Java XI. Interfész

Tág értelemben egy osztály interfészének nevezzük az osztály azon metódusainak halmazát, melyek a külvilággal kommunikálnak (azaz publikusak).

Fontos megjegyezni, hogy ezen metódusok csak deklarálva vannak, implementációjuk nincs, azaz absztraktak.

Interfésznek nevezzük azon osztályokat, melyek minden metódusa absztrakt. Az absztrakt metódusok jelölésére az abstract kulcsszót használjuk.

A java nyelvben egy interfész deklarálásához az interface kulcsszót használjuk:

public interface Rational {

int getNumerator();

int getDenominator();

Rational times(Rational that);

Rational times(int that);

Rational plus(Rational that);

...

}

Az interfészekben szereplő tagok implicit módon publikusak és absztraktak.

public interface Rational {

abstract public int getNumerator();

abstract public int getDenominator();

abstract public Rational times(Rational that);

abstract public Rational times(int that);

abstract public Rational plus(Rational that);

...

}

Tehát ez a két interfész ekvivalens.

Egy interfész tartalma pontosabban:

* **példánymetódus deklaráció (esetleg default implementáció)**
* konstansok
* osztályszintű metódusok
* beágyazott típusok

# Megvalósítás:

public interface Rational {

int getNumerator();

int getDenominator();

Rational times(Rational that);

}

Egy interfész megvalósításának nevezzük azt az osztályt, mely implementálja az interfész metódusait.

Ennek jelzésére használjuk az *implements* kulcsszót:

public class Fraction implements Rational {

private final int numerator, denominator;

public Fraction(int numerator, int denominator){ ... }

public int getNumerator(){return numerator;}

public int getDenominator(){return denominator;}

public Rational times(Rational that){ ... }

}

Egy interfésznek akár több megvalósítása is lehet:

public class Simplified implements Rational {

...

public int getNumerator(){ ... }

public int getDenominator(){ ... }

Rational times(Rational that){ ... }

}

És egy osztály akár több interfészt is megvalósíthat:

public class List implements Cloneable, Iterable<E>, ... { ...

Az osztályokhoz hasonlóan az interfészek is lehetnek generikusak:

package java.util;

public interface List<T> {

T get(int index);

void set(int index, T item);

void add(T item);

...

}

package java.util;

public class ArrayList<T> implements List<T>{

public ArrayList(){ ... }

public T get(int index){ ... }

...

}

# Interfészek és az altípusosság:

class Fraction implements Rational { ... }

class ArrayList<T> implements List<T> { ... }

class LinkedList<T> implements List<T> { ... }

Az interfészek megvalósításával altípus jön létre. A Rational interfész Fraction megvalósítása a Rational osztály egy altípusa. Ez alapján észrevehetjük az interfészek valójában olyan típusok, melyeket nem lehet példányosítani.

List<String> names;

Ezek alapján, a Liskov helyettesíti elv értelmében egy adott interfész helyettesíthető egy azt megvalósító osztállyal. Például:

List<String> names = new ArrayList<>();

Fontos megjegyezni, hogy az interfész nem példányosítható:

List<String> names = new List<String>(); // fordítási hiba

Ajánlott a következő programozási stílus követése:

List<String> names = new ArrayList<>();

(Deklarálás interfésszel, példányosítás osztállyal, megjegyzés: itt a tényleges típus az adott osztály nem az interfész [mivel az nem példányosítható] lesz)

# Speciális jelentésű interfészek:

Vannak bizonyos interfészek, melyek megvalósítása szükséges ahhoz, hogy bizonyos nyelvi funkciók alkalmazhatóak legyenek az adott típusra.

* **for-each** ciklus olyan adatszerkezeteket képes bejárni, melyek megvalósítják a **java.lang.Iterable<T>** interfészt.
* megnyitható erőforrásként (**try-with-resources**): **java.lang.AutoClosable**
* elérhető a **sekély másolás**: **java.lang.Cloneable**
* objektumszerializáció (majd haladó javán): java.io.Serializable

## Az Iterable interfész:

Egy típus iterálható, ha az biztosít egy olyan objektumot (ún. *iterátort*), mellyel bejártható. Azaz az osztály megvalósítja az

public interface Iterable<T> {

java.util.Iterator<T> iterator();

...

}

Iterable<T> interfészt. Ehhez szükséges a

public interface Iterator<T> {

boolean hasNext();

T next();

...

}

java.util.Iterator interfész megvalósítása.

Egy elképzelt megvalósítás:

package java.util;

public class ArrayList<T> implements Iterable<T> {

Object[] data;

int size = 0;

...

public Iterator<T> iterator(){ return new ALIterator<>(this); }

}

class ALIterator<T> implements Iterator<T> {

private final ArrayList<T> theArrayList;

private int index = 0;

ALIterator( ArrayList<T> al ){ theArrayList = al; }

public boolean hasNext(){ return index < theArrayList.size; }

@SuppressWarnings("unchecked") public T next(){

return (T)theArrayList.data[index++];

}

}

Az iterátor objektumok nagy előnye, (azzal szemben, ha maga az obejktum végezné a bejárást), hogy egyszerre több iterátor is végezhet felsorolást egy objektumon. Ezt az előnyt mutatja be a következő példa is.Egy bonyolultabb példa:

A pairs metódus lényegében egy adatszerkezet önmagával vett Descartes-szorzatát adja meg, azaz az összes olyan rendezett párt állítja elő, melynek első és második tagja is egy elem az adatszerkezetből.

(kis diszkrét matek kitérő )

List<Pair<Integer,Integer>> pairs( List<Integer> ns ){

List<Pair<Integer,Integer>> ps = new LinkedList<>();

Iterator<Integer> it = ns.iterator();

while(it.hasNext()){

Integer item = it.next();

Iterator<Integer> it2 = ns.iterator();

while(it2.hasNext()){

ps.add(new Pair<Integer,Integer>(item,it2.next()));

}

}

return ps;

}

A példában megfigyelhető:

* generikus osztályok és interfészek használata
* az iterátorok használata
* az interfészek és megvalósításuk altípusossága