Hopper Language

Hopper is a reaction to working professionally in C# and C++ for more than two decades and then being disappointed by the language choices available in the retro 8 bit and Arduino domains.
Hopper builds on the familiarity of C, just like Java, JavaScript and C# did, while retaining the implementation simplicity required to exist on smaller footprint platforms.

Influences

While years of C++ and C# have left me disillusioned with conventional OOP, I am still a fan of their generic compound types, specifically lists, dictionaries and arrays. A rich string type is also essential in a modern language.
Turbo Pascal played an influential role in my earlier development years and I enjoy the benefits that *units* bring for both encapsulation and for breaking complex systems into modules.

Improvements

Many of these will be subjective. If you don't like them, feel free to build your own system.

Expressions are not Statements

Ever done this?

```
if (a = 0)
{
    ...
}
```

This is the root cause of an entire class of defects that stem from allowing a statement to behave like an expression (zero is assigned to rather than compared to .. and the "value" returned by this "expression" is "a").
Unlike C, this would be illegal in Hopper.
One downside of this is that return values from functions must be consumed, even if you don't need them: `functions` are expressions because they return a value, `method` calls (`functions` that don't return values) are statements.

Spaces in paths are illegal

The Hopper environment simplifies paths a lot:

- no spaces in paths
- forward slashes as path separators: /
- only letters and digits in paths

Spaces can now be used unambiguously to delimit command line arguments. No more defects because you forgot to quote a path or argument.
Dash is illegal in file and folder names so now it can unambiguously be used in option arguments.

No pointers

No new or malloc, so no memory leaks. No pointers, so no null pointers. The execution engine is a stack machine which makes the stack itself a trivial reachable live object list for garbage collection (along with a ruthless copy on assignment policy).

No threads

A single threaded platform simplifies language design a lot and results in an incredibly stable implementation. Consider memory allocation and garbage collection: much simpler with only one thread.

Blocks are required

Ever done this?

```
if (a == 0)
    foo();
```

Followed by this?

```
if (a == 0)
    foo();
    bar();
```

This would be illegal in Hopper since blocks are required. Structural statements all *require* blocks, as-in:

```
if (a == 0)
{
    foo();
}
```

Switch cases are structural so they require blocks too.

Break means break

In C-like languages the same keyword, `break`, is used to do two completely different things:

a) to exit the current loop
b) to jump to the end of a `switch` statement

In Hopper, `break` is not abused by the `switch` statement which means `break` always exits the current loop. Switch cases *never* fall through to the next case (another common source of program errors not possible in Hopper) so `break` is not needed.

Returning Errors

In C we have APIs like `atoi` which converts an ascii string to an `int`. When the string cannot be converted to an `int`, it returns zero. So how do you tell the difference between zero and an error?

The other form of overloading of return values in traditional APIs is when `NULL` is returned on failure like when `fopen` fails or when `malloc` fails. In all these cases we don't really learn a lot about the error (that's what `errno` is for - a horrible hack).

C# brought the concept of TryParse to the foreground: a reasonable way to return two values: a boolean to represent success or failure and a value to contain the success. Hopper follows this pattern where possible for errors that are not "exceptional". For example:

```
bool TryParseInt(string source, ref int result);
```

For files (and directories) there are many possible failure opportunities for every API so using this style would have created serious code clutter. I chose to simplify error handling for IO by making a single API to check for errors and then implementing the file APIs so they *never* fail catastrophically. For example, if you read a line from a bad file handle, you just get an empty string, not a crash or other failure. `File.IsValid` is the property you use to tell that everything is ok: the file opened fine, the end-of-file has not been reached yet, etc. For example:

```
file myFile = File.Open(“myfile.txt);
string text;
while (file.IsValid)
{
    text = text + file.ReadLine();
    text = text + char(0x0A); // line break
}
```

Broad Strokes

This summary assumes familiarity with C flavoured languages. Hopper is similar to C#, JavaScript and Java in that it is compiled to byte code which is interpreted by a runtime (yes, in theory it could be JITed too ...).

Value Types

The Hopper runtime stack slots are 16 bits wide. This makes it easier to implement on smaller platforms and it influences the available value types. All value types are initialised to zero by default.
They are:

- `uint` - 16 bit unsigned
- `int` - 16 bit signed
- `bool` - true or false (but 16 bits on the stack)
- `byte` - 8 bit unsigned (but 16 bits on the stack)
- `char` - 8 bit unsigned (but 16 bits on the stack)
- `enum` - 16 bit unsigned
- `flags` - 16 bit unsigned - more about this type later..

Reference Types

References are Hopper’s replacement for pointers. References are unsigned 16 bit values for the same implementation reasons mentioned above. References are the mechanism that allows more complex types to exist in Hopper.
They are:

- list - collection class that supports insertion and deletion
- dictionary - collection of key-value pairs, keys can be strings or any value type
- array - size is fixed at declaration time, can only contain value types
- `string` - special type of list that contains `char`
- variant - a container to “box” value types to insert them as members of lists or dictionaries
- `long` - 32 bit signed type
- `float` - 32 bit floating point numeric type

References are copied on assignment. These may seem expensive but so far has not proven to be a problem. This is how new complex types are created.
References are garbage collected. This is how complex types are destroyed. Most garbage collectors work on the basis of walking reachable objects, marking them, and then sweeping all objects to release the unmarked ones.

The Hopper garbage collector treats the stack as a map of reachable objects: when a reference is popped from the stack, it is automatically collected. This simple scheme works for two reasons:

- Hopper is single threaded
- references are copied on assignment

Flags

C# has a [`flags`] attribute that can be used in conjunction with `enum` types. Hopper goes one step further and makes flags into a type of their own. In Hopper, `enum` is a simple type. Unlike the C variants, actual values cannot be assigned to enum members. `enum` members do not support arithmetic at all.
`flags` are intended to be used as bit flags. Their members can be assigned values and arithmetic is supported. `&` and `|` are the most obvious.

Switch

Conventionally, `switch` supports ordinal value types. In addition, Hopper also supports the ordinal reference types `string` and `long`. Basically, switch cases can be any type which is reduced to a literal constant at compile time, even strings.

Units

Hopper units are based on Turbo Pascal units, a way of breaking complex systems into logical modules. The `uses` statement is a cross between C's `include` statement and C#'s `using` statement.
The `uses` statement includes units as source into the compilation process and replace the need for project or make files. Pascal had the same advantage.
However, unlike the C's `include`, order doesn't matter. C and C++ were designed to be compiled in a single pass so header files are used to declare types and functions before their definition is encountered. Pascal solves this problem with forward declarations.
C# solves the issue by simply doing more than one pass. Hopper is the same: it compiles in two passes:

- first all the declarations are collected, all used units are walked
- second, all the functions and methods are compiled

Order doesn't matter in Hopper. Units are included only once even if they are mentioned in multiple uses statements.

First Pass

Hopper collects declarations for the following in the first pass:

- constants
- member variables
- enum types
- flags types
- functions
- methods

Second Pass

Hopper compiles the methods and functions on the second pass (the parts between the { and the }).
Member variables are automatically initialised to their zero values. In addition, they can be initialised by assignment to an expression. This expression is compiled during the second pass so it can depend on constants, methods and functions.

Preprocessor Directives

Hopper supports the following preprocessor directives:

- #define
- #ifdef
- #ifndef
- #else
- #endif

Hopper also supports #if with expressions the way C# does but the symbols are purely boolean. For #ifdef and #ifndef they see symbols that are defined as true and those that are not defined as false. For #if, symbols can be defined, not defined, true or false. Expressions can be combined as boolean expressions using && and ||.
Where Hopper differs from C is the way preprocessor #ifdef .. #endif directive scope works: it aligns with the two passes. An #ifdef at the file/declaration level needs to have a matching #endif at the file/decleration level. An #ifdef inside a method, needs a matching #endif within the same method.

Public and Private

Unit members can be private to the current unit or visible outside the unit if public. The way Hopper declares members and public or private is simple:

- if the member name starts with a lowercase character, the member is private
- if the member name starts with an uppercase character, the member is public

Hopper identifiers include letters and numbers. They must start with a letter. Underscores are not allowed (since they are ugly).

Delegates

Delegates are Hopper's 'function pointer' type. They support all the features of Hopper methods and functions (`ref` arguments, overloads, etc.) and are strongly typed.
In a nutshell, they are simply 2 byte method indices that can be passed around as variables.
Delegates are one of the very few instances in Hopper where zero initialization is not enough: delegates must be initialised to reference a method of function before they are called or else a runtime error will result.

Scope

As mentioned earlier, a "global" member name starting with a lowercase letter is only visible within the current unit. This includes constants, member variables, enum types, flags types, properties, methods and functions. An uppercase first letter implies "public" which means visible to the entire Hopper program. However, member variables may never be "public". So public members can include constants, enum types, flags types, properties, methods and functions.
Members that are uniquely named across the entire Hopper program do not need to be qualified with their unit name. For example, `PrintLn(...)` from the `Screen` unit can be used

just like that as long as nobody defines a new method called `PrintLn(..)` with the same argument signature. If the argument signature is no longer unique, then calling it will require the unit qualifier as-in `Screen.PrintLn(...)`.
The same applies for enumeration members like `Key.Escape` in the `Keyboard` unit. If someone declares a new public member called `Key`, then we'll need to qualify the one from `Keyboard` as `Keyboard.Key.Escape`.
There is an obvious desirable exception to this: if you use a so-called ambiguously named member from within the unit in which it is defined, then there is never any reason to qualify it. The locally defined member will always take precedence over any similarly named member in a different unit.

this

Hopper methods and functions support some syntactic sugar in the form of a "this" argument. Illustration by example is the simplest. Consider these method and function definitions (which are qualified by their unit names for clarity):

```
	string File.ReadLine(file f)
	string String.Substring(string this, uint start, uint length)
```

Conventional use of these would look something like:

```
	string line = File.ReadLine(f);
	string substring = String.Substring(source, pos, len);
```

With the "this" syntax we can write it as follows instead:

```
	string line = f.ReadLine();
	string substring = source.Substring(pos, len);
```

In short, if the type of the variable is the same as the type of the system unit where the method is defined and the first argument of the method is that type, then this abbreviated "this" syntax can be used.

Child Programs

Hopper runs a single process but it can launch a child program with the System.Execute API:

```
	uint Execute(string programPath, <string> arguments)
```

On returning from the child program, execution of the parent/calling program continues. The calling program is protected from errors or or failures in the child program. The `uint` return value will contain the error value if the child program called `Diagnostics.Die(uint error)` in the event of a catastrophic or irrecoverable failure (either a system failure like division by zero for example or a failure intercepted in the user code).

Constraints

Hopper is designed to generate bytecode which will be efficient and easy to execute on smaller platforms (like legacy 8 bit processors). Most of these constraints exist for this reason.

Instructions

Each instruction is a byte. This means there will never be more than 256 instructions in the Hopper instruction set. Since the Hopper runtime is a stack machine, many instructions have no operands and truly are one byte wide. For those with operands, we try to squeeze the operand down to just one more byte. This is why we have short jump (offset -128..+127) and long jump instructions (2 byte offset).

Method Size

Each method is a stand alone code "segment" unrelated to other methods. Based on the above, jumps limit the size of method code: in the worst case scenario, a forward jump from the start to the end of a method could not exceed 32767 bytes in distance. That's bytes of code, not instructions.

Total Methods

Short call instructions are two bytes long where the second is the method index (0..255). The most commonly called methods are allocated one byte indices. There is also a long call instruction that takes a 2 byte index and that defines the limit: a Hopper program cannot have more than 65535 methods. Delegates are always stored as 2 byte (stack slots) method indices with the msb cleared for short indices.

Dictionaries

Dictionaries are implemented using hash tables. This is why the order of the elements in a dictionary is indeterminate. Only two types of hash keys are supported: `string` and `uint` (`byte` and `char` work too as a 'subset' of the supported key types).

Lists

Although lists are implemented as a unidirectional linked list, they use `uint` for indices, length and implementation optimizations. This means the maximum number of elements in a list is 65535.

Arrays

Arrays are intended to be a more efficient collection at runtime than lists and dictionaries. Members of lists and dictionaries are reference types which means they need to be kept track of by the object manager (allocation and garbage collection) and they need to be copied on assignment.
Arrays can *only* contain value types (`char`, `int`, `uint`, `bool`, `enum` and `flags`).
Unlike lists and dictionaries, the size of an array is fixed at creation: it cannot grow or shrink after it is created. The array itself is a reference type which means it can be inserted into a list or dictionary as a member. This allows arrays to be easily optimised for speed and size so, for example, an array of bool can be implemented as 1 bit per entry and provide similar efficiency to Turbo Pascal sets but without the 256 member limit.

Walk Through Presentations

Hello World

So, what does this actually look like?
https://docs.google.com/presentation/d/1ZfqbBSqEAHBDnZLWSe-3sygjJOas6WJYlS9MmkPlFR8/

Hopper Projects

No make files. No project files. Just units and uses statements.
https://docs.google.com/presentation/d/1AP5HfBwkcK6xj-8Z5_-iPWaELDSroDuUKiATp3Y2iq4/

Runtime Library

No programming language lives in isolation. A good runtime library is required to do useful things with your language. The .NET Framework was a “planned” runtime library rather than the typical way in which these things evolve incrementally over time. They looked over their shoulders at the success of the Java library, learned from some mistakes, and tried to build a single large library that covered all domains and made sense. In the early days they used the word “orthogonal” a lot to describe the consistency of behaviours across the different unrelated sections of the library: if you know how `TryParse` works for `int`, you’ll quickly figure out how it works for `TimeSpan`. If `foreach` works with lists, then it should work with strings and dictionaries too.

What does this have to do with Hopper? Well, between Java and C# (and to a lesser extent, JavaScript) they have standardised how we come to expect very common APIs to work. For example, string helper methods and properties: Length, Substring, Replace, Trim, Split, etc. should all behave exactly the same way as they do in both Java, C# and JavaScript.

The same applies in many other areas where it would be senseless to invent yet another API rather than making Hopper behave exactly as you would already expect it to. The following are all Hopper API methods and properties. You already know exactly how they behave:

```
System.CurrentDirectory
Path.Combine(...)
Path.GetFileName(...)
File.Exists(...)
Char.IsUpper
Char.ToLower
List.Append(...)
List.Clear()
UInt.ToString()
```

Other familiar patterns concern the Hopper collection classes: Dictionary and Array have a Count property, String and List have a Length property - just like you’d expect. The syntax for defining collections should also look very familiar to you if you have either a C++ template or C# background:

```
<string,string> dictionaryOfStrings;
<string, <int> > dictionaryOfListOfInts;
<bool> listOfBools;
uint[1000] arrayOfUInt;
```

That sums it up for the motivation of why the runtime library looks the way it does. The Hopper runtime is compiled from source and is broken into one unit per type. Some APIs are “system” (part of the runtime) and others are Hopper source.

What follows for the remainder of this document is a detailed reference section describing the runtime library broken down by unit.

Value Types

Boolean

Byte

Char

Int

Type

UInt

Reference Types

Float

Array

Dictionary

List

Long

Pair

String

Utility

Color

Path

Time

IO

Directory

File

Keyboard

Screen

Platform

Diagnostics

System

VM