

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/>



>>> Whoops = _ #Magic Happens

Ako to bolo v Python

(encapsulácia)

Enkapsulácia je základný princíp OOP (Rosumie tomu každý, až na G.Rossuma)

- všetko je public by default (katastrofa)
- protected – začínajú jedným _ / ale to je len konvencia
prefix _ znamená, že nepoužívaj ma mimo podtriedy (doporučenie...)
- private – začínajú dvomi __ / to nie je konvencia

class Bod:

 _totoJeProtectedVariable = 5

 __totoJePrivateVariable = 15

bod = Bod()

bod._totoJeProtectedVariable

bod.__totoJePrivateVariable

bod._Bod__totoJePrivateVariable

```
Python 3.4.0 (default, Apr 11
[GCC 4.8.2] on linux
5
15
```

niekto mi to zakáže ??? ☹️

niekto mi to zakáže ??? 😊

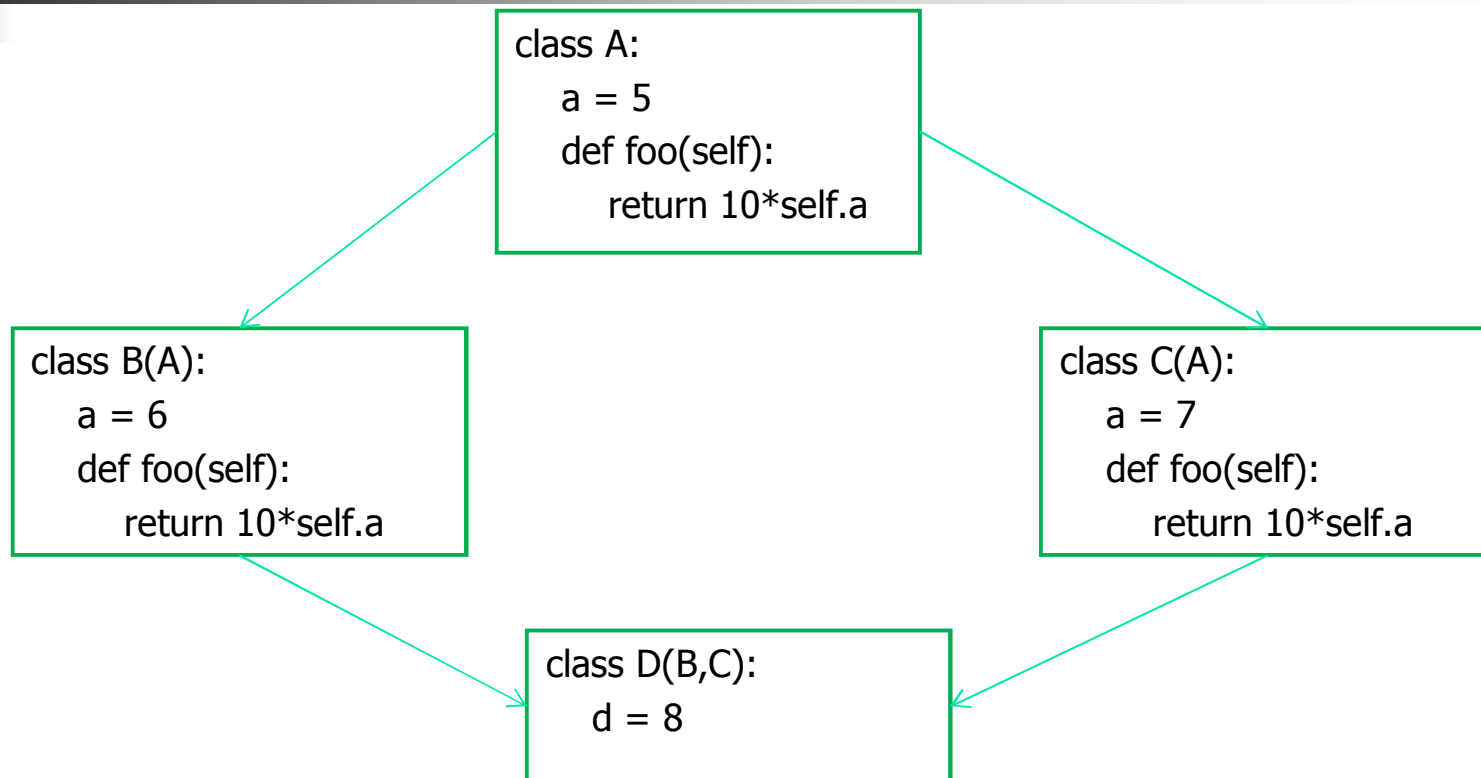
niekto mi to zakáže ???



```
>>> Whoops = _ #Magic Happens
```

Ako to bolo v Python

(dedenie)



Diamond Problem

```
x = D()
print(x.a)
print(x.foo())
```

```
Python 3.4.0 (default, Apr 1
[GCC 4.8.2] on linux
6
60
```



Abstract vs. Interface

(rekapitulácia – tentokrát už v Jave)

aký je rozdiel medzi abstraktnou triedou a interface:

- `abstract class XXX { ... foo(...) ; }` a `interface XXX { ... foo(...) ; }`
- 1. trieda **dedí** od abstraktnej triedy, pričom trieda **implementuje** interface
- 2. rovnako **nejde urobiť new** od abstraktnej triedy ani od interface
- 3. abstraktná trieda môže predpísať defaultné správanie v neabstraktných metódach
- 4. abstraktná trieda vás donúti v podtriedach dodefinovať správanie abstraktných metód
- 5. trieda môže zároveň **implementovať viac interface**, ale nemôže dediť od viacerých

abstraktná trieda	interface
môže mať abstraktné aj neabstraktné metódy	len abstraktné public, takže public abstract ani nepíšeme
dve abstraktné triedy nemôžeme podediť do jednej	interface podporuje viacnásobné dedenie
môže mať final/non-final, static/non-static premenné	len static a final, takže k nim static final ani nepíšeme
môže mať statické metódy (napr. main), aj konštruktor	... nič z toho
abstraktná trieda môže implementovať interface	Interface nie je implementáciou abstraktnej triedy



Konštruktor v abstraktnej ???

```
public abstract class AbstractConstructor {
    int value;

    private AbstractConstructor() { // konštruktor, dokonca privátny v abstraktnej triede
        this(0);
    }

    public AbstractConstructor(int val) { // explicitne sa nedá zavolať pomocou new
        value = val;
    }

    abstract public int lenAbyTuBoloNiecoAbstract();
}

class JejPodtrieda extends AbstractConstructor {
    public JejPodtrieda() { // konštruktor v podtriede zavolá konštruktor v abstraktnej
        super(0);          // nadtriede, ktorá zavolá privátny konštruktor
    }

    public int lenAbyTuBoloNiecoAbstract() {
        return value;
    }
}
```

[Súbor:AbstractConstructor.java](#)



Interface a dedenie

(Môže interface dediť ?)

- podtrieda dedí z nadtriedy metódy a atribúty (dedenie = class inheritance)
- interface sa tiež môže dediť (z typu dostaneme jeho podtyp)
- hodnota podtypu je použiteľná všade tam, kde sa čaká nadtyp

ALE:

trieda implementujúca podtyp nie je podtriedou triedy implementujúcej nadtyp

```
interface NadInterface { // nadtyp, NI
```

```
interface PodInterface extends NadInterface { // podtyp, PI
```

```
// trieda implementujúca nadtyp
```

```
class NadInterfaceExample implements NadInterface // NIE
```

```
// trieda implementujúca podtyp
```

```
class PodInterfaceExample implements PodInterface // PIE
```

[NadInterface.java](#), [PodInterface.java](#)



Interface vs. class inheritance

```
public interface NadInterface {  
    public void add(String s);  
    int aa = 9;  
}
```

NI

dedenie

```
public interface PodInterface  
    extends NadInterface {  
    public void addAll(String[] p);  
    int bb = 10;  
}
```

PI

implementácia

```
public class NadInterfaceExample  
    implements NadInterface {  
    public void add(String s) { }  
    public int a;  
}
```

NIE

implementácia

```
public class PodInterfaceExample  
    implements PodInterface {  
    public void add(String s) { }  
    public void addAll(String[] p) { }  
    public int b;  
}
```

PIE

Je toto nadtrieda tejto ?

[NadInterfaceExample.java](#), [PodInterfaceExample.java](#)



Interface vs. class inheritance

```
NadInterfaceExample nie = new NadInterfaceExample();  
PodInterfaceExample pie = new PodInterfaceExample();
```

```
pie.addAll(null); 😊
```

```
nie.addAll(null); ☹️
```

NadInterfaceExample a PodInterfaceExample nie sú podtriedy

```
NadInterfaceExample nie2 = pie; ☹️
```

```
PodInterfaceExample pie1 = nie; ☹️
```

NadInterfaceExample a PodInterfaceExample sú ako bratranci

```
System.out.print(pie.b); 😊
```

```
System.out.print(pie.a); ☹️
```

```
System.out.print(pie.bb); 😊
```

```
System.out.print(pie.aa); 😊
```




Interface vs. class inheritance

```
NadInterface ni = new NadInterfaceExample(); // nie  
PodInterface pi = new PodInterfaceExample(); // pie
```

NadInterface je nadtyp PodInterface

```
NadInterface nie1 = ni; 😊
```

```
NadInterface nie2 = pi; 😊
```

```
PodInterface pie1 = ni; 😞
```

```
PodInterface pie2 = pi; 😊
```

Uffff....

Ďalšie čítanie: <http://www.wideskills.com/java-tutorial/java-abstract-class-and-interface>



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 front, balík, ...
 - implementácie: polia, jednoduché a obojsmerne spájané zoznamy

cvičenia:

- interface a implementation pre ADT
- 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.out.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.out.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 (už sme to kedysi videli),
 - 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,
 - 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**

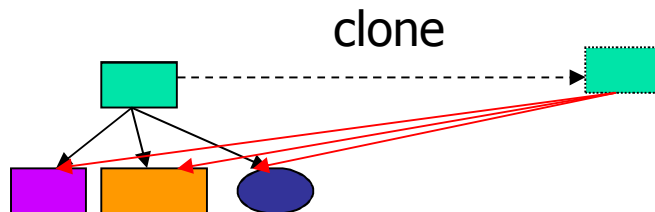
ak $o : Object$ a hodnotou je objekt triedy E , potom $(E)o : E$
explicitný *cast* predstavuje typovú kontrolu v runtime,
napr. $(Integer)o : Integer$, $(String)o : String$

ak však hodnota objektu o nie je triedy E , potom runtime check
 $(E)o$ zlyhá (cast exception)

- 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):

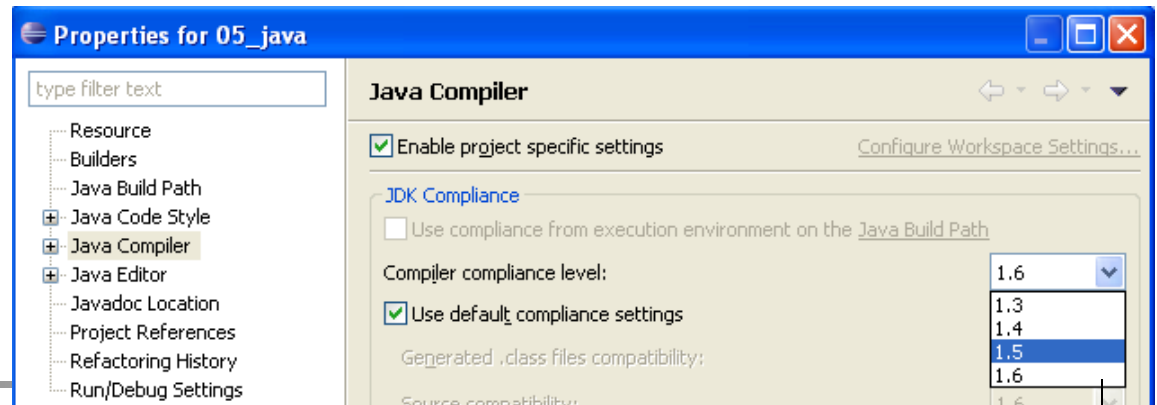


- **boolean equals(Object obj)** – porovná referencie na objekty,

`x.clone() != x`

`x.clone().getClass() == x.getClass()`

Generics



```
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];  
    }  
}
```

Ak ich nemáte zapnuté,
zapnite si ich

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

V Jave (na rozdiel od 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 pod `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; // pObj[0]:Object, stS: Stack50<String>
Integer i = pObj[0].pop(); // "bude problem" nie je typu Integer,
// lebo pObj[0] nie je Stack50<Integer> ale
// Stack50<String>



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

Použitie generického typu môže byť ohraňované kvalifikátormi na typový parameter, napr. metóda `genMethod2` sa dá použiť len pre číselné typy, t.j. typy podedené z typu `Number` (`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
```

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);  
    }  
}
```

Súbory: [StackInterface.java](#),
[EmptyStackException.java](#),
[FullStackException.java](#)



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());
```

Súbor: [ArrayStack.java](#)

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;
```

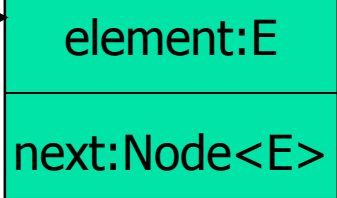
// referencia na nasledujúci vagónik zásobníka

```
}
```

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;  
    }  
}
```





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  
    }  
}
```



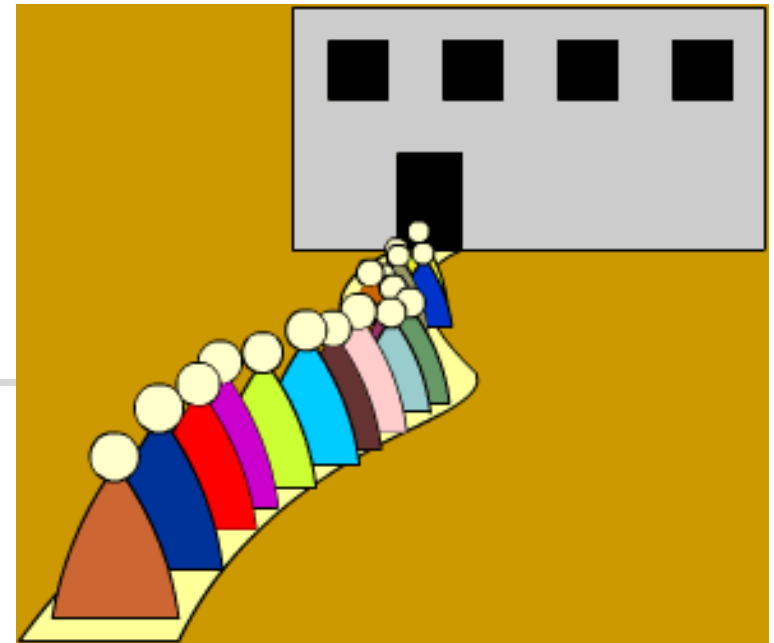
NodeStack – pokračovanie

```
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;
}

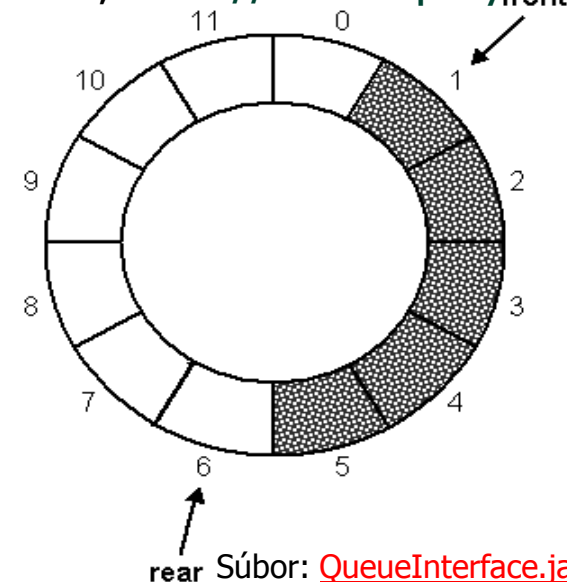
NodeStack<Integer> sn = new NodeStack<Integer>();
for(int i=0; i<10; i++)
    sn.push(i);
while (!sn.isEmpty())
    System.out.println(sn.pop());
```


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;  
}
```

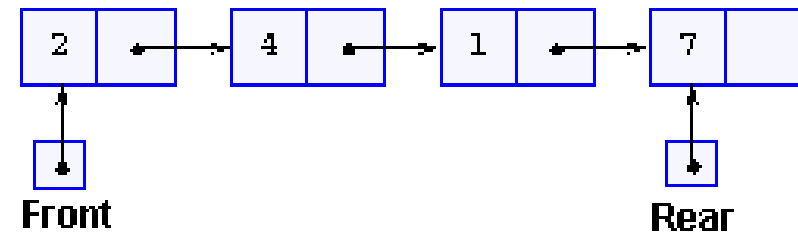


// prvý
// pridaj posledný
// zober prvý



Súbor: [QueueInterface.java](#)

Queue

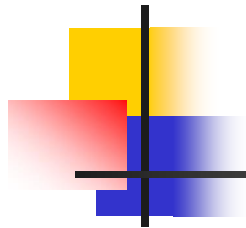


Reprezentácia:

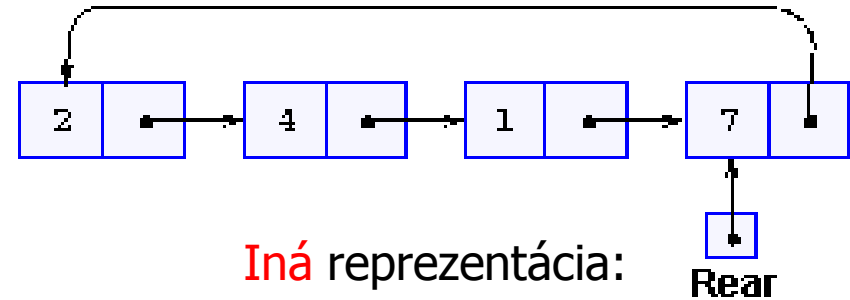
```
public void enqueue(E elem) {
    Node<E> node = new Node<E>();
    node.setElement(elem);
    node.setNext(null);
    if (size == 0) // prvý prvok prázdneho frontu
        front = node;
    else
        rear.setNext(node);
    rear = node;
    size++;
}
```

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

```
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

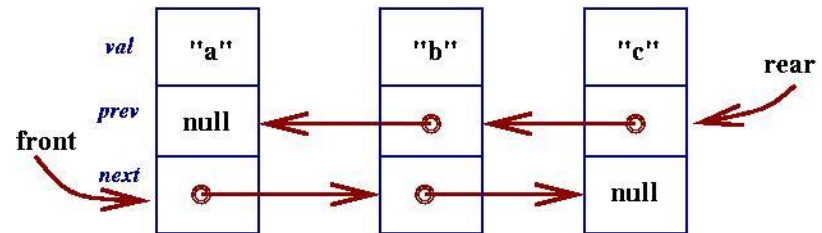


Iná reprezentácia:
`Node<E> rear;`
`int size = 0;`

```
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++;  
}
```

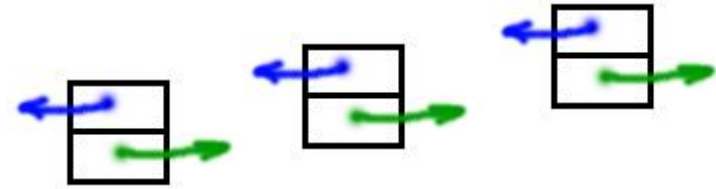
```
public E dequeue()  
    throws EmptyQueueException {  
    if (size == 0)  
        throw new EmptyQueueException(  
            "Queue is empty.");  
    size--;  
    E tmp = rear.getNext().getElement();  
    if (size == 0)  
        rear = null;  
    else  
        rear.setNext(rear.getNext().getNext());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

Balík - interface



```
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

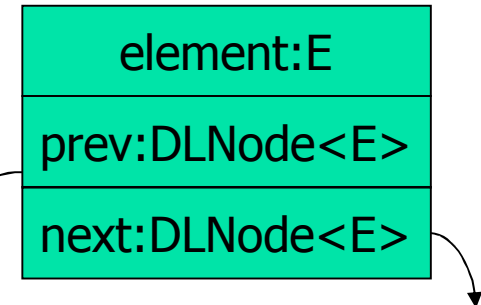


```
public class DLNode<E> {  
    private E element;  
    private DLNode<E> prev, next;
```

// 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;  
    }  
}
```

.....



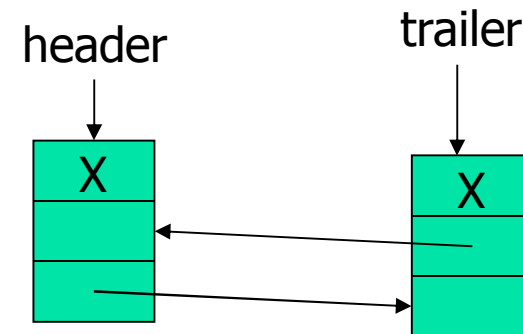
Balík – implementácia

```
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();  
    }
```





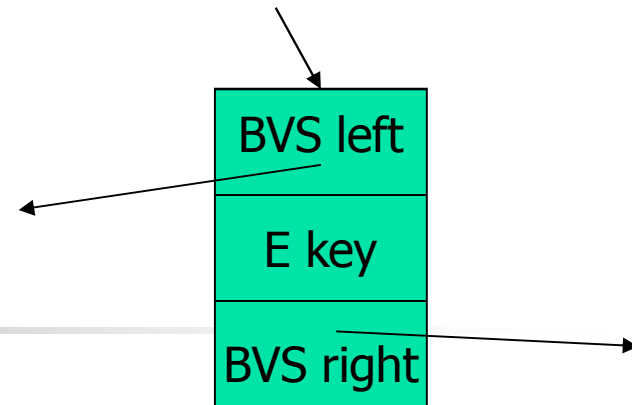
Balík - implementácia

```
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;
}
```



BVSNode

parametrizovateľný model:



```
public class BVSNode<E extends Comparable<E>> {  
    BVSNode left;  
    E key;  
    BVSNode right;  
    public BVSNode(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));  
    }  
}
```

find je cvičenie

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 BVSTreeNode<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)

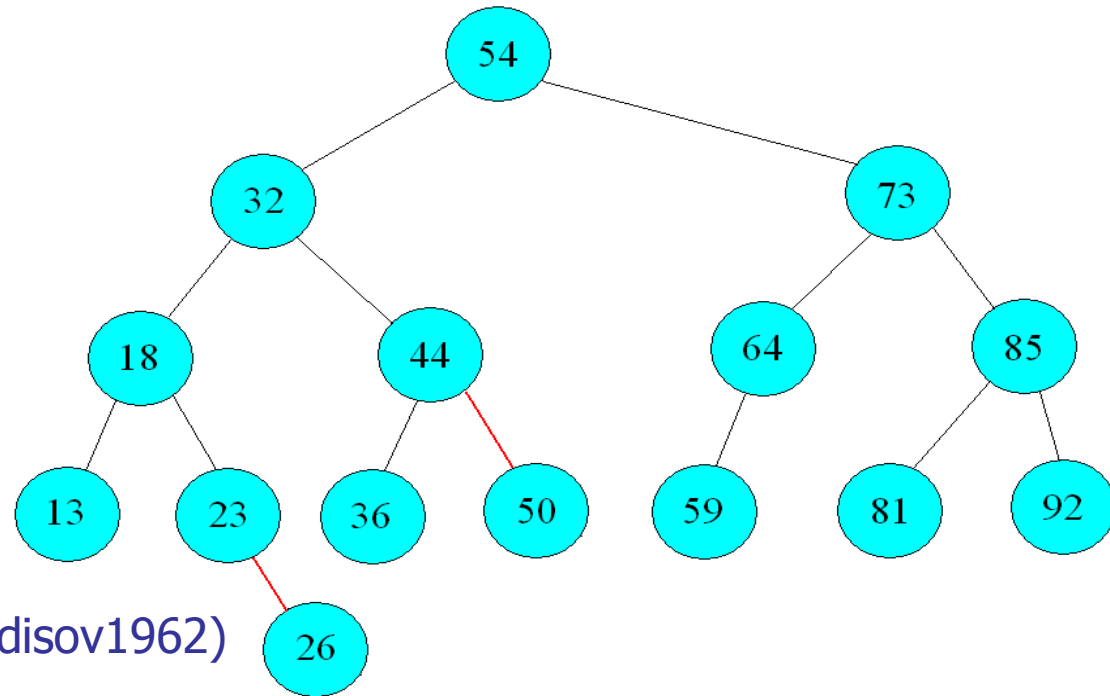
pravdepodobne indikujú chybu

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

```
private BVSTree<E> delete(E k, BVSTree<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  
// nájsť min.pravého podstromu  
// rekurz.zmaž minimum  
// pravého podstromu  
// ak nemá 2 synov, je to ľahké
```

Vyvážené stromy



AVL Tree (Adelson-Velsky Landisov1962)

2-3, B-Tree (Bayer,1971)

Red-Black (Sedgewick,1978)

Splay Tree (Tarjan,1986)

Tree-Forest

vyskúšajte si demo, applet ilustrujúci stromové ADT:

<http://www.qmatica.com/DataStructures/Trees/BST.html>

<http://www.qmatica.com/DataStructures/Trees/AVL/AVLTree.html>