Problem Set 3 Solution

Interfaces

---

1. Short questions

    1. Will the following code compile?

    ```
    public class D { }

    public class C implements Comparable<D> {
       public int compareTo(D o) { return 0; }
    }
    ```

    ANSWER:

    Yes.

    2. Will the following code compile?

    ```
    public class D { }

    public class C implements Comparable<C>, Comparable<D> {
       public int compareTo(C o) { return 0; }
       public int compareTo(D o) { return 0; }
    }
    ```

    ANSWER:

    No, you can't implement the same interface with different generic type arguments.

    3. Will the following code compile?

    ```
    public class A implements Comparable<A> {
       public int compareTo(A o) { return 0; }
    }

    public class B extends A implements Comparable<B> {
       public int compareTo(B b) { return 0; }
    }
    ```

    ANSWER:

    No, same as previous because when B extends A, it implicitly implements any interface(s) that A implements, namely Comparable<A>. So B ends up implementing Comparable twice, with A and B as generic type arguments, which is not allowed.

    4. Will the following code compile?

    ```
    public interface I { void stuff(); }
    public interface J { void stuff(); }
    public class F implements I,J { }
    ```

    ANSWER:

    No. F must implement methods of all the interfaces it implements.

    5. Will the following code compile?

    ```
    public interface I { void stuff(); }
    public interface J { void stuff(); }
    public class F implements I,J { public void stuff() { } }
    ```

    ANSWER:

    Yes.

6. Will the following code compile?

```
public interface I { int stuff(); }
public interface J { void stuff(); }
public class F implements I,J { public int stuff() { return 3;}}
```

ANSWER:

No. Since the interface methods have the same name but different return types, there is no way for F to implement both without a naming conflict.

7. Will the following code compile?

```
class X implements Comparable<X> {
    public int compareTo(X o) {
        return 0;
    }
}
public class Searcher {

    public static <T extends Comparable<T>>
    boolean binarySearch(T[] list, T item) {
        return false;
    }

    public static void main(String[] args) {
        X[] xs = new X[2];
        xs[0] = new X();
        xs[1] = new X();
        binarySearch(xs,new X());
    }
}
```

ANSWER: Yes.

8. Will the following code compile?

```
class X implements Comparable<X> {
    public int compareTo(X o) {
        return 0;
    }
}
class Y extends X { }
public class Searcher {

    public static <T extends Comparable<T>>
    boolean binarySearch(T[] list, T item) {
        return false;
    }

    public static void main(String[] args) {
        Y[] ys = new Y[2];
        ys[0] = new Y();
        ys[1] = new Y();
        binarySearch(ys,new Y());
    }
}
```

ANSWER: Yes.

9. Will the following code compile?

```
class X implements Comparable<X> {
    public int compareTo(X o) {
        return 0;
    }
```

```
}
class Z implements Comparable<X> { }
public class Searcher {

    public static <T extends Comparable<T>>
    boolean binarySearch(T[] list, T item) {
        return false;
    }

    public static void main(String[] args) {
        Z[] zs = new Z[2];
        zs[0] = new Z();
        zs[1] = new Z();
        binarySearch(zs,new Z());
    }
}
```

ANSWER: No. The type T that is required by the search must either itself implement Comparable<T> or must have an ancestor in the inheritance hierarchy that implements Comparable<T>. Z implements Comparable<X>, but Z is not X, and neither is it a subclass of X.

10. Will the following code compile?

```
class X implements Comparable<X> {
    public int compareTo(X o) {
        return 0;
    }
}
class Y extends X { }
class Z extends Y { }

public class Searcher {

    public static <T extends Comparable<T>>
    boolean binarySearch(T[] list, T item) {
        return false;
    }

    public static void main(String[] args) {
        Z[] zs = new Z[2];
        zs[0] = new Z();
        zs[1] = new Z();
        binarySearch(zs,new Z());
    }
}
```

ANSWER:

Yes.

---

2. Suppose you built a Java library of sorting algorithms: insertion sort, quicksort, and heapsort. You want to sell this library. How would you package your library *using interfaces*, so users could use any of these algorithms in their applications, and switch from using one to another (*interface polymorphism*), with the least amount of code rewrite?

SOLUTION

Write an interface called `SortingAlgorithm` with one or more methods called `sort`, and then write various sorting classes for the different sorting algorithms, that implement the `SortingAlgorithm` interface.

```
// interface definition
public interface SortingAlgorithm<T extends Comparable<T>> {
    void sort(T[] list);
}

/** sample implementations **/
```

```
public class InsertionSort<T extends Comparable<T>>
implements SortingAlgorithm<T> {   // note the syntax
    public void sort(T[] list) {
         ...  // implement insertion sort
    }
}

public class Quicksort<T extends Comparable<T>>
implements SortingAlgorithm<T> {
    public void sort(T[] list) {
         ...  // implement quicksort
    }
}
/** end sample implementations **/


/** application call examples, interface polymorphism **/
...
// use insertion sort
SortingAlgorithm<String> sort_algo = new InsertionSort<String>();
sort_algo.sort(new String[]{"one","two","three"}); // anonymous array creation and initialization
...
// morph to using quicksort, only sorting object creation needs to change
sort_algo = new Quicksort<String>();   // plug-n-play
sort_algo.sort(new String[]{"one","two","three"});  // no change in this line

/** end application call examples **/
```

---

3. Aside from the `java.lang.Comparable<T>` interface used for comparing objects of a class, the `java.util` package has an interface, `Comparator<T>` that may also be used to compare objects. What is the difference between these two interfaces, and how may this difference be usefully employed in applications?

   SOLUTION

   Here's an example of a simplified user-defined type:

```
public class Customer {
  private String firstName;
  private String middleName;
  private String lastName;

  // setter & getter methods..
}
```

   In the above class, the most commonly *natural ordering* would be by `lastName`. This can easily be implemented using the `Comparable<T>` interface, i.e.

```
public class Customer implements Comparable<Customer> {

  public int compareTo (Customer c) {

    if (c == null) { return -1; // assuming you want null values shown last}

    String cLastName = c.getLastName();
    if (lastName == null && cLastName == null) { return 0; }
    // assuming you want null values shown last
    if (lastName != null && cLastName == null) { return -1; }
    if (lastName == null && cLastName != null) { return 1; }
    return lastName.compareTo (cLastName);
  }
}
```

Sorting a `List` of `Customer` objects would be as simple as:

```
  Collections.sort (customerList);
```

But, if we want to use a different ordering, e.g. order by the first name, then we cannot use the natural ordering as defined within the `Customer` class. Instead, we have to define an alternative ordering, in the form of a `Comparator` class.

```
public class CustomerFirstNameComparator
implements Comparator<Customer> {

  public int compare (Customer c1, Customer c2) {
    if (c1 == null && c2 == null) { return 0; }
    // assuming you want null values shown last
    if (c1 != null && c2 == null) { return -1; }
    if (c1 == null && c2 != null) { return 1; }

    String firstName1 = c1.getFirstName();
    String firstName2 = c2.getFirstName();

    if (firstName1 == null && firstName2 == null) { return 0; }
    // assuming you want null values shown last
    if (firstName1 != null && firstName2 == null) { return -1; }
    if (firstName1 == null && firstName2 != null) { return 1; }
    return firstName1.compareTo (firstName2);
  }
}
```

Sorting a `List` of `Customer` objects by their first name, would be:

```
  Comparator<Customer> comparator = new CustomerFirstNameComparator();
  Collections.sort (customerList, new CustomerFirstNameComparator());
```