OOP dedenie
POsledne dva principy OOP su dedenie a polymorfizmus
HOvorile sme si o abstrakcii a zapuzdreni a teraz tu mame dedenie a polymorfizmus.

Snazime sa vytvarat softver tak ze nie su izolovani, nevznikaju nezavisle od inych ale stavaju na pred tym vytvorenych produktov. Ak vytvorime softver a potom vytvarame novy tak mozeme prevziat nejake riesenia, prerobime ho, resp doplnime nove funkcie a mame novy produkt. Nezaciname vsetko robit od zaciatku ale mame nejaku hotovu cast.

A teraz chcem umoznit programovacom jazyku rozsirovat ten existujuci kod dalej. CO sme naprogramovali nechame nemenne a na tom budeme stavat dalej. Rozsirovat. Napor datova struktura bude uchovavat dalsie informacie ale navonol bude stale nemenna.

S tym suvisi znovupouzitelnost a rozsiritelnost. CHceme znovu pouzivat existujuci kod. (Zobrat a pouzit v inom kontexte - napr triedenie cisla ale rovnakym sposobom chcem triedit zoznamy pouzivatelov, zoznamov. Aby sme nemuseli znova pisat program pre vsetko zvlast tak napiseme program tak aby sme to vedeli vyuzit pre kazdy typ. Rozsiritelnost mame uz nejaku funkcionalitu a dorobime nejake nove funkcie. (na prikaldu obchodu - chceme dorobit akciove ceny)
Zadefinujeme si novu strukturu discount article, ktore je podtypom triedy article. Discount article ma vsetky vlastnosti toho article plus nieco navyse. z matematickeho hladiska je mensi specificky typ discount article ako article.

Subtypying len poh+lad typov - nejaky typ viem nahradit inym typom (vsetko co robim s tovarom s robi v article)
pokial sa pozriem hlbsie na implementaciu typu to uz je dedenie (discount article prevezme vsetko z artcile a doda nova metodu discount)

samotnysubtyping hovori ze discounta rticle dedi od article

implementuje znovupouzije tu implementaciu ktora bola pre article + nieco navyse

subclassing iny pojem pre dedenie

ked dedim vlastnosti dedim aj

vzdy sa zavola aj konsturktor nadriadenej triedy a vieme si to riadit ktory konstruktor. zapuzdrenie a information hiding zachovvava vsetko tak ako mame

Abstrakciu si predstavujeme tak ze zlavu riesime v triede discount article.

abstrakcie-vsetka funkcionalita ktora suvisi s danym objektom vychadza z pojmu mnozina a operacia nad mnozinou
zapuzdrenie - v ramci triedy, abstrakcia realizovana zapuzdrenim, napr. getPrice, zlava je zapuzdrena v triede
skryvanie informacie volanie nadradeneho konstruktora namiesto toho aby sme priamo volali name,price,quantity
pristup k cene napiseme len cez getter (super je dolezite aby sme nevolalai rekurzivne funkciu) k comu nemame pristup nepokazime :), povolujeme len to co je dolezite
dedenie dedi vlastnosti, pridava nove vlastnosti, niektore vlastnosti vieme aj zmenit

ako programator ako najlepsie vyuzijem oop aby bol softver bol znovupouzitelny a rozsirenie
open/closed princip - otvoreny pre nove funkcie uzavrety pre zmeny

polymorfizmus - tovar je nielen article aj aj discount article

listkovej princip substitucie

niektore vlastnosti si treba uvedomit getpriceje aj v articleaj discounnt article

neskore viazanie metod, neskore volanie metod - az pocas behu programu - preto neskor, vieme presne povedat z ktorej triedy bude funkcia zavolana problem prekrytych funkcii

virtualne fukncie v C# apparent type vzdy ak sa nepouzije virtual

ak je prekryta a chcema zavolat prekrytu tak this
ak je prekryta a chceme povodnu tak super (resp. base)

clenske premenne len private !
protected len metody

- neodporuca sa pouzivat dedenie ale kompozicia ! subtyping ano ale dedenie nie

subtyping jeden druh nahradit druhym
dedenie zoberiem implementaciu a prepisem, overriding

SLIDE
Na predchadzajucich prednaskach sme sa zaoberali abstrakciou a zapuzdrenim. Dnes si nieco povieme k poslednymi dvom principom a to k dedicnosti a polymorfizmu. A tym budeme mat zakladne principi OOP zvládnute.

SLIDE SLIDE
Zacneme kratkym opakovanim. Co je to trieda? Trieda je sablona resp. vzor, ktora predpisuje ake clenske premenne a metody maju objekty danej triedy a co sa bude diat ak ich zavolame.

SLIDE
O co sa pri navrhu softveru snazime? Snazime sa vytvarat taky softver, ktory nie je izolovany a nezavisly od inych. Snazime sa tvorit softver tak aby bolo mozne dalej nad nim pracovat, staviame na veciach, ktore sme uz urobili/ niekto iny urobil. Nezaciname vsetko robit od zaciatku, vyuzijeme riesenia, ktore uz su, a ktore doplnime o nove funkcie a tak nam vznikne novy produkt.

Chceme umoznit rozsirovat existujuci kod dalej. Co sme naprogramovali ostane nemenne a na tom budeme stavat dalej. Napr. datova struktura bude uchovavat dalsie informacie ale navonok bude stale nemenna.

SLIDE
S tym suvisia pojmy ako znovupouzitelnost (reusability) a rozsiritelnost (expendability). Chceme znovu pouzivat existujuci kod.

Reusability: Zobrat a pouzit v inom kontexte - majme napr program na triedenie cisel ale rovnakym sposobom chceme napr. triedit zoznamy pouzivatelov, alebo zoznam nieco. Aby sme nemuseli znova pisat program pre vsetko zvlast tak napiseme program tak, aby sme to vedeli vyuzit pre kazdy typ.

Rozsiritelnost mame uz nejaku funkcionalitu a dorobime nejake nove funkcie. (Vezmeme si naprikla obchod, ktory je charakterizovany polozkami, polozky maju svoj nazov, cenu a mnozstvo. My chceme implementovat metodu, ktora nastavi ceny poloziek na akciove ceny produktom. Zadefinujeme si novu strukturu discount article, ktore je podtypom triedy article. Discount article ma vsetky vlastnosti toho article plus nieco navyse.
Z matematickeho hladiska je mensi specificky typ discount article ako article.

Alebo si vezmime priklad animal. Je charakterizovany menom, velkost, celad, zvukom, ktory vydava. Chceme zadefinovat novu triedu pes, ma rovnake vlastnosti ako animal a k tomu nejake navyse.

SLIDE
Tak mozeme povedat ze dedicnost je ked vytvarame nove triedy pouzitim vlastnosti uz existujucich tried. Nova tried znovupouziva, rozsiruje alebo modifikuje spravanie povodnej triedy. Povodnu triedu nazyvame base, super alebo parent - rodic. Triedu vzniknutu z povodnej nazyvame odvodena (derived), subb (pod), alebo child - potomok

SLIDE
C# nepodporuje multinasobne dedenie.Co je multinasobne dedenie? Ked jedna trieda dedi od viacerych nadtried. Vyuzitim Interfacov sa da dosiahnut nieco podobne, viac info na dalsej prednaske.
(C# does not support multiple inheritance. However multiple inheritance can be achieved using Interfaces.)

Dedenie je tranzitivne co znamena ze ak mame styri triedy A,B,C,D. Trieda A je zakladna trieda, trieda B je od nej odvodena, trieda C je odovodena od tridy B a trieda D je odvodena od triedy C, tak potom cleny triedy A su dostupne aj triede D. Je potrebne pamat na to, ze nie vsetky cleny triedy sa da dedit. Co sa neda dedit je staticke konstruktory (inicializuju staticke premenne, konstruktor na vytvorenie novej instancie triedy a finalize cize destruktor na uvolnenie pamate. Aka je pristupnost ostatnych cleno triedy nastavuju modifikatory pristupu a to private, protected, internal a public.

SLIDE
Privatne cleny su dostupne iba v konkretnej triede, nie su dostupne odvodenym triedam ale je potrebne si dat pozor na nested triedy.

Protected cleny su dostupne odvodenym triedam. Klucove slovo protected pouzivame len pre metodty, clenske premenne ponechavame private. Internal oznacuje cleny pristupne iba v ramci jedneho assembly. Public slovom su oznacene cleny, ktore su sucastou odvodenych tried a verejnych rozhrani odvodenej triedy. Ak nenapiseme nic, tak sa nemozeme spoliehat, ze to program urobi za nas. Neurobi!

SLIDE
Ako mozu byt tvorene nove metody? Odovdene triedy maju moznost prepisat zdedenych clenov. Ak chceme prepisovat cleny v base triede musime ich nastavit klucovym slovom virtual. Niekedy chceme aby vytvorena trieda prepisal parent triedu. Vtedy ak pouzijeme klucove slovo new tak skryjeme metody base triedy a pouzivame len metody odvodenej triedy.

SLIDE
Dalsim modifikatorom oznacovania tried a metod je abstrakt. Z abstraktnej triedy sa neda vytvorit objekt. Obsahuje len formalne zapisy metod. Ak chcem pristupovat k abstraktnym clenom musim zadefinovat metodu a triedu v povodnej triede. A posledne slovicko je slovicko seal. Ak nechcem aby od danej triedy dedili dalsie metody je nacase to zastavit.

SLIDE
AKo som povedala, dedicnost sa vztahuje len na triedy a rozhrania. Ine typy ako su napriklad struktury enumeracne typy neumoznuju dedicnost

SLIDE
Polymorfizmus vychadza z greckeho poly a morph. Poly oznacuje viac a morph tvar. Volny preklad je mnohotvary. To jest napr. jeden nazov ma vela foriem. V C# pozname 2 typy polymorfizmu a to parametricky alebo subtyping. Existuju aj dalsie pomenovanie polymorfizmu ako casting a ad-hoc.

SLIDE
Subtyping je pouzitie jedneho typu tam, kde sa ocakava pouzitie druheho typu. Metody resp. funkcie su pisane pre supertypy (nadtypy) ale budu fungovat aj na podtypoch . Napr. AK je trieda B podtriedou triedy A, tak vsade kde sa ocakava, ze pride A mozeme zadat aj B. Subtyping je late binding, a spada tu aj prepisovanie metod, kedy prekladac rozhodne az pocas behu programu, ktora metoda sa ma vykonat. Nazov metod a ich popis musi byt rovnaky.

- prepisovanie metód (overriding)
- názov metód a ich popis (typ a počet parametrov) musí byť rovnaký
- prekladač rozhodne počas behu programu, ktorú metódu zavolať (ak nie je žiadna k dispozícií vráti nám chybu)

subtyping is a relation between types
inheritance is a relation between implementations stemming from a language feature that allows new objects to be created from existing ones
In a number of object-oriented languages, subtyping is called **interface inheritance**, with inheritance referred to as *implementation inheritance*

SLIDE
Parametricky polymorfizmus pouziva typ ako parameter. v C# pod to spadaju generika. Mame vytvorenu funkcionalitu, ale typy do nej vstupujuce su neurcene. Napr. chceme zoradit nejaky zoznam ale chceme to vyuzivat aj pre typy integer, aj pre double, aj pre string. Napiseme genericku metodu. Prekladac vyhodnoti typy pri preklade programu to znamena ze hovorime o early bindingu.

SLIDE
V praxi sa zvycajne miesto dedicnosti vyuziva kompozicia. O tom blizsie nabuduce.

```
using System;

namespace Lecture03
{
    class Program
    {
        static void Main(string[] args)
        {
            Dog pes1 = new Dog("Benny");
            Cat macka1 = new Cat("Cecil");

            Animal a1 = new Animal("Benn");

            pes1.Pet();
            macka1.Pet();

            pes1.Eating();
            pes1.Seeing();

            //a1.Eating();

            Animal a = pes1;
            a.Seeing();
            //a.Pet();
            //a.Eating();

            //Animal b = macka1;
            //b.Pet();
            //b.Eating();


        }
    }
}
```

Caption

Lecture03/Animal.cs

```
using System;
namespace Lecture03
{
    public class Animal
    {
        //pet - ako sloveso maznat

        private string Name;

        public Animal()
        {
            Console.WriteLine("konstruktor bez parametrov");
        }

        public Animal(string Name)
        {
            Console.WriteLine("Konstruktor s parametrom");
            this.Name = Name;
        }

        public virtual void Pet()
        {
            Console.WriteLine("no sound");
        }

        public virtual void Eating()
        {
            Console.WriteLine("jedlo");
        }

        public virtual void Seeing()
        {
            Console.WriteLine("Nevidim");
        }

}
}
```

```
using System;
namespace Lecture03
{
    public class Dog : Animal
    {
        private string Name;

        public Dog(string Name) : base(Name)
        {
            this.Name = Name;
        }

        public override void Seeing()
        {
            Console.WriteLine("Vidim");
        }

        //public new void Pet()

        //polymorfizmus
        public override void Pet()
        {
            Console.WriteLine(Name + " : hav hav");
        }

        public void Pet(string X)
        {
            Console.WriteLine(Name + X + " : hav hav");
        }

        public override void Eating()
        {
            base.Eating();
            Console.WriteLine(Name + " zerie granule");
        }

    }
}
```

```
using System;
namespace Lecture03
{

    class Cat : Animal
    {
        private string Name;
        public Cat(string Name) : base(Name)
        {
            this.Name = Name;
        }

        //public new void Pet()

        //polymorfizmus
        public override void Pet()
        {
            Console.WriteLine(Name + " : miau maiu");
        }

    }
}
```

```
using System;
namespace Lecture03
{
    public class PetOwner<T> where T : Animal
    {
        public T animal;

        public void SetPet(T p)
        {
            this.animal = p;
        }
    }

}
```

```
//PetOwner<Cat> catOwner = new PetOwner<Cat>();     //
//PetOwner<Dog> dogOwner = new PetOwner<Dog>();     //
//PetOwner<Animal> petOwner = new PetOwner<Animal>();

//petOwner.animal = new Cat("Helena");     // OK
//petOwner.animal = new Dog("Jozi");

//catOwner.animal = new Cat("Helena");     // OK
//catOwner.animal = new Dog("Jozi");
```