Gränssnitt, Samlingar, Instans kontra Klass

Bildserie 4

Sammanslagning av Strängar

Strängar kan inte ändras, det skapas nya hela tiden

- Använder man + (sammanslagning, konkatenering) och strängar kan det bli ineffektivt (speciellt i loopar)
- Ett bättre sätt är att använda en StringBuilder och append()-metoden. När man är klar omvandlar man till String m.h.a. toString() metoden

Uppdelning av Strängar

Finns flera sätt att dela upp en sträng

- split(…), delar upp en sträng i många strängar. Delningen görs utifrån ett matchande "teckenmönster" (kallas **reguljärt uttryck**, regular expression) se http://docs.oracle.com/javase/7/docs/api/java/util/regex/Pattern.html#sum *)

*) Används mycket vid t.ex. Web-programmering då man söker mönster (fraser, ord) i texter (på Web sidor)

Gränssnitt*

Ett **gränssnitt** (**interface**) är en specifikation för ett objekts beteende (ett bindande kontrakt)

- Gränssnittet listar de metoder objektet måste ha
- Hur metoderna fungerar anges inte (metodkroppar saknas)

I Java skapas gränssnitt på liknande sätt som en klass förutom att man använder order interface istf class

- Kan inte instansiera gränssnitt, ingen körbar kod

*) Senare tittar vi på grafiska användargränssnitt, vilket är något helt annat

Gränssnitt och Typ

Ett gränssnitt introducerar en typ (precis som en klass)

Givet: Ett gränssnitt introducerar en typ

Givet: Alla variabler måste ha en typ

Slutsats: Vi kan deklarera variabler av gränssnittstypen

Som ett idiom inleder jag (men inte Java) gränssnittstyper med “I” (stort i)

Implementera Gränssnitt

Om vi deklarerar en variabel av gränssnittstyp ...

- Typsystemet vet att objektet som refereras har vissa metoder (metodanrop på "gränssnitts"-variabeln är ok för type checker:n) ...
- ... men vi kan inte skapa objekt av typen ... ??? Vilken kod körs???

Problemet löser man genom att låta någon klass implementera gränssnittet, (**implements**) d.v.s. alla metoder i gränssnittet finns garanterat i klassen (med körbar kod i metoderna)

Implementera Gränssnitt, forts

Vi säger att klassen som implementerar gränssnittet **överskuggar** (**override**) metoderna i gränssnittet (ersätter dessa)

- För att det skall fungera måste returtyp, namn och parameterlista vara exakt så som i gränssnittet
- Ovanför metoderna i klassen anger man **@Override**, en kontroll att det verkligen är överskuggning (kompilator kontrollerar)

Gränssnitt och Typsystem

Om vi har detta (gränssnitt, ny typ)...

```
public interface IMyInterface {
     // No code in metod!!!
     public void doIt();
}
```

```
public class MyClass  implements IMyInterface {
     // Code here. Must have else compile error
     public void doIt(){ System.out.println
("...");}
}
```

.. och detta (klass) ...

```
// Ok for type system, object referenced by i
guaranteed (by implements) to have method
IMyInterface i = new MyClass(); // Ok
i.doIt();    // We have code to run!
```

… så är tilldelning OK!

Varför Gränssnitt?

Genom att använda gränssnitt skiljer vi på specifikation och implementation

- Om vi i programmet använder variabler av gränssnittstypen (specifikationstypen, d.v.s. abstraherar bort implementationen) skyddar vi koden mot förändring
- Om vi byter implementation behöver vi inte ändra något i resten av programmet (vi kanske kommer på ett bättre sätt att implementera, eller vill kunna byta implementation (under körning))
- Programmet blir modifierbart … (om vi tar fram en bra specifikation, en dålig leder till mycket problem …)

Samlingar

Mycket vanligt att man i ett program behöver samlingar av objekt (många av samma typ)

Samlingarna kategoriseras utifrån hur de lagrar objekt och vilka regler som gäller för åtkomst av objekten (alltså inte utifrån vilka objekt de innehåller)

- Man säger att samlingarna är generiska (**generic**), de kan innehålla "vilka objekt som helst" (av samma typ)

Typiska Samlingar

Mängd, samma idé som en matematisk mängd, inga dubbletter, ingen ordning

Lista, … en ordnad sekvens, första, n:te, sista, före, efter, ...

Lexikon, givet en nyckel kan man slå upp ett värde … (namn/telefonnummer, svenskt ord/engelsk översättning)

Java Collection Framework

Java har färdiga klasser för samlingar

- The Java Collection Framework (JCF), är samlade :-) i paketet java.util.*
- JCF skiljer konsekvent på specifikation och implementation
- JCF:s gränssnitt och klasser är generiska, man anger vid deklarationen vilken typ av objekt samlingen skall innehålla (t.ex. List<String>)
- I dokumentationen visas List<T>, där T står för någon typ

JCF Exempel

Samling	Specifikation (gränssnitt)	Implementationer (klasser som implements)
Mängd	Set	HashSet, LinkedHashSet, TreeSet, ...
Lista	List	ArrayList, LinkedList, RoleList, ...
Lexikon	Map	HashMap, ConcurrentSkipListMap, ...

```
// Declaring and instantiating collections

Set<Integer> is = new HashSet<>();

List<String> sl = new ArrayList<>();

Map<String, Integer> m = new HashMap<>();
```

List och ArrayList

Vi kommer bara att använda gränssnittet List och implementationen ArrayList

- Alla parametrar och returtyper anges som List<...> inte ArrayList<...>

List påminner om vanliga Array:er men...

- Vi föredrar List framför Array (Array:er för primitiva, på för låg abstraktionsnivå)
- Listor växer och krymper dynamiskt efter behov
- Listor har metoder för att lägga till/ta bort/uppdatera/hitta index för objekt, tömma listan, dellistor, m. m.
- Om vi behöver fixa (fasta) positioner är dock Array att föredra (försök att hålla Array:er "inom klassen")

Exakt hur fungerar List?

Vad händer om vi ...

- lägger till ett element, var hamnar det? Först, sist,...?
- tar bort ett element i mitten (blir det ett hål)?
- lägger till en dubblett? Skrivs gamla värdet över?
- har dubbletter, vilken returneras?
- lägger till null?
- anropar add(), vi får en boolean i retur?
- anropar contains(...)

Equals Metoden

Ofta behöver två objekt jämföras t.ex. om man anropar list.contains(someObject) för någon lista

- Returnerar sant om someObject finns i listan (om det är "lika" något annat objekt i listan)

Som standar har alla objekt i Java en equals()-metod

- Den fördefinierade equals-metoden jämför referenser (**samma**). Vi vill vanligen jämföra tillstånd (**lika**)
- Om en klass skall hanteras av en samling skall klassen normalt ha en egen version av equals-metoden (dessutom skall det finnas en egen version av metoden hashCode, de hör ihop)
- Även i detta fall säger vi att vi överskuggar equals-metoden (equals-metoden finns egentligen i klassen Object mer senare…)

TileBag

TileBag hanterar många Tiles alltså en samling av Tiles

- List eller Array?
- Tile equals() metod ...!?

Holder

En holder hanterar också många Tiles

- List eller Array?

Lab 1: Klassen DiceCup

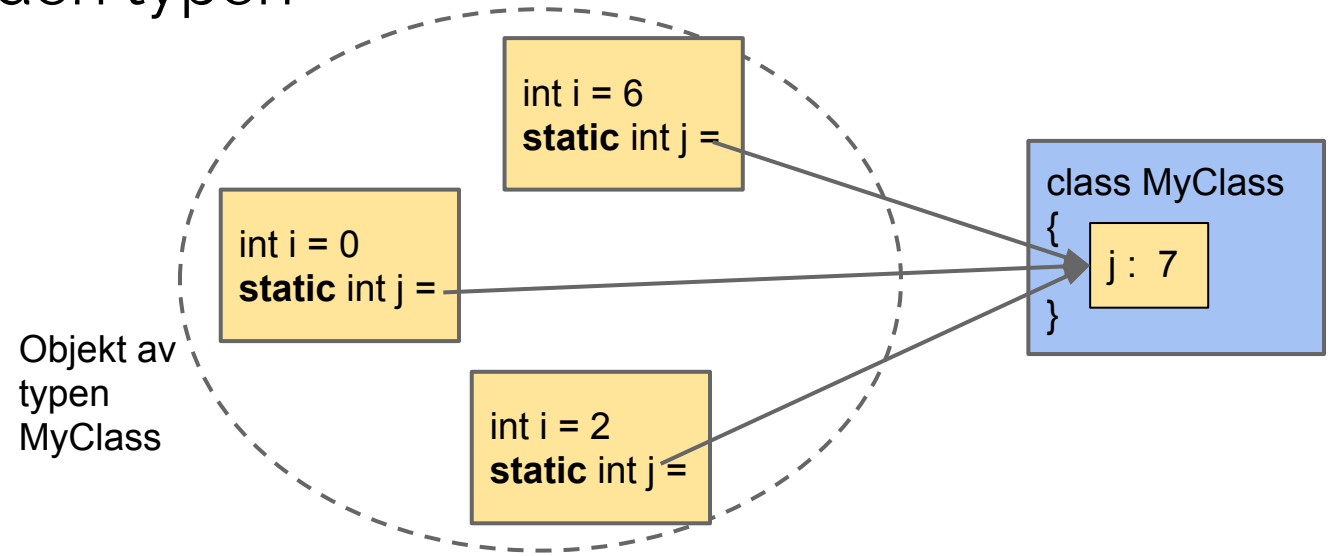
Klassen hanterar ett antal tärningar, hur spara dessa?

- Array eller List?
- "Utåt" hanterar klassen bara List (tar emot List som argument, resulterar List som resultat)
- Ev. byte mellan Array och List format (mer senare)
- Finns färdig test för klassen

Instans och Klassvariabler

En instansvariabel, tillhör ett objekt (del av objektets tillstånd)

En **klassvariabel** (**static**) är gemensam för alla objekt av den typen



Klassvariabler

Användning av klassvariabler

- Eftersom variabeln delas av alla kan vilket objekt som helst ändra värdet för alla andra (gör ett objekt fel drabbas alla andra)), måste vara final
- Använd enbart (i denna kurs) som **konstant** (= static final), se nedan

```
// The only way in course to use a static variable
// (as a constant). Note Naming convention
public static final String MY_CONSTANT;
```

Instans och Klassmetoder

En instansmetod, kan använda instansvariabler och klassvariabler

- Måste anropas på ett objekt (det objekt vars värden för instansvariablerna metoden skall använda)

En klassmetod (**static**) kan bara använda klassvariabler

- Anropas direkt "på klassen" (klassnamnet)
- Behöver inte skapa något objekt
- Kan inte använda this-referensen (this är ju det aktuella objektet, orimligt...)

Klassmetoder

Används då man inte behöver några instansvariabler för att utföra uppgiften

- Motsvarar rena funktioner (som i matematik), indata -> utdata
- T.ex. Math.sin(...)
- Metoden main (den som anropas först av allt i alla Java program) måste vara static

Collections och Arrays

Två hjälpklasser då man arbetar med samlingar.

- Klasserna är rena funktionssamlingar, allt är static

```
// Some useful methods
Collections.sort( Collection c );
Collections.min( Collection c );
Collections.max( Collection c );

// Transform array to (unmodifiable) List
List< … > aList = Arrays.asList( anArray );
```

Lab 1: Klassen Score

Representerar en (spelarens) kolumn i resultattabellen

- Använder hjälpklass RowValue (en samling av RowValues)
- Instansmetoder, för spelarens värden
- Klassmetod för att beräkna möjliga utfall (givet ett resultat). Komplicerad måste brytas ned i många hjälpmetoder
- Sortering av inkommande resultat underlättar

Lab 1: Testa Score

Finns påbörjad testklass för Score

Ni skall själva komplettera denna

- Sätt hjälpmetoder till private när testningen är klar

Lab 1: Klassen Player

Enkel klass som representerar en spelare

Använder Score för att bokföra spelarens resultat

Lab 1: Klassen Yatzy

Klassen Yatzy representerar “hela” spelet

- Klassen administrerar och använder många andra modellklasser för att åstadkomma den logik och datahantering som krävs för spelet (hur många spelare är det? hur många tärningskast har aktuell spelare gjort? Är spelet slut?...)

Kommandoradsmeny

För att testa flera samverkande klasser skapar vi en **kommandoradsmeny**

- Skapas med en switch-sats i en while-loop, en Scanner och System.out

```
// Commandline principle
while( … ){
     action = scanner.read..()         // Ask user what to do
     switch( action ){
     case ...: doSomething(); break;      // Do it, possible display result
     case ..:  doSomethingElse(); break;
     }
}
```

Lab 1: Testa Yatzy

När klassen Yatzy är klar har vi ett logiskt fungerande spel

För att testa klassen skall ni göra klart en påbörjad kommandoradsmeny (i yatzy.test.TestCommandLine)