E1. (příklad na spojové seznamy) Doprogramujte kód metody `add` tak, aby vkládala svůj parametr do spojového seznamu, na který ukazuje proměnná `head`. Výsledný seznam by měl být vzestupně seřazen podle obsahu instančních proměnných `c`. Dejte si pozor na krajní případy.

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

    static Node head;

    static void add(Node n) {
        /* 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 = b;
        a.n.n = c;
        c.n = a;
        b.n = d;
        b.n.n = a;
    }
}
```

E3. Ve třídě `Queue` je něco špatně. Napište co, proč a opravte to.

```
class Queue {
    int[] contents;
    int first, last;

    /** Metoda init musi byt zavolana
        pred prvnim pouzitim teto instance. */
    void init() { contents = new int[10000]; }

    /** Vlozi parametr dovnitr fronty. */
    void insert(int c) { contents[last++] = c; }

    /** Vrati nejstarsi prvek ve fronte. */
    int remove() { return contents[++first]; }
}
```

E4. Zredukujte kód třídy `RedBlueSet`, zachovejte funkčnost metod `addRed`, `containsRed`, `removeRed`, `addBlue`, `containsBlue` a `removeBlue`. Pokud chcete, můžete část funkčnosti třídy `RedBlueSet` přesunout do jiné třídy (to je nápověda.

```
class RedBlueSet {
    int[] redContents;
    int redCount;
    int[] blueContents;
    int blueCount;

    void addRed(int c) {
        redContents[redCount++] = c;
    }

    int indexOfRed(int c) {
        for (int i = 0; i < redCount; i++)
            if (redContents[i] == c) return i;
        return -1;
    }

    boolean containsRed(int c) {
        return indexOfRed(c) != -1;
    }

    void removeRed(int c) {
        int i = indexOfRed(c);
        if (i == -1) return;
        redContents[i] = redContents[--redCount];
    }

    void addBlue(int c) {
        if (containsBlue(c)) return;
        blueContents[blueCount++] = c;
    }

    int indexOfBlue(int c) {
        for (int i = 0; i < blueCount; i++)
            if (blueContents[i] == c) return i;
        return -1;
    }

    boolean containsBlue(int c) {
        return indexOfBlue(c) != -1;
    }

    void removeBlue(int c) {
        int i = indexOfBlue(c);
        if (i == -1) return;
        blueContents[i] = blueContents[--blueCount];
    }
}
```

E5. Vytvořte alternativní implementaci třídy `ExtendedStack`, která se chová stejně, ale nepoužívá dědičnost. Vyvarujte se duplikace kódu třídy `Stack`. Svoji volbu řešení podrobně zdůvodněte.

```
class Stack {
    int[] contents = new int[1000];
    int offset = 0;

    void push(int number) { contents[offset++] = number;}
    int pop() { return contents[--offset]; }
}

class ExtendedStack extends Stack {
    int top() { return contents[offset]; }
}
```

D1. Kód metody `sort` není naprogramován ideálně – napište proč a zlepšete ho (při zachování funkčnosti). Tip: budete muset upravit celou třídu. Bubble sort, quick sort a heap sort jsou různé typy algoritmů pro třídění.

```
class Sorter {
    static void bubbleSort(int[] array) {
        /* kod */
    }

    static void quickSort(int[] array) {
        /* kod */
    }

    static void heapSort(int[] array) {
        /* kod */
    }

    int preferredSortType = 1;

    void setPreferredSortType(int type) {
        preferredSortType = type;
    }

    void sort(int[] array) {
        if (preferredSortType == 1) bubbleSort(array);
        if (preferredSortType == 2) quickSort(array);
        if (preferredSortType == 3) heapSort(array);
    }
}
```

D2. Dopište vhodné asserty do metody `sort` ze zadání předchozího příkladu.

D3. Třída `Account` je „natvrdo“ svázaná se třídou `Marketing` – to není dobré. Navrhněte způsob jak to odstranit, naimplementujte relevantní části kódu ve třídě `Account` a naznačte jak by se měl změnit zbytek aplikace. Zachovejte funkčnost – instance třídy `Marketing` se musí dozvědět o každém vlastníkovi milionového zůstatku.

```
/** Reprezentuje ucet v bance. */
class Account {
    /** Zustatek na uctu. */
    int balance;
    /** Vlastnik tohoto uctu. */
    Person owner;

    /** Vlozi dane mnozstvi penez na ucet. */
    void deposit(int amount) {
        /* kod pro vlozeni penez na ucet */
        if (balance > 1000000)

Marketing.getMarketing().possiblyANewMillionaireOfferThem
IcelandicBonds(owner);
    }

    /** Zbytek kodu. */
}
```

D4. Do třídy `Account` z předchozího příkladu dopište tovární metodu tak, aby při pokusu o vytvoření druhého účtu se stejným vlastníkem vrátila `null`.

D5. Třída `Set` reprezentuje množinu pomocí kruhového spojového seznamu (tzn. poslední prvek ukazuje zpátky na první). Určete její konzistentní stav a napište metodu `bool OK()`, která vrátí true nebo false podle toho, jestli je daná instance v konzistentním stavu nebo ne.

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

    Node(int c, Node n) { contents = c; next = n; }
}

class Set {
    Node head;
    int count;

    void add(int number) {
        if (head == null) {
            head = new Node(number, null);
            head.next = head;
        }
        Node tmp = head;
        while (tmp.next != head) tmp = tmp.next;
        tmp.next = new Node(number, tmp.next);
        count++;
    }

    boolean contains(int number) {
        Node tmp = head;
        for (int i = 0; i < count; i++) {
            if (tmp.contents == number) return true;
            tmp = tmp.next;
        }
        return false;
    }
}
```

C1. V JDBC se spojení s databází naváže pomocí kódu napsaného níže. Instance vrácená voláním `createStatement` implementuje rozhraní `Statement`, ale její skutečnou třídu JDBC určí podle řetězce zadaného jako parametr metody `getConnection`. Který návrhový vzor (z těch, které jsme se učili) je tomu nejblíže a proč?

```
Connection con = DriverManager.getConnection
    ("jdbc:myDriver:wombat","myLogin","myPassword");
Statement stmt = con.createStatement();
```

C2. Uživatel vašeho matematického systému chce ve svých programech psát zápisy typu `A + B` (sjednocení dvou množin), `A – B` (množinový rozdíl) a `A * B` (kartézský součin dvou množin). Navrhněte strukturu tříd implementující návrhový vzor interpretr, která by tyto operace byla schopná realizovat.

C3. V kódu níže je neoptimalita související s tím, že všechny třídy implementující rozhraní `Sort` nemají žádné instanční proměnné a jsou tudíž bezestavové. Zlepšete ho.

```
class Sorter {
    static void bubbleSort(int[] array) {
        /* kod */
    }

    static void quickSort(int[] array) {
        /* kod */
    }

    static void heapSort(int[] array) {
        /* kod */
    }

    Sort s = new BubbleSort();

    void setPreferredSortType(int type) {
        if (type == 1) s = new BubbleSort();
        if (type == 2) s = new QuickSort();
        if (type == 3) s = new HeapSort();
    }

    void sort(int[] array) {
        s.sort(array);
    }
}

interface Sort {
    void sort(int[] array);
}

class BubbleSort implements Sort {
    public void sort(int[] array) {
        Sorter.bubbleSort(array);
    }
}

class QuickSort implements Sort {
    public void sort(int[] array) {
        Sorter.quickSort(array);
    }
}

class HeapSort implements Sort {
    public void sort(int[] array) {
        Sorter.heapSort(array);
    }
}
```

C4. Třída `Triangle` implementuje návrhový vzor state se stavy pravoúhlý, ostroúhlý a tupoúhlý. Naimplementujte do této třídy podporu pro návrhový vzor visitor, která se bude rozhodovat podle stavu dané instance.

```
class Triangle {
    TriangleState s;

    void visit(TriangleVisitor v) { /* … */ }

    /* … */
}

interface TriangleVisitor {
    void visitRectangular(Triangle t);
    void visitObtuse(Triangle t);
    void visitAcute(Triangle t);
}
```

C5. Napište a vysvětlete (na příkladu) Deméterovo pravidlo.