Object Model

Object Oriented Programming

236703

Winter 2015-6
Class Representation In Memory

A class is an abstract entity, so why should it be represented in the runtime environment?

Answer #1: Dynamic Binding

- The actual method bound to a message is determined at run time based on the type (class) of the receiver
- There must be a link between the receiver and its methods
- Since links are the same for all objects of the same class, it makes sense to share the representation
Dynamic Binding – C++ Style

// class B declares virtual f(), g() & h()
B* b1 = GetSomeDerived(); // return new D1
B* b2 = GetSomeDerived(); // return new D1
B* b3 = GetSomeDerived(); // return new D2
Class Representation In Memory

- A class is an abstract entity, so why should it be represented in the runtime environment?
- **Answer #2: Run-time type information**
  - Being able to recognize an object’s type is sometimes useful:
    - C++ – is a downcast safe?
    - Java – are we comparing objects of the same type?
    - Squeak – are those arguments of the expected type?
  - Better rely on polymorphism when possible, though
public boolean equals(Object obj) {
    if (obj instanceof MyClass) {
        // cast and compare
    }
}

obj

obj class

MyClass

Must this be an object?
Class Representation In Memory

A class is an abstract entity, so why should it be represented in the runtime environment?

Answer #3: Reflection

- The ability of the program to examine and possibly modify itself
  - Serialization, cloning, code annotation, garbage collection etc.
Object Serialization – Squeak Style

storeOn: aStream

...  
1 to: (self class instSize) do: [
    :i |
    aStream
    nextPutAll: ' instVarAt: ';
    store: i;
    nextPutAll: ' put: '; 
    store: (self instVarAt: i);
...

Must this be an object?
A Class as an Object?

- There are various reasons for classes to have some memory representation
- Why not as an object?
- Answer: Let's go for it!
  - Each object $o$ has a pointer to its class object $c$, which is the object representing the class to which $o$ belongs.
Meta-classes

- **Meta**: Beyond; transcending; more comprehensive
- **Class**: generates instances
- **Metaclass**: a class whose instances are classes
- **Terminal Instance**: cannot be instantiated further

---

**Objects**

- **Classes**: Instantiable objects
- **Metaclasses**: Instances are classes
- **Terminal Instances**: Non-Instantiable objects ("regular" objects)
Taxonomy of Metaclass Systems

1-Level System: Objects only
- Objects describe themselves
- No classes: objects are “instantiated” or “inherited” from other objects
  - A.K.A. *prototype inheritance*
- Example: Self, JavaScript

2-Level System: Objects, Classes
- Objects are described by classes
- Classes do not exist in run-time. Not first-class objects:
  - Not instances of classes
  - Not created by constructors
  - Cannot receive messages
- Examples: Eiffel, C++
Taxonomy of Metaclass Systems

- **3-Level System: Objects, Classes, One Metaclass**
  - Objects described by classes
  - Classes described by the metaclass
  - The metaclass describes itself
  - Examples: Little Smalltalk, Java, C#

- **4/5-Level System: Objects, Classes, Metaclasses, ...**
  - Objects described by classes
  - Classes described by matching metaclasses
  - Metaclasses described by meta-metaclass/es
  - Example: Squeak
The 1 Level System

- In a 1-level system, an object is essentially a map from names (strings) to either values or methods
  - Very flexible, but static type checking is hard
- No distinction between objects and classes
  - Every object is the class of itself
- Instantiation of a new object can’t be done using a class – every object is its own class! So, we can:
  - Clone an existing object
  - Create an empty object (“ex nihilo”)
    - Possibly initialize using an object literal: obj = {a: 1, b: 2}
The 1 Level System

Object Prototype

Print <method>

Book Prototype

Prototype

Borrow <method>

Prototype

Title

Oliver Twist

Author

Dickens

Prototype

1984

Title: <method>

Author: Orwell

Clone

Prototype

Clone

Prototype

Title: ???

Author: ???
Test case: JavaScript

▶ Prototype inheritance: run-time linking instead of compile time subclassing
▶ Lookup relies on delegation – traverse prototype objects

```javascript
var obj1 = {};  // empty object, “inherits” Object
obj1.m = 1;     // create and set field m
var obj2 = {f: function(){}};  // object with method
obj2.__proto__ = obj1;         // obj2 now "inherits" obj1
console.log(obj2.m);           // print 1
obj2.m = 2;                    // new field shadows obj1.m
console.log(obj1.m);           // still prints 1
```
1 Level System – Pros and Cons

Pros:
- Extremely flexible
  - Delegation can replace inheritance, but not the other way around
  - As *Object Oriented* as it gets... (literally)

Cons:
- Performance penalty on member lookup
  - Associative arrays, linked “sub-objects”
- No compile time validations and optimizations
- Not very common
The 2 Level System

- Objects are instances of classes
- Classes are compile time creatures
  - No run time representation
- Minimal run time overhead
  - Zero overhead ("supermarket") principle
  - But objects’ types cannot always be determined
    - On some cases, run-time type information is available
- No class (static) members
  - More on that when we discuss the 3 level system
Test case: C++

class C {
public:
    int i;
    void f();
};

int main() {
    C* c = new C();
    c->i = 7;
    c->f();
}

Note: Polymorphic types have a vtable and RTTI, which can be considered a 2.5 level
The 3 Level System

- Objects are instances of classes
- Classes are also objects
- All classes are instances of one *metaclass*
  - In some languages, the metaclass class is named *Class*...
- Simple and elegant model: courtesy of Smalltalk-76, the first language to introduce the metaclass concept
  - Later adopted by Little-Smalltalk
A user-defined class can not affect the structure of its class object

- It can, however, affect the class object’s state
  - Can it affect the meta-class structure or state?
The Metaclass Class (3 levels)

- The only metaclass in the system
- Instance of itself
  - Avoids the infinite regress of the instantiation relationship
- Inherits from Object
- Holds behavior common to all class objects:
  - How to add methods
  - How to instantiate a class
- Holds structure common to all class objects:
  - Super class
  - Instance variables
  - List of methods
- Uniform behavior and structure of all classes:
  - Different classes cannot have different new methods
  - The new defined in class `Class` calls the constructor defined in the object’s class
The 3 Level System

Object

Instance of
Subclass of
Fields
Methods

Class

Instance of
Subclass of
Fields
Methods

Oliver Twist
Instance of
Author: Dickens
Year: 1838

Novel

Instance of
Subclass of
Fields
Methods

Print
...

Author
Year

Set author
Get author
Set year
Get year

Subclass of
Fields
Methods

New
Add subclass
...

Instance of

Class Members in 3 Level Systems

- Object members are defined in the object’s class
- Similarly, class members are defined in the metaclass. But there is only one, hence shared among all classes!
- Static binding allows emulation
  - Static members are stored in some global space
  - The class is merely a namespace
  - e.g., Math.abs(x)
  - Java, C#, C++ etc.
The 4 Level System

- Each object is an instance of a class
- Each class is an instance of a metaclass
  - May have its own singleton metaclass
  - May share a metaclass with other classes
- Each metaclass is an instance of a meta-metaclass
- One meta-metaclass:
  - Instance of itself
  - Inherits from Object (or Class)
- Examples: (with minor variations)
  - LOOPS: Lisp Object and data Oriented Programming System
  - ObjVlisp: OBJect Virtual extension of LISP
    - A model more than a real language
A 5 Level System

Three Kinds of Entities

1. Objects – as before
2. Classes – as before
3. Metaclasses – the object describing class X is an instance of a singleton metaclass “X class”
   - Integer → Integer class
   - Class → Class class
   - Metaclass → Metaclass Class

The metaclass “Integer class” is also the result of evaluating the Smalltalk expression `Integer class`
Inheritance in the 5 Level System

- **Metaclasses hierarchy:**
  - X inherits from Y ⇔ “X class” inherits from “Y class”

- **A metaclass object “X class”:**
  - is an instance of “Metaclass”
  - inherits from “Class”

- **Class:**
  - an abstract class
  - parent of all “X class” objects
  - “Breaks” the model...
Infinite Object-Class Regression

- 3-KRS: an infinite number of levels system
  - Objects $\rightarrow$ Classes $\rightarrow$ Metaclasses $\rightarrow$ Meta-metaclasses $\rightarrow$ Meta-meta-metaclasses $\rightarrow$ ...

- For each object $X$, there is a meta-object $\uparrow X$
  - Meta-object $\approx$ Class
    - Class = object abstraction; meta-object = object reflection
    - Lazy creation of meta-objects enables the system to work in a finite world

- In real world: Ruby’s singleton class!