Object-oriented programming

Lecture №8

Type, Polymorphism

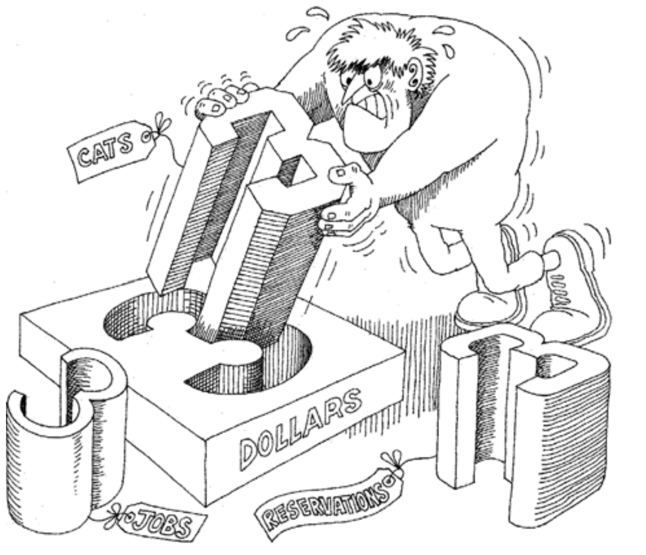
PhD, Alexander Vlasov

Question?

- class & interface
- extend & implement
- λ-functions

Type system

- class=type
- We will use the terms *type* and *class* interchangeably
- static / dynamic typing
- strong / weak typing
- explicit / implicit typing



speed=length/time, but
?????=time/length

Typing is the enforcement of the class of an object, such that objects of different types may not be interchanged, or at the most, they may be interchanged only in very restricted ways.

Using strongly typed languages

- Without type checking, a program in most languages can ‘crash’ in mysterious ways at runtime.
- In most systems, the edit-compile-debug cycle is so tedious that early error detection is indispensable.
- Type declarations help to document programs.
- Most compilers can generate more efficient object code if types are declared.

Remark: In almost all cases, the programmer in fact knows what sorts of objects are expected as the arguments of a message, and what sort of object will be returned

Another point of view

"Now! *That* should clear up a few things around here!"

```
double a=3.6;    //explicit typing
var b=a;         //implicit typing, java 10
int c=3.7f;      // weak typing, java is strong typing
```

```
// static typing              //dynamic typing
int add(int x,int y){         def add(x,y):
    return x+y                        return x+y
}
```

Java?

```
Operationable operation1 = (x,y)-> x + y;
```

Duck typing

If it walks like a duck and it quacks like a duck, then it must be a duck

```
class Duck:
    def fly(self):
        print("Duck flying")

class Airplane:
    def fly(self):
        print("Airplane flying")

class Whale:
    def swim(self):
        print("Whale swimming")

for animal in Duck(), Airplane(), Whale():
    animal.fly()
```

```
Duck flying
Airplane flying
AttributeError: 'Whale' object has no attribute 'fly'
```

Polymorphism

- *Ad hoc polymorphism*: defines a common interface for an arbitrary set of individually specified types.
- *Parametric polymorphism*: when one or more types are not specified by name but by abstract symbols that can represent any type.
- *Subtyping* (also called *subtype polymorphism* or *inclusion polymorphism*): when a name denotes instances of many different classes related by some common superclass.

Christopher Strachey in 1967

Ad hoc polymorphism

```
public class AdHocPolymorphism {
  public static void f(double x) {
                System.out.println(“double”);
                System.out.println(x);
  }
 //public static int f(int x) {
//              System.out.println(“int2”);
//              System.out.println(x);
 //}
 public static void f(char x) {
                System.out.println(“char”);
                System.out.println(x);
 }
 public static void f(int x) {
                System.out.println(“int”);
                System.out.println(x);
  }
}
```

```
public static void main(String args[]) {
  f((byte)1);
  f((short)2);
  f('a');
  f(3);
  f(4L);
  f(5.6F);
  f(5.6);
}
```

Output:

```
int
1
int
2
char
a
int
3
double
4.0
double
5.599999904632568
double
5.6
```

Parametric polymorphism

```
public class Tree<T>{
        private T value;
        private Tree<T> left;
        private Tree<T> right;

        public void replaceAll(T value){
                this.value = value;
                if (left != null)
                        left.replaceAll(value);
                if (right != null)
                        right.replaceAll(value);
        }
}
```

```
Tree<String> tree=new Tree<String>()
```

Subtype polymorphism

```
class A {
    public String print() {
        return "A";
    }
}
class B extends A {
    @Override
    public String print() {
        return"B";
    }
}
```

```
List<A> list = new ArrayList<A>();
list.add(new A());
list.add(new A());
list.add(new B());

public void printAll() {
    for(A i : list) {
        System.out.println(i.print());
    }
}
```

Output:

```
A
A
B
```

Polymorphism via class

In Java, all methods are virtual!

Polymorphism via Interfaces

- An interface name can be used as the type of an object reference variable

```
Speaker current;
```

- The `current` reference can be used to point to any object of any class that implements the `Speaker` interface
- The version of `speak` that the following line invokes depends on the type of object that `current` is referencing

```
current.speak();
```

Polymorphism via Interfaces

- Suppose two classes, `Philosopher` and `Dog`, both implement the `Speaker` interface, providing distinct versions of the `speak` method
- In the following code, the first call to `speak` invokes one version and the second invokes another:

```
Speaker guest = new Philospher();
guest.speak();
guest = new Dog();
guest.speak();
```

final

- ***final* member data**
  Constant member

- ***final* member function**
  The method can’t be overridden.

- ***final* class**
  ‘Base’ is final, thus it can’t be extended

(String class is final)

```
final class Base {
  final int i=5;
  final void foo() {
    i=10;
//what will the compiler say
about this?
  }
}

class Derived extends Base {
// Error
  // another foo ...
  void foo() {

  }
}
```

Static

- **Member data** - Same data is used for all the instances (objects) of some Class.

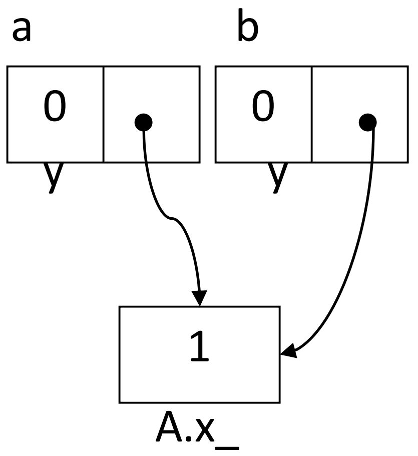
```
Class A {
   public int y = 0;
   public static int x_ = 1;
};

A a = new A();
A b = new A();
System.out.println(b.x_);
a.x_ = 5;
System.out.println(b.x_);
A.x_ = 10;
System.out.println(b.x_);
```

Assignment performed on the first access to the Class.
Only one instance of 'x' exists in memory

Output:

```
1
5
10
```



Static

- **Member function**
  - Static member function can access only static members
  - Static member function can be called without an instance.

```
Class TeaPot {
        private static int numOfTP = 0;
        private Color myColor_;
        public TeaPot(Color c) {
                myColor_ = c;
                numOfTP++;
        }
        public static int howManyTeaPots()
                { return numOfTP; }

        // error :
        public static Color getColor()
                { return myColor_; }
}
```

Static

```
Usage:

TeaPot tp1 = new TeaPot(Color.RED);

TeaPot tp2 = new TeaPot(Color.GREEN);

System.out.println("We have " +
        TeaPot.howManyTeaPots()+ "Tea Pots");
```

Static

- **Block**
  - Code that is executed in the first reference to the class.
  - Several static blocks can exist in the same class (Execution order is by the appearance order in the class definition).
  - Only static members can be accessed.

```
class RandomGenerator {
   private static int seed_;

   static {
        int t = System.getTime() % 100;
        seed_ = System.getTime();
        while(t-- > 0)
          seed_ = getNextNumber(seed_);
        }
   }
}
```

Inner Classes

- Description
  - Class defined in scope of another class
- Property
  - Can directly access all variables & methods of enclosing class (including private fields & methods)
- Example

```
public class OuterClass {
    public class InnerClass {
        ...
    }
}
```

Anonymous Inner Class

- Doesn’t name the class
- inner class defined at the place where you create an instance of it (in the middle of a method)
  - Useful if the only thing you want to do with an inner class is create instances of it in one location
- In addition to referring to fields/methods of the outer class, can refer to final local variables

Anonymous inner classes

- use
  ```
  new Foo() {
      public int one() { return 1; }
      public int add(int x, int y) { return x+y; }
      };
  ```
- to define an anonymous inner class that:
  - extends class Foo
  - defines methods one and add

MyList without anonymous inner class

```
public class MyList implements Iterable {
   private Object [ ] a;
   private int size;
   public Iterator iterator() {
    return new MyIterator();
    }
   public class MyIterator implements Iterator {
      private int pos = 0;
      public boolean hasNext() { return pos < size; }
      public Object next()      { return a[pos++]; }
   }
}
```

MyList with anonymous inner class

```
public class MyList implements Iterable {
    private Object [ ] a;
    private int size;
    public Iterator iterator() {
     return new Iterator () {
        private int pos = 0;
        public boolean hasNext() { return pos < size; }
        public Object next()      { return a[pos++]; }
    }
}
```

Nested class

- Declared like a standard inner class, except you say “static class” rather than “class”.
- For example:

```
class LinkedList {
  static class Node {
    Object head;
    Node tail;
    }
  Node head;
  }
```

Nested classes

- An instance of an inner class does not contain an implicit reference to an instance of the outer class
- Still defined within outer class, has access to all the private fields
- Use if inner object might be associated with different outer objects, or survive longer than the outer object
  - Or just don’t want the overhead of the extra pointer in each instance of the inner object