain at most one module for a given module *name*.

When this hook is called: When a module body is parsed, once per distinct module specifier in that module body.

After calling this hook, if the full module *name* is in the registry or the load table, no new Load Record is created. Otherwise the loader kicks off a new Load, starting by calling the **`locate`** hook.

Default behavior: Return the module *name* unchanged.

When the normalize method is called, the following steps are taken:

1. Return *name*.

1.6.3.17 Loader.prototype.locate (load)

Given a normalized module name, determine the resource address (URL, path, etc.) to *load*.

The loader passes an argument, *load*, which is an ordinary Object with two own properties. **`load.name`** is the normalized name of the module to be located. **`load.metadata`** is a new

Object which the hook may use for any purpose. The Loader does not use this Object except to pass it to the subsequent loader hooks.

The hook returns either the resource address (any non-thenable value) or a thenable for the resource address. If the hook returns a thenable, loading will continue with the **`fetch()`** hook once the promise is fulfilled.

When this hook is called: For all imports, immediately after the **`normalize`** hook returns successfully, unless the module is already loaded or loading.

Default behavior: Return the module name unchanged.

NOTE The browser's **`System.locate`** hook may be considerably more complex.

When the locate method is called, the following steps are taken:

1. Return the result of Get(*load*, **`"name"`**).

1.6.3.18 Loader.prototype.fetch (load)

Fetch the requested source from the given address (produced by the **`locate`** hook).

This is the hook that must be overloaded in order to make the **`import`** keyword work.

The loader passes an argument, *load*, which is an ordinary Object with three own properties. **`load.name`** and **`load.metadata`** are the same values passed to the **`locate`** hook. **`load.address`** is the address of the resource to fetch. (This is the value produced by the **`locate`** hook.)

The fetch hook returns either module source (any non-thenable value) or a thenable for module source.

When this hook is called: For all modules whose source is not directly provided by the caller. It is not called for the module bodies provided as arguments to **`loader.module()`** or **`loader.define()`**, since those do not need to be fetched. (However, this hook may be called when loading dependencies of such modules.)

Default behavior: Throw a **`TypeError`**.

When the fetch method is called, the following steps are taken:

1. Throw a TypeError exception.

1.6.3.19 Loader.prototype.translate (load)

Optionally translate the given source from some other language into ECMAScript.

The loader passes an argument, *load*, which is an ordinary Object with four own properties. **`load.name`**, **`load.metadata`**, and **`load.address`** are the same values passed to the **`fetch`** hook. **`load.source`** is the source code to be translated. (This is the value produced by the **`fetch`** hook.)

The hook returns either an ECMAScript ModuleBody (any non-Promise value) or a thenable for a ModuleBody.

When this hook is called: For all modules, including module bodies passed to **`loader.module()`** or **`loader.define()`**.

Default behavior: Return the source unchanged.

When the translate method is called, the following steps are taken:

1. Return the result of Get(*load*, **`"source"`**).

1.6.3.20 Loader.prototype.instantiate (load)

Allow a loader to optionally provide interoperability with other module systems.

The loader passes an argument, *load*, which is an ordinary Object with four own properties. **`load.name`**, **`load.metadata`**, and **`load.address`** are the same values passed to the **`fetch`** and **`translate`** hooks. **`load.source`** is the translated module source. (This is the value produced by the **`translate`** hook.)

If the **`instantiate`** hook returns **undefined** or a thenable for the value **undefined**, then the loader uses the default linking behavior. It parses src as a Module, looks at its imports, loads its dependencies asynchronously, and finally links them together and adds them to the registry.

Otherwise, the hook should return a factory object (or a thenable for a factory object) which the loader will use to create the module and link it with its clients and dependencies.

The form of a factory object is:

```
{ deps: <array of strings (module names)>, execute: <function (Module, Module, ...) -> Module> }
```

The module is executed during the linking process. First all of its dependencies are executed and linked, and then passed to the **`execute`** function. Then the resulting module is linked with the downstream dependencies.

NOTE This feature is provided in order to permit custom loaders to support using **`import`** to import pre-ES6 modules such as AMD modules. The design requires incremental linking when such modules are present, but it ensures that modules implemented with standard source-level module declarations can still be statically validated.

When this hook is called: For all modules, after the **`translate`** hook.

Default behavior: Return undefined.

When the instantiate method is called, the following steps are taken:

1. Return **undefined**.

1.6.3.21 Loader.prototype[@@iterator] ( )

The initial value of the @@iterator property is the same function object as the initial value of the entries property.

1.6.4 Loader Iterator Objects

A Loader Iterator object represents a specific iteration over the module registry of some specific Loader instance object.

Loader Iterator objects are similar in structure to Map Iterator objects. They are created with three internal slots:

[[Loader]]	The Loader object whose module registry is being iterated.
[[ModuleMapNextIndex]]	The integer index of the next element of [[Loader]].[[Modules]] to be examined by this iteration.
[[MapIterationKind]]	A string value that identifies what is to be returned for each element of the iteration. The possible values are: **"key"**, **"value"**, **"key+value"**.

1.6.4.1 CreateLoaderIterator(loader, kind) Abstract Operation

Several methods of Loader objects return Loader Iterator objects. The abstract iteration CreateLoaderIterator is used to create such *iterator* objects. It performs the following steps:

1. Assert: Type(*loader*) is Object.
2. Assert: *loader* has all the internal slots of a Loader object.
3. Let *iterator* be the result of ObjectCreate(%LoaderIteratorPrototype%, ([[Loader]], [[ModuleMapNextIndex]], [[MapIterationKind]])).
4. Set *iterator*.[[Loader]] to *loader*.
5. Set *iterator*.[[ModuleMapNextIndex]] to 0.
6. Set *iterator*.[[MapIterationKind]] to *kind*.
7. Return *iterator*.

1.6.4.2 The %LoaderIteratorPrototype% Object

All Loader Iterator Objects inherit properties from the %LoaderIteratorPrototype% intrinsic object. The %LoaderIteratorPrototype% intrinsic object is an ordinary object and its [[Prototype]] internal slot is the %ObjectPrototype% intrinsic object. In addition, %LoaderIteratorPrototype% has the following properties:

1.6.4.2.1 %LoaderIteratorPrototype%.next ( )

1. Let *O* be the **this** value.
2. If Type(*O*) is not Object, throw a TypeError exception.
3. If *O* does not have all of the internal properties of a Loader Iterator Instance, throw a TypeError exception.
4. Let *loader* be the value of the [[Loader]] internal slot of *O*.
5. Let *index* be the value of the [[ModuleMapNextIndex]] internal slot of *O*.
6. Let *itemKind* be the value of the [[MapIterationKind]] internal slot of *O*.
7. Assert: *loader* has a [[Modules]] internal slot and *loader* has been initialised so the value of *loader*.[[Modules]] is not undefined.
8. Repeat while *index* is less than the total number of elements of *loader*.[[Modules]],

a. Let *e* be the Record {[[key]], [[value]]} at 0-origined insertion position *index* of *loader*.[[Modules]].
b. Set *index* to *index* + 1.
c. Set the [[ModuleMapNextIndex]] internal slot of *O* to *index*.
d. If *e*.[[key]] is not empty, then
    i. If *itemKind* is **"key"**, then let *result* be *e*.[[key]].
    ii. Else if *itemKind* is **"value"**, then let *result* be *e*.[[value]].
    iii. Else,
        1. Assert: *itemKind* is **"key+value"**.
        2. Let *result* be the result of ArrayCreate(2).
        3. Assert: *result* is a new, well-formed Array object so the following operations will never fail.
        4. Call CreateOwnDataProperty(*result*, **"0"**, *e*.[[key]]).
        5. Call CreateOwnDataProperty(*result*, **"1"**, *e*.[[value]]).
    iv. Return CreateIterResultObject(*result*, false).
9. Return CreateIterResultObject(undefined, true).

1.6.4.2.2 %LoaderIteratorPrototype% [@@iterator] ()

The following steps are taken:

1. Return the **this** value.

The value of the **name** property of this function is **"[Symbol.iterator]"**.

1.6.4.2.3 %LoaderIteratorPrototype% [@@toStringTag]

The initial value of the @@toStringTag property is the string value **"Loader Iterator"**.

New section: Modules and Module Loaders

Subsection: Module Instance Objects

CreateUnlinkedModuleInstance (body, boundNames, knownExports, unknownExports, imports)

When the abstract operation CreateUnlinkedModuleInstance is called with arguments *body*, *boundNames*, *knownExports*, *unknownExports*, and *imports*, the following steps are taken:

1. Let *M* be a new object with [[Prototype]] **null**.
2. Set *M*.[[Body]] to *body*.
3. Set *M*.[[BoundNames]] to *boundNames*.
4. Set *M*.[[KnownExportEntries]] to *knownExports*.
5. Set *M*.[[UnknownExportEntries]] to *unknownExports*.
6. Set *M*.[[ExportDefinitions]] to **undefined**.
7. Set *M*.[[Exports]] to **undefined**.
8. Set *M*.[[Dependencies]] to **undefined**.
9. Set *M*.[[UnlinkedDependencies]] to **undefined**.
10. Set *M*.[[ImportEntries]] to *imports*.
11. Set *M*.[[ImportDefinitions]] to **undefined**.
12. Set *M*.[[LinkErrors]] to a new empty List.
13. Let *realm* be the current realm.
14. Let *globalEnv* be *realm*.[[*globalEnv*]].
15. Let *env* be the result of calling the NewModuleEnvironment abstract operation passing *globalEnv* as the argument.
16. Set *M*.[[Environment]] to *env*.
17. Return *M*.

LookupModuleDependency (M, requestName)

When the abstract operation LookupModuleDependency is called with arguments *M* and *requestName*, the following steps are taken:

1. If *requestName* is **null** then return *M*.
2. Let *pair* be the record in *M*.[[Dependencies]] such that *pair*.[[Key]] is equal to *requestName*.
3. Return *pair*.[[Module]].

LookupExport (M, exportName)

When the abstract operation LookupExport is called with arguments *M* and *exportName*, the following steps are taken:

1. If *M*.[[Exports]] does not contain a record *export* such that *export*.[[ExportName]] is equal to *exportName*, then return **undefined**.
2. Let *export* be the record in *M*.[[Exports]] such that *export*.[[ExportName]] is equal to *exportName*.
3. Return *export*.[[Binding]].

Subsection: Module Linking

ResolveExportEntries (M, visited)

When the abstract operation ResolveExportEntries is called with arguments *M* and *visited*, the following steps are taken:

1. If *M*.[[ExportDefinitions]] is not **undefined**, then return *M*.[[ExportDefinitions]].
2. Let *defs* be a new empty List.
3. Let *boundNames* be *M*.[[BoundNames]].
4. For each *entry* in *M*.[[KnownExportEntries]], do
    a. Let *modReq* be *entry*.[[ModuleRequest]].
    b. Let *otherMod* be the result of calling the LookupModuleDependency abstract operation passing *M* and *modReq* as arguments.
    c. If *entry*.[[Module]] is **null** and *entry*.[[LocalName]] is not **null** and *boundNames* does not contain *entry*.[[LocalName]], then the following steps are taken:
        i. Let *error* be a new Reference Error.
        ii. Add *error* to *M*.[[LinkErrors]].
    d. Add the record {[[Module]]: *otherMod*, [[ImportName]]: *entry*.[[ImportName]], [[LocalName]]: *entry*.[[LocalName]], [[ExportName]]: *entry*.[[ExportName]], [[Explicit]]: **true**} to *defs*.
5. For each *modReq* in *M*.[[UnknownExportEntries]], do
    a. Let *otherMod* be the result of calling the LookupModuleDependency abstract operation passing *M* and *modReq* as arguments.
    b. If *otherMod* is in *visited*, then the following steps are taken:
        i. Let *error* be a new Syntax Error.
        ii. Add *error* to *M*.[[LinkErrors]].
    c. Otherwise the following steps are taken:
        i. Add *otherMod* to *visited*.
        ii. Let *otherDefs* be the result of calling the ResolveExportEntries abstract operation passing *otherMod* and *visited* as arguments.
        iii. For each *def* of *otherDefs*, do
            1. Add the record {[[Module]]: *otherMod*, [[ImportName]]: *def*.[[ExportName]], [[LocalName]]: **null**, [[ExportName]]: *def*.[[ExportName]], [[Explicit]]: **false**} to *defs*.
6. Set *M*.[[ExportDefinitions]] to *defs*.
7. Return *defs*.

ResolveExports (M)

When the abstract operation ResolveExports is called with argument *M*, the following steps are taken:

1. For each *def* in *M*.[[ExportDefinitions]], do
    a. Call the ResolveExport abstract operation with arguments *M*, *def*.[[ExportName]], and a new empty List.

ResolveExport (M, exportName, visited)

When the abstract operation ResolveExport is called with arguments *M*, *exportName*, and *importName*, the following steps are taken:

1. Let *exports* be *M*.[[Exports]].

2. If *exports* has a record *export* such that *export*.[[ExportName]] is equal to *exportName*, return *export*.[[Binding]].
3. Let *ref* be {[[Module]]: *M*, [[ExportName]]: *exportName*}.
4. If *visited* contains a record equal to *ref* then the following steps are taken:
    a. Let *error* be a new Syntax Error.
    b. Add *error* to *M*.[[LinkErrors]].
    c. Return *error*.
5. Let *defs* be *M*.[[ExportDefinitions]].
6. Let *overlappingDefs* be the List of records *def* in *defs* such that *def*.[[ExportName]] is equal to *exportName*.
7. If *overlappingDefs* is empty, then the following steps are taken:
    a. Let *error* be a new Reference Error.
    b. Add *error* to *M*.[[LinkErrors]].
    c. Return *error*.
8. If *overlappingDefs* has more than one record *def* such that *def*.[[Explicit]] is **true**, or if it has length greater than 1 but contains no records *def* such that *def*.[[Explicit]] is **true**, then the following steps are taken:
    a. Let *error* be a new Syntax Error.
    b. Add *error* to *M*.[[LinkErrors]].
    c. Return *error*.
9. Let *def* be the unique record in *overlappingDefs* such that *def*.[[Explicit]] is **true**, or if there is no such record let *def* be the unique record in *overlappingDefs*.
10. If *def*.[[LocalName]] is not **null**, then the following steps are taken:
    a. Let *binding* be the record {[[Module]]: *M*, [[LocalName]]: *def*.[[LocalName]]}.
    b. Let *export* be the record {[[ExportName]]: *exportName*, [[Binding]]: *binding*}.
    c. Add *export* to *exports*.
    d. Return *binding*.
11. Add *ref* to *visited*.
12. Let *binding* be the result of calling the ResolveExport abstract operation passing *def*.[[Module]] and *def*.[[ImportName]] as arguments.
13. Return *binding*.

ResolveImportEntries (M)

When the abstract operation ResolveImportEntries is called with argument *M*, the following steps are taken:

1. Let *entries* be *M*.[[ImportEntries]].
2. Let *defs* be a new empty List.
3. For each *entry* in *entries*, do
    a. Let *modReq* be *entry*.[[ModuleRequest]].
    b. Let *otherMod* be the result of calling the LookupModuleDependency abstract operation passing *M* and *modReq* as arguments.
    c. Add the record {[[Module]]: *otherMod*, [[ImportName]]: *entry*.[[ImportName]], [[LocalName]]: *entry*.[[LocalName]]} to *defs*.
4. Return *defs*.

LinkImports (M)

When the abstract operation LinkImports is called with argument *M*, the following steps are taken:

1. Let *envRec* be *M*.[[Environment]].
2. Let *defs* be *M*.[[ImportDefinitions]].
3. For each *def* in *defs*, do

a. If *def*.[[ImportName]] is **module**, then the following steps are taken:
    i. Call the CreateImmutableBinding concrete method of *envRec* passing *def*.[[LocalName]] as the argument.
    ii. Call the InitializeImmutableBinding concrete method of *envRec* passing *def*.[[LocalName]] and *def*.[[Module]] as the arguments.
b. Otherwise, the following steps are taken:
    i. Let *binding* be the result of calling the ResolveExport abstract operation passing *def*.[[Module]] and *def*.[[ImportName]] as the arguments.
    ii. If *binding* is **undefined**, then the following steps are taken:
        1. Let *error* be a new Reference Error.
        2. Add *error* to *M*.[[LinkErrors]].
    iii. Otherwise, call the CreateImportBinding concrete method of *envRec* passing *def*.[[LocalName]] and *binding* as the arguments.

LinkDeclarativeModules (loads, loader)

When the abstract operation LinkDeclarativeModules is called with arguments *loads* and *loader*, the following steps are taken:

1. Let *unlinked* be a new empty List.
2. For each *load* in *loads*, do
    a. If *load*.[[Status]] is not **linked** then the following steps are taken:
        i. Let *body* be *load*.[[Body]].
        ii. Let *boundNames* be the BoundNames of *body*.
        iii. Let *knownExports* be the KnownExportEntries of *body*.
        iv. Let *unknownExports* be the UnknownExportEntries of *body*.
        v. Let *imports* be the ImportEntries of *body*.
        vi. Let *module* be the result of calling the CreateUnlinkedModuleInstance abstract operation passing *body*, *boundNames*, *knownExports*, *unknownExports*, and *imports* as the arguments.
        vii. Let *pair* be the record {[[Module]]: *module*, [[Load]]: *load*}.
        viii. Add *pair* to *unlinked*.
3. For each *pair* in *unlinked*, do
    a. Let *resolvedDeps* be a new empty List.
    b. Let *unlinkedDeps* be a new empty List.
    c. For each *dep* in *pair*.[[Load]].[[Dependencies]], do
        i. Let *requestName* be *dep*.[[Key]].
        ii. Let *normalizedName* be *dep*.[[Value]].
        iii. If *loads* contains a record *load* such that *load*.[[Name]] equals *normalizedName*, then the following steps are taken:
            1. If *load*.[[Status]] is **linked** then the following steps are taken:
                a. Let *resolvedDep* be the record {[[Key]]: *requestName*, [[Value]]: *load*.[[Module]]}.
                b. Add *resolvedDep* to *resolvedDeps*.
            2. Otherwise, the following steps are taken:
                a. Let *otherPair* be the record in *unlinked* such that *otherPair*.[[Load]].[[Name]] is equal to *normalizedName*.
                b. Add the record {[[Key]]: *requestName*, [[Value]]: *otherPair*.[[Module]]} to *resolvedDeps*.
                c. Add *otherPair*.[[Load]] to *unlinkedDeps*.
        iv. Otherwise, the following steps are taken:
            1. Let *module* be the result of calling the LoaderRegistryLookup abstract operation with arguments *loader* and *normalizedName*.
            2. If *module* is **null** then the following steps are taken:
                a. Let *error* be a new Reference Error.

            b. Add *error* to *pair*.[[Module]].[[LinkErrors]].
        3. Otherwise, add the record {[[Key]]: *requestName*, [[Value]]: *module*} to *resolvedDeps*.
    d. Set *pair*.[[Module]].[[Dependencies]] to *resolvedDeps*.
    e. Set *pair*.[[Module]].[[UnlinkedDependencies]] to *unlinkedDeps*.
4. For each *pair* in *unlinked*, do
    a. Call the ResolveExportEntries abstract operation passing *pair*.[[Module]] and a new empty List as arguments.
    b. Call the ResolveExports abstract operation passing *pair*.[[Module]] as the argument.
5. For each *pair* in *unlinked*, do
    a. Call the ResolveImportEntries abstract operation passing *pair*.[[Module]] as the argument.
    b. Call the LinkImports abstract operation with argument *pair*.[[Module]].
6. If there exists a *pair* in *unlinked* such that *pair*.[[Module]].[[LinkErrors]] is not empty, choose one of the link errors and throw it.
7. For each *pair* in *unlinked*, do
    a. Set *pair*.[[Load]].[[Module]] to *pair*.[[Module]].
    b. Set *pair*.[[Load]].[[Status]] to **linked**.
    c. Let *r* be the result of calling the FinishLoad abstract operation passing *loader* and *pair*.[[Load]] as the arguments.
    d. ReturnIfAbrupt(*r*).

LinkDynamicModules (loads, loader)

When the abstract operation LinkDynamicModules is called with arguments *loads* and *loader*, the following steps are taken:

1. For each *load* in *loads*, do
    a. Let *factory* be *load*.[[Factory]].
    b. Let *module* be the result of calling *factory* with no arguments.
    c. ReturnIfAbrupt(*module*).
    d. If *module* does not have all the internal data properties of a Module Instance Object, then throw a new Type Error.
    e. Set *load*.[[Module]] to *module*.
    f. Set *load*.[[Status]] to **linked**.
    g. Let *r* be the result of calling the FinishLoad abstract operation passing *loader* and *load* as the arguments.
    h. ReturnIfAbrupt(*r*).

Link (start, loader)

When the abstract operation Link is called with argument *start*, the following steps are taken:

1. Let *groups* be the result of calling the LinkageGroups abstract operation with argument *start*.
2. For each *group* in *groups*:
    a. If the [[Kind]] of each element of *group* is **declarative**, then call the LinkDeclarativeModules abstract operation passing *group* and *loader* as arguments.
    b. Otherwise, call the LinkDynamicModules abstract operation passing *group* and *loader* as arguments.

Subsection: Module Linking Groups

A load record $load_1$ has a ***linkage dependency*** on a load record $load_2$ if $load_2$ is contained in $load_1$.[[UnlinkedDependencies]] or there exists a load record *load* in $load_1$.[[UnlinkedDependencies]] such that *load* has a linkage dependency on $load_2$.

The ***linkage graph*** of a List of load records is the set of load records *load* such that some load record in the list has a linkage dependency on *load*.

A ***dependency chain*** from $load_1$ to $load_2$ is a List of load records demonstrating the transitive linkage dependency from $load_1$ to $load_2$.

A ***dependency cycle*** is a dependency chain whose first and last elements' [[Name]] fields have the same value.

A dependency chain is ***cyclic*** if it contains a subsequence that is a dependency cycle. A dependency chain is ***acyclic*** if it is not cyclic.

A dependency chain is ***mixed*** if there are two elements with distinct values for their [[Kind]] fields.

A ***dependency group transition*** of kind *kind* is a two-element subsequence $load_1$, $load_2$ of a dependency chain such that $load_1$.[[Kind]] is not equal to *kind* and $load_2$.[[Kind]] is equal to *kind*.

The ***dependency group count*** of a dependency chain with first element $load_1$ is the number of distinct dependency group transitions of kind $load_1$.[[Kind]].

LinkageGroups (start)

When the abstract operation LinkageGroups is called with argument *start*, the following steps are taken:

1. Let *G* be the linkage graph of *start*.
2. If there are any mixed dependency cycles in *G*, throw a new Syntax Error.
3. For each *load* in *G*, do
    a. Let *n* be the largest dependency group count of all acyclic dependency chains in *G* starting from *load*.
    b. Set *load*.[[GroupIndex]] to *n*.
4. Let *declarativeGroupCount* be the largest [[GroupIndex]] of any *load* in *G* such that *load*.[[Kind]] is **declarative**.
5. Let *declarativeGroups* be a new List of length *declarativeGroupCount* where each element is a new empty List.
6. Let *dynamicGroupCount* be the largest [[GroupIndex]] of any *load* in *G* such that *load*.[[Kind]] is **dynamic**.
7. Let *dynamicGroups* be a new List of length *dynamicGroupCount* where each element is a new empty List.
8. Let *visited* be a new empty List.
9. For each *load* in *start*, do
    a. Call the BuildLinkageGroups abstract operation passing *load*, *declarativeGroups*, *dynamicGroups*, and *visited* as arguments.
10. If any *load* in the first element of *declarativeGroups* has a dependency on a load record of [[Kind]] **dynamic**, then let *groups* be a List constructed by interleaving the elements of *dynamicGroups* and *declarativeGroups*, starting with the former; otherwise let *groups* be a List constructed by interleaving the elements of *declarativeGroups* and *dynamicGroups*, starting with the former.
11. Return *groups*.

BuildLinkageGroups (load, declarativeGroups, dynamicGroups, visited)

When the abstract operation BuildLinkageGroups is called with arguments *load*, *declarativeGroups*, and *dynamicGroups*, the following steps are taken:

1. If *visited* contains an element whose [[Name]] is equal to *load*.[[Name]], then return.
2. Add *load* to *visited*.
3. For each *dep* of *load*.[[UnlinkedDependencies]], do
    a. Call the BuildLinkageGroups abstract operation passing *dep*, *declarativeGroups*, *dynamicGroups*, and *visited* as arguments.
4. Let *i* be *load*.[[GroupIndex]].
5. If *load*.[[Kind]] is **declarative** let *groups* be *declarativeGroups*; otherwise let *groups* be *dynamicGroups*.
6. Let *group* be the *i*th element of *groups*.
7. Add *load* to *group*.