

# Objektno-orientisano programiranje

## Nasleđivanje

Nasleđivanje predstavlja bitan deo bilo kog objektno-orientisanog jezika. Klase mogu da nasleđuju jedna drugu, odnosno klasa koja nasleđuje dobija sve osobine nasleđene klase, poznate i kao bazna klasa. Za početak dat je primer klase **Person**, sa dva svojstva (**FirstName** i **LastName**), kao i jednom metodom (**GetFullName**). Ova klasa je nasleđena od strane **Employee** klase, što se označava dvotačkom: **Employee : Person**.

```
public class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public string GetFullName()
    {
        return FirstName + " " + LastName;
    }
}

public class Employee : Person
{
    public decimal Salary { get; set; }
}
```

Kako sve klase nasleđuju klasu **Object**, klasa **Employee** zapravo nasleđuje od klase **Person**, dok klasa **Person** nasleđuje od klase **Object**, iako to nije eksplicitno navedeno. U narednom primeru dat je primer upotrebe klase **Employee**.

```
Employee employee = new Employee();
employee.FirstName = "Sander";
employee.LastName = "Rossel";
string fullName = employee.GetFullName();
employee.Salary = 1000000; // I wish! :-)
```

Kao što se može primetiti, sve što važi za klasu **Person**, važi i za klasu **Employee**. U ovom slučaju **Person** može biti nazvano baznom klasom ili superklasom, a **Employee** izvedenom klasom ili podklasom. Takođe, može se reći i da **Employee** proširuje **Person**.

Ukoliko bismo hteli da dozvolimo podklasama da izmene ponašanje metode **GetFullName** iz prethodnog primera, potrebno je upotrebiti ključnu reč **virtual** u baznoj klasi i ključnu reč **override** u podklasi.

```
public class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public virtual string GetFullName()
    {
        return FirstName + " " + LastName;
    }
}
```

```
public class Employee : Person
{
    public decimal Salary { get; set; }
    public override string GetFullName()
    {
        string originalValue = base.GetFullName();
        return LastName + ", " + FirstName;
    }
}
```

Upotrebom ključne reči **virtual** dobijamo mogućnost menjanja/redefinisavanja metode **GetFullName**. Nakon ovoga, možemo pozvati originalnu metodu upotrebom ključne reči **base**, koja pokazuje na implementaciju bazne klase, ili vratiti potpuno drugačiju vrednost.

Employee je trenutno tipa **Employee**, ali je u isto vreme i tipa **Person**. Ovo znači da možemo upotrebiti tip **Employee**, gde god nam je potreban tip **Person**. Ukoliko ovo uradimo, ne možemo pristupiti nijednom specifičnom članu klase **Employee**, kao što je recimo **Salary**. Sledi primer ovakve upotrebe podklasa:

```
Person person = new Employee();
person.FirstName = "Sander";
person.LastName = "Rossel";
string fullName = person.GetFullName();
Console.WriteLine(fullName);
// Press any key to quit.
Console.ReadKey();
```

Ono što je bitno zaključiti iz prethodnog primera, jeste da će vrednost promenljive **fullName** biti **"Rossel, Sander"**, jer se radi o objektu tipa **Employee**, a ne **"Sander Rossel"**, kao što bi to bio slučaj za objekat tipa **Person**.

Ukoliko želimo da nateramo podklasu na nasledi određene članove bazne klase, neophodno je označiti metode, a samim tim i celu klasu kao **abstract** metodu/klasu. Apstraktna klasa ne može biti instancirana i mora biti nasleđena, odnosno njene apstraktne metode moraju biti nadjačane(*eng. overridden*).

```
public abstract class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public abstract string GetFullName();
}

public class Employee : Person
{
    public decimal Salary { get; set; }
    public override string GetFullName()
    {
        return LastName + ", " + FirstName;
    }
}
```

U ovom slučaju poziv **base.GetFullName** metode više nije moguć, jer nema implementaciju, a nadjačavanje te metode je obavezno. Osim toga, upotreba klase **Employee** ostaje ista.

S druge strane, moguće je eksplicitno naznačiti da klasa ili metoda ne smeju biti nasleđene ili nadjačane. Ovo se može uraditi pomoću ključne reči **sealed**.

```
public sealed class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public string GetFullName()
    {
        return FirstName + " " + LastName;
    }
}
```

Sada kada je klasa **Person** zapečaćena (eng. *Sealed*), ne možemo više nasleđivati od nje. Ovo znači da više nije moguće kreirati klasu **Employee**, koja upotrebljava klasu **Person** kao baznu klasu.

Metode mogu biti zapečaćene samo u podklasama. Ukoliko ne želimo da drugi ljudi budu u mogućnosti da nam nadjačaju metodu, dovoljno je ne obeležiti je kao virtuelnu. Međutim, ukoliko imamo virtuelnu metodu, a podklasa želi da spreči njeno dalje nadjačavanje, ona može biti obeležena kao **sealed**.

```
public class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public virtual string GetFullName()
    {
        return FirstName + " " + LastName;
    }
}

public class Employee : Person
{
    public decimal Salary { get; set; }
    public sealed override string GetFullName()
    {
        return LastName + ", " + FirstName;
    }
}
```

Nijedna podklasa klase **Employee** sada ne može da nadjača **GetFullName** metodu. Ključna reč **sealed** se koristi ukoliko na primer klasa implementira neke bezbednosne provere, koje ne bi trebalo proširivati ni na koji način.

## Nasleđivanje ili Kompozicija

Iako je nasleđivanje jako važno za objekto-orjentisano programiranje, ono može biti i mučno ukoliko imamo ogromno stablo nasleđivanja. Kao alternativa nasleđivanju javlja se kompozicija. Nasleđivanje se odnosi na vezu tipa "je" (*Zaposleni je osoba*), dok se upotrebom kompozicije definiše veza tipa "ima" (*Auto ima motor*). Primer kompozicije je sledeći:

```
public class Engine
{
    // ...
}

public class Car
{
    private Engine engine = new Engine();
    // ...
}
```

## Enkapsulacija

Enkapsulacija predstavlja proces sakrivanja unutrašnjeg rada klase. Ovo znači da određujemo javnu specifikaciju za korisnike naše klase, dok je stvarni posao sakriven. Prednost ovoga je ta što klasa može da izmeni svoj način rada, bez potrebe za izmenom njenih potrošača.

U C#-u postoje četiri ključne reči za modifikovanje pristupa, koje nam omogućavaju pet načina za kontrolisanje vidljivosti koda:

- **private** – vidljivo samo u trenutnoj klasi
- **protected** – vidljivo samo u trenutnoj klasi i njenim naslednicima
- **internal** – vidljivo samo klasama u okviru istog sklopa(*eng. assembly - .exe ili .dll fajl*)
- **protected internal** – vidljivo samo u trenutnoj klasi i njenim naslednicima u okviru istog sklopa
- **public** – vidljivo svima

**Primer.** Ukoliko bismo pravili klasu koja postavlja upite nad bazom podataka, morali bismo imati neku metodu RunQuery, koja bi bila vidljiva svakom korisniku naše klase. Metoda za pristup bazi podataka bi mogla biti različita za svaku bazu, tako da bismo nju mogli ostaviti otvorenu za naslednike. Dodatno, koristimo i neke pomoćne klase koje su vidljive samo u okviru našeg projekta. Za kraj, potrebno nam je da čuvamo i neko privatno stanje, koje ne sme biti menjano izvan naše klase, jer bi to moglo dovesti do nevažećeg stanja. U nastavku je dat predloženi kod:

```
public class QueryRunner
{
    private IDbConnection connection;

    public void RunQuery(string query)
    {
        Helper helper = new Helper();
        if (helper.Validate(query))
        {
            OpenConnection();
            // Run the query...
            CloseConnection();
        }
    }

    protected void OpenConnection()
    {
        // ...
    }

    protected void CloseConnection()
    {
        // ...
    }
}

internal class Helper
{
    internal bool Validate(string query)
    {
        // ...
        return true;
    }
}
```

Ukoliko se ovo kompajlira u neki sklop i tom sklopu se pristupi iz drugog projekta, mogli bismo da vidimo samo **QueryRunner** klasu. Ukoliko bismo kreirali i instancirali **QueryRunner** klasu mogli bismo da pozovemo samo **RunQuery** metodu. Ukoliko bismo nasledili **QueryRunner** klasu, imali bismo i pristup **OpenConnection** i **CloseConnection** metodama. Klasa **Helper** i polje **connection** bi međutim bili zauvek sakriveni od nas.

Klase mogu da sadrže ugnježdene privatne klase, koje su vidljive samo trenutnoj klasi. Te privatne klase imaju pristup privatnim članovima trenutne klase. U nastavku je dat primer ovoga:

```
public class SomeClass
{
    private string someField;

    public void SomeMethod(SomeClass otherInstance)
    {
        otherInstance.someField = "Some value";
    }

    private class InnerClass
    {
        public void SomeMethod(SomeClass param)
        {
            param.someField = "Some value";
        }
    }
}
```

Ukoliko se zaboravi modifikator pristupa, dodeljuje se uobičajena vrednost(**internal** za klase i **private** za sve ostalo).

Podklase ne mogu imati veću pristupačnost od njihove bazne klase, što znači da ukoliko neka klasa ima modifikator pristupa **internal**, njena podklasa ne može postati **public**(ali može biti **private**).

## Polimorfizam

Videli smo nasleđivanje i kako je moguće izmeniti ponašanje tipa kroz nasleđivanje. Naša klasa **Person** je imala **GetFullName** metodu, koja je bila izmenjena u podklasi **Employee**. Takođe, videli smo da ukoliko se traži objekat tipa **Person**, možemo da ubacimo bilo koju podklasu klase **Person**, kao što je **Employee**. Ova pojava se naziva polimorfizam. Polimorfizam dozvoljava izvedenim klasama da specijalizuju implementaciju bazne klase.

```
class Program
{
    static void Main(string[] args)
    {
        Person p = new Employee();
        p.FirstName = "Sander";
        p.LastName = "Rossel";
        PrintFullName(p);
        // Press any key to quit.
        Console.ReadKey();
    }
}
```

```

    public static void PrintFullName(Person p)
    {
        Console.WriteLine(p.GetFullName());
    }
}

public class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public virtual string GetFullName()
    {
        return FirstName + " " + LastName;
    }
}

public class Employee : Person
{
    public decimal Salary { get; set; }
    public sealed override string GetFullName()
    {
        return LastName + ", " + FirstName;
    }
}

```

U ovom primeru metoda **PrintFullName** prima objekat tipa **Person**, ali ispisuje **"Rossel, Sander"**, jer je prosleđeni parametar zapravo podtipa **Employee**, koji nadjačava funkcionalnost **GetFullName** metode. **Employee**, kao instanca izvedene klase predstavlja runtime tip (stvarni tip kada se program izvršava), a bazna klasa **Person** predstavlja tip za vreme kompajliranja koda. Nadjačavanje metoda se dešavaju za vreme izvršavanja koda.

Još jedan primer upotrebe polimorfizma jeste definisanje jednakosti referentnih tipova. Referentni tipovi su uobičajeno jednaki samo ukoliko su njihove reference jednake. Sledeći primer nam pokazuje kako se može kontrolisati ova jednakost:

```

public class Customer
{
    int id;
    string name;

    public Customer(int id, string name)
    {
        this.id = id;
        this.name = name;
    }

    public override bool Equals(object obj)
    {
        if (obj == null)
            return false;

        if (obj.GetType() != typeof(Customer))
            return false;

        Customer cust = obj as Customer;

        return id == cust.id;
    }
}

```

```

public static bool operator ==(Customer cust1, Customer cust2)
{
    return cust1.Equals(cust2);
}

public static bool operator !=(Customer cust1, Customer cust2)
{
    return !cust1.Equals(cust2);
}

public override int GetHashCode()
{
    return id;
}

public override string ToString()
{
    return $"{{ id: {id}, name: {name} }}";
}
}

```

Kako sve klase implicitno nasleđuju od klase **Object**, one mogu da nadjačaju virtualne metode **Equals**, **GetHashCode** i **ToString**, **Object** klase. Kada nadjačavamo **Equals** metodu, trebalo bi prvo proveriti da nema **null** vrednosti i da su tipovi prosleđenih objekata jednaki, kako pozivaoci ove metode ne bi poredili **null** vrednosti ili nekompatibilne tipove. Instance klase **Customer** su jednake ukoliko imaju isti **id**.

Klasa **Customer** ima konstruktor koji inicijalizuje stanje klase. Operator **this** nam dozvoljava da pristupimo članovima trenutne instance i pomaže da se izbegne dvoznačnost.

U nastavku je dat primer koji proverava jednakost instanci tipa **Customer**:

```

using System;

class Program
{
    static void Main()
    {
        Customer cust1 = new Customer(1, "May");
        Customer cust2 = new Customer(2, "Joe");

        Console.WriteLine($"cust1 == cust2: {cust1 == cust2}");

        Customer cust3 = new Customer(1, "May");

        Console.WriteLine($"\\ncust1 == cust3: {cust1 == cust3}");
        Console.WriteLine($"cust1.Equals(cust3): {cust1.Equals(cust3)}");
        Console.WriteLine($"object.ReferenceEquals(cust1, cust3):
{object.ReferenceEquals(cust1, cust3)}");

        Console.WriteLine($"\\ncust1: {cust1}");
        Console.WriteLine($"cust2: {cust2}");
        Console.WriteLine($"cust3: {cust3}");

        Console.ReadKey();
    }
}

```

Operator `==` proverava da li dva objekta predstavljaju istu instancu. Metoda **ReferenceEquals** predstavlja statičku metodu klase **Object**, i radi istu stvar kao i operator `==`, uz razliku da ovu metodu nije moguće nadjačati, što ne važi za operator. Operator `==` i metoda **ReferenceEquals** se ne pozivaju nad instancom objekta, tako da je moguće uporediti **null** vrednost sa nekom drugo vrednošću. Metoda **Equals** se poziva nad instancom, koja iz tog razloga ne sme biti **null** vrednosti, a poređene vrednosti ne moraju predstavljati istu instancu kako bi bile jednake. S obzirom da je u našem primeru metoda **Equals** nadjačana, ona će vršiti poređene polja **id** prosleđenih objekata.

Klasa **Customer** nadjačava i **ToString** metodu, tako da poslednje tri **Console.WriteLine** naredbe neće ispisivati nazive tipova prosleđenih objekata, kao što je uobičajeno.



# Interfejsi

Interfejs predstavlja nešto poput apstraktne klase, ali samo sa apstraktnim metodama. Ne radi ništa više osim definisanja tipa. Dobra stvar kod interfejsa je ta što je moguće naslediti/implementirati više interfejsa, što kod klasa nije moguće.

Primer interfesja:

```
public interface ISomeInterface
{
    string SomeProperty { get; set; }
    string SomeMethod();
    void SomethingElse();
}
```

Kao što se može primetiti, nijedno polje interfejsa ne sadrži modifikator pristupa. Ukoliko klasa implementira interfejs, sve unutar tog interfejsa je javno dostupno. Ovako bi izgledala njegova implementacija:

```
public class SomeClass : ISomeInterface
{
    public string SomeProperty { get; set; }

    public string SomeMethod()
    {
        // ...
    }

    public void SomethingElse()
    {
        // ...
    }
}
```

Ovde klasa **SomeClass** ima tipove **object**, **SomeClass** i **ISomeInterface**. Prefiks “I” u nazivu **ISomeInterface** je uobičajena praksa u C#-u, ali nije obavezan.

Kada se za jednu klasu u isto vreme korsiti nasleđivanje i implementacija interfejsa, prvo se navodi bazna klasa, a zatim lista interfejsa razdvojena zarezima. U nastavku je dat primer primer sa klasama **Person**, **Employee** i **StampCollector**. U prvom delu je prikazano kako bi ove klase bile definisane ukoliko bi se koristilo samo nasleđivanje, a zatim je u sve to ubačena i implementacija interfejsa.

```
public class Person
{
    // ...
}

public class Employee : Person
{
    // ...
}

public class StampCollector : Person
{
    // ...
}
```

```
public class EmployeeStampCollector : Employee
{
    // ...
}
```

Kao što se može primetiti zaposleni(**Employee**) je u isto vreme i osoba(**Person**). Isto važi i za sakupljača markica(**StampCollector**). Međutim, šta se dešava ukoliko je osoba u isto vreme zaposlena, a bavi se i sakupljanjem markica kao hobiem? Klasa **EmployeeStampCollector** u našem primeru može naslediti samo jednu baznu klasu i odabrana je klasa **Employee**. Problem nastaje kada vidimo da sve što klasa **EmployeeStampCollector** i **StampCollector** imaju zajedničko jesu bazna klasa **Person**. Zbog izostanka višestrukog nasleđivanja, ovaj problem moramo rešiti upotrebom interfejsa, na sledeći način:

```
public interface IEmployee
{
    // ...
}

public interface IStampCollector
{
    // ...
}

public class Person
{
    // ...
}

public class Employee : Person, IEmployee
{
    // ...
}

public class StampCollector : Person, IStampCollector
{
    // ...
}

public class EmployeeStampCollector : Employee, IStampCollector
{
    // ...
}
```

Sada su i klasa **StampCollector**, kao i klasa **EmployeeStampCollector** tipa **Person**, ali i tipa **IStampCollector**. **EmployeeStampCollector** je još i tipa **Employee**(i **IEmployee**), jer nasleđuje klasu **Employee**, koja implementira **IEmployee** interfejs.

Kako klasa može da implementira više interfejsa, moguće je imati interfejs koji nasleđuje više interfejsa. Klasa u ovom slučaju mora implementirati sve interfejsje koji su nasleđeni od strane tog interfejsa. Iz očiglednih razloga, interfejs ne može da nasleđuje klasu.

```
public interface IPerson { }
public interface IEmployee : IPerson
{ }
public interface IStampCollector : IPerson
{ }
public interface IEmployeeStampCollector : IEmployee, IStampCollector
{ }
```

Prethodni primer je jako teorijski, tako da će u nastavku biti dat primer iz realnog sveta. Recimo da naša aplikacija zahteva logovanje nekih informacija. Možda hoćemo da logujemo podatke u bazu podataka, ako se radi o proizvodnom prostoru, ali bismo u isto vreme hteli da logujemo sve i na konzolu, zbog debugovanja. Dodatno, ukoliko baza nije dostupna, želimo da logujemo sve i u Windows Event logove.

```
class Program
{
    static void Main(string[] args)
    {
        List<ILogger> loggers = new List<ILogger>();
        loggers.Add(new ConsoleLogger());
        loggers.Add(new WindowsLogLogger());
        loggers.Add(new DatabaseLogger());

        foreach (ILogger logger in loggers)
        {
            logger.LogError("Some error occurred.");
            logger.LogInfo("All's well!");
        }
        Console.ReadKey();
    }
}

public interface ILogger
{
    void LogError(string error);
    void LogInfo(string info);
}

public class ConsoleLogger : ILogger
{
    public void LogError(string error)
    {
        Console.WriteLine("Error: " + error);
    }

    public void LogInfo(string info)
    {
        Console.WriteLine("Info: " + info);
    }
}

public class WindowsEventLogLogger : ILogger
{
    public void LogError(string error)
    {
        Console.WriteLine("Logging error to Windows Event log: " + error);
    }

    public void LogInfo(string info)
    {
        Console.WriteLine("Logging info to Windows Event log: " + info);
    }
}
```

```

public class DatabaseLogger : ILogger
{
    public void LogError(string error)
    {
        Console.WriteLine("Logging error to database: " + error);
    }

    public void LogInfo(string info)
    {
        Console.WriteLine("Logging info to database: " + info);
    }
}

```

Ovo je prilično lepo rešenje, gde je moguće dodavati i uklanjati logger-e, kad god poželimo.

### **Eksplicitna implementacija interfejsa**

Već je pomenuto da ukoliko se implemetira interfejs, svi njegovi članovi postaju javni. Međutim, moguće je i sakriti članove interfejsa. Kako bismo to uradili, moramo eksplicitno implementirati interfejs. Kada se interfejs implementira eksplicitno, njegove članove je moguće pozvati samo ukoliko se klasa upotrebi kao tip interfejsa.

```

public interface ISomeInterface
{
    void MethodA();
    void MethodB();
}

public class SomeClass : ISomeInterface
{
    public void MethodA()
    {
        // Even SomeClass can't invoke MethodB without a cast.
        ISomeInterface me = (ISomeInterface)this;
        me.MethodB();
    }

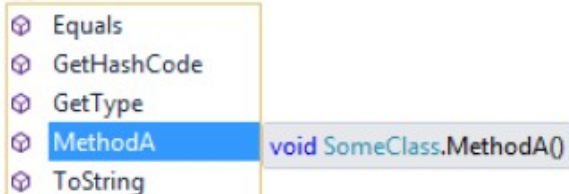
    // Explicitly implemented interface member.
    // Not visible in SomeClass.
    void ISomeInterface.MethodB()
    {
        throw new NotImplementedException();
    }
}

```

```

class Program
{
    static void Main(string[] args)
    {
        SomeClass o = new SomeClass();
        o.MethodA();
        o.M
    }
}

```



Čak ni korisnici klase **SomeClass** ne mogu pozvati metodu **MethodB**, ukoliko tu klasu ne iskoriste kao (odnosno ne koriste je u) **ISomeInterface**.

```
ISomeInterface obj = new SomeClass();  
obj.MethodA();  
obj.MethodB();
```