i219 Software Design Methodology
2. Basic concepts of object-oriented technology

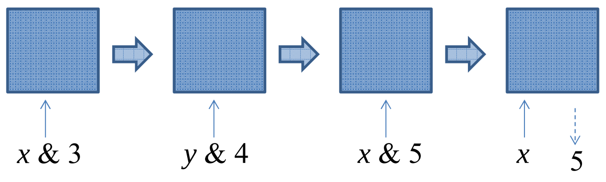
Kazuhiro Ogata (JAIST)

Outline of lecture

- Object
- Attribute
- Message & method
- Class
- Inheritance
- Interface
- Abstract class

Object (1)

- Let us consider the following thing:
  - Feeding a key and a value into it, it associates the value with the key.
  - Feeding a key into it, if there exit some values associated with the key, it returns the value most recently associated with the key.



An example of objects

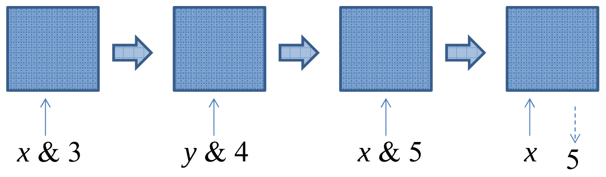
Object (2)

- What we can do for objects are essentially:
  - to send messages to them, and
  - to send messages to them and get some results from them.
- On receipt of a message, what an object does is basically:
  - to create new objects,
  - to send messages to objects (including itself) (and get some results from them),
  - to change something inside the object, and/or
  - to return something as the result of the message.

Object (3)

- When we use software components, we are more interested in what they can do for us than how they do those things (how they are implemented).
- Objects can be such components.

When we want to associate values with keys, we can use the object (let's call it the blue object):

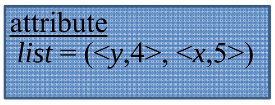


But, not interested in how it is implemented inside, e.g., hash table and binary search tree.

Attribute (1)

- An object has its own internal state that are made of what are called attributes.

If the blue object is implemented with a list of pairs of keys and values, then it has such a list as its attribute.



Attribute (2)

- In Smalltalk, given an object, its attributes can only be (essentially) directly modified and observed (accessed) by the object itself.

  What holds attributes are called instance variables.
- In Java, attributes can be given four different access levels.

  Attributes can be declared with one of the three access modifiers (private, protected and public), and with none of them (meaning package).

  Even if an attribute is declared as private, it can also be accessed by other objects.

  What holds attributes are called fields.

Message & method (1)

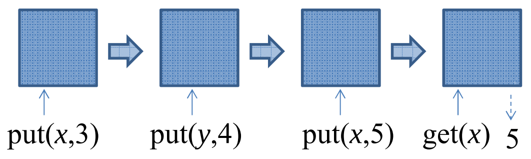
- An object cannot accept all messages.
- What messages can be accepted by an object are described as methods.
- A method has a name; it may have a list of parameters (together with their types); it may have a type whose value is returned by the method.

If an object has a method such that its name is get, and it takes one parameter whose type is String and returns an integer, then it can accept the message get("abc") and returns an integer, such as 3.

Message & method (2)

- The blue object can accept two kinds of messages
  - to associate a value with a key and
  - to get a value associated with a key
- The object has the following two methods for the two kinds of messages:
  - put(*k*: Key, *v*: Value): Void
  - get(*k*: Key): Value

<u>attribute</u>
list = (<*y*,4>, <*x*,5>)
<u>method</u>
put(k: Key, v: Value): Void
get(k: Key): Value



Message & method (3)

- In Smalltalk, if an object has a method, any (other) objects can send the object the message corresponding to the method.
- In Java, also possible to make an access control to methods, making them private, protected, public & package.

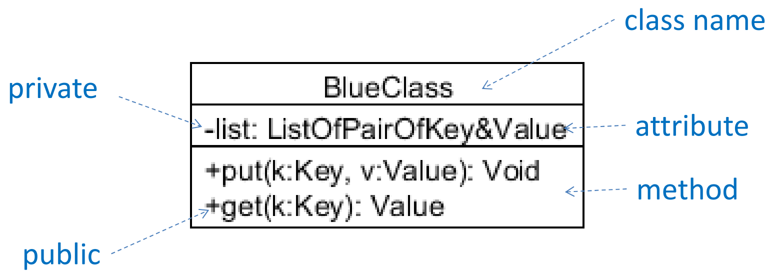
Class (1)

- A class describes what objects look like.
- Each object has its class and is called an instance of the class.
- In a class, attributes and methods are declared; an instance (object) of the class has those attributes and methods.

Units of access control to attributes (or encapsulation) are instances in Smalltalk, but classes in Java.
If an attribute is declared as private in a class in Java, then the attribute of an object (instance) of the class can be accessed by not only the object itself but also any other objects of the class.

Class (2)

- The class from which the blue object is instantiated is described as follows (in UML):



Class (3)

- An implementation of BlueClass in Java is as follows:

```
public class BlueClass {
  private ArrayList<KeyValPair> list;

  public BlueClass() {
    list = new ArrayList<KeyValPair>(); }

  public void put(String k, int v) { … }

  public Integer get(String k) { … }

  …
}
```

attribute

making an object of ArrayList<…>

constructor making instances of the class

primitive type for integers

method

wrapper class for integers

Class (4)

- put() & get() are as follows:

```
public void put(String k, int v) {
  for (int i = 0; i < list.size(); i++)
    if (k.equals(list.get(i).getKey())) {
      list.get(i).setVal(v);
      return; }
  list.add(0,new KeyValPair(k,v)); }

public Integer get(String k) {
  for (int i = 0; i < list.size(); i++)
    if (k.equals(list.get(i).getKey()))
      return list.get(i).getVal();
  return null; }
```

Message get(i) is sent to list, obtaining the pair at position i in the list. Then, message getKey() is sent to the pair, obtaining the key (a String) in the pair. Then, message equals(…) together with the key as the argument is sent to k, obtaining true or false.

Class (4)

- In Smalltalk, classes are also objects (which are instances of the class Metaclass).

  (Almost everything such as integers and messages are objects in Smalltalk)
- For each class, in Java, there exists an object (which is an instance of the class Class) that represents the class.

Class (5)

- Note that there are some object-oriented programming languages that have no classes:
  - Self : designed & developed at Xerox PARC, Stanford Univ., & Sun Microsystems; one of the main designers is David Unger; purer than Smalltalk.
  - ABCL/1 : designed & developed at Titech & U. of Tokyo; one of the main designers is Akinori Yonezawa; an object-oriented concurrent programming language

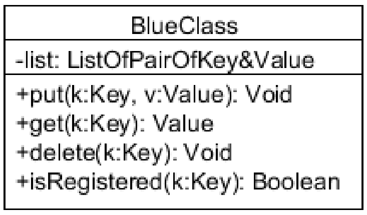
Inheritance (1)

- An existing class can be extended (or specialized) to make a new class.
- The new class basically inherits all attributes and methods owned by the existing class.

  (some of them may not be directly accessed by the new class in Java)

Let us add two new methods to BlueClass:

1. delete(k: Key): Void
2. isRegistered(k: K): Boolean

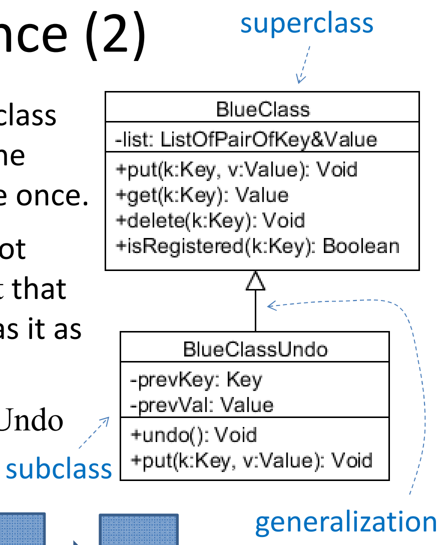

BlueClass
-list: ListOfPairOfKey&Value
+put(k:Key, v:Value): Void
+get(k:Key): Value
+delete(k:Key): Void
+isRegistered(k:Key): Boolean

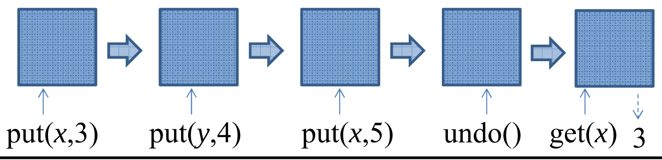
Inheritance (2)

Extend BlueClass to make a new class called BlueClassUndo such that the effect with put(k,v) can be undone once.

An object of BlueClassUndo cannot directly access to the attribute list that is inherited from BlueClass but has it as one of the attributes.

The method put(…) in BlueClassUndo overrides put(…) in BlueClass.





Inheritance (3)

- Implementations of isRegistered() and delete() in Java are as follows:

```
public boolean isRegistered(String k) {
    for (int i = 0; i < list.size(); i++)
        if (k.equals(list.get(i).getKey()))
            return true;
    return false;  }

public void delete(String k) {
    for (int i = 0; i < list.size(); i++)
        if (k.equals(list.get(i).getKey())) {
            list.remove(i);
            return;  }  }
```

Inheritance (4)

- An implementation of BlueClassUndo in Java is as follows:

BlueClass is extended to make BlueClassUndo.

```
public class BlueClassUndo extends BlueClass {
    private String prevKey;
    private Integer prevVal;
    public BlueClassUndo() {
        super();
        prevKey = null;
        prevVal = null;
    }
    public void put(String k, int v) { … }
    public void undo() { … }
}
```

The constructor in BlueClass is invoked.

Inheritance (5)

- put(…) & undo() are as follows:

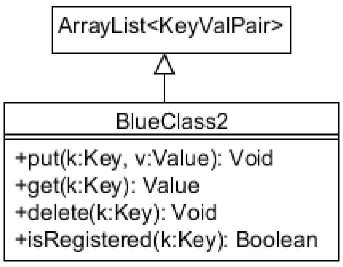
```
public void put(String k, int v)
{
   prevKey = k;
   if (super.isRegistered(k)) {
      prevVal = super.get(k);
   } else {
      prevVal = null;  }
   super.put(k,v);  }
```

```
public void undo() {
   if (prevKey != null) {
      if (prevVal != null) {
         super.put(prevKey,prevVal);
         prevVal = null;
      } else {
         super.delete(prevKey);  }
      prevKey = null;  }  }
```

The method put(…) in BlueClass is invoked.

Inheritance (6)

- Another design & implementation of BlueClass



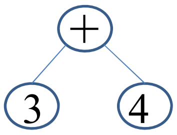
```
public class BlueClass2
            extends  ArrayList<KeyValPair> {
   public BlueClass2() { super(); }
   public void put(String k, int v) {
      for (int i = 0; i < super.size(); i++)
         if (k.equals(this.get(i).getKey())) {
            this.get(i).setVal(v);
            return;  }
      super.add(0,new KeyValPair(k,v));  }
   …  }
```

The method get(…) in the class of which the object is executing the method put(…) is invoked.
Keyword this refers to the currently running object.

Interface (1)

- An interface is what specifies methods such that any of its implementations (classes) is supposed to provide them.
- An interface can be used as type as a class.
- But, no objects are made from interfaces.

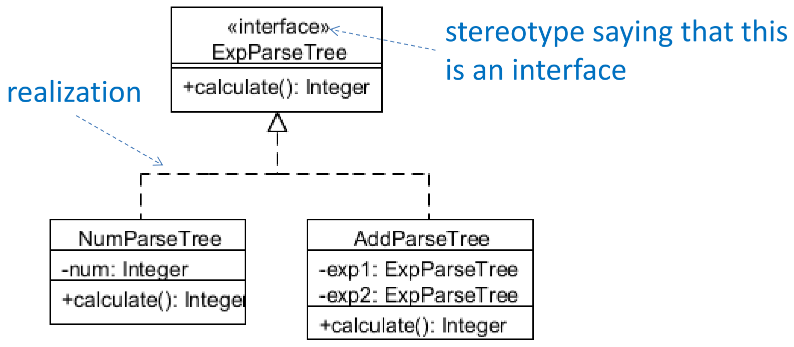
Let us consider parse trees of arithmetic expressions such as



There are several kinds of nodes such as addition and number.
Each kind of nodes is expressed as one class.
Each of such classes is supposed to provide a method calculate() that calculates (the expression corresponding to) the parse tree whose top is an object of that class.

Interface (2)

As a type for any expression whose top is either addition or number, an interface can be used; it also specifies that its implementations are supposed to provide a method calculate().



Interface (3)

- ExpParseTree, NumParseTree & AddParseTree in Java:

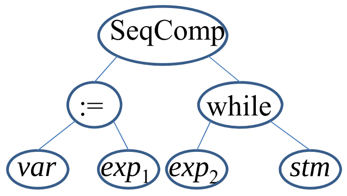
```
public interface ExpParseTree {  int calculate();  }
                    declaration of interface
public class NumParseTree  implements ExpParseTree {
   private int val;
   public NumParseTree(int x) { val = x; }            ExpParseTree is
   public int calculate() { return val; }  }          implemented

public class AddParseTree implements ExpParseTree {
   private ExpParseTree ept1, ept2;
   public AddParseTree(ExpParseTree e1, ExpParseTree e2) {
      ept1 = e1;  ept2 = e2; }
   public int calculate() {
      int n1 = ept1.calculate();  int n2 = ept2.calculate();
      return n1 + n2; }  }
```

Abstract Class (1)

- An abstract class is what is between a class and an interface.
- It can have attributes and methods such that some methods are not implemented and called abstract methods.
- No objects are made from abstract classes.

Let us consider parse trees of imperative programs such as



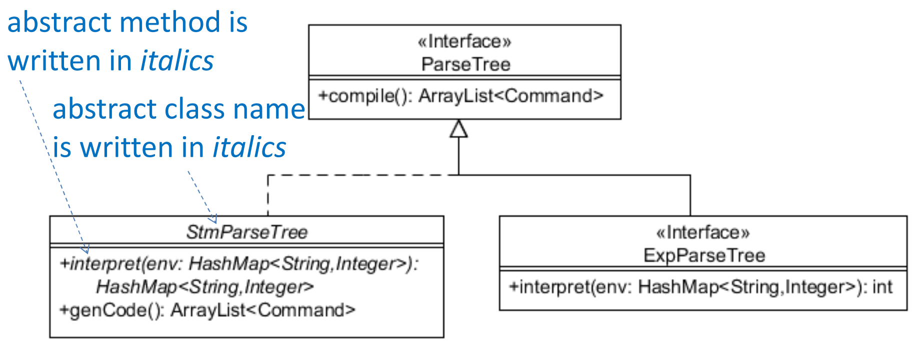
var := *exp*$_1$;
while *exp*$_2$ **do** *stm* **od**

Each class expressing each kind of nodes is supposed to provide interpret(…) & compile(…); the former interprets the corresponding program and the latter generates a list of commands (instructions) for the program.

Abstract Class (2)

A list of commands is supposed to end with command “quit”.

As a type for both statements and expressions, an interface (ParseTree) is used; As a type for any statements, an abstract class (StmParseTree) is used, providing a method that puts “quit” at the end of each list generated.


abstract method is written in *italics*
abstract class name is written in *italics*
«Interface»
ParseTree
+compile(): ArrayList<Command>
StmParseTree
+interpret(env: HashMap<String,Integer>): HashMap<String,Integer>
+genCode(): ArrayList<Command>
«Interface»
ExpParseTree
+interpret(env: HashMap<String,Integer>): int

Abstract Class (3)

- ParseTree & StmParseTree in Java:

declaration of abstract class

declaration of abstract method

```
public interface ParseTree {
   List<Command> compile();  }

abstract class StmParseTree implements ParseTree {
   abstract Map<String,Integer>
      interpret(Map<String,Integer> env)
      throws InterpreterException;
   public List<Command> genCode() {
      List<Command> cl;
      cl = this.compile();
      cl.add(new Command(CommandName.QUIT));
      return cl; }  }
```

Summary

- Object
- Attribute
- Message & method
- Class
- Inheritance
- Interface
- Abstract class