Class-based languages

Rafał Łasocha
ii@UWr, 5.03.2014

Definitions

- there are many definitions of class, depends on language
- the most popular: classes are describing objects, with specified fields and methods (full code)
- attributes = fields + methods
- object = instance of class
- `InstanceTypeOf(foo)` = type of **class** Foo (note that class ≠ type)
- Special keyword, `self` (`this`), refers to current object

Real world

- Simula 67 – first language with classes, objects, inheritance
- Smalltalk
  - canonical example of class-based language, influential to many other popular today (c++, obj-c, c#, java, python, ruby...), it's pure oo language
  - Implications of the fact, that class is an object
- currently, there are huge amount of class-based languages

Smalltalk few examples

```
3 + 4                     → send ’+ 4’ to 3
20 factorial              → send factorial to 20
20 factorial class        → LargePositiveInteger
Object new class          → Object
MyStudent := Customer new

Object subclass: #TrainSet
     instanceVariableNames: 'engine cars'
     classVariableNames: ' '
     poolDictionaries: ' '
```

Storage models

- naive (attribute record)
- records with fields and references to method suites

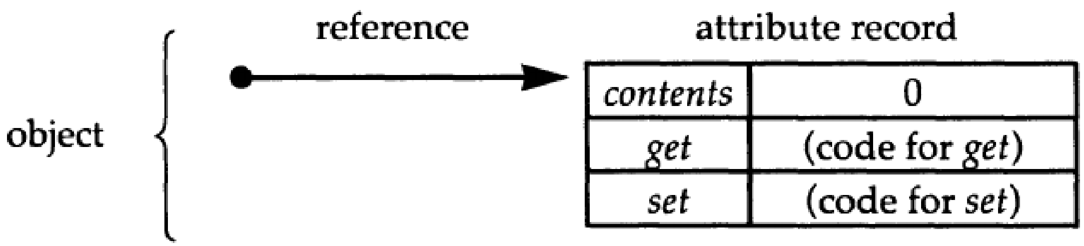


Figure 2-1. Naive storage model.

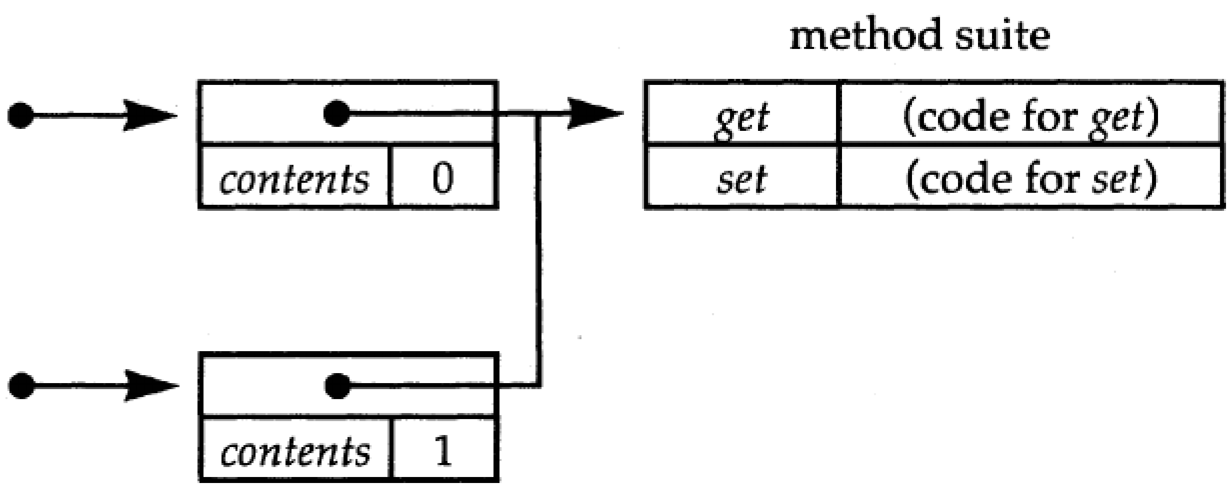


Figure 2-2. Method suites.

Method lookup

- may be performed during compilation or runtime
- method suites organized in trees or directed graphs (multiple inheritance)
- method suites assigned to more specific subclasses, delegate calls to its' parents
- implementation of lookup should include lookup of class variables
- some languages (beta, ruby) allows to assign methods to objects

Example (ruby):

```
bar = Foo.new
def bar.x
   ...
end
```

Subclasses, inheritance

- „c' inherits from c” ⇔ „c' is a subclass of c”
- fields from superclass (parent) are present in subclass
- methods from superclass are present in subclass, but may be overriden
- some languages (simula, c++) allow to override only those methods, which have been declared as virtual
- `self` keyword – refers to a subclass
- multiple inheritance – when class can inherit from more than one class (notice that conflicts are possible and `super` may be tricky)

```
class cell is
    var contents: Integer := 0;
    method get() : Integer is
            return self.contents;
    end;
    method set(n: Integer) is
            self.contents := n;
    end;
end;
```

```
subclass reCell of cell is
    var backup: Integer := 0;
    override set(n: Integer) is
            self.backup := self.contents;
            super.set(n);
    end;
    method restore() is
            self.contents := self.backup;
    end;
end;
```

```
var myCell : InstanceTypeOf(cell) := new cell;
```

Storage model (with inheritance)

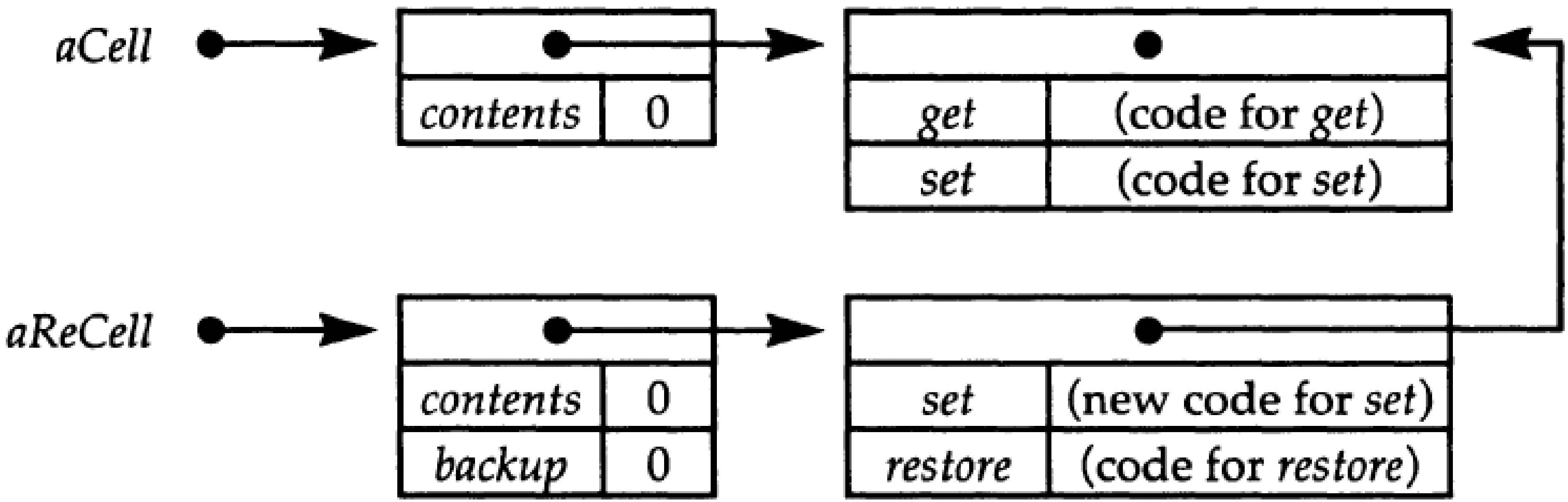


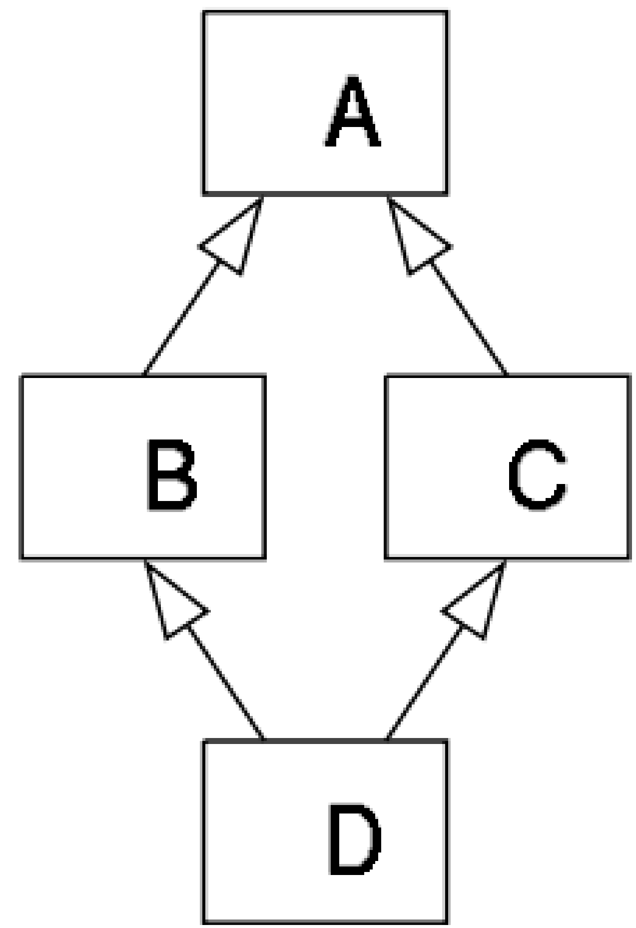
Figure 2-3. Hierarchical method suites.

Super keyword

- makes possible running original implementation of redefined methods
- many different implementations
  - `super` as a „parent self"
  - `super` as a parent method (CLOS: `call-next-method`)
  - none; instead of `super` keyword, some languages provides syntax to access methods directly from specific class
  - instead of specifying where to call parent method, specify where to call subclass' method (`inner` keyword in Beta), so method extension rather than method overriding
  - exists even composition of `super` and `inner`, described in (2)
- `self` **inside** `super` methods refers to subclass

Diamond problem

- A implements method foo()
- B and C are overriding it
- what about (new D).foo() call?
- many solutions
  - just don't allow this situation
  - take implementation from the first language (first - in some order)
  - Eiffel-way: provide syntax to solve name conflicts (by renaming and selecting)



Eiffel-way solution

```
class TEACHING_ASSISTANT inherit
    TEACHER
        rename
           computer_account as faculty_account
        select
           faculty_account
        end

    STUDENT
        rename
           computer_account as student_account
        end
```

Subsumption

- subtype polimorphism – If c' is a subclass of c, and `o' : InstanceTypeOf(c')`, then `o' : InstanceTypeOf(c)`
- subtype relation (partial order): „`A <: B`"
  - If `a : A` and `A <: B`, then `a : B` (subsumption)
  - `InstanceTypeOf(c') <: InstanceTypeOf(c)` iff c' is a subclass of c

Subsumption and method dispatch - examples

```
var myCell : InstanceTypeOf(cell) := new cell;
var myReCell : InstanceTypeOf(reCell) := new reCell;
procedure f(x : InstanceTypeOf(cell)) is … end;

myCell := myReCell;

f(myReCell);
```

```
procedure g(x : InstanceTypeOf(cell)) is
    x.set(3);
end;
g(myReCell);
```

Method dispatch

- static dispatch – choosing the method during compilation
- dynamic dispatch – choosing the method in run-time
- different dispatch may result in different methods selected to call
- dynamic dispatch is available in virtually all object-oriented languages
- some languages provide both kinds of dispatch
- With dynamic dispatch, there is no runtime side-effects of subsumption (think about set() from reCell, after assignment `variableOfTypeCell := reCell`)

Type information / typecase

- during subsumption from reCell to cell, we lost information that instance of reCell have method `restore()`
- `typecase` – useful feature (Simula, Modula-3), but language shouldn't require to change all related typecases when defining new subclass

```
typecase x
    when rc: InstanceTypeOf(reCell) do … rc.restore() …;
    when c: InstanceTypeOf(cell) do …;
end;
```

Covariance, contravariance, invariance

- type operator T is:
  - Covariant if `A' <: A` ⇔ `T(A') <: T(A)`
  - Contravariant if `A' <: A` ⇔ `T(A) <: T(A')`
  - Invariant if neither of above applies
- `A×B <: A'×B'` if `A <: A'` and `B <: B'`
- `A→B <: A'→B'` if `A' <: A` and `B <: B'` (function is contravariant in it's argument, and covariant in return type)
- mutable structures (mutable pairs, arrays) are invariant

Method specialization

- fields cannot change their types between subclasses (mutable structures are invariant)
- methods can be specialized (simple implication from $A \rightarrow B$ variance)
- older versions of Eiffel were supporting covariance of argument (catcall problem)
- self is covariantly specialized by inheritance
- what about methods of c, which return `InstanceTypeOf(c)`?

Self type specialization

```
class c is
    method m(): InstanceTypeOf(c) is
        … return self;
    end;
end;
subclass c' of c is
end;
```

- `self` in c' is `InstanceTypeOf(c')`
- we would like `o'.m()` to be `InstanceTypeOf(c')`
- solution: introduction of `Self` type
- we could even use Self as type for fields, but it's not always type-safe

```
class c is
    method m(): Self is … return self; end;
end;
```

```
// C++
class A
{
    public:
    A* Hello()
    {
        return this;
    }
}

class B : public class A
{
    public:
    B* World()
    {
        return this;
    }
}

// b.Hello() returns
//  A type
```

Object types

- until now, types were almost the same as classes
- introduction of object types (interfaces-like), which are independent from implementations (classes)
- on example with cells:
  `ObjectTypeOf(cell)` = Cell
- notice, that different classes may have the same ObjectType
- object protocol – type signature of the object

```
ObjectType Cell is
    var contents : Int;
    method get() : Int;
    method set(n : Int);
end;

ObjectType ReCell is
    var contents : Int;
    var backup : Int;
    method get() : Int;
    method set(n : Int);
    method restore();
end;
```

Subclassing and subtyping

- subtyping partial order: „`O' <: O` if O' has the same components as O and possibly more”
- notice, that following is true: „If c' is a subclass of c, then `ObjectTypeOf(c') <: ObjectTypeOf(c)`”
- using the definition on the right and from previous slide, we may conclude that `ReCell <: Cell` and `ReCell <: ReInteger`
- previously: subclassing-is-subtyping, currently: subclassing-implies-subtyping

```
ObjectType ReInteger is
    var contents : Integer;
    var backup : Integer;
    method restore();
end;
```

Method specialization problem

```
ObjectType Person is
    …
    method eat( food: Food );
end;
```

```
ObjectType Vegetarian is
    …
    method eat( food: Vegetables );
end;
```

- problem – if `Vegetables <: Food`, then we can't make Vegetarian a subclass of Person, because we would allow Vegetarian to eat meat (by subsumption)
- solution – type parameters

```
ObjectOperator PersonEating[F <: Food] is
    …
    method eat( food: F );
end;

ObjectOperator VegetarianEating[F <: Vegetables] is
    …
    method eat( food: F );
end;

var p : PersonEating[Vegetables];
p := new VegetarianEating[Vegetables];
```

Method specialization problem (bounded type parametrization)

```
// Java
public class Box<T> {
    private T t;
    public <U extends Number> void inspect(U u){
        ...
    }
}
```

- templates (java)
- bounded type specialization supported also by Eiffel

Method specialization problem (partially abstract types)

```
ObjectType Person is
    type F <: Food;
    …
    var lunch : F;
    method eat( food: F );
end;
```

```
ObjectType Vegetarian is
    type F <: Vegetables
    …
    var lunch : F;
    method eat( food: F );
end;
```

- by specifying the `lunch` field, we are specifying `F` type
- however, after specyfing this field, `Person/Vegetarian` can't eat anything more general

Subclassing without subtyping

- inheritance isn't connected with subtyping at all

```
ObjectType Max is
    var n : Integer;
    method max(other : Max) : Max;
end;

ObjectType MinMax is
    var n : Integer
    method max(other: MinMax) : MinMax;
    method min(other: MinMax) : MinMax;
end;
```

```
class maxClass is
    var n : Integer := 0;
    method max(other : Self) : Self is
        if self.n > other.n then return self else return other end;
    end;
end;

subclass minMaxClass of maxClass is
    method min(other : Self) : Self is
        if self.n < other.n then return self else return other end;
    end;
end;
```

`MinMax <: Max`?

```
subclass minMaxClass' of minMaxClass is
    override max(other : Self) : Self is
        if other.min(self) = other then return self else return other end;
    end;
end;
```

- assumption: `MinMax <: Max`
- `mm' : MinMax`, then by subsumption `mm' : Max`, but then, `other` may be `Max`, and `Max` doesn't have `min()` method

Object protocols

```
ObjectOperator MaxProtocol[X] is
    var n : Integer;
    method max(other : X) : X;
end;

ObjectOperator MinMaxProtocol[X] is
    var n : Integer
    method max(other: X) : X;
    method min(other: X) : X;
end;
```

- we can define higher-order subtype relation between type operators:
  - `P <<: P'` <u>iff</u> `P[T] <: P'[T]` for all types `T`
- notice that `MinMax <: MaxProtocol[MinMax]`
- then, `MinMaxProtocol <<: MaxProtocol`
- for type T, we can always create T-protocol (with Self changed to X)
- we can define subprotocol definition between types:
  - S subprotocol T if S <: T-Protocol[S]
  - S subprotocol T if S-Protocol <<: T-Protocol

Homework

- interpreter (only running correct AST, without lexer/parser) for class-based language
- language: any, if it will work on 64-bit linux (Sorry, C#, F# and Visual Basic programmers)
- details specified with resources at weekend
- deadline: 30.03.2014
- classes, inheritance

Sources

(1)„Theory of objects”, Martin Abadi, Luca Cardelli
(2)„Super and Inner — Together at Last!”, David S. Goldberg, Robert Bruce Findler, Matthew Flatt
(3)http://archive.eiffel.com/doc/online/eiffel50/intro/language/tutorial-10.html
(4)http://daimi.au.dk/~beta/Books/betabook.pdf
(5)http://wikipedia.org
(6)http://www.wobblini.net/singletons.html (ruby singletons' methods)
(7)http://pharo.gforge.inria.fr/PBE1/PBE1ch6.html
(8)http://stackoverflow.com/questions/11761506/inheritance-function-that-returns-self-type (c++ example)