# Implementing Object Oriented Languages

(A sketch)

February 18, 2008

# Methods

- C#: Methods, generics, objects, interfaces...
- Machine code:
  - operators: add, subtract, xor...
  - conditionals: if
  - jump
  - take CIS 371 for more details.
- Common Intermediate Language
  - object oriented byte code
  - .Net equivalent to Java byte code
  - closer to C# than machine code

How do we compile an object oriented program to machine code?

# Functions and Methods

**Functions**

- take arguments, compute, and return a result.
- have access to arguments and global variables.
- always “means the same thing” (static dispatch).
- easy to implement in machine code.

**Methods**

- take arguments, compute, and return a result.
- has access to arguments, global variables, and object members.
- have context dependent meanings (dynamic dispatch).
- are be implemented in terms of functions.

# From functions to methods

Translating methods to functions requires emulating two key method behaviors

- Access to object members:
- Dynamic dispatch:

Will also need simpleClasses (or records) which contain multiple fields but no methods.

# From functions to methods

Translating methods to functions requires emulating two key method behaviors

- Access to object members:
  Represent methods as a functions that takes special argument, this, that contains an object reference.
- Dynamic dispatch:

Will also need simpleClasses (or records) which contain multiple fields but no methods.

# From functions to methods

Translating methods to functions requires emulating two key method behaviors

- Access to object members:
  Represent methods as a functions that takes special argument, this, that contains an object reference.
- Dynamic dispatch:
  Lookup the right function to call in a table (the *vtable*) at runtime.

Will also need simpleClasses (or records) which contain multiple fields but no methods.

# Example: Adding a this argument

```
class Counter{
    int C=0;
    void inc(int x) {C += x;}
    void incTwice(int x) {inc(x); inc(x)}
}
```

$\rightsquigarrow$

```
simpleClass Counter{ int C; }

function void Counter_inc(Counter this, int x){
    this.C += x;}

function void Counter_incTwice(Counter this,int x){
    call Counter_inc(this, x);
    call Counter_inc(this, x)}

function void Counter_ctor(Counter this) {
    C = 0;
    call Object_ctr(); }
```

# Example: Using the Counter class

```
Counter c = new Counter();
c.inc();
```

$\rightsquigarrow$

```
c = allocate(sizeof Counter);
call Counter_ctor(c);
call Counter_inc(c);
```

# Example: Virtual methods through v-tables

```
class Counter{
    int C;
    virtual void inc(int x) {C += x;} }

class FastCounter: Counter{
    override void inc(int x) {C += 2*x;} }

class Runner{
    static void Main(string[] args)
    {
        Counter c = new FastCounter();

        // Should call FastCounter method and get 6
        c.inc(3);
    }
}
```

# Example: Virtual methods through v-tables

```
class Counter{
    int C=0;
    virtual void inc(int x) {C += x;}
}
```

$\rightsquigarrow$

```
simpleClass Counter{
    int C;
    // compiler remembers 0 -> Counter_inc
    function[] vtable = {Counter_inc};
}

function void Counter_inc(Counter this, int x){
    this.C += x;}
```

# Example: Virtual methods through v-tables

```
class FastCounter: Counter{
    override void inc(int x) {C += 2*x;} }
```

⇝

```
simpleClass FastCounter{
    // copied from base class
    int C;
    // compiler remembers 0 -> FastCounter_inc
    function[] vtable = {FastCounter_inc};
}

function void FastCounter_inc(Counter this, int x){
    C += 2*x;}
```

# Example: Virtual methods through v-tables

```
static void Main(string[] args)
{
    Counter c = new FastCounter();

    c.inc(3);
}

//static methods can compile to functions w/o this
function void Runner_Main(string[] args){
    // call FastCounter's default constructor
    c = allocate(sizeof FastCounter);
    call FastCounter_ctor(c);

    // do the virtual call
    function f = c.vtable[0];
    call f (c, 3)
}
```

# Questions

Suppose Counter contained the following (non-virtual) method.

```
void incTwice(int x) {inc(x); inc(x)}
```

How does this compile? What determines whether this eventually calls Counter_inc or FastCounter_inc?

# Other features

**Interfaces** Each interface gets an interface table—analogous to a vtable.

**Reflection** Each object keeps a reference to metadata describing its own class. This is used reflection and checking casts.