Тема 4

ОБЕКТНО-ОРИЕНТИРАНО ПРОГРАМИРАНЕ

1. Познава концепцията за типизиране на класове, чрез шаблонни класове и методи. Демонстрира създаването и употребата на шаблонни класове и методи.

Когато за работата на един метод е нужна **допълнителна информация**, тази информация се подава **на метода чрез параметри**. По време на изпълнение на програмата, при извикване на метода, подаваме аргументи на метода, те се присвояват на параметрите му и след това се използват в тялото на метода.

По подобие на методите, когато знаем, че **функционалността (действията) капсулирана в един клас**, може да бъде **приложена** не само към обекти от един, а **от много (разнородни) типове**, и тези типове не са известни по време на деклариране на класа, можем да използваме една функционалност на езика C# наречена **шаблонни типове (generics).**

Типизирането на метод (известно още с термина “шаблонен метод” / “generic method”) се прави, като веднага след името и преди отварящата кръгла скоба на метода, се добави <K>, където K е заместителят на типа, който ще се използва в последствие:

```
<return_type> <methods_name><K>(<params>)
```

Типизирането на клас (създаването на шаблонен клас) представлява добавянето към декларацията на един клас, 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.

Декларация на типизиран (шаблонен клас)

Формално, типизирането на класове се прави, като към декларацията на класа, след самото име на класа се добави **<T>**, където **T** е заместителят (параметърът) на типа, който ще се използва в последствие:

```
[<modifiers>] class <class_name><T>
{
}
```

Създаването и употребата на шаблонни класове и методи.

Приют за животни-**AnimalShelter**

```
class AnimalShelter<T>
{
   // Class body here ...
}
```

- шаблон на нашия клас AnimalShelter, който в последствие ще конкретизираме, заменяйки T с конкретен тип, например Dog.

Използването на типизирани класове става по следния начин:

```
<class_name><concrete_type> <variable_name> =
new <class_name><concrete_type>();
```

Ако искаме да създадем два приюта, един за кучета и един за котки,
ще трябва да използваме следния код:

```
AnimalShelter<Dog> dogsShelter = new AnimalShelter<Dog>();
AnimalShelter<Cat> catsShelter = new AnimalShelter<Cat>();
```

По този начин сме сигурни, че приютът **dogsShelter** винаги ще съдържа обекти от тип Dog, а променливата **catsShelter** ще оперира винаги с обекти от тип Cat.

2. **Описва и обяснява концепцията за наследяване на класове. Демонстрира наследяването на класове.**

- **Наследяването** е основен принцип от обектно-ориентираното програмиране.
- **Наследяването** позволява на един клас да "наследява" (**поведение** и **характеристики**) от друг, по-общ клас.
- Йерархийте от класове подобряват четимостта на кода и позволяват преизползване на функционалност.
- В C# наследяването се отбелязва чрез **:** оператора

```
class Person { … }

class Student : Person { … }
class Employee : Person { … }
```

Ключовата дума base:

- Тя указва да бъде използван базовият клас и позволява достъп до негови методи, конструктори и член-променливи.
- С **base**() можем да извикваме конструктор на базовия клас.
- С **base.method**(…) можем да извикваме метод на базовия клас, да му подаваме параметри и да използваме резултата от него.
- С **base.field** можем да вземем стойността на член-променлива на базовия клас или да й присвоим друга стойност.

Модификатори на достъп на членове на класа при наследяване

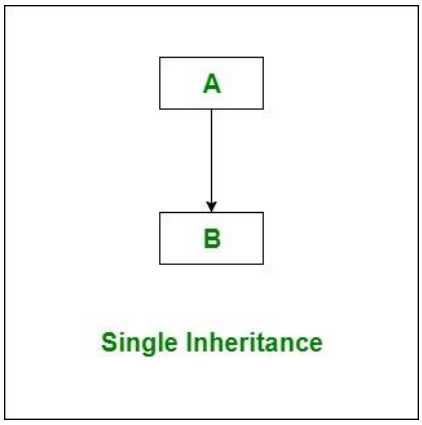
За членовете на един клас (методи, свойства, член-променливи) бяха разгледани **public**, **private**, **internal**. Има още два модификатора - **protected** и **internal protected**. Ето какво означават те:

- **protected** дефинира членове на класа, които са невидими за ползвателите на класа (тези, които го инстанцират и използват), но са видими за класовете наследници.

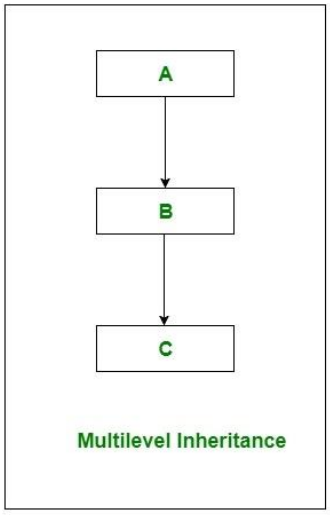
- **protected internal** дефинира членове на класа, които са едновременно internal, тоест видими за ползвателите в цялото асембли, но едновременно с това са и protected - невидими за ползвателите на класа (извън асемблито), но са видими за класовете наследници (дори и тези извън асемблито).

Видове наследяване:

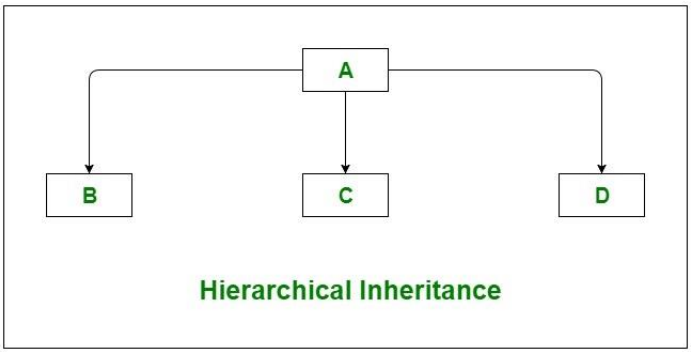
Единично наследяване: При единично наследяване подкласовете наследяват характеристиките на един суперклас.



Наследяване на много нива: При наследяване на много нива производният клас ще наследява базов клас и също ще действа като базов за друг клас.



Йерархично наследяване: При йерархично наследяване един клас служи като суперклас (базов клас) за повече от един подклас.



Множествено наследяване (чрез интерфейси): При множествено наследяване един клас може да има повече от един суперклас и да наследява характеристики от всички родителски класове. Имайте предвид, че C# не поддържа множествено наследяване с класове. **В C# можем да постигнем множествено наследяване само чрез интерфейси.**

- В C# не се поддържа множествено наследяване
- Поддържа се само имплементиране на множество интерфейси
- Наследяване на конструктори: Подкласът наследява всички членове (полета, методи) от своя суперклас. Конструкторите не са членове, така че не се наследяват от подкласове, но конструкторът на суперкласа може да бъде извикан от подкласа.
- Наследяване на private член: Подкласът не наследява частните членове на своя родителски клас. Ако обаче суперкласът има свойства (методи get и set) за достъп до неговите частни полета, тогава подкласът може да ги наследи.

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

И двете концепции се използват при полиморфизма:

Полиморфизъм:

- Третият основен принцип на ООП
- Многообразие
- Еднотипна работа с разнотипни данни
- Обобщаване на възможности, но с различна реализация
- Обединяване на сходни функционалности в нещо общо
- Гъвкав и лесен начин за поддръжка на кода

Има два вида полиморфизъм: статичен и динамичен.

- При статичния се прилага презаписването (overload) на членове, оператори.
- При динамичния се прилага предефинирането (override) на членове. Обвързан е с абстракцията.

Предефиниране (overload):

- Презаписване на членове“ (member overloading) - Повече от един член с едно и също наименование, но различна сигнатура.
- Членове, които могат да се презаписват: методи, конструктори, индексатори и оператори
- **Компилаторът** открива най-подходящият член, който да се използва, в зависимост от подаваните аргументи и съответните параметри.

Приложение на презаписвнето на членове

- Прилагане на сходна функционалност върху различни типове данни
- Прави кода по-четим, тъй като сходните операции имат еднакви наименования
- **Да не се злоупотребява:** прилага се, когато поведението е сходно, а типът на резултата – един и същи

- Видове презаписваеми оператори
  - Унарни (например: “+”, “-”, “~”, “!”, “++” и “--”)
  - Бинарни (например: „+“, „-“, „*“, „/“ и „%“)
  - Превръщане на данни

```
public struct Color
{
  public int R { get; set; }
  public int G { get; set; }
```

```
  public int B { get; set; }
  // презаписваме проверката дали е true
  public static bool operator true(Color c)
  {
    return c.R != 0 || c.G != 0 || c.B != 0;
  }
  // презаписваме проверката дали е false
  // забележка: винаги се презаписват в заедно
  public static bool operator false(Color c)
  {
    return c.R == 0 && c.G == 0 && c.B == 0;
  }
}
```

```
public struct Color
{
  public int R { get; set; }
  public int G { get; set; }
  public int B { get; set; }
  // презаписваме смесването на ДВА цвята
  public static Color operator +(
    Color l, Color r)
  {
    return new Color
    {
      R = l.R + r.R,
      G = l.G + r.G,
      B = l.B + r.B,
    };
  }
}
```

Презаписване (override):

В .NET наследените от базовия клас методи, които са декларирани като виртуални (virtual), могат да се пренаписват (override). Това означава да им се **подмени имплементацията, като оригиналният сорс код от базовия клас се игнорира, а на негово място се написва друг код**.
Метод, който може да се пренапише в клас наследник, се нарича виртуален метод (virtual method). Методите в .NET не са виртуални по подразбиране. Ако искаме да бъдат виртуални, ги маркираме с ключовата дума virtual. Тогава клас-наследник може да декларира и дефинира метод със същата сигнатура.
Виртуалните методи са важни за пренаписването на методи (method overriding), което е в сърцето на полиморфизма.

Декларирането на виртуален член

```
class Animal
{
  public virtual void MakeSound()
  {
    // не правим нищо, защото животните
    // не издават звук в общия случай
  }
}
```

- Деклариране на предефиниран член
  - Използва се ключовата дума **override** след модификатора за достъп
  - В родителски клас трябва да е деклариран виртуален член със същата сигнатура и видимост

Не е задължително всички виртуални членове да бъдат предефинирани в класовете-наследници.

```
class Dog : Animal
{
   public override void MakeSound()
   {
      Console.WriteLine("Woof woof!");
   }
}

class Cat : Animal
{
   public override void MakeSound()
   {
      Console.WriteLine("Meow!");
   }
}

class Fish : Animal
{
}
```

- Извикваният метод от йерархията се определя от истинския тип на обекта

```
// взимаме _някакво_ животно от зоопарка
Animal animal = zoo.GetAnimal();
// караме животното да издаде звук
animal.MakeSound();
// какъв звук ще издаде, зависи от вида
// животно
```

4. **Посочва принципите на обектно-ориентираното програмиране и дава примери за приложението им.**

Принципи на ООП:

- **Капсулация** - Тя се нарича още "скриване на информацията" (information hiding). Един обект трябва да предоставя на ползвателя си само необходимите средства за управление. Една Секретарка ползваща един Лаптоп знае само за екран, клавиатура и мишка, а всичко останало е скрито. Тя няма нужда да знае за вътрешността на Лаптопа, защото не ѝ е нужно и може да оплеска нещо. Тогава част от свойствата и методите остават скрити за нея.
- **Наследяване** - 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.
- **Абстракция** - Абстракцията означава да работим с нещо, което знаем как да използваме, но не знаем как работи вътрешно. Например имаме телевизор. Не е нужно да знаем как работи телевизорът отвътре, за да го ползваме. Нужно ни е само дистанционното и с малък брой бутони (интерфейс на дистанционното) можем да гледаме телевизия.
- **Полиморфизъм** - 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. Полиморфизмът дава възможността да третираме произволна голяма котка просто като голяма котка и да кажем "хвани жертвата си", без значение каква точно е голямата котка.

5. **Описва абстрактни класове и интерфейси . Различава абстрактен клас и интерфейс. Демонстрира и прави заключения и изводи за употребата на интерфейси и абстрактни класове.**

Интерфейс:

- 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**.
- Интерфейсът в .NET се дефинира с ключовата думичка **interface**.
- В него може да има само декларации на методи и свойства, както и статични променливи (за константи например).

- Всички членове на интерфейсите са “**public abstract**”.

Разлики между абстрактен клас и интерфейс:

Абстрактен клас	**Интерфейс**
1. Класът може да наследи само един абстрактен клас. 2. Абстрактните класове могат да предоставят целия код и/или само детайлите, които трябва да се презапишат. 3. Абстрактния клас може да съдържа модификатори за достъп 4. Ако множество имплементации са от сходен вид и имат общо поведение или статут, то абстрактния клас е по-добър избор. 5. Абстрактният клас може да притежава полета и константи 6. Ако добавим нов метод към абстрактен клас, то имаме опцията да създадем имплементация по подразбиране и така съществуващият код ще може да работи коректно.	1. Класът може да имплементира няколко интерфейса. 2. Интерфейсът не може да предоставя никакъв код, предоставя само описание. 3. Интерфейсите нямат модификатори за достъп. Всичко е публично по подразбиране. 4. Ако множество имплементации споделят само сигнатурата на методите и нищо друго, то тогава интерфейсът е по-добър избор. 5. Не поддържа полета 6. Ако добавим нов метод към интерйфес, то трябва да проследим всичките му имплементации и да дефинираме имплементация за новия метод.

6. **Дефинира понятието полиморфизъм и различава видовете полиморфизъм.**

- Полиморфизмът позволява **третирането на обекти от наследен клас като обекти от негов базов клас**.
- Полиморфизмът може много да напомня на абстракцията, но в програмирането се свързва най-вече с **пренаписването** (override) **на методи в наследените класове** с цел промяна на оригиналното им поведение, наследено от базовия клас.

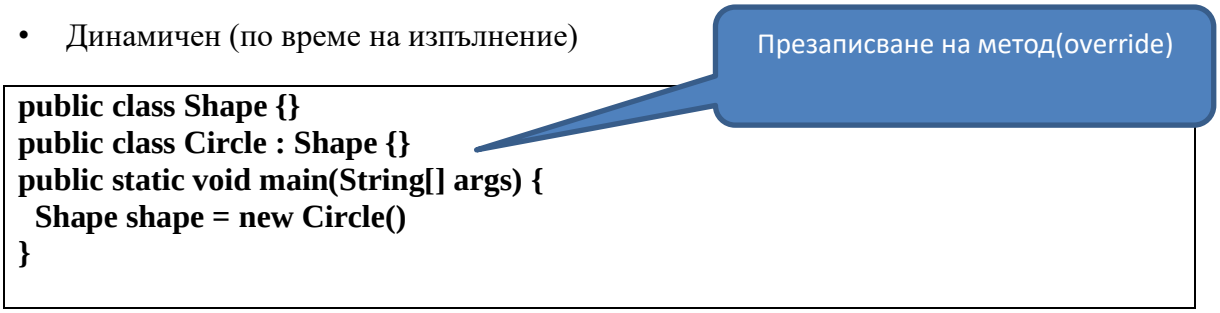
Видове полиморфизъм

- Статичен (по време на компилиране)

Презареждане на метод(overload)

```
public static void main(String[] args) {
  int Sum(int a, int b, int c)
  double Sum(Double a, Double b)
}
```

- Динамичен (по време на изпълнение)



Полиморфизъм може да се постигне чрез абстрактен клас или чрез интерфейс.

Възможността на обект да заема много форми:

```
public interface IAnimal {}
public abstract class Mammal {}
public class Person : Mammal, IAnimal {}
```

- Интерфейсите ни позволяват да постигнем полиморфистично поведение, което да не зависи от кода и да е лесно за промяна.

7. **Анализира фрагмент/и от код и идентифицира и поправя правилно грешките в написания програмен код, така че да реши поставената задача. Допълва кода, ако и когато това е необходимо.**

Задача 1 Създайте абстрактен клас **ColoredFigure**, който притежава:

- Поле color за отбелязване на цвета (като низ)
- Поле size за отбелязване на размер на фигурата
- Конструктор, който приема за параметри цвят и размер
- Метод Show(), който отпечатва цвета и размера на обекта.
- Абстрактен метод GetName(), който връща името на фигурата
- Абстрактен метод GetArea(), който връща лицето на фигурата

Създайте клас **Triangle**, който наследява ColoredFigure, като този клас има:

- Конструктор, който извиква конструктора на суперкласа
- Дефиниция за абстрактния метод GetName(), като този метод връща низа "Triangle".
- Дефиниция за абстрактния метод GetArea(), като този метод връща лицето на триъгълника, като

триъгълникът се приема за равностранен, със страна size. Използвайте формулата:

$$S = \frac{(size)^2 * \sqrt{3}}{4}$$

Създайте клас **Square**, който наследява ColoredFigure, като този клас има:

- Конструктор, който извиква конструктора на суперкласа

- Дефиниция за абстрактния метод GetName(), като този метод връща низа "Square".
- Дефиниция за абстрактния метод GetArea(), като този метод връща лицето на квадрата със страна

size.

Създайте клас **Circle**, който наследява ColoredFigure, като този клас има:

- Конструктор, който извиква конструктора на суперкласа
- Дефиниция за абстрактния метод GetName(), като този метод връща низа "Circle".
- Дефиниция за абстрактния метод GetArea(), като този метод връща лицето на кръга, с радиус size.

Вход:

На първия ред на входа има единствено цяло число N – брой заявки. От следващите N реда се подава
заявка в един от следните формати:

• Triangle <цвят> <размер>
• Circle <цвят> <размер>
• Square <цвят> <размер>

Изход:

За всяка заявка трябва да създаде обект от съответния клас, след което трябва да изпечатате 4 реда:
<име на фигурата>:
Color: <цвят>
Size: <размер>
Area: <лице>
Отпечатайте лицето с точно два знака след запетаята.

Вход	Изход
3 Circle blue 1 Square red 2 Triangle green 3	Circle: Color: blue Size: 1 Area: 3.14 Square: Color: red Size: 2 Area: 4.00 Triangle: Color: green Size: 3 Area: 7.79

ColoredFigure.cs

```
public abstract class ColoredFigure {
        protected string color;
        protected double size;

public ColoredFigure(string color, double size) {
      this.color = color;
      this.size = size;
 }
```

```
   public void Show() {
       Console.WriteLine(string.Format("Color: {0}", this.color));
       Console.WriteLine(string.Format("Size: {0}", this.size));
   }
   public abstract string GetName();
   public abstract double GetArea();
  }
```

Circle.cs

```
public class Circle: ColoredFigure {
  public Circle(string color, double size): base(color, size) { }

  public override double GetArea() {
    return Math.PI * Math.Pow(base.size, 2.0);
  }
public override string GetName() {
     return "Circle";
}
}
```

Triangle.cs

```
public class Triangle: ColoredFigure {
  public Triangle(string color, double size): base(color, size) { }

  public override double GetArea() {
      return (Math.Pow(base.size, 2.0) * Math.Sqrt(3)) / 4.0;
  }
public override string GetName() {
     return "Trangle";
}
}
```

Square.cs

```
public class Square: ColoredFigure {
  public Square(string color, double size): base(color, size) { }

  public override double GetArea() {
      return Math.Pow(base.size, 2.0);
  }
public override string GetName() {
     return "Square";
}
}
```

Program.cs

```
class Program {
 static void Main(string[] args) {
       int n = int.Parse(Console.ReadLine());
```

```
    for (int i = 0; i < n; i++) {
        var cmd = Console.ReadLine().Split();
        switch (cmd[0]) {
            case "Circle": {
                Circle circle = new Circle(cmd[1], double.Parse(cmd[2]));
                Console.WriteLine(string.Format("Name: {0}", circle.GetName()));
                circle.Show();
                Console.WriteLine(string.Format("Area: {0:f2}", circle.GetArea()));
                break; }
            case "Square": {
                Square square = new Square(cmd[1], double.Parse(cmd[2]));
                Console.WriteLine(string.Format("Name: {0}", square.GetName()));
                square.Show();
                Console.WriteLine(string.Format("Area: {0:f2}", square.GetArea()));
                break; }
            case "Triangle": {
                Triangle triangle = new Triangle(cmd[1], double.Parse(cmd[2]));
                Console.WriteLine(string.Format("Name: {0}", triangle.GetName()));
                triangle.Show();
                Console.WriteLine(string.Format("Area: {0:f2}", triangle.GetArea()));
                break;
    }
  }
 }
}
```