E1. (příklad na spojové seznamy) Doprogramujte kód metody `extractSmallest` tak, aby ze spojového seznamu odkazovaného proměnnou `head` odstranila nejmenší prvek a vrátila jeho hodnotu. Dejte si pozor na krajní případy.

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

    static Node head;

    static int extractSmallest() {
        /* 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
        b.n = a;
        a.n.n = c;
        c.n = a;
        b.n = d;
        b.n.n = a;
    }
}
```

E3. Metoda `top` ve třídě `ExtendedStack` vypisuje logovací hlášky o vybrání a opětovném vložení prvku, ale to se vám nelíbí. Zařiďte to tak, aby žádnou hlášku nevypisovala, vyhněte se duplikaci kódu.

```
class Stack {
    /* atributy */

    /** Vlozi prvek. */
    void push(int e) { /* kod */ }

    /** Vyjme a vrati posledni vlozeny prvek. */
    int pop() { /* kod */ }

    /** Vrati hodnotu posledniho vlozeneho prvku
        (bez vyjmuti). */
    int top() {
        int result = pop();
        push(result);
        return result;
    }
}

class ExtendedStack extends Stack {
    void push(int e) {
        System.out.println("Prvek vlozen.");
        super.push(e);
    }

    int pop() {
        System.out.println("Prvek vybran.");
        return super.pop();
    }
}
```

E4. Rozhraní `Animal` nemá žádné metody, takže třídy `Dog` a `Cat` nemusí mít žádné společné metody. Je takové rozhraní k něčemu? Pokud ano, uveďte příklad.

```
interface Animal {}
class Dog implements Animal { /* ... */ }
class Cat implements Animal { /* ... */ }
```

E5. Je v následujícím kódu něco špatně (kromě chybějících těl metod)? Pokud ano, co?

```
interface Stack {
    void push(int e);
    int pop();
}

class StackImpl implements Stack {
    /* atributy */

    public void push(int e) { /* kod */ }
    public int pop() { /* kod */ }
    int top() { /* kod */ }

    static void f() {
        Stack s = new StackImpl();
        s.push(7);
        s.pop();
    }
}
```

D1. Třída `Rectangle` reprezentuje obdélník, třída `Square` čtverec. Napište tovární metodu `Shape getShape(int height, int width)`, která vrátí instanci vhodné třídy. Vhodné konstruktory si domyslete.

D2. Dopište implementaci třídy `Stack` z příkladu E3 (pole nebo spojový seznam), do metody `pop` dopište vhodný assert.

D3. Do třídy `Stack` z předchozího příkladu dopište podporu pro návrhový vzor iterátor, který postupně vrátí všechny prvky v daném zásobníku.

D4. Fronta se od zásobníku liší tím, že metoda `pop` vrací nejstarší prvek, zatímco u zásobníku ta samá metoda vrací prvek nejmladší. Naimplementujte třídu `Container`, která se chová jako fronta nebo jako zásobník podle toho, co si uživatel zvolí. Při implementaci využijte návrhový vzor state nebo strategy.

D5. Do třídy `Cell` dopište podporu pro návrhový vzor observer s událostí `contentsChanged`. Událost rozesílejte pouze pokud se opravdu hodnota proměnné contents změnila, tzn. ne pokud uživatel nastavuje hodnotu, která už v dané instanci byla.

```
class Cell {
    int contents;
    void set(int c) { contents = c; }
    int get() { return contents; }
}
```

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. Naimplementujte abstraktní továrnu: třídy továren by měly být `Zerg`, `Protoss` a `Terran`, jejich společné rozhraní `Race` s tovární metodou `Unit getUnit()`, statická metoda vracející abstraktní továrnu by měla být ve třídě `Game` a jmenovat se `getMyRace()`.

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

```
class MyInteger {
    int c;

    private MyInteger(int cc) { c = cc; }
    int getValue() { return c; }
}
```

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 {
    void f(D d);
}

class B implements A {
    public void f(D d) {
        d.g(this);
    }
}

class C implements A {
    public void f(D d) {
        d.h(this);
    }
}

interface D {
    void g(B b);
    void h(C c);
}
```