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
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 }
```

Problems @ Javadoc Declaration Search Console

```
<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

```java
import java.util.ArrayList;
import java.util.Collection;
import java.util.Iterator;
public class Main2 {
    public static void main(String[] args) {
        // 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";
            System.out.println(element);
        }
        System.out.println();
        System.out.println("Print myCollection's content:");
        iter = myCollection.iterator();
        while (iter.hasNext()) {
            System.out.println(iter.next());
        }
    }
}
```

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 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)

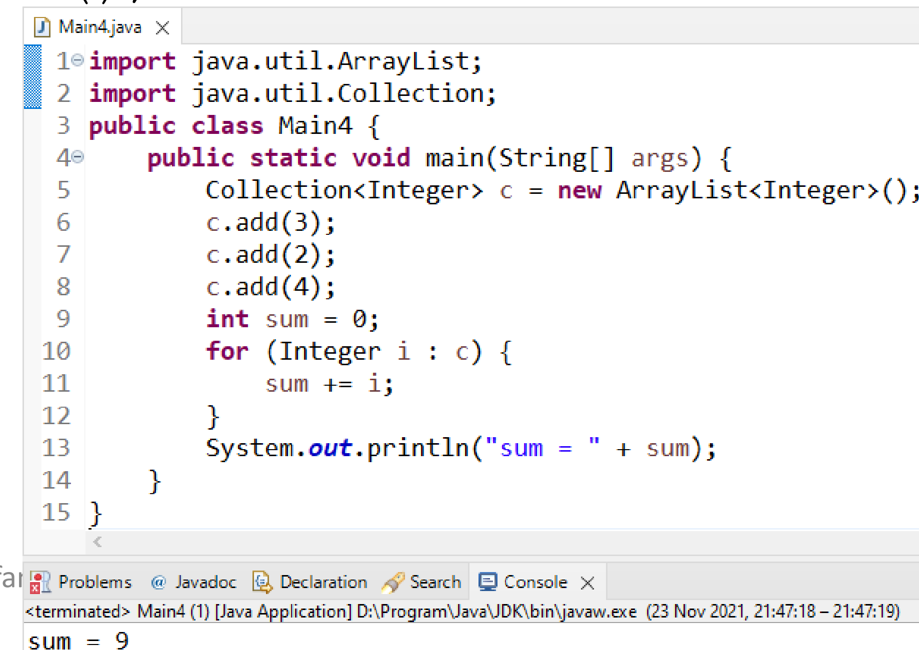
```
Print myList's content:
Ace, Portgas D.
Luffy, Monkey D.
Luffy, Monkey D.
Sanji, Vinsmoke
Sanji, Vinsmoke
Zoro, Roronoa
Zoro, Roronoa
Zoro, Roronoa

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);
```



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

```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         System.out.println();
45         iter = myList.iterator();
46         System.out.println("Update the list (index 2 = Sanji) in the middle of iterating:");
47         while (iter.hasNext()) {
48             System.out.println(iter.next());
49             // ConcurrentModificationException: ADD & REMOVE
50             // But, it's not happened for SET
51 //          myList.add("Buggy"); // It CANNOT! Adding in the middle of iterating
52             myList.set(2, "Sabo"); // It CAN! Updating in the middle of iterating
53 //          myList.remove(2); // It CANNOT! Removing in the middle of iterating
54         }
55     }
56 }
```

Problems | Javadoc | Declaration | Search | Console

<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
```

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 `Comparator` 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)

```java
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

```java
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

```java
package student;
public class Student {
    final private int age;
    Student(int age) {
        this.age = age;
    }
    public int getAge() {
        return age;
    }
}
```

MyCollection.java

```java
package student;
import java.util.ArrayList;
import java.util.Collection;
import java.util.Collections;
import java.util.Iterator;
import java.util.List;
public class MyCollection {
    public static void main(String[] args) {
        Collection<Student> students = new ArrayList<>();
        Student s = new Student(15);
        students.add(s);
        Student s1 = new Student(20);
        students.add(s1);
        Student s2 = new Student(12);
        students.add(s2);
        Iterator<Student> itStudents = students.iterator();
        System.out.println("Print the list of student:");
        while (itStudents.hasNext()) {
            System.out.println(itStudents.next().getAge());
        }
        AgeComparator age = new AgeComparator();
        System.out.println();
        System.out.printf("Age difference between s1 (age %d) and s2 (age %d) = %d\n",
                s1.getAge(), s2.getAge(), age.compare(s1, s2));
        System.out.println();
        System.out.println("After sort is called, the list of student becomes:");
        Collections.sort((List<Student>) students, new AgeComparator());
        Iterator<Student> itStudents2 = students.iterator();
        while (itStudents2.hasNext()) {
            System.out.println(itStudents2.next().getAge());
        }
    }
}
```

Console: <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

Name.java

```java
package name;
public class Name implements Comparable<Name> {
    private String firstName, lastName;
    public Name(String firstName, String lastName) {
        this.firstName = firstName;
        this.lastName = lastName;
    }
    @Override
    public int compareTo(Name other) {
        int result = Integer.MIN_VALUE;
        switch (firstName.compareTo(other.firstName)) {
        case -1:
            result = -1;
            break;
        case 0:
            result = lastName.compareTo(other.lastName);
            break;
        case 1:
            result = 1;
            break;
        }
        return result;
    }
    @Override
    public String toString() {
        return "[" + firstName + ", " + lastName + "]";
    }
}
```

Main.java

```java
package name;
public class Main {
    public static void main(String[] args) {
        Name n1 = new Name("Luffy", "Monkey D.");
        Name n2 = new Name("Garp", "Monkey D.");
        Name n3 = new Name("Dragon", "Monkey D.");
        Name n4 = new Name("Rouge", "Portgas D.");
        Name n5 = new Name("Ace", "Portgas D.");
        Name names[] = {n1, n2, n3, n4, n5};
        System.out.println("Case 1");
        System.out.println("Original order:");
        for (Name n : names) {
            System.out.println(n);
        }
        java.util.Arrays.sort(names);
        System.out.println();
        System.out.println("After sorting:");
        for (Name n : names) {
            System.out.println(n);
        }
        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);
        }
    }
}
```

Problems @ Javadoc Dec

<terminated> Main (9) [Java Applicat

```
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.]
```

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

Name2.java

```java
package name;
public class Name2 implements Comparable<Name2> {
    private String lastName, firstName;
    public Name2(String lastName, String firstName) {
        this.lastName = lastName;
        this.firstName = firstName;
    }
    @Override
    public int compareTo(Name2 other) {
        int result = Integer.MIN_VALUE;
        switch (lastName.compareTo(other.lastName)) {
        case -1:
            result = -1;
            break;
        case 0:
            result = firstName.compareTo(other.firstName);
            break;
        case 1:
            result = 1;
            break;
        }
        return result;
    }
    @Override
    public String toString() {
        return "[" + lastName + ", " + firstName + "]";
    }
}
```

Main2.java

```java
package name;
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;
public class Main2 {
    public static void main(String[] args) {
        Name2 n1 = new Name2("Monkey D.", "Luffy");
        Name2 n2 = new Name2("Monkey D.", "Garp");
        Name2 n3 = new Name2("Monkey D.", "Dragon");
        Name2 n4 = new Name2("Portgas D.", "Rouge");
        Name2 n5 = new Name2("Portgas D.", "Ace");
        List<Name2> names = new ArrayList<>();
        names.add(n1);
        names.add(n2);
        names.add(n3);
        names.add(n4);
        names.add(n5);
        System.out.println("Case 1");
        System.out.println("Original order:");
        for (Name2 n : names) {
            System.out.println(n);
        }
        Collections.sort(names);
        System.out.println();
        System.out.println("After sorting:");
        for (Name2 n : names) {
            System.out.println(n);
        }
        System.out.println();
        n1 = new Name2("Monkey D.", "Luffy");
        n2 = new Name2("Portgas D.", "Rouge");
        n3 = new Name2("Monkey D.", "Garp");
        n4 = new Name2("Portgas D.", "Ace");
        n5 = new Name2("Monkey D.", "Dragon");
        List<Name2> names2 = new ArrayList<>();
        names2.add(n1);
        names2.add(n2);
        names2.add(n3);
        names2.add(n4);
        names2.add(n5);
        System.out.println("Case 2");
        System.out.println("Original order:");
        for (Name2 n : names2) {
            System.out.println(n);
        }
        Collections.sort(names2);
        System.out.println();
        System.out.println("After sorting:");
        for (Name2 n : names2) {
            System.out.println(n);
        }
    }
}
```

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]

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

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

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

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

```java
package name;
public class LastFirstName {
    private String lastName, firstName;
    LastFirstName(String lastName, String firstName) {
        this.lastName = lastName;
        this.firstName = firstName;
    }
    public String getLastName() {
        return lastName;
    }
    public void setLastName(String lastName) {
        this.lastName = lastName;
    }
    public String getFirstName() {
        return firstName;
    }
    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }
    @Override
    public String toString() {
        return "[" + lastName + ", " + firstName + "]";
    }
}
```

NameComparator.java

```java
package name;
import java.util.Comparator;
public class NameComparator implements Comparator<LastFirstName> {
    @Override
    public int compare(LastFirstName n1, LastFirstName n2) {
        String s1 = n1.getLastName() + n1.getFirstName();
        String s2 = n2.getLastName() + n2.getFirstName();
        return s1.compareTo(s2);
    }
}
```

Last name + first name: Code (continued)

LastFirstNameTest.java

```
1  package name;
2  import java.util.ArrayList;
3  import java.util.Collections;
4  import java.util.List;
5  public class LastFirstNameTest {
6      public static void main(String[] args) {
7          LastFirstName n1 = new LastFirstName("Monkey D.", "Luffy");
8          LastFirstName n2 = new LastFirstName("Monkey D.", "Garp");
9          LastFirstName n3 = new LastFirstName("Monkey D.", "Dragon");
10         LastFirstName n4 = new LastFirstName("Portgas D.", "Rouge");
11         LastFirstName n5 = new LastFirstName("Portgas D.", "Ace");
12         List<LastFirstName> 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 (LastFirstName n : names) {
21             System.out.println(n);
22         }
23         Collections.sort((List<LastFirstName>) names, new NameComparator());
24         System.out.println();
25         System.out.println("After sorting:");
26         for (LastFirstName n : names) {
27             System.out.println(n);
28         }
29         System.out.println();
30         n1 = new LastFirstName("Monkey D.", "Luffy");
31         n2 = new LastFirstName("Portgas D.", "Rouge");
32         n3 = new LastFirstName("Monkey D.", "Garp");
33         n4 = new LastFirstName("Portgas D.", "Ace");
34         n5 = new LastFirstName("Monkey D.", "Dragon");
35         List<LastFirstName> 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 (LastFirstName n : names2) {
44             System.out.println(n);
45         }
46         Collections.sort((List<LastFirstName>) names2, new NameComparator());
47         System.out.println();
48         System.out.println("After sorting:");
49         for (LastFirstName n : names2) {
50             System.out.println(n);
51         }
52     }
53 }
```

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)] `[Brook]` `[Nami]`, but what should the *original list* look like?
- Most people would probably expect the *original list* to look like
  - [Person(Usopp), Person(Nami)] `[Usopp]` `[Nami]`.
- However, also feasible is – [Person(Brook), Person(Nami)] `[Brook]` `[Nami]`.
- 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

```java
public class Person {
    private String name;
    Person(String name) {
        this.name = name;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    @Override
    public String toString() {
        return "[" + name + "]";
    }
}
```

Main5.java

```java
import java.util.ArrayList;
import java.util.Collections;
import java.util.Iterator;
import java.util.List;
public class Main5 {
    private static List<Person> copyList(List<Person> people) {
        List<Person> result = new ArrayList<Person>(people);
        Collections.copy(result, people);
        return result;
    }
    public static void main(String[] args) {
        List<Person> people = new ArrayList<>();
        people.add(new Person("Usopp"));
        people.add(new Person("Nami")) ;
        System.out.println("Print the original list:");
        Iterator<Person> iter = people.iterator();
        while (iter.hasNext()) {
            System.out.println(iter.next());
        }
        System.out.println();
        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());
        }
    }
}
```

Problems @ Javadoc Declaration Search Console

<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]
```

Deep copying

Person.java

```java
public class Person {
    private String name;
    Person(String name) {
        this.name = name;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    @Override
    public String toString() {
        return "[" + name + "]";
    }
}
```

Main6.java

```java
import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;
public class Main6 {
    private static List<Person> deepCopy(List<Person> people) {
        List<Person> result = new ArrayList<>(people);
        for (int i = 0; i < people.size(); i++) {
            result.set(i, new Person(people.get(i).getName()));
        }
        return result;
    }
    public static void main(String[] args) {
        List<Person> people = new ArrayList<>();
        people.add(new Person("Usopp"));
        people.add(new Person("Nami")) ;
        System.out.println("Print the original list:");
        Iterator<Person> iter = people.iterator();
        while (iter.hasNext()) {
            System.out.println(iter.next());
        }
        System.out.println();
        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());
        }
    }
}
```

Console

```
<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]
```

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

```
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 }
```