Object Oriented Programming in Java

11: Collections and custom classes

Licence

You are free to

- **Share** — copy and redistribute the material in any medium or format
- **Adapt** — remix, transform, and build upon the material

under the following terms

- **Attribution** — You must give appropriate credit, provide a link to the license, and indicate if changes were made. You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use.
- **NonCommercial** — You may not use the material for commercial purposes.
- **ShareAlike** — If you remix, transform, or build upon the material, you must distribute your contributions under the same license as the original.

- The samples and slides are inspired by the Object Oriented Programming course at the University of Zagreb, Faculty of Electrical Engineering and Computing, Zagreb, Croatia.
  - Original materials in Croatian were created by (in alphabetical order): Ivica Botički, Marko Čupić, Mario Kušek, Boris Milašinović, and Krešimir Pripužić under CC-BY-NC-SA licence.
  - Adapted for this course by Boris Milašinović and shared under the same licence
  - https://creativecommons.org/licenses/by-nc-sa/4.0/

Using custom classes in collections

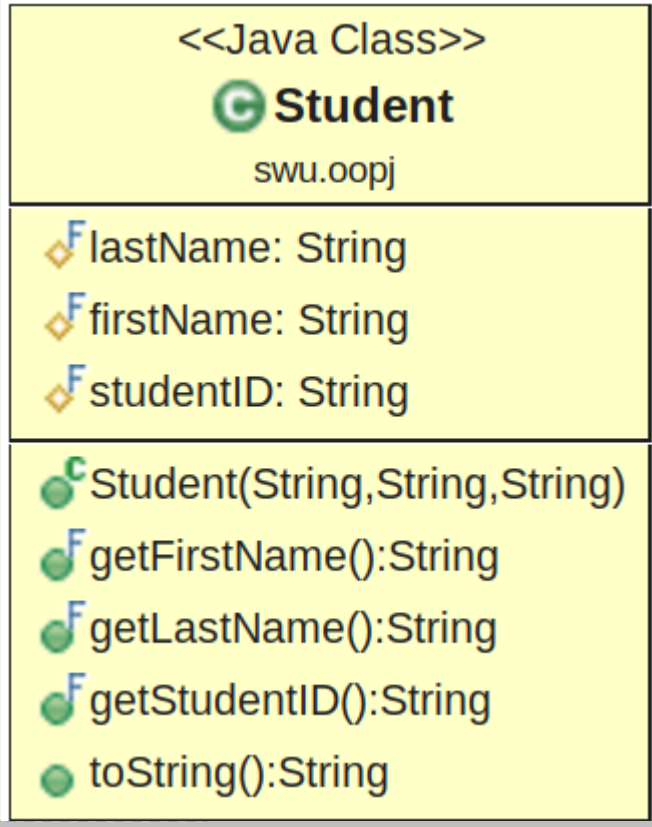
- In previous examples we had used built-in classes (String, Integer, …) as keys in maps or elements of lists and sets
- This presentation describe what is needed to use custom classes in built-in collections
- We shall start with the simple implementation of class Student and extend it through the examples by overriding or implementing
  - equals, hashCode, Comparable, …

Initial version of the custom class

11_.../swu/oopj/Student.java

- Initial implementation of Student contains constructor with parameters for last name, first name, and id
  - Values are stored as *protected final* variables having only *public final* (not overridable) getters and overridden method *toString*

```
public class Student {
...
   public Student(String lastName, String
        firstName, String studentID) {
               this.lastName = lastName;
               this.firstName = firstName;
               this.studentID = studentID;
        }
   @Override
   public String toString() {
        return String.format("(%s) %s %s",
              studentID, firstName, lastName);
   } ...
```



Common methods – printing collection

- One of the common tasks is to print the content of the collection
  - The code iterates through the collection and prints each element
    - *toString* method is called for each element
  - This code could be applied for anything that is Iterable
    - Does not have any restrictions about parameter type T
      - It would be class *Student* or some of its subclasses but does not use anything specific for *Student*

11_.../swu/oopj/Common.java

```
public class Common {
	public static <T> void printCollection(Iterable<T> col){
		for(T element : col) {
			System.out.println(element);
		}
		System.out.println();
	}
	...
```

Common methods – fill collection (1)

- All examples (i.e. collections in examples) should contain same sample data. The initial idea would be to write something like

```
public static void fillStudentsCollection(Collection<Student> col) {
   Student s1 = new Student("Black", "Joe", "1234567890");
   Student s2 = new Student("Poe", "Edgar Allan", "2345678901");
   ...
   col.add(s1);
   col.add(s2);
...
```

- The problem with this approach is that we would use different versions of class Student, not by changing the initial one, but by extending it
  - e.g *class Student2 extends Student*
  - Thus constructor *new Student* is not good for example 2, and must be replaced with *new Student2*

Common methods – fill collection (2)

- We would like to achieve something like

```
public static <S extends Student> void
                        fillStudentsCollection(Collection<S> col) {
   Student s1 = new S("Black", "Joe", "1234567890");
   Student s2 = new S("Poe", "Edgar Allan", "2345678901");
   ...
   col.add(s1);
   col.add(s2);
...
```

  where each subclass of Student would have constructor with 3 parameters

- A nice idea, but unfortunately not possible in Java
  - Java generics does not allow creating new objects of generic type using *new*

Common methods – fill collection (3)

- Instead we can define a function interface

```
@FunctionalInterface
public interface StudentFactory <S extends Student> {
   S create(String lastName, String firstName, String studentID);
}
```

11_.../swu/oopj/StudentFactory.java

with method create implemented in such way that it creates a new (subclass of) *Student*

  - Methods that create new objects (as an alternative to using new) are called factory methods

- How to implement this interface using lambda expression

```
StudentFactory<Student> factory =
       (last, first, id) -> new Student(last, first, id);
```

11_.../swu/oopj/example1/ArrayListMain.java

Common methods – fill collection (4)

- For this functional interface

```
@FunctionalInterface
public interface StudentFactory <S extends Student> {
   S create(String lastName, String firstName, String studentID);
}
```

11_.../swu/oopj/StudentFactory.java

we can use lambda expression

11_.../swu/oopj/example/ArrayListMain.java

```
StudentFactory<Student> factory =
       (last, first, id) -> new Student(last, first, id);
```

or reference to a method that have the required signature

- It must have 3 String parameters and return Student
- At first it looks like we do not have such method, but there is exactly such method – Student's constructor

```
StudentFactory<Student> factory = Student::new;
```

11_.../swu/oopj/example1/ArrayListMain.java

Common methods – fill collection (5)

- Now we can complete out fill collection method

```
public static <S extends Student> void fillStudentsCollection(
                Collection<S> col, StudentFactory<S> factory){
   S s1 = factory.create("Black", "Joe", "1234567890");
   S s2 = factory.create("Poe", "Edgar Allan", "2345678901");
   ...
   col.add(s1);
   col.add(s2);
...
```

11_.../swu/oopj/Common.java

Example #1 – Find element in a list

- We are searching for student that we "think" is a member of list, but the results is *false*
  - We have another student with the same last name, first name, and id, but these are not the same objects (references) in memory
  - *ArrayList's contains* uses *equals* to compare objects

```
I have following students:
(1234567890) Joe Black
(2345678901) Edgar Allan Poe
(3456789012) Immanuel Kant
(0123456789) Joe Rock
(5687461359) Joe Black

Poe present: false
```

```
List<Student> students = new ArrayList<>();
StudentFactory<Student> factory = Student::new;
Common.fillStudentsCollection(students, factory);

System.out.println("I have following students:");
Common.printCollection(students);

Student s = new Student("Poe", "Edgar Allan", "2345678901");
System.out.println("Poe present: " + students.contains(s));
```

11_.../swu/oopj/example1/ArrayListMain.java

Rule #1 – override *equals* method

- We should define when two objects should be considered equal
  - otherwise the reference equality is used
  - Custom rule: Two students should be equal if they have same id

```
public class Student2 extends Student {
   public Student2(String lastName, String firstName, String id) {
       super(lastName, firstName, id);
   }
   @Override
   public boolean equals(Object obj) {
       if(!(obj instanceof Student2)) return false;
       Student2 other = (Student2)obj;
       return this.studentID.equals(other.studentID);
   }
}
```

11_.../swu/oopj/example2/Student2.java

Example #2 – Find element in a list (“fixed”)

- ArrayList’s is still using equals but class *Student2* has overridden equals method and compare id’s

```
I have following students:
(1234567890) Joe Black
(2345678901) Edgar Allan Poe
(3456789012) Immanuel Kant
(0123456789) Joe Rock
(5687461359) Joe Black

Poe present: true
```

11_.../swu/oopj/example2/ArrayListMain.java

```
List<Student2> students = new ArrayList<>();
Common.fillStudentsCollection(students, Student2::new);

System.out.println("I have following students:");
Common.printCollection(students);

Student2 s = new Student2("Poe", "Edgar Allan", "2345678901");
System.out.println("Poe present: " + students.contains(s));
```

Example #2 – Find element in a hashset

- Student2 has equals, but it does not help in case of HashSet
  - *HashSet* puts elements in buckets and uses hasCode to calculate in which bucket an element should be put and searched for

```
I have following students:
(1234567890) Joe Black
(2345678901) Edgar Allan Poe
(3456789012) Immanuel Kant
(0123456789) Joe Rock
(5687461359) Joe Black

Poe present: false
```

11_.../swu/oopj/example2/HashSetMain.java

```
Set<Student2> students = new HashSet<>();
Common.fillStudentsCollection(students, Student2::new);

System.out.println("I have following students:");
Common.printCollection(students);

Student2 s = new Student2("Poe", "Edgar Allan", "2345678901");
System.out.println("Poe present: " + students.contains(s));
```

Rule #2 – override *hashCode* method

- *hashCode* must be implemented in such way that
  - two same objects (in terms of equals) must have same hash
  - Note: Two objects having same hash does not have to be equal
    - Different hash means that objects are different
- *hashCode* should be implemented in a ways that evenly distributes elements into buckets

```
public class Student3 extends Student {
    public Student3(String lastName, String firstName, String id) {
        super(lastName, firstName, id);
    }
    @Override
    public int hashCode() {
        return this.studentID.hashCode();
    }
}
```

11_.../swu/oopj/example3/Student3.java

Example #3 – Find element in a hashset

- Student3 has hashcode but it still does not work properly
  - Bucket is calculated correctly, but as bucket can contain multiple elements, search in the bucket is done using equals

```
I have following students:
(1234567890) Joe Black
(2345678901) Edgar Allan Poe
(3456789012) Immanuel Kant
(0123456789) Joe Rock
(5687461359) Joe Black

Poe present: false
```

11_.../swu/oopj/example3/HashSetMain.java

```
Set<Student3> students = new HashSet<>();
Common.fillStudentsCollection(students, Student3::new);

System.out.println("I have following students:");
Common.printCollection(students);

Student3 s = new Student3("Poe", "Edgar Allan", "2345678901");
System.out.println("Poe present: " + students.contains(s));
```

Rule #3 – override both *equals* and *hashCode*

- Both *hashCode* and *equals* must be implemented
  - Must be mutually consistent

```
public class Student4 extends Student {
    public Student4(String lastName, String firstName, String id) {
        super(lastName, firstName, id);
    }
    @Override
    public boolean equals(Object obj) {
        if(!(obj instanceof Student4)) return false;
        Student4 other = (Student4)obj;
        return this.studentID.equals(other.studentID);
    }
    @Override
    public int hashCode() {
        return this.studentID.hashCode();
    }
}
```

11_.../swu/oopj/example4/Student4.java

Example #4 – Add elements to TreeSet (1)

- An attempt to add student to a TreeSet causes an exception

```
Exception in thread "main" java.lang.ClassCastException: class
swu.oopj.example4.Student4 cannot be cast to class java.lang.Comparable
        at java.base/java.util.TreeMap.compare(TreeMap.java:1291)
        at java.base/java.util.TreeMap.put(TreeMap.java:536)
        at java.base/java.util.TreeSet.add(TreeSet.java:255)
        at swu.oopj.Common.fillStudentsCollection(Common.java:20)
        at swu.oopj.example4.TreeSetMain.main(TreeSetMain.java:13)
```

```
Set<Student4> students = new TreeSet<>();
Common.fillStudentsCollection(students, Student4::new);

System.out.println("I have following students:");
Common.printCollection(students);
```

11_…/swu/oopj/example4/TreeSetMain.java

```
Student4 s = new Student4("Poe", "Edgar Allan", "2345678901");
System.out.println("Poe present: " + students.contains(s));
```

Example #4 – Add elements to TreeSet (2)

- TreeSet must compare objects in order to build the tree
  - Assumes that used type implements *Comparable* (in order to invoke *compare* method), and produces *ClassCastException* if does not
- Two possible solutions
  1. Implement *Comparable* interface in class Student
     - *Comparable* imposes a total ordering on the objects of each class that implements it. It is referred as **natural ordering**, and the class's *compareTo* method is **natural comparison method**.
     - The natural ordering for a class C is consistent with equals if and only if e1.compareTo(e2) == 0 has the same boolean value as e1.equals(e2) for every e1 and e2 of class C.
  2. Specify another comparator in a TreeSet constructor
     - Functional interface *Comparator* with *compare* method used as comparison function, imposing a total ordering on some collection of objects.

Comparable and *Comparator*

- Comparable

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

- Comparator

```
public interface Comparator<T> {
    int compare(T o1, T o2);
    … and several default methods …
}
```

- Comparison should return 0 if objects should be treated as same, less than zero if the first one is before the second one, or greater than zero otherwise
- Note the difference in number of arguments!

Example #5 – implementing natural order

- Should be consistent with equals
  - In this example derived from the version that has equals and hashcode and compares students by their ids

11_.../swu/oopj/example5/Student5.java

```
public class Student5 extends Student4 {
   public Student5(String lastName, String firstName, String id) {
       super(lastName, firstName, id);
   }
   @Override
   public int compareTo(Student5 other) {
     return this.studentID.compareTo(other.studentID);
   }
}
```

Example #6 – writing a comparator

- Could be written as a separate class, anonymous class or using lambda expression
  - Does not require existence of natural comparator
  - An example with a separate class

```
public class StudentComparator implements Comparator<Student4> {
   @Override
   public int compare(Student4 s1, Student4 s2) {
       return s1.getStudentID().compareTo(s2.getStudentID());
   }
}
```

11_.../swu/oopj/example6/StudentComparator.java

```
Set<Student4> students = new TreeSet<>(new StudentComparator());
```

  - An example with a lambda expression

11_...example6/TreeSetMain.java

```
Set<Student> students = new TreeSet<>(
  (s1, s2) -> s1.getStudentID().compareTo(s2.getStudentID())
);
```

Example #7 – complex comparisons

- A comparator that compares first by last name, then (if same) by first name, and then by id

11_.../swu/oopj/example7/StudentComparator.java

  - It is not consistent with equals…
    - … and it is not expected as we can have many comparators

```
public class StudentComparator implements Comparator<Student4> {
   @Override
   public int compare(Student4 s1, Student4 s2) {
       int r = s1.getLastName().compareTo(s2.getLastName());
       if (r != 0)
              return r;
       r = s1.getFirstName().compareTo(s2.getFirstName());
       if (r != 0)
              return r;
       return s1.getStudentID().compareTo(s2.getStudentID());
   }
}
```

Reverse comparator as a decorator example

- Suppose that we would like to implement comparison as in previous example but in descending order
  - We could write a new class (or new lambda expression)
- Better solution: write a new universal comparator that uses an existing comparator (receive it as an argument in the constructor and wraps it) to return an opposite result as result of own compare method
  - The concept is also known as ***decorator*** pattern

Example #7 – Reverse comparator

- Applicable for any comparator, not only for comparator of students
  - "Does not how to compare objects by itself, but wraps an existing comparator that knows"

```
public class ReverseComparator<T> implements Comparator<T> {
    private Comparator<T> original;
    public ReverseComparator(Comparator<T> original) {
        this.original = original;
    }

    @Override
    public int compare(T o1, T o2) {
        int r = original.compare(o1, o2);
        return -r;
    }
}
```

11_.../swu/oopj/example7/ReverseComparator.java

Example #7 – Reverse comparator (usage)

- First, we need some concrete comparator, and then we create new comparator based on it that do reverse comparison

```
I have following students:
(0123456789) Joe Rock
(2345678901) Edgar Allan Poe
(3456789012) Immanuel Kant
(5687461359) Joe Black
(1234567890) Joe Black
```

11_.../swu/oopj/example7/Main.java

```
StudentComparator comparator = new StudentComparator();
Comparator<Student4> reverse = new ReverseComparator<>(comparator);

Set<Student4> students = new TreeSet<>(reverse);
Common.fillStudentsCollection(students, Student4::new);

System.out.println("I have following students:");
Common.printCollection(students);
```

Example #7 – Built in reverse comparators

- Instead of writing custom reverse comparator we can use built in methods:
  - Default method in interface Comparator

    Comparator<Student> reverse = comparator.reversed();
  - *Collections static method reverseOrder with a comparator as argument*

    Comparator<Student> reverse = Collections.reverseOrder(comparator);
  - *Collections* static method *reverseOrder* without arguments
    - Only if class (Student) has natural order defined (it creates reverse comparator of natural order)

    Comparator<Student> reverse = Collections.~~<Student>~~reverseOrder();

```
StudentComparator comparator = new StudentComparator();
Comparator<Student4> reverse = comparator.reversed();
Set<Student4> students = new TreeSet<>(reverse);
...
```

11_.../swu/oopj/example7/Main.java

Combining multiple sorting criteria

- In the previous examples we saw an example of complex comparisons (by last name, then by first name, and then by id)
- What if we want to cover all combinations?
  - Three attributes + ascending and descending yields $3!*2^3 = 48$ combinations
  - E.g. 4 attributes + ascending and descending yields $4!*2^4 = 384$
  - It does make sense to write all combinations!
- Solution:
  - Write comparator for each attribute
  - Use reverse comparator
  - Create composite comparator that contains list of desired comparators
  - With this, we enable a user to create any possible combinations

Example #8 – Composite comparator (1)

- Uses a list of existing comparators for comparison

```
public class CompositeComparator<T> implements Comparator<T> {
   private List<Comparator<T>> comparators;
   public CompositeComparator(List<Comparator<T>> comparators) {
       this.comparators = new ArrayList<>(comparators.size());
       this.comparators.addAll(comparators);
   }
   @Override
   public int compare(T o1, T o2) {
       for (Comparator<T> c : comparators) {
              int r = c.compare(o1, o2);
              if (r != 0)
                     return r;
       }
       return 0;
   }
...
```

11_.../swu/oopj/example8/CompositeComparator.java

Example #8 – Composite comparator (2)

- Instead of list it can have variable number of comparators
  - Eases usage
- Note: Instead of *Comparator<T>, Comparator<? super T>* may be used
  - e.g. if we would like to compare two *Student4* objects, a Comparator<Student> also it could be used

```
public class CompositeComparator<T> implements Comparator<T> {
   private List<Comparator<T>> comparators;
   @SafeVarargs
   public CompositeComparator(Comparator<T>... comparators) {
       this.comparators = new ArrayList<>(comparators.length);
       Collections.addAll(this.comparators, comparators);
   }
...
```

11_.../swu/oopj/example8/CompositeComparator.java

Example #8 – comparator for each attribute

- Besides natural order defined in Student5, let's define comparator for each attribute
  - Could be part of the class or defined somewhere else

11_.../swu/oopj/example8/Student8.java

```
public class Student8 extends Student5 {
   ...
   public static final Comparator<Student8> BY_LAST_NAME =
              (s1,s2) -> s1.lastName.compareTo(s2.lastName);
   public static final Comparator<Student8> BY_FIRST_NAME =
               Comparator.comparing(Student8::getFirstName);
   public static final Comparator<Student8> BY_STUDENT_ID =
              (s1,s2) -> s1.studentID.compareTo(s2.studentID);

}
```

Example #8 – using composite comparator

- We can create new comparator by combining any combination of existing comparators
  - To get natural order use *Comparator.<T>naturalOrder()*

11_.../swu/oopj/example8/Main.java

```
Comparator<Student8> comparator = new CompositeComparator<>(
        Student8.BY_FIRST_NAME.reversed(),
        Student8.BY_LAST_NAME,
        Comparator.naturalOrder()
                        //same as Comparator.<Student>naturalOrder()
);
Set<Student8> students = new TreeSet<>(comparator);
...
```

Example #8 – built in composite comparator

- The same comparator as in the previous example can be created with built-in composite comparator using *thenComparing*
  - default method in interface *Comparator*

11_.../swu/oopj/example8/Main.java

```
Comparator<Student> comparator =
                        Student.BY_FIRST_NAME
                       .reversed()
                       .thenComparing(Student.BY_LAST_NAME)
                       .thenComparing(Comparator.naturalOrder());
Set<Student8> students = new TreeSet<>(comparator);
...
```