The *8th*™ Programming Environment

Best Practices
ver. 17.01

Best Practices

For most programmers, *8th* will be a bit unfamiliar. The paradigm shift can be difficult at times, but once you are comfortable with it, we believe *8th* to be a more powerful and productive language than most. In this paper we will try to help you become a more effective *8th* programmer.

8th is a language which is designed to be secure. That means: it makes it *easy* to write applications which cannot be easily subverted, and it also makes it *difficult* for hackers to get your IP or modify your application.

However, as with all programming languages, there are "best practices" which should be followed to help *8th* fulfill those goals. By "best practices" we mean the techniques and guidelines you *should* use when writing *8th* code. Adhering to them will make your experience with *8th* much more productive and enjoyable, and will make debugging your code much easier:

Coding practices

- **do iterative development** Write your program starting with pseudocode-like `words`. Iteratively break those into their components, and *test each word as you write it*. This way you will find bugs in your code as you go, and the foundation of your program will be much more stable.
- **use qualified words** Say `n:+` rather than just `+`. This will make your code more clear, and avoid unexpected behavior.
- **phrases** Write your code in "phrases" of several `words` which accomplish some task. Break your code with white-space after each phrase. Then when you examine your code, you'll see the phrases and know what the code does.
- **factor** Break your code into smaller pieces. Ideally your words should be between *three and seven phrases long*, at the most (excluding whitespace for clarity, and comments). Extract commonly used phrases into their own words, and give those words an evocative name so you remember what they do (and so you don't mis-type them).
- **comment** Use comments liberally! Put a comment before any word you create explaining what it's for, what it expects and what it returns. Put comments inside your word if there's any chance you will not remember what you did a month from now.
- **choose meaningful names** Whether you maintain your code by yourself or not, using *meaningful* names for your words will help you understand what the word is for, and acts as another layer of documentation. Any sequence of non-whitespace can be a name, and it can be in any language you understand, so there's no reason to use `x1` when `update-client-name` is what the word actually does.
- **namespaces** Use a namespace of your own for your application's words. Doing that will allow you to reuse word names which 8th uses, without overriding them. Note that by default, *8th* will already put your words in the `user` namespace.
- **stack picture** Keep the "stack picture" (including the current `word`'s "stack-effect diagram", or SED) in your mind as you code. Help keep your mental picture straight by documenting it in comments in your code.
- **verify** You know that SED you documented so carefully? Use the phrase `.s cr` to verify that the stack actually looks like you expect it to. Put that phrase at the beginning and end of the word to check the stack-picture. Verify that the phrases you factored out do, in fact, do

what you claimed they do. When your words are debugged, your application will be more likely to work. Don't wait until you've written a lot of code: verify as you go!

- **leverage data** *8th* uses a powerful dialect of JSON as its data declaration syntax. Use words such as **`caseof`** and **`when`** to implement multi-branch decisions (similar to "switch" and the like from C type languages)
- **avoid complexity** *8th* has many powerful words and data structures built-in. Leverage them rather than building your own. Avoid complicated nested conditionals or long loops. Factor your code into bits that you can grasp at a glance.
- **use the stack** If you're coming from a more traditional language, you are used to thinking in terms of variables (and local variables). Don't. Instead, try to think in terms of the stack, and leverage it. After all, you get the stack for free…
- **check results** Many words in *8th* return **`null`** to indicate failure (or lack of a resource). Do not assume a word will always return what you expect: if it is documented to return **`null`** as well, use **`null?`** to check for that!
- **use whitespace** Indent your code so logical blocks are at the same indentation level. *8th* does not enforce any whitespace convention except the requirement there be at least some between words; if you're not careful you can end up with unreadable code!

As an example of code which is unreadable:

```
: x1 n:sqr swap n:sqr n:+ ;
```

What does it do? Even though it's a short bit of code, it could be made easier to understand. Document it, give it a descriptive name, and break the lines:

```
\ return sum of squares of two items
: sum-squares   \ m n -- m2+n2
  n:sqr swap    \ n2 m
  n:sqr n:+ ;   \ n2+m2
```

Using indentation for logical blocks:

```
: ifthen  flag --
  if
    \ indented 'true' case
  else
    \ indented 'false' case
  then
  \ outdented continuation...
  something ;
```

Note that in general, we prefer two spaces of indentation, but that's just a preference. However, we do think that indenting the body of a word so it is clearly separate is a good idea, and we think that indenting the contents of an **`if… else… then`** or a **`repeat… while`** or similar is important for legibility. We also generally place the terminating **`;`** of a word on the last line of the word.

Using data structures instead of conditionals or loops:

```
: one ... ;
: two ... ;
: three ... ;
[ ' one , ' two , ' three ] var, decider
: decide            \ n -- do-something
  0 2 n:clamp       \ Ensure input parameter is in range of valid indices
  decider @
  swap caseof ;     \ use 'caseof' to do the actual branching
```

In this instance we used an array to replace a nested "if" statement. We could also have used an object, if we wished to decide based on strings instead of numbers.

Using your own namespace:

```
ns: myapp
: assert ... ;
ns: G
...
```

Since 8th already defines a word **G:assert**, if you were to define **assert** in your own code, it would overwrite the default one. By using your own namespace, **myapp** in this case, you avoid that by creating your version as **myapp:assert**. Of course you may use any namespace name you like, and by default *8th* will put your words in the **user** namespace.

Debugging practices

To paraphrase Helmuth Von Moltke, "no code survives contact with the users". There *will* be bugs in your code. Because *8th* does not come with the equivalent of **gdb** or **Eclipse** you might find it difficult to know where to begin debugging your code. Here are some tips:

- **check the stack**! Most of the problems you're likely to encounter with *8th* will be due to an *unbalanced stack*. Check that your words conform to their documented SEDs (and keep those diagrams updated!).
- **check the stack**!! No, not a typo: you really need to make sure the stack is correctly used! As mentioned in the previous section, that is as simple as inserting **.s cr** in strategic locations in your code. You can also use the **dbg:trace** facility to help, although that is global in scope.
- **check the documentation**. Another source of problems is using *8th*'s built-in words incorrectly. Check the documentation to make sure you are correctly using them.
- **check types**. Ensure the data you put on the stack is of the expected type!
- **keep your words short**. The shorter your own words, the easier it will be to understand if they do what they are supposed to do by "inspection".
- **use the REPL**. The "REPL" (Read-Eval-Print-Loop) is your friend when developing. You can invoke any word which you have written, and verify manually that it is working. *Use it* to write a test-suite which exercises your words and ensures they handle all inputs correctly.
- **attend the prompt**. The default prompt in the REPL will show you if you have left a word-definition or string, array or map definition open. If the prompt is not **ok>**, then things are probably not "OK".

Security practices

There are a number of steps you can take to help *8th* make your programs more secure:

- **use the commercial version** to create encrypted applications for deployment
- **don't use eval** on user-supplied text unless you've also used the word **only** to restrict the words known to the REPL. **Don't use eval** to evaluate external JSON! Use **json>** instead.
- **use https** instead of **http** for your APIs (if you can).
- **use the fully encrypted local database** to store data locally, and *do not* store the password

- **use the db:prepare** and **`db:bind`** words in your SQL statements, rather than creating a SQL statement using string concatenation
- **don't keep passwords** or plaintext or keys beyond their useful lifespan. Use **`s:clear`** or **`b:clear`** to wipe their data after you're done using them.
- **don't store passwords**. Ever.
- **prefer AES+GCM** (the default) for encryption. The other ciphers and modes are provided as a courtesy, but they are not as secure or robust.
- **encourage or enforce** longer passwords from your users and use **`cr:genkey`** with at least 10,000 iterations on their provided passwords
- **use `rand` rather than `rand-pcg`** if you need cryptographically random numbers
- **use the FFI with care**. If you pass a string or buffer to a 3rd-party library using the FFI, ensure it does not overwrite the bounds of your data. Use only trusted libraries!

Conclusion

We are sure that by putting our recommendations into practice in your own 8th programs, you will have a much better experience and be able to produce more stable and secure applications than ever before.