IDT, Přednáška 4

Libor Váša

Katedra informatiky a výpočetní techniky, Západočeská univerzita v Plzni

4. 3. 2024

Dědičnost

Příklad

třídy hledající v seřazeném poli

- sekvenční vyhledávání
- půlení intervalu

Příklad

třídy hledající v seřazeném poli

- sekvenční vyhledávání
- půlení intervalu

možné řešení: rozhraní `IFinder`

```
interface IFinder {
        void SetData(int[] data);
        bool Find(int x);
}
```

Třídy implementující rozhraní

```
class SequentialFinder : IFinder {
  int[] data;
  void SetData(int[] data) throws Exception {
    for (int i = 1;i<data.Length;i++)
      if (data[i]<data[i-1])
        throw new Exception();
    this.data = data;
  }
```

Třídy implementující rozhraní

```
class SequentialFinder : IFinder {
  int[] data;
  void SetData(int[] data) throws Exception {
    for (int i = 1;i<data.Length;i++)
      if (data[i]<data[i-1])
        throw new Exception();
    this.data = data;
  }
  bool Find(int x) {
    for(int i = 0;i<data.Length;i++)
    if (data[i] == x)
      return true;
    else if (data[i]>x)
      return false
    return false;
  }
}
```

Třídy implementující rozhraní

```
class SequentialFinder : IFinder {
  int[] data;
  void SetData(int[] data) throws Exception {
    for (int i = 1;i<data.Length;i++)
      if (data[i]<data[i-1])
        throw new Exception();
    this.data = data;
  }
  bool Find(int x) {
    for(int i = 0;i<data.Length;i++)
    if (data[i] == x)
      return true;
    else if (data[i]>x)
      return false
    return false;
  }
}
class IntervalSubdivisionFinder : IFinder {
  int[] data;
  void SetData(int[] data) throws Exception {
    for (int i = 1;i<data.Length;i++)
      if (data[i]<data[i-1])
        throw new Exception();
    this.data = data;
  }

  bool Find(int x) {...}
}
```

Problém

- obě třídy mají shodnou implementaci metody `SetData`
- pokud se v metodě najde chyba, musí se opravit na dvou (několika) místech

Problém

- obě třídy mají shodnou implementaci metody `SetData`
- pokud se v metodě najde chyba, musí se opravit na dvou (několika) místech

Řešení

zahrnout implementaci do rozhraní,

Problém

- obě třídy mají shodnou implementaci metody `SetData`
- pokud se v metodě najde chyba, musí se opravit na dvou (několika) místech

Řešení

zahrnout implementaci do rozhraní, což ale koncept rozhraní neumožňuje

Abstraktní třída

chová se podobně jako rozhraní

- říká co musí být implementováno (hlavičky metod)

Abstraktní třída

chová se podobně jako rozhraní

- říká co musí být implementováno (hlavičky metod)

může obsahovat i implementaci některých metod

Abstraktní třída

chová se podobně jako rozhraní

- říká co musí být implementováno (hlavičky metod)

může obsahovat i implementaci některých metod

nelze vytvořit instanci

- třída není úplná

Abstraktní třída

chová se podobně jako rozhraní

- říká co musí být implementováno (hlavičky metod)

může obsahovat i implementaci některých metod

nelze vytvořit instanci

- třída není úplná

ostatní třídy pak mohou od abstraktní třídy „dědit“

- musí poskytnout implementaci chybějících metod (jako u rozhraní)
- atributy abstraktní třídy se stanou součástí odděděné třídy
- metody implementované v abstraktní třídě se stanou součástí odděděné třídy

Abstraktní třída

```
abstract class AbstractFinder {
  int[] data;
  void SetData(int[] data) throws Exception {
    this.data = data;
    for (int i = 1;i<data.Length;i++)
      if (data[i]<data[i-1])
        throw new Exception();
  }
  abstract bool Find(int x);
}
```

Abstraktní třída

```
abstract class AbstractFinder {
  int[] data;
  void SetData(int[] data) throws Exception {
    this.data = data;
    for (int i = 1;i<data.Length;i++)
      if (data[i]<data[i-1])
        throw new Exception();
  }
  abstract bool Find(int x);
}

public class SequentialFinder : AbstractFinder {
  bool Find(int x) {
    for(int i = 0;i<data.Length;i++)
      if (data[i] == x)
        return true;
      else if (data[i]>x)
        return false
    return false;
  }
}
```

Abstraktní třída

```
abstract class AbstractFinder {
  int[] data;
  void SetData(int[] data) throws Exception {
    this.data = data;
    for (int i = 1;i<data.Length;i++)
      if (data[i]<data[i-1])
        throw new Exception();
  }
  abstract bool Find(int x);
}

public class SequentialFinder : AbstractFinder {
  bool Find(int x) {
    for(int i = 0;i<data.Length;i++)
      if (data[i] == x)
        return true;
      else if (data[i]>x)
        return false
    return false;
  }
}

public class IntervalSubdivisionFinder : AbstractFinder {
  bool Find(int x) {
  ...
  }
}
```

Použití abstraktní třídy

Abstraktní třídu je možné použít jako typ referenční proměnné

Použití abstraktní třídy

Abstraktní třídu je možné použít jako typ referenční proměnné

```
AbstractFinder finder;
```

Použití abstraktní třídy

Abstraktní třídu je možné použít jako typ referenční proměnné

```
AbstractFinder finder;
if (Console.ReadLine() == "b")
  finder = new SequentialFinder();
else
  finder = new IntervalSubdivisionFinder();
// zbytek kodu uz je bez vetveni
finder.SetData(new int[] {1, 2, 3});
bool isThereFifteen = finder.Find(15);
```

Použití abstraktní třídy

Abstraktní třídu je možné použít jako typ referenční proměnné

```
AbstractFinder finder;
if (Console.ReadLine() == "b")
  finder = new SequentialFinder();
else
  finder = new IntervalSubdivisionFinder();
// zbytek kodu uz je bez vetveni
finder.SetData(new int[] {1, 2, 3});
bool isThereFifteen = finder.Find(15);
```

Toto ale samozřejmě nelze:

```
AbstractFinder finder = new AbstractFinder();
finder.SetData(new int[] {1, 2, 3});
bool isThereFifteen = finder.Find(15);
```

Možnosti abstraktních tříd

```
public abstract class AbstractFinder {
  int[] data;
  void SetData(int[] data) throws Exception {
    ...
  }

  abstract bool Find(int x);

  double MeasureSearchTime(int count) {
    Random r = new Random();
    DateTime start = DateTime.Now;
    for (int i = 0;i<count;i++)
      Find(r.Next(data[data.Length-1]));
    return (DateTime.Now - start).TotalMilliseconds;
  }
}
```

Možnosti abstraktních tříd

```
public abstract class AbstractFinder {
  int[] data;
  void SetData(int[] data) throws Exception {
    ...
  }

  abstract bool Find(int x);

  double MeasureSearchTime(int count) {
    Random r = new Random();
    DateTime start = DateTime.Now;
    for (int i = 0;i<count;i++)
      Find(r.Next(data[data.Length-1]));
    return (DateTime.Now - start).TotalMilliseconds;
  }
}
```

abstraktní metodu je možné použít, ačkoli ještě není implementována

Dědičnost

Dědit lze i od třídy, která není abstraktní

- je možné dodat další funkcionalitu
- je možné upravit existující funkcionalitu (tzv. přepsání metody)
  - v předkovi musí být označena `virtual`
  - v potomkovi musí být označena `override`

Dědičnost

Dědit lze i od třídy, která není abstraktní

- je možné dodat další funkcionalitu
- je možné upravit existující funkcionalitu (tzv. přepsání metody)
  - v předkovi musí být označena `virtual`
  - v potomkovi musí být označena `override`

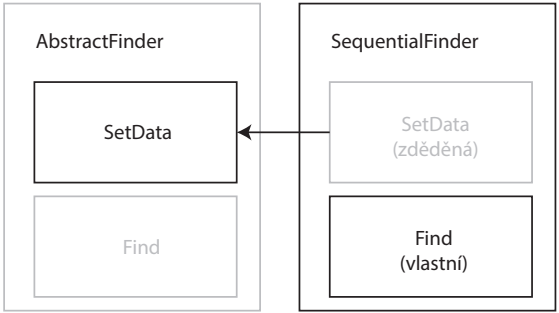
```
public abstract class AbstractFinder {
  int[] data;
  virtual void SetData(int[] data) throws Exception {
    ...
  }
  ...
```

Příklad na dědičnost

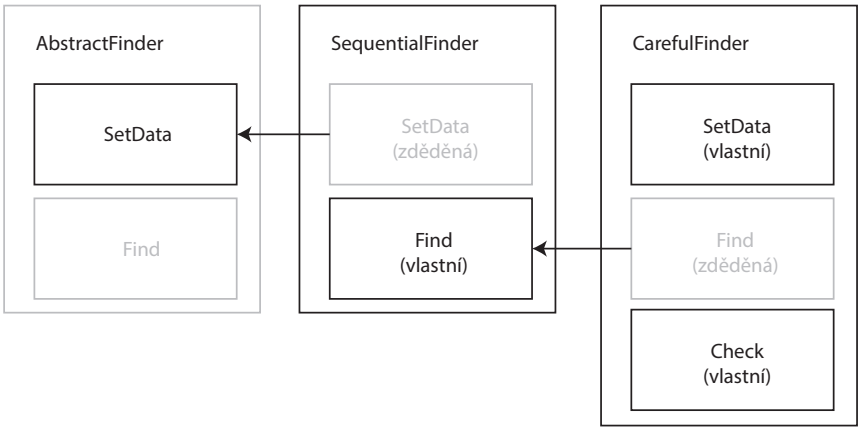
```
public class CarefulFinder : SequentialFinder {
  public override void SetData(int[] data) {
    this.data = (int[])data.Clone();
    Array.Sort(this.data);
  }

  public bool Check() {
    for (int i = 1;i<data.Length;i++)
      if (data[i]<data[i-1])
        return false;
    return true;
  }
}
```

Dědičnost

Dědičnost


AbstractFinder
SetData
Find
SequentialFinder
SetData
(zděděná)
Find
(vlastní)

Dědičnost


AbstractFinder
SetData
Find
SequentialFinder
SetData
(zděděná)
Find
(vlastní)
CarefulFinder
SetData
(vlastní)
Find
(zděděná)
Check
(vlastní)

Dědičnost

referenci na potomka je možné kdykoli „přetypovat“ na referenci na předka

- přetypovává se reference, nikoli instance samotná
- obráceně ne (runtime error)
- ...

Dědičnost

Pokud předka neuvedeme, je předkem třída `Object`

- `Object` je tudíž (pra)předkem každé třídy
- každou referenci je možné přetypovat na referenci na `Object`
- dědí se některé metody (uvidíme časem jaké a proč)

Dědičnost

Pokud předka neuvedeme, je předkem třída `Object`

- `Object` je tudíž (pra)předkem každé třídy
- každou referenci je možné přetypovat na referenci na `Object`
- dědí se některé metody (uvidíme časem jaké a proč)

Dědit lze vždy jen od jednoho předka!

- jinak hrozí kolize implementací

Referenční proměnné

```
CarefulFinder cf = new CarefulFinder();
```

Referenční proměnné

```
CarefulFinder cf = new CarefulFinder();
SequentialFinder sf = (SequentialFinder)cf;
```

Referenční proměnné

```
CarefulFinder cf = new CarefulFinder();
SequentialFinder sf = (SequentialFinder)cf;
SequentialFinder sf2 = cf;
```

Referenční proměnné

```
CarefulFinder cf = new CarefulFinder();
SequentialFinder sf = (SequentialFinder)cf;
SequentialFinder sf2 = cf;

SequentialFinder sf3 = new SequentialFinder();
```

Referenční proměnné

```
CarefulFinder cf = new CarefulFinder();
SequentialFinder sf = (SequentialFinder)cf;
SequentialFinder sf2 = cf;

SequentialFinder sf3 = new SequentialFinder();
CarefulFinder cf2 = sf3;
```

Referenční proměnné

```
CarefulFinder cf = new CarefulFinder();
SequentialFinder sf = (SequentialFinder)cf;
SequentialFinder sf2 = cf;

SequentialFinder sf3 = new SequentialFinder();
CarefulFinder cf2 = sf3; // compile error
```

Referenční proměnné

```
CarefulFinder cf = new CarefulFinder();
SequentialFinder sf = (SequentialFinder)cf;
SequentialFinder sf2 = cf;

SequentialFinder sf3 = new SequentialFinder();
CarefulFinder cf2 = sf3; // compile error
CarefulFinder cf3 = sf;
```

Referenční proměnné

```
CarefulFinder cf = new CarefulFinder();
SequentialFinder sf = (SequentialFinder)cf;
SequentialFinder sf2 = cf;

SequentialFinder sf3 = new SequentialFinder();
CarefulFinder cf2 = sf3; // compile error
CarefulFinder cf3 = sf; // compile error
```

Referenční proměnné

```
CarefulFinder cf = new CarefulFinder();
SequentialFinder sf = (SequentialFinder)cf;
SequentialFinder sf2 = cf;

SequentialFinder sf3 = new SequentialFinder();
CarefulFinder cf2 = sf3; // compile error
CarefulFinder cf3 = sf; // compile error
CarefulFinder cf4 = (CarefulFinder)sf;
```

Referenční proměnné

```
CarefulFinder cf = new CarefulFinder();
SequentialFinder sf = (SequentialFinder)cf;
SequentialFinder sf2 = cf;

SequentialFinder sf3 = new SequentialFinder();
CarefulFinder cf2 = sf3; // compile error
CarefulFinder cf3 = sf; // compile error
CarefulFinder cf4 = (CarefulFinder)sf; //OK
```

Referenční proměnné

```
CarefulFinder cf = new CarefulFinder();
SequentialFinder sf = (SequentialFinder)cf;
SequentialFinder sf2 = cf;

SequentialFinder sf3 = new SequentialFinder();
CarefulFinder cf2 = sf3; // compile error
CarefulFinder cf3 = sf; // compile error
CarefulFinder cf4 = (CarefulFinder)sf; //OK
CarefulFinder cf5 = (CarefulFinder)sf3;
```

Referenční proměnné

```
CarefulFinder cf = new CarefulFinder();
SequentialFinder sf = (SequentialFinder)cf;
SequentialFinder sf2 = cf;

SequentialFinder sf3 = new SequentialFinder();
CarefulFinder cf2 = sf3; // compile error
CarefulFinder cf3 = sf; // compile error
CarefulFinder cf4 = (CarefulFinder)sf; //OK
CarefulFinder cf5 = (CarefulFinder)sf3; // runtime error
```

Referenční proměnné

```
CarefulFinder cf = new CarefulFinder();
SequentialFinder sf = (SequentialFinder)cf;
SequentialFinder sf2 = cf;

SequentialFinder sf3 = new SequentialFinder();
CarefulFinder cf2 = sf3; // compile error
CarefulFinder cf3 = sf; // compile error
CarefulFinder cf4 = (CarefulFinder)sf; //OK
CarefulFinder cf5 = (CarefulFinder)sf3; // runtime error
```

Zjištění třídy - operátor is

```
Console.WriteLine(cf is CarefulFinder)
```

Referenční proměnné

```
CarefulFinder cf = new CarefulFinder();
SequentialFinder sf = (SequentialFinder)cf;
SequentialFinder sf2 = cf;

SequentialFinder sf3 = new SequentialFinder();
CarefulFinder cf2 = sf3; // compile error
CarefulFinder cf3 = sf; // compile error
CarefulFinder cf4 = (CarefulFinder)sf; //OK
CarefulFinder cf5 = (CarefulFinder)sf3; // runtime error
```

Zjištění třídy - operátor is

```
Console.WriteLine(cf is CarefulFinder) //true
```

Referenční proměnné

```
CarefulFinder cf = new CarefulFinder();
SequentialFinder sf = (SequentialFinder)cf;
SequentialFinder sf2 = cf;

SequentialFinder sf3 = new SequentialFinder();
CarefulFinder cf2 = sf3; // compile error
CarefulFinder cf3 = sf; // compile error
CarefulFinder cf4 = (CarefulFinder)sf; //OK
CarefulFinder cf5 = (CarefulFinder)sf3; // runtime error
```

Zjištění třídy - operátor is

```
Console.WriteLine(cf is CarefulFinder) //true
Console.WriteLine(sf is CarefulFinder)
```

Referenční proměnné

```
CarefulFinder cf = new CarefulFinder();
SequentialFinder sf = (SequentialFinder)cf;
SequentialFinder sf2 = cf;

SequentialFinder sf3 = new SequentialFinder();
CarefulFinder cf2 = sf3; // compile error
CarefulFinder cf3 = sf; // compile error
CarefulFinder cf4 = (CarefulFinder)sf; //OK
CarefulFinder cf5 = (CarefulFinder)sf3; // runtime error
```

Zjištění třídy - operátor is

```
Console.WriteLine(cf is CarefulFinder) //true
Console.WriteLine(sf is CarefulFinder) //true
```

Referenční proměnné

```
CarefulFinder cf = new CarefulFinder();
SequentialFinder sf = (SequentialFinder)cf;
SequentialFinder sf2 = cf;

SequentialFinder sf3 = new SequentialFinder();
CarefulFinder cf2 = sf3; // compile error
CarefulFinder cf3 = sf; // compile error
CarefulFinder cf4 = (CarefulFinder)sf; //OK
CarefulFinder cf5 = (CarefulFinder)sf3; // runtime error
```

Zjištění třídy - operátor is

```
Console.WriteLine(cf is CarefulFinder) //true
Console.WriteLine(sf is CarefulFinder) //true
Console.WriteLine(sf3 is CarefulFinder)
```

Referenční proměnné

```
CarefulFinder cf = new CarefulFinder();
SequentialFinder sf = (SequentialFinder)cf;
SequentialFinder sf2 = cf;

SequentialFinder sf3 = new SequentialFinder();
CarefulFinder cf2 = sf3; // compile error
CarefulFinder cf3 = sf; // compile error
CarefulFinder cf4 = (CarefulFinder)sf; //OK
CarefulFinder cf5 = (CarefulFinder)sf3; // runtime error
```

Zjištění třídy - operátor is

```
Console.WriteLine(cf is CarefulFinder) //true
Console.WriteLine(sf is CarefulFinder) //true
Console.WriteLine(sf3 is CarefulFinder) //false
```

Referenční proměnné

```
CarefulFinder cf = new CarefulFinder();
SequentialFinder sf = (SequentialFinder)cf;
SequentialFinder sf2 = cf;

SequentialFinder sf3 = new SequentialFinder();
CarefulFinder cf2 = sf3; // compile error
CarefulFinder cf3 = sf; // compile error
CarefulFinder cf4 = (CarefulFinder)sf; //OK
CarefulFinder cf5 = (CarefulFinder)sf3; // runtime error
```

Zjištění třídy - operátor is

```
Console.WriteLine(cf is CarefulFinder) //true
Console.WriteLine(sf is CarefulFinder) //true
Console.WriteLine(sf3 is CarefulFinder) //false
Console.WriteLine(cf is SequentialFinder)
```

Referenční proměnné

```
CarefulFinder cf = new CarefulFinder();
SequentialFinder sf = (SequentialFinder)cf;
SequentialFinder sf2 = cf;

SequentialFinder sf3 = new SequentialFinder();
CarefulFinder cf2 = sf3; // compile error
CarefulFinder cf3 = sf; // compile error
CarefulFinder cf4 = (CarefulFinder)sf; //OK
CarefulFinder cf5 = (CarefulFinder)sf3; // runtime error
```

Zjištění třídy - operátor is

```
Console.WriteLine(cf is CarefulFinder) //true
Console.WriteLine(sf is CarefulFinder) //true
Console.WriteLine(sf3 is CarefulFinder) //false
Console.WriteLine(cf is SequentialFinder) //true
```

Referenční proměnné

```
CarefulFinder cf = new CarefulFinder();
SequentialFinder sf = (SequentialFinder)cf;
SequentialFinder sf2 = cf;

SequentialFinder sf3 = new SequentialFinder();
CarefulFinder cf2 = sf3; // compile error
CarefulFinder cf3 = sf; // compile error
CarefulFinder cf4 = (CarefulFinder)sf; //OK
CarefulFinder cf5 = (CarefulFinder)sf3; // runtime error
```

Zjištění třídy - operátor is

```
Console.WriteLine(cf is CarefulFinder) //true
Console.WriteLine(sf is CarefulFinder) //true
Console.WriteLine(sf3 is CarefulFinder) //false
Console.WriteLine(cf is SequentialFinder) //true
Console.WriteLine(cf is AbstractFinder)
```

Referenční proměnné

```
CarefulFinder cf = new CarefulFinder();
SequentialFinder sf = (SequentialFinder)cf;
SequentialFinder sf2 = cf;

SequentialFinder sf3 = new SequentialFinder();
CarefulFinder cf2 = sf3; // compile error
CarefulFinder cf3 = sf; // compile error
CarefulFinder cf4 = (CarefulFinder)sf; //OK
CarefulFinder cf5 = (CarefulFinder)sf3; // runtime error
```

Zjištění třídy - operátor is

```
Console.WriteLine(cf is CarefulFinder) //true
Console.WriteLine(sf is CarefulFinder) //true
Console.WriteLine(sf3 is CarefulFinder) //false
Console.WriteLine(cf is SequentialFinder) //true
Console.WriteLine(cf is AbstractFinder) //true
```

Polymorfismus

instanci potomka lze použít všude kde se očekává předek

- parametr metody
- pole předků
- ...

```
void TestFinder(AbstractFinder f) {
  Console.WriteLine(f.MeasureSearchTime(1000));
}
```

Polymorfismus

instanci potomka lze použít všude kde se očekává předek

- parametr metody
- pole předků
- ...

```
void TestFinder(AbstractFinder f) {
  Console.WriteLine(f.MeasureSearchTime(1000));
}

AbstractFinder[] finders = new AbstractFinder[2];
finders[0] = new SequentialFinder();
finders[1] = new IntervalSubdivisionFinder();
for (int i = 0;i<finders.Length;i++) {
  finders[i].SetData(...);
  TestFinder(finders[i]);
}
```

Polymorfismus

funguje to i pro pole:

```
SequentialFinder[] finders = new SequentialFinder[2];
finders[0] = new SequentialFinder();
finders[1] = new SequentialFinder();
```

Polymorfismus

funguje to i pro pole:

```
SequentialFinder[] finders = new SequentialFinder[2];
finders[0] = new SequentialFinder();
finders[1] = new SequentialFinder();
AbstractFinder[] abstractFinders = finders;
```

Polymorfismus

funguje to i pro pole:

```
SequentialFinder[] finders = new SequentialFinder[2];
finders[0] = new SequentialFinder();
finders[1] = new SequentialFinder();
AbstractFinder[] abstractFinders = finders;
```

reference `finders` a `abstractFinders` teď odkazují na stejné pole.

Třídy s typovým parametrem

Problém

Často vytváříme podobné třídy, lišící se jen datovým typem některé z položek

- Příklad: dvojice (Pair)
- Dovede: přiřadit hodnotu (první, druhou), získat hodnotu (první, druhou), vyměnit hodnoty
  - `IntPair`
  - `StringPair`
  - `StudentPair`
  - ...

Problém

Často vytváříme podobné třídy, lišící se jen datovým typem některé z položek

- Příklad: dvojice (Pair)
- Dovede: přiřadit hodnotu (první, druhou), získat hodnotu (první, druhou), vyměnit hodnoty
  - `IntPair`
  - `StringPair`
  - `StudentPair`
  - ...

Problém

- většina zdrojového kódu je shodná
- změny se obtížně udržují na několika místech

Řešení 1

Kolekce pro typ `Object`

```
public class ObjectPair {
  Object item1, item2;
  ...
  public void SetItem1(Object i) {...}
  public Object GetItem1() {...}
  public void SetItem2(Object i) {...}
  public Object GetItem2() {...}
  public void SwapItems() {...}
  public ObjectPair() {...}
  ...
}
```

- instanci jakékoli třídy je možné přetypovat na typ `Object`

Použití

```
ObjectPair pair = new ObjectPair();
String s1 = "Zvahlav";
pair.SetItem1(s1);
String s2 = "Nekav";
pair.SetItem2(s2);
Console.WriteLine(pair.GetItem1());
pair.SwapItems();
String s3 = (String)pair.GetItem2();
```

Použití

```
ObjectPair pair = new ObjectPair();
String s1 = "Zvahlav";
pair.SetItem1(s1);
String s2 = "Nekav";
pair.SetItem2(s2);
Console.WriteLine(pair.GetItem1());
pair.SwapItems();
String s3 = (String)pair.GetItem2();
```

kde se očekává `Object`, tam je možné použít instanci jakékoli třídy

Použití

```
ObjectPair pair = new ObjectPair();
String s1 = "Zvahlav";
pair.SetItem1(s1);
String s2 = "Nekav";
pair.SetItem2(s2);
Console.WriteLine(pair.GetItem1());
pair.SwapItems();
String s3 = (String)pair.GetItem2();
```

kde se očekává `Object`, tam je možné použít instanci jakékoli třídy

- `SetItem1()` očekává `Object`
- `WriteLine()` si umí poradit s `Object`em

Použití

```
ObjectPair pair = new ObjectPair();
String s1 = "Zvahlav";
pair.SetItem1(s1);
String s2 = "Nekav";
pair.SetItem2(s2);
Console.WriteLine(pair.GetItem1());
pair.SwapItems();
String s3 = (String)pair.GetItem2();
```

kde se očekává `Object`, tam je možné použít instanci jakékoli třídy

- `SetItem1()` očekává `Object`
- `WriteLine()` si umí poradit s `Object`em

kde se očekává `String` (nebo jiná specifická třída), tam je nutné provést přetypování

Problém tohoto řešení

- do páru je možné vložit instance různých tříd
- při vybrání musí klientský program provést správné přetypování
- pokud je v přetypování chyba, pak se projeví až za běhu programu (ne při překladu)

Problém tohoto řešení

- do páru je možné vložit instance různých tříd
- při vybrání musí klientský program provést správné přetypování
- pokud je v přetypování chyba, pak se projeví až za běhu programu (ne při překladu)

```
ObjectPair pair = new ObjectPair();
int[] array = new int[5];
pair.SetItem1(array);
```

Problém tohoto řešení

- do páru je možné vložit instance různých tříd
- při vybrání musí klientský program provést správné přetypování
- pokud je v přetypování chyba, pak se projeví až za běhu programu (ne při překladu)

```
ObjectPair pair = new ObjectPair();
int[] array = new int[5];
pair.SetItem1(array);
String s = "Orlosup␣dravy";
pair.SetItem2(s);
```

Problém tohoto řešení

- do páru je možné vložit instance různých tříd
- při vybrání musí klientský program provést správné přetypování
- pokud je v přetypování chyba, pak se projeví až za běhu programu (ne při překladu)

```
ObjectPair pair = new ObjectPair();
int[] array = new int[5];
pair.SetItem1(array);
String s = "Orlosup␣dravy";
pair.SetItem2(s);

String bird = (String)pair.GetItem1();
pair.SwapItems();
int[] numbers = (int[])pair.GetItem1();
```

Problém tohoto řešení

- do páru je možné vložit instance různých tříd
- při vybrání musí klientský program provést správné přetypování
- pokud je v přetypování chyba, pak se projeví až za běhu programu (ne při překladu)

```
ObjectPair pair = new ObjectPair();
int[] array = new int[5];
pair.SetItem1(array);
String s = "Orlosup␣dravy";
pair.SetItem2(s);

String bird = (String)pair.GetItem1();
pair.SwapItems();
int[] numbers = (int[])pair.GetItem1();
```

Pokud se volání `SwapItems()` neprovede, pak se program bez problému přeloží, ale běh skončí vyjímkou

Řešení: obalovací třída

Obalovací třída (wrapper)

- poskytuje metody, které provádějí přetypování na správnou třídu
- skrývá vnitřní `ObjectStack`

Řešení: obalovací třída

Obalovací třída (wrapper)

- poskytuje metody, které provádějí přetypování na správnou třídu
- skrývá vnitřní `ObjectStack`

```
public class StringPair {
  private ObjectPair innerPair;

  public void SetItem1(String s) {
    innerPair.SetItem1(s)
  }
  public String GetItem1() {
    return (String)innerStack.GetItem1();
  }
  public StringPair() {
    innerStack = new ObjectPair();
  }
  ...
}
```

Obalovací třída

- metody třídy `StringPair` umožňují jen práci s referencemi na `String`

Obalovací třída

- metody třídy `StringPair` umožňují jen práci s referencemi na `String`
- klientská třída nemá přístup k vnitřní třídě reprezentující pár objektů

Obalovací třída

- metody třídy `StringPair` umožňují jen práci s referencemi na `String`
- klientská třída nemá přístup k vnitřní třídě reprezentující pár objektů
- obalovací třídu je nutné vyrobit pro každou specifickou třídu prvku
  - implementace metod je triviální
  - změny implementace páru se provádějí na jednom místě ve třídě `ObjectPair`

StringPair

```
public class StringPair {
  private ObjectPair innerPair;

  public void SetItem1(String s) {
    innerPair.SetItem1(s)
  }
  public String GetItem1() {
    return (String)innerPair.GetItem1();
  }
  ...
  public StringPair() {
    innerPair = new ObjectPair();
  }
}

StringPair sPair = new StringPair();
sPair.SetItem1("Nejneobhospodarovavatelnejsimi");
```

PersonStack

```
public class PersonPair {
  private ObjectPair innerPair;

  public void SetItem1(Person p) {
    innerPair.SetItem1(p)
  }
  public Person GetItem1() {
    return (Person)innerPair.Get();
  }
  ...
  public PersonPair() {
    innerPair = new ObjectPair();
  }
}

PersonPair pPair = new PersonPair();
pPair.SetItem1(new Person("Bucivoj␣Bral"));
```

BrushStack

```
public class BrushPair {
  private ObjectPair innerPair;

  public void SetItem1(Brush b) {
    innerStack.SetItem1(b)
  }
  public Brush GetItem1() {
    return (Brush)innerPair.GetItem1();
  }
  ...
  public BrushPair() {
    innerPair = new ObjectPair();
  }
}

BrushPair bPair = new BrushPair();
bPair.SetItem1(new Brush(3));
```

Otázka

Nešlo by definovat formu, ze které by se vytvářely třídy lišící se pouze nějakým datovým typem?

Třída s typovým parametrem

```
public class MyPair<T> {
  private ObjectPair innerPair;

  public void SetItem1(T t) {
    innerPair.SetItem1(t)
  }
  public T GetItem1() {
    return (T)innerStack.GetItem1();
  }
  ...
  public MyPair() {
    innerPair = new ObjectPair();
  }
}
```

Třída s typovým parametrem

```
public class MyPair<T> {
  private ObjectPair innerPair;

  public void SetItem1(T t) {
    innerPair.SetItem1(t)
  }
  public T GetItem1() {
    return (T)innerStack.GetItem1();
  }
  ...
  public MyPair() {
    innerPair = new ObjectPair();
  }
}

MyPair<String> sPair = new MyPair<String>();
sPair.SetItem1("Olovo");
```

Třída s typovým parametrem

```
public class MyPair<T> {
  private ObjectPair innerPair;

  public void SetItem1(T t) {
    innerPair.SetItem1(t)
  }
  public T GetItem1() {
    return (T)innerStack.GetItem1();
  }
  ...
  public MyPair() {
    innerPair = new ObjectPair();
  }
}

MyPair<String> sPair = new MyPair<String>();
sPair.SetItem1("Olovo");
MyPair<Person> pPair = new MyPair<Person>();
pPair.SetItem1(new Person("Milivoj␣Tydlitat"));
```

Třída s typovým parametrem

```
public class MyPair<T> {
  private ObjectPair innerPair;

  public void SetItem1(T t) {
    innerPair.SetItem1(t)
  }
  public T GetItem1() {
    return (T)innerStack.GetItem1();
  }
  ...
  public MyPair() {
    innerPair = new ObjectPair();
  }
}

MyPair<String> sPair = new MyPair<String>();
sPair.SetItem1("Olovo");
MyPair<Person> pPair = new MyPair<Person>();
pPair.SetItem1(new Person("Milivoj␣Tydlitat"));
MyPair<Brush> bPair = new MyPair<Brush>();
bPair.SetItem1(new Brush(8));
```

Generické třídy

Třídám s typovým parametrem se také říká generické třídy

Generické třídy

Třídám s typovým parametrem se také říká generické třídy

Můžeme vytvořit generickou obalovací třídu, nebo rovnou generický pár

Generický pár

```
class Pair<T> {
  T item1;
  T item2;

  public Pair(T i1, T i2) {
    this.item1 = i1;
    this.item2 = i2;
  }
```

Generický pár

```
class Pair<T> {
  T item1;
  T item2;

  public Pair(T i1, T i2) {
    this.item1 = i1;
    this.item2 = i2;
  }
  public T GetItem1() {
    return item1;
  }
```

Generický pár

```
class Pair<T> {
  T item1;
  T item2;

  public Pair(T i1, T i2) {
    this.item1 = i1;
    this.item2 = i2;
  }
  public T GetItem1() {
    return item1;
  }
  public void SetItem1(T i1) {
    item1 = i1;
  }
```

Generický pár

```
class Pair<T> {
  T item1;
  T item2;

  public Pair(T i1, T i2) {
    this.item1 = i1;
    this.item2 = i2;
  }
  public T GetItem1() {
    return item1;
  }
  public void SetItem1(T i1) {
    item1 = i1;
  }
  public void SwapItems() {
    T tmp = item1;
    item1 = item2;
    item2 = tmp;
  }
  ...
}
```

Omezení typových parametrů

Problém

Co když potřebuji, aby typový parametr něco uměl?

Omezení typových parametrů

Problém

Co když potřebuji, aby typový parametr něco uměl?

Definujeme rozhraní (co má umět)

```
public interface IProcessable {
  void Process();
}
```

Omezení typových parametrů

Problém

Co když potřebuji, aby typový parametr něco uměl?

Definujeme rozhraní (co má umět)

```
public interface IProcessable {
  void Process();
}
```

Určíme, jaká rozhraní má typový parametr implementovat

```
class ProcessingPair<T : IProcessable> {
```

Omezení typových parametrů

Problém

Co když potřebuji, aby typový parametr něco uměl?

Definujeme rozhraní (co má umět)

```
public interface IProcessable {
  void Process();
}
```

Určíme, jaká rozhraní má typový parametr implementovat

```
class ProcessingPair<T : IProcessable> {
  T item1;
  T item2;
```

Omezení typových parametrů

Problém

Co když potřebuji, aby typový parametr něco uměl?

Definujeme rozhraní (co má umět)

```
public interface IProcessable {
  void Process();
}
```

Určíme, jaká rozhraní má typový parametr implementovat

```
class ProcessingPair<T : IProcessable> {
  T item1;
  T item2;
  ...
  void processItems() {
    item1.Process();
    item2.Process();
  }
}
```

U instancí třídy `T` pak můžeme volat metody rozhraní.

Příště

- výpočetní složitost