Software Development I
Univ.-Prof. Dr. Alois Ferscha
Examination, January 27, 2015

Institute for Pervasive Computing

Last name:	**SAMPLE SOLUTION**	Lecture hall:	
First name:		Seat:	
ID:	SKZ:	Points / Grade:	

- **Please fill in the header of your test** and make sure that you have your **student ID** („Keplerkarte") at hand
- **Open book:** You may use resources such as Java textbooks, lecture slides, assignments, and other written material
- It is not allowed to use any electronic devices (cell phones, PDAs, notebooks, iPads, calculators)
- It is not allowed to use extra paper
- **Please do not use pencils or red/green pens**
- Solutions have to be provided in the framed placeholders
- The score of each (part of an) example is an indication of the intended editing time (i.e**., 90 points in total corresponds to 90 minutes editing time**).

GOOD LUCK!

1. Data types (2+2+5+3+1=13 points)

Strings and character arrays. In the following problem you should complete the implementation of the `indexOf()`-function according to the following specs.

Specification

- Function/method interface: `int indexOf (String str, String subStr)`
  - returns the start index of the String `'subStr'` within `'str'` or -1 if not contained.
- Both strings, `'str'` and `'subStr'` are case sensitive.
- Except for the method `toCharArray()`, it is not allowed to use functions from the String class (in particular, the `String.indexOf()`-function ;-)).

Hints

- Convert all the strings into character-arrays before processing.

```
public class StringIndexOf {

    public static int indexOf(String str, String subStr) {

        // convert 'str' and 'subStr' into arrays of characters
        char[]  strArr    = str.toCharArray();
        char[]  subStrArr = subStr.toCharArray();
        int len = strArr.length;
        int subLen = subStrArr.length;
        int count = 0;

        // check if 'subStr' > 'str'; if so, 'subStr' cannot be contained in 'str'!
        if (subLen > len) {
            return -1;
        }
```

```
        for (int i = 0; i <= len - subLen; i++) {
            for (int j = 0; j < subLen; j++) {
                if (strArr[j+i] == subStrArr[j]) {
                    count++;
                    if (count == subLen) {
                          return i;
                    }
                } else {
                    count = 0;
                    break;
                }
            }
        }
        return -1;
    }

    public static void main(String[] args) {
        String str = "Hello World";
        String subStr = "World";

        System.out.println(str.indexOf(subStr)); // internal indexOf
        System.out.println(indexOf(str, subStr)); // my indexOf function
    }
}
```

2. Arrays/Loops (11 points)

Sum of odd numbers. An integer array `'intArr'` is given, containing any number of odd and even integers in any combination (see the example below). You should complete the algorithm in the main method below to sum up all the odd numbers in the array (and skip the even numbers).

Hints

- Even = "gerade" (2, 4, 6, etc.), odd = "ungerade" (1, 3, 5, etc.)

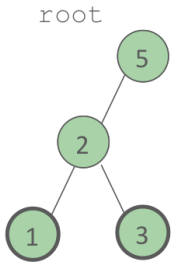
```
public class ArrayLooping {

    public static void main(String[] args) {
        int[] intArr = {1, 3, 5, 7, 2, 4, 8, 7, 5, 3, 2, 0, 5, 2, -1, -4};
```

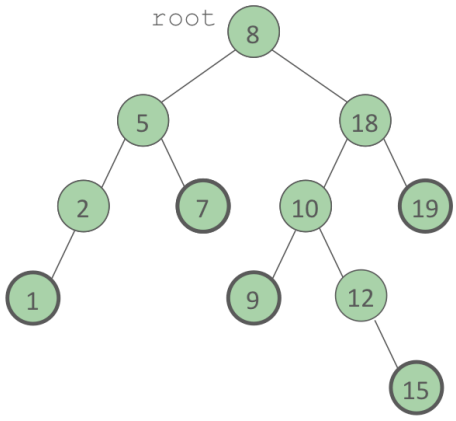
```
        // loop over array elements and sum up odd numbers (result: 35)
        int result = 0;
        for (int i=0; i < intArr.length; i++)
        {
            if (intArr[i] % 2 != 0)
                result += intArr[i];
        }

        System.out.println("Result: "+result);
    }
}
```

3. Binary Trees (2+4+4+5=15 points)

Counting leaves. Given a binary tree, compute the number of leaf nodes in that tree. A leaf node is defined as containing data (in our case, a value of type `int`) but neither a left nor a right child (subtree). The `countLeaves()` method should return 0 (zero) for an empty tree, or the number of leaves of the tree otherwise (see the examples below).

```
public class Node {
    int data;
    Node left;
    Node right;

    // constructor
    Node (int data) {
        left = null;
        right = null;
        this.data = data;
    }
}
```





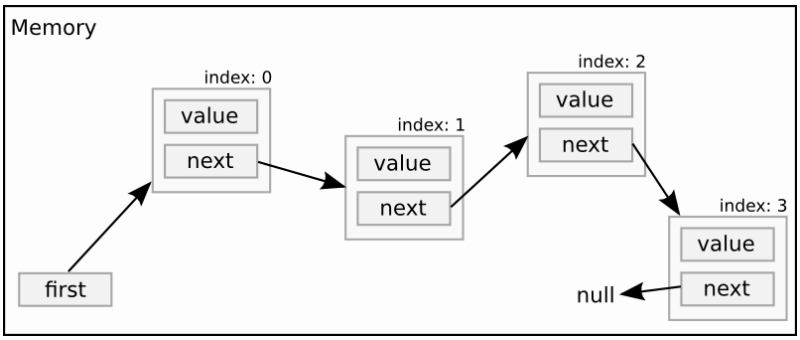
`countLeaves(root) = 2` `countLeaves(root) = 5`

```
public class BinTree {

    // Root node pointer ('null' for an empty tree)
    private Node root;

    /**
     * 'countLeaves' returns the number of leaves of that tree (or 0 for an emptry tree).
     * It uses a recursive helper that recurs down to find all the leaves.
     */
    public int countLeaves() {
        return countLeaves(root);

    }
```

```
    private int countLeaves(Node node) {

        // recursion anchor: number of leafs for an empty tree is zero.
        if (node==null) {
            return 0;
        }

        // leaf
        else if (node.left==null && node.right == null)

        {
            return 1;
        }
        // recursive call
        return countLeaves(node.left) + countLeaves (node.right);

    }
}
```

4. Singly Linked List (3+3+2+3+10+8+3=32 points)

Below, you see a figure describing a singly linked list and assume that you have also been provided with the `LinkedListItem` class that already implements the following variables and methods:




```
public class LinkedListItem {
    private String value;
    private LinkedListItem next;

    public LinkedListItem(String contents);    // The constructor!

    public void setNext(LinkedListItem next); // Set the next item in chain
    public LinkedListItem getNext();          // Get the next item in chain
    public String getContents();              // Get the contents of the item
}
```

Fill in the missing code in the partial implementation below to make the source code complete. Note that the list index starts at 0, i.e. the first element in the list has index 0.

Notes

- The list is sorted ascending and contains no duplicate items.
- Look at (and use) the methods provided in the `LinkedListItem`-class!
- **`int`** `search(`**`String`** `searchContent)`
  Please note that, if a content is not found, the search method should return -1.
- **`boolean`** `addTo(`**`String`** `text)`
  Before inserting an element into the list, you should make use of the `String.compareTo(otherString)`-function. This function will return 0 for equality, a value below 0 if the string is less (lexicographically smaller) than the other String (e.g., string "a" will be less than "b") or a value above 0 otherwise.

```
public class LinkedList {
    LinkedListItem first;

    public LinkedList() {
    }

    public int search(String searchContent) {
        int idx = 0;
        if (first != null) {

            // complete the for loop (condition)

            for (LinkedListItem item = first; item != null; item=item.getNext()  ) {


                // compare elements

                if ( item.getContents().equals(searchContent)                ) {

                    return idx;
                }
                idx++;
            }
        }
        // return reference
        return -1;
    }

    public boolean addTo(String text) {
        // 1) Special case: list is empty
        if (first == null) {
            // instantiate and parametrize new node
            first = new LinkedListItem(text);

            return true;
        }
        LinkedListItem newItem = new LinkedListItem(text);
```

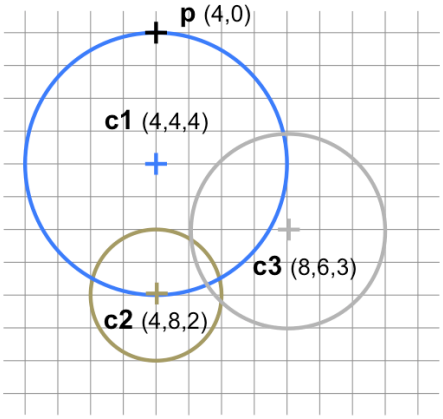
```
        // 2) Special case: add element as first element to the list
        if (first.getContents().compareTo(text) == 0) {
            return false;
        } else if (text.compareTo(first.getContents()) < 0) {
            // insert at first position
            
            
            newItem.setNext(first);
            first = newItem;
            return true;
            
            
            
            
        }

        LinkedListItem item = first.getNext();
        LinkedListItem prev = first;

        for (; item != null; item = item.getNext()) {
            if (item.getContents().compareTo(text) == 0) {
                return false;
            } else if (text.compareTo(item.getContents()) < 0) {
                // add new item at the correct place,
                // set 'next' pointer of the previous node, etc.
                
                
                
                newItem.setNext(item);
                prev.setNext(newItem);
                return true;
                
                
                
                
            }
            prev = item;
        }
        // set 'next' pointer of the previous node
        
        prev.setNext(newItem);
        
        
        return true;
    }
}
```

5. Objects/Classes/Methods (3+3+5+3+3+2=19 points)

The class `Circle` models a circle in 2D-space with x and y coordinates and a radius r. Complete the implementation of the class as described in the following.

1. **Instance variables**

   Define three private instance variables for the x and y coordinates and thre radius r (all of type `int`).



2. **Constructors**

   a) Add a constructor that constructs a circle with given x and y coordinates and radius r.

3. **Instance methods**

   a) Implement a method called `isOriginOf` that takes two `int` values x and y as input and returns a `boolean` indicating whether or not the given point (x, y parameters) is **located on the circumference of the circle**.

   Use the following circle equation to check the condition:

$$(x - x_{CC})^2 + (y - y_{CC})^2 = r^2$$

$$x, y : \text{Coordinates of the point (method parameters)}$$

$$x_{CC}, y_{CC} : \text{Center coordinates of the circle}$$

   `Math.pow (a, b)` returns the value of the first argument raised to the power of the second argument.

   b) Implement an **overloaded** method `isOriginOf` that takes another circle as input and returns `true/false` whether or not the **center of this circle** is located on the circumference of the circle the method is called on.

4. **Class methods**

   a) Add a class method `isOriginOf` that takes two circles (each instances of `Circle`) as input, and returns true/false whether or not the center of the 2nd circle is located on the circumference of the 1st circle.

Notes

- Your solutions should be as efficient as possible, e.g., try to **re-use code**
- Be careful with **visibility** of variables and methods

```
public class Circle {
    // 1. Instance variables
    private int x;
    private int y;
    private int r;

    // 2. Constructors
    public Circle(int x, int y, int r) {
        this.x = x;
        this.y = y;
        this.r = r;
    }
```

```
    // 3. Instance methods 'isOriginOf'
    public boolean isOriginOf(int x, int y) {
        return 0 == Math.pow(x - this.x, 2) + Math.pow(y - this.y, 2)-
        Math.pow(this.r, 2);
    }

    public boolean isOriginOf(Circle p) {
        return isOriginOf(p.x, p.y);
    }

    // 3. Class method.
    public static boolean isOriginOf(Circle c1, Circle c2) {
        return c1.isOriginOf(c2);
    }
}
```

Complete finally the test program outlined below.

```
public class CircleTest {
    public static void main(String[] args) {
        Circle c1 = new Circle(4,4,4);
        Circle c2 = new Circle(4,8,2);
        Circle c3 = new Circle(8,6,3);

        // check instance method 'isOriginOf' on c1 with a point x=4,y=0:
        System.out.println(c1.isOriginOf(4,0)); // output: true (see above)

        // check instance method 'isOriginOf' from c1 with c2:
        System.out.println(c1.isOriginOf(c2));

        // check class method 'isOriginOf' with c1 and c3:
        System.out.println(Circle.isOriginOf(c1, c3));

    }
}
```