# Generic types, Collections in Java

## Objectives:

- understanding the generic types mechanisms; generic classes; generic interfaces; generic methods;
- using the main collection types;

## Generic types

The mechanism of generic types allows data types to be parameters when defining classes, methods and interfaces.

- Avantages:
  - code robustness (type checks on compilation)
  - cast conversions not required
  1. more general algorithms

**Syntax:**

**Generic class**

```
class name<T1, T2, ..., Tn> { /* ... */ }
```

**Generic interface**

```
interface name<T1, T2, ..., Tn> { /* ... */ }
```

**Generic method**

```
[specificator_acces] [static] <K, V> tip_returnat name(generic_type <K, V> p1, generic_type<K, V> p2) { /*...*/}
```

Generic types naming conventions:

- E - Element (used extensively by Java Collections Framework)
- K - Key
- N - Number
- T - Type
- V - Value

**Generic types – restriction (specialization) of parameters**

**Simple restriction**

```
<T extends B1>
```

**Restriction through classes and interfaces**

```
<T extends MyClass & MyInterface>
```

**Generic types - wildcard parameters ?**

În Java aceste metode generice utilizează ca parametru “?”, wildcard, in locul tipului generic T.

De asemenea se poate folosi cu “?” si un mecanism de limitare – specializare:

`<? extends Superclass>` (Superclass -> upperbound limit)
`<? super Subclass>` (Subclass -> lowerbound limit)

### Java Collections Framework (JCF)

Java Collections Framework serves for handling uni-dimensional arrays of class instances. The number of elements is not known from the beginning and is permanently updated.

Definition: *a collection is a group of objects, ordered or not, that may or may not contain repeated* values *or not*. There are collections that can contain only a certain type of elements (for example instances of a single class) and there are collections that can store objects of various types (using the generics mechanism). Some collections allow introducing *null* elements, others do not.

Java collections provide mechanisms for inserting elements into collections, searching for certain components, ordering, etc.

### Interface `Collection`

This interface is at the base of collection interfaces and classes hierarchy.

**Interfaces derived from `Collection`:** `BeanContext, BeanContextServices, BlockingDeque<E>, BlockingQueue<E>, Deque<E>, List<E>, NavigableSet<E>, Queue<E>, Set<E>, SortedSet<E>, TransferQueue<E>`

**Classes that implement `Collection`:** `AbstractCollection, AbstractList, AbstractQueue, AbstractSequentialList, AbstractSet, ArrayBlockingQueue, ArrayDeque, ArrayList, AttributeList, BeanContextServicesSupport, BeanContextSupport, ConcurrentHashMap.KeySetView, ConcurrentLinkedDeque, ConcurrentLinkedQueue, ConcurrentSkipListSet, CopyOnWriteArrayList, CopyOnWriteArraySet, DelayQueue, EnumSet, HashSet, JobStateReasons, LinkedBlockingDeque, LinkedBlockingQueue, LinkedHashSet, LinkedList, LinkedTransferQueue, PriorityBlockingQueue, PriorityQueue, RoleList, RoleUnresolvedList, Stack, SynchronousQueue, TreeSet, Vector`

The main methods from the `Collection` interface.

- *boolean add(Object o)*
  => adds an item to the collection
- *boolean addAll(Collection c)*
  => adds all items in the collection received as a parameter
- *void clear()*
  => deletes all items in the collection
- *boolean contains(Object o)*
  => returns a boolean value based on the existence of the searched object
- *containsAll(Collection c)*
  => returns a Boolean value based on the existence of an entire collection
- *boolean equals(Object o)*
  => compares 2 collection items
- *int hashCode()*
  => returns the *hash* value of the collection
- *boolean isEmpty()*
  => tests whether or not the collection has elements
- *Iterator iterator()*

=> returns a iterator for collection items

- *boolean remove(Object o)*
  => removes (if any) from the collection the item received as a parameter
- *boolean removeAll(Collection c)*
  => removes from the collection all items from the collection received as a parameter
- *boolean retainAll(Collection c)*
  => preserves only the items that are contained in the collection received as a parameter
- *int size()*
  => returns the number of items in the collection
- *Object[] toArray()*
  => returns an array of objects populated with items in the collection

- Collections can be traversed with: *forEach* method or with *Iterator* instances

**Iterator** = object used to browse a container

```
public interface Iterator<E> {
boolean hasNext( );
E next( );
void remove( ); //optional
}
```

- Ordering the collections can be done with the interface *Comparable*:

```
interface Comparable<T> {
        public int compareTo(T o);
}
```

or with interface *Comparator* that has the methods:

*naturalOrder(), reverseOrder(), comparing(), compare(), …*

*Comparable* interface is a good choice when used to define the default ordering or if it represents the main method for comparing object.
Using *Comparator* several different comparison strategies can be defined.

### Interface `List`

Provides a mechanism for controlling an ordered collection of objects. The programmer can control the insertion position of an element and can access any stored object through the index.

Lists support object duplicates.

### Classes that implement interface `List`:

`AbstractList, AbstractSequentialList, ArrayList, AttributeList, CopyOnWriteArrayList, LinkedList, RoleList, RoleUnresolvedList, Stack, Vector`

### Interface `Set/SortedSet`

A set is a collection without duplicate items.

`SortedSet` implementations need to use `Comparable` interface.

**Classes that impment interface `Set/SortedSet`**

`AbstractSet, ConcurrentHashMap.KeySetView, ConcurrentSkipListSet, CopyOnWriteArraySet, EnumSet, HashSet, JobStateReasons, LinkedHashSet, TreeSet`

**Interface `Map/SortedMap`**

An object of this type makes associations between keys and values/objects. The keys are unique. A key can be associated with a single value.

There are 3 ways of traversing the stored items:

- by key
- by value
- by key-value associations

The *SortedMap* interface extends the *Map* interface and allows elements comparing.

**Classes that implement interface `Map/SortedMap`**

`AbstractMap, Attributes, AuthProvider, ConcurrentHashMap, ConcurrentSkipListMap, EnumMap, HashMap, Hashtable, IdentityHashMap, LinkedHashMap, PrinterStateReasons, Properties, Provider, RenderingHints, SimpleBindings, TabularDataSupport, TreeMap, UIDefaults, WeakHashMap`

**Examples**

**Ex. 1 - generic classes and interfaces**

```
interface Int1<T>{
	public void printType(T var);
}

class C1<T> implements Int1<T>{
	public void printType(T var){
		System.out.println("Tipul variabilei curente: "+var.getClass());
	}
}

public class Generics_01{
	public static void main(String[] args) {
		//C1 ob1 = new C1(); //compiler warnings
		C1<String> ob1 = new C1<>();
		String x = "xxx";
		ob1.printType(x);

		C1<Integer> ob2 = new C1<>();
		Integer y = 7;
		ob2.printType(y);
	}
}
```

**Ex. 2 - generic methods**

```
import java.util.*;
```

```
class Generics_02{
	public <T> void fromArrayToCollection(T[] a, Collection<T> c) {
		for (T o : a) {
			c.add(o);
		}
	}
}

class Test{
	public static void main(String[] args) {
		Generics_02 ob = new Generics_02();
		String strings[] = {"aaa", "bbb", "ccc"};
		Collection <String> collection = new ArrayList<>();
		ob.fromArrayToCollection(strings, collection);
		System.out.println(collection);
	}
}
```

**Ex. 3 - wildcards**

```
class TwoD{
	int x, y;
	TwoD(int x, int y){
		this.x = x;
		this.y = y;
	}
}

class ThreeD extends TwoD{
	int z;
	ThreeD(int x, int y, int z){
		super(x, y);
		this.z = z;
	}
}

class FourD extends ThreeD{
	int t;
	FourD(int x, int y, int z, int t){
		super(x, y, z);
		this.t = t;
	}
}

class Coords<T extends TwoD>{
	T [ ] coords;
	Coords(T [] o){
		coords=o;
	}
}

class BoundedWildcard{
	void showXY(Coords<?> c){
		System.out.println("x y coordinates: ");

		for(int i=0; i<c.coords.length; i++)
```

```
            System.out.println(c.coords[i].x + " " +c.coords[i].y);

        System.out.println();
    }

    void showXYZ(Coords<? extends ThreeD> c){
        System.out.println("x y z coordinates: ");

        for(int i=0; i<c.coords.length; i++)
            System.out.println(c.coords[i].x + " " + c.coords[i].y+" "
            +c.coords[i].z);
        System.out.println();
    }

    void showAll(Coords<? extends FourD> c){
        System.out.println("x y z t coordinates: ");

        for(int i=0; i<c.coords.length; i++)
            System.out.println(c.coords[i].x + " " +c.coords[i].y+ " "
            +c.coords[i].z+ " " +c.coords[i].t);
        System.out.println();
    }
}

class Test{
    public static void main(String[] args) {

        BoundedWildcard bw = new BoundedWildcard();

        TwoD[ ] td ={
            new TwoD(0,0),
            new TwoD(7,9),
            new TwoD(18,4),
            new TwoD(-1,-2)
        };
        Coords<TwoD> td1ocs = new Coords<TwoD> (td);

        System.out.println("Contents of td1ocs");

        bw.showXY(td1ocs);
        //bw.showXYZ(td1ocs); //tip nepotrivit
        //bw.showAll(td1ocs); //tip nepotrivit

        FourD[ ] fd= {
            new FourD(1,2,3,4),
            new FourD(6,8,14,8),
            new FourD(22, 9, 7, 5),
            new FourD(-3, 5, 7, -17)
        };

        Coords<FourD> fd1ocs = new Coords<FourD> (fd);

        System.out.println("Contents of fd1ocs");

        bw.showXY(fd1ocs);
        bw.showXYZ(fd1ocs);
        bw.showAll(fd1ocs);
```

```
	}
}
```

**Ex. 4 - Vector based collection**

```
import java.util.Vector;

public class Vector_01 {

	public static void main(String[] args) {

		Vector <Integer> v = new Vector<>( );
		v.add(7);
		v.add(8);
		v.add(9);

		for(Integer crt_int : v){
			System.out.println(crt_int.intValue());
		}

		v.removeAllElements();
		//v.setSize(3);

		v.add(0, 7);
		v.add(0, 8);
		v.add(0, 9);


		for(int i=0; i<v.size(); i++){
			System.out.println(i+": "+v.elementAt(i).intValue());
		}

		v.removeElementAt(2);

		for(Integer crt_int : v){
			System.out.println(crt_int.intValue());
		}

		v.removeAllElements();

	}
}
```

**Ex.5 - SortedSet based collection**

```
// crearea unei liste de tip SortedSet, in care elementele de tip Persoana sunt
ordonate crescator dupa nume si descrescator dupa varsta.

import java.util.Comparator;
import java.util.Iterator;
import java.util.Scanner;
import java.util.SortedSet;
import java.util.TreeSet;

public class Persoana {
```

```
       private String nume;
       private int varsta;

       public String getNume() {
            return nume;
       }
       public void setNume(String nume) {
            this.nume = nume;
       }

       public int getVarsta() {
            return varsta;
       }
       public void setVarsta(int varsta) {
            this.varsta = varsta;
       }

       public String toString()
       {
            return new String("Nume: "+nume+" varsta: "+varsta);
       }
} // class Persoana


class PComparator implements Comparator<Persoana> {
    public int compare(Persoana p1, Persoana p2)
    {
      if(p1.getNume().compareTo(p2.getNume())<0)
           return -1;
        if(p1.getNume().compareTo(p2.getNume())>0) return 1;
        return p2.getVarsta()-p1.getVarsta();
    }
} // class PComparator


public class TestSetPersoana {
       public static void main(String[] args) {
       // crearea unui set sortat TreeSet
            Scanner sc=new Scanner(System.in);
            Persoana tab[]=new Persoana[4];
            String nume;
            int v;

            SortedSet <Persoana>set = new TreeSet<Persoana>(new PComparator());

            for(int i=0; i<tab.length;i++)
            {
                  System.out.println("Nume: ");
                  nume=sc.next();
                  System.out.println("Varsta: ");
                  v=sc.nextInt();

                  tab[i]=new Persoana();
                  tab[i].setNume(nume);
                  tab[i].setVarsta(v);
                  set.add(tab[i]);
            }
```

```
        // parcurgerea elementelor din set
        Iterator <Persoana>it = set.iterator();
        while (it.hasNext()) {
                  // preluarea elementului curent
             Persoana element = it.next();
             System.out.println(element);

         }
    }
}
```

**Ex. 6 - sorting collections**

```
import java.util.*;
import java.lang.Comparable;

class Name implements Comparable<Name> {
       private final String firstName, lastName;

       public Name(String firstName, String lastName) {
            this.firstName = firstName;
            this.lastName = lastName;
       }

       public String getFirstName() { return firstName; }
       public String getLastName()  { return lastName;  }

       @Override
       public boolean equals(Object o) { //override din clasa Object
            if (!(o instanceof Name))
                  return false;
            Name temp = (Name) o;
            return temp.firstName.equals(firstName) &&
                  temp.lastName.equals(lastName);
       }
       @Override
       public int hashCode() {//override din clasa Object
            return firstName.hashCode() + lastName.hashCode();
       }
       @Override
       public String toString() {//override din clasa Object
            return firstName + " " + lastName;
       }

       //metoda preluata din interfata Comparable
       @Override
       public int compareTo(Name n) {
            int lastCmp = lastName.compareTo(n.getLastName());
            return (lastCmp != 0 ? lastCmp : firstName.compareTo(n.getFirstName()));
       }
}

class Test {
       public static void main(String... args) {
```

```
            Name bandsArray[] = {
                  new Name("Foo", "Fighters"),
                  new Name("Nir", "Vana"),
                  new Name("Met", "Allica"),
                  new Name("AC", "DC")
            };
            List<Name> names = Arrays.asList(bandsArray);
            Arrays.sort(bandsArray);//ordonare compareTo()
            //Collections.sort(names);//ordonare compareTo()
            System.out.println(names);
      Name ob= new Name("Nir", "Vana");
System.out.println("names contains Nirvana = "+names.contains(ob));//needs equals()
& hashCode()
      }
}
```

**Individual work**

**Complexity ***

1. Create an interface called *Generator<T>* with a single method *next(T var)*. Implement the interface so that you can generate the following values when applying it to certain data types (Integer, Character, etc.). The class will be instantiated in the *main( )* method, located in a separate class.

2. Write a class called *Calculator* which has the methods to do addition, subtraction, multiplication and division. The methods will take generic input variables and will return the corresponding type. For example, the sum of two integers should return an integer, and for floats it should return a float. Same for division. Adding and subtracting is allowed for *String* variables as well, but the multiplication and division will print an error message.

Write the same class, but use method overloading.

3. Build an application which contains a generic class *SetterGetter* which allows the user to *set()* and *get()* the attribute values for different types of objects. For example, given the classes *Kid, Adult* and *Retired*, enable the class to set and get the names and ages of the associated objects. Create collections with unique entries of type *Kid, Adult* and *Retired*, and which are populated with data read from the console. Print the read data using different methods.

**Complexity ****

4. Implement a class called *UserFile* (name, extension, type, size in bytes, constructors, mutators, accesors). The types of files are predefined and stored in a *Hashtable* object (for example "image"->0, "text"->1, "application"->2, etc.) Create a list of objects from this class and read from the keyboard the associated info. Print all the specific extensions of the predefined file types. Order the file list based on size and print the result.

5. Write a class named *Student* that has the private fields *name, group, average_mark*, and getter / setter methods for all the attributes. In the *main()* method, placed in another class, create a sortedSet collection, with *Student* type objects, initialized with values read from the keyboard, that will keep the elements in descending order by *average_mark* and in ascending order by name (the students that have the same average will appear in alphabetic order). Browse the collection using *for-loop* and display all the items. Then browse the collection with an *iterator* and display all students with *average_mark*> = 8. Browse the collection with *forEach ()* and display all student data in a particular group.

**Complexity ***

6. Write an *Employee* class with the private fields name, age, salary and getter / setter methods for the fields. In the *main ()* method, placed in another class, create a list of *Employee* type objects, initialized with values read from the keyboard. Sort the the list in:
- ascending order by name, using the *Comparable* interface
- descending order by age, using the *Comparable* interface and a lambda expression
- ascending order by name and descending order by salary, using the *Comparator* interface

7. Create a generic class called *VirtualLibrary* with a single attribute, *totalNumberOfEntries*, and with methods which enables the user to set and return an entry. The types of entries are *Book, Article, MediaResource, Magazine and Manual*. Implement the specific classes for each type of entry.

In the *main( )* method create a *SortedSet<VirtualLibrary>* variable which maintains all the library entries. Use the methods to add, add multiple, return a specific entry and check if an entry exists in the library.