

## 8. Óra

### Miről lesz szó

Virtuális (**virtual**) metódusok és ezek felüldefiniálása (**override**)

```
class Pet {  
    public virtual void MakeSound() {  
        Console.WriteLine("valami hang");  
    }  
}  
  
class Cat : Pet {  
    public override void MakeSound() {  
        Console.WriteLine("miau.");  
    }  
}
```

Futás idejű polimorfizmus (**Runtime polymorphism**)

```
public class Shape {  
    public int X { get; private set; }  
    public int Y { get; private set; }  
    public int Height { get; set; }  
    public int Width { get; set; }  
  
    public virtual void Draw() {  
        Console.WriteLine("Performing base class drawing tasks");  
    }  
}  
  
public class Circle : Shape { // Shape base classa a Circle-nek, max 1 base classa lehet!  
    public override void Draw() {  
        Console.WriteLine("Drawing a circle");  
        base.Draw(); // A base classra utal (Shape jelen esetben)  
    }  
}  
  
public class Rectangle : Shape { // Shape base classa a Rectangle-nek, max 1 base classa lehet!  
    public override void Draw() {  
        Console.WriteLine("Drawing a rectangle");  
        base.Draw(); // A base classra utal (Shape jelen esetben)  
    }  
}  
  
public class Triangle : Shape { // Shape base classa a Triangle-nek, max 1 base classa lehet!  
    public override void Draw() {  
        Console.WriteLine("Drawing a triangle");  
        base.Draw(); // A base classra utal (Shape jelen esetben)  
    }  
}
```

```
}  
}
```

```
List<Shape> shapes = new List<Shape> {  
    new Rectangle(),  
    new Triangle(),  
    new Circle()  
};  
  
// Futás időben először Rectangle, majd Triangle, majd Circle osztály specifikus Draw()  
metódusát hívja meg  
foreach (Shape shape in shapes) {  
    shape.Draw();  
}  
  
/* Output:  
    Drawing a rectangle  
    Performing base class drawing tasks  
    Drawing a triangle  
    Performing base class drawing tasks  
    Drawing a circle  
    Performing base class drawing tasks  
*/
```

```
public abstract class Person {  
    private readonly string _name;  
    private readonly string _taj;  
  
    public string Name { get { return _name; } }  
    public string Taj { get { return _taj; } }  
  
    public Person(string name, string taj) {  
        _name = name;  
        _taj = taj;  
    }  
}  
  
public class Employee : Person {  
    public Employee(string name, string taj) : base(name, taj) {  
        // helyes base class konstruktor hívás  
    }  
}
```

## abstract vs virtual

### abstract

- **Kötelező** overrideolni
- Lehet:
  - Class
    - Ilyenkor nem lehet instanceot létrehozni belőle, csak az osztályokból amik ebből származnak (ha nem abstractok)
  - Method

- Property
- Indexer

## virtual

- **Nem kötelező**, de lehet overrideolni
- Lehet:
  - Method
  - Property
  - Indexer

 `Object.ToString()` abstract vagy virtual?

## ?? null-összevonó operátor (**null-coalescing operator**)

Ha az első érték `null`, akkor a másodikat adja vissza.

```
string? favCatName = null; // null

string catName1 = favCatName ?? "Garfield"; // Garfield
string catName2 = "Tom" ?? "Garfield" // Tom
```

## ??= (**null-coalescing assignment operator**)

Ha a változó `null`, akkor beállítja az értékét.

```
int? favNumber = null; // null

favNumber ??= 42; // 42
favNumber ??= 55; // 42
```



```
favNumber = favNumber ?? 42;
```

És akár:

```
int? favNumber = null; // null

int number = (favNumber ??= 42);
// 42 lesz egyaránt a number és a favNumber változó, mert az egész assignment kifejezés
értéke lesz az újonnan beállított érték
```

## Egyke (**Singleton**) tervminta

```
class Potato {
    // EGY statikus instance létezik belőle
    private static Potato? _instance;

    // Statikus instance ha nincs létrehozva, majd visszatér vele
    public static Potato Instance => _instance ??= new Potato();
}
```

```
// nem lehet példányosítani mert privát a konstruktora
private Potato() { }
}
```

Egy kicsit máshogy lekódolva, goofy operátorok nélkül:

```
class Potato {
    private static Potato? _instance;

    public static Potato Instance {
        get {
            if (_instance == null) {
                _instance = new Potato();
            }
            return _instance;
        }
    }

    private Potato() { }
}
```

## Runtime típusellenőrzés, cast-olás

```
class Animal { }
class Dog : Animal {
    public void Woof();
}
class Cat : Animal {
    public void Meow();
}

Dog doggo = new Dog();
Cat cat = new Cat();

/*
  Animal
  /    \
Dog     Cat
*/
```

### typeof()

*Típusnevet* (nem objektumot!) kér ami **fordítási időben** eldönthető, visszatér a típussal

```
typeof(Cat); // namespace.Cat
```

### GetType()

*Objektumot* kér, visszaadja a **runtime** típusát

```
doggo.GetType(); // namespace.Dog

Animal garfield = new Cat();
garfield.GetType(); // namespace.Cat
```

Ellenőrzés ezekkel: (pontos típus ellenőrzés)

```
doggo.GetType() == typeof(Dog) // true
```

## is

Igazzal tér vissza ha **típusnak** az öröklődési fájában van az **objektum**

```
doggo is Dog    // true
doggo is Animal // true
doggo is Cat    // false

Animal animal = new Cat();
// Kombinált check és cast
if (animal is Cat castedCat) { // lokális változóba castolás egyből ha sikeres
    // true és casting sikeres
    castedCat.Meow();
} else {
    // false és casting sikertelen
}
```

## (Class)object (unsafe cast)

Ha nem sikerül castolni, akkor `InvalidCastException`-t dob

```
try {
    Cat castedCat = (Cat)animal;
    // Cast sikeres
} catch (InvalidCastException) {
    // Cast sikertelen
}
```

## as (safe cast)

Ha nem sikerül castolni, akkor `null`-al tér vissza

- csak referencia típusokra
- nincs type checking, ezt manuálisan kell

```
Cat castedCat = animal as Cat;
if (castedCat != null) {
    // Cast sikeres
} else {
    // Cast sikertelen
}
```

# Feladatok

## Kert



