Datové abstrakce v programovacích jazycích

Motivace

•Strukturovat rozsáhlé programy

•Dovolit separátní překlad

Možné formy strukturování:

•Podprogramy – původní forma abstrakce -abstrakce výpočtů

•Moduly – kontejnery vzájemně souvisejících podprogramů a dat (od 80. let běžná forma datové abstrakce)

•Kompilační jednotky – kolekce podprogramů a dat přeložitelných bez nutnosti současně překládat zbytek programu

Datové abstrakce v programovacích jazycích

Zapouzdření = seskupení logicky souvisejících podprogramů do celků, které lze obvykle separátně překládat Nap**ř. :**

- Vnořované podpr. (Pascal),
- Soubory s podpr. (C a Fortranu lze samostatně překládat), **př 1FrontaC** verze int a double.
- Package Ada, Java, Fortran90, **př. 2STACKADA**

Abstraktní datový typ je zapouzdření datového typu a podprogramů poskytujících operace pro tento typ.

- ✓ Je použitelný k deklaraci proměnných
- ✓ Skutečná reprezentace je uživateli skryta
- ✓ Instanci ADT nazýváme objektem

Datové abstrakce v programovacích jazycích

ADT splňuje podmínky:

1. Definice typu a operací nad ním je obsažena v jedné syntaktické jednotce
2. Reprezentace objektů tohoto typu je ukryta před programovými jednotkami, které jej využívají

ad 1 splňuj package ADA
třídy C++
třídy Java

ad 2 splňují dtto pomocí konstrukcí private

sloužící jako interface značené public
pro styk s příbuznými protected

Výhody ad 1:

- Lepší organizace programu
- Lepší modifikovatelnost programu
- Separátní překlad

Výhody ad 2:

- Spolehlivější – důsledkem ukrytí dat
- Nezávislost uživatele na konkrétní implementaci ADT

Je typ int jazyka C také ADT ?

Reprezentace je skrytá

Operace jsou vestavěné

Uživatel může definovat *objekty* typu int

Uživatelem definované ADT musí mít stejné vlastnosti jako vestavěné ADT

Jazykové požadavky na ADT:

1. Syntaktická konstrukce pro zapouzdření definice typu
2. Prostředek pro vytváření jmen typů a podprogramů viditelných klientům, při zakrytí aktuálních definic.
3. Určité primitivní operace jsou zabudovány do jazyka (většinou jen přiřazování a test rovnosti) Další potřebné operace definuje návrhář ADT (konstruktory, destruktory, …)

př. zásobník

abstraktní operace
- vytvoř(zásobník)
- zruš(zásobník)
- vlož(zásobník, element)
- vyber(zásobník)
- vrchol(zásobník)
- prázdný(zásobník)

využíván

- ADT v Adě — Př. 2StackSeparADA
- ADT v C++ — Př. 3Stack1CPP
- ADT v Javě — př. 4StackJava

Ad ADA

Zapouzdřující konstrukcí je package

Má obvykle dvě části

1. Package specification (tj. interface)
2. Package body (implementace v 1. specifikovaných prostředků)

- Exportovat lze libovolný typ
- Ukrytí typu se zajistí ve specifikaci konstrukcí

  type JMENO is private;
- Neprivátní typy mají uživateli zpřístupněnou strukturu

Ad ADA

private klauzule je „zneviditelněna“ klientovi

```
package JMENO_BALIKU  is
   type JM_TYPU is private;
        …
   type JM_TYPU is
        record
        …
        end record;

End JMENO_BALIKU;
```

Ad ADA

Důvody pro privátní a neprivátní definice:

1. Kompilátor musí znát strukturu typů po prohlédnutí specif. části (pro separátní překlad
2. Klient musí vidět jméno typu, ale nesmí „vidět“ klauzuli private

-private typy vlastní operace :=, =, /=

-limited private typy nemají žádné operace (všechny je třeba dodefinovat).

-ADT lze parametrizovat proměnnou

Př.5Stack2ADA

Ad C++

- Založeno na C struct a Simula class (plus ukrývání)
- Prostředkem zapouzdření je třída (t.j. typ)
- Všechny instance třídy sdílí jednu kopii členské funkce
- Každá instance třídy má svou vlastní kopii členských dat
- Instance mohou být statické, dynamické v zásobníku, dynamické na haldě
- členská data mohou být i na haldě (new/delete)

Ad C++

- Členské funkce mohou být *inlined* (hlavička i tělo fce jsou v definiční části třídy, realizace uzavřeného podprogramu)
- Ukrývání formou:

  -private klazule *skryté entity*

  -public klauzule *entity tvořící interface*

  protected klauzule *viz příště dědičnost*

  př. 6Stack1CPP

Ad C++

Konstruktory

- Funkce inicializující členská data
- Nevytváří objekty, mohou ale alokovat paměť je-li část objektu dynamická na haldě
- Mohou mít parametry = parametrizovat objekty
- Jméno mají stejné jako třída
- Mohou být volány explicitně
- Implicitně jsou volány při vytváření instance

  ještě př. 6Stack1CPP

Ad C++

- Destruktory

-funkce úklidu po zrušení instance (obv. z haldy)

-implicitně volané, když končí život objektu

-mohou být volány explicitně

-jménem je ~JmenoTridy

- Friend funkce a třídy dovolí přístup k nepříbuzným jednotkám a funkcím
- Rozdíl package versus class:

  class definuje typ

  package definuje zapouzdření

ADT C++ lze parametrizovat proměnnou

Viz př.7Stack2CPP

Ad C++ mechanismus namespace

Umožňuje explicitní určení rozsahu platnosti jmen pro vyloučení kolizí jmen z různých separátně překládaných knihoven

Např. v 1Fronta.C je-li ADT Queue součástí více knihoven, které jsou uvedeny v příkazu include, pak nutno zjednoznačnit kvalifikací v queue.cpp

kvalifikace jménem prostoru

```
struct MyQueue::Queuerep
{ void* data;
  Queue next;
};
```

V queue.h je specifikován prostor zápisem

Ad C++ mechanismus namespace

```
#ifndef QUEUE_H  /*zamezi opakovani natazeni*/
#define QUEUE_H
namespace MyQueue
{struct Queuerep;
 typedef struct Queuerep * Queue;

 Queue createq(void);
 Queue enqueue(Queue q, void* elem);
 void* frontq(Queue q);
 Queue dequeue(Queue q);
 int emptyq(Queue q);
}
#endif
```

Klient pak může použít

- Klauzuli using namespace JmenoProstoru;
- Kvalifikaci jménem prostoru

```
#include <stdlib.h>
#include <stdio.h>
#include "queue.h„
using namespace MyQueue;
main()
{  int *x = new int ;
   int *y = new int ;
   int *z;
   Queue q = MyQueue::createq(); // zde jiz neni nutna
   *x = 21;
   *y = 39;
   q = enqueue(q,x);    . . .
```

Ad Java

Obdoba C++ kromě

-všechny uživatelem definované typy jsou třídami

-všechny objekty jsou umístěny na haldě a zpřístupněny referenční proměnnou

-jednotlivé entity tříd mohou mít modifikátory přístupu (private/public)

-Java má druhou úroveň řízení rozsahu platnosti – package (slouží místo namespaces v C++). Jiný význam než ADA

-všechny entity všech tříd balíku nemají-li modifikátor, jsou viditelné v celém balíku.

-soubory balíku jsou umístěny v adresáři stejného jména,

začínají klauzulí package jméno;

Př 4StackJava

Ad Java

- Balíky jsou seskupeními tříd
- Každý balík je ve vlastním adresáři
- Příkaz package může obsahovat více souborů, určuje pouze, kterému balíku patří třídy definované v souboru
- Třídy definované uvnitř balíku musí být zpřístupněny jménem balíku. Např.mějme package myqueue;

  pak lze odkazovat na public jméno:

  explicitně plným jménem: myqueue.Queue

  nebo zviditelnit deklarací: import myqueue.Queue ;

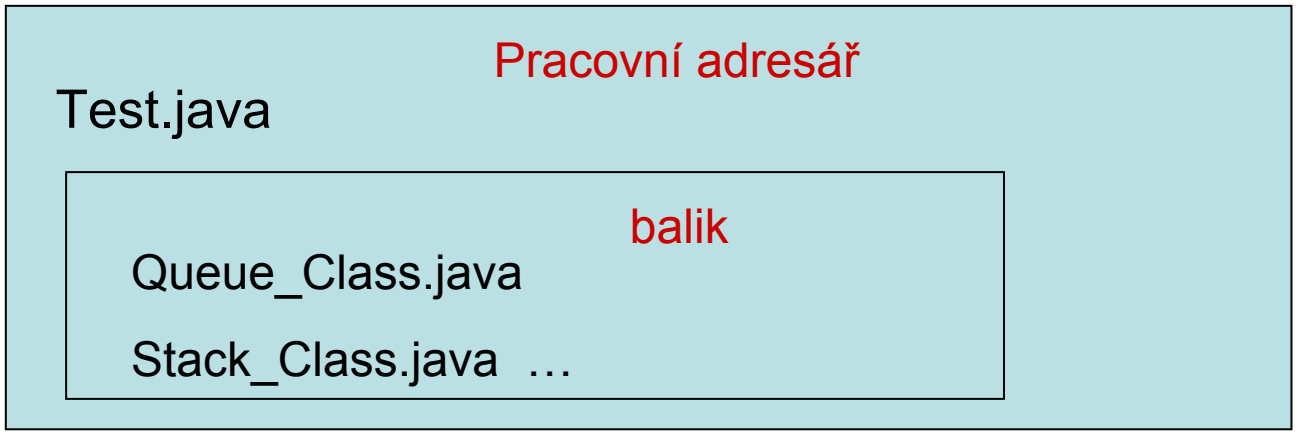
  nebo importováním všech: import myqueue.* ;
- Každý separátně překládaný soubor Javy smí mít jen 1 public třídu

Ad Java

- Balíky (jako třídy) mohou vytvářet hierarchie

  package x.y.z; musí být pak v directory …/x/y/z
- Nepoužije-li se příkaz package, použije se implicitní balík (nemá žádné jméno)
- Aby hlavní program našel balík který použije, musí buď - být nastavena syst. proměnná CLASSPATH

  -nebo je umístěn v bezprostředně nadřazeném adresáři



Ad Java balíky a přístup k metodám a proměnným

- Viditelnost každého prvku Javy je dána modifikátory (specifikátory) přístupu *private, public, protected nebo nic*
- Viditelnost určuje viditelnost uvnitř třídy a viditelnost třídy uvnitř balíku
- Metody a proměnné označené public jsou viditelné všude včetně různých tříd v různých balících. Public třída je přístupná i mimo svůj balík
- Privátní proměnné a metody jsou přístupné jen metodám své třídy
- Chráněná proměnná a metoda je přístupná i v potomcích své třídy (i pokud jsou potomci v jiných balících) a ve třídách ze stejného balíku
- Když nemá proměnná, metoda, třída žádný modifikátor, pak uvnitř třídy viditelná

  je a vně třídy viditelná

  není s výjimkou tříd ze stejného balíku, kde viditelná je
- Implicitní přístup proto užívají takové elementy, které mají být privátní vně balíku, ale veřejné uvnitř balíku

Ad Java balíky a přístup k metodám a proměnným

Modifikátor	v téže třídě	v potomkovi	v tomtéž balíku	v podtřídách různých balíků	odkudkoliv
Public	+	+	+	+	+
Implicitní	+	+	+	-	-
Protected	+	+	+	+	-
Private	+	-	-	-	-

Ad Java balíky

Java Application Programming Interface – hierarchie knihoven

Vrchol = balík java

Z něj odvozeny:

- java.lang různé obecně použitelné třídy, (např třída Systém)
  importuje se automaticky do každého programu,
- java.io Třídy pro vstup/výstup
- java.net Třídy pro práci v síti
- java.applet „ „ konstrukci appletů
- java.awt „ pro podporu systému Abstr. Wind. Toolkit
- . . .

ADT C#

- Založené na C++ a Java principech
- Používá modifikátory private, public, protected téměř přesně jako Java
- Přidal dva přístupové modifikátory, internal (viditelné všem třídám balíku) a protected internal (protectedUinternal)
- Všechny instance class jsou dynamické v haldě
- Defaultní konstruktory jsou k dispozici pro všechny třídy a pokud to neprovede uživatelský konstruktor, tak inicializují instance proměnných standardní hodnotou
- Garbage collection se používá pro většinu objektů v haldě, proto destruktory se užívají řídce
- `structs` jsou třídami, které nepodporují dědičnost, mají ale konstruktory, metody, vlastnosti, datové položky. Jsou alokovány v RT zásobníku (ne v haldě)

ADT C#

- Zavádí „assemblies“´= „balíky“, zapouzdřovací prostředky mocnější než třídy
- Balík (assembly) je kolekce souborů, která se jeví aplikačním programům jako spustitelný (exe) soubor nebo jako jedna DLL
- DLL je kolekcí tříd a metod, které jsou individuálně propojovány s prováděným programem, jsou-li potřebné při jeho exekuci
- Každý soubor z balíku definuje modul, který může být separátně zpracován
- C# má modifikátor přístupu `internal`; pak `internal` člen třídy je viditelný všem třídám balíku ve kterých se vyskytne

Generické ADT

Umožňují tzv parametrický polymorfismus = šablona
= násobné využití operace nad různými typy
Př. operace se zásobníkem pro int, bool, záznam, …

Obecně existují 3 formy polymorfismu:

1. přetížení
2. parametrický
3. objektový

- Generické moduly ADA
- Generické třídy C++
- Generické typy Java (od 1.5 omezené parametrizování)
- C # parametrický polymorfismus zavedl od vánoc

Generické ADT - ADA

```
generic
      generické parametry
package Jméno is                    --specifik.cast package
      exportované prostředky
end Jméno;

package body Jméno is               --implementace package
      exportované prostředky a jejich těla
end Jméno;
```

Generické ADT - ADA

generické parametry:

- Numerické typy (skutečný parametr je odvozený z integer, fixed a float)
- Diskrétní typ (skutečným par. je libovolný diskrétní typ)
- Privátní typ („ ----------“ typ dovolující operace přiřazení, rovnosti a nerovnosti. Ostatní potřebné operace je nutno dodefinovat)
- Obecný typ (skut. par. může být jakýkoliv typ, všechny operace nutno ale naprogramovat)

Př. 8GenerSTACKADA

Generické ADT C++

- Generické třídy – popisují obecný algoritmus,
- konkrétní typ dat bude určen jako parametr při vytváření objektu.

Obecný formát:

template <class Ttyp1, Ttyp2, …> class jmeno-tridy {

. . .

}

Ttyp je jméno generického (obecného) typu

Objekt je vytvořen příkazem

jméno-třídy <typ1, typ2, …> objekt;

Kde typ je jméno konkrétního typu

Členské fce gen. třídy jsou automaticky generické (viz jindy

Generické ADT C++

- Stejně jako ADA instaluje C++ generické třídy v době překladu
- Nová instalace je vytvořena kdykoliv je vytvářen objekt, který požaduje dosud neexistující verzi generické třídy

Př. 9GenerStack3CPP

Generické datové typy a metody - Java

- **Generický=parametrizovaný typ Javy je typ (třída / interface), který má formální parametr(y) v podobě typu(ů). Je parametrizován typem.**
- **Formální typy jsou při instalaci typu nahrazeny skutečnými typy**
- **Běžnými generickými typy jsou kontejnery typů jako** ArrayList **a** LinkedList**, které jsou součástí Javy před v.5.0**
- **Neumožńuje se jakýkoliv typ, generickými parametry musí být třídy. Nelze jednoduše přeprogramovat příklady z C++ a Ady**
- **Genericita se projevuje jako automatická konverze.**

Generické datové typy a metody - Java

Konvence značení typových parametrů

- **<T>** **Typ**
- **<S>** **také typ, je-li T již použito**
- **<E>** **Element, využíváno hlavně v Java Collection Framework**
- **<K>** **Klíč**
- **<V>** **Value**
- **<N>** **Number**

Formální typové parametry se deklarují v hlavičce třídy za názvem třídy a uzavírají se do špičatých závorek

Skutečné typové parametry se uvádějí v konstruktoru za jménem třídy ve špičatých závorkách

Př 91GenerJava Prumery.java

```
class Prumerovac<T extends Number> {
// pole prvku Number podtypu
  T[] poleCisel;
  Prumerovac(T[] o) {//konstruktor dostane odkaz na pole
    poleCisel = o;
  }
  double prumer() {
    double suma = 0.0;
    for(int i=0; i < poleCisel.length; i++)
      suma += poleCisel[i].doubleValue();//metoda tridy Double
//metoda vraci vysledek typu double
    return suma / poleCisel.length;
  }
}
```

```
// Hlavni program
public class Prumery {
  public static void main(String args[]) {

    Integer ipoleCisel[] = { 40, 30, 20, 10 };
    Prumerovac<Integer> iobjekt = new Prumerovac<Integer>(ipoleCisel);
    double vysledek = iobjekt.prumer();
    System.out.println("prumer iobjektu je " + vysledek);

    Double dpoleCisel[] = { 10.1, 20.1, 30.1, 40.1, 50.1 };
    Prumerovac<Double> dobjekt = new Prumerovac<Double>(dpoleCisel);
    double wysledek = dobjekt.prumer();
    System.out.println("prumer dobjektu je " + wysledek);
    Float fpoleCisel[] = { 10.1f, 20.1f, 30.1f, 40.1f, 50.1f };

    Prumerovac<Float> fobjekt = new Prumerovac<Float>(fpoleCisel);
    double fysledek = fobjekt.prumer();
    System.out.println("prumer fobjektu je " + fysledek);
  }
}
```

Generické datové typy a metody - Java

Generické (parametrizované) metody

- Parametrizovaná metoda ≠ metoda s parametry
- „ „ „ „ ≠ přetížená metoda
- Má typové parametry, které informují překladač o skutečných parametrech a návratové hodnotě při volání metody
- Jsou javovskou obdobou generických podprogramů a generických fcí jiných jazyků
- Typové parametry se uvádí v definici metody před typem návratové hodnoty
- Při volání generické metody překladač většinou odvodí správný typový parametr z typu parametrů metody, pak typové parametry se již neuvádí.

Př.91GenerMetoda

```
//Zjistuje vyskyt prvku v poli
public class GenerMetoda {
  static <T, V extends T> boolean obsahuje(T x, V[] y) {
    for(int i=0; i < y.length; i++)
      if(x.equals(y[i])) return true;
    return false;
  }
  public static void main(String args[]) {
    //obsahuje() v Integer poli
    Integer cisla[] = { 10, 20, 30, 40, 50 };
    if(obsahuje(20, cisla))
      System.out.println("20 tam je");
    if(!obsahuje(99, cisla))
      System.out.println("99 tam neni");
    System.out.println();
    //obsahuje() v String poli
    String jmena[] = { "Jan", "Marie", "Eduard", "Kuba" };
    if(obsahuje("Kuba", jmena))
      System.out.println("Kuba tam je");
    if(!obsahuje("Filip", jmena))
      System.out.println("Filip tam neni");
  }
}
```

```
public class PretizeneMetody     {

      // Metoda tiskPole tiskne pole Integer
      public static void tiskPole( Integer[] inputArray )   {
         // tisk prvku
        for ( Integer element : inputArray )
           System.out.printf( "%s ", element );
        System.out.println();
     } // konec metody tiskPole

     // Metoda tiskPole tiskne pole Double
     public static void tiskPole( Double[] inputArray )  {
        // display array elements
        for ( Double element : inputArray )
           System.out.printf( "%s ", element );
        System.out.println();
     } // konec metody tiskPole

     // Metoda tiskPole tiskne pole Character
     public static void tiskPole( Character[] inputArray )   {
        // display array elements
        for ( Character element : inputArray )
           System.out.printf( "%s ", element );
        System.out.println();
     } // konec metody tiskPole
```

```
public static void main( String args[] )
      {
        // vytvori pole Integer, Double a Character
        Integer[] integerArray = { 1, 2, 3, 4, 5, 6 };
        Double[] doubleArray = { 1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7 };
        Character[] characterArray = { 'A', 'H', 'O', 'J' };

        System.out.println("Pole integerArray obsahuje:" );
        tiskPole( integerArray ); // predava pole Integer
        System.out.println( "\nPole doubleArray obsahuje:" );
        tiskPole( doubleArray ); // predava pole Double
        System.out.println( "\nPole characterArray obsahuje:" );
        tiskPole( characterArray ); // predava pole Character
      } // end main
   } // end class PretizeneMetody
```

```
public class GenerickaMetoda     {
      // genericka methoda tiskPole
      public static < E > void tiskPole( E[] inputArray )       {
        // zobraz prvky pole
        for ( E element : inputArray )
          System.out.printf( "%s ", element );

        System.out.println();
      } // end methody tiskPole

      public static void main( String args[] )        {
        // vytvor pole Integer, Double a Char
        Integer[] intArray = { 1, 2, 3, 4, 5, 6 };
        Double[] doubleArray = { 1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7 };
        Character[] charArray = { 'A', 'H', 'O', 'J' };

        System.out.println( „Pole integerArray obsahuje:" );
        tiskPole( intArray ); // preda Integer pole
        System.out.println( "\nPole doubleArray obsahuje:" );
        tiskPole( doubleArray ); // preda Double pole
        System.out.println( "\nPole characterArray obsahuje:" );
        tiskPole( charArray ); // preda pole znaku
      } // end main
    } // end class GenerickaMetoda
```

Generické datové typy a metody - Java

Java 5.0 podporuje *wildcard types (žolíky).*

Zápis <?> má význam jako <NejakyTyp> = libovolný typ vyhovující případným podmínkám manipulace s ním.

Parametrizovaný typ, který má žolík jako typový parametr, je pak považován za rodičovský typ všech typů, v nichž byl za ? dosazen konkrétní typ.

Např.

```
void tisknikolekci(Collection<?> c) {
    for (Object e: c)        { System.out.println(e);
    }
}
```

Bude tisknout prvky jakékoliv kolekce bez ohledu na třídu jejích komponent