7. Dědičnost

- object → všechny třídy automaticky dědí od tř. object (GetType(), ToString(), Equals(), GetHashCode())
- třída dědí od jiné třídy

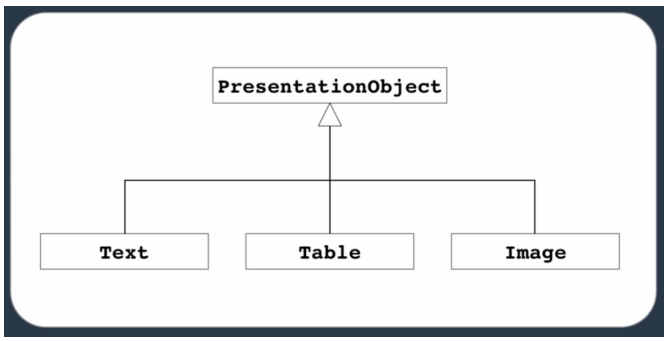
↳ auto, stejně jako motorka, jsou vozidlo (funkce Move)

Př. Powerpoint

↳ máme obrázek, tvar, text a tabulku

- každý z nich je reprezentovaný třídou
- všechny tyto objekty lze v Powerpointu:
  - přesunout
  - duplikovat
  - změnit velikost
  - smazat, kopírovat
  - .....
  - pozice na snímku
  - velikost
  - pořadí vrstvy
  - .....

Všechny tyto vlastnosti a metody můžeme zaobalit jako PresentationObject ⇒ Nepaluje se kód!!!



Obrázek, text a další budou tuto implementaci dědit od svého rodiče

↳ nemusíme stejný kód psát znovu

Příklad dědičnosti

Třída Zaměstnanec

```
public class Zamestnanec
{
    public string Jmeno;
    public string Pozice;

    Počet odkazů: 3
    public Zamestnanec(string jmeno, string pozice)
    {
        Jmeno = jmeno;
        Pozice = pozice;
    }

    // Virtuální metoda, kterou lze přepsat v potomcích
    Počet odkazů: 6
    public virtual void Pracuj()
    {
        Console.WriteLine($"{Jmeno} pracuje na obecných úkolech.");
    }

    // Metoda není virtual, takže ji potomoci pouze zdědí, ale nemůžou ji změnit
    Počet odkazů: 3
    public void Info()
    {
        Console.WriteLine($"Jméno: {Jmeno}, Pozice: {Pozice}");
    }
}
```

virtual – umožňuje potomky přepsat funkci

Programátor dědí od zaměstnance

```
public class Programator : Zamestnanec
{
    public string ProgramovaciJazyk;

    Počet odkazů: 1
    public Programator(string jmeno, string jazyk) : base(jmeno, "Programátor") // : base(jmeno, "Programátor") volá konstruktor předka
    {
        ProgramovaciJazyk = jazyk;
    }

    // Přepis metody Pracuj
    Počet odkazů: 4
    public override void Pracuj()
    {
        Console.WriteLine($"{Jmeno} píše kód v jazyce {ProgramovaciJazyk}.");
    }
}
```

:base(...) volá konstruktor předka – musí splnit

override přepisuje funkci Pracuj – programátor má tak jiné chování

Programátor má také vlastnost navíc – jazyk

Manažer dědí od zaměstnance

```
public class Manager : Zamestnanec
{
    public int PocetPodrizenych;

    Počet odkazů: 1
    public Manager(string jmeno, int pocetPodrizenych) : base(jmeno, "Manažer") // : base(jmeno, "Manažer") volá konstruktor předka
    {
        PocetPodrizenych = pocetPodrizenych;
    }

    // Přepis metody Pracuj a použití base pro volání původní metody
    Počet odkazů: 4
    public override void Pracuj()
    {
        base.Pracuj(); // zavolá se metoda Pracuj z rodiče
        Console.WriteLine($"{Jmeno} také řídí tým o {PocetPodrizenych} lidech.");
    }
}
```

:base(...) volá konstruktor předka - musí splnit

override přepisuje funkci Pracuj

↳ volá implementaci předka

↳ přidá něco navíc

Manažer má také vlastnost navíc – PocetPodrizenych

Použití

```
public class Program
{
    Počet odkazů: 0
    static void Main(string[] args)
    {
        Zamestnanec zamestnanec = new Zamestnanec("Jan Novák", "Asistent");
        Programator programator = new Programator("Petr Dvořák", "C#");
        Manager manager = new Manager("Eva Svobodová", 5);

        // Ukázka obecných informací
        zamestnanec.Info();
        programator.Info();
        manager.Info();

        Console.WriteLine();

        // Ukázka přepisu a volání metod
        zamestnanec.Pracuj();
        programator.Pracuj();
        manager.Pracuj();
    }
}
```

```
Vybrat Konzola ladění sady Microsoft Visual Studio
Jméno: Jan Novák, Pozice: Asistent
Jméno: Petr Dvořák, Pozice: Programátor
Jméno: Eva Svobodová, Pozice: Manažer

Jan Novák pracuje na obecných úkolech.
Petr Dvořák píše kód v jazyce C#.
Eva Svobodová pracuje na obecných úkolech.
Eva Svobodová také řídí tým o 5 lidech.
```

Dědičnost (Is-a vztah)

↳ jeden objekt specializovaná verze jiného objektu (rodiče)

Př. Pes je Zvíře, Kruh je Tvar

Kompozice (Has-a)

↳ objekt má jiný objekt jako vlastnost

Př. Auto má motor, Student má adresu

Dedičnost vs. Kompozice

- **dědičnost** použijeme, když chceme sdílet vlastnosti nebo chování mezi třídami a zachovat hierarchii
- **kompozice** použijeme, když chceme vytvořit flexibilnější a znovupoužitelné komponenty bez pevného vázání na hierarchii tříd

Příklad

Dědičnost

```
Počet odkazů: 2
public class Savec
{
    Počet odkazů: 0
    public Savec() { }

    Počet odkazů: 1
    public void Dychat()
    {
        Console.WriteLine("Dýchám plícemi");
    }
}
Počet odkazů: 2
public class Pes : Savec
{
    Počet odkazů: 1
    public void Stekat()
    {
        Console.WriteLine("haf haf");
    }
}
Počet odkazů: 0
public class Program
{
    Počet odkazů: 0
    static void Main(string[] args)
    {
        Pes pes = new Pes();
        pes.Dychat();
        pes.Stekat();
    }
}
```

- sdílí vlastnosti/metody

Kompozice

```
Počet odkazů: 4
public class Motor
{
    Počet odkazů: 1
    public void Startuj()
    {
        Console.WriteLine("Motor nastartován.");
    }
}

Počet odkazů: 3
public class Auto
{
    private Motor _motor;

    Počet odkazů: 1
    public Auto(Motor motor)
    {
        _motor = motor;
    }

    Počet odkazů: 1
    public void StartujAuto()
    {
        _motor.Startuj();
    }
}
Počet odkazů: 0
public class Program
{
    Počet odkazů: 0
    static void Main(string[] args)
    {
        Motor motor = new Motor();
        Auto auto = new Auto(motor);
        auto.StartujAuto(); // Auto používá Motor
    }
}
```

- využíváme jiné třídu
- flexibilní, nevázané

Modifikátory přístupu

- zamezit přístupu ostatních tříd/objektů

- **public**
  - dostupné, viditelné
- **private**
  - nedostupné, viditelné pouze uvnitř třídy
- **static**
  - dostupné, viditelné, lze použít i bez vytvoření objektu

  např. `Console.WriteLine();`
- **protected**
  - něco jako private

  ↳ lze zavolat pouze ze třídy nebo tříd, které z dané třídy dědí
  - ruší enkapsulaci
- **radši se vyhnout**

Upcasting/Downcasting

```
Circle circle = new Circle();
Shape shape = circle;
```

Upcasting

```
Circle anotherCircle = (Circle)shape;
```

Downcasting

Abstraktní třída

- slouží pouze pro definici
  - ↳ nemá implementaci
- vše co se dědí z ABS třídy se musí override
  - ↳ nemá implementaci

```
public abstract class Shape
{
    Počet odkazů: 2
    public abstract void Draw();
}

Počet odkazů: 1
public class Circle : Shape
{
    Počet odkazů: 2
    public override void Draw()
    {
        // vykreslení kruhu
    }
}
Počet odkazů: 0
public class Program
{
    Počet odkazů: 0
    static void Main(string[] args)
    {
        Shape shape = new Shape(); // nelze, protože z abstraktní třídy nejde vytvořit objekt
        Shape circle = new Circle(); // lze, protože se jedná o instanci třídy Circle
        circle.Draw();
    }
}
```

- nelze vytvořit objekt/instanci třídy Shape
  - ↳ nemá implementaci

použití:

Víme, že každý tvar se vykresluje jinak, proto každý tvar bude vždycky přepisovat funkci draw

⇒ Shape tak nepotřebuje mít implementaci

Abstraktní třídy se dolík nepoužívají

↳ lepší řešení → interface/rozhraní