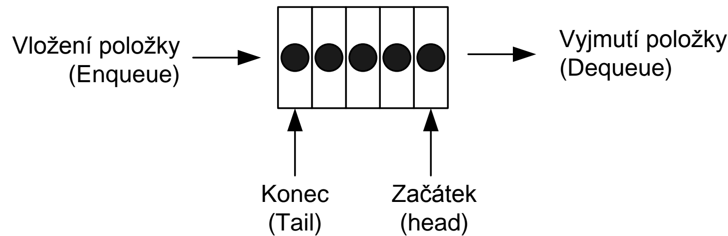
Fronta (Queue)

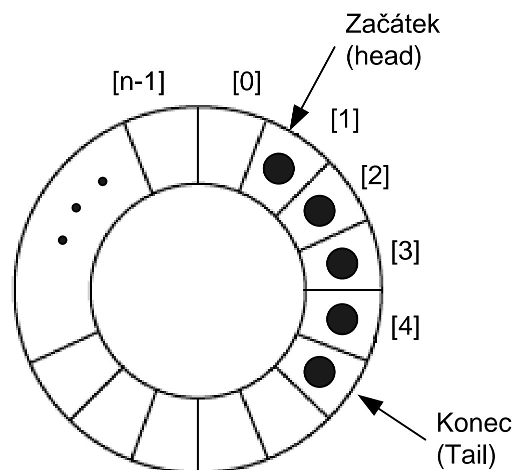
- FIFO – First In First Out (první dovnitř první ven)



- Přidávání prvků → na jedné straně (konec)
- Odebírání prvků →
  - na opačné straně (začátek)
  - ve stejném pořadí jako byly vloženy
- použití: Vyrovnávací paměť dvou spolupracujících objektů
  - dva nezávislé objekty → díky dvěma ukazatelům
  - první objekt plní frontu, druhý prvky odebírá

Implementace

- Přes pole – nutno pamatovat konec i začátek
  - Většinou nelze „běžně" – nutná tzv. kruhová fronta (cyclic queue)
    - pole `Q[0,1,…,n-1]`, po `Q[n-1]` následuje `Q[0]`



- Spojový (dopředný) seznam – neomezená délka

Prioritní fronta

- fronta s předbíháním
- každý prvek → různá priorita
- přednostně jsou odebírány prvky s vyšší prioritou → nutno projít celou frontu

Fronta v BCL .NET

- Třída `Queue<T>` = cyklická fronta, data v poli, schopnost automaticky růst („ale" jako u `Stack`)
- Konstruktory:

  - `Queue()` – vytvoří frontu o kapacitě 4 prvky
  - `Queue(int capacity)` – vytvoří frontu o kapacitě `capacity` prvků
- Vkládání:
  - `void Enqueue(T item)` – vloží `item` na konec fronty (plná fronta = autozvětšení)
- Vyjímání:
  - `T Dequeue()` – vrací prvek ze začátku fronty
  - `T Peek()` – jako `Dequeue()`, ale prvek ve frontě zůstává
  - `void Clear()` – vymaže všechny prvky z fronty (kapacita se nemění)
- Převody na pole:
  - Jako u `Stack`
- Vlastnosti jen pro čtení:
  - `Int Count` – počet prvků ve frontě (ne kapacita!)
- Příklad použití:

```
Queue<int> q = new Queue<int>(5);
q.Enqueue(1);
q.Enqueue(2);
q.Enqueue(3);
q.Enqueue(4);
int a = q.Dequeue();
a = q.Dequeue();
q.Enqueue(31);
q.Enqueue(41);
```

Negenerické varianty

- Původně v .NET 1.0
- I v .NET 2.0 a novější – jmenný prostor `System.Collection`
- Pracují pouze s prvky typu `object`
  - vkládání:
    - Hodnotové typy → boxing
    - Odkazové typy → dědičnost (předek zastupuje potomka)
  - Vyjímání: nutno přetypovat
- Pomalejší, nebezpečnější než generické varianty
- Příklad: Zásobník typu `double`

```
Stack s = new Stack();
s.Push(10.3);                 // void Stack.Push(object obj) -> boxing
double x = (double)s.Pop();   // object Stack.Push() -> unboxing
```

Delegáty (Delegate)

- = způsob, jak prostřednictvím jedné metody spustit metodu jinou
  - Chová se jako „zástupce“ pro jinou metodu
- Příklad – delegát pro metody typu `double (double)`:

```
class Program
{
      delegate double Zastupce(double param);      // prototyp metod
```

```
        static double Metoda(double x)
        {
                return 10 * x;
        }

        static void Main(string[] args)
        {
                Zastupce d;
                d = new Zastupce(Math.Sin);      // d je ted Math.Sin()
                double prom = d(Math.PI / 2);    // = 1.0
                d = new Zastupce(Metoda);        // d je ted Metoda()
                prom = d(5);                     // = 50.0
        }
}
```

- Delegát – statut třídy, lze definice:
    - Jako člen jiné třídy (lze `public`, `private`, …, `static`) – viz příklad výše
    - Mimo třídu
- Na úrovni .NET potomek třídy `Delegate` skrytý za „syntaktický cukr“ (viz pole a `Array`)
- Požadavky na parametry – žádné:
    - Počet parametrů: 0 až ∞
    - Typy návratové hodnoty a parametrů: libovolné
- „instance delegáta se může odkazovat na libovolnou instanční či statickou metodu libovolného objektu jakékoli třídy. Jediná podmínka → hlavička metody = delegát.“
    - Příklad chyby:

```
delegate double Zastupce(double param)
static void Metoda(string text)
{
        Console.WriteLine(text);
}
Zastupce = new Zastupce(Metoda);
// ERROR: No overload for 'Metoda' matches delegate 'Zastupce'
```

- Využití:
    - událostmi řízené programování (typicky GUI ve Windows)
    - paralelní programování (vlákna)
    - spouštění metod, které nejsou známy v době psaní programu (viz cvičení)
        - Bubble srt tak, aby uměla setřídit cokoli.

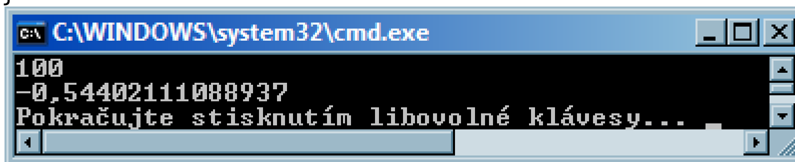
Vícenásobný delegát (Multicast Delegate)

- Jeden delegát = může být zástupce pro více metod (spouští se postupně)
    - Návratová hodnota jedině `void` (jak uložit do jedné proměnné výsledky více metod?)
    - Počet neomezen
    - Pořadí spuštění není definováno
- .NET: potomek třídy `MulticastDelegate`
- Příklad:

```
class Program
```

```
{
        delegate void Zastupce(double param);

        static void Metoda(double x)
        {
                Console.WriteLine(10 * x);
        }
        static void JinaMetoda(double x)
        {
                Console.WriteLine(Math.Sin(x));
        }
        static void Main(string[] args)
        {
                Zastupce d = new Zastupce(Metoda);
                d += new Zastupce(JinaMetoda);
                d(10);
        }
}
```



- Lze i:

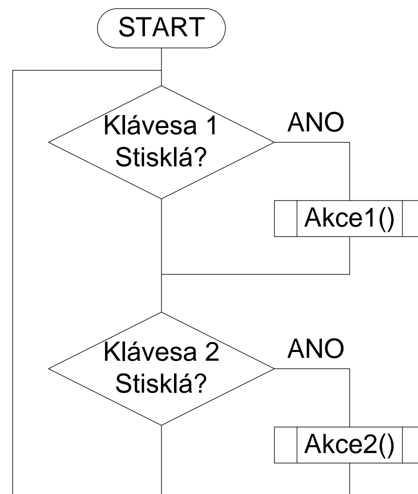
```
Zastupce d1 = new Zastupce(Metoda);
Zastupce d2 = new Zastupce(JinaMetoda);
Zastupce d = d1 + d2;
d(10);
```

- Odebrání metody z delegáta:

```
d -= JinaMetoda;
```

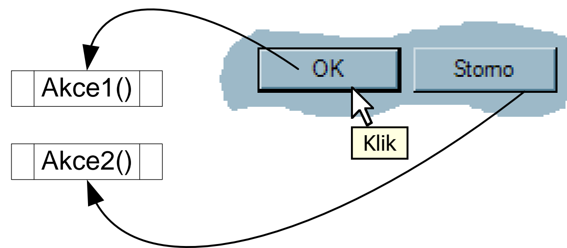
Události (Events)

- Jak napsat program, který je ovládán klávesami (tlačítky)?
  - Konzolová aplikace



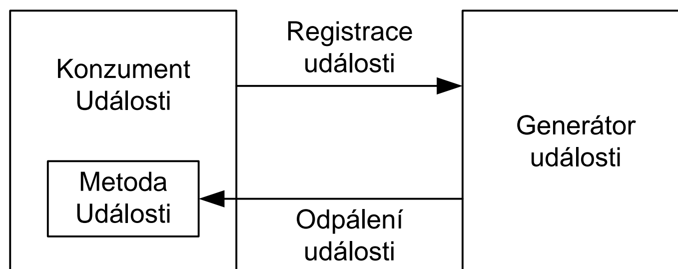
    - Nevýhoda: spotřeba 100% výkonu procesoru (OS Windows obrazně, MS-DOS skutečně)
  - GUI aplikace (OS Windows, Linux, apod.) – událostmi řízené programování (event based programming)

- Princip:



- Program nelze popsat spojitým vývojovým diagramem
- Čekání realizuje OS (0% spotřeba výkonu)
- Stisk tlačítka (= událost) → OS spustí určenou metodu (programátorem) – ošetření události

- C# → `event`
  - netýká se jen programování GUI, určeno pro komunikaci mezi objekty
  - jakákoli třída může generovat události, např. `Sklenice`: události „přetekla“, „je prázdná“ apod.
- princip a základní pojmy:
  - „až bude ráno, tak mě vzbuď"



  - konzument, generátor = objekty
  - Metoda události – event handler
  - Odpálení události – firing event
- Příklad:

```
class Trida  // generátor
{
        public delegate void PrototypHandleru();
        public event PrototypHandleru Udalost;

        private int prom;

        public void InkrementProm()
        {
                prom++;
                if (prom == 2)
                {
                        Udalost();   // odpálení
                        prom = 0;
                }
        }
}
```

```
class Program        // konzument
{
        static void Main(string[] args)
        {
                Trida objekt = new Trida();
                objekt.Udalost += Handler;        // registrace
                for (int i = 0; i < 5; i++)
                {
                        Console.WriteLine(i);
                        objekt.InkrementProm();
                }
        }
        // Event Handler
        static void Handler()
        {
                Console.WriteLine("Nastala udalost");
        }
}
```

```
C:\WINDOWS\system32\cmd.exe
0
1
Nastala udalost
2
3
Nastala udalost
4
Pokračujte stisknutím libovolné klávesy...
```

- Pokus o odpálení nezaregistrované události → výjimka, nutno v generátoru **vždy** ošetřit

```
// jinak stejné
public void InkrementProm()
{
        prom++;
        if (prom == 2)
        {
                if (Udalost != null)
                {
                        Udalost();
                }
                prom = 0;
        }
}
```

- založeno na multicast delegates:
  - registrace +=, odregistrace -=
  - návratová hodnota event handleru vždy `void`
- události mohou být řetězeny – v handleru jedné události odpálíme jinou

Události dle .NET Framework Guidelines

- http://msdn2.microsoft.com/en-us/library/w369ty8x(VS.80).aspx
- Jen doporučení (použito ve Winforms)
- delegát:

```
delegate void EventHandler(object sender, EventArgs e)
```

- `sender` – informace o generátoru (např. více generátorů je zaregistrováno na jeden handler a tam je nutné generátory rozlišit)
- `e` – argumenty události
  - žádné požadavky na argumenty – `EventArgs`
  - vlastní parametry události – potomek `EventArgs` (viz dále)
- není nutné deklarovat – v BCL je již obsažen

- předchozí příklad (jen změny)

```
class Program
{
        // Main() je stejny
        static void Handler(object sender, EventArgs e)
        {
                Console.WriteLine("Nastala udalost");
        }
}
class Trida
{
        // gelegata EventHandler nedeklarovat
        public event EventHandler Udalost;
        // v public void InkrementProm():
        if (Udalost != null)
        {
                Udalost(this, new EventArgs());
        }
}
```

- definice a použití vlastního argumentu události:
  - třída, podědit od `EventArgs`, název by měl končit na …`EventArgs`
  - nutno definovat vlastního delegáta (název by měl končit na …`Handler`)
  - upravený min. příklad:

```
class TridaEventArgs: EventArgs
{
        private string zprava;
        public string Zprava
        {
                get { return zprava; }
        }
        public TridaEventArgs(string text)
        {
                this.zprava = text;
        }
}
class Trida
{
        public delegate void TridaEventHandler(object sender,
                TridaEventArgs e);
        public event TridaEventHandler Udalost;
        // v public void InkrementProm():
        if (Udalost != null)
        {
                Udalost(this, new TridaEventArgs("Ahoj"));
        }
}
class Program
{
        // Main() je stejny
```

```
        static void Handler(object sender, TridaEventArgs e)
        {
                Console.WriteLine("Nastala udalost, zprava: {0}", e.Zprava);
        }
}
```

- nutnost definice vlastního delegáta lze obejít pomocí generické verze

```
public event EventHandler<TridaEventArgs> Udalost;
```