Prohlížeč obchodní sítě

Program.cs

```
Console.CursorVisible = false;

Cursor mainCursor = new Cursor(); //Kurzor pro hlavní interface
Cursor listCursor = new Cursor(); //Kurzor pro prohlížení seznamu
Cursor warningCursor = new Cursor(); //Kurzor pro ukládací menu
Tree t = new Tree();
Render r = new Render();
List l = new List();
```

Na začátku vypnu viditelnost kurzoru, aby mi tam neblikal. Dále zde vytvářím instance jednotlivých tříd. Jsou zde 3 kurzory, protože každá nabídka má svůj. Např. *mainCursor* je kurzor pro hlavní nabídku a *listCursor* je pro procházení seznamu.

```
while (true)
{
    ConsoleKeyInfo keyInfo = Console.ReadKey(true);
}
```

Déle je zde while, který vždy zjistí, jakou klávesu uživatel zmáčknul a dále podle hromady podmínek udělá danou akci.

```
else if (keyInfo.Key == ConsoleKey.RightArrow)
{
    if (mainCursor.X == 0)
    {
        mainCursor.X++;
    }
}
```

Zde je ukázka toho, jak program pracuje se vstupem. Pokud uživatel zmáčkne pravou šipku, tak program zjistí, zda je možné pohnout kurzor a pokud ano, tak ho pohne.

Render.cs

Render obsahuje 3 hlavní metody. **RenderMainInterface**, **RenderList**, **RenderWarning**. Každý vykresluje jinou nabídku, ovšem všechny fungujou na principu, že je zde hromada podmínek.

```
if (c.X == 0 && c.Y == 0)
{
    Console.ForegroundColor = ConsoleColor.Green;
    Console.BackgroundColor = ConsoleColor.Yellow;
    Console.Write("ANO");
    Console.ResetColor();
}
else
{
    Console.ForegroundColor = ConsoleColor.Green;
    Console.Write("ANO");
    Console.ResetColor();
}
```

Zde je ukázka z třídy **RenderWarning**, kde nejdřív zjistím, zda je kurzor na „**ANO**“ a pokud ano, tak nastaví jeho barvu na pozadí na žlutou a pokud ne, tak to jenom vykreslí bez barvy na pozadí.

```
static int GetTotalSalesRecursive(Salesman node)
{
    int sum = node.Sales;
    foreach (var sub in node.Subordinates)
        sum += GetTotalSalesRecursive(sub);
    return sum;
}
```

V této class je také metoda **GetTotalSalesRecursive**, která pomocí rekurze sečte celkové prodeje sítě.

Salesman.cs

```
public override bool Equals(object obj)
{
    if (obj is Salesman other)
    {
        return ID == other.ID;
    }
    return false;
}

public override int GetHashCode()
{
    return ID.GetHashCode();
}
```

Tohle jsou jediné věci, které jsem přidal do této třídy. Dlouho jsem nechápal, proč mi nefunguje **List.Contains()**. Když jsem chtěl přidat obchodníka na seznam, tak jsem to vždy pomocí této metody zkontroloval, zda už na seznamu není, ovšem nikdy to nefungovalo správně. Stávalo se tedy, že na jednom listu bylo více stejných obchodníků. Pak jsem na https://stackoverflow.com/ zjistil, že pokud chci porovnávat objekty, potřebuji přepsat metodu **Equals()** a také přepsat metodu **GetHashCode()**. Tohle tedy porovnává podle unikátního ID a díky tomu už metoda **List.Contains()**, fungovala správně.

Cursor.cs

```
internal class Cursor
{
    public int X { get; set; } = 0;
    public int Y { get; set; } = 0;
}
```

Tato class je vpodstatě jenom obálkou na **X** a **Y** souřadnice.

Tree.cs

```
internal class Tree
{
    public Salesman Root { get; set; } = Salesman.DeserializeTree(File.ReadAllText("smalltree.json"));
    public Salesman Node { get; set; } = Salesman.DeserializeTree(File.ReadAllText("smalltree.json"));
    public Stack<Salesman>? Stack { get; set; } = new Stack<Salesman>();
    public List<Salesman>? List { get; set; } = new List<Salesman>();
    public string Path { get; set; } = "";
}
```

Root se v průběhu programu nemění a je to celý strom, zatímco **Node** se v průběhu mění a je to pouze část stromu, kterou jsem právě vybral. **Stack** si pamatuje nadřízené. Takže když kliknu na podřízeného, tak se do **Stacku** přidá aktuální node a pokud budu chtít jít zpět nahoru, tak stačí pouze **Stack.Pop()** a mám ndřízeného. Dále je zde **List**, který uchovává aktuální seznam obchodníků. A **Path** je cesta k souboru.

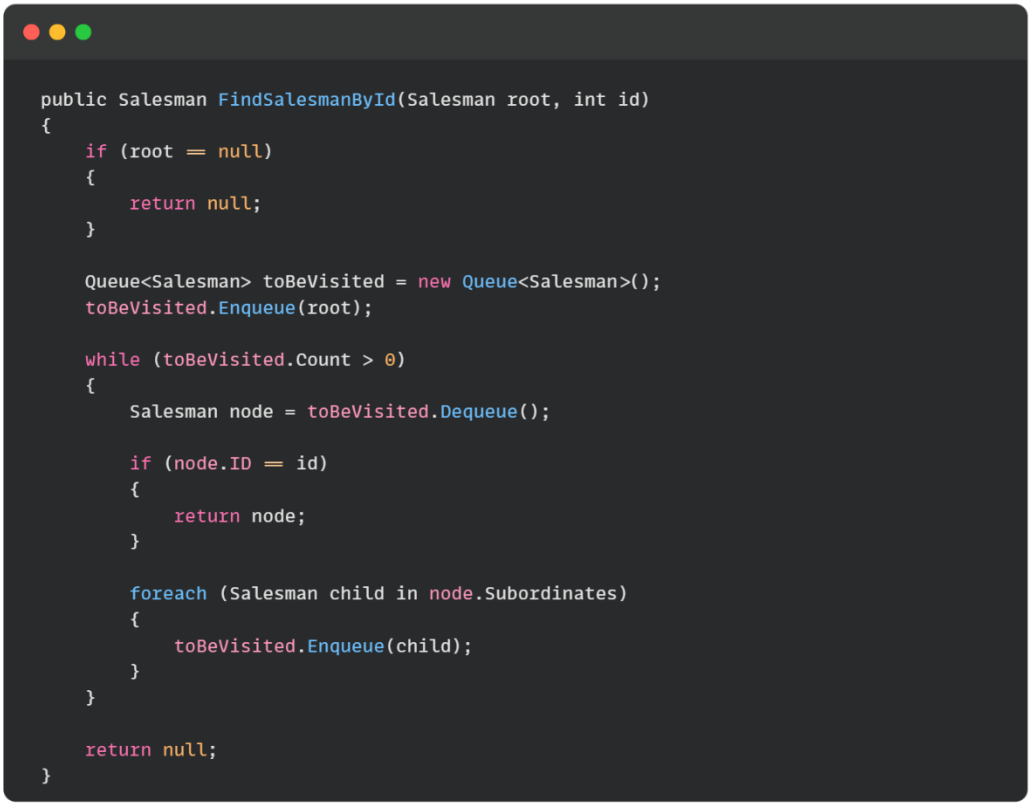
List.cs

```
public bool RemoveSalesmanFromList(Salesman salesman, List<Salesman> list)
{
    if (list.Contains(salesman))
    {
        list.Remove(salesman);
        return true;
    }
    else
    {
        return false;
    }
}
```

Tato class obsahuje věci, které se týkají seznamu. Zde je ukázka metody **RemoveSalesmanFromList**, která zkontroluje, zda na seznamu je daný obchodník a pokud ano, tak ho odebere a vrátí true, což znamená, že se operace povedla, jinak vrátí false.

```
public bool CreateList(Tree t)
{

    Console.Write($"Zadej název souboru: ");
    string? input = Console.ReadLine();
    Console.Clear();

    try
    {
        using (FileStream fs = File.Create($"{input}.txt"))
        {
            fs.Close();
        }

        t.Path = input + ".txt";
        t.List.Clear();

        return true;

    }
    catch
    {
        return false;

    }
}
```

Pak chci ještě vysvětlit, jak funguje metoda **CreateList**, která vytvoří nový textový soubor se zadaným názvem. Nejprve jsem používal klasický **File.Create()**, ale kvůli tomu se tam vyskytl hodně zvláštní bug. Vždy když jsem vytvořil soubor a chtěl jsem do něj něco uložit, tak program spadl s chybovou hláškou, že soubor je otevřený jinde a nejde do něj tedy zapisovat. Pak jsem pomocí **ChatGPT** zjistil, že **File.Create()** nechává soubor otevřený, proto do něj nelze nic ukládat. A poradilo mi použít **FileStream** podle jejich dokumentace je to lepší nástroj na práci se soubory a podle dokumentace jsem udělal tuto metodu. *(Omlouvám se, že jsem použil ChatGPT, ale i po debugování jsem nevěděl, proč soubor zůstává otevřený, tak jsem tam zadal tento problém a našla mi toto řešení. Logiku ostatních metod jsem vždy vymýšlel sám a celkově jsem ji použil 3x v podobných situacích, kdy jsem byl bezradný. Vždy jsem si prošel dokumentaci, abych tomu rozuměl a nejenom kopíroval kód.)*



```
public Salesman FindSalesmanById(Salesman root, int id)
{
    if (root == null)
    {
        return null;
    }

    Queue<Salesman> toBeVisited = new Queue<Salesman>();
    toBeVisited.Enqueue(root);

    while (toBeVisited.Count > 0)
    {
        Salesman node = toBeVisited.Dequeue();

        if (node.ID == id)
        {
            return node;
        }

        foreach (Salesman child in node.Subordinates)
        {
            toBeVisited.Enqueue(child);
        }
    }

    return null;
}
```


Zde je metoda **FindSalesmanById**, kterou vždy používám při načítání souboru. Strukturu souborů jsem nechal základní, takže na každém řádku je **ID** obchodníka. Při načítání souboru používám tuto metodu, která dostane **ID** a v **root** najde, jaký obchodních má toto **ID**, pak ho jenom vrátí.