Collections

Sorting Data

Sorting data in a collection

- we are already partly familiar with `sort()` method
- if elements in the collection are primitives, they are sorted by natural order
- if elements are Strings, then numbers sort before letters, and uppercase letters sort before lowercase letters
- but elements in the collection can be any type of object
- in that case, you have to define your own criteria of sorting
- in order to do this you can choose one of two approaches
    1. use a class which implements `Comparable<T>` interface, or
    2. pass the implementation of `Comparator<T>` interface in `sort()` method

Comparable<T> interface

- this interface has one abstract method: `int compareTo(T o)`
  - this methods has to be implemented in a concrete class
- this method returns an integer according to these rules:
  1. if the current object is equivalent to the argument it returns 0
  2. if the current object is smaller than the argument it returns a negative number
  3. if the current object is larger than the argument it returns a positive number
- let's look at some examples...

```
public class Person implements Comparable<Person> {
  private String name;
  private int age;
  public Person(String name, int age) {
    this.name = name;
    this.age = age;
  }
  @Override
  public int compareTo(Person other) {
    return this.age - other.age;
  }
  // toString() implementation
}
```

0 if ages are equal

<0 if age is smaller than age in the argument

>0 if age is greater than age in the argument

```
public class Main {
  public static void main(String[] args) {
    List<Person> people = Arrays.asList(
      new Person("John", 25),
      new Person("George", 20),
      new Person("Ben", 30)
    );
    Collections.sort(people);
    System.out.println(people);
  }
}
```

sorted by age

```
[Person{name='George', age=20}, Person{name='John', age=25}, Person{name='Ben', age=30}]
```

```
public class Person implements Comparable<Person> {
  private String name;
  private int age;
  public Person(String name, int age) {
    this.name = name;
    this.age = age;
  }
  @Override
  public int compareTo(Person other) {
    return this.name.compareTo(other.name);
  }
  // toString() implementation
}
```

`String` class has the implementation of `compareTo()` method, so you can just use it here

```
public class Main {
  public static void main(String[] args) {
    List<Person> people = Arrays.asList(
      new Person("John", 25),
      new Person("George", 20),
      new Person("Ben", 30)
    );
    Collections.sort(people);
    System.out.println(people);
  }
}
```

sorted by name

```
[Person{name='Ben', age=30}, Person{name='George', age=20}, Person{name='John', age=25}]
```

Comparator<T> interface

- in the last example we had to define a criterium for sorting when designing a class Person (either by name or age)
- but what if we don't want to make that commitment?
  - i.e. what if we want to sort by name in one case, and by age in another?
- in that case we can use `Comparator<T>` interface
  - and provide the implementation for `compare(T o1, T o2)` method
- this implementation is than passed to `sort()` method
  - to do this we usually use lambda expression or method reference

```
public class Person {     no "implements Comparable" here
  private String name;
  private int age;
  public Person(String name, int age) {
    this.name = name;
    this.age = age;
  }
  public String getName() { return name; }
  public int getAge() { return age; }
  // toString() implementation
}
```

```
public class Main {
  public static void main(String[] args) {
    List<Person> people = Arrays.asList(
      new Person("John", 25),
      new Person("George", 20),
      new Person("Ben", 30)
    );
    Collections.sort(people, (p1, p2) -> p1.getAge() - p2.getAge());
    System.out.println(people);
  }
}
```

implementation of `compare()` method to sort by age

```
[Person{name='George', age=20}, Person{name='John', age=25}, Person{name='Ben', age=30}]
```

```
public class Main {
  public static void main(String[] args) {
    List<Person> people = Arrays.asList(
      new Person("John", 25),
      new Person("George", 20),
      new Person("Ben", 30)
    );
    Collections.sort(people, (p1, p2) -> p1.getName().compareTo(p2.getName()));
    System.out.println(people);
  }
}
```

implementation of `compare()` method to sort by name using `compareTo()` as it is implemented in `String` class

```
[Person{name='Ben', age=30}, Person{name='George', age=20}, Person{name='John', age=25}]
```

```
// same thing without lambda (the old way)
public class Main {
  public static void main(String[] args) {
    List<Person> people = Arrays.asList(
      new Person("John", 25), new Person("George", 20), new Person("Ben", 30));
    Comparator<Person> byAge = new Comparator<Person>() {
      public int compare (Person p1, Person p2) {
        return p1.getAge() - p2.getAge();
      }
    };
    Collections.sort(people, byAge);
    System.out.println(people);
  }
}
```

implementation of `compare()` method using anonymous class

passing the implementation in the `sort()` method

```
// using comparing() method with method reference


// to sort by name
Comparator<Person> c = Comparator.comparing(Person::getName);


// to sort by name in reversed order
Comparator<Person> c = Comparator.comparing(Person::getName).reversed();


// to sort by name and then by age (if names are the same)
Comparator<Person> c =
  Comparator.comparing(Person::getName).thenComparingInt(Person::getAge);
```

Comparable vs. Comparator Summary

	Comparable	**Comparator**
package name (for import)	java.lang	java.util
must me implemented by a class	Yes	No
method name in interface	`compareTo()`	`compare()`
number of method parameters	1	2
usually used with lambda	No	Yes