KATEDRA INFORMATIKY
UNIVERZITA PALACKÉHO V OLOMOUCI
www.inf.upol.cz

Jan Laštovička
https://www.inf.upol.cz/lide/jan-lastovicka
jan.lastovicka@upol.cz
17. listopadu 12, 771 46 Olomouc

Úvod do programovacích stylů ◇ poznámky k přednášce

3. Třídy

verze z 9. října 2024

Z minulé přednášky víme, že data objektu tvoří jeho vnitřní stav. Vnitřní stav objektu je rozdělen do pojmenovaných položek nazývaných **atributy**. Názvy zpráv, kterým objekt rozumí, musí být různé od jmen jeho atributů. Toho docílíme tak, že názvy zpráv (na rozdíl od jmen atributů) budou obsahovat sloveso. Například `set`, `get` nebo `move`.

Vykonání příkazu:

```
class Class:
    def __init__(self):
        self.atribute1 = None
        self.atribute2 = None
        ⋮
```

vytvoří novou třídu, kde *`Class`* je jméno nové třídy a *`atribute1`*, *`atribute2`*, ... jsou názvy atributů. Vytvořením instance třídy *`Class`* vznikne objekt, který bude mít atributy *`atribute1`*, *`atribute2`*, ... Hodnotou každého atributu bude hodnota `None`.

Například třídu `Point` můžeme definovat takto:

```
class Point:
    def __init__(self):
        self.x = None
        self.y = None
```

Můžeme si vytvořit její instanci:

```
>>> point = Point()
>>> point
<Point object at 0x1056510f0>
```

Pokud *`object`* je objekt a *`attribute`* jeho atribut, pak

```
object.attribute
```

je výraz, jehož hodnota je hodnota atributu *`attribute`* objektu *`object`*. Například hodnota atributu `x` objektu `point`:

```
>>> point.x
>>>
```

Interpret nic nevytiskne, protože hodnota atributu `x` objektu `point` je `None`, a Shell hodnotu `None` netiskne.

Pokud *`object`* je objekt, *`attribute`* jeho atribut a *`value`* hodnota, pak

```
object.attribute = value
```

je příkaz, který se vykoná tak, že nastaví hodnotu atributu *`attribute`* objektu *`object`* na hodnotu *`value`*. Pokud by objekt zadaný atribut neměl, příkaz jej nejprve vytvoří.

Nastavíme atributy `x` a `y` objektu `point`:

```
>>> point.x = 3
>>> point.y = 4
```

Ověříme:

```
>>> point.x
3
>>> point.y
4
```

Kód, který objekt může při přijmutí zprávy vykonat, je rozdělen do metod. **Metoda** je pojmenovaná funkce objektu.

Rozšíříme si definici třídy:

```
class Class:
    methods
```

Část *`methods`* je blok obsahující definice metod. Definice metody má tvar:

```
def method(self, parameter1, arameter2, ...):
    block
```

kde *`method`* je jméno metody, *`parameter1`*, *`arameter2`*, ... jsou proměnné a *`block`* je blok kódu. Definice třídy bude vždy začínat definicí metody `__init__`. Objektový systém při vytváření instance třídy zašle nově vytvořené instanci zprávu `__init__` bez argumentů. Říkáme, že metoda **inicializuje** instanci. Instance třídy získá všechny metody, které třída definuje. Zavedeme si omezení, že vytvářet objektu atributy je možné jen při jeho inicializaci.

Rozšíříme si definici třídy `Point` o definice metod:

```
class Point:
    def __init__(self):
        self.x = None
        self.y = None

    def get_x(self):
        return self.x

    def get_y(self):
        return self.y

    def set_x(self, x):
        self.x = x
        return self

    def set_y(self, y):
        self.y = y
        return self
```

a vytvoříme novou instanci:

```
>>> point = Point()
```

Třída `Point` objektu `point` definovala vlastnosti `x` a `y`.

Podívejme se podrobněji na to, co se stane, když objektu *`object`* zašleme zprávu *`message`* s argumenty *`arg1`*, *`arg2`*, ... Objekt se pokusí u sebe najít metodu jménem *`message`*. Pokud se to nepovede, objekt zaslanou zprávu odmítne. Jinak ji přijme a zavolá nalezenou metodu s argumenty *`object`*, *`arg1`*, *`arg2`*, ... Vidíme, že příjemce zprávy se stává prvním argumentem volané funkce. Návratová hodnota funkce je návratovou hodnotou zaslané zprávy.

Tedy například pokud zašleme zprávu `set_x` objektu `point` s argumentem `3`, pak se najde jeho metoda `set_x`, která se zavolá s dvěma argumenty: `point` a `3`. Kód metody nastaví hodnotu atributu `x` objektu `point` na `3` a vrátí objekt `point`, který bude i návratovou hodnotou zaslané zprávy:

```
>>> point.set_x(3)
<Point object at 0x10bb8fc10>
```

Podobně zaslání zprávy `get_x` bez argumentu objektu `point` vede k zavolání jeho metody `get_x` s argumentem `point`. Metoda vrátí hodnotu atributu `x` objektu `point`:

```
>>> point.get_x()
3
```

Metoda, která se pro zaslanou zprávu v objektu nalezne, se nazývá **obsluha zprávy**. Procesu vykonávání těla obsluhy zprávy se říká **obsloužení zprávy**. Můžeme například říci, že obsloužení zprávy `set_x` způsobilo změnu atributu `x`.

Budeme dodržovat **princip zapouzdření**:

> Hodnoty atributů objektu smí přímo číst a měnit pouze metody tohoto objektu.

Ostatní můžou k atributům objektu přistupovat pouze přes jeho metody. Například přístup k atributům `x` a `y` instancí třídy `Point` je zajištěn pomocí stejnojmenných vlastností.

Navíc, pokud nějaká třída definuje vlastnost odpovídající atributu, pak jsou metody třídy povinny tuto vlastnost využívat a nepřistupovat přímo k atributům. Například metoda bodu na zjištění vzdálenosti od počátku musí používat vlastnosti `x` a `y`:

```
def get_r(self):
    x = self.get_x()
    y = self.get_y()
    return (x ** 2 + y ** 2) ** 0.5
```

Zkusíme bodu zaslat zprávu `get_r`:

```
>>> point = Point()
>>> point.set_x(3)
<Point object at 0x10afcbc10>
>>> point.set_y(4)
<Point object at 0x10afcbc10>
>>> point.get_r()
5.0
```

Nově vytvořená instance třídy `Point` není v konzistentním stavu. Zprávy `get_x` a `get_y` vrací nepřípustnou hodnotu `None`:

```
>>> point = Point()
>>> point.get_x()
>>> point.get_y()
>>>
```

Objekt `point` nereprezentuje žádný geometrický bod v rovině.

Protože inicializaci instance obstarává metoda `__init__`, můžeme její definici změnit tak, aby vytvořené atributy měly smysluplné výchozí hodnoty:

```
def __init__(self):
    self.atribute1 = expr1
    self.atribute2 = expr2
    ⋮
```

Nově vytvořená instance pak bude obsahovat atributy *atribute1*, *atribute2*, ..., jejichž hodnoty postupně vzniknou vyhodnocováním výrazů *expr1*, *expr2*, ...

Změníme definici inicializace instance třídy `Point`:

```
def __init__(self):
    self.x = 0
    self.y = 0
```

Nově vytvořené instance reprezentují bod o souřadnicích $(0, 0)$:

```
>>> point = Point()
>>> point.get_x()
0
>>> point.get_y()
0
```

Bod můžeme snadno uvést do nekonzistentního stavu:

```
>>> point = Point()
>>> point.set_x("1")
<Point object at 0x7f906a3cb9d0>
>>> point.get_x()
'1'
```

Chyba se může projevit úplně v jiném místě programu za nějaký čas:

```
>>> point.get_x() + 5
TypeError: can only concatenate str (not "int") to str
```

Lepší by bylo nedovolit nastavit hodnotu atributu na hodnotu, která není celé číslo.

Změníme definice metod třídy `Point`:

```
def set_x(self, x):
    if type(x) != int:
        raise TypeError("x coordinate of a point should be an integer")
    self.x = x
    return self

def set_y(self, y):
    if type(y) != int:
        raise TypeError("y coordinate of a point should be an integer")
    self.y = y
    return self
```

Nyní pokus o nastavení vlastnosti `x` na nečíselnou hodnotu způsobí vyvolání výjimky, kterou objekt změnu odmítne. Jeho vnitřní stav se nezmění:

```
>>> point = Point()
>>> point.set_x(1)
<Point object at 0x7fd0c62f8100>
>>> point.get_x()
1
>>> point.set_x("1")
TypeError: x coordinate of a point should be an integer
>>> point.get_x()
1
```

Vlastnosti objektu nemusí jednoznačně odpovídat jeho atributům. Pro příklad se podívejme na následující třídu pro počítadla.

```
class Counter:
    def __init__(self):
        self.value = 0

    def get_value(self):
        return self.value

    def set_value(self, value):
        if type(value) != int:
            raise TypeError("value of a counter should be an integer")
        self.value = value
        return self

    def inc(self):
        value = self.get_value()
        self.set_value(value + 1)
        return self

    def get_text(self):
        return str(self.get_value())

    def set_text(self, text):
        return self.set_value(int(text))
```

Třídu si vyzkoušíme:

```
>>> counter = Counter()
>>> counter.get_value()
0
>>> counter.inc()
<Counter object at 0x10212b290>
```

```
>>> counter.get_text()
'1'
>>> counter.set_text("5")
<Counter object at 0x10f07f440>
>>> counter.get_value()
5
```

Jak je vidět, počítadlo má vlastnosti `value` a `text`. Obě však mají vliv na atribut `value`. Rozdíl mezi nimi je, že `value` je číselná hodnota počítadla a `text` její textová reprezentace.

Zavedeme si třídu pro popisky:

```
class Label:
    def __init__(self):
        self.x = 0
        self.y = 0
        self.text = ""

    def get_x(self):
        return self.x

    def get_y(self):
        return self.y

    def set_x(self, x):
        if type(x) != int:
            raise TypeError("x coordinate of a label should be an integer")
        self.x = x
        return self

    def set_y(self, y):
        if type(y) != int:
            raise TypeError("y coordinate of a label should be an integer")
        self.y = y
        return self

    def get_text(self):
        return self.text

    def set_text(self, text):
        if not type(text) == str:
            raise TypeError("text of a label should be a string")
        self.text = text
        return self
```

Popisek je podobný bodu (má také vlastnosti `x` a `y`), ale navíc má vlastnost `text`.

Dále si zavedeme třídu pro skupiny ovládacích prvků:

```
class Group:
    def __init__(self):
        self.items = []

    def get_items(self):
        return self.items

    def set_items(self, items):
        self.items = items
        return self
```

Skupinu snadno uvedeme do nekonzistentního stavu:

```
>>> group = Group()
>>> group.set_items([1])
>>> <Group object at 0x10aa32c90>
>>> group.get_items()
[1]
```

Hodnotou vlastnosti `items` musí být pole ovládacích prvků. Ovládací prvky jsou přímé instance tříd `Label` a `Group`. Přidáme ochranu do metody `set_items`:

```
def set_items(self, items):
    for item in items:
        if not (isinstance(item, Label)
                or isinstance(item, Group)):
            raise TypeError("items of a group \
have to be an array of widgets")
    self.items = items
    return self
```

Kontrola ochrany:

```
>>> group1 = Group()
>>> label = Label()
>>> group1.set_items([label])
<Group object at 0x1027b8f90>
>>> group1.get_items()
[<Label object at 0x102b7d4d0>]
>>> group2 = Group()
>>> group1.set_items([label, group2])
<Group object at 0x1027b8f90>
>>> group1.get_items()
[<Label object at 0x102b7d4d0>, <Group object at 0x102505790>]
```

```
>>> group1.set_items([label, 1])
TypeError: items of a group have to be an array of widgets
>>> group1.set_items(1)
TypeError: 'int' object is not iterable
```

Skupinu lze stále uvést do nekonzistentního stavu tak, že nastavované pole změníme z vnějšku:

```
>>> group = Group()
>>> items = [label]
>>> group.set_items(items)
<Group object at 0x1027b8f90>
>>> group.get_items()
[<Label object at 0x102b7d4d0>]
>>> items[0] = 1
>>> items
[1]
>>> group.get_items()
[1]
```

Provedeme změnu metody `get_items`:

```
def set_items(self, items):
    for item in items:
        if not (isinstance(item, Label)
                or isinstance(item, Group)):
            raise TypeError("items of a group \
have to be an array of widgets")
    self.items = items[:]
    return self
```

Operátor řezu `[:]` je zde použit pro kopírování pole. Nyní již změna pole, které jsme nastavili jako hodnotu vlastnosti `items`, neovlivní vnitřní stav skupiny:

```
>>> items = [Label()]
>>> group = Group()
>>> group.set_items(items)
<Group object at 0x106f3be10>
>>> group.get_items()
[<Label object at 0x107721a90>]
>>> items[0] = 1
>>> group.get_items()
[<Label object at 0x107721a90>]
```

Stále je však možné skupinu uvést do nekonzistentního stavu tak, že změníme pole vrácené zprávou `get_items`:

```
>>> group = Group().set_items([Label()])
>>> items = group.get_items()
>>> items
[<Label object at 0x107721a90>]
>>> items[0] = 1
>>> group.get_items()
[1]
```

Musíme změnit metodu `get_items` tak, aby místo pole prvků vracela jeho kopii:

```
def get_items(self):
    return self.items[:]
```

Provedeme kontrolu ochrany:

```
>>> group = Group().set_items([Label()])
>>> items = group.get_items()
>>> items
[<Label object at 0x107b77530>]
>>> items[0] = 1
>>> items
[1]
>>> group.get_items()
[<Label object at 0x107b77530>]
```

Teprve nyní jsme docílili plné ochrany konzistence vlastnosti `items` skupin ovládacích prvků.

Popisky i skupiny jsou ovládací prvky. Každý ovládací prvek by mělo jít posunout. Přesněji by každý ovládací prvek měl rozumět následující zprávě.

widget`.move(`*dx*`, `*dy*`) => `*widget*

widget: ovládací prvek
dx, *dy*: celá čísla

Přidáme obsluhu zprávy `move` do třídy `Label`:

```
def move(self, dx, dy):
    self.set_x(self.get_x() + dx)
    self.set_y(self.get_y() + dy)
    return self
```

a do třídy `Group`:

```
def move(self, dx, dy):
     for item in self.get_items():
         item.move(dx, dy)
     return self
```

Tedy posunout popisek znamená změnit jeho kartézské souřadnice a posun skupiny uskutečníme tak, že posuneme každý prvek ve skupině. Vyzkoušíme:

```
>>> label = Label()
>>> label.move(10, 20)
<Label object at 0x10f3bcb90>
>>> print(label.get_x(), label.get_y())
10 20
>>> label2 = Label()
>>> group = Group().set_items([label, label2])
>>> group.move(5, 10)
<Group object at 0x10f906f50>
>>> print(label.get_x(), label.get_y())
15 30
>>> print(label2.get_x(), label2.get_y())
5 10
```

Posun popisku i skupiny provedeme v obou případech zasláním zprávy `move`. Ovšem v každém z případů se zavolá jiná metoda zodpovědná za posun.

Právě jsme si demonstrovali obecný princip **polymorfizmu** v objektovém programování:

> Různé třídy mohou definovat pro tutéž zprávu různé metody.

Otázky a úkoly na cvičení

1. Vytvořte třídu `Window` pro grafická okna. Okna budou mít jedinou vlastnost `widget` s výchozí hodnotou `None`. Zajistěte, aby hodnota vlastnosti byla vždy `None` nebo ovládací prvek, což je instance třídy `Label` nebo `Group`. Vytvořte okno se dvěma popisky s texty `"A"` a `"B"`.

2. Stvořte třídu `Radiobutton` pro ovládací prvek přepínač. Postupujte následovně.

   (a) Napište třídu `Radiobutton` definující atributy `x`, `y` a `state` (stav přepínače). Stav přepínače je logická hodnota (`True` nebo `False`) určující, zda je přepínač vybrán.

   (b) Definujte vlastnosti `x`, `y` a `state`.

   (c) Přidejte ochrany při nastavování hodnot vlastností.

   (d) Definujte obsluhy zpráv `move`, `is_selected` a `toggle`. Zpráva `radiobutton.move(dx, dy)` posune přepínač o `dx` a `dy`. Zpráva `radiobutton.is_selected()` rozhoduje, zda je přepínač vybrán. Zpráva `radiobutton.toggle()` přepne stav přepínače.

   Kód otestujte:

```
>>> r = Radiobutton()
>>> r.get_state()
False
>>> r.toggle().get_state()
True
>>> r.move(20, 30)
<Radiobutton object at 0x1106bda50>
>>> print(r.get_x(), r.get_y())
20 30
```

3. Vytvořte třídu `RadiobuttonGroup` pro skupinu přepínačů. Postup:

   (a) Napište třídu `RadiobuttonGroup` definující atribut `items`.

   (b) Přidejte vlastnost `items` (metody `get_items` a `set_items`). Zajistěte, aby hodnota vlastnosti `items` byla vždy pole přepínačů.

   (c) Napište obsluhu zprávy `move` posunující skupinu.

   Novou třídu otestujte:

```
>>> rg = RadiobuttonGroup().set_items([Radiobutton(),
                                       Radiobutton().move(0, 20)])
>>> rg.get_items()
[<Radiobutton object at 0x110823190>,
 <Radiobutton object at 0x110823250>]
```

4. Zajistěte, aby vždy platilo: Ve skupině přepínačů může být vybrán nejvýše jeden přepínač. Postupujte následovně.

   (a) Skupině přepínačů přidejte vlastnost `selected`, jejíž hodnotou je vybraný přepínač. Pokud žádný přepínač není vybrán, vlastnost má hodnotu `None`. Nastavení hodnoty vlastnosti `selected` zruší vybrání dříve vybraného přepínače. Počítejte s tím, že uživatel třídy smí zapínat přepínače pouze nastavením vlastnosti `selected`. Zamyslete se, zda musíte přidávat atribut `selected` do skupiny přepínačů.

   (b) Nakonec zajistěte, aby nastavení vlastnosti `items` zrušilo vybrání všech přepínačů. Napište si k tomu pomocnou metodu `deselect_all`.

   (c) Přidejte skupině přepínačů vlastnosti `x` a `y` určující souřadnice skupiny.

   Kód otestujte:

```
 >>> r1 = Radiobutton()
>>> r1
<Radiobutton object at 0x103beda10>
>>> r2 = Radiobutton()
>>> r2
<Radiobutton object at 0x10444ffd0>
```

```
>>> rg = RadiobuttonGroup().set_items([r1, r2])
>>> print(r1.is_selected(), r2.is_selected())
False False
>>> rg.get_selected()
>>> rg.set_selected(r1)
<RadiobuttonGroup object at 0x10444ff90>
>>> print(r1.is_selected(), r2.is_selected())
True False
>>> rg.get_selected()
<Radiobutton object at 0x103beda10>
>>> rg.set_selected(r2)
<RadiobuttonGroup object at 0x10444ff90>
>>> print(r1.is_selected(), r2.is_selected())
False True
>>> rg.get_selected()
<Radiobutton object at 0x10444ffd0>
>>> rg.set_selected(None)
<RadiobuttonGroup object at 0x10444ff90>
>>> print(r1.is_selected(), r2.is_selected())
False False
>>> rg.get_selected()
>>>
```