

Analýza a návrh informačných systémov II 3

objektovo orientovaný návrh

Peter Bednár

Dedičnosť

Dedičnosť (1)

- V reálnom svete môžeme triedy zotriediť podľa spoločných vlastností do **hierarchií**, od všeobecnejších pojmov po špecifickejšie, napr.:
 - Osoba → Študent → Vysokoškolský študent
 - Univerzitný zamestnanec → Pedagogický pracovník → Odborný asistent
 - Dopravný prostriedok → Motorové vozidlo → Osobné auto

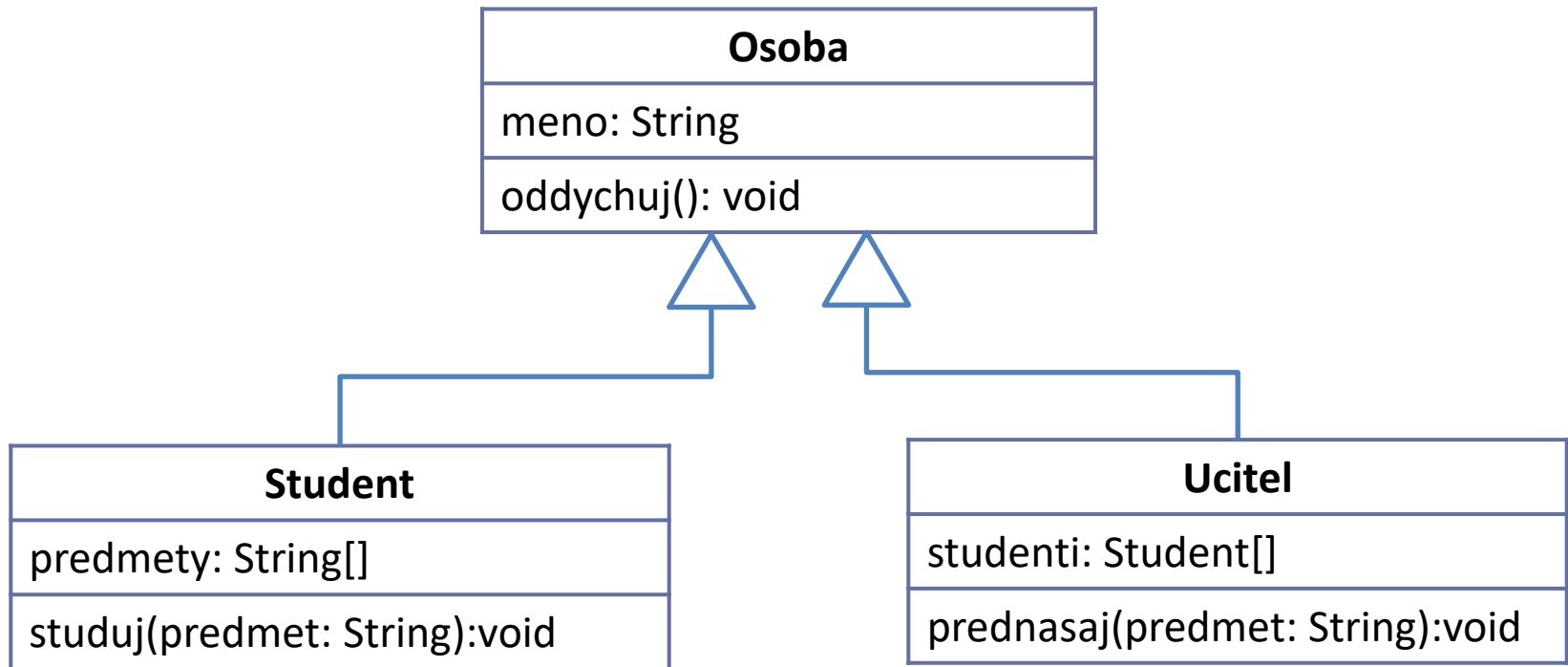
Dedičnosť (2)

- Osoba → Študent
 - Osoba – všeobecnejší pojem – **nadtrieda**
 - Študent – špecifickejší pojem – **podtrieda**
- Dôležité pre umiestnenie tried do hierarchie je, že špecifickejší objekt podtriedy má všetky vlastnosti, ktoré má všeobecnejšia nadtrieda:
 - tzn. napr. každý Študent je zároveň Osoba, a musí mať všetky vlastnosti definované pre osoby ako napr. meno, adresu, dátum narodenia a pod. (môže mať však aj dodatočné vlastnosti špecifické pre Študenta, napr. zoznam predmetov)

Dedičnosť (3)

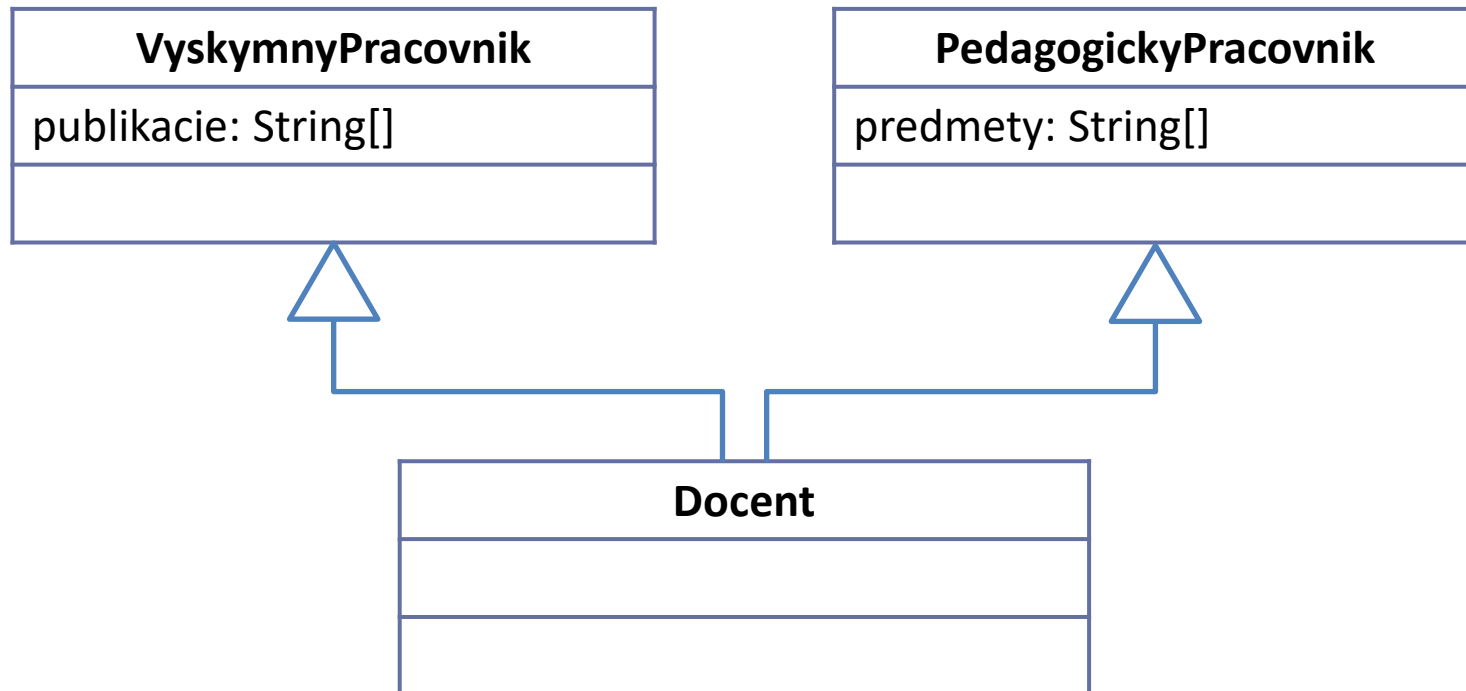
- Vlastnosti sa dedia rekurzívne, tzn. napr. pre:
Dopravný prostriedok → Motorové vozidlo → Osobné auto
auto má všetky vlastnosti motorového vozidla (napr. motor) a zároveň všetky vlastnosti dopravného prostriedku (napr. majiteľa)

Dedičnosť v UML - príklad



Viacnásobná dedičnosť

- Vo všeobecnosti môžeme mať aj viacnásobnú dedičnosť: docent je vedecký aj pedagogický pracovník



Dedičnosť v Java (1)

- V Java môže mať každá trieda iba jednu nadtriedu (jednoduchá dedičnosť), ktorá je uvedená za názvom triedy a kľúčovým slovom **extends**

```
class Trieda extends Nadtrieda {  
    ...  
}
```

- Java definuje najvšeobecnejšiu triedu Object
 - Všetky triedy sú podtriedou Object (aj keď nevedieme žiadnu triedu za **extends**)

Dedičnosť v Jave (2)

- Dedičnosť pri objektovo-orientovanom návrhu umožňuje dosiahnuť lepšiu znovupoužiteľnosť kódu
 - Kód ktorý je spoločný pre viacero tried je možné umiestniť do spoločnej nadtriedy

Dedičnosť – príklad (1)

```
class Osoba {
```

```
    String meno;
```

```
    void oddychuj() {
```

```
        System.out.println(meno + ": oddychujem");
```

```
    }
```

```
}
```

Nadtriedou triedy Osoba je Object

Nadtriedou triedy Student je Osoba a Object

```
class Student extends Osoba {
```

```
    String predmety[];
```

```
    void studuj(String predmet) {
```

```
        System.out.println(meno + ": študujem " + predmet);
```

```
    }
```

```
}
```

V podtriede zdedíme všetky členské premenné a metódy nadtried

Dedičnosť – príklad (2)

```
class Ucitel extends Osoba {  
  
    Student studenti[];  
  
    void prednasaj(String predmet) {  
        System.out.println(meno + ": prednášam " + predmet);  
        if (studenti != null) {  
            for (int i = 0; i < studenti.length; i++) {  
                studenti[i].studuj(predmet);  
            }  
        }  
    }  
}
```

Dedičnosť v Jave (3)

- Objekt podtriedy môžeme použiť všade, kde sa očakáva trieda nadtriedy:
 - tzn. pre **class** A **extends** B
 - ...
 - B objekt = **new** A();
- V podtriede sa zdedia všetky členské premenné a všetky metódy z nadtried

Dedičnosť – príklad (3)

```

public static void main(String args[]) {
    Ucitel lucia = new Ucitel();
    Osoba peter = new Student();

    lucia.oddychuj();
    lucia.prednasaj("ANIS II");

    peter.oddychuj();
    peter.studuj("ANIS II");

    Student student = (Student)peter;
    student.studuj("ANIS II");

    ((Student)peter).studuj("ANIS II");

    ((Ucitel)peter).prednasaj("ANIS II");
}

```

rovnaký typ premennej a triedy
podtriedu môžeme priradiť ako
nadtriedu
môžeme volať metódy nadtriedy
aj vlastné metódy

môžeme priamo volať metódy triedy Osoba
CHYBA – nemôžeme volať metódy triedy
Student, pretože premenná peter je
typu Osoba
samotný objekt je však typu Student
môžeme ho **pretypovať** a volať metódy
triedy Student
pretypovanie vieme urobiť aj bez
pomocnej premennej

CHYBA - nemôžeme však pretypovať
študenta na učiteľa

Dedičnosť a konštruktory

- Keďže objekty podtriedy zdedia členské premenné všetkých nadtried, pri vytváraní objektu sa volajú aj konštruktory nadtried
- V konštruktore podtriedy môžeme volať priamo konštruktor nadtriedy pomocou kľúčového slova **super**
 - Volanie bezparametrického konštruktora nadtriedy vloží kompilátor automaticky a nemusíme ho písať
 - Volanie konštruktorov s parametrami musíme zapísať cez **super**

Dedičnosť a konštruktory – príklad (1)

```
class Osoba {  
  
    Osoba() {  
        System.out.println("konštruktor: Osoba");  
    }  
}
```

```
class Student extends Osoba {  
    Student() {  
        System.out.println("konštruktor: Student");  
    }  
}  
...  
Student student = new Student();
```

Kompilátor automaticky doplní volanie konštruktora nadtriedy `super()` ako prvý príkaz

najprv sa zavolá konštruktor nadtriedy a vypíše sa „konštruktor: Osoba“
potom sa vykoná kód konštruktora Student a vypíše sa „konštruktor: Student“

Dedičnosť a konštruktory – príklad (2)

```
class Osoba {  
    String meno;  
    Osoba(String meno) {  
        this.meno = meno;  
    }  
}
```

```
class Student extends Osoba {  
    String predmety[];  
    Student() {  
        super("neznámi");  
    }  
    Student(String meno, String predmety[]) {  
        super(meno);  
        this.predmety = predmety;  
    }  
}
```

Všimnite si, že trieda Student má dva preťažené konštruktory, jeden bez parametrov a druhý s parametrami typu String a String[]

Ak nadtrieda nemá bezparametrický konštruktor, musíme zavolať konštruktor nadreďenej triedy cez kľúčové slovo super a predať mu požadované parameter (meno)

Za volaním super môžu nasledovať ďalšie inicializačné príkazy špecifické pre podtriedu

Trieda Object

- Každý objekt v Jave je inštanciou triedy `Object` (pretože každá trieda je potriedou triedy `Object`)
- **Konštruktor:**
 - `Object() { }` – prázdny bezparametrický konštruktor
- **Členské premenné:** žiadne
- **Metódy:** všeobecne použiteľné metódy, napr.:
 - `String toString()` – vráti textovú reprezentáciu objektu pre výpis

Polymorfizmus

Polymorfizmus (1)

- Zdedené metódy je možné v podtriede predefinovať
- Tzn. nad rôznymi objektami je možné zavolať tú istú metódu s tými istými parametrami, ale každý objekt sa bude chovať inak – ide o tzv. **polymorfizmus**
- Predefinované metódy by sme mali v podtriede označiť tzv. anotáciou: `@Override`
- V predefinovaných metódach môžeme volať pôvodnú zdedenú metódu pomocou kľúčového slova **super**

Polymorfizmus – príklad (1)

```
class Osoba {  
    void oddychuj() { System.out.println("nerobím nič"); }  
}  
class Student extends Osoba {  
    @Override  
    void oddychuj() { System.out.println("párty!!!"); }  
}  
class Ucitel extends Osoba {  
    @Override  
    void oddychuj() { System.out.println("čítam knihu..."); }  
}
```

Predefinované metódy označené anotáciou `@override`

```
Osoba osoba1 = new Osoba(); osoba1.oddychuj();  
Osoba osoba2 = new Student(); osoba2.oddychuj();  
Osoba osoba3 = new Ucitel(); osoba3.oddychuj();
```

Vypíše sa: nerobím nič
Vypíše sa: párty!!!
Vypíše sa: čítam knihu...

Polymorfizmus – príklad (2)

```
class Osoba {  
    void oddychuj() {  
        System.out.println("nerobim nič");  
    }  
}
```

```
class Student extends Osoba {  
    @Override  
    void oddychuj() {  
        System.out.println("párty!!! a potom");  
        super.oddychuj();  
    }  
}
```

Volanie pôvodnej metódy, vypíše sa:
párty!!! a potom
nerobím nič

Testovanie typu

- To, či objekt patrí do danej triedy môžeme testovať operátorom `instanceof`
- Príklad:

```
Student student = new Student();
student instanceof Student    // true
student instanceof Osoba      // true - je inštanciou pre každú
                               // nadtriedu

student = null;
student instanceof Student    // null nepatrí do žiadnej triedy
Osoba osoba = new Student();
osoba instanceof Student      // true, nezáleží na typu
                               // premennej ale typu objektu
```

Identita objektov (1)

- Operátory `==` alebo `!=` porovnávajú referenciu na objekt, nie jeho hodnoty

- To platí napr. aj pre reťazce, takže:

```
String s1 = "Ahoj";  
if (s1 == "Ahoj") { // môže byť false !!!  
    ...  
}
```

- Ak chceme porovnávať objekty podľa hodnoty, môžeme použiť metódu `boolean equals(Object obj)` ktorá je definovaná v triede `Object`

Identita objektov (2)

- Pre reťazce:

```
if (s1.equals("Ahoj")) {  
    ...  
}
```
- Pre vlastné triedy sa implementácia `equals` z `Object`,
 - Štandardne vnútorne testuje rovnosť objektov podľa odkazu operátorom `==`
- Môžeme ju predefinovať a porovnávať objekty podľa hodnôt

Identita objektov – príklad (1)

```
class Osoba {  
    String meno;  
  
    Osoba(String meno) {  
        this.meno = meno;  
    }  
  
    @Override  
    public boolean equals(Object obj) {  
        if (obj instanceof Osoba) {  
            Osoba osoba = (Osoba)obj;  
            return meno.equals(osoba.meno);  
        } else {  
            return false;  
        }  
    }  
}
```

Otestujeme, či je objekt obj z triedy Osoba

Ak áno, pretypujeme ho a porovnáme podľa mena

Ak obj nie je typu Osoba, vrátime false

Identita objektov – príklad (2)

```
class Student extends Osoba {  
    Student(String meno) {  
        super(meno);  
    }  
}
```

```
Osoba osoba1 = new Osoba("Peter");  
Osoba osoba2 = new Osoba("Peter");  
Student student = new Student("Peter");  
osoba1 == osoba2           // false, premenné sa neodkazujú na tie isté  
                           // objekty  
osoba1.equals(osoba2)     // true, keďže sa mená rovnajú  
osoba1.equals(null)      // false, null nepatrí žiadnej triede  
osoba1.equals(student)   // true, student je aj inštanciou triedy  
                           // Osoba a mená sa rovnajú
```

Zapúzdrenie

Zapúzdenie (1)

- K vnútornému stavu objektu by mal mať prístup iba samotný objekt, resp. podtriedy, ktoré ho zdedia
 - Zlepšenie bezpečnosti programovania
- Prístup k členským premenným a metódam (aj konštruktorom) môžeme ohraničiť modifikátormi **public**, **protected** a **private**
- Podobne môžeme obmedziť prístup k celým triedam

Zapúzdrenie (1)

	public	protected	bez ohraničenia	private
Daná trieda	X	X	X	X
Všetky triedy v balíku	X	X	X	
Podtrieda v tom istom balíku	X	X		
Podtrieda v inom balíku	X	X		
Ostatné triedy	X			

Zapúzdrenie – príklad (1)

```
public class Motor {  
  
    private long startovacíKod;  
    protected int otacky;  
    protected int teplota;  
    public String typ;  
  
    public void nastartuj(long kod) {  
        if (kod == startovacíKod) {  
            nastavOtacky(10);  
        }  
    }  
    ...  
}
```

Samotná trieda je public takže k nej môžu pristupovať triedy aj z iných balíkov

K privátnym premenným môže priamo pristupovať len daná trieda

Zapúzdrenie – príklad (2)

```
public class Motor {  
  
    private long startovacíKod;  
    protected int otacky;  
    protected int teplota;  
    public String typ;  
  
    ...  
    public int getOtacky() {  
        return otacky;  
    }  
  
    ...  
    protected void nastavOtacky(double vykon) {  
        otacky = vykon * 50 + 2000;  
    }  
}
```

Ak chceme aby ostatné triedy mali prístup iba na čítanie, premennú zdefinujeme ako `private` alebo `protected` a pridáme verejnú metódu pre prístup

Abstraktné triedy a metódy

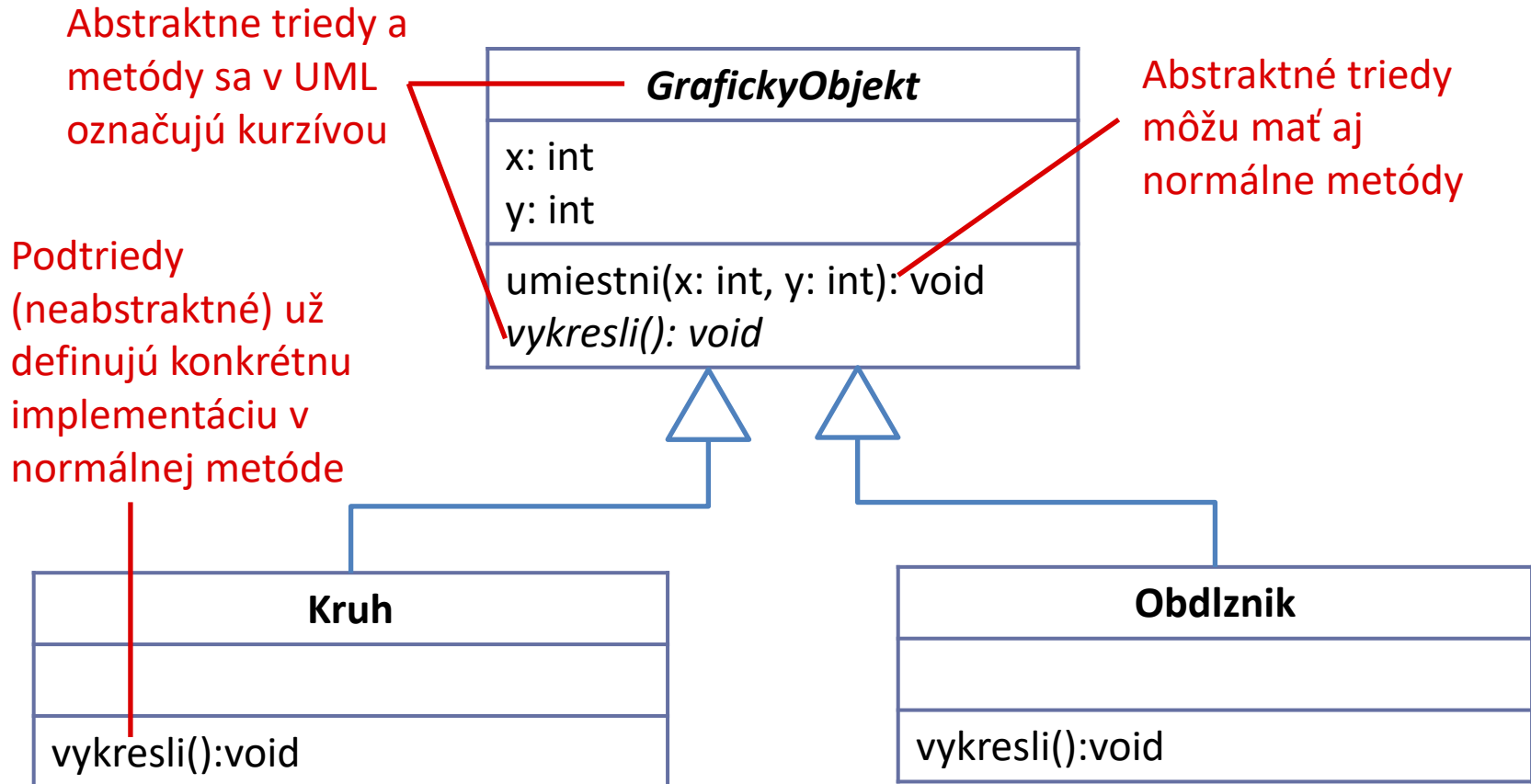
Abstraktné metódy

- Niekedy chceme v nadtriede predpísať, že každá podtrieda musí mať nejakú metódu, ale nechceme ju naprogramovať
- Príklad:
 - Máme triedu `GrafickyObjekt` a jej podtriedy `Kruh`, `Obdĺžnik` atď.
 - Chceme aby sa dal každý objekt vykresliť metódou `vykresli()`, ale keďže vykreslenie závisí na konkrétne tvare objektu, implementácia metódy `vykresli()` má zmysel iba pre konkrétne podtriedy
 - Preto v triede `GrafickyObjekt` len zadefinujeme metódu bez jej implementácie a označíme ju kľúčovým slovom **abstract**

Abstraktné triedy

- Podobne môžeme definovať abstraktnú triedu
- Platí:
 - Ak má trieda aspoň jednu abstraktnú metódu, musí byť abstraktná
 - Ako abstraktnú triedu môžeme označiť aj triedy bez abstraktných metód
 - Z abstraktných tried nie je možné vytvoriť inštancie, slúžia iba na to aby sa z nich odvodili ďalšie triedy

Abstraktné metódy a triedy - UML



Abstraktné metódy a triedy – príklad (1)

```
public abstract class GrafickyObjekt {  
  
    protected int x;  
    protected int y;  
  
    public GrafickyObjekt() {  
        umiestni(10, 10);  
    }  
  
    public void umiestni(int x, int y) {  
        this.x = x;  
        this.y = y;  
    }  
  
    public abstract void vykresli();  
}
```

Triedu označíme ako abstraktnú

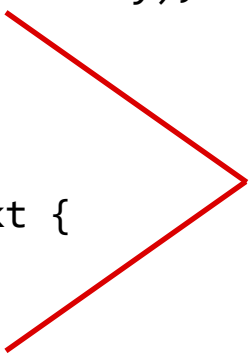
Abstraktná trieda môže definovať členské premenné, konštruktory, alebo normálne (neabstraktné) metódy

Pre abstraktnú metódu nevedieme žiadne telo

Abstraktné metódy a triedy – príklad (2)

```
public class Kruh extends GrafickyObjekt {  
    @Override  
    public void vykresli() {  
        System.out.println("kruh: " + x + ":" + y);  
    }  
}  
  
public class Obdlznik extends GrafickyObjekt {  
    @Override  
    public void vykresli() {  
        System.out.println("obdlznik: " + x + ":" + y);  
    }  
}
```

Konkrétne implementácie



Abstraktné metódy a triedy – príklad (3)

```
public class Program {  
  
    public static void main(String args[]) {  
        GrafickyObjekt objekty[] = new GrafickyObjekt[]{  
            new Kruh(),  
            new Obdlznik(),  
            new Kruh()  
        };  
  
        for (int i = 0; i < objekty.length; i++) {  
            objekty[i].vykresli();  
        }  
    }  
}
```

Pole má spoločný typ – abstraktnú nadtriedu

Postupne sa zavolajú špecifické implementácie a vypíše sa kruh, obdĺžnik a kruh

Zhrnutie

- Dedičnosť
- Polymorfizmus a predefinovanie metód
- Zapúzdrenie a prístup k členským premenným a metódam
 - `private`
 - `protected`
 - `public`
- Abstraktné triedy a metódy