Kolekce, cyklus foreach

- Jen informativně
- Kolekce = seskupení prvků (objektů)
- Jednu již známe – pole (`Array`)
- Kolekce v C# = třída, která implementuje `IEnumerable` (`ICollection`)

Cyklus `foreach`

- „pro každý …“
- Určen pro procházení kolekcí – **jen pro čtení, jen celé** (pořadí určuje kolekce)
- Syntaxe:

```
foreach (typPrvku prvek in kolekce)
```

- Příklad, pole typu `int`, vytištění na konzoli – porovnání s `for`

```
int[] pole = { 1, 2, 3, 4 };
// for
for (int i = 0; i < pole.Length; i++)
{
	Console.WriteLine(pole[i]);
}
// foreach
foreach (int prvek in pole)
{
	Console.WriteLine(prvek);
}
```

`ArrayList`

- Pole s automatickým růstem velikosti
- Jmenný prostor `System.Collections`
- Konstruktor:
  - `ArrayList()` – počáteční kapacita 16
  - `ArrayList(int capacity)` – počáteční kapacita `capacity`

```
ArrayList al = new ArrayList(10);
```

- Pracuje s `object` – pomalé
- Vlastnosti:
  - `int Capacity` – velikost kolekce
- Vlastnosti jen pro čtení
  - `int Count` – aktuální počet prvků v kolekci
- Přístup přes metody:
  - `public virtual int Add(object value)`
    - přidá objekt na konec kolekce
    - pokud `Count` < `Capacity` → O(1)
    - pokud `Count` = `Capacity` (kolekce je plná – nutné autozvětšení `Capacity`) → O(`Count`) – vytvoření nového úložiště + kopie všech prvků
  - `public virtual void Insert(int index, object value)`
    - vloží `value` na pozici `index`
    - pokud `index` < `Count` → odsunutí prvků s vyšším indexem

        - pokud `index` > `Count` → `ArgumentOutOfRangeException`
    - `public virtual void Remove(object obj)`
        - pokusí se najít `obj` a vyjme jej z kolekce (nenalezen = nic se neděje)
        - porovnání objektů realizuje `Object.Equals()` (je virtuální!)
    - Další viz MSDN – obsahuje např. i `Sort()`, `BinarySearch()`, `Reverse()`, `RemoveRange()`, …
    - Příklad:

```
ArrayList al = new ArrayList(); // 16 prvků
al.Add("Malá ");
al.Add("černá ");
al.Add("kočka");          // Malá černá kočka
al.Insert(1, "bílá ");    // Malá bílá černá kočka
al.Remove("černá ");      // Malá bílá kočka
foreach (string slovo in al)
{
      Console.Write(slovo);
}
```

- Přístup jako do pole (přes indexy)
    - **Pozor** – jen v rozsahu ⟨0; `Count`⟩, jinak `ArgumentOutOfRangeException`

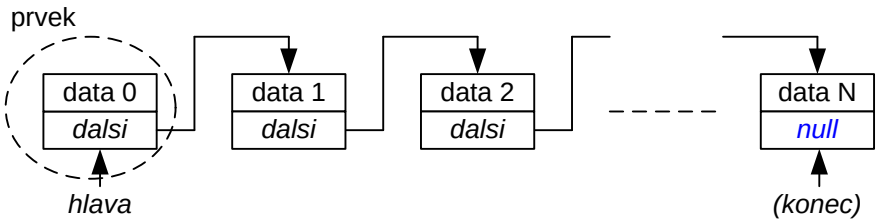
```
al[0] = Velká ";              // přepíše prvek!!!
string text = (string)al[2]; // přetypování nutné
```

Datové struktury

- datová struktura = „kontejner“ pro uložení informací, obsahuje
    - data
    - algoritmy (operace nad daty)
- dělení
    - základní (obvykle v každém VPJ) – proměnná, pole, objekt
    - odvozené – zásobník, fronta, seznam, strom, slovník, graf…
        - používají základní datové struktury
        - Starší jazyky nemají, moderní ano (C# v BCL)

Spojový seznam

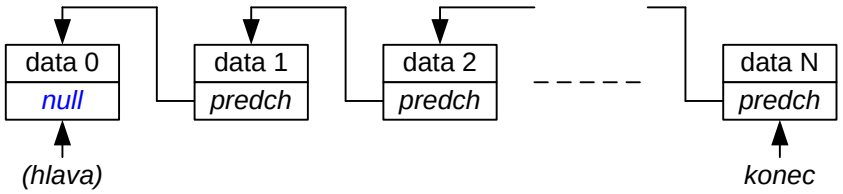
- (Linked list)
- pro data neomezené délky
- varianty:
    - dopředně vázaný



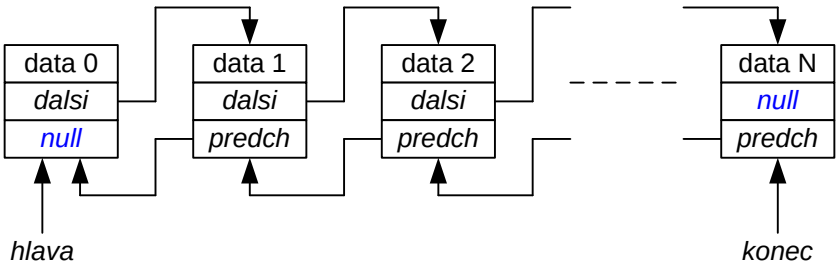
Odkazy (referenční proměnné)

        - nutno pamatovat: ukazatel na první prvek (hlava)

    - lze procházet pouze dopředu
  - zpětně vázaný



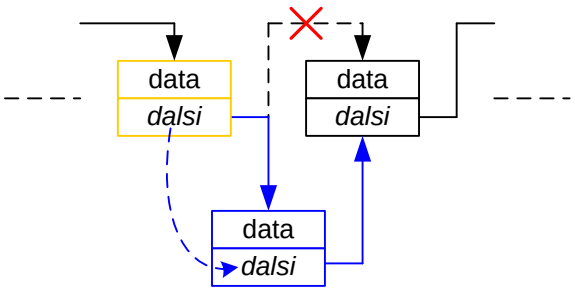
    - nutno pamatovat: ukazatel na poslední prvek (konec, pata, ocas)
    - lze procházet pouze odzadu
  - obousměrně vázaný (doubly linked list)



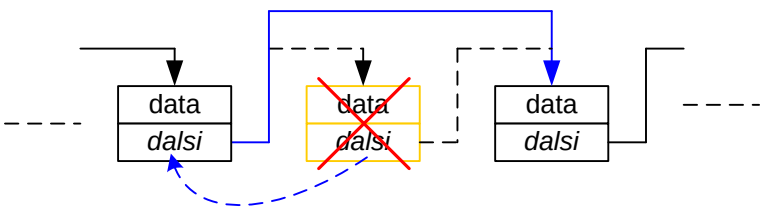
    - průchod oběma směry
- nevýhoda: sekvenční přístup

Základní operace (dopředně vázaný seznam)

- přidat na konec
- vložit dovnitř seznamu
  - vstup: reference na prvek, za (před) který se bude vkládat



- vyhledat hodnotu
  - návratová hodnota operace = reference na vyhledaný prvek
- smazat poslední prvek
- vyjmout zevnitř seznamu
  - vstup: reference na mazaný prvek



- smazat celý seznam

Implementace v C#

- příklad dopředně vázaného seznamu prvků typu `int`
- prvek seznamu (node)

```
class Prvek
{
	public int hodnota;
	public Prvek dalsi = null;	// odkaz na sebe samu (typ)
}
```

- vlastní seznam

```
class Seznam
{
	Prvek hlava = null;
	Prvek konec; // při přidávání nemusíme procházet seznam od začátku

	public void PridejNaKonec(int hodnota)
	{
		Prvek prvek = new Prvek();
		prvek.hodnota = hodnota;
		if (hlava == null) // = prazdny seznam
		{
			hlava = konec = prvek;
			return;
		}
		konec.dalsi = prvek;	// predchazeji prvek spojime s novym
		konec = prvek;		// novy prvek = novy konec
	}
	// dalsi operace viz cviceni
}
```

- v Main:

```
Seznam S = new Seznam();
S.PridejNaKonec(5);
S.PridejNaKonec(10);
```

- Generická implementace:

```
class Prvek<T>
{
	public T hodnota;
	public Prvek<T> dalsi = null;	// odkaz na sebe samu (typ)
}
class Seznam<T>
{
	Prvek<T> hlava = null;
	Prvek<T> konec;

	public void PridejNaKonec(T hodnota)
	{
		Prvek<T> prvek = new Prvek<T>();
		// dale stejne
	}
}
// v Main:
Seznam<int> S = new Seznam<int>();
S.PridejNaKonec(5);
```

Seznam v BCL

- Ve jmenném prostoru `System.Collections.Generic` (platí i pro další datové struktury)
- `LinkedList<T>` –obousměrný spoj. seznam, prvky = `LinkedListNode<T>`
- Vkládání:
  - `void AddFirst(T value)`
    - Na začátek vloží `value`
  - `void AddLast(T value)`
    - Na konec vloží `value`
  - `void AddAfter(LinkedListNode<T> node, T value)`
    - Za prvek `node` vloží `value`
  - `void AddBefore(LinkedListNode<T> node, T value)`
    - Před prvek `node` vloží `value`
  - Kromě `T value` lze vkládat i `LinkedListNode<T> node` (přetížení)
- Mazání:
  - `bool Remove(T value)`
    - vyjme první výskyt `value`
    - vrací `true` prvek nalezen a vyjmut, jinak `false`
    - přetížena i pro `LinkedListNode<T> node`
  - `void RemoveFirst()`
    - vyjme první prvek
  - `void RemoveLast()`
    - vyjme poslední prvek
  - `void Clear()`
    - vymaže všechny prvky ze seznamu
- Vyhledávání:
  - `bool Contains(T value)`
    - vrací `true` pokud seznam obsahuje `value`
  - `LinkedListNode<T> Find(T value)`
    - Vyhledá první výskyt `value` (hledá zepředu)
    - Vrací: odkaz na prvek, pokud nenalezeno – `null`
  - `LinkedListNode<T> FindLast(T value)`
    - Jako `Find()`, ale hledá odzadu
- Informace:
  - Vlastnosti jen pro čtení
  - `int Count` – aktuální počet prvků v seznamu
  - `LinkedListNode<T> First` – odkaz na první prvek
  - `LinkedListNode<T> Last` – odkaz na poslední prvek
- Příklad použití:

```
LinkedList<string> veta = new LinkedList<string>();
veta.AddLast("Dobrý ");
veta.AddLast("večer");
veta.AddLast(", mami");
veta.Remove("večer");
LinkedListNode<string> node = veta.Find("Dobrý ");
// chybí kontrola node na null!
```

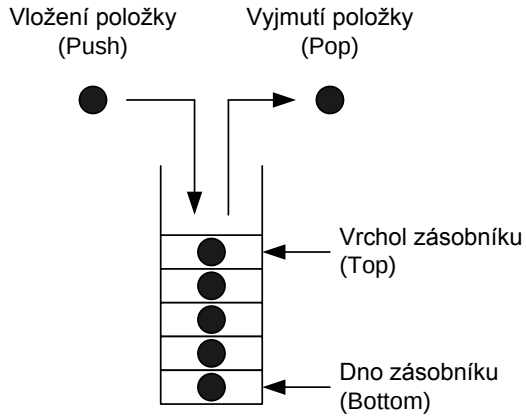
```
veta.AddAfter(node, "den");
```

- Tvoří zároveň kolekci (implementuje `ICollection<T>` a další)

```
foreach (string slovo in veta)
{
        Console.Write(slovo);      // Dobrý den, mami
}
```

Zásobník (Stack)

- LIFO – Last In First Out (poslední dovnitř, první ven)



- Přidávání prvků → na vrchol
- Odebírání →
  - v opačném pořadí než byly vloženy
  - z vrcholu (znalost pozice → i zvnitřku)
- Použití: Přechodné úložiště, systémové programování (mikroprocesory)
- Implementace
  - Přes pole
  - zpětně vázaný spojový seznam – hlava seznamu = vrchol
- Chybové stavy
  - Přetečení zásobníku (stack overflow) = push() do plného zásobníku
  - Podtečení zásobníku (stack underflow) = pop() z prázdného zásobníku
- Využití
  - v architektuře počítače
  - překladač VPJ
  - rekurze

Příklad implementace

- zásobník přes pole – nutno pamatovat vrchol

```
class Zasobnik<T>
{
        T[] data;
        int vrchol = 0;     // ukazuje na prvni volny prvek

        public Zasobnik(int kapacita)
        {
                data = new T[kapacita];
        }
```

```
        public void Push(T co)
        {
                if (vrchol >= data.Length)
                        throw new InvalidOperationException("Stack Overflow");
                data[vrchol++] = co;
        }

        public T Pop()
        {
                if (vrchol <= 0)
                        throw new InvalidOperationException("Stack Underflow");
                return data[--vrchol];
        }
}
// v Main:
Zasobnik<int> z = new Zasobnik<int>(5);
z.Push(-10);
z.Push(55);
int prvek = z.Pop();        // 55
z.Push(1234);       // z: -10, 1243
```

Zásobník v BCL

- Třída `Stack<T>`
  - Může měnit svoji kapacitu – rozšiřování automaticky, zmenšování ručně (oboje časově náročné!!! – O(n), viz `ArrayList`)
- Konstruktory:
  - `Stack()` – vytvoří zásobník o kapacitě 4 prvky
  - `Stack(int capacity)` – vytvoří zásobník o kapacitě `capacity` prvků
- Vkládání:
  - `void Push(T item)` – vloží `item` na vrchol zásobníku (plný zásobník = autozvětšení)
- Vyjímání:
  - `T Pop()` – vrací prvek z vrcholu zásobníku
  - `T Peek()` – jako `Pop()`, ale prvek v zásobníku zůstává
  - `void Clear()` – vymaže všechny prvky ze zásobníku (kapacita se nemění)
- Převody na pole:
  - `T[] ToArray()` – vytvoří nové pole a zkopíruje do něj obsah zásobníku
  - `void CopyTo(T[] array, int arrayIndex)` – zkopíruje obsah zásobníku do již existujícího pole `array` od indexu `arrayIndex`
- Vlastnosti jen pro čtení:
  - `Int Count` – počet prvků v zásobníku (ne kapacita!)
- Příklad použití:

```
Stack<int> S = new Stack<int>(10);
S.Push(10);
S.Push(22);
S.Push(-33);
S.Pop();
S.Push(-1256);
int posledni = S.Peek();  // -1256
```

```
// zasobnik jako kolekce
foreach (int prvek in S)
{
	Console.WriteLine(prvek); // 10, 22, -1256
}
// prevod na pole
int[] pole = S.ToArray();	// pole: 10, 22, -1256
```