Chapter 7

Recursion

7.1 Introduction

Let us implement the method

```java
/**
 * Prints the given number of *s.
 *
 * @param number of *s to be printed.
 * @pre. n >= 0
 */
public static void stars(int n)
```

of the class `Print`. An obvious way to implement this method is the following.

```java
public static void stars(int n)
{
   for (int i = 0; i < n; i++)
   {
      System.out.print('*');
   }
}
```

However, there are other ways to implement this method. Obviously, if `n` is zero, then we do not have to print anything at all. Otherwise, we have to print at least one `*`. This leads us to the following skeleton.

```java
public static void stars(int n)
{
   if (n == 0)
   {
      // do nothing
   }
7     else
8     {
9        System.out.print('*');
10
11    }
12 }
```

What remains to be done at line 10 is printing $n - 1$ `*`s. In order to accomplish that, we can delegate to the `stars` method:

```
10 Print.stars(n - 1);
```

As usual, when we invoke a method, such as `stars`, we have to make sure that its precondition is satisfied. If we reach line 10, then we have that $n > 0$ and, hence, $n - 1 \geq 0$. So the precondition of `stars` is indeed satisfied at line 10.

Note that the `stars` method is invoked within the body of the `stars` method. This makes it a so-called *recursive* method. The invocations of the method within itself are known as *recursive invocations*. For example, the invocation `Print.stars(n - 1)` on line 10 is a recursive invocation.

Within the body of a recursive method we find one or more *base cases* and one or more *recursive cases*. In a base case, there is no need to delegate to the method itself, whereas a recursive case contains one or more recursive invocations. The above recursive method `stars` has a single base case, namely when `n` is zero. In that case, line 5 is executed. The method also has a single recursive case, namely when `n` is greater than zero. In that case, line 9 and 10 are executed.

The method `starsAndPlusses` prints a number of `*`s —this number is given as an argument— followed by the same number of `+`s.

```
/**
 * Prints the given number of *s followed by the given number of +s.
 *
 * @param the number of *s and +s to be printed.
 * @pre. n >= 0
*/
public static void starsAndPlusses(int n)
```

When implementing a method recursively, we first determine base cases, i.e. those cases in which we do not need to delegate to the method itself. If `n` is zero, we do not have to do anything. This is an example of a base case. Now assume that `n` is greater than zero. In that case we have to print the following.

$$\overbrace{* \underbrace{* \cdots *}_{\mathtt{n}-1}}^{\mathtt{n}} \overbrace{\underbrace{+ \cdots +}_{\mathtt{n}-1} +}^{\mathtt{n}}$$

Note that the substring starting from the second `*` and ending with the one but last `+` can be printed by delegating to `Print.starsAndPlusses(n - 1)`. Hence, to print `n` `*`s followed by `n` `+`s we can

1. print a `*`,
2. print $n - 1$ `*`s followed by $n - 1$ `+`s by delegating to `Print.starsAndPlusses(n - 1)`, and
3. print a `+`.

Combining the above leads us to the following.

```
1  public static void starsAndPlusses(int n)
2  {
3     if (n == 0)
4     {
5        // do nothing
6     }
7     else
8     {
9        System.out.print('*');
10       Print.starsAndPlusses(n - 1);
11       System.out.print('+');
12    }
13 }
```

As in the previous example, we have to make sure that the precondition of the method `starsAndPlusses` is satisfied when we invoke it in line 10. If we reach line 10, then we have that $n > 0$ and, hence, $n - 1 \geq 0$. So the precondition is satisfied. The method has a single base case. In that case line 5 is executed. The method also has a single recursive case. In that case line 9–11 are executed.

To simplify matters a little, all methods in this chapter are static. Note, however, that recursion can also be used to implement non-static methods.

In the remainder of this chapter, we will provide numerous examples of recursive methods. We will start with simple examples, such as the `stars` method presented above, to illustrate recursion. To implement these methods, using iteration, i.e. using for-, while- and do-loops, may be simpler than using recursion. However, when we come to more intricate examples, the strength of recursion will become apparent.[1]

7.2 Proving Correctness and Termination

First, we present a strategy to prove recursive methods correct and a technique to prove that recursive methods terminate. Subsequently, we apply these to prove the correctness and termination of the recursive methods introduced in the previous section.

The correctness proof of a recursive method is split into two parts.

[1]If a method can be implemented using recursion, then it can also be implemented using iteration. Conversely, if a method can be implemented using iteration, then it can also be implemented using recursion. So, what should we use, recursion or iteration? Generally, we choose the technique that gives rise to the simplest code. We will violate this guideline at the beginning of this chapter, since we want to illustrate recursion by means of simple examples and we are not aware of any simple examples for which recursion gives rise to simpler code than iteration.

- For each base case, we prove that it is correct.
- For each recursive case, we assume that the recursive invocations are correct and we prove under that assumption that the recursive case is correct as well.

The termination proof of a recursive method consists of two steps.

1. We define the size of each invocation of the method. The size has to be a natural number (i.e. a non-negative integer).
2. We prove that each recursive invocation has a smaller size than the original invocation.

Our strategy to prove correctness can only be used in combination with a proof of termination using the above proof technique.[2]

The method `stars` of the class `Print` prints a number of `*`s.

```
1  /**
2   * Prints the given number of *s.
3   *
4   *  @param the number of *s to be printed.
5   *  @pre. n >= 0
6   */
7  public static void stars(int n)
8  {
9     if (n == 0)
10    {
11       // do nothing
12    }
13    else
14    {
15       System.out.print('*');
16       Print.stars(n - 1);
17    }
18 }
```

Let us first prove the method correct.

- In the base case (line 11), when `n` equals zero, we do nothing, which is equivalent to printing zero `*`s.
- Consider the recursive case (line 15 and 16). Assume that the recursive invocation `Print.stars(n - 1)` is correct, i.e. it prints `n - 1` `*`s. Then the recursive case (line 15 and 16) prints `n` `*`s.

$$\underbrace{*}_{\text{line 15}}\ \underbrace{\overbrace{*\cdots*}^{\texttt{n}-1}}_{\text{line 16}}$$

[2]Our correctness proof and our termination proof can be combined into a proof that

for each invocation of the method, the invocation is correct and terminates

by strong induction on the size of the invocation.

Next, let us prove termination.

1. We define the size of the invocation `Print.stars(n)` by

   $$\text{size}(\texttt{Print.stars(n)}) = \texttt{n}.$$

   From the precondition we can conclude that the size of an invocation is a natural number.

2. The size of the recursive invocation `Print.stars(n - 1)` is $\texttt{n} - 1$, which is smaller than `n`, the size of the original invocation `Print.stars(n)`.

The method `starsAndPlusses` prints a number of `*`s followed by the same number of `+`s.

```
1  /**
2   * Prints the given number of *s followed by the given number of +s.
3   *
4   * @param the number of *s and +s to be printed.
5   * @pre. n >= 0
6   */
7  public static void starsAndPlusses(int n)
8  {
9     if (n == 0)
10    {
11       // do nothing
12    }
13    else
14    {
15       System.out.print('*');
16       Print.starsAndPlusses(n - 1);
17       System.out.print('+');
18    }
19 }
```

Again, we first prove the correctness of the recursive method.

- In the base case (line 11), when `n` equals zero, we do nothing, which is equivalent to printing zero `*`s followed by zero `+`s.
- Consider the recursive case (line 15–17). Assume that the recursive invocation `Print.starsAndPlusses(n - 1)` is correct, i.e. it prints `n - 1` `*`s followed by `n - 1` `+`s. Then the recursive case (line 15–17) prints `n` `*`s followed by `n` `+`s.

$$\underbrace{*}_{\text{line 15}}\ \underbrace{\overbrace{*\cdots*}^{\texttt{n}-1}\overbrace{+\cdots+}^{\texttt{n}-1}}_{\text{line 16}}\ \underbrace{+}_{\text{line 17}}$$

Termination of this method can be proved in the same way as the previous method.

7.3 Recursive Methods that Return an Integer

Recall that the factorial of n, often denoted as $n!$, is defined as

$$n! = \prod_{i=1}^{n} i = 1 \times 2 \times \cdots (n-1) \times n.$$

Let us implement the method

```java
/**
 * Returns the factorial of the given number.
 *
 * @param a number whose factorial is returned.
 * @pre. n >= 1
 * @return n!
 */
public static int factorial(int n)
```

of the `Math` class using recursion. Note that, in contrast to all recursive methods that we have developed so far, this method returns a result.

If `n` is one, then `n`! is one. This is a base case. Otherwise, `n` is greater than one. In that case,

$$\mathtt{n}! = 1 \times 2 \times \cdots \times (\mathtt{n} - 1) \times \mathtt{n} = (\mathtt{n} - 1)! \times \mathtt{n}.$$

To compute $(\mathtt{n} - 1)!$ we can delegate to the method itself using `Math.factorial(n - 1)`. Hence, this is a recursive case. Combining the above, we arrive at the following recursive method.

```java
1  public static int factorial(int n)
2  {
3     int factorial;
4     if (n == 1)
5     {
6        factorial = 1;
7     }
8     else
9     {
10       factorial = Math.factorial(n - 1) * n;
11    }
12    return factorial;
13 }
```

Note that if we arrive at line 10, then $\mathtt{n} \geq 2$ and, hence, $\mathtt{n} - 1 \geq 1$ and, hence, the precondition for the invocation `Math.factorial(n - 1)` is satisfied.

Let us prove the method `factorial` correct.

- In the base case (line 6), when `n` is one, we assign the variable `factorial` the value of `n`! and, hence, this case is correct.

- Consider the recursive case (line 10). Assume that the recursive invocation `Math.factorial(n - 1)` is correct, i.e. it returns $(\mathtt{n}-1)!$. Then we assign $(\mathtt{n}-1)! \times \mathtt{n} = \mathtt{n}!$ to the variable `factorial` in line 10. Hence, this case is correct as well.

Next, we show that the method terminates.

1. We define the size of the invocation `Math.factorial(n)` by

   $$\text{size}(\mathtt{Math.factorial}(\mathtt{n})) = \mathtt{n}.$$

   From the precondition we can conclude that the size of an invocation is a natural number.

2. The size of the recursive invocation `Math.factorial(n - 1)` is $\mathtt{n} - 1$, which is smaller than `n`, the size of the original invocation `Math.factorial(n)`.

7.4 Recursive Methods for Lists

The method `contains` of the class `Collections` checks if a element is part of a list.

```
/**
 * Tests if the given list contains the given element.
 *
 * @param element the element.
 * @pre. element != null
 * @param list the list.
 * @pre. list != null
 * @return true if the given list contains the given element, false otherwise.
*/
public static <T> boolean contains(T element, List<T> list)
```

Obviously, an empty list, i.e. one of size zero, does not contain the given element. This is a base case. If the first element of the list is the element for which we are looking, then we have found it right away. This is a base case as well. Otherwise, the list is nonempty and the first element is not the one for which we are looking. Hence, we have to look in the rest of the list for the element. This can be done be delegating and, hence, this is a recursive case.

To obtain the rest of the list (without destroying the original list), we can make a shallow copy as follows.

```
List<T> rest = new ArrayList<T>(list.subList(1, list.size()));
```

The method `subList` returns a view of a portion of the list. In the above case, it returns a view of the portion starting at index 1 and ending just before index `list.size()`, that is, all but the first element (the element at index 0) of the list. We use the copy constructor of the `ArrayList` class to copy this portion.

We can implement the `contains` method as follows.

```
 1 public static <T> boolean contains(T element, List<T> list)
 2 {
 3    boolean contains;
 4    if (list.size() == 0)
 5    {
 6       contains = false;
 7    }
 8    else
 9    {
10       if (element.equals(list.get(0))
11       {
12          contains = true;
13       }
14       else
15       {
16          List<T> rest = new ArrayList<T>(list.subList(1, list.size()));
17          contains = Collections.contains(element, rest);
18       }
19    }
20    return contains;
21 }
```

In line 17, we have that `element != null` and `rest != null` and, hence, the precondition of `contains` is satisfied.

The correctness of the `contains` method can be proved as follows.

- In the first base case (line 6), `list` is empty. Obviously, an empty list does not contain `element` and, hence, setting `contains` to false is correct.

- In the second base case (line 12), `list` is nonempty and the first element of `list` is equal to `element`. In this case we have found the element and, hence, setting `contains` to true is correct.

- When we reach the recursive case (line 16–17), we know that `list` is nonempty and the first element of `list` is not equal to `element`. We assume that the recursive invocation `Collections.contains(element, rest)` is correct, i.e. it returns whether `element` is contained in `rest`.

  $$\underbrace{[\text{first}, \overbrace{\cdots}^{\texttt{rest}}]}_{\texttt{list}}$$

  Since the first element of `list` is not equal to `element`, `list` contains `element` if and only if `rest` contains `element`. Therefore,

```
contains = Collections.contains(element, rest);
```

is correct.

We can prove that the `contains` method terminates as follows.

1. We define the size of the invocation `Collections.contains(element, list)` by

   $$\text{size}(\texttt{Collections.contains}(\texttt{element}, \texttt{list})) = \texttt{list.size}().$$

   Obviously, `list.size()` is a natural number.

2. Since `rest` is obtained by removing the first element of `list`, `rest.size()` is smaller than `list.size()`. Hence, the size of the recursive invocation `Collections.contains(element, rest)` is smaller than the size of the original invocation `Collections.contains(element, list)`.

The method `isConstant` of the class `Collections` checks if a list is constant, i.e. if all its elements are equal.

```
/**
 * Tests if the given list is constant, that is, if all elements are the same.
 *
 * @param list the list.
 * @pre. list != null and list does not contain null
 * @return true if the given list is constant, false otherwise.
 */
public static <T> boolean isConstant(List<T> list)
```

If the list is empty, then all elements are equal and, hence, the list is constant. If the list contains a single element, then the list is constant as well. Both are base cases. If the first and the second element are not equal, then the list is not constant. This is yet another base case. Otherwise, the list contains at least two elements and the first and second element are equal. In that case, the list is constant if and only if the sublist starting with the second element is constant. The latter fact can be determined by invoking the method recursively on the rest of the sublist. Therefore, the `isConstant` method can be implemented as follows.

```
public static <T> boolean isConstant(List<T> list)
{
   boolean constant;
   if (list.size() <= 1)
   {
      constant = true;
   }
   else
   {
```

```
10     if (!list.get(0).equals(list.get(1)))
11     {
12       constant = false;
13     }
14     else
15     {
16       List<T> rest = new ArrayList<T>(list.subList(1, list.size()));
17       constant = Collections.isConstant(rest);
18     }
19   }
20   return constant;
21 }
```

We leave it to the reader to check that the precondition of `isConstant` is satisfied at line 17.

Let us prove the `isConstant` method correct.

- In the first base case (line 6), the list is either empty or contains a single element. In both cases all elements are equal and, hence, the assignment `constant = true` is correct.
- In the second base case (line 12), the first and second element are not equal and, therefore, the list is not constant, so the assignment `constant = false` is correct.
- Let us consider the recursive case (line 16–17). Assume that the recursive call is correct, i.e. it returns whether `rest` is constant. If we reach the recursive case, we know that the list has at least two elements and the first element is equal to the second element. Hence, `list` is constant only if `rest` is constant. Therefore, the assignment `constant = Collections.isConstant(rest)` is correct.

We can prove that the method `isConstant` terminates in the same way as we proved the termination of the `contains` method.

The `minimum` method of the `Collections` class returns the minimum of a nonempty list of integers.

```
/**
 * Returns the minimum element of the given list.
 *
 * @param list the list.
 * @pre. list != null and list.size() > 0 and list does not contain null
 * @return the minimum element of the given list.
 */
public static int minimum(List<Integer> list)
```

If the list contains a single integer, then obviously that integer is the minimum. This is a base case. Assume that the list contains more than one element. Then the minimum is the minimum of

- the first element,

- the minimum of the rest of the list.

The latter can be obtained by a recursive call.

```
 1 public static int minimum(List<Integer> list)
 2 {
 3   int minimum;
 4   if (list.size() == 1)
 5   {
 6      minimum = list.get(0);
 7   }
 8   else
 9   {
10      List<Integer> rest = new ArrayList<Integer>(list.subList(1, list.size()))
           ;
11      minimum = Math.min(list.get(0), Collections.minimum(rest));
12   }
13   return minimum;
14 }
```

Note that `rest != null`. Since `list` does not contain null, `rest` does not either. If we reach line 10, `list.size()` is greater than one and, hence, `rest.size()` is greater than zero. Therefore, precondition for the invocation of `minimum` in line 10 is satisfied.

The correctness of the `minimum` method can be proved as follows.

- In the base case (line 6), the list has a single element, which is the minimum.
- For the recursive case (line 10–11), assume that the recursive invocation `Collections.minimum(rest)` is correct, i.e. it returns the minimum of `rest`. As we already mentioned above, in this case the minimum of `list` is the minimum of the first element and the minimum of the rest of the list.

The proof of termination is similar to the one presented above.

In all the recursive methods for lists presented above, the method is recursively invoked on a copy of the list with the first element removed. Rather than removing the first element, we can also reduce the size of the list in other ways. For example, we can split a list in two as follows.

```
int middle = list.size() / 2;
List<T> left = new ArrayList<T>(list.subList(0, middle));
List<T> right = new ArrayList<T>(list.subList(middle, list.size()));
```

As an example, let us consider the `minimum` method again. In the recursive case we can split `list` into `left` and `right`. Since the minimum element of `list` is the minimum of

- the minimum element of `left` and
- the minimum element of `right`

we can implement the `minimum` method also as follows.

```
 1 public static int minimum(List<Integer> list)
 2 {
 3   int minimum;
 4   if (list.size() == 1)
 5   {
 6      minimum = list.get(0);
 7   }
 8   else
 9   {
10      int middle = list.size() / 2;
11      List<Integer> left = new ArrayList<Integer>(list.subList(0, middle));
12      List<Integer> right = new ArrayList<Integer>(list.subList(middle, list.
            size()));
13      minimum = Math.min(Collections.minimum(left), Collections.minimum(right))
            ;
14   }
15   return minimum;
16 }
```

Note that, since `list.size()` is greater than one when we reach line 10, the lists `left` and `right` are nonempty. This is needed to verify that the preconditions for `Collections.minimum(left)` and `Collections.minimum(right)` are satisfied. The details are left to the reader.

To modify the above correctness proof, we only have to revisit the recursive case (line 10–13). In that case, we assume that the recursive invocations `Collections.minimum(left)` and `Collections.minimum(right)` are correct, i.e. return the minimum element of the left and the right part of `list`, respectively. As we already mentioned above, the minimum element of `list` is the minimum of the minimum element of `left` and the minimum element of `right`. As a consequence, the assignment of line 13 is correct.

Termination can be proved as follows.

1. We define the size of the invocation `minimum(list)` by

   $$\text{size}(\texttt{minimum}(\texttt{list})) = \texttt{list}.\texttt{size}()$$

2. When we reach line 10 `list.size()` is greater than one. As a consequence, `left.size()` and `right.size()` are smaller than `list.size()`. Hence, the size of the recursive invocations `Collections.minimum(left)` and `Collections.minimum(right)` are smaller than the size of the original invocation `Collections.minimum(list)`.