PROGRAMIRANJE II

Abstraktni razredi

- Pomembna aktivnost pri načrtovanju programa je poiskati potrebne razrede za dani problem. Ko razrede identificiramo jih poskušamo uvrstiti v hierarhijo (relacija nadrazred – podrazred)
- Nadaljnji korak je poiskati še neidentificirane razrede, ki so posplošitev (generalizacija) danih razredov.

Abstraktni razredi

- Postopek generalizacije nas lahko pripelje do takšnih razredov za katere vemo, da njihovih objektov ne bomo kreirali.
- Razred želimo imeti v hierarhiji samo zato, ker so posplošitev že znanih razredov in zaradi uporabe polimorfnih (virtualnih) metod.
- Vsem objektom bomo poslali isto sporočilo, objekti pa se bodo odzvali na njim svojstven način.

Abstraktni razredi

- Takšnim razredom pravimo **abstraktni razredi** (*abstract classes*)
- Ostalim razredom pravimo tudi **konkretni razredi** (*concrete classes*)
- Primeri abstraktni razredov:
  - Žival – Pes, Muca, …
  - Lik – Krog, Kvadrat, …
  - Vozilo – Letalo, Vlak, Kolo, …

Abstraktni razredi

- Mnogo metod v abstraktnih razredih ne znamo implementirati (npr. kako definirati ploščino Lika).
- Takšne metode imenujemo abstraktne metode ali tudi čisto virtualne (*pure virtual*).

Abstraktni razredi

- Abstraktne metode zato nimajo implementacije telesa metode.
- To zapišemo v C++ kot

```
tip ImeFun(Args) = 0;
```

- Prav tako ne moremo ustvarjati objektov abstraktnih razredov, saj se takšni objekti ne bi znali odzivati na sporočila.

Primer 9

// Animal.h

```
class Animal {   // abstract class
public:
    virtual ~Animal() {}
    virtual void voice() const = 0;   // abstract method
};
```

// Cat.h

```
#include <iostream>

class Cat : public Animal {
public:
    void voice() const {
        std::cout << "meow" << std::endl;
    }
};
```

// Dog.h

```
#include <iostream>

class Dog : public Animal {
public:
    void voice() const {
        std::cout << "bark" << std::endl;
    }
};
```

// Cow.h

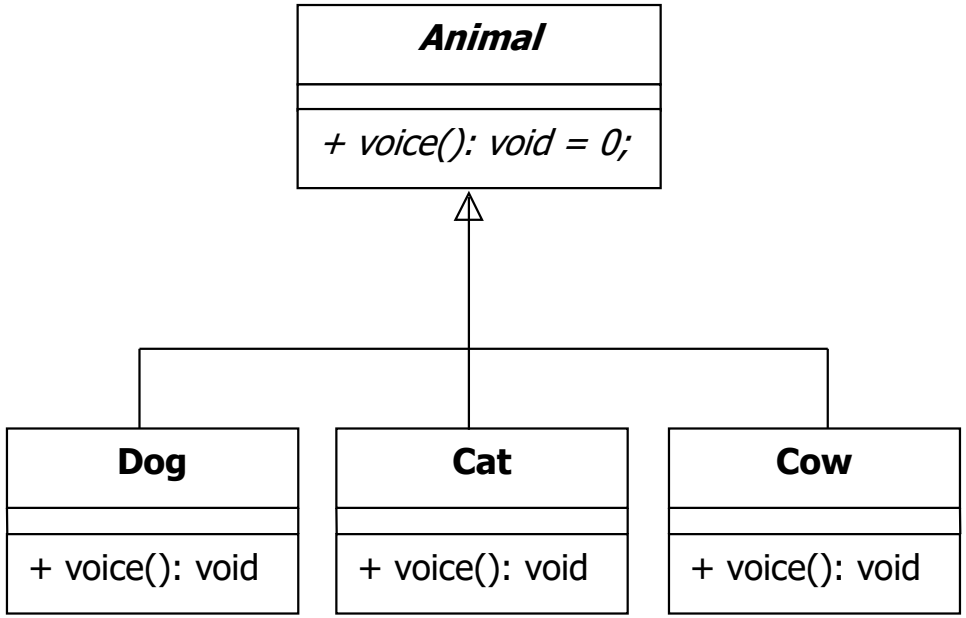
```
#include <iostream>

class Cow : public Animal {
public:
    void voice() const {
        std::cout << "moo" << std::endl;
    }
};
```

Primer 9

// Example09.cpp

```
#include <iostream>
#include "Animal.h"
#include "Dog.h"
#include "Cat.h"
#include "Cow.h"

int main() {
    Animal* zoo[4];

    //zoo[0] = new Animal;
    zoo[0] = new Dog;
    zoo[1] = new Cat;
    zoo[2] = new Dog;
    zoo[3] = new Cow;
    for (int i=0; i < 4; i++)
      zoo[i]->voice();

    for (int i=0; i<4; i++)
        delete zoo[i];
   return 0;
}
```

UML notacija



Primer 10 (agregacija)

//Employee.h

```
#include <iostream>

class Employee {
private:
    std::string name;
public:
    Employee(std::string name) : name(name) {
        std::cout << "Employee::constructor" << std::endl;
    }
    virtual ~Employee() {
        std::cout << "Employee::destructor" << std::endl;
    }
    virtual const std::string& toString() const {
        return name;
    };
};
```

Primer 10 (agregacija)

//Company.h

```
#include <iostream>
#include "Employee.h"

class Company {
private:
    std::string name;
    Employee* ptrEmployee;  // aggregation
public:
    Company(std::string cname, Employee* ename) : name(cname), ptrEmployee(ename) {
        std::cout << "Company::constructor" << std::endl;
    }
    virtual ~Company() {
        std::cout << "Company::destructor" << std::endl;
    }
    virtual const std::string& toString() const {
        return name;
    };
    virtual void employed() const {
        std::cout << ptrEmployee->toString();
    }
};
```

Primer 10 (agregacija)

//Example10.cpp

```
#include <iostream>
#include "Employee.h"
#include "Company.h"

int main() {
    std::cout << "Aggregation example" << std::endl;
    {
        std::cout << "----------- nested block 1 begin -----------" << std::endl;
        Employee* ptrEmp = new Employee("John");
        {
            std::cout << "----------- nested block 2 begin -----------" << std::endl;
            Company c("SmartCo", ptrEmp);
            std::cout << "Employee ";
            c.employed();
            std::cout << " works for company " << c.toString() << std::endl;
            std::cout << "----------- nested block 2 end -----------" << std::endl;
        }
        std::cout << "Company doesn't exist anymore" << std::endl;
        std::cout << "But, employee " << ptrEmp->toString();
        std::cout << " still exists!" << std::endl;
        std::cout << "----------- nested block 1 end -----------" << std::endl;
        delete ptrEmp;
    }
    return 0;
}
```

Primer 11 (kompozicija)

//Room.h

```
#include <iostream>

class Room {
private:
    std::string name;
public:
    Room(std::string name) : name(name) {
        std::cout << "Room::constructor" << std::endl;
    }
    virtual ~Room() {
        std::cout << "Room::destructor" << std::endl;
    }
    virtual void print() const {
        std::cout << "Room " << name << std::endl;
    }
};
```

Primer 11 (kompozicija)

//House.h

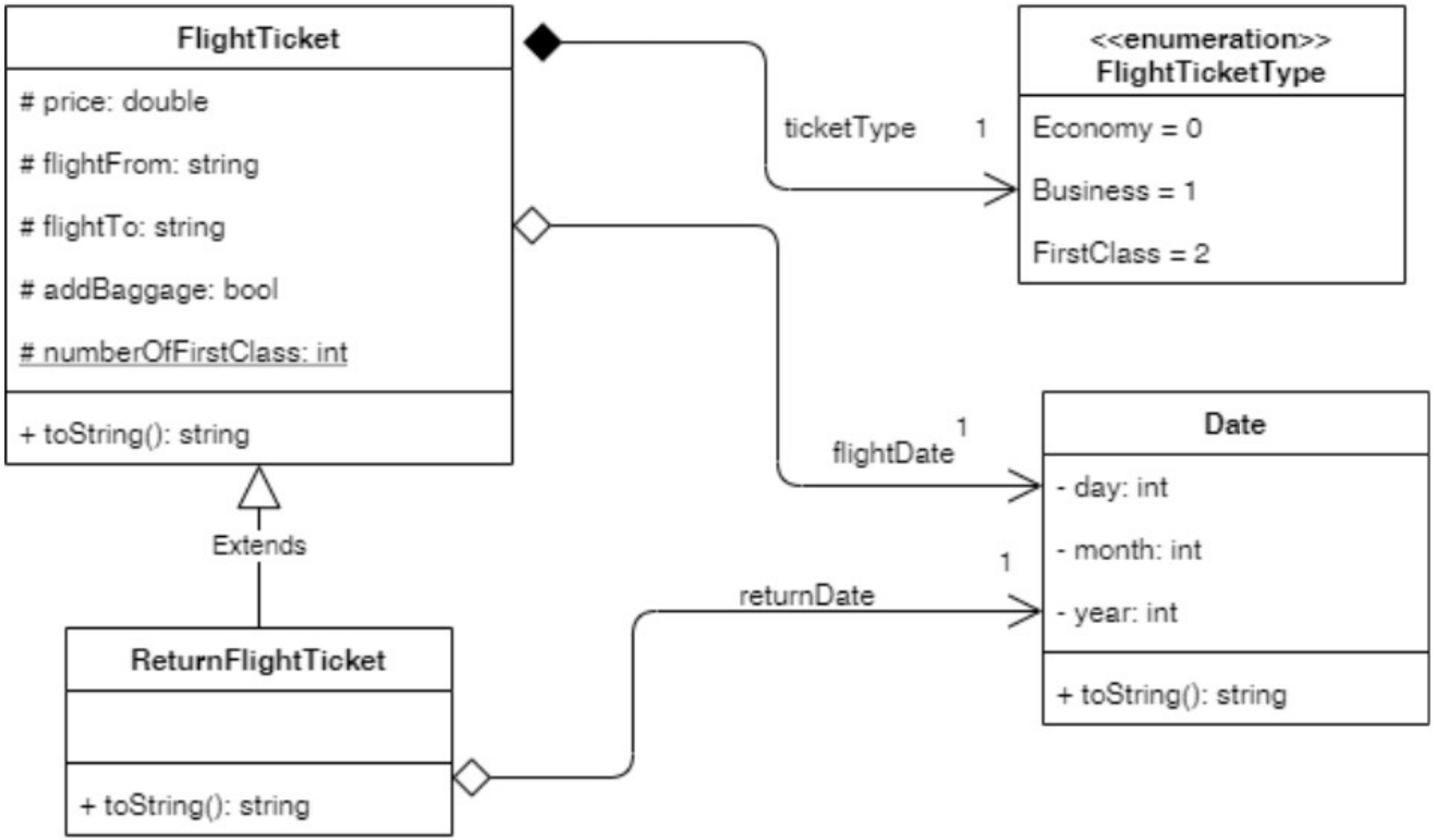
```
#include <iostream>
#include "Room.h"

class House {
private:
    std::string name;
    Room kitchen, livingRoom, bedroom;
public:
    House(std::string name) : name(name), kitchen("Kitchen1"), livingRoom("living room 1"),
                              bedroom("TV bedroom") {
        std::cout << "House::constructor" << std::endl;
    }
    virtual ~House() {
        std::cout << "House::destructor" << std::endl;
    }
    virtual void print() const {
        std::cout << "House name: " << name << std::endl;
        kitchen.print();
        livingRoom.print();
        bedroom.print();
    }
};
```

Primer 11 (kompozicija)

//Example11.cpp

```
int main() {
    std::cout << "Composition example" << std::endl;
    {
        std::cout << "----------- nested block begin -----------" << std::endl;
        House h1("My home");
        h1.print();
        std::cout << "----------- nested block end -----------" << std::endl;
    }
    std::cout<<"House and rooms don't exist anymore!" << std::endl;
    return 0;
}
```

Primer 12 (dedovanje in vsebovanje)



Primer 12 (dedovanje in vsebovanje)

Naštevni tip enum in enum class

Nekateri programski jeziki omogočajo uporabnikom definirati popolnoma nov osnovni tip z naštevanjem vrednosti. Takšen osnovni tip imenujemo naštevni tip ("enumeration type") in posamezno vrednost enumerand ("enumerand").

V C++ definiramo naštevni tip z naštevanjem identifikatorjev, ki predstavljajo njihove vrednosti.

```
enum BarvaKarte {srce, karo, pik, kriz};
```

Enumerandom pa lahko priredimo celoštevilčne vrednosti:

```
enum StevEnum { ena = 1, dva = 2, tri = 3, stiri = 4, pet = 5,
                sest = 6, sedem = 7, osem = 8, devet = 9, deset = 10 };
```

Primer 12 (dedovanje in vsebovanje)

```
struct Karta {
        BarvaKarte barva;
        StevEnum stev;
};
Karta mojaKarta;
mojaKarta.barva = karo;
mojaKarta.stev = dva;
cout << (mojaKarta.barva+1 == mojaKarta.stev) << endl;
```

C++11 uvede **enum class**, kjer

- ni dovoljena implicitna pretvorba v int
- ni možna primerjava med različnimi naštevnimi tipi

Primer 12 (dedovanje in vsebovanje)

//Example12.cpp

```
#include <iostream>
#include "FlightTicket.h"
#include "ReturnFlightTicket.h"

int main() {
    Date* ptrDate1 = new Date(17, 3, 2020);
    Date* ptrDate2 = new Date(18, 3, 2020);
    Date* ptrDate3 = new Date(19, 3, 2020);
    Date* ptrDate4 = new Date(29, 3, 2020);
    Date* ptrDate5 = new Date(31, 3, 2020);
    FlightTicket* ptrFlights[3];
    ptrFlights[0] = new FlightTicket(400, "Moskva", "Ljubljana", false, FlightTicketType::Economy, ptrDate1);
    ptrFlights[1] = new ReturnFlightTicket(1000, "Hong Kong", "Washington", true,
                                           FlightTicketType::FirstClass, ptrDate2, ptrDate4);
    ptrFlights[2] = new ReturnFlightTicket(800, "Vienna", "Chicago",
                                           false, FlightTicketType::Economy, ptrDate3, ptrDate5);
    for (int i=0; i<3; i++)
        std::cout << ptrFlights[i]->toString() << std::endl;

    delete ptrDate1;
    delete ptrDate2;
    delete ptrDate3;
    delete ptrDate4;
    delete ptrDate5;
    for (int i=0; i<3; i++)
        delete ptrFlights[i];
    return 0;
}
```

Vprašanja