# Java 3. Polymorfismus

## 1 Polymorfismus

Třetí základní vlastností OOP je polymorfismus. Je to vlastnost programovacího jazyka, která využívá dědičnost tak, že umožňuje zacházet s objekty se stejným rozhraním jednotným způsobem.

Určitý typ úloh vede u jazyků bez polymorfismu k algoritmům, v nichž se nevyhneme složitým podmíněným příkazům a větvení s přepínači (příkaz switch). Pomocí polymorfismu lze tyto úlohy řešit mnohem elegantněji a hlavně flexibilněji. Takové programy s polymorfismem lze snadno rozšiřovat a to dokonce i beze změny samotného algoritmu.

Nejčastěji se pro polymorfismus využívá faktu, že potomek realizuje stejné rozhraní (tedy množinu veřejných metod) jako jeho předek. V Javě lze ke stejnému účelu využívat i datový typ **interface**, díky čemuž lze stejně zacházet i s objekty, které nejsou spolu příbuzné, ale přesto vykonávají stejný druh činnosti. Například létat dokáže letadlo, holub i papírová vlaštovka. Zjevně ovšem nejde o příbuzné třídy. Možnost používat typ **interface**, který všechny tyto třídy implementují, nám umožní vyvarovat se absurdním pokusům vytvářet pro ně společného předka.

### 1.1 Polymorfismus se třídami

Základní schéma využití polymorfismu vyžaduje mít několik tříd se společným předkem. Tyto třídy budou překrývat virtuální metody předka[1] a budou tak reagovat na stejnou zprávu vždy po svém.[2] Velice často se v tomto případě používají pro předky abstraktní třídy.

```
public abstract class Tvar {
  public abstract double getObsah();

  // další metody...
}
```

1 V Javě jsou všechny běžné metody virtuální. Výjimku tvoří finální metody.

2 *Reagovat na zprávu po svém* znamená, že když potomek překryl metodu předka, bude volání této metody v potomkovi dělat něco jiného, než když tuto metodu zavolám v objektu typu předek.

```
public class Ctverec {
  @Override
  public double getObsah()
  { return this.sirka*this.sirka; }
  //...
}
```

```
public class Obdelnik {
  @Override
  public double getObsah()
  { return this.sirka*this.vyska; }
  //...
}
```

Dále polymorfismus vyžaduje vyrobit proměnnou typu předek, do které se ale vloží reference na dynamicky vyrobený objekt typu potomek. Obvyklé je, že se rovnou pracuje s polem nebo kolekcí takových proměnných.

```
Tvar[] tvary = new Tvar[5];
tvary[0] = new Ctverec(10);
tvary[1] = new Obdelnik(3, 5);
//...

double plocha = 0;
for (Tvar t : tvary){
  plocha += t.getObsah();
}
```

Všimněte si, že do proměnné typu předek (Tvar) jde přiřadit reference na dynamicky vyrobeného potomka. V tomto směru je to typově kompatibilní, protože potomek v sobě *obsahuje* objekt typu předek. Proměnná typu předek (t) pak může volat pouze metody svého typu, tedy předka. Finta je ale v tom, že když potomek překryl virtuální metodu předka, bude se v tomto případě volat metoda potomka. V naší ukázce pěkně každý tvar spočítá svůj obsah podle toho, jestli je to čtverec nebo obdélník.

### 1.2 Polymorfismus s rozhraními

V Javě jde vyrobit datový typ **interface**. Je velice podobný abstraktní třídě, ale nemůže obsahovat žádné atributy a obsahuje pouze hlavičky metod (klíčové slovo **abstract** se teď neuvádí).

Tento typ se používá tak, že jej jiné třídy implementují, viz klíčové slovo implements, a zavazují se tak, že překryjí poskytnuté metody.

Rozhraní se obvykle pojmenovávají přídavnými jmény.

```
public interface Plosny {
  public double getObsah();
}
```

```
public class Obrazek implements Plosny
{
  @Override
  public double getObsah(){
    return getVyska()*getSirka();
  }

//...
}
```

Podobně jako u polymorfismu nad třídami, i u rozhraní jde vyrábět proměnné typu rozhraní a přiřazovat do nich objekty tříd, které toto rozhraní implementují. Mechanismus polymorfismu zde funguje totožně.

Výhodou použití rozhraní je, že takto lze stejně zacházet s objekty, které sice nemají žádný příbuzenský vztah, ale dokáží poskytovat služby stejného významu.[3]

Třída může dokonce implementovat více rozhraní najednou, na rozdíl od dědičnosti, kde Java vícenásobnou dědičnost zakazuje.

# 2 Úkol

Vytvoř projekt TvaryPrijmeniJmeno – doplň do názvu své jméno a příjmení.

1. Vytvoř program, který bude počítat s objekty různých tvarů: rovnostranné trojúhelníky, kruhy, obdélníky. Spodní strana trojúhelníků i obdélníků bude vždy rovnoběžná s osou x. Také budeš potřebovat objekty typu bod (není to tvar). Program na začátku vyrobí sadu tvarů s náhodnými rozměry a bude nad nimi provádět různé výpočty s využitím polymorfismu.
2. Vyrob třídu `Tvar` jako abstraktní. Bude obsahovat položky společné všem tvarům. Vytvoř v ní alespoň jednu abstraktní metodu (musí to mít smysl). Konkrétní tvary budou jejími potomky.
3. Každý objekt bude mít konstruktory, které umožní inicializovat jeho atributy těmito hodnotami:
   - kruh – střed a poloměr,
   - trojúhelník – střed a délka strany,
   - obdélník – střed, výška a šířka.
4. Každý objekt umí spočítat svou plochu a obvod.
5. Vytvoř pole o alespoň 10 tvarech a vlož do něj náhodně tvary s náhodnými rozměry v těchto mezích:
   - poloměr, výška, šířka: 1–20
   - souřadnice středu (x a y): 20–80
6. Využij polymorfismus pro výpočet
   - součtu obsahů všech tvarů,
   - součtu obvodů všech tvarů.

   Vypiš tyto hodnoty.
7. **Bonus:** Zajisti, aby se objekty generované do pole nepřekrývaly (stačí, když si ohlídají pravoúhlý prostor kolem sebe). Pokud vyrobíš objekt, který zasahuje do prostoru jiného objektu v poli, zahoď jej a zkus vyrobit nový. Nakonec vypiš v procentech, kolik volného prostoru zůstalo ve čtverci vymezeném body [0, 0], [100, 100] po zaplnění tvými objekty.

**Pomůcka:** Generátor náhodných čísel uděláš pomocí třídy Random.

```
Random nc = new Random();
int n = nc.nextInt(100); // <0, 99>
double x = nc.nextDouble(); // <0, 1)
```

3 U objektů, které jsou z logiky věci příbuzné, je samozřejmě lepší použít dědičnost tříd. Předek může uchovávat společné atributy a poskytovat metody, které by bylo zbytečné implementovat znovu v každém potomkovi.