Pokud zadání nerozumíte nebo se vám zdá nejednoznačné, zeptejte se. Odpovídejte jasně a k věci. Pište čitelně a srozumitelně. při psaní kódu nepoužívejte řezničinky.

1. Odkrokujte následující kód a napište stavy virtuálního stroje i) na začátku programu před provedením prvního příkazu, ii) těsně po prvním návratu z metody `Cell.g`, iii) těsně před druhým voláním metody `Cell.g` a iv) na konci programu po provedení všech příkazů:

```
new Cell();
Cell c = new Cell();
new Cell().f(c);
```

```
class Cell {
    Cell c1;
    static Cell c2;

    Cell() {
        c1 = c2;
        c2 = this;
    }

    void f(Cell c) {
        c.c1.g(c);
        c = new Cell();
        c.c1.g(c);
    }

    void g(Cell c) {
        c.c1 = c2;
    }
}
```

Řešení:

$$\left(\begin{array}{c|c|l|l} & & \texttt{new Cell();} & Cell.c2 = null \\ & & \texttt{Cell c = new Cell();} & \\ & & \texttt{new Cell().f(c);} & \end{array} \right)$$

$$\left(\begin{array}{l|l|l|l} \#1 \to Cell(c1 = null) & this = \#3 & \texttt{c = new Cell();} & Cell.c2 = \#3 \\ \#2 \to Cell(c1 = \#3) & c = \#2 & \texttt{c.c1.g(c);} & \\ \#3 \to Cell(c1 = \#2) & \hline c = \#2 & & \end{array} \right)$$

$$\left(\begin{array}{l|l|l|l} \#1 \to Cell(c1 = null) & this = \#3 & \texttt{c.c1.g(c);} & Cell.c2 = \#4 \\ \#2 \to Cell(c1 = \#3) & c = \#4 & & \\ \#3 \to Cell(c1 = \#2) & \hline c = \#2 & & \\ \#4 \to Cell(c1 = \#3) & & & \end{array} \right)$$

$$\left(\begin{array}{l|l|l|l} \#1 \to Cell(c1 = null) & c = \#2 & & Cell.c2 = \#4 \\ \#2 \to Cell(c1 = \#3) & & & \\ \#3 \to Cell(c1 = \#2) & & & \\ \#4 \to Cell(c1 = \#4) & & & \end{array} \right)$$

2. V kódu níže dochází k porušení substitučního principu. Napište kde a vysvětlete proč (obecně i na příkladu).

```
/**
 * Trida A reprezentuje seznam celych cisel.
 */
class A {
    LinkedList<Integer> l = new LinkedList<Integer>();

    /** Vlozi cislo i na konec tohoto seznamu. */
    void push(Integer i) {
        l.addFirst(i);
    }

    /** Odstrani a vrati posledni cislo z tohoto seznamu. */
    Integer pop() {
        return l.removeLast();
    }

    /** Vrati delku tohoto seznamu. */
    int size() {
        return l.size();
    }

    /** Vrati cislo z tohoto seznamu na indexu i. i musi byt v rozmezi
     * 0...size-1. 0 znamena zacatek, size-1 konec seznamu.
     */
    Integer get(int i) {
        assert 0 <= i;
        assert i < l.size();
        return l.get(i);
    }
}

/**
 * Trida B reprezentuje seznam celych cisel.
 */
class B extends A {
    /** Vlozi cislo i na zacatek tohoto seznamu. */
    void push(Integer i) {
        l.addLast(i);
    }
}
```

Řešení: metoda `B.push` by podle dokumentace nadtřídy měla vkládat na konec seznamu, ne na začátek. Chyba proto může být v kódu, který výpadá takto:

```
void f(A a) { /* kod */ }

f(new B());
```

Chyba v implementaci oproti dokumentaci je v tomto případě irelevantní.

3. Třída `LinkedList` reprezentuje oboustranně zřetězený cyklický vzestupně setříděný spojový seznam. Dopište do ní metodu `insert`, která vloží danou hodnotu na správné místo v seznamu. Pokud již daná hodnota v seznamu je, nedělá nic. Dejte si pozor, abyste správně nastavili ukazatele `next` i `prev`.

```
class Node {
    int contents;
    Node prev, next;

    Node(int c) {
        contents = c;
    }
}

class LinkedList {
    Node first;

    boolean contains(int elem) {
        if (first == null) return false;
        Node temp = first;
        do {
            if (temp.contents == elem) return true;
            if (temp.contents > elem) return false;
            temp = temp.next;
        } while (temp != first);
        return false;
    }

    void insert(int elem) {
```

Řešení:

```
    void insert(int elem) {
        if (first == null) {
            first = new Node(elem);
            first.next = first;
            first.prev = first;
            return;
        }
        if (first.contents > elem) {
            first = insertBefore(first, elem);
            return;
        }
        if (first.prev.contents < elem) {
            insertBefore(first, elem);
            return;
        }
        for (Node temp = first; true; temp = temp.next) {
            if (temp.contents == elem) return;
            if (temp.contents > elem) {
                insertBefore(temp, elem);
                return;
            }
        }
```

```
}

static Node insertBefore(Node n, int elem) {
    Node newNode = new Node(elem);
    newNode.next = n;
    newNode.prev = n.prev;
    n.prev.next = newNode;
    n.prev = newNode;
    return newNode;
}
```

4. Co vypíše volání `new A().f(new B(new D()))`?

```
class A {
    void f(B b) {
        System.out.println("F1");
        b.i(this);
        System.out.println("F2");
    }
    void g(C c) { System.out.println("G"); }
    void h(D d) { System.out.println("H"); }
}

class B {
    C c;

    B(C c) { this.c = c; }
    void i(A a) {
        System.out.println("I1");
        c.j(a);
        System.out.println("I2");
    }
}

class C {
    void j(A a) {
        System.out.println("J1");
        a.g(this);
        System.out.println("J2");
    }
}

class D extends C {
    void j(A a) {
        System.out.println("K1");
        a.h(this);
        System.out.println("K2");
    }
}
```

Řešení:

```
F1
I1
K1
H
K2
I2
F2
```

5. Dopište implementace metody `String[] Node.allStrings()`, která vrátí seznam všech řetězců v instancích třídy `Positive` z celého podstromu.

```
interface Node {
    String[] allStrings();
}

class NodeWithTwoChildren implements Node {
    Node left, right;

    public String[] allStrings() { /* kod */ }
}

class Negative implements Node {
    String s;

    public String[] allStrings() { /* kod */ }
}

class Positive implements Node {
    Strings s;

    public String[] allStrings() { /* kod */ }
}
```

Řešení:

```
class NodeWithTwoChildren implements Node {
    Node left, right;

    public String[] allStrings() {
        String[] l = left.allStrings();
        String[] r = right.allStrings();
        String[] result = new String[l.length + r.length];
        System.arraycopy(l, 0, result, 0, l.length);
        System.arraycopy(r, 0, result, l.length, r.length);
        return result;
    }
}

class Negative implements Node {
    String s;

    public String[] allStrings() { return new String[0]; }
}

class Positive implements Node {
    Strings s;

    public String[] allStrings() { return new String[] { s } }
}
```