

Solutions to questions of lecture 5  
by Kasper B. Graversen & Kasper Østerbye

version 1 - 8/03-04

---

ex 1.1  
substring has the following

preconditions (conjugated):  
- beginindex >= 0 and  
- endindex >= beginindex  
- endindex <= string length

postconditions:  
return is string beginning at beginindex, upto, but not including endindex  
original string is unchanged

ex 1.2  
Yes it is indeed a good idea. The focus of the paper is that not both the client and the supplier should check, but only one of them. Although there is no exact rule other than to achieve the simplest architecture (p. 5). personally I find it advantageous to have the check in the supplier, as it then is "reused" when other clients uses it. As noted on page 7, assertion violations are not special cases but are the result of bugs, which this exception certainly is trying to prevent.

1.3

from the javadoc

true if the character sequence represented by the argument is a prefix of the character sequence represented by this string; false otherwise. Note also that true will be returned if the argument is an empty string or is equal to this String object as determined by the equals(Object) method

```
public boolean startsWith(String prefix) {
    if(prefix == null) throw new NullPointerException();
    if (prefix.length() > this.length()) return false;
    for(int i = 0; i < prefix.length(); i++)
        if (this.charAt(i) != prefix.charAt(i) )
            return false;
    return true;
}
```

preconditions:  
- prefix string length >= 0

postconditions  
- true if prefix.equals( s.substring(0,prefix.length() ) )

(The postcondition shows that the prefix method could be written using the equals and substring methods.)

2.1

The following method works like this:

First it skips any non-letters to make sure we are at the beginning of a letter  
Then we loop, at the beginning of the loop we print out the word that starts there  
at the end of the loop, we make sure to skip any leading non-letters, so we are ready for next round in the loop.

```
static void toWords(String str){
    int i = 0;
    final int N = str.length();
    // skip leading non letters
    while (i < N && !Character.isLetter(str.charAt(i)))
        i++;
    while (i < N){
```

```

                                4-solutions.txt
                                // print the word
                                while (i<N && Character.isLetter(str.charAt(i))){
                                    System.out.print(str.charAt(i));
                                    i++;
                                }
                                System.out.println();
                                // skip non letters
                                while (i<N && !Character.isLetter(str.charAt(i))){
                                    i++;
                                }
                            }
                        }
                    }
                }
            }
        }
    }
}

```

## 2.2

```

class WordIterator implements OOPIterator{
    // Exercise 2.3
    // inv: if nextWord.size() = 0, the iterator is empty
    //      else nextWord is the head.
    //      i is either N or the first unprocessed letter in str
    //      N is the length of str.
    final String str;
    final int N;
    int i;
    String nextWord;

    WordIterator(String str){
        this.str= str;
        i=0;
        N = str.length();
        findNext();
    }
    /* pre: i is first non-processed letter in str
       post: if hasNext is true then nextWord is head.
            i is first non-processed letter in str.
    */
    private void findNext(){
        // skip non letters
        while( i<N && !Character.isLetter(str.charAt(i)) )
            i++;
        // find the word
        int start = i;
        while( i<N && Character.isLetter(str.charAt(i)) )
            i++;
        nextWord = str.substring(start,i);
    }

    public boolean hasNext(){
        return nextWord.length()>0;
    }

    public Object peek(){
        return nextWord;
    }

    public Object next(){
        String ret = nextWord;
        findNext();
        return ret;
    }

    public OOPIterator cloneMe(){
        try{
            return (OOPIterator)super.clone();
        }catch(Exception shouldNotHappen){
            return null;
        }
    }
}

```

## 2.4

According to the invariant, i is either str.length or the first unprocessed letter in str. N is the length of the str, and nextWord is the head of the iterator.

In the constructor, N is set to str.length, which is according to the invariant. i is set to 0, which is the first unprocessed character in str. Therefore the pre-condition for the private method findNext is fulfilled. The post condition of findNext is that the invariant is fulfilled.

2.5

The argument is pretty much the same, it depends on the correct functioning of findNext. i is the first unprocessed element in str. That is all which is needed in the pre condition for findNext. The post condition for findNext is that the invariant is fulfilled. Since returning from a method will not change the state of any fields, the invariant is still true when we return from next.

2.6

The point is that the counter in odd filter is initialized to its default value (0) when the object is created. The default value is given to ALL fields before any initializers or constructor code is executed. The initialization starts in the super class (FilterIterator). The constructor in FilterIterator calls the private method findNext, which in turn calls the abstract method condition, which is redefined in the subclass OddFilter. The condition method in OddFilter increments count from 0 to 1. At some point we return from the initialization of the superclass part, and starts to initialize the subclass part. At this point count is already 1. Therefore, if we initialize it to 0, we will do wrong. We can initialize it to 1, but that is kind of strange, as this is the value it already has.

The invariant regarding count is:

```
// count indicates the number of times
// condition has been called
```

Therefore the default value 0 is right.

Setting it explicitly to 1 will not break this invariant.

3.

See the code attached.