16. Rozhraní, dědičnost, abstraktní třídy a přetěžování

Interface

Rozhraním objektu se myslí to, jak je objekt viditelný zvenku.
Rozhraní také můžeme chápat jako abstraktní třídu, která má všechny členy abstraktní.
Stejně jako třída může mít rozhraní jako členy metody, vlastnosti (get a set), události.
Rozhraní ale budou obsahovat pouze deklaraci členů.
Implementaci členů rozhraní zajistí třída, která implementuje rozhraní implicitně nebo explicitně.
Rozhraní (Interface) je kontrakt, který specifikuje operace, které má třída, která ho implementuje, musí splňovat, a který se již nezabývá tím, jak toho bude konkrétně dosaženo.
Rozhraní objektu tvoří právě jeho veřejné metody, je to způsob, jakým s určitým typem objektu můžeme komunikovat.
Výhodou rozhraní je, že pro třídu můžeme implementovat více rozhraní. Při dědění přebírá třída od rodičovské třídy jeho rozhraní.

Vše co může být napsáno do interface je automaticky public abstract, kromě konstant které jsou automaticky jen public, protože abstract není platnej pro instanci.

Výhody třeba u kolekcí

Máme produkty, třeba fyzické nebo digitální, tak dáme, ať obě implementují interface IProdukt
A pak vytvoříme kolekci s datovým typem IProdukt, takže do ní půjdou vkládat objekty, které implementují IProdukt
List<IProdukt> list = new List<IProduct>();
Interface může také implementovat (doplňuje) od ostatních interface, třída nebo struktura, která ho poté implementuje, musí implementovat i ty od kterých interface dědí (dědící řetěz)
Nemůžeme vytvořit instanci interface, ale můžeme vytvořit referenčí proměnou.
Třeba u listu si můžeme vybrat z těchto interface
ICollection<T> IEnumerable<T> IList<T> IReadOnlyCollection<T> IReadOnlyList<T> ICollection IEnumerable IList
A napsat IList list = new List<int>();
Tak budeme mít k dispozici jenom metody, které jsou napsané v IList nebo v rozhraní, které IList implementuje (ICollection, IEnumerable). V tomto přídapě je IList negenerickej takže ani neví, co tam mám dát za datový typ

Abstraktní třída

```
Abstraktní třídy označené klíčovým slovem "abstrakt" v definici třídy se
obvykle používají k definování základní třídy v hierarchii.
Něco mezi interface a třídou
Abstraktní metody jsou reprezentovány pouze hlavičkou funkce a jejich
implementace je ponechána na odvozené třídy

V děděných třídách se používá u metod override,
Také nemůžem vytvořit instanci této třídy
Platí stejná pravidla dědění jako u tříd, může se dědit pouze jedna třída
nebo abstraktní třída
Dají se použít, třeba když nějaké metody chceme definovat s defaultním
základem a definici některých chceme nechat na třídě, které jí zdědí
Neabstraktní metody nejsou automaticky public

Dobrý příklad je udělat abstraktní třídu Connection a od toho oddědit
AcceptedConnection a RefusedConnection. U třídy Connection bude např.
metoda Disconnect nebo broadcastToClient() úplně stejná.Tím se vyhneme
opakovanému psaní kódu.
```

Třída může zdědit pouze jednu abstraktní třídu	**Třída může implementovat více interfaců**
Abstraktní třída může obsahovat konstruktor	Interface nemůže obsahovat konstruktor
V abstraktní třídě můžou být statické prvky	V interfacu nemůžou být statické prvky
V přístupu prvků a metod může být public, private a protected.	V přístupu může být public a protected (ale protected se neuplatní asi).
Rychlej.	Pomalej.
Abstraktní třída může obsahovat metody, definované metody, konstanty, proměnné...	Interface může obsahovat pouze abstraktní metody a konstanty
Může mít vlastní soukromé proměnné a metody.	Nemůže

Dědičnost

Třída může dědit pouze jednu třídu v Javě.U C++ je mnohonásobná dědičnost. Když dědí, tak získává všechny vlastnosti, metody (ty které nejsou private) Pro dědění se používá ochrana protected, dědící třída získá všechny protected od děděné.
Můžeme ukládat třídu jako datový typ třídy, kterou daná třída dědí, poté máme přístup jenom k vlastnostem a metodám jako třída která je použita jako datový typ proměný

Přetěžování

Jde o jednoduchou myšlenku umožnit deklarování funkcí se stejným jménem, které by se chovaly jinak v závislosti na typech vstupních parametrů. Při přetěžování je možné vytvořit libovolné množství funkcí stejného jména za předpokladu, že budou rozlišitelné. Aby se překladač mohl rozhodnout, kterou funkci použít, vyžaduje odlišnosti v počtu nebo v typech parametrů. Při výběru vhodné funkce překladač postupuje tak, že vybere tu s vhodným počtem parametrů a následně se řídí v pořadí těmito pravidly:
1. Přesná shoda v typech.
2. Shoda v typech po rozšíření typů (char a short na int, float na double).
3. Shoda v typech po implicitním přetypování (např int na double). Všechny konverze jsou považovány za stejně významné, žádná není upřednostněna.

```
    public int Add(int a, int b){
        return a+b;
    }
    public int Add(int a,int b,int c){
        return a+b+c;
    }
```

Přetížené metody, se stejným názvem. Jedna má 2 vstupní inty, druhá 3. Konstruktor se také často přetěžuje, třeba několik s povinnými vstupy a jeden bez vstupů.