Introduction

The standard import offered by D has problems. We know this quite well, since it seems every problem people run into with modules are related in some way to namespace collisions. It's been going on for years now, reflected in the newsgroup posts from as long as the author has used the language.

For those who don't know what the issues are, there are some examples of issues noted early on in this most recent saga. Or, do a search on the newsgroup for import, or name-collision, or some such.

The existing import mechanism gathers all importee symbols into the importer namespace, and it looks like this:

```
import lib.text.locale;
```

The text lib.text.locale identifies a module within a package.

Walter (the language designer & developer) has finally acknowledged there is a latent issue with respect to namespace collisions, and had initially suggested extending the import syntax to permit **explicit** import to occur. It would look like this:

```
import lib.text.locale.Date;
```

Where, in this example, Date is perhaps a class or some other top-level symbol. This would be just fine, although may perhaps become a bit verbose when used often.

One alternative proposed at the time was to effectively combine an alias with the import, in a fashion as follows:

```
import lib.text.locale as locale;
```

The above actually imports all symbols from the module, just like the standard import, but does so in such a manner whereby each imported symbol must be addressed using the provided prefix. The result is akin to the imported module being wrapped in its own little namespace. Thus we'd have something like this:

```
auto time = new locale.Date;
```

Contrast this with the more generic module namespace, where the prefix would not be required at all. You can quickly see where and how namespace collisions can and do take place.

The issue: reloaded

Why is any of this important? I suggest everyone should read the earlier posts in the various threads, and do some analysis of why namespace collisions occur. But in the meantime, this short explanation might suffice:

Each module in D has a single namespace to which all imported symbols are effectively added (top-level imported symbols). The problems begin when you import two modules where each of them uses a common symbol. Let's use the symbol *Result* for illustrative purposes:

```
module lib.text.locale.convert;

struct Result
{
// bunch of stuff
}

struct Formatter
{
  // other stuff
}
```

```
module lib.db.model.odbc;

class Result
{
// a bunch of different stuff`
}
```

```
module main;

import lib.text.locale.convert;
import lib.db.model.odbc;

void main()
{
   // error: namespace collision
   auto result = new Result;

   // create() name may collide with something soon?
   auto db = create (“db”);
}
```

In this case, we *could* alias the problem away. However, serious problems take place when said collision occurs, let's say, a year or two later when the provider of either module quite innocently adds a symbol that collides with another from some other module provider.

When this happens, and it most certainly will, somebody now has to go and re-factor existing code that had previously been compiling and working just fine, simply because they installed some vendors' latest modules. In any software company the author has heard of, such compiler behavior is simply inexcusable. The mere fact that it *can* happen would, without question, keep D out of any place the author has ever been paid; it probably should at any and all software shops involved in anything resembling medium to large-scale software projects.

The upshot is this: the existing D import is quite sufficient for short-term or personal projects. But the single-namespace nature is susceptible to brittleness and breakdowns. These problems can easily be avoided with some simple compiler modifications.

This very problem is why Walter suggested the extended import syntax in the first place. By using it, there's no way an innocent change in one module can lead to breakage via namespace collisions. Unfortunately, we don't have the extended import, and that particular resolution has been removed from the table for some unstated reason.

In light of the potential for breakage, what should a software house do to avoid such things?

If it were the authors' project, or any of his prior colleagues, we'd simply avoid using the existing D import. The reasons are at least twofold:

- We wouldn't want the post-product collision described above to happen; not even once.
- Consistency in import style makes it reasonable to maintain, comprehend, and standardize upon.

In other words, for code with any kind of longevity in mind (such as the kind that generates recurring revenue) we simply wouldn't *use* an import syntax that is prone to breakage.

This surely has some serious implications for the widespread usage of D.

Options

So what would we do to avoid these kinds of namespace collision? What would the options be?

Well, even with the current import behavior, one could fully-qualify all imported symbols at all times. This, unfortunately, leads to rather long-winded code which is then so much harder to work with and maintain. I think we can all agree that fully-qualified-symbols (FQS) will avoid the principal collision issue, but at what cost? Let's look at a trivial example:

```
module main;

import lib.text.locale.convert;
import lib.db.model.odbc;

void main()
{
   auto result = new lib.text.locale.convert.Result;
   auto db = lib.db.model.odbc.create("db");
}
```

Yes, it is certainly possible to use FQS all over the place, but there's a certain elegance which D imparts to code (such as foreach), and it's hard for the above example to be considered elegant in its verbosity. Keep in mind that this is just a trivial example. Larger bodies of code would be notably more *opaque*, for want of a better word (note that the font used for code within this document was changed several times just to get the above example to fit on the page).

We need some means of segregating namespaces such that names cannot collide, whilst being able to select a more usable prefix than the entire FQS path.

So, how about using aliases instead? Well, given that we need to ensure no namespace collisions will occur, we'll need to alias *every* import. We need to do this because, as noted above:

- To avoid namespace collisions, we need some kind of namespace segregation
- FQS are simply too verbose in the general case

Returning to our example:

```
module main;

import lib.text.locale.convert;
alias lib.text.locale.convert convert;
import lib.db.model.odbc;
alias import lib.db.model.odbc odbc;

void main()
{
   auto result = new locale.Result;
   auto db = odbc.create("db");
}
```

This is better than FQS, and just as effective. The alias effectively sets up a shortcut for the FQS. It's just as though the imported module had been originally wrapped within a struct, or class.

We should distinguish between the two principal forms of import. The FQS and the aliased forms will be referred to as the *safe* import, whereas traditional import form is referred to as *unsafe* import.

So, we have the ability to replace long names with shorter or more appropriate versions, and the namespace collisions should not occur since we are solely responsible for these short names (not some other company or individual). Of course, these short names should be private to the module; otherwise they will spill out into the namespace of anything importing this particular module.

In fact, all imports should also be private unless there is explicit need to expose the content therein for some reason. In short, both alias and import should probably be private by default, to lessen the likelihood of subtle mistakes. This is not shown in the trivial examples, and the question of default behavior it is not considered further here.

A large number of D regulars are now begging for a subtle syntax extension, which would combine the import and alias into a single entity. A number of proposals have been put forth, but let's look at the basic one only:

```
module main;

import lib.text.locale.convert as convert;
import lib.db.model.odbc as odbc;

void main()
{
   auto result = new locale.Result;
   auto db = odbc.create ("db");
}
```

This is identical to the version using aliases, but the alias is combined with the import. It certainly appears to be a whole lot cleaner, and probably easier to maintain also. Add a private keyword in front, and both the import & the alias are now private. Likewise with package, public, etc.

The example uses a new keyword as in the syntax, though that could easily be replaced by a symbolic ':' or '=' instead, if the number of keywords should be kept to a minimum.

Let's also consider how many modules actually use the import keyword. It would appear that virtually every module in the D language will use import; typically several times.

How many of those will need to use the above style of import versus the existing style? Well, we're talking about software houses intending to avoid what are otherwise inevitable namespace collisions. For that kind of language usage, *all* imports should use the *safe* mechanism rather than the *unsafe* existing mechanism.

Basically this means every import in every module should likely be using *safe* import instead of the *unsafe* variety.

Placing these options into context, let's see how they look:

```
module main;

import lib.text.locale.convert;
import lib.db.model.odbc;

void main()
{
   auto result = new lib.db.model.odbc.Result;
   auto db = lib.db.model.odbc.create("db");
}
```

```
module main;

import lib.text.locale.convert;
alias lib.text.locale.convert convert;
import lib.db.model.odbc;
alias import lib.db.model.odbc odbc;

void main()
{
   auto result = new locale.Result;
   auto db = odbc.create("db");
}
```

```
module main;

import lib.text.locale.convert as convert;
import lib.db.model.odbc as odbc;

void main()
{
   auto result = new locale.Result;
   auto db = odbc.create("db");
}
```

It's not too hard to see which one exhibits better qualities in terms of readability, maintainability, and comprehension at a glance.

Walter's resolution

This is where the water turns murky, since the language designer and sole developer insists the functionality is handled fully and perfectly adequately via the FQS or the alias example; the combined import has been deemed unnecessary sugar, for something that is used infrequently only.

The resolution is supposed to address namespace collision issues, and applies the keyword static to distinguish it from *unsafe* import:

```
module main;

static import lib.text.locale.convert;
alias lib.text.locale.convert convert;

static import lib.db.model.odbc;
alias import lib.db.model.odbc odbc;

void main()
{
   auto result = new locale.Result;
   auto db = odbc.create("db");
}
```

The use of static in this context has been roundly criticized by newsgroup regulars as wholly either inappropriate or meaningless. Given that imports are public by default, we should also apply the typical private keyword also:

```
module main;

private static import lib.text.locale.convert;
private alias lib.text.locale.convert convert;

private static import lib.db.model.odbc;
private alias import lib.db.model.odbc odbc;

void main()
{
   auto result = new locale.Result;
   auto db = odbc.create("db");
}
```

The use of alias is required to avoid FQS, which would simply become yet more verbose and unwieldy. Again, keep in mind this is a trivial example.

The use of static here is supposed to *ensure* that one uses FQS to address entities. That's certainly a reasonable approach, but since FQS is just *so* verbose, who is going to use it for *safe* imports?

More importantly, who will be *encouraged* to use the *safe* import, when it simply offers notably more effort on the part of the developer? Please raise your hands?

Instead, we're asking (nay, *begging*) for a syntax that makes it as effortless and painless as possible, in order to **encourage** the use of *safe* imports.

The fact that it seems all the D regulars are asking for this, in one form or another, speaks volumes as to how important it is.

The D website proudly displays the motto "*D goes where the community wants to take it*" …

Well, here is one very good example of anything *but* that. If this is truly the level of sincerity, wisdom, and real-world experience behind the D language, the author suspects all serious software development houses should give D a very wide berth.