# Changes to 13.2.1 (“Let and Const Declarations”)

**Syntax**

*BindingIdentifier* :

*Identifier*

**default**

**Static Semantics: Early Errors**

*BindingIdentifier* : **default**

* It is a Syntax Error if the innermost *Declaration* or *VariableStatement* containing this production is not immediately contained in an *ExportDeclaration*.

# Changes to 15.1 (“Script”)

**Syntax**

*Script* :

*ScriptBodyopt*

*ScriptBody* :

*ScriptOuterItemList*

*ScriptOuterItemList* :

*ScriptOuterItem*

*ScriptOuterItemList* *ScriptOuterItem*

*ScriptOuterItem* :

*ModuleDeclaration*

*ImportDeclaration*

*StatementListItem*

# Changes to 15.3 (“Modules”)

**Syntax**

*Module* :

*ModuleBodyopt*

*ModuleBody* :

*ModuleOuterItemList*

*ModuleOuterItemList* :

*ModuleOuterItem*

*ModuleOuterItemList* *ModuleOuterItem*

*ModuleOuterItem* :

*ModuleDeclaration*

*ImportDeclaration*

*ExportDeclaration*

*StatementListItem*

*ModuleDeclaration* :

**module** [no *LineTerminator* here] *Identifier* **from** *ModuleSpecifier* **;**

*ModuleSpecifier* :

*StringLiteral*

*ImportDeclaration* :

**import** *ImportSpecifierSet* **from** *ModuleSpecifier* ;

**import** *ModuleSpecifier* **;**

*ImportSpecifierSet* :

*Identifier*

**{** **}**

**{** *ImportSpecifierList* **}**

**{** *ImportSpecifierList* **,** **}**

*ImportSpecifierList* :

*ImportSpecifier*

*ImportSpecifierList* **,** *ImportSpecifier*

*ImportSpecifier* :

*Identifier*

*Identifier* **as** *Identifier*

*ReservedWord* **as** *Identifier*

*ExportDeclaration* :

**export** *ExportSpecifierSet* **;**

**export** *ExportSpecifierSet* **from** *ModuleSpecifier* **;**

**export** *VariableStatement* **;**

**export** *Declaration* **;**

**export** *BindingList* **;**

*ExportSpecifierSet* :

*ExplictExportSpecifierSet*

**\***

*ExplicitExportSpecifierSet* :

**{** **}**

**{** *ExportSpecifierList* **}**

**{** *ExportSpecifierList* **,** **}**

*ExportSpecifierList* :

*ExportSpecifier*

*ExportSpecifierList* **,** *ExportSpecifier*

*ExportSpecifier* :

*Identifier*

*Identifier* **as** *IdentifierName*

**Static Semantics: Early Errors**

*Script* : *ScriptBodyopt*

* It is a Syntax Error if the ImportedNames of *ScriptBodyopt* contains any duplicate elements.

*Module* : *ModuleBodyopt*

* It is a Syntax Error if the ImportedNames of *ModuleBodyopt* contains any duplicate elements.

*ImportDeclaration* : **import** *ImportSpecifierSet* **from** *ModuleSpecifier* **;**

* It is a Syntax Error if the BoundNames of *ImportSpecifierSet* contains the string “**eval**” or the string “**arguments**”.

**Static Semantics: BoundNames**

*ImportDeclaration* : **import** *ImportSpecifierSet* **from** *ModuleSpecifier* **;**

1. Let *names* be the ImportedNames of *ImportSpecifierSet*.
2. Return *names*.

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

1. Return a new empty List.

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

1. Let *names* be the BoundNames of *VariableStatement*.
2. Return *names*.

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

1. Let *names* be the BoundNames of *Declaration*.
2. Return *names*.

*ExportDeclaration* : **export** *BindingList* **;**

1. Let *names* be the BoundNames of *BindingList*.
2. Return *names*.

*ModuleDeclaration* : **module** [no *LineTerminator* here] *Identifier* **from** *ModuleSpecifier* **;**

1. Let *names* be a new List containing *Identifier*.
2. Return *names*.

**Static Semantics: ImportedNames**

*ImportDeclaration* : **import** *ImportSpecifierSet* **from** *ModuleSpecifier* **;**

1. Return the ImportedNames of *ImportSpecifierSet*.

*ImportSpecifierSet* : *Identifier*

1. Return a new List containing *Identifier*.

*ImportSpecifierSet* : **{** **}**

1. Return a new empty List.

*ImportSpecifierSet* : **{** *ImportSpecifierList* **}**

*ImportSpecifierSet* : **{** *ImportSpecifierList* **,** **}**

1. Return the ImportedNames of *ImportSpecifierList*.

*ImportSpecifierList* : *ImportSpecifier*

1. Return the ImportedNames of *ImportSpecifier*.

*ImportSpecifierList* : *ImportSpecifierList* **,** *ImportSpecifier*

1. Let *names* be the ImportedNames of *ImportSpecifierList*.
2. Append to *names* the elements of the ImportedNames of *ImportSpecifier*.
3. Return *names*.

*ImportSpecifier* : *Identifier*

1. Return a new List containing *Identifier*.

*ImportSpecifier* : *Identifier* **as** *Identifier*

1. Return a new List containing the second *Identifier*.

*ImportSpecifier* : *ReservedWord* **as** *Identifier*

1. Return a new List containing *Identifier*.

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

1. Return a new empty List.

**Static Semantics: KnownExportNames**

*ExportDeclaration* : **export** **\*;**

1. Return a new List containing the record {[[Name]]: **all**, [[Kind]]: **local**}.

*ExportDeclaration* : **export** *ExplicitExportSpecifierSet* **;**

1. Let *exports* be a new empty List.
2. Let *pairs* be the ExportPairs of *ExplicitExportSpecifierSet*.
3. For each *pair* in *pairs*, do:
   1. Let *e* be the record {[[Name]]: *pair*.[[External]], [[Kind]]: **local**}.
   2. Add *e* to *exports*.
4. Return *exports*.

*ExportDeclaration* : **export** *ExplicitExportSpecifierSet* **from** *ModuleSpecifier* **;**

1. Let *pairs* be the ExportPairs of *ExportSpecifierSet*.
2. Let *exports* be a new empty List.
3. For each *pair* in *pairs*, do:
   1. Let *e* be *pair*.[[External]].
   2. Add the record {[[Name]]: *e*, [[Kind]]: **reexport**} to *exports*.
4. Return *exports*.

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

1. Let *names* be the BoundNames of *VariableStatement*.
2. Let *exports* be a new empty List.
3. For each *name* in *names*, do:
   1. Add the record {[[Name]]: *name*, [[Kind]]: **local**} to *exports*.
4. Return *exports*.

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

1. Let *names* be the BoundNames of *Declaration*.
2. Let *exports* be a new empty List.
3. For each *name* in *names*, do:
   1. Add the record {[[Name]]: *name*, [[Kind]]: **local**} to *exports*.
4. Return *exports*.

*ExportDeclaration* : **export** *BindingList* **;**

1. Let *names* be the BoundNames of *BindingList*.
2. Let *exports* be a new empty List.
3. For each *name* in *names*, do:
   1. Add the record {[[Name]]: *name*, [[Kind]]: **local**} to *exports*.
4. Return *exports*.

**Static Semantics: ExportPairs**

*ExplicitExportSpecifierSet* : **{** **}**

1. Return a new empty List.

*ExplicitExportSpecifierSet* : **{** *ExportSpecifierList* **}**

*ExplicitExportSpecifierSet* : **{** *ExportSpecifierList* **,** **}**

1. Let *pairs* be the ExportPairs of *ExportSpecifierList*.
2. Return *pairs.*

*ExportSpecifierList* : *ExportSpecifier*

1. Let *pairs* be the ExportPairs of *ExportSpecifier*.
2. Return *pairs*.

*ExportSpecifierList* : *ExportSpecifierList* **,** *ExportSpecifier*

1. Let *pairs* be the ExportPairs of *ExportSpecifierList*.
2. Append the ExportPairs of *ExportSpecifier* to *pairs*.
3. Return *pairs*.

*ExportSpecifier* : *Identifier*

1. Let *pairs* be a new List containing the record {[[Internal]]: *Identifier*, [[External]]: *Identifier*}.
2. Return *pairs*.

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

1. Let *pairs* be a new List containing the record {[[Internal]]: *Identifier*, [[External]]: *IdentifierName*}.
2. Return *pairs*.

**Static Semantics: UnknownExportNames**

*ExportDeclaration* : **export** **\*****from** *ModuleSpecifier* **;**

1. Return a new List containing *ModuleSpecifier*.

**Static Semantics: ModuleExportNames**

*Module* : *ModuleBodyopt*

1. If *ModuleBodyopt* is not present, return a new empty List.
2. Let *boundNames* be the BoundNames of *ModuleBodyopt*.
3. Let *importedNames* be the ImportedNames of *ModuleBodyopt*.
4. Let *localNames* be the set difference of *boundNames* and *importedNames*.
5. Let *exports* be a new empty List.
6. For each *known* in the KnownExportNames of *ModuleBodyopt*, do
   1. If *known*.[[Name]] is **all**, append *localNames* to *exports*.
   2. Otherwise if *known*.[[Kind]] is **reexport**, add *known*.[[Name]] to *exports*.
   3. Otherwise *known*.[[Kind]] is **local** and the following steps are taken.
      1. Let *name* be *known*.[[Name]].
      2. If *name* is not in *boundNames*, then
         1. Let *error* be a new ReferenceError.
         2. Add {[[Error]]: *error*} to *exports*.
      3. Otherwise, add *name* to *exports*.
7. Let *unknown* be the UnknownExportNames of *ModuleBodyopt*.
8. For each *module* in *unknown*, do
   1. Add {[[Unknown]]: *module*} to *exports*.
9. Return *exports*.

**Static Semantics: LinkageConstraints**

*ModuleDeclaration* : **module** [no *LineTerminator* here] *Identifier* **from** *ModuleSpecifier* **;**

1. Return a new List containing the record {[[Module]]: *ModuleSpecifier*, [[ImportName]]: **module**, [[LocalName]]: *Identifier*, [[ExportName]]: **null**, [[Implicit]]: **false**}.

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

1. Return a new List containing the record {[[Module]]: *ModuleSpecifier*, [[ImportName]]: **module**, [[LocalName]]: **null**, [[ExportName]]: **null**, [[Implicit]]: **false**}.

*ImportDeclaration* : **import** *ImportSpecifierSet* **from** *ModuleSpecifier* **;**

1. Let *constraints* be a new List containing the record {[[Module]]: *ModuleSpecifier*, [[ImportName]]: **module**, [[LocalName]]: **null**, [[ExportName]]: **null**, [[Implicit]]: **false** }.
2. Let *pairs* be the ImportPairs of *ImportSpecifierSet*.
3. For each *pair* of *pairs*, do
   1. Add the record {[[Module]]: *ModuleSpecifier*, [[ImportName]]: *pair*.[[External]], [[LocalName]]: *pair*.[[Internal]], [[ExportName]]: **null**, [[Implicit]]: **false**} to *constraints*.
4. Return *constraints*.

*ExportDeclaration* : **export** **\*;**

1. Return a new List containing the record {[[Module]]: **null**, [[ImportName]]: **null**, [[LocalName]]: **all**, [[ExportName]]: **all**, [[Implicit]]: **false**}.

*ExportDeclaration* : **export** *ExplicitExportSpecifierSet* **;**

1. Let *constraints* be a new empty List.
2. Let *pairs* be the ExportPairs of *ExplicitExportSpecifierSet*.
3. For each *pair* in *pairs*, do
   1. Add the record {[[Module]]: **null**, [[ImportName]]: **null**, [[LocalName]]: *pair*.[[Internal]], [[ExportName]]: *pair*.[[External]], [[Implicit]]: **false**} to *constraints*.
4. Return *constraints*.

*ExportDeclaration* : **export** *ExplicitExportSpecifierSet* **from** *ModuleSpecifier* **;**

1. Let *constraints* be a new empty List.
2. Let *pairs* be the ExportPairs of *ExplicitExportSpecifierSet*.
3. For each *pair* in *pairs*, do
   1. Add the record {[[Module]]: *ModuleSpecifier*, [[ImportName]]: *pair*.[[Internal]], [[LocalName]]: **null**, [[ExportName]]: *pair*.[[External]], [[Implicit]]: **false**} to *constraints*.
4. Return *constraints*.

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

1. Let *constraints* be a new empty List.
2. Let *names* be the BoundNames of *VariableStatement*.
3. For each *name* in *names*, do
   1. Add the record {[[Module]]: **null**, [[ImportName]]: **null**, [[LocalName]]: *name*, [[ExportName]]: *name*, [[Implicit]]: **false**} to *constraints*.
4. Return *constraints*.

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

1. Let *constraints* be a new empty List.
2. Let *names* be the BoundNames of *Declaration*.
3. For each *name* in *names*, do
   1. Add the record {[[Module]]: **null**, [[ImportName]]: **null**, [[LocalName]]: *name*, [[ExportName]]: *name*, [[Implicit]]: **false**} to *constraints*.
4. Return *constraints*.

*ExportDeclaration* : **export** *BindingList* **;**

1. Let *constraints* be a new empty List.
2. Let *names* be the BoundNames of *BindingList*.
3. For each *name* in *names*, do
   1. Add the record {[[Module]]: **null**, [[ImportName]]: **null**, [[LocalName]]: *name*, [[ExportName]]: *name*, [[Implicit]]: **false**} to *constraints*.
4. Return *constraints*.

*ExportDeclaration* : **export** **\*****from** *ModuleSpecifier* **;**

1. Return a new List containing the record {[[Module]]: *ModuleSpecifier*, [[ImportName]]: **all**, [[LocalName]]: **null**, [[ExportName]]: **all**, [[Implicit]]: **false**}.

**Static Semantics: ImportPairs**

*ImportSpecifierSet* : *Identifier*

1. Return a new List containing the record {[[Internal]]: *Identifier*, [[External]]: “**default**”}.

*ImportSpecifierSet* : **{** **}**

1. Return a new empty List.

*ImportSpecifierSet* : **{** *ImportSpecifierList* **}**

*ImportSpecifierSet* : **{** *ImportSpecifierList* **,** **}**

1. Return the ImportPairs of *ImportSpecifierList*.

*ImportSpecifierList* : *ImportSpecifier*

1. Let *pairs* be the ImportPairs of *ImportSpecifier*.
2. Return *pairs*.

*ImportSpecifierList* : *ImportSpecifierList* **,** *ImportSpecifier*

1. Let *pairs* be the ImportPairs of *ImportSpecifierList*.
2. Append the ImportPairs of *ImportSpecifier* to *pairs*.
3. Return *pairs*.

*ImportSpecifier* : *Identifier*

1. Return a new List containing the record {[[Internal]]: *Identifier*, [[External]]: *Identifier*}.

*ImportSpecifier* : *IdentifierName* **as** *Identifier*

1. Return a new List containing the record {[[Internal]]: *Identifier*, [[External]]: *IdentifierName*}.

**Static Semantics: ModuleLinkageConstraints**

*Module* : *ModuleBodyopt*

1. If *ModuleBodyopt* is not present, return a new empty List.
2. Let *boundNames* be the BoundNames of *ModuleBodyopt*.
3. Let *importedNames* be the ImportedNames of *ModuleBodyopt*.
4. Let *localNames* be the set difference of *boundNames* and *importedNames*.
5. Let *constraints* be a new empty List.
6. For each *constraint* in the LinkageConstraints of *ModuleBodyopt*, do
   1. If *constraint*.[[Module]] is **null** and *constraint*.[[ExportName]] is **all**, then the following steps are taken:
      1. For each *localName* in *localNames*, do
         1. Add the record {[[Module]]: **null**, [[ImportName]]: **null**, [[LocalName]]: *localName*, [[ExportName]]: *localName*, [[Implicit]]: **true**} to *constraints*.
   2. Otherwise, add *constraint* to *constraints*.
7. Return *constraints*.

## New section: Modules and Module Loaders

**ModuleInstanceExportNames ( M, visited )**

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

1. If *M*.[[ExportedNames]].[[Resolved]] is **true**, then return *M*.[[ExportedNames]].[[Value]].
2. Let *exports* be a new empty List.
3. For each *export* in *M*.[[ExportedNames]].[[Value]], do
   1. If *export* is a string or a record with an [[Error]] field, add *export* to *exports*.
   2. Otherwise *export* is a record with an [[Unknown]] field, and the following steps are taken:
      1. Let *modSpec* be *export*.[[Unknown]].
      2. Let *otherMod* be the result of calling the abstract operation LookupModuleDependency with arguments *M* and *modSpec*.
      3. Let *otherMod* be *r*.[[Module]].
      4. If *otherMod* is in *visited*, then
         1. Let *error* be a new SyntaxError.
         2. Add the record {[[Error]]: *error*} to *exports*.
      5. Otherwise, the following steps are taken:
         1. Add *otherMod* to *visited*.
         2. Let *otherExports* be the result of calling the ModuleInstanceExportNames abstract operation with arguments *otherMod* and *visited*.
         3. Append *otherExports* to *exports*.
4. Set *M*.[[ExportedNames]] to the record {[[Resolved]]: **true**, [[Value]]: *exports*}.
5. Return *exports*.

**ModuleInstanceLinkageConstraints ( M )**

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

1. If *M*.[[LinkageConstraints]].[[Resolved]] is **true**, then return *M*.[[LinkageConstraints]].[[Value]].
2. Let *constraints* be a new empty List.
3. For each *constraint* in *M*.[[LinkageConstraints]].[[Value]], do
   1. Let *modSpec* be *constraint*.[[Module]].
   2. If *modSpec* is **null**, then add the record {[[Module]]: *M*, [[ImportName]]: *constraint*.[[ImportName]], [[LocalName]]: *constraint*.[[LocalName]], [[ExportName]]: *constraint*.[[ExportName]], [[Implicit]]: *constraint*.[[Implicit]]} to *constraints*.
   3. Otherwise, the following steps are taken:
      1. Let *otherMod* be the result of calling the LookupModuleDependency abstract operation with arguments *M* and *modSpec*.
      2. If *constraint*.[[ExportName]] is **all**, then the following steps are taken:
         1. Let *visited* be a new empty List.
         2. Let *names* be the result of calling the ModuleInstanceExportNames abstract operation with arguments *otherMod* and *visited*.
         3. For each *name* in *names*, do
            1. Add the record {[[Module]]: *otherMod*, [[ImportName]]: *name*, [[LocalName]]: **null**, [[ExportName]]: *name*, [[Implicit]]: **true**} to *constraints*.
      3. Otherwise, add the record {[[Module]]: *otherMod*, [[ImportName]]: *constraint*.[[ImportName]], [[LocalName]]: *constraint*.[[LocalName]], [[ExportName]]: *constraint*.[[ExportName]], [[Implicit]]: *constraint*.[[Implicit]]} to *constraints*.
4. Set *M*.[[LinkageConstraints]] to the record {[[Resolved]]: **true**, [[Value]]: *constraints*}.
5. Return *constraints*.

**ResolveExports ( M )**

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

1. Let *constraints* be *M*.[[LinkageConstraints]].[[Value]].
2. For each *constraint* in *constraints*, do
   1. If *constraint*.[[ExportName]] is not **null** then call the ResolveExport abstract operation with arguments *M*, *constraint*.[[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 *pair* be {[[Module]]: *M*, [[ExportName]]: *exportName*}.
4. If *visited* contains a record equal to *pair* then the following steps are taken:
   1. Let *error* be a new SyntaxError.
   2. Let *export* be the record {[[ExportName]]: *exportName*, [[Binding]]: *error*}.
   3. Add *export* to *exports*.
   4. Return *error*.
5. Let *constraints* be *M*.[[LinkageConstraints]].[[Value]].
6. Let *overlappingConstraints* be the List of records *constraint* in *constraints* such that *constraint*.[[ExportName]] is equal to *exportName*.
7. If *overlappingConstraints* is empty, or if it has more than one record *constraint* such that *constraint*.[[Implicit]] is **false**, or if it has length greater than 1 but contains no records *constraint* such that *constraint*.[[Implicit]] is **false**, then the following steps are taken:
   1. Let *error* be a new SyntaxError.
   2. Let *export* be the record {[[ExportName]]: *exportName*, [[Binding]]: *error*}.
   3. Add *export* to *exports*.
   4. Return *error*.
8. Let *constraint* be the unique record in *overlappingConstraints* such that *constraint*.[[Implicit]] is **false**, or if there is no such record let *export* be the unique record in *overlappingConstraints*.
9. If *constraint*.[[LocalName]] is not **null**, then the following steps are taken:
   1. Let *binding* be the record {[[Module]]: *M*, [[LocalName]]: *constraints*.[[LocalName]]}.
   2. Let *export* be the record {[[ExportName]]: *exportName*, [[Binding]]: *binding*}.
   3. Add *export* to *exports*.
   4. Return *binding*.
10. Assert: *constraint*.[[ImportName]] is not **null**.
11. Add *pair* to *visited*.
12. Let *binding* be the result of calling the ResolveExport abstract operation with arguments *constraint*.[[Module]] and *constraint*.[[ImportName]].
13. Return *binding*.

**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
   1. If *load*.[[Status]] is not **linked** then the following steps are taken:
      1. Let *module* be the result of calling the CreateModuleInstance abstract operation.
      2. Let *pair* be the record {[[Module]]: *module*, [[Load]]: *load*}.
      3. Add *pair* to *unlinked*.
3. For each *pair* in *unlinked*, do
   1. Let *resolvedDeps* be a new empty List.
   2. Let *unlinkedDeps* be a new empty List.
   3. For each *dep* in *pair*.[[Load]].[[Dependencies]], do
      1. Let *requestName* be *dep*.[[RequestName]].
      2. Let *normalizedName* be *dep*.[[NormalizedName]].
      3. If *loads* contains a record *load* such that *load*.[[NormalizedName]] equals *normalizedName*, then the following steps are taken:
         1. If *load*.[[Status]] is **linked** then the following steps are taken:
            1. Let *resolvedDep* be the record {[[RequestName]]: *requestName*, [[Module]]: *load*.[[Module]]}.
            2. Add *resolvedDep* to *resolvedDeps*.
         2. Otherwise, the following steps are taken:
            1. Let *otherPair* be the record in *unlinked* such that *otherPair*.[[Load]].[[NormalizedName]] is equal to *normalizedName*.
            2. Let *resolvedDep* be the record {[[RequestName]]: *requestName*, [[Module]]: *otherPair*.[[Module]]}.
            3. Add *resolvedDep* to *resolvedDeps*.
            4. Add *otherPair*.[[Load]] to *unlinkedDeps*.
      4. 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 set *module* to a new ReferenceError.
         3. Let *resolvedDep* be the record {[[RequestName]]: *requestName*, [[Module]]: *module*}.
         4. Add *resolvedDep* to *resolvedDeps*.
   4. Set *pair*.[[Module]].[[Dependencies]] to *resolvedDeps*.
   5. Set *pair*.[[Module]].[[UnlinkedDependencies]] to *unlinkedDeps*.
4. For each *pair* in *unlinked*, do
   1. Call the ResolveExports abstract operation with argument *pair*.[[Module]].
5. For each *pair* in *unlinked*, do
   1. Call the LinkImports abstract operation with argument *pair*.[[Module]].
6. FIXME: If there are any errors anywhere within *unlinked*, throw one of them.
7. For each *pair* in *unlinked*, do
   1. Set *pair*.[[Load]].[[Module]] to *pair*.[[Module]].
   2. Set *pair*.[[Load]].[[Status]] to **linked**.
   3. Call the FinalizeLoadRecord abstract operation passing *pair*.[[Load]] as the argument.

**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 *constraints* be *M*.[[LinkageConstraints]].[[Value]].
3. For each *constraint* in *constraints*, do
   1. If *constraint*.[[ExportName]] is **null** and *constraint*.[[LocalName]] is not **null** and *constraint*.[[ImportName]] is **module**, then the following steps are taken:
      1. Call the CreateImmutableBinding concrete method of *envRec* passing *constraint*.[[LocalName]] as the argument.
      2. Call the InitializeImmutableBinding concrete method of *envRec* passing *constraint*.[[LocalName]] and *constraint*.[[Module]] as the arguments.
   2. Otherwise, if *constraint*.[[ExportName]] is **null** and *constraint*.[[LocalName]] is not **null**, then the following steps are taken:
      1. Let *binding* be the result of calling the ResolveExport abstract operation passing *constraint*.[[Module]] and *constraint*.[[ImportName]] as the arguments.
      2. Call the CreateImportBinding concrete method of *envRec* passing *constraint*.[[LocalName]] and *binding* as the arguments.

**LookupModuleDependency ( M, requestName )**

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

1. Let *pair* be the record in *M*.[[Dependencies]] such that *pair*.[[RequestName]] is equal to *requestName*.
2. Return *pair*.[[Module]].

A load record *load1* has a ***linkage dependency*** on a load record *load2* if *load2* is contained in *load1*.[[UnlinkedDependencies]] or there exists a load record *load* in *load1*.[[UnlinkedDependencies]] such that *load* has a linkage dependency on *load2*.

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 *load1* to *load2* is a list of load records demonstrating the transitive linkage dependency from *load1* to *load2*.

A ***dependency cycle*** is a dependency chain whose first and last elements’ [[NormalizedName]] 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 *load1*, *load2* of a dependency chain such that *load1*.[[Kind]] is not equal to *kind* and *load2*.[[Kind]] is equal to *kind*.

The ***dependency group count*** of a dependency chain with first element *load1* is the number of distinct dependency group transitions of kind *load1*.[[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 SyntaxError.
3. For each *load* in *G*, do
   1. Let *n* be the largest dependency group count of all acyclic dependency chains in *G* starting from *load*.
   2. 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
   1. Call the BuildLinkageGroups abstract operation passing *load*, *declarativeGroups*, *dynamicGroups*, and *visited* as arguments.
10. Let *firstDeclarativeGroup* be the first element of *declarativeGroups*.
11. Let *firstDynamicGroup* be the first element of *dynamicGroups*.
12. If any *load* in *firstDeclarativeGroup* has a dependency on a load record of [[Kind]] **dynamic**, then let *firstGroup* be **dynamic** and *secondGroup* be **declarative**; otherwise let *firstGroup* be **declarative** and *secondGroup* be **dynamic**.
13. Return {[[FirstGroup]]: *firstGroup*, [[SecondGroup]]: *secondGroup*, [[Declarative]]: *declarativeGroups*, [[Dynamic]]: *dynamicGroups*}.

**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 [[NormalizedName]] is equal to *load*.[[NormalizedName]], then return.
2. Add *load* to *visited*.
3. For each *dep* of *load*.[[UnlinkedDependencies]], do
   1. 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*.

**LinkModules ( loads, loader, kind )**

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

1. If *kind* is **declarative** then return the result of calling LinkDeclarativeModules passing *loads* and *loader* as the arguments.
2. Otherwise, return the result of calling LinkDynamicModules passing *loads* and *loader* as the arguments.

**LinkDynamicModules ( loads, loader )**

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

1. Let.

**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. Let *firstKind* be *groups*.[[FirstGroup]].
3. Let *secondKind* be *groups*.[[SecondGroup]].
4. If *firstKind* is **declarative** then let *firstGroup* be *groups*.[[Declarative]] and *secondGroup* be *groups*.[[Dynamic]]; otherwise let *firstGroup* be *groups*.[[Dynamic]] and *secondGroup* be *groups*.[[Declarative]].
5. Let *firstGroupCount* be the length of *firstGroup*.
6. Let *secondGroupCount* be the length of *secondGroup*.
7. Let *n* be the maximum of *firstGroupCount* and *secondGroupCount*.
8. For each *i* from 0 to *n – 1* (inclusive):
   1. If *i < firstGroupCount*, then the following steps are taken:
      1. Let *r* be the result of calling the LinkModules abstract operation passing *group*, *loader*, and *firstKind* as arguments.
      2. ReturnIfAbrupt(*r*).
   2. If *i < secondGroupCount*, then the following steps are taken:
      1. Let *r* be the result of calling the LinkModules abstract operation passing *group*, *loader*, and *secondKind* as arguments.

**DefineBuiltinProperties ( O )**

When the abstract operation DefineBuiltinProperties is called with object *O*, the following steps are taken:

1. For each name *P* of the standard properties of the global object (see 18), in alphabetical order, do:
   1. Let *V* be the value of the standard built-in property *P* for the current realm.
   2. Let *W* be the value of the [[Writable]] attribute of the standard built-in property *P*.
   3. Let *E* be the value of the [[Enumerable]] attribute of the standard built-in property *P*.
   4. Let *C* be the value of the [[Configurable]] attribute of the standard built-in property *P*.
   5. Let *newDesc* be the Property Descriptor {[[Value]]: *V*, [[Writable]]: *W*, [[Enumerable]]: *E*, [[Configurable]]: *C*}.
   6. Let *status* be the result of calling the [[DefineOwnProperty]] internal method of *O* passing *P* and *newDesc* as arguments.
   7. ReturnIfAbrupt(*status*).
2. Return *O*.

# Changes to 15.1.2 (“Script Evaluation”)

**Runtime Semantics: Script Evaluation**

*Script* : *ScriptBodyopt*

With arguments *realm*, *deletableBindings*, and *moduleDependencies*.

(before the existing step 1)

1. For each *mod* in *moduleDependencies*, do:
   1. Let *status* be the result of performing Module Evaluation of *mod* using *realm*, *deletableBindings*, and *mod*.[[Dependencies]] as arguments.
   2. ReturnIfAbrupt(*status*).