# Web Development with Python Lesson 14

>>>

Ing. Patrik Dendis

STEP IT ACADEMY BRATISLAVA


DESIGN PATTERN?
I THINK I HAVE HEARD ABOUT IT BEFORE
imgflip.com

# OBSAH PREZENTÁCIE

- Opakovanie
- Abstraktné triedy
- SOLID princípy
- S - single Responsibility Principle
- O - open-closed principle
- L - liskov Substitution Principle
- I - interface Segregation Principle
- D - dependency Inversion Principle

# OPAKOVANIE

- Čo sú návrhové vzory?
- Aké tri typy návrhových vzorov poznáte?
- Opíšte Singleton
- Opíšte Factory Method
- Opíšte Observer
- Opíšte MVC

# ABSTRAKTNÉ TRIEDY

Abstraktné triedy v Pythone slúžia ako základ pre iné triedy, ktoré z nich odvodzujú, a definujú sadu abstraktných metód, ktoré musia byť implementované v odvodených triedach.

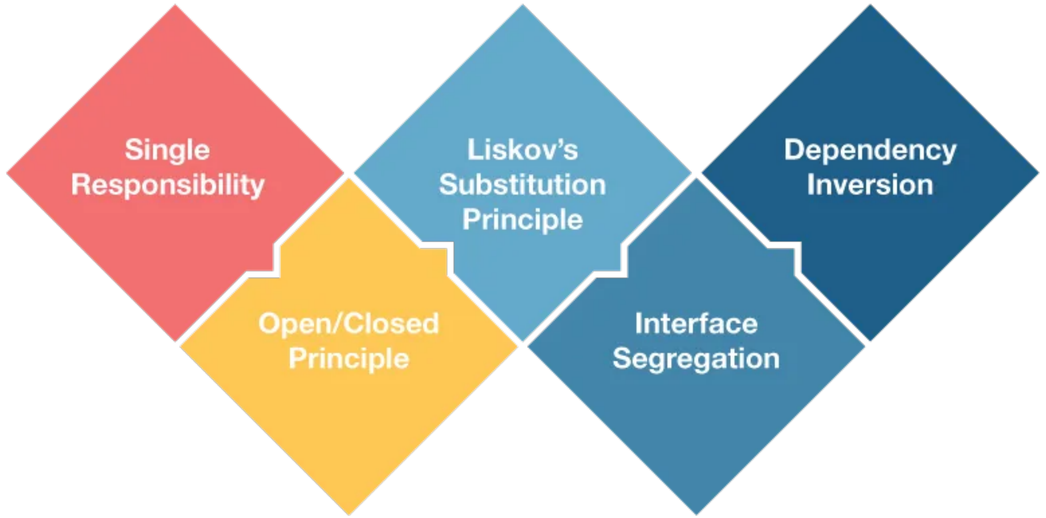
Abstraktná trieda sama o sebe nemôže byť inštanciovaná, čo znamená, že z nej nemôžete vytvoriť priamy objekt.

# ABSTRAKTNÉ TRIEDY

```
1   from abc import ABC, abstractmethod
2
3   class Zviera(ABC):
4       @abstractmethod
5       def zvuk(self):
6           pass
7
8   class Pes(Zviera):
9       def zvuk(self):
10          return "Haf"
11
12  moje_zviera = Zviera()
13
14  moj_pes = Pes()
15  print(moj_pes.zvuk())
```

# SOLID PRINCÍPY

princípy pomáhajú pri vytváraní softvérových systémov, ktoré sú ľahko udržiavateľné, rozšíriteľné a testovateľné



# S - Single Responsibility Principle (SRP)

Zásada jedinej zodpovednosti hovorí, že trieda by mala mať len jeden dôvod na zmenu. To znamená, že trieda by mala byť zodpovedná len za jednu funkcionalitu alebo oblasť záujmu, čo uľahčuje jej údržbu a testovanie.

# S - Single Responsibility Principle (SRP)



main.py s_bad.py

```python
class User:
    def __init__(self, name, last_name, age):
        self.name = name
        self.last_name = last_name
        self.age = age

    @property
    def age(self):
        return self._age

    @age.setter
    def age(self, age):
        if age < 0or age >= 130:
            raise ValueError("Age must be between 0 and 130 ")
        self._age = age

    # Displays user information to the console.
    def display(self):
        print(f"{self.name}{self.last_name}{self.age}")

    def input(self):
        self.name = input("Input name:")
        self.last_name = input("Input last name:")
        self.age = int(input("Input age:"))

# Create a User object, display its information,
# update it with user input, then display it again.
obj = User( name: "Bill", last_name: "Windows", age: 34)
obj.display()
obj.input()
obj.display()
```

# S - Single Responsibility Principle (SRP)



main.py s_bad.py s_good.py

```python
class User:
    def __init__(self, name, last_name, age):
        self.name = name
        self.last_name = last_name
        self.age = age

    @property
    def age(self):
        return self._age

    # Setter for age includes validation.
    @age.setter
    def age(self, age):
        if age < 0or age >= 130:
            raise ValueError("Age must be between 0 and 130 ")
        self._age = age

# Console class handles user I/O operations.
class Console:

    @staticmethod
    def display(obj):
        print(f"{obj.name}{obj.last_name}{obj.age}")

    @staticmethod
    def input():
        name = input("Input name:")
        last_name = input("Input last name:")
        age = int(input("Input age:"))
        return User(name, last_name, age)

obj = User( name: "Bill", last_name: "Windows", age: 34)
Console.display(obj)
obj = Console.input()
Console.display(obj)
```

# VÝZNAM

- **Lepšia udržateľnosť**: Keď má trieda alebo modul len jednu zodpovednosť, je oveľa jednoduchšie pochopiť, testovať a opravovať ju. Úpravy v jednej časti systému majú menšiu šancu nezamýšľane ovplyvniť iné časti.

- **Zjednodušené testovanie**: Triedy s jednou zodpovednosťou sú oveľa jednoduchšie na testovanie, pretože potrebujete testovať menej funkcionality v rámci jednej jednotky. To vedie k presnejším a efektívnejším testom.

- **Vyššia flexibilita a možnosti opätovného použitia**: Keď sú komponenty navrhnuté tak, aby riešili len jednu zodpovednosť, je ľahšie ich použiť v iných častiach aplikácie alebo dokonca v iných projektoch, pretože ich funkcionality sú dobre definované a izolované.

- **Lepšie paralelné vývojové práce**: Keďže zodpovednosti sú jasne oddelené, viacerí vývojári môžu pracovať na rôznych častiach systému súčasne bez veľkého rizika konfliktov alebo potreby častého zlučovania zmien.

# O - Open-Closed Principle (OCP)

Zásada otvorenosti/zatvorenosti uvádza, že softvérové entity (triedy, moduly, funkcie atď.) by mali byť otvorené pre rozšírenie, ale zatvorené pre úpravy. To znamená, že je možné pridávať nové funkcionality bez menenia existujúceho kódu.

# O - Open-Closed Principle (OCP)

main.py | s_bad.py | s_good.py | o_bad.py

```python
import io
import os

class Output:
    def __init__(self, data, output_type):
        # output_type determines the destination type for the data.
        self.output_type = output_type
        self.data = data

    def display(self):
        if self.output_type == "console":
            print(f"{self.data}")
        elif self.output_type == "file":
            file_dir = os.path.dirname(os.path.abspath(__file__))
            os.chdir(file_dir)
            with open('output.txt', 'w') as f:
                f.write(self.data)
        else:
            raise ValueError("Wrong type of output")

# Create an Output object with data "some string"
# and output_type "file", then call its display method.
obj = Output( data: "some string", output_type: "file")
obj.display()
```

# O - Open-Closed Principle (OCP)



main.py | s_bad.py | s_good.py | o_bad.py | o_good.py

```python
from abc import ABC, abstractmethod

# Abstract base class for outputs.
class Output(ABC):
    def __init__(self, data):
        self.data = data

    # Declare an abstract display method.
    @abstractmethod
    def display(self):
        pass

# Console output implementation.
class ConsoleOutput(Output):
    def display(self):
        print(f"{self.data}")

# File output implementation.
class FileOutput(Output):
    def display(self):
        with open('output.txt', 'w') as f:
            f.write(self.data)

# Create a ConsoleOutput object and display.
obj = ConsoleOutput("some string")
obj.display()

# Create a FileOutput object and display.
obj2 = FileOutput("another string")
obj2.display()
```

# VÝZNAM

- **Zlepšená udržateľnosť**: Keďže existujúci kód zostáva nezmenený pri pridávaní novej funkcionality, je menšia pravdepodobnosť, že rozšírenie zavedie chyby do už overeného a stabilného kódu.

- **Zvýšená flexibilita**: OCP umožňuje vývojárom pridávať nové funkcionality alebo reagovať na zmeny požiadaviek bez nutnosti zasahovať do existujúceho kódu. To znamená, že systém je flexibilnejší a lepšie sa prispôsobí budúcim potrebám.

- **Lepšia modularita**: Dodržiavanie OCP často vedie k návrhu, kde systém je rozdelený do dobre definovaných, modulárnych komponentov, čo uľahčuje pochopenie, testovanie a rozvoj softvéru.

- **Znížené riziko pri aktualizáciách**: Keďže existujúci kód zostáva nezmenený, aktualizácie a rozšírenia prinášajú menšie riziko narušenia funkcionality systému, čo vedie k stabilnejšiemu vývojovému cyklu.

# L - Liskov Substitution Principle (LSP)

Zásada substitúcie Liskovovej hovorí, že objekty v programe by mali byť nahraditeľné ich podtypmi bez ovplyvnenia správnosti programu. To znamená, že odvodené triedy musia byť schopné úplne nahradiť svoje nadtriedy.

# L - Liskov Substitution Principle (LSP)



```python
class Vtak:
    def lietat(self):
        pass

class Vrabec(Vtak):
    def lietat(self):
        return "Vrabec lieta"

class Kura(Vtak):
    def lietat(self):
        return "Kura nemôže lietať"

def spustit_lietanie(vtak):
    print(vtak.lietat())

vrabec = Vrabec()
kura = Kura()

spustit_lietanie(vrabec)
spustit_lietanie(kura)
```

# L - Liskov Substitution Principle (LSP)



```python
class Vtak:
    pass

class LietajuciVtak(Vtak):
    def lietat(self):
        pass

class Vrabec(LietajuciVtak):
    def lietat(self):
        return "Vrabec lieta"

class Kura(Vtak):
    pass

def spustit_lietanie(lietajuci_vtak):
    print(lietajuci_vtak.lietat())

vrabec = Vrabec()
spustit_lietanie(vrabec)
```

# VYŽITIE

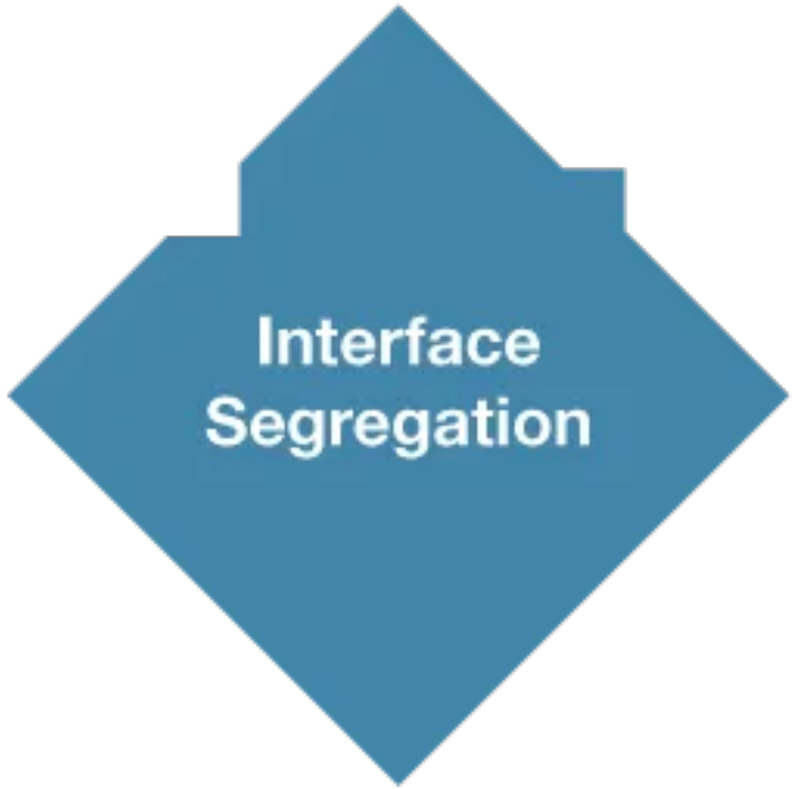
- **Zvýšená modulárnosť**: LSP umožňuje vytvárať systémy s vysokou úrovňou modularity. Komponenty sú navrhnuté tak, aby boli zameniteľné, čo uľahčuje výmenu alebo aktualizáciu implementácií bez ovplyvnenia zvyšku systému.

- **Lepšia udržateľnosť**: Keďže LSP zabezpečuje, že zmeny v podtypoch neovplyvnia nadtypy, je jednoduchšie udržiavať a rozširovať existujúci kód. To vedie k znižovaniu nákladov na údržbu a zlepšeniu celkovej kvality kódu.

- **Zvýšená opätovná použiteľnosť kódu**: Podtypy, ktoré dodržiavajú LSP, môžu byť použité v akomkoľvek kontexte, kde je očakávaný ich nadtyp. To umožňuje opätovné použitie kódu v rôznych častiach aplikácie alebo dokonca v rôznych projektoch.

# I - Interface Segregation Principle (ISP)

Zásada segregácie rozhraní uvádza, že klienti by nemali byť nútení závisieť na rozhraniach, ktoré nepoužívajú. To vedie k vytváraniu špecifických rozhraní namiesto jedného, veľkého, univerzálneho rozhrania.



# I - Interface Segregation Principle (ISP)



```python
from abc import ABC, abstractmethod

class Vtak(ABC):
    @abstractmethod
    def lietaj(self):
        pass

    @abstractmethod
    def chod(self):
        pass

class Pstros(Vtak):
    def fly(self):
        raise Exception("Pstros is not flying")

    def walk(self):
        print("Pstros is walking")

class Orol(Vtak):
    def fly(self):
        print("Orol is flying")

    def walk(self):
        print("Orol is walking")

try:
    obj = Orol()
    obj.fly()
    obj.walk()
    obj2 = Pstros()
    obj2.walk()
    obj2.fly()
except Exception as e:
    print(e)
```

# I - Interface Segregation Principle (ISP)



```python
from abc import ABC, abstractmethod

class Walkable(ABC):
    @abstractmethod
    def chod(self):
        pass

class Flyable(ABC):
    @abstractmethod
    def lietaj(self):
        pass

class Pstros(Walkable):
    def chod(self):
        print("Pstros is walking")

class Orol(Walkable, Flyable):
    def lietaj(self):
        print("Orol is flying")
    def chod(self):
        print("Orol is walking")

try:
    obj = Orol()
    obj.lietaj()
    obj.chod()

    obj2 = Pstros()
    obj2.chod()

except Exception as e:
    print(e)
```

# VYŽITIE

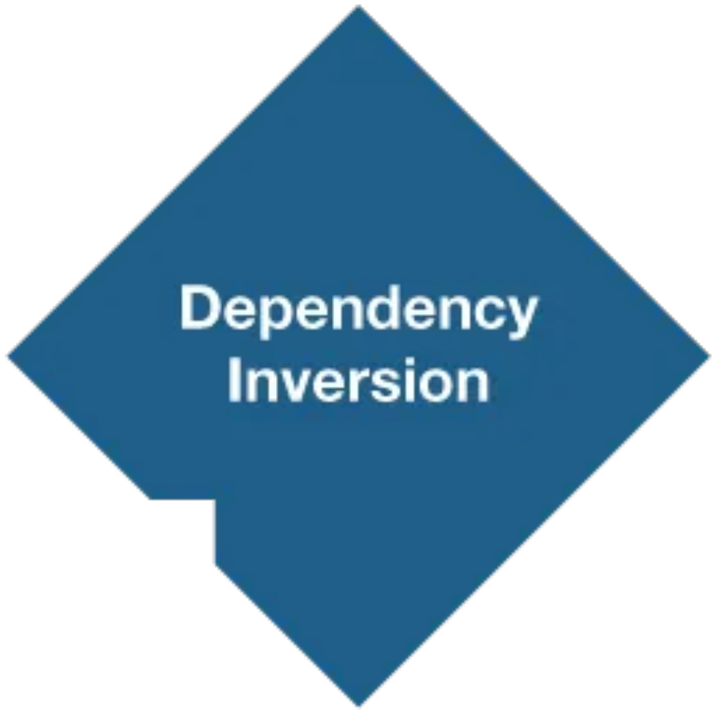
- **Zníženie závislostí**: Triedy nebudú závislé na metódach, ktoré nepoužívajú. Tým sa znižuje ich vzájomná závislosť, čo vedie k modularitejšiemu a pružnejšiemu dizajnu.

- **Lepšia čitateľnosť a udržateľnosť kódu**: Keď triedy implementujú len tie rozhrania, ktoré potrebujú, stáva sa kód jednoduchším na pochopenie a údržbu. Zmeny v jednom rozhraní majú menší dopad na triedy, ktoré toto rozhranie nepoužívajú.

- **Zvýšenie opätovnej použiteľnosti kódu**: Menšie, špecifické rozhrania sú ľahšie opätovne použiteľné v rôznych kontextoch, pretože sú zamerané na konkrétne funkcionality. Tým sa zvyšuje flexibilita pri návrhu a implementácii nových funkcií alebo systémov.

# D - Dependency Inversion Principle (DIP)

Zásada inverzie závislostí hovorí, že moduly vyššej úrovne by nemali závisieť na moduloch nižšej úrovne, ale oba by mali závisieť na abstrakciách. Taktiež abstrakcie by nemali závisieť na detailoch, ale detaily by mali závisieť na abstrakciách.



# D - Dependency Inversion Principle (DIP)



```python
class TextFileOperations:
    def __init__(self,path):
        self.path = path

    def read_text_data(self):
        with open(self.path, 'r') as file:
            data = file.read()
        return data

    def write_text_data(self,data):
        with open(self.path,'w') as file:
            file.write(data)

class TextOperations:
    def __init__(self,text_source):
        self.text_source = text_source
        self.data = self.text_source.read_text_data()

    # Searches for a
    def search_for_word(self, word):
        return word in self.data

    # Counts the occurrences of a word in the data
    def count_occurences(self, word):
        return self.data.count(word)

file = TextFileOperations("output.txt")

obj = TextOperations(file)
print(f"{obj.search_for_word('more')}")
print(f"{obj.count_occurences('be')}")
```

# D - Dependency Inversion Principle (DIP)



```python
from abc import ABC, abstractmethod

class DataSource(ABC):
    def __init__(self,path):
        self.path = path

    @abstractmethod
    def read_data(self):
        pass
    @abstractmethod
    def write_data(self):
        pass

class TextDataSource(DataSource):
    def read_data(self):
        with open(self.path, 'r') as file:
            data = file.read()
        return data
    def write_data(self,data):
        with open(self.path,'w') as file:
            file.write(data)

class DbDataSource(DataSource):
    def read_data(self):
        return "data from database"
    def write_data(self,data):
        print(f"write {data} to database")

class TextOperations:
    def __init__(self,data_source):
        self.data_source = data_source
        self.data = self.data_source.read_data()

    def search_for_word(self, word):
        return word in self.data

    def count_occurences(self, word):
        return self.data.count(word)

file = TextDataSource("output.txt")
db = DbDataSource("customers")

obj = TextOperations(file)
print(f"{obj.search_for_word('more')}")
print(f"{obj.count_occurences('be')}")

obj = TextOperations(db)
print(f"{obj.search_for_word('data')}")
print(f"{obj.count_occurences('from')}")
```

# VYŽITIE

- **Zvýšená modulárnosť**: DIP podporuje návrh, ktorý je viac modulárny, pretože moduly nie sú pevne viazané na konkrétne implementácie, ale na abstrakcie. To umožňuje ľahké vymieňanie a aktualizáciu modulov bez potreby zmeny v moduloch, ktoré od nich závisia.

- **Nižšia vzájomná závislosť komponentov**: DIP znižuje priamu závislosť medzi konkrétnymi modulmi, čo vedie k väčšej nezávislosti a izolácii komponentov. To zjednodušuje údržbu a aktualizáciu systému.

- **Zlepšená možnosť opätovného použitia kódu**: Abstrakcie umožňujú, aby boli komponenty navrhnuté tak, aby boli ľahko opätovne použiteľné v rôznych častiach aplikácie alebo dokonca v rôznych projektoch.

# ZHRNUTIE

https://www.youtube.com/watch?v=kF7rQmSRlq0

# ZADANIE

## Task 1

Create a class Shoes. Store the following data:

- type of shoes:
  - male,
  - female;
- type of shoes (sneakers, boots, sandals, dress shoes, etc.);
- color;
- price;
- brand;
- size.

Create necessary methods for this class. Implement the MVC pattern for the Shoes class and code to use a model, controller, and view.

# ZADANIE

## Task 2

Create a class Recipe. Store the following data:

- recipe name;
- recipe author;
- type of recipe (soups, salads, etc.);
- recipe text;
- a link to the recipe video;
- ingredients;
- cuisine (Italian, French, Ukrainian, etc.)

Create necessary methods for this class. Implement the MVC pattern for the Recipe class and code to use a model, controller, and view.

# ZADANIE

### Task 1

Create an app simulating a hot dog spot. The app should have the following functionality:

1. The user can choose among three standard hot dog recipes or create a custom one.
2. The user can choose whether to add mayonnaise, mustard, ketchup, toppings (sweet onions, jalapenos, chile, pickles, etc.)
3. Information about the ordered hot dog should be displayed on the screen and saved to a file.
4. If the user orders three and more hot dogs, provide a discount. The discound depends on the number of hot dogs.
5. Payment can be in cash or by card.
6. Provide the possibility to view the number of hot dogs sold, returns, profit.
7. Provide the possibility to view information about the availability of hot dog ingredients.
8. If the ingredients are about to run out, display a message about ingredients that need to be purchased.
9. Application classes must be created based on the SOLID principles and design patterns.

# ĎAKUJEM ZA POZORNOSŤ