Types abstraits

Types abstraits

Abstract Data Type — ADT

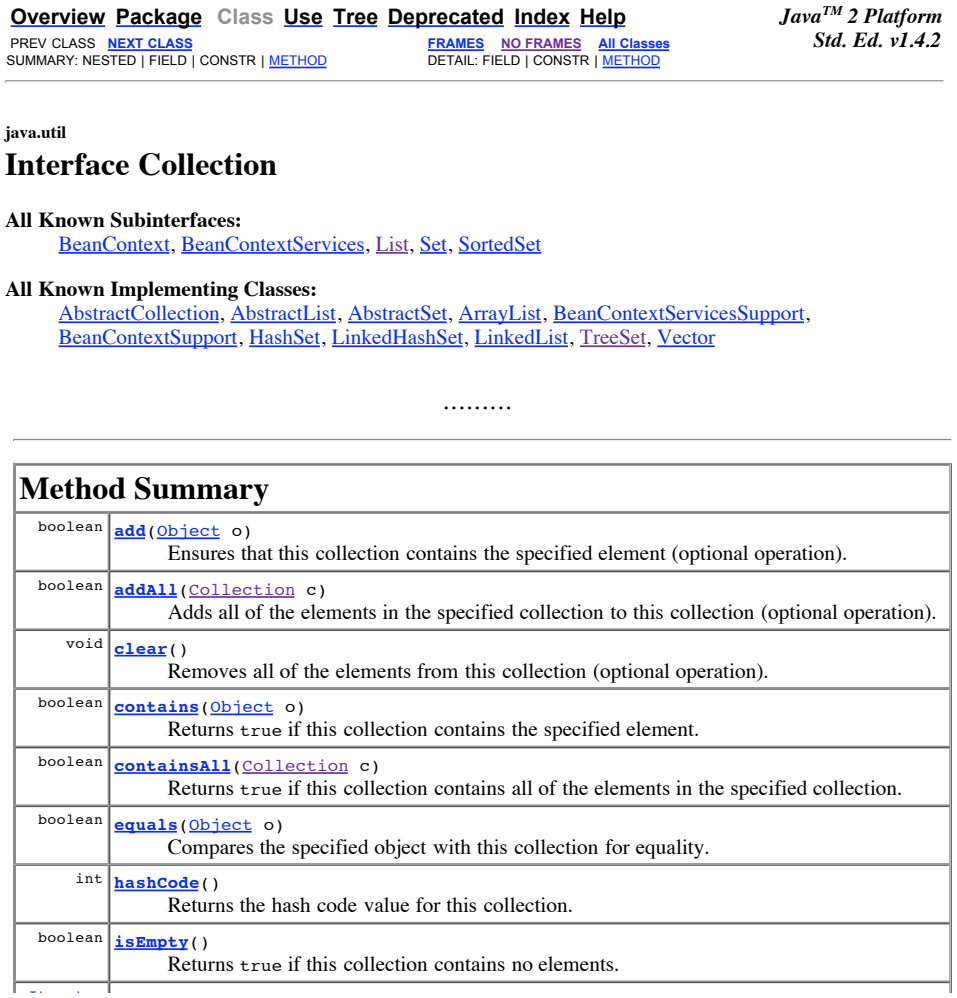
Type abstrait : un ensemble d'objets et un ensemble d'opérations

But : comparaison d'implantations qui offrent la même fonctionnalité

Pensez, p.e., à des `interfaces` de Java : signature des méthodes+sémantique (en Javadoc) mais plusieurs implantations possibles

Notez que `interface` ne définit que le syntaxe des opérations donc il ne correspond pas à la notion de l'ADT

Java interface



Overview Package Class Use Tree Deprecated Index Help

Java™ 2 Platform Std. Ed. v1.4.2

PREV CLASS NEXT CLASS | FRAMES NO FRAMES All Classes

SUMMARY: NESTED | FIELD | CONSTR | METHOD | DETAIL: FIELD | CONSTR | METHOD

java.util

Interface Collection

All Known Subinterfaces:

BeanContext, BeanContextServices, List, Set, SortedSet

All Known Implementing Classes:

AbstractCollection, AbstractList, AbstractSet, ArrayList, BeanContextServicesSupport, BeanContextSupport, HashSet, LinkedHashSet, LinkedList, TreeSet, Vector

.....

Method Summary

boolean	**add**(Object o) Ensures that this collection contains the specified element (optional operation).
boolean	**addAll**(Collection c) Adds all of the elements in the specified collection to this collection (optional operation).
void	**clear**() Removes all of the elements from this collection (optional operation).
boolean	**contains**(Object o) Returns true if this collection contains the specified element.
boolean	**containsAll**(Collection c) Returns true if this collection contains all of the elements in the specified collection.
boolean	**equals**(Object o) Compares the specified object with this collection for equality.
int	**hashCode**() Returns the hash code value for this collection.
boolean	**isEmpty**() Returns true if this collection contains no elements.

pas un ADT mais une abstraction . . .

Nombres naturels

Objets : $N = \{0, 1, \ldots\}$

Opérations :
add : $N \times N \mapsto N$;
succ : $N \mapsto N$;
eq ? : $N \times N \mapsto \{\text{vrai}, \text{faux}\}$;
less : $N \times N \mapsto \{\text{vrai}, \text{faux}\}$

Beaucoup d'implantations possibles :
chaînes de chiffres (base 10, `String`), factorisation en primes (`int[]`), $i \in N$ representé par un `char[]` de longueur i

Grande différence dans l'efficacité de l'exécution d'opérations

Nombres naturels — implantation 1

```
public class N {
   private int valeur ;  // implantation basé sur le type int
   public N(int v){this.valeur = v;}
   public N add(N a, N b){
      return new N(a.valeur+b.valeur);
   }
   ...
}
```

implantation facile

problème de représentation (si $v \geq 2^{31}$)

Nombres naturels — implantation 2

```
public class N {
   private String valeur ;  // chaîne de chiffres
   public N(String v){this.valeur = v;}
   public N add(N a, N b){
      // ... l'«algorithme» d'addition sur papier
   }
   ...
}
```

implantation un peu plus compliqué

pas de problème de représentation (au moins jusqu'à $10^{2^{31}}$ — longueur max de `String` est `Integer.MAX_VALUE`)

Nombres naturels — implantation 3

factorisation : $312 = 2^3 \cdot 3 \cdot 13 = 2^3 \cdot 3^1 \cdot 5^0 \cdot 7^0 \cdot 11^0 \cdot 13^1$ representé par `int[] fact = {3,1,0,0,0,1}`

```
public class N {
   private int[] fact ;  //  factorisation en primes
   public N(String in){
    // factorisation
   }
   public N add(N a, N b){
      // ... algorithme d'addition + factorisation du résultat
   }
   ...
}
```

implantation assez compliqué

par contre, pgcd est facile à calculer sans l'algo d'Euclid :
si $a = \sum_i p_i^{a_i}$ et $b = \sum_i p_i^{b_i}$, alors $\mathsf{pgcd}(a,b) = \sum_i p_i^{\min\{a_i,b_i\}}$.
(avec les nombres premiers $p_1 = 2, p_2 = 3, p_3 = 5, \ldots$)