Clase incluse

Programare Orientată pe Obiecte

Nested classes vs *inner classes*

Inner classes:

- datorită relaţiei pe care o au cu clasa exterioară (depind de o instanţă a acesteia).
- Cuprind:
  - Clasele interne normale
  - Clasele interne anonime
  - Clasele interne (locale) blocurilor şi metodelor

Nested classes :

- definirea unei clase în interiorul altei clase
- cuprinde atât *inner classes* cât şi *clasele statice interne*.
- clasele statice interne: *static nested classes* (nu *static inner classes)*.

Clase interne statice

- Clasele interne pot avea modificatorul ***static*** (clasele exterioare nu pot fi statice!)
- Putem obține o referință către o clasă internă statică fără a avea nevoie de o instanță a clasei exterioare!
- Diferența clase interne statice şi cele nestatice: **clasele nestatice ţin legătura cu obiectul exterior** în vreme ce **clasele statice nu păstrează această legătură**.

Pentru clasele interne statice:

- nu avem nevoie de un obiect al clasei externe pentru a crea un obiect al clasei interne
- nu putem accesa câmpuri nestatice ale clasei externe din clasă internă (nu avem o instanţă a clasei externe)

Exemplu

```
class Outer {
    public int data= 9;

    class NonStaticInner {
        private int i = 1;
        public int value() {
         return i + data;   //Outer.this.data;
                        // OK, acces membru  clasă exterioară
        }
    }
     static class StaticInner {
        public int k = 99;
         public int value() {
            k += data; // EROARE,  acces membru nestatic
         return k;
     }
    }
}
```

Exemplu

```
public class Test {
  public static void main(String[] args) {
     Outer out                  = new Outer ();
    Outer.NonStaticInner nonSt = out.new NonStaticInner();
    // instantiere CORECTA pt o clasa nestatica

    Outer.StaticInner st       = out.new StaticInner();
    // instantiere INCORECTA a clasei statice

    Outer.StaticInner st2      = new Outer.StaticInner();
     // instantiere CORECTA a clasei statice
    }
}
```

De ce clase interne statice?

- Pentru a grupa clasele:
  - dacă o clasă internă statică A.B este folosită doar de A, atunci nu are rost să o facem top-level.

- Dacă avem o clasă internă A.B, o declarăm statică dacă în interiorul clasei B nu avem nevoie de nimic specific instanţei clasei externe A
  - nu avem nevoie de o instanță a acesteia

Interface Segregation Principle (ISP)

Programare Orientată pe Obiecte

Interface Segregation Principle (ISP)

Motivatie

- Proiectare aplicatie: modul abstract care contine mai multe submodule
- Daca modulul e implementat intr-o clasa – abstractizarea va fi o interfata
- Daca vrem sa adaugam un alt modul ce contine doar o parte din submodulele sistemului original – obligatia de a implementa intreaga interfata (scrierea unor dummy methods)
- O astfel de interfata este numita “fat interface” sau “polluted interface”
- **Interface Segregation Principle**: clientii nu trebuie fortati sa implementeze interfetele pe care nu le folosesc

Exemplu

```
// ISP – nu este luat in calcul
// Interfata IWorker - polluted interface

interface IWorker {
        public void work();
        public void eat();
}

class Worker implements IWorker{
        public void work() {
                // ....working
        }
        public void eat() {
                // ..... eating in launch break
        }
}
```

Exemplu - continuare

```
class SuperWorker implements IWorker{
        public void work() {
                //.... working much more
        }

        public void eat() {
                //.... eating in launch break
        }
}
class Manager {
        IWorker worker;

        public void setWorker(IWorker w) {
                worker=w;
        }

        public void manage() {
                worker.work();
        }
}
```

Interface Segregation Principle (ISP)

- In firma apar roboti.
- Cum ii integram?
- Lucreaza dar nu mananca.
- Rezolvare:

  In locul unei interfete mari e de preferat sa definim mai multe interfete mici pe baza grupurilor de metode, fiecare dintre ele corespunzand unui submodul.
- Interfata IWorker trebuie impartita in 2 interfete diferite

Rezolvare - Interface Segregation Principle

```
// interface segregation principle - good example

interface IWorker extends Feedable, Workable { }

interface IWorkable {
	public void work();
}

interface IFeedable{
	public void eat();
}

class Worker implements IWorkable, IFeedable{
	public void work() {
		// ....working
	}

	public void eat() {
		//.... eating in launch break
	}
}
```

Exemplu - continuare

```
class Robot implements IWorkable{
        public void work() {
                // ....working
        }
    }
class SuperWorker implements IWorkable, IFeedable{
        public void work() {
                //.... working much more
        }

        public void eat() {
                //.... eating in launch break
        }
    }
class Manager {
        Workable worker;

        public void setWorker(Workable w) {
                worker=w;
        }

        public void manage() {
                worker.work();
        }
    }
```

Concluzii

Ca orice principiu, principiul impartirii interfetei necesita

- timp si efort in plus la proiectare
- cresterea complexitatii codului.

- Dar rezulta un design flexibil.

Aplicarea sa nu inseamna interfete cu o singura metoda:

- se identifica zonele in care se prevede o extindere a codului in viitor
- experienta si bun simt.

Genericitate

Programare Orientată pe Obiecte

Introducere

- concept nou - JDK 5.0.
- oferă un mijloc de **abstractizare a tipurilor de date**
- util mai ales în ierarhia de colecţii
- din unele puncte de vedere, se poate asemăna cu conceptul de *template* din C++.
- permite parametrizarea tipurilor de date (clase și interfețe), parametrii fiind tot tipuri de date
- Control sporit asupra lucrului cu tipuri de date, verificarea tipului de date se face la compilare
- Eliminarea operatorului de cast
- Scrierea de algoritmi generici

Fără genericitate

Exemplu:

```
List myList = new ArrayList();
myList.add(new Integer(0));
Integer x = (Integer) myList.get(0);
```

Obs: necesitatea operaţiei de cast pentru a identifica corect variabila obţinută din listă.

Dezavantaje:

- Este îngreunată citirea codului
- Apare posibilitatea unor erori la execuţie
  - în momentul în care în listă se introduce un obiect care nu este de tipul *Integer*.

- *Genericitatea* intervine pentru a elimina aceste probleme!

Tipuri generice (parametrizate)

- **Tipizarea elementelor unei colecţii:**
  TipColectie < TipDeDate >

```
// Inainte de 1.5
    List list = new ArrayList();
    list.add(new Integer(123));
    int val = ((Integer)list.get(0)).intValue();

// Dupa 1.5, folosind si autoboxing
    List<Integer> list = new ArrayList<Integer>();
    list.add(123);
    int val = list.get(0);
```

- **Avantaje**: simplitate, control (eroare la compilare vs. ClassCastException)

Exemplu cu genericitate

```
List<Integer> myList = new ArrayList<Integer>();
myList.add(new Integer(0));
Integer x = myList.get(0);
```

- lista nu mai conţine obiecte oarecare, ci poate conţine doar obiecte de tipul *Integer*.
- a dispărut şi cast-ul.
- **verificarea tipurilor este efectuată de compilator**, ceea ce elimină potenţialele erori de execuţie cauzate de eventuale cast-uri incorecte.

Beneficiile utilizării genericităţii:

- îmbunătăţirea lizibilităţii codului
- creşterea gradului de robusteţe

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 parametrilor de tip

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

Exemple:

public class **Node<T>** { … }

public interface **Pair<K, V>** { … }

public class PairImpl<K, V> implements Pair<K, V> {...}

Definirea unor structuri generice simple

- Exemple - definiţia oferită de Java pentru tipurile *List* şi *Iterator*.

```
public interface List<E> {
    void add(E x);
    Iterator<E> iterator();
}
public interface Iterator<E> {
    E next();
    boolean hasNext();
}
```

- Sintaxa <E> - folosită pentru a defini ***tipuri formale*** în cadrul interfeţelor.
- În momentul în care invocăm efectiv o structură generică, ele vor fi înlocuite cu tipurile efective utilizate în învocare.

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* <>

De la versiunea 7, atata timp cat compilatorul poate determina tipul argumentelor din context

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

Instantierea unor structuri generice simple

Exemplu:

```
ArrayList<Integer> myList = new
    ArrayList<Integer>();
Iterator<Integer> it = myList.iterator();
```

- În această situaţie, tipul formal *E* a fost înlocuit (la compilare) cu tipul efectiv *Integer*.

- Observaţie: analogie cu parametrii funcţiilor: se definesc utilizând parametri *formali*, urmând ca, în momentul unui apel, aceşti parametri să fie înlocuiţi cu parametrii *actuali*.

Observaţii

- Deoarece colecţiile sunt construite peste tipul de date Object, metodele de tip *next* sau *prev* ale iteratorilor vor returna tipul Object, fiind responsabilitatea noastră de a face conversie la alte tipuri de date, dacă e cazul!

```
ArrayList<Integer> list=new ArrayList<Integer>();
for (Iterator i = list.iterator(); i.hasNext();) {
    Integer val=(Integer)i.next();
    ...
}
```

sau:

```
ArrayList<Integer> list=new ArrayList<Integer>();
for (Iterator <Integer> i = list.iterator(); i.hasNext();) {
    Integer val=i.next();
    ...
}
```

sau:

```
ArrayList<Integer> list=new ArrayList<Integer>();
for (Integer val : list) {
    ...
}
```

Exemple

```
Collection<String> c = new ArrayList<String>();
c.add("Test");
c.add(2); // EROARE!
Iterator<String> it = c.iterator();
while (it.hasNext()) {
   String s = it.next();
}
```

- O **iterare** obişnuită pe un map de perechi (String, Student) se va face în felul următor:

```
for (Map.Entry<String,Student> entry:
   students.entrySet())
   System.out.println("Media “+ entry.getKey()+“este“
   +entry.getValue().getAverage());
```

- bucla for-each⇔ iteratorul mulţimii de perechi întoarse de entrySet.

Raw Types

= Numele unei clase sau interfete generice fara argumente de tip

Exemplu:

```
public class Box<T> {
        public void set(T t) { /* ... */ }
        // ...
}
```

- Pt a crea un tip parametrizat pentru Box<T>:

  ```
  Box<Integer> intBox = new Box<>();
  ```

- Daca argumentul de tip este omis => un tip de baza (raw type) pt Box<T>:

  ```
  Box rawBox = new Box();
  ```

- Asadar, Box este tipul de baza (raw type) pentru tipul generic Box<T>
- Atentie! Un tip corespunzator unei clase non-generice sau unei interfete nu este un raw type!
- Raw types – apar pentru ca multe din clasele API nu au fost generice pana la versiunea JDK 5.0.
- Cand utilizam raw types, obtinem comportamentul pre-generic!

Raw types

- Pentru compatibilitate, este permisa atribuirea unui tip parametrizat tipului sau de baza:

```
Box<String> stringBox = new Box<>();
Box rawBox = stringBox; // OK
```

- Dar, invers apare warning:

```
Box rawBox = new Box();
        // rawBox is a raw type of Box<T>
Box<Integer> intBox = rawBox;
        // warning: unchecked conversion
```

- Apare warning si daca folosim un raw type pentru a invoca metode generice

```
Box<String> stringBox = new Box<>();
Box rawBox = stringBox;
rawBox.set(8);
        // warning: unchecked invocation to set(T)
```

- Indicatie: A se evita utilizarea raw types.

Genericitatea în subtipuri

Exemplu:

```
List<String> stringList = new ArrayList<String>();
                                        // corect!
List<Object> objectList = stringList;
                                        // operaţie corectă?
```

Presupunem că operaţia e corectă =>

- am putea introduce în *objectList* orice fel de obiect, nu doar obiecte de tip String =>
- Potenţiale erori de execuţie.
- Exemplu

```
objectList.add(new Object());
String s = stringList.get(0); // operatie ilegala!
```

=> Operaţia **nu va fi permisă de către compilator**!

Observaţie importantă

- Integer extends Object
- Integer extends Number

Dar:

- **Stack<Integer> extends Stack<Object>**
- **Stack<Integer> extends Stack<Number>**

Dacă *SubType* este un subtip

- (clasă descendentă sau
- subinterfaţă)

al lui *SuperType:*

Atenţie!

O structură generică:

GenericStructure <SubType>

NU este un subtip al lui:

GenericStructure < SuperType >!

Tip – Subtip in genericitate - concluzie

- 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”

Wildcards

- Utilizate atunci când dorim să intrebuinţăm o structură generică drept *parametru* într-o funcţie şi nu dorim să limităm tipul de date din colecţia respectivă.
- Exemplu fără wildcard:

```
void printCollection(Collection<Object> c) {
    for (Object e : c) System.out.println(e);
}
```

- ne restricţionează să folosim la apelul funcţiei doar o colecţie cu elemente de tip *Object* (care ***nu*** *poate fi convertită la o colecţie de un alt tip*)!

Wildcards

- Această restricţie este eliminată de folosirea **wildcard**-urilor:
- Exemplu:

```
void printCollection(Collection<?> c) {
    for (Object e : c) System.out.println(e);
}
```

- Limitare: **nu putem adăuga elemente arbitrare** într-o colecţie cu wildcard-uri:

```
Collection<?> c = new ArrayList<String>();
                        // Operatie permisa
c.add(new Object());
                        // Eroare la compilare
```

- De ce?

Wildcards - observatii

- Nu putem adăuga într-o colecţie generică decât elemente **de un anumit tip**, iar wildcard-ul **nu indică un tip anume!**
- Putem adăuga elemente de tip *String?*

```
List<?> myList = new ArrayList<String>();
myList.add("Some String");
    // Eroare compilare!!
```

- Singurul element care poate fi adăugat este *null*, întrucât acesta este membru al oricărui tip referinţă!

- Rezolvare:

```
List<?> someList = new ArrayList<String>();
((ArrayList<String>)someList).add("Some String");
```

- Operaţiile de tip *getter* sunt posibile
  - rezultatul acestora poate fi mereu interpretat drept *Object*:

```
Object item = someList.get(0);
```

Exemple

- Corect? Dacă da, ce afişează?

```
List<?> someList = new ArrayList<String>();
((ArrayList<Integer>)someList).add(1);
System.out.print(someList);
```

- Corect? Dacă da, ce afişează?

```
List<?> someList = new ArrayList<String>();
((ArrayList<String>)someList).add("Some String");
((ArrayList<Integer>)someList).add(1);
Object item = someList.get(0);
System.out.println(item);
item = someList.get(1);
System.out.println(item);
```

Bounded Wildcards

- un *wildcard* poate fi înlocuit cu orice tip => poate deveni un inconvenient!

Mecanismul bazat pe **Bounded Wildcards:**

- permite introducerea unor restricţii asupra tipurilor ce pot inlocui un wildcard
- le obligă să se afle într-o relaţie ierarhică (de moştenire) faţă de un tip fix specificat.

*Clasă(sau interfaţă) <? **extends** Bază>*

- impune ca tipul elementelor clasei (interfeţei) să fie de tip **Bază** sau un **subtip** al acesteia!

*Clasă(sau interfaţă) <? **super** Bază>*

- impune ca tipul elementelor listei să fie de tip **Bază** sau o **superclasă** a acesteia

Wildcards - exemple

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

BOUNDED

- Delimitate superior

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

- Delimitate inferior

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

Exemplu

```
class User{
    protected String name = "User";
    public String getName() {
        return name;
    }
}
class Admin extends User {
    public Admin () {
        name = “Admin ";
    }
}
class Student extends User {
    public Student () {
        name = " Student ";
    }
}
```

Exemplu

```
class MyApplication {
    // bounded wildcards
    public static void listUser(List<? extends User> userList) {
        for(User item : userList)
            System.out.println(item.getName());
    }

    public static void main(String[] args) {
        List<User> pList = new ArrayList<User>();

        pList.add(new Admin());
        pList.add(new Student());
        pList.add(new User());

        MyApplication.listUser(pList);
        // Se va afisa: “Admin", “Student", "User"
    }
}
```

Exemple

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

-prima este clasa, daca exista o clasa

```
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!
	}
}
```

Metode *generice*

= Metode care introduc un parametru de tip pentru a facilita prelucrarea unor structuri generice (date ca parametru), altul decât cei ai clasei din care fac parte.

Exemplu:

```
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});
```

Metode generice - exemple

- Exemple de implementare ale unei metode ce copiază elementele unui vector intrinsec într-o colecţie:
- Metodă corectă

```
static <T> void correctCopy(T[] a, Collection<T> c){
    for (T o : a) c.add(o); // Operatia va fi permisa
}
```

- Metodă incorectă

```
static void incorrectCopy(Object[] a, Collection<?> c){
    for (Object o : a) c.add(o);
/* Operatie incorecta, eroare la compilare: adăugarea
   elementelor într-o colecţie generică cu tip
   specificat*/
}
```

Metode generice

- putem folosi *wildcards* sau *bounded wildcards*.

- Exemple corecte:
- metodă ce copiaza elementele dintr-o listă în altă listă

```
public static <T> void copy (List<T> dest,
                             List<? extends T> src) { …}
```

- metodă de adăugare a unor elemente într-o colecţie, cu restricţionarea tipului generic

```
public boolean addAll (int index, Collection<? extends
    E> c)
```

Metode generice si parametri de tip Bounded - exemplu

```
public static <T> int countGreaterThan(T[] anArray, T elem) {
    int count = 0;
    for (T e : anArray)
        if (e > elem) // compiler error
            ++count;
    return count;
}
```

- Operatorul > se aplica doar pentru tipurile primitive (short, int, double, long, float, byte, char), nu pentru a compara obiecte.
- Pentru a rezolva problema se foloseste un parametru de tip marginit de interfata Comparable<T>:

```
public interface Comparable<T> {
    public int compareTo(T o); }
public static <T extends Comparable<T>> int
    countGreaterThan(T[] anArray, T elem) {
    int count = 0;
    for (T e : anArray)
        if (e.compareTo(elem) > 0) ++count;
    return count;
}
```

Stergerea tipurilor - Type Erasure

- Genericitatea a fost introdusa in Java pentru a realiza un control al tipurilor la compilare si pentru a furniza suportul pentru programarea generica.
- Pentru a implementa genericitatea compilatorul Java aplica stergerea tipului astfel:
- Inlocuieste toti parametrii de tip din tipurile generice cu limitele lor sau cu Object daca parametrii de tip sunt nemarginiti
- Bytecod-ul generat astfel, contine doar clase, interfete si metode obisnuite
- Insereaza conversii de tip daca sunt necesare pentru a asigura type safety.
- Genereaza bridge methods pentru a asigura polimorfismul in tipurile generice extinse.
- Type erasure asigura faptul ca nicio noua clasa nu va fi creata pentru tipurile parametrizate; ca urmare, genericitatea nu genereaza runtime overhead.

Type Erasure – exemplul 1

```
public class Node<T> {
    private T data;
    private Node<T> next;
    public Node(T data, Node<T> next){
        this.data = data;
        this.next = next;
    }
    public T getData() { return data; }
        // ...
}
```

- Rezultatul dupa type erasure – inlocuire cu Object:

```
public class Node {
    private Object data;
    private Node next;
    public Node(Object data, Node next) {
        this.data = data; this.next = next; }
    public Object getData() { return data; }
    // ...
}
```

Type Erasure – exemplul 2

```
public class Node<T extends Comparable<T>> {
    private T data;
    private Node<T> next;
    public Node(T data, Node<T> next) {
        this.data = data; this.next = next; }
    public T getData() { return data; }
    // ...
}
```

- Compilatorul inlocuieste parametrul de tip T marginit cu prima clasa care-l margineste - Comparable:

```
public class Node {
    private Comparable data;
    private Node next;
    public Node(Comparable data, Node next) {
        this.data = data; this.next = next; }
    public Comparable getData() { return data; }
    // ...
}
```

Restrictii in genericitate

NU SE POT:

- Instantia tipuri generice folosind tipuri primitive
- Crea instante pentru un parametru de tip
- Declara campuri statice al caror tip este un parametru de tip
- Utiliza *cast* sau *instanceof* cu un tip parametrizat
- Crea *arrays* de un tip parametrizat
- Crea, prinde, arunca obiecte de tip parametrizat
- Supraincarca o metoda cu parametri de tip care au acelasi raw type

Instantiere tipuri generice folosind tipuri primitive - NU

```
class Pair<K, V> {
    private K key;
    private V value;
    public Pair(K key, V value) {
        this.key = key;
        this.value = value; }
    // ...
}
```

- Pair<**int, char**> p = new Pair<>(8, 'a');
  // compile-time error
- Pair<**Integer, Character**> p = new Pair<>(8, 'a');
  Se face autoboxing 8 la Integer.valueOf(8) si 'a' la Character('a'):
  Pair<Integer, Character> p = new Pair<>(Integer.valueOf(8), new Character('a'));

Creare instante pentru un parametru de tip - NU

- compile-time error:

```
public static <E> void append(List<E> list) {
    E elem = new E(); // compile-time error
    list.add(elem);
}
```

- Se poate crea un obiect de tip generic prin reflexie:

```
public static <E> void append(List<E> list,
    Class<E> cls) throws Exception {
     E elem = cls.newInstance(); // OK
     list.add(elem);
}
```

- Apel metoda:

```
List<String> ls = new ArrayList<>();
append(ls, String.class);
```

Declarare campuri statice ce au ca tip un parametru de tip - NU

```
public class MobileDevice<T> {
    private static T os;
    // ...
}
```

- Daca ar fi permis, urmatorul cod ar fi confuz:

```
MobileDevice<Smartphone> phone = new
    MobileDevice<>();
MobileDevice<Pager> pager = new
    MobileDevice<>();
MobileDevice<TabletPC> pc = new
    MobileDevice<>();
```

- Deoarece campul static *os* este comun atat pentru phone, pager, cat si pc, care ar fi de fapt tipul lui *os*? Nu poate fi Smartphone, Pager, si TabletPC in acelasi timp.

Utilizare cast sau instanceof cu un tip parametrizat - NU

- Datorita stergerii tuturor parametrilor de tip din codul generic, nu se poate verifica ce tip parametrizat este folosit la executie pentru un tip generic:

```
public static <E> void test (List<E> list) {
    if (list instanceof ArrayList<Integer>)
      { // compile-time error // ... }
}
```

- Multimea tipurilor parametrizate pe care le poate primi metoda *test*:

  S = { ArrayList<Integer>, ArrayList<String> LinkedList<Character>, ... }

- La executie nu se poate face diferenta intre un ArrayList<Integer> si un ArrayList<String>. Ce putem face este sa folosim un wildcard pentru a verifica ca *list* este un ArrayList:

```
public static void test (List<?> list) {
      if (list instanceof ArrayList<?>) {
            // OK; instanceof requires a reifiable type
      }
}
```

Utilizare cast tip parametrizat - NU

- In mod normal, nu se poate converti un tip parametrizat decat daca el este parametrizat printr-un wildcard.

- Exemplu:

  List<Integer> li = new ArrayList<>();

  List<Number> ln = (List<Number>) li; // compile-time error

- Totusi, in unele cazuri compilatorul stie ca un parametru de tip este intotdeauna valid si permite conversia.

- Exemplu:

  List<String> l1 = ...;

  ArrayList<String> l2 = (ArrayList<String>)l1; // OK

Creare *vectori intrinseci* de un tip parametrizat - NU

```
List<Integer>[] arrayOfLists = new List<Integer>[2];
// compile-time error
```

- Inserare tipuri diferite intr-un vector intrinsec:

```
Object[] strings = new String[2];
strings[0] = "hi"; // OK
strings[1] = 100; //  ArrayStoreException
```

- Daca am incerca acelasi lucru cu o lista generica:

```
Object[] stringLists = new List<String>[];
    // compiler error, dar sa presupunem ca ar fi corect
stringLists[0] = new ArrayList<String>(); // OK
stringLists[1] = new ArrayList<Integer>();
/* Ar trebui sa apara ArrayStoreException, dar nu
    poate fi detectata!! */
```

Creare, prindere, aruncare obiecte de tip parametrizat - NU

- O clasa generica nu poate extinde direct sau indirect clasa Throwable:

```
// Mostenire indirecta Throwable
class MathException<T> extends Exception { /* ... */ }
        // compile-time error
// Mostenire directa Throwable
class QueueFullException<T> extends Throwable { /* ...*/ }
        // compile-time error
```

- O metoda nu poate prinde o instanta a unui parametru de tip:

```
public static <T extends Exception, J> void execute(List<J>
    jobs){
    try {
        for (J job : jobs) // ...
     } catch (T e) { // compile-time error // ... }
}
```

- Totusi, se poate folosi un parametru de tip in clauza throws:

```
class Parser<T extends Exception> {
    public void parse(File file) throws T { // OK // ... }
}
```

Supraincarcare metoda cu parametri de tip care au acelasi raw type - NU

- O clasa nu poate avea doua metode supraincarcate care vor avea aceeasi signatura – antet – dupa stergerea tipurilor (type erasure)

```
public class Example {
    public void print(Set<String> strSet) { }
    public void print(Set<Integer> intSet) { }
}
```

- compile-time error.

Aplicatii

Intrebări și exerciții

- *The Java Tutorials*, Generics

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