IDT, Přednáška 3

Libor Váša

Katedra informatiky a výpočetní techniky, Západočeská univerzita v Plzni

26. 2. 2024

Vykonání programu v C#

Vykonání programu v C#

Paměť: zásobník (Stack) a halda (Heap)

Vykonání programu v C#

Paměť: zásobník (Stack) a halda (Heap)

- pozor na terminologii - neplést s datovými strukturami zásobník a halda
  - v češtině i v angličtině stejný název (heap) pro různé věci (oblast paměti/datová struktura)
  - je to něco zcela jiného! (zásobník je alespoň principiálně podobný, ale halda je úplně jiná!)

Vykonání programu v C#

Paměť: zásobník (Stack) a halda (Heap)

- pozor na terminologii - neplést s datovými strukturami zásobník a halda
  - v češtině i v angličtině stejný název (heap) pro různé věci (oblast paměti/datová struktura)
  - je to něco zcela jiného! (zásobník je alespoň principiálně podobný, ale halda je úplně jiná!)

- fyzicky jsou samozřejmě zásobník i halda ve stejné paměti, jen na jiných místech

Zásobník (Stack)

- při každém volání metody se alokuje místo pro skutečné parametry a lokální proměnné
- při ukončení metody se místo uvolní
- tady žijí lokální referenční proměnné
- omezená kapacita: když je zanoření do metod příliš hluboké, dojde k přetečení zásobníku (stack overflow)

Příklad

```
static double SmallerRoot(double a, double b, double c) throws Exception{
  double d = DiscriminantSqrt(a, b, c);
  double r1 = (-b+d)/(2*a);
  double r2 = (-b-d)/(2*a);
  if (Math.Abs(r1)<Math.Abs(r2))
    return r1;
  else
    return r2;
}
```

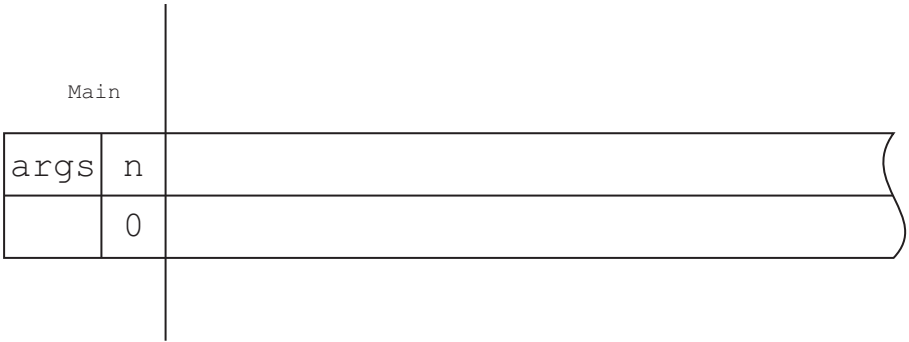
Příklad

```
static double SmallerRoot(double a, double b, double c) throws Exception{
  double d = DiscriminantSqrt(a, b, c);
  double r1 = (-b+d)/(2*a);
  double r2 = (-b-d)/(2*a);
  if (Math.Abs(r1)<Math.Abs(r2))
    return r1;
  else
    return r2;
}

static double DiscriminantSqrt(double a, double b, double c) throws Exception{
  double d = b*b-4*a*c;
  if (d>=0)
    return(Math.Sqrt(d));
  else throw new Exception();
}
```

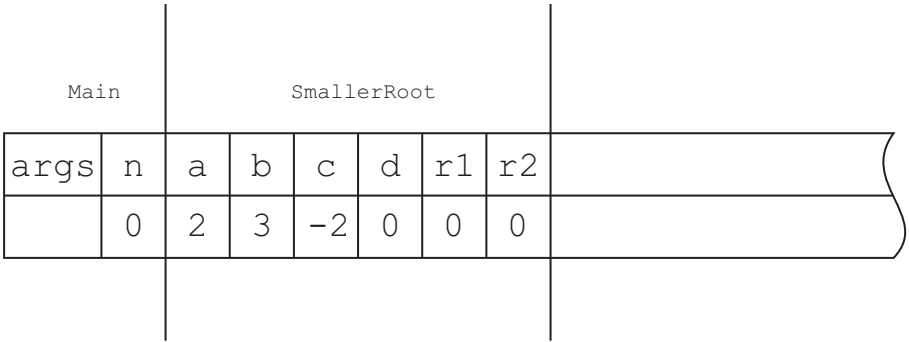
Příklad

```
static double SmallerRoot(double a, double b, double c) throws Exception{
  double d = DiscriminantSqrt(a, b, c);
  double r1 = (-b+d)/(2*a);
  double r2 = (-b-d)/(2*a);
  if (Math.Abs(r1)<Math.Abs(r2))
    return r1;
  else
    return r2;
}

static double DiscriminantSqrt(double a, double b, double c) throws Exception{
  double d = b*b-4*a*c;
  if (d>=0)
    return(Math.Sqrt(d));
  else throw new Exception();
}

static void Main(String[] args) throws Exception{
  double n = SmallerRoot(2, 3, -2);
  Console.WriteLine(n);
}
```

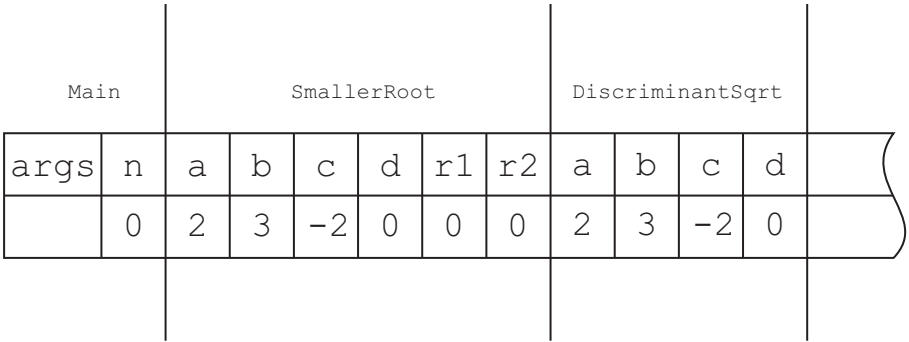
Zásobníkové rámce (Stack frames)



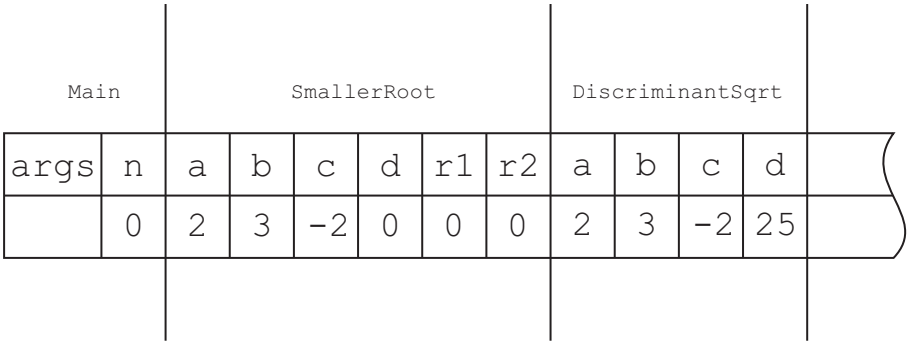
Zásobníkové rámce (Stack frames)



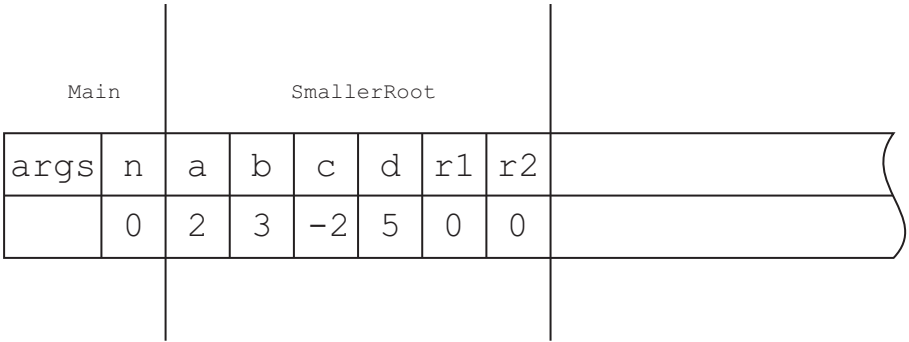
Zásobníkové rámce (Stack frames)



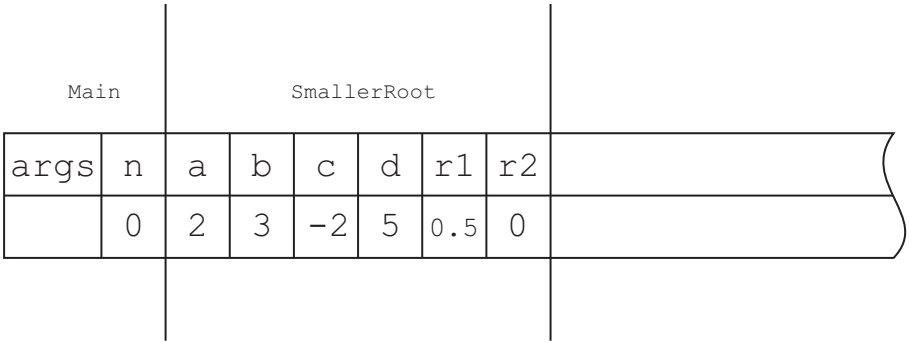
Zásobníkové rámce (Stack frames)



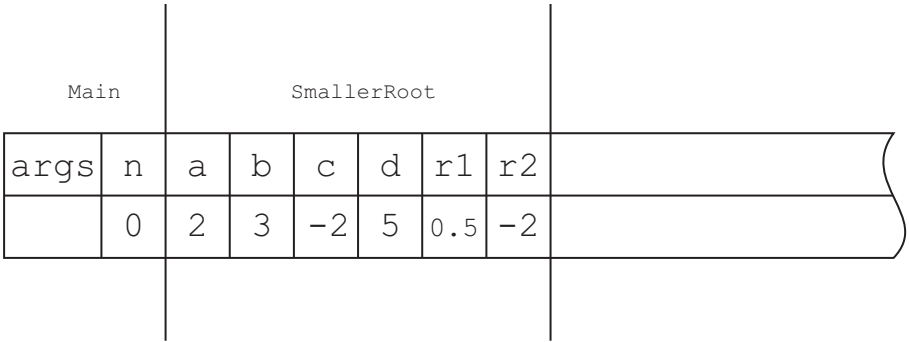
Zásobníkové rámce (Stack frames)



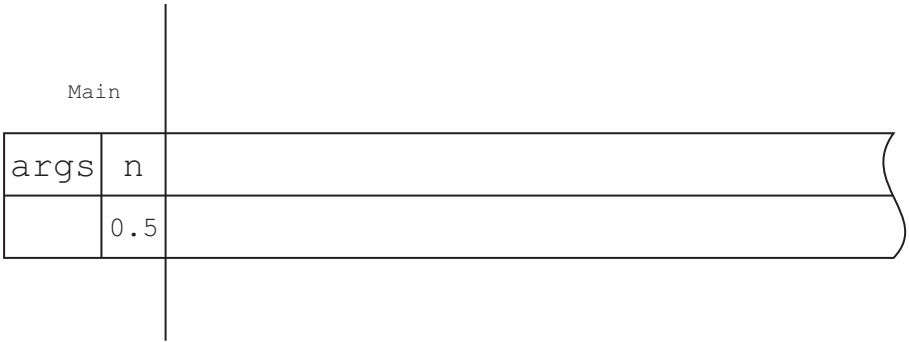
Zásobníkové rámce (Stack frames)



Zásobníkové rámce (Stack frames)



Zásobníkové rámce (Stack frames)



Halda (Heap)

- představuje většinu dostupné paměti (až gigabyty)
- tady žijí instance tříd
- paměť na haldě alokují konstruktory
- alokovaná paměť se uvolňuje, když na instanci neukazují žádné referenční proměnné
  - .NET hlídá reference ze zásobníku
  - odstranění zařizuje automaticky tzv. garbage collector

Důsledky

- alokování paměti v zásobníku je možné jedině deklarací lokální proměnné
- paměť je tudíž „statická“ - dopředu lze určit, kolik se jí bude alokovat (.NET to dělá)

Důsledky

- alokování paměti v zásobníku je možné jedině deklarací lokální proměnné
- paměť je tudíž „statická“ - dopředu lze určit, kolik se jí bude alokovat (.NET to dělá)

Dynamická data

- velikost není při překladu známá (záleží např. na vstupu od uživatele, velikosti vstupního souboru, délce běhu programu apod.)
- uložena jedině na haldě!

Důsledky

- alokování paměti v zásobníku je možné jedině deklarací lokální proměnné
- paměť je tudíž „statická“ - dopředu lze určit, kolik se jí bude alokovat (.NET to dělá)

Dynamická data

- velikost není při překladu známá (záleží např. na vstupu od uživatele, velikosti vstupního souboru, délce běhu programu apod.)
- uložena jedině na haldě!

Pravidlo

- data v zásobníku mají jméno (proměnné, parametry)
- data na haldě nikdy jméno nemají (jen reference)

Námitka 1: pole

Pole si přece můžu udělat jak velké chci...

Námitka 1: pole

Pole si přece můžu udělat jak velké chci...

Pole je třída!

Námitka 1: pole

Pole si přece můžu udělat jak velké chci...

Pole je třída!

```
String s = Console.ReadLine();
int length = Int32.Parse(s);
int[] array; // referencni promenna!
array = new int[length]; // konstruktor!
```

Námitka 1: pole

Pole si přece můžu udělat jak velké chci...

Pole je třída!

```
String s = Console.ReadLine();
int length = Int32.Parse(s);
int[] array; // referencni promenna!
array = new int[length]; // konstruktor!
```

```
int[] ref2 = array;
ref2[0] = 10;
Console.WriteLine(array[0]);
```

Námitka 2: řetězec

String je přece tak dlouhý, jak ho uživatel napíše

Námitka 2: řetězec

String je přece tak dlouhý, jak ho uživatel napíše

String je třída!

Námitka 2: řetězec

String je přece tak dlouhý, jak ho uživatel napíše

String je třída!

```
String s; // referencni promenna!
String s = Console.ReadLine(); // nextLine vraci referenci
String s2 = s; // s2 ukazuje na stejnou instanci
```

Pozor

Řetězce v C# jsou tzv. neměnné (immutable), nelze měnit jejich obsah. Všechny metody, které zdánlivě obsah mění, ve skutečnosti vracejí referenci na novou instanci!

Pravda o řetězcích

```
String s1 = "Pokus";
String s2 = "Pokus";
Console.WriteLine(s1==s2);
```

Řetězec v uvozovkách funguje (částečně) jako konstruktor

- vrací referenci na instanci třídy `String`

Opterátor == vyhodnocuje shodu obsahu

- zvláštní chování třídy String, obvykle se vyhodnocuje shoda referencí!

Příklad - vykonání programu

```
class Person{
  int a;
  Person father;
```

Příklad - vykonání programu

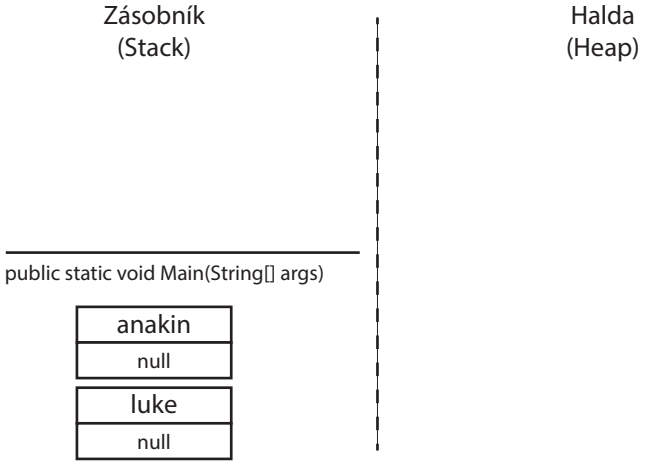
```
class Person{
  int a;
  Person father;

  public Person(int p)  {
    this.a = p;
  }

  public void SetFather(Person p)  {
    this.father = p;
  }
}

static void Main(String[] args){
  Person luke = new Person(25);
  Person anakin = new Person(50);
  luke.SetFather(anakin);
}
```

Příklad - vykonání programu

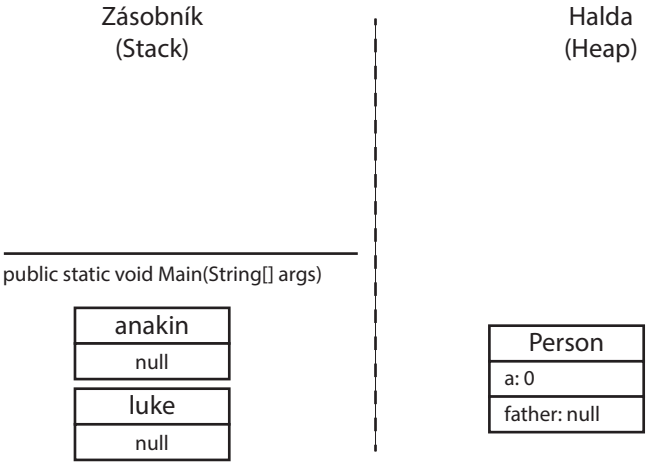
Zásobník
(Stack)

Halda
(Heap)

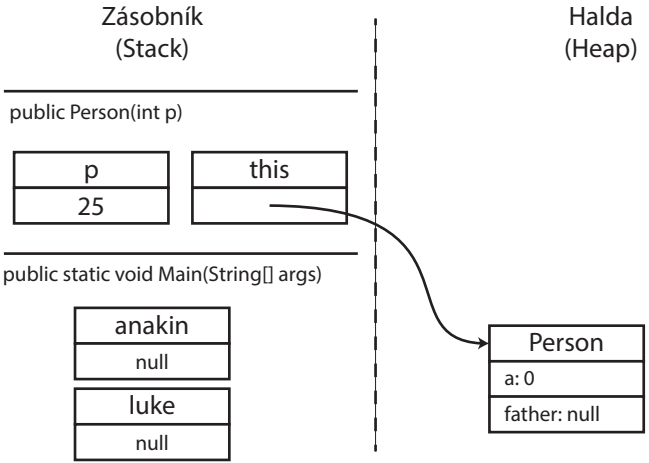
Příklad - vykonání programu



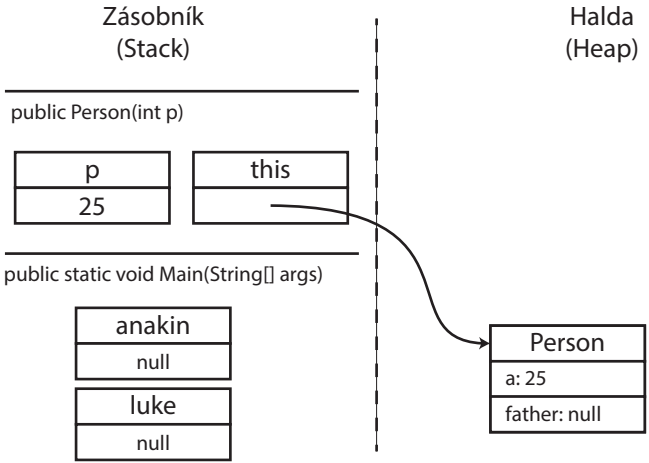
Příklad - vykonání programu



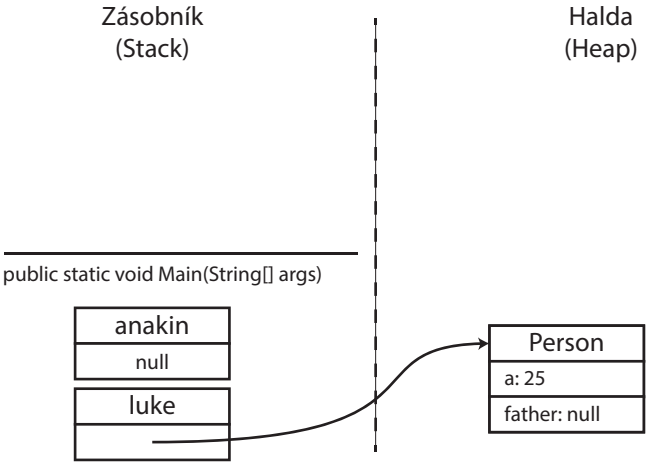
Příklad - vykonání programu



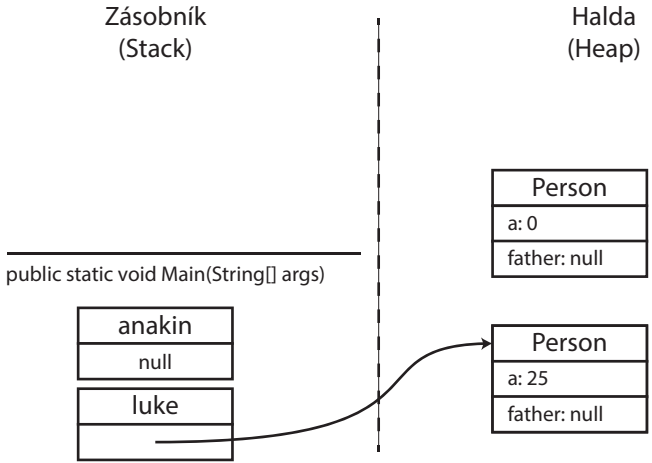
Příklad - vykonání programu



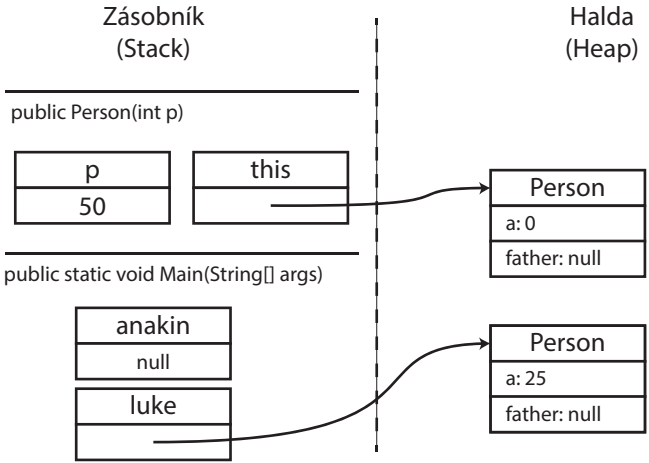
Příklad - vykonání programu



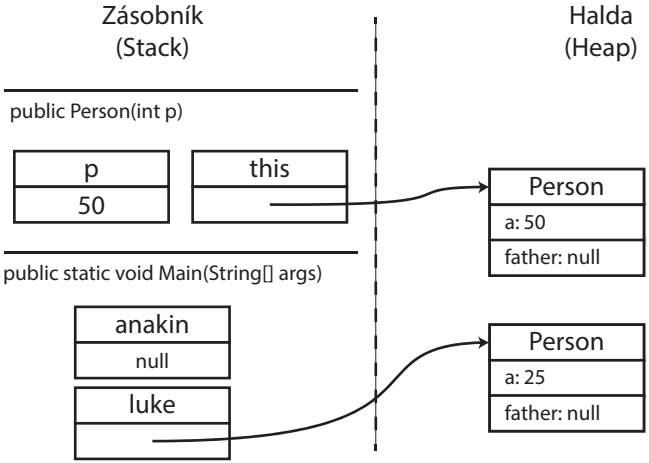
Příklad - vykonání programu



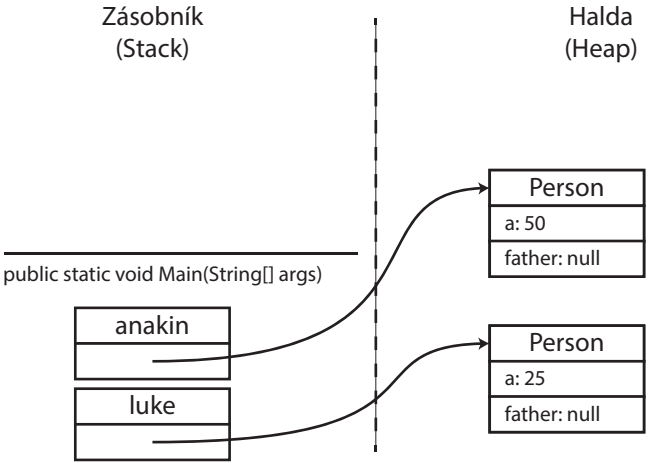
Příklad - vykonání programu



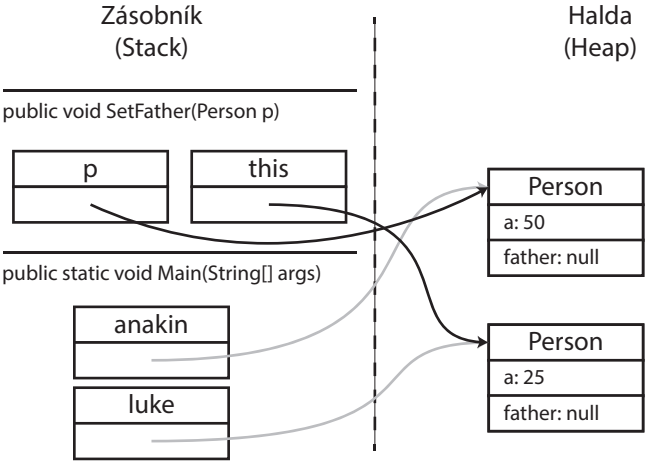
Příklad - vykonání programu



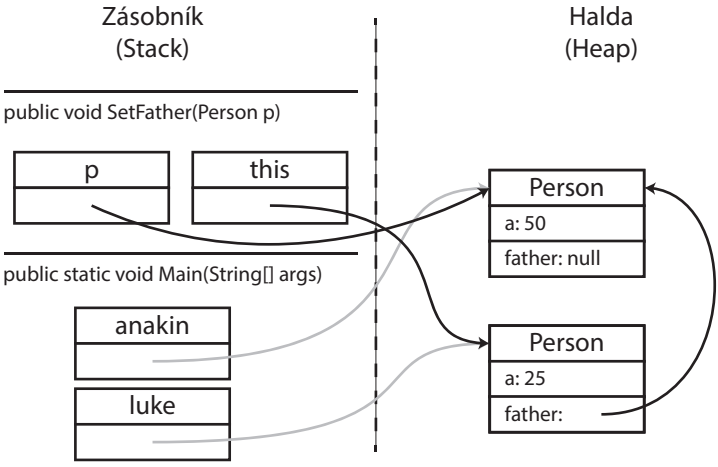
Příklad - vykonání programu



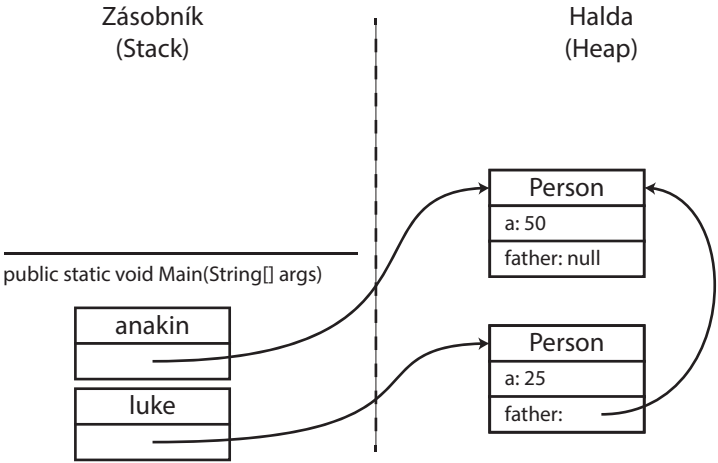
Příklad - vykonání programu



Příklad - vykonání programu



Příklad - vykonání programu



Poznámka

- instance třídy nijak neeviduje reference které na ni ukazují

```
anakin = null;
Person darthVader = luke.father;
```

Poznámka

- instance třídy nijak neeviduje reference které na ni ukazují

```
anakin = null;
Person darthVader = luke.father;
```

- Garbage collector uvolňuje paměť na haldě

Poznámka

- instance třídy nijak neeviduje reference které na ni ukazují

```
anakin = null;
Person darthVader = luke.father;
```

- Garbage collector uvolňuje paměť na haldě
- všechny instance na které vedou reference ze zásobníku označí jako neodstranitelné

Poznámka

- instance třídy nijak neeviduje reference které na ni ukazují

```
anakin = null;
Person darthVader = luke.father;
```

- Garbage collector uvolňuje paměť na haldě
- všechny instance na které vedou reference ze zásobníku označí jako neodstranitelné
- všechny instance na které vedou reference z neodstranitelných instancí označí jako neodstranitelné (iterativně)

Poznámka

- instance třídy nijak neeviduje reference které na ni ukazují

```
anakin = null;
Person darthVader = luke.father;
```

- Garbage collector uvolňuje paměť na haldě
- všechny instance na které vedou reference ze zásobníku označí jako neodstranitelné
- všechny instance na které vedou reference z neodstranitelných instancí označí jako neodstranitelné (iterativně)
- všechny ostatní instance jsou odstranitelné a jejich paměť bude recyklována

Poznámka

- instance třídy nijak neeviduje reference které na ni ukazují

```
anakin = null;
Person darthVader = luke.father;
```

- Garbage collector uvolňuje paměť na haldě
- všechny instance na které vedou reference ze zásobníku označí jako neodstranitelné
- všechny instance na které vedou reference z neodstranitelných instancí označí jako neodstranitelné (iterativně)
- všechny ostatní instance jsou odstranitelné a jejich paměť bude recyklována

```
System.GC.Collect(); // pozada o recyklaci odpadu
```

Generické programy

Generické programování

Význam

Generické = obecné, společné

Generické programování

Význam

Generické = obecné, společné

snaha podchytit společné vlastnosti různých algoritmů/datových struktur

Generické programování

Význam

Generické = obecné, společné

snaha podchytit společné vlastnosti různých algoritmů/datových struktur

Antonymum

Specifické = konkrétní, speciální, zvláštní

Generické programování

Význam

Generické = obecné, společné

snaha podchytit společné vlastnosti různých algoritmů/datových struktur

Antonymum

Specifické = konkrétní, speciální, zvláštní

Cíle:

- umožnit záměnu různých specifických implementací splňujících nějaká generická kritéria
- umožnit sdílení generického kódu různými specifickými implementacemi

Problém

- různé algoritmy řazení (bubblesort, insertsort, ...)
- není dopředu jasné, který bude lepší použít

Problém

- různé algoritmy řazení (bubblesort, insertsort, ...)
- není dopředu jasné, který bude lepší použít
- implementujeme oba/všechny a vyzkoušíme na reálných datech
- budeme chtít snadno přecházet od jednoho algoritmu k druhému

Možné řešení

Vytvoříme pro každý algoritmus třídu, která ho svými metodami implementuje

```
class BubbleSort{
  public void BubbleSort(int[] data){
    ...
  }
}

class InsertSort{
  public void InsertSort(int[] data){
    ...
  }
}
```

Použití

```
...
int[] data1 = GenerateData1();
int[] data2 = GenerateData2();
BubbleSort sorter = new BubbleSort();
sorter.BubbleSort(data1);
sorter.BubbleSort(data2);
...
```

Použití

```
...
int[] data1 = GenerateData1();
int[] data2 = GenerateData2();
BubbleSort sorter = new BubbleSort();
sorter.BubbleSort(data1);
sorter.BubbleSort(data2);
...
```

```
...
int[] data1 = GenerateData1();
int[] data2 = GenerateData2();
InsertSort sorter = new InsertSort();
sorter.InsertSort(data1);
sorter.InsertSort(data2);
...
```

Dobrý nápad

Dáme metodám se stejným významem stejný název

```
class BubbleSort{
  public void Sort(int[] data){
    ...
  }
}

class InsertSort{
  public void Sort(int[] data){
    ...
  }
}
```

Použití

```
...
int[] data1 = GenerateData1();
int[] data2 = GenerateData2();
BubbleSort sorter = new BubbleSort();
sorter.Sort(data1);
sorter.Sort(data2);
...
```

Použití

```
...
int[] data1 = GenerateData1();
int[] data2 = GenerateData2();
BubbleSort sorter = new BubbleSort();
sorter.Sort(data1);
sorter.Sort(data2);
...
```

```
...
int[] data1 = GenerateData1();
int[] data2 = GenerateData2();
InsertSort sorter = new InsertSort();
sorter.Sort(data1);
sorter.Sort(data2);
...
```

změna algoritmu ale stále vyžaduje rekompilaci

Použití?

```
...
int[] data1 = GenerateData1();
int[] data2 = GenerateData2();
if (Console.ReadLine() == "b")
  BubbleSort sorter = new BubbleSort();
else
  InsertSort sorter = new InsertSort();
sorter.Sort(data1);
sorter.Sort(data2);
...
```

Jde to?

Použití?

```
...
int[] data1 = GenerateData1();
int[] data2 = GenerateData2();
if (Console.ReadLine() == "b")
  BubbleSort sorter = new BubbleSort();
else
  InsertSort sorter = new InsertSort();
sorter.Sort(data1);
sorter.Sort(data2);
...
```

Jde to?

Nejde! je-li tělem podmínky jediný příkaz, nesmí to být deklarace proměnné

Nešlo by toto?

```
...
int[] data1 = GenerateData1();
int[] data2 = GenerateData2();
if (Console.ReadLine() == "b") {
  BubbleSort sorter = new BubbleSort();
}
else {
  InsertSort sorter = new InsertSort();
}
sorter.Sort(data1);
sorter.Sort(data2);
...
```

Nešlo by toto?

```
...
int[] data1 = GenerateData1();
int[] data2 = GenerateData2();
if (Console.ReadLine() == "b") {
  BubbleSort sorter = new BubbleSort();
}
else {
  InsertSort sorter = new InsertSort();
}
sorter.Sort(data1);
sorter.Sort(data2);
...
```

Také nejde! proměnné definované v bloku za koncem bloku už neexistují

Co by šlo

```
...
int[] data1 = GenerateData1();
int[] data2 = GenerateData2();
if (Console.ReadLine() == "b") {
  BubbleSort sorter = new BubbleSort();
  sorter.Sort(data1);
  sorter.Sort(data2);
}
else {
  InsertSort sorter = new InsertSort();
  sorter.Sort(data1);
  sorter.Sort(data2);
}
...
```

Výkonná část kódu je zduplikovaná, změny budeme muset dělat na dvou místech!

Elegantnější řešení

zavedeme tzv. rozhraní (Interface)

- popisuje, co nějaká třída umí: hlavičky metod
- neříká nic o tom, jak jsou metody implementovány: nemají tělo

Elegantnější řešení

zavedeme tzv. rozhraní (Interface)

- popisuje, co nějaká třída umí: hlavičky metod
- neříká nic o tom, jak jsou metody implementovány: nemají tělo

název rozhraní se může objevit všude tam, kde se očekává název třídy

Elegantnější řešení

zavedeme tzv. rozhraní (Interface)

- popisuje, co nějaká třída umí: hlavičky metod
- neříká nic o tom, jak jsou metody implementovány: nemají tělo

název rozhraní se může objevit všude tam, kde se očekává název třídy různé třídy mohou implementovat stejné rozhraní

Elegantnější řešení

zavedeme tzv. rozhraní (Interface)

- popisuje, co nějaká třída umí: hlavičky metod
- neříká nic o tom, jak jsou metody implementovány: nemají tělo

název rozhraní se může objevit všude tam, kde se očekává název třídy různé třídy mohou implementovat stejné rozhraní

jedna třída může implementovat více rozhraní

Elegantnější řešení

zavedeme tzv. rozhraní (Interface)

- popisuje, co nějaká třída umí: hlavičky metod
- neříká nic o tom, jak jsou metody implementovány: nemají tělo

název rozhraní se může objevit všude tam, kde se očekává název třídy různé třídy mohou

implementovat stejné rozhraní

jedna třída může implementovat více rozhraní

klientský kód pracuje s rozhraním, jako by to byla třída

- volá metody definované v rozhraní

Příklad

```
interface ISorter {
  void Sort(int[] i);
}
```

Příklad

```
interface ISorter {
  void Sort(int[] i);
}

class BubbleSort : ISorter{
  public void Sort(int[] data){
    ...
  }
}
```

Příklad

```
interface ISorter {
  void Sort(int[] i);
}

class BubbleSort : ISorter{
  public void Sort(int[] data){
    ...
  }
}

class InsertSort : ISorter{
  public void Sort(int[] data){
    ...
  }
}
```

konvence: názvy rozhraní začínají velkým písmenem I

Použití

```
...
int[] data1 = GenerateData1();
int[] data2 = GenerateData2();
ISorter sorter = new BubbleSort();
sorter.Sort(data1);
sorter.Sort(data2);
...
```

Použití

```
...
int[] data1 = GenerateData1();
int[] data2 = GenerateData2();
ISorter sorter = new BubbleSort();
sorter.Sort(data1);
sorter.Sort(data2);
...
```

```
...
int[] data1 = GenerateData1();
int[] data2 = GenerateData2();
ISorter sorter = new InsertSort();
sorter.Sort(data1);
sorter.Sort(data2);
...
```

Hlavní rozdíl

změnu implementace je možné provést za běhu

- bez rekompilace
- větvení jen při vytváření instance, zbytek kódu je jen jednou

Hlavní rozdíl

změnu implementace je možné provést za běhu

- bez rekompilace
- větvení jen při vytváření instance, zbytek kódu je jen jednou

```
...
int[] data1 = GenerateData1();
int[] data2 = GenerateData2();
ISorter sorter;
if (Console.ReadLine() == "b")
  sorter = new BubbleSort();
else
  sorter = new InsertSort();
// zbytek kodu uz je bez vetveni
sorter.Sort(data1);
sorter.Sort(data2);
...
```

Proč se to jmenuje zrovna „rozhraní“?

typické použití:

- 2 programátoři, 1 problém

Proč se to jmenuje zrovna „rozhraní“?

typické použití:

- 2 programátoři, 1 problém
- programátor Trautenberk píše třídu, kterou programátor Krakonoš využívá

Proč se to jmenuje zrovna „rozhraní“?

typické použití:

- 2 programátoři, 1 problém
- programátor Trautenberk píše třídu, kterou programátor Krakonoš využívá

dohodnou se, co bude třída určitě poskytovat

- Trautenberk ví, co musí naimplementovat
- Krakonoš ví, na co se může spolehnout
- ohlídá to překladač

Proč se to jmenuje zrovna „rozhraní“?

typické použití:

- 2 programátoři, 1 problém
- programátor Trautenberk píše třídu, kterou programátor Krakonoš využívá

dohodnou se, co bude třída určitě poskytovat

- Trautenberk ví, co musí naimplementovat
- Krakonoš ví, na co se může spolehnout
- ohlídá to překladač

Navíc

- jasně definované rozhraní předchází sporům

Navíc

- jasně definované rozhraní předchází sporům
- pokud Trautenberk dodrží rozhraní, může bez obav z konfliktu vylepšovat svoji implementaci

Navíc

- jasně definované rozhraní předchází sporům
- pokud Trautenberk dodrží rozhraní, může bez obav z konfliktu vylepšovat svoji implementaci
- Trautenberkových služeb mohou využívat i jiní sousedé

Navíc

- jasně definované rozhraní předchází sporům
- pokud Trautenberk dodrží rozhraní, může bez obav z konfliktu vylepšovat svoji implementaci
- Trautenberkových služeb mohou využívat i jiní sousedé
- časem je snadno možné Trautenberkovu práci nahradit nějakou jinou/lepší implementací, pokud opět vyhoví rozhraní

Jiný příklad

máme následující rafinovanou metodu pro řazení celých čísel:

```
void Sort(int[] data) {
  for (int i = 1;i<data.Length;i++)
    for (int j = i;j<data.Length;j++)
      if (data[j]<data[j-1]) {
        int tmp = data[j];
        data[j] = data[j-1];
        data[j-1] = tmp;
      }
}
```

Jiný příklad

máme následující rafinovanou metodu pro řazení celých čísel:

```
void Sort(int[] data) {
  for (int i = 1;i<data.Length;i++)
    for (int j = i;j<data.Length;j++)
      if (data[j]<data[j-1]) {
        int tmp = data[j];
        data[j] = data[j-1];
        data[j-1] = tmp;
      }
}
```

... potřebujeme ale místo celých čísel řadit rozvrhové události (instance `PlanEvent`)

PlanEvent

```
class PlanEvent {
  int dayOfWeek;
  int start;
  int end;
}
```

Možné řešení

Dvě metody:

```
SortIntegers(int[] data)
SortPlanEvents(PlanEvent[] data)
```

Možné řešení

Dvě metody:

```
SortIntegers(int[] data)
SortPlanEvents(PlanEvent[] data)
```

```
void SortPlanEvents(PlanEvent[] data) {
  for (int i = 1;i<data.Length;i++)
    for (int j = i;j<data.Length;j++)
      if (data[j]<data[j-1]) {
        PlanEvent tmp = data[j];
        data[j] = data[j-1];
        data[j-1] = tmp;
      }
}
```

Možné řešení

Dvě metody:

`SortIntegers(int[] data)`

`SortPlanEvents(PlanEvent[] data)`

```
void SortPlanEvents(PlanEvent[] data) {
  for (int i = 1;i<data.Length;i++)
    for (int j = i;j<data.Length;j++)
      if (data[j]<data[j-1]) {
        PlanEvent tmp = data[j];
        data[j] = data[j-1];
        data[j-1] = tmp;
      }
}
```

Nepřeloží se, pro reference na `PlanEvent` nedává smysl operátor $<$.

Fungující řešení

```
void SortPlanEvents(PlanEvent[] data) {
  for (int i = 1;i<data.Length;i++)
    for (int j = i;j<data.Length;j++)
      if (data[j].BelongsBefore(data[j-1])) {
        PlanEvent tmp = data[j];
        data[j] = data[j-1];
        data[j-1] = tmp;
      }
}
```

Fungující řešení

```
void SortPlanEvents(PlanEvent[] data) {
  for (int i = 1;i<data.Length;i++)
    for (int j = i;j<data.Length;j++)
      if (data[j].BelongsBefore(data[j-1])) {
        PlanEvent tmp = data[j];
        data[j] = data[j-1];
        data[j-1] = tmp;
      }
}

class PlanEvent {
  ...
  public bool BelongsBefore(PlanEvent e) {
    if (this.dayOfWeek<e.dayOfWeek)
      return true;
    if (this.dayOfWeek == e.dayOfWeek)
      if (this.start<e.start)
        return true;
    return false;
  }
}
```

Generalizace

Vtíravá myšlenka

Metodou `sortPlanEvents` by mohlo jít řadit všechno co poskytuje metodu `belongsBefore`...?

Generalizace

Vtíravá myšlenka

Metodou `sortPlanEvents` by mohlo jít řadit všechno co poskytuje metodu `belongsBefore`...?

```
interface ISortable {
  bool BelongsBefore(ISortable x);
}
```

Generalizace

Vtíravá myšlenka

Metodou `sortPlanEvents` by mohlo jít řadit všechno co poskytuje metodu `belongsBefore`...?

```
interface ISortable {
  bool BelongsBefore(ISortable x);
}
void sortAnything(ISortable[] data) {
  for (int i = 1;i<data.Length;i++)
    for (int j = i;j<data.Length;j++)
      if (data[j].BelongsBefore(data[j-1])) {
        ISortable tmp = data[j];
        data[j] = data[j-1];
        data[j-1] = tmp;
      }
}
```

Řaditelná třída

```
class PlanEvent : ISortable{
  ...
  public bool BelongsBefore(ISortable x) {
    PlanEvent e = (PlanEvent)x;
    if (this.dayOfWeek<e.dayOfWeek)
      return true;
    if (this.dayOfWeek == e.dayOfWeek)
      if (this.start<e.start)
        return true;
    return false;
  }
}
```

Řaditelná třída

```
class PlanEvent : ISortable{
  ...
  public bool BelongsBefore(ISortable x) {
    PlanEvent e = (PlanEvent)x;
    if (this.dayOfWeek<e.dayOfWeek)
      return true;
    if (this.dayOfWeek == e.dayOfWeek)
      if (this.start<e.start)
        return true;
    return false;
  }
}
```

metoda způsobuje vyjímku pokud parametr není reference na `PlanEvent`

Řaditelná třída

```
class PlanEvent : ISortable{
  ...
  public bool BelongsBefore(ISortable x) {
    PlanEvent e = (PlanEvent)x;
    if (this.dayOfWeek<e.dayOfWeek)
      return true;
    if (this.dayOfWeek == e.dayOfWeek)
      if (this.start<e.start)
        return true;
    return false;
  }
}
```

metoda způsobuje vyjímku pokud parametr není reference na `PlanEvent`

- s tím co zatím známe to nejde obejít
- později si ukážeme jak to obejít jde

Technické aspekty

.NET již zavádí podobné rozhraní `IComparable`

```
interface IComparable {
  public int CompareTo(Object o);
}
```

- vrací záporné číslo pokud patří před, nulu pokud je „stejné“ a kladné číslo pokud patří za předaný prvek
- lepší je použít tzv. generické rozhraní `IComparable<T>`, ukážeme si později

Technické aspekty

.NET již zavádí podobné rozhraní `IComparable`

```
interface IComparable {
  public int CompareTo(Object o);
}
```

- vrací záporné číslo pokud patří před, nulu pokud je „stejné“ a kladné číslo pokud patří za předaný prvek
- lepší je použít tzv. generické rozhraní `IComparable<T>`, ukážeme si později

Řazení primitivních datových typů

- i primitivní datové typy mohou poskytovat metody a implementovat rozhraní
- datové typy `int`, `double` a další rozhraní `IComparable` implementují

Generalizace z pohledu Krakonoše

```
void SortAnything(IComparable[] data) {
  for (int i = 1;i<data.Length;i++)
    for (int j = i;j<data.Length;j++)
      if (data[j].CompareTo(data[j-1])<0) {
        IComparable tmp = data[j];
        data[j] = data[j-1];
        data[j-1] = tmp;
      }
}
```

... a z pohledu Trautenberka

`Array.Sort` je ochotná seřadit cokoli co implementuje `IComparable`

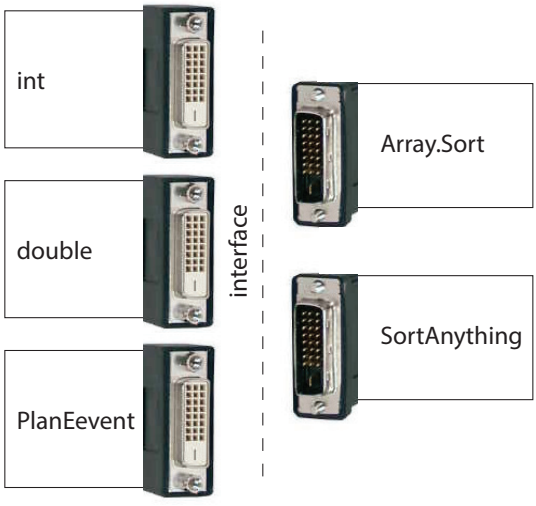
```
class PlanEvent : IComparable {
  ...
  public int CompareTo(Object other) {
    PlanEvent e = (PlanEvent)other;
    if (this.dayOfWeek<e.dayOfWeek)
      return -1;
    if (this.dayOfWeek == e.dayOfWeek){
      return this.start - e.start;
    }
    return 1;
  }
}
```

... a z pohledu Trautenberka

`Array.Sort` je ochotná seřadit cokoli co implementuje `IComparable`

```
class PlanEvent : IComparable {
  ...
  public int CompareTo(Object other) {
    PlanEvent e = (PlanEvent)other;
    if (this.dayOfWeek<e.dayOfWeek)
      return -1;
    if (this.dayOfWeek == e.dayOfWeek){
      return this.start - e.start;
    }
    return 1;
  }
}

...
PlanEvent[] events = ...
Array.Sort(events);
```

Ještě jednou co je rozhraní



Příště

- dědičnost
- třídy s typovým parametrem