Algoritmer och datastrukturer
HT 2015

skriven av mbm
reviderad av dhe 2009-03-06
reviderad av jackson 2012-02-27
reviderad av perjee 13 november 2015

Laboration 1: De abstrakta datatyperna Stack och Kö

Syfte

Syftet med laborationen är

- att visa hur abstrakta datatyper kan skapas i Java med hjälp av klasser och interface;
- att ge förståelse för vikten av att separera mellan abstrakta datatypers gränssnitt och deras implementation;
- att ge färdighet i att använda de abstraka datatyperna stack och kö
- att illustrera hur ett gränssnitt för en lista kan skapas i något programmeringsspråk, till exempel Java; samt
- att ge en introduktion till enhetstestning med Junit i Java.

Förberedelse

Ha kurslitteraturen och litteraturhandledningen för denna uppgift nära till hands när du arbetar med denna uppgift. Föreläsning 1 och 2, med tillhörande avsnitt i kurslitteraturen, är också relevanta. Det är starkt rekommenderat att du genomför de icke obligatoriska övningarna och att du har läst igenom hela uppgiften innan du börjar arbeta.

Uppgifter

1. Skriv ett program som med hjälp av en stack testar om ett inmatat uttryck är parentesbalanserat eller inte. Programmet ska be användaren mata in en sträng, och sedan skriva ut om strängen var parentesbalanserad eller inte. Programmet ska avslutas när användaren matar in en tom sträng.
2. Skapa ett antal testfall som sedan används vid testning av parentesbalanseringsprogrammet. Dokumentera testresultaten med skärmdumpar som läggs till som bilagor i laborationsrapporten.
3. Skriv testfall för att med hjälp av Junit eller på ett annat sätt testa förekomsten av fel hos list-implementationen av stacken. Alla metoder skall testas om deras funktionalitet följer dokumentationen.
4. Utveckla ett interface som beskriver ADT:n kö. Skriv testfall för att med hjälp av Junit eller på ett annat sätt testa förekomsten av fel hos en implementation av interfacet. Alla metoder skall testas om deras funktionalitet följer dokumentationen.

Genomförande

Börja med att skapa en katalog för kursens laborationer på ditt konto eller din dator, och spara ner `aod_lab1.zip` till den nyskapade katalogen. (Om filen inte packas upp automatiskt, gör det med hjälp av något packprogram, t.ex. 7zip.) När du packar upp ZIP-filen så kommer bl. a. katalogen `src`, som innehåller några underkataloger och ett antal filer, att skapas i katalogen `aod_lab1`.

Katalogen innehåller även `jar`-filen `aod_lab1.jar`. I den finns färdigkompilerade `class`-filer som du behöver för att kunna kompilera och testköra programmen i laborationen. Filen behöver inte packas upp. Koden i `class`-filerna är kompatibel med JDK 7 och 8.

API-dokumentationen till klasserna i `aod_lab1.jar` hittar du i underkatalogen `doc`!

Uppgift 1. Testprogram för en stack: Balanserade parenteser

Börja med att studera de metoder som specificeras i interfacet `Stack` (filen `Stack.java` i en av under-katalogen i `src`). Interfacet `Stack` implementeras av den färdigkompilerade klassen `ListStack`. Klassen tillhör paketet `se.hig.aod.lab1`.

Implementera sedan testprogrammet, som alltså ska undersöka om stränguttryck som matas in av användaren innehåller balanserade parentesuttryck eller ej.

- Exempel på balanserade parentesuttryck: `()`, `(())`, `()()`, `((())())`
- Exempel på obalanserade parentesuttryck: `)(`, `(()`, `())`, `(()(`

Utgå gärna från skelettkoden som finns i filen `CheckBalance.java` (finns under katalogen `src` i paket `se.hig.aod.lab1`). Skriv den statiska metoden `isBalanced`, som tar ett parentesuttryck (en textsträng) som inparameter, och som returnerar `true` om uttrycket är parentesbalanserat, annars `false`. Anropa metoden från huvudprogrammet där du läser in ett parentesuttryck från tangentbordet.

Testprogrammet får gärna ha ett grafiskt användargränssnitt, men ett enkelt textbaserat gränssnitt räcker. Det går bra att använda `Scanner`-klassen från övning 1.

Algoritm för balanserade parenteser

Algoritmens grundidé beskrivs i avsnitt 6.6.2 i kurslitteraturen. Figur 1 visar samma algoritm i en mer detaljerad pseudokod.

Tips

- För att kunna kompilera programmet behöver du tala om för kompilatorn att de filer som finns i `aod_lab1.jar` ska ingå i projektets `classpath`. Om du arbetar i terminalfönster görs det genom att ange flaggan `-classpath` (följt av filens namn, inklusive sökvägen till filen) till kompilatorn.

  Om du arbetar med någon IDE lägger du till filen i projektets sökvägar. I Eclipse så högerklickar man på projektet och väljer ”Build Path” → ”Add External Archives...” Där navigerar man till katalogen som innehåller `jar`-filen och väljer till slutet `jar`-filen.

```
Skapa en tom teckenstack.

Antag att uttrycket är balanserat.

Så länge uttrycket är balanserat och det fortfarande finns
tecken i strängen:
    Hämta nästa tecken från strängen.

    Om nästa tecken är '(':
      Placera '(' på stacken.
    Annars:
      Om nästa tecken är en ')':
        Om stacken ej är tom:
          Plocka bort toppen av stacken.
        Annars:
          // Vi har hittat en högerparentes som saknar maka.
          Markera att uttrycket ej är balanserat.

Om uttrycket är balanserat och stacken ej tom:
    // Vi har kvar vänsterparenteser som saknar makar.
    Markera att uttrycket ej är balanserat.

Töm stacken.
```

Figur 1: Algoritm som testar om en sträng innehåller ett balanserat parentesuttryck.

- Använd metoden `charAt` för att komma åt ett tecken på en viss position i strängen. Se avsnitt 2.3.4 i kurslitteraturen.
- `Stack` är ett generiskt interface, dvs. användaren måste bestämma för vilken sorts objekt det ska användas. I vårt fall ska det kunna användas för att spara tecken (`char`):

  ```
  Stack <Character> charStack = new ListStack <Character> ();
  ```

  Eftersom det finns en implicit konvertering (se nedan) från `char` (primitiv datatyp) till `Character` (wrapper-klass för `char`-data) i Java kan man använda följande kod för att lägga till ett tecken på stacken:

  ```
  char aChar;
  ...
  // Plocka ut det i:te tecknet:
  aChar = parentesuttrycket.charAt(i);
  // "Slå in" tecknet (implicit) och pusha:
  charStack.push(aChar);
  ```

Men man kan göra en explicit konvertering till wrapper-klassen `Character` också:

```
charStack.push(new Character(aChar)); // "Slå in" tecknet och pusha
```

Varför måste vi göra på detta sätt? Svaret är att en `char` är en av de primitiva datatyperna, och de primitiva datatyperna är inga objekt. Alltså ärver de inte egenskaper från klassen `Object`, till skillnad från vår "wrapper" (`Character`) som gör det. Alltså kan `Character`-objekt pushas på stacken, men däremot inte `char`-värden. Läs mer om wrapper-klasser i kurslitteraturen, avsnitt 4.6.2. Notera att exemplet i figur 4.24 använder ungefär samma mönster som beskrivs ovan.

Uppgift 2. Tester

Skapa ett antal "testfall" för att testa programmet. Dokumentera utfallen av testerna genom att lägga skärmutskrifterna till rapporten. Beskriv testfallen och resultaten i resultatdelen, så att det tydligt framgår att stacken fungerar. Exempel på lämpliga testfall:

1. Kontroll av uttrycket `()`
2. Kontroll av uttrycket `)(`
3. Kontroll av uttrycket `(()`
4. ...

Uppgift 3. Att testa implementationer

Filen `ListStackTest.java` (under katalogen `tests`) är ett exempel hur man kan testa klassen `ListStack` med hjälp av Junit. Testfallen 'övertäcker' bara en del av klassens funktionalitet och är bara delvis implementerade.

Implementera tomma testfallen resp. definiera och implementera de som saknas!

Du kan använda Junit (rekommenderas!) eller ett annat testprogram. Viktigt är att testet är fullständigt och korrekt.

Se dokumenten till kursen OODP på Blackboard under 'Innehåll → Junit' för mer information.

Uppgift 4. Java-interface och testprogram för en kö

Försök att med papper och penna skapa ett Java-interface för en kö (FIFO-kö). Specificera vilka metoder som ingår i interfacet, deras eventuella inparametrar och returtyper. Diskutera gärna med din grupp om du kör fast eller om något är oklart. När du är klar, jämför ditt eget förslag med interfacet `Queue.java` (se katalogen `src`).

Interfacet `Queue` implementeras av den färdigkompilerade klassen `ListQueue` i paketet `se.hig.ad1.lab1`. Dokumentationen av interfacet finns i katalogen `doc`. Observera att vissa metoder kastar en "runtime exception" (klassen `QueueEmptyException`) om användaren försöker att komma åt ett element i en tom lista.

Skapa en Junit-test eller ett liknande program som testar `ListQueue`-klassen på ett liknande sätt som `ListStack`.

Diskussionsfrågor

Att förbereda inför seminariet:

1. Föreläsningsanteckningarna räknar upp några ”standardoperationer” på en stack. Jämför med det givna interfacet `Stack` som beskrivs under rubriken 'Genomförande'. Är det någon stackoperation som inte verkar ha någon motsvarighet i interfacet? Är det någon operation i gränssnittet för en stack som verkar överflödig i javasammanhang?

2. Jämför standardbibliotekets stack-klass (klassen `java.util.Stack`) med stack-interfacet från uppgift 1, och försök hitta likheter och skillnader.

3. När det gäller stack-ADT:n i standardbiblioteket så menar många att konstruktörerna har gjort några designmissar. Kan ni komma på några brister i designen av standardbibliotekets stack?

4. Stacken och kön i uppgift 1 och 3 använder 'runtime exceptions' för att tala om för användaren att en operation har använts på ett felaktigt sätt, till exempel att en borttagningsoperation har anropats för en stack/kö som är tom. Ge exempel på ett par andra sätt för en abstrakt datatyp att signalera felaktig användning, och diskutera för- och nackdelar med de olika metoderna.

5. Hur signaleras felaktig användning av `java.util.Stack`, `java.util.Queue` respektive `java.util.List`?

6. Skriv ett Java-interface `List` för den abstrakta datatypen (osorterad) Lista. Interfacet behöver inte innehålla alla tänkbara operationer på en osorterad lista, men åtminstone de viktigaste standardoperationerna (skapa lista, tömma lista, kontrollera om listan är tom/full, antal element) och några varianter av insättnings- och borttagningsoperationerna.

Tips fråga 2-3

I Javas standardbibliotek finns klassen `java.util.Stack`, som är en implementation av en stack-ADT. Vi kommer inte använda oss av standardbibliotekets stack i programmeringsuppgifterna, men vi bör känna till att den finns.

Tips fråga 6

Varje gruppmedlem skapar ett eget interface i en fil `List.java`. `List`-interfacet ska vara generisk, dvs. användaren ska kunna bestämma klassen av objekten som ska kunna sparas i listan. Följ exemplet i interfacen `Stack.java` och `Queue.java`.

Observera att du inte behöver skapa någon klass som implementerar interfacet. Du kan alltså inte testköra ditt listinterface, men se till att kompilera interfacet och rätta eventuella kompileringsfel. Jämför ditt förslag med dina gruppmedlemmars förslag. Diskutera fram ett ”vinnande” förslag eller jämka ihop era förslag till ett slutligt förslag från gruppen.

Redovisning

Programmeringsuppgifterna och rapportskrivning utförs gruppvis. Följande är en checklista över alla delar som ingår i redovisningen:

- En skriftlig laborationsrapport som innehåller svar på de frågor som ställs i uppgiften, och i övrigt utgår från de typografiska mallar som finns tillgängliga på Blackboard. Glöm inte att ange författarens/författarnas namn, personnummer och e-postadress på rapportens framsida. Lämna inte in Word-filer, på grund av risken för spridning av makrovirus. Konvertera istället till PDF-format. Instruktioner för hur du gör det finns på Blackboard.
- Dokumentation av testresultaten i form av skärmdumpar i labbrapporten.
- Alla källkodsfiler som hör till uppgiften, även de som du inte själv har skrivit resp. modifierat. Dokumentera i kod som du själv skrev att du är författare (’`@author`’). Till kod som du har modifierat ska du lägga till en kommentar att du är upphovsmannen till modifikationen.
- `javadoc`-genererad HTML-dokumentation av de klasser du själv har skrivit resp. modifierat.

Packa ihop laborationsrapporten, skärmdumpar, källkoden och javadoc-dokumentationen i en `jar`-fil eller `zip`-fil, och ladda upp filen via inlämningslänken i Blackboard som tillhör laborationen.

Lycka till! :-)