The Young Person's Guide to zkl

zkl doesn't attempt to boil the ocean,
zkl doesn't want to warm the ocean.
zkl wants to make a nice cup of tea.
--Zander Kale

zkl[1] is a general purpose object oriented programming language. It is imperative but borrows concepts from many programming paradigms, including functional. It is curly-bracketed, dynamic, reflective, and threaded. It has built in garbage collection, strings, lists, dictionaries, threads, fibers (continuations and co-routines) and more. The syntax strongly resembles the C programming language while the data model is closer to that of Python and Smalltalk. The goal of zkl is to enable rapid prototyping, quickly developing solutions to problems and does this at the expense of the strong static checking needed for production oriented languages. On the other hand, zkl has a level of verbosity that slows development somewhat, compared, for example, to Python or (especially) Perl.

In zkl, everything is an object, even numbers. Classes and functions behave like any other object, including numbers. Classes and functions can have the same anonymity as numbers[2]. They usually don't but it is nice to be able to create them on the fly, like 123, if you need to. Even threads are objects.

The "Hello world" program looks pretty much the same as it does in most languages:

```
println("Hello world");
```
or
```
"Hello again".println();
```

There is no need for include files or main(). Even the compile step is invisible.

Another example, the ever popular factorial program:

```
fcn fact(x){  // Input: x  output: x!
   if (0==x) return(1);         // 0! → 1
   return( x * fact(x - 1) );   // x! → x * (x-1)!
}
```

Looks very similar to C, Java or Python. Since zkl integers are 64 bits long, it can generate big numbers:

```
fact(37)
```
→ 10969079327018188803

1 Pronounced zee·kay·el or zee·kale (for American English speakers). Or however you like.
2 "Lambda" functions
3 Using the BigNum extension: `fact(BigNum(50)) : "%,d".fmt(_)` → 30,414,093,201, 713,378,043,612, 608,166,064,768,844,377,641,568,960,512,000,000,000,000

zkl has core support for preemptive threads. Every class is threadable, as are functions:
```
fcn thread{ println("I am running in ",vm); }
thread.launch() → "I am running in VM#543"
```
zkl includes a fair amount of threading support, such as locks, events, semaphores, waiters, timers, heartbeat timers and pipes (not to be confused with Unix pipes although they are used for interprocess communication).

The zkl core objects have been designed to try and have the same "look and feel" so they can be used interchangeably. For example, strings, lists and files have common functionality that allows them to be used in the same way in foreach loops:
```
foreach char in ("123")                // String: Print 1 2 3
   { println(char); }
foreach item in ( List("1","2",3) )    // Print 1 2 3
   { println(item); }
foreach line in (File("f.txt","r"))    // Print every line
   { print(line); }
```
If you look closely at the list example, you'll notice that lists are heterogeneous, that is, they can contain *any* zkl object. In this case, two strings and a number.

The zkl exception system is very similar to that of Java, Python and other languages.
```
try{
   gettingOutOfBedWasABadIdea:=consultHoroscope(today);
   if (gettingOutOfBedWasABadIdea)
      throw(Exception.BadDay("Coffee please"));
}
catch(BadDay){
   println("I heard somebody say: ",__exception.text);
}
```
When this code is run and my horoscope for today isn't so good, the code will print:
```
I heard somebody say: Coffee please
```

Getting Started

The zkl executable is self contained; it is all you need to get started programming in zkl. Running zkl gives you an interactive shell:
```
> zkl
zkl: 1+2
3
zkl: var x=L(1, "2", 3.4, L(5,"six"), self, fcn(x){ x + 1 })
L(1,"2",3.4,L(5,"six"),Class(__class#0),Fcn(__fcn#1))
zkl: x.apply("type")
L("Int","StringConst","Float","List","Class","Fcn")
zkl: ^C
>
```
To create programs, use your favorite text editor to create a .zkl file and then run it:
```
zkl myprogram.zkl
```

```
zkl myprogram
```

Data Things: Numbers, Strings, etc

Numbers

Numbers can be integer or floating point. Which one is determined by the number itself. For example, 1 is an integer and 1.2 is a floating point number. You can do all the number things: add, subtract, etc. It is important to note that the first number in a calculation determines which type of number is the result of a calculation. For example:

```
1 + 2          → 3    // an integer
1.5 + 2        → 3.5  // a float
1 + 2 * 3      → 7    // an integer
1 + 2.5 * 3 → 8    // an integer BUT since multiplication is performed before addition,
// the calculation becomes:
1 + (2.5 * 3) → 1 + 7.5 → 8
```

Strings

A string is a bunch of characters in double quotes. For example: “This is a string”. Strings are immutable, that is, they can't change (just like numbers). To change a string, you create a new string from the old one:

```
"New " + "string" → “New String”
```

Lists

A list is an ordered collection of objects. You create them by giving objects to a list and it will hand you back a new list with those objects in it. “List” is the “mother” list (and is usually referred to as “L”). So, to create a list of two objects: `L(1,"two")`. Lists are mutable and can change. To add objects to the end a list, you can use the append method or the plus operator:

```
a:=L(1,"two"); a.append(3.5)      → L(1,”two”,3.5)
a + "four"                        → L(1,”two”,3.5,”four”)
```

You can do many things with lists. For example, to sort it:

```
L(5,12,3.5,7).sort()              → L(3.5,5,7,12)
```

To create a list of hex strings from a list of integers:

```
L(9,10,11).apply("toString",16) → L("9","a","b")
```

Dictionaries

Dictionaries are like the book: a word (the key) has a definition (the value). Anything can be a key and anything can be value.

```
d:=Dictionary();    // Create a dictionary
d["one"]=2;       // The key is “one” and value is the integer 2
d[3]="four";
d → D( 3:four one:2 )
```

Etc

Other objects include TCP/IP sockets, regular expressions, unit tests, time and date, fibers (continuations and co-routines), threads, a data container/byte stream editor and atomic objects. You can see a list by typing “`Vault.dir()`”.

Branching

Conditional branching is the traditional if/else:

```
if (1) println("This is done always");
if ("one"==2) println("Nope")
else          println("Yep");
```

An “if” actually has a value:

```
x:=1;
println( if (x==1) "one" else "not one" ); → “one”
```

Loops

zkl provides three types of looping: while/do loops, foreach loops and “transform the collection” loops.

```
while(condition){ code } // run code while condition is true
do{ code }               // run code while condition
while(condition);        // is true, at least once
do(n){ code }            // run code n times
```

Foreach loops walk through an object using a Walker (which would be called an iterator in languages such as C++, Java or Python). Most objects support a walker so you can traverse them.

```
foreach n in (object) // run code for each item in object
   { code }
foreach n in (L(1,"two",3)){ println(n); } → “1”, “two”, “3”
```

Additionally, walkers can be used by themselves:

```
L(1,"two",3).walker().walk() → L(1,”two”,3)
w:=L(1,"two",3).walker(); w.peek() → 1
```

Functional languages support various ways of applying a function to a collection of data in one statement. This is a very powerful idea, which zkl supports through these methods: apply/pump, filter and reduce.

```
L(1,2,3).apply('+(5))              → L(6,7,8)  // aka map, mapcar
L(1,2,"three",4).filter("isEven") → L(2,4)
L(5,3,7,99,8).reduce((0).max)      → 99   // aka fold
fcn enum(x,ref){return(ref.inc(),x)}
L("one","two").apply(enum,Ref(0)) → L( L(0,”one”), L(1,”two”) )
```

Most container objects (such as lists, files, strings) support these methods, as do Walkers.

Functions

Code is organized in functions, indicated by the “fcn” keyword. A simple function might be:

```
fcn hello{ println("Hello World"); }
```

`hello()` runs the function. There is no requirement for functions to be named, so the previous example could be written as a lambda function:

```
fcn{ println("Hello World") }(); // run the lambda function
```

This notation is handy when you want to pass functions to other functions:

```
f(fcn{ println("Hello World") });
```

is the same as `f(hello)` (the function hello). Functions can return more than one thing,

```
fcn f{ return(1,2,3) }
```

is a function that returns a list of three integers. Then, `a,b,c:=f()` would set a to 1, b to 2 and c to 3.

Classes

Classes are containers, they hold other objects: classes, functions, and data.

```
class C{
   var c;      // instance variable
   println("This is part of the class constructor");
   fcn init(v){
     println("This function is run when "
             "a new instance is created");
     c=v;            // set the class variable
   }
   fcn hello{ println("Hello"); }
}
```

Calling the class creates a new instance (copy) of the class:

```
c:=C(1) →
```

This is part of the class constructor
This function is run when a new instance is created

You can also access the class variables and functions:

`c.hello()` → "Hello"
`c.c` → 1

The class C is just an "reference" instance; in such classes, all variables are initialized to Void (unless they are set in the constructor).

`C.hello()` → "Hello"
`C.c` → Void

Scope

zkl is block (lexically) scoped; if something is created in a block, it wants to stays there. Except functions and classes, which are Class scoped. Closures and partial function application can be used to give functions one way lexical scoping.

The zkl Shell

The shell provides an interactive REPL[4] environment where you can get immediate feedback to your experiments. Often, while writing programs, it is convenient to test a few things in the shell and then paste them into your program. Classes, functions and variables stick around so you can reference them later.
The shell is documented in Objects.startup.

4 Read-eval-print loop: http://en.wikipedia.org/wiki/REPL

Shared Libraries

Shared libraries (DLLs) enable objects written in C to be added to the system. Example objects include a zlib (gzip) interface, allowing both streaming and static compression and inflation and a LZO compression object.

```
Zeelib:=Import("ZeeLib");
text:=Zeelib.Compressor(True).write("This is a test").close();
(f:=File("foo.txt.z","wb")).write(text.drain()); f.close();
```

That creates the file “foo.text.z” with the compressed text in it. To check:

```
>gzip -dc foo.txt.z
This is a test
```

Concept to Topic Mapping

Closure, Partial application:
- Function: 'wrap, '{}, Partial (Objects.Deferred)
- Objects: .fp (Object).

Command line processing:
- Objects: Utils.Argh

Lambda:
- Keywords: Fcn (anonymous functions)
- Objects: Deferred ('wrap)

Lazy and infinite lists:
- Objects: Walker

Lazy objects:
- Objects: Deferred

Looping, functional:
- Methods: apply, filter, pump, reduce, Utils.zipWith, Utils.Helpers.cycle
- Objects: Fcn (tail recursion),Walker

Looping, imperative
- Keywords: do, foreach, while, continue, break
- Objects: Utils.Generator, VM (yield), Walker

Networking:
- Objects: Network.TCP

Packaging:
- Keywords: pimport
- Objects: Import, Utils.Wad

Parameters (arg lists), manipulating:
- Objects: List (xplode), VM (arglist, nthArg, numArgs, pasteArgs)

Repl (read-evel-print-loop), the command line:
- Objects: startup

Sugar:
- Keywords: ask, print.
- Objects: Deferred ('wrap and '{}), Op ('+,'-,'*,'/)

Threads, cooperative (green):
- Objects: Fcn (strand, stranded)

Threads, native:
- Keywords: class (threading)
- Objects: Class (launch, liftoff, splashdown), Fcn (launch, future), Thread.Pipe, Thread.Straw

Unit testing:
- Objects: Test

> *Use zkl to focus on solutions! Order before midnight and receive a FREE microscope!*
> -- Zander Kale