OOP: konstruktor (výchozí, implicitní, obecný), výchozí parametry, dynamické objekty, metody, druhy dědičnosti

Konstruktor

- Konstruktor je v OOP speciální metoda třídy, která se volá ve chvíli vytváření instance této třídy
- Konstruktor je podobný ostatním metodám s tím rozdílem, že nevrací (return) žádnou hodnotu
- Úkolem konstruktoru je inicializace datových členů instance
  - Např.:

```
class User {
    constructor(name, surname) {
        this.name = name;
        this.surname = surname;
    }

    printName() {
        return `${this.name} ${this.surname}`
    }
}

const instance = new User("Vojta", "Jelinek")
const fullName = instance.printName();
console.log(fullName)
```

  -
  - Je vytvořena třída s názvem User
  - V této třídě je metoda construcor()
  - Tato metoda má dva parametry name, surname
  - Když je vytvořena instance této třídy pomocí příkazu new User zadávají se dva parametry, které přijímá metoda constructor
  - Díky vložení těchto dvou parametrů jsou nyní tyto dvě proměnné nadefinované na zvolené jméno a příjmení
- **Výchozí konstruktor**
  - Výchozí konstruktor je konstruktor, který lze vyvolat bez zadání jakýchkoliv argumentů

- Např.:
  -
```
class User {
    public:
        string name, surname;

        User() {
            name = "Vojta";
            surname = "Jelinek";
        }

        void printName() {
            cout << name + " " + surname << endl;
        }
};


int main() {
    User user;
    user.printName();
}
```
  - V tomto případě je konstruktér metoda User()
  - Jak si lže všimnout je použit výchozí konstruktér protože, tato metoda nemá žádné argumenty

Výchozí parametry

- Příklad:
  -
```
int fun(int x = 10, int y = 20) {
    return x + y;
}

int main() {
    int answer = fun();
    cout << answer << endl;
}
```
  - Tato funkce má jako výchozí parametry hodnoty x = 10 a y = 20
  - Pokud zavoláme tuto funkce a nezadáme jí žádné parametry dostaneme výsledek 30
  - Pokud by tato funkce bylo zavolána s jedním parametrem tento parametr by nahradil hodnotu x, pokud s dvěma byli by nahrazeny obě hodnoty
- Pokud je zadán jeden parametr jako výchozí, všechny následní parametry musí být také nastaveny jako výchozí

Dynamické objekty

- K vytváření a mazání dynamických objektů se používají operátory **new** a **delete**
- Dynamické objekty jsou v paměti alokovány při run-time
- Je zapotřebí použít pointer, aby bylo možné k tomuto objektu přistoupit
- Výhodou dynamických objektů je možnost pracovat s různě velikými daty, beztoho abychom je napřed znali
- Jsou pomalejší než statické objekty
- Dynamické objekty nejsou dealokovány po proběhnutí kódu, toto musí ohlídat sám programátor pomocí příkazu **delete**
- Příklad:
  - ```
    class Dynamic {
        public:
            Dynamic() {
                cout<<"Hello"<<endl;
            }
    };

    int main() {
        Dynamic *obj;
        obj = new Dynamic;
        delete obj;

        return 0;
    }
    ```
  - Objekt Dynamic byl vytvořen dynamicky pomocí pointeru a příkazu **new**
  - Následně byl z paměti dealokován pomocí příkazu **delete**

Metody

- Metody jsou funkci patřící nějaké třídě
- Metody umožňují práci s daty, které jsou uložené v třídě (instanci)
- Metody také poskytují rozhraní, které mohou jiné třídy využít k přístupu k datům uložených v objektu
    ```

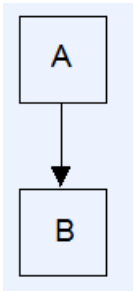
- Příklad:
  - ```
    class Metody {
        public:
            string name, surname;
            int vek;

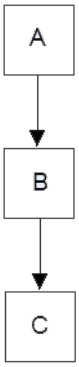
            Metody(string initName, string initSurname, int initVek) {
                name = initName;
                surname = initSurname;
                vek = initVek;
            };

            string metodaRekniAhoj() {
                return "Ahoj " + name + " " + surname;
            }
    };

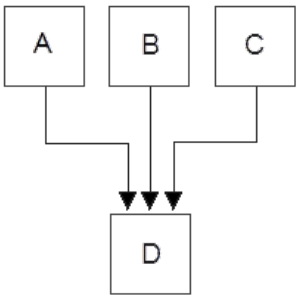
    int main() {
        Metody ukazka("John", "Doe", 19);
        string pozdraveni = ukazka.metodaRekniAhoj();
        cout << pozdraveni << endl;
    }
    ```
  - Tento objekt má metodu s názvem metodaRekniAhoj

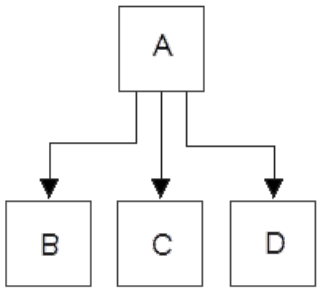
Druhy dědičností

- V OOP je dědičnost mechanismus přenášení vlastností jedné třídy na jinou třídu
- Single
  - 

  - Jedna třída dědí pouze z jedné jiné
- Multilevel
  - 

  - Třída dědí z jedné třídy a následně další třída dědí z této třídy
    ```

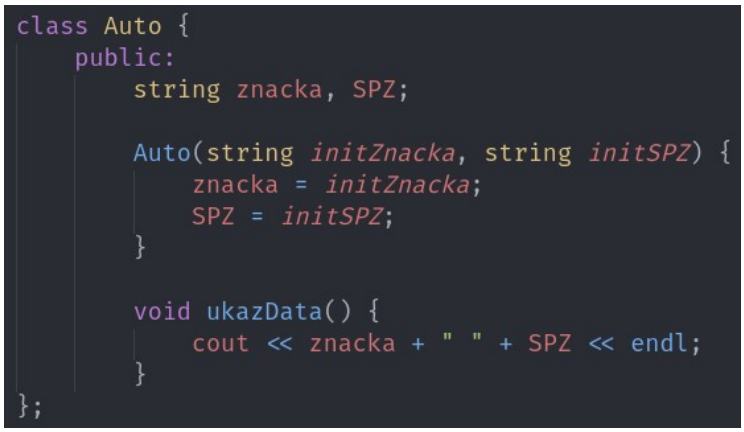
- Multiple
  - 

    - Jedna třída dědí z více tříd najednou
- Hiearchical
  - 

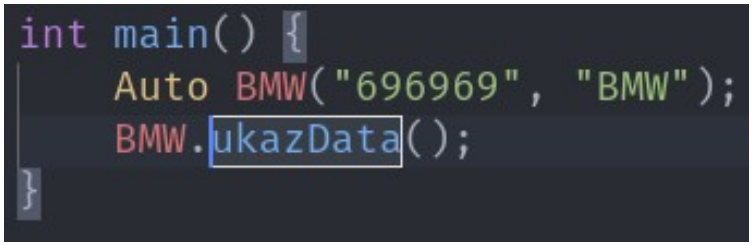
    - Z jedné hlavní třídy dědí více tříd najednou

OOP: Třída a instance třídy, modifikátory přístupu

Třída a instance třídy

- **Třída**
  - V OOP je třída vzor pro vytváření konkrétních instancí
  - Ve třídě jsou předdefinované její proměnné a metody
  - Příklad třídy:
    - 
      ```
      class Auto {
          public:
              string znacka, SPZ;

              Auto(string initZnacka, string initSPZ) {
                  znacka = initZnacka;
                  SPZ = initSPZ;
              }

              void ukazData() {
                  cout << znacka + " " + SPZ << endl;
              }
      };
      ```
      - Je vytvořena třída Auto
      - Tato třída funkce jako vzor pro vytváření instance jakéhokoliv auta
      - V této třídě nejsou předem nastavené hodnoty znacka a SPZ
      - Tyto hodnoty se nastaví až při vytvoření konkrétní instance
- **Instance**
  - Instance třídy je konkrétní datový objekt v paměti odvozený z nějakého vzoru
  - Každý objekt má své atributy a metody podle jeho vzoru (třídy)
  - Příklad:
    - 
      ```
      int main() {
          Auto BMW("696969", "BMW");
          BMW.ukazData();
      }
      ```
      - Zde je konkrétní instance třídy Auto
      - Je zde už nastavena značka tohoto auta i jeho SPZ
      - Dále je použita metoda, která je vytvořena ve třídě a tato metoda vypíše do konzole jaké data jsou uloženy v této instanci třídy

Modifikátory přístupu

- **public**
  - K těmto vlastnostem má přístup kdokoliv
  - Příklad:
    - 
      ```
      class Parent {
          public:
              int parentData = 1000;

              void showMyPrivate() {
                  cout << parentData << endl;
              }
      };

      class Child : public Parent {
          public:
              void showParentData() {
                  cout << this→parentData << endl;
              }
      };

      int main() {
          Parent p;
          cout << p.parentData << endl;

          Child c;
          c.showParentData();
      }
      ```
    - V tomto příkladě jsou parentData uložena jako public
    - To znamená že k nim má přístup kdokoliv
- **private**
  - K těmto vlastnostem má pouze přístup samotná třída
  - Příklad:
    - 
      ```
      class Parent {
          private:
              int parentData = 1000;
          public:
              void showMyPrivate() {
                  cout << parentData << endl;
              }
      };

      class Child : public Parent {
          public:
              void showParentData() {
                  cout << this→parentData << endl;
              }
      };

      int main() {
          Parent p;
          p.showMyPrivate();

          Child c;
      }
      ```
    - V tomto příkladě má k datům přístup pouze samotná třída parent
    - Takže ani třída Child, která dědí z této třídy k těmto datům nemá přístup
- **protected**
  - K těmto vlastnostem má pouze přístup třída, která dědí z této třídy

- Příklad:
  - 

```
class Parent {
    protected:
        int parentData = 1000;
    public:
        void showMyPrivate() {
            cout << parentData << endl;
        }
};

class Child : public Parent {
    public:
        void showParentData() {
            cout << this→parentData << endl;
        }
};

int main() {
    Parent p;
    p.showMyPrivate();

    Child c;
    c.showParentData();
}
```

    - V tomto příkladě má k datům přístup pouze třída Parent a všechny ostatní třídy, které z této třídy dědí

Výjimky, ladění, druhy a ošetření chyb: try, except, chybová událost

Výjimka

- Je to výjimečná situace, která může nastat za běhu programu
- Jedná se o zobecnění vnitřního přerušení vyvolaného chybou při provádění programu
- Ve většině programovacích jazyků se vznik výjimky v kódu hlídá pomocí klíčového slova **try**
- Výskyt výjimky se ošetřuje pomocí bloku kódu, který většinou uvozuje klíčové slovo **catch,** nebo **except**
- V tomto bloku kódu je k dispozici datová struktura nesoucí informace o chybovém stavu
- Výjimky mohou nastat buď nějakou chybou při běhu kódu nebo mohou být vyvolány úmyslně pomocí klíčového slova **throw**

Ladění (Debugging)

- Je to proces pro detekování a odstraňování existujících čí potencionálních chyb ("bugů")
- Buggem je myšlena nějaká část kódu, která se chová neočekávaně a může způsobit nějaký problém (error)
- Existuje mnoho způsobů jak debuggovat program od vypisování důležitých proměnných do konzole po využívání nějakých programů na debuggovaní (debugger)
- Debugger může například postupně vykonávat jeden příkaz za druhým a programátor může při tomto procesu sledovat různé stavy proměnných apod.

Try except

- Příklad:
  - ```
    def division():
        a = int(input("Number 1: "))
        b = int(input("Number 2: "))

        if b == 0:
            raise Exception("Can divide by zero")

        print(a/b)

    try:
        division()
    except Exception as err:
        print(err)
    ```
  - Je vytvořena funkce divsion, díky které se zadají dvě hodnoty a, b
    ```

- Pokud bude hodnota b rovna nule vytvoří se výjimka
- Následně pomocí bloku **try** se tato funkce zavolá
- Pokud nastane nějaká výjimka při vyvolání této funkce zachytí se pomocí bloku **except**
- Tento blok následně vypíše na konzoly výjimku, která byla vytvořena ve funkce division