# **ČÁST**

## PROMĚNNÉ, DATOVÉ TYPY…

**Proměnná**

Vyhrazené místo v paměti, které si nějak pojmenujeme a můžeme do něj ukládat a číst data. Každá proměnná má datový typ.

Proměnné rozlišujeme na ***lokální proměnné*** a ***členské proměnné***.

*Lokální proměnná* se deklaruje vždy v nějaké metodě, kde ji zrovna potřebujeme.

***Lokální proměnná*** se dá používat pouze v té metodě, kde byla nadeklarována. Proměnná se vytvoří vždy, když je metoda zavolána. Platnost této proměnné končí po provedení všech příkazů v metodě, jakmile tedy metoda skončí, proměnná zanikne.

private void Func()

{

int local;

}

***Členská proměnná***musí být deklarována uvnitř třídy nebo struktury. Proměnná bude platná po celou dobu kdy je třída využívána. Zanikne společně se třídou.

class Test

{

int variable;

private void Func()

{

}

}

**Datový typ**

Protože je C# tvrdě psaný jazyk, musíme při každé deklaraci proměnné informovat kompilátor o tom, jaký datový typ chceme v dané situaci použít.

**Druhy datových typů:**

**Hodnotové X Referenční**

Hodnotové datové typy jsou obecně **jednoduché struktury** – jedno číslo, jeden znak. Pracuje se s nimi **rychle** a v programu se jich objevuje hodně a **zabírají málo místa**. Mají **pevnou velikost**. Hodnotové datové typy se ukládají do Stacku (zásobníku). Stack je velmi rychlá pamět s přímým přístupem. Její prostředky jsou přidělovány operačním systémem.

**Hodnotové**

*Celočíselné* – int (**4 byte**), short (2 byte), long (8 byte), uint, ushort, ulong

int, sbyte xxx – I kladné i záporné

Datové typy začínající na u neumožňují ukládat záporné hodnoty, tím pádem na kladnou část čísla můžou uložit 2x vyšší hodnotu. Říká se jím Unsigned.

C# kód chápe:

int a = 10;

jako:

System.Int32 a = 10;

V paměti se tento INT uložil takto:

|  |  |
| --- | --- |
| **STACK (Zásobník)** | **HEAP (Halda)** |
| int a (10) |  |

*Desetinné* – float (4 byte), double (8 byte), decimal (16 byte)

Double je 2x přesnější než float.

„Desetinný separátor” ve zdrojovém kódu je vždy tečka, nehledě na to, jaké máme regionální nastavení ve Windows

Decimal se používá k uchovávání peněžních hodnot, protože ukládá desetinné číslo vnitřně podobně jako text

*Textové* – char (2 byte) –

Znaky do charu ukládáme do apostrofů ‘ ’

podle ASCI tabulky

Console.ReadKey() vrací právě hodnotu char

print char vrátí číslo symbolu v ASCI tabulce

Znakové sady – např. čínština, japonština

*Boolean* – můžu ukládat true nebo false (1 byte)

Do boolu můžeme ukládat jak hodnoty true/false, nebo logický výraz, např:

bool b = false;

bool logic = (15 > 5);

* False
* True

Bool se používá u logických operátorů a podmínky if.

**Referenční**

Referenční datové typy se v paměti ukládají dvakrát. Jednou v Stacku a jednou v Heapu. V Stacku se vytvoří reference, odkaz do Heapu, kde se nachází opravdový objekt.

Referenční datové typy ukládáme do STACKu (zásobníku) a HEAPu (Haldy). Je to z toho důvodu, že Stack má omezenou velikost. Proto složitější datové typy ukládáme do Heapu, který je sice pomalejší a přístup do něj je složitější, ale má prakticky neomezenou velikost.

Např.:

class User

{

public int age;

public string name;

public string surname;

public User(string name, string surname, int age)

{

this.name = name;

this.surname = surname;

this.age = age;

}

public override string ToString()

{

return name;

}

}

class Program

{

static void Main(string[] args)

{

int a = 10;

User u = new User("Jan", "Novák", 20);

}

}

V paměti to vypadá takto:

|  |  |
| --- | --- |
| **STACK (Zásobník)** | **HEAP (Halda)** |
| int a (10) |  |
| User u = Reference | Jan Novák, 20 |

Proměnná u (datového typu objektu User) je referenčního datového typu. Tato proměnná ukazuje na složitější objekt než hodnotový datový typ.

class Program

{

static void Main(string[] args)

{

int a = 10;

int b = 25;

User u = new User("Jan", "Novák", 20);

User v = new User("Adam", "Novotný", 30);

a = b;

u = v;

}

}

Nastavíme hodnotu proměnné a na b a hodnotu proměnné u na v, v paměti to vypadá takto:

|  |  |
| --- | --- |
| **STACK (Zásobník)** | **HEAP (Halda)** |
| int a (25) |  |
| int b (25) |  |
| User u = Reference | Jan Novák, 20 |
| User v = Reference | Adam Novotný, 30 |

Hodnotový typ v Stacku se jen zkopíruje, ale u objektu se zkopíruje pouze reference, ne samotný objekt.

Proměnná u je nyní referenčního typu.

Můžeme na jedno místo v paměti uložit několik proměnných - pole

Int[] arr = {10, 20, 30};

***string je pole charů***

string s = “ahoj” (píše se do “ ”) – zkratka pro pole charu

Referenční datové typy, na rozdíl od hodnotových, můžou nabývat speciální hodnoty null. Slovo null znamená, že reference neukazuje na žádná data.

**Přímý kód**

Znaménkový byt znázorňuje kladnost či zápor čísla – 0 je plus, 1 je minus.

Musíme vědět kolikabitový je kód, např.

1 + 3 bity (1 | 101 (2) = -5 (10))

1 + 4 bity (0 | 1101 (2) = +13 (10))

*Odčítání v přímém kódu:*

26 – 9 = 17

26 (10) = 11010 (2)

9 (10) = 1001 (2)

11010 – 01001 = 10001 (2)

Při operaci 0 – 1 se vypůjčí 1 zleva

**Doplňkový kód**

-3 pro 1+4bit:

0 | 0011 = 3

1 | 1100 = inverze

1 | 1101 = +1 => -3

*Přetečení:*

Situace, kdy je snaha o přenos do neexistujícího bitu (nestačí bity/řády) –

maximum + 1 = minimum

15 + 1 pro 1+4bit:

0 | 1111 = 15

0 | 1111 + 1 = 1 | 0000

1 | 0000 = -16

6 – 3

Odčítání obejdeme sčítáním:

6 + (-3) = 3

0110 (6)

0011 (3) -> 1101 (-3 = 13)

0110 (6)

+ 1101 (-3)

* 1 ... 0011 = 3

1 je za hranicí datového typu, proto se tato jednička zahodí a zbyde 0011 = 3

Pokud zajdeme za hranici řádů, dojde k přetečení.

**Desetinná čísla**

0010 | 1110

2 | 14 (DK = -2)

2 \* 10-2

=> 0.02

**ByVal X ByRef**

Když předáváme proměnné do parametrů funkce, máme na výběr ze dvou možností.

ByVal je defaultní. ByVal vytvoří kopii proměnné. Pokud tuto předanou proměnnou změníme například uvnitř funkce, tak se to nijak originální proměnné nedotkne.

ByRef nevytváří kopii, ale přímo referenci na původní proměnnou.

Modifikátor – public, private, protected, internal (výchozí)

**Enum**

Pro vytvoření konstant.

enum Level

{

Low,

Medium,

High

}

public void GetLevel()

{

Level lvl = Level.Medium;

Console.WriteLine(lvl);

}

**Operátory**

*Matematické / aritmetické*

a + b ; a - b ; a \* b ; a / b ; a % b (modulo, dělení se zbytkem)

*Relační / porovnávací*

a < b ; a > b ; a <= b ; a >= b ; a ==b ; a != b

*Logické*

&& (logické AND) ; || (logické OR) ; ! (logická negace)

*Bitové*

& (bitové AND) ; | (bitové OR) ; ^ (bitové Exclusive OR) ; << nebo >> (bitové posuny)

**Pravdivostní tabulky**

*AND*

4 & 5 = 4

0100

0101

0100 = 4

*OR*

4 | 2 = 6

0100

0010

0110 = 6

*EXCLUSIVE OR*

2^3 = 1

0010

0011

*0001* = 1

*Bitové posuny*

(násobení nebo dělení základem soustavy – např. 2 nebo 10)

Rychlejší pro PC.

3 << 2

3 \* 2 \* 2 = 12

011 -> 01100

Inkrementace ++

Dekrementace - -

++a : Nejprve je zvýšena hodnota proměnné a teprve poté je použíta v příslušném výrazu.

b-- : Nejprve je vyhodnocení výrazu - výraz je vyhodnocen s původní hodnotou proměnné a teprve poté je snížen.

a++ sufix

++a prefix

*Přetížení operátorů*

Přetížení operátoru –

Využívá se k aritmetickým operacím s nadřazenými datovými typy.

Např. Můžeme sečíst dvě vlastnosti dvou objektů, dostaneme výsledek.

Box box = new Box { length = 3 } + new Box { length = 4 } => Console.WriteLine(box.length) => 7

## PROCEDURY, FUNKCE

Procedury, funkce, nebo metody slouží k přehlednosti kódu, zaručují snadnější modifikaci, nebo rozšiřitelnost programu.

*Procedura* nevrací hodnotu – **VOID**.

*Funkce* vrací hodnotu.

*Metoda* vrací hodnotu, ale je součástí objektu.

Parametry funkce můžeme ovlivňovat chování procedury.

static int Dvojmoc(int n)

{

return n \* n;

}

static int Dvojmoc (int n) – hlavička metody, z ní se můžeme dozvědět, že název metody je Dvojmoc, je statická, vrací výsledek typu int, při volání jí musíme předat jeden parametr n.

static void Main(string[] args)

{

Console.Write("Vyber si číslo: ");

int n = Convert.ToInt32(Console.ReadLine());

Console.WriteLine("Druhá mocnina čísla {0} je {1}", n, Dvojmoc(n));

Console.ReadLine();

}

Funkce a procedury můžeme předčasně ukončit returnem a vyjímkou.

*Statická metoda*

U nestatických metod musíme vytvářet instanci objektu, kterému patří. U statických instanci můžeme vynechat a rovnou volat metodu jménem.

Jedná se zejména o pomocné metody, které potřebujeme často používat a nevyplatí se nám tvořit instanci.

Např. metoda WriteLine() na třídě Console je statická. To samé u Round() na třídě Math.

*Statická metoda v kódu*:

public **static** int StatickaMetoda(int a, int b)

{

return a + b;

}

Program, nemusíme vytvářet instanci objektu, do nějž patří metoda:

Console.WriteLine(StatickaMetoda(2, 3));

*Nestatická metoda v kódu*:

public int NeStaticka(int a, int b)

{

return a + b;

}

Program – musíme vytvořit instanci objektu, do kterého metoda patří:

Program pr = new Program();

Console.WriteLine(pr.NeStaticka(x, y));

*Přetížená funkce*

Dvě funkce můžou mít jeden název, různé return typy a různý počet parametrů s různými typy parametrů.

Volitelný parametr (musí být poslední)

Static int xxx(int a = 3) -> v konstruktoru volame bez parametru xxx();

static int Foo2(params int[] pole)

{

int result = 0;

foreach (int item in pole)

{

result += item;

}

return result;

}

Console.WriteLine(Foo2(1,2,3));

Params umožňuje rovnou do parametru psát hodnoty v poli, které budou odděleny čárkou

Params – pouze jednou, musí být poslední

Finally {} se provede po každé – i když to spadne do podmínky

Ref, out – předávání proměnných referencí

*Ref – oboustranné předávání proměnných*

public static void Plus(ref int inside)

{

inside = inside + 10;

}

Program:

int outside = 20;

Plus(ref outside);

Console.WriteLine(outside);

* 30

*Out – jednostranné předávání proměnných*

Jakékoliv údaje, které předáme metodě při volání metody se zahodí a nahradí tím, co vrátí metoda

U out musíme přiřadit hodnotu všem proměnným v parametru

public static void Plus(out int inside)

{

inside = 0;

inside = inside + 10;

}

Program:

int outside = 20;

Plus(out outside);

Console.WriteLine(outside);

* 10

*Defaultně se proměnné předávají hodnotou. Příklad:*

public static void Plus(int inside)

{

inside = inside + 10;

}

Program:

int outside = 20;

Plus(outside);

Console.WriteLine(outside);

* 20

**Garbage Collector**

Stack se umí vyčistit sám, ale Heap to neumí, proto máme Garbage Collector.

Memory Leak – situace, ke které nesmí dojít (např. u Reference Counter)

Situace, kdy se Heap postupně přeplnňuje a časem dojde k tomu, že tam nemůžeme vytvářet nové objekty

Situace, kdy programátor neuvolní místo v paměti, které už nevyužívá.

Dangling Pointer – reference, která se pojí na objekt v Heapu, který už neexistuje, nebo který byl přesunut

V C# nemůže memory leak vzniknout, v C++ však ano

Garbage Collector odstraní všechny objekty, které mají reference counter na 0

Reference Counter označuje počet referencí na nějaký objekt

Mark and Sweep

1. fáze – označování – označí všechny objekty, ke kterým se můžeme dostat
2. fáze – likvidace – odstraní všechny objekty, které nebyly označeny v první fázi

Pokud máme dva objekty v cyklu a nemáme se k nim jak dostat, tak budou likvidovány

Paměť se dělí na generace podle toho, jak často se na daném místě čistí.

**Řídící struktury**

Podmínky - if, switch

Cyklus – while (for a foreach jsou “vychytávky”)

for (int a = 0; b < 5; a++)

{}

int a = 0 - je kód před cyklem

b < 5 – je podmínka cyklu

a++ - poslední řádek cyklu

while (true) = for (;;) = nekonečné cykly

Continue – přeskočí krok

Break – úplně vyruší cyklus

Foreach – k procházení kolekcí

List <int> data = new List<int>() {1, 2, 3, ,4 ,5}

Foreach (int item in data)

{

Console.WriteLine(item);

}

Throw – vyhodí vyjímku

Exception ex = new Exception();

Throw new Exception(“zprava”)

NullReferenceException – když je objekt nulový (např. chybí instance)

Index(Argument)OutOfRange – když voláme číslo za hranicí pole, kolekce

IOException – když havarují soubory

**Pole**

Datová struktura, která slouží k ukládání několik hodnot jednoho datového typu

int[] pole = new int[10 {hodnoty}];

V paměti jsou ukládány jako několik proměnných vedle sebe (v Heapu)

K jednotlivým proměnným přistupujeme indexem – pole[0]

pole.Length – zjištění délky pole

P[3]

#5000 (místo v paměti Heap) + 4 (velikost INT) \* 3 (index 3)

int[,] pole2 = new int[5, 10];

5 sloupců, 10 řádků

int[,] pole2 = new int[5, 10];

, určuje dimenzi, může jich být několik

Přistupujeme k poli [2, 3, 4]

P[1, 1, 2]

1 \* (3\*4) + 1\*4 + 2

**Kolekce**

List

Kolekce nemá pevnou velikost jako pole.

Collection.Count();

Sklepávání

Add -

RemoveAt -

Clear – nastaví délku pole na 0

Insert –

DU

Add(value)

Get(index) -> value

Set(index, value)

Count()

RemoveAt(index)

Insert(index, value)

Clear()

Remove(value)

RemoveAll(value)

Throw new Exception(); - pokud máme nedodělanou část kódu, kterou momentálně nechceme řešit

Array je rychlejší než kolekce.

Kolekce může měnit velikost, narozdíl od array.

Kolekce má předpřipravené metody.

Array je homogenní, ale kolekce je heterogenní.

**Dictionary**

Dictionary<int, int> dic = new Dictionary<int, int>();

dic[2] = 5;

dic[10] = 1;

dic[15] = 2;

foreach (KeyValuePair<int, int> item in dic)

{

Console.WriteLine(item.Key + ":" + item.Value);

}

Dictionary<string, int> dic2 = new Dictionary<string, int>();

Console.Read();

String – pole charů

Char x = ‘a’;

Referenční, ukládá se v HEAPu

String je “immutable”, jakmile se jednou naplní, už se nedá změnit

string.PadLeft/Right(20, ‘x’)

string.Replace(‘l’, ‘x’)

string[] parts = string.Split(‘o’)

string.Substring(2, 4) – od druheho indexu oddeli string

str.Trim(‘x’) – ocisti retezec od zacatku nebo konce

Escape sequence –

\n – odentruje na konci

\t – tabulator

\r – odradkuje, ale neposune (jako u psacího stroje, přepisuje hodnoty)

\\ = \

@ vypina vsechny escape sequence

$ zapina interpolaci promennych

Console.WriteLine($”Vek: {age}, Jmeno: neni”)

V Linuxu – enter = \n

Ve Windowsu – enter = \r\n

StringBuilder = kolekce stringu

StringBuilder sb = new StringBuilder();

For (int I =0; I < 1000; i++)

{ sb.Appemd(“asdf”) }

Console.WriteLine(sb.ToString());

Split – Podle daného charu/separátoru (v parametru), rozdělí string.

Substring – Určímé počáteční index a počet znaků od počátku a tím určíme novou velikost stringu

ToUpper, ToLower – změní na velké nebo malé písmo, nepřijíma žádné argumenty

Replace – Přijímá dva argumenty – hledaný výraz a výraz, který nahradí hledaný výraz v stringu

Trim – Bez parametru – vymaže white space, můžeme do parametru napsat znaky (params charů), které se odstraní

Pad – Do parametrů se napíše počet znaků (musí být větší než velikost stringu) a o tolik znaků se posune string doprava a doleva.

Spojování stringů – buď plusem, nebo stringBuilderem (Append) – lepší, rychlejší

White Space – část stringu kde je hodnota charu rovna mezeře

**Soubory**

(using System.IO;)

StreamReader, StreamWriter

StreamWriter writer = new StreamWriter(“C:\\data.txt”);

Musíme zavírat Writer a Reader

Writer.Dispose();

Writer.Close();

Writer.Flush() – zapise udaje do souboru

Před Flushem se údaje ukládají do cache

using (StreamWriter writer = new StreamWriter("C:\\data.txt");)

{

}

S using nemusíme StreamWriter uzavírat

Interface:

public class Foo : IDisposable

{

public void Dispose()

{

throw new NotImplementedException();

}

}

Do Using muzeme dat jen to, co implementuje IDisposable

using (StreamWriter writer = new StreamWriter("C:\\data.txt", true))

true nastavuje append, append pripisuje udaje do jednoho souboru, write cely soubor prepise

Pozor na velikost souboru!

EOF – end of file

Jakmile reader narazi na EOF, prestane cist

using (StreamReader reader = new StreamReader("C:\\Users"))

FileInfo file = new FileInfo(“C:\\data.txt”);

FileInfo zjistí pomocí systémového souborového systému informaci o souboru – nazev, velikost v bytech, zda existuje, atributy

Umoznuje pracovat se soubory – vytvaret, mazat, presouvat atd

DirectoryInfo directory = new DirectoryInfo(“C:\\data.txt”);

Vraci informaci o souborech ve slozce

Delete – muzeme mazat pouze prazdne slozky, ale muzeme pouzit

Rekurzivni struktura – zacyklena struktura

Root nema parenta

Copy neexistuje

DriveInfo – informace o discích

Drivetype – typ disku

Tree

Rekurzivni funkce ve svem tele vola sama sebe

Muze se snadno zacyklit

Vim jak vyresit jednodussi ukol, napr 1! = 1 \* 1

Kdyz zjistim toto, pak uz vim jak vyjde 5! = 5 \* 4!

Kazdy algoritmus muzeme napsat rekurzivni funkci

Kazda funkce se ulozi v STACKu, kde se ukladaji promenne ktere funkce vyzaduje, nebo s nimi pocita

Stack Overflow (cca 65000 funkci)

2 4 7 5 1 -> 7 vs 2

4 7 5 1 -> 7 vs 4

7 5 1 -> 5 vs 7

5 1 -> 1 vs 5

1 -> 1

* 7

ternarni operator:

return (data[0] < x) ? data[0] : x;

=

static int Kdyz(bool cond, int a, int b)

{

if (cond)

{

return a;

}

else

{

return b;

}

}

DU, rekurzivne:

static bool Contains(List<int> data, int value)

{

foreach (int item in data)

{

if (item == value)

{

return true;

}

}

return false;

}

Prochazeni do hloubky – rekurzivni funkce se nejprve dostane do nejvetsi hloubky a potom se vynori

**Datove Struktury**

Zasobnik – Stack (lifo)

Fronta – Queue (fifo)

Stack ma 3 metody –

Push(value) – hodi hodnotu na Stack

Pop() -> value – likviduje horni hodnotu v Stacku a vrati ji

Peek -> value – vrati horni hodnotu, ale nelikviduje, pouze “nahlidne”

LIFO

Stack je pole v pameti

class MyStack

{

private int[] array;

private int count = 0;

public MyStack()

{

array = new int[4];

}

public void Push(int value)

{

array[count++] = value;

}

public int Pop()

{

return array[--count];

}

public int Peek()

{

return array[count - 1];

}

public void Empty()

{

if (top == 0)

throw new IndexOutOfRangeException();

}

public void Full()

{

if (top == arr.Length) {

int[] arr2 = new int[arr.Length \* 2];

for (int i = 0; i < arr.Length; i++) {

arr2[i] = arr[i];

}

arr = arr2;

}

}

}

Queue – ma “dve diry”,

Enqueue - zafrontit

Dequeue - vyfrontit

Peek – nahlednuti

Push [t]; t++]

Peek -> [t-1]

Pop t--; -> [t]

Empty t==0

Full t==Length

DU

Full – Resize pole nebo exception

Fixed – ex, Resizable – Resize

Abstract

Public je dostupnej vsude

Private pouze uvnitr funkce

Protected take pouze uvnitr funkce, ale umi dedit

Public virtual void X – virtual zarucuje, ze ji muzeme prekryt (public override void X)

Abstraktni tridy (public abstract class Foo) – nemuzeme udelat instance tridy Foo f = new Foo nepujde

Abstraktni metoda nema telo

Kvuli tomu, ze nema telo, nemuzeme delat instance

Abstraktni metoda nemuze byt privatni

Equals – pro porovnani objektu na zaklade hodnot – porovnava hash code

GetHashCode – objekt ma vratit

GetType -

ToString – defaultne se vole – zobrazuje

**Queue**

3 2 1

Enqueue – posune hodnotu a snizi end

Dequeue – odstrani hodnotu na endu

Peek – vrati hodnotu na startu

E: p[e] = value

e++ (vlozime hodnotu na end a zvetsime end)

D: -> p[s]

s++ (vratime hodnotu start a zvetsime start)

Empty: s == e

Full: (e + 1) %= s

class MyQueue

{

private int[] array;

private int start = 0;

private int count = 0;

public MyQueue()

{

array = new int[4];

}

public void Enqueue(int value)

{

Full();

array[(start + count++) % array.Length] = value;

}

public int Dequeue()

{

Empty();

count--;

return array[start++ % array.Length];

}

public int Peek()

{

return array[start];

}

public void Full()

{

if (start + count == array.Length)

throw new IndexOutOfRangeException();

}

public void Empty()

{

if (count == 0)

throw new IndexOutOfRangeException();

}

}

“Obetujem” misto v pameti pred startem

(e+1) % Length

Pole:

| 6 | ? | 3 | 4 | 5 |

S

(s+c) % Length

Queue = p(start+count) % Length = value; count++

Dequeue = p[start], start++, count--

Peek p[start]

Empty: c == 0

Full: c == Length

LINKED LIST

class LinkedList

{

private Item first = null;

public void Add(int value)

{

Item n = new Item { value = value };

if (first == null) {

first = n;

}

else {

while (first.next != null) {

Item item = first;

while (item.next != null) {

item = item.next;

}

item.next = n;

}

}

}

public void RemoveAt(int index)

{

if (index == 0) {

first = first.next;

}

else {

Item item = GetItem(index - 1);

item.next = item.next.next;

}

}

public void Insertion(int index, int value)

{

Item n = new Item() { value = value };

if (index == 0) {

n.next = first;

first = n;

}

else {

Item item = GetItem(index - 1);

n.next = item.next;

item.next = n;

}

}

public void Clear()

{

first = null;

}

public Item GetItem(int index)

{

Item item = first;

for (int i = 0; i < index; i++) {

item = item.next;

}

return item;

}

}

class Item

{

public int value;

public Item next;

}

Push:

n.next = t

t = n

Peek:

<- t

Pop:

<- t

t = t.next

Enqueue:

e.next = n

e = n

Peek:

<- s

Dequeue:

<- s

s = s.next

**Generické metody, třídy**

Pole T (novy datovy typ)

Za nazev tridy - <T>

Nested Class – vnitrni trida

**OOP**

**Zapouzdření** – schopnost objektu schovat své vlastnosti a metody před okolím

Modifikátory přístupu – public, private, protected, internal

Public – je dostupné všude

Private – je dostupné pouze this (uvnitř trídy), nedědí se, nemůže být abstraktní

Protected – z vnějšku se tváří jako private, ale můžeme ji dědit

Internal – public, ale v rámci projektu

Internal je výchozí přístupový modifikátor

Dynamic Linked Library - .dll

Modul, který můžeme linkovat na několik aplikací

.Net Framework – sada kódu, díky kterým jsou naše aplikace velikostně menší

Knihovny, které nám usnadňují práci. Můžeme je připojit k projektu (References)

Konstruktor

Speciální metoda, která se volá, jakmile se vytváří nová instance

() značí, že se volá konstruktor

Neexistuje třída bez konstruktoru, i když tam není napsán

Parametry – v závorce

: this() za konstruktor

Zavolá se na prvním místě bezparametrický konstruktor

Při dědění:

Na prvním místě se zavolá rodičovský konstruktor

: base(parametry) – volá rodiče

public class Foo

{

public Foo(int a)

{

}

}

public class Bar : Foo

{

public Bar(int a) : base(a)

{

}

}

Bar předá a rodičovskému konstruktoru Foo

**Dědičnost**

Když dědíme snažíme se o to abychom neměli duplikovaný kód (Eliminate redundant code)

Všechny třídy dědí z Objectu

Dědí tyto metody: Equals, HashCode, ToString, GetType

**Překrytí třídy**

Přepsání metody, můžeme z jedné metody udělat jinou metodu

Override

**Polyformismus**

Poly známená Many

Morphism známená Form

Můžeme změnit nepěkné switch/case statements

Můžeme zavolat potomka

Foo f = new Bar();

Díky polyformismu, můžeme vyměnit kód za běhu

Polymorfismus funguje pouze s virtual a override

**Abstraktní metody**

Třída může mít abstraktního potomka

**Rozhraní / Interface**

Konvence: do názvu I, IFoo např

Sada veřejných metod

public interface IFoo

{

void XXX();

}

XXX implementujeme ve třídě, musíme ji implementovat

Každá třída může mít pouze jednoho rodiče, může dědit pouze od jedné třídy

Ale můžeme dědit nekonečno interfaců

Interface může dědit od několika interfaců, protože nemají těla

Polymorfismus – možnost dosadit na libovolné místo, kde se vyskytuje datový typ rodiče, instance potomka

Třída s privátním konstruktorem nejde zavolat

**Statika**

Static int – metodu volame tridou

Private static void Hello – Hello je ulozena ve tride

Nemusíme dělat instanci statické třídy

Statická třída má jenom statické metody a je celá statická

Statická metoda se volá jako první

Např. Math

**Tovární metoda**

Do instance dáme třídu GetInstance, protože konstruktor je např. privátní

Do GetInstance můžeme dát nějaké podmínky, které se budou volat při volání třídy

**Jedináček – Singleton**

(Lazy implementace)

Privátní konstruktor nutností

Privátní statická proměnná datového typu třídy

Tovární metoda, s if, která kontroluje jestli proměnná je null a vytvoří jí do proměnné

(== porovnává pointery v paměti)

Nebo

(Early implementace)

Public static readonly Bar Instance new Bar()

Private Bar() {}

**Řadící algoritmy**

**Bubble Sort**

Porovná čísla a prohodí je, pokud jedno číslo je vetší než druhy

Command – návrhový vzor

Porovnání čísel

Slouží k tomu, abychom přidali funkci do parametru další funkce

Přetížená funkce

**Delegát**

Datový typ, který reprezentuje funkci

Public delegate int MyFunc(int a);

MyFunc je nový datový typ

Budou pasovat funkce, které berou int + vracejí a

MyFunc a = Xxx;

Xxx(1) -> a(1); - tím funkci zavoláme

Xxx je nazev funkce

U delegatu neni zavorka u funkce, jinak by se ta funkce spustila

Davame kod funkce do promenne

**Anonymní zápis**

**Lambda Funkce**

MyFunc m1 = delegate(int val)

{

return val + 10;

};

MyFunc m2 = (int val) =>

{

return val + 1;

};

MyFunc m3 = (int val) => val + 10;

MyFunc m4 = val => val + 10;

U listu:

Nastudovat:

Data.Select()

Data.Where()

Delegaty

Singleton

Public delegate bool Delegate(int a, int b)

Delegát který vrací bool a vstupuje do něj a a b

Funkce vraci hodnotu (public bool Neco() {})

Procedura nevraci hodnotu

Metoda je funkce ktera patri objektu (public class AAA{public bool Neco(){}})

DU

Public MyList Filter(Lambda) -> vraci kolekci po Where

Public MyList Map(Lambda) -> vraci kolekce po Select

Public bool Every(Lambda) -> vraci true pokud vsechny prvky splnuji where (po All)

Public … Reduce(Lambda, … init) -> vraci soucet vsech prvku

1 2 3 4

* 10 (Reduce)

Reduce(current, carry) -> current + carry, (init) 0

Do current zapisujeme hodnotu na prvku

Do carry děláme součet current + carry

Predvytvorene delegaty:

Action – void -> Action<int> a

Func – dva parametry -> Func<int, string> vraci int, bere string

Predicate

Návrhový vzor –

**Memento – paměťník**

Originator – původce, ze kterého se tvoří verze objektu

Memento – verze (objekt, který uchovává potřebné informace, abych se k tomu objektu mohl vrátit)

Caretaker – objekt, který uchovává verze

Memento může mít stejné vlastnosti jako originator

**Closure**

**Observer**

While cyklus, který stále něco kontroluje

Jakmile se něco vrátí, spustí se funkce

Během toho while cyklu nemůžeme dělat nic jiného

**Událost**

Public event TYP JMENO;

Object sender – o jaký objekt se jedna

EventArgs – parametry eventu – poloha myši atd.

**Decorator**

Extract Interface – na nazvu třídy

“Obaluje” původní objekt

Pokud potřebujeme, aby jedna funkce občas dělala i něco navíc

**Abstraktní třida**

Obecná třída, ze které nemůžu vytvořit instanci

Zaručí, že každý objekt, který jí dědí, bude mít ty metody

U abstraktní třídy můžu metody předdefinovat

**Interface**

Nemůžeme předdefinovat metody

Karel karel = new Karel();

^ určuje rozhrání, přes které komunikuju

Nevýhoda lambdy – nemůžeme lambdu zavolat znovu

**Iterator**

Projíždí data prvek po prvku (jako foreach)

Dávkovač

HasNext vrací true, pokud má co vracet

GetNext použije další prvek

MyListIterator – dávkuje data

Enumerator : IEnumerator

Dispose – odstrani tento objekt

MoveNext – vraci jestli se povedlo posunout – poslední bude mít false (this.current++;

Return this.current < this.data.Count;)

Reset – na začátek (current = -1

)

**Thread (vlákno) - nepovinné**

Zapojí druhý procesor

Defaultně využíváme pouze jeden procesor

Thread.Sleep(5000) – pauza na 5 sekund

Yield Return – pauzne funkci, pote pokračuje dal od místa zastavení

DU

Single Linked list

Add

Da se prochazet forearchem, whilem pomoci yieldu

Iterator

Místo Listu vlozit Single Linked List

For u Single Linked List je pomaly, místo toho tam dame Iterator

Current schováme v Iteratoru

**State – stav**

Lepší if

Zadame text

Vyhledani v nejake slozce soubory, které kdekoliv obsahuji text

Vratime všechny soubory které vyhovuji podmínce

1. Rekurze
2. Bez Rekurze (stack, queue)

While(!st.Empty())

Postupne prochazi všechny soubory a slozky

Prohledavani do hloubky (vždy potomek) DFS

Queue prochazi po slupkách BFS

**Algoritmy**

Vlastnosti algoritmů – elementárnost (algoritmus se skládá z elementárních kroků), konečnost (musí zkončit), resultativnost (musí vrátit správný výsledek), determinismus (se stejnými daty vždy vrátí stejný výsledek za stejný čas)

Postup, jak řešit problém

Každý algoritmus má složitost

Počet množství elementárních kroků, které program musí provést

Řády asymptotického růstu

Konstantní – ArrayList

Logaritmus2(n) – Binary Search - musí být uspořádaný, píchnu doprostřed a potom se vydám do části listu, kde se nachází

Linear Search – 4 MLD s

Binary Search – 2^32 -> 32s

Složitost:

Konstantní

Log(n) – Binary Search

Odmocnina(n)

n – Max, min, průměr

n \* log(n) – Match Sort, Heap Sort, Quick Sort

n^2 – Bubble Sort, Selection Sort, Insertion Sort

n^3

2^n – exponencionální algoritmy

n!

n^n

n – jeden for()

log – počet pater stromu, dělení listu

Třídy asymptotické složitosti

Omikron Ο – stejně a lepší

Omega Ω – stejně a horší

Théta Θ – (řádově) stejné

Porovnání algoritmů

F(x) ε O(n)

F(x) ε O(2^n)

Pokud porovnávám algoritmy z jedné třídy, k existuje (nikdy ho nedožene)

Pokud porovnávám algoritmy z různých tříd, k neexistuje (jednou ho dožene)

**Návrhový vzor – Strategy**

Algorithm

**Řadící algoritmy**

**Bubble Sort**

5; 2; 1; 8; 6; 3

Porovnává dvojice a pokud jsou špatně, tak je prohodí.

2; 5; 1; 8; 6; 3 (prohodí 2 a 5)

* 2; 1; 5; 8; 6 ;3 (prohodí 1 a 5)

Složitost: n \* (n - 1) = n^2 – n

* n^2

(n(n + 1))/2 = ½ n^2 + ½ n -> složitost n^2 (asymptoticky odstraníme všechno menší než n^2) => n/2 \* (n1 + nk) => n/2 \* (n-1 + 1) = n \* ((n-1) + 1)/2 = ((n^2)/2 => n^2

Optimalizace:

Nebudeme porovnávat poslední prvky, které už jsou seřazené

A nebudeme vždy porovnávat se složitostí n^2

Složitost znázorňuje jak složitý je algoritmus.

Složitost se pohybuje mezi n a n^2

n je když je to pole seřazené na začátku

Přirozenost: je přirozený (složitost se odvíjí od seřazenosti prvků)

Stabilita: Bubble Sort je stabilní (neprohazuje mezi sebou prvky (2; 2; **2**))

2; 4; 5; 2; 1; 3; **2**

1; 2; 2; **2**; 3; 4; 5

Stabilní podmínka: a > b

Nestabilní podmínka: a >= b

**Shaker / Cocktail Sort**

Bubble Sort jede jedním směrem

Cocktail Sort střídá směry

BS

----------------

-------------

----------

CS

---------------

-----------

-------

DU:

BS bez optimalizace, s optimalizací odřezávaní a s optimalizací testu seřazenosti

**Selection Sort**

Řazení výběrem

Ukládá minimum

Vždycky najde nejmenší prvek v poli a ten nejmenší prvek uloží na správné místo.

Uložené hodnoty se už neporovnávají.

Posouváme “začátek” pole a hledáme minimum

Nestabilní – např. (9, 2, 3, 9, 1) – algoritm zjistí, že 1 je nejmenší prvek v poli, prohodí 1 s 9. Tím pádem první 9 se stane druhou 9. Pořadí 9 se prohodí.

Nepřirozený

DÚ

**Insertion Sort**

Složitost: n^2

Reálná složitost: n(n+1)/2

Přirozený (n – n^2)

Stabilní

**Quick Sort**

Rozdělíme pole na malé a velké části a rekurzivně se to opakuje

Kurzory left a right jedou, dokud nenajdou prvky nevyhovující podmínce

Vezmeme první hodnotu pole a tím určíme pivot

Složitost: n \* log2(n) až n^2

Reálná složitost: n(n+1)/2 = n^2

Nepřirozený

Nestabilní

**Merge Sort**

Rozdělí pole na dvě poloviny (a, b) a pustí merge sort na každé polovině a vrátí hodnotu spojeného a a b

MERGE(a, b)

Složitost: n \* log2(n)

Stabilní

Nepřirozený

Paměťově náročný (2x pole)

**Heap Sort**

Heap je binární strom. Je to strom, který má v kořenu nejmenší/největší prvek. Jeho potomci jsou větší/menší

Pravidlo: Heap je rekurzivní

(i = 1) – (potomci) (2i) – (2i + 1)

1; 2; 3; 4; 5; 6; 7;

1 –> 2 + 3

2 –> 4 + 5

3 –> 6 + 7

Složitost: n \* log2(n)

**Radix Sort**

Řazení podle číselných řádů

Složitost: m \* n (m = počet řádů)

Složitost záleží na počtu řádů

Stabilní

Nepřirozený

**Bogo Sort**

Nedeterministický algoritm

**Dynamické datové typy (29)**

Do proměnné var můžeme uložit cokoliv. Je pro líné programátory. Pokud ho nastavíme na číslo, nemůžeme ho změnit na text. Nemůžeme ho dát do parametrů funkce a nemůžeme ho vrátit.

Do proměnné dynamic můžeme uložit cokoliv a můžeme později změnit datový typ. Můžeme ho dát do parametrů funkce a můžeme ho vrátit.

**Regulární výrazy (15)**

Definujeme vzor a snažíme se namatchovat string na vzor

Text se speciálními znaky

Výrazu “abc” budou vyhovovat jakékoliv stringy, které obsahují tento výraz

^ - výskyt na začátku výrazu

$ - výskyt na konci výrazu

“^ahoj$” – bude vyhovovat pouze string “ahoj”

| - alternativní možnost (OR)

“ahoj|nazdar” – vyhovují výrazy, které kdekoliv obsahují ahoj nebo nazdar

( ) – vytvoření skupiny podvýrazů

“^Pep(a|o)” – musí začínat na Pep, následovat musí buď a nebo o

. – libovolný znak

{ } – opakování znaků předcházející tomuto znaku

“a{1, 3}” – vyhovují “a”, “aa”, “aaa”, “aaahoj”, “pepaa”

a; a; a; a; a; a – vyhovuje ^a{1, 3}

{ minimum ,} – minimum x krát, ale maximum je bez limitu (parametry nejsou povinné)

{ 3 } – pouze 3x se může opakovat

{0, } -> **\*** - libovolný počet

{1, } -> **+** - minimálně jednou

{0, 1} -> **?** – 0x nebo 1x

[ ] – výčet možných znaků, zastupuje jeden znak, ale jen určité vybrané znaků (omezená tečka)

[ab5] – a, nebo b, nebo 5

[a-z] – jakýkoliv znak od a do z

Rozsah znaků vychází z ASCII tabulky

[^ ] – negace výčtu – všechny znaky kromě těch, které jsou vypsané

[^0-9] – vše kromě číslic

\ - Escape sequence, díky kterému můžeme použít symboly, které se využívají

Using System.Text.RegularExpressions;

Regex.IsMatch(“string k testování”, “pattern”) – můžeme uložit do boolu

jmeno.prijmeni -> @”^[a-z]+\.[a-z]+$”

\d – digits (čísla)

\D – písmena

\w – čísla i písmena

**Testování software (22)**

Důležitá činnost, která se provádí při vývoji aplikace

Fáze testování aplikace je prakticky nekonečná

Zajišťuje kvalitu kódu

V praxi mnohdy provádí specializovaný tým

Nalézt chyby v implementaci, nalézt chyby v zabezpečení, hledání konceptuálních chyb (zda vyhovuje požadavku zákazníka), testování zátěže (stresování namáhání programu), testy uživatelského rozhraní (zjišťování, zda se s programem dobře pracuje)

Testovatelnost – není možné plně testovat libovolný program

Komponentová sktruktura / pro efektivní testování je potřeba rozdělit program na části

Testy zdrojových kódů

White-Box testing – testování při znalosti kódu, testují se všechny průchody programem

UNIT TESTY

Black-Box testing – testování bez znalosti kódu, komponenty se testují na základě signatury – zda funguje, tak jak se očekává

INTEGRAČNÍ TESTY

Testy uživatelského rozhraní – automatizované (monkey), Usability Test (testování na skutečných uživatelích)

Git – několik branchů najednou

Alpha testy – reálné testování v laboratoři

Beta testy – testování v reálném prostředí

[ … ] - atributy, pouze značka, která nic nedělá, ale musíme ji před testem uvést, aby se objevil v Test Exploreru

Třída Assert

Zavoláme třídu, kterou testujeme (třeba z jiného projektu)

Uložíme do proměnné result metodu sčítání s čísly

Assert.AreEqual(8, result)

Databáze – uživatelé

Registrace uživatele – informace o sobě, profilový obrázek

Můžeme si někoho přidat do friend-listu, musí se požadavek potvrdit

Mezi lidma si můžeme posílat zprávy

Můžeme PM, nebo skupinové chaty

Textové zprávy, soubory, gify, emoji

Sent/Delivered/Read

Upravit zprávu, odstranit zprávu

Můžeme odstraňovat pouze svoje zprávy, v group chatu může odstraňovat zprávy a uživatele pouze admin

Desktop aplikace, webová aplikace (Angular)

Úkoly:

Tour of Heroes

Server – ASP .Net server, Entity Framework

Aplikace jednou za cca 10s zkontroluje, jestli jsou nové zprávy

Git, GitHub

MySQL

Rest API

Vymyslet datový model databáze

Začátek – návrh databáze a návrh uživatelského rozhraní

Use case diagramy

MyContext – třída, která reprezentuje připojení k databázi

DbSet – kolekce, která tahá data z databáze

<http://mysqlstudenti.litv.sssvt.cz/phpmyadmin/>

ORM – mapuje třídy na tabulky

Třída Person (vlastnosti třídy – sloupce v tabulce, datové typy musí odpovídat)

PersonRepository – context (databáze), obsahuje funkce pro práci s databázi (FindAll)

List personů, vrací this.context.People.ToList()

Person FindById(int id)

Vrací this.context.People.Where(x => x.id = id).FirstOrDefault NEBO this.context.People(id)

Void Create(Person person)

This.context.People.Add(person) + context.SaveChanges (commit tran)

Void Delete(Person person)

Context.people.remove(person) + context.SaveChanges

Void Update(Person person)

Uložíme si do proměnné current person = find.byid(person.id)

Vytáhneme data z databáze do kontextu a tím se propojí

Current.Name = person.Name …

SaveChanges

NEBO

Context.Entry(person).State = System.Data.Entity.EntityState.Modified

SaveChanges (změní “špinavé” persony)

Muzeme psat i SQL

[Table(“tb\_person”)] – anotace před třídu Person

API – programové rozhraní, jakým se komunikuje aplikace s jinou aplikaci

GUI – interface pro uživatele

RestAPI – standardizované API, funguje na principu http protokolu, get post atd

Json / XML – textově zakódované struktury dat

http metody – get (pro získaní dat ze serveru), post (pro vytvoření nových person)

put (pro update person, updatuje celý person), patch (pro update, mění jen některé),

delete (pro smazaní personu)

http status cody – 200, 201, 204

Controller – ověří, zda jsou data správné

WEB API with readwrite actions

PeopleController – instance PersonRepository

Get – kolekce Personu – vraci repository.FindAll()

Get(id) – vraci repository.FindById(id)

Post Person value – repository.Create(value)

Put int id, Person value – value.id = id; repository.Update

Delete int id – instan88ce Person – repository.FindById(id)

Repository.Delete(person.id)

Testovani – PostMan – Post – vrátí data z databáze v jsonu

Angular – nodejs -> npm ->

NPM – balíčkovací systém, který stahuje pluginy – nugety

SCSS (SASS)

Každá stránka má vlastní komponentu (angular schematic - component)

Ng generate component edit-page

Vytváření routů – do routing modelů dáme do proměnné Routes routy, musíme vyplnit cestu a název komponenty, musí existovat v app.module

Router links

App.component = skořápka aplikace

A routerlink = a href

Jedna stránka, kde měníme její obsah

Style.scss platí pro celou stránku

Pro jednotlivé stránky – nazevstranky.component.scss

Vytvoříme komponentu a do komponenty html stránky dáme tag komponenty a v html té podkomponenty nastavíme její strukturu a styl v css podkomponenty

Table-stripe

V phpstormu – model – typescript file – export class Person(Id: number, name: string, age: number)

HttpClientModule

Do Imports přidáme httpClientModule

Service – třida, která zajišťuje komunikaci se serverem

Service – person-service

Do konstruktoru:

Private http HttpClient

FindAll – Observerable<Person>

Vrací pole Person a stahuje je z nějaké url (tu dostaneme z PostMana)

V tabulce – do component.ts – do constructoru private services PersonService

Data:Person[] – musíme naimportovat

Ngoninit

This.service.findall().subscribe(next people => this.data = people)

V html - \*ngFor=let item of data – opakuje tag stejný počet jako proměnná data

Na serveru – config.EnableCors

ReactiveFormModule – pro formuláře

Kazda komponenta má vstupy I výstupy

U Formuláře dáme atributy do závorek [formGroup]=form (vstup)

(ngSubmit)=save()

V person.service

Save(persons person)

Return this.http.post(this.url, person)

V form – save()

This.service.save(this.form.value).subscribe(next () => console.log())

Metoda Refresh

Pipe – před subscribem můžeme přidat modifikátor pipe

? = if

V NgModule musí být zaregistrované všechny komponenty (v declarations)

Komponenta defaultně dědí z OnInit, který musíme po každé vypsat

OnChanged, OnDestroyed

[] – vstup

() – vystup

Index.html – hlavni stranka, vklada se sem skeleton aplikace <app-root></app-root> - nazev v @component -> selector

App.component.html – nav, div, uvnitr - <router-outlet></router-outlet> - vkládá se do něj komponenta definovana app-routing.module, routy určuje jestli bude edit-page, nebo list-page (úprava / seznam osob)

Styles.scss / App.component.scss – stylovaní stránky

Do app-person-table se vlozi kod tabulky – vytvoříme komponentu, do které vložíme kód a tu komponentu odkážeme v edit/list-page

Guard – služba, která hlídá přístup k jednotlivým routum – například pokud není přihlášený, vykopne ho, nebo jestli nemá oprávnění – nepustí ho

Observerable – (rxJS) – objekt, který vyhazuje události, na něj dáváme subscribe (jakmile Observerable vyhodí událost, my na to reagujeme)

Observerable<Person> - vyhodi Person kdyz probehne Event

Tap – nic nedělá

Map

SwitchMap

Of – jedno pole

From – po jednom

CHECKPOINT – 7.1.

Dva clienty + server

Odeslání zprávy z jednoho clientu – zpráva dorazí na druhý client (cross-platform) – nemusí být design na 100%

Obě dvě platformy musí být funkční – bugy nevadí

JWT – přihlašování

Server vrací token při přihlášení, má platnost, s každým požadavkem musí uživatel posílat ten token

Json web token – standard pro tokeny

Token se skládá z headeru, payloadu (tam si můžeme dát co chceme)

Verify signature – podpis

Na serveru je jedinecný podpis, který zná jenom server

.NET knihovna

+ tabulka Login – token

**UML (14)**

Unified Modelling Language

Jazyk pro popis jednotlivých částí programu

Využívá se v analýze a návrhu (vymýšlíme, jak program bude vypadat) a pro dokumentaci

Na základě těchto návrhů programátoři vytváří program

Skládá se z 12-13 diagramů pro různé pohledy programu (strukturální, behaviorální, use-case diagram)

Use Case

System Boundary, Actor, Use-case

Mezi Actorem a Use-casem je vztah

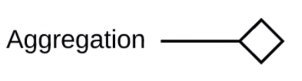
Mezi use-casema mohou být vztahy (extend – rozšiřující use-case, není povinný; include – povinný, součástí use-case)



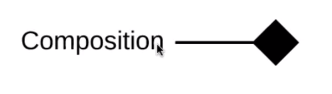
* -Při dědění



* + Asociace určuje základní vztah mezi dvěma entitami. Ty mohou existovat nezávisle na sobě.



* + Agregace reprezentuje vztah typu celek - část.



* + Kompozice je podobná agregaci, avšak reprezentuje silnější vztah. Entita části nemá bez celku smysl. Pokud zanikne celek, zanikají automaticky i jeho části.

Class Diagram

Diagram tříd, popisuje jak třídy vypadají z pohledu rozložení

Třída je rozdělena do tří částí, název (povinná), vlastnosti, metody

(User | +name: string; +age: int | +sayHello(): string)

Datové typy za dvojtečku, parametry metody se klasicky píšou do závorek

Minus – private, plus – public, # - protected

Abstraktní vlastnosti se píšou kurzívou

Statické vlastnosti se píšou podtrženě

Dědičnost se znázorňuje šipkou, která ukazuje na třídu, ze které dědí

Stereotyp se píše do “divných” uvozovek ((interface)) nad třídou

Agregace (prázdný kosočtverec), Kompozice (vybarvený kosočteverec) – znázorňuje, že jsou objekty v sobě

V případe kompozice je objekt povinný, u agregace tam nemusí

Asociace – obyčejná čára – znázorňuje provázanost objektů, může být bez šipek, s šipkami na jednu stranu, nebo na obě

Čárkovaná čára – Dependence / závislost, znázorňuje to, že objekt využívá druhý objekt, například jako součást metody jako vstupní parametr

Sekvenční diagram

Znázorňuje tok metod, závislost volání jednotlivých metod

Z objektů jde lifeline

Na lifelině se při zavolání metody znázorňuje tok metod obdélníkem

Plná čára – volání

Čárkovaná čára – vracení

Pokud potřebujeme objekt likvidovat, můžeme na konci jeho lifelinu udělat křížek, který znázorňuje jeho konec

State Diagram

Popisuje stav objektů

Otevřené okno, okno na ventilačku, zavřené okno, zničené okno

**Formuláře (17)**

Datagrid pořebuje Data Model, který implementuje IListSource

IListSource předdefinuje dvě metody IList GetList a bool ContainsListCollection

ContainsListCollection => true

[Browsable(false)] – schová sloupec v Datagridu

[DisplayName(“Jmeno”)] – pojmenuje názvy sloupců v Datagridu

ShowDialog zablokuje předchozí dialog, Show ne – formy nejsou na sobě závislé

ShowDialog vrací DialogResult

BindingList je asi nejlepší zdroj pro DataSource

ErrorProvider – komponenta pro signalizovaní chyb

Validace na úrovni položky – eventy Validating, Validated

**GDI grafika**

Picturebox – event Paint

Graphics g = e.Graphics;

Pen p = new Pen(Color.Red, tloustka10)

DrawLine(p, 10, 10, 200, 200)

Solidbrush b = new SolidBrush(Color.FromArgb(100(opacity), 0, 255, 0));

g.MeasureString

g.TranslateTransform – ke každé souřadnici přičte rozdíl

g.RotateTransform – zmena uhlu

g.ScaleTransform – zmena meritka

g.ResetTransform

g.VisibleClipBounds / g.ClipBounds – rozsah kreslene plochy

Bitmap bmp = new Bitmap

Graphics gr = Graphics.FromImage(bmp)

Bmp.Save()

Lazy loading – stahovani dat az kdyz je potrebujeme

Razor – možnost psaní HTML a C# zároveň (jako html + php) – C# s tagem @

**Reflexe**

Čtení kódu kódem

Čtu jaké má kód třídy, metody, vlastnosti…

Jsem schopen díky tomu dynamicky měnit názvy objektů

Programové přečtení kódu

Equals, GetHashCode, ToString, GetType – dědí se z objectu

GetType vrací instanci typu

Type type = f.GetType();

type.Name – vypise nazev tridy

type.FullName – vypise nazev namespace + tridy

foreach (PropertyInfo prop in type.GetProperties())

prop.Name – vypise nazvy vsech vlastnosti (napr. public string Name {get; set;})

foreach (FieldInfo field in type.GetFields())

Field.Name – vypise nazvy vsech fieldu (napr. public ing Age;)

Field.FieldType.Name – vypise nazev promenne (napr. int32)

Foreach (MethodInfo method in type.GetMethods())

Method.Invoke – zavola metodu

U reflexi neplati modifikatory pristupu – precte I privatni metody

Flagy modifikatoru pristupu – do parametru GetMethod() – BindingFlags.NonPublic (private objekty)

BindingFlags.NonPublic | BindingFlags.

U reflexi muzeme zavolat privatni metodu, nebo kontstruktor

Private int year;

Public int getYear() { return year; }

foreach (ConstructorInfo constructor in Type.GetConstructors())

object x = constructor.Invoke(new object[] {})

Type t = Type.GetType(“ConsoleApp1.Foo”)

Object f = Activator.CreateInstance(t);

MethodInfo method = Type.GetMethod(“SayHello”);

method.Invoke(f, new object[] {});

Atributy jsou ctene reflexi, ale doopravdy nic nedelaji

Čtení atributů (atributy – u datagridu, ASP – http/post/get, unit testy)

Možnost volání privátní a public metody

Volání konstruktoru

**UNIT TESTY**

Calculator calculator = new Calculator();

Assert.AreEqual(10, calculator.Sum(8, 2))

Assert.ThrowsException<DivideByZeroException>(() => calculator.Sum(5, 0));

CollectionAssert – pro testovani kolekci

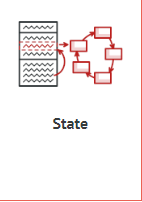
Metody Init a Clean se vzdy provedou pred kazdym testem a po testu s atributy [TestInit] a [TestCleanup]. Pokud mame nekolik testu v jednom projektu, tak se Init a Cleanup zavola po kazdem jednotlivem testu. Muzeme ale pouzivat ClassInitialize a ClassCleanup (ktere se provedou pouze jednou) s metodou:

Public static void SingleInit(TestContext context)

# **Část**

## 10) Návrhové vzory Gof

### **State**



Co řeší: Předcházíme spousty If or Switch statementu, v malém počtu Stavů není tak špatné použit if or switch ale když budeme mít stavů více, tak skončíme s velikou spoustou podmínek

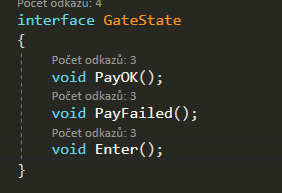
Řešení:

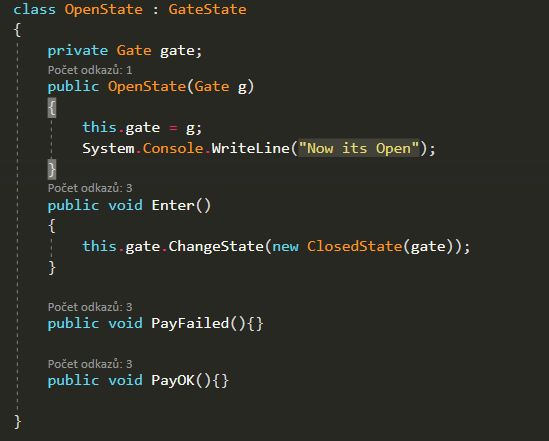
Stavový vzor doporučuje udělat novou třídu pro každý stav.

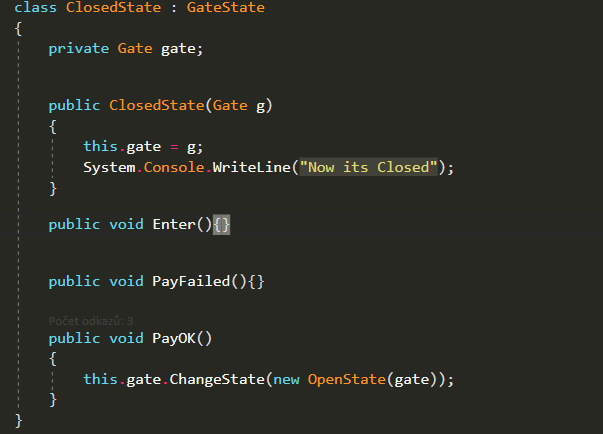
Příklad by mohl být turniket může být zavřený nebo otevřený což jsou dva stavy

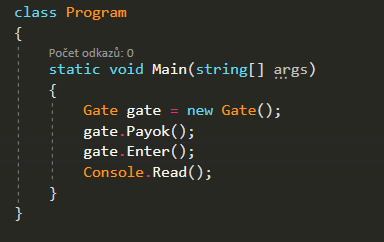
Implementace:







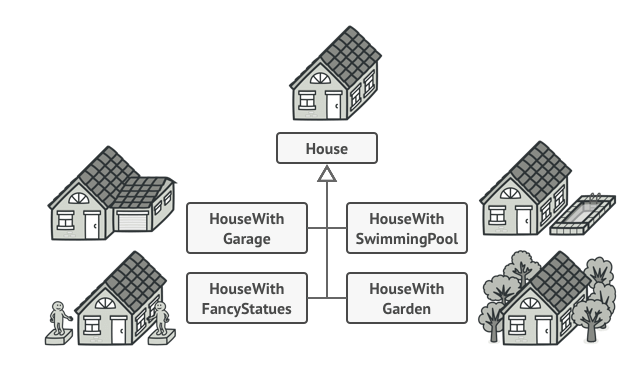




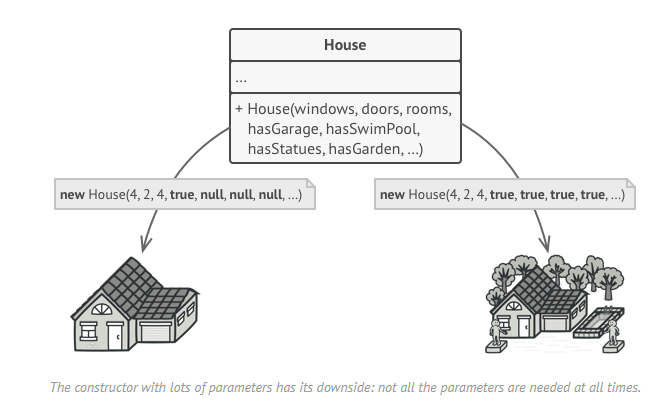
### **Builder**



Problém, který řeší:



Chceme postavit barák, tak si uděláme metodu pro vytvoření baráku, ale pak se rozhodneme, že ho chceme změnit např: že chceme přidat zahradu. Tak budeme muset změnit metodu a brát nový parametr a tak dále.



Řešení:

