softserve experience matters

Java Core

# Generic Collections

IT Academy
05/2016

# Agenda

- Wrapper Pattern
- Generic in Java
- Arrays in Java
- Collections in Java
  - List, Set, Map
  - Iterators
  - Methods, sorting
- Practical tasks

# Wrapper Pattern

Non-generic Box class

```
public class Box {
  private Object obj;
  public void set(Object obj) { this.obj = obj; }
  public Object get( ) { return obj; }
}
```

Since its methods accept or return an Object, you are free to pass in whatever you want, provided that it is not one of the primitive types.

There is no way to verify, at compile time, how the class is used.

# Wrapper Pattern

```
public class Appl {
  public static void main(String[ ] args) {
      String text = "Hello World";
      Box box = new Box();
      box.set(text);
      Integer i = (Integer) box.get();
    }
}
```

Runtime Error

One part of the code may place an Integer in the box and expect to get Integers out of it, while another part of the code may mistakenly pass in a String, resulting in a runtime error.

# Wrapper Pattern

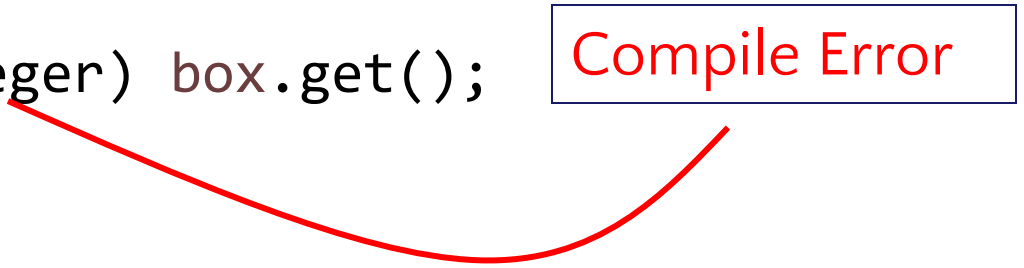
Wrapper (or Decorator) is one of the most important design patterns.

One class takes in another class, both of which extend the same abstract class, and adds functionality.

```
public class WrapperBox {
  private Box box = new Box();
  public void set(String text) {
   this.box.set(text);
  }
  public String get( ) { return box.get(); }
}
```

# Wrapper Pattern

```
public class Appl {
  public static void main(String[ ] args) {
      String text = "Hello World";
      WrapperBox box = new WrapperBox();
      box.set(text);
      Integer i = (Integer) box.get();
    }
}
```




The basic idea of a wrapper is to call-forward to an underlying object, while simultaneously allowing for new code to be executed just before and/or just after the call.

# Generic in Java

Generics, introduced in Java SE 5.0

- A generic type is a generic class or interface that is parameterized over types.
- Generics add a way to specify concrete types to general purpose classes and methods that operated on Object before.
- With Java's Generics features you can set the type for classes.

Generic class is defined with the following format:

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

The type parameter section, delimited by angle brackets (<>), follows the class name.

# Generic in Java

To update the `Box` class to use generics, you create a generic type declaration by changing the code

```
public class Box
```

to

```
public class Box<T>
```

This introduces the type variable, `T`, that can be used anywhere inside the class.

To instantiate this class, use the new keyword, as usual, but place <Integer> between the class name and the parenthesis:

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

# Generic in Java

```
public class Box<T> {
    // T stands for "Type".
    private T t;
    public void set(T t) { this.t = t; }
    public T get( ) { return t; }
}
```

All occurrences of Object are replaced by `T`.

A type variable can be any non-primitive type you specify: any class type, any interface type, any array type, or even another type variable.

The same technique can be applied to create generic interfaces.

# Generic in Java

```
public class Appl {
   public static void main(String[ ] args) {
      String text = "Hello World";
      Box<String> box = new Box<String>();
      box.set(text);
      Integer i = (Integer) box.get();
   }
}
```

Compile Error

Generics also provide compile-time type safety that allows programmers to catch invalid types at compile time.

# Generic in Java

Java method can be parametrized, too:

```
<T> getRandomElement(List<T> list) { … }
```

As with class definitions, you often want to restrict the type parameter in the method.

For example, a method which takes a list of Vehicles and returns the fastest vehicle in the list can have the following type.

```
<T extends Vehicle> T getFastest(List<T> list)
{…}
```

# Generic in Java

## Disadvantages

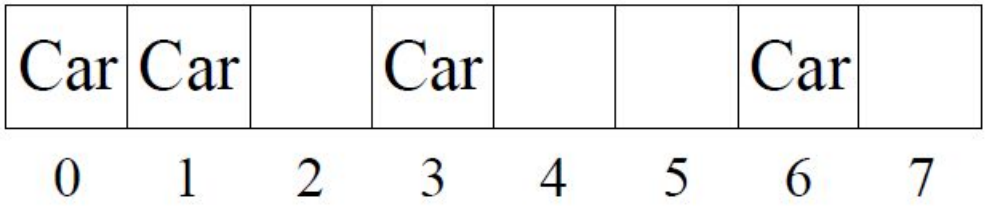
- Generic-fields can not be static.
- Static methods can not have generic parameters or use generic fields.
- Can not be made an explicit call to the constructor generic-type:

```
class Optional<T> {
      T value = new T();
}
```

The compiler does not know what constructor can be caused and the amount of memory to be allocated when an object.

# Arrays in Java

```
class Car{ };          // minimal dummy class
Car[ ] cars1;          // null reference
Car[ ] cars2 = new Car[10];   // null references
for (int i = 0; i < cars2.length; i++) {
    cars2[i] = new Car( );
}
```




```
// Aggregated initialization
Car[ ] cars3 = {new Car( ), new Car( ), new Car( )};
cars1 = {new Car( ), new Car( ), new Car( )};
```

# Arrays in Java

Most efficient way to hold references to objects.

## Advantages

- An array know the type it holds, i.e., compile-time type checking.
- An array knows its size, i.e., ask for the length.
- An array can hold primitive types directly.

## Disadvantages

- An array can only hold one type of objects (including primitives).
- Arrays are fixed size.
- How to add element inside?

# Collections in Java

Collection is a container of Objects, it groups many Objects into a single one.

Collections – dynamic arrays, linked lists, trees, sets, hash tables, stacks, queues.

All collections frameworks contain the following:

- interfaces
- implementations
- algorithms (there are the methods such as searching and sorting)

# Benefits of collections

- reduces programming effort
- increases program speed and quality
- allows interoperability among unrelated APIs
- reduce effort to design new APIs
- helps to reuse the code

# Wrappers

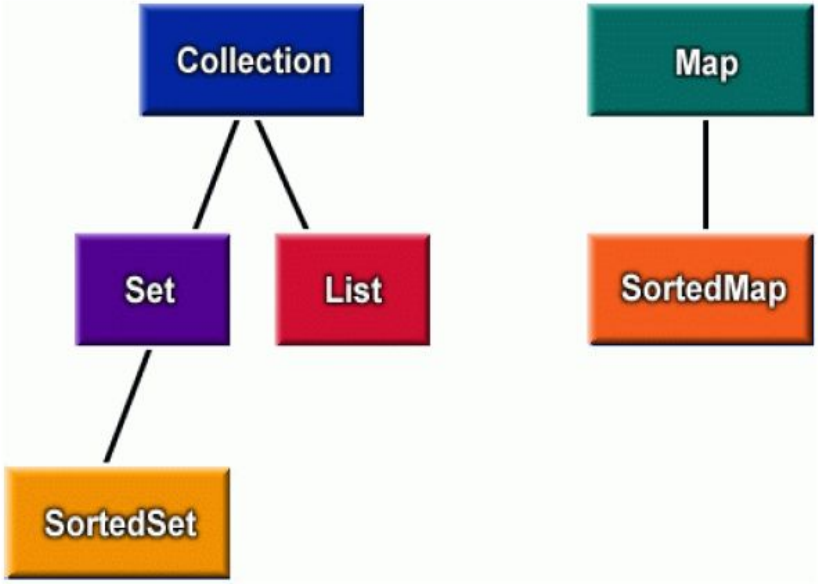
For all primitive types in Java are correspond type-wrappers (object types):

| Primitive type | Wrapper class | Constructor Arguments |
|---|---|---|
| `byte` | `Byte` | byte or String |
| `short` | `Short` | short or String |
| `int` | `Integer` | int or String |
| `long` | `Long` | long or String |
| `float` | `Float` | float, double or String |
| `double` | `Double` | double or String |
| `char` | `Character` | char |
| `boolean` | `Boolean` | boolean or String |

# Interfaces

There are data types that represent collections.

Classes that implement interfaces `List<E>` and `Set<E>`, implement the interface `Iterable<E>`.



# Collections in Java

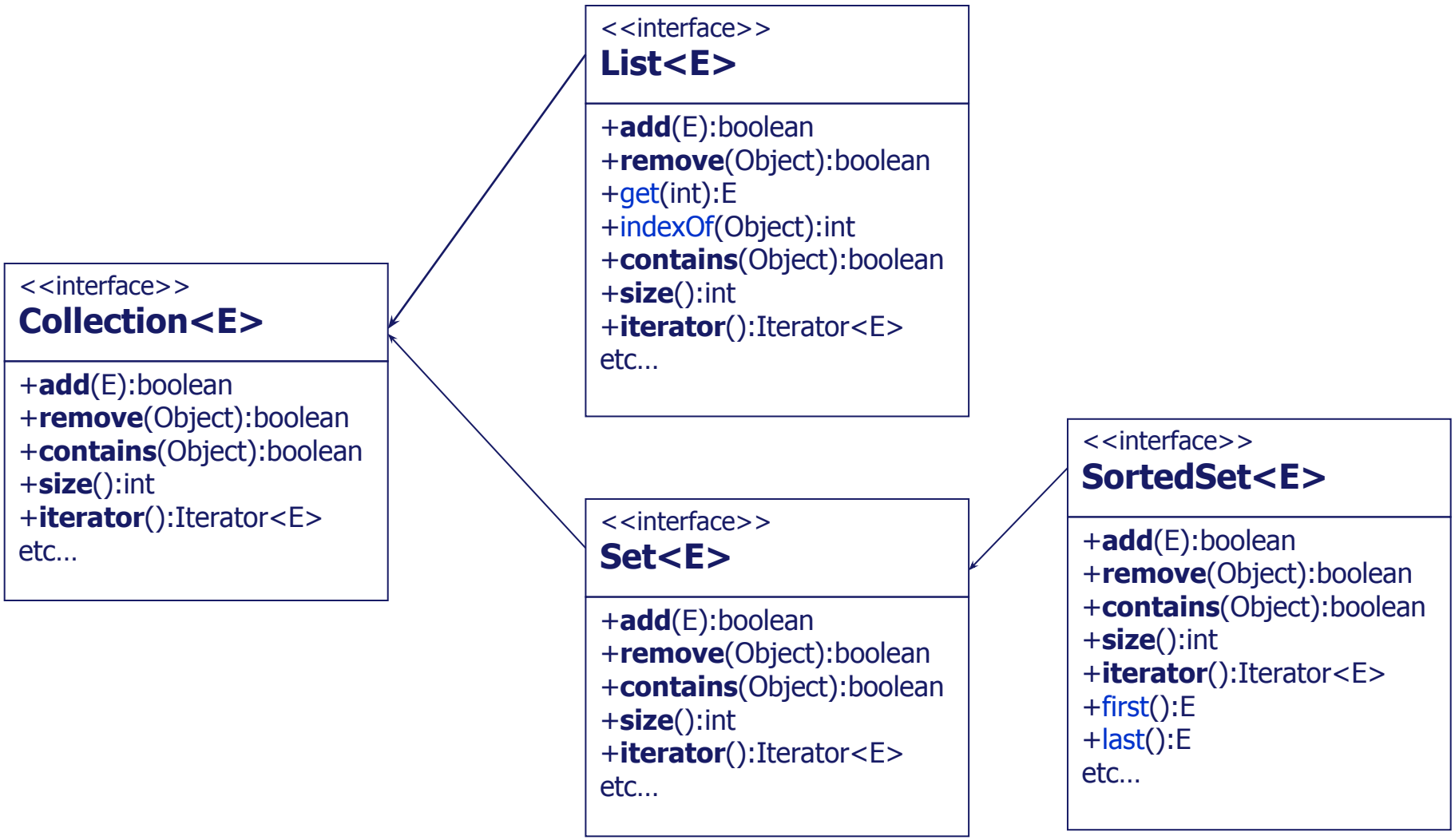
The `interface Collection<E>` defined methods:

- `boolean add(E obj)` – adds obj to the collection, it returns true, if the object is added;
- `boolean addAll(Collection<? extends E> c)` – adds all the elements;
- `void clear()` – removes all items from the collection;
- `boolean contains(Object obj)` – returns true, if the collection contains an element of obj;
- `boolean equals(Object obj)` – returns true, if the collections are equivalent;

# Collections in Java

- `boolean isEmpty()` – returns true, if the collection is empty;
- `Iterator<E> iterator()` – retrieves the iterator;
- `boolean remove(Object obj)` – removes the object from the collection;
- `int size()` – the number of items in the collection;
- `Object[] toArray()` – copies the collection to an array of objects;
- `<T>T[] toArray(T a[])` – copies the elements of the collection to an array of objects of a particular type.

# Interfaces

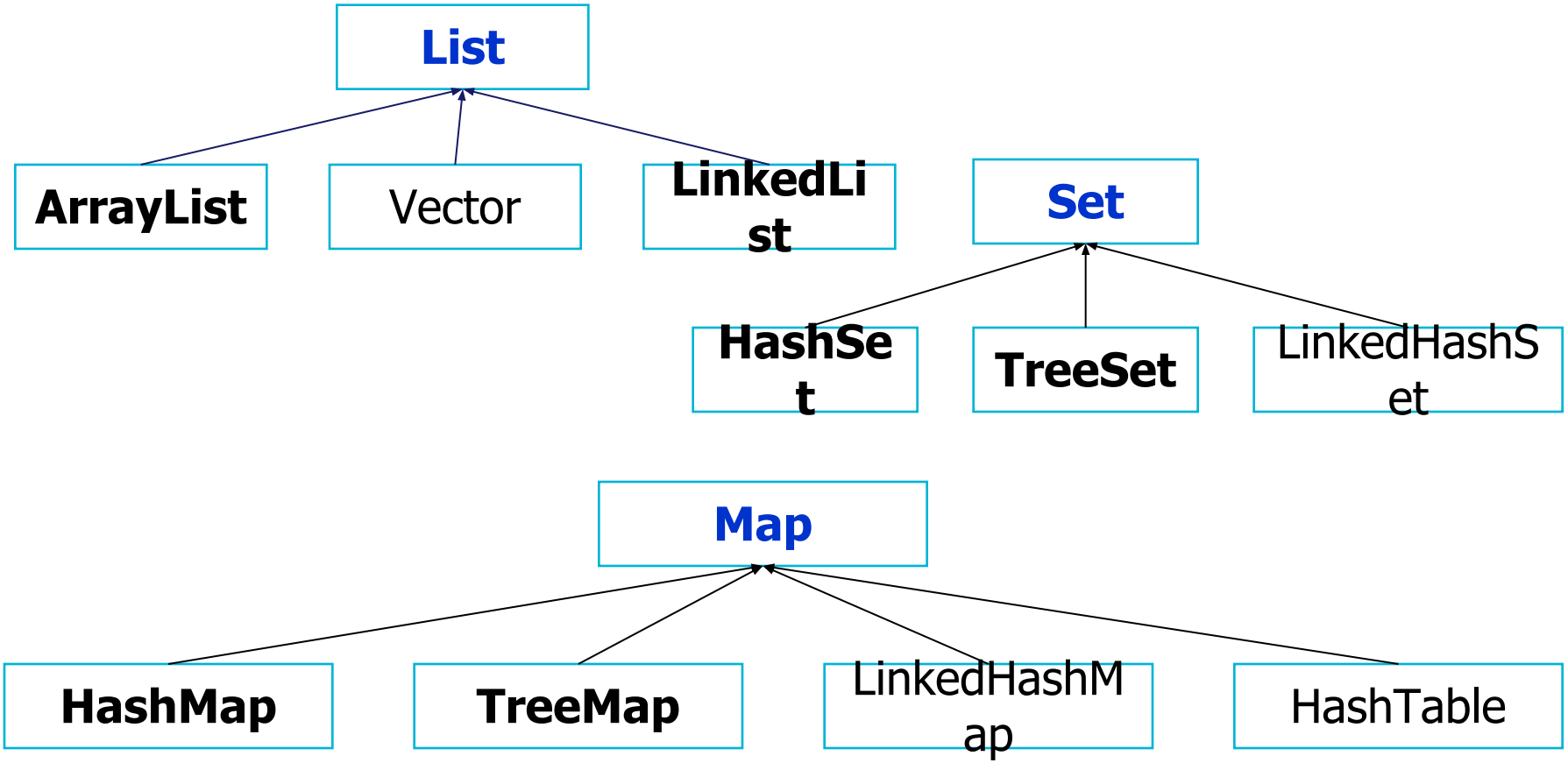

<<interface>>
Collection<E>
+add(E):boolean
+remove(Object):boolean
+contains(Object):boolean
+size():int
+iterator():Iterator<E>
etc...
<<interface>>
List<E>
+add(E):boolean
+remove(Object):boolean
+get(int):E
+indexOf(Object):int
+contains(Object):boolean
+size():int
+iterator():Iterator<E>
etc...
<<interface>>
Set<E>
+add(E):boolean
+remove(Object):boolean
+contains(Object):boolean
+size():int
+iterator():Iterator<E>
etc...
<<interface>>
SortedSet<E>
+add(E):boolean
+remove(Object):boolean
+contains(Object):boolean
+size():int
+iterator():Iterator<E>
+first():E
+last():E
etc...

# Collections in Java

- **List** – a list of objects. Objects can be added to the list (the method `add()`), replace the list (method `set()`), removed from the list (the method `remove()`), extract (method `get()`). There the ability to pass on the list of organizations with an *iterator*.
- **Set** – a set of objects. The same features as that of the List, but the object can be part of set only once. Double addition of one and the same object in the set is not change the set.
- **Map** – a map or associative array. In Map we add pair of objects (key, value). Accordingly, the search operation is a key value. Adding a couple with an existing key in the Map leads to the replacement, not to upload it. From Map can be obtain key and a list of values.

# Implementations

There are the implementations of the collection interfaces. In essence, they are reusable data structures



# List

Since List is an interface you need to instantiate a concrete implementation of the interface in order to use it.

There are two general-purpose List implementations — ArrayList and LinkedList

For example

- java.util.ArrayList
- java.util.LinkedList

`listArr` and `listLink` have objects of type Object

Here are a few examples of how to create a List instance:

```
List listArr = new ArrayList();
List listLink = new LinkedList();
```

# ArrayList

- Adding elements

```
List list = new ArrayList();
list.add("First element");
list.add("Second element");
list.add(0, "One more first element");
```

- Access through index

```
String element2 = list.get(1);
```

- Access through new for-loop

```
for(Object object : list) {
String element = (String) object;
}
```

# Iterator

Iterator – a helper object. Used to iterate over collection of objects.

Iterators are based on the interface

- `boolean hasNext()` – checks whether there are elements in the collection
- `Object next()` – shows the next item in the collection
- `void remove()` – removes the last selected item from the collection.

Collection interface has a method `Iterator iterator();`

Access through iterator

```
Iterator iterator = list.iterator();
while(iterator.hasNext(){
   String element = (String) iterator.next();
}
```

# ArrayList

- Removing Elements
  ```
  remove(Object element)
  remove(int index)
  ```
- Cleaning a list
  ```
  list.clear();
  ```
- List size
  ```
  int size = list.size();
  ```
- Generic List

```
List<MyType> myType = new ArrayList<MyType>( );
myType.add(new MyType( ));
MyType my = myType.get(0);
```

# LinkedList

**`LinkedList`** has the same functionality as the `ArrayList`.

Different way of implementation and efficiency of operations.

- Adding to the LinkedList is faster
- Pass through the list is almost as effective as the ArrayList,
- Arbitrary removal from the list is slower than ArrayList.

# Example

```
public static void main (String[ ] args) {
    ArrayList cars = new ArrayList( );
    for (int i = 0; i < 12; i++) {
        cars.add (new Car( ));
    }
    Iterator it = cars.iterator( );
    while (it.hasNext( )) {
        System.out.println ((Car)it.next( ));
    }
}
```

# Set

- A Set is a collection that does not contain any duplicate element.
- Element that are put in a set must override equals() method to establish uniqueness
- It is unsorted, unordered Set
- Can contain null

# Example

```
import java.util.*;
public class FindDups {
    public static void main(String args[ ]){
        Set s = new HashSet( );
        for (int i = 0; i < args.length; i++) {
            if (!s.add(args[i]))
            System.out.println("Duplicate detected: " +
           args[i]);
        }
        System.out.println(s.size( ) +
                    " distinct words detected: " + s);
    }
}
```

# Wrapper

```
public class MyList {
    private ArrayList v = new ArrayList( );
    public void add(MyType obj) {
        v.add(obj);
    }
    public MyType get(int index) {
        return (MyType)v.get(index);
    }
    public int size( ) {
        return v.size( );
    }
}
```

- Collections work with the Object class.
- We can add to the collection any objects of Java.
- When read from the collection, we also obtain Object.
- Add only objects of type MyType for MyList

# Sorting

```
public static void main(String[] args) {
      int[] x = new int[10];
      for (int i = 0; i < x.length; i++) {
         Random rand = new Random();
         x[i] = rand.nextInt(10);
      }
      Arrays.sort(x);
      for (int i = 0; i < x.length; i++) {
         System.out.println(x[i]);
      }
   }
```

What is wrong in the code

- Write a new code for type double, etc.
- Do I need to constantly create "bicycle" ?
- You may use an existing solution

# Class Arrays. Sorting

```
import java.util.Arrays;
public class Appl {
    public static void main(String[ ] args) {
        Student[ ] students = new Student[3];
        students[0] = new Student(52645, "Oksana");
        students[1] = new Student(98765, "Bogdan");
        students[2] = new Student(1354, "Orest");
        Arrays.sort(students);
        for (int i = 0; i < students.length; i++) {
            System.out.println(students);
     }
    }
}
```

What will happen?

# Compare elements

To specify the order of the following interfaces: `Comparable` and `Comparator`

```
public class MyType implements Comparable {
    String name;
    public int compareTo(Object obj) {
        return name.compareTo(((MyType)obj).name);
    }
}
```

`Comparable` to specify only one order.

Method **`compareTo`** can return

- 0, if objects are equal
- <0 (-1), if first object is less than second object
- >0 (1), if first object is great than second object

# Compare elements

`Comparator` interface has two methods

```
public int compare(Object o1, Object o2)
// and
public boolean equals(Object obj)
```

Methods **`compareTo`** and **`compare`** can throw an exception *ClassCastException*, if the object types are not compatible in the comparison.

# Example 1

```
public class Employee {
    int tabNumber;
    String name;

    public Employee(String name, int tabNumber) {
        this.name = name;
        this.tabNumber = tabNumber;
    }

    @Override
    public String toString() {
        return "Employee [tabNumber=" + tabNumber + ",
          name=" + name + "]";
    }
}
```

# Example 1

```java
import java.util.Comparator;
public class NameComparator implements Comparator<Employee>{
    @Override
    public int compare(Employee o1, Employee o2) {
        return o1.name.compareTo(o2.name);
    }
}
```

```java
import java.util.Comparator;
public class TabComparator implements Comparator<Employee>{
    @Override
    public int compare(Employee o1, Employee o2) {
        return o1.tabNumber - o2.tabNumber;
    }
}
```

# Example 1

```
import java.util.ArrayList;
import java.util.List;
public class Main {
    public static void main(String[] args) {
        List<Employee> list = new ArrayList<Employee>();

        list.add(new Employee("Vasya", 15));
        list.add(new Employee("Anna", 2));
        list.add(new Employee("Alina", 40));

        list.sort(new NameComparator());
        for (Employee employee : list) {
            System.out.println(employee);
        }
     list.sort(new TabComparator());
        for (Employee employee : list) {
            System.out.println(employee);
        }
    }
}
```

# Example 2

```
public class Employee {
    int tabNumber;
    String name;
    static NameComparator nameComparator =
                new NameComparator( );
    static TabComparator tabComparator =
                new TabComparator();
    public static Comparator getNameComparator( ) {
        return nameComparator;
    }
    public static Comparator getTabComparator( ) {
        return tabComparator;
    }
```

Add `get()` and `set()` methods

# Example 2

```
static class NameComparator implements Comparator {
  public int compare(Object o1, Object o2) {
    return ((Employer)o1).name.compareTo(((Employer)o2).name);
  }
}

static class TabComparator implements Comparator {
  public int compare(Object o1, Object o2) {
    return ((Employer)o1).tabNumber – ((Employer)o2).tabNumber;
  }
}
. . .
}
```

# Example 2

```
public static void main(String[] args) {
Set<Employee> set =
      new TreeSet(Employee.getNameComparator());
set.add(new Employee(15, "Vasya"));
set.add(new Employee(2, "Anna"));
set.add(new Employee(40, "Alina"));
System.out.println(set);
Set<Employee> set1 =
      new TreeSet(Employee.getTabComparator());
set1.addAll(set);
System.out.println(set1);
}
```

# Map

- The most commonly used Map implementations are HashMap and TreeMap.

```
Map mapA = new HashMap();
Map mapB = new TreeMap();
```

- Adding elements

```
mapA.put("key1", "one");
mapA.put("key2", "two");

String value2 = (String) mapA.get("key2");
```

- Removing element

```
mapA.remove(Object key);
```

# Map

```
for (Map.Entry<String, String> entry : map.entrySet()) {
    System.out.println(entry.getKey() + " "
                    + entry.getValue());
}
```

```
for (Iterator i = map.entrySet().iterator(); i.hasNext();){
   Map.Entry entry = (Map.Entry) i.next();
   System.out.println(entry.getKey() + " "
                    + entry.getValue());
}
```

# Practical task 1

Declare collection *myCollection* of 10 integers and fill it (from the console or random).

- Find and save in list *newCollection* all positions of element more than 5 in the collection. Print *newCollection*
- Remove from collection *myCollection* elements, which are greater then 20. Print result
- Insert elements 1, -3, -4 in positions 2, 8, 5. Print result in the format: “position – xxx, value of element – xxx”
- Sort and print collection

*Use next Collections for this tasks: List, ArrayList, LinkedList*

# Practical task 2

In the *main*() method declare map *employeeMap* of pairs <Integer, String>.

- Add to *employeeMap* seven pairs (ID, name) of some persons. Display the map on the screen.
- Ask user to enter ID, then find and write corresponding *name* from your map. If you can't find this ID - say about it to user (use function *containsKey*()).
- Ask user to enter name, verify than you have *name* in your map and write corresponding ID. If you can't find this name - say about it to user (use function *containsValue*()).

# Homework

1. Write parameterized methods *union*(Set set1, Set set2) and *intersect*(Set set1, Set set2), realizing the operations of union and intersection of two sets. Test the operation of these techniques on two pre-filled sets.

2. Create map *personMap* and add to it ten persons of type <String, String> (*lastName*, *firstName*).
   - Output the entities of the map on the screen.
   - There are at less two persons with the same *firstName* among these 10 people?
   - Remove from the map person whose *firstName* is ”Orest” (or other). Print result.

# Homework

3. Write class Student that provides information about the name of the student and his course. Class Student should consists of
   a) properties for access to these fields
   b) constructor with parameters
   c) method *printStudents* (List students, Integer course), which receives a list of students and the course number and prints to the console the names of the students from the list, which are taught in this course (use an iterator)
   d) methods to compare students by name and by course
   e) In the *main*() method
      - declare List students and add to the list five different students
      - display the list of students ordered by name
      - display the list of students ordered by course.

softserve experience matters

# The end

**USA HQ**
Toll Free: 866-687-3588
Tel: +1-512-516-8880

**Ukraine HQ**
Tel: +380-32-240-9090

**Bulgaria**
Tel: +359-2-902-3760