

2023/2024(1)

EF234302 Object Oriented Programming

Lecture #10

# Collections: More & Immutability

Misbakhul Munir **IRFAN SUBAKTI**

司馬伊凡

Мисбакхул Мунир **Ирфан Субакти**

# Iterators

- `Iterators` are objects that allow us to iterate (or examine each element) in a collection in turn
- They are **useful** because otherwise we *don't have the ability to look at* contents of a collection if the collection is not a *list*
  - Because of the ordered nature of *lists* – we can examine the element at each index

```
// This code will print out all the strings in the collection
Collection<String> myCollection = new ArrayList<String>();
myCollection.add("Luffy, Monkey D.");
myCollection.add("Zoro, Roronoa");
myCollection.add("Robin, Nico");
Iterator<String> iter = myCollection.iterator();
while (iter.hasNext()) {
    System.out.println(iter.next());
}
```

# Iterators (continued)

```
Main.java x
1 import java.util.ArrayList;
4 public class Main {
5     public static void main(String[] args) {
6         // This code will print out all the strings in the collection
7         Collection<String> myCollection = new ArrayList<String>();
8         myCollection.add("Luffy, Monkey D.");
9         myCollection.add("Zoro, Roronoa");
10        myCollection.add("Robin, Nico");
11        Iterator<String> iter = myCollection.iterator();
12        while (iter.hasNext()) {
13            System.out.println(iter.next());
14        }
15    }
16 }
```

<terminated> Main (8) [Java Application] D:\Program\Java\JDK\bin\javaw.exe (25 Nov 2021, 13:15:33 - 13:15:33)

```
Luffy, Monkey D.
Zoro, Roronoa
Robin, Nico
```

# Iterators (continued)

- The only problem with `iterators` is that they *only* allow us *read the contents* of the underlying collection and *not change it*. If we have a **mutable object** then we can *change the object*, but *cannot set a new object in its place*. For example, the following *will not change* the contents of the collection.

```
// This code will print out all the strings in the collection
Collection<String> myCollection = new ArrayList<String>();
myCollection.add("Luffy, Monkey D.");
Iterator<String> iter = myCollection.iterator();
while (iter.hasNext()) {
    String element = iter.next();
    element += " and Zoro, Roronoa";
}
```

# Iterators (continued)

```
Main2.java ×
1 import java.util.ArrayList;
2 import java.util.Collection;
3 import java.util.Iterator;
4 public class Main2 {
5     public static void main(String[] args) {
6         // This code will print out all the strings in the collection
7         Collection<String> myCollection = new ArrayList<String>();
8         myCollection.add("Luffy, Monkey D.");
9         Iterator<String> iter = myCollection.iterator();
10        while (iter.hasNext()) {
11            String element = iter.next();
12            element += " and Zoro, Roronoa";
13            System.out.println(element);
14        }
15        System.out.println();
16        System.out.println("Print myCollection's content:");
17        iter = myCollection.iterator();
18        while (iter.hasNext()) {
19            System.out.println(iter.next());
20        }
21    }
22 }
```

Problems @ Javadoc Declaration Search Console ×

<terminated> Main2 (1) [Java Application] D:\Program\Java\JDK\bin\javaw.exe (25 Nov 2021, 13:07:27 – 13:07:30)

Luffy, Monkey D. and Zoro, Roronoa

Print myCollection's content:  
Luffy, Monkey D.

# Iterators (continued)

- Instead we would have to *create a new collection* and *add element* to the new collection as we iterated over the contents of the old collection.
- Additionally the `iterator` class does **not allow** us to iterate **backwards** over a collection, *only forwards*.
- Hence, if we wanted to implement `removeDuplicates` (as in the question from previous assignment) using collections and also in an imperative way we would have to use a *variable to remember* the previous element and a *new collection* to put the unique elements in.

# Iterators (continued)

```
public static <T> List<T> removeDuplicatesFromSortedList(List<T> input) {
    // Note that this method can't cope with removing duplicate null values
    T previous = null;
    List<T> result = new ArrayList<T>();
    result.add(input.get(0));
    Iterator<T> iter = input.iterator();
    while (iter.hasNext()) {
        T current = iter.next();
        if (previous != null && !previous.equals(current)) {
            result.add(current);
        }
        previous = current;
    }
    return result;
}
```

- `ListIterator`. There is one exception to these rules, and that is for `Lists`. `Lists` also provide a `ListIterator` that allows us to add, set, remove and iterate **backwards** (as well as **forwards**) over the list. We can find out more from the Java API.

# Iterators (continued)

```
Main3.java ×
1 import java.util.ArrayList;
2 import java.util.Iterator;
3 import java.util.List;
4 public class Main3 {
5     public static <T> List<T> removeDuplicatesFromSortedList(List<T> input) {
6         // Note that this method can't cope with removing duplicate null values
7         T previous = null;
8         List<T> result = new ArrayList<T>();
9         result.add(input.get(0));
10        Iterator<T> iter = input.iterator();
11        while (iter.hasNext()) {
12            T current = iter.next();
13            if (previous != null && !previous.equals(current)) {
14                result.add(current);
15            }
16            previous = current;
17        }
18        return result;
19    }
}
```

```
20 public static void main(String[] args) {
21     List<String> myList = new ArrayList<>();
22     myList.add("Ace, Portgas D.");
23     myList.add("Luffy, Monkey D.");
24     myList.add("Luffy, Monkey D.");
25     myList.add("Sanji, Vinsmoke");
26     myList.add("Sanji, Vinsmoke");
27     myList.add("Zoro, Roronoa");
28     myList.add("Zoro, Roronoa");
29     myList.add("Zoro, Roronoa");
30     System.out.println("Print myList's content:");
31     Iterator<String> iter = myList.iterator();
32     while (iter.hasNext()) {
33         System.out.println(iter.next());
34     }
35     System.out.println();
36     System.out.println("Call removeDuplicatesFromSortedList().");
37     myList = removeDuplicatesFromSortedList(myList);
38     System.out.println("Print myList's content after " +
39         "removeDuplicatesFromSortedList():");
40     iter = myList.iterator();
41     while (iter.hasNext()) {
42         System.out.println(iter.next());
43     }
44 }
45 }
```

```
Problems @ Javadoc Declaration Search Console ×
<terminated> Main3 (1) [Java Application] D:\Program\Java\JDK\bin\javaw.exe (23 Nov 2021, 21:43:36 – 21:43:37)
Print myList's content:
Ace, Portgas D.
Luffy, Monkey D.
Luffy, Monkey D.
Sanji, Vinsmoke
Sanji, Vinsmoke
Zoro, Roronoa
Zoro, Roronoa
Zoro, Roronoa

Call removeDuplicatesFromSortedList().
Print myList's content after removeDuplicatesFromSortedList():
Ace, Portgas D.
Luffy, Monkey D.
Sanji, Vinsmoke
Zoro, Roronoa
```



# Iterable

- All classes that implement `Collection` also implement `Iterable` as well. Any class that implements the `Iterable` interface can be more easily iterated over using the **enhanced for-loop** construct. The enhanced for loop is similar to the normal for loop, but only allows iterating over the elements in a collection in a **forward** manner and one at a time.

```
Collection<Integer> c = new ArrayList<Integer> ();
```

```
c.add(3);
```

```
c.add(2);
```

```
c.add(4);
```

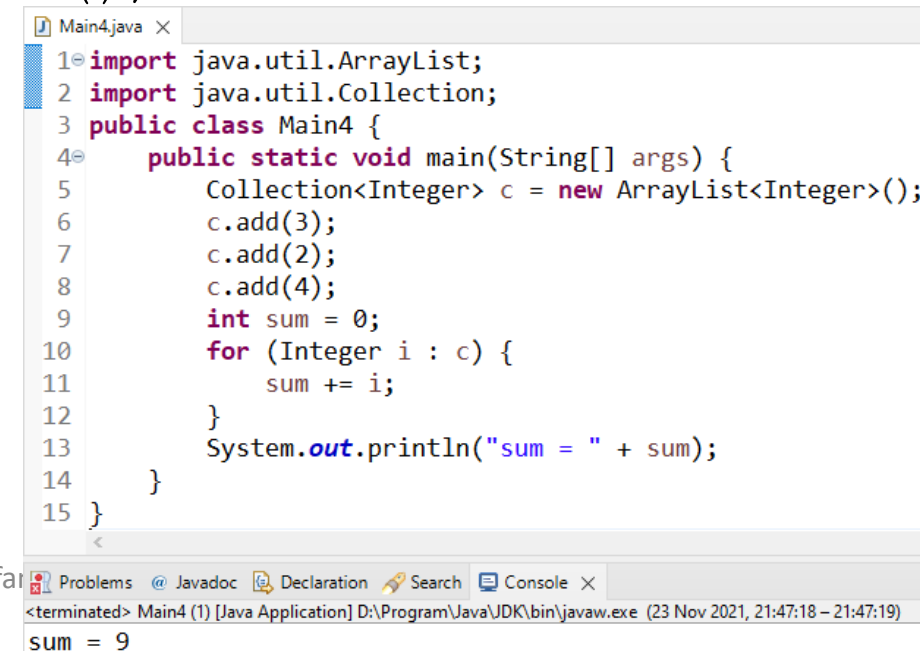
```
int sum = 0;
```

```
for (Integer i : c) {
```

```
    sum += i;
```

```
}
```

```
System.out.println("sum = " + sum);
```



```
Main4.java x
1 import java.util.ArrayList;
2 import java.util.Collection;
3 public class Main4 {
4     public static void main(String[] args) {
5         Collection<Integer> c = new ArrayList<Integer>();
6         c.add(3);
7         c.add(2);
8         c.add(4);
9         int sum = 0;
10        for (Integer i : c) {
11            sum += i;
12        }
13        System.out.println("sum = " + sum);
14    }
15 }
```

<terminated> Main4 (1) [Java Application] D:\Program\Java\JDK\bin\javaw.exe (23 Nov 2021, 21:47:18 - 21:47:19)  
sum = 9

# Concurrent modification

- We've already been warned that we *cannot modify collection whilst iterating over it* (because the iterator does *not provide sufficient methods* to access it).
  - List: add & remove → it cannot, but set → it can!
- This is also because if we modify the contents of a collection then it usually does not make sense which should be the next object to be retrieved by the iterator.
- For instance, if iterating over the list ["a", "b", "c", "d", "e"] and the iterator has just returned "b" and so next element should be "c" at index 2
  - Then what should happen if we removed "a" from the beginning of the list?
  - Should the iterator still return "c" (now at index 1), or should it return the element at index 2 ("d")?
  - The answer is unclear, and so instead the iterator throws a runtime exception of the type `"ConcurrentModificationException"`.
- This is something to be aware of, and to know why this exception has been thrown (because the list was *modified whilst the list was being iterated over*). We should also be aware that this behaviour is not *fool proof* and so should not be relied upon.

# Concurrent modification (cont'd)

```
Main3.java x
1 import java.util.ArrayList;
2 import java.util.Iterator;
3 import java.util.List;
4 public class Main3 {
5     public static <T> List<T> removeDuplicatesFromSortedList(List<T> input) {
6         // Note that this method can't cope with removing duplicate null values
7         T previous = null;
8         List<T> result = new ArrayList<T>();
9         result.add(input.get(0));
10        Iterator<T> iter = input.iterator();
11        while (iter.hasNext()) {
12            T current = iter.next();
13            if (previous != null && !previous.equals(current)) {
14                result.add(current);
15            }
16            previous = current;
17        }
18        return result;
19    }
20    public static void main(String[] args) {
21        List<String> myList = new ArrayList<>();
22        myList.add("Ace, Portgas D.");
23        myList.add("Luffy, Monkey D.");
24        myList.add("Luffy, Monkey D.");
25        myList.add("Sanji, Vinsmoke");
26        myList.add("Sanji, Vinsmoke");
27        myList.add("Zoro, Roronoa");
28        myList.add("Zoro, Roronoa");
29        myList.add("Zoro, Roronoa");
30    }
}
```

20.11.2023

2023/2024(1) – Object O

```
Problems @ Javadoc Declaration Search Console x
<terminated> Main3 (1) [Java Application] D:\Program\Java\JDK\bin\javaw.exe (25 Nov 2021, 18:23:36 – 18:23:36)
Print myList's content:
Ace, Portgas D.
Luffy, Monkey D.
Luffy, Monkey D.
Sanji, Vinsmoke
Sanji, Vinsmoke
Zoro, Roronoa
Zoro, Roronoa
Zoro, Roronoa

Call removeDuplicatesFromSortedList().
Print myList's content after removeDuplicatesFromSortedList():
Ace, Portgas D.
Luffy, Monkey D.
Sanji, Vinsmoke
Zoro, Roronoa

Update the list (index 2 = Sanji) in the middle of iterating:
Ace, Portgas D.
Luffy, Monkey D.
Sabo
Zoro, Roronoa

System.out.println("Print myList's content:");
Iterator<String> iter = myList.iterator();
while (iter.hasNext()) {
    System.out.println(iter.next());
}
System.out.println();
System.out.println("Call removeDuplicatesFromSortedList().");
myList = removeDuplicatesFromSortedList(myList);
System.out.println("Print myList's content after " +
    "removeDuplicatesFromSortedList():");
iter = myList.iterator();
while (iter.hasNext()) {
    System.out.println(iter.next());
}
System.out.println();
iter = myList.iterator();
System.out.println("Update the list (index 2 = Sanji) in the middle of iterating:");
while (iter.hasNext()) {
    System.out.println(iter.next());
    // ConcurrentModificationException: ADD & REMOVE
    // But, it's not happened for SET
    myList.add("Buggy"); // It CANNOT! Adding in the middle of iterating
    myList.set(2, "Sabo"); // It CAN! Updating in the middle of iterating
    myList.remove(2); // It CANNOT! Removing in the middle of iterating
}
}
```

# Comparable & comparators

- When using lists, it can often be useful to *sort* all the elements into order
- This can be done by using the `Collections.sort` method
- However, **not all lists can be sorted**
- To sort a list then the generic type of the list must either implement `Comparable` or we must have written a `Comparator` class for the objects we wish to sort
- We saw the `Comparable` interface weeks ago, when we looked at the bounded types for generics. Indeed, the `Collections.sort` method uses bounded types to restrict the lists that it can sort

```
public static <E extends Comparable<E>> void sort(List<E> list) {  
    ...  
}
```

# Comparable & comparators (continued)

- All the `Comparable` interface is saying is that this class has an inherent order and knows how to *sort* itself.
- But what if it is **not obvious** *how a class should be sorted* – then including the `Comparable` interface would *confusing* more than anything
  - For example, with student records: Each student has a first name, last name, age, degree course, marks, etc.
  - But if I had a list students, how should they be sorted?
  - Well, it depends on what information I need to know from that list
- And this is why the `Comparator` interface exists
- `Comparators` are able to **compare two instances of a class** and say which *order* they should come in
  - For example, we could write an *age* `Comparator` for the student class

# Comparable & comparators (continued)

```
public class AgeComparator implements Comparator<Student> {
    @Override
    public int compare(Student s1, Student s2) {
        if (s1 == s2) {
            return 0; // The same age
        } else if (s1 == null) {
            return -1; // s1 should come before s2
        } else if (s2 == null) {
            return 1; // s1 should come after s2
        } else {
            return s1.getAge() - s2.getAge();
        }
    }
}
```

```
AgeComparator.java X
1 package student;
2 import java.util.Comparator;
3 //Using list -> sort all the element in order, by Collections.sort()
4 //Not all lists can be sorted.
5 //To sort a generic type of list:
6 //(1) implement Comparable
7 //(2) write a Comparator class for the objects we want to sort
8 public class AgeComparator implements Comparator<Student> {
9     @Override
10    public int compare(Student s1, Student s2) {
11        if (s1 == s2) {
12            return 0; // The same age
13        } else if (s1 == null) {
14            return -1; // s1 should come before s2
15        } else if (s2 == null) {
16            return 1; // s1 should come after s2
17        } else {
18            return s1.getAge() - s2.getAge();
19        }
20    }
21 }
```

# Comparable & comparators (continued)

```
Student.java x MyCollection.java x
1 package student;
2 public class Student {
3     final private int age;
4     Student(int age) {
5         this.age = age;
6     }
7     public int getAge() {
8         return age;
9     }
10 }

1 package student;
2 import java.util.ArrayList;
3 import java.util.Collection;
4 import java.util.Collections;
5 import java.util.Iterator;
6 import java.util.List;
7 public class MyCollection {
8     public static void main(String[] args) {
9         Collection<Student> students = new ArrayList<>();
10        Student s = new Student(15);
11        students.add(s);
12        Student s1 = new Student(20);
13        students.add(s1);
14        Student s2 = new Student(12);
15        students.add(s2);
16        Iterator<Student> itStudents = students.iterator();
17        System.out.println("Print the list of student:");
18        while (itStudents.hasNext()) {
19            System.out.println(itStudents.next().getAge());
20        }
21        AgeComparator age = new AgeComparator();
22        System.out.println();
23        System.out.printf("Age difference between s1 (age %d) and s2 (age %d) = %d\n",
24            s1.getAge(), s2.getAge(), age.compare(s1, s2));
25        System.out.println();
26        System.out.println("After sort is called, the list of student becomes:");
27        Collections.sort((List<Student>) students, new AgeComparator());
28        Iterator<Student> itStudents2 = students.iterator();
29        while (itStudents2.hasNext()) {
30            System.out.println(itStudents2.next().getAge());
31        }
32    }
33 }
```

```
Problems @ Javadoc Declaration Search Console x
<terminated> MyCollection (1) [Java Application] D:\Program\Java\JDK\bin\javaw.exe (23 Nov 2021, 2
Print the list of student:
15
20
12

Age difference between s1 (age 20) and s2 (age 12) = 8

After sort is called, the list of student becomes:
12
15
20
```

# First name, last name: Sorting

- Exercise: How would we write a comparator that orders the students alphabetically by their full name (first name compared first, then last name)?
- Case 1
  1. Luffy, Monkey D.
  2. Garp, Monkey D.
  3. Dragon, Monkey D.
  4. Rouge, Portgas D.
  5. Ace, Portgas D.
- Case 2
  1. Luffy, Monkey D.
  2. Rouge, Portgas D.
  3. Garp, Monkey D.
  4. Ace, Portgas D.
  5. Dragon, Monkey D.



# First name, last name: Code

```
1 package name;
2 public class Name implements Comparable<Name> {
3     private String firstName, lastName;
4     public Name(String firstName, String lastName) {
5         this.firstName = firstName;
6         this.lastName = lastName;
7     }
8     @Override
9     public int compareTo(Name other) {
10        int result = Integer.MIN_VALUE;
11        switch (firstName.compareTo(other.firstName)) {
12            case -1:
13                result = -1;
14                break;
15            case 0:
16                result = lastName.compareTo(other.lastName);
17                break;
18            case 1:
19                result = 1;
20                break;
21        }
22        return result;
23    }
24    @Override
25    public String toString() {
26        return "[" + firstName + ", " + lastName + "];"
27    }
28 }
```

```
1 package name;
2 public class Main {
3     public static void main(String[] args) {
4         Name n1 = new Name("Luffy", "Monkey D.");
5         Name n2 = new Name("Garp", "Monkey D.");
6         Name n3 = new Name("Dragon", "Monkey D.");
7         Name n4 = new Name("Rouge", "Portgas D.");
8         Name n5 = new Name("Ace", "Portgas D.");
9         Name names[] = {n1, n2, n3, n4, n5};
10        System.out.println("Case 1");
11        System.out.println("Original order:");
12        for (Name n : names) {
13            System.out.println(n);
14        }
15        java.util.Arrays.sort(names);
16        System.out.println();
17        System.out.println("After sorting:");
18        for (Name n : names) {
19            System.out.println(n);
20        }
21    }
22 }
```

Problems @ Javadoc Dec  
<terminated> Main (9) [Java Appli

Case 1  
Original order:  
[Luffy, Monkey D.]  
[Garp, Monkey D.]  
[Dragon, Monkey D.]  
[Rouge, Portgas D.]  
[Ace, Portgas D.]

After sorting:  
[Ace, Portgas D.]  
[Rouge, Portgas D.]  
[Dragon, Monkey D.]  
[Garp, Monkey D.]  
[Luffy, Monkey D.]

Case 2  
Original order:  
[Luffy, Monkey D.]  
[Rouge, Portgas D.]  
[Garp, Monkey D.]  
[Ace, Portgas D.]  
[Dragon, Monkey D.]

After sorting:  
[Dragon, Monkey D.]  
[Ace, Portgas D.]  
[Garp, Monkey D.]  
[Rouge, Portgas D.]  
[Luffy, Monkey D.]

```
System.out.println();
n1 = new Name("Luffy", "Monkey D.");
n2 = new Name("Rouge", "Portgas D.");
n3 = new Name("Garp", "Monkey D.");
n4 = new Name("Ace", "Portgas D.");
n5 = new Name("Dragon", "Monkey D.");
Name names2[] = {n1, n2, n3, n4, n5};
System.out.println("Case 2");
System.out.println("Original order:");
for (Name n : names2) {
    System.out.println(n);
}
java.util.Arrays.sort(names2);
System.out.println();
System.out.println("After sorting:");
for (Name n : names2) {
    System.out.println(n);
}
}
```

# Last name, first name: Sorting

- Exercise: How would we write a comparator that orders the students alphabetically by their full name (last name compared first, then first name)?
- Case 1
  1. Luffy, Monkey D.
  2. Garp, Monkey D.
  3. Dragon, Monkey D.
  4. Rouge, Portgas D.
  5. Ace, Portgas D.
- Case 2
  1. Luffy, Monkey D.
  2. Rouge, Portgas D.
  3. Garp, Monkey D.
  4. Ace, Portgas D.
  5. Dragon, Monkey D.

# Last name, first name: Code

```
1 package name;
2 public class Name2 implements Comparable<Name2> {
3     private String lastName, firstName;
4     public Name2(String lastName, String firstName) {
5         this.lastName = lastName;
6         this.firstName = firstName;
7     }
8     @Override
9     public int compareTo(Name2 other) {
10        int result = Integer.MIN_VALUE;
11        switch (lastName.compareTo(other.lastName)) {
12            case -1:
13                result = -1;
14                break;
15            case 0:
16                result = firstName.compareTo(other.firstName);
17                break;
18            case 1:
19                result = 1;
20                break;
21        }
22        return result;
23    }
24    @Override
25    public String toString() {
26        return "[" + lastName + ", " + firstName + "];"
27    }
28 }
```

20.11.2023

```
1 package name;
2 import java.util.ArrayList;
3 import java.util.Collections;
4 import java.util.List;
5 public class Main2 {
6     public static void main(String[] args) {
7         Name2 n1 = new Name2("Monkey D.", "Luffy");
8         Name2 n2 = new Name2("Monkey D.", "Garp");
9         Name2 n3 = new Name2("Monkey D.", "Dragon");
10        Name2 n4 = new Name2("Portgas D.", "Rouge");
11        Name2 n5 = new Name2("Portgas D.", "Ace");
12        List<Name2> names = new ArrayList<>();
13        names.add(n1);
14        names.add(n2);
15        names.add(n3);
16        names.add(n4);
17        names.add(n5);
18        System.out.println("Case 1");
19        System.out.println("Original order:");
20        for (Name2 n : names) {
21            System.out.println(n);
22        }
23        Collections.sort(names);
24        System.out.println();
25        System.out.println("After sorting:");
26        for (Name2 n : names) {
27            System.out.println(n);
28        }
29        System.out.println();
30        n1 = new Name2("Monkey D.", "Luffy");
31        n2 = new Name2("Portgas D.", "Rouge");
32        n3 = new Name2("Monkey D.", "Garp");
33        n4 = new Name2("Portgas D.", "Ace");
34        n5 = new Name2("Monkey D.", "Dragon");
```

20

```
Problems @ Javadoc Dec
<terminated> Main2 (2) [Java Applic
Case 1
Original order:
[Monkey D., Luffy]
[Monkey D., Garp]
[Monkey D., Dragon]
[Portgas D., Rouge]
[Portgas D., Ace]
Case 2
Original order:
[Monkey D., Luffy]
[Portgas D., Rouge]
[Monkey D., Garp]
[Portgas D., Ace]
[Monkey D., Dragon]
After sorting:
[Portgas D., Ace]
[Portgas D., Rouge]
[Monkey D., Dragon]
[Monkey D., Garp]
[Monkey D., Luffy]
After sorting:
[Monkey D., Dragon]
[Portgas D., Ace]
[Monkey D., Garp]
[Portgas D., Rouge]
[Monkey D., Luffy]
35 List<Name2> names2 = new ArrayList<>();
36 names2.add(n1);
37 names2.add(n2);
38 names2.add(n3);
39 names2.add(n4);
40 names2.add(n5);
41 System.out.println("Case 2");
42 System.out.println("Original order:");
43 for (Name2 n : names2) {
44     System.out.println(n);
45 }
46 Collections.sort(names2);
47 System.out.println();
48 System.out.println("After sorting:");
49 for (Name2 n : names2) {
50     System.out.println(n);
51 }
52 }
53 }
```

19

# Last name + first name: Sorting

- Exercise: How would we write a comparator that orders the students alphabetically by their full name (last name + first name)?
- Case 1
  1. Luffy, Monkey D.
  2. Garp, Monkey D.
  3. Dragon, Monkey D.
  4. Rouge, Portgas D.
  5. Ace, Portgas D.
- Case 2
  1. Luffy, Monkey D.
  2. Rouge, Portgas D.
  3. Garp, Monkey D.
  4. Ace, Portgas D.
  5. Dragon, Monkey D.

# Last name + first name: Code

```
LastFirstName.java ×
1 package name;
2 public class LastFirstName {
3     private String lastName, firstName;
4     LastFirstName(String lastName, String firstName) {
5         this.lastName = lastName;
6         this.firstName = firstName;
7     }
8     public String getLastName() {
9         return lastName;
10    }
11    public void setLastName(String lastName) {
12        this.lastName = lastName;
13    }
14    public String getFirstName() {
15        return firstName;
16    }
17    public void setFirstName(String firstName) {
18        this.firstName = firstName;
19    }
20    @Override
21    public String toString() {
22        return "[" + lastName + ", " + firstName + "];"
23    }
24 }
```

```
NameComparator.java ×
1 package name;
2 import java.util.Comparator;
3 public class NameComparator implements Comparator<LastFirstName> {
4     @Override
5     public int compare(LastFirstName n1, LastFirstName n2) {
6         String s1 = n1.getLastName() + n1.getFirstName();
7         String s2 = n2.getLastName() + n2.getFirstName();
8         return s1.compareTo(s2);
9     }
10 }
```

# Last name + first name: Code (continued)

```
LastFirstNameTest.java × 25
1 package name; 26
2 import java.util.ArrayList; 27
3 import java.util.Collections; 28
4 import java.util.List; 29
5 public class LastFirstNameTest { 30
6     public static void main(String[] args) { 31
7         LastFirstName n1 = new LastFirstName("Monkey D.", "Luffy"); 32
8         LastFirstName n2 = new LastFirstName("Monkey D.", "Garp"); 33
9         LastFirstName n3 = new LastFirstName("Monkey D.", "Dragon"); 34
10        LastFirstName n4 = new LastFirstName("Portgas D.", "Rouge"); 35
11        LastFirstName n5 = new LastFirstName("Portgas D.", "Ace"); 36
12        List<LastFirstName> names = new ArrayList<>(); 37
13        names.add(n1); 38
14        names.add(n2); 39
15        names.add(n3); 40
16        names.add(n4); 41
17        names.add(n5); 42
18        System.out.println("Case 1"); 43
19        System.out.println("Original order:"); 44
20        for (LastFirstName n : names) { 45
21            System.out.println(n); 46
22        } 47
23        Collections.sort((List<LastFirstName>) names, new NameComparator()); 48
24        System.out.println(); 49
25 50
26 51
27 52
28 53
29 54
30 55
31 56
32 57
33 58
34 59
35 60
36 61
37 62
38 63
39 64
40 65
41 66
42 67
43 68
44 69
45 70
46 71
47 72
48 73
49 74
50 75
51 76
52 77
53 78
54 79
55 80
56 81
57 82
58 83
59 84
60 85
61 86
62 87
63 88
64 89
65 90
66 91
67 92
68 93
69 94
70 95
71 96
72 97
73 98
74 99
75 100
76 101
77 102
78 103
79 104
80 105
81 106
82 107
83 108
84 109
85 110
86 111
87 112
88 113
89 114
90 115
91 116
92 117
93 118
94 119
95 120
96 121
97 122
98 123
99 124
100 125
101 126
102 127
103 128
104 129
105 130
106 131
107 132
108 133
109 134
110 135
111 136
112 137
113 138
114 139
115 140
116 141
117 142
118 143
119 144
120 145
121 146
122 147
123 148
124 149
125 150
126 151
127 152
128 153
129 154
130 155
131 156
132 157
133 158
134 159
135 160
136 161
137 162
138 163
139 164
140 165
141 166
142 167
143 168
144 169
145 170
146 171
147 172
148 173
149 174
150 175
151 176
152 177
153 178
154 179
155 180
156 181
157 182
158 183
159 184
160 185
161 186
162 187
163 188
164 189
165 190
166 191
167 192
168 193
169 194
170 195
171 196
172 197
173 198
174 199
175 200
176 201
177 202
178 203
179 204
180 205
181 206
182 207
183 208
184 209
185 210
186 211
187 212
188 213
189 214
190 215
191 216
192 217
193 218
194 219
195 220
196 221
197 222
198 223
199 224
200 225
201 226
202 227
203 228
204 229
205 230
206 231
207 232
208 233
209 234
210 235
211 236
212 237
213 238
214 239
215 240
216 241
217 242
218 243
219 244
220 245
221 246
222 247
223 248
224 249
225 250
226 251
227 252
228 253
229 254
230 255
231 256
232 257
233 258
234 259
235 260
236 261
237 262
238 263
239 264
240 265
241 266
242 267
243 268
244 269
245 270
246 271
247 272
248 273
249 274
250 275
251 276
252 277
253 278
254 279
255 280
256 281
257 282
258 283
259 284
260 285
261 286
262 287
263 288
264 289
265 290
266 291
267 292
268 293
269 294
270 295
271 296
272 297
273 298
274 299
275 300
276 301
277 302
278 303
279 304
280 305
281 306
282 307
283 308
284 309
285 310
286 311
287 312
288 313
289 314
290 315
291 316
292 317
293 318
294 319
295 320
296 321
297 322
298 323
299 324
300 325
301 326
302 327
303 328
304 329
305 330
306 331
307 332
308 333
309 334
310 335
311 336
312 337
313 338
314 339
315 340
316 341
317 342
318 343
319 344
320 345
321 346
322 347
323 348
324 349
325 350
326 351
327 352
328 353
329 354
330 355
331 356
332 357
333 358
334 359
335 360
336 361
337 362
338 363
339 364
340 365
341 366
342 367
343 368
344 369
345 370
346 371
347 372
348 373
349 374
350 375
351 376
352 377
353 378
354 379
355 380
356 381
357 382
358 383
359 384
360 385
361 386
362 387
363 388
364 389
365 390
366 391
367 392
368 393
369 394
370 395
371 396
372 397
373 398
374 399
375 400
376 401
377 402
378 403
379 404
380 405
381 406
382 407
383 408
384 409
385 410
386 411
387 412
388 413
389 414
390 415
391 416
392 417
393 418
394 419
395 420
396 421
397 422
398 423
399 424
400 425
401 426
402 427
403 428
404 429
405 430
406 431
407 432
408 433
409 434
410 435
411 436
412 437
413 438
414 439
415 440
416 441
417 442
418 443
419 444
420 445
421 446
422 447
423 448
424 449
425 450
426 451
427 452
428 453
429 454
430 455
431 456
432 457
433 458
434 459
435 460
436 461
437 462
438 463
439 464
440 465
441 466
442 467
443 468
444 469
445 470
446 471
447 472
448 473
449 474
450 475
451 476
452 477
453 478
454 479
455 480
456 481
457 482
458 483
459 484
460 485
461 486
462 487
463 488
464 489
465 490
466 491
467 492
468 493
469 494
470 495
471 496
472 497
473 498
474 499
475 500
476 501
477 502
478 503
479 504
480 505
481 506
482 507
483 508
484 509
485 510
486 511
487 512
488 513
489 514
490 515
491 516
492 517
493 518
494 519
495 520
496 521
497 522
498 523
499 524
500 525
501 526
502 527
503 528
504 529
505 530
506 531
507 532
508 533
509 534
510 535
511 536
512 537
513 538
514 539
515 540
516 541
517 542
518 543
519 544
520 545
521 546
522 547
523 548
524 549
525 550
526 551
527 552
528 553
529 554
530 555
531 556
532 557
533 558
534 559
535 560
536 561
537 562
538 563
539 564
540 565
541 566
542 567
543 568
544 569
545 570
546 571
547 572
548 573
549 574
550 575
551 576
552 577
553 578
554 579
555 580
556 581
557 582
558 583
559 584
560 585
561 586
562 587
563 588
564 589
565 590
566 591
567 592
568 593
569 594
570 595
571 596
572 597
573 598
574 599
575 600
576 601
577 602
578 603
579 604
580 605
581 606
582 607
583 608
584 609
585 610
586 611
587 612
588 613
589 614
590 615
591 616
592 617
593 618
594 619
595 620
596 621
597 622
598 623
599 624
600 625
601 626
602 627
603 628
604 629
605 630
606 631
607 632
608 633
609 634
610 635
611 636
612 637
613 638
614 639
615 640
616 641
617 642
618 643
619 644
620 645
621 646
622 647
623 648
624 649
625 650
626 651
627 652
628 653
629 654
630 655
631 656
632 657
633 658
634 659
635 660
636 661
637 662
638 663
639 664
640 665
641 666
642 667
643 668
644 669
645 670
646 671
647 672
648 673
649 674
650 675
651 676
652 677
653 678
654 679
655 680
656 681
657 682
658 683
659 684
660 685
661 686
662 687
663 688
664 689
665 690
666 691
667 692
668 693
669 694
670 695
671 696
672 697
673 698
674 699
675 700
676 701
677 702
678 703
679 704
680 705
681 706
682 707
683 708
684 709
685 710
686 711
687 712
688 713
689 714
690 715
691 716
692 717
693 718
694 719
695 720
696 721
697 722
698 723
699 724
700 725
701 726
702 727
703 728
704 729
705 730
706 731
707 732
708 733
709 734
710 735
711 736
712 737
713 738
714 739
715 740
716 741
717 742
718 743
719 744
720 745
721 746
722 747
723 748
724 749
725 750
726 751
727 752
728 753
729 754
730 755
731 756
732 757
733 758
734 759
735 760
736 761
737 762
738 763
739 764
740 765
741 766
742 767
743 768
744 769
745 770
746 771
747 772
748 773
749 774
750 775
751 776
752 777
753 778
754 779
755 780
756 781
757 782
758 783
759 784
760 785
761 786
762 787
763 788
764 789
765 790
766 791
767 792
768 793
769 794
770 795
771 796
772 797
773 798
774 799
775 800
776 801
777 802
778 803
779 804
780 805
781 806
782 807
783 808
784 809
785 810
786 811
787 812
788 813
789 814
790 815
791 816
792 817
793 818
794 819
795 820
796 821
797 822
798 823
799 824
800 825
801 826
802 827
803 828
804 829
805 830
806 831
807 832
808 833
809 834
810 835
811 836
812 837
813 838
814 839
815 840
816 841
817 842
818 843
819 844
820 845
821 846
822 847
823 848
824 849
825 850
826 851
827 852
828 853
829 854
830 855
831 856
832 857
833 858
834 859
835 860
836 861
837 862
838 863
839 864
840 865
841 866
842 867
843 868
844 869
845 870
846 871
847 872
848 873
849 874
850 875
851 876
852 877
853 878
854 879
855 880
856 881
857 882
858 883
859 884
860 885
861 886
862 887
863 888
864 889
865 890
866 891
867 892
868 893
869 894
870 895
871 896
872 897
873 898
874 899
875 900
876 901
877 902
878 903
879 904
880 905
881 906
882 907
883 908
884 909
885 910
886 911
887 912
888 913
889 914
890 915
891 916
892 917
893 918
894 919
895 920
896 921
897 922
898 923
899 924
900 925
901 926
902 927
903 928
904 929
905 930
906 931
907 932
908 933
909 934
910 935
911 936
912 937
913 938
914 939
915 940
916 941
917 942
918 943
919 944
920 945
921 946
922 947
923 948
924 949
925 950
926 951
927 952
928 953
929 954
930 955
931 956
932 957
933 958
934 959
935 960
936 961
937 962
938 963
939 964
940 965
941 966
942 967
943 968
944 969
945 970
946 971
947 972
948 973
949 974
950 975
951 976
952 977
953 978
954 979
955 980
956 981
957 982
958 983
959 984
960 985
961 986
962 987
963 988
964 989
965 990
966 991
967 992
968 993
969 994
970 995
971 996
972 997
973 998
974 999
975 1000
976 1001
977 1002
978 1003
979 1004
980 1005
981 1006
982 1007
983 1008
984 1009
985 1010
986 1011
987 1012
988 1013
989 1014
990 1015
991 1016
992 1017
993 1018
994 1019
995 1020
996 1021
997 1022
998 1023
999 1024
1000 1025
1001 1026
1002 1027
1003 1028
1004 1029
1005 1030
1006 1031
1007 1032
1008 1033
1009 1034
1010 1035
1011 1036
1012 1037
1013 1038
1014 1039
1015 1040
1016 1041
1017 1042
1018 1043
1019 1044
1020 1045
1021 1046
1022 1047
1023 1048
1024 1049
1025 1050
1026 1051
1027 1052
1028 1053
1029 1054
1030 1055
1031 1056
1032 1057
1033 1058
1034 1059
1035 1060
1036 1061
1037 1062
1038 1063
1039 1064
1040 1065
1041 1066
1042 1067
1043 1068
1044 1069
1045 1070
1046 1071
1047 1072
1048 1073
1049 1074
1050 1075
1051 1076
1052 1077
1053 1078
1054 1079
1055 1080
1056 1081
1057 1082
1058 1083
1059 1084
1060 1085
1061 1086
1062 1087
1063 1088
1064 1089
1065 1090
1066 1091
1067 1092
1068 1093
1069 1094
1070 1095
1071 1096
1072 1097
1073 1098
1074 1099
1075 1100
1076 1101
1077 1102
1078 1103
1079 1104
1080 1105
1081 1106
1082 1107
1083 1108
1084 1109
1085 1110
1086 1111
1087 1112
1088 1113
1089 1114
1090 1115
1091 1116
1092 1117
1093 1118
1094 1119
1095 1120
1096 1121
1097 1122
1098 1123
1099 1124
1100 1125
1101 1126
1102 1127
1103 1128
1104 1129
1105 1130
1106 1131
1107 1132
1108 1133
1109 1134
1110 1135
1111 1136
1112 1137
1113 1138
1114 1139
1115 1140
1116 1141
1117 1142
1118 1143
1119 1144
1120 1145
1121 1146
1122 1147
1123 1148
1124 1149
1125 1150
1126 1151
1127 1152
1128 1153
1129 1154
1130 1155
1131 1156
1132 1157
1133 1158
1134 1159
1135 1160
1136 1161
1137 1162
1138 1163
1139 1164
1140 1165
1141 1166
1142 1167
1143 1168
1144 1169
1145 1170
1146 1171
1147 1172
1148 1173
1149 1174
1150 1175
1151 1176
1152 1177
1153 1178
1154 1179
1155 1180
1156 1181
1157 1182
1158 1183
1159 1184
1160 1185
1161 1186
1162 1187
1163 1188
1164 1189
1165 1190
1166 1191
1167 1192
1168 1193
1169 1194
1170 1195
1171 1196
1172 1197
1173 1198
1174 1199
1175 1200
1176 1201
1177 1202
1178 1203
1179 1204
1180 1205
1181 1206
1182 1207
1183 1208
1184 1209
1185 1210
1186 1211
1187 1212
1188 1213
1189 1214
1190 1215
1191 1216
1192 1217
1193 1218
1194 1219
1195 1220
1196 1221
1197 1222
1198 1223
1199 1224
1200 1225
1201 1226
1202 1227
1203 1228
1204 1229
1205 1230
1206 1231
1207 1232
1208 1233
1209 1234
1210 1235
1211 1236
1212 1237
1213 1238
1214 1239
1215 1240
1216 1241
1217 1242
1218 1243
1219 1244
1220 1245
1221 1246
1222 1247
1223 1248
1224 1249
1225 1250
1226 1251
1227 1252
1228 1253
1229 1254
1230 1255
1231 1256
1232 1257
1233 1258
1234 1259
1235 1260
1236 1261
1237 1262
1238 1263
1239 1264
1240 1265
1241 1266
1242 1267
1243 1268
1244 1269
1245 1270
1246 1271
1247 1272
1248 1273
1249 1274
1250 1275
1251 1276
1252 1277
1253 1278
1254 1279
1255 1280
1256 1281
1257 1282
1258 1283
1259 1284
1260 1285
1261 1286
1262 1287
1263 1288
1264 1289
1265 1290
1266 1291
1267 1292
1268 1293
1269 1294
1270 1295
1271 1296
1272 1297
1273 1298
1274 1299
1275 1300
1276 1301
1277 1302
1278 1303
1279 1304
1280 1305
1281 1306
1282 1307
1283 1308
1284 1309
1285 1310
1286 1311
1287 1312
1288 1313
1289 1314
1290 1315
1291 1316
1292 1317
1293 1318
1294 1319
1295 1320
1296 1321
1297 1322
1298 1323
1299 1324
1300 1325
1301 1326
1302 1327
1303 1328
1304 1329
1305 1330
1306 1331
1307 1332
1308 1333
1309 1334
1310 1335
1311 1336
1312 1337
1313 1338
1314 1339
1315 1340
1316 1341
1317 1342
1318 1343
1319 1344
1320 1345
1321 1346
1322 1347
1323 1348
1324 1349
1325 1350
1326 1351
1327 1352
1328 1353
1329 1354
1330 1355
1331 1356
1332 1357
1333 1358
1334 1359
1335 1360
1336 1361
1337 1362
1338 1363
1339 1364
1340 1365
1341 1366
1342 1367
1343 1368
1344 1369
1345 1370
1346 1371
1347 1372
1348 1373
1349 1374
1350 1375
1351 1376
1352 1377
1353 1378
1354 1379
1355 1380
1356 1381
1357 1382
1358 1383
1359 1384
1360 1385
1361 1386
1362 1387
1363 1388
1364 1389
1365 1390
1366 1391
1367 1392
1368 1393
1369 1394
1370 1395
1371 1396
1372 1397
1373 1398
1374 1399
1375 1400
1376 1401
1377 1402
1378 1403
1379 1404
1380 1405
1381 1406
1382 1407
1383 1408
1384 1409
1385 1410
1386 1411
1387 1412
1388 1413
1389 1414
1390 1415
1391 1416
1392 1417
1393 1418
1394 1419
1395 1420
1396 1421
1397 1422
1398 1423
1399 1424
1400 1425
1401 1426
1402 1427
1403 1428
1404 1429
1405 1430
1406 1431
1407 1432
1408 1433
1409 1434
1410 1435
1411 1436
1412 1437
1413 1438
1414 1439
1415 1440
1416 1441
1417 1442
1418 1443
1419 1444
1420 1445
1421 1446
1422 1447
1423 1448
1
```

# Last name + first name: Output

```
Problems @ Javadoc Decl.
<terminated> LastFirstNameTest [Java]
Case 1
Original order:
[Monkey D., Luffy]
[Monkey D., Garp]
[Monkey D., Dragon]
[Portgas D., Rouge]
[Portgas D., Ace]

After sorting:
[Monkey D., Dragon]
[Monkey D., Garp]
[Monkey D., Luffy]
[Portgas D., Ace]
[Portgas D., Rouge]

Case 2
Original order:
[Monkey D., Luffy]
[Portgas D., Rouge]
[Monkey D., Garp]
[Portgas D., Ace]
[Monkey D., Dragon]

After sorting:
[Monkey D., Dragon]
[Monkey D., Garp]
[Monkey D., Luffy]
[Portgas D., Ace]
[Portgas D., Rouge]
```

# Shallow & deep copying

- When programming it can be useful to create *copies* of objects
- However, when copying an object the issue of **shallow** copying and **deep** copying arises
- The issue can most easily be demonstrated with lists

```
List<Person> people = new ArrayList<Person> ();  
people.add(new Person ("Usopp"));  
people.add(new Person ("Nami"));  
List<Person> copy = copyList (people);  
Person p = copy.get (0);  
p.setName ("Brook");
```



# Shallow & deep copying (continued)

- Now its very easy to guess what the **copied list** should look like
  - [Person(Brook), Person(Nami)] <sup>[Brook]</sup><sub>[Nami]</sub>, but what should the *original list* look like?
- Most people would probably expect the *original list* to look like
  - [Person(Usopp), Person(Nami)] <sup>[Usopp]</sup><sub>[Nami]</sub>.
- However, also feasible is – [Person(Brook), Person(Nami)] <sup>[Brook]</sup><sub>[Nami]</sub>.
- But why is this?
- The reason is that *second list* (**copied list**) is only a **shallow copy**.
- That is, it has *copied the order of the objects*, but the *actual objects* in the list are the *same objects*.
- Hence, any *change* to the persons in the *original list* will also be **reflected** in the *copied list* and *vice versa*.
- If the *original list* remains *unchanged* then a **deep copy** has to be performed.
- That is the *persons themselves* were also *copied*.

# Shallow copying

```
Person.java x Main5.java x
1 public class Person {
2     private String name;
3     Person(String name) {
4         this.name = name;
5     }
6     public String getName() {
7         return name;
8     }
9     public void setName(String name) {
10        this.name = name;
11    }
12    @Override
13    public String toString() {
14        return "[" + name + "]";
15    }
16 }

1 import java.util.ArrayList;
2 import java.util.Collections;
3 import java.util.Iterator;
4 import java.util.List;
5 public class Main5 {
6     private static List<Person> copyList(List<Person> people) {
7         List<Person> result = new ArrayList<Person>(people);
8         Collections.copy(result, people);
9         return result;
10    }
11    public static void main(String[] args) {
12        List<Person> people = new ArrayList<>();
13        people.add(new Person("Usopp"));
14        people.add(new Person("Nami"));
15        System.out.println("Print the original list:");
16        Iterator<Person> iter = people.iterator();
17        while (iter.hasNext()) {
18            System.out.println(iter.next());
19        }
20        System.out.println();
21    }
22 }
23 }
24 }
25 }
26 }
27 }
28 }
29 }
30 }
31 }
32 }
33 }
34 }
35 }
36 }
37 }
38 }
39 }
40 }
41 }
```

```
Problems @ Javadoc Declaration Search Console x
<terminated> Main5 [Java Application] D:\Program\Java\JDK\bin\javaw.exe
Print the original list:
[Usopp]
[Nami]

Shallow copying

Update the copied list: Usopp -> Brook

Print the copied list:
[Brook]
[Nami]

Print the original list:
[Brook]
[Nami]

System.out.println("Shallow copying");
List<Person> copy = copyList(people);
System.out.println();
Person p = copy.get(0);
System.out.println("Update the copied list:" +
    " Usopp -> Brook");
p.setName("Brook");
System.out.println();
System.out.println("Print the copied list:");
iter = copy.iterator();
while (iter.hasNext()) {
    System.out.println(iter.next());
}
System.out.println();
System.out.println("Print the original list:");
iter = people.iterator();
while (iter.hasNext()) {
    System.out.println(iter.next());
}
}
```

# Deep copying

```
Person.java x Main6.java x
1 public class Person {
2     private String name;
3     Person(String name) {
4         this.name = name;
5     }
6     public String getName() {
7         return name;
8     }
9     public void setName(String name) {
10        this.name = name;
11    }
12    @Override
13    public String toString() {
14        return "[" + name + "]";
15    }
16 }

1 import java.util.ArrayList;
2 import java.util.Iterator;
3 import java.util.List;
4 public class Main6 {
5     private static List<Person> deepCopy(List<Person> people) {
6         List<Person> result = new ArrayList<>(people);
7         for (int i = 0; i < people.size(); i++) {
8             result.set(i, new Person(people.get(i).getName()));
9         }
10        return result;
11    }
12    public static void main(String[] args) {
13        List<Person> people = new ArrayList<>();
14        people.add(new Person("Usopp"));
15        people.add(new Person("Nami"));
16        System.out.println("Print the original list:");
17        Iterator<Person> iter = people.iterator();
18        while (iter.hasNext()) {
19            System.out.println(iter.next());
20        }
21        System.out.println();
22    }
23 }
24 }
25 }
26 }
27 }
28 }
29 }
30 }
31 }
32 }
33 }
34 }
35 }
36 }
37 }
38 }
39 }
40 }
41 }
42 }
```

```
Problems @ Javadoc Declaration Search Console x
<terminated> Main6 [Java Application] D:\Program\Java\JDK\bin\javaw.exe
Print the original list:
[Usopp]
[Nami]

Deep copying

Update the copied list: Usopp -> Brook

Print the copied list:
[Brook]
[Nami]

Print the original list:
[Usopp]
[Nami]

System.out.println("Deep copying");
List<Person> copy = deepCopy(people);
System.out.println();
Person p = copy.get(0);
System.out.println("Update the copied list:" +
    " Usopp -> Brook");
p.setName("Brook");
System.out.println();
System.out.println("Print the copied list:");
iter = copy.iterator();
while (iter.hasNext()) {
    System.out.println(iter.next());
}
System.out.println();
System.out.println("Print the original list:");
iter = people.iterator();
while (iter.hasNext()) {
    System.out.println(iter.next());
}
}
```

# Mutability vs immutability

- Though we may not know it, we have already encountered *mutable* and *immutable* objects in Java.
- Simply put, an **immutable object cannot have its state changed after it is created** (remember an *object* is different to a *variable*).
- One example of *immutable objects* are `Strings`.
- Once created a `String` *cannot be modified*, though we can still create new `Strings` as and when needed.
- **Mutable objects**, on the other hand, can have their *state altered once created*.
- There are a couple of *immutable collection classes* in the Collections API.

# Mutability vs immutability (continued)

- The Java website states that "*Maximum* reliance on **immutable objects** is widely *accepted* as a sound strategy for creating simple, reliable code. The benefits of immutable objects".
  - Why would this be? What possible benefits can arise from not being allowed to modify an object?
- The simplest benefit is that immutable objects make for **good keys in a HashMap** (remember when it was explained that objects should *never change whilst they are in use as keys in a map*).
- Another point is that immutable objects mean we do **not need to worry what will happen** when we *pass an object to another method*.
  - For instance, if we pass a List to a method called `medianAverage` then it has to sort the list to find the median or it may make its own copy of the list and sort that list.
  - The former behaviour is undesired as the order of the list might hold significance. However, with strings this is not a problem as the `String` class is immutable. The method that was passed the string can create new strings based on the string parameter, but it cannot alter the string we gave it.

# Mutability vs immutability (continued)

- One last thing, but important point is that **immutable objects** are very useful in **threaded applications** (we'll learn more about `Threads` later).
- That is if we wish to share objects between `threads` (think of a `thread` as **another bit of code to working at the same time** as our main program) then making our object *immutable is a safe way to share the object*.
- A simple analogy is to imagine a *global marksheet* which all the lecturers and TAs input your labs/assignments marks to. Now imagine if two TAs open this file at the same time and begin editing the document. TA A finishes first and saves the file, then TA B finishes and saves his changes (thus overwriting TA A's changes and so the marks file is left in a broken state).
- Programmers sometimes feel that *creating new objects is costly*. Now, whilst there is some **overhead**, Java is *very efficient at creating and disposing of objects*, especially small *short lived* objects (*immutable objects normally are **short lived** since they can not be modified*).

# Mutability vs immutability (continued)

- Exercise: Which of the following would it be useful to make *mutable* and which to make *immutable* and why?
  - Colour
  - Bank account
  - A 2D co-ordinate or "*point*"
  - The Integer class (a class designed to hold `int` values – useful for generics as you cannot have generics of primitive types).
  - A student class that must have a name, DoB, address and ID number

# Immutability: Practising

- Immutability can be enforced in two ways:
  - The simplest way is to *not provide any setter* methods for our class.
  - A more robust way is to declare fields `final`. Final is an additional *keyword* that is used during the declaration of a field or variable that lets the compiler know that the value of a variable is **not allowed to be changed** after it is first assigned to or that the value of a field is **not allowed to be assigned** outside of a constructor or its declaration.



# Immutability: Practising (continued)

- For an object to be *fully immutable* then its fields must also be comprised of **primitives** or **immutable objects**.
- This is the case with the Student class as `id` is an `int` (primitive) and `String` is **immutable**.

```
Student.java x
1 public class Student {
2     private static int ID_COUNTER = 1;
3     private final int id = ID_COUNTER++;
4     private final String name;
5
6     public Student(String name) {
7         this.name = name;
8     }
9     public int getId() {
10        return id;
11    }
12    public String getName() {
13        return name;
14    }
15 }
```