Изпитна тема № 4: Обектно-ориентирано програмиране

1.

1.1. Познава концепцията за типизиране на класове, чрез шаблонни класове и методи. (4 т.)

Типизирането на клас (създаването на шаблонен клас) представлява добавяне, към декларацията на един клас, на параметър (заместител) на неизвестен тип, с който класът ще работи по време на изпълнение на програмата. В последствие, когато класът бива инстанциран, този параметър се замества с името на някой конкретен тип.

1.2. Демонстрира създаването и употребата на шаблонни класове и методи. (4 т.)

```
using System;

class Point<T>
{
   private T m_x;
   private T m_y;

   public Point(T x, T y)
   {
      m_x = x;
      m_y = y;
   }

   public T X
   {
      get { return m_x; }
      set { m_x = value; }
   }

   public T Y
   {
      get { return m_y; }
      set { m_y = value; }
   }
}
```

2.

2.1 Описва и обяснява концепцията за наследяване на класове. (6 т.)

Наследяването е основен принцип от обектно-ориентираното програмиране. То позволява на един клас да "наследява" (поведение и характеристики) от друг, по-общ клас. Например лъвът е от семейство котки. Всички котки имат четири лапи, хищници са, преследват жертвите си. Тази функционалност може да се напише веднъж в клас Котка и всички хищници да я преизползват – тигър, пума, рис и т.н. Концепцията за наследяване ни помага да „изнесем“ общата функционалност на няколко класове, които са написани, в един общ. Ползата от това е, че с вдигането на нивото на абстракция, се получава по-четим и по-лесно можем да рефакторираме кода.

2.2 Демонстрира наследяването на класове. (6 т.)

```
2.  using System;
3.
4.  public class Character
5.  {
6.      protected int m_height;
7.      protected string m_hairColor;
8.      protected string m_name;
9.      protected string m_weapon;
10.
11.     public Character(int heihgt, string hairColor, string name, string weapon)
12.     {
13.         m_height = heihgt;
14.         m_hairColor = hairColor;
15.         m_name = name;
16.         m_weapon = weapon;
17.     }
18.
19.     public void Print()
20.     {
21.         Console.WriteLine($"Name: {m_name}\nHeihgt: {m_height}\nHair color: {m_hairColor}\nWeapon: {m_weapon}\n");
22.     }
23. }
24.
25. public class Warrior : Character
26. {
27.     public Warrior(int heihgt, string hairColor, string name, string weapon = "sword")
28.         : base(heihgt, hairColor, name, weapon)
29.     {}
30.
31.
32. }
```

```
33. public class Archer : Character
34. {
35.     public Archer(int heihgt, string hairColor, string name, string weapon = "bow")
36.         : base(heihgt, hairColor, name, weapon)
37.     { }
38. }
39.
40. public class Program
41. {
```

```
42.     static void Main(string[] args)
43.     {
44.         Character warrior = new Warrior(180, "blonde", "Kokala");
45.         Character archer = new Archer(170, "black", "Dimitur Penev", "crossbow");
46.
47.         warrior.Print();
48.         archer.Print();
49.     }
50. }
```

3.Различава презаписване (override) и презареждане (overload) на метод. (4 т.)

Method Overloading

Когато в даден клас декларираме един метод, чието име съвпада с името на друг метод, но сигнатурите на двата метода се различават по списъка от параметри (броят на елементите в него или подредбата им), казваме, че имаме различни варианти на този метод (method overloading).

Пример:

```
static void Draw(string str)
{
// Draw string
}
static void Draw(int number)
{
// Draw integer
}
static void Draw(float number)
{
// Draw float number
}
```

Method Overriding

Презаписването на метод означава промяна на имплементацията на наследен метод.

- Ако декларираме метод като виртуален в базовия клас, можем да го заменим производен клас.

Пример:

```
public class Shape
{
 public virtual Draw()
 {
 // Default implementation for all derived classes
 }
}

public class Circle : Shape
{
```

```
 public override Draw()
 {
 // Changed implementation
 }
}
```

4.

4.1. Посочва принципите на обектно-ориентираното програмиране. (4 т.)

Абстракция – Способността на една програма да игнорира някои аспекти на информацията, с която работи – способността да се съсредоточава върху съществено. Всеки обект в системата служи като модел на един абстрактен „агент“, който може да извършва дадена работа, да променя състоянието си и да докладва за него, и да „общува“ с други обекти в системата без да разкрива как са реализирани тези свойства. Процесите, функциите или методите също подлежат на абстракция и когато те са абстрактни се използват разнообразни техники за разширяване на една абстракция:

Капсулиране – наричано също „скриване на информация“: Прави невъзможно за потребителите на даден обект да променят неговото вътрешно състояние по неочакван начин; само вътрешните методи на обекта имат достъп до неговото състояние. Всеки клас има интерфейс, който определя как другите класове могат да взаимодействат с него като по този начин е черна кутия за другите класове и останалата част от програмите, които го използват. Това предпазва потребителите от разстройване на инвариантите на класа, което е полезно, защото позволява реализацията на един клас обекти да бъде променена в аспекти, които не са достъпни чрез интерфейса без това да изисква промени в потребителските програми.

Полиморфизъм – Различни неща или обекти могат да имат еднакъв интерфейс или да отговарят на едно и също (по наименование) съобщение и да реагират подходящо в зависимост от природата или типа на обекта. Това позволява много различни неща да бъдат взаимозаменими. Например, ако една птица получи съобщение „движи се бързо“, тя ще маха с крила и ще лети. Ако един лъв получи същото съобщение, той ще тича, използвайки краката си. И двете животни отговарят на една и съща молба по начини, които са подходящи за всяко от тях. По този начин, една променлива в програмния текст може да съдържа различни обекти по време на изпълнение на програмата и да извиква различни методи по различно време на изпълнение.

Наследяване – Организира и подпомага полиморфизма и капсулирането, като позволява да бъдат дефинирани и създавани обекти, които са специализирани варианти на вече съществуващи обекти. Новите обекти могат да използват (и разширяват) вече дефинираното поведение, без да е необходимо да реализират

това поведение отново. Това обикновено се прави чрез групиране на обектите в класове и дефиниране на нови класове, които разширяват съществуващи класове и подреждане по този начин на класовете в дървета или решетки, отразяващи сходствата в тяхното поведение. Докато използването на класове е най-популярната техника за наследяване, има и друга известна техника, наречена Прототипно-базирано програмиране.

4.2. Дава примери за приложението им. (8 т.)

Наследяване:

```
public class Felidae
{
private bool male;
public Felidae() : this(true) { }
public Felidae(bool male)
{
this.male = male;
}
public bool Male
{
get => this.male;
set => this.male = value;
}
}

public class Lion : Felidae
{
private int weight;
public Lion(bool male, int weight) : base(male) =>
this.weight = weight;
public int Weight
{
get => this.weight;
set => this.weight = value;
}
}
```

Абстракция:

```
public class AbstractionExample
{
public static void Main()
{
Lion lion = new Lion(true, 150);
Felidae bigCat1 = lion;
```

```
AfricanLion africanLion = new AfricanLion(true, 80);
Felidae bigCat2 = africanLion;
}
}
```

Капсулация:

```
public class Felidae
{
public virtual void Walk()
{
// ...
}
// ...
}
public class Lion : Felidae, Reproducible<Lion>
{
// ...
private Paw frontLeft;
private Paw frontRight;
private Paw bottomLeft;
private Paw bottomRight;
private void MovePaw(Paw paw)
{
// ...
}
public override void Walk()
{
this.MovePaw(frontLeft);
this.MovePaw(frontRight);
this.MovePaw(bottomLeft);
this.MovePaw(bottomRight);
}
// ...
}
```

Публичният метод Walk() извиква 4 пъти някакъв друг скрит (private) метод. Така базовият клас е кратък – само един метод. Имплементацията обаче извиква друг метод, също част от имплементацията, но скрит за ползвателя на класа. Така класът Lion не разкрива публично информация за това как работи вътрешно и това му дава възможност на по-късен етап да промени имплементацията си без останалите класове да разберат.

Полиморфизъм:

```
class Zoo
 {
```

```
class Animal
 {
 abstract string MakeNoise ();
 }

class Cat : Animal {

string MakeNoise () {
 return "Meow";
 }
 }

 class Dog : Animal {

 string MakeNoise () {
 return "Bark";
 }
 }

 public static void Main ()
 {
 //този метод не знае за типа на шум който животното ще вдига
 Animal animal = Zoo.GetAnimal ();
 Console.WriteLine(animal.MakeNoise());
 }
```

5

5.1. Описва абстрактни класове и интерфейси. (2 т.)

Какво става, ако искаме да кажем, че класът Felidae е непълен и само наследниците му могат да имат инстанции? Това става с ключовата дума abstract пред името на класа и означава, че класът не е готов и не може да бъде инстанциран. Такъв клас се нарича абстрактен клас. А как да укажем коя точно част от класа не е пълна? Това отново става с ключовата дума abstract пред името на метода, който трябва да бъде имплементиран. Този метод се нарича абстрактен метод и не може да притежава имплементация, а само декларация.

В езика C# интерфейсът е дефиниция на роля (на група абстрактни действия). Той дефинира какво поведение трябва да има един обект, без да указва как точно се реализира това поведение. Интерфейсите са още познати като договори. Един обект може да има много роли (да имплементира много интерфейси) и ползвателите му могат да го използват от различни гледни точки.

5.2. Различава абстрактен клас и интерфейс. (4 т.)

Интерфейсът е напълно нереализиран клас, използван за деклариране на набор от операции върху обектите на класа.

Абстрактният клас е частично реализиран клас. Някои от неговите методи са дефинирани и са общи за всички подкласове от следващо ниво. Останалите методи се дефинират в подкласовете на следващото ниво според тяхните изисквания.

Интерфейсът позволява да се разработи множествена наследственост.

5.3. Демонстрира употребата на интерфейси и абстрактни класове. (12 т.)

Интерфейси

```
interface ICharacter
{
   void Print();
}
public class Character : ICharacter
{
   protected int m_height;
   protected string m_hairColor;
   protected string m_name;
   protected string m_weapon;

   public Character(int heihgt, string hairColor, string name, string weapon)
   {
      m_height = heihgt;
      m_hairColor = hairColor;
      m_name = name;
      m_weapon = weapon;
   }

   public void Print()
   {
      Console.WriteLine($"Name: {m_name}\nHeihgt: {m_height}\nHair color: {m_hairColor}\nWeapon: {m_weapon}\n");
   }
}
```

Абстрактни класове

```
public abstract class Character
{
   protected int m_height;
   protected string m_hairColor;
   protected string m_name;
   protected string m_weapon;

   public Character(int heihgt, string hairColor, string name, string weapon)
   {
      m_height = heihgt;
      m_hairColor = hairColor;
      m_name = name;
      m_weapon = weapon;
   }

   public virtual void Print()
   {
      Console.WriteLine($"Name: {m_name}\nHeihgt: {m_height}\nHair color: {m_hairColor}\nWeapon: {m_weapon}\n");
   }
}

public class Warrior : Character
```

```
{
   public Warrior(int heihgt, string hairColor, string name, string weapon = "sword")
      :base(heihgt, hairColor, name, weapon)
   {}

   public override void Print()
   {
      base.Print();
   }
}
```

5.4. Прави заключения и изводи за употребата на интерфейси и абстрактни класове. (8 т.)

Кога да използваме абстракция и интерфейси?

Отговорът на този въпрос е: винаги, когато искаме да постигнем абстракция на данни или действия, чиято имплементация по-късно може да се подмени. Код, който комуникира с друг код чрез интерфейси, е много по-издръжлив срещу промени, отколкото код, написан срещу конкретни класове. Работата през интерфейси е често срещана и силно препоръчвана практика – едно от основните правила за писане на качествен код.

Кога да пишем интерфейси?

Винаги е добра идея да се използват интерфейси, когато се предоставя функционалност на друг компонент. В интерфейса се слага само функционалността (като декларация), която другите трябва да виждат. Вътрешно в една програма/компонент интерфейсите могат да се използват за дефиниране на роли. Така един обект може да се използва от много класове чрез различните му роли.

6. Дефинира понятието полиморфизъм и различава видовете полиморфизъм.

6.1. Дефинира понятието полиморфизъм. (2 т.)

Полиморфизмът позволява третирането на обекти от наследен клас като обекти от негов базов клас. Например големите котки (базов клас) хващат жертвите си (метод) по различен начин. Лъвът (клас наследник) ги дебне, докато Гепардът (друг клас-наследник) просто ги надбягва.

.

6.2. Различава видовете полиморфизъм. (4 т.)

Има два вида полиморфизъм:

- Полиморфизъм на времето за компилиране (method overloading)
- Полиморфизъм по време на изпълнение (method overriding)

Полиморфизмът по време на компилиране в C# е съществуването на множество методи с едно и също име, но с различни аргументи по тип и/или брой. Наричаме го също method overloading. Можем да го използваме в ситуации, в които трябва да внедрим множество методи с подобна функционалност и избираме да им дадем едно и също име.

Полиморфизмът по време на изпълнение е известен още като късно свързване(late binding). Тук името на методите и броят на параметрите и типа (сигнатурата на метода) трябва да са еднакви, но методите може да имат различна реализация. Отмяната(overriding) на метода е пример за динамичен полиморфизъм. Отмяната на метода може да се извърши с помощта на наследяване. При отмяната на метод е необходимо методът в основният клас и методът в производният клас да имат една и съща сигнатура. Компилаторът ще реши кой метод да извика по време на изпълнение и ако не бъде намерен метод, тогава хвърля грешка.