Pokud zadání nerozumíte nebo se vám zdá nejednoznačné, zeptejte se. 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:

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

```
class Cell {
    Cell c;
    static int contents;

    Cell() {
        c = this;
    }

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

    void g() {
        contents++;
        c.contents++;
    }
}
```

Řešení:

$$\left(\;\middle|\; \begin{array}{l} \texttt{Cell c = new Cell();} \\ \texttt{c.f(c);} \end{array} \;\middle|\; Cell.contents = 0 \right)$$

$$\left(\begin{array}{l} \#1 \to Cell(c = \#1) \end{array} \;\middle|\; \begin{array}{l} this = \#1 \\ \hline c = \#1 \\ \hline c = \#1 \end{array} \;\middle|\; \begin{array}{l} \texttt{c = new Cell();} \\ \hline \texttt{this.g();} \end{array} \;\middle|\; Cell.contents = 2 \right)$$

$$\left(\begin{array}{l} \#1 \to Cell(c = \#1) \\ \#2 \to Cell(c = \#2) \end{array} \;\middle|\; \begin{array}{l} this = \#1 \\ \hline c = \#2 \\ \hline c = \#1 \end{array} \;\middle|\; \begin{array}{l} \texttt{this.g();} \end{array} \;\middle|\; Cell.contents = 2 \right)$$

$$\left(\begin{array}{l} \#1 \to Cell(c = \#1) \\ \#2 \to Cell(c = \#2) \end{array} \;\middle|\; c = \#1 \;\middle|\; \;\middle|\; Cell.contents = 4 \right)$$

2. Funguje třída `Buffer` správně, tzn. podle dokumentace? Pokud ne, napište konkrétní důvod proč.

```
class Buffer {
    int size; Object[] elems; boolean[] flags;

    /**
     * Vytvori buffer s indexy 1,...,size a nastavi vsechny jeho
     * indexy do stavu "bez hodnoty". size musi byt alespon 1.
     */
    Buffer(int size) {
        this.size = size;
        elems = new Object[size];
        flags = new boolean[size];
        // implicitni hodnota booleanu je false
    }

    /**
     * Prepne index i do stavu "s hodnotou" a nastavi jeho hodnotu
     * na elem. Pro i musi platit 0 < i <= size.
     */
    void set(int i, Object elem) {
        assert 0 < i; assert i <= elems.length;
        elems[i - 1] = elem;
        flags[size - i] = true;
    }

    /**
     * V pripade, ze je index i ve stavu "bez hodnoty", vyhodi
     * vyjimku, pokud je index i ve stavu "s hodnotou", vrati
     * prvek na indexu i. Pro i musi platit 0 < i <= size.
     */
    Object get(int i) {
        assert 0 < i; assert i <= elems.length;
        if (flags[size - i]) return elems[i - 1];
        else throw new IllegalStateException();
    }

    /**
     * V pripade, ze je index i ve stavu "bez hodnoty", vyhodi
     * vyjimku, pokud je index i ve stavu "s hodnotou", vrati
     * prvek na indexu i a prepne index i do stavu "bez hodnoty".
     * Pro i musi platit 0 < i <= size.
     */
    Object remove(int i) {
        assert 0 < i; assert i <= elems.length;
        if (flags[size - i]) {
            flags[size - i] = false;
            return elems[i - 1];
        } else throw new IllegalStateException();
    }
}
```

Řešení: Ano, třída `Buffer` funguje podle dokumentace.

3. Třída `LinkedList` reprezentuje cyklický spojový seznam — spojový seznam, který má v posledním prvku ukazatel na první. Dopište kód metody `remove`, která odstraní prvek na zadaném indexu (první prvek má index 1, indexy menší než nula nebo větší než délka seznamu nemusíte řešit). Existující kód nesmíte měnit.

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

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

class LinkedList {
    Node first;

    int size() {
        if (first == null) return 0;
        int size = 1;
        for (Node temp = first; temp.next != first;
            temp = temp.next) size++;
        return size;
    }

    void remove(int index) {
        assert 1 <= index; assert index <= size();
```

Řešení:

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

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

class LinkedList {
    Node first;

    int size() {
        if (first == null) return 0;
        int size = 1;
        for (Node temp = first; temp.next != first;
                temp = temp.next) size++;
        return size;
    }

    void remove(int index) {
        assert 1 <= index;
        assert index <= size();
```

```
        if (index == 1) {
            Node tmp = first;
            while (tmp.next != first) tmp = tmp.next;
            first = first.next;
            tmp.next = first;
            return;
        }
        Node tmp = first;
        while (index > 2) {
            tmp = tmp.next;
            index--;
        }
        tmp.next = tmp.next.next;
    }
}
```

4. Předpokládejte, že třída `Queue` korektně implementuje neomezenou frontu — seznam prvků, do kterého se dá metodou `add` na začátek vkládat a ze kterého se dá metodou `remove` z konce odebírat. Co je špatně na implementaci třídy `PriorityQueue`? Nápověda: chyba souvisí s porušením zapouzdření, ale nejde pouze o modifikátory `public` nebo `private`.

```
class Element {
    int value; boolean priority;

    Element(int v, boolean p) {
        value = v;
        priority = p;
    }
    void setValue(int v) { value = v; }
    int getValue() { return value; }
    void setPriority(boolean p) { priority = p; }
    boolean getPriority() { return priority; }
}

class Queue {
    /* atributy */

    void add(Element e) { /* kod */ }
    Element remove() { /* kod */ }
    boolean isEmpty() { /* kod */ }
}

class PriorityQueue {
    Queue highPriorityElements = new Queue();
    Queue lowPriorityElements = new Queue();

    /* Prida do fronty element e. e nesmi byt null. */
    void add(Element e) {
        assert e != null;
        if (e.getPriority()) highPriorityElements.add(e);
        else lowPriorityElements.add(e);
    }

    /*
     * Vrati (a odebere) z fronty nejdrive vlozeny prvek e
     * s vyssi prioritou (tzn. e.getPriority() == true).
     * Pokud takovy prvek neexistuje, vrati a odebere
     * nejdrive vlozeny prvek e s nizsi prioritou (tzn.
     * e.getPriority() == false). Pokud ani takovy prvek
     * neexistuje, vrati null.
     */
    Element remove() {
        if (! highPriorityElements.isEmpty())
            return highPriorityElements.remove();
        if (! lowPriorityElements.isEmpty())
            return lowPriorityElements.remove();
        return null;
    }
}
```

Řešení: třída `Queue` nevrací prvky ve správném pořadí, pokud uživatel mění priority už vloženych prvků.

5. Dopište implementaci metody `NodeProcessor.getNumberOfChildren` aniž byste použili operátor `instanceof` nebo metodu `Object.getClass`. Do rozhraní `Node` a tříd `NodeWithTwoChildren`, `NodeWithOneChild` a `NodeWithNoChildren` můžete přidat maximálně jednu metodu.

```
interface Node {

}

class NodeWithTwoChildren implements Node {
    Node left, right;
}

class NodeWithOneChild implements Node {
    Node child;
}

class NodeWithNoChildren implements Node {

}

class NodeProcessor {
    int getNumberOfChildren(Node n) {
```

Řešení:

```
interface Node {
    int getNumberOfChildren();
}

class NodeWithTwoChildren implements Node {
    Node left, right;

    public int getNumberOfChildren() {
        return left.getNumberOfChildren() +
               right.getNumberOfChildren() + 2;
    }
}

class NodeWithOneChild implements Node {
    Node child;

    public int getNumberOfChildren() {
        return child.getNumberOfChildren() + 1;
    }
}

class NodeWithNoChildren implements Node {
    public int getNumberOfChildren() {
        return 0;
    }
}

class NodeProcessor {
```

```
    int getNumberOfChildren(Node n) {
        return n.getNumberOfChildren();
    }
}
```