ПРЕДСТАВЯНЕ НА ОБЕКТИ ОТ РЕАЛНИЯ СВЯТ С ПРОГРАМЕН КОД

1. Дефинира: клас, конструктор, поле, свойство. Описва създаване на обекти от клас.

Клас (class) в ООП наричаме описание (спецификация) на даден клас обекти от реалността. Класът представлява шаблон, който описва видовете състояния и поведението на конкретните обекти (екземплярите), които биват създавани от този клас (шаблон).
В обектно-ориентираното програмиране, класът дефинира някои свойства, полета, събития, методи и т.н. Класът дефинира видовете данни и функционалността, която техните обекти ще имат.

Обект (object) наричаме екземпляр, създаден по дефиницията (описанието) на даден клас. Когато един обект е създаден по описанието, което един клас дефинира, казваме, че обектът е от тип "името на този клас".

Елементи на класа

- Декларация на класа (class declaration) – това е редът, на който декларираме името на класа. Например:

public class Dog

- Тяло на клас – по подобие на методите, класовете също имат част, която следва декларацията им, оградена с фигурни скоби – "{" и "}", между които се намира съдържанието на класа. Тя се нарича тяло на класа.

```
public class Dog
{
// Тяло на класа
}
```

- Конструктор (constructor) – това е псевдометод, който се използва за създаване на нови обекти. Конструкторът е специален тип метод, който ще бъде извикан автоматично, когато създадете екземпляр на клас. Конструкторът се дефинира чрез използване на модификатор за достъп и име на клас:
  <access-modifier> <class-name>(){ }.
- **Името на конструктора трябва да е същото като името на класа.**
- **Конструкторът може да бъде публичен, частен или защитен.**
- **Конструкторът не може да върне никаква стойност, така че не може да има тип на връщане.**
- **Един клас може да има множество конструктори с различни параметри, но може да има само един конструктор без параметри.**
- **Ако не е дефиниран конструктор, C# компилаторът ще го създаде вътрешно.**

public Dog() { }

- Полета (fields) – те са променливи, декларирани в класа (понякога в литературата се срещат като член-променливи). В тях се пазят данни, които отразяват състоянието на обекта и са нужни за работата на методите на класа. Стойността, която се пази в полетата, отразява конкретното състояние на дадения обект, но съществуват и такива полета, наречени статични, които са общи за всички обекти.

private string name;

- Свойства (properties) – така наричаме характеристиките на даден клас. Обикновено стойността на тези характеристики се пази в полета. Подобно на полетата, свойствата могат да бъдат притежавани само от конкретен обект или да са споделени между всички обекти от тип даден клас.

private string Name { get; set; }

В C# може да се създаде обект от клас с помощта на ключовата дума new и да се присвои този обект на променлива от тип клас. Например, следното създава обект от класа Student и го присвоява на променлива от типа Student.

```
Student mystudent = new Student();
```

Вече можете да получите достъп до публични членове на клас, като използвате нотацията object.MemberName.

```
Student mystudent = new Student();
mystudent.FirstName = "Steve";
mystudent.LastName = "Jobs";

mystudent.GetFullName();
```

2. Дефинира понятията функции/методи, тип и стойност на връщане, параметри и посочва видове параметри на функция/метод. Демонстрира дефинирането и употребата на функции/методи.

Метод (method) е съставна част от програмата, която решава даден проблем, може да приема параметри и да връща стойност. Понякога се нарича функция.
Чрез методите правим една програма добре структурирана и лесно четима, избягва се повторението на код, може да бъде извикан толкова пъти колкото ни е нужно.
Деклариране на метод наричаме регистрирането на метода в програмата, за да бъде разпознаван в останалата част от нея.
Имплементация (създаване) на метода е реалното написване на кода, който решава конкретната задача, която методът решава. Този код се съдържа в самия метод и реализира неговата логика.
Извикване е процесът на стартиране на изпълнението на вече декларирания и създаден метод, от друго място на програмата, където трябва да се реши проблемът, за който е създаден извикваният метод.
Задължителните елементи в декларацията на един метод.

- Тип на връщаната стойност.

- Име на метода.
- Списък с параметри - Декларира се между скобите (и), които изписваме след името му. Тук изброяваме **поредицата от параметри**, които метода ще използва. Може да присъства само един параметър, няколко такива или да е празен списък. Ако няма параметри, то ще запишем само скобите ().

След като сме декларирали метода, следва неговата **имплементация** (тяло). В тялото на метода описваме алгоритъма, по който той решава даден проблем.

Извикването на метод представлява стартирането на изпълнението на кода, който се намира в тялото на метода. Това става като изпишем името му, последвано от кръглите скоби () и знака ; за край на реда.

- Метода може да бъде извикан от главния метод Main().
- Метод може да бъде извикан и от тялото на друг метод, който не е главния метод на програмата ни.

Декларирането на метод представлява регистриране на метода в нашата програма.
То става чрез следната декларация:

```
[public] [static] <return_type> <method_name>([<param_list>])
```

- Тип на връщаната от метода стойност – **<return_type>**.
- Име на метода – **<method_name>**.
- Списък с параметри на метода – **([<param_list>])**– може да е празен списък или да съдържа поредица от декларации на параметри.

След като декларираме метода, следва да напишем неговата **имплементация**.

- Имплементацията (тялото) на метода се състои от кода, който ще бъде изпълнен при извикването на метода.
- Този код трябва да бъде поставен в тялото на метода и той реализира неговата логика.
- Тяло на метод наричаме програмният код, който се намира между **фигурните скоби** "{" и "} "

```
static <return_type> <method_name>(<parameters_list>)
{

}
```

Видове методи:

1.Метод, който не връща резултат (тип **void**).

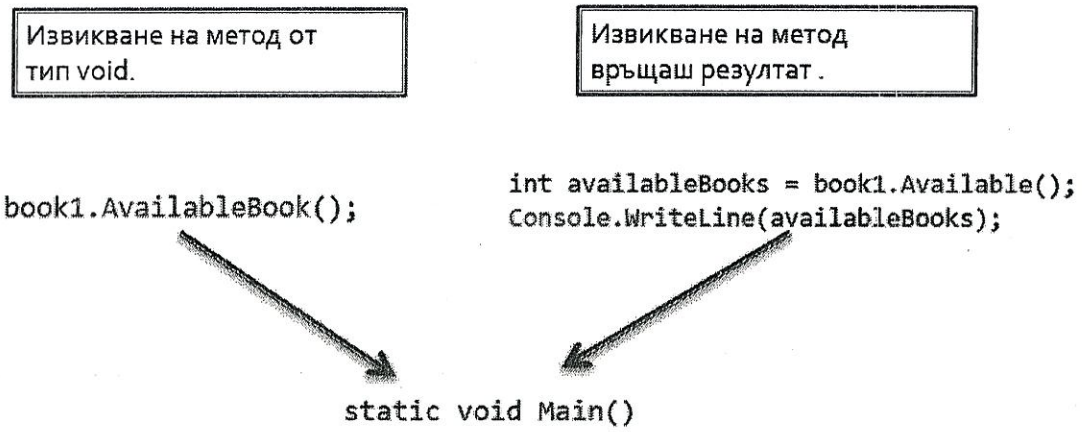
```
public void AvailableBook()
{
    Console.WriteLine(this.total - this.rentBook);
}
```

- Такъв метод може да няма в тялото си оператор **return.** Затова казваме, че не връща резултат.
- Ако има оператор return, то след него не се поставя никакъв израз, а само ";".

2.Метод, връщаш резултат (всеки един от скаларните типове **int, long, decimal, char**).

```
public int Available()
{
    return this.total - this.rentBook;
}
```

- Типа на метода може да бъде всеки един от скаларните типове в езика C#.
- В тялото на метода се използва оператор **return <израз>;**



Методи с един или няколко параметъра (може да бъде както от тип void, така и от скаларните типове).

3. Прави заключения и изводи за употребата на ключовата дума this и дава пример за ситуации, които показват необходимостта от употребата на ключовата дума this.

Ключовата дума this в C# дава достъп до референцията към текущия обект, когато се използва от метод в даден клас. Това е обектът, чийто метод или конструктор бива извикван. Можем да я разглеждаме като указател (референция), дадена ни априори от създателите на езика, с която да достъпваме елементите (полета, методи, конструктори) на собствения ни клас:

```
this.myField; // достъп до поле
this.DoMyMethod(); // достъп до метод
this(3, 4); // достъп до конструктор с два параметра
```

Следват различните начини за използване на ключова дума 'this' в C#:
- Използване на ключова дума „this“ за препращане към текущи членове на екземпляр на клас.
- Използване на this() за извикване на конструктора в същия клас.

```
public Geeks() : this("geeks")
{
    Console.WriteLine("Non-Parameter Constructor Called");
}
```

-Използване на ключова дума „this“ за извикване на метод на текущия клас.

```
void display()
{

    this.show();

    Console.WriteLine("Inside display function");
}

void show()
{
    Console.WriteLine("Inside show function");
}
```

4. Обяснява енкапсулирането на данни в класовете. Дава пример за употребата на методите за достъп и промяна на енкапсулираните данни.

Капсулация (encapsulation) наричаме скриването на физическото представяне на данните в един клас, така че, ако в последствие променим това представяне, това да не рефлектира върху останалите класове, които използват този клас. Капсулирането може да бъде постигнато чрез: **Деклариране на всички променливи в класа като частни и използване на C# Properties в класа за задаване и получаване на стойностите на променливите.**
Чрез синтаксиса на C# това се реализира като декларираме полета (физическото представяне на данните) с възможно най-ограничено ниво на видимост (най-често с модификатор private) и декларираме достъпът до тези полета (четене и модифициране) да може да се осъществява единствено чрез специални методи за достъп (accessor methods).

Пример:

```
using System;

public class DemoEncap {

   private String studentName;
   private int studentAge;

   public String Name
```

```
    {

        get
        {
            return studentName;
        }

        set
        {
            studentName = value;
        }

    }

    public int Age
    {

        get
        {
            return studentAge;
        }

        set
        {
            studentAge = value;
        }

    }

}

class GFG {

    static public void Main()
    {

        DemoEncap obj = new DemoEncap();

        obj.Name = "Ankita";

        obj.Age = 21;

        Console.WriteLine("Name: " + obj.Name);
        Console.WriteLine("Age: " + obj.Age);
    }
}
```

Обяснение: В горната програма класът DemoEncap е капсулиран, тъй като променливите са декларирани като частни. За достъп до тези частни променливи използваме инструментите за достъп Name и Age, които съдържат метода get и set за извличане и задаване на стойностите на частните полета. Аксесорите се определят като публични, така че да имат достъп в друг клас.

Предимства на капсулирането:

- **Скриване на данни**: Капсулацията скрива имплементацията (реализацията на обекта - неговите компоненти – полета, свойства, методи). Потребителят няма да има представа за вътрешната реализация на класа. Потребителят няма да види как класът съхранява стойности в променливите. Той знае само, че ние предаваме стойностите на аксесори и променливите се инициализират към тази стойност.
- **Тестването на код е лесно:** Капсулираният код е лесен за тестване за тестване. Капсулацията намалява сложността.
- **Повишена гъвкавост**: Можем да направим променливите на класа само за четене или само за запис в зависимост от нашите изисквания. Ако искаме да направим променливите само за четене, тогава трябва да използваме само Get Accessor в кода. Ако искаме да направим променливите само за запис, тогава трябва да използваме само Set Accessor.
- **Възможност за повторна употреба**: Капсулирането също така подобрява възможността за повторна употреба и лесно се променя с нови изисквания. Осигурява **структурните промени** да останат **локални.**

5. Дефинира и описва статичен клас и статични членове на класа. Дава пример за употребата на статични членове в клас.

- Когато един елемент на класа е деклариран с модификатор **static**, го наричаме статичен.
- В C# като **статични** могат да бъдат декларирани **полетата**, **методите**, **свойствата**, **конструкторите** и **класовете**.

Статичен член (static member) на класа наричаме всяко поле, свойство, метод или друг член, който има модификатор **static** в декларацията си.

- Това означава, че полета, методи и свойства маркирани като статични, **принадлежат на самия клас**, а не на някой конкретен обект от дадения клас.
- Следователно, когато маркираме поле, метод или свойство като статични, можем да ги използваме, без да създаваме нито един обект от дадения клас.

Статичните полета декларираме по същия начин, както се декларира поле на клас, като след модификатора за достъп (ако има такъв), добавяме ключовата дума **static**:

```
[<access_modifier>] static <field_type> <field_name>
```


15

```
public class Dog
{
	// Static (class) variable
	static int dogCount;

	// Instance variables
	private string name;
	private int age;
}
```

Достъп до статични полета:

За разлика от обикновените (нестатични) полета на класа, статичните полета, бидейки асоциирани с класа, а не с конкретен обект, могат да бъдат достъпвани от външен клас като към името на класа, чрез точкова нотация, достъпим името на съответното статично поле:

```
public static void Main()
{
	// Access to the static variable through class name
	Console.WriteLine("Dog count is now " + Dog.dogCount);
}
```

Статичните полета в класа

- Принадлежат на самия клас
- Имат една и съща стойност за всеки обект
- Могат да бъдат достъпени и само чрез класа - без създаване на обект от този клас

Статични методи:

- По подобие на статичните полета, когато искаме един метод да е асоцииран само с класа, но не и с конкретен обект от класа, тогава го декларираме като статичен.
- Синтактично да декларираме статичен метод означава в декларацията на метода, да добавим ключовата дума **static**:

```
[<access_modifier>] static <return_type> <method_name>()
```

Статични свойства:

- Статични свойства носят същите характеристики като свойствата, свързани с конкретен обект от даден клас, но с тази разлика, че статичните свойства се отнасят за класа.

- Да превърнем едно обикновено свойство в статично, е да добавим ключовата дума static при декларацията му.
- Статичните свойства се декларират по следния начин:

```
[<modifiers>] static <property_type> <property_name>
{

}
```

6. Дефинира по-сложни класове. Различава модификатори за достъп. Разработва по-сложни класове с правилна енкапсулация на членовете на класа.

Модификаторите определят степента на капсулация на данните

- Private - за полета
- Protected – за наследници (подкласове)
- Internal – за класове от същия проект (namespace)
- Public – за класове и интерфейси в целия .Net

Private Модификатор за достъп

- Основен начин за капсулиране на обект и скриване на данни от външния свят

```
private string name;
Person (string name)
{
 this.name = name;
}
```

- Класовете и интерфейсите не могат да са private. Идеята за интерфейс е да се даде възможност за връзка с „външния свят“ – т.е. – трябва да са достъпни
- Могат да бъдат достъпни само в декларацията на класа

Protected Модификатор за достъп

- Могат да бъдат достъпни само от подкласове

```
class Person
{
 protected string FullName { get; set; }
}
```

- Модификаторът за достъп Protected не може да бъде приложен за класове и интерфейси
- Предотвратява външни класове да се опитват да го използват

Internal модификатор за достъп

- Internal е модификатор по подразбиране в C#

```
class Person
{
  string Name { get; set; }
  internal int Age { get; set; }
}
```

- Дава достъп на всеки друг клас в същия проект

```
Team rm = new Team("Real");
rm.Name("Real Madrid");
```

Public модификатор за достъп

- Клас, метод, конструктор, деклариран в public клас може да бъде достъпен от всеки клас, принадлежащ на .NET света

```
public class Person
{
  public string Name { get; set; }
  public int Age { get; set; }
}
```

- Употребата се налага ако се опитваме да достъпим public клас в друг namespace
- Методът main() в приложението трябва да е public
- Интерфейсите са public. Тъй като смисълът им е да дават връзка с външния свят

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.