Zkouška - PROG 2 - přehled

#shrnuti

Zkouška

- vnější a vnitřní třídění

0. Upřesnění zadání, pokud je potřeba
1. Postřehy
2. Zdůvodnění volby algoritmu
3. Reprezentace dat - jaké datové struktury, aktuální předmět atd. , počet , a jiné vlastnosti
4. Dekompozice programu - např. *data-flow diagram* algoritmu
5. diskuze

Uzávorkování matic

Externí třídění

- setřídit rozsáhlá data uložená v souboru, která se nevejdou do pole
- nelze tedy použít většinu algoritmů vnitřního třídění.
- **slévání** = ze dvou setříděných souborů vytvoří jeden setříděný soubor
- úseky se rozdělí střídavě do dvou souborů, pak úseky se po dvou slévají a ukládají se do jednoho výsledného souboru
- V každém kroku je každý záznam jednu nebo dvakrát čten a zapisován - O(N) (=1krok)
- Po K-tém kroku výpočtu mají úseky délku 2^K => $2^K = N$ => $K = log2N$ => O(Nlog2N)
  Virtuální metody vs abstraktní
  Diskrétní simulace
  Dědičnost/polymorfismus
  Hashování
  Stromy
  Dynamické programování
  Grafové algoritmy
  Dijkstra
  Tabulka virtuálních metod
  Abstraktní třídy
  Generické třídy Generické metody
  Unit testy
  Halda

Garbage Collector

- pracuje tak, že pravidelně prohledává paměť aplikace a zjišťuje, které objekty se ještě používají a které již nejsou potřeba.

1. přednáška

vlastnosti "C#"

- Program = hromada tříd, třída Program má v sobě Main()
- třídy obsahují datové složky a metody. Instance tříd = Objekty
  - this. - odkaz na tuhle proměnnou
- program v C# = projekt s více soubory
- **namespace/jmenný prostor** = zabalení a identifikace programu - jak se program reprezentuje ostatním programům -
- na začátku programu - *using System* (=using System Library in our project)
- Syntaxe: 1. Intro to CSharp > Cim se lisi C od Python
- veřejné a neveřejné proměnné

datové typy

- každá proměnná má svůj typ
- se dělí na **hodnotové** a **referenční**
  - **hodnotové typy**: int, double, enum, struct, char
    - např. int i = 100; Systém uloží 100 do adresy přidělené proměnné i
    - pokud hodnotovou proměnnou, definovanou v jedné funkci, chceme změnit v jiné funkci, a pak ji opět vyvoláme ve své funkci, její původní hodnota se nezmění
  - **referenční typy**: string, pole, třídy, delegáty
    - např. string s = "Hello World"; Referenční typ obsahuje ukazatel (hodnota s je 0x600000) na jiné místo v paměti (0x803200 - zvoleno náhodně), kde je uložena hodnota s.
    - v jiné metodě můžeme změnit její hodnotu
- celé číslo - int, System.Int32
- desetinné číslo - double, (64 bit)
- znak - char (16 bit Unicode)
- pole - [], System.Array
  - všechny prvky jsou stejného typu
  - obdélníkové [,], nepravidelné [ ] [ ] - ty můžou v sobě mít různé délky polí. V jiných slovech: je to pole ukazatelů na jednorozměrné pole

```
int[][] aaa = new int[3][];
aaa[0] = new int[4]; //kazde pole musime jeste definovat
aaa[1] = new int[6];
aaa[2] = new int[2];
```

- řetězec - string, System.String
  - obsah nelze měnit, na to je třída StringBuilder
  - formátování:

```
Console.WriteLine($"a={a} b={b} a*b={a*b}");
```

- objekt - StringBuilder, ArrayList, List<>
- tuple - System.Tuple, System.ValueTuple
- *při překladu probíhá kontrola typů*

enum

- výčtový typ, reprezentuje konstanty

```
enum Level {
  LOW,  //0
  MEDIUM,  //1
  HIGH  //2
};

enum Level {
  LOW = 25,
  MEDIUM = 50,
  HIGH = 75
};

enum Level {
  LOW = 5,
  MEDIUM, // 6
  HIGH // 7
};

int main() {
  // Create an enum variable and assign a value to it
  enum Level myVar = MEDIUM; //1 //50 //6

  // Print the enum variable
  printf("%d", myVar);  //1 //50 //6


  return 0;
}
```

struct

- hodnotový typ - struktura - obsahuje různé datové typy (konstruktory, konstanty, fields, methods, properties, events atd.)
- má bezparametrický konstruktor, neumí dědit, se nemusí alokovat

```
public struct Person
{
    // Declaring different data types
    public string Name;
    public int Age;
    public int Weight;
}

//in main
 Person P1;

// P1's data
```

```
P1.Name = "Verasik";
P1.Age = 20;
P1.Weight = 54;
```

AL spojky

- &&, || - logický
- &, | - bitový a logický - úplné vyhodnocení
- ! - not
- ^ - xor - (p or q) and neg(p and q)
- *aritmetické* - +=, ++, / ... dělení celočíselné a desetinné

checked, unchecked

- klíčová slova - použijí se na blok, nebo na výraz
- určuje, zda došlo k aritmetickému přetečení

```
int a = int.MaxValue;
//unchecked { Console.WriteLine(a + 3); // output: 2 }
checked
{
        Console.WriteLine(a + 3); //Unhandled Exception: System.OverflowException: Arithmetic
operation resulted in an overflow.
}
```

funkce

- void, vs vraci nejaky datovy typ
- **předávání parametrů**: standardní hodnoty, *odkaz (ref)* - v hlavičce i při volání, *výstupní parametr (out)* - v hlavičce i při volání
  - ref - musíme deklarovat proměnnou před její změnou v jiné metodě
  - out - vhodné pro nedeklarované proměnné, deklarujeme je až v nějaké metodě

Main()

- výchozí statická metoda ve třídě - plní funkci hlavního programu
- jediná v aplikaci

static void Main(string[] args)

dynamicky alokované proměnné

- vytvářejí se pomocí `new + konstruktor`, new (funkce, ukazatel) - vrací vytvořenou instanci
- referenční typy : `new String, new Object, new string[]`
- default: null

2. přednáška

- druhy programování

- **Strukturované** - blok, funkce
- **Modulární** - modul, unit
- **Objektové** - objekt

Objekty

- referenční typ, dynamicky alokované proměnné

```
Human p = new Human("female", 21, "green", "brown");
```

- u každého objektu se volá Konstruktor

Třída

- mohou obsahovat jiné objekty (tělo obsahuje buňky, letadlo součásti)
- podobné objekty = instance jedné třídy (můžou mít různé vlastnosti)
- v definici třídy mohou být **data (properties)** a **metody**
- *this.* - řešení kolize jmen, v Konstruktoru
  - `if (this is Zakaznik)` tenhle objekt je typu Zakaznik?
- třída může mít více konstruktorů a stejnojmenných funkcí (musí se lišit parametry)

Konstruktor

- metoda volaná při vytváření instance
- jmenuje se stejně jako Třída

Dědičnost

- odvozený datový typ dědí od svého rodiče/předka *všechny datové položky a metody*
  - může přidávat datové položky a metody
  - nebo přepisovat metody
- konstruktor : **base**(parametry rodiče)
  - volá konstruktor či metody svého rodiče (super())
- v C# - nelze dědit od více předků

Metody

- můžeme metodu předefinovat
- bez konstruktoru nelze vytvořit objekt

Předefinování metody

- klíčové slovo **new** - skryje přístupnou metodu základní třídy
- in public new void method(...) {base.method(...) }

virtuální metody

- deklarace: **virtual** = metoda základní třídy
- předdefinování (v potomkovi): **override**

- třída si pamatuje **tabulku virtuálních metod (VMT)** – virtual method table – ta třída má v paměti svoji tabulku virtuálních metod. V té tabulce jsou položky, kde v paměti jsou **ukazatelé** dané funkce.
  - obsahuje adresy virtuálních metod, a při volání přepsané override metody, změní se odkaz na tu override metodu
- když se zavolá konstruktor, tak do dat svého objektu napíše údaje o tom, jakého je typu - dosadí odkaz na VMT
- VMT tabulka je stejná pro všechny objekty té Třídy
- v C++, kompilátor vytvoří samostatnou VMT pro každou třidu. Když je objekt vytvořen, přidá se pointer do VMT (a to je skrytá součást objektu)

```
object obj = "asdf";
string text = obj.ToString();
//String.ToString() - obj will call virtual method of String, not of the Object class
```

- přetížené operátory
  - operátory jsou pro nějaké definované třídy hned zabudované. Ale když vytvoříme novou třídu - dojde k Overloadu.
  - Musíme proto vytvořit metodu, kde zabudejeme operátory
  - v počítače máme jen celá a necelá čísla
  - pokud chceme například jiný obor (komplexní čísla) - přetížíme operátory - dodefinujeme je, např. přerdefinujeme virtuální metodu ToString

```
public override string ToString() { return ""+re+"+ "+im+"i";}
```

abstraktní metody

- **abstraktní třída** = nevytváříme instance této třídy, slouží jen pro dědění. Může obsahovat i abstraktní i neabstraktní metody, lokální proměnné. Třída může dědit jenom z jedné (abstraktní) třídy.
- **abstraktní metoda** = virtuální, nemůže být volána (nemá tělo), jenom předefinovaná v potomkách její třídy (může být předefinovaná zase abstraktní metodou)
- **Interface** = abstraktní třída, která zahrnuje jenom abstraktní funkce, které potomky musí (všechny) implementovat. objekt může splňovat více interfaců. Nemá lokální proměnné

Polymorfismus

- (skupina) tříd, která sděďuje součásti (proměnné, metody) hlavní třídy
- volá se metoda příslušná aktuální třídě objektu

statické členy a třídy

- **statické členy**
  - nemají instance, jsou přípustné pomocí jména třídy
  - jsou alokovány ve třídě, ne instanci
- **statická třída**
  - obsahuje pouze statické členy
  - nelze z ní vytvářet instance pomocí new
  - nelze z ní dědit

viditelnost

- **public** = přístupné všem
- **protected** = přístupné jen z této třídy/struktury, a z potomků
- **private (default)** = přístupné jen z této třídy/struktury
- internal = viditelné pouze ze současného souboru

zapouzdření / encapsulation

- aby objekt toho zveřejňoval co nejméně - má privátní metody a vlastnosti
- metody pro přístup k proměnným:
  - get(x) - dovolí objektu to jen číst
  - set(x) - dovolí objektu měnit hodnotu proměnné

```
`private` `String studentName;`

`private` `int` `studentAge;`

`public` `String Name`

    `{`

        `get` `{` `return` `studentName; }`

        `set` `{ studentName = value; }`

    `}`
`public` `int` `Age`

    `{`

        `get` `{` `return` `studentAge; }`

        `set` `{ studentAge = value; }`

    `}`
```

sealed class

- nelze od této třídy dědit (říká to kompilátoru)

3. přednáška

- simulace - spojitá a diskrétní
  - spojitá = simuluje každou vteřinu
  - diskrétní = zabývá se jen momenty, **kdy se něco děje**
- **události / events**, seznam událostí = **kalendář**
  - 3 atributy - kdy, kdo, co

- např. 0 - A přijíždí do M, 0 - B přijíždí do M, 0 - A počíná nakládat, 120 - A je naloženo a vyjíždí z M, 120 - B začíná nakládat, 300 - A přijíždí do N

Diskrétní simulace

Mám nějaké události -> do fronty -> setridena podle casu -> tu udalost zpracuju
Funguje jako Dijkstrův algoritmus

=pro ní je typické, že pokud nastane nějaká událost, proměnná se v modelu okamžitě mění

- příklad: Supermarket, obchod - obsluha zákazníka na pokladnách. Pomocí diskrétní simulace, vedoucí prodejní může optimalizovat počet pokladních na pokladnách, tak aby nedocházelo k dlouhým frontám / aby nebylo příliš hodně pokladníků v pustém obchodu
- složky: **čas, události, fronta** atd.

4. přednáška

program v jazyku CSharp

- zdrojový kód: .cs
- soubor .csproj - popisuje projekt, ve kterém může být více .cs souborů
- .sln (solution) - úroveň nad .csproj - může obsahovat více projektů
- v adresáři programu jsou bin a obj (podadresáře)
  - obj - nejprve se přeloží tam objekty
  - bin - z těchhle objektů se tam sleje zdrojový kód. Tam je soubor .exe, který se spouští
- pokud máme .cs .csproj .sln, adresáře obj a bin se automaticky vytvoří

vstupy, vystupy

- System.IO.StreamReader("soubor.txt") , System.IO.StreamWriter("soubor.txt")
- 1. Intro to CSharp > switch - něco o switch

5. přednáška

Programování řízené událostmi

- už víme z diskrétní simulace:
  - složitý decentralizovaný systém - skládá se z mnoha částí (tříd), které připraveny na zpracování událostí. Každá z nich dělá to, co umí
- události
  - (např. od myše) - stisknout (poté se otevře např. soubor), pustit, pohnout.
    - (Button) sender
  - události zpracovávají všichni (hierarchie objektů, úřad apod.)
  - **low-level** = jednoduchý pohyb myší (jedním směrem)
  - **high-level** = složité pohyby myší ((dvojité) stisknutí)
- Program má podřízené (**komponenty**), ty má nějak seřazené, postupně jim říká – nastala tahle událost, zpracuj ji, první říká – to se mě netýká, tak se zeptá dalšího. Až najde toho, který se k tomu přihlásí, tak ten to nějak zpracuje. Ten to řekne svým podřízeným.
- **knihovny s komponenty**: WinForms, WinUI, WPF

- Windows - fce zpracuj_udalost()
- hry, jako Pexeso

6. přednáška

Dynamické programování

- technika, jak počítat větší hodnoty z menších
- hledá nejlepší řešení nebo počet možností dohromady
- memoizace = rekurzivní funkce s pamětí (cache) - aby neopakovala stejné výpočty
  - bez memoizace - T(n) je exponenciální
  - s memoizací - T(n) je polynomiální
- příklady:
  - BVS o K vrcholech
    - máme k možností, jak najít kořen
    - pak i-1 vrcholů v LP, a k-i vrcholů v PP
    - určíme triviální případy, pak rekurentní vzorec
  - Kolika způsoby se dá 20 zapsat pomocí 1, 2, 5
  - Násobení matic
    - triviální případ: $P_{1,i} = 0$
    - rekurentní vzorec: $P_{k,i} = min(P_{j,i} + P_{k-j,i+j} + R_i * R_{i+j} * R_{i+k})$
  - Editační vzdálenost
  - Nejdelší cesta v topologicky uspořádaném grafu
  - atd.

8. přednáška

Výjimky

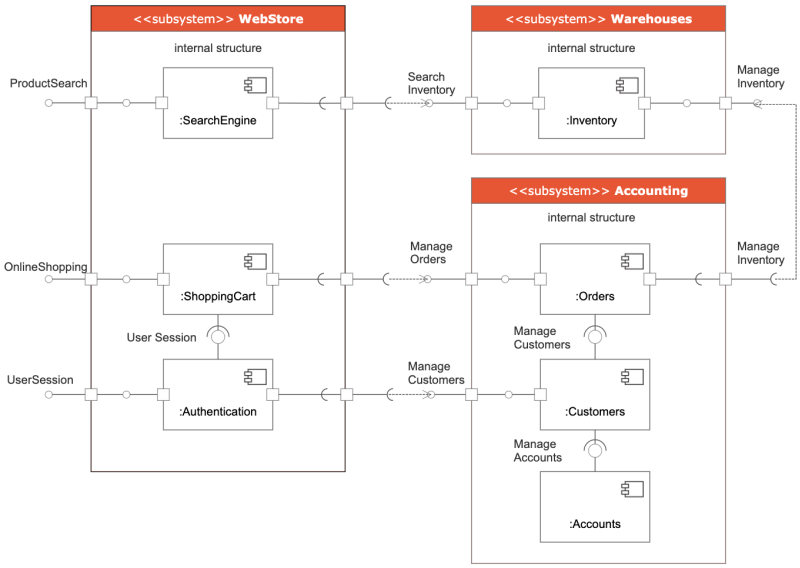
- odchytávají chyby v kódu
- v OOP jazycích
- chyby jsou **běhové** a **kompilační**
  - běhové / výjimka - když se program spustí a zhroutí
  - kompilační - se najde při kompilaci
- C# výjimky: try {} catch {název_výjimky} finally*
  - finally se používá pro zavření souboru, uvolnění paměti - např. return
- výjimka se vyvolává pomocí **throw**
- všechny výjimky = potomky Exception
  - např. IndexOutOfRangeException

Unit Testy

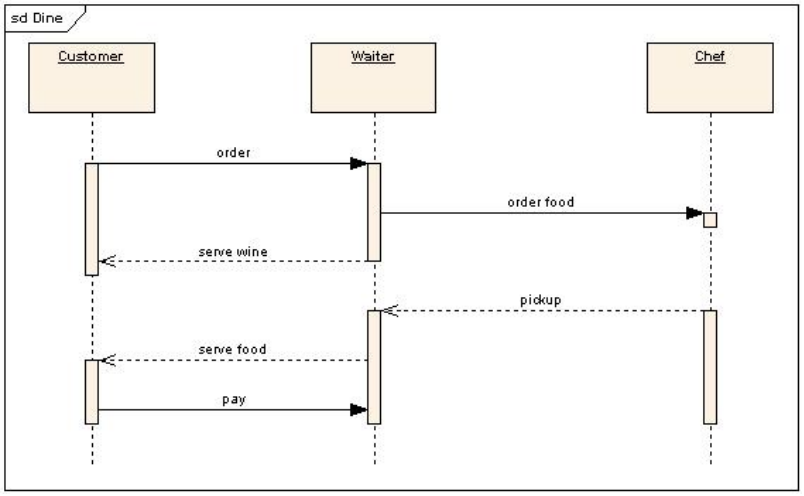
- testovací projekt
- testují se jednotky (např. funkce)
- k Solution se přidá nový projekt např. MSTest Test Project
- jsou **internal** - viditelné jen ze současného souboru
- např. Assert.AreEqual(a, b)

10. přednáška

- Uml2
  - = pravidla, jak se mají kreslit obrázky



- Diagram tříd
  - obrázek - znázorňuje přehled všech tříd
- Diagram stavů
  - stavy a přechody mezi nimi
- Sekvenční diagram
  - průběh spolupráce objektů, shora dolů běží čas



Objektový návrh

- jaké třídy, veřejné členy (funkce + data) tam budou

1. Princip jedné zodpovědnosti / Single Responsibility Principle

= Jedna třida (objekt) by měla vykonávat 1 věc

Princip otevřenosti a zavřenosti / Open-closed principle

=Můžeme tvořit či rozšiřovat nové třídy, ale už potom mají definitivní rozhrání (jsou uzavřené, nebudeme je měnit)

Příklad: Přelévání

- **Fronta** = Přidej, Vyber
- **Stav a počet kroků**
- **Stav** - Stav(), Nastav(), Nacti(), VratNasledniky(), ObsahIteNadoby()
- **Zvané stavy**
- **Objemy** - JestliJesteNeznasTakPridej(int kroky)
- **Velikosti nádob**

```
Veliksoti = Velikosti();
Stav stav = new Stav();
stav.Nacti();
Fronta f = new Fronta();
f.pridej(stav)

while ...{
        (stav, k) = f.Vyber();
        foreach (stav s in stav.VratNasledniky())
                if (!(stav in znameStavy)) { f.pridej(s, k+1)}
}
```

přidavné jméno = kandidát na třidu
sloveso = kandidát na funkci

Zákon substituce / Liskov substitution principle

- Místo objektu A můžeme použit objekt B, který je odvozený od A
- napr. Tigr (B) je taky kočka (A)
- odvozená třída by neměla ztrácet schopnosti předchozí třídy

Princip oddělených rozhraní

= jedná třída může plnit spoustu rozhraní

Princip obrácení závislosti / Dependency Inversion Principle

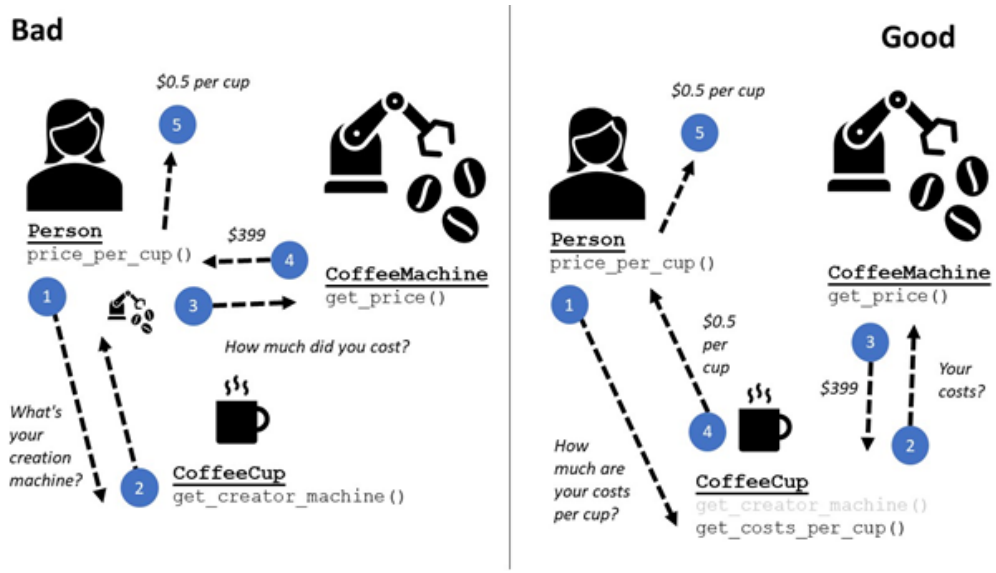
= pravidla SOLID

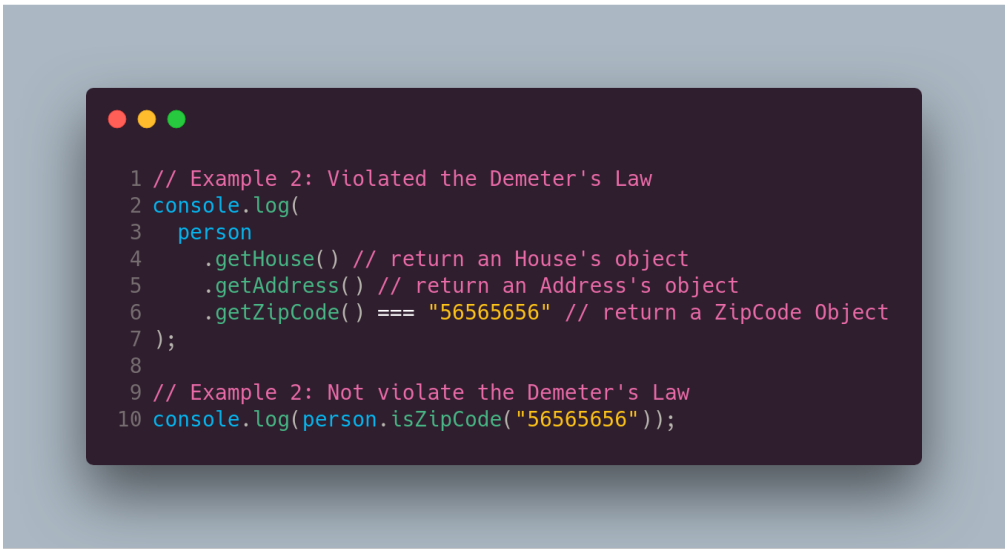
- moduly vyšší úrovně by neměly záviset na modulech nižší úrovně;
- oba by měly záviset na abstrakcích.

1. Deméteřin zákon / Law Of Demeter

- Funkce FFF() ve třídě TTT by měla volat pouze:
  - funkce třídy TTT
  - funkce objektů = předané jako parametry funkci FFF
  - funkce objektů vytvořené fci FFF
  - funkce objektů patřící třídě TTT

- globální funkce





```
// Example 2: Violated the Demeter's Law
console.log(
  person
    .getHouse() // return an House's object
    .getAddress() // return an Address's object
    .getZipCode() === "56565656" // return a ZipCode Object
);

// Example 2: Not violate the Demeter's Law
console.log(person.isZipCode("56565656"));
```

```
seznam.Vyber().Split()[0].ToItem().Name.ToString()
```

2. Dry - don't repeat yourself

= Každá informace by měla být v programu jen jednou

3. High cohesion, Low coupling

- High cohesion = Věci, které spolu souvisí, by měly být blízko
- Low coupling = věci by měly být provázané jen zlehka (přes interface)

4. YAGHI - You Ain't Gonna Need It

11. přednáška

Měření času

DateTime

=třída: a date and time of day

TimeSpan

=třída: rozdíl mezi dvěma kalendářními daty

```
DateTime date1 = new DateTime(2003, 6, 13, 15, 00, 15);
DateTime date2 = new DateTime(2024, 4, 30, 14, 22, 30);


TimeSpan interval = date2 - date1;
Console.WriteLine("{0} - {1} = {2}", date2, date1, interval.ToString())
```

Delegáty (C#)

```
public delegate int Funkce(int x, int y);

public static int Soucet(int x, int y) { return x + y; }
public static int Soucin(int x, int y) { return x * y; }
Funkce f = Soucet; //dosazujeme adresu funkce, nevoláme ji
Console.WriteLine(f(10, 20));
Funkce f = Soucin; //funkce přidá tam další funkci
Console.WriteLine(f(10, 20));
```

- funkce, která má 2 intové parametry a vrací int
- obsahuje adresu / referenci na funkci se stejným seznamem parametrů a vrácenou hodnotu, jakou má delegát
- v delegátu může být víc funkcí, provedených za sebou
- Delegáty lze předávat jako parametry funkcím
- Multicasting: f += Soucin

Anonymní funkce

- do proměnné se může dosadit funkce, která se vyrobí na místě, pomocí delegate
- = inline delegate

```
  public delegate void petanim(string pet);
  petanim p = delegate(string mypet)
    {
        Console.WriteLine("My favorite pet is: {0}",
                                                mypet);
    };
  p("Dog");
```

Lambda funkce

- při vstupu se neuvádí typy, takže univerzálnější

```
(input-parameters) => expression
(input-parameters) => { <sequence-of-statements> }

Func<int, int> square = x => x * x;
Console.WriteLine(square(5));
//25
```

Vlákna / Thread

- v počítači běží více programů "najednou"
- thready (výpočetní vlákna) běží najednou - jejich rychlost závisí na počtu jader procesoru
- když vytvoříme více vláken, než jader:
  - **Race condition** =
    - Vytvoří se více **threadů**,
    - 1. thread A = A + 10
    - 2. thread A = A + 50
  - **Thread-save** =
  - **Zámky** = když jedna funkce běží, nemůžou běžet jiné funkce / nemůžou jiné objekty něco dělat

    ```
    if (Volno) {
            nastav Nevolno
    }
    ```

  - **Deadlock**
    - potřebuju vářičku
    - a pak se vzpomenu, že potřebuju pánvičku

12. přednáška

Generické metody a třídy

- ref
- Genericke funkce = bere parametry stejného libovolného datového typu - je všeobecná (general)
- return type - name - <> (T parameters) {}
- je obnovitelná / reusable

```
static void Replace(ref int a,ref int b)
{
        int temp = a;
        a = b;
        b = temp;

}
```

```
nebo

//Genericke funkce
static void Replace<T>(ref T a,ref T b)
{
        where T: IComparable; //implements interface - returns int

        T temp = a;
        a = b;
        b = temp;
}

staic void Main(string[] args) {

        int a = 2;
        int b = 5;
        Console.WriteLine(a, b);
        Replace(ref a, ref b);


        float fa =m 2.0f;
        float fb = 7.0f;
        //Replace<float>(ref fa, ref fb)

}

//genericka trida
```

```
class KeyValuePair<TKey, TValue>
{
    public TKey Key { get; set; }
    public TValue Value { get; set; }
}
```

```
```