
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

4. Dědičnost

verze z 11. října 2023

Jednoduchá definice třídy `Entry` by vypadala následovně.

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

    def get_x(self):
        return self.x

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

    def get_y(self):
        return self.y

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

    def get_text(self):
        return self.text

    def set_text(self, text):
        self.text = text
        return self
```

Třída definuje atributy a vlastnosti `x`, `y` a `text`.

Zárodek třídy `Checkbox` je podobný třídě `Entry`:

```
class Checkbox:
    def __init__(self):
        self.x = 0
        self.y = 0
        self.value = False
```

```
    def get_x(self):
        return self.x

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

    def get_y(self):
        return self.y

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

    def get_value(self):
        return self.value

    def set_value(self, value):
        self.value = value
        return self
```

Obě třídy se shodují v definicích atributů a vlastností `x` a `y`.

Problém s opakujícím se kódem vyřešíme tak, že umožníme sdílet definice atributů a metod mezi třídami. Třída je bude dědit po svých předcích.

Přímého předka třídy můžeme uvést v definici třídy:

```
class Class(ParentClass):
    def __init__(self):
        super().__init__()
        ⋮
```

Class: jméno nové třídy
ParentClass: jméno existující třídy

Třída `AtomicWidget` bude obsahovat společné definice tříd `Entry` a `Checkbox`:

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

    def get_x(self):
        return self.x
```

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

    def get_y(self):
        return self.y

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

Třídy `Entry` a `Checkbox` budou mít za přímého předka třídu `AtomicWidget`:

```
class Entry(AtomicWidget):
    def __init__(self):
        super().__init__()
        self.text = ""

    def get_text(self):
        return self.text

    def set_text(self, text):
        self.text = text
        return self

class Checkbox(AtomicWidget):
    def __init__(self):
        super().__init__()
        self.value = False

    def get_value(self):
        return self.value

    def set_value(self, value):
        self.value = value
        return self
```

Pomocí přímého předka můžeme definovat následující tři pojmy.

Třída C je *předkem* třídy D, pokud je C přímým předkem D nebo C je přímým předkem třídy E a třída E je předkem třídy D.

Třída D je *přímý potomek* třídy C, pokud třída C je přímý předek třídy D.

Třída D je *potomek* třídy C, pokud třída C je předkem třídy D.

Třída získá (říkáme, že *zdědí*) definice všech atributů a metod svých *předků*.

Tedy třídy `Entry` a `Checkbox` zdědily od třídy `AtomicWidget` atributy `x`, `y` a metody `get_x`, `set_x`, `get_y` a `set_y`. Například dostáváme:

```
>>> entry = Entry()
>>> entry.set_x(5)
<__main__.Entry object at 0x10a463a10>
>>> entry.get_x()
5
>>> checkbox = Checkbox()
>>> checkbox.set_x(5)
<__main__.Checkbox object at 0x10a9bbe50>
>>> checkbox.get_x()
5
```

Třídy musí splňovat následující pravidlo nazývané *is-a*.

> Je-li třída D potomkem třídy C, pak věta „každé D je C“ musí dávat smysl.

Například třída `Entry` může být potomkem třídy `AtomicWidget`, protože každé textové pole je atomický ovládací prvek. (Every entry is an atomic widget). Třída `Window` nemůže být potomkem třídy `AtomicWidget`, protože každé okno není atomickým ovládacím prvkem. (Every window is not an atomic widget.)

Objekt O je *instancí* třídy C, pokud O je přímou instancí třídy C nebo O je přímou instancí třídy D a třída D je potomek třídy C.

Vestavěný predikát `isinstance` rozhoduje, zda je objekt instancí třídy:

```
>>> isinstance(checkbox, AtomicWidget)
True
>>> isinstance(entry, AtomicWidget)
True
```

Jak jsme viděli v případě tříd `Entry` a `Checkbox`, tak potomci tříd mohou definovat nové atributy a metody. Například třída `Entry` přidává vlastnost `text`.

Uvažujme třídu `Group`

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

    def get_items(self):
        return self.items[:] # kopie pole

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

a jejího potomka `RadiobuttonGroup`

```
class RadiobuttonGroup(Group):
    def __init__(self):
        super().__init__()

    def set_items(self, items):
        for item in items:
            if not isinstance(item, RadioButton):
                raise TypeError("item is not a radiobuton")
        self.items = items[:]
        return self
```

V instanci třídy `RadiobuttonGroup` nyní existují dvě metody jména `set_items`. První je metoda přidaná třídou `Group` a druhá třídou `RadiobuttonGroup`. Nastává otázka, jakou metodu vybrat pro obsluhu zprávy `set_items`.

Při zaslání zprávy *`message`* objektu se k obsluze vybere metoda jména *`message`* přidaná třídou C pro kterou platí, že neexistuje metoda jména *`message`* přidaná třídou D, kde třída D by byla potomkem třídy C.

Tedy k obsluze `set_items` se vybere metoda přidaná třídou `RadiobuttonGroup`.

Pokud třída definuje metodu, kterou zdědila, říkáme, že ji *přepisuje*. Například třída `RadiobuttonGroup` přepisuje metodu `set_items`.

Pokud metoda M přepisuje metodu N, můžeme v těle metody M zavolat metodu N následovně.

Výraz

`super().`*`message`*`(`*`arg1`*`, `*`arg2`*`, ...) => `*`value`*

`message`: zpráva
`value``, `*`arg1`*`, `*`arg2`*`, ...`: hodnoty

v těle M zavolá metodu N s argumenty *`receiver`*, *`arg1`*, *`arg2`*, ..., kde *`receiver`* je příjemce zprávy. Hodnota *`value`* je návratová hodnota meotdy N.

Do třídy `Polygon` přidáme volání přepsané metody:

```
def set_items(self, items):
    for item in items:
        if not isinstance(item, Point):
            raise ValueError("Items of polygon must be points.")
    super().set_items(items)
    return self
```

Výraz `super().set_items(items)` zavolá metodu třídy `Group`:

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

Již víme, že definice inicializace instance (`__init__`) je metoda. Systém při vytváření objektu zašle nově vytvořenému objektu zprávu `__init__` bez argumentů. Pokud metodu `__init__` přepisujeme, musíme nejprve zavolat přepisovanou metodu výrazem `super().__init__()`.

Metoda `__init__` ve třídě `RadiobuttonGroup`:

```
def __init__(self):
    super().__init__()
```

pouze volá přepsanou metodu. Metodu `__init__` tedy můžeme odstranit:

```
class RadiobuttonGroup(Group):
    def set_items(self, items):
        for item in items:
            if not isinstance(item, RadioButton):
                raise TypeError("item is not a radiobuton")
        super().set_items(items)
        return self
```