

Contexts and Shadowing

Contexts are a mechanism for ensuring that symbols created by a package are distinct from symbols created by the user or by other packages. The biggest problem with contexts is shadowing, a situation in which a symbol in one context prevents the user from referring to a symbol of the same name in a different context. In this article, we first describe how packages use contexts and then we survey tools and techniques that exist for dealing with the shadowing problem. Finally, we develop a new package, `AntiShadow.m`, that prevents the most common form of shadowing.

David B. Wagner

Even the most experienced *Mathematica* user makes a mistake like this occasionally:

```
In[1]:= Show[Polyhedron[Dodecahedron]]
Show::gtype: Polyhedron is not a type of graphics.
Out[1]= Show[Polyhedron[Dodecahedron]]
```

The symbols `Polyhedron` and `Dodecahedron` are defined in one of the standard packages, `Graphics`Polyhedra``. Realizing the error, the user belatedly loads the package. Strange warning messages appear.

```
In[2]:= Needs["Graphics`Polyhedra`"]
Polyhedron::shdw:
Warning: Symbol Polyhedron
appears in multiple contexts
{Graphics`Polyhedra`, Global`}; definitions in context
Graphics`Polyhedra` may shadow or be shadowed by other
definitions.
Dodecahedron::shdw:
Warning: Symbol Dodecahedron
appears in multiple contexts
{Graphics`Polyhedra`, Global`}; definitions in context
Graphics`Polyhedra` may shadow or be shadowed by other
definitions.
```

Even though the package has been loaded, it appears that *Mathematica* still doesn't know about these symbols.

```
In[3]:= Show[Polyhedron[Dodecahedron]]
Show::gtype: Polyhedron is not a type of graphics.
Out[3]= Show[Polyhedron[Dodecahedron]]
```

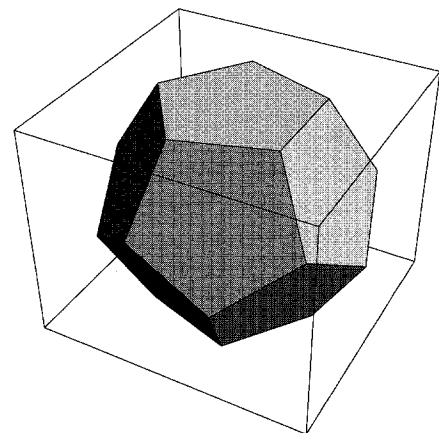
David B. Wagner is the president and founder of Principia Consulting, a training and consulting firm specializing in Mathematica and other technical software packages. He is the author of Power Programming with Mathematica.

This problem results from the fact that the initial use of the name `Polyhedron` (and `Dodecahedron`) created a symbol named `Polyhedron` that is distinct from the symbol of the same name defined in the package.

```
In[4]:= ?*`Polyhedron
Polyhedron
Graphics`Polyhedra`Polyhedron
```

Experienced users know that to remedy this situation, all that is necessary is to remove the offending symbols using the `Remove` function:

```
In[5]:= Remove[Polyhedron, Dodecahedron]
In[6]:= Show[Polyhedron[Dodecahedron]]
```



Shadowing can be a major annoyance if many symbols are involved. Furthermore, judging from the frequency with which questions about this topic appear in the *Mathematica* Internet discussion group (`comp.soft-sys.math.mathematica`), shadowing is a common and frustrating problem for inexperienced *Mathematica* users.

The shadowing problem can not be eliminated, but it can be ameliorated. The goal of this article is to develop a tool for preventing shadowing in the most common cases. Along the way, we will survey other techniques and tools for dealing with the problem. First, we present a detailed introduction to contexts and how packages use them.

Contexts

A properly designed package should not modify the definitions of any symbols that are in existence at the time the package is loaded – with the possible exception of system-defined symbols that are enhanced by the package. Obviously, the author of a package can't possibly anticipate what symbol names will be in use already. Therefore, multiple distinct symbols with the same name need to be able to co-exist. Contexts are the mechanism used to manage such symbol names in a *Mathematica* session.

Every symbol in a *Mathematica* session belongs to some context. Built-in symbols, such as `Plus`, are in the `System`` context.

```
In[7]:= Context[Plus]
Out[7]= System`
```

(The grave accent, or backquote, following a context name is called a context mark.)

Upon the first use of a new symbol name, a symbol with that name is created in whatever context happens to be the current context. This input creates a symbol `x` in the `Global`` context:

```
In[8]:= x = 5;
In[9]:= Context[x]
Out[9]= Global`
```

The current context is kept in the system variable `$Context`:

```
In[10]:= $Context
Out[10]= Global`
```

Most of the time, the current context is `Global``, so most symbols defined by the user in the course of a *Mathematica* session fall into that context.

Note that it is not necessary to assign a value to a symbol in order to create it; the mere utterance of a symbol name (metaphorically speaking, of course) causes its creation.

```
In[11]:= Context[y]
Out[11]= Global`
```

Symbol names are unique within any one context, but the same symbol name appearing in different contexts refers to different symbols. How can these different symbols be specified unambiguously? Every symbol has a long name and a short name; the long name is of the form `contextname`short-`

`name`. Normally, you need to use only the short name of a symbol to refer to that symbol, and symbols print as short names. The `?` operator shows the long name of a symbol.

```
In[12]:= ?x
Global`x
x = 5
```

This input creates a context called `temp`, as well as a symbol `x` within that context:

```
In[13]:= temp`x = 6;
```

Note that this symbol is distinct from the symbol `Global`x` that was created above. You can refer to a symbol in another context by using its long name.

```
In[14]:= {x, temp`x}
Out[14]= {5, 6}
```

Or, you can change the current context to the other context,

```
In[15]:= Begin["temp`"]
Out[15]= temp`
```

and use the symbol's short name. The short name `x` now refers to `temp`x`, whereas to specify `Global`x`, you must use that symbol's full name.

```
In[16]:= {x, Global`x}
Out[16]= {6, 5}
```

This leaves the `temp`` context and returns to the previous (`Global``) context:

```
In[17]:= End[]
Out[17]= temp`
```

The short name `x` once again refers to the global symbol `x`.

```
In[18]:= ?x
Global`x
x = 5
```

Note that `Begin` returns the name of the context that is entered; the same name is returned by `End` when the context is left. You should always use `Begin` and `End` rather than manipulating the value of `$Context` directly.

There is one important caveat regarding the use of `Begin`: always place it on its own input line. The reason is that no part of an input line is evaluated until the entire line has been parsed; however, the parser uses the value of `$Context` at the time the input is read to decide how to resolve the symbol names. For example:

```
In[19]= Begin["temp`"]; Print[{x, Global`x, temp`x}]; End[];
      {5, 5, 6}
```

Since the entire input line is evaluated at one time, the current context does not become `temp`` until after the parsing is completed. Therefore, the parser resolves the name `x` to the symbol `Global`x`, because `Global`` is the current context. Placing the `Begin["temp`"]` command on a separate input line gives the behavior one might expect.

```
In[20]= Begin["temp`"];
      Print[{x, Global`x, temp`x}]
      End[];
      {6, 5, 6}
```

Nested Contexts

Contexts can be nested. For example, this input creates a symbol `x` in a subcontext `foo` of the context `temp``.

```
In[23]= temp`foo`x
Out[23]= temp`foo`x
```

Here we evaluate `x` in three different contexts: `Global`` (the current context), `temp``, and `temp`foo``:

```
In[24]= x
Out[24]= 5

In[25]= Begin["temp`"]
Out[25]= temp`

In[26]= x
Out[26]= 6

In[27]= Begin["`foo`"]
Out[27]= temp`foo`

In[28]= x
Out[28]= x
```

The `End` command undoes the action of the most recent `Begin`, thus it exits `temp`foo`` and returns to the context `temp``.

```
In[29]= End[]
Out[29]= temp`foo`

In[30]= $Context
Out[30]= temp`
```

Note that the specification of the subcontext name ``foo`` begins with a context mark. If the leading context mark is omitted, that `Begin` command creates and enters a new top-level context called `foo``.

```
In[31]= Begin["foo`"]
Out[31]= foo`
```

```
In[32]= z
Out[32]= z

In[33]= ?z
      foo`z
```

The current context is now `foo``, not `temp`foo``. The function `Contexts[]`, which returns a list of all existing contexts, shows that `foo`` and `temp`foo`` are different contexts.

```
In[34]= Contexts[]
Out[34]= {DSolve`, FE`, foo`, Format`, Global`,
      Graphics`Animation`, Graphics`Polyhedra`,
      Graphics`Polyhedra`Private`, Graphics`Private`,
      Integrate`, Limit`, NullSpace`, Obsolete`, Series`,
      Solve`, System`, System`ComplexExpand`,
      System`Private`, temp`, temp`foo`}
```

(The other contexts in this menagerie were created as part of the kernel's initialization procedure.)

An `End` command returns to the context `temp``, since that was the current context before `Begin[foo`]` was executed. (`End` returns to the previous context, not to the "parent" context.)

```
In[35]= End[]
Out[35]= foo`

In[36]= $Context
Out[36]= temp`
```

A second `End` command returns to the `Global`` context.

```
In[37]= End[]
Out[37]= temp`

In[38]= $Context
Out[38]= Global`
```

There is a strong analogy between contexts and directories in a hierarchical file system. The context mark corresponds to the directory pathname separator ("`/`" in UNIX, "`\`" in DOS, "`:`" in Macintosh OS), symbol names correspond to file names, and the current context corresponds to the working directory. However, unlike UNIX and DOS (but like the Macintosh OS), a context mark at the beginning of a context name (as in ``foo``) does not correspond to a "root" context. When the name of a context stands alone, it is an absolute name, but when it is preceded by a context mark it is relative to the current context.

`Begin` and `End` do not correspond precisely to the UNIX `cd` or the DOS `chdir`, since they keep track of the order in which contexts are entered and left. Instead, `Begin` and `End` are analogous to the UNIX shell commands `pushd` and `popd`.

The Context Search Path

To allow you to use symbols from multiple contexts conveniently, *Mathematica* maintains a *context search path*, which is simply a list of context names in the order in which they will be searched when symbols are encountered. The current context is always searched first. Continuing the analogy with file system directories, the context search path is analogous to the `PATH` environment variable of UNIX and DOS systems, while the current context is analogous to the working directory. The context search path is kept in the system variable `$ContextPath`.

```
In[39]:= $ContextPath
Out[39]= {Graphics`Polyhedra`, Global`, System`}
```

This shows that contexts will be searched in the following order: first the current context, then the `Graphics`Polyhedra`` context, then the `Global`` context, and finally the `System`` context. (Do not confuse the context search path `$ContextPath` with the directory search path `$Path`, which contains the list of directories that the kernel searches when attempting to find a file.)

For example, when the symbol `I` is used, and the current context is `Global``, the system does not find symbols called `Graphics`Polyhedra`I` or `Global`I` so it uses the symbol `System`I`.

```
In[40]:= ?I
I represents the imaginary unit Sqrt[-1].
```

If you were to define a symbol in the `Global`` context named `I`, then you would no longer be able to access `System`I` by its short name.

```
In[41]:= Global`I = 2;
I::shdw: Warning: Symbol I appears in multiple contexts
{Global`, System`}; definitions in context Global`
may shadow or be shadowed by other definitions.
```

The warning message says that this definition has shadowed another symbol having the same name, that is, it has made that symbol inaccessible by its short name. (We'll discuss shadowing in detail later in this article.) There are now two distinct symbols named `I`: `Global`I` and `System`I`.

```
In[42]:= ?*`I
I          System`I
```

Because the current context is `Global``, an unqualified `I` now refers to `Global`I`, which has the value 2.

```
In[43]:= I^2
Out[43]= 4
```

You can always refer to the imaginary unit `I` by its long name.

```
In[44]:= System`I ^2
Out[44]= -1
```

Alternatively, you can set the current context to `System``; then references to `I` are resolved to `System`I`.

```
In[45]:= Begin["System`"]
Out[45]= System`
In[46]:= I^2
Out[46]= -1
```

Note that `$ContextPath` is not changed.

```
In[47]:= $ContextPath
Out[47]= {Graphics`Polyhedra`, Global`, System`}
```

Nevertheless, `I` resolves to `System`I` because `System`` is the current context.

```
In[48]:= $Context
Out[48]= System`
In[49]:= End[]
Out[49]= System`
```

Only the current context and the contexts in `$ContextPath` are searched automatically; to refer to a symbol in any other context, you have to use its long name.

At this point, we should get rid of the bogus definition of `I`. The function `Clear` isn't sufficient for this purpose, as it removes only the values of a symbol, and not the symbol itself.

```
In[50]:= Clear[I]
In[51]:= Context[I]
Out[51]= Global`
```

The correct function for this job is `Remove`:

```
In[52]:= Remove[I]
In[53]:= Context[I]
Out[53]= System`
```

BeginPackage and EndPackage

There are two other context-related functions, `BeginPackage` and `EndPackage`, that make context management for a package much easier. The first thing that every package must do is call `BeginPackage`, which creates a new context and changes the context search path to consist of only the newly-defined context and the `System`` context:

```
In[54]:= BeginPackage["example`"]
Out[54]= example`
In[55]:= $ContextPath
Out[55]= {example`, System`}
```

```
In[19]= Begin["temp`"]; Print[{x, Global`x, temp`x}]; End[];
      {5, 5, 6}
```

Since the entire input line is evaluated at one time, the current context does not become `temp`` until after the parsing is completed. Therefore, the parser resolves the name `x` to the symbol `Global`x`, because `Global`` is the current context. Placing the `Begin["temp`"]` command on a separate input line gives the behavior one might expect.

```
In[20]= Begin["temp`"];
      Print[{x, Global`x, temp`x}]
      End[];
      {6, 5, 6}
```

Nested Contexts

Contexts can be nested. For example, this input creates a symbol `x` in a subcontext `foo` of the context `temp``.

```
In[23]= temp`foo`x
Out[23]= temp`foo`x
```

Here we evaluate `x` in three different contexts: `Global`` (the current context), `temp``, and `temp`foo``:

```
In[24]= x
Out[24]= 5

In[25]= Begin["temp`"]
Out[25]= temp`

In[26]= x
Out[26]= 6

In[27]= Begin["`foo`"]
Out[27]= temp`foo`

In[28]= x
Out[28]= x
```

The `End` command undoes the action of the most recent `Begin`, thus it exits `temp`foo`` and returns to the context `temp``.

```
In[29]= End[]
Out[29]= temp`foo`

In[30]= $Context
Out[30]= temp`
```

Note that the specification of the subcontext name ``foo`` begins with a context mark. If the leading context mark is omitted, that `Begin` command creates and enters a new top-level context called `foo``.

```
In[31]= Begin["foo`"]
Out[31]= foo`
```

```
In[32]= z
Out[32]= z

In[33]= ?z
      foo`z
```

The current context is now `foo``, not `temp`foo``. The function `Contexts[]`, which returns a list of all existing contexts, shows that `foo`` and `temp`foo`` are different contexts.

```
In[34]= Contexts[]
Out[34]= {DSolve`, FE`, foo`, Format`, Global`,
      Graphics`Animation`, Graphics`Polyhedra`,
      Graphics`Polyhedra`Private`, Graphics`Private`,
      Integrate`, Limit`, NullSpace`, Obsolete`, Series`,
      Solve`, System`, System`ComplexExpand`,
      System`Private`, temp`, temp`foo`}
```

(The other contexts in this menagerie were created as part of the kernel's initialization procedure.)

An `End` command returns to the context `temp``, since that was the current context before `Begin[foo`]` was executed. (`End` returns to the previous context, not to the "parent" context.)

```
In[35]= End[]
Out[35]= foo`

In[36]= $Context
Out[36]= temp`
```

A second `End` command returns to the `Global`` context.

```
In[37]= End[]
Out[37]= temp`

In[38]= $Context
Out[38]= Global`
```

There is a strong analogy between contexts and directories in a hierarchical file system. The context mark corresponds to the directory pathname separator ("`/`" in UNIX, "`\`" in DOS, "`:`" in Macintosh OS), symbol names correspond to file names, and the current context corresponds to the working directory. However, unlike UNIX and DOS (but like the Macintosh OS), a context mark at the beginning of a context name (as in ``foo``) does not correspond to a "root" context. When the name of a context stands alone, it is an absolute name, but when it is preceded by a context mark it is relative to the current context.

`Begin` and `End` do not correspond precisely to the UNIX `cd` or the DOS `chdir`, since they keep track of the order in which contexts are entered and left. Instead, `Begin` and `End` are analogous to the UNIX shell commands `pushd` and `popd`.

`BeginPackage` temporarily removes all other contexts except `System`` from the search path, which guarantees that symbols defined by the package will be distinct from symbols defined by other packages or by the user. The `System`` context is left on the context search path so that the package can use built-in functions. (If a package needs the services of other packages, a call to

```
BeginPackage[packagecontext, othercontext1, othercontext2,...]
```

can be used to load the other packages, if necessary, and prepend their contexts to the context search path.)

Here are some symbols created in the package context:

```
In[56]:= f = "a package symbol";
        g = "another package symbol";
```

It's considered good programming practice to place all definitions that are not meant to be seen by the user into a private subcontext of the package context. By convention, this context is called `Private``.

```
In[58]:= Begin["Private`"]
Out[58]= example`Private`
In[59]:= a = "a very shy symbol";
In[60]:= End[];
In[61]:= EndPackage[];
```

`EndPackage[]` is used at the end of the package to return the value of `$Context` to its previous value and to set the context search path to its previous value with the name of the new context prepended to it.

```
In[62]:= $Context
Out[62]= Global`
In[63]:= $ContextPath
Out[63]= {example`, Graphics`Polyhedra`, Global`, System`}
```

Prepending the new context name to the context path means that symbols defined in the package context can be referred to by their short names:

```
In[64]:= ?f
        example`f
        f = "a package symbol"
```

Since the private subcontext is not on the context search path, symbols in that context are "hidden" from the user.

```
In[65]:= ?a
Information::notfound: Symbol a not found.
```

Note that since the private subcontext is nested within the package context, it is distinct from the private subcontext of any other package that follows the standard package structure outlined here.

Digression: CleanSlate

Since we will be doing a lot of experimentation with context creation, it will be convenient to have a mechanism for removing entire contexts from a kernel session without having to quit and restart the kernel. The `CleanSlate` package [Gayley 1993] (*MathSource* item 0204-310) provides this functionality. The author recommends it highly. If you intend to work through the examples in the remainder of this article, you are advised to load this package now. Otherwise, you will find it necessary to quit and restart the kernel several times in order to wipe out certain contexts.

`CleanSlate` can remove only the symbols and contexts created after the `CleanSlate.m` package was loaded. Therefore, it is necessary to begin a new kernel session before loading the `CleanSlate.m` package. (You may wish to insert a command in your `init.m` file to load this package automatically when the kernel starts up; for the location of the `init.m` file, check the documentation for your version of *Mathematica*.) The following command is the first in the new kernel session.

```
In[1]:= Needs["CleanSlate`"]
In[2]:= ?CleanSlate
CleanSlate[] purges all symbols and their values in all contexts that have been added to the context search path ($ContextPath), since the CleanSlate package was read in. This includes user-defined symbols (in the Global` context) as well as any packages that may have been read in. It also removes most, but possibly not all, of the additional rules for System symbols that these packages may have defined. It also clears the In[] and Out[] values, and resets the $Line number, so new input begins as In[1].
CleanSlate["Context1`", "Context2`"] purges only the listed contexts.
```

The following option is used to disable some extra status information that normally would be printed by each call to the `CleanSlate` function.

```
In[3]:= SetOptions[CleanSlate, Verbose -> False];
```

Ways to Prevent Shadowing

Shadowing occurs when more than one context on the context search path defines a symbol of a given name. The most common cause of shadowing is using a symbol from a package before loading that package, as illustrated at the beginning of this article.

Several techniques and tools exist for preventing, or at least minimizing, the shadowing problem. Some of these are aimed at package developers, while others are meant for users. In the next few sections, we'll survey the existing tools and then develop one of our own.

We'll start with two tricks that package designers can use to spare users of their packages the frustration of shadowing. Unfortunately, each of these techniques has drawbacks as well.

Roman Maeder suggests including the `Global`` context in the `BeginPackage` command [Maeder 1991, 47–49]:

```
BeginPackage["MyPackage`", "Global`", ...]
```

This adds the `Global`` context to the context search path at the time the package is loaded. Any symbol created by the package that does not already exist in the `Global`` context will be created in the `MyPackage`` context. If such a symbol already exists in the `Global`` context (because a user tried to use it too soon), then that symbol – in the `Global`` context – will be redefined by the package. Hence, there will be only one symbol with the given name, and no shadowing will occur.

Of course, this technique involves a risk, since the user may have deliberately associated a definition with one of these conflicting symbol names, and the package might wipe out that definition. To get around this problem, Nancy Blachman recommends the following approach [Blachman 1992, 259–263]. Start the package with these lines:

```
BeginPackage["MyPackage`"]
EndPackage[]
```

The purpose is to prepend the package context to the context search path. Now continue with the rest of the package; all symbols will be created in the context in which the package was loaded, which probably is `Global``. (Be sure to declare the package's private subcontext as `MyPackage`Private`` rather than just `Private``, to ensure that the package's private symbols do not end up in a context called `Global`Private``.)

It may appear that this method has the same potential for problems as Maeder's. The difference is that if a sophisticated user wishes to create the package's symbols in a different context – which cannot be done using Maeder's method – she can always do this:

```
BeginPackage["NewContext`"]
Needs["MyPackage`"]
EndPackage[]
```

This prepends `NewContext`` to the context search path and causes all nonprivate symbols defined by `MyPackage` to be created in that context. Thus, the user's definitions of any conflicting symbols will be preserved (although they will still shadow the definitions from the package).

Pre-Declaring Package Symbols

Every standard package directory (such as `Graphics`) contains a file called `Master.m`, which contains declarations for the contexts of each of the packages in the directory (such as `Graphics`Polyhedra``). These declarations have the following general form:

```
In[4]:= DeclarePackage["Graphics`Polyhedra`",
                    {"Polyhedron", "Dodecahedron"}]
Out[4]= Graphics`Polyhedra`
```

This statement doesn't actually load the `Graphics`Polyhedra`` package; it simply tells the kernel that the symbols `Polyhedron` and `Dodecahedron` can be found there.

```
In[5]:= ?Polyhedron
Graphics`Polyhedra`Polyhedron
Attributes[Polyhedron] = {Stub}
```

The `Stub` attribute signifies that the package that defines this symbol has not yet been loaded. Upon the first use of any symbol having this attribute, the kernel will automatically load the package that defines that symbol.

```
In[6]:= Polyhedron
Out[6]= Polyhedron
```

```
In[7]:= ?Polyhedron
Polyhedron[name] gives a Graphics3D object representing
the specified solid centered at the origin and with
unit distance to the midpoints of the edges.
Polyhedron[name, center, size] uses the given center
and size. The possible names are in the list Polyhedra.
```

If there are certain standard packages that you use on occasion but which would occupy too much memory if they were all loaded simultaneously, you can avoid any possibility of shadowing problems with those packages by loading their master packages at the beginning of each *Mathematica* session. You can automate this procedure by putting the statements to load the master packages into your `init.m` file.

If you are a *Mathematica* developer and you are writing a particularly large package that uses a lot of memory, or a set of interrelated packages, you may want to create a master package as a convenience for your users. A master package should be structured as follows, with a `DeclarePackage` statement for each package:

```
BeginPackage["MyPackage`Master`"];
EndPackage[];
DeclarePackage["Package1", {symbol1, symbol2, ...}];
DeclarePackage["Package2", {symbol3, symbol4, ...}];
```

For more information on `Master` packages, see [Blachman 1993]. Also, take a look at the standard `Statistics`` packages, which provide a good example of this and other techniques for organizing a large set of interrelated packages.

Of course, the `Master` package can only prevent shadowing of package symbols if it has been loaded. Users who have shadowing problems because they forget to load a standard package often forget to load the `Master` package as well.

Removing Shadows

The vast majority of packages, including standard packages, do not take any steps to prevent the shadowing problem (aside from defining `Master` packages). In response to this situation, Ulrich Jentschura has written a package called `Unshadow.m` (*MathSource* item 0204-815), which defines a function that seeks out and destroys all global symbols that are shadowing other symbols. This package is especially useful for cases in which the user has shadowed so many symbols that the error message mechanism suppresses some of the `::shdw` messages!

To demonstrate, we deliberately make the same mistake as before.

```
In[8]:= CleanSlate[];
In[11]:= g = Polyhedron[Dodecahedron];
In[2]:= Show[g]
Show::gtype: Polyhedron is not a type of graphics.
Out[2]= Show[Polyhedron[Dodecahedron]]
In[3]:= Needs["Graphics`Polyhedra`"]
Polyhedron::shdw:
Warning: Symbol Polyhedron
  appears in multiple contexts
  {Graphics`Polyhedra`, Global`}; definitions in context
  Graphics`Polyhedra` may shadow or be shadowed by other
  definitions.
Dodecahedron::shdw:
Warning: Symbol Dodecahedron
  appears in multiple contexts
  {Graphics`Polyhedra`, Global`}; definitions in context
  Graphics`Polyhedra` may shadow or be shadowed by other
  definitions.
```

`Unshadow` removes the symbols in the global context that are shadowing symbols in other contexts and returns a list of the symbols that have been removed.

```
In[4]:= Needs["Unshadow`"]
Unshadow Package, Version of April, 1993
For information, type ?Unshadow
In[5]:= Unshadow[]
The following symbols appeared both in the global and in
another context.
They have been removed in the global context to prevent
shadowing.
Out[5]= {Dodecahedron, Polyhedron}
```

Indeed, the shadowing symbols are gone.

```
In[6]:= Context /@ {Polyhedron, Dodecahedron}
Out[6]= {Graphics`Polyhedra`, Graphics`Polyhedra`}
```

The strategy used by `Unshadow` is to get a list of symbol names in each of the contexts on the context search path. For example:

```
In[7]:= Names["Graphics`Polyhedra`*"]
Out[7]= {Cube, Dodecahedron, Faces, Geodesate, GreatDodecahedron,
  GreatIcosahedron, GreatStellatedDodecahedron,
  Hexahedron, Icosahedron, Octahedron, OpenTruncate,
  Polyhedra, Polyhedron, SmallStellatedDodecahedron,
  Stellate, Tetrahedron, Truncate, Vertices}
```

After a bit of string massaging (see the package source for details), a list of symbols that are common to the global context and any other context on the search path is constructed; this list is then passed to `Remove`.

Removing Symbols Without Values

`Unshadow` does not make any attempt to distinguish between symbols that the user might or might not need. One obvious way to do this would be simply to return the list of shadowing symbols without removing them; the user could then verify that none of the symbols are important and remove them with the command `Remove @@ %`. Alternatively, `Unshadow` could prompt the user for confirmation (using the `Input` function) before removing each symbol. Better yet, it could give the user the best of both worlds by providing an option that controls its behavior in this regard.

On the other hand, there is something to be said for making the process as automatic as possible. A more sophisticated approach that subscribes to this philosophy would be to determine whether or not the shadowing symbol has any values; if it does not, then it probably is not important and can be removed automatically. This is the approach that we shall take next. Our efforts will culminate in a new package, `AntiShadow.m`, that removes shadows during the package-loading process, taking care never to remove a symbol that has any values.

A symbol can have many different kinds of values. For example, this assignment creates a type of value called an own-value:

```
In[8]:= test := 1/0
```

We can list the own-values of a symbol using the `OwnValues` function:

```
In[9]:= OwnValues[test]
Out[9]= {Literal[test] =>  $\frac{1}{0}$ }
```

There are five other kinds of rules that can be attached to a symbol: down-values, up-values, sub-values, format-values, and N-values, which are listed by the corresponding functions `DownValues`, `UpValues`, `SubValues`, `FormatValues`, and `NValues`. In addition to checking for the six types of rules, we will check for the existence of attributes, options, and messages.

If a symbol has the `ReadProtected` attribute, `OwnValues` and friends will cause an error message to be printed. Therefore, we must check for attributes before checking any of the other types of values. Furthermore, `Options` does not hold its argument; therefore, we can check for options only after determining that the symbol has no rules. Here is a function that does all of this:

```
In[10]:= SetAttributes[hasValues, HoldFirst];
In[11]:= hasValues[s_Symbol] :=
    Attributes[s] != {} ||
    OwnValues[s] != {} ||
    UpValues[s] != {} ||
    DownValues[s] != {} ||
    SubValues[s] != {} ||
    NValues[s] != {} ||
    FormatValues[s] != {} ||
    Options[s] != {} ||
    Messages[s] != {}
```

The `hasValues` function is given the `HoldFirst` attribute so that its argument will not evaluate. Here is an example:

```
In[12]:= hasValues[test]
Out[12]= True
```

The `Unshadow` package could easily be enhanced to `Select` all symbols for which `hasValues` returns `False` out of a list of candidates for removal, before passing the list to the `Remove` function. We leave experimentation with the `Unshadow` package to the interested reader. Rather than modify that package, we will take a different approach in developing our package, `AntiShadow`.

Like the `Unshadow` package, `AntiShadow` works with symbol names in string form, for reasons that will become clear later. Unfortunately, `OwnValues` and the like do not accept strings as arguments.

```
In[13]:= OwnValues["test"]
    OwnValues::sym:
    Argument test at position 1 is expected to be a symbol.
Out[13]= OwnValues[test]
```

We could convert the string to an expression using `ToExpression`, but that would result in an evaluation of the symbol.

```
In[14]:= ToExpression["test"]
    Power::infy: Infinite expression  $\frac{1}{0}$  encountered.
Out[14]= ComplexInfinity
```

The solution to this problem is to use `ToHeldExpression` to convert the string to an expression and wrap `Hold` around it at the same time.

```
In[15]:= ToHeldExpression["test"]
Out[15]= Hold[test]
```

The own-values of the symbol can then be found by using `Apply` to replace the head `Hold` with `OwnValues`.

```
In[16]:= OwnValues @@ %
Out[16]= {Literal[test] :>  $\frac{1}{0}$ }
```

The `hasValues` function can be applied to symbol names using the same technique.

```
In[17]:= hasValues @@ ToHeldExpression["test"]
Out[17]= True
```

There still remains another, more subtle problem. Even if a symbol has no values, removing it can have adverse consequences on other symbols. For example, the symbol `g` defined above is still a function of the removed symbols `Polyhedron` and `Dodecahedron`.

```
In[18]:= g
Out[18]= Removed[Polyhedron][Removed[Dodecahedron]]
```

The expressions of the form `Removed[symbol]` aren't normal expressions; in fact, they are a pathological output format for symbols that have been marked internally for removal.

```
In[19]:= FullForm[ g[[1]] ]
Out[19]/FullForm= Removed["Dodecahedron"]
```

```
In[20]:= Head[%]
Out[20]= Symbol
```

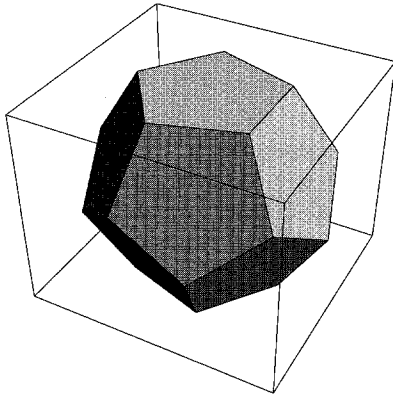
This means that the pattern `Removed[x_]` will not match either of the removed symbols:

```
In[21]:= g /. Removed[x_] :> ToExpression[x]
Out[21]= Removed[Polyhedron][Removed[Dodecahedron]]
```

If we extract the removed symbols from `g` using the pattern `_Symbol`, we can use them in replacement rules.

```
In[22]:= Cases[g, _Symbol, Infinity, Heads -> True]
Out[22]= {Removed[Polyhedron], Removed[Dodecahedron]}
In[23]:= Thread[Rule[%, {Polyhedron, Dodecahedron}]]
Out[23]= {Removed[Polyhedron] -> Polyhedron,
    Removed[Dodecahedron] -> Dodecahedron}
```

```
In[24]= Show[g /. %]
```



It is even possible to prevent the removal of symbols upon which other symbols depend by using the `DependencyAnalysis` package developed in the previous installment of this column [Wagner 1996]. That package defines the function `DependsOn`, which returns the list of all symbols upon which a given symbol depends. `DependsOn` could be applied to every symbol in the `Global`` context, and the resulting collection of symbols could be eliminated from the list of candidates for removal. The drawback of this approach is that, in a *Mathematica* session with a lot of symbols, it could slow down the shadow removal process quite dramatically. We will not pursue this idea any further here; the interested reader may wish to try it out.

We will not clean the `Global`` context at this point because we still need the definition of `hasValues`.

```
In[25]= CleanSlate["Graphics`Polyhedra`"];
```

Avoiding Shadowing Dynamically

To make the process of eliminating shadowing symbols as automatic as possible, we would like to remove the offending symbols without requiring any action on the part of the user. The observation that makes this possible is that whenever a symbol is defined that might shadow some other symbol, the message `symbol::shdw` is generated:

```
In[1]= General::shdw
Out[1]= Warning: Symbol `1` appears in multiple contexts `2`; \
        definitions in context `3` may shadow or be shadowed by \
        other definitions.
```

Therefore, our next course of action will be to override calls to `Message[sym_::shdw, ___]`.

Notice that `Message` already has some down-values:

```
In[2]= Length[DownValues[Message]]
Out[2]= 5
```

These down-values are created during the initialization of the kernel. In the case of `Message`, the down-values are expendable; they simply define usage messages for a handful of obsolete functions. However, many built-in functions are

implemented partially or exclusively by down-values. Needless to say, it is extremely ill-advised to wipe out any pre-defined down-values for a built-in function. Therefore, we'll save the down-values for `Message` now and restore them after we've finished creating our own rules for `Message`.

```
In[3]= mdv = DownValues[Message];
        Unprotect[Message];
        DownValues[Message] = {};
```

From the text of the `General::shdw` message, we deduce that `Message` will be called with four arguments: the name of the message plus the three string substitution arguments. Let's write a rule for `Message` that will allow us to see the full form of each of these arguments:

```
In[6]= Message[arg0_, arg1_, arg2_, arg3_] :=
        (Print[FullForm[#]]& /@ {arg0, arg1, arg2, arg3};
        Print[""])
```

Next, we recreate the problematic global symbols and load the package.

```
In[7]= Show[Polyhedron[Dodecahedron]]
        Show::gtype: Polyhedron is not a type of graphics.
Out[7]= Show[Polyhedron[Dodecahedron]]

In[8]= Needs["Graphics`Polyhedra`"]
        MessageName[Polyhedron, "shdw"]
        HoldForm["Polyhedron"]
        HoldForm[List["Graphics`Polyhedra`", "Global`"]]
        HoldForm["Graphics`Polyhedra`"]

        MessageName[Dodecahedron, "shdw"]
        HoldForm["Dodecahedron"]
        HoldForm[List["Graphics`Polyhedra`", "Global`"]]
        HoldForm["Graphics`Polyhedra`"]
```

The first argument is a `MessageName` object. Since `MessageName` has the `HoldFirst` attribute, the symbol inside of it does not evaluate. The rest of the arguments, which are substituted for ``1``, ``2``, and ``3``, are strings wrapped in `HoldForm`. Let's refine our definition for `Message` to use destructuring to extract the symbol, symbol name, and context names. We will, of course, keep the symbol itself wrapped in `HoldForm` to prevent its evaluation.

```
In[9]= Clear[Message]

In[10]= Message[sym_::shdw, HoldForm[symName_],
              HoldForm[cxNames_], HoldForm[newcxName_] :=
        ( Print["sym = ", HoldForm[sym]];
          Print["symName = ", InputForm[symName]];
          Print["cxNames = ", InputForm[cxNames]];
          Print["newcxName = ", InputForm[newcxName], "\n"];
        )

In[11]= CleanSlate["Graphics`Polyhedra`"];
```

```

In[1]:= Needs["Graphics`Polyhedra`"]
sym = Polyhedron
symName = "Polyhedron"
cxNames = {"Graphics`Polyhedra`", "Global`"}
newcxName = "Graphics`Polyhedra`"

sym = Dodecahedron
symName = "Dodecahedron"
cxNames = {"Graphics`Polyhedra`", "Global`"}
newcxName = "Graphics`Polyhedra`"

```

When passing `sym` to `hasvalues` to check if the shadowing symbol has any definitions in the `Global`` context, we must remember that at the time `hasvalues` is called, the current context will be the new package's context, not `Global``. Therefore, we must construct the long name `"Global`"<>symName` explicitly and use `hasvalues@@ToHeldExpression[#]&` on it.

We also can use the `symName` argument to generate our own warning messages whenever we remove a symbol to avoid a shadow. This is the text of the message that will be generated:

```

In[2]:= General::noshdw =
"Warning: the symbol `1` has been removed from \
the global context to prevent shadowing.";

```

Here, then, is the definition of `Message` that will do the trick.

```

In[3]:= Clear[Message]
In[4]:= Message[sym::shdw, HoldForm[symName_],
HoldForm[cxNames_], HoldForm[newcxName_]] /;
MemberQ[cxNames, "Global`"] && !hasvalues @@
ToHeldExpression["Global`" <> symName] :=
With[{globalsym = "Global`" <> symName},
Message[sym::noshdw, globalsym];
Remove[globalsym] ]

```

If `Global`` is one of the contexts containing the symbol named `symName` (`MemberQ[cxNames, "Global`"]`), and `Global`symName` has no values (`!hasvalues @@ ToHeldExpression["Global`"<>symName]`), the warning message is printed and `Global`symName` is removed.

Here is a demonstration. We create the symbols `Polyhedron` and `Dodecahedron` in the global context, and we give the latter a value.

```

In[5]:= Polyhedron[Dodecahedron];
In[6]:= Dodecahedron = "I have a value!";
In[7]:= CleanSlate["Graphics`Polyhedra`"];

```

```

In[1]:= Needs["Graphics`Polyhedra`"]
Polyhedron::noshdw:
Warning: the symbol Global`Polyhedron
has been removed from the global context to prevent
shadowing.
Dodecahedron::shdw:
Warning: Symbol Dodecahedron
appears in multiple contexts
{Graphics`Polyhedra`, Global`}; definitions in context
Graphics`Polyhedra` may shadow or be shadowed by other
definitions.

```

The symbol `Global`Polyhedron` has been removed:

```

In[2]:= ?*Polyhedron
Polyhedron[name] gives a Graphics3D object representing
the specified solid centered at the origin and with
unit distance to the midpoints of the edges.
Polyhedron[name, center, size] uses the given center
and size. The possible names are in the list Polyhedra.

```

On the other hand, `Global`Dodecahedron` has not been removed. Because this symbol has an own-value, the package leaves it up to the user to decide whether or not to remove it.

```

In[3]:= ?*Dodecahedron
Dodecahedron
Graphics`Polyhedra`Dodecahedron

```

Now that we're satisfied with the new behavior of `Message`, we should restore its pre-defined down-values and reprotect it. Note that we place the down-value we have created before any of the built-in down-values. In this case, it makes no difference, but for symbols with nontrivial down-values, this order ensures that the user-defined down-value is tried before the built-in ones.

```

In[4]:= DownValues[Message] = Join[DownValues[Message], mdv];
Protect[Message];

```

Note that the new rule for `Message` prevents the user from creating a symbol in the `Global`` context that would shadow a symbol in some other context.

```

In[6]:= Global`GreatStellatedDodecahedron
GreatStellatedDodecahedron::noshdw:
Warning: the symbol Global`GreatStellatedDodecahedron
has been removed from the global context to prevent
shadowing.
Out[6]= Removed[GreatStellatedDodecahedron]

```

The return value is a reference to the symbol just created and then removed. This behavior is either a bug or a feature, depending on your point of view. If you think it's a bug, you can easily fix it: simply add another condition to the new rule for `Message` that checks to see if `newcxName` is not equal to `"Global`"`.

The functions developed in the last two sections are collected in the `AntiShadow` package, included in the electronic supplement. Since the new rule for `Message` can perform its function only if it already exists at the time a package is loaded, `AntiShadow` will be most effective if it is loaded automatically at the beginning of every *Mathematica* session.

Conclusion

The use of contexts to segregate symbol names in *Mathematica* can sometimes cause the problem of shadowing. In the most common case, shadowing results from attempting to use a package-defined symbol before loading the package. We have reviewed several existing techniques for dealing with this problem. Loading master packages at the beginning of a session is effective for the standard packages, but usually is not an option for other packages. The `Unshadow` package can remove shadows after-the-fact, but it is not very discriminating about which symbols it removes and it requires an extra step on the part of the user. The `AntiShadow` package developed in this article has the advantages that it works with any package, it completely automates the process of shadow removal, and it never removes a symbol for which the user has spent time and effort defining rules. It thus represents an advance over existing techniques.

References

- Blachman, Nancy. 1992. *Mathematica: A Practical Approach*. Prentice-Hall.
- Blachman, Nancy. 1993. Using the standard packages. *The Mathematica Journal* 3(2): 31–35.
- Gayley, Todd. 1993. CleanSlate: A package for clearing symbols and freeing memory. *The Mathematica Journal* 3(3): 46–51.
- Maeder, Roman. 1991. *Programming in Mathematica*. 2nd ed. Addison-Wesley.
- Wagner, David B. 1996. Dependency analysis. *The Mathematica Journal* 6(1): 54–65.

David B. Wagner
Principia Consulting
3841 Orion Court, Boulder, CO 80304-1024
dbwagner@princon.com

 The electronic supplement contains the package `AntiShadow.m`.