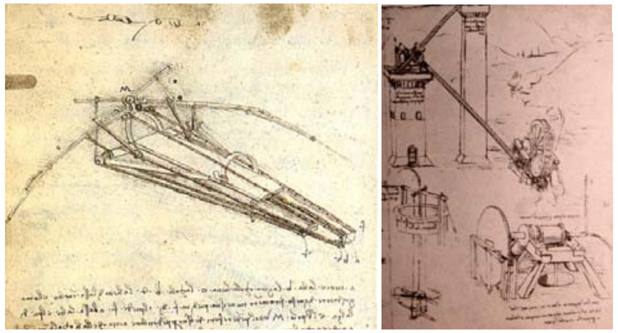
HY540 – Advanced Topics in Programming Language Development



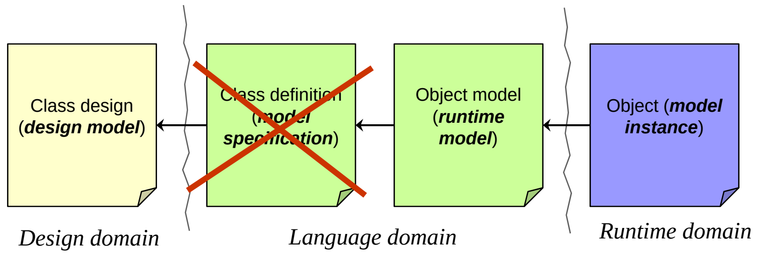
Lecture 2

Untyped classes – introduction (1/4)

- Relate to languages where no class-related data type is supported, even *class* is not a keyword
- However, objects are supported, though not typed by classes but by a generic *object* type
- Object creation is allowed via library functions, special expressions, or all variables are objects

⇨ *The languages of our focus are classless*

Untyped classes – introduction (2/4)


Class design (**design model**)
~~Class definition (**model specification**)~~
Object model (**runtime model**)
Object (**model instance**)
Design domain
Language domain
Runtime domain

Untyped classes – introduction (3/4)

- Classless languages have commonly reference semantics for object values (i.e. no deep assignment, no call by value)
- Replication maybe possible with a reserved library function or operator, typically as shallow copy
- In the vast majority object disposal is automatic (garbage collection)
- They are considered (type) unsafe due to absence of typing (though some language adopt type tags)
- But they are very popular due to simplicity of use and economy of expression (making objects of a particular *kind* is straightforward in code)

Untyped classes – introduction (4/4)

- Next we discuss
  - Language examples
  - Object model
  - Prototypes
  - Duck typing (object protocols)
  - Implementation details

Language examples (1/14)

- *Self language (1/5)*

```
student name: 'john' age: 10 residence: 'New York'.
student name print.
student _AddSlots: (| nationality <- 'Greek'|).
student _AddSlots: (| doWork = ( statements (method) ) |).
self x: ((self x) + add)          // ala C++ this->x += d
[i < thresh] whileTrue: [i: i * 2]
[condition block] whileTrue: block     // sending whileTrue message to block
n even ifTrue: [h: n / 2]              // var: rv is an assignment message for var
condition ifTrue: block                // sending ifTrue message to expression
```

The OO programming model in Self adopts the message passing style **object** *(***message** *[***parameters***])** where a message carries information about the invocation to be done (it is left associative if a parameter is message invocation too).

Language examples (2/14)

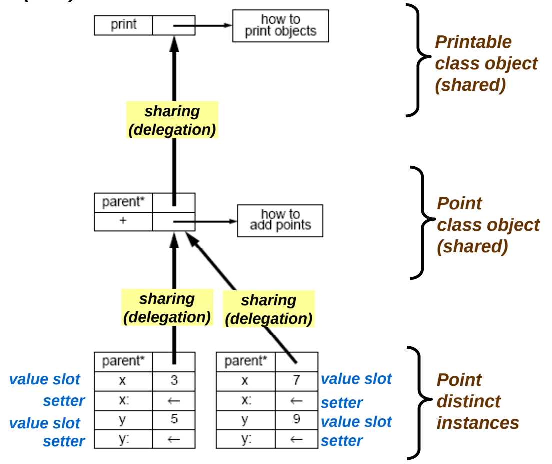
- *Self language (2/5)*
  - The first large-scale object-based language - Sun, 1986-2006, still available via `http://selflanguage.org`
  - Objects as containers of indexed slot values (via ids)
  - Strongly affected by Smalltalk, however, dropping all the class-based stuff and static typing
  - *Everything is an object* (a Smalltalk slogan), even control flow statements
    - Sending *ifTrue*, *whileTrue*, *ifFalse*, *whileFalse* messages (with code as a parameter) to expressions or code blocks (evaluated, not merely executed)
  - Every slot is late bound (always runtime dispatching)
  - Compiled to byte code with a comprehensive virtual machine

Language examples (3/14)

- *Self language (3/5)*
  - The Self language introduced many innovations
    - *traits*, polymorphic inline caching, code of practice for prototype programming, *inheritance by delegation*
  - The idea of traits is Self is that objects looking alike (of the same class) should not copy their methods
  - Instead, they should all share from a special object carrying such common methods representing the class
  - This *class object* is called a *trait* in Self and aims to provide a model of making classes real objects
  - In Self, editing of such a trait slot is possible through a *parent** slot that is a list (meaning can have multiple traits)

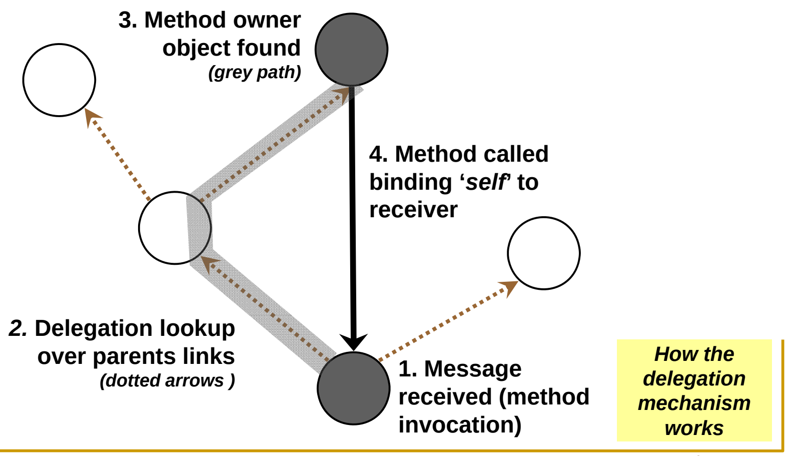
Language examples (4/14)

Self language (4/5)



Language examples (5/14)

Self language (5/5)


3. Method owner object found (grey path)
4. Method called binding 'self' to receiver
2. Delegation lookup over parents links (dotted arrows)
1. Message received (method invocation)
How the delegation mechanism works

Language examples (6/14)

- *Perl language (1/2)*

```
%student = (  #hash (object) constructor expression
      name => 'John',
      age => 10,
      residence => 'New York'
);
$student{nationality} = 'Greek'  #setting a scalar in %student hash
$student{f} = sub (formal arguments here (proto in Perl) ) {  body  }
$student{f}(%student,  actual arguments here)    #method invocation style
```

Language examples (7/14)

- *Perl language (2/2)*
  - The language evolved in a very surprising way with many changes looking mostly arbitrary and not quite thoughtful
  - Originally with an object-based language kernel, lacking support for *self* or *this* in method invocations (as Python does)
  - Adopting type tags (*sigils*) to declare the variable type with declaration by use style (i.e. **array x** is abbreviated to @**x**)
  - Latter (version 5.0) introduced typed classes and explicit class model specification for better typing
    - but still looks that the underlying implementation (interpreter) adopts universally the field dictionary model
  - We return back to Perl (and Python and other languages) latter when discussing inheritance and methods

Language examples (8/14)

- *Java Script (1/3)*

```
// Functions setting this by default call Object() to create a new object on which
// this applies inside the function
function student (name, age, nationality, rename) {
        this.name = name;
        this.age = age;
        this.nationality = nationality;
        this.rename = rename;
}
function student_rename (name) // A method is a function using 'this'
        { this.name = name;}
st = new student("John", 10, "Greek", student_rename);
st.registry_number = 13;        // Can add slots dynamically
address = new Object();         // Makes a default empty object
address.city = "Gotham";        // Can freely add or edit slots
```

Language examples (9/14)

- *Java Script (2/3)*
  - Probably the first popular languages with a clean, truly dynamic, untyped object-based model
  - Entirely typeless, with a generic object model as an extensible dictionary, and a language design pattern for object constructors (object templates)
    - We will revisit this pattern latter
  - Methods are syntactically normal global functions accessing *this*, something that requires naming conventions to avoid name conflicts
    - or else can assign directly inside constructors as anonymous functions
  - Whether a function is called as a constructor or not is statically resolved and affects the way the function works internally
  - Interpreted, but there is a plan for a compiled JavaScript on the DLR (Dynamic Language Runtime, on top of CLR) running under Silver Light 2.0 (.NET)

Language examples (10/14)

- *Java Script (3/3)*

Constructor is any function assigning internally to this slots and encompasses no return statements in its body. We trivially turn constructor invocation as normal functions.

In constructor functions we always add an extra first argument this and the following two extra statements in the function body:

Head: `if (!this) this = new Object();`

Tail (fallback): `return this;`

If in a function call expression an object slot (field) is the function value, then we pass as the first argument to the call the object value

```
ObjectExpr . Id (Args) =>
        a = <ObjectExpr>.Var
        t_f = get(a, <Id>)
        push a
        emit push for every <Args>
        call t_f
```

```
function student_rename (name) {
        if (!this) this = new Object();
        this.name = name;
        return this;
}
st.rename("Lee")  ≡ student_rename(st, "Lee");
rename("Lee")     ≡ student_rename(nil, "Lee");
```

Language examples (11/14)

- *Lua (1/2)*

```
student = { name = "John", age = 10}    // no semicolons needed
student.nationality = "Greek"
student:rename(this, name)  // 'this' not a keyword
        { this.name = name }
function sudent.rename(this, name)  // equivalent to the previous code
        { this.name = name }
student.rename = function(this, name)  // equivalent to the previous code
        { this.name = name }
student:rename("Lee")    // method invocation convention
student.rename(student, "Lee")  // equivalent to previous expression
student..rename("Lee")  // alpha language (CS340) syntactic sugar for methods
```

Lua does not have true methods (they are only syntax), meaning they are not first-class methods

Language examples (12/14)

- *Lua (2/2)*
  - Has a long history (1991 – today), PUC, Rio (Brazil)
  - Was not designed initially with a high-level goal, but evolved by demands, experimentation and good will
  - Not proposing innovative features as such, but combines *simplicity*, good set of features and *performance* (written in C, 18 *KLOC* for the 2009 version), with great *portability*
  - Objects as hash tables model initially inspired by Smalltalk, Perl and JavaScript
  - Compiled to byte code with dedicated instructions for object management
  - Pseudo methods are an important weakness of the language for OOP

Language examples (13/14)

- *Delta (1/3)*

```
student = [
        {.name : “John”}, {.age : 10 },
        {.rename : ( method(name){ self.name = name; } )}
];
student.nationality = “Greek”;
rn = student.rename;          // First class methods, ‘self’ is keyword
rn(“Lee”); ≡ student.rename(“Lee”); // Method values retain ‘self’ binding (owner)
rn.owner = []; // But objects can change their owner on-the-fly
rn(“Bruce”); // Will apply the method to the previous empty object
```

- First-class methods are implemented as pairs of a function address (number) and an owner (self) object (pointer).
- Invoking a method value is handled as a function call with the extra work of implicitly passing the owner as an extra first argument (meaning the user need not do it, as in Lua).
- Similar to AS3 bound methods and Python methods, though in Python the ‘self’ is not reserved but is strongly suggested (…a poor design choice) and the owner is immutable (so can’t move methods around).

Language examples (14/14)

- *Delta (2/3)*

- Callable methods retaining internally ‘self’ object are adequate to allow program very fast generic binders (together with the argument passing support of the language).

```
function bind_1st (f, x) {
    return [
        {.x : x}, { .f : f }, {.rebind: (method(x) { self.x = x;}) },
        ( method(){ return self.f(self.x, |arguments|); } )
    ][0];
}
bind_1st(print, 10)(); ≡ print(10);
bind_1st(bind_1st, print)(20)(); ≡ print(20);
bind_1st((function(f){f();}), bind_1st(print, 30))(); ≡ print(30);
```

$\texttt{bind\_1st(bind\_1st(…bind\_1st(bind\_1st(f, } a_1\texttt{),} a_2\texttt{)…,} a_{n-1}\texttt{),} a_n\texttt{)()} \equiv \texttt{f(}a_1, a_2, …, a_{n-1}, a_n\texttt{)}$

- The previous has theoretical value only, as making a generic *lazy (delayed) evaluation* binder call is very easy (need to store all arguments). *You know binders from C++, right?*

Footnote (1/2)

- *And what binders are?*
  - In general, higher-order functions = *functions returning other functions or accepting functions as arguments*
    - The result of a binder *f* is a function *g* that can be normally called where *f()() ≡ g() ⇔ g ≡ f()*
  - In particular, a binder is a restriction of a function by ‘freezing’ some of its arguments to constant supplied values
  - Thus, to make binders the language should support *function values* which may refer to some sort of *associated memory*
    - The latter is called a *function closure*
    - For methods the closure is the owner object
    - Persistent variables retained across invocations are member variables referred through ***self*** or ***this***

Footnote (2/2)

- Can we support closures in *C*?
  - No by retaining the original syntax and semantics for functions
- Can you do that in *Java*?
  - Still not, the reason is mainly the lack of a syntactic abstraction: should adopt *a.f()* instead of *f()* directly, given method *f* with closure *a* being its owning object
- What about *C++?*
  - Yes through instances *a* of a class implementing *operator()* to have the semantics of *f* with its closure being *a* and invocation allowed with the syntax *a()*
- And in *Lua*?
  - No way, since there are no true methods, neither any facility for treating entire objects as functions (functors) like in C++
- In *Python*?
  - Sure, in two ways: callable method values (as in Delta), and classes defining __call__ method (similar to *operator()* in C++), with object-based support also in Delta / alpha
- In C#?
  - Yes via instance method delegates, similar to method values of Python and Delta (method binders freezing '*this*' value). Delegates are superior to *operator()* overloading
  - ***f' = delegate x.f => f '(1,2) ≡ x.f(1,2)***

Language examples (12/12)

- *Delta (3/3)*
  - Compiled, virtual machine, two generations: 1999-2000 and 2004-present; nothing was done for the language in-between
  - Still built on the original language source, although radically evolved by refactoring and extensibility
  - Design emphasizing *compositionality* and *reusability*
    - continuously in progress, soon on the quest for other Holly Grails (current mission is *meta-programming*, next mission is *concurrency*)
  - Delivery emphasizing support by extensible high-end tools (on which research is also done) – *Sparrow (IDE), Zen* and *Disco* (debuggers)
  - Various features inspired by Lua, Python, Smalltalk, Action Script and Self

Object model (1/7)

- The untyped classless object model
  - Should allow craft any object
  - Should allow add or remove members freely
  - Can't predict how objects evolve, meaning models of static calculations for member access can't apply
- *Member dictionary* model is the only viable alternative
  - A member can carry any value allowed in the language
  - Can't treat methods differently since can't attach methods to classes because classes do not exist
  - So methods are normal members of a *method* value type with a special treatment of ***self***

Object model (2/7)

- The object model is an generalization of the member dictionary where entries are:
  - Dynamically typed and freely updatable
  - Commonly indexed by identifiers with the syntactic sugar of field-like access: `a.x ≡ a["x"]`
  - Late bound so the compiler may tell nothing about `a.x`
- Reference semantics for object values
  - assignment, comparison, argument passing
- Copy or comparison semantics (shallow or deep) offered as library functions
  - Sometimes serialization is offered as a built-in feature

Object model (3/7)

- *Object* is *Dictionary [Key : Value]*
- *Key* commonly allowed as *String* or *Number*
- *Value* is a *disjoint union* type (*discriminated record* or *algebraic data type*) as follows:

disjoint union ***value*** {
    tag is $\{T_1,\ldots,T_n\}$
    variant is $S_1+\ldots+S_n$ for distinct type domains S_i
}
domain(***value***) = (*tag* T_1 | *variant* $\in S_1$) $\cup \ldots \cup$ (*tag* T_n | *variant* $\in S_n$)

Object model (4/7)

- In most languages disjoint unions are directly supported or else they have to be emulated using objects
- Theoretically the following compile-time operators are required for disjoint unions:
  - *construction* of a disjoint-union *value* from its *tag* and *variant*
  - *tag test* to determine from which S_i the variant value was chosen
  - *projection* to recover the variant value from S_i

Object model (5/7)

- Implementation (untyped objects)
  - Basic operators
    - *object new()*
    - *value set (object, key, value)*
    - *value get (object, key)*
  - Message passing (depends on the language)
    - *value send (object, message)* | *message = list [key, value]*
    - *message make_message ([key,value]∗)*
  - Copy and comparison (preferable is the lib style)
    - *object deep_copy* (*object*) *bool deep_cmp* (*object*, *object*)
    - *object shallow_copy* (*object*) *bool shallow_cmp* (*object*, *object*)

Object model (6/7)

- Choice of syntax to reflect underlying semantics
  - construction styles
    - object construction separate from field editing
      - `a = new object` *// via a new meta operator*
      - `a = []` *// via object constructor expression*
      - `a = new_object()` *// via a library function*
    - object construction embedding field definition
      - `a = new object (name:"John")`
      - `a = [  name = "John" ]`
      - `a = new_object(new_field("name", "John"))`
    - method definition
      - similarly separate from, or embedded with, object construction

Object model (7/7)

- Choice of syntax to reflect underlying semantics
  - *Field access*
    - dot and index operators
      - `a.x = y;  a[i] = y; y = a.x; y = a[i];`
      - optionally untyped support for properties, as in Delta

```
@x { @set ( method(x) {@x = x;} )
     @get ( method(){ return @x; } )
}
```

    - message passing style
      - `a :set x y.      y = a :get x.`
  - *Invocation (functorization)*
    - `a(1,2)` *// call operator style*
    - `a :call 1 2.` *// message passing style*
    - `a :get x :call 3 4 :set y 30.` ≡ `a.x(3,4).y = 30;`

Prototypes (1/7)

- A prototype is a special object that
  - plays the role of producing objects of a respective class
  - carries class-based behavior (functions) and state (data) sharable by all respective objects
- The notion of a class is used here only as a concept of a design domain having no language-specific mapping
- The term prototype came from the assumption that prototypes are just prototypical objects from which new objects are produced by replication

Prototypes (2/7)

- Question (latter)
  - *if a prototype **P** produces objects of class **A** then is the class of object **P** also **A**?*
- Prototypes make designers think in an object-first fashion and then solve the problem of how objects are created
  - The notion of a class is implied latter at the design domain as a model of objects
- ☝ *Prototypes are a design pattern in classless languages to emulate classes as object factories*

Prototypes (3/7)

- Think of an object, say a point, generally *Point,* and define its state and behavior *x, y, move*
- The language should allow to either:
  - implement a function that produces such objects
    - factory method
  - define an object with a method to produce such objects
    - factory object

```
function Point (x,y) { // JavaScript, factory method
   this.x = x; this.y = y;
   this.move = function (dx,dy) { this.x+=dx; this.y+=dy; };
}
pt = new Point(10,20); // Special syntax, not as normal function
function Point(x,y) { // Delta, factory method
   return [    {.x : x}, {.y : y},
               method move(dx,dy){ self.x += dx; self.y += dy;}
          ];
}
pt = Point(10, 20);   // Uniform syntax, as any other function
```

Prototypes (4/7)

- The differentiation on JavaScript amongst prototype functions (i.e. constructors) from normal functions relates to program interpretation:
  - **`new Point(x,y)` ⇒ `Point(new Object(), x, y)`**
  - We presented earlier an interpretation pattern avoiding this separation, making constructors first-class values
  - However, as we see there is reason JavaScript distinguishes such functions, since they are treated implicitly like *classes*
- Factory methods make prototypes look also like design concepts, since a prototype is never referenceable
- But factory objects turn prototypes to first-class values, though requiring more definitions
- ☝ *Prototypes are first-class values allowing emulate in the language first-class classes by sharing (delegation)*

Prototypes (5/7)

```
function Point() {              // Method returning factory object
   if (isundefined(static proto))
       proto = [
              {.x : 0}, {.y : 0},
              method move (dx,dy){ self.x += dx; self.y += dy;},
              method new (x, y) {
                         local a = tabcopy(self);
                         if (arguments[0]) a.x = x;
                         if (arguments[1]) a.y = y;
                         return a;
                       }
       ];
       return proto;
}
pt = Point().new(10,20);        // Construction via prototype accessor
proto = Point();                // Prototype is first class value
pt2 = proto.new();              // Possible to abstract over prototypes
```

Delta (example shows a prototype only as an object factory)

Prototype accessor pattern: `function` *<Class name>* `(…) {…}`
Instance creation pattern: *<Class name>*`().new();`

Prototypes (6/7)

- If factory methods are supported they can emulate classes only regarding the needs of object construction
- In this case the constructor plays the role of a prototype value if construction is the only task required in the program
- To cope with variable arguments (alternative construction signatures) the language should support:
  - *overloading*: requires constructor functions to be distinguished in the language by signatures
    - for this to make sense typed signatures are always needed
  - *argument handling*: to dispatch internally to the constructor according to dynamically extracted signature
    - probable choice, transfer the responsibility to the programmer

Prototypes (7/7)

- Prototypes may have no special meaning in the language, meaning they are like any other object
- So their distinction as prototypes or not is only a matter of program design
  - if they are used for object construction and behavior sharing (as in *Self*) then their role is that of a prototype
- In some languages prototypes are explicit
  - associated to objects
  - being sharable items
  - editable during runtime
  - allowing extend the behavior of existing objects
- *Firstly we show that in all cases prototypes are singletons, meaning they are not prototypical objects*

Prototypes as singletons (1/3)

- Let prototype ***P*** producing *class(**A**)* objects. We prove that *class(P)* $\neq$ *A*
- *Assume **P** is prototypical, meaning **P** is an **A** object*
- $\exists$ *object **A x** where **x** $\neq$ **P** $\Leftrightarrow$ x is created through **P***
- $\exists$ *only one **A** object **x** in the program $\Rightarrow$ x=P*
- *no A object exists in the program $\Leftrightarrow$ no A object can be made*
- *can produce **A** objects $\Leftrightarrow$ the **P** object exists (retained)*

Prototypes as singletons (2/3)

- $\forall$ *C class C.invariant() $\Rightarrow$ any **C** object **a** exists $\Leftrightarrow$ the application state reached a point implying that **a** had to be constructed*
- *given any object **x** satisfying its class invariant any copy **x'** is not guaranteed to satisfy the invariant too*
- *if classes should retain an object during the entire runtime then this may contradict their invariant*
- ***P** may break **A** class invariant $\Rightarrow$ class(P) $\neq$ A*

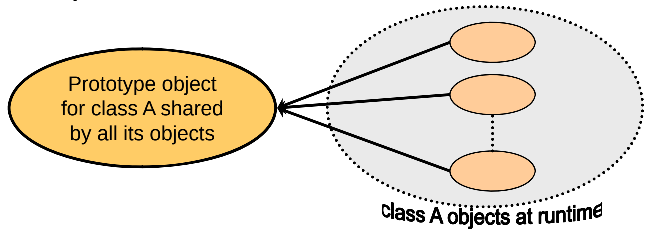
Prototypes as singletons (3/3)

- The proof shows prototypes to be the wrong metaphor, although they are widely used. *Class objects* is a better term.
- With prototypes we should always distinguish amongst two different classes
  - The class concerning the produced objects, say ***A***
  - The class concerning the prototype itself, say ***A_IMPL***
  - ☞ *The behavior of a prototype is different from the behavior of the produced objects, meaning they semantically map to different classes*
- Reflecting our remark, given any class ***A*** and object **x** the following hold:
  - ***x*** produces ***A*** objects $\Leftrightarrow$ ***x*** is an ***A_IMPL*** object
  - ***x*** is an ***A_IMPL*** object $\Leftrightarrow$ ***x*** is an ***A*** prototype

☞ *In analogy to class-based languages where there is only one implementation of a given class, prototypes are by definition singletons*

Prototype details (1/10)

- *A prototype represents the default behavior for a concept, and new objects can re-use part of the knowledge stored in the prototype by saying how the new object differs from the prototype (Lieberman, 1986)*
- In some languages prototypes are part of the language with sharing semantics, meaning they are not merely object factories
- Prototype is a special object shared by all objects of a class, offering an extra dynamic lookup space for members not found originally in the object


Prototype object for class A shared by all its objects
class A objects at runtime

Prototype details (2/10)

```
emit_field_get (Class* c, Expr* a, char* x, Expr* lv) { // compile time a.x
        FieldPtr* f = c->GetFieldPtr(x);
        if (f) {  // field statically-defined within its class
                emit t1 = mem[a->GetAddress() + f->GetOffset()]        t1 = a[x]
                emit mem[lv->GetAddress()] = t1                         lv = t1
        }
        else {    // allow field to be looked-up dynamically within the prototype object
                emit t1 = mem[a->GetAddress()  + a->PrototypeOffset()]  t1 = a.proto
                emit param x, t1                                        push x, t1
                emit call _getfield_                                    call _getfield_
                emit getretval t2                                       t2 = retval
                emit mem[lv->GetAddress()] = t2                         lv = t2
        }
}                                                  class-based languages (e.g. AS3)
bool field_get (Object& a, string& x, Value* at) {    // runtime time a.x
        return   straight_get(a, x, at) || straight_get(a.prototype, x, at);
}                                                  object-based languages (e.g. JS)
```

Prototype details (3/10)

- *Action Script 3.0 (1/2)*
  - Although mainly a class-based language, has object-based roots with a preference by programmers to deploy mostly the dynamic features
  - It supports prototypes as objects shared by all instances of a class, meaning extending prototypes practically extends the class itself
  - Currently supported only for objects of dynamic or static classes, meaning code generation is that of the class-based language family

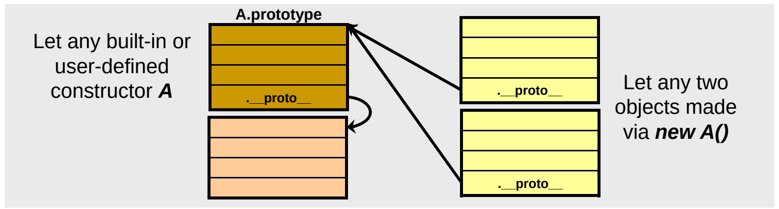
Prototype details (4/10)

- *Action Script 3.0* (2/2)

```
var student1 : Object = {  // can make objects without classes
      name : "John", age: 256
};
public class Student {
     public var name:String;
     public var age: Number;
     public function Student (iName:String, iAge: Number) // ctor
       { name = iName; age = iAge; }
     public function print():void
       { trace("Name:" + name); }
}
var student2 = new Student("Lee", 1024);
var boundMethod:Function = student2.print(); // methods with closures
boundMethod(); // like student2.foo()
Student.prototype.rename = function (newName: String):void {
      this.name = newName;
}
student2.rename("Lee"); // calls student2.prototype.rename
```

Prototype details (5/10)

- *JavaScript (1/3)*
  - In JavaScript one prototype entry is reserved for every constructor function
    - thus JavaScript implicitly treats a constructor as a class
  - While every object has a reference to its prototype, initially set to be the prototype of its constructor function (if made through a constructor, else nil)
    - **A.prototype** can be changed (all objects using it will be affected)
      - internally it is A.prototype.__proto__ which actually changes
    - **x.__proto__** can be changed (only x is affected)


Let any built-in or user-defined constructor *A*
A.prototype
.__proto__
.__proto__
.__proto__
Let any two objects made via *new A()*

Prototype details (6/10)

▪JavaScript (2/3)

```
var circle = { radius:10, cx:0, cy:0 };
var myCircle = {};
myCircle.__proto__ = circle;
```

```
function Circle(r, x, y) {
  this.radius = r;
  this.cx = x;
  this.cy = y;
};
Circle.prototype.pi = 3.141628;
Circle.prototype.area = function()
   { return this.pi * this.radius * this.radius; };
Circle.prototype.perimeter = function()
   { return 2 * this.pi * this.radius; };
circle = new Circle(10, 0, 0);
circle.area();
circle.perimeter();
```

Prototype details (7/10)

▪JavaScript (3/3)

```
function ColoredCircle(r, x, y, c) {
  this.radius = r;
  this.cx = x;
  this.cy = y;
  this.color = c;
};
ColoredCircle.prototype = new Circle(0,0,0);
coloredCircle = new ColoredCircle(20, 0, 0,'red');
coloredCircle.area();
coloredCircle.perimeter();
```

```
Object.prototype.trace = function()
   { alert(this); };
'hello, world'.trace();
(coloredCircle).trace();
```

The last object in the prototype chain of every object is `Object.prototype`

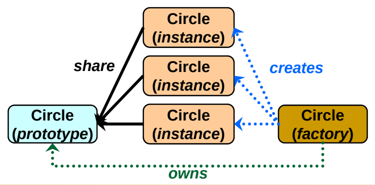
Prototype details (8/10)

- *Delta (1/3)*
  - To support prototypes as class objects shared by all respective instances, the model of the ***Self*** language has been implemented:
    - Although it supports prototypes is has no reserved syntax and semantics for prototypes
    - Some objects can be shared by other objects through the *delegation* mechanism
    - It is up to the programmer to let the shared object populate state and behavior that imply a prototype and link new instances to this shared object
    - Multiple delegates are allowed, cycles are detected on lookup

Prototype details (9/10)

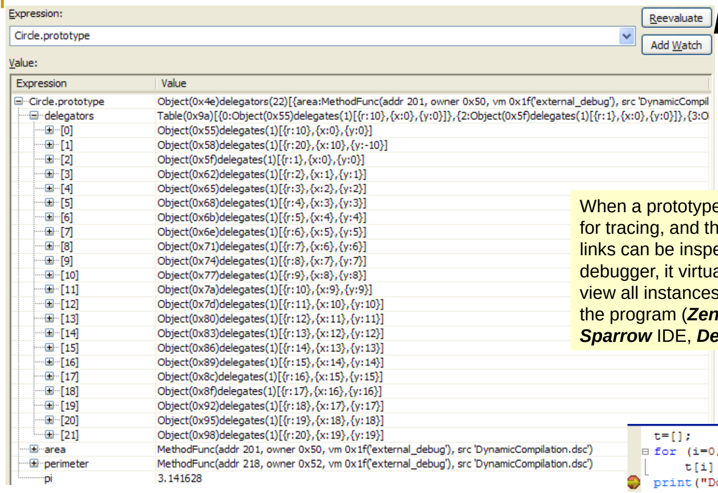
Delta (2/3)

```
Circle = [     ←Circle is an object that plays the role of a factory
    { .prototype : [] },     ←Circle stores locally the object shared by all its instances (i.e. prototype)
    method @operator()(r, x, y) {     ←Circle behaves as a functor emulating the constructor method
        local c = [ {.r:r}, {.x:x }, {.y:y} ];
        delegate(c, self.prototype);          ←Delegation applied upon construction
        return c;
    }
];

Circle.prototype.pi          = 3.141628;
Circle.prototype.area        = [ method(){ return self.pi*self.r*self.r; } ][0];
Circle.prototype.perimeter   = [ method(){ return 2*self.pi*self.r; } ][0];

c1 = Circle(10, 0, 0);
c2 = Circle(20, 10, -10);
print(c1.area(), nl, c2.perimeter(), nl);
```



Prototype details (10/10)

Delta (3/3)



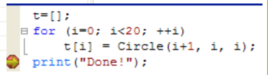
Expression:

Circle.prototype

Value:

Expression	Value
Circle.prototype	Object(0x4e)delegators(22)[{area:MethodFunc(addr 201, owner 0x50, vm 0x1f('external_debug'), src 'DynamicCompil
delegators	Table(0x9a)[{0:Object(0x55)delegates(1)[{r:10},{x:0},{y:0}]},{2:Object(0x5f)delegates(1)[{r:1},{x:0},{y:0}]},{3:O
[0]	Object(0x55)delegates(1)[{r:10},{x:0},{y:0}]
[1]	Object(0x58)delegates(1)[{r:20},{x:10},{y:-10}]
[2]	Object(0x5f)delegates(1)[{r:1},{x:0},{y:0}]
[3]	Object(0x62)delegates(1)[{r:2},{x:1},{y:1}]
[4]	Object(0x65)delegates(1)[{r:3},{x:2},{y:2}]
[5]	Object(0x68)delegates(1)[{r:4},{x:3},{y:3}]
[6]	Object(0x6b)delegates(1)[{r:5},{x:4},{y:4}]
[7]	Object(0x6e)delegates(1)[{r:6},{x:5},{y:5}]
[8]	Object(0x71)delegates(1)[{r:7},{x:6},{y:6}]
[9]	Object(0x74)delegates(1)[{r:8},{x:7},{y:7}]
[10]	Object(0x77)delegates(1)[{r:9},{x:8},{y:8}]
[11]	Object(0x7a)delegates(1)[{r:10},{x:9},{y:9}]
[12]	Object(0x7d)delegates(1)[{r:11},{x:10},{y:10}]
[13]	Object(0x80)delegates(1)[{r:12},{x:11},{y:11}]
[14]	Object(0x83)delegates(1)[{r:13},{x:12},{y:12}]
[15]	Object(0x86)delegates(1)[{r:14},{x:13},{y:13}]
[16]	Object(0x89)delegates(1)[{r:15},{x:14},{y:14}]
[17]	Object(0x8c)delegates(1)[{r:16},{x:15},{y:15}]
[18]	Object(0x8f)delegates(1)[{r:17},{x:16},{y:16}]
[19]	Object(0x92)delegates(1)[{r:18},{x:17},{y:17}]
[20]	Object(0x95)delegates(1)[{r:19},{x:18},{y:18}]
[21]	Object(0x98)delegates(1)[{r:20},{x:19},{y:19}]
area	MethodFunc(addr 201, owner 0x50, vm 0x1f('external_debug'), src 'DynamicCompilation.dsc')
perimeter	MethodFunc(addr 218, owner 0x52, vm 0x1f('external_debug'), src 'DynamicCompilation.dsc')
pi	3.141628

When a prototype is accessible for tracing, and the delegation links can be inspected by the debugger, it virtually enables view all instances of a class in the program (***Zen*** debugger, in ***Sparrow*** IDE, ***Delta*** language).



```
t=[];
for (i=0; i<20; ++i)
    t[i] = Circle(i+1, i, i);
print("Done!");
```

Duck typing (object protocols) (1/6)

- It is an approach to dynamic typing where the assumed type of an object does not depend on typing relationships
- Instead, it is defined by its current external view in terms of the properties and methods that it offers to the outside world
- In other words, typing is defined by the capability to use it with a certain deployment image in a specific context
- *Meaning we are concerned with just those aspects of an object that are used, rather than with the precise type of the object itself*

Duck typing (object protocols) (2/6)

- The terminology comes from the, not very well known, duck test of inductive reasoning
- ☺ *If it looks like a duck, swims like a duck and quacks like a duck, then it probably is a duck*
- Thus, duck typing allows polymorphism without inheritance
  - Meaning an object is eligible to an operation once it displays the required properties and methods
- Duck typing is a bad term anyway, as it was far earlier defined with a theoretical foundation by the notion of *objector protocols*
  - *[Abadi & Cardelli, A Theory of Objects]*

Duck typing (object protocols) (3/6)

- In general, an object protocol is a view of an object, like: it has property ***x*** and method ***f(a,b)***
- Notably, everything mentioned in a protocol should obey further protocol constraints (i.e., what is the protocol of ***x***)
- Object protocols allow generic functions (or polymorphic ones) without relying on subtyping but on requirements expressed about
  - Genericity should rely on what we expect from an object to offer rather than on its specific designated data type
- In classless languages the only way to define protocols is as documented design constraints
  - and the only way to ensure code will not fail, since can't test statically, is via unit testing

Duck typing (object protocols) (4/6)

Class-based (!) sample: C++

Requirements (object protocols, or concept in C++ forthcoming standard):

```
        Iterator<I>,
        Addable<value_type>,
        Assignable<value_type>,
        CopyConstructible<value_type>

template<typename I>
typename iterator_traits<T>::value_type
sum(I start, I end, typename iterator_traits<T>::value_type init) {
  for (I current = start; current != end; ++current)
    init = init + *current;
  return init;
}
```

Duck typing (object protocols) (5/6)

Classless sample: Delta

```
function Intersection (a, b) {
        local box = [ {.x, .y, .w, .h : 0 } ];
        if (a.x + a.w <= b.x or b.x + b.w <= a.x) return box;
        if (a.y + a.h <= b.y or b.y + b.h <= a.y) return box;
        box.x = max(a.x, a.y);
        box.y = max(a.y, b.y);
        box.w = (b.x < a.x ?
                       min(b.w – (a.x-b.x), a.w) :
                       min(a.w - (b.x-a.x), b.w));
        box.h = (b.y < a.y ?
                       min(b.h – (a.y-b.y), a.h) :
                       min(a.h - (b.y-a.y), b.h));
        return box;
}
b1 = Intersection(aSprite, aRect);
b2 = Intersection(anImage, aWindow);
```

required members:
x, y, w, h
required operators on members:
arithmetic, relational

Duck typing (object protocols) (6/6)

- Protocols relate to *interfaces* of class-based languages
  - static commitment of a class in implementing the number of methods appearing in the interface
  - in fact, some languages use the term *protocol* for interface protocols (like Objective C)
- *Protocol-based polymorphism* is when a generic function is written whose object parameters are constrained by protocol (or interface) compliance
  - it is more liberate to subtype polymorphism, i.e. when argument types should be derived from a supertype
  - some languages impose protocol compliance to be expressed during class definition (Java and C# interfaces)
  - others may allow express how classes can adapt to protocols independently of their original definitions (C++ concepts)

Implementation details (1/3)

Common instruction set for objects

`new_object`	Dynamic object creation (construction)
`new_object_by_constructor`	Invocation of a constructor to make an object
`call_method`	Invocation of a method (using name)
`new_prototype`	Creation of a new prototype associated to a class
`get_prototype`	Extraction of an prototype from an object
`set_member / get_member`	Member access
`set_attribute / get_attribute`	Attribute / property member access
`delete_member`	Removal of members
`has_member`	Queries member presence
`add_delegate`	In case sharing is done by delegates.
`remove_delegate`	Can remove a delegate on the fly
`get_delegates`	Getting list of shared objects

Implementation details (2/3)

Dynamic optimization (1/2)

Good optimization for classless languages is a real hell. As everything is late bound, and everything can change due to a call, can't tell statically what is constant and what not. Consider the following example:

```
for (…) {
      p.f(); // Is p.f a loop invariant?
      q.g(); // Is q.g invariant as well?
}
```

The code generation situation is as follows:

```
N:    get_member p "f" _t
      call _t
```

or at the best case (depending on the VM instruction set):

```
N:    call_method p "f"
```

But both require a lookup (hashing) in **p** with index **"f"** at every call.

Implementation details (3/3)

Dynamic optimization (2/2)

We may try a dynamic optimization scheme as follows:

- After the first invocation we set the method address directly on the target instruction.
- Store an update function which restores the original instruction if ***p.f*** or ***p*** change (meaning we return to ***p.f*** calculation only when ***f*** or ***p*** change).
- Automatically unregister the update function from ***p*** and ***p.f*** when exiting from the scope of ***p*** and restore the original instruction.

The previous should be applied for loops, and may work, but complicates heavily the Virtual Machine implementation. This means Virtual Machine vendors should perform such dynamic optimizations, something that was traditionally the job of compiler writers.

Lessons learned (1/2)

- Member dictionary is the only viable object model
- Object management support means provide methods
  - as syntactic abstractions (sugar) if they are invoked always with an object - *caller is explicit*
    - makes the implementation easier, but still allows moving methods around since methods are just global functions
    - should avoid commit with a ***self*** or ***this*** keyword since they are pseudo methods
  - as values carrying the object owner if they can be invoked standalone - *caller is implicit*
    - this way methods become functions with closure (*closure*= *owner*) enabling program function generators in the language
    - allow extract and mutate the owner object: the implementation is easy, while increasing the programming power of the language

Lessons learned (2/2)

- Prototypes allow emulate classes as shared mutable objects carrying
  - *class functionality* - as *methods* in the prototype
  - *static class* data - as *fields* in the prototype
- Sharing is possible by implementing a delegation mechanism
  - the language may further expose prototypes as first-class concepts, but delegation is always the engine
  - if delegation links can be inspected via the debugger it is analogous to enabling track down all instances of a class
- Object management functions are included as high-level instructions of the virtual machine
  - In some cases you offer some key language features as library functions that you transform latter to instructions
  - This makes execution faster, but may require extra syntax while downgrades such features from first-class values (functions)