Chapter 13
Object Oriented Programming

Quick Review

Abstract Data Types

Object Model

- *Class, constructor, destructor, object-oriented languages, client of a class, class methods, instance methods, inner class*
- *Visibility*
- *Inheritance*
  - Is-a v.s. Has-a
  - Single inheritance v.s. Multiple inheritance
- *Templates and generics*
- *Abstract class and interface*

Polymophism

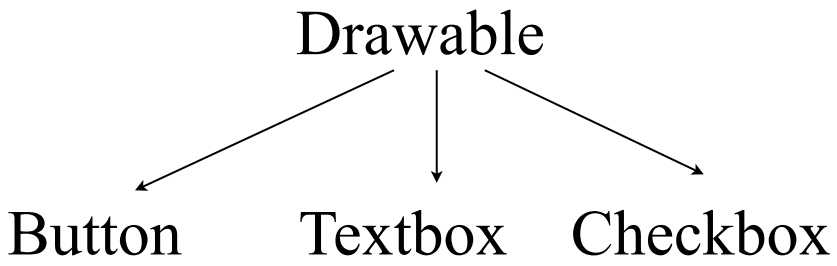
In OO: it is materialized by late binding of method calls.

Virtual Method:

A method whose behavior can be overridden within an inheriting class by a function with the same signature.

In Java, by default, all methods are virtual.

```
for (Drawable obj : myList)
    obj.paint( );
```



Principle: substitutability.

Abstract Classes

Defn: An *abstract class* is one that is either declared to be abstract or has one or more abstract methods.

Defn: An *abstract method* is a method that contains no code beyond its signature.

Any subclass of an abstract class that does not provide an implementation of an inherited abstract method is itself abstract.

Because abstract classes have methods that cannot be executed, client programs cannot initialize an object that is a member an abstract class.

This restriction ensures that a call will not be made to an abstract (unimplemented) method.

```
abstract class Expression { ... }
    class Variable extends Expression { ... }
    abstract class Value extends Expression { ... }
        class IntValue extends Value { ... }
        class BoolValue extends Value { ... }
        class FloatValue extends Value { ... }
        class CharValue extends Value { ... }
    class Binary extends Expression { ... }
    class Unary extends Expression { ... }
```

Interfaces

Defn: An *interface* encapsulates a collection of constants and abstract method signatures.

An interface may not include either variables, constructors, or non-abstract methods.

```
public interface Map {

    public abstract boolean containsKey(Object key);

    public abstract boolean containsValue(Object value);

    public abstract boolean equals(Object o);

    public abstract Object get(Object key);

    public abstract Object remove(Object key);

    ...

}
```

Because it is not a class, an interface does not have a constructor, but an abstract class does.

Some like to think of an interface as an alternative to multiple inheritance.

Strictly speaking, however, an interface is not quite the same since it doesn't provide a means of reusing code;

i.e., all of its methods must be abstract.

An interface is similar to multiple inheritance in the sense that an interface is a type.

A class that implements multiple interfaces appears to be many different types, one for each interface.

Virtual Method Table (VMT)

How is the appropriate virtual method called at run time?

At compile time the actual run time class of any object may be unknown.

```
MyList myList;
...
System.out.println(myList.toString( ));
```

Each class has its own VMT, with each instance of the class having a reference (or pointer) to the VMT.

A simple implementation of the VMT would be a hash table, using the method name (or signature, in the case of overloading) as the key and the run time address of the method invoked as the value.

For statically typed languages,

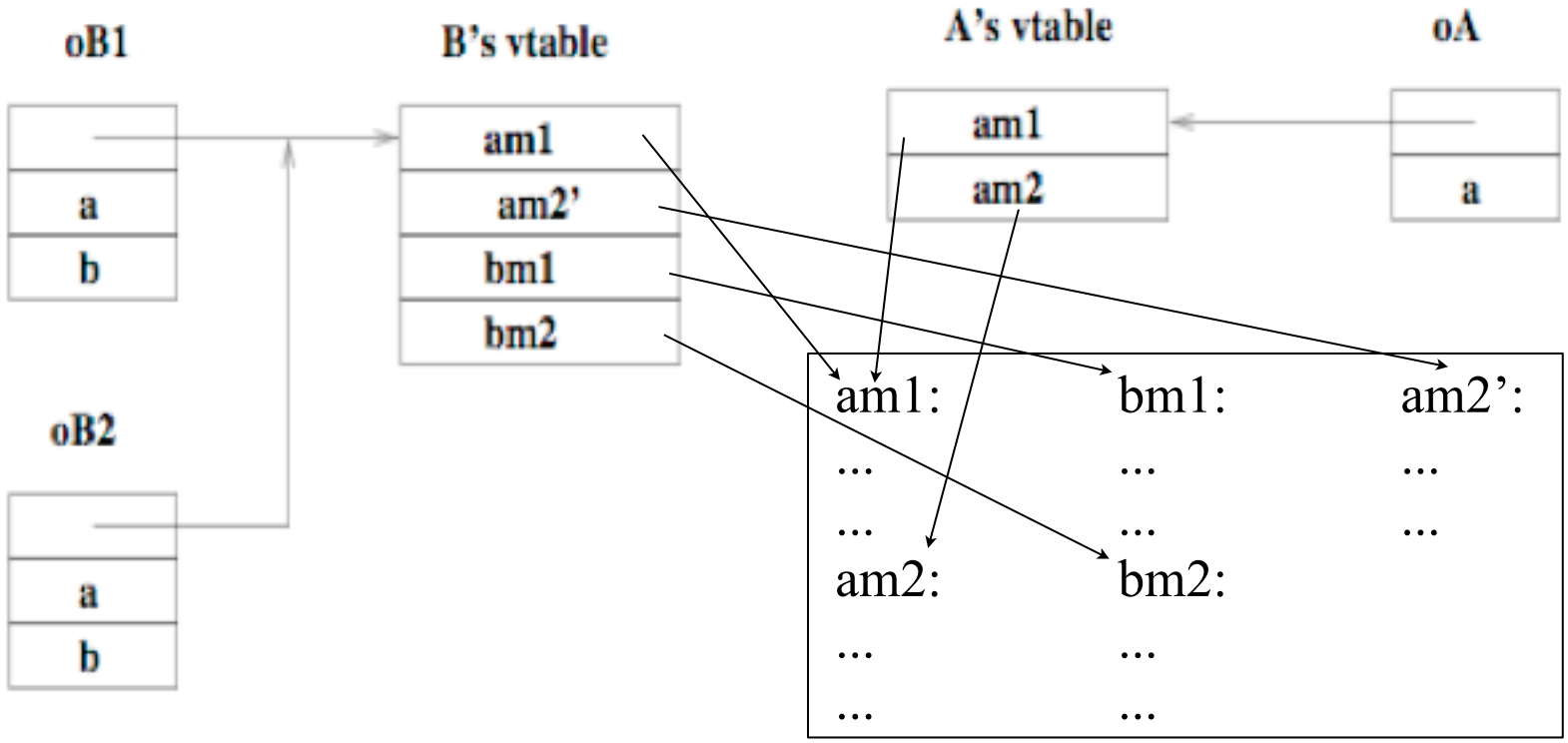
the VMT is kept as an array.

The method being invoked is converted to an index into the VMT at compile time.

```
class A {
    Obj  a;
    void am1( ) { ... }
    void am2( ) { ... }
}
class B extends A {
    Obj  b;
    void bm1( ) { ... }
    void bm2( ) { ... }
    void am2( ) { ... }
}
```

```
class A {
    Obj  a;
    void am1( ) { ... }
    void am2( ) { ... }
}
```

```
class B extends A {
    Obj  b;
    void bm1( ) { ... }
    void bm2( ) { ... }
    void am2( ) { ... }
}
```



Run Time Type Identification

Defn: Run time type identification (RTTI) is the ability of the language to identify at run time the actual type or class of an object.

All dynamically typed languages have this ability, whereas most statically typed imperative languages, such as C, lack this ability.

At the machine level, recall that data is basically untyped.

In Java, for example, given any object reference, we can determine its class via:

```
Class c = obj.getClass( );
```

Reflection

Reflection is a mechanism whereby a program can discover and use the methods of any of its objects and classes.

Reflection is essential for programming tools that allow plugins (such as Eclipse -- www.eclipse.org) and for JavaBeans components.

In Java the *Class* class provides the following information about an object:

- The superclass or parent class.
- The names and types of all fields.
- The names and signatures of all methods.
- The signatures of all constructors.
- The interfaces that the class implements.

```
Class class = obj.getClass( );
Constructor[ ] cons = class.getDeclaredConstructors( );
for (int i=0; i < cons.length; i++) {
    System.out.print(class.getName( ) + "(" );
    Class[ ] param = cons[i].getParameterTypes( );
    for (int j=0; j < param.length; j++) {
        if (j > 0) System.out.print(", ");
            System.out.print(param[j].getName( );
    }
    System.out.println( ")" );
}
```

13.4 Java

- mixed language
  - *primitive types: int, double, boolean*
  - *objects*
- statically typed with some dynamic flavor
- single inheritance

Direct support for:

- inner classes
- visibility modifiers
- abstract classes
- interfaces
- generics
- run time type identification
- reflection

Example: Symbolic Differentiation

- Implement symbolic differentiation
- State rules
- Separate simplification (not included in this example)
- Use of abstract syntax

Symbolic Differentiation Rules

Figure 13.19

$$\frac{d}{dx}(c) = 0 \qquad c \text{ is a constant}$$

$$\frac{d}{dx}(x) = 1$$

$$\frac{d}{dx}(u + v) = \frac{du}{dx} + \frac{dv}{dx} \qquad u \text{ and } v \text{ are functions of } x$$

$$\frac{d}{dx}(u - v) = \frac{du}{dx} - \frac{dv}{dx}$$

$$\frac{d}{dx}(uv) = u\frac{dv}{dx} + v\frac{du}{dx}$$

$$\frac{d}{dx}\left(\frac{u}{v}\right) = \left(v\frac{du}{dx} - u\frac{dv}{dx}\right) \Big/ v^2$$

Example

d/dx (2*x+1)

d/dx (2*x+1) = d/dx (2*x) + d/dx 1

*= x * d/dx 2 + 2 * d/dx x + 0*

*= x * 0 + 2 * 1 + 0*

-- simplified algebraically: 2

Abstract Syntax of Expressions

Expression = *Variable* | *Value* | *Binary*

Variable = *String* id

Value = *int* value

Binary = *Add* | *Subtract* | *Multiply* | *Divide*

Add = *Expression* left, right

Subtract = *Expression* left, right

Multiply = *Expression* left, right

Divide = *Expression* left, right

```
public abstract class Expression {

    public abstract Expression diff(Variable x);

}
```

```
class Value extends Expression {
    private int value;
    public Value(int v) { value = v; }
    public Expression diff(Variable x) {
        return new Value(0);
    }
}
```

```
class Variable extends Expression {
    private String id;
    static final private Value zero = new Value(0);
    static final private Value one  = new Value(1);
    public Variable(String s) { id = s; }
    public Expression diff(Variable x) {
        return id.equals(x.id) ? one : zero;
    }
}
```

```
abstract class Binary extends Expression {

    protected Expression left, right;

    protected Binary(Expression u, Expression v) {

        left = u; right = v;

    }

}
```

```
class Add extends Binary {
    public Add(Expression u, Expression v) {
        super(u, v);
    }
    public Expression diff(Variable x) {
        return new Add(left.diff(x), right.diff(x));
    }
}
```

... ...

differentiation of 2*x + 1 on x (with appropriate print function):

2*1 + x * 0 + 0

Prolog Program

```
d(X, U+V, DU+DV) :- d(X, U, DU), d(X, V, DV).
d(X, U-V, DU-DV) :- d(X, U, DU), d(X, V, DV).
d(X, U*V, U*DV + V*DU) :- d(X, U, DU), d(X, V, DV).
d(X, U/V, (V*DU - U*DV)/(V*V)) :- d(X, U, DU), d(X, V, DV).
d(X, C, 0) :- atomic(C), C\=X.
d(X, X, 1).
```

differentiation of 2*x + 1 on x:

2*1 + x * 0 + 0

Haskell Program

```
data Expr = Num Int | Var String | Add  Expr Expr |
                    Sub Expr Expr | Mul Expr Expr |
                    Div Expr Expr deriving (Eq, Ord, Show)


diff :: String -> Expr -> Expr
diff x (Num c) = Num 0
diff x (Var y) = if x == y then Num 1 else Num 0
diff x (Add u v) = Add (diff x u) (diff x v)
diff x (Sub u v) = Sub (diff x u) (diff x v)
diff x (Mul u v) = Add (Mul u (diff x v))
                                  (Mul v (diff x u))
diff x (Div u v) = Div (Sub (Mul v (diff x u))
                                        (Mul u (diff x v))) (Mul v v)
```

Haskell Output

differentiation of 2*x + 1 on x:

```
Add (Add (Mul (Num 2) (Num 1))
         (Mul (Var "x") (Num 0))) (Num 0))
```