E1. (příklad na spojové seznamy) Dopište kód metod `insertFirst` a `insertLast` tak, aby `insertFirst` vkládala na začátek spojového seznamu odkazovaného proměnnou `head` a `insertLast` vkládala na jeho konec.

```
class Node {
    Node n;
    int c;

    static Node head;

    static void insertFirst(int c) {
        /* kod */
    }

    static void insertLast(int c) {
        /* kod */
    }
}
```

E2. (příklad na spojové seznamy) Nakreslete, jak vypadají objekty v paměti po vykonání metody `f`.

```
class Node {
    Node n;
    int c;

    Node(int cont) { c = cont; }

    static void f() {
        Node a = new Node(1);
        Node b = new Node(2);
        Node c = new Node(3);
        Node d = new Node(4);
        a.n = d;
        b.n = a;
        a.n.n = b;
        c.n = b;
        b.n = c;
        b.n.n = d;
    }
}
```

Následující dva příklady se budou odvolávat na třídy `Stack` a `Queue`. Zde je jejich kód.

```
class Stack {
    int[] contents = new int[10000];
    int count;

    void add(int e) {
        contents[count++] = e;
        count %= 10000;
    }

    int remove() {
        return contents[--count];
    }

    int[] removeMultiple(int c) {
        int[] result = new int[c];
        for (int i = 0; i < c; i++) result[i] = remove();
        return result;
    }
}

class Queue {
    int[] contents = new int[10000];
    int first, last;

    void add(int e) {
        contents[first++] = e;
        first %= 10000;
    }

    int remove() {
        int result = contents[last++];
        last %= 10000;
        return result;
    }

    int[] removeMultiple(int c) {
        int[] result = new int[c];
        for (int i = 0; i < c; i++) result[i] = remove();
        return result;
    }
}
```

E3. Zredukujte duplikaci kódu ve třídách `Stack` a `Queue`.

E4. Zredukujte duplikaci v kódu níže. Pokud chcete, můžete třídy `Stack` a `Queue` modifikovat.

```
static void insertMultiple(Stack s, int[] array) {
    for (int i = 0; i < array.length; i++)
        s.add(array[i]);
}

static void insertMultiple(Queue q, int[] array) {
    for (int i = 0; i < array.length; i++)
        q.add(array[i]);
}
```

E5. V následujícím kódu je něco špatně (kromě chybějících těl metod). Co a proč?

```
interface VrchniPrchni {
    public void steraceStiraji();
    public void ostrikovaceStrikaji();
}

class DaliborVrana implements VrchniPrchni {
    public static void steraceStiraji() { /* kod */ }
    public void ostrikovaceStrikaji() { /* kod */ }
}
```

D1. Třída `Queue` z příkladu E3 za určitých okolností selže – metoda `remove` už nebude vracet hodnotu nejstaršího vloženého (a nevybraného) prvku. Popište za jakých a do metody `add` dopište assert, který tomu zabrání.

D2. Změňte třídu `Node` tak, aby instance vracely svoje děti pomocí návrhového vzoru iterátor.

```
class Node {
    Node[] children;

    Node() { /* kod */ }
    int getNumberOfChildren() { return children.length; }
    Node getChild(int index) { return children[index]; }
}
```

D3. Do třídy `Queue` z příkladu E3 dopište podporu návrhového vzoru observer, která rozesílá události při vložení každého desátého prvku.

D4. Třída `InfiniteArray` reprezentuje nekonečné pole (s indexy od nuly výš), které je na počátku celé naplněné nulami. Má dvě možné reprezentace – pole, které se automaticky "prodlužuje", nebo spojový seznam dvojic (index, hodnota). Napište implementaci třídy `InfiniteArray`, která mezi těmito dvěma reprezentacemi přepíná – pokud počet nulových prvků v intervalu (0, nejvyšší index s nenulovým prvkem) klesne pod třetinu, přepne na reprezentaci polem, pokud naopak vzroste nad dvě třetiny, přepne na reprezentaci spojovým seznamem. Při implementaci použijte návrhový vzor state/strategy.

```
class InfiniteArray {
    /* atributy */
    void set(int index, int value) { /* kod */ }
    int get(int index) { /* kod */ }
}
```

D5. Má smysl vytvářet interface s naprosto stejnými metodami jako má třída, která tento interface implementuje (tzn. žádné navíc, vizte níže)? Proč (stačí příklad)?

```
interface ListInterface {
  void append(Element e);
  Element get( int index);
  void remove( int index);
}

class List {
  public void append(Element e) { /* kod */ }
  public Element get( int index) { /* kod */ }
  public void remove( int index) { /* kod */ }
}
```

C1. Co vypíše metoda f a proč?

```
class Mac {
    private String getName() { return "Mac"; }
    public void printName() {
        System.out.println( getName());
    }
}

class PC extends Mac {
    private String getName() { return "PC"; }

    static void f() {
        new PC().printName();
    }
}
```

C2. Do rozhraní `Zajic` a tříd `Bob` a `Bobek` doplňte podporu pro návrhový vzor visitor a využijte ji v metodě `Vecernicek.zacni` tak, abyste se zbavili použití operátoru `instanceof`.

```
interface Zajic {
    void vstavej();
    void spi();
}

class Bob implements Zajic {
    public void vstavej() { /* kod */ }
    public void spi() { /* kod */ }
}

class Bobek implements Zajic {
    public void vstavej() { /* kod */ }
    public void spi() { /* kod */ }
}

class Vecernicek {
    void zacni(Zajic z) {
        if (z instanceof Bob) z.vstavej();
        if (z instanceof Bobek) z.spi();
    }
}
```

C3. Instance třídy `Fraction` jsou imutabilní – během svého života nemění svoji hodnotu. Napište tovární metodu `Fraction get(int n, int d)`, která pro stejné hodnoty `n` a `d` vrací stejné instance. Pokud chcete, můžete použít třídu `HashMap`.

```
class Fraction {
    int nominator, denominator;

    Fraction(int n, int d) {
        nominator = n;
        denominator = d;
    }

    Fraction times(Fraction f) {
        return new Fraction( nominator * f.nominator,
                             denominator * f.denominator);
    }
}
```

C4. Napište třídy implementující návrhový vzor interpret, který by byl schopen interpretovat níže uvedený kousek kódu.

```
a = 10;
b = a;
c = b + 1;
print c;
```

C5. Níže uvedený kód implementuje návrhový vzor. Který?

```
interface A {
    B getB();
    C getC();
}

interface B {}

interface C {}

class D implements B {}
class E implements B {}

class F implements C {}
class G implements C {}

class H implements A {
    public B getB() {
        return new D();
    }

    public C getC() {
        return new F();
    }
}

class I implements A {
    public B getB() {
        return new E();
    }

    public C getC() {
        return new G();
    }
}

class J {
    A a = new H();

    void h() {
        a = new H();
    }

    void i() {
        a = new I();
    }

    A getA() {
        return a;
    }
}
```