Università degli Studi di Verona
Dipartimento di Informatica
A.A. 2018-2019

APPUNTI DI “PROGRAMMAZIONE 2”

Creato da *Davide Zampieri*

```
// NUOVI CAMPI SEMPRE “PRIVATE FINAL”
private final Field_type field_name;

// COSTRUTTORE STANDARD
public Class_name(Field_type1 field_name1, Field_type2 field_name2, ...)
{
      this.field_name1 = field_name1;
      this.field_name2 = field_name2;
      ...
}

// COSTRUTTORE PRIVATO (PER AVERE UN’UNICO OGGETTO ESISTENTE)
final static Class_type INSTANCE = new Class_type();
private Class_name () {}

// METODO EQUALS() STANDARD
@Override
public boolean equals(Object other) {
      if(other instanceof Class_type) {
            Class_type otherAsClass_type = (Class_type) other;
            return field_name1 == otherAsClass_type.field_name1 &&
                  field_name2.equals(otherAsClass_type.field_name1) &&
                  ...;
      } else {
            return false;
      }
}

// METODO HASHCODE() STANDARD:
//    - LE CLASSI HANNO IL PROPRIO HASHCODE
//    - PER I TIPI PRIMITIVI (DOUBLE, CHAR, ...) BASTA FARE IL CAST
@Override
public int hashCode() {
      return field_name1 ^ field_name2.hashCode() ^ ...;
}

// METODO COMPARETO() STANDARD:
//    - <0 → THIS < OTHER, =0 → THIS = OTHER, >0 → THIS > OTHER
//    - LE CLASSI HANNO IL PROPRIO COMPARETO
//    - ALLA FINE SI RITORNA IL RISULTATO DI UNA DIFF O DI UN COMPARETO
public int compareTo(Class_type other) {
      int diff = field_name1 - other.field_name1;
            if(diff != 0)
                  return diff;

      diff = field_name2 - other.field_name2;
            if(diff != 0)
                  return diff;

      ...

      return field_name3.compareTo(other.field_name3);
}
```

```
// ECCEZIONI CONTROLLATE:
//    - I METODI CHE LE LANCIANO DEVONO DICHIARARLE CON “THROWS”
//    - POSSONO AVERE UN COSTRUTTORE CHE MANDA UN MESSAGGIO A SUPER
public class MyException extends Exception {
      public MyException(String message) {
            super(message);
      }
}

// ECCEZIONE NON CONTROLLATA STANDARD (PUO’ ESSERE VUOTA)
public class MyException extends RuntimeException {
      public MyException(String message) {
            super(message);
      }
}

// CATTURA DELLE ECCEZIONI (PER EVITARE LA TERMINAZIONE DEL PROGRAMMA)
try {
      ...codice che potrebbe lanciare un’eccezione...
} catch(Exception E) {
      ...stampa di un messaggio...
} finally {
      ...codice che viene eseguito sempre...
}

// LETTURA E SCRITTURA FILE DI TESTO (TRY WITH RESOURCES)
try (Reader reader = new BufferedReader(new FileReader(src));
      Writer writer = new BufferedWriter(new FileWriter(dst))) {
      int c = reader.read();
      while (c != -1) {
            writer.write(c);
            c = reader.read();
      }
} // chiude automaticamente reader e writer
catch (FileNotFoundException e) {
      System.out.println("Il file " + src + " non esiste");
}
catch (IOException e) {
      System.out.println("Problema di I/O");
}

// INTERFACCE: FORNISCONO LE FIRME DI UNA SERIE DI METODI
public interface MyInterface {
      Method_type1 method_name1(parameters);
      Method_type2 method_name2(parameters);
      ...
}

// QUALE COLLEZIONE USARE:
```

- collections ending with *-Set* are children of java.util.AbstractSet (which implements Iterable, Collection and Set)
- collections ending with *-Map* are children of java.util.AbstractMap (which implements Map)
- collections ending with *-List* are children of java.util.AbstractList (which implements Iterable, Collection and List)
- se devo memorizzare oggetti in base a qualche criterio di ordinamento utilizzo un *SortedSet* (dovrò implementare il compareTo() oppure passare un comparator come argomento alla creazione)
- se non è importante l'ordine, uso un *HashSet*

- se invece ho bisogno che gli oggetti mantengano l'ordine di inserimento uso una *List* (di solito *ArrayList*)
- se ho bisogno di memorizzare degli oggetti indicizzati da altri oggetti uso una *Map* (*TreeMap* se mi serve l'ordine di inserimento, *HashMap* altrimenti)

```
// METODI COMUNI A TUTTE LE COLLEZIONI:
collection.add(object);
collection.addAll(anotherCollection);
collection.clear();
collection.contains(object);
collection.containsAll(anotherCollection);
collection.equals(anotherCollection);
collection.hashCode();
collection.isEmpty();
collection.iterator();
collection.remove(object);
collection.removeAll(anotherCollection);
collection.retainAll(anotherCollection); // contrario di removeAll
collection.size();
collection.toArray();
Collection_type<E> copy = new Collection_type<>(anotherCollection);

// METODI DELLE MAPPE:
map.put(key, value);
map.putAll(anotherMap);
map.putIfAbsent(key, value);
map.get(key);
map.remove(key);
map.containsKey(key);
map.containsValue(value);
map.keySet(); // ritorna la collezione delle chiavi
map.values(); // ritorna la collezione dei valori

// METODI DELLE LISTE:
list.get(index);
list.add(index, object);
list.indexOf(object);
list.lastIndexOf(object);
list.remove(index);

// ITERATORI:
//    - SE UNA CLASSE IMPLEMENTA ITERABLE<*> DEVO DEFINIRE AL SUO INTERNO
//      IL METODO ITERATOR()
//    - SE * È GIA' ITERABILE (AD ESEMPIO PERCHÈ È UNA COLLEZIONE)
public Iterator<*> iterator() {
      return campo_di_tipo_*.iterator();
}
//    - SE * NON È ITERABILE DEVO DEFINIRE UNA CLASSE INTERNA PRIVATA
private class MyIterator implements Iterator<T> {
      private final ...;
      private MyIterator() {...}

      @Override
      public boolean hasNext() {...}

      @Override
      public T next() {...}
}
```

```
// METODO TOSTRING() CON STRING.FORMAT
@Override
public String toString() {
	String result = "";
	float percentuale = voti.get(p)*100/numeroVotiEspressi();
	result += String.format("%.02f", percentuale);
	return result;
}

// ALTRO:
// Main method
public static void main(String[] args) {
	...
}
// Read int/String
import java.util.Scanner;
Scanner scanner = new Scanner(System.in);
int n = scanner.nextInt();
String s = scanner.next(); // just one word
String s = scanner.nextLine(); // entire line
// Generate random
import java.util.Random
Random random = new Random();
int n = random.nextInt(4); // from 0 to 3
int n = random.nextInt((max - min) + 1) + min; // between min and max
// For each
for (Item item : items) {
	...
}
// Convert numbers
public String intTo(AnyBase)(int n) {
	final String values = "0123456789ABCDEF";
	String res = "";
	do {
		res = values.charAt(n % base) + res;
		n /= base;
	} while(n > 0);
	return res;
}
// Visibility
```

Access modifier	Within class	Within package	Outside package by subclass only	Outside package
private	Y	N	N	N
“package”	Y	Y	N	N
protected	Y	Y	Y	N
public	Y	Y	Y	Y

```
// NOTE: if private fields are immutable (e.g. value do not change
after initialization), add final keyword to declaration
```

```java
// ESEMPI DAGLI ESAMI:
//    - COLLEZIONI
public class Shop {
    //USO UNA MAPPA PER SAPERE I PRODOTTI DISPONIBILI
    private final HashMap<Product, Integer> products = new HashMap<>();

    public void add(Product product, int howMany) {
        //aggiunge howMany volte il prodotto indicato, che poteva gia' essere presente in negozio
        products.putIfAbsent(product, 0);
        products.put(product, products.get(product) + howMany);
    }

    void buy(Product[] productsToBuy) throws MissingProductException {
        //rimuove i prodotti indicati da quelli disponibili in questo negozio
        //se non fossero disponibili tutti i prodotti, lancia una MissingProductException
        //in tal caso, il negozio dovra' restare immutato e nessun prodotto dovra' essere tolto
        HashMap<Product, Integer> copy = new HashMap<>(products);
        for(Product p : productsToBuy) {
            copy.put(p, copy.get(p)-1);
            if(copy.get(p) < 0)
                throw new MissingProductException();
        }

        for(Product p : productsToBuy)
            products.put(p, products.get(p)-1);
    }
}

public class SplitOrder extends Order {
    public SplitOrder(Shop shop, Product... products) {
        super(shop, products);
    }

    @Override
    public Iterable<Shipping> ship() throws MissingProductException {
        //compra i prodotti e ritorna una o piu' spedizioni
        buy();

        //CREO TANTE SPEDIZIONI IN BASE AI GIORNI DI ATTESA DEI PRODOTTI
        HashMap<Integer, List<Product>> shippings = new HashMap<>();
        for(Product p : getProducts()) {
            shippings.putIfAbsent(p.getDaysBeforeShipping(), new ArrayList<>());
            shippings.get(p.getDaysBeforeShipping()).add(p);
        }

        //CREO UNA LISTA (ITERABILE) CHE CONTIENE TANTE SPEDIZIONI
        List<Shipping> result = new ArrayList<>();
        for(List<Product> shipping : shippings.values()) {      //ITERO SULLA COLLEZIONE DEI VALORI
            Shipping s = new Shipping(shipping);
            result.add(s);
        }

        return result;
    }
}
```

```
//    - COMPARATORI
private class MyComparator implements Comparator<T> {
      @Override
      public int compare(T o1, T o2) {...}
}
```

```
// queryByPrice devono essere le stanze per almeno una persona, che costano al massimo 70 al giorno,
// ordinate per prezzo crescente (cioe' prima quelle che costano meno)
SortedSet<Room> queryByPrice = bnb.sortedAvailableFor(1, 70, new Comparator<Room>() {
    // restituisce:
    // un numero negativo se o1 viene prima di o2;
    // un numero positivo se o2 viene prima di o1;
    // 0 se o1 e o2 sono uguali
    @Override
    public int compare(Room o1, Room o2) {
        return o1.getPriceForDay() - o2.getPriceForDay();
    }
});

// come queryByPrice, ma ordinate per valutazione decrescente (cioe' prima quelle con valutazione maggiore)
SortedSet<Room> queryByStars = bnb.sortedAvailableFor(1, 70, new Comparator<Room>() {
    // restituisce:
    // un numero negativo se o1 viene prima di o2;
    // un numero positivo se o2 viene prima di o1;
    // 0 se o1 e o2 sono uguali
    @Override
    public int compare(Room o1, Room o2) {
        if(o1.averageStars() < o2.averageStars())
            return 1;
        else if(o1.averageStars() > o2.averageStars())
            return -1;
        else
            return 0;
    }
});
```

```
public class EsamePerEsito extends Esame {
    @Override
    protected Comparator<Verbalizzazione> getComparator() {
        return new Comparator<Verbalizzazione>() {
            @Override
            public int compare(Verbalizzazione v1, Verbalizzazione v2) {
                int diff = v1.getEsito().compareTo(v2.getEsito());
                if (diff != 0)
                    return diff;
                else
                    return v1.getStudente().matricola - v2.getStudente().matricola;
            }
        };
    }
}
```

```
public class EsamePerMatricola extends Esame {
    @Override
    protected Comparator<Verbalizzazione> getComparator() {
        return new Comparator<Verbalizzazione>() {
            @Override
            public int compare(Verbalizzazione v1, Verbalizzazione v2) {
                return v1.getStudente().matricola - v2.getStudente().matricola;
            }
        };
    }
}
```

```
//    - ITERATORI
class Items implements Iterable<Item> {
    private Item[] items;

    @Override
    public Iterator<Item> iterator() {
        return new Iterator<Item> {
            private int pos;

            public boolean hasNext() {
                return pos < items.length; // items is a private
                array
            }

            public Item next() {
                return items[pos];
            }
        }
    }
}

public class Elezioni implements Iterable<VotiPerPartito> {


       @Override
       public final Iterator<VotiPerPartito> iterator() {
              return new ElezioniIterator();
       }

       private class ElezioniIterator implements Iterator<VotiPerPartito> {
              private int numeroPartito;

              @Override
              public boolean hasNext() {
                     return numeroPartito < Partito.NUMERO_PARTITI;
              }

              @Override
              public VotiPerPartito next() {
                     Partito prossimoPartito = Partito.elementi()[numeroPartito++];
                     return new VotiPerPartito(prossimoPartito, votiPer(prossimoPartito));

              }
       }
}
```

APPUNTI ITERATOR e SUB-CLASSI

SexView

```
import PhoneBook.Entry;

import java.util.Iterator;

public class SexView extends View {
	private final View parent;
	private final boolean sex;

	public SexView(View parent, boolean sex) {
		this.parent = parent;
		this.sex = sex;
	}

	@Override
	public Iterator<Entry> iterator() {
		return new SexViewIterator();
	}

	private class SexViewIterator implements Iterator<Entry> {
		private final Iterator<Entry> iteratorOfParent;
		private Entry next;

		private SexViewIterator() {
			iteratorOfParent = parent.iterator();
			lookForSex();
		}

		private void lookForSex() {
			next = null;

			while (iteratorOfParent.hasNext()) {
				Entry cursor = iteratorOfParent.next();
				if (cursor.sex == sex) {
					next = cursor;
					return;
				}
			}
		}

		@Override
		public boolean hasNext() {
			return next != null;
		}

		@Override
		public Entry next() {
			Entry result = next;
			lookForSex();

			return result;
		}

		@Override
		public void remove() {
		}
	}
}
```

SortedView

```
import it.univr.phones.PhoneBook.Entry;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.Comparator;
import java.util.Iterator;
import java.util.List;

public class SortedView extends View {
	private final View parent;
	private final Comparator<Entry> comparator;

	public SortedView(View parent, Comparator<Entry> comparator) {
		this.parent = parent;
		this.comparator = comparator;
	}

	@Override
	public Iterator<Entry> iterator() {
		return new SortedViewIterator();
	}

	private class SortedViewIterator implements Iterator<Entry> {
		private final Entry[] sortedEntries;
		private int pos;

		private SortedViewIterator() {
			List<Entry> entries = new ArrayList<>();
			for (Entry entry: parent)
				entries.add(entry);

			sortedEntries = entries.toArray(new Entry[entries.size()]);
			Arrays.sort(sortedEntries, comparator);
		}

		@Override
		public boolean hasNext() {
			return pos < sortedEntries.length;
		}

		@Override
		public Entry next() {
			return sortedEntries[pos++];
		}

		@Override
		public void remove() {
		}
	}
}
```

view

```
import PhoneBook.Entry;

public abstract class View implements Iterable<Entry> {

	protected View() {
	}

	@Override
	public final String toString() {
		String result = "";
		for (Entry entry: this)
			result += entry.toString() + "\n";

		return result;
	}
}
```

PhoneBook

```
import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;

public class PhoneBook extends View {
   private final List<Entry> entries = new ArrayList<>();

   public static class Entry {
     public final String name;
     public final String surname;
     public final int phone;
     public final boolean sex;
     public final static boolean MALE = false;
     public final static boolean FEMALE = true;

     private Entry(String name, String surname, int phone, boolean sex) {
            this.name = name;
            this.surname = surname;
            this.phone = phone;
            this.sex = sex;
     }

     @Override
     public String toString() {
            return name + " " + surname + ": " + phone + (sex == MALE ? " [male]" : " [female]");
     }
   }

   public PhoneBook() {
   }

   public void add(String name, String surname, int phone, boolean sex) {
     for (Entry entry: entries)
            if (entry.name.equals(name) && entry.surname.equals(surname)) {
                   entries.remove(entry);
                   break;
            }

     entries.add(new Entry(name, surname, phone, sex));
   }

   public void remove(String name, String surname) {
     for (Entry entry: entries)
            if (entry.name.equals(name) && entry.surname.equals(surname)) {
                   entries.remove(entry);
                   return;
            }

     throw new UnknownEntryException();
   }

   @Override
   public Iterator<Entry> iterator() {
     return entries.iterator();
   }
}
```