# 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 ImportNames of *ScriptBodyopt* contains any duplicate elements.

*Module* : *ModuleBodyopt*

* It is a Syntax Error if the ImportNames 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 ImportNames 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: ImportNames**

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

1. Let *names* be a new empty List.
2. Let *pairs* be the ImportPairs of *ImportSpecifierSet*.
3. For each *pair* of *pairs*, do
   1. Add *pair*.[[LocalName]] to *names*.
4. Return *names*.

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

1. Return a new empty List.

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

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

**Static Semantics: ImportPairs**

*ImportSpecifierSet* : *Identifier*

1. Return a new List containing the record {[[LocalName]]: *Identifier*, [[ImportName]]: “**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 {[[LocalName]]: *Identifier*, [[ImportName]]: *Identifier*}.

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

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

**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 {[[LocalName]]: *Identifier*, [[ExportName]]: *Identifier*}.
2. Return *pairs*.

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

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

**Static Semantics: ModuleRequests**

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

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

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

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

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

1. Return a new List containing *ModuleSpecifier*.

**Static Semantics: KnownExportSpecs**

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

1. Return a new List containing the record {[[ModuleRequest]]: **null**, [[ImportName]]: **null**, [[LocalName]]: **all**, [[ExportName]]: **all**, [[Explicit]]: **true**}.

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

1. Let *specs* be a new empty List.
2. Let *pairs* be the ExportPairs of *ExplicitExportSpecifierSet*.
3. For each *pair* in *pairs*, do
   1. Add the record {[[ModuleRequest]]: **null**, [[ImportName]]: **null**, [[LocalName]]: *pair*.[[LocalName]], [[ExportName]]: *pair*.[[ExportName]], [[Explicit]]: **true**} to *specs*.
4. Return *specs*.

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

1. Let *specs* be a new empty List.
2. Let *pairs* be the ExportPairs of *ExplicitExportSpecifierSet*.
3. For each *pair* in *pairs*, do
   1. Add the record {[[ModuleRequest]]: *ModuleSpecifier*, [[ImportName]]: *pair*.[[LocalName]], [[LocalName]]: **null**, [[ExportName]]: *pair*.[[ExportName]], [[Explicit]]: **true**} to *specs*.
4. Return *specs*.

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

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

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

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

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

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

**Static Semantics: UnknownExportSpecs**

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

1. Return a new List containing the record {[[ModuleRequest]]: *ModuleSpecifier*, [[ImportName]]: **all**, [[LocalName]]: **null**, [[ExportName]]: **all**, [[Explicit]]: **true**}.

**Static Semantics: ImportSpecs**

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

1. Return a new List containing the record {[[ModuleRequest]]: *ModuleSpecifier*, [[ImportName]]: **module**, [[LocalName]]: *Identifier*}.

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

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

**Static Semantics: ModuleKnownExportSpecs**

*Module* : *ModuleBodyopt*

1. If *ModuleBodyopt* is not present, return a new empty List.
2. Let *boundNames* be the BoundNames of *ModuleBodyopt*.
3. Let *importNames* be the ImportNames of *ModuleBodyopt*.
4. Let *localNames* be the set difference of *boundNames* and *importNames*.
5. Let *specs* be a new empty List.
6. For each *spec* in the KnownExportSpecs of *ModuleBodyopt*, do
   1. If *spec*.[[ModuleRequest]] is **null** and *spec*.[[ExportName]] is **all**, then the following steps are taken:
      1. For each *localName* in *localNames*, do
         1. Add the record {[[ModuleRequest]]: **null**, [[ImportName]]: **null**, [[LocalName]]: *localName*, [[ExportName]]: *localName*, [[Explicit]]: **false**} to *specs*.
   2. Otherwise, if *spec*.[[ModuleRequest]] is **null** and *spec*.[[LocalName]] is not **null** and *boundNames* does not contain *spec*.[[LocalName]], then the following steps are taken:
      1. Let *error* be a new Reference Error.
      2. Add *error* to *specs*.
   3. Otherwise, add *spec* to *specs*.
7. Return *specs*.

## New section: Modules and Module Loaders

### Subsection: Module Instance Objects

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

When the abstract operation CreateUnlinkedModuleInstance is called with arguments *body*, *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*.[[KnownExportSpecs]] to *knownExports*.
4. Set *M*.[[UnknownExportSpecs]] to *unknownExports*.
5. Set *M*.[[ExportDefinitions]] to **undefined**.
6. Set *M*.[[Exports]] to **undefined**.
7. Set *M*.[[Dependencies]] to **undefined**.
8. Set *M*.[[UnlinkedDependencies]] to **undefined**.
9. Set *M*.[[ImportSpecs]] to *imports*.
10. Set *M*.[[ImportDefinitions]] to **undefined**.
11. Set *M*.[[LinkErrors]] to a new empty List.
12. Let *realm* be the current realm.
13. Let *globalEnv* be *realm*.[[*globalEnv*]].
14. Let *env* be the result of calling the NewModuleEnvironment abstract operation passing *globalEnv* as the argument.
15. Set *M*.[[Environment]] to *env*.
16. Return *M*.

**LookupModuleDependency ( M, requestName )**

When the abstract operation LookupModuleDependency is called with arguments *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]].

**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

**ResolveExportSpecs ( M, visited )**

When the abstract operation ResolveExportSpecs 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. For each *spec* in *M*.[[KnownExportSpecs]], do
   1. Let *modReq* be *spec*.[[ModuleRequest]].
   2. Let *otherMod* be the result of calling the LookupModuleDependency abstract operation passing *M* and *modReq* as arguments.
   3. Add the record {[[Module]]: *otherMod*, [[ImportName]]: *spec*.[[ImportName]], [[LocalName]]: *spec*.[[LocalName]], [[ExportName]]: *spec*.[[ExportName]], [[Explicit]]: *spec*.[[Explicit]]} to *defs*.
4. For each *modReq* in *M*.[[UnknownExportSpecs]], do
   1. Let *otherMod* be the result of calling the LookupModuleDependency abstract operation passing *M* and *modReq* as arguments.
   2. If *otherMod* is in *visited*, then the following steps are taken:
      1. Let *error* be a new Syntax Error.
      2. Add *error* to *M*.[[LinkErrors]].
   3. Otherwise the following steps are taken:
      1. Add *otherMod* to *visited*.
      2. Let *otherDefs* be the result of calling the ResolveExportSpecs abstract operation passing *otherMod* and *visited* as arguments.
      3. For each *def* of *otherDefs*, do
         1. Add the record {[[Module]]: *otherMod*, [[ImportName]]: *def*.[[ExportName]], [[LocalName]]: **null**, [[ExportName]]: *def*.[[ExportName]], [[Explicit]]: **false**} to *defs*.
5. Set *M*.[[ExportDefinitions]] to *defs*.
6. 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
   1. 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:
   1. Let *error* be a new Syntax Error.
   2. Add *error* to *M*.[[LinkErrors]].
   3. 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:
   1. Let *error* be a new Reference Error.
   2. Add *error* to *M*.[[LinkErrors]].
   3. 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:
   1. Let *error* be a new Syntax Error.
   2. Add *error* to *M*.[[LinkErrors]].
   3. 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:
    1. Let *binding* be the record {[[Module]]: *M*, [[LocalName]]: *def*.[[LocalName]]}.
    2. Let *export* be the record {[[ExportName]]: *exportName*, [[Binding]]: *binding*}.
    3. Add *export* to *exports*.
    4. 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*.

**ResolveImportSpecs ( M )**

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

1. Let *specs* be *M*.[[ImportSpecs]].
2. Let *defs* be a new empty List.
3. For each *spec* in *specs*, do
   1. Let *modReq* be *spec*.[[ModuleRequest]].
   2. Let *otherMod* be the result of calling the LookupModuleDependency abstract operation passing *M* and *modReq* as arguments.
   3. Add the record {[[Module]]: *otherMod*, [[ImportName]]: *spec*.[[ImportName]], [[LocalName]]: *spec*.[[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
   1. If *def*.[[ImportName]] is **module**, then the following steps are taken:
      1. Call the CreateImmutableBinding concrete method of *envRec* passing *def*.[[LocalName]] as the argument.
      2. Call the InitializeImmutableBinding concrete method of *envRec* passing *def*.[[LocalName]] and *def*.[[Module]] as the arguments.
   2. Otherwise, the following steps are taken:
      1. Let *binding* be the result of calling the ResolveExport abstract operation passing *def*.[[Module]] and *def*.[[ImportName]] as the arguments.
      2. If *binding* is **undefined**, then the following steps are taken:
         1. Let *error* be a new Reference Error.
         2. Add *error* to *M*.[[LinkErrors]].
      3. 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
   1. If *load*.[[Status]] is not **linked** then the following steps are taken:
      1. Let *body* be *load*.[[Body]].
      2. Let *knownExports* be the KnownExportSpecs of *body*.
      3. Let *unknownExports* be the UnknownExportSpecs of *body*.
      4. Let *imports* be the ImportSpecs of *body*.
      5. Let *module* be the result of calling the CreateUnlinkedModuleInstance abstract operation passing *body*, *knownExports*, *unknownExports*, and *imports* as the arguments.
      6. Let *pair* be the record {[[Module]]: *module*, [[Load]]: *load*}.
      7. 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. Add the record {[[RequestName]]: *requestName*, [[Module]]: *otherPair*.[[Module]]} to *resolvedDeps*.
            3. 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 the following steps are taken:
            1. Let *error* be a new Reference Error.
            2. Add *error* to *pair*.[[Module]].[[LinkErrors]].
         3. Otherwise, add the record {[[RequestName]]: *requestName*, [[Module]]: *module*} 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 ResolveExportSpecs abstract operation passing *pair*.[[Module]] and a new empty List as arguments.
   2. Call the ResolveExports abstract operation passing *pair*.[[Module]] as the argument.
5. For each *pair* in *unlinked*, do
   1. Call the ResolveImportSpecs abstract operation passing *pair*.[[Module]] as the argument.
   2. 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
   1. Set *pair*.[[Load]].[[Module]] to *pair*.[[Module]].
   2. Set *pair*.[[Load]].[[Status]] to **linked**.
   3. Let *r* be the result of calling the FinalizeLoadRecord abstract operation passing *loader* and *pair*.[[Load]] as the arguments.
   4. 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
   1. Let *factory* be *load*.[[Factory]].
   2. Let *module* be the result of calling *factory* with no arguments.
   3. ReturnIfAbrupt(*module*).
   4. If *module* does not have all the internal data properties of a Module Instance Object, then throw a new Type Error.
   5. Set *load*.[[Module]] to *module*.
   6. Set *load*.[[Status]] to **linked**.
   7. Let *r* be the result of calling the FinalizeLoadRecord abstract operation passing *loader* and *load* as the arguments.
   8. 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*:
   1. If the [[Kind]] of each element of *group* is **declarative**, then call the LinkDeclarativeModules abstract operation passing *group* and *loader* as arguments.
   2. Otherwise, call the LinkDynamicModules abstract operation passing *group* and *loader* as arguments.

### Subsection: Module Linking Groups

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 Syntax Error.
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. 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 [[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*.

### Subsection: Module Loaders

**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*.