# Objektovo orientované programovanie

Učiteľ:
**Ing. Jozef Wagner PhD.**

Učebnica:
https://oop.wagjo.com/

# OPG
# Teória 12

1. Statický polymorfizmus
2. Dynamický polymorfizmus
3. Parametrický polymorfizmus

# Polymorfizmus

**Mnohotvárnosť**

Metóda môže mať rôzne správanie v závislosti od triedy, teda typu objektu, ktorý bol pri volaní metódy použitý

- Statický polymorfizmus
- Dynamický polymorfizmus
- Parametrický polymorfizmus

# Statický polymorfizmus

Ad-hoc polymorfizmus

Deje sa **počas kompilácie**

Realizuje sa cez **preťažovanie metód** - method overloading

Máme viacero metód s rovnakým názvom - kompilátor rozhodne, ktorá sa zavolá

Ak žiadna verzia metódy nevyhovuje, hľadá sa verzia s argumentom rodičovského typu

# Statický polymorfizmus

```
class StaticPoly {
  void foo(String x) {
    System.out.println("String");
  }
  void foo(Object x) {
    System.out.println("Object");
  }
  void foo(int x) {
    System.out.println("int");
  }
  void foo(Number x) {
    System.out.println("Number");
  }
}
```

```
foo("Test");          // String
foo(true);            // Object
foo(4);               // int
foo(4.0);             // Number
```

Výber vieme ovplyvniť explicitným pretypovaním pri volaní

```
foo((int)4.0);        // int
foo((Object)"Test"); // Object
```

# Statický polymorfizmus

Rozhodovanie sa deje na základe kódu,
na skutočný typ objektu sa neprihliada

```
public class StaticPoly {
    public static void foo(String x) {...}

    public static void foo(Object x) {...}

    public static void main(String[] args) {
        String s = "Test";
        Object o = s;
        foo(o); // Object!
    }
}
```

# Statický polymorfizmus

Ak je rovnako vhodných metód viacero, kompilátor skončí chybou, že volanie je nejednoznačné (anglicky ambiguous)

```
public class StaticPoly {
    void foo(String x) {...}
    void foo(Number x) {...}

    void foo(int x, long y) {...}
    void foo(long x, int y) {...}
}
```

```
foo(null); // ERROR
foo((String)null); // String

foo(1, 2); // ERROR
foo(1, 2L);
foo(1, (long) 2);
```

# Dynamický polymorfizmus

Subtypový polymorfizmus

Deje sa **počas behu programu** - runtime

Realizuje sa cez **prekrývanie metód** - method overriding

Pri volaní polymorfickej metódy sa na typ uvedený v kóde neprihliada

Počas behu programu sa zavolá metóda, ktorá zodpovedá skutočnému typu objektu

Záleží iba na type objektu, nad ktorým sa volá daná metóda, nie na argumentoch

# Dynamický polymorfizmus

```
class Parent {
   public void foo() {
      System.out.println("Parent");
   }
}

class Child extends Parent {
   @Override
   public void foo() {
      System.out.println("Child");
   }
}
```

V kóde máme triedu Parent, ale zavolá sa prekrytá metóda podľa skutočného typu

```
public class DynamicPoly {
   void zavolajFoo(Parent p) {
      p.foo();
   }

   void main(String[] args) {
      Parent x = new Child();
      zavolajFoo(x); // Child foo
   }
}
```

# Dynamický polymorfizmus

Statické metódy nepodporujú prekrytie metód ani dynamický polymorfizmus

Je mierne pomalší kvôli tabuľke virtuálnych metód (VMT, vtable)

```
Animal myPet;          // Referencia typu Animal

myPet = new Dog();  // Ale objekt je typu Dog
myPet.makeSound();  // Výstup: Pes šteká: Haf!

myPet = new Cat();
myPet.makeSound();  // Výstup: Mačka mňauká: Miau!
```

# Parametrický polymorfizmus

Generický polymorfizmus

Deje sa **počas kompilácie**

Realizuje sa cez **generické triedy**, ktorý parametrizujú typy atribútov

Používa sa, keď klasický typ je príliš obmedzujúci a znemožňoval by znovupoužitie triedy

Použitie typu `Object` by ale malo za následok slabú typovú kontrolu - radšej generickú triedu

Problém: príliš obmedzujúci HAS-A typ znemožňujúci znovupoužitie

Ak by sme chceli prvky iného typu, museli by sme vytvoriť novú triedu

```
public class Pair {
    String x;
    String y;
    Pair(String x, String y) {
        this.x = x;
        this.y = y;
    }
    public String getX() {
        return x;
    }
    public String getY() {
        return y;
    }
}
```

```
Pair p = new Pair("A", "B");
String x = p.getX();

Pair p = new Pair(3, 4); // ERROR
```

Použitie triedy `Object` je príliš všeobecné a prináša slabú typovú kontrolu

Metódy nevedia, aký typ vracajú. Musíme mať explicitné pretypovanie.

```
public class Pair {
    Object x;
    Object y;
    Pair(Object x, Object y) {
        this.x = x;
        this.y = y;
    }
    public Object getX() {
        return x;
    }
    public Object getY() {
        return y;
    }
}
```

```
Pair p = new Pair(3, 4);
```

```
Pair p = new Pair("A", "B");
String x = p.getX(); // ERROR
String x = (String)p.getX();
```

# Parametrický polymorfizmus

Parametrizovaný typ sa pri definícii triedy uvedie do lomených zátvoriek

```
public class Pair<T> {...}
```

Pri vytváraní objektov sa do lomených zátvoriek uvedie, aký konkrétny typ sa použije

```
Pair<String> p1 = new Pair<>("Hello", "World");
```

Namiesto konkrétneho typu, napr. `String` sa použije typová premenná `T`

```
public class Pair<T> {
    T x;
    T y;
    Pair(T x, T y) {
        this.x = x;
        this.y = y;
    }
    public T getX() {
        return x;
    }
    public T getY() {
        return y;
    }
}
```

```
Pair<String> p1 = new Pair<>("A", "B");
String s = p1.getX();

Pair<Integer> p2 = new Pair<>(3, 4);
int i = p2.getX();
```

# Ak máme viac parametrizovaných typov, oddelíme ich čiarkami

```
public class Pair<T, U> {
    T x;
    U y;
    Pair(T x, U y) {
        this.x = x;
        this.y = y;
    }
    public T getX() {
        return x;
    }
    public U getY() {
        return y;
    }
}
```

```
Pair<String, Integer> p1 = new
Pair<>("A", 3);

String s = p1.getX();
int i = p2.getY();
```

# Parametrický polymorfizmus

Používa sa hlavne pri kolekciách, (zoznamy, množiny, zásobníky, a pod)

`ArrayList<String>`

`Set<Integer>`

Parametrizované typy v Jave existujú iba pri kompilácii, počas behu programu sa použije typ `Object`.

Toto nahradenie sa odborne volá **type erasure**, teda vymazanie typu