

# JAVA

`java.lang.StringBuffer`  
`java.lang.StringBuilder`

# Přehled

- "měnitelný" řetězec
  - instance třídy `String` jsou neměnitelné
- nejsou potomky `String`
  - `String`, `StringBuffer`, `StringBuilder` jsou **final**
- `StringBuffer`
  - bezpečný vůči vláknům
- `StringBuilder`
  - není bezpečný vůči vláknům
  - od Java 5
- mají stejné metody
  - vše pro `StringBuffer` platí i pro `StringBuilder`

# Přehled

- operátor `+` na řetězcích je implementován pomocí třídy `StringBuffer`

výraz `x = "a" + 4 + "c"`

je přeložen do

```
x = new
```

```
    StringBuffer().append("a").append(4).  
    append("c").toString()
```

- základní metody – `append` a `insert`
  - definovány pro všechny typy

# Konstruktory

- `StringBuffer()`
  - prázdný bufer
- `StringBuffer(String str)`
  - bufer obsahující `str`
- `StringBuffer(int length)`
  - prázdný bufer s iniciální kapacitou `length`
    - kapacita je během práce s buferem dle potřeby zvětšována
- `StringBuffer(CharSequence chs)`
  - `CharSequence`
    - interface
    - implementují ho `String`, `StringBuffer`, `StringBuilder`,...

# Metody

- `StringBuffer append(typ o)`
  - definována pro všechny primitivní typy, `Object`, `String` a `StringBuffer`
  - převede parametr na řetězec a připojí na konec
  - vrací referenci na sebe (`this`)
- `StringBuffer insert(int offset, typ o)`
  - definována pro všechny typy jako `append`
  - vloží řetězec na danou pozici
  - `offset` musí být  $\geq 0$  a  $<$  aktuální délka řetězce v bufru
- `StringBuffer replace(int start, int end, String str)`
  - nahradí znaky v bufru daným řetězcem
- `StringBuffer reverse()`
  - převrátí pořadí znaků v bufru

# Metody

- `StringBuffer delete(int start, int end)`
  - odstraní znaky z bufu
  - `start`, `end` – indexy do bufu
- `StringBuffer deleteCharAt(int i)`
  - odstraní znak na dané pozici
- `char charAt(int i)`
  - znak na dané pozici
- `int length()`
  - aktuální délka řetězce v bufu
- `String substring(int start)`
- `String substring(int start, int end)`
  - vrátí podřetězec

# JAVA

## Kolekce

# Přehled

- (collections)
- kolekce ~ objekt obsahující jiné objekty
- např. – pole
  - jen pole nestačí
    - mnoho výhod (vestavěný typ, rychlý přístup, prvky i primitivní typy, ...)
    - omezení – např. pevná velikost
- Java collection library
  - sada interfaců a tříd poskytujících dynamická pole, hašovací tabulky, stromy, ...
  - součást balíku **java.util**
    - ne vše v java.util je kolekce



# Kolekce a Java 5

- Java < 5
  - prvky kolekcí – typ **Object**
    - nelze vkládat primitivní typy
- Java 5
  - kolekce pomocí generických typu
    - fungují kolekce i bez <> - "raw" types
  - stále nelze pro primitivní typy
    - List<int> - chyba při překladu
  - metody zůstaly „stejně“

# Ještě k polím: `java.util.Arrays`

- `java.util.Arrays`
  - sada metod pro práci s poli
  - součást knihovny kolekcí
- metody
  - všechny jsou statické
  - většinou definovány pro všechny primitivní typy a pro `Object`
- `int binarySearch(typ[] arr, typ key)`
  - hledání prvku v poli
  - binární vyhledávání
  - pole musí být setříděno vzestupně
  - vrací index prvku pokud v poli je nebo zápornou hodnotu indexu, kam by prvek patřil, kdyby v poli byl

# Ještě k polím: `java.util.Arrays`

- `boolean equals(typ[] a1, typ[] a2)`
  - porovnává, zda jsou pole stejná, tj. stejně dlouhá a obsahují stejné prvky
  - prvky jsou stejné, pokud  
(`e1==null ? e2==null : e1.equals(e2)`)
- `void fill(typ[] arr, typ val)`
  - vyplní všechny prvky pole parametrem `val`
- `void fill(typ[] arr, int from, int to, typ val)`
  - vyplní zadanou část pole parametrem `val`
- `void sort(typ[] arr)`
  - seřídí pole vzestupně
  - quicksort pro primitivní typy, mergesort pro `Object`
- `void sort(typ[] arr, int from, int to)`
  - seřídí zadanou část pole

# Třídění pole

- `void sort(Object[] arr)`
  - prvky pole musí být porovnatelné, tj. implementovat interface `java.lang.Comparable<T>`
    - metoda `int compareTo(T o)`
- `void sort(T[] arr, Comparator<? super T> c)`
  - prvky stále musí být porovnatelné
  - na porovnávání se použije objekt `c`
  - interface `java.util.Comparator<T>`
    - `int compare(T o1, T o2)`
  - pro vyhledávání
    - `int binarySearch(T[] a, T key, Comparator<? super T> c)`
- `void parallelSort(typ[] a)`
  - paralelní mergesort
    - `ForkJoinPool`

# java.util.Arrays

- `typ[] copyOf(typ[] original, int newLength)`
- `typ[] copyOfRange(typ[] original, int from, int to)`
- `<T,U> T[] copyOf(U[] original, int newLength, Class<? extends T[]> newType)`
  - kopie pole
- `<T> List<T> asList(T... a)`
  - pole => list

# Základní kolekce

- dva základní druhy – interface **Collection** a **Map**
- **Collection<E>**
  - skupina jednotlivých prvků
  - **List<E>**
    - drží prvky v nějakém daném pořadí
  - **Set<E>**
    - každý prvek obsahuje právě jednou
  - **Queue<E>** (od Java 5)
    - fronta prvků
  - **Deque<E>** (od Java 6)
    - oboustranná fronta prvků
- **Map<K,V>**
  - skupina dvojic klíč–hodnota
- pro každý druh kolekce existuje alespoň jedna implementace
  - většinou více

# Hierarchie kolekcí

- kolekce neimplementují přímo daný interface
- implementují třídy `AbstractSet`, `AbstractList`, `AbstractMap`, `AbstractQueue`, `AbstractDeque`
  - abstraktní třídy
  - poskytují základní funkčnost dané kolekce
  - každá implementace interfacu `Set`, `List`,... by měla rozšiřovat danou `Abstract` třídu
- používání kolekcí
  - obvykle přes interface daného druhu kolekce
  - př. `List c = new ArrayList()`
  - lze potom v aplikaci snadno vyměnit implementaci

# Iterator<E>

- kolekce nemusí přímo podporovat přístup k prvkům
- kolekce mají metodu
  - `Iterator<E> iterator()`
  - vrací objekt typu `Iterator<E>`, který umožní projít všechny prvky v kolekci
- metody
  - `E next()` - vrací další prvek kolekce
  - `boolean hasNext()` - `true`, pokud jsou další prvky
  - `void remove()`
    - odstraní poslední vrácený prvek z kolekce
    - default od Java 8 (vyhazuje `UnsupportedOperationException`)
  - default `void forEachRemaining(Consumer<? super E> action)`
    - od Java 8



# Iterator<E>

- implementace iteratoru a vztah k prvkům kolekce záleží na konkrétní kolekci

```
List c = new ....  
...  
Iterator e = c.iterator();  
while (e.hasNext()) {  
    System.out.println(e.next());  
}
```

- cyklus **for** na kolekce s iterátorem
  - tj. implementující interface `Iterable`

```
for (x:c) {  
    System.out.println(x);  
}
```

# Iterable<T>

- `Iterator<T> iterator()`
  - vrací iterátor
- `default void forEach(Consumer<? super T> action)`
  - provede akci na všechny elementy
  - od Java 8
- `default Splitter<T> splitter()`
  - vrací spliterator
  - od Java 8

# Collection<E>

- `boolean add(E o)`
  - přidá objekt do kolekce
  - vrací `false`, pokud se nepodařilo objekt přidat
  - volitelná metoda
- `boolean addAll(Collection<? extends E> c)`
  - přidá všechny prvky
  - vrací `true`, pokud přidala nějaký prvek
  - volitelná metoda
- `void clear()`
  - odstraní všechny objekty
  - volitelná metoda
- `boolean contains(E o)`
  - vrací `true`, pokud je objekt v kolekci

# Collection<E>

- `boolean containsAll(Collection<?> c)`
  - vrací `true`, pokud jsou všechny dané objekty v kolekci
- `boolean isEmpty()`
- `Iterator<E> iterator()`
- `boolean remove(E o)`
  - vrací `true`, pokud odstranila objekt z kolekce
  - volitelná metoda
- `boolean removeAll(Collection<?> c)`
  - snaží se odstranit dané objekty z kolekce
  - vrací `true`, pokud něco odstranila
  - volitelná metoda
- `boolean retainAll(Collection<?> c)`
  - odstraní objekty, které nejsou v `c`
  - volitelná metoda

# Collection<E>

- `int size()`
  - počet prvků v kolekci
- `default Splitter<E> splitter()`
- `default boolean removeIf(Predicate<? super E> filter)`
  - odstraní objekty splňující podmínku
- `Object[] toArray()`
  - vrátí pole obsahující všechny prvky kolekce
- `T[] toArray(T[] a)`
  - vrátí pole obsahující všechny prvky kolekce
  - vrácené pole je stejného typu jako je pole `a`

```
List<String> c;
```

```
.....
```

```
String[] str = c.toArray(new String[1]);
```

# List<E>

- potomek Collection
- udržuje prvky v nějakém pořadí
- může obsahovat jeden prvek vícekrát
- má metodu `E get(int index)`
  - vrátí prvek na dané pozici v kolekci
- default `void sort(Comparator<? super E>)`
  - od Java 8
- kromě Iteratoru umožňuje získat i ListIterator
- ListIterator
  - potomek iteratoru
  - umožňuje
    - procházet prvky i v obráceném pořadí – metody `previous()`, `hasPrevious()`
    - přidávat a nahrazovat prvky – metody `add()`, `set()`

# List<E>

- dvě implementace
- **ArrayList**
  - implementován polem
  - rychlý náhodný přístup k položkám
  - pomalé přidávání doprostřed
- **LinkedList**
  - rychlý sekvenční přístup
  - pomalý náhodný přístup
  - má navíc metody
    - addFirst()
    - removeFirst()
    - addLast()
    - removeLast()
    - getFirst()
    - getLast()

# Set<E>

- potomek Collection
- nepřidává žádnou novou metodu
- každý prvek může obsahovat pouze jednou
- několik implementací
- **HashSet**
  - velmi rychlé vyhledání prvku v kolekci
  - nedodrží pořadí
- **TreeSet**
  - Set implementovaný pomocí červeno-černých stromů
  - implementuje **SortedSet**
    - prvky jsou setříděny
    - umožňuje vrátit část (podmnožinu) kolekce
- **LinkedHashSet**
  - jako HashSet, ale udržuje pořadí prvků (pořadí vkládání)



# Queue<E>

- potomek Collection
- „fronta“ prvků
- typicky FIFO
- může mít pevnou velikost
- 2 druhy metod pro stejné činnosti
  - pokud selžou, vracejí výjimku (add, remove, element)
  - pokud selžou, vracejí speciální hodnotu (offer, poll, peek)
- `add(E e)`, `offer(E e)`
  - přidává prvek
- `E remove()`, `E poll()`
  - odstraní a vrátí prvek
- `E element()`, `E peek()`
  - vrátí prvek, ale nechá ho ve frontě

# Deque<E>

- „double ended queue“
- potomek od Queue
- podobné metody jako Queue, ale 2x
  - na první prvek
  - na poslední prvek
  
- **addFirst(E)**            **offerFirst(E)**
- **removeFirst()**        **pollFirst()**
- **getFirst()**            **peekFirst()**
  
- **addLast(E)**            **offerLast(E)**
- **removeLast()**        **pollLast()**
- **getLast()**            **peekLast()**

# Map<K, V>

- není potomek **Collection**
- kolekce dvojic klíč–hodnota
  - ~ asociativní pole
- každý klíč obsahuje pouze jednou
- metody
  - `V put(K key, V value)`
    - asociuje klíč s hodnotou
    - vrátí původní hodnotu asociovanou s klíčem (null, pokud klíč v kolekci nebyl)
  - `V get(K key)`
    - vrátí hodnotu asociovanou s klíčem
  - `boolean containsKey(Object key)`
  - `boolean containsValue(Object val)`
  - `Set<K> keySet()`
  - `Collection<V> values()`

# Map<K,V>: implementace

- několik implementací
- **HashMap**
  - implementace pomocí hašovací tabulky
  - konstantní čas přidávání a vyhledávání
- **LinkedHashMap**
  - jako HashMap
  - navíc při iterování dodržuje pořadí (pořadí přidávání nebo LRU)
  - o něco pomalejší
    - iterování je rychlejší
- **TreeMap**
  - implementace pomocí červeno-černých stromů
  - implementuje interface SortedMap
    - prvky jsou setříděny

# HashMap<K, V>

- prvky musí správně implementovat metody `hashCode()`
- dva objekty, které jsou stejné (ve smyslu metody `equals()`) musí vracet stejný `hashCode`
- různé objekty nemusí nutně vracet různý `hashCode`
- používá se hašování s řetězci
  - různé objekty se stejným `hashCode` budou ve stejném řetězci
- HashMap má na začátku nějaký počet "políček", do kterých se hašuje = kapacita
- faktor využití = počet prvků / kapacita
- při dosažení daného faktoru (implicitně 0.75) se kapacita zvětší a tabulka se "přehašuje"
  - kvůli rychlosti přístupu

# Třída Collections

- obdoba třídy Arrays
- sada statických metod pro práci s kolekcemi
- metody
  - binarySearch
  - fill
  - sort
  - rotate
  - shuffle
  - reverse
  - ...

# Synchronizace

- většina kolekcí není bezpečná vůči vláknům
- bezpečné (synchronizované) kolekce se vytvářejí stejně jako nemodifikovatelné
- metody na na Collections
  - synchronizedCollection
  - synchronizedList
  - synchronizedSet
  - synchronizedMap

# Nemodifikovatelné kolekce

- metody na Collections
  - unmodifiableCollection
  - unmodifiableList
  - unmodifiableSet
  - unmodifiableMap
- mají jeden parametr (daný druh kolekce)
- vrací "read-only verzi" kolekce, která obsahuje stejné položky jako dodaná kolekce



# Nemodifikovatelné kolekce

- metoda `of` pro snadné vytváření
  - od Java 9
  - pro všechny typy kolekcí
    - List, Set, Map

```
List<String> list = List.of("foo", "bar", "baz");  
Set<String> set = Set.of("foo", "bar", "baz");  
Map<String, String> map = Map.of("foo", "a",  
    "bar", "b", "baz", "c");
```

- 12 přetížených metod `of`
  - `of()`, `of(E e)`, `of(E e1, E e2)`, ..., až `of` s 10 parametry
  - `of(E... elems)`

# "Staré" kolekce (Java 1.0, 1.1)

- Java collection library od verze Javy 1.2 přetvořena (List, Set, Map)
- původní kolekce
  - neměly by se používat
    - ale občas se použití nelze vyhnout
  - byly zahrnuty do nové verze (tj. také implementují List, Set nebo Map)
- Vector
  - obdoba ArrayList
- Enumeration
  - obdoba Iterator
- Hashtable
  - obdoba HashMap
- ...

# JAVA

`java.util.stream`

# Přehled

- od Java 8
  - používají lambda výrazy
- zpracovávání kolekcí
  - programátor specifikuje, co chce provést
  - plánování operací je ponecháno na implementaci
  - funkcionální přístup
    - „map & reduce“
- proudy (streams) dat
  - lze získat z kolekcí, polí,...
- v podstatě náhrada za iterátor
  - iterátor předepisuje strategii pro procházení
  - neumožňuje paralelizaci

# Příklad

- `List<String> words = ...// list slov`
- počet slov delších než 10
  - pomocí iterátoru

```
int count = 0;
for (String w : words) {
    if (w.length() > 10) count++;
}
```
  - pomocí streamů

```
long count = words.stream().filter(w ->
w.length() > 10).count();
```
- obě řešení jsou správná a funkční
- ale iterátor předepisuje procházení a nelze paralelizovat

# Vlastnosti streamů

- `java.util.stream.Stream<T>`
  - interface
- stream neдрží v sobě elementy
  - jsou v kolekci „pod“ streamem nebo generovány
- operace na streamu nemění zdroj, ale vytvářejí nový stream
- pokud to lze, operace na streamu jsou „líné“
- lze je snadno paralelizovat
  - `long count = words.parallelStream().filter(w -> w.length() > 12).count();`

# Operace na streamech

- stream pipeline
  - posloupnost operací na streamu
- druhy stream operací
  - intermediate
    - vracejí nový stream
    - vždy jsou líné
    - vyhodnocují se až při aplikaci „terminal“ operace
  - terminal
    - (téměř vždy) nejsou líné
    - „konzumují“ stream pipeline
    - produkují něco jiného než stream

# Operace na streamech

- parametry operací – funkcionální interfacery
  - skutečné parametry – lambda výrazy
- balíček `java.util.function`
  - `Function<T, R>`  
`R apply(T t)`
  - `Predicate<T>`  
`boolean test(T t)`
  - `Supplier<T>`  
`T get()`
  - `UnaryOperator<T>` extends `Function<T, T>`  
`T apply(T t)`
  - `BinaryOperator<T>` extends  
`BiFunction<T, T, T>`
  - ...



# Vytváření streamů

- `kolekce.stream()`
  - `kolekce.parallelStream()`
  - metody na interfacu Stream
    - `static <T> Stream<T> of(T... values)`
    - `static <T> Stream<T> empty()`
    - `static <T> Stream<T> generate(Supplier<T> s)`
      - generuje nekonečné streamy
- ```
interface Supplier<T> {  
    T get();  
}
```

# Vytváření streamů

- metody na interfacu Stream (pokrač.)
  - `static <T> Stream<T> iterate(T seed, UnaryOperator<T> f)`
    - generuje nekonečné streamy
    - seed – první element
    - další elementy – `f(seed), f(f(seed)),...`
- `java.nio.files.Files`
  - `static Stream<String> lines(Path path)`
- ...

# Intermediate operace

- **Stream<T> filter(Predicate<? super T> predicate)**
  - vrátí stream s elementy, které „projdou“ predikátem
- **<R> Stream<R> map(Function<? super T, ? extends R> mapper)**
  - vrací stream výsledků funkce aplikované na elementy zdrojového streamu
- **<R> Stream<R> flatMap(Function<? super T, ? extends Stream<? extends R>> mapper)**
  - jako map, ale pokud funkce vrací také stream, výsledek je sloučen v jednom streamu, tj. není to stream streamů

# Intermediate operace

- `Stream<T> skip(long n)`
- `Stream<T> limit(long maxSize)`
- `static <T> Stream<T> concat(Stream<? extends T> a, Stream<? extends T> b)`
  
- `Stream<T> distinct()`
- `Stream<T> sorted()`
- `Stream<T> sorted(Comparator<? super T> comparator)`

# Terminal operace

- `Optional<T> max (Comparator<? super T> comparator)`
- `Optional<T> min (Comparator<? super T> comparator)`
- `Optional<T> findFirst ()`
- `long count ()`
  
- `Optional<T> reduce (BinaryOperator<T> accumulator)`
- `T reduce (T identity, BinaryOperator<T> accumulator)`

# Terminal operace

- `Object[] toArray()`
- `<A> A[] toArray(IntFunction<A[]> generator)`
  
- `<R,A> R collect(Collector<? super T,A,R> collector)`
- `<R> R collect(Supplier<R> supplier, BiConsumer<R,? super T> accumulator, BiConsumer<R,R> combiner)`
  - předpřipravené kolektory
    - `toList`, `toSet`, `toMap`

# Terminal operace

- `void forEach(Consumer<? super T> action)`
- `void forEachOrdered(Consumer<? super T> action)`

# „Primitivní“ streamy

- `interface Stream<T>`
  - nefunguje rozumně na primitivní typy
- `IntStream`
  - pro `int`, ale i pro `byte`, `short`, `char`, `boolean`
- `LongStream`
- `DoubleStream`
  - pro `double` i `float`
- metody na `Stream<T>`
  - `IntStream mapToInt (ToIntFunction<? super T> mapper)`
  - `LongStream mapToLong (ToLongFunction<? super T> mapper)`
  - `DoubleStream mapToDouble (ToDoubleFunction<? super T> mapper)`



# JAVA

Ještě k funkcionálnímu programování

# Funkcionální programování



- funkce ve FP ~ „matematická funkce“
  - má parametry
  - vrací výsledek(ky)
  - **nemá vedlejší efekty!!!**
    - **POZOR: I/O operace jsou také vedlejší efekty**
  - nevyhazují výjimky
    - také lze posuzovat jako vedlejší efekt
  - pokud možno jsou líné
- data (seznamy) jsou nemodifikovatelná
  - funkce vrací nová

# Líné funkce

- příklad

```
class DebugPrint {  
    private boolean debug;  
    public void setDebug(boolean d) { debug = d; }  
    public void println(String s) {  
        if (debug) { System.out.println(s); }  
    }  
}
```

...

```
DebugPrint db = new DebugPrint();
```

...

```
db.println("Name of the user: " + userName);
```

- řetězec je potřeba pouze pokud `debug == true`

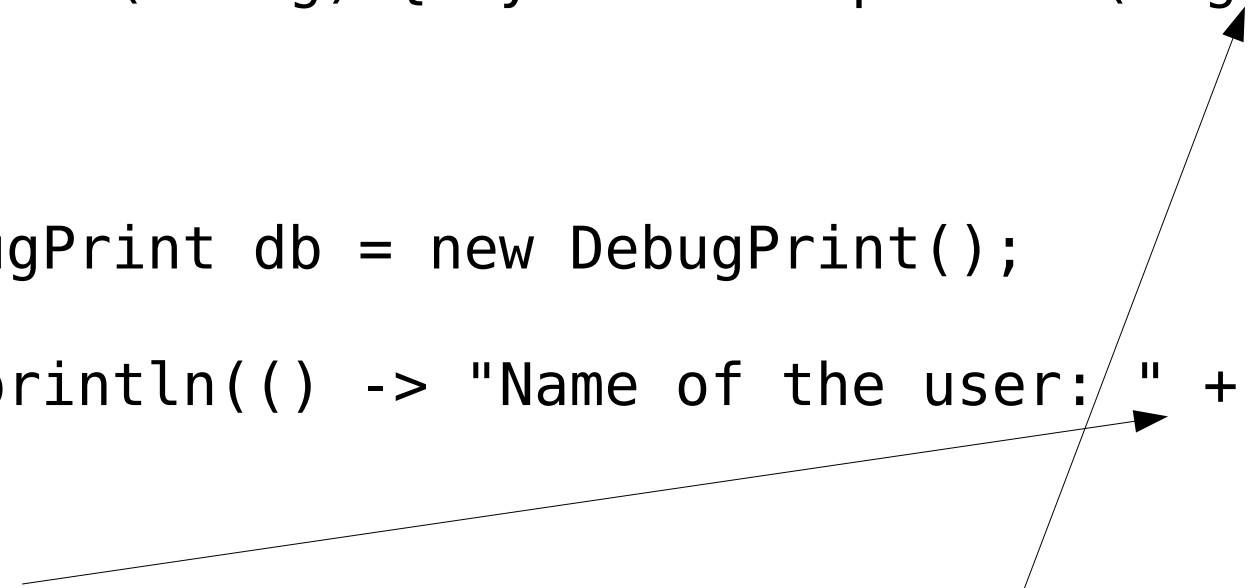
## **ALE vytváří se vždy**

- `new StringBuffer().append(...`

# Líné funkce

- lépe

```
class DebugPrint {  
    private boolean debug;  
    public void setDebug(boolean d) { debug = d; }  
    public void println(Supplier<String> c) {  
        if (debug) { System.out.println(c.get()); }  
    }  
}  
  
...  
DebugPrint db = new DebugPrint();  
...  
db.println(() -> "Name of the user: " + userName);
```



- řetězec se vytvoří pouze pokud je potřeba

# Nevyhazování výjimek

- při chybě vracet speciální hodnotu
- null není vhodné
  - nelze řetězit volání
- Optional<T>
  - třída
  - kontejner pro hodnotu, které může být null
  - metody
    - boolean isPresent()
    - T get()
    - void ifPresent(Consumer<? super T> consumer)
    - ...
  - získání
    - static <T> Optional<T> empty()
    - static <T> Optional<T> of(T value)
    - static <T> Optional<T> ofNullable(T value)



Verze prezentace J09.cz.2019.01

Tato prezentace podléhá licenci [Creative Commons Uved'te autora-Neužívejte komerčně 4.0 Mezinárodní License](https://creativecommons.org/licenses/by-nc/4.0/).