# OMO

## 3b - Funkcionální programování v Java

- Pure funkce
- Funkce první třídy a funkce vyššího řádu
- Lambda expressions
- Closures
- Curying
- Representation transparency
- Lazy evaluation

Ing. David Kadleček, PhD.
kadlecd@fel.cvut.cz, david.kadlecek@cz.ibm.com

Verze 18.10.2020

# Klíčová pravidla funkcionálního programování

- **Immutable**: to už známe :-)
- **No implicit state** (bez implicitního stavu): nesmí mít skrytý či implicitní stav. Stav musí být exlicitní a transparentní (viditelný):
- **Pure functions:**
  a. ***No side effects*** (bez vedlejších efektů): Funkce či operace nesmí měnit vnější stav (jiné než vlastní lokální proměnné) - funkce pouze vrací hodnotu funkci, která ji volá
  b. ***Idempotence*** = funkce vrací hodnoty, které jsou závislé pouze na argumentech předaných při volání => nezávisí tedy na ničem jiném. Pokud funkci zavoláte vícekrát s těmi samými parametry, tak bude vždy vracet to samé

**Pure funkcionální jazyky neumožňují mutabilitu a side efekty - příklad Haskel**
**Vyšší jazyky to povolují - příklad Java**

# Pure funkce

Příklad pure funkce:

```
public class ObjectWithPureFunction{
   public int sum(int a, int b) {
       return a + b;
   }
}
```

Příklad **non** pure funkce:

```
public class ObjectWithNonPureFunction{
   private int value = 0;
   public int add(int nextValue) {
       this.value += nextValue;
       return this.value;
   }
}
```

# Lambda expressions

Lambda expression je forma ve tvaru: **(seznam argumentů funkce) -> tělo funkce**

```
/* Java 1.8+ Funkce, která sečte dvě čísla */
(int x, int y) -> x + y
/* Bezparametrická funkce */
() -> 42
/* Procedura */
(String s) -> { System.out.println(s); }
/* Komparátor */
List<Person> personList = Person.createShortList();
Collections.sort(personList, (Person p1, Person p2) -> p1.getSurName().compareTo(p2.getSurName()));
```

- Lambda výrazy se používají především k definování implementace funkčního **rozhraní s jedinou metodou** tzv. **inline formou** což vede k výrazné redukci kódu a přináší např. do Javy některé výhody funkcionálního programování.
- Lambda expression v programovacích jazycích je funkce, kterou je možné definovat a volat bez bindingu s identifikátorem

*Pozn. Lambda calculus je formální systém matematické logiky a informatiky pro vyjádření výpočtu pomocí bindingu proměnných a jejich substituce - nezaměňovat s lambda expressions*

# Funkce první třídy a funkce vyššího řádu

Objektem **první třídy** (first-class citizen) v programovacích jazycích je entita, která podporuje následující operace: být předána jako parametr, přiřazená proměnné a být vrácená z funkce. Funkce první třídy je tedy taková funkce, která splňuje výše uvedené vlastnosti.

*Pozn. Metody a  třídy, jelikož to nejsou hodnoty, tak jsou považovány za objekty druhé třídy.*

**Funkce vyššího řádu** je funkce, které splňuje přinejmenším jednu z vlastností:

- Jedním či více parametry je funkce
- Vrací funkci jako parametr

# Funkce první třídy a funkce vyššího řádu

Klasický přístup:

```
public List filterPersonByAge(List<Person>
list) {
   List result = new ArrayList();
   for (Person person : list) {
       if(p.age > 65){
           result.add(person);
       }
   }
   return result;
}
```

Filtruji a vracím každého, kdo je starší než 65 let. Problém je, že když chci filtrovat podle jiného atributu, tak musím celý tento kód zduplikovat, abych modifikoval pouze jednu řádku kódu.

# Funkce první třídy a funkce vyššího řádu

Přepis v Java 1.8+:

```
import java.util.function.Predicate;
public class HigherOrderFunctionExample {

   public List filterPerson(List<Person> list, Predicate<Person> p) {
       List result = new ArrayList();
       for (Person person : list) {
           if(p.test(person)) {result.add(person);}
       }
       return result;
   }
   public boolean ageFilter(Person p){
       return p.age > 65;
   }
}
```

Přidali jsme nový parametr typu *Predicate*, který obsahuje podmínku, kterou testujeme. Dále pak metoda *ageFilter*, kterou vkládáme jako parametr *p*.

Funkce je volána následovně: `filterPerson(personList, FirstClassFunctionExample::ageFilter);`

Jestliže chceme filtrovat podle jiného atributu, tak uděláme drobnou změnu do implementace filtru a vlastní kód na filtrování je přepoužit.

# Funkce první třídy a funkce vyššího řádu

Java neumožňuje pracovat s funkcí jako s typem.

Místo toho použilo tzv. Funkcionální Interfacy :

- Function

```
public interface Function<T,R> {
   public <R> apply(T parameter);
}
```

- Predicate

```
public interface Predicate {
   boolean test(T t);
}
```

- UnaryOperator T->T
- BinaryOperator T,S->R
- Supplier ()->T
- Consumer T->()

# Funkce první třídy a funkce vyššího řádu

Přepis v Java 1.8+:

```
import java.util.function.Predicate;
public class HigherOrderFunctionExample {

   public List filterPerson(List<Person> list, Predicate<Person> p) {
       List result = new ArrayList();
       for (Person person : list) {
           if(p.test(person)) {result.add(person);}
       }
       return result;
   }
   public boolean ageFilter(Person p){
       return p.age > 65;
   }
}
```

Přidali jsme nový parametr typu *Predicate*, který obsahuje podmínku, kterou testujeme. Dále pak metoda *ageFilter*, kterou vkládáme jako parametr *p*.

Funkce je volána následovně:

```
filterPerson(personList, FirstClassFunctionExample::ageFilter);
```

Jestliže chceme filtrovat podle jiného atributu, tak uděláme drobnou změnu do implementace filtru a vlastní kód na filtrování je přepoužit.

# Příklad Kotlin

```
fun <T> ArrayList<T>.filterOnCondition(condition: (T) -> Boolean): ArrayList<T>{
    val result = arrayListOf<T>()
        for (item in this){
            if (condition(item)){
                result.add(item)
            }
        }
        return result
    }
}

fun isMultipleOf (number: Int, multipleOf : Int): Boolean{
    return number % multipleOf == 0
}

var list = arrayListOf<Int>()
for (number in 1..10){
    list.add(number)
}
var resultList = list.filterOnCondition { isMultipleOf(it, 5) }
```

# Closures

Closure je funkce, která si při deklaraci vytvoří lokální proměnnou, kterou si vezme z kontextu ve kterém je

```
public class ClosureSample {
    // this is a higher-order-function that returns an instance of Function interface
    Function<Integer, Integer> add(final int x) {
        // The lambda expression is returned here as closure
        // x is obtained from the outer scope of this method which is declared as final
        return y -> x + y;
    }
    public static void main(String[] args) {
        ClosureSample sample = new ClosureSample();
        // we are currying the add method to create more variations
        Function<Integer, Integer> add10 = sample.add(10);
        Function<Integer, Integer> add20 = sample.add(20);
        Function<Integer, Integer> add30 = sample.add(30);
        System.out.println(add10.apply(5));
        System.out.println(add20.apply(5));
        System.out.println(add30.apply(5));
    }
}
```

# Currying

Currying spočívá ve vyhodnocování argumentů funkce per partes, kdy po každém kroku získáme funkci, která má o jeden argument méně.

Např. pro funkci

***f(x, y, z) = x * y + z***

můžeme aplikovat argumenty  3, 4, 5 a dostaneme:

***f(3, 4, 5) = 3 * 4 + 5 = 17***

Současně ale můžeme aplikovat pouze 3 a získáme novou funkci *f*

***(3, y, z) = g(y, z) = 3 * y + z***

currying podruhé pro 4 nám dá:

***g(4, z) = h(z) = 3 * 4 + z***

# Currying - zanoření volání

Příklad vytvoření složené funkce při deklaraci:

```
/*Java 1.8+*/
public class Currying {
   public void currying() {
       // Create a function that adds 2 integers
       BiFunction<Integer,Integer,Integer> adder = ( a, b ) -> a + b ;
       // And a function that takes an integer and returns a function
       Function<Integer,Function<Integer,Integer>> currier = a -> b -> adder.apply( a, b ) ;
       // Call apply 4 to currier (to get a function back)
       Function<Integer,Integer> curried = currier.apply( 4 ) ;
       // Results
       System.out.printf( "Curry : %d\n", curried.apply( 3 ) ) ; // ( 4 + 3 )
   }
```

# Currying - kompozice

Vytvoření složené funkce ex post po jejich deklaraci:

```
public void composition() {
        // A function that adds 3
        Function<Integer,Integer> add3   = (a) -> a + 3 ;
        // And a function that multiplies by 2
        Function<Integer,Integer> times2 = (a) -> a * 2 ;
        // Compose add with times
        Function<Integer,Integer> composedA = add3.compose( times2 ) ;
        // And compose times with add
        Function<Integer,Integer> composedB = times2.compose( add3 ) ;
        // Results
        System.out.printf( "Times then add: %d\n", composedA.apply( 6 ) ) ;  // ( 6 * 2 ) + 3
        System.out.printf( "Add then times: %d\n", composedB.apply( 6 ) ) ;  // ( 6 + 3 ) * 2
}
public static void main( String[] args ) {
        new Currying().currying() ;
        new Currying().composition() ;
        }
}
```

# Referential transparency

Vychází z idempotentnosti pure funkcí. Hezkým důsledkem toho je, že mohu volání funkce nahradit hodnotou, kterou funkce vrátila naposledy. Tzv. ***memoizace*** nebo ***caching funkčního volání***, abychom nemuseli provádět vícekrát tu samou funkci

# Lazy evaluace

Process kdy zpozdím vyhodnocení výrazu až do doby než potřebuju výsledek. Opak **lazy** je **eager**
Java ve většině případů funguje eager - kromě operandů &&, || and ?:, které jsou lazy. Lambda expressions umožňují psát “lazy” kód v Java.

```
public class EagerSample {
   public static void main(String[] args) {
       System.out.println(addOrMultiply(true, add(4), multiply(4))); // 8
       System.out.println(addOrMultiply(false, add(4), multiply(4))); // 16
   }
   static int add(int x) {
       System.out.println("executing add");
       return x + x;
   }
   static int multiply(int x) {
       System.out.println("executing multiply");
       return x * x;
   }
   static int addOrMultiply(boolean add, int onAdd, int onMultiply) {
       return (add) ? onAdd : onMultiply;
   }
}
```

```
executing add
executing multiply
8
executing add
executing multiply
16
```

# Lazy evaluace

```
public class LazySample {
   public static void main(String[] args) {
       // This is a lambda expression behaving as a closure
       Function<Integer, Integer> add = t -> {
           System.out.println("executing add");
           return t + t;
       };
       // This is a lambda expression behaving as a closure
       Function<Integer, Integer> multiply = t -> {
           System.out.println("executing multiply");
           return t * t;
       };
       // Lambda closures are passed instead of plain functions
       System.out.println(addOrMultiply(true, add, multiply, 4));
       System.out.println(addOrMultiply(false, add, multiply, 4));
   }
   // This is a higher-order-function
   static <T, R> R addOrMultiply( boolean add, Function<T, R> onAdd,
                                  Function<T, R> onMultiply, T t) {
           return (add ? onAdd.apply(t) : onMultiply.apply(t));
   }
}
```

```
executing add
8
executing multiply
16
```