Algoritmer och datastrukturer I, 5p
2005-11-15, mbm
N-institutionen, avdelningen för datavetenskap

HÖGSKOLAN I GÄVLE

Handledning till kurslitteraturen

Weiss, M.A.: *Data Structures and Problem Solving Using Java*.

Hänvisningar till avsnitt i kurslitteraturen markeras som vanligt med överstrykning. Läsanvisningarna till föreläsning 2 är också relevanta för laboration 1.

Laboration 1: Listor, Stackar och Köer

Arv

Att använda sig av *arv* (inheritance) för att skapa nya klasser som ärver egenskaper från andra klasser är en grundläggande teknik inom objektorienterad programmering. Vi kommer inte att behöva använda oss så mycket av arv i denna kurs, men eftersom Java använder sig väldigt mycket av arv (till exempel i standardbiblioteket) så kan det underlätta förståelsen att ha ett hum om vad arv är och hur det fungerar i Java. Arv introduceras i avsnitt 4.1.

Fortsätt läsa avsnitt 4.1.1, som handlar om hur nya klasser kan skapas med hjälp av arv, och avsnitt 4.1.2 som handlar om typkompatibilitet. Avsnitt 4.1.3 - 4.1.9 kan du hoppa över tills vidare, liksom avsnitt 4.2 - 4.3.

Du bör också känna till att alla klasser i Java ärver vissa datafält och metoder från klassen `Object`. Detta förklaras i avsnitt 4.5 - 4.5.1.

Generiska komponenter

Avsnitt 4.6 handlar om *generiska* metoder, det vill säga metoder som kan operera på vilka klasser som helst. Exempel på metoder som kan göras generiska är *sorteringsmetoder*. Det enda som skiljer sorteringsmetoder åt är hur elementen som ska sorteras jämförs med varandra. Heltal jämförs på ett sätt, tecken på ett annat sätt, och strängar på ett tredje sätt. Själva logiken för sorteringen är däremot oförändrad. Detta att alla klasser ärver från klassen `Object` kan utnyttjas för att skapa generiska metoder. Hur det går till beskrivs i avsnitt 4.6.2. För att kunna betrakta variabler av de primitiva datatyperna (int, double, char o.s.v.) som objekt behöver de först "slås in" i en så kallad wrapper-klass. Detta förklaras i avsnitt 4.6.2. Studera exemplet i figur 4.24 noga.

Ett annat sätt att skapa generiska abstrakta datatyper är att använda sig av språkkonstruktionen *generics*, som introducerades i Java 2 version 1.5. Generics beskrivs närmare i kursbokens tredje upplaga. I C++ finns en motsvarighet till generics som kallas för *templates*.

Interface

I föreläsning 2 nämndes att varje abstrakt datatyp har ett gränssnitt som talar om vilka operationer som hör ihop med denna datatyp. För att specificera gränssnitt så finns i Java en särskild språkkonstruktion som kallas `interface` (vilket helt enkelt är det engelska ordet för gränssnitt). För att undvika sammanblandning mellan det generella begreppet 'gränssnitt' och den specifika javakonstruktionen 'interface' kommer vi att försöka använda det svenska ordet när vi menar gränssnitt i allmänhet, och det engelska ordet när vi syftar på Java-interface.

Interface introduceras i avsnitt 4.4. Läs också följande avsnitt:

- 4.4.1 **Specificera ett interface**
- 4.4.2 **Hur en klass implementerar ett interface**
- 4.4.3 **Multipla interface**

Avsnitt 4.4.4 kan hoppas över, liksom avsnitt 4.5.2 - 4.5.3, 4.6.3 - 4.6.4, 4.7 och 4.8.

HÖGSKOLAN I GÄVLE

Exceptions (Undantag)

Avsnitt 2.5 introducerar språkkonstruktionen *exceptions* (undantag) som kan användas i många språk, däribland Java, för att rapportera "fel" (eller snarare *undantagsfall*) som uppstår då en metod exekveras. Exempel på sådana undantagsfall kan vara felindexering i en array (ett försök att komma åt en position som inte finns, till exempel position 0), ett försök att anropa en metod för en null-referens, eller ett försök att poppa från en tom stack.

Att använda exceptions är bättre än att bara skriva ut ett felmeddelande inuti den metod där felet uppstod. Anledningen till det är att felmeddelandet visserligen ger information till en mänsklig användare, men inte kan förstås av programmet självt. En exception däremot kan skickas som en signal från en metod till en annan metod, till exempel huvudprogrammet, så att programmet kan vidta de åtgärder som eventuellt krävs (till exempel be användaren mata in ett nytt array-index).

I avsnitt 2.5.1 beskrivs hur exceptions används. Grundregeln (som har ett undantag) är att varje *anrop* till en metod som kan orsaka[1] en exception ska omges av ett *try-catch-block*, enligt följande mönster:

```
try {
      …
      ettObjekt.enMetodSomKanKastaEnException();
      … } catch(här anges vilken sorts exception som kan hanteras) {
      …
      (här anges vad som ska göras om undantaget har inträffat)
      …
      }
```

Både `try` och `catch` är reserverade ord i Java.

I avsnitt 2.5.2 förklaras hur det reserverade ordet `finally` kan användas för att utöka try-catch-blocket till ett *try-catch-finally-block*. I finally-delen läggs kod som måste exekveras ovillkorligen, vare sig undantaget har inträffat eller ej. Detta kan vara viktigt i olika sammanhang, till exempel vid filhantering där det är viktigt att stänga filer som har öppnats.

Avsnitt 2.5.3 introducerar de tre huvudgrupperna av exceptions: *runtime exceptions*, *checked exceptions* och *errors*. Vi fokuserar bara på den första kategorin här, och återkommer till de andra två kategorierna senare. Det är viktigt att komma ihåg att alla exceptions är *objekt*, precis som i princip allt annat som hanteras i Java. Med andra ord så är en exception en instans av en exception-klass.

En *runtime exception* är den sortens undantag som på sätt och vis är lättast att använda. För att kunna fånga upp ett sådant undantag så måste metodanropet omges av try-catch som vanligt, men kompilatorn kommer inte att protestera om try-catch utelämnas. Därför är det lätt om man att snabbt skriva program som anropar sådana metoder, även om man är ovan att använda exceptions. När man har fått programmet att fungera för de fall som inte leder till att ett undantagsobjekt kastas, så kan koden för att fånga upp undantaget läggas till.

Runtime exceptions är praktiska att använda för den typen av undantagsfall som kan förekomma på väldigt många ställen i ett program. I sådana lägen vore det väldigt bökigt att hela tiden använda try-catch. Till exempel vet vi att varje gång vi ska använda punktoperatorn för att anropa en metod för ett visst ovjekt så finns risken att objektreferensen är *null*, vilket leder till att en *NullPointerException* kastas. Föreställ dig hur koden skulle se ut om varje metodanrop skulle omges av try-catch för att kunna fånga upp en *NullPointerException*!

Förutom den nyss nämnda klassen finns det några fler *standard runtime exceptions* i Java. Några av dem räknas upp i figur 2.12.

[1] Detta kallas också att *kasta* en Exception, på engelska *throw*.

HÖGSKOLAN I GÄVLE

Några frågor återstår att besvara:

1. Om vi vill *anropa* en metod, hur vet vi om den kan orsaka en exception eller inte, och i så fall vilken slags exception?
2. Kan man skapa egna ”skräddarsydda” undantagsklasser, och hur går det i så fall till?
3. Om vi vill *skriva* en egen metod som kan orsaka ett undantag, hur gör man för att kasta en exception inuti metoden? Går det att deklarera på något sätt att metoden kan kasta ett undantag, så att användaren av metoden får reda på det?

Vi återkommer till dessa frågor i nästa laboration.