Generics in Java and Haskell

<T>

Modified the description of AcceptsAny and AcceptsAll on 12/4 at 12:30pm

Java Generics

Overview

For this problem we define an interface named `Predicate`. `Predicate` is a generic interface that declares a single method named `accept`. The `accept` method takes an object of T type and returns true if the predicate accepts the object and false otherwise.

```
package predicates;

public interface Predicate<T> {
  public boolean accepts(T t);
}
```

Implementation Examples

Consider, for example, the following implementation. This predicate is not generic and operates on Integers. The accept method returns true if the integer is positive. In other words, this predicate accepts all positive integers.

```
public class IsPositive implements Predicate<Integer> {
    public boolean accepts(Integer t) {
      return t > 0;
   }
}
```

The predicate can then be used as shown in the following code fragment. While this code fragment is not particularly useful, it concisely demonstrates how to use the single method of the interface. After executing this code fragment, the value of `b1` is true while the value of `b2` is false.

```
Predicate<Integer> p = new IsPositive();
boolean b1 = p.accepts(13);
boolean b2 = p.accepts(-3);
```

Consider another example. The GreaterThan predicate accepts Comparable objects that are greater than some reference object. The behavior of this class is shown, by example, in the following code fragment. The variable p1 uses “Brazil” as a reference object and accepts only strings that are greater than Brazil. Hence, variable b1 is true, b2 is true and b3 is false after executing the code fragment. Variable p2 uses the Integer 10 as a reference object and only accepts Integers that are greater than 10. Hence, b4 is false and b5 is true after executing the code fragment.

```
Predicate<String> p1 = new GreaterThan<String>(“Brazil”);
boolean b1 = p1.accepts(“China”);
boolean b2 = p1.accepts(“USA”);
boolean b3 = p1.accepts(“Australia”);

Predicate<Integer> p2 = new GreaterThan<Integer>(10);
boolean b4 = p2.accepts(5);
boolean b5 = p2.accepts(28);
```

Problems

You must write several classes that implement the Predicate interface. These classes are described in the following list and each must be in the "predicates" package.

1. `SimilarColor` An implementation of `Predicate` that
   - Is not generic
   - Has a constructor that accepts a Color object known as the reference.
   - Accepts only Color objects that are similar to the reference object. Two colors are similar if the sum of the absolute differences in red, green, and blue is less than or equal to 30.

2. `StartsWith` An implementation of `Predicate` that
   - Is not generic
   - Has a constructor that accepts a String object known as the reference.
   - Accepts only Strings start with the references string.

3. `GreaterThan` An implementation of `Predicate` that
   - Is generic
   - Has a constructor that accepts a Comparable object known as the reference.
   - Accepts operates only on objects of the type specified in the constructor and only accepts objects that are greater than the reference.

4. `Subset` An implementation of `Predicate` that
   - Is generic
   - Has a constructor that accepts a java.util.List object known as the reference list.
   - Accepts operates only on lists of the same type as that specified in the constructor and only accepts lists where each of it's elements are contained in the reference list.

5. `Negation` An implementation of `Predicate` that
   - Is generic
   - Has a constructor that accepts a `Predicate` object known as the reference.
   - Accepts operates only on those types of objects that the reference operates on. The Negation predicate accepts any object that is not accepted by the reference.

6. `AcceptsAll` An implementation of `Predicate` that
   - Is generic
   - Has a constructor that accepts a java.util.List of objects known as the reference list.
   - Accepts operates only `Predicates` that operate on the type of objects found in the reference list. `AcceptsAll` will only accept a predicate that accepts each of the elements in the reference list.

7. `AcceptsAny` An implementation of `Predicate` that
   - Is generic
   - Has a constructor that accepts a java.util.List of objects known as the reference list.
   - Accepts operates only `Predicates` that operate on the type of objects found in the reference list. `AcceptsAny` will only accept a predicate that accepts at least one of the elements in the reference list.

8. `And` An implementation of `Predicate` that
   - Is generic
   - Has a constructor that accepts zero-or-more `Predicate` objects (each of exactly the same type) known as the reference predicates.
   - Accepts operates only on objects of the type operated on by the reference predicates. `And` accepts an element if it is accepted by all of the reference predicates.

In addition to the classes above, also complete the following.

9. `PredicateUtilities` A class that contains a public static function named `listFilter` that has the following features.
   - The method has two parameters: a `Predicate` and a `java.util.List`. The predicate must operate only on those elements that are contained in the list.
   - The method returns a list of those elements of the input that are accepted by the predicate.

Haskell List Functions

Just as in Scheme, list processing is a basic feature of Haskell. Write the following Haskell functions and for each function, you must include a signature.

1. Write a function named `seal` that accepts two lists of sorted (ascending order) values xs and ys. The function returns a list of all elements in xs and ys such that the list is in sorted order (ascending).
   a. seal [3, 6, 7] [1, 2] => [1, 2, 3, 6, 7]
   b. seal [] [3, 12] => [3, 12]
   c. seal [1, 2, 3] [4, 5, 6] => [1, 2, 3, 4, 5, 6]
   d. seal ["a", "d"] ["b"] => ["a", "b", "d"]

2. Write a function named `isSublist` that accepts two lists of Integers xs and ys. The function returns True iff xs occurs anywhere as a sublist within ys. For example,
   a. isSublist [3, 4] [1, 2, 3 4] => True
   b. isSublist [3, 4] [1, 3, 2, 4] => False
   c. isSublist [1, 2, 3] [1, 2, 3, 0, 1] => True
   d. isSublist [1, 2, 3] [1, 2, 2, 3, 1, 2] => False

3. Write a function named `combinator` that accepts two lists of elements xs and ys. The function returns a list containing all possible pairs of elements (given as a list) [x, y] where x is an element of xs and y is an element of ys. For example,
   a. combinator [3, 4] [1, 2] => [[3,1], [3, 2], [4, 1], [4, 2]]
   b. combinator [] [3, 12, 9] => []
   c. combinator [1, 2] [1, 3] => [[1, 1], [1, 3], [2, 1], [2, 3]]
   d. combinator ["a", "b"] ["c", "d"] => [["a", "c"], ["a", "d"], ["b", "c"], ["b", "d"]]

