Object orientation

Amer Diwan

Main components of object-orientation?

- Encapsulation
  - Group data and associated operations
- Information hiding
  - Need to know principle!
- Inheritance

Inheritance

- ```
  TYPE Shape = OBJECT
    color: Colors;
  END;
  ```
- ```
  TYPE Circle = Shape OBJECT
    radius: INTEGER;
  END;
  ```

One way to think about inheritance

- An object contains
  - All the fields of its supertype, followed by all the fields declared in the object
  - (All the methods of the supertype followed by all the methods in the object)
- What's a subtype?

What do they look like in memory?

- Run time representation needs
  - space for data
  - (mapping of methods)
- What do Shape and Circle objects look like?
- How are they different from records?

Another example

```
TYPE FilledCircle = Circle OBJECT
   fill_color: Colors;
END;
```

- A FilledCircle is an object with
  - all the fields of a shape, followed by
  - all the fields of a circle, followed by
  - all the fields declared in FilledCircle
- FilledCircle <: Circle <: Shape

Inheritance and subtyping

- Is a FilledCircle bigger or smaller than a Circle?
- Does this make sense from a subtyping point of view?

An example

- ```
  T = OBJECT ... END;
  S = T OBJECT ... END;
  x: REFANY;
  x := NEW(S);
  NARROW (x, T)
  ```
- What representation does this need?
  ```

Another example

- ```
  l: ShapeList.T;
  l := ShapeList.Cons(NEW(Circle), l)
  l := ShapeList.Cons(NEW(FilledCircle),
  l)
  l := ShapeList.Cons(NEW(Square), l)
  ```
- Is this legal?

Continuing example

- ```
  WHILE l.tail # NIL DO
    l.head.radius := 10
    l := l.tail;
  END
  ```
- ```
  WHILE l.tail # NIL DO
    IF l.head.color = Colors.White THEN
      ...
    END;
    l := l.tail;
  END;
  ```
  ```

Continuing example

- ```
  PROCEDURE f(s: Shape) =
      IF s.color # Colors.White THEN
        ...
      END
  ```

Polymorphism

- Contrast with monomorphic:
  - Functions and their operands have a unique type
  - Every value and variable can be interpreted to be of one and only one type
- More directly
  - Functions work uniformly on a range of operand types
  - Some values and variables may have more than one type
  ```

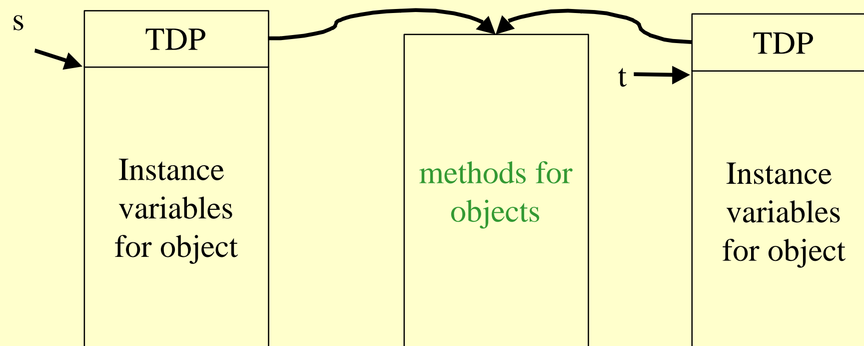
Polymorphism versus monomorphism

- Expressiveness
- Simplicity
- Ease of implementation
- Efficiency

Adding methods to objects: support for encapsulation

- ```
  TYPE Shape = OBJECT
    color: Colors;
  METHODS
    setColor(to: Color) := ...
  END;
  ```
- ```
  TYPE Circle = Shape OBJECT
    radius: INTEGER;
  END;
  ```

Representation



Let's look at the guts of a method

- ```
  TYPE Shape = OBJECT
     color: Colors;
  METHODS
     setColor(to: Color) := setColorShape;
  END;
  ```
- ```
  PROCEDURE setColorShape(to: Color) =
  ...
  ```
- How does setColorShape access the "color" instance variable?

Two approaches

- Java, C++, Smalltalk:
  - pass implicit parameter called “this” or “self”
- Modula-3:
  - s.setColor(c) => setColorShape(s, c)

Overriding methods

- Overriding gives a new implementation for an existing method
- ```
  TYPE Shape = OBJECT
    color: Colors;
  METHODS render(); END;
  TYPE Circle = Shape OBJECT
    radius: INTEGER;
  OVERRIDES render := ...; END;
  TYPE Square = Shape OBJECT
    side: INTEGER;
  OVERRIDES render := ...; END;
  ```
  ```

Representation

- Each type has its own mapping of methods to implementations
- The correct implementation is picked at run time!

Some implications of inheritance

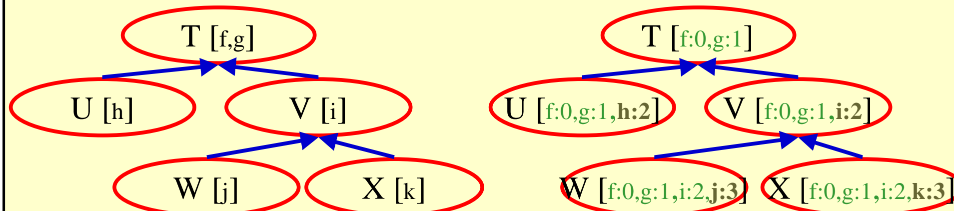
- ```
  WHILE l.tail # NIL DO
       f(l.head)
       l := l.tail;
  END;
  ```
- ```
  PROCEDURE f(s: Shape) =
     s.render();
  ```
- What method is executed here?

How to implement method dispatch?

- Assumptions:
  - Static typing
  - Single inheritance (we will look at multiple inheritance later)

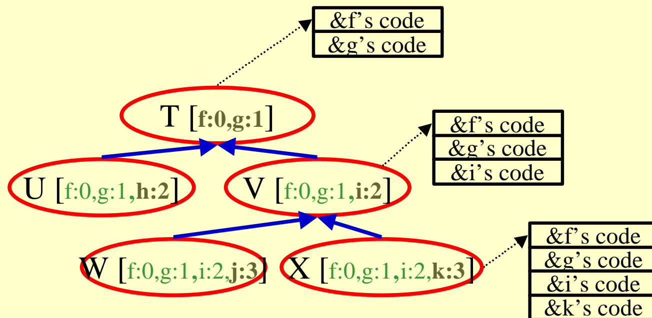
V-Tables

- Idea:
  - Prepend the methods of a supertype to a subtype
  - A method T::m appears in the same position in all T's subclasses



V-tables (cont.)

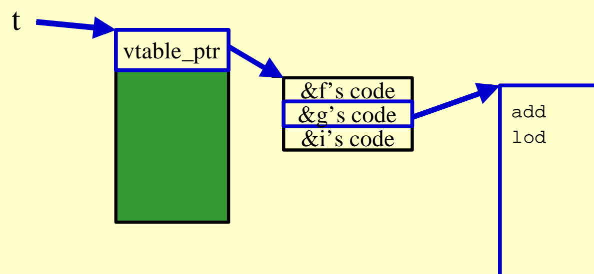
Construct a v-table for each class



v-tables are typically part of the type descriptor

V-tables(cont.)

t->g() becomes
vp = t->vtable_ptr
gaddr =*(vp+g's offset)
(*gaddr)()



More examples

- ```
  t: Shape;
  t := NEW Circle;
  t.render();
  ```
- ```
  t: Circle;
  t := NEW Circle;
  t.render();
  NARROW(t, Shape).render();
  ```

Overriding versus redefining

- ```
  TYPE Shape = OBJECT
    color: Colors;
  METHODS render(); END;
  TYPE Circle = Shape OBJECT
    radius: INTEGER;
  METHODS render := ...; END;
  TYPE Square = Shape OBJECT
    side: INTEGER;
  METHODS render := ...; END;
  ```
  ```

Back to our old examples

- ```
  WHILE l.tail # NIL DO
       f(l.head)
  END;
  ```
- ```
  PROCEDURE f(s: Shape) =
     s.render();
  ```

Back to our old examples (cont.)

- ```
  t: Shape;
  t := NEW Circle;
  t.render();
  ```
- ```
  t: Circle;
  t := NEW Circle;
  t.render();
  NARROW(t, Shape).render();
  ```

Contravariance and inheritance

- Issues with
  - The “id” function
  - The “clone” method

Next topic: Case study

- Reading: Read up about objects in Java in your favorite Java book