

Výčtové typy a kolekce v Javě, generické typy

Jiří Vokřínek

Katedra počítačů

Fakulta elektrotechnická

České vysoké učení technické v Praze

Přednáška 3

B0B36PJV – Programování v JAVA

Obsah přednášky

Výčtové typy

Kolekce a JFC

Iterátory

Přehled JFC

Generické typy

Výčtové typy

Pojmenované hodnoty

- Vyjmenované hodnoty reprezentují množinu pojmenovaných hodnot
- Historicky se pojmenované hodnoty dají v Javě realizovat jako konstanty

Podobně jako v jiných jazycích

```
public static final int CLUBS = 0;  
public static final int DIAMONDS = 1;  
public static final int HEARTS = 2;  
public static final int SPADES = 3;
```

- Mezi hlavní problémy tohoto přístupu je, že není typově bezpečný

Jak zajistíme přípustné hodnoty příslušné proměnné?

- Například se jedná o hodnoty celých čísel
- Dále nemůžeme jednoduše vytisknout definované hodnoty

Pojmenované hodnoty

- Vyjmenované hodnoty reprezentují množinu pojmenovaných hodnot
- Historicky se pojmenované hodnoty dají v Javě realizovat jako konstanty

Podobně jako v jiných jazycích

```
public static final int CLUBS = 0;  
public static final int DIAMONDS = 1;  
public static final int HEARTS = 2;  
public static final int SPADES = 3;
```

- Mezi hlavní problémy tohoto přístupu je, že není typově bezpečný

Jak zajistíme přípustné hodnoty příslušné proměnné?

- Například se jedná o hodnoty celých čísel
- Dále nemůžeme jednoduše vytisknout definované hodnoty

Výčtové typy

- Java 5 rozšiřuje jazyk o definování výčtového typu
- Výčtový typ se deklaruje podobně jako třída, ale s klíčovým slovem **enum** místo class

```
public enum Suit { CLUBS, DIAMONDS, HEARTS, SPADES }
```

- V základní podobě se jedná o čárkou oddělený seznam jmen reprezentující příslušné hodnoty
- Výčtové typy jsou typově bezpečné

```
public boolean checkClubs(Suit suit) {  
    return suit == Suit.CLUBS;  
}
```

Možné hodnoty jsou kontrolovány kompilátorem při překladu.

<http://docs.oracle.com/javase/tutorial/java/java00/enum.html>

Vlastnosti výčtových typů

- Uložení dalších informací
- Tisk hodnoty
- Načtení všech hodnot výčtového typu
- Porovnání hodnot
- Výčtový typ je objekt
 - Může mít datové položky a metody
 - Výčtový typ má metodu `values()`
 - Může být použit v řídicí struktuře `switch()`

```
import java.awt.Color;

public enum Suit {

    CLUBS(Color.BLACK),
    DIAMONDS(Color.RED),
    HEARTS(Color.BLACK),
    SPADES(Color.RED);

    private Color color;

    Suit(Color c) {
        this.color = c;
    }

    public Color getColor() {
        return color;
    }

    public boolean isRed() {
        return color == Color.RED;
    }
}
```

lec03/Suit

Příklad použití 1/2

```
public class DemoEnum {  
    public boolean checkClubs(Suit suit) {  
        return suit == Suit.CLUBS;  
    }  
    public void start() {  
        Suit suit = Suit.valueOf("SPADES"); //parse string  
        System.out.println("Card: " + suit);  
  
        Suit[] suits = Suit.values();  
        for (Suit s : suits) {  
            System.out.println(  
                "Suit: " + s + " color: " + s.getColor());  
        }  
    }  
    public static void main(String[] args) {  
        DemoEnum demo = new DemoEnum();  
        demo.start();  
    }  
}
```

lec03/DemoEnum

Příklad použití 2/2

■ Příklad výpisu:

```
java DemoEnum  
Card: SPADES color: java.awt.Color[r=255,g=0,b=0]  
suit: CLUBS color: java.awt.Color[r=0,g=0,b=0]  
suit: DIAMONDS color: java.awt.Color[r=255,g=0,b=0]  
suit: HEARTS color: java.awt.Color[r=0,g=0,b=0]  
suit: SPADES color: java.awt.Color[r=255,g=0,b=0]
```

■ Příklad použití v příkazu **switch**

```
Suit suit = Suit.HEARTS;
```

```
switch (suit) {  
    case CLUBS:  
    case HEARTS:  
        // do with black  
        break;  
    case DIAMONDS:  
    case SPADES:  
        // do with red  
        break;  
}
```

Reference na výčet

- Výčet je jen jeden

Singleton

- Referenční proměnná výčtového typu je buď **null** nebo odkazuje na validní hodnotu z výčtu
- Důsledek: pro porovnání dvou referenčních hodnot není nutné používat `equals`, ale lze využít přímo operátor `==`

Jak porovnáváme objekty?

Kolekce a JFC

Kolekce (kontejnery) v Javě

Java Collection Framework (JFC)

- Množina třídy a rozhraní implementující sadu obecných a znovupoužitelných datových struktur
- Navržena a implementována převážně Joshua Blochem

J. Bloch: Effective Java (2nd Edition), Addison-Wesley, 2008

- Příklad aplikace principů objektově orientovaného programování návrhu klasických datových struktur

Dobrý příklad návrhu

- JFC poskytuje unifikovaný rámec pro reprezentaci a manipulacemi s kolekcemi

Kolekce

- Kolekce (též nazývaná kontejner) je objekt, který obsahuje množinu prvků v jediné datové struktuře
- Základními datovými struktury jsou
 - Pole (statické délky) – nevýhody: konečný počet prvků, přístup přes index, implementace datových typů je neflexibilní
 - Seznamy – nevýhody: jednoúčelový program, primitivní struktura
- **Java Collection Framework** – jednotné prostředí pro manipulaci se skupinami objektů
 - Implementační prostředí datových typů **polymorfního charakteru**
 - Typickými skupinami objektů jsou **abstraktní datové typy**: množiny, seznamy, fronty, mapy, tabulky, ...
 - Umožňuje nejen ukládání objektů, získávání a jejich zpracování, ale také výpočet souhrnných údajů apod.
 - Realizuje se prostřednictvím: **rozhraní** a **tříd**

Java Collection Framework (JFC)

- Rozhraní (interfaces) – hierarchie abstraktních datových typů (ADT)
 - Umožňují kolekcím manipulovat s prvky nezávislé na konkrétní implementaci
 - `java.util.Collection`, ...
- Implementace – konkrétní implementace rozhraní poskytují základní podporu pro znovupoužitelné datové struktury
 - `java.util.ArrayList`, ...
- Algoritmy – užitečné metody pro výpočty, hledání, řazení nad objekty implementující rozhraní kolekcí.
 - Algoritmy jsou polymorfní
 - `java.util.Collections`

<http://docs.oracle.com/javase/tutorial/collections>

JFC – výhody

- Výkonné implementace – umožňují rychlé a kvalitní programy, možnosti přizpůsobení implementace
- Jednotné API (*Application Programming Interface*)
 - Standardizace API pro další rozvoj
 - Genericita
- Jednoduchost, konzistentnost (jednotný přístup), rychlé naučení
- Podpora rozvoje sw a jeho znovupoužitelnost

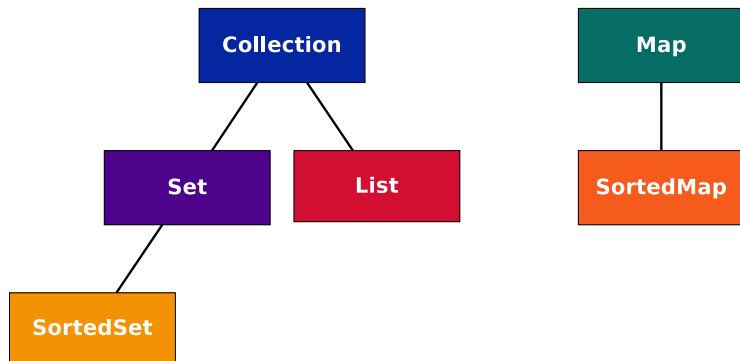
Jednotné API podporuje interoperabilitu i částí vytvořených nezávisle.

- Odstínění od implementačních podrobností

Kromě JFC je dobrý příklad kolekci také například knihovna STL (Standard Template Library) pro C++.

- Nevýhody
 - Rozsáhlejší kód
 - Široká nabídka možností

Struktura rozhraní kolekce



- **Collection** lze získat z **Map** prostřednictvím **Map.values()**
- Některé operace jsou navrženy jako „*optional*“, proto konkrétní implementace nemusí podporovat všechny operace

UnsupportedOperationException

Procházení kolekcí v Javě

■ Iterátory – **iterator**

- Objekt umožňující procházet kolekci
- a selektivně odstraňovat prvky

■ Rozšířený příkaz **for-each**

- Zkrácený zápis, který je přeložen na volání s použitím **`o.iterator()`**

```
public interface Iterator {  
    boolean hasNext();  
    Object next();  
    void remove(); //Optional  
}
```

```
Collection collection =  
    getCollection();  
for (Object o: collection) {  
    System.out.println(o);  
}
```

Iterátory

Iterátor

- Iterátor lze získat voláním metody **iterator** objektu kolekce
- Příklad průchodu kolekce **collection**

```
Iterator it = collection.iterator();  
while(it.hasNext()) {  
    System.out.println(it.next());  
}
```

- Metoda **next()**:

1. Vrací aktuální prvek iterátoru

Po vytvoření iterátoru je to první prvek

2. Postoupí na další prvek, který se stane aktuálním prvkem iterátoru

Iterátor – metody rozhraní

■ Rozhraní **Iterator**

```
public interface Iterator {  
    boolean hasNext();  
    Object next();  
    void remove(); //Optional  
}
```

- **hasNext()** – true pokud iterace má ještě další prvek
- **next()** – vrací aktuální prvek a postoupí na další prvek
 - Vyvolá **NoSuchElementException** pokud již byly navštíveny všechny prvky
- **remove()** – odstraní poslední prvek vrácený **next**
 - Lze volat pouze jednou po volání **next**
 - Jinak vyvolá výjimku **IllegalStateException**
 - Jediný korektní způsob modifikace kolekce během iterování

Iterátor a způsoby implementace

■ Vytvoření kopie kolekce

- + vytvořením privátní kopie nemohou jiné objekty změnit kolekci během iterování
- náročné vytvoření $O(n)$

■ Přímé využití vlastní kolekce

Běžný způsob

- + Vytvoření, **hasNext** a **next** jsou $O(1)$
- Jiný objekt může modifikovat strukturu kolekce, což může vést na nespecifikované chování operací

Rozhraní **Iterable**

- Umožňuje asociovat **Iterator** s objektem
- Především předepisuje metodu

```
public interface Iterable {  
    ...  
    Iterator iterator();  
    ...  
}
```

Iterator: hasNext(); next(); remove(); – jednoduché rozhraní a z toho plynoucí obecnost (genericita).

- V Java 8 rozšíření o další metody

<http://docs.oracle.com/javase/8/docs/api/java/lang/Iterable.html>

- Iterátory v Javě

http://www.tutorialspoint.com/java/java_using_iterator.htm

- Iterator Design Pattern

http://sourcecmaking.com/design_patterns/Iterator/java/1

<http://java.dzone.com/articles/design-patterns-iterator>

Rozhraní **Iterable**

- Umožňuje asociovat **Iterator** s objektem
- Především předepisuje metodu

```
public interface Iterable {  
    ...  
    Iterator iterator();  
    ...  
}
```

Iterator: hasNext(); next(); remove(); – jednoduché rozhraní a z toho plynoucí obecnost (genericita).

- V Java 8 rozšíření o další metody

<http://docs.oracle.com/javase/8/docs/api/java/lang/Iterable.html>

- Iterátory v Javě

http://www.tutorialspoint.com/java/java_using_iterator.htm

- Iterator Design Pattern

http://sourcecmaking.com/design_patterns/Iterator/java/1

<http://java.dzone.com/articles/design-patterns-iterator>

Iterátory a jejich zobecnění

- Iterátory mohou být aplikovány na libovolné kolekce
- Iterátory mohou reprezentovat posloupnost, množinu nebo mapu
- Mohou být implementovány použitím polí nebo spojových seznamů
- Příkladem rozšíření pro spojové seznamy je **ListIterator**, který umožňuje
 - Přístup k celočíselné pozici (index) prvku
 - Dopředný (forward) nebo zpětný (backward) průchod
 - Změnu a vložení prvků

add, hasNext, hasPrevious, previous, next, nextIndex, previousIndex, set, remove

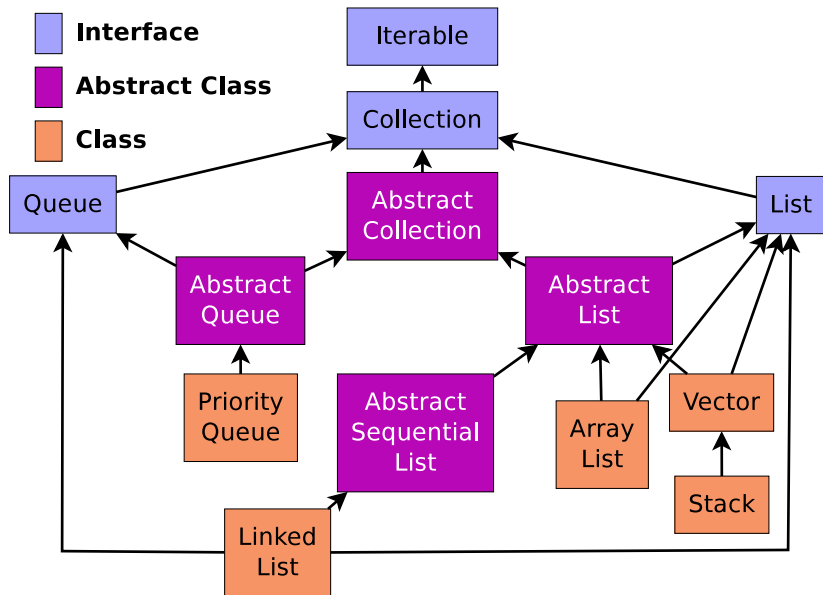
Iterátory a jejich zobecnění

- Iterátory mohou být aplikovány na libovolné kolekce
- Iterátory mohou reprezentovat posloupnost, množinu nebo mapu
- Mohou být implementovány použitím polí nebo spojových seznamů
- Příkladem rozšíření pro spojové seznamy je **ListIterator**, který umožňuje
 - Přístup k celočíselné pozici (index) prvku
 - Dopředný (forward) nebo zpětný (backward) průchod
 - Změnu a vložení prvků

add, hasNext, hasPrevious, previous, next, nextIndex, previousIndex, set, remove

Přehled JFC

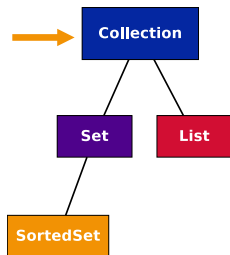
JFC overview



Rozhraní Collection

- Co možná nejobecnější rozhraní pro předávání kolekcí objektů

```
public interface Collection {  
    // Basic Operations  
    int size();  
    boolean isEmpty();  
    boolean contains(Object element);  
    boolean add(Object element);    // Optional  
    boolean remove(Object element); // Optional  
    Iterator iterator();  
  
    // Bulk Operations  
    boolean containsAll(Collection c);  
    boolean addAll(Collection c);    // Optional  
    boolean removeAll(Collection c); // Optional  
    boolean retainAll(Collection c); // Optional  
    boolean clear();                // Optional  
  
    // Array Operations  
    Object[] toArray();  
    <T> T[] toArray(T a[]);
```

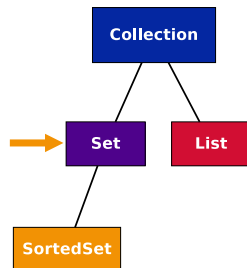


Třída **AbstractCollection**

- Základní implementace rozhraní **Collection**
- Pro **neměnitelnou** kolekci je nutné implementovat
 - **iterator** spolu s **hasNext** a **next**
 - **size**
- Pro **měnitelnou** kolekci je dále nutné implementovat
 - **remove** pro **iterator**
 - **add**

Rozhraní Set

- **Set** je **Collection**, ve které nejsou duplicitní prvky
- Využívá metod **equals** a **hashCode** pro identifikaci stejných prvků
- Dva objekty **Set** jsou stejné pokud obsahují stejné prvky
- JDK implementace
 - **HashSet** – velmi dobrý výkon (využívá hašovací tabulku)
 - **TreeMap** – garantuje uspořádání, red-black strom



- **podmnožina**
`s1.containsAll(s2)`
- **sjednocení**
`s1.addAll(s2)`
- **průnik**
`s1.retainAll(s2)`
- **rozdíl**
`s1.removeAll(s2)`

Rozhraní **List**

- Rozšiřuje rozhraní **Collection** pro model dat jako **uspořádanou posloupnost** prvků, indexovanou celými čísly udávající pozici prvku (od 0)

```
public interface List extends Collection {  
    // Positional Access  
    Object get(int index);  
    Object set(int index, Object element); // Optional  
    void add(int index, Object element); // Optional  
    Object remove(int index); // Optional  
    abstract boolean addAll(int index,  
                           Collection c); // Optional  
  
    // Search  
    int indexOf(Object o);  
    int lastIndexOf(Object o);  
  
    // Iteration  
    ListIterator listIterator();  
    ListIterator listIterator(int index);  
  
    // Range-view  
    List subList(int from, int to);  
}
```

```
public interface ListIterator  
    extends Iterator {  
  
    boolean hasNext();  
    Object next();  
  
    boolean hasPrevious();  
    Object previous();  
  
    int nextIndex();  
    int previousIndex();  
  
    void remove(); // Optional  
    void set(Object o); // Optional  
    void add(Object o); // Optional  
}
```

- Většina polymorfních algoritmů v JFC je aplikovatelná na **List** a ne na **Collection**.

```
sort(List); shuffle(List); reverse(List); fill(List, Object);  
copy(List dest, List src); binarySearch(List, Object);
```

Rozhraní **AbstractList**

- Základní implementace rozhraní **List**
- Pro **neměnitelný** list je nutné implementovat
 - **get**
 - **size**
- Pro **měnitelný** list je dále nutné implementovat
 - **set**
- Pro měnitelný list **variabilní délky** je dále nutné implementovat
 - **add**
 - **remove**

Třída `ArrayList`

- Náhodný přístup k prvkům implementující rozhraní `List`
- Používá pole (array)
- Umožňuje automatickou změnu velikosti pole
- Přidává metody:
 - `trimToSize()`
 - `ensureCapacity(n)`
 - `clone()`
 - `removeRange(int fromIndex, int toIndex)`
 - `writeObject(s)` – zápis seznamu do výstupního proudu `s`
 - `readObject(s)` – načtení seznamu ze vstupního proudu `s`

<http://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html>

- `ArrayList` obecně poskytuje velmi dobrý výkon (využívá hašovací tabulky)
- `LinkedList` může být někdy rychlejší
- `Vector` – *synchronizovaná* „varianta“ `ArrayList`, ale lze též přes *synchronized wrappers*

Třída **ArrayList**

- Náhodný přístup k prvkům implementující rozhraní **List**
- Používá pole (array)
- Umožňuje automatickou změnu velikosti pole
- Přidává metody:
 - **trimToSize()**
 - **ensureCapacity(n)**
 - **clone()**
 - **removeRange(int fromIndex, int toIndex)**
 - **writeObject(s)** – zápis seznamu do výstupního proudu **s**
 - **readObject(s)** – načtení seznamu ze vstupního proudu **s**

<http://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html>

- **ArrayList** obecně poskytuje velmi dobrý výkon (využívá hašovací tabulky)
- **LinkedList** může být někdy rychlejší
- **Vector** – **synchronizovaná** „varianta“ **ArrayList**, ale lze též přes *synchronized wrappers*

Rozhraní Map

- **Map** je kolekce, která mapuje klíče na hodnoty
- Každý klíč může mapovat nejvýše jednu hodnotu
- Standardní JDK implementace:
 - **HashMap** – uloženy v hašovací tabulce
 - **TreeMap** – garantuje uspořádání, red-black strom
 - **Hashtable** – hašovací tabulka implementující rozhraní **Map**
*synchronizovaný přístup, neumožňuje **null** prvky a klíče*

```
public interface Map {  
    // Basic Operations  
    Object put(Object key, Object value);  
    Object get(Object key);  
    Object remove(Object key);  
    boolean containsKey(Object key);  
    boolean containsValue(Object value);  
    int size();  
    boolean isEmpty();  
    // Bulk Operations  
    void putAll(Map t);  
    void clear();  
    // Collection Views  
    public Set keySet();  
    public Collection values();  
    public Set entrySet();  
    // Intergace for entrySet  
    // elements  
    public interface Entry {  
        Object getKey();  
        Object getValue();  
        Object setValue(Object val);  
    }  
}
```

Třída `SortedSet`

- `SortedSet` je `Set`, který udržuje prvky v rostoucím pořadí tříděné podle:
 - přirozeného pořadí prvků, nebo dle implementace `Comparator` předaného při vytvoření
- Dále `SortedSet` nabízí operace pro
 - `Range-view` – rozsahové operace
 - `Endpoints` – vrací první a poslední prvek
 - `Comparator access` – vrací `Comparator` použitý pro řazení

```
public interface SortedSet extends Set {  
    // Range-view  
    SortedSet subSet(Object fromElement, Object toElement);  
    SortedSet headSet(Object toElement);  
    SortedSet tailSet(Object fromElement);  
  
    // Endpoints  
    Object first();  
    Object last();  
  
    //Comparator access  
    Comparator comparator();  
}
```

Implementace kolekcí

■ Obecně použitelné implementace

Veřejné (**public**) třídy, které poskytují základní implementaci hlavních rozhraní kolekcí, například **ArrayList**, **HashMap**

■ Komfortní implementace

Mini-implementace, typicky dostupné přes takzvané statické tovární metody (`static factory method`), které poskytují komfortní a efektivní implementace pro speciální kolekce, například **`Collections.singletonList()`**.

■ Zapouzdřující implementace

Implementace kombinované s jinými implementacemi (s obecně použitelnými implementacemi) a poskytují tak dodatečné vlastnosti, např. **`Collections.unmodifiableCollection()`**

Obecně použitelné implementace

- Pro každé rozhraní (kromě obecného rozhraní Collection) jsou poskytovány dvě implementace

		<i>Implementace</i>			
		Hašovací tabulky	Variabilní pole	Vyvážený strom	Spojový seznam
	Set	HashSet		TreeSet	
<i>Rozhraní</i>	List		ArrayList, Vector		LinkedList
	Map	HashMap		TreeMap	

Generické typy

Generické typy a nevýhody polymorfismu

- Flexibilita (znovupoužitelnost) tříd je tradičně v Javě řešena dědičností a polymorfismem
- Polymorfismus nám dovoluje vytvořit třídu (např. nějaký kontejner), která umožňuje uložit libovolný objekt (jako referenci na objekt typu **Object**)

Např. **ArrayList** z JFC

- Dynamická vazba polymorfismu však neposkytuje kontrolu správného (nebo očekávaného) typu během kompilace
- Případná chyba v důsledku „špatného“ typu se tak projeví až za běhu programu
- Tato forma polymorfismu také vyžaduje explicitní přetypování objektu získaného z obecné kolekce

Například zmiňovaný **ArrayList** pro ukládání objektů typu **Object**.

Příklad použití kolekce `ArrayList`

```
package cz.cvut.fel.pr2;

import java.util.ArrayList;

public class Simulator {
    World world;
    ArrayList participants;

    Simulator(World world) {
        this.world = world;
        participants = new ArrayList();
    }

    public void nextRound() {
        for (int i = 0; i < participants.size(); ++i) {
            Participant player = (Participant) participants.get(i);
            Bet bet = world.doStep(player);
        }
    }
}
```

- Explicitní přetypování (`Participant`) je nutné.

Generické typy

- Java 5 dovoluje použít generických tříd a metod
- Generický typ umožňuje určit typ instance tříd, které lze do kolekce ukládat
- Generický typ tak poskytuje statickou typovou kontrolu během překladu
- Generické typy představují parametrizované definice třídy typu nějaké datové položky
- Parametr typu se zapisuje mezi `<>`, například

```
List<Participant> partList = new ArrayList<Participant>();
```

<http://docs.oracle.com/javase/tutorial/java/generics/index.html>

Příklad použití parametrizované kolekce **ArrayList**

```
package cz.cvut.fel.pr2;

import java.util.ArrayList;

public class Simulator {
    World world;
    ArrayList<Participant> participants;

    Simulator(World world) {
        this.world = world;
        participants = new ArrayList();
    }

    public void nextRound() {
        for (int i = 0; i < participants.size(); ++i) {
            Participant player = participants.get(i);
            Bet bet = world.doStep(player);
        }
    }
}
```

- Explicitní přetypování (**Participant**) není nutné

Příklad – generický a negenerický typ

```
ArrayList participants;  
participants = new ArrayList();  
participants.push(new PlayerRed());
```

```
// vložit libovolny objekt je mozne  
participants.push(new Bet());
```

```
ArrayList<Participant> participants2;  
participants2 = new ArrayList<Participant>();  
participants2.push(new PlayerRed());
```

```
// nelze prelozit  
// typova kontrola na urovni prekladace  
participants2.push(new Bet());
```

Příklad parametrizované třídy

```
import java.util.List;  
import java.util.ArrayList;  
  
class Library<E> {  
    private List<E> resources = new ArrayList<E>();  
  
    public void add(E x) {  
        resources.add(x);  
    }  
  
    public E getLast() {  
        int size = resources.size();  
        return size > 0 ? resources.get(size-1) : null;  
    }  
}
```

Generické metody

- Generické metody mohou být členy generických tříd nebo normálních tříd

```
public class Methods {  
    public <T> void print(T o) {  
        System.out.println("Print Object: " + o);  
    }  
    public static void main(String[] args) {  
        Integer i = 10;  
        Double d = 5.5;  
  
        Methods m1 = new Methods();  
  
        m1.print(i);  
        m1.print(d);  
  
        m1.<Integer>print(i);  
  
        /// nelze -- typova kontrola  
        m1.<Integer>print(d);  
    }  
}
```

lec03/Methods

Příklad implementace spojového seznamu

- Třída **LinkedList** pro uchování objektů
- Implementujeme metody **push** a **print**

```
public class LinkedList {  
    class ListNode {  
        ListNode next;  
        Object item;  
        ListNode(Object item) { ... }  
    }  
  
    ListNode start;  
  
    public LinkedList() { ... }  
    public LinkedList push(Object obj) { ... }  
    public void print() { ... }  
}
```

lec03/LinkedList

Příklad použití

- Do seznamu můžeme přidávat libovolné objekty, např. **String**
- Tisk seznamu však realizuje vlastní metodou **print**

```
LinkedList lst = new LinkedList();  
lst.push("Joe");  
lst.push("Barbara");  
lst.push("Charles");  
lst.push("Jill");  
  
lst.print();
```

- Využití konstrukce **for-each** vyžaduje, aby třída **LinkedList** implementovala rozhraní **Iterable**

```
for (Object o : lst) {  
    System.out.println("Object:" + o);  
}
```


Rozhraní `Iterable` a `Iterator`

- Rozhraní `Iterable` předepisuje metodu `iterator`, která vrací iterátor instanci třídy implementující rozhraní `Iterator`
- `Iterator` je objekt umožňující postupný přístup na položky seznamu
- Rozšíříme třídu `LinkedList` o implementaci rozhraní `Iterable` a vnitřní třídu `LLIterator` implementující rozhraní `Iterator`

<http://docs.oracle.com/javase/tutorial/java/java00/innerclasses.html>

```
public class LinkedListIterable extends LinkedList
    implements Iterable {

    private class LLIterator implements Iterator { ... }

    @Override
    public Iterator iterator() {
        return new LLIterator(start); //kurzor <- start
    }
}
```

lec03/LinkedListIterable

Implementace rozhraní **Iterator**

- Rozhraní **Iterator** předepisuje metody **hasNext** a **next**

```
private class LLIterator implements Iterator {  
    private ListNode cur;  
  
    private LLIterator(ListNode cur) {  
        this.cur = cur; // nastaveni kurzoru  
    }  
  
    @Override  
    public boolean hasNext() {  
        return cur != null;  
    }  
  
    @Override  
    public Object next() {  
        if (cur == null) {  
            throw new NoSuchElementException();  
        }  
        Object ret = cur.item;  
        cur = cur.next; //move forward  
        return ret;  
    }  
}
```

lec03/LinkedListIterable

Příklad využití iterátoru v příkazu **for-each**

- Nahradíme implementace **LinkedList** za **LinkedListIterable**

```
// LinkedList lst = new LinkedList();
LinkedListIterable lst = new LinkedListIterable();
lst.push("Joe");
lst.push("Barbara");
lst.push("Charles");
lst.push("Jill");

lst.print();

for (Object o : lst) {
    System.out.println("Object:" + o);
}
```

lec03/LinkedListDemo

Spojový seznam specifických objektů

- Do spojového seznamu **LinkedList** můžeme ukládat libovolné objekty, což má i přes své výhody také nevýhody:
 - Nemáme statickou typovou kontrolu prvků seznamu
 - Musíme objekty explicitně přetypovat, například pro volání metody **toNiceString** objektu **Person**

```
public class Person {  
  
    private final String name;  
    private final int age;  
  
    public Person(String name, int age) { ... }  
    public String toNiceString() {  
        return "Person name: " + name + " age: " + age;  
    }  
}
```

Příklad přetypování na **Person**

```
LinkedListIterable lst = new LinkedListIterable();  
lst.push(new Person("Joe", 30));  
lst.push(new Person("Barbara", 40));  
lst.push(new Person("Charles", 50));  
lst.push(new Person("Jill", 60));  
  
for (Object o : lst) {  
    System.out.println("Object: " + ((Person)o).  
        toNiceString());  
}
```

Generický typ

- Využitím generického typu můžeme předepsat konkrétní typ objektu
- Vytvoříme proto LinkedList přímo jako generický typ deklarací `class LinkedListGeneric<E>` a záměnou **Object** za **E**

```
public class LinkedListGeneric<E> {  
    class ListNode {  
        ListNode next;  
        E item;  
        ListNode(E item) { ... }  
    }  
    ListNode start  
    public LinkedListGeneric() { ... }  
    public LinkedListGeneric push(E obj) { ... }  
    public void print() { ... }  
}
```

lec03/LinkedListGeneric

Generický typ – `Iterable` a `Iterator`

- Podobně upravíme odvozený iterátor a doplníme typ také v rozhraní `Iterable` a `Iterator`

```
public class LinkedListGenericIterable<E> extends
    LinkedListGeneric<E> implements Iterable<E> {

    // vnitřní třída pro iterátor
    private class LLIterator implements Iterator<E> { ... }

    @Override
    public Iterator iterator() {
        return new LLIterator(start);
    }
}
```

lec03/LinkedListGenericIterable

Generický typ – Iterator

- Implementace iterátoru je identická jako v případě

`LinkedListIterable`

```
private class LLIterator implements Iterator<E> {  
    private ListNode cur;  
  
    private LLIterator(ListNode cur) {  
        this.cur = cur;  
    }  
    @Override  
    public boolean hasNext() {  
        return cur != null;  
    }  
    @Override  
    public E next() {  
        if (cur == null) {  
            throw new NoSuchElementException();  
        }  
        E ret = cur.item;  
        cur = cur.next; //move forward  
        return ret;  
    }  
}
```

lec03/LinkedListGenericIterable

Příklad použití

```
LinkedListGenericIterable<Person> lst = new  
LinkedListGenericIterable();
```

```
lst.push(new Person("Joe", 30));  
lst.push(new Person("Barbara", 40));  
lst.push(new Person("Charles", 50));  
lst.push(new Person("Jill", 60));
```

```
lst.print();
```

```
for (Person o : lst) {  
    System.out.println("Object: " + o.toNiceString());  
}
```

lec03/LinkedListGenericDemo

Shrnutí přednášky

Diskutovaná témata

- Výčtové typy – **enum**
 - Kolekce – **Java Collection Framework** (JFC)
 - Iterátory
 - Generické typy
-
- Příště: Výjimky a soubory

Diskutovaná témata

- Výčtové typy – **enum**
 - Kolekce – **Java Collection Framework** (JFC)
 - Iterátory
 - Generické typy
-
- **Příště: Výjimky a soubory**