Programare avansată

Tipuri generice

Colecții de date

Problema

- “Construiți o structură de date:
  - *o stivă de date, o listă înlănțuită, un vector,*
  - *un graf, un arbore, etc.”*
- Care este tipul de date pe care îl vom folosi pentru reprezentarea elementelor?

Structură omogenă	Structură eterogenă

`class Stack {`
`  private int[] items;`
`  public void push (int item) { ...}`
`  public int peek() { … }`
`}`
`…`
`Stack stack = new Stack();`
`stack.push(100);`
`stack.push(200);`
~~`stack.push("Hello World!");`~~
	`class Stack {`
	`  private Object[] items;`
	`  public void push (Object item) {...}`
	`  public Object peek() { … }`
	`}`
	`…`
	`Stack stack = new Stack();`
	`stack.push(100);`
	`stack.push(new Rectangle());`
	`stack.push("Hello World!");`
	`String s = (String) stack.peek;`

Tipuri *generice*

- Permit parametrizarea tipurilor de date (clase și interfețe), parametrii fiind tot tipuri de date

```
public Stack<String> { … }
```

- Control sporit asupra lucrului cu tipuri de date

```
stack.push(new Rectangle());
```

verificarea tipului de date se face la compilare!

- Eliminarea operatorului de cast

```
String s = (String) stack.peek();
```

Definirea unui tip generic

```
class ClassName<T1, T2, ..., Tn> { … }
```

sau

```
interface IName<T1, T2, ..., Tn> { … }
```

```
/**
 * Versiunea generica a clasei Stack
 * @param <E> tipul elementelor
 */
public class Stack<E> {

    // E reprezinta un tip generic de date
    private E[] items;

    public void push(E item) { ... }
    public E peek() { .. }
}
```

Convenții de numire a tipurilor

- E - Element (folosit intensiv de Java Collections Framework)
- K - Key
- N - Number
- T - Type
- V - Value
- S,U,V etc. - 2nd, 3rd, 4th types

```
public class Node<T> { … }
public interface Pair<K, V> { … }
public class PairImpl<K, V> implements Pair<K, V> {...}
```

Instanțierea tipurilor generice

- Invocarea generică a unui tip

```
Stack<String> stack = new Stack<String>();

Pair<Integer,String> pair =

          new PairImpl<Integer,String>(0, "ab");
Stack<Node<Integer>> nodes = new Stack<Node<Integer>>();
```

- Operatorul *diamond* <>

```
Stack<String> stack = new Stack<>();

Pair<Integer,String> pair = new PairImpl<>(0, "ab");

Stack<Node<Integer>> nodes = new Stack<>();
```

Metode *generice*

Sunt metode care introduc parametri proprii, alții decât cei ai clasei din care fac parte

```
public class Util {
  public static <T> int countNullValues(T[] anArray) {
    int count = 0;
    for (T e : anArray)
        if (e == null) {
            ++count;
        }
    return count;
  }
}
Util.countNullValues(new String[]{"a", null, "b"});
Util.countNullValues(new Integer[]{1, 2, null, 3, null});
```

Tipuri generice delimitate *(bounded)*

```
class D <T extends A & B & C> { /* ... */ }
```

```
public class Node<T extends Number> {
    private T t;
    public void set(T t) { this.t = t; }
    public T get() { return t; }

    // Metoda generica
    public <U extends Integer> void inspect(U u){
        System.out.println("T: " + t.getClass().getName());
        System.out.println("U: " + u.getClass().getName());
    }

    public static void main(String[] args) {
        Node<Double> node = new Node<>();
        node.set(12.34);                //OK
        node.inspect(1234);             //OK
        node.inspect(12.34);            //compile error!
        node.inspect("some text");      //compile error!
    }
}
```

Tipuri generice delimitate *(bounded)*

```
class D <T extends A & B & C> { /* ... */ }
```

```
public class Node<T extends Number> {
    private T t;
    public void set(T t) { this.t = t; }
    public T get() { return t; }

    // Metoda generica
    public <U extends Integer> void inspect(U u){
        System.out.println("T: " + t.getClass().getName());
        System.out.println("U: " + u.getClass().getName());
    }

    public static void main(String[] args) {
        Node<Double> node = new Node<>();
        node.set(12.34);                //OK
        node.inspect(1234);             //OK
        node.inspect(12.34);            //compile error!
        node.inspect("some text");      //compile error!
    }
}
```

Moștenirea
"is a"

- ✔ `Integer extends Object`
- ✔ `Integer extends Number`
- **~~`Stack<Integer> extends Stack<Object>`~~**
- **~~`Stack<Integer> extends Stack<Number>`~~**

Fie două tipuri de date A și B (de exemplu, *Number* și *Integer*). Indiferent dacă A sau B sunt în relație de moștenire sau nu, *MyClass<A>* nu este în nici o relație cu *MyClass<B>.*

Părintele comun al claselor *MyClass<A>* și *MyClass<B>* este *Object*.

"This is a common misunderstanding when it comes to programming with generics, but it is an important concept to learn "

Wildcard-uri

- Delimitate superior

```
public double sumOfList(List<? extends Number> list) {
    double s = 0.0;
    for (Number n : list)
        s += n.doubleValue();
    return s;
}
```

- Fără delimitare

```
public void printList(List<?> list) {
    for (Object elem: list)
        System.out.print(elem + " ");
}
```

- Delimitate inferior

```
public void addNumbers(List<? super Integer> list) {
    for (int i = 1; i <= 10; i++) {
        list.add(i);
    }
}
```

Intrebări și exerciții

- *The Java Tutorials*, Generics

  http://docs.oracle.com/javase/tutorial/java/generics/QandE/generics-questions.html

Colecții de date
Java Collections Framework

- O *colecție* este un obiect care grupează mai multe elemente într-o singură unitate.
- *Vetori, liste înlănțuite, stive, mulțimi matematice, dicționare, tabele hash, etc.*
- Reutilizarea codului
- Reducerea efortului de programare
- Creșterea vitezei și calității unei aplicații
- Algoritmi *polimorfici*
- Folosesc *tipuri generice*

Arhitectura colecțiilor

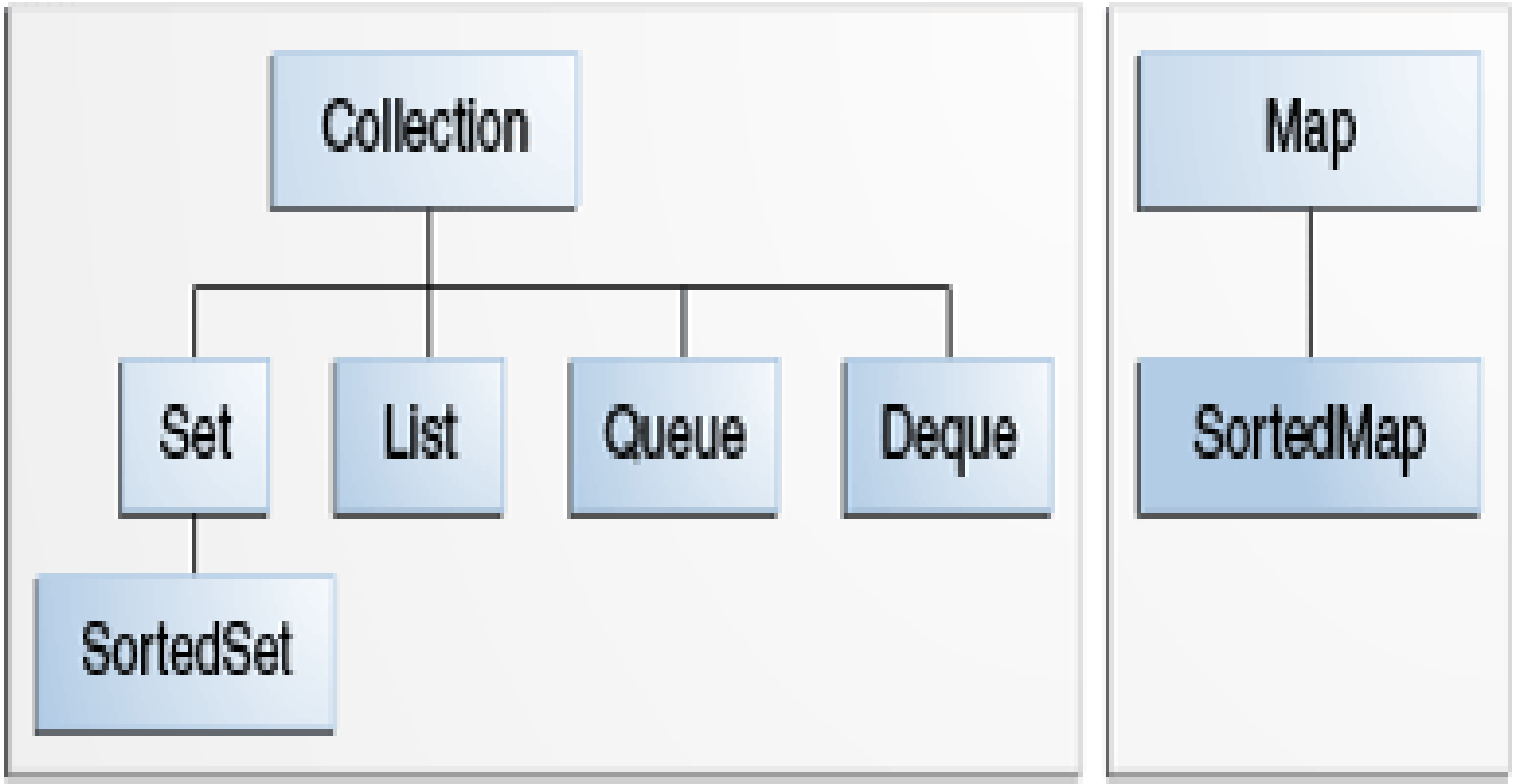
- Interfață
- Clasă abstractă
- Implementări concrete

List

AbstractList

ArrayList
LinkedList
Vector

Interfețe

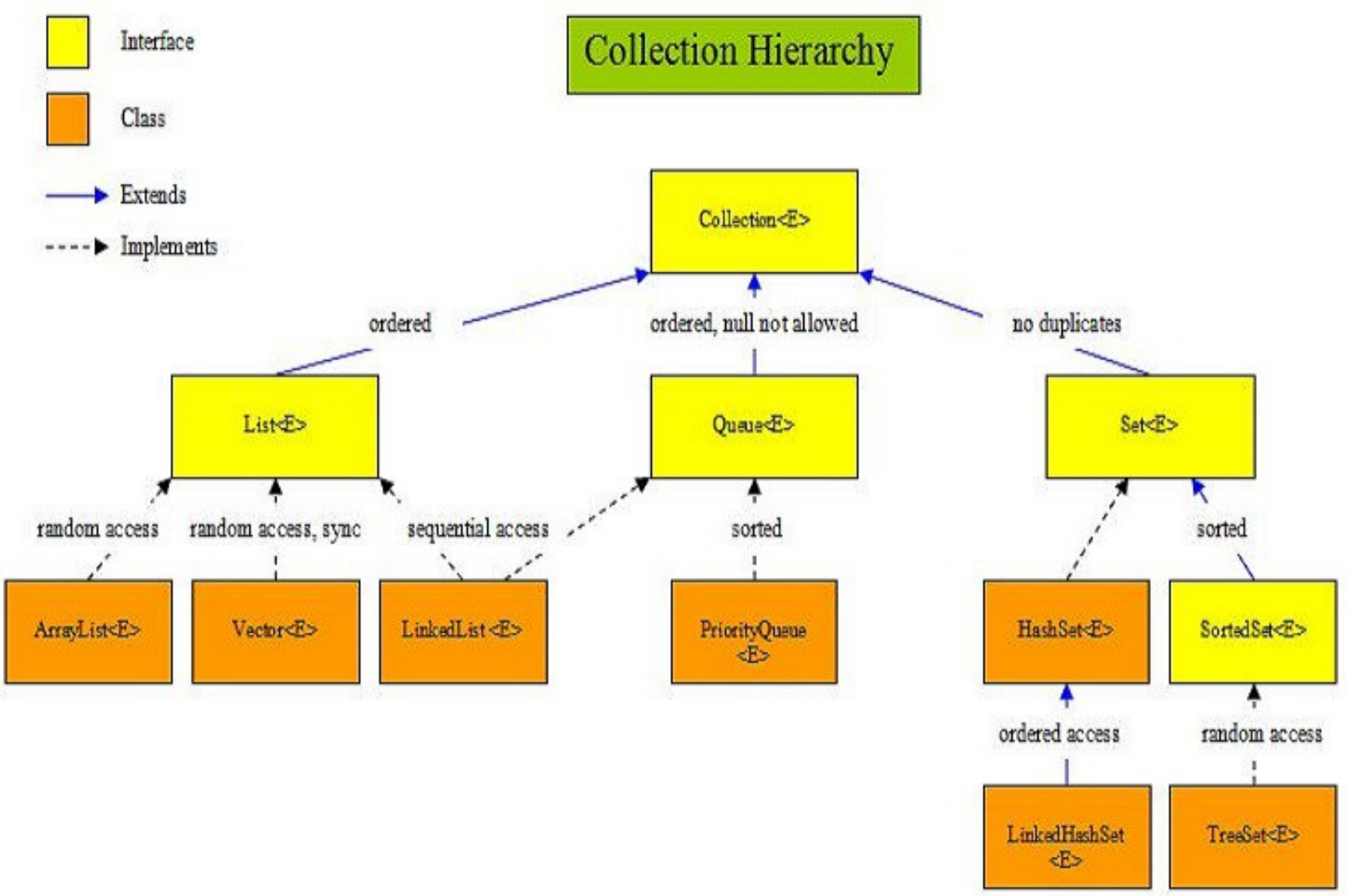


Implementări

Interfața	Hash	Array	Tree	Linked	Hash+Linked
Set	HashSet		TreeSet		LinkedHashSet
List		ArrayList Vector		LinkedList	
Queue					
Deque		ArrayDeque		LinkedDeque	
Map	HashMap Hashtable		TreeMap		LinkedHashMap

```
Set set = new HashSet(); --raw generic type (Object)
ArrayList<Integer> list = new ArrayList<>();
List<Integer> list = new ArrayList<>();
List<Integer> list = new LinkedList<>();
List<Integer> list = new Vector<>();
Map<Integer, String> map = new HashMap<>();
```

Ierarhia colecțiilor



Parcurgerea colecțiilor

- Colecții indexate

```
for (i=0; i < lista.size(); i++ ) {
    System.out.println( lista.get(i) );
}
```

- Iteratori și enumerări

```
for (Iterator it = colectie.iterator(); it.hasNext(); ) {
  System.out.println(it.next());
  it.remove();
}
```

- *for-each*

```
List<Student> studenti = new ArrayList<Student>();
...
for (Student student : studenti) {
    student.setNota(10);
}
```

Algoritmi polimorfici

java.util.Collections

- sort
- shuffle
- binarySearch
- reverse
- fill
- copy
- min
- max
- swap
- enumeration
- unmodifiable*TipColectie*

```
List<String> immutablelist = Collections.unmodifiableList(list);
immutablelist.add("Oops...?!");
```

- synchronized*TipColectie*

Ce DesignPattern?

ArrayList sau *LinkedList*?

```
import java . util .*;
public class Test {
  final static int N = 100000;
  public static void testAdd ( List lst) {
    long t1 = System . currentTimeMillis ();
    for (int i=0; i < N; i++)
      lst.add (new Integer (i));
    long t2 = System . currentTimeMillis ();
    System . out. println ("Add: " + (t2 - t1));
}
  public static void testGet ( List lst) {
    long t1 = System . currentTimeMillis ();
    for (int i=0; i < N; i++)
      lst.get(i);
    long t2 = System . currentTimeMillisA ();
    System . out. println ("Get: " + (t2 - t1));
  }
  public static void testRemove ( List lst ) {
    long t1 = System . currentTimeMillis ();
    for (int i=0; i < N; i++)
      lst.remove(0);
    long t2 = System . currentTimeMillis ();
    System . out. println (" Remove : " + (t2 - t1));
  }
  public static void main ( String args []) {
    List lst1 = new ArrayList ();
    testAdd ( lst1 ); testGet ( lst1 ); testRemove ( lst1 );
    List lst2 = new LinkedList ();
    testAdd ( lst2 ); testGet ( lst2 ); testRemove ( lst2 );
  }
}
```

	ArayList	LinkedList
add	0.12	0.14
get	0.01	87.45
renmove	12.05	0.01

Concluzia: alegerea unei anumite implementări depinde de natura problemei ce trebuie rezolvată.