Software and Programming II (SP2) — Lab sheet 8

Wildcards and Recursive Methods

2018/19

1. In Lab sheet 7, we implemented a generic class `Pair` to store two objects of arbitrary (and possibly different) types. Our code looked like this:

```
1  public class Pair<A, B> {
2
3      private A x;
4      private B y;
5
6      public Pair(A x, B y) {
7          this.x = x;
8          this.y = y;
9      }
10
11     public Pair(Pair<A, B> other) {
12         this(other.x, other.y);
13     }
14
15     . . .
16 }
```

   For the second constructor, we can use a more general parameter type that will make the constructor usable for more inputs. The body of the constructor does not need to be changed. What would the first line of the second constructor look like? *Hint: Wildcards.* Can you come up with an example use of the new constructor that was not possible with its old version?

2. In Lab sheet 7, we wrote an interface `Dictionary`:

```
1 import java.util.Collection;
2
3 public interface Dictionary<K, V> {
4     V get(K key);
5     V put(K key, V value);
6     boolean isEmpty();
7     Collection<K> keys();
8     Collection<V> values();
9 }
```

   Now we have learned about wildcards, and in the previous question we have seen that they can give us some additional flexibility. Would it be a sensible change to replace the line

```
Collection<K> keys();
```

   by the following line?

```
Collection<? extends K> keys();
```

3. Recursion is an elegant, almost magical way to solve problems. Problems that are difficult to solve with an iterative technique are sometimes quite simple when solved with a recursive technique. Good programmers understand how to program in both styles. The basic idea is to take a "large" problem and imagine how a solution to a "smaller" version of the problem could be used to solve the original problem.

   Consider the problem of reversing the letters in a string. We will need a couple of `String` methods. Assume `String s = "abcde"`.

   (a) `s.charAt(0)` returns the character `'a'`

   (b) `s.substring(1)` returns the string `"bcde"`

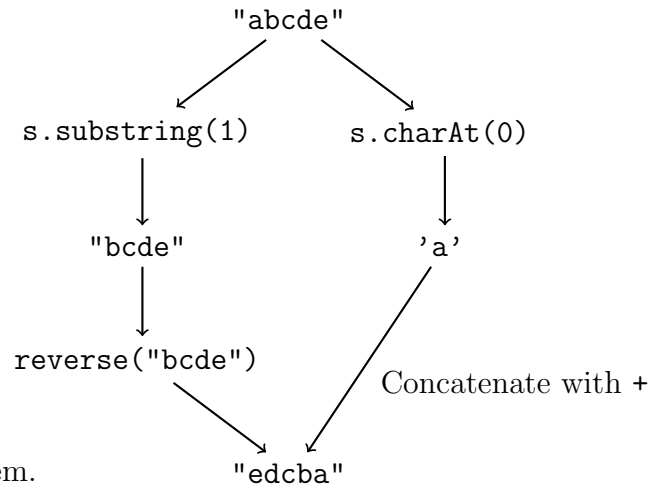
   Let's plan a method for reversing the string that uses these methods. If the problem at hand is reversing `s`, we first need to think of a smaller related problem. Can you imagine one?

   Suppose we consider the problem of reversing the string `"bcde"`?
   Could the solution to this problem help us solve the original problem?

   Original problem: Reverse `"abcde"`

   Shrink the problem: Reverse `"bcde"`

   Use this solution to solve the original problem.



   One of the steps in the above process involves calling `reverse("bcde")`. The interesting thing about recursive methods is that you can solve the smaller problems by calling the method you are writing. Keep in mind you don't have to show how to solve all the smaller problems, you just have to show how a solution to a smaller problem can be used to solve the larger problem.

   The process of "shrinking" the original problem can't continue forever, so we also have to specify the solutions to the smallest problems we encounter. In the example above, if the string `s` has a single (or no) character, we can simply return it as the solution (we can reverse a string with zero or one characters). You can test for this by examining whether `s.length() <= 1`. In a recursive solution, you have to test for the smallest cases first.

   Write a recursive method called `reverse` that is passed a `String` and that returns a `String` with the characters of the original string reversed. Use the following `main` method for testing your method.

```
public class Reverse {

    // Put your code here

    public static void main(String[] args) {
```

```
        String word = "abcdefg";
        System.out.println("Word:  " + word );
        System.out.println("Word reversed: " + reverse(word));
    }
}
```

4. A very good sorting algorithm that uses recursion is named *merge sort*. It works as follows:

    1. To start, consider each element of the `ArrayList` as a "section" of size 1 (numbered 0 through $n-1$).
    2. Merge neighbouring *sections* together so that the resulting section (of double the size of the original section) is sorted.
    3. Repeat the previous step until there is only one section left.

    The above looks well and good in concept but there are a couple of snags. The first is how to handle weirdly-sized sections (*leftovers*). Another is how to store the sections while we are working with them.).

    Here are some example runs of merge sort:

```
Input: arr =        { 1 8 4 13 99 23 17 7 25 }
Initial sections =  {1} {8} {4} {13} {99} {23} {17} {7} {25}
Merge #1 =          {1 8} {4 13} {23 99} {7 17} {25}
Merge #2 =          {1 4 8 13} {7 17 23 99} {25}
Merge #3 =          {1 4 7 8 13 17 23 99} {25}
Cleanup =           {1 4 7 8 13 17 23 25 99}

Input: arr =        { 13 99 47 0 23 13 86 }
Merge #1 =          { 13 99 } { 0 47 } { 13 23 } { 86 }
Merge #2 =          { 0 13 47 99 } {13 23 86 } //Note odd sized section
Merge #3 =          { 0 13 13 23 47 86 99 }
```

    You are required to:

    (a) Perform a *merge sort* on the following `ArrayList` (showing all steps as above):

    ```
    { 13 47 200 53 0 100 33 8 31 75 123 47 99 }
    ```

    (b) Write a recursive method, `mergeSort(ArrayList<Integer> arr)` that sorts the input `ArrayList` using merge sort.
    Hint: You'll need a *helper function*, `merge`, that merges two sections together.