TDDC77 Objektorienterad programmering
Laborationsinstruktioner period 2

Ahmed Rezine (ahmed.rezine@liu.se)
Bearbetning av tidigare versioner av Johan Jernlås, John Edvardsson, John Wilander och Jonas Wallgren

November 5, 2013

1 Laboration 4 – Ett par småprogram

Den här laborationen är främst till för att ni skall komma igång ordentligt. Men den illustrerar också lösningar på ett klassiskt problem som är olika effektiva.

NumberPrinter

Skapa en klass `NumberPrinter` i paketet `lab1`. Ta i `main()` emot ett positivt heltal n från användaren och använd sedan n som parameter till en metod som skriver ut en $4 \times n$-tabell innehållande alla heltal mellan 1 och n i formerna: decimalt, binärt, oktalt och hexadecimalt. Kolumnerna ska var högerjusterade och ha samma bredd. Nedan ges ett exempel där $n = 16$.

```
Enter the number of rows: 16
        Decimal          Binary           Octal     Hexadecimal

              1               1               1               1
              2              10               2               2
            ...             ...             ...             ...
             15            1111              17               f
             16           10000              20              10
```

Lägg till kod som kontrollerar att det inmatade värdet är ett positivt heltal. I annat fall ska ett felmeddelande ges. I denna uppgift kan du ha nytta av `Integer.toBinaryString()`, `Integer.toHexString()` samt `Integer.toOctalString()`. Utskriftsformateringen ska göras med `System.out.format()` eller `java.util.Formatter`.

Metoden `format` har argument och resultat enligt

```
public PrintStream format(String format,
                          Object... args)
```

där `format` är en s.k. formatsträng. Exempel på användning är

```
System.out.format("%1$16s","text");
```

vilket innebär att `text` skrivs ut högerjusterat i ett fält med bredden 16 tecken. (`%`=Här kommer en formatsträng, `1$`=Använd det första av dom följande argumenten, `16`=Skriv i ett fält 16 tecken brett, `s`=Skriv en sträng. För en beskrivning av hur formatsträngen är uppbyggd se `http://java.sun.com/javase/6/docs/api/java/util/Formatter.html#syntax` .)

FibonacciCalculator

Fibonaccis talföljd är en serie av heltal, vars användande kan spåras tillbaka till 1202, då den användes av Leonardo av Pisa för att beskriva förökningstakten hos kaniner (under perfekta förhållanden). De två första talen är ettor och varje efterföljande tal är lika med summan av de två föregående talen i serien. Början på serien är alltså: 1 1 2 3 5 8 13 21 34. Nedan ser vi den formella definitionen.

$$fib(n) = \begin{cases} 1 & \text{om } n = 0 \\ 1 & \text{om } n = 1 \\ f(n-1) + f(n-2) & \text{annars.} \end{cases}$$

Iteration

Implementera Fibonacci-följden med en iterativ funktion `long fibIterative(int n)`.

Fibonacciföljden är växande. Skriv ut vart tionde tal och observera. Vad händer? Varför? Skriv programmet så att en lista på 20 tal skrivs ut och så att användaren kan mata in startargument och steg (t.ex. Jag vill se var 13:e värde med start från n=17.)

Rekursion

Implementera en rekursiv version av Fibonacci-följden `long fibRecursive(int n)`.

Vad händer om ni matar in lite större värden på n, t.ex. 100? Var beredd att stoppa körningen genom att klicka på den röda fyrkanten till höger om *Console*-fliken. Varför skiljer det sig från den iterativa versionen?

Ändra de båda Fibonacciversionerna så att en räknare från början 0-ställs och därefter stegas upp varje gång ett Fibonaccivärde beräknas. Till sist ska räknarens värde skrivas ut. Förklara skillnaden hos det värdet mellan Fibonacciversionerna.

Extrauppgift: Rekursion + array

Du märkte säkert att den rekursiva versionen av Fibonacci inte var speciellt snabb. För att beräkna $fib(80)$ måste ju både $fib(78)$ och $fib(79)$ beräknas. Vi ser att många saker som vi måste räkna ut under vägen redan har räknats ut men inte tagits till vara.

I den här uppgiften ska du skiva en ny rekursiv Fibonacci-funktion, men se till att funktionen tar till vara redan uträknade delresultat. Detta kan du göra genom att skapa en array `long[] fibNum`. Tanken är att för varje Fibonacci-tal som räknas ut så ska det sparas i listan. Den rekursiva Fibonacci-funktionen kommer då att kontrollera om det redan ligger i `fibNum` innan den räknar ut ett delresultat. För alla $n > 0$ gäller ju att $fib(n) > 0$. Så om `fibNum[i] > 0` innebär att `fib(i)` inte behöver beräknas på nytt.

T.ex. antag att `fib(i)` söks. Innan vi gör anropet `fib(i-1)` kollar vi först om det ligger ett tal större än 0 i `fibNum[i-1]`. Om det inte gör det så sätts `fibNum[n-1] = fib(n-1)`.

På detta sätt kollar alltså den nya rekursiva Fibonacci-funktionen hela tiden om ett visst delresultat redan är beräknat, vilket i sin tur gör att många rekursiva anrop inte behöver utföras.

Notera

Den första Fibonacci-implementationen har linjär komplexitet. Detta innebär, förenklat, att det finns ett linjärt förhållande mellan värdet på n och den tid funktionen tar att exekvera. För

den andra funktionen är förhållandet exponentiellt (2^n). Den tredje funktionen är i värsta fallet linjär, men i bästa fallet så finns ju redan resultatet uträknat i `fibNum` och då är den konstant. Mer om detta kommer i kommande kurs (TDDC91).

Inlämning

Lämna in din källkod samt svaren på frågorna uppgiften.

2 Laboration 5 – Arv vs. inkapsling

Denhär laborationen består av flera deluppgifter. Flera av deluppgifterna ska redovisas så spara koden från en del innan ni börjar ändra/skriva över den i nästa del.

MultiplicationTable

I den här uppgiften ska ni skriva ett en klass `MultiplicationTable`. Tabellen har en konstruktor som tar antalet rader och antalet kolumner som argument. Det ska finnas en metod `print()` som skriver ut följande tabell:

```
% java MultiplicationTable 9 8

  * |  0  1  2  3  4  5  6  7  8
----+---------------------------
  0 |  0  0  0  0  0  0  0  0  0
  1 |  0  1  2  3  4  5  6  7  8
  2 |  0  2  4  6  8 10 12 14 16
  3 |  0  3  6  9 12 15 18 21 24
  4 |  0  4  8 12 16 20 24 28 32
  5 |  0  5 10 15 20 25 30 35 40
  6 |  0  6 12 18 24 30 36 42 48
  7 |  0  7 14 21 28 35 42 49 56
  8 |  0  8 16 24 32 40 48 56 64
  9 |  0  9 18 27 36 45 54 63 72
```

Alla kolumner i tabellen ska vara av samma bredd. Bredden ska vara ett högre än bredden på det element i tabellen som tar störst plats, oavsett räknesätt (nästa delmoment).

Arv

Antag att vi vill lägga till tabeller för fler räknesätt t.ex. addition. Man skulle då kunna tänka sig att vi har en superklass `ArithmeticTable` som definierar en abstrakt metod `int evaluate(int, int)` som utför en aritmetisk heltalsoperation på sina två argument och returerar resultatet.

Inför den nya superklassen och visa hur den ska användas genom att implementera subtyperna `MultiplicationTable`, `AdditionTable`, `SubtractionTable`, och `DivisionTable`.

Hur kan vi lösa problemet utskrift av rätt operatortecken och rätt utskriftsbredd på ett smidigt sätt?

Inkapsling

I stället för arv kan vi lösa föregående uppgift med inkapsling. Istället för att `ArithmeticTable` har en abstrakt metod så skickar vi med ett objekt som är ansvarigt för att beräkna cellvärdena. Inför först följande interface:

```
public interface Operation {

    public char symbol();

    public int width(intg rows, int cols);
```

```
    public int evaluate(int a, int b);
}
```

Nu kan vi ändra konstruktorn till `ArithmeticTable` till:

```
public ArithmeticTable(Operation op, int r, int c);
```

Istället för att arithmetic table anropar den abstrakta metoden `evaluate` så anropas evaluate på den inkapslade operationen. Instansiera ett antal objekt av typen `Operation`: `addition`, `substraction`, `multiplication`, `division`.

```
ArithmeticTable tab = new ArithmeticTable(addition,
9, 8);
```

Extrauppgift: Uppräkningar

Skapa en uppräkning `EnumOperation` som implementerar `Operation` och definierar värdena: `ADDITION`, `SUBTRACTION`, `MULTIPLICATION`, `DIVISION`. Utgå från följande mall.

```
public enum EnumOperation implements Operation {

    // Er kod ska in här

    private final char symbol;

    private EnumOperation(char symbol) {
        this.symbol = symbol;
    }
    public char symbol() {
        return symbol;
    }
}
```

Inlämning

Lämna in er källkod till assistenten . Om ni har gjort extrauppgiften så lämna in den med.

3 Laboration 6 – Ordlista

I denna uppgift ska ni skriva ett glosprogram, dvs ett program som förhör användaren på glosor. Er uppgift är att skriva en klass `WordQuiz` som förhör användaren på ett antal termer i en given ordlista. Användaren måste fortsätta förhöret ända tills alla ord har klarats. När alla termer översatts korrekt skriver programmet ut antalet gjorda försök.

```
import java.util.Scanner;

public class WordQuiz {
    private Scanner input = new Scanner(System.in);
    public WordQuiz(Dictionary dictionary) {
        // Er kod ska in här...
    }
    public void runQuiz() {
        // ...och här
    }
    public static void main(String[] args) {
        // Skapa en tom ordlista på ngt sätt och fyll den med ord.
        Dictionary sweng = ...;
        sweng.add("hej", "hello");
        sweng.add("hej", "hi");
        sweng.add("hej", "hey");
        sweng.add("godnatt", "good night");
        sweng.add("nattinatti", "good night");
        sweng.add("fågel", "bird");
        sweng.add("hund", "dog");
        sweng.add("katt", "cat");
        WordQuiz wq = new WordQuiz(sweng);
        wq.runQuiz();
    }
}
```

```
% java WordQuiz
fågel = bird
katt = katz
Fel!
hund = dogg
Fel!
hej = hi
godnatt = good nite
Fel!
nattinatti = good night
Du missade 3 ord.Jag förhör dig på dessa igen.
katt = catz
Fel!
hund = dog
godnatt = good night
Du missade 1 ord.Jag förhör dig på dessa igen.
katt = cat
Grattis, du klarade alla glosorna på 3 försök.
```

Ni kommer att behöva skriva ett antal hjälpklasser: `Word` och `Dictionary`. Dessa bör innehålla följande metoder:

```
/**
 * Denna klass representerar ett ord, vilket består av en
 * teckensekvens kallad text.
 */
public class Word {
    /**
     * Skapar ett nytt ord med den givna texten.
     */
    public Word(String text);

    /**
     * Jämför detta ord med det specificerade objektet. Resultatet är
     * true om och endast om obj också är ett ord (Word) och har
     * samma text.
     */
    public boolean equals(Word obj);

    /**
     * Returnerar hashkoden för detta ord beräknat på ordets text.
     */
    public int hashCode();

    /**
     * Returnerar texten för detta ord.
     */
    public String toString();
}
```

Vi ser att `Word` inte definerar några nya metoder utan endast åsidosätter tre metoder definierade i `Object`. För att kunna göra det definierar vi en egen klass `Word` istf för att använda `String`.

```
import java.io.InputStream;
import java.io.OutputStream;
import java.util.Set;

/**
 * Denna klass modellerar en ordlista (dictionary). En ordlista
 * associerar termer med betydelser. En term kan mappas till flera betydelser.
 * Både term och betydelse representeras med klassen Word.
 */
public class Dictionary {
    /**
     * Lägger till termen t till ordlistan med betydelsen m. Om termen
     * redan är tillagd med angiven betydelse  händer ingenting.
     */
    public void add(Word t, Word m);

    /**
     * Bekvämare sätt att anropa add för 2 strängar än
     * add(new Word(w1), new Word(w2)).
     */
    public void add(String t, String m);

    /**
```

```
     * Returnerar en icke-null mängd med ordlistans alla
     * termer.
     */
    public Set<Word> terms();

    /**
     * Slår upp och returnerar en mängd av betydelser till t, eller
     * null om t inte finns i ordlistan.
     */
    public Set<Word> lookup(Word t);

    /**
     * Skapar och returnerar en ny ordlista på det motsatta språket, dvs, alla
     * betydelser blir termer och alla termer blir betydelser. T.ex. en
     * svensk-engelsk ordlista blir efter invertering engelsk-svensk.
     */
    public Dictionary inverse();

    /**
     * Läser in orden från den givna strömmen och lägger dessa i
     * ordlistan.
     */
    public void load(InputStream is);

    /**
     * Spara ordlistan på den givna strömmen.
     */
    public void save(OutputStream os);
}
```

Tänk på att om oordlistan ännu inte har några ord inlagda så ska ändå inte `terms` returnera `null` utan en tom mängd. Skillnaden är bl.a. att en tom mängd ändå är en mängd som man kan använda t.ex. motoden `isEmpty` på. Ett `null`-värde kan man inte göra någonting speciellt med.

Internt bör ordlistan använda sig av en `Map`-struktur (se `java.util`). Lämpligtvis mappar vi ord mot mängder av betydelser. Alltså `Map<Word, Set<Word>>`. De konkreta typer som kommer på fråga är alltså: `HashMap` och `HashSet`. Se vidare föreläsningarna. Ni ska på ett översiktligt sätt förstå hur dessa fungerar och vilka krav som ställs på nycklarna. (Detaljer kommer i TDDC91.)

Ni ska också skriva metoder för att spara och ladda ord till och från en ström. Här får ni *inte* använda er av `Serialization`, vilket är Javas egna API för att skriva och läsa datastrukturer. Tanken är att ni ska hitta på ett eget sätt för detta.

När programmet startas ska det fråga efter namnet på filen där ordlistan är lagrad.

Inlämning

Lämna in er källkod till assistenten.

4 Laboration 7 – Menysystem

I den här uppgiften ska ni skriva ett ramverk för ett konsolbaserat menysystem.

```
HUVUDMENY
=========
0. Avsluta
1. Varulista
2. Lägg till ny vara

Ange 0-2: 2

LÄGG TILL NY VARA
=================
0. Tillbaka
1. Bok
2. Film
3. Kläder
4. Mat

Ange 0-4: 1

LÄGG TILL BOK
=============
Varunummer: 6
Titel: Emil i Lönneberga
Författare: Astrid Lindgren
Pris: 45
Miljömärkt: n

LÄGG TILL NY VARA
=================
0. Tillbaka
1. Bok
2. Film
3. Kläder
4. Mat

Ange 0-4:
```

Menysystemet är uppbyggt kring två typer av menyer (`Menu`) och menyval (`MenuItem`). En meny är en så kallad behållare (eng. container) för menyval. Genom att låta meny också vara ett menyval kan en meny innehålla undermenyer.

```
public interface MenuItem {
    /**
     * Returnerar menyvalets rubrik.
     */
    public String getTitle();

    /**
     * Exekverar/väljer menyvalet.
     */
    public void execute();
}
```

```
public class Menu implements MenuItem {
    /**
     * Skapar en tom meny med den givna rubriken.
     */
    public Menu(String title);

    /**
     * Lägger till ett menyval till menyn.
     */
    public void add(MenuItem item);

    public String getTitle();

    /**
     * Exekverar menyn enligt loopen definierad i punkterna (1) till (4).
     * (1) Skriver ut menyns rubrik med stora bokstäverunderstruket med '='.
     * Därefter följer en numrerad lista över alla menyelement i denna
     * meny, numrerade från 0.
     *
     * (2) Användaren får sedan välja ett av alterntiven genom att ange
     * talet framför menyvalet. Vad händer om man inte anger ett
     * giltigt tal? Användarens menyval exekveras.
     * (3) Om menyval 0 valts så returnerar metoden. 0 motsvarar
     * alltså alltid av avsluta/tillbaka/återgå.
     *
     * (4) gå till (1)
     */
    public void execute();
}

public abstract class AbstractMenuItem implements MenuItem {
    private String title;
    public AbstractMenuItem(String title) {
        this.title = title;
    }
    public abstract void execute();

    public String getTitle() {
        return title;
    }
}
```

Givet klassen `class AbstractMenuItem` ovan ser vi ett exempel på hur ett mycket enkelt menysystem kan skapas.

```
Menu testMenu = new Menu("En liten testmeny");
testMenu.add(new AbstractMenuItem("Tillbaka") {      <---- Här skapas en så kallad
    public void execute() {                                anonym klass
            // Gör ingenting.
    }
});
testMenu.add(new AbstractMenuItem("Skriv ut hej") {
    public void execute() {
        System.out.println("Hej!!!");
    }
});
testMenu.add(testMenu); // Wow, en cirkulär meny!
```

Om menyvalen sväller upp och blir väldigt stora kan man för läsbarhetens skull behöva definiera dessa som fullvärdiga klasser istället för anonyma klasser.

Inlämning

Lämna in er källkod till assistenten.

5 Laboration 8 – Varudatabas

I den här uppgiften ska ni skriva en varudatabas enligt utskriften i menysystemslabben. Ert system ska hantera minst tre kategorier av varor (`Item`) – böcker, film och mat. Förutom att lägga till nya varor ska man även kunna söka och lista redan inlagda varor. Listorna ska antingen vara sorterade på varunummer eller alfabetiskt på namn. Som topptyp i vårt system har vi interfacet `Item`.

```
public interface Item {
   /**
    * Returnerar varans namn.
    */
   public String getName();

   /**
    * Returnerar varans typ.
    */
   public String getType();  // Att returnera en enum ItemType med någon
                             // av de varutyper man har är snyggare

   /**
    * Returnerar varans pris.
    */
   public Price getPrice();

   /**
    * Returerar varunumret.
    */
   public int getNumber();

   /**
    * Returerar sant om varan är miljömärkt.
    */
   public boolean isGreen();

   /**
    * Den här metoden ansvarar för att spara objektets tillstånd i den givna
    * källan. Implentatören kan använda valfritt format.
    */
   public void writeItem(PrintWriter out);

   /**
    * Den här metoden ansvarar för att läsa in ett sparat tillstånd från den
    * givna källan.
    */
   public void readItem(Scanner in);
}
```

Varje vara anses alltså ha ett namn, en typ, ett pris, ett varunummer och egenskapen att vara miljömärkt eller inte. Vidare kan man spara respektive läsa varor till och från olika källor.

Förutom de attribut som är förknippade med `Item` har dess subtyper även följande attribut: böcker (`Book`) har titel och författare, filmer (`Movie`) har titel och år, medan mat (`Grocery`) bara ett namn på matvaran.

Vidare ska ni implementera en persistent (sparas mellan körningar) databas för inlagda varor. Utgå från följande klasskelett:

```
import java.util.Collection;

public class Database {
    /**
     * Skapar en ny databas av innehållet i den givna filen.
     */
    public Database(String filename);

    /**
     * Lägger till varan till databasen.
     */
    public void add(Item item);

    /**
     * Slår upp givet varunummer i databasen.
     *
     * @return varan eller <code>null</code> om varunumret ej finns i
     *         databasen.
     */
    public Item get(int i);

    /**
     * Returnerar nästa lediga varunummer.
     */
    public int nextItemNumber();

    /**
     * Returnerar alla varor i databasen i form av en samling.
     */
    public Collection<Item> items();
}
```

En av svårigheterna i den här labben är att lösa problemet med lagringen av databasen på fil. En lösning är att när databasen ska spara sina varor på fil så itererar den över dessa och för varje vara skriver den ner en identifierare för vilken typ av vara det är frågan om och sedan låter varan själv skriva ner sitt tillstånd (m.h.a. writeItem(PrintWriter)).

När databasen senare ska läsa in varorna så läser den först in vilken typ av vara det är, t.ex. `Book`, och skapar en instans av den typen, för att sedan anropa `instance.readItem(...)`.

Filformatet skulle då kunna se ut så här:

```
Class: ttit71.shop.Grocery
Number: 2
Price: 10.0 LITER
Green: true
Name: Mjölk

Class: ttit71.shop.Grocery
Number: 4
Price: 12.0 ITEM
Green: false
Name: Minipizza
```

```
Class: ttit71.shop.Book
Number: 1
Price: 58.0 ITEM
Green: true
Title: Pippi Långstrump
Author: Astrid Lindgren

Class: ttit71.shop.Movie
Number: 3
Price: 45.0 ITEM
Green: false
Title: Hotshots
Director: skdksk

Class: ttit71.shop.Grocery
Number: 5
Price: 6.0 KILO
Green: true
Name: Havregryn

Class: ttit71.shop.Book
Number: 0
Price: 105.0 ITEM
Green: false
Title: Moby Dick
Author: sdf sdf sdk
```

Notera att det inte är nödvändigt att det interna formatet är så här läsbart. Så länge programmet kan avgöra vad varje rad eller ord står för kan vilket format som helst väljas. Inte heller i denna labb får ni använda er av `Serializable`.

Klasskelettet till `Price` och definitionen av uppräkningen `Unit` ser ut så här:

```
/**
 * Denna typ modellerar ett pris per Unit.
 */
public class Price {
    public Price(int amount, Unit unit);

    /**
     * Returnerar prisbeloppet.
     */
    public double getAmount();

    /**
     * Returnerar den enhet beloppet är baserat på. Dvs kr/ENHET.
     */
    public Unit getUnit();
}
```

```
public enum Unit {
    /** Indikerar att varan betalas per styck. */
    ITEM("st"),

    /** Indikerar att varan betalas per kilo. */
    KILO("kg"),

    /** Indikerar att varan betals per liter. */
    LITER("l");

    private String symbol;

    private Unit(String symbol) {
        this.symbol = symbol;
    }

    public String getSymbol() {
        return symbol;
    }
}
```

Inlämning

Lämna in er källkod till assistenten. Bifoga de hjälpklasser ni skrivit i föregående labb så att assistenten förstår hur ni använder era klasser.