ooeForth

SVFIG

Chen-Hanson Ting
June 26, 2021

Java Forth

- There were several Forth implemented in Java.
- There was even an Java eForth implemented by Michael A. Losh in 1997.
- They were all very complicated beyond my comprehension.

Java Eforth

- I wanted a simple Java Forth modeled after jeforth614.
- Every Forth word should be an object.
- Java is a better host to Forth than JavaScript.
- ooeForth is a truly object oriented Forth.

ooeEforth

- There are only two types of words:
  - Primitive words
  - Colon words
- All system words are primitive objects.
- All user defined words are colon objects.

ooeForth

- A single class Code constructs all Forth words as objects.
- A single method with a giant `HashMap` executes all primitive objects.
- `nest()` method executes colon objects.

ooeForth

- **All colon objects contain linear object lists.**
- **All colon objects are executed by this very simple inner interpreter:**

```
nest(){for(var w:pf) w.xt();}
```

- **Great appreciation to Shawn Chen and Brad Nelson.**

Eforth112 Object

- Stack: value list
- Rstack: value list
- Dictionary:
  - Primitive list + Colon list
- Method:
  - main(), Outer Interpreter
- Class Code constructs all objects

class Eforth112

stack

rstack

dictionary

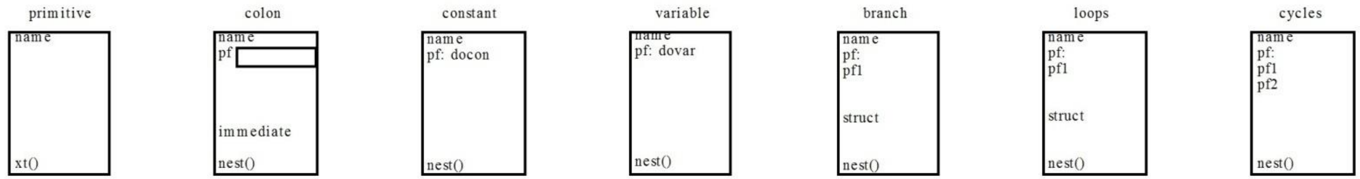
compiling

base = 10;

fence = 0;

wp,ip;

idiom;



Dictionary Instantiation

```
main()
colon = new Code(":"); colon.token=fence++; dictionary.add(colon);
semi = new Code(";"); semi.token=fence++; semi.immediate=true; dictionary.add(semi)
dup = new Code("dup"); dup.token=fence++; dictionary.add(dup);
over = new Code("over"); over.token=fence++; dictionary.add(over);
qdup = new Code("4dup"); qdup.token=fence++; dictionary.add(qdup);
swap = new Code("swap"); swap.token=fence++; dictionary.add(swap);
rot = new Code("rot"); rot.token=fence++; dictionary.add(rot);
rrot = new Code("-rot"); rrot.token=fence++; dictionary.add(rrot);
dswap = new Code("2swap"); dswap.token=fence++; dictionary.add(dswap);
pick = new Code("pick"); pick.token=fence++; dictionary.add(pick);
roll = new Code("roll"); roll.token=fence++; dictionary.add(roll);
ddup = new Code("2dup"); ddup.token=fence++; dictionary.add(ddup);
dover = new Code("2over"); dover.token=fence++; dictionary.add(dover);
drop = new Code("drop"); drop.token=fence++; dictionary.add(drop);
nip = new Code("nip"); nip.token=fence++; dictionary.add(nip);
ddrop = new Code("2drop"); ddrop.token=fence++; dictionary.add(ddrop);
tor = new Code(">r"); tor.token=fence++; dictionary.add(tor);
rfrom = new Code("r>"); rfrom.token=fence++; dictionary.add(rfrom);
rat = new Code("r@"); rat.token=fence++; dictionary.add(rat);
plus = new Code("+"); plus.token=fence++; dictionary.add(plus);
minus = new Code("-"); minus.token=fence++; dictionary.add(minus);
mult = new Code("*");  mult.token=fence++; dictionary.add(mult);
div = new Code("/"); div.token=fence++; dictionary.add(div);
mod = new Code("mod"); mod.token=fence++; dictionary.add(mod);
starsl = new Code("*/"); starsl.token=fence++; dictionary.add(starsl);
ssmod = new Code("*/mod"); ssmod.token=fence++; dictionary.add(ssmod);
and = new Code("and"); and.token=fence++; dictionary.add(and);
or = new Code("or"); or.token=fence++; dictionary.add(or);
xor = new Code("xor"); xor.token=fence++; dictionary.add(xor);
negate = new Code("negate"); negate.token=fence++; dictionary.add(negate);
zequal = new Code("0="); zequal.token=fence++; dictionary.add(zequal);
zless = new Code("0<"); zless.token=fence++; dictionary.add(zless);
zgreat = new Code("0>"); zgreat.token=fence++; dictionary.add(zgreat);
equal = new Code("="); equal.token=fence++; dictionary.add(equal);
less = new Code("<"); less.token=fence++; dictionary.add(less);
great = new Code(">"); great.token=fence++; dictionary.add(great);
nequal = new Code("<>"); nequal.token=fence++; dictionary.add(nequal);
gequal = new Code(">="); gequal.token=fence++; dictionary.add(gequal);
lequal = new Code("<="); lequal.token=fence++; dictionary.add(lequal);
baseat = new Code("base@"); baseat.token=fence++; dictionary.add(baseat);
basest = new Code("base!"); basest.token=fence++; dictionary.add(basest);
hex = new Code("hex"); hex.token=fence++; dictionary.add(hex);
decimal = new Code("decimal"); decimal.token=fence++; dictionary.add(decimal);
cr = new Code("cr"); cr.token=fence++; dictionary.add(cr);
dot = new Code("."); dot.token=fence++; dictionary.add(dot);
dotr = new Code(".r"); dotr.token=fence++; dictionary.add(dotr);
udotr = new Code("u.r"); udotr.token=fence++; dictionary.add(udotr);
key = new Code("key"); key.token=fence++; dictionary.add(key);
emit = new Code("emit"); emit.token=fence++; dictionary.add(emit);
```

Forth Outer Interpreter

```
while(!(idiom=in.next()).equals("bye")) {  // parse input
Code newWordObject = null;
for (var w : dictionary) {  // search dictionary
if (w.name.equals(idiom))
{newWordObject = w;break;};}
if(newWordObject != null) {  // word found
if((!compiling) || newWordObject.immediate) {
try {newWordObject.xt(); } // execute
catch (Exception e) {System.out.print(e);}}
else {  // or compile
Code latestWord = dictionary.get(dictionary.size()-1);
latestWord.addCode(newWordObject);}}
else {
try {int n=Integer.parseInt(idiom, base); // not word, try number
if (compiling) {  // compile integer literal
latestWord = dictionary.get(dictionary.size()-1);
latestWord.addCode(new Code("dolit",n));}
else { stack.push(n);}}// or push number on stack
catch (NumberFormatException  ex) {// catch all errors
System.out.println(idiom + " ?");
compiling = false; stack.clear();}}}
System.out.println("Thank you.");
in.close();}
```

Class Code

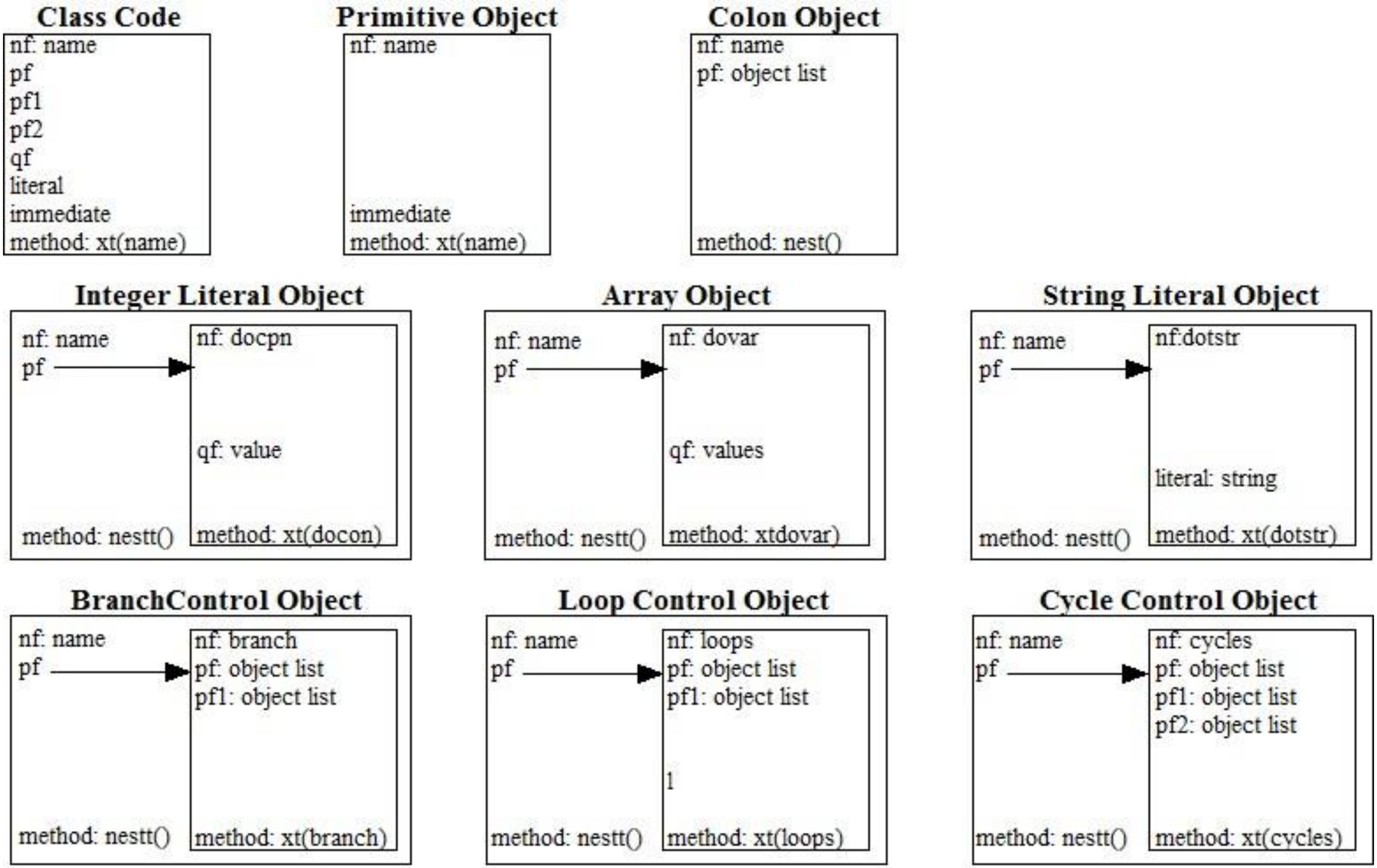
```
class Code {
name;
pf = new ArrayList<>();
pf1 = new ArrayList<>();
pf2 = new ArrayList<>();
qf = new ArrayList<>() ;
struct = 0;
immediate = false;
literal;
token = 0;
Code(String n) {name=n;}
Code(String n, int d) {name=n;qf.add(d);}
Code(String n, String l) {name=n;literal=l;}
void xt() {
            if (lookUp.containsKey(name)) {
            lookUp.get(name).run();
            } else { rstack.push(wp); rstack.push(ip);
            wp=token; ip = 0;// wp points to current colon object
            for(Code w:pf) {if (w.name.equals("exit")) break; w.xt();ip++;}
            ip=rstack.pop(); wp=rstack.pop();}
            }
void addCode(Code w) { this.pf.add(w);}
HashMap<String, Runnable> lookUp = new HashMap<>() {{
```

Primitive Methods

```
// stacks
put( "dup", ()-> { stack.push(stack.peek());});
put( "over", ()-> { stack.push(stack.get(stack.size()-2));});
put( "2dup", ()-> { stack.addAll(stack.subList(stack.size()-2,stack.size()));});
put( "2over", ()-> { stack.addAll(stack.subList(stack.size()-4,stack.size()-2));});
put( "4dup", ()-> { stack.addAll(stack.subList(stack.size()-4,stack.size()));});
put( "swap", ()-> { stack.add(stack.size()-2,stack.pop());});
put( "rot", ()-> { stack.push(stack.remove(stack.size()-3));});
put( "-rot", ()-> { stack.push(stack.remove(stack.size()-3));stack.push(stack.remove(stack.size()-3));});
put( "2swap", ()-> { stack.push(stack.remove(stack.size()-4));stack.push(stack.remove(stack.size()-4));});
put( "pick", ()-> { int i=stack.pop();int n=stack.get(stack.size()-i-1);stack.push(n);});
put( "roll", ()-> { int i=stack.pop();int n=stack.remove(stack.size()-i-1);stack.push(n);});
put( "drop", ()-> { stack.pop();});
put( "nip", ()-> { stack.remove(stack.size()-2);});
put( "2drop", ()-> { stack.pop();stack.pop();});
put( ">r", ()-> { rstack.push(stack.pop());});
put( "r>", ()-> { stack.push(rstack.pop());});
put( "r@", ()-> { stack.push(rstack.peek());});
put( "push", ()-> { rstack.push(stack.pop());});
put( "pop", ()-> { stack.push(rstack.pop());});
// math
put( "+", ()-> { stack.push(stack.pop()+stack.pop());});
put( "-", ()-> { int n= stack.pop();stack.push(stack.pop()-n);});
put( "*", ()-> { stack.push(stack.pop()*stack.pop());});
put( "/", ()-> { int n= stack.pop();stack.push(stack.pop()/n);});
put( "*/", ()-> { int n=stack.pop();stack.push(stack.pop()*stack.pop()/n);});
put( "*/mod", ()-> { int n=stack.pop();int m=stack.pop()*stack.pop();
stack.push(m%n);stack.push(m/n);});
put( "mod", ()-> { int n= stack.pop();stack.push(stack.pop()%n);});
put( "and", ()-> { stack.push(stack.pop()&stack.pop());});
put( "or", ()-> { stack.push(stack.pop()|stack.pop());});
put( "xor", ()-> { stack.push(stack.pop()^stack.pop());});
put( "negate", ()-> { stack.push(-stack.pop());});
// logic
put( "0=", ()-> { stack.push((stack.pop()==0)?-1:0);});
put( "0<", ()-> { stack.push((stack.pop()<0)?-1:0);});
put( "0>", ()-> { stack.push((stack.pop()>0)?-1:0);});
put( "=", ()-> { int n= stack.pop();stack.push((stack.pop()==n)?-1:0);});
put( ">", ()-> { int n= stack.pop();stack.push((stack.pop()>n)?-1:0);});
put( "<", ()-> { int n= stack.pop();stack.push((stack.pop()<n)?-1:0);});
put( "<>", ()-> { int n= stack.pop();stack.push((stack.pop()!=n)?-1:0);});
put( ">=", ()-> { int n= stack.pop();stack.push((stack.pop()>=n)?-1:0);});
put( "<=", ()-> { int n= stack.pop();stack.push((stack.pop()<=n)?-1:0);});
// output
put( "base@", ()-> { stack.push(base);});
put( "base!", ()-> { base = stack.pop();});
put( "hex", ()-> { base = 16; });
put( "decimal", ()-> { base = 10; });
put( "cr", ()-> { System.out.println();});
put( ".", ()-> { System.out.print(Integer.toString(stack.pop(),base)+" ");});
```

Class Code

- It is an one-size-fits-all object constructor.
- It constructs all primitive objects.
- The Outer Interpreter uses it to compile all colon objects defined by user.



Class Code

nf: name
pf
pf1
pf2
qf
literal
immediate
method: xt(name)

Primitive Object

nf: name
immediate
method: xt(name)

Colon Object

nf: name
pf: object list
method: nest()

Integer Literal Object

nf: name
pf
method: nestt()

nf: docpn
qf: value
method: xt(docon)

Array Object

nf: name
pf
method: nestt()

nf: dovar
qf: values
method: xtdovar)

String Literal Object

nf: name
pf
method: nestt()

nf:dotstr
literal: string
method: xt(dotstr)

BranchControl Object

nf: name
pf
method: nestt()

nf: branch
pf: object list
pf1: object list
method: xt(branch)

Loop Control Object

nf: name
pf
method: nestt()

nf: loops
pf: object list
pf1: object list
1
method: xt(loops)

Cycle Control Object

nf: name
pf
method: nestt()

nf: cycles
pf: object list
pf1: object list
pf2: object list
method: xt(cycles)

Primitive Objects

- **nf: name**
- **token: id**
- pf
- pf1
- pf2
- qf
- **immediate: flag**
- **method: `xt(name)`**

Colon Objects

- **nf: name**
- **token: id**
- **pf: object list**
- pf1
- pf2
- qf
- immediate
- **method:** **`next()`**

Literals

- There are data literals in an object list.
- All literals are colon objects which has embedded literals:
  - Constants
  - Variables
  - Arrays
  - Strings

Constant Objects

- **nf: name**
- **token: id**
- **pf:** `docon`
- pf1
- pf2
- qf
- immediate
- **method:** **`next()`**

docon Objects

- **nf: docon**
- **token: id**
- pf:
- pf1
- pf2
- **qf: value**
- immediate
- **method:** `xt(docon)`

dovar Objects

- **nf: dovar**
- **token: id**
- pf:
- pf1
- pf2
- **qf: value**
- immediate
- **method:** `xt(dovar)`

Array Objects

- **nf: dovar**
- **token: id**
- pf:
- pf1
- pf2
- **qf: value list**
- immediate
- **method:** `xt(dovar)`

String Objects

- **nf: name**
- **token: id**
- **pf:** `dostr[dotstr]`
- pf1
- pf2
- qf
- immediate
- **method:** `next()`

dostr Objects

- **nf: dostr[dotstr]**
- **token: id**
- pf:
- pf1
- pf2
- **literal: string**
- immediate
- **method:** `xt(dostr[dotstr])`

```
Usage: $" xxx" , ." yyy"
```

Control Structures

- There are branches and loops in an object list.
- All control structures are colon objects with alternate paths:
  - `if pf else pf1 then`
  - `begin pf again`
  - `begin pf until`
  - `begin pf while pf1 repeat`
  - `for pf aft pf1 then pf2 next`

IF Object

- **nf: name**
- **token: id**
- **pf:** `branch`
- pf1
- pf2
- qf
- immediate
- **method:** `next()`

branch Object

- **nf: branch**
- **token: id**
- **pf: object list**
- **pf1: object list**
- pf2
- qf
- immediate
- **method:** **`xt(branch)`**

```
Usage: if pf else pf1 then
```

BEGIN Object

- **nf: name**
- **token: id**
- **pf:** `branch`
- pf1
- pf2
- qf
- immediate
- **method:** `next()`

loops Object

- **nf: loops**
- **token: id**
- **pf: object list**
- **pf1: object list**
- pf2
- qf
- immediate
- **method:** `xt(loops)`

```
Usage: begin pf while pf1 repeat
```

FOR Object

- **nf: name**
- **token: id**
- **pf:** `donext`
- pf1
- pf2
- qf
- immediate
- **method:** **`next()`**

cycles Object

- **nf: cycles**
- **token: id**
- **pf: object list**
- **pf1: object list**
- **pf2: object list**
- qf
- immediate
- **method:** `xt(cycles)`

`Usage:` `for pf aft pf1 then pf2 next`

Outer Interpreter

- The Forth outer interpreter is the `main()` method in Eforth112 class.
- The parser is a single Java method: `Scanner.in.next()`.
- To use `in.next()`. I sacrificed the universal Forth prompt `OK`, and the opportunity to dump the data stack.

```
in=new Scanner(System.in);String idiom;
while(!(idiom=in.next()).equals("bye")){
Code newWordObject=null;
  for (var w : dictionary){
    if (w.name.equals(idiom)) {newWordObject=w
    if(newWordObject != null){
      if((!compiling) || newWordObject.immedia
      else{  Code latestWord=dictionary.get(di
      latestWord.addWord(newWordObject);}}
    else{try {int n=Integer.parseInt(idiom, ba
      if (compiling){Code latestWord=dictionar
        latestWord.addWord(new Code("dolit",n)
      else{stack.push(n);}}
      catch (NumberFormatException  ex) {Syste
        compiling=false,stack.clear();}}}
  System.out.println("Thank you.");in.close();
```

Linear Object Lists

- Colon objects compile linear object lists in their `pf` fields.
- Linear lists can be executed conveniently.
- Linear lists can be nested indefinitely to solve complicated problems.

ooeForth

- Complicated data structures like arrays and strings are reduced to objects.
- Complicated control structures like branches and loops are reduced to objects.
- Hence the new name ooeForth.

Law of Structures

- The Third Law of Computing is the Law of Structures in my *Laws of Computing*.
- It states that all computable problems can be reduced to nested linear lists of structures.
- ooeForth proves this law.

Conclusions

- Eforth112 implements Forth words as true objects.
- It is my first Java project and shows my lack of understanding of this extremely complicated language.
- Eforth112 is logically correct but can use lots of improvements.

Link to Eforth112

- **Link to Eforth112:** https://drive.google.com/file/d/1rRlCiVuUx6jqx4axNwyX6nwQvP-_qGQ5/view?usp=sharing
- **Email comments to me:**
  - **chenhansunding@gmail.com**

Demo

Thank You!