

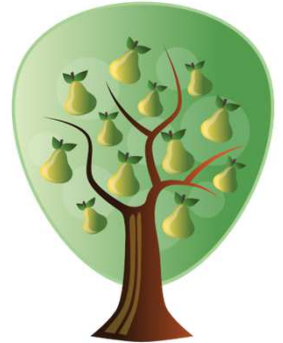
Front of  
Queue



Rear (end)  
of Queue

# Parametrický polymorfizmus

(na lineárnych dátových štruktúrach)



Peter Borovanský  
KAI, I-18

borovan 'at' ii.fmph.uniba.sk

<http://dai.fmph.uniba.sk/courses/JAVA/>

# Copy-paste je smrt' programátora

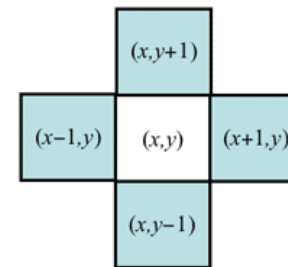
9/10 otevřených kódů  
Voltage malo tento  
copy-paste pattern

```
if (j > 0 && k > 0 && n[j - 1][k - 1] % 10 != 0) {
    n[j - 1][k - 1] = (n[j - 1][k - 1] + 1);
    if (n[j - 1][k - 1] % 10 == 0) {
        countDracula++;
        zeroCount++;
    }
}
if (j > 0 && n[j - 1][k] % 10 != 0) {
    n[j - 1][k] = (n[j - 1][k] + 1);
    if (n[j - 1][k] % 10 == 0) {
        countDracula++;
        zeroCount++;
    }
}
if (j > 0 && k < n[0].length - 1 && n[j - 1][k + 1] % 10 != 0) {
    n[j - 1][k + 1] = (n[j - 1][k + 1] + 1);
    if (n[j - 1][k + 1] % 10 == 0) {
        countDracula++;
        zeroCount++;
    }
}
if (k < n[0].length - 1 && n[j][k + 1] % 10 != 0) {
    n[j][k + 1] = (n[j][k + 1] + 1);
    if (n[j][k + 1] % 10 == 0) {
        countDracula++;
        zeroCount++;
    }
}
if (j < n.length - 1 && k < n[0].length - 1 && n[j + 1][k + 1] % 10 != 0) {
    n[j + 1][k + 1] = (n[j + 1][k + 1] + 1);
    if (n[j + 1][k + 1] % 10 == 0) {
        countDracula++;
        zeroCount++;
    }
}
if (j < n.length - 1 && n[j + 1][k] % 10 != 0) {
    n[j + 1][k] = (n[j + 1][k] + 1);
    if (n[j + 1][k] % 10 == 0) {
        countDracula++;
        zeroCount++;
    }
}
if (j < n.length - 1 && k > 0 && n[j + 1][k - 1] % 10 != 0) {
    n[j + 1][k - 1] = (n[j + 1][k - 1] + 1);
    if (n[j + 1][k - 1] % 10 == 0) {
        countDracula++;
        zeroCount++;
    }
}
if (k > 0 && n[j][k - 1] % 10 != 0) {
    n[j][k - 1] = (n[j][k - 1] + 1);
    if (n[j][k - 1] % 10 == 0) {
        countDracula++;
        zeroCount++;
    }
}
```

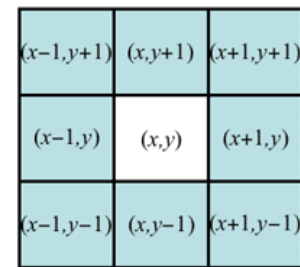
```
if (voltages[i - 1][j] != 0) {
    cellsToUpdate.add((i - 1) + "," + j);
}
if (j != 0) {
    if (voltages[i - 1][j - 1] != 0) {
        cellsToUpdate.add((i - 1) + "," + (j - 1));
    }
}
if (j != voltages[i].length - 1) {
    if (voltages[i - 1][j + 1] != 0) {
        cellsToUpdate.add((i - 1) + "," + (j + 1));
    }
}
if (i != voltages.length - 1) {
    if (voltages[i + 1][j] != 0) {
        cellsToUpdate.add((i + 1) + "," + j);
    }
}
if (j != 0) {
    if (voltages[i + 1][j - 1] != 0) {
        cellsToUpdate.add((i + 1) + "," + (j - 1));
    }
}
if (j != voltages[i].length - 1) {
    if (voltages[i + 1][j + 1] != 0) {
        cellsToUpdate.add((i + 1) + "," + (j + 1));
    }
}
if (j != 0) {
    if (voltages[i][j - 1] != 0) {
        cellsToUpdate.add(i + "," + (j - 1));
    }
}
if (j != voltages[i].length - 1) {
    if (voltages[i][j + 1] != 0) {
        cellsToUpdate.add(i + "," + (j + 1));
    }
}
```

```
if (voltages[i - 1][j] != 0) {
    cellsToUpdate.add((i - 1) + "," + j);
}
if (j != 0) {
    if (voltages[i - 1][j - 1] != 0) {
        cellsToUpdate.add((i - 1) + "," + (j - 1));
    }
}
if (j != voltages[i].length - 1) {
    if (voltages[i - 1][j + 1] != 0) {
        cellsToUpdate.add((i - 1) + "," + (j + 1));
    }
}
if (i != voltages.length - 1) {
    if (voltages[i + 1][j] != 0) {
        cellsToUpdate.add((i + 1) + "," + j);
    }
}
if (j != 0) {
    if (voltages[i + 1][j - 1] != 0) {
        cellsToUpdate.add((i + 1) + "," + (j - 1));
    }
}
if (j != voltages[i].length - 1) {
    if (voltages[i + 1][j + 1] != 0) {
        cellsToUpdate.add((i + 1) + "," + (j + 1));
    }
}
if (j != 0) {
    if (voltages[i][j - 1] != 0) {
        cellsToUpdate.add(i + "," + (j - 1));
    }
}
if (j != voltages[i].length - 1) {
    if (voltages[i][j + 1] != 0) {
        cellsToUpdate.add(i + "," + (j + 1));
    }
}
```

# Skôr ako dámy DÚ3



4-neighbourhood



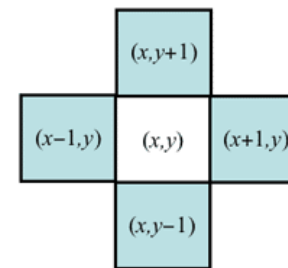
8-neighbourhood

Ak stojím na políčku  $[x, y]$  a potrebujem niečo riešiť v 4/8 smeroch  
... dva vnorené mini-cykly

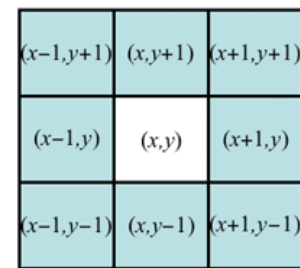
```
for (var dx = -1; dx <= 1; dx++) { // -1, 0 1
    for (var dy = -1; dy <= 1; dy++) { // -1, 0 1
        if (dx == 0 && dy == 0) continue; // 8 smerov // 4 smery
                                           if (Math.abs(dx) == Math.abs(dy)) continue;

        var nx = x + dx;
        var ny = y + dy;
        if (0 <= nx && nx < pole.length &&
            0 <= ny && ny < pole[nx].length) {
            //.. a riešim [nx, ny]
            System.out.println(nx + "," + ny);
        }
    }
}
```

# Skôr ako dámy DÚ3



4-neighbourhood

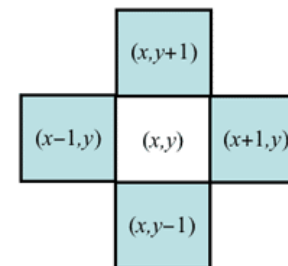


8-neighbourhood

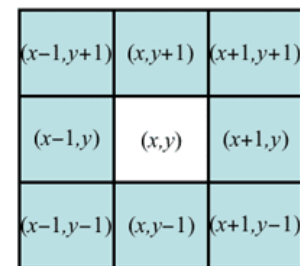
Ak stojím na políčku  $[x,y]$  a potrebujem niečo riešiť v 4/8 smeroch ... smery mám v dátovej štruktúre, klúadne aj List, ...

```
int[][] directions = {{-1,-1}, {-1,0}, {-1,1},  
                      {0,-1},      {0,1},  
                      {1,-1}, {1,0},  {1,1}};  
for (var dir : directions) {  
    var dx = dir[0];  
    var dy = dir[1];  
    var nx = x+dx;  
    var ny = y+dy;  
    if (0 <= nx && nx < pole.length &&  
        0 <= ny && ny < pole[nx].length) {  
        //.. a riesim [nx, ny]  
        System.out.println(nx+ "," + ny);  
    }  
}
```

# Skôr ako dámy DÚ3



4-neighbourhood

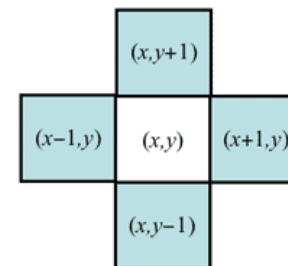


8-neighbourhood

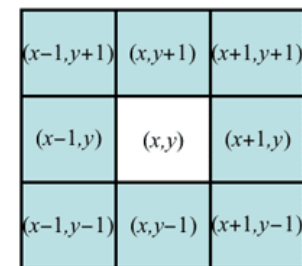
Ak stojím na políčku [x,y] a potrebujem niečo riešiť v 4/8 smeroch  
... počul som o výnimkách

```
int[][] directions = {{-1,-1}, {-1,0}, {-1,1},  
                      {0,-1},      {0,1},  
                      {1,-1}, {1,0}, {1,1}};  
for (var dir : directions) {  
    var dx = dir[0];  
    var dy = dir[1];  
    var nx = x+dx;  
    var ny = y+dy;  
    try {  
        //.. a riesim pole[nx, ny]  
        System.out.println(pole[nx][ny]);  
    } catch (IndexOutOfBoundsException e) {  
        // indexoval som mimo  
    }  
}
```

# Skôr ako dámy DÚ3



4-neighbourhood



8-neighbourhood

Ak stojím na políčku  $[x,y]$  a potrebujem niečo riešiť v 4/8 smeroch  
... a náhodou hľadám niečo ako *piškvorku* dĺžky 5

```
int[][] directions = {{-1,-1}, {-1,0}, {-1,1}, {0,-1}, {0,1}, {1,-1}, {1,0}, {1,1}};
for (var dir : directions) {
    var dx = dir[0];
    var dy = dir[1];
    for (int k = 0; k < 5; k++) {
        var nx = x + k * dx; // rovnica priamky
        var ny = y + k * dy;
        try {
            //.. a riesim pole[nx, ny]
            System.out.println(pole[nx][ny]);
        } catch (IndexOutOfBoundsException e) {
            // indexoval som mimo
        }
    }
}
```



# Triedy a objekty

---

dnes bude:

- trieda **Object**,
- klonovanie a boxovanie,
- generics (generické typy) - parametrický polymorfizmus,
- **interface** a **implementation**,
- výnimky na príkladoch, **throw(s)**, **try catch (Exception)**,
- príklady lineárnych dátových štruktúr
  - interface pre stack, front, balík, ...
  - implementácie: polia, jednoduché a obojsmerné spájané zoznamy
- príklady stromových dátových štruktúr

cvičenia:

- interface a implementation pre ADT (prioritný front)
- parametrické typy

literatúra:

- <http://docs.oracle.com/javase/tutorial/java/generics/index.html>,
- <https://docs.oracle.com/javase/tutorial/extra/generics/simple.html>,



# Prvý Stack

(motivačný príklad)

- vytvoríme zásobník ako triedu Stack
- implementuje operácie push, pop, ...
- s obmedzeniami:
  - na maximálnu veľkosť zásobníka,
  - typ prvkov v zásobníku,
  - neošetrené chybové stavy

```
public class Stack {  
    protected int[] S;  
    protected int top = -1;  
  
    public Stack(int size) {  
        S = new int[size];  
    }  
    public boolean isEmpty() {  
        return top < 0;  
    }  
    public void push(int element) {  
        if (top+1 == S.length) // test, kedy už nemôžeme pridať prvok  
            System.err.println("Stack is full"); // vypíš chybu  
        else // ak môžeme  
            S[++top] = element; // tak pridáme  
    }  
    // reprezentácia ako pole int  
    // vrchol zásobníka, index vrchného prvku  
    // konštruktor vytvorí pole int[] veľkosti  
    // size  
    // test, či zásobník neobsahuje prvky
```



# Prvý Stack – pokračovanie



```
public int pop() {  
    int element;  
    if (isEmpty()) {  
        System.err.println("Stack is empty"); // vypíš chybu  
        return -1; // nevieme čo vrátiť, tak "čokoľvek":int  
    }  
    element = S[top--];  
    return element;  
}
```

```
public class StackMain {  
    public static void main(String[] args) {  
        final int SSIZE = 100;  
        Stack s = new Stack(SSIZE);  
        for(int i=0; i<SSIZE; i++)  
            s.push(i);  
        while (!(s.isEmpty()))  
            System.out.println(s.pop());  
    }  
}
```

99  
98  
...  
6  
5  
4  
3  
2  
1  
0



# Čo s obmedzeniami

---

Zamyslenie nad predchádzajúcim príkladom:

- fixná veľkosť poľa pre reprezentáciu zásobníka
  - dynamická realokácia,
  - na budúce prídu java-hotové štruktúry: Vector, ArrayList, ...
  - použiť štruktúru, ktorej to nevadí (napr. spájané zoznamy),
- typ prvkov je obmedzený (na int) v implementácii (ako sa rozumne vyhnúť kopírovaniu kódu, ak potrebujeme zásobníky double, String, alebo užívateľom definované typy Ratio, Complex, ...):
  - nájsť „matku všetkých typov“ (trieda Object),
  - zaviesť parametrické typy – parametrický polymorfizmus (generics),
- chybové stavy
  - chybové hlášky a „hausnumerické“ výstupné hodnoty,
  - **System.err.print**
  - výnimky (definícia výnimky, vytvorenie a odchytenie výnimky)



# Trieda Object

- class Object je nadtrieda všetkých tried
- vytvoríme heterogénny zásobník pre elementy ľubovoľného typu
- implementácia v poli,
- realokácia pri pretečení

```
public class StackObj {  
    protected Object[] S;           // reprezentácia ako pole Object-ov  
    protected int top;              // vrchol  
  
    public StackObj (int Size) {     // konštruktor naalokuje pole Object-ov  
        S = new Object[Size];       // požadovanej veľkosti  
        top = 0;  
    }  
    public boolean isEmpty () {  
        return top == 0;  
    }  
    public void push (Object item) { // push netestuje pretečenie ☹  
        S[top++] = item;  
    }  
    public Object pop () {           // ani pop netestuje podtečenie ☹  
        return S[--top];  
    }  
}
```

# Pretečenie poľa realokácia

- implementácia v poli, čo „puchne“
- ak sa pokúsime pretypovať hodnotu z typu Object na iný (napr. String), môžeme dostať **runtime cast exception**

```
public void push (Object item) {  
    if (top == S.length) {                                // problém pretečenia  
        Object[] newS = new Object[S.length * 2];        // naalokuj pole 2*väčšie  
        for (int i=0; i<S.length; i++) newS[i] = S[i];    // presyp  
        S = newS;                                          // poves miesto starého poľa  
    }  
    S[top++] = item;                                     // a konečne pridaj prvok
```

```
StackObj pd = new StackObj(SSIZE);  
pd.push(new Integer(123456)); // heterogénny stack  
pd.push("ahoj");              // zoženie Integer aj String  
String str = (String)pd.pop(); System.out.println(str);  
Integer num = (Integer)pd.pop(); System.out.println(num);  
ak posledné dva riadky vymeníme, runtime cast exception,  
lebo "ahoj" nie je Integer ani 123456 nie je String
```

ahoj  
123456

- takto sa programovalo do verzie 1.4
- potom prišli generics - templates(C++)  
a parametrické dátové typy

# Trieda Object

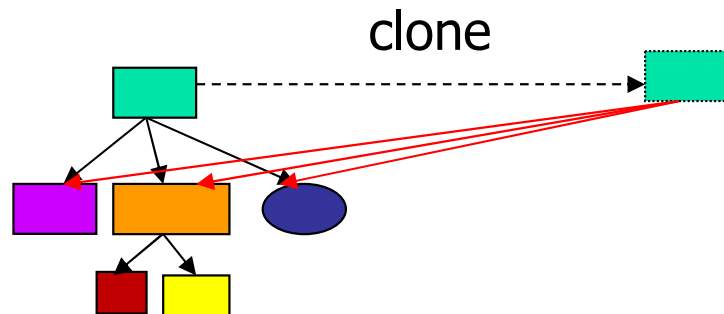
nadtrieda všetkých tried, ale inak normálna trieda, napr.

```
Object[] S = new Object[Size];
```

- pretypovanie **do triedy Object**  
ak  $x : E$ , potom  $(Object)x : Object$  – explicitne,  
resp.  $x : Object$  – implicitne
- pretypovanie **z triedy Object**  
ak  $o : Object$  a hodnotou je objekt triedy  $E$ , potom  $(E)o : E$   
explicitný *cast* predstavuje typovú kontrolu v runtime,  
napr.  $(Integer)o : Integer$ ,  $(String)o : String$
- ak však hodnota objektu  $o$  nie je triedy  $E$ , potom runtime check  
 $(E)o$  zlyhá (cast exception)
- ak chcete byť opatrný, tak sa spýtajte  $x$  **instanceof**  $E$
- primitívne typy (int, double, boolean, ...) **boxujeme** do skutočných  
tried (Integer, Double, Boolean, ...)

# Čo vie každý Object

- **String toString()** - textová reprezentácia,
- **int hashCode()** - pretransformuje referenciu na objekt na int, vráti,
- **void finalize()** - deštruktor volá garbage collector,
- **Class getClass()** – vráti Class objekt (triedy Class),
- **Object clone()** – vytvorí **nerekurzívnu** kópiu objektu, ak objekt je z klonovateľnej triedy (Cloneable), inak CloneNotSupportedException.  
Polia, Integer, String sú klonovateľné. Nerekurzívna (shallow):



- **boolean equals(Object obj)** – porovná referencie na objekty,

$x.clone() \neq x$

$x.clone().getClass() == x.getClass()$

# Clone v príkladoch



```
public class Pole3D {
    private Hruska[][][] mojePole;
    public Pole3D(Hruska[][][] tvojePole) {
        mojePole = tvojePole;
        mojePole = tvojePole.clone();
        for (int i = 0; i < tvojePole.length; i++) {
            mojePole[i] = tvojePole[i].clone();
            for (int j = 0; j < tvojePole[i].length; j++) {
                mojePole[i][j] = tvojePole[i][j].clone();
                for (int k = 0; k < tvojePole[i][j].length; k++) {
                    mojePole[i][j][k] =
                        (Hruska)tvojePole[i][j][k].clone();
                }
            }
        }
    }
}

public class Hruska implements Cloneable {
    @Override
    protected Object clone() { ... }
}
```

# Generics

Properties for 05\_java

type filter text

- Resource
- Builders
- Java Build Path
- Java Code Style
- Java Compiler
- Java Editor
- Javadoc Location
- Project References
- Refactoring History
- Run/Debug Settings

Java Compiler

☒ Enable project specific settings

[Configure Workspace Settings...](#)

JDK Compliance

☐ Use compliance from execution environment on the Java Build Path

Compiler compliance level:

1.6  
1.3  
1.4  
1.5  
1.6

Generated .class files compatibility:

Source compatibility:

```
public class Stack50<E> {
```

```
    protected E[] S;
```

```
    protected int top;
```

```
    public Stack50(int Size) {
```

```
        S = (E[]) new Object[Size];
```

```
        top = 0;
```

```
    }
```

```
    public boolean isEmpty() {
```

```
        return top == 0;
```

```
    }
```

```
    public void push(E item) {
```

```
        S[top++] = item;
```

```
    }
```

```
    public E pop() {
```

```
        return S[--top];
```

```
    }
```

```
}
```

Java život pred JDK 5 bol Python škaredý:

- miesto `ArrayList<String>`
- sa písalo `ArrayList`
- bola to kolekcia plná `Objects`
- zo staticky typovaného jazyka to robilo dynamicky typovaný ☹

použitím typovej premennej sa z definície triedy, metódy, ... stáva šablóna, do ktorej skutočný typ musíme dosadiť

Súbor: [Stack50.java](#)





# Stack50

- hlavným rozdielom je, že Stack50 je homogénny, všetky prvky sú tohoistého typu
- ak však naozaj treba miešať typy, Stack50<Object> je to, čo sme mali

```
public class Stack50<E> {  
    protected E[] S;  
    protected int top;
```

```
    public Stack50(int Size) {  
        S = (E[]) new Object[Size];  
        // toto nejde: S = new E[Size]; // kvôli typovej bezpečnosti  
        top = 0;  
    }
```

```
        Stack50<String> st50 =                // E = String  
            new Stack50<String>(SSIZE);  
        st50.push("caf");  
        st50.push("hello");  
        st50.push("salut");  
        // st50.push(new Integer(12345)); // String != Integer  
        System.out.println(st50.pop());
```



# Boxovanie

Mnohé rozhodnutia prijaté  
v jazyku Java sú poplatné tomu,  
byť as fast as possible,  
konkurentom doby bol jazyk C++

V Java (na rozdiel od napr. C#) nemožno vytvoriť generický typ  
parametrizovaný primitívnym typom:

Stack50<**int**> je **ilegálny typ**

miesto toho treba:

Stack50<Integer> je legálny typ

---

**Primitívne typy:** byte, short, int, long, float, double, ...

**Referenčný typ:** trieda

**Boxovanie typov:** int->Integer, float->Float, double->Double,...

```
int bb = 5;                // primitivny typ, modifikovateľný  
Integer cc = new Integer(15); // trieda/objekt, nemodifikovateľný
```

```
bb = cc;                   // bb = cc.intValue();  
cc = bb;                   // cc = new Integer(bb);
```

# Kovariancia a polia

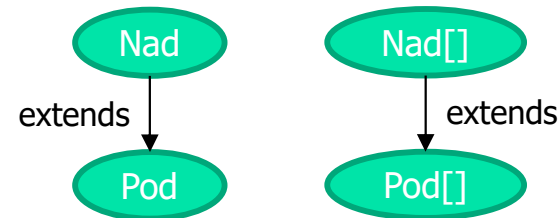
- generics sa realizujú v kompilátore, výsledný byte kód je negenerický,
- generics nie je makro, ktoré sa expanduje (ako templates v C++),
- **kovariancia** znamená, že ak T1 je podtrieda T2, tak  $\psi(T1)$  je podtrieda  $\psi(T2)$
- logicky... , polia sú kovariantné, t.j. T1[] je podtriedou T2[], príklad:

z predošlého slajdu:

E[] je podtrieda Object[], lebo E je podtrieda Object

iný príklad

nech Podtrieda je podtriedou Nadtrieda:



```
Podtrieda[] a = { new Podtrieda(), new Podtrieda()};
```

```
Nadtrieda[] b = a; // kovariancia polí, lebo Podtrieda[] podtrieda Nadtrieda[]
```

```
// Podtrieda[] c = b; nejde, lebo neplatí Nadtrieda[] podtrieda Podtrieda[]
```

# Nekovariancia generických typov

- na prvý pohľad nelogický, ale **generické typy nie sú kovariantné**,  
napr. `Stack50<T1>` **NIE JE** podtriedou `Stack50<T2>`, ak `T1` je podtrieda `T2`.

Ak by to tak bolo (kontrapríklad nabúra typovú bezpečnosť):

```
Stack50<Podtrieda> stA = new Stack50<Podtrieda>(100);
```

```
stA.push(new Podtrieda());
```

```
Stack50<Nadtrieda> stB = stA;
```

```
// ak by to tak bolo, tak toto by išlo
```

```
// ale ono to v skutočnosti nejde...
```

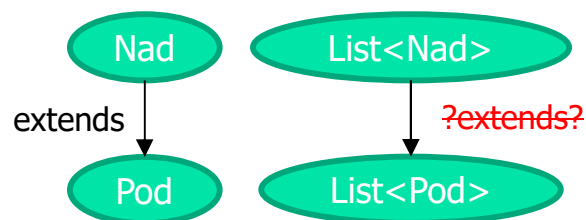
- dôvod (nabúrame typovú kontrolu):  
`stB.push(new Nadtrieda());`

```
// ak by sme to dopustili, potom
```

```
// je korektný výraz, ktorý pomieša
```

```
// objekty Podtriedy a Nadtriedy v stB
```

```
// Stack50 už nie je homogénny
```





# Dôsledky kovariancie

keďže polia sú kovariantné, generics nie, potom **nie je možné vytvoriť pole prvkov generického typu**, napríklad:

```
// S = new E[Size];           // vid' konštruktor Stack50  
alebo                         // je síce korektná deklarácia  
Stack50<Integer>[] p;         // ale nekorektná alokácia  
// p = new Stack50<Integer>[5]; // cannot create generic array
```

---

- dôvod (ak by to išlo, takto nabúrame typovú kontrolu):

```
Object[] pObj = p;           // Stack50<Integer> je podtrieda Object, preto  
                             // z kovariancie Stack50<Integer>[] je podtrieda Object[]  
                             // vytvoríme malý stack Stack50<String>  
Stack50<String> stS = new Stack50<String>(100);  
stS.push("bude problem"); // s elementmi typu String
```

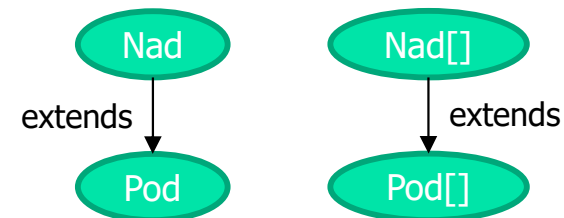
```
pObj[0] = stS;  
Integer i = pObj[0].pop();
```

```
// pObj[0]:Object, stS: Stack50<String>  
// "bude problem" nie je typu Integer,  
// lebo pObj[0] nie je Stack50<Integer> ale  
// Stack50<String>
```

# Kovariancia „útočila“

```
{ // príklad z prednášky
  Stack50<Podtrieda> stA = new Stack50<Podtrieda>();
  stA.push(new Podtrieda());
  Stack50<Nadtrieda> stB = stA; // ak by to tak bolo, tak toto by išlo
                                // ale ono to v skutočnosti nejde...
                                // dôvod (nabúrame typovú kontrolu)
  //stB.push(new Nadtrieda()); // ak by sme to dopustili, potom
}
```

```
{ // otázka študenta : skúsme to s poliami, ktoré sú kovariantné
  Podtrieda[] stA = new Podtrieda[]{ new Podtrieda(), null };
  Nadtrieda[] stB = stA;
  stB[1] = new Nadtrieda();
  System.out.println(stA[1]);
}
```



- kód je skompilovateľný, statická typová kontrola nenájde chybu ☹
- ale počas behu nastane `java.lang.ArrayStoreException: Nadtrieda`
- aspoň že typová homogénnosť poľa je zachovaná/uchránená ☺



# Generické generického ?

---

- `Stack50<Stack50<Integer>>`
  - ide, ale kto potrebuje stack stackov ... ?
- `ArrayList<ArrayList<Integer>>, HashMap<String, HashSet<String>>`
  - potrebujem úplne bežne, a to ide ☺
- v cvičení budete implementovať **prioritný front** s hodnotami typu `E`
  - ak bude predpísaná implementácia poľom `[]`, tak by to chcelo
  - ```
private class Elem<E> implements Comparable<Elem<E>> { // dvojica elem:E, priorita:int
    public E elem;
    public int prio;
    ...
}
```
  - lenže ako dôsledok predchádzajúceho sa nepodarí vytvoriť pole klasické dvojíc:  
`Elem<E>[] front`
  - ale bez problémov sa podarí vytvoriť `ArrayList<Elem<E>> front`
  - riešenie, ktoré máte objaviť (ak zadanie zakazuje použiť kolekcie)
  - `E[] hodnoty = (E[])new Object[size];`
  - `int[] priority = new int[size];`

# Generické metódy

(v negenerickej triede)

Nie len celá definícia triedy (ADT) môže byť parametrizovaná typom, ale aj jednotlivá metóda či konštruktor v neparametrickej triede.

```
public static <T> String genMethod(T value) {  
    System.out.println(value.getClass());  
    return value.toString();  
}
```

```
public static <E> void printArray(E[] p) {  
    for ( E elem : p )  
        System.out.print( elem + ",");  
    System.out.println();  
}
```

```
System.out.println(genMethod(1));  
System.out.println(genMethod("wow"));  
Integer[] p = {1,2,3}; System.out.println(genMethod(p)); printArray(p);  
Double[] r = {1.1,2.2,3.3}; System.out.println(genMethod(r)); printArray(r);
```

```
class java.lang.Integer  
1  
class java.lang.String  
wow  
class [Ljava.lang.Integer;  
[Ljava.lang.Integer;@42e..  
1,2,3,  
class [Ljava.lang.Double;  
[Ljava.lang.Double;@930..  
1.1,2.2,3.3,
```



# Generické metódy

(v negenerickej triede)

Použitie generického typu môže byť ohrozené kvalifikátormi na typový parameter, napr. metóda `genMethod2` sa dá použiť len pre číselné typy, t.j. typy podedené od typu/triedy `Number` (čo sú `BigDecimal`, `BigInteger`, `Byte`, `Double`, `Float`, `Integer`, `Long`, `Short`)

```
public static <T extends Number> T genMethod2(T value) {  
    System.out.println(value.getClass());  
    return value;  
}
```

```
System.out.println(genMethod2(1));  
//System.out.println(genMethod2("wow"));  
System.out.println(genMethod2(Math.PI));
```

```
class java.lang.Integer  
1  
class java.lang.Double  
3.141592653589793
```

Iný príklad: Binárny vyhľadávací strom má zmysel, len ak vieme porovnávať hodnoty prvkov vo vrcholoch

```
public class BVSTree<E extends Comparable<E>> { ...
```



# Generické metódy

(v negenerickej triede)

```
static <T> T[] append(T[] arr, T element) {  
    final int N = arr.length;  
    arr = Arrays.copyOf(arr, N+1); // N+N  
    arr[N] = element;  
    return arr;  
}
```

1  
2  
3  
4  
5  
6  
...

1.048.576 = N =  $2^{20}$

---

550 mld =  $N(N+1)/2$

1  
2  
4  
8  
16  
32  
...

1.048.576 = N =  $2^{20}$

---

2 ml =  $2^{21}-1$



# Pole3D

aj statická metóda môže byť generická

---

```
public class Pole3D__ {  
  
    // generická statická metóda  
    public static <T> boolean obeNull(T[] a, T[] b) {  
        return a == null && b == null;  
    }  
  
    // generická statická metóda  
    public static <T> boolean roznePolia(T[] a, T[] b) {  
        if (a == null && b == null) return false;  
        if (a == null && b != null) return true;  
        if (a != null && b == null) return true;  
        return a.length != b.length;  
    }  
}
```

# Interface pre Stack

Definícia interface predpisuje metódy, ktoré implementátor musí zrealizovať

```
public interface StackInterface<E> {  
    public int size();  
    public boolean isEmpty();  
    public E top() throws EmptyStackException;  
    public void push (E element) throws FullStackException;  
    public E pop() throws EmptyStackException;  
}
```

```
public class EmptyStackException extends RuntimeException {  
    public EmptyStackException(String err) {  
        super(err);  
    }  
}
```

```
public class FullStackException extends RuntimeException {  
    public FullStackException(String err) {  
        super(err);  
    }  
}
```



# Implementation - ArrayStack

Implementujeme poľom parametrický zásobník s výnimkami:

```
public class ArrayStack<E> implements StackInterface<E> {  
    protected int capacity;  
    protected E S[];                // reprezentácia  
    protected int top = -1;  
  
    public ArrayStack(int cap) {      // konštruktor pre Stack danej veľkosti  
        capacity = cap;  
        S = (E[]) new Object[capacity];  
    }  
  
    public void push(E element) throws FullStackException {  
        if (size() == capacity)      // ak už nemôžem pridať  
            throw new FullStackException("Stack is full."); // hodím výnimku  
        S[++top] = element;          // inak pridám  
    }  
}
```



# ArrayStack - pokračovanie

```
public E top() throws EmptyStackException {  
    if (isEmpty()) // ak je prázdny  
        throw new EmptyStackException("Stack is empty."); // výnimka  
    return S[top];  
}  
public E pop() throws EmptyStackException {  
    E element;  
    if (isEmpty()) // ak niet čo vybrať  
        throw new EmptyStackException("Stack is empty."); // výnimka  
    element = S[top];  
    S[top--] = null; // odviazanie objektu S[top] pre garbage collector  
    return element;  
}
```

```
ArrayStack<String> B = new ArrayStack<String>();  
B.push("Boris");  
B.push("Alenka");  
System.out.println((String)B.pop());  
B.push("Elena");  
System.out.println((String)B.pop());
```

# Vagóniková implementácia



- implementácia pomocou poľa nie je jediná, a má niektoré nedostatky
- chceme implementovať zásobník ako spájaný zoznam
- v C++/Pascale sme na to potrebovali pointer

```
typedef struct node {
```

```
    int element;
```

```
    node *next;
```

// pointer na nasledujúci vagónik zásobníka

```
};
```

- v Java

```
public class Node {
```

```
    private int element;
```

```
    private Node next;
```

// referencia na nasledujúci vagónik zásobníka

```
}
```

Iná implementácia, pomocou  
pospájaných krabíc typu Node

# Spájaný zoznam - Node

```
public class Node<E> {  
    private E element;           // reprezentácia krabice  
    private Node<E> next;
```

```
    public Node() { this(null, null); }
```

```
    public Node(E e, Node<E> n) { // konštruktor krabice typu Node  
        element = e;  
        next = n;  
    }
```

```
    // enkapsulacia: getter a setter
```

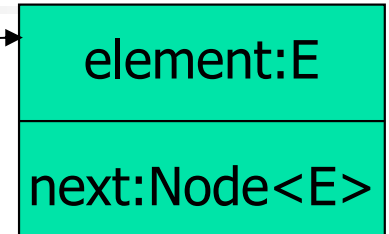
```
    public E getElement() {  
        return element;  
    }
```

```
    public Node<E> getNext() {  
        return next;  
    }
```

```
}
```

```
    public void setElement(E newElem) {  
        element = newElem;  
    }
```

```
    public void setNext(Node<E> newNext) {  
        next = newNext;  
    }
```





# Hádanka na zamyslenie

veľmi krátkodobá prémia

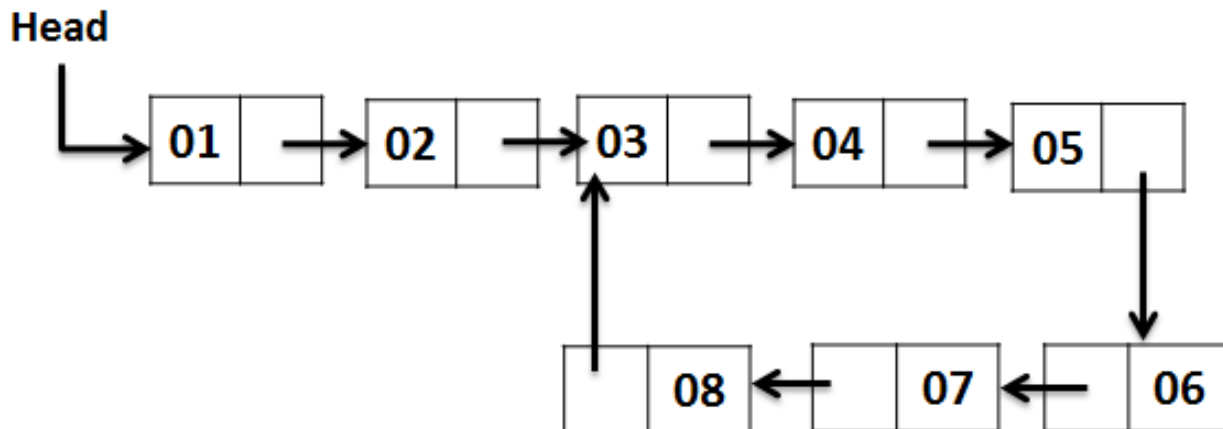
máte štruktúru definovanú na predchádzajúcom slajde

```
public class Node<E> {  
    private E element;    // reprezentácia krabice  
    private Node<E> next; // obsah krabice  
}
```

viete zistiť, či zoznam končí null, alebo je zacyklený ? Definujte

```
public boolean infinite() { ... true/false }
```

- pozor: testovacie vstupy budú niekoľko miliónov krabíc dlhé...





# NodeStack - implementation

```
public class NodeStack<E> implements StackInterface<E> {  
    protected Node<E> top; // reprezentácia triedy NodeStack  
    protected int size;      // ako pointer na prvú krabicu  
  
    public NodeStack() { top = null; size = 0; } // prázdny stack  
  
    public int size() {  
        return size; // pamätáme si dĺžku, aby sme ju nemuseli počítat'  
    }  
    public boolean isEmpty() { // test na prázdny stack  
        return size==0;  
    }  
    public void push(E elem) { // push už nemá problém s pretečením  
        Node<E> v = new Node<E>(elem, top); // vytvor novú krabicu elem+top  
        top = v; // tá sa stáva vrcholom stacku  
        size++; // dopočítaj size  
    }  
}
```



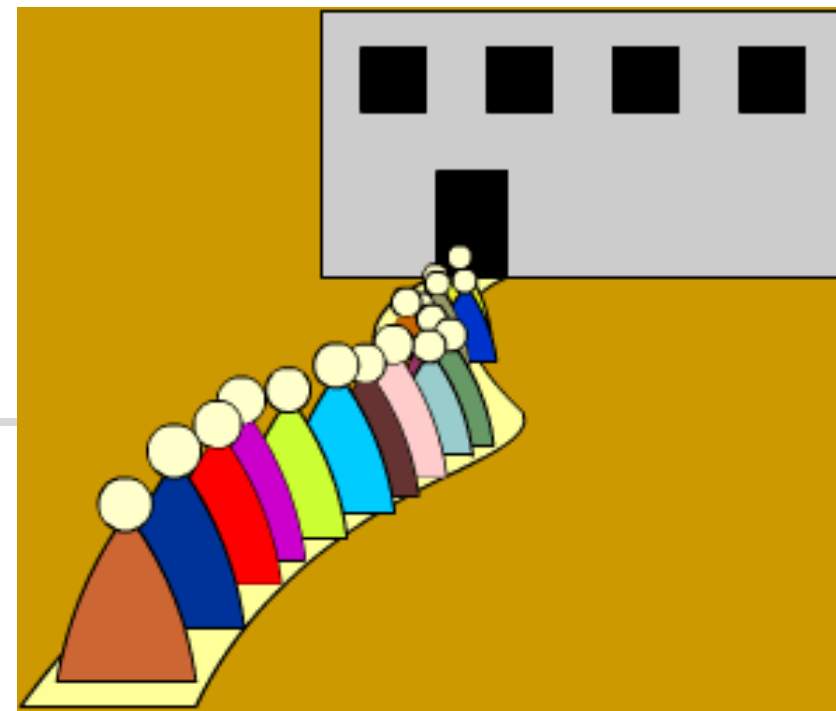
# NodeStack – pokračovanie

```
public E top() throws EmptyStackException {  
    if (isEmpty()) throw new EmptyStackException("empty.");  
    return top.getElement();           // daj hodnotu prvého prvku  
}  
public E pop() throws EmptyStackException {  
    if (isEmpty()) throw new EmptyStackException("empty.");  
    E temp = top.getElement();         // zapamätaj si vrchnú hodnotu  
    top = top.getNext();               // zahod' vrchnú krabicu  
    size--;                           // dopočítaj size  
    return temp;  
}  
}
```

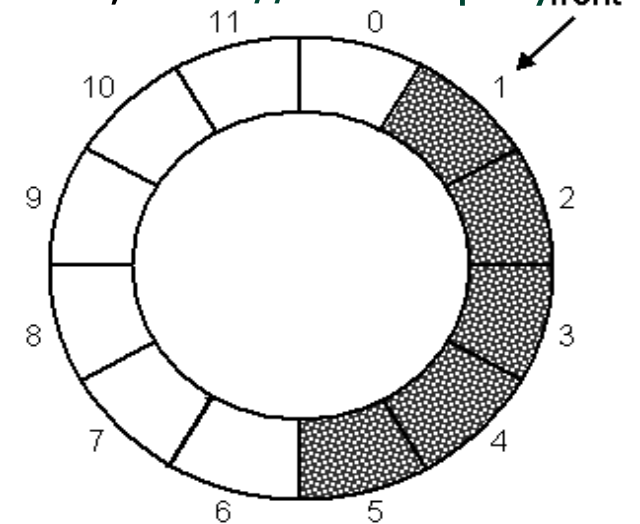
```
NodeStack<Integer> sn = new NodeStack<Integer>();  
for(int i=0; i<10; i++)  
    sn.push(i);  
while (!sn.isEmpty())  
    System.out.println(sn.pop());
```

# Queue - interface

```
public interface QueueInterface<E> {  
    public int size();  
    public boolean isEmpty();  
    public E front() throws EmptyQueueException;  
    public void enqueue (E element);  
    public E dequeue() throws EmptyQueueException;  
}
```



// prvý  
// pridaj posledný  
// zober prvý



rear Súbor: [QueueInterface.java](#)

```

/**
 * zarad prvok elem s prioritou prio
 */
@Override
public void enqueue(E elem, int prio) {
    int index = first;
    int index_1 = Math.floorMod(first-1,f.length);
    while (priority[index] < prio ) {

        f[index_1] = f[index];
        priority[index_1] = priority[index];
        index_1 = index;
        index = Math.floorMod(index+1,f.length);
    }
    f[index] = elem;
    priority[index] = prio;
}

/**
 * vyber prvok s najmensou prioritou
 */
@Override
public E dequeue() {
    if (isEmpty()) {
        return null;
    }
    E el = f[first];
    //%%
    first = Math.floorMod(first +1, f.length);
    return el;
}

```

```

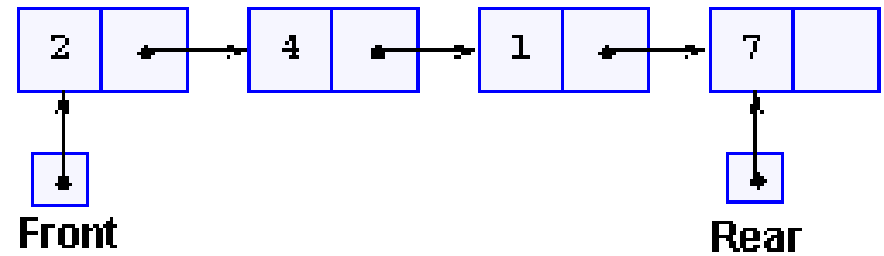
/**
 * zarad prvok elem s prioritou prio
 */
@Override
public void enqueue(E elem, int prio) {
    int index = first;
    int index_1 = Math.floorMod(first-1,f.length);
    while (priority[index] < prio &&) {

        f[index_1] = f[index];
        priority[index_1] = priority[index];
        index_1 = index;
        index = Math.floorMod(index+1,f.length);
    }
    f[index] = elem;
    priority[index] = prio;
}

/**
 * vyber prvok s najmensou prioritou
 */
@Override
public E dequeue() {
    if (isEmpty()) {
        return null;
    }
    E el = f[first];
    //%%
    first = Math.floorMod(first +1, f.length );
    return el;
}

```

# Queue



Reprezentácia:

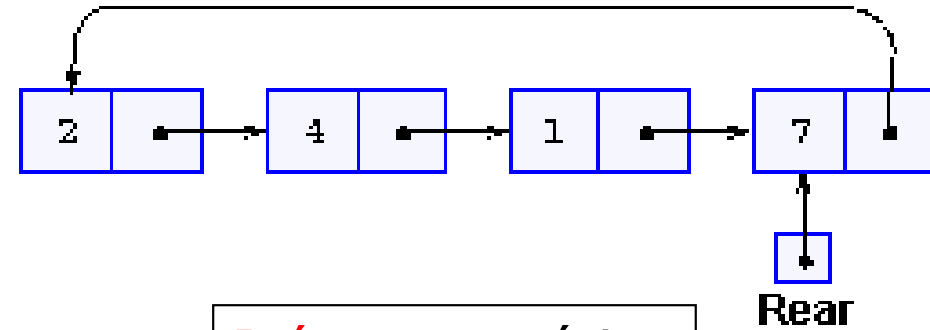
```
Node<E> front;    // prvý  
Node<E> rear;     // posledný  
int size = 0;     // veľkosť
```

```
public void enqueue(E elem) {  
    Node<E> node = new Node<E>();  
    node.setElement(elem);  
    node.setNext(null);  
    if (size == 0) // prvý prvok prázdneho frontu  
        front = node;  
    else  
        rear.setNext(node);  
    rear = node;  
    size++;  
}
```

```
public E dequeue() throws EmptyQueueException {  
    if (size == 0)  
        throw new  
            EmptyQueueException("Queue is empty.");  
    E tmp = front.getElement();  
    front = front.getNext();  
    size--;  
    if (size == 0) // bol to posledný prvok frontu  
        rear = null;  
    return tmp;  
}
```

# Queue2

iná reprezentácia



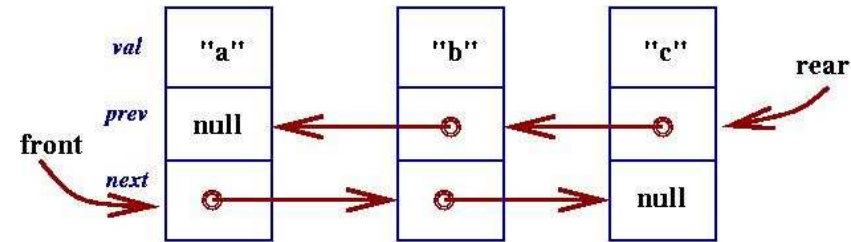
Iná reprezentácia:  
`Node<E> rear;`  
`int size = 0;`

```
public void enqueue(E elem) {  
    Node<E> node = new Node<E>();  
    node.setElement(elem);  
    if (size == 0)  
        node.setNext(node);  
    else {  
        node.setNext(rear.getNext());  
        rear.setNext(node);  
    }  
    rear = node;  
    size++;  
}
```

```
public E dequeue()  
    throws EmptyQueueException {  
    if (size == 0)  
        throw new EmptyQueueException(  
            "Queue is empty.");  
    size--;  
    E tmp = rear.getNext().getElement();  
    if (size == 0)  
        rear = null;  
    else  
        rear.setNext(rear.getNext().getNext());  
    return tmp;  
}
```

# Balík – interface

obojstranne spájaný zoznam  
double linked list

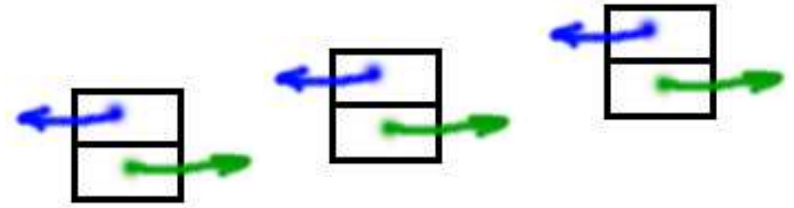


```
public interface DequeInterface<E> {  
  
    public int size();  
    public boolean isEmpty();  
  
    public E getFirst() throws EmptyDequeException;  
    public E getLast() throws EmptyDequeException;  
  
    public void addFirst (E element);  
    public void addLast (E element);  
  
    public E removeFirst() throws EmptyDequeException;  
    public E removeLast() throws EmptyDequeException;  
}
```



# DLNode

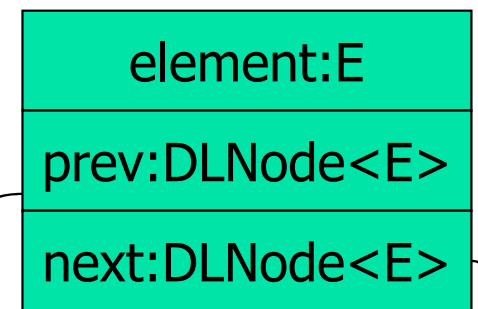
double linked node



```
public class DLNode<E> {  
    private E element;  
    private DLNode<E> prev, next;
```

// obojsmerne spájaný zoznam

```
    public DLNode() { this(null, null, null); }  
    public DLNode(E e, DLNode<E> p, DLNode<E> n) {  
        element = e;  
        next = n;  
        prev = p;  
    }  
    public E getElement() { return element; }  
    public DLNode<E> getNext() { return next; }  
    public void setElement(E newElem) {  
        element = newElem;  
    }  
}
```



# Balík – implementácia

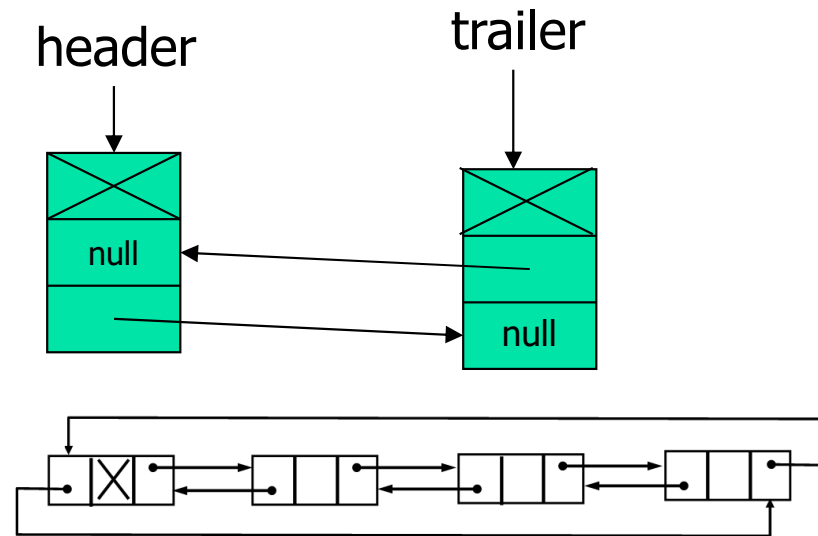
sentinel nodes

```
public class Deque<E> implements DequeInterface<E> {
```

```
    protected DLNode<E> header, trailer; // reprezentácia balíka dvomi  
    protected int size; // pointerami na zač. a koniec
```

```
    public Deque() { // konštruktor  
        header = new DLNode<E>();  
        trailer = new DLNode<E>();  
        header.setNext(trailer);  
        trailer.setPrev(header);  
        size = 0;  
    }
```

```
    public E getFirst() throws Exception {  
        if (isEmpty()) throw new Exception("Deque is empty.");  
        return header.getNext().getElement();  
    }
```





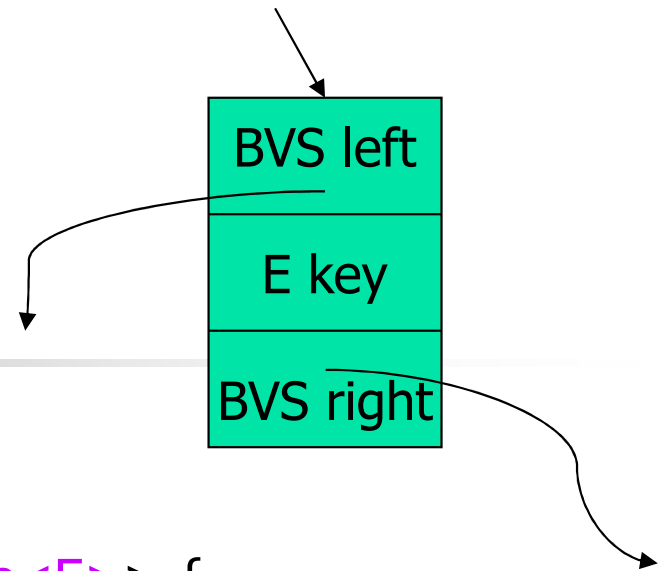
# Balík – implementácia

sentinel nodes

Zmyslom „nárazníkov“/zakážok (resp. sentinel nodes) je odľahčiť kód

```
public void addFirst(E o) {  
    DLNode<E> second = header.getNext();  
    DLNode<E> first = new DLNode<E>(o, header, second);  
    second.setPrev(first);  
    header.setNext(first);  
    size++;  
}  
public E removeLast() throws Exception {  
    if (isEmpty()) throw new Exception("Deque is empty.");  
    DLNode<E> last = trailer.getPrev();  
    E o = last.getElement();  
    DLNode<E> secondtolast = last.getPrev();  
    trailer.setPrev(secondtolast);  
    secondtolast.setNext(trailer);  
    size--;  
    return o;  
}
```

# BVSNNode



parametrizovateľný model:

```
public class BVSNNode<E extends Comparable<E>> {  
    BVSNNode left;  
    E key;  
    BVSNNode right;  
    public BVSNNode(E key) { // konštruktor  
        this.key = key;  
        left = right = null;  
    }  
}
```

- Comparable ([Comparable<E>](#)) je interface predpisujúci jedinú metódu:  
**int compareTo**(Object o), [<E>int compareTo\(E e\)](#)
- základné triedy implementujú interface Comparable (ak to dáva zmysel):  
Integer, Long, ..., String, Date, ...
- pre iné triedy môžeme dodefinovať metódu `int compareTo()`



# Interface Comparable

ak typ nie je primitívny musíme mu prezradiť, ako porovnávať hodnoty tohto typu

```
public class Zamestanec implements Comparable<Zamestanec> {  
    private final String meno, priezvisko;  
    public Zamestanec(String meno, String priezvisko) { // konštruktor  
        this.meno = meno; this.priezvisko = priezvisko;  
    }  
    public int compareTo(Zamestanec n) {  
        int lastCmp = priezvisko.compareTo(n.priezvisko);  
        return (lastCmp != 0 ? lastCmp : meno.compareTo(n.meno));  
    }  
    // alternatíva  
    public int compareTo(Object o) {  
        if (!(o instanceof Zamestanec)) return -9999;  
        Zamestanec n = (Zamestanec)o;  
        int lastCmp = priezvisko.compareTo(n.priezvisko);  
        return (lastCmp != 0 ? lastCmp : meno.compareTo(n.meno));  
    }  
}
```



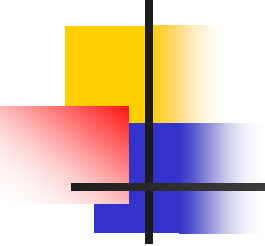
# BVSTree

(insert)

```
public class BVSTree<E extends Comparable<E>> {  
    BVSNode<E> root;    // smerník na vrchol stromu
```

```
    public BVSTree() {  
        root = null;  
    }  
    public void insert(E x) {  
        root = (root == null)? // je prázdny ?  
            new BVSNode<E>(x): // vytvor  
                                // jediný uzol  
        root.insert(x); // inak vsuň do  
                        // existujúceho stromu  
    }
```

```
    public BVSNode<E> insert (E k) {  
        if (k.compareTo(key) < 0)  
            if (left == null)  
                left = new BVSNode<E>(k);  
            else  
                left = left.insert(k);  
        else  
            if (right == null)  
                right = new BVSNode<E>(k);  
            else  
                right = right.insert(k);  
        return this;  
    }
```



# BVSTree – zlé riešenie

## (delete)

---

```
public void delete(E k) { root = root.delete(k); }
```

```
private BVSTree<E> delete(E k) {
```

```
    if (this == null)
```

```
        return null;
```

```
    if (k.compareTo(key) < 0)
```

```
        left = left.delete(k);
```

```
    else if (k.compareTo(key) > 0)
```

```
        right = right.delete(k);
```

```
    else // k == key
```

```
        this = null;
```

Pozor na konštrukcie:

- if (this == null)
- this = null,  
pravdepodobne indikujú chybu



# BVSTree

## (delete)

Pozor na konštrukcie:

- this = null,
  - if (this == null)
- pravdepodobne indikujú chybu

```
public void delete(E k) { root = delete(k, root); }
```

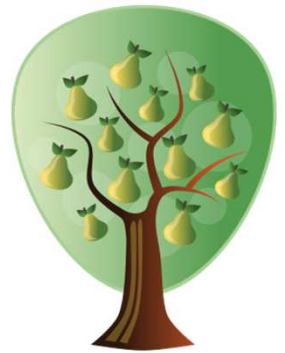
```
private BVSNode<E> delete(E k, BVSNode<E> t ) {  
    if (t == null )  
        return t;  
    if (k.compareTo(t.key) < 0)  
        t.left = delete(k, t.left);  
    else if(k.compareTo(t.key) > 0)  
        t.right = delete(k, t.right);  
    else if( t.left != null && t.right != null ) {  
        t.key = findMin(t.right).key;  
        t.right = delete(t.key, t.right);  
    } else  
        t = (t.left != null) ? t.left : t.right;  
    return t;  
}
```

```
// element je v ľavom podstrome  
// delete v ľavom podstrome  
// element je v pravom podstrome  
// delete v pravom podstrome  
// je to on, a má oboch synov  
// nájdi min.pravého podstromu  
// rekurz.zmaž minimum  
// pravého podstromu  
// ak nemá 2 synov, je to ľahké
```



# Klonovanie

(trieda Hruska)



```
interface Clonable {           // vlastná analógia clon(e)able
    public Object copy();      // z istého dôvodu úplne vo vlastnej réžii
}

public class Hruska implements Comparable<Hruska>, Clonable {

    static int allInstances = 0;           // počítadlo všetkých inštancií
    private int instanceIndex;             // koľkatá inštancia v poradí
    private int size;                      // veľkosť hrušky

    public Hruska(int size) { this.size = size;
        instanceIndex = allInstances++;
        System.out.println("create Hruska " + instanceIndex);
    }

    public Hruska copy() {
        System.out.println("copy Hruska " + instanceIndex);
        return new Hruska(size);
    }

    public int compareTo(Hruska inaHruska) {
        return Integer.compare(this.size, inaHruska.size);
    }
}
```

# Klonovanie

(trieda BVSNode)

```
class BVSNode<E extends Comparable<E> & Clonable> implements Clonable {
    BVSNode<E> left, right; E key;

    static int allInstances = 0;           // počítadlo všetkých inštancií
    private int instanceIndex;           // koľkatá inštancia v poradí

    public BVSNode(E theKey) { key = theKey; left = right = null;
        instanceIndex = allInstances++;
        System.out.println("create BVSNode " + instanceIndex);
    }

    public BVSNode<E> copy() {
        System.out.println("copy BVSNode " + instanceIndex);
        BVSNode<E> clone = new BVSNode<E>(
            (key!=null)?(E)(key.copy()):null    );
        clone.left = (left != null) ? left.copy():null;
        clone.right = (right != null) ? right.copy():null;
        return clone;
    }
}
```



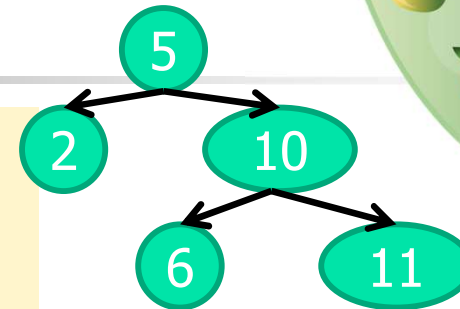
# Klonovanie

(trieda BVSTree)

```
class BVSTree<E extends Comparable<E> & Clonable> implements Clonable {  
    BVSNode<E> root;    // pointer na koreň stromu  
  
    static int allInstances = 0;  
    private int instanceIndex;  
  
    public BVSTree () {  
        instanceIndex = allInstances++;  
        System.out.println("create BVSTree " + instanceIndex);  
        root = null;  
    }  
  
    public BVSTree<E> copy() {  
        System.out.println("copy BVSTree " + instanceIndex);  
        BVSTree<E> clone = new BVSTree<E>();  
        clone.root = (root != null)?root.copy():null;  
        return clone;  
    }  
}
```

# Pear Tree Copy

(Klonovanie stromu hrušiek)



```
create BVSTree 0
create Hruska 0
create BVSTree 0
create Hruska 1
create BVSTree 1
create Hruska 2
create BVSTree 2
create Hruska 3
create BVSTree 3
create Hruska 4
create BVSTree 4
```

```
<key:som hruska 5:> - <left:som hruska 2>, <right:som hruska 10>
<key:som hruska 2:> - <x>, <x>
<key:som hruska 10:> - <left:som hruska 6>, <right:som hruska 11>
<key:som hruska 6:> - <x>, <x>
<key:som hruska 11:> - <x>, <x>
```

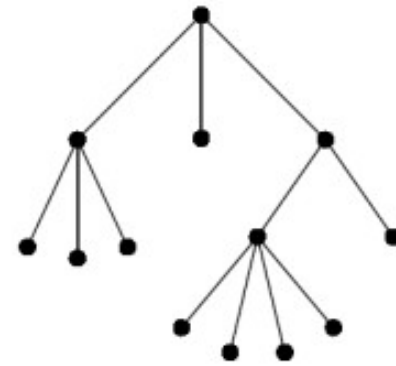
```
BVSTree<Hruska> s =
    new BVSTree<Hruska>();
Random r = new Random();
for(int i=0; i<5; i++)
    s.insert(new Hruska(r.nextInt(19)));
```

(BVSTree<Hruska>)

```
s.copy();
copy BVSTree 0
create BVSTree 1
copy BVSTree 0
copy Hruska 0
create Hruska 5
create BVSTree 5
copy BVSTree 3
copy Hruska 3
create Hruska 6
create BVSTree 6
```

```
copy BVSTree 1
copy Hruska 1
create Hruska 7
create BVSTree 7
copy BVSTree 4
copy Hruska 4
create Hruska 8
create BVSTree 8
copy BVSTree 2
copy Hruska 2
create Hruska 9
create BVSTree 9
```

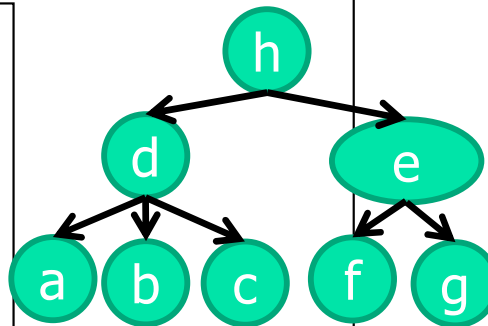
# Všeobecný strom



```
class Node<T extends Comparable<T>> implements Comparable<Node<T>> {  
    private T          value;  
    private List<Node<T>> sons;
```

- `T extends Comparable<T>` - znamená, že predpokladáme porovnávanie na type `T`
- `implements Comparable<Node<T>>` - znamená, že chceme porovnávať celé stromy
  - a preto musíme definovať
  - `@Override`  
`public int compareTo(Node<T> o) { ... }`
- príklad rekurzcie cez strom, inicializácie:

```
public int size() {  
    int count = 1;  
    for (Node<T> son : sons)  
        if (son != null)  
            count += son.size();  
    return count;  
}
```



```
Node<String> tree = new Node("h",  
    List.of(  
        new Node("d",  
            List.of(  
                new Node("a", null),  
                new Node("b", null),  
                new Node("c", null))  
            ),  
        new Node("e",  
            List.of(  
                new Node("f", null),  
                new Node("g", null))  
            )  
    ));
```