

# Java

## Enum

# Výčty

- **<= Java 1.4**

```
public static final int COLOR_BLUE = 0;  
public static final int COLOR_RED = 1;  
public static final int COLOR_GREEN = 2;
```

- **možné problémy**

- typová (ne)bezpečnost
- žádný namespace
- konstanty napevno přeložené v klientech
- při výpisu jen hodnoty

# Enum

```
public enum Color { BLUE, RED, GREEN }  
...  
public Color clr = Color.BLUE;
```

- „normální“ třída
  - atributy, metody, i metoda main
  - potomek třídy `java.lang.Enum`
  - pro každou konstantu - jedna instance
    - `public static final` atribut
    - `protected` konstruktor

# „Enum bez enumu“

- jak udělat enum v Java 1.4
  - (a jak je enum implementovaný)

```
class Color {
    private int value;

    public static final Color RED = new Color(0);
    public static final Color GREEN = new Color(1);
    public static final Color BLUE = new Color(2);

    private Color(int v) {
        value = v;
    }
    ...
}
```

# java.lang.Enum

```
public abstract class Enum <E> extends  
    Enum<E>> { ... }
```

- **metody**
  - `String name()`
  - `int ordinal()`
- **každý enum má metodu `values()`**
  - **vrací pole se všemi konstantami**

```
public Colors clr = Colors.BLUE;  
System.out.println(clr);    →  BLUE
```

# Atributy a metody

```
public enum Planet {
    MERCURY (3.303e+23, 2.4397e6),
    VENUS (4.869e+24, 6.0518e6),
    EARTH (5.976e+24, 6.37814e6),
    ...

    private final double mass;
    private final double radius;

    Planet(double mass, double radius) {
        this.mass = mass;
        this.radius = radius;
    }

    double surfaceGravity() {
        return G * mass / (radius * radius);
    }
}
```

# Atributy a metody

- příklad

```
public enum Operation {
    PLUS, MINUS, TIMES, DIVIDE;

    double eval(double x, double y) {
        switch(this) {
            case PLUS:    return x + y;
            case MINUS:   return x - y;
            case TIMES:   return x * y;
            case DIVIDE:  return x / y;
        }
        throw new AssertionError("Unknown op: " + this);
    }
}
```

# Atributy a metody

- abstraktní metody
- konkrétní implementace u každé konstanty

```
public enum Operation {  
    PLUS { double eval(double x, double y) { return x+y; }},  
    MINUS { double eval(double x, double y) { return x-y; }},  
    TIMES { double eval(double x, double y) { return x*y; }},  
    DIVIDE { double eval(double x, double y) { return x/y;}};  
  
    abstract double eval(double x, double y);  
}
```



# enum

- nelze dědit
  - ~~enum MoreColors extends Colors~~
- proč?

```
enum Color { Red, Green }
```



```
final class Color extends java.lang.Enum<Color> {  
    public static final Color Red;  
    public static final Color Green;  
    ...  
}
```

# Java

## Proměnný počet parametrů



- „tři tečky“
- pouze jako poslední parametr
- lze předat pole nebo seznam parametrů
- v metodě dostupné jako pole

```
void argtest(Object... args) {  
    for (int i=0;i <args.length; i++) {  
        System.out.println(args[i]);  
    }  
}  
  
argtest("Ahoj", "jak", "se", "vede");  
argtest(new Object[] {"Ahoj", "jak", "se",  
    "vede"});
```

- metoda printf
  - System.out.printf("%s %d\n", user, total);

# Příklad

- Jsou volání ekvivalentní?

```
argtest("Ahoj", "jak", "se", "vede");  
argtest(new Object[] {"Ahoj", "jak", "se", "vede"});  
argtest((Object) new Object[] {"Ahoj", "jak", "se",  
    "vede"});
```

- a) Všechna ekvivalentní
- b) Ekvivalentní 1. a 2.
- c) Ekvivalentní 2. a 3.
- d) Každé dělá něco jiného

# JAVA

## Anotace

# Anotace

- (metadata)
- od Java 5
- umožňují přidat informace k elementům v programu (ke třídám, metodám, atributům,...)
  - obecně – lze použít všude tam, kde lze napsat nějaký modifikátor viditelnosti
    - ale i jinde
- zapisují se **@JmenoAnotace**
- lze definovat vlastní
  - určit, kde je lze napsat, jak používat,...
- předdefinované anotace v balíku java.lang
  - @Deprecated
  - @Override
  - @SuppressWarnings
  - ...

# Anotace

- mohou mít parametry

```
@Deprecated(since="1.2", forRemoval=true)
```


- parametry mohou mít implicitní hodnoty
    - tj. není nutné uvést hodnotu
- ```
@Deprecated
```

- kde lze použít
  - třídy, atributy, metody, ...
  - parametry metod, balíčky
  - použití typů
- lze omezit v definici anotace


# Předdefinované anotace

- **@Override**
  - označení, že metoda předefinovává metodu z předka
  - pokud nic nepředefinovává => kompilátor odmítne třídu přeložit
  - použití je volitelné (nicméně silně doporučené)


```
class A {  
    public void foo() {}  
}  
class B extends A {  
    @Override  
    public void foo() {}  
}
```



```
interface Ice {  
    void foo() {}  
}  
class C implements Ice {  
    @Override  
    public void foo() {}  
}
```



```
class D {  
    public void foo() {}  
}  
class E extends D {  
    @Override  
    public void bar() {}  
}
```





# Předdefinované anotace

- `@Deprecated`
  - označení API, které by se nemělo používat
  - při použití => varování při překladu
  - parametry
    - `String since`
      - default ""
    - `boolean forRemoval`
      - default false

# Předdefinované anotace

- `@SuppressWarnings`
  - zamezí vypisování varování při překladu
  - parametr – třída varování
    - `String[]` value
  - podporované třídy jsou závislé na překladači
  - vždy dostupné třídy varování
    - *unchecked* – varování při „nevhodném“ používání generických typů
    - *deprecation* – varování při použití „deprecated“ elementů
  - př. `@SuppressWarnings("unchecked")`  
`@SuppressWarnings({"unchecked", "deprecation"})`

# JAVA

## Lambda výrazy

# Motivace

- obsluha událostí v GUI
- implementace komparátoru
- implementace vlákna
- ...
  - běžně pomocí anonymní vnitřní třídy

vždy interface  
s jednou metodou

```
interface Comparator<T> {  
    int compare(T o1, T o2);  
}
```

```
class Arrays {  
    ...  
    void sort(T[] a, Comparator<T> c);  
}
```

```
Arrays.sort(array, new Comparator<AClass> () {  
    public int compare(AClass o1, AClass o2) {  
        return o1.x - o2.x;  
    }  
});
```

# Motivace

- předchozí příklad pomocí lambda výrazů

```
Arrays.sort(array, (o1, o2) -> o1.x - o2.x );
```

- zjednodušeně:  
lambda výraz ~ blok kódu s parametry
- od Java 8

# Funkcionální interface

- kde lze lambda výrazy použít?

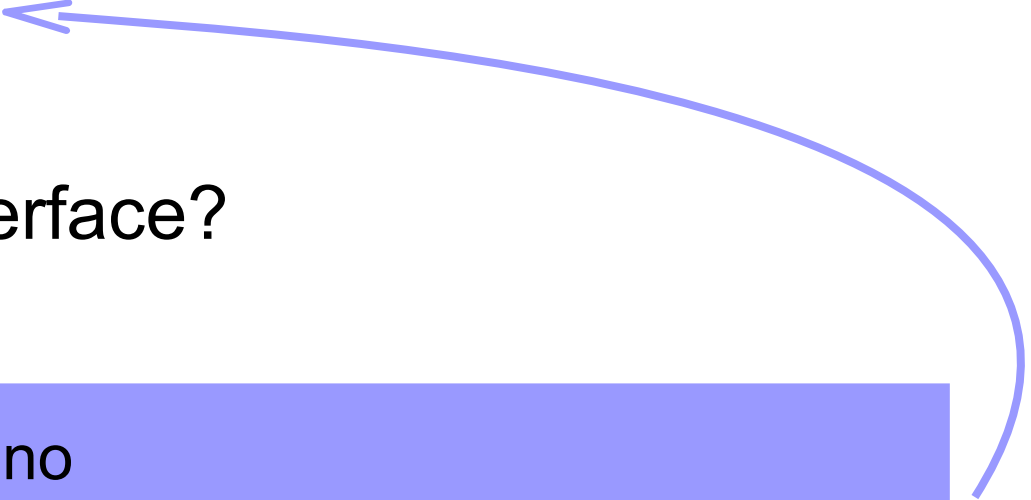
tam, kde se očekává instance **interfacu s jednou abstraktní metodou**

**= funkcionální interface**

- lambda výraz je instance funkcionálního interfacu
- ale  
lambda výraz neobsahuje informaci o tom, který funkcionální interface implementuje

# Funkcionální interface

```
interface Predicate<T> {  
    default Predicate<T> and(Predicate<? super T> other);  
    static <T> Predicate<T> isEqual(Object targetRef);  
    default Predicate<T> negate();  
    default Predicate<T> or(Predicate<? super T> other);  
    boolean test(T t);  
}
```



- Je to funkcionální interface?

ano  
pouze jedna **abstraktní** metoda

# Typ lambda výrazu

- stejný lambda výraz lze přiřadit do různých interfaců

```
Runnable r = () -> {};  
AutoCloseable r = () -> {};
```

```
public interface Runnable {  
    void run();  
}
```

```
public interface AutoCloseable {  
    void close();  
}
```



# Typ lambda výrazu

- lambda výrazy jsou objekty

```
Runnable r = () -> {};  
Object o = r;
```

- ale  
lambda výrazy nelze (přímo) přiřadit do typu Object

```
Object r = () -> {};
```

- protože Object není funkcionální interface

# Syntaxe lambda výrazu

- seznam parametrů v závorkách
  - typy lze vynechat
    - od Java 11 lze použít i **var**
  - při jednom parametru lze závorky vynechat
- “šipka” ->
- tělo
  - jeden výraz
    - lze vynechat return
    - bez závorek
      - nelze vynechat, pokud je použit return
  - blok
    - ve složených závorkách

# Příklady lambda výrazů

- `(int x, int y) -> x + y`
- `(x, y) -> x - y`
- `(var x, var y) -> x - y`
- `() -> 42`
- `(String s) -> System.out.println(s)`
- `x -> 2 * x`
- `c -> { int s = c.size(); c.clear();  
return s; }`

# Funkcionální interface

- `@FunctionalInterface`
  - anotace
  - pro označení funkcionálního interface
    - použití není nutné
      - podobně jako `@Override`

# Reference na metody

- `String::valueOf`
  - reference na statickou metodu
  - ekvivalent: `x -> String.valueOf(x)`
- `Object::toString`
  - reference na nestatickou metodu
  - ekvivalent: `x -> x.toString()`
- `x::toString`
  - reference na metodu konkrétního objektu
  - ekvivalent: `() -> x.toString()`
- `ArrayList::new`
  - reference na konstruktor
  - ekvivalent: `() -> new ArrayList<>()`

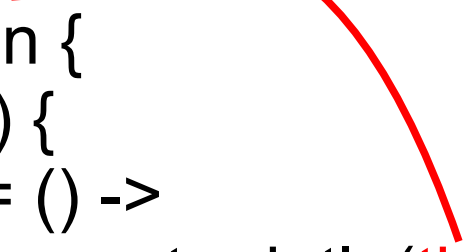
# Lambda výrazy

- lambda výrazy nepřidávají nový prostor (scope) viditelnosti proměnných

```
Path first = Paths.get("/usr/bin");  
Comparator<String> comp = (first, second) ->  
    Integer.compare(first.length(), second.length());
```

- `this` v lambda výrazu odkazuje na `this` metody, ve které je výraz vytvořen

```
public class Application {  
    public void doWork() {  
        Runnable runner = () ->  
            {System.out.println(this.toString());};  
    }  
}
```



# Překlad lambda výrazů

```
public class AClass {  
    ...  
    public void foo(AClass[] array) {  
        Arrays.sort(array, new Comparator<AClass> () {  
            public int compare(AClass o1, AClass o2) {  
                return o1.x - o2.x;  
            }  
        });  
    }  
}
```

```
javac AClass.java  
  
=> AClass.class  
    AClass@1.class
```

- ale

```
public class AClass {  
    ...  
    public void foo(AClass[] array) {  
        Arrays.sort(array, (o1, o2) -> o1.x - o2.x);  
    }  
}
```

```
javac AClass.java  
  
=> AClass.class
```

# JAVA

`java.lang.Object`



# Metody

- clone
- equals
- finalize
- getClass
- hashCode
- notify
- notifyAll
- toString
- wait

# equals

- boolean equals(Object obj)
  - pozor na signaturu metody
  - definována s parametrem typu **Object**
  - při předefinování je nutno zachovat typ **Object**
  - příklad

```
class Complex {
    long x,y;
    public boolean equals(Object obj) {
        if (obj instanceof Complex) {
            Complex c = (Complex) obj;
            if (c.x == x && c.y == y) {
                return true;
            }
        }
        return false;
    }
}
```

# equals

- je vhodné definovat metodu s anotací `@Override`
  - `@Override public boolean equals(Object obj)`
- při definici s jiným typem je metoda **přetížena**, ale ne předefinována

```
class Complex {  
    long x,y;  
    public boolean equals(Complex obj) {  
        ...  
    }  
}
```

- třída obsahuje 2 metody equals

# hashCode

- `int hashCode()`
- hashovací kód objektu
- používá se např. v `java.util.Hashtable` a dalších
- pro stejný objekt vrací stále stejnou hodnotu
  - nemusí být stejná mezi různými běhy programu
- pokud jsou dva objekty stejné ve smyslu metody `equals()`, pak `hashCode` musí vracet u obou stejné číslo
- dva různé objekty nemusí mít nutně různý `hashCode`
  - je to ale velmi vhodné

# clone

- Object `clone()` throws `CloneNotSupportedException`
- vytvoří kopii objektu
- platí
  - `x.clone() != x`
- mělo by platit (ale nemusí)
  - `x.clone().equals(x)`
- aby metoda fungovala, musí objekt implementovat interface `Cloneable`
  - jinak vyhodí výjimku `CloneNotSupportedException`
- pole se berou jako by implementovali `Cloneable`
- "klonují" se jen objekty, ne jejich atributy
  - mělká kopie
  - pokud chcete jinak, je nutno metodu předefinovat

# clone

- predefinování clone

- typická implementace

- ne nezbytně nutná

```
protected Object clone() {  
    Object clonedObj = super.clone();  
    ....  
    return clonedObj;  
}
```

- při klonování by mělo platit

```
a.clone() != a
```

```
a.clone().equals(a)
```

# toString

- vrací textovou reprezentaci objektu
- implicitně vrací
  - getClass().getName() + '@' + Integer.toHexString(hashCode())
- vhodné předefinovat

```
class MyClass { .... }  
...  
MyClass o = new MyClass();  
System.out.println(o); // zavola se toString()
```



Verze prezentace J05.cz.2018.01

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