E1. (příklad na spojové seznamy) Přepište třídu `Set` tak, aby se chovala stále stejně, avšak místo pole používala spojový seznam (neomezené délky). Spojový seznam naprogramujte od začátku, tzn. nesmíte použít třídy typu `LinkedList` apod.

```
/**
 * Implementace mnoziny, tzn. kontejneru prvku
 * bez vzajemneho poradi a bez opakovani.
 */
class Set {
    int[] contents = new int[1000];
    int size = 0;

    void add(int e) {
        if (!contains(e))
            contents[size++] = e;
    }

    boolean contains(int e) {
        return indexOf(e) != -1;
    }

    int indexOf(int e) {
        for (int i = 0; i < size; i++)
            if (contents[i] == e)
                return i;
        return -1;
    }

    void remove(int e) {
        int i = indexOf(e);
        if (i == -1)
            return;
        contents[i] = contents[--size];
    }
}
```

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

```
class Node {
    int contents;
    Node left, right;

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

    void add(Node n) {
        if (n.contents < contents)
            if (left == null)
                left = n;
            else
                left.add(n);
        else
            if (right == null)
                right = n;
            else
                right.add(n);
    }

    static void f() {
        Node n = new Node(5);
        Node t = new Node(3);
        n.add(t);
        t = new Node(1);
        n.add(t);
        t = new Node(4);
        n.add(t);
        t = new Node(7);
        n.add(t);
    }
}
```

E3. Jedna z metod třídy `Set` z příkladu E1 by měla být privátní. Určete která a napište proč. Na zdůvodnění si dejte záležet, o uznatelnosti příkladu se bude rozhodovat především podle něj.

E4. Změňte metodu `plus` na statickou, zachovejte funkčnost.

```
class Complex {
    int re, im;

    Complex(int r, int i) {
        re = r;
        im = i;
    }

    Complex plus(Complex c) {
        return new Complex(re + c.re, im + c.im);
    }
}
```

E5. Na třídě `Ir` není něco v pořádku. Co?

```
class Kim {
    int contents;

    /* Nastavi hodnotu */
    public void set(int c) {
          contents = c;
    }
    /* Vrati hodnotu */
    public int get() {
          return contents;
    }
}

class Ir extends Kim {
    /* Nastavi hodnotu pouze pokud je parametr zaporny */
    public void set(int c) {
          if ( c < 0) super.set( c);
    }
}
```

V následujících příkladech se budeme odvolávat na interface `Number` a třídy `Complex` a `Real`. Zde je jejich kód.

```
interface Number {
    int getReal();
    int getImaginary();
}

class Real implements Number {
    int value;

    Real(int v) {
        value = v;
    }

    public int getReal() {
        return value;
    }

    public int getImaginary() {
        return 0;
    }
}

class Complex implements Number {
    int re, im;

    Complex(int r, int i) {
        re = r;
        im = i;
    }

    public int getReal() {
        return re;
    }

    public int getImaginary() {
        return im;
    }
}
```

D1. Naprogramujte tovární metodu `static Number getNumber(int re, int im)`, která bude podle zadaných parametrů vracet instance tříd `Real` nebo `Complex`.

D2. Napište třídu `Num`, která bude pro svoji implementaci používat návrhový vzor stav se stavy reálné číslo a imaginární číslo (reprezentované třídami `Real` a `Complex`).

```
class Num {
    int getReal() { /* ... */ }
    int getImaginary(){ /* ... */ }
    void setReal(int r) { /* ... */ }
    void setImaginary(int i) { /* ... */ }
}
```

D3. Napište příklad překrytí metody (overriding) a přetížení metody (overloading). V čem je rozdíl?

D4. Naprogramujte třídu `Fraction` reprezentující zlomek. Použijte návrhový vzor kompozit tak, aby v čitateli i jmenovateli zlomku mohla být jak reálná, tak komplexní třída.

D5. Naprogramujte tovární metodu `static Number getNumber(int re, int im)`, která bude pro reprezentaci stejných čísel vracet stejné objekty (tzn. například volání `getNumber(1, 2)` pokaždé vrátí ukazatel na stejnou instanci). Pokud chcete, můžete použít `HashMap` nebo jinou kolekci.

C1. Do kódu níže dopište podporu pro visitora a visitora, který dané dvě instance sečte.

```
interface Number {
    int getReal();
    int getImaginary();
}

class Real implements Number {
    int value;

    Real(int v) {
        value = v;
    }

    public int getReal() {
        return value;
    }

    public int getImaginary() {
        return 0;
    }
}

class Complex implements Number {
    int re, im;

    Complex(int r, int i) {
        re = r;
        im = i;
    }

    public int getReal() {
        return re;
    }

    public int getImaginary() {
        return im;
    }
}
```

C2. Třídy `Vladimir` a `Iljic` implementují interface `Lenin`. Napište tovární metodu, která pomocí abstract factory při prvních pěti voláních vrátí novou instanci třídy `Vladimir` a pak už jen instance třídy `Iljic`.

C3. Předpokládejte, že třída `X` má tovární metodu a dopište do ní iterátor, který postupně vrátí všechny její vytvořené instance.

C4. Upravte následující kód tak, aby striktně dodržoval Deméterovo pravidlo. Hlavičku metody `f` upravovat nesmíte.

```
class Vaclav {
    void f(Y y) {
        y.getZ().setA(7);
    }
}

class Premysl {
    Z z;
    Z getZ() { return z; }
}

class Karel {
    int a;
    void setA(int a) { this.a = a; }
}
```

C5. Předpokládejte, že třída `X` má tovární metodu a dopište do ní podporu pro observer, který rozešle události při vytvoření nové instance.