Programare Orientată pe Obiecte

Genericitate

sau polimorfism parametric

Dr. Petru Florin Mihancea

A

Elemente fundamentale de genericitate în Java

O cerință

Dorim să definim o clasă ce reprezintă un 2-tuplu

- **o pereche de două elemente de orice tip**
- **constructor/metode pentru inițializarea/setarea perechii**
- **câte o metodă pentru accesarea fiecărui element**

Soluție: ne bazăm pe polimorfism (de subtip), deoarece o referință Object poate referii orice fel de obiect

```
public class Pair {
    private Object p1,p2;
    public Pair(Object p1, Object p2) {
        this.p1 = p1;
        this.p2 = p2;
    }
    public void setFirst(Object p1) {
        this.p1 = p1;
    }
    public Object getFirst() {
        return p1;
    }
    public void setSecond(Object p2) {
        this.p2 = p2;
    }
    public Object getSecond() {
        return p2;
    }
}
```

```
public class Utilities {
    private static Pair doSet(Pair p) {
        Ceasornicar om = p.getFirst(); //Eroare compilare
        ClockType aparat = p.getSecond(); //Eroare compilare
        om.regleaza(aparat);
        return p;
    }
    public static void createSetAndDisplay(Clock aparat) {
        Ceasornicar om = new Ceasornicar();
        Pair p = new Pair(om,aparat);
        Pair res = doSet(p);
        ClockType c = res.getSecond(); //Eroare compilare
        System.out.println(c);
    }
}
```

DAR ...

```
public class Utilities {
    private static Pair doSet(Pair p) {
        Ceasornicar om = (Ceasornicar)p.getFirst();
        ClockType aparat = (ClockType)p.getSecond();
        om.regleaza(aparat);
        return p;
    }
    public static void createSetAndDisplay(Clock aparat) {
        Ceasornicar om = new Ceasornicar();
        Pair p = new Pair(om,aparat);
        Pair res = doSet(p);
        ClockType c = (ClockType)res.getSecond();
        System.out.println(c);
    }
}
```

DAR ...

... pot să se facă o sumedenie de erori care se observă doar la execuția programului

```
public class Utilities {
    private static Pair doSet(Pair p) {
        Ceasornicar om = (Ceasornicar)p.getFirst();
        ClockType aparat = (ClockType)p.getSecond();
        om.regleaza(aparat);
        return p;
    }
    public static void createSetAndDisplay(Clock aparat) {
        Ceasornicar om = new Ceasornicar();
        Pair p = new Pair(om, aparat);
        Pair res = doSet(p);
        ClockType c = (ClockType)res.getSecond();
        System.out.println(c);
    }
    public static void iWouldLikeToKnowAtCompileTime() {
        Pair p1 = new Pair(new Integer(5),new Integer(6));
        doSet(p1); //Eroare de execuție
        Pair p2 = new Pair(new Ceasornicar(),new Clock());
        doSet(p2);
        Pair p3;
        p3 = p2;   //Riscant, dacă elementele perechii
        doSet(p3); //nu sunt de tipurile corespunzătoare ?
        p3 = p1;   //Riscant, dacă elementele perechii
                   //nu sunt de tipurile corespunzătoare ?
        doSet(p3); //Eroare de execuție
    }
}
```

DAR ...

... pot să se facă o sumedenie de erori care se observă doar la execuția programului

”Soluția”

```
public class Utilities {
 private static PairCeasornicarClockType doSet(PairCeasornicarClockType p) {
   Ceasornicar om = p.getFirst();
   ClockType aparat = p.getSecond();
   om.regleaza(aparat);
   return p;
 }
 public static void createSetAndDisplay(Clock aparat) {
   Ceasornicar om = new Ceasornicar();
   PairCeasornicarClockType p =
      new PairCeasornicarClockType(om,aparat);
   PairCeasornicarClockType res =  doSet(p);
   ClockType c = res.getSecond();
   System.out.println(c);
 }
 public static void iWouldLikeToKnowAtCompileTime() {
   PairIntegerInteger p1 =
      new PairIntegerInteger(new Integer(5),new Integer(6));
   doSet(p1);//Eroare de compilare
   PairCeasornicarClockType p2 =
      new PairCeasornicarClockType(
            new Ceasornicar(),new Clock());
   doSet(p2);
   PairCeasornicarClockType p3;
   p3 = p2;
   doSet(p3);
   p3 = p1; //Eroare de compilare
   doSet(p3);
  }
}
```

```
public class PairCeasornicarClockType {
 private Ceasornicar p1; ClockType p2;
 public Pair(Ceasornicar p1, ClockType p2) {
   this.p1 = p1; this.p2 = p2;
 }
 public void setFirst(Ceasornicar p1) {this.p1 = p1;}
 public Ceasornicar getFirst() {return p1;}
 public void setSecond(ClockType p2) {this.p2 = p2;}
 public ClockType getSecond() {return p2;}
}
```

```
public class PairIntegerInteger {
 private Integer p1, p2;
 public Pair(Integer p1, Integer p2) {
   this.p1 = p1; this.p2 = p2;
 }
 public void setFirst(Integer p1) { this.p1 = p1; }
 public Integer getFirst() { return p1; }
 public void setSecond(Integer p2) { this.p2 = p2; }
 public Integer getSecond() { return p2; }
}
```

și multe altele ...

Soluția(I)

parametrii de tip

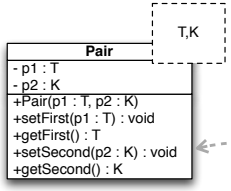
```
public class Pair<T,K> {
    private T p1;
    private K p2;
    public Pair(T p1, K p2) {
        this.p1 = p1;
        this.p2 = p2;
    }
    public void setFirst(T p1) {
        this.p1 = p1;
    }
    public T getFirst() {
        return p1;
    }
    public void setSecond(K p2) {
        this.p2 = p2;
    }
    public K getSecond() {
        return p2;
    }
}
```

tip/clasă generică

argument de tip ex. Integer

Pair<Integer, Integer>
Pair<Integer,Clock>
...

Tipuri parametrizate
- sau “incovări” ale tipului generic



În UML

Crearea de obiecte ale unui tip parametrizat

new *Pair<Integer, Integer>* ();

Soluția(II)

```
public class ClockCommand {
    public static Pair<Ceasornicar,ClockType>
doSet(Pair<Ceasornicar,ClockType> p) {
        Ceasornicar om = p.getFirst();
        ClockType aparat = p.getSecond();
        om.regleaza(aparat);
        return p;
    }
    public static void createSetAndDisplay(Clock aparat) {
        Ceasornicar om = new Ceasornicar();
        Pair<Ceasornicar,ClockType> p =
new Pair<Ceasornicar,ClockType>(om, aparat);
        Pair<Ceasornicar,ClockType> res = doSet(p);
        ClockType c = res.getSecond();
        System.out.println(c);
    }
    public static void iWouldLikeToKnowAtCompileTime() {
        Pair<Integer,Integer> p1 =
 new Pair<Integer,Integer>(new Integer(5),new Integer(6));
        doSet(p1);//Eroare compilare
        Pair<Ceasornicar, ClockType> p2 =
new Pair<Ceasornicar, ClockType>(
new Ceasornicar(),new Clock());// Putem pune subtipuri
        doSet(p2);
        Pair<Ceasornicar, ClockType> p3;
        p3 = p2;
        doSet(p3);
        p3 = p1; //Eroare compilare
        doSet(p3);
    }
}
```

```
public class Pair<T,K> {
    private T p1;
    private K p2;
    public Pair(T p1, K p2) {
        this.p1 = p1;
        this.p2 = p2;
    }
    public void setFirst(T p1) {
        this.p1 = p1;
    }
    public T getFirst() {
        return p1;
    }
    public void setSecond(K p2) {
        this.p2 = p2;
    }
    public K getSecond() {
        return p2;
    }
}
```

Implements/Extends (I)

```
public interface PairIntereface<T,K> {
    public T getFirst();
    public K getSecond();
}
```

```
public class Pair<T,K>
implements PairIntereface<T, K>{
    private T p1;
    private K p2;
    public Pair(T p1, K p2) {
        this.p1 = p1;
        this.p2 = p2;
    }
    public void setFirst(T p1) {
        this.p1 = p1;
    }
    public T getFirst() {
        return p1;
    }
    public void setSecond(K p2) {
        this.p2 = p2;
    }
    public K getSecond() {
        return p2;
    }
}
```

```
public class Triple<T,K,Z> extends Pair<T,K> {
    private Z p3;
    public Triple(T p1, K p2, Z p3) {
        super(p1, p2);
        this.p3 = p3;
    }
    public void setThird(Z p3) {
        this.p3 = p3;
    }
    public Z getThird() {
        return p3;
    }
}
```

Implements/Extends (II)

```
public class MySpecialBooleanPair implements PairIntereface<Boolean, Boolean> {
    private boolean b1,b2;
    public MySpecialBooleanPair(boolean b1, boolean b2) {
        this.b1 = b1;
        this.b2 = b2;
    }
    public Boolean getFirst() {
        return b1;
    }
    public Boolean getSecond() {
        return b2;
    }
}
```

```
public class TripleWith2Integers extends Pair<Integer,Integer> {
    private Integer p3;
    public TripleWith2Integers(Integer p1, Integer p2, Integer p3) {
        super(p1, p2);
        this.p3 = p3;
    }
...
}
```

Polimorfism (de subtip)

```
PairIntereface<Integer,Integer> a;
a = new Pair<Integer, Integer>(new Integer(5), new Integer(6));
System.out.println(a);
a = new Triple<Integer,Integer,Integer>(new Integer(5), new Integer(6), new Integer(8));
System.out.println(a);
a = new TripleWith2Integers(new Integer(0), new Integer(9), 0);
System.out.println(a);
a = new MySpecialBooleanPair(false, false); //Eroare de compilare
```

Cât timp nu variază argumentele de tip, relația supertip/subtip se menține cu toate implicațiile de rigoare

Multe limitări

a. Nu putem folosii tipuri primitive ca argumente de tip

ex. Pair<int,int> - eroare compilare (soluție: clase înfășurătoare ex. Integer)

b. Nu putem crea instanțe de-a unui parametru de tip

ex. new T(); - eroare compilare (soluție: prin reflexion)

c. Nu putem crea tablouri de-a unui parametru de tip

ex. new T[...]; - eroare compilare (dar putem declara referințe la tablouri T[] x;)
(soluție: folosim colecții/containere)

d. Nu putem folosii instanceof

ex. x instanceof T - eroare compilare (soluție: prin reflexion)

... (vezi următorul slide)

Datorate de multe ori de modul în care genericitatea este implementată în Java (prin erasure - nu există / nu se menține informație legată de ce tipuri ia un parametru de tip); în alte limbaje genericitatea este implementată prin alte modalități mai puternică

Bounded type parameters (I)

```
public class NamedGenericPair<T,K> {
    private T p1;
    private K p2;
    public NamedGenericPair(T p1, K p2) {
        this.p1 = p1;
        this.p2 = p2;
    }
    public void setFirst(T p1) {
        this.p1 = p1;
    }
    public T getFirst() {
        return p1;
    }
    public void setSecond(K p2) {
        this.p2 = p2;
    }
    public K getSecond() {
        return p2;
    }
    public String toString() {
        return "(" + p1.toString() + "," + p2.toString() + ")";
    }
}
```

pe o variabilă având ca tip un parametru de tip se pot apela numai metode din Object

Bounded type parameters (II)

```
public class NamedGenericPair<T extends NamedEntity,K extends NamedEntity> {
    private T p1;
    private K p2;
    public NamedGenericPair(T p1, K p2) {
        this.p1 = p1;
        this.p2 = p2;
    }
    public void setFirst(T p1) {
        this.p1 = p1;
    }
    public T getFirst() {
        return p1;
    }
    public void setSecond(K p2) {
        this.p2 = p2;
    }
    public K getSecond() {
        return p2;
    }
    public String toString() {
        return "(" + p1.getName() + ":" + p1.toString() + "," + p2.getName() + ":" + p2.toString() + ")";
    }
}
```

```
public interface NamedEntity {
   public String getName();
}
```

În general forma e T extends Tip1 & Tip2 & ... & Tn (cel mult o clasă și se pune prima)

```
Se pune o limita superioara la argumentele de tip: să fie tipul NamedEntity sau subtip de-al său
NamedGenericPair<Integer,Integer> x; //Eroare compilare
NamedGenericPair<NamedEntity,NamedEntity> y;
```

B

Câteva elemente mai avansate ...

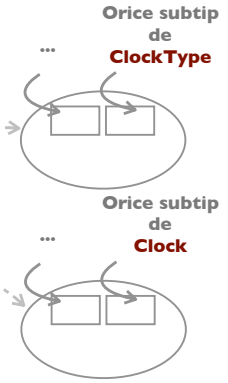
(lucrurile nu sunt chiar așa triviale)

Lucruri "evidente" și unele "ciudățenii" la o primă vedere

```
public class Atentie {
 public static void main(String[] args) {
  Ceasornicar person = new Ceasornicar();
  Clock c = new Clock();
  EnhancedClock ec = new EnhancedClock();

  Pair<Ceasornicar,ClockType> a = new Pair<Ceasornicar,ClockType>(person,c);
  a.setSecond(ec); //Se accepta orice referinta de tipul dar ori de un subtip de-al sau

  Pair<Ceasornicar,Clock> b = new Pair<Ceasornicar,Clock>(person,c);
  b.setSecond(c);
  b.setSecond(ec);//Eroare compilare - la b al doilea element e neaparat tip/subtip de-al
                  //lui Clock
  a = b; //Eroare compilare !!!
  a.setSecond(ec);
 }
}
```



a - este o referință spre un obiect pereche ce poate conține un Ceasornicar și orice fel de ClockType

b - este o referință spre un obiect pereche ce poate conține un Ceasornicar și un Clock

Explicație: Dacă s-ar permite atribuirea în obiectul referit de a și b am putea pune ca al doilea element un EnhancedClock (linia următoare) deși obiectul NU poate ține un EnhancedClock (el poate ține numai Clock) și deci ar apare o eroare de execuție

Quiz

Este ok să spunem ?

Pair<Ceasornicar,ClockType> x = new Pair<Ceasornicar,Clock>(person,c);

NU!

x - este o referință spre un obiect pereche ce poate conține un Ceasornicar și orice fel de ClockType

... iar obiectul creat poate conține numai Ceasornicar și Clock!!!

Wildcard limitat superior

```
public class Atentie {
 public static void main(String[] args) {
  Ceasornicar person = new Ceasornicar();
  Clock c = new Clock();
  EnhancedClock ec = new EnhancedClock();

  Pair<Ceasornicar,? extends ClockType> a = new Pair<Ceasornicar,Clock>(person,c);
  a.setSecond(c); //Eroare compilare pentru ca nu se mai poate garanta la compilare
                  //ca situatia anterioara nu apare la rulare (ca obiectul referic chiar poate tine un Clock
  Pair<Ceasornicar,Clock> b = new Pair<Ceasornicar,Clock>(person,c);

   a = b; //E ok.
   a.setSecond(ec);//Eroare compilare ca mai sus; se poate face numai a.setSecond(null);
 }
}
```

a - este o referință spre un obiect pereche ce poate conține un Ceasornicar și orice fel de ClockType SAU un obiect pereche ce poate conține un Ceasornicar și orice fel de Clock SAU ...

există și wildcard limitat inferior :)

Dacă avem `class G<T> {}` și o referință cu `G<? extends Something> g;` la apelarea metodelor ce au un argument T putem da doar null ca valoare la acel argument