1 What are enums really?

I view an enum as a type which has a limited and enumerated list of possible values. That's it.

1.1 What should enums do?

In C, they're represented just as numbers, and are interconvertible with numbers. This is bad for a couple reasons:

- You can pass a number into a function as an enum in C even if that number may not necessarily correspond to any of the enum values! This is very bad because people using enums as parameters have to account for the possibility that someone is going to pass a plain number in as an enum that might not match any of their enumerated types.
- You can't have any extra properties associated with an enum value. For example, if you wanted to enumerate all the keywords of a programming language (words like "for" and "while" and "void", depending on the language), you can number all the keywords, but you can't store the strings represented by those keywords. You'd have to do something like making a separate array of strings and then do something like `keyword_strings[keyword_enum]` or something equally ugly.

\English definitely needs to do this better. Here is what we need to do with enums:

- Enum values should be objects, not just numbers.
- Further, enums should have indexed values like C enums, but they should not be implicitly convertible because it reduces readability. There should be an operator like `<enum>'s ordinal` or `<enum>'s index`.
- There should also be a way to convert a number to an enum value, something like `<enum>s[<index>]`, making it like taking the list of the enum's values and grabbing one value by list index.

So, the initial question is how to best do this for the sake of efficiency, sure, but mostly this is a question of syntax. What should enums LOOK like?

Below are some ideas I've heard of or come up with so far. I used the example of all the special kinds of "tokens" in this languages used for some CSE340 assignments. It was a C-like language where some tokens were keywords like "while" and "switch" and some were symbols like the curly braces.

The initial solution I came up with would be declared in \English as something like this:

```
a new type of enum with <literal list Enum Values> where <op=>enum value Value Constructor>
```

The idea is to make a list of the values, then process each of the values given unsing the constructor. It lets you declare all the enums together in a concise way while letting you also make constructor-like operators to create each value to do any processes needed to construct custom enum types with extra properties and whatnot.

I used the example of all the special kinds of "tokens" in this languages used for some CSE340 assignments. It was a C-like language where some tokens were keywords like "while" and "switch" and some were symbols like the curly braces. Here's the example:

```
define <type of enum> token type:  a new type of enum with VAR, WHILE, INT, REAL, STRING <5>, BOOLEAN,
TYPE, LONG, DO, CASE <10>, SWITCH, PLUS as '+', MINUS as '-', DIV as '/', MULT as '*' <15>, EQUAL as
'=', COLON as ':', COMMA as ',', SEMICOLON as ';', LBRAC <20> as '[', RBRAC as ']', LPAREN as '(', RPAREN
as ')', LBRACE as '', RBRACE <25> as '', NOTEQUAL as "<>", GREATER as '>', LESS as '<', LTEQ as "<=",
GTEQ as ">=" <30>, DOT as '.', I.D., NUM, REALNUM, and ERROR where <token type value> ([A-Z.]+) (as <string
Symbol>)?  := a new enum value  with implicit string Symbol := Symbol or  lowercased;
```

The "where..." portion of the expression could even be optional if it's declared as a plain literal enum, i.e. each item is just the literal enum value's name and there are no extra properties. Maybe the "where" clause could be placed at the beginning of the operator instead of the end; that's somehting to consider.

1.1.1 Inheritance?

There's another big question here: should you be able to have new types of enums based on other enum types? For example, in Minecraft, there is a static set of different kinds of blocks. Within that, they have 16

different colors/kinds of wool blocks. How can you easily declare a block type enum and a wool type enum? *Should* you be able to?

Assuming you should be able to, I can think of a few ways to implement such functionality currently. The examples below relate to that Minecraft example; `DIRT` and `GRASS` are types of non-wool blocks and `YELLOW WOOL` and `ORANGE WOOL` are types of wool:

- Enums with sub-types could be delcared with the subtypes as values.
  - Example:
    define ¡type of enum¿ block type: a new type of enum with DIRT and GRASS + wool types;
    define ¡type of block type¿ wool type: a new type of block type with YELLOW WOOL and ORANGE WOOL;
  - Pros:
    * It's easy to implement in \English.
    * It's clean to read.
    * It maintains that all enum values are declared with the enum type itself, preventing weird stuff like letting people add more values to an enum at runtime.
  - Cons:
    * You can't add more values to a pre-existing enum at runtime... but you also can't do it at compile time! If `block type` was already made for you in a library, but you want to have `wool types` as well for your code, you would have to go back to the library source code and modify it to add that "`+ wool types`", which is something no one wants to have to do and no one should do. In some cases, going back and modifying a utils library may not even be possible.
- Declare wool types as a type of `block type` and behind the scenes, the list of `wool types` will be tacked onto the end of the list of `block types`.
  - Example:
    define ¡type of enum¿ block type: a new type of enum with DIRT and GRASS;
    define ¡type of block type¿ wool type: a new type of block type with YELLOW WOOL and ORANGE WOOL;
  - Pros:
    * It's easy to implement in \English.
    * It's even cleaner to read.
  - Cons:
    * It "obfuscates" potentially large parts of the list of `block type`'s values. Imagine that you're reading this code for the first time and you want to know what kinds of blocks are available. You would find the `block type` declaration and think that `DIRT` and `GRASS` are the only `block types`. Who would guess that there could be other things like `wool type` declared in other parts of the code in different files in different folders that could change that?
- Disallow inheritance completely. In C-like languages, this is what is done, and in those language, it leaves you hanging. However, in \English, we do have a way to make it work pretty nicely without inheritance by simply making the `wool type` type a `block type` with some kind of restriction, e.g. its name ends in "WOOL":
  - Example:
    define ¡type of enum¿ block type: a new type of enum with DIRT, GRASS, YELLOW WOOL, and ORANGE WOOL;
    define ¡type of block type¿ wool type: a new type of block type where ¡wool type¿'s name ends with "WOOL";

- Pros:
  - It's easy to implement in \English.
  - It's still clean to read, though maybe not as much so as the above possibilities.
  - It requires no modification of higher types which may be coming from a library.
- Cons:
  - Like the first option, it won't let you add more items to an enum's list of values at compile time.

While considering these solutions or others, we have to consider not only the ease of use, but also possible conceptual dangers. We have to try to make it so that it's not easily abusable; don't let bad coders do bad things.