# *Programmering i Java – fortsättningskurs*

Datorlaborationer

**EDAA30**

`http://cs.lth.se/edaa30`

LUNDS UNIVERSITET
Lunds Tekniska Högskola

HT 2017

# Datorlaborationer – anvisningar

Datorlaborationerna ger exempel på tillämpningar av det material som behandlas under kursen och ger träning på att skriva program. I kursen ingår 6 sådana laborationer.

- Uppgifterna i laborationerna löses normalt i par om två. I samband med anmälan till till laborationerna får du möjlighet att ange vem du vill samarbeta med. Anmälan till laborationsgrupp sker via kursens hemsida (`cs.lth.se/edaa30`).

- Du måste arbeta med varje laboration under ”rätt” vecka (om du inte är sjuk, se nedan). Om du är mycket ambitiös kan du dock arbeta i förväg och få flera uppgifter godkända vid ett redovisningstillfälle, under förutsättning att laborationsledaren har tid med detta. Du kan däremot inte komma i efterhand och kräva att bli godkänd på flera uppgifter (utom om du har varit sjuk).

- *Laborationerna kräver en hel del förberedelser.* I början av varje laboration finns anvisningar om förberedelser. Under rubriken Förberedelser anges vilka av laborationsuppgifterna som *ska* lösas före laborationstillfället. Du ska också ha läst igenom de övriga uppgifterna och gärna försökt lösa dem. Det är inget krav att att du kommer med helt färdiga lösningar. Men det är ditt och din laborationspartners ansvar att ha förberett er så att ni bedömer att ni hinner bli klara under laborationen. Kontakta kursansvarig om det dyker upp några frågor när du förbereder laborationen.

  Om du har förberett dig väl bör du hinna med alla uppgifterna under laborationen. Om du inte gör det så får du göra de resterande uppgifterna på egen hand och redovisa dem vid påföljande laborationstillfälle eller resurstillfälle (och förbereda dig mera till nästa laboration).

- Se till att laborationshandledaren noterar dig som godkänd på listan på nästa sida när du är godkänd på en laboration.

- Om du är sjuk vid något laborationstillfälle så måste du anmäla detta till kursansvarig (niklas.fors@cs.lth.se) före laborationen. Annars riskerar du att bli hänvisad till nästa gång kursen går, och du kan då inte slutföra kursen (och få poäng) förrän då.

  Om du varit sjuk bör du göra uppgiften på egen hand och redovisa den vid ett senare tillfälle. Har du varit sjuk och behöver hjälp för att lösa laborationen kontakta kursansvarig. Det kommer också att anordnas en ”uppsamlingslaboration” i slutet av kursen.

# EDAA30 Programmering i Java – fortsättningskurs, godkända laborationer HT 2017

Skriv ditt namn och din namnteckning nedan:

Namn: .....

Namnteckning: .....

| Godkända uppgifter | Datum | Handledarens namnteckning |
|---|---|---|
| 1 | | |
| 2 | | |
| 3 | | |
| 4 | | |
| 5 | | |
| 6 | | |

För att bli godkänd på en uppgift måste du lösa deluppgifterna och diskutera dina lösningar med en labhandledare. Denna diskussion är din möjlighet att få feedback på ditt program. Ta vara på den!

Se till att handledaren noterar dina uppgifter som godkända på detta godkännandeblad. Dessa underskrifter är ditt kvitto på att du är godkänd på laborationerna. Spara dem tills du fått slutbetyg i kursen.

# Innehåll

# Laboration 1 – använda abstrakta datatyper

*Mål:* Du ska träna på att implementera algoritmer med hjälp av de abstrakta datatyperna `List`, `Set` och `Map`. Du ska också träna på att använda interface och klasser från Java Collection Framework som beskriver respektive implementerar dessa abstrakta datatyper.

## Förberedelser

Besvara frågorna med hjälp av kurslitteraturen, föreläsningsbilderna och/eller dokumentationen för respektive interface.

F1. Betrakta följande kodavsnitt:

```
List<Integer> nbrs = new ArrayList<Integer>();
for (int i = 0; i < 100; i += 10) {
    nbrs.add(i);
    nbrs.add(i);    // notera: talet läggs till två gånger
}

for (int a : nbrs) {
    System.out.println(a);
}
```

a) Hur många rader skrivs ut? **Svar:** ☐

b) Kan vi vara säkra på talens ordning i utskriften? **Ja** ☐ **Nej** ☐

F2. Anta att vi byter ut den första raden mot följande:

```
Set<Integer> nbrs = new HashSet<Integer>();
```

a) Hur många rader skrivs nu ut? **Svar:** ☐

b) Kan vi vara säkra på talens ordning i utskriften? **Ja** ☐ **Nej** ☐

F3. Betrakta följande kodavsnitt:

```
Map<__________, __________> m = new HashMap<__________, __________>();
m.put("albatross", 12);
m.put("pelikan", 27);
m.put("lunnefågel", 19);
m.put("albatross", 7);
System.out.println(m.get("albatross"));
```

Fyll i typargument på de streckade raderna ovan, så att typerna stämmer överens med koden. (Du ska inte använda `Object` här.)

F4. Vad skrivs ut i exemplet i föregående uppgift? **Svar:** ☐

F5. I interfacet `Map` finns en metod för att undersöka om en given nyckel förekommer. Vilken då? **Svar:** ☐

F6. Läs igenom texten under rubriken ”Bakgrund”.

F7. Lös uppgifterna D1, D2, D3 och D4.

F8. Läs igenom övriga uppgifter.

## Bakgrund

Historiker och litteraturvetare använder ibland datorbaserade metoder för att få översiktlig information om stora textmassor. Genom att exempelvis räkna förekomster av ortnamn (och liknande) kan man skapa sig en uppfattning om den geografi som beskrivs i texten.[1]

Med en sådan metod kan vi undersöka Selma Lagerlöfs bok *Nils Holgerssons underbara resa genom Sverige* och räkna förekomster av landskapens namn. På så sätt kan vi förstå något om den bild av Sverige som tecknas i boken. Lagerlöfs bok gavs ut 1906–1907, under den nationalromantiska perioden, då många av 1900-talets föreställningar om den svenska nationen tog form.

När vi undersöker en text, som Lagerlöfs bok, räknar vi alltså förekomster av vissa ord, som exempelvis platser. I den här uppgiften kommer du att konstruera klasser för att räkna ord på olika sätt. Vi kommer särskilt att fokusera på vilka abstrakta datatyper man kan använda.

## Datorarbete

D1. Börja med att bekanta dig med interfacet `TextProcessor`. Det innehåller metoder för att behandla inläst text, ett ord i taget, samt presentera ett resultat. Vi kommer att använda detta interface för att hantera olika slags textanalyser på ett enhetligt sätt.

```
public interface TextProcessor {

    /**
     * Anropas när ett ord lästs in.
     * Metoden ska uppdatera statistiken därefter.
     */
    void process(String w);

    /**
     * Anropas när samtliga ord i sekvensen lästs in.
     * Metoden ska skriva ut en sammanställning av statistiken.
     */
    void report();
}
```

(I just detta fall skulle vi även kunna använda en abstrakt klass, men interface passar bättre eftersom `TextProcessor` inte har några egna attribut eller metoder.)

D2. I projektet finns också en klass `SingleWordCounter`, som implementerar interfacet ovan. Denna klass är till för att räkna hur många gånger ett givet ord förekommer.

Klassen innehåller ett fel, som gör att antalet alltid blir 0 (noll). Finn felet och åtgärda det.

D3. Projektet innehåller därtill ett program `Holgersson.java`. Där skapas ett `SingleWordCounter`-objekt, som ska räkna antalet förekomster av ordet ”nils”. Vi stavar namnet med små bokstäver (gemener), eftersom programmet `Holgersson` omvandlar alla ord till gemener vid inläsning (med hjälp av `String`-metoden `toLowerCase`).

Därefter läses boken in. Bokens text finns i textfilen `nilsholg.txt`, och inleds med ett par dikter innan kapitel 1. Öppna gärna filen och se hur den ser ut.

Alla filens gås ord igenom och vårt `SingleWordCounter`-objekt uppdateras. Slutligen skrivs resultatet ut.

[1] Ett intressant exempel på en sådan historisk undersökning, större än vad vi har möjlighet att göra här, är: Cameron Blevins, ”Space, Nation, and the Triumph of Region: A View of the World from Houston”: *Journal of American History*, vol. 101, no. 1, 2014.

Kör programmet och kontrollera att det utskrivna resultatet stämmer. Namnet ”Nils” förekommer 75 gånger i Lagerlöfs roman. Om ditt resultat inte stämmer kan det bero på felet i föregående uppgift.

**Kommentar** till raden `s.useDelimiter(...)`: här konfigureras `Scanner`-objektet så att skiljetecknen `,.:;!?'"` filtreras bort från de inlästa orden. Anropet ser lite märkligt ut, eftersom vi använt ett *reguljärt uttryck* för att ange att alla dessa tecken ska betraktas som skiljetecken.[2] Du behöver inte bry dig om hur denna rad fungerar.

D4. Inför fortsättningen vill vi utöka programmet `Holgersson` så att det kan hantera flera `TextProcessor`-implementationer på samma text.

Ändra programmet så att det har en lista av `TextProcessor`-objekt. Till att börja med ska listan bara innehålla det enda objekt som finns sedan tidigare (och som räknar ”Nils”).

Varje gång ett ord lästs in ska alla `TextProcessor`-objekt i listan få sin `process`-metod anropad. När all text har lästs in ska alla `TextProcessor`-objekt skriva ut sina respektive resultat.

Lägg till en rad i ditt `Holgersson`-program, så att även antalet förekomster av ordet ”norge” räknas. Din lista ska alltså innehålla två `TextProcessor`-objekt, och du ska få följande resultat:

```
nils: 75
norge: 1
```

D5. Hittills har vi behövt skapa ett nytt objekt för varje ord vi räknar. Nu vill vi införa en ny typ av textanalys, där inte bara ett enda ord räknas, utan flera. Vi ska räkna hur många gånger de olika svenska landskapen nämns i boken.

Skapa en ny klass `MultiWordCounter`, som implementerar interfacet `TextProcessor` och fungerar enligt följande:

- Konstruktorn ska ta en vektor av strängar som parameter. Vektorn innehåller de ord vi vill räkna. Följande exempel visar hur en sådan konstruktor ska fungera:

  ```
  String[] landskap = { "blekinge", "bohuslän" /* , ... */ };
  TextProcessor r = new MultiWordCounter(landskap);
  ```

- Din klass `MultiWordCounter` ska ha exakt **ett** attribut, och det attributet ska vara av typen `Map` (med lämpliga typargument – jämför med förberedelseuppgifterna).

  Detta `Map`-attribut används för att hålla reda på hur många gånger de sökta orden (landskapsnamn i exemplet ovan) förekommer. Inledningsvis innehåller denna `Map` värdet 0 (noll) för varje sökt ord (landskapsnamn). Metoden `process` ökar antalet **om** ett givet ordet är ett av de sökta orden.

- Även om attributet har typen `Map`, så ska det objekt som skapas för det vara av den konkreta klassen `HashMap`. På så vis blir ditt program, så långt som möjligt, oberoende av vilken implementation av `Map`-interfacet som faktiskt används.

  Namnet `HashMap` ska alltså bara förekomma på **ett** ställe i klassen (förutom ev. import-satser).

- Metoden `report` ska skriva ut alla nycklar och respektive värden i din `Map`.

[2] Om du är särskilt intresserad kan du läsa mer om denna `Scanner`-finess här: `https://docs.oracle.com/javase/8/docs/api/java/util/Scanner.html`. Du kan även läsa mer om reguljära uttryck på exempelvis Wikipedia (`https://en.wikipedia.org/wiki/Regular_expression`).

**Tips:** Man kan gå igenom alla nycklar i en `Map` så här (om vi antar att `m` är en `Map` vars nycklar är av typ `String`):

```
for (String key : m.keySet()) {
    // gör något med key och m.get(key)
}
```

D6. Lägg till ett `MultiWordCounter`-objekt för landskapen i din lista i programmet `Holgersson`. Notera att det finns en användbar strängvektor given i programmet.

Kör programmet. I resultatet ser vi att gränstrakterna (Skåne, Lappland) nämns relativt ofta. Kanske var det angeläget att visa att Sverige ännu var rätt stort, trots att Norge lämnat unionen året innan? (Norge nämns ju endast en gång.)

D7. Vi ska nu ta fram information om boken på ett annat sätt. Genom att räkna alla ord, inte bara landskap, kan vi skapa oss en uppfattning om bokens innehåll. Vi måste emellertid utesluta vissa vanliga ord, som ”och”, ”ett” och ”att”, för att få ett meningsfullt resultat. Vi behöver alltså återigen en tabell av ord, men nu för att räkna *alla* ord, *utom* ett antal undantagsord.

Skapa en klass `GeneralWordCounter`, som implementerar interfacet `TextProcessor` och fungerar enligt följande:

- Konstruktorn ska ta en `Scanner` som parameter, så att undantagsord på så vis kan läsas in från filen `undantagsord.txt`:

  ```
  Scanner stopwords = new Scanner(new File("undantagsord.txt"));
  TextProcessor r = new GeneralWordCounter(stopwords);
  ```

  Filen `undantagsord.txt` finns i ditt Eclipse-projekt. Öppna gärna den och se hur den ser ut.
- Använd en `Map` för att hålla reda på hur många gånger respektive ord förekommer, samt en mängd (`Set`) för att hålla reda på undantagsorden.
- Även här ska du använda `HashMap`, men även här ska namnet `HashMap` bara förekomma på ett ställe i klassen (förutom ev. import-satser). För din mängd (`Set`) kan du välja implementation själv.
- Metoden `process` räknar alla ord, såvida de inte finns i mängden av undantagsord. Första gången ett nytt ord upptäcks läggs det till med antalet 1, och påföljande gånger samma ord upptäcks ökas dess antal med ett.
- Metoden `report` ska skriva ut alla ord som förekommer 200 gånger eller fler. (Om du vill får du gärna göra detta värde till en konstruktorparameter.)

D8. Lägg till ett `GeneralWordCounter`-objekt i din lista i programmet `Holgersson`. Kör programmet. Vilka är de vanligaste orden i Lagerlöfs bok?

Intresset för gränstrakterna framgår indirekt även här. Ledargåsen Akka är döpt efter ett lappländskt fjällmassiv, och pojken kommer från Skåne. Kanske skymtar vi bland orden även nationalromantikens fascination för den vilda naturen?

D9. Man kan mäta exekveringstiden för ett Java-avsnitt med hjälp av `System.nanoTime`. Denna metod returnerar antalet nanosekunder som förflutit sedan någon ospecificerad tidpunkt. Genom att subtrahera två sådana tidpunkter kan man få ett mått på förfluten tid, exempelvis i millisekunder:

```
long t0 = System.nanoTime();
... // kod vars exekveringstid vi vill mäta
long t1 = System.nanoTime();
System.out.println("tid: " + (t1 - t0) / 1000000.0 + " ms");
```

Justera programmet `Holgersson` så att tiden för programmet skrivs ut, så som ovan. Kör programmet tre gånger och beräkna medelvärdet av exekveringstiderna. Notera detta medelvärde.

D10. Justera din klass `GeneralWordCounter`, så att den använder `TreeMap` istället för `HashMap`. Om du har gjort rätt så räcker det ändra på ett ställe.

- Fungerar ditt program fortfarande?
- Hur påverkas ordningen i det utskrivna resultatet?
- Hur påverkas exekveringstiden? (Beräkna även här medelvärdet från tre körningar.)

D11. Fundera igenom följande, och diskutera med din handledare:

- Vad är det för skillnad på `Map` och `HashMap`?
- Vad är det för skillnad på `HashMap` och `TreeMap`? Vad beror skillnaderna på?

# Laboration 2 – länkad lista

*Mål:* Du ska lära dig implementera den abstrakta datatypen kö (FIFO queue) på två olika sätt; dels genom att delegera till Javas klass LinkedList och dels genom att implementera en från grunden med en länkad datastruktur. Du ska också lära dig att testa en klass genom att skriva testmetoder och använda testverktyget JUnit.

## Förberedelser

Läs igenom den inledande texten under rubriken ”Bakgrund” nedan. Lös uppgift D1-D4. Läs igenom övriga uppgifter. Läs PM om jUnit som finns på kursens webbsida.

## Bakgrund

På den här laborationen ska du på två olika sätt implementera en generisk klass `FifoQueue` med följande klassrubrik:

```
public class FifoQueue<E> extends AbstractQueue<E> implements Queue<E>
```

Klassen representerar en kö och ska implementera interfacet `Queue` i klassbiblioteket `java.util`.

Förklaring till varför klassen `FifoQueue` ärver `AbstractQueue`: Ibland kan man implementera vissa metoder med hjälp av andra metoder. Ex:

```
public boolean isEmpty () {
    return size() == 0;
}
```

För att underlätta för den som ska implementera det (stora) interfacet `Queue` finns den abstrakta klassen `AbstractQueue` som innehåller många av `Queue`-metoderna implementerade enligt detta mönster. Det som återstår att göra i klassen `FifoQueue` är att implementera metoderna `offer`, `size`, `peek`, `poll` och och `iterator`.

```
/**
 * Inserts the specified element into this queue, if possible.
 * post: the specified element is added to the rear of this queue.
 * @param x the element to insert
 * @return true if it was possible to add the element to this queue, else false
 */
boolean offer(E x);

/**
 * Returns the number of elements in this queue.
 * @return the number of elements in this queue
 */
int size();

/**
 * Retrieves, but does not remove, the head of this queue,
 * or returns null if this queue is empty.
 * @return the head of this queue, or null if the queue is empty
 */
E peek();

/**
 * Retrieves and removes the head of this queue,
 * or returns null if this queue is empty.
 * post: the head of the queue is removed if the queue was not empty
```

```
     * @return the head of this queue, or null if the queue is empty
     */
    E poll();

  /**
     * Returns an iterator over the elements in this queue.
     * @return an iterator over the elements in this queue
     */
    Iterator<E> iterator();
```

En kommentar till metoden `offer`: enligt specifikationen ska det element som är parameter sättas in enbart om det möjligt. I beskrivningen av interfacet `Queue` i Java-dokumentationen kan man utläsa att det är tillåtet att införa begränsningar på köer, t ex att en kö bara får innehålla ett visst antal element. Om man anropar metoden `offer` när kön redan innehåller det maximalt tillåtna antalet ska i sådana fall ingen insättning göras och metoden ska returnera `false`. I vår implementering ska inte någon sådan begränsning göras. Metoden ska därför i klassen `FifoQueue` alltid sätta in elementet och returnera `true`.

## Datorarbete

D1. Först ska du implementera kö-klassen genom att delegera till klassen `LinkedList` i paketet `java.util`. I projektet för laborationen finns det i paketet `queue_delegate` en fil med namnet `FifoQueue.java`. I klassen finns attributet `list` som du ska använda för att hålla reda på elementen i kön.

Implementera alla metoderna. Du ska inte behöva lägga till mer än en rad i varje metod. Läs dokumentationen av `LinkedList` på nätet så att du väljer rätt metoder. (Det finns t.ex. flera metoder i klassen för att hämta och ta bort element. De skiljer sig åt när det gäller vad som händer om listan är tom).

D2. I filen `TestFifoQueue` i paketet `testqueue` finns det testmetoder som kontrollerar funktionaliteten hos de metoder som implementeras i denna uppgift. Bekanta dig med testklassen så att du förstår vad som testas.

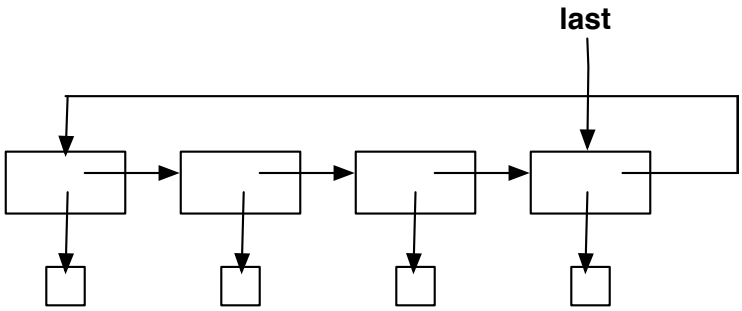
Testa din kö-klass och rätta till den tills du får grönt ljus.

D3. Nu ska du göra en ny implementering av klassen `FifoQueue`. En enkellänkad lista ska användas för elementen i kön. Noderna i listan ska representeras av följande privata nästlade klass (deklarerad i klassen `FifoQueue`):

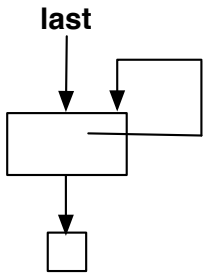
```
private static class QueueNode<E>  {
    E element;          // refererar till elementet
    QueueNode<E>  next; // refererar till efterföljande nod

    /* Konstruktor */
    QueueNode(E element) {
        this.element = element;
        next = null;
    }
}
```

Listan ska vara cirkulär, dvs i det sista elementet är inte referensen (`next`) till efterföljaren `null` utan i stället refererar den till det äldsta (första) elementet i listan. I en tom kö har attributet `last` värdet `null`. Kön representeras i klassen `FifoQueue` av ett attribut (`last`), som refererar till den listnod som innehåller det sist insatta elementet. Se fig. 1. Det är *inte* tillåtet att lägga till ett extra attribut `first` i köklassen.




Cirkulär lista att använda för att implementera en kö.
next-referenser och referenser till insatta element är utritade.
Noden längst till vänster innehåller det äldsta elementet i kön. Noden längst till höger, som attributet last refererar till, innehåller det senast insatta elementet.




Kö med ett enda element

Figur 1: Kö som representeras av cirkulär enkellänkad lista.

Observera att det bara är implementeringen av listan som är cirkulär. Utifrån sett är det en vanlig lista med början och slut.

I paketet `queue_singlelinkedlist` finns en fil med namnet `FifoQueue.java`. I klassen finns attributet `last`, metoderna som ska implementeras samt klassen `QueueNode`. Dessutom finns det ytterligare ett attribut `size` som representerar antalet element i kön.

Implementera alla metoder utom `iterator()`, som vi återkommer till i nästa uppgift. Testa metoderna parallellt. OBS! Ändra i testklassen så att du importerar `FifoQueue` från rätt paket.

Du bör börja med att implementera och testa metoderna `offer` och `size`. När du känner dig säker på att insättning fungerar kan du gå vidare till metoderna `peek` och `poll`.

D4. I denna uppgift ska `iterator()` implementeras. Metoden ska returnera ett objekt av en klass som implementerar interfacet `Iterator<E>`. En skiss över hur detta kan göras följer:

```
public class FifoQueue<E>  extends AbstractQueue<E> implements Queue<E> {
    ...
    public Iterator<E> iterator () {
        return new QueueIterator();
    }

    private class QueueIterator implements Iterator<E> {
        private QueueNode<E> pos;
        ...
        /* Konstruktor */
        private QueueIterator() {...}

        public boolean hasNext() {...}
```

```
        public E next() {...}
    }
}
```

Lägg märke till att i förslaget ovan är klassen `QueueIterator` en privat inre klass i klassen `FifoQueue`. Det innebär att man i `QueueIterator` har tillgång till alla attribut i sitt omgivande objekt av typen `FifoQueue`. Man kan alltså inne i ett objekt av typen `QueueIterator` använda attributen i klassen `FifoQueue`. Klassen `QueueIterator` och dess konstruktor kan vara privata eftersom det bara är den omgivande klassen som kommer att använda dem.

Läs specifikationen för metoderna i interfacet `Iterator<E>` i Javas dokumentation på nätet. Lägg in klassen `QueueIterator` i klassen `FifoQueue` enligt ovan och implementera konstruktorn, `hasNext` och `next`.

Testa!

D5. Ibland behöver man slå samman (konkatenera) två köer `q1` och `q2` till en kö bestående av alla element i `q1` följda av alla element i `q2`. Om man bara har tillgång till de befintliga metoderna på listan kan man successivt ta ut elementen ur `q2` med metoden `poll` och sätta in dem i `q1` med metoden `offer`. Om det finns $n$ element i `q2` anropas alltså båda metoderna $n$ gånger.

*OBS: Du ska göra en effektivare lösning genom att i stället utföra konkateneringen i en metod i klassen `FifoQueue`. Utnyttja den interna datastrukturen hos `FifoQueue` istället för att använda metoderna `offer` och `poll`.*

Implementera följande metod i `FifoQueue`:

```
/**
 * Appends the specified queue to this queue
 * post: all elements from the specified queue are appended
 * to this queue. The specified queue (q) is empty after the call.
 * @param q the queue to append
 * @throws IllegalArgumentException if this queue and q are identical
 */
public void append(FifoQueue<E> q);
```

I kommentaren står att `IllagalArgumentException` ska genereras som `queue` och `q1` är identiska. Med det menas att att man inte ska kunna slå ihop en kö med sig själv:

```
q1.append(q1);
```

D6. Skapa i paketet `testqueue` en fil `TestAppendFifoQueue.java` genom att i menyn `File` välja `New -> JUnit TestCase`. Ange gärna i dialogen att den klass som ska testas är `queue.FifoQueue` så får du automatiskt inlagt en importsats i filen. (Om du inte anger detta kan du manuellt lägga till `import queue_singlelinkedlist.FifoQueue` i början av testklassen). Lägg i denna fil in test för `append`-metoden. Testen ska åtminstone täcka in fyra fall för konkatenering:

- två tomma köer
- tom kö som konkateneras till icke-tom kö
- icke-tom kö som konkateneras till tom kö
- två icke-tomma köer
- försök att slå ihop en kö med sig själv

I testen ska du både kontrollera storlek *och* att elementen hamnat i rätt ordning. Glöm inte att kontrollera att den andra kön är tom efter sammanslagningen.

Kör testen och korrigera eventuella fel i `append`-metoden tills alla test lyckas.

D7. Fundera igenom följande, och diskutera med din handledare.

- Skulle du lika gärna kunna använda `ArrayList` för att lagra elementen i `FifoQueue` i uppgift D1?
- Jämför de två olika sätten att implementera `FifoQueue` (uppgift D1 resp. uppgift D3). Fördelar/nackdelar?
- Istället för att implementera en egen kö-klass skulle man helt enkelt kunna använda någon av kö-klasserna i java.util (`LinkedList` eller `Dequeue`). Ofta är det klokt att återanvända en befintlig implementering på detta sätt. I vilka situationer kan det vara olämpligt?

# Laboration 3 – grafiska användargränssnitt

*Mål:* Du ska träna på att skapa grafiska användargränssnitt med JavaFX och mönstret Model-View-Controller. Du ska också träna på att formulera lambdauttryck och använda dem för att bland annat hantera användarinteraktion på ett smidigt sätt.

## Förberedelser

F1. Läs igenom texten under rubriken ”Bakgrund”.

F2. Sök upp dokumentationen för `Map.Entry`.[3] Interfacet används för att beskriva ett par, av en nyckel och ett värde, i en `Map`. Bekanta dig med vilka metoder som finns i interfacet.

Det finns en metod i `Map` för att hämta en mängd (`Set`) av sådana `Map.Entry`-objekt. Vad heter metoden? **Svar:**

F3. Anta nu att vi har en lista (inte en mängd) av `Map.Entry` enligt ovan, med typargumenten `<Object, Integer>`. Vi vill använda den inbyggda metoden `sort` för att sortera listan, i fallande ordning, med avseende på *värden* (inte nycklar).

Fyll i lambdauttrycket nedan så att ordningen blir den önskade.

```
List<Map.Entry<Object, Integer>> l = ...;    // listan skapas

l.sort((e1, e2) -> ____________________________________________________);
```

Du ska **inte** använda metoden `comparingByValue` i `Map.Entry`, utan ta tillfället att träna på att skriva lambdauttryck själv.

**Tips:** lambdauttrycket ska ge ett värde som är <0, ==0 eller >0 beroende på hur de två jämförda elementen (`e1` och `e2`) förhåller sig till varandra. Det rör sig alltså om samma slags jämförelsevärde som exempelvis i metoden `compareTo` för strängar.

F4. Sök upp dokumentationen för följande JavaFX-klasser på nätet:[4]
`ListView`, `Button`, `TextField`, `HBox`

Du behöver inte läsa alla detaljer, men försök skapa dig en idé om vad klasserna är till för. (**Tips:** jämför med bilden i figur 1 i följande avsnitt.) Notera även metoder eller korta exempel som besvarar följande frågor:

Med vilken metod kan man få en `ListView` att scrollas till en given rad? **Svar:**

Med vilken metod anges vilken kod som körs när en `Button` klickas? (**Tips:** titta i superklassen.) **Svar:**

Vilken metod returnerar texten som användaren har skrivit i ett `TextField`? (**Tips:** titta i superklassen.) **Svar:**

Hur gör man för att lägga till ett element i en `HBox`? **Svar:**

[3] `https://docs.oracle.com/javase/8/docs/api/java/util/Map.Entry.html`
[4] `https://docs.oracle.com/javase/8/javafx/api/javafx/scene/control/ListView.html`
`https://docs.oracle.com/javase/8/javafx/api/javafx/scene/control/Button.html`
`https://docs.oracle.com/javase/8/javafx/api/javafx/scene/control/TextField.html`
`https://docs.oracle.com/javase/8/javafx/api/javafx/scene/layout/HBox.html`

F5. Sök upp dokumentationen för interfacet `ObservableList` på nätet.[5] Hur skiljer sig detta interface från `List`? Kan du förstå något av vad skillnaden kan betyda?

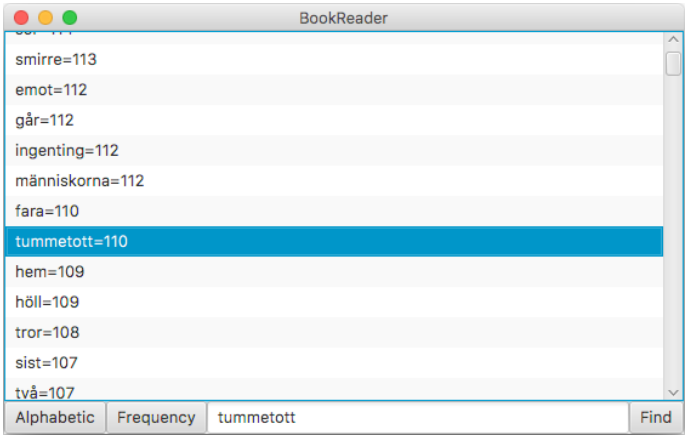
**Ledtråd:** notera vilka superinterface `List` respektive `ObservableList` har. De skiljer sig åt på en punkt i detta avseende.

F6. Lös uppgifterna D1, D2, D3 och D4.

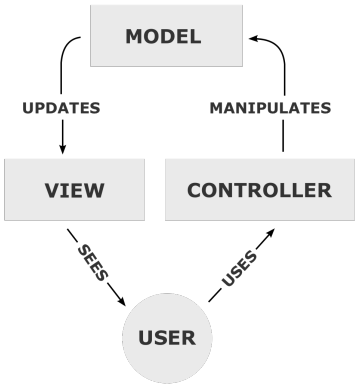
F7. Läs igenom övriga uppgifter.

## Bakgrund

I den här laborationen kommer du att skapa ett grafiskt användargränssnitt till det textanalysprogram du konstruerade i laboration 1. En bild på hur det kan se ut visas i figur 1 nedan.




Figur 1: Grafiskt användargränssnitt för vårt textanalysprogram.




Figur 2: Idén bakom mönstret Model-View-Controller. (Källa: Wikipedia, public domain.)

Huvuddelen av fönstret upptas av en lista, där bokens ord finns listade. På varje rad står ett ord och motsvarande antal.

Under listan finns några knappar och ett textfält. De två knapparna till vänster låter oss välja hur listan ska sorteras (alfabetiskt eller efter antal förekomster). I textfältet kan man skriva in ett ord, och när man trycker på knappen ”Find” söks motsvarande ord i listan upp och markeras.

### Ett sätt att strukturera grafiska applikationer

Program med grafiska användargränssnitt struktureras ofta enligt ett mönster som kallas *Model-View-Controller* (se figur 2 ovan).[6] Vi delar då in klasserna i tre olika delsystem, där varje delsystem består av en eller flera klasser:

**Modellen** hanterar data och algoritmer i applikationen. Det är alltså en central del av applikationen. Vi eftersträvar att göra modellen *oberoende* av användargränssnittet (vy och kontroll), så att samma modell kan återanvändas även med ett annat användargränssnitt.

I vårt system utgörs modellen av den klass `GeneralWordCounter`, som du implementerade i laboration 1. Då använde du denna modell med ett textbaserat gränssnitt.

[5] `https://docs.oracle.com/javase/8/javafx/api/javafx/collections/ObservableList.html`
[6] Mer att läsa finns exempelvis på Wikipedia: `https://en.wikipedia.org/wiki/Model-view-controller`

**Vyn** används för att visa modellens innehåll grafiskt, samt låta användaren interagera med den. Vyns klasser består alltså av knappar, inmatningsfält och annat. Sådana klasser finns ofta tillgängliga i färdiga paket, som JavaFX, Android eller Swing. I denna laboration använder vi JavaFX, ett standardpaket i Java.

**Kontrollen** förmedlar indata från vyn till modellen. Omvänt förmedlas även information från modellen för att presenteras i vyn. Med andra ord kopplar kontrollen samman de färdiga klasserna från JavaFX med vår modell.

Kontrollen utgörs här av två klasser: en JavaFX-baserad applikation, som du kommer att skriva, samt `ObservableList`, som är en del av JavaFX.

Det är inte alltid helt lätt att dra skarpa gränser mellan de tre delsystemen, och exakt hur mönstret Model-View-Controller tillämpas skiljer från applikation till applikation. (Som alternativ till ovanstående kan man exempelvis betrakta ObservableList som en del av modellen, snarare än kontrollen.) Indelningen i modell, vy och kontroll är ändå generell och användbar, och tillämpas ofta när man konstruerar applikationer med grafiska användargränssnitt. JavaFX, liksom de flesta andra ramverk av samma slag, är konstruerat utifrån detta synsätt.

### Något om interfacet `ObservableList`

Från vår modell får vi alltså en mängd av ord och deras respektive antal. För vår applikation vill vi istället ha en *lista* av ord och antal. Vi vill dessutom ha en speciell sorts lista, en *observerbar* lista. I detta sammanhang betyder det att listobjektet kan meddela förändringar till ett annat objekt, exempelvis en del av användargränssnittet.

Interfacet `ObservableList` passar bra för detta. Detta interface, samt en användbar färdig implementation av det, är en del av JavaFX.

För listan behöver vi bestämma oss för en elementtyp. Vi kommer att se i datoruppgifterna nedan att det är praktiskt att använda typen `Map.Entry` (med lämpliga typargument) till detta.

## Datorarbete

D1. Öppna din klass `GeneralWordCounter` från laboration 1. Vi behöver utöka klassen med en metod för att få tillgång till en mängd av ord-antal-par. Metoden finns i den klassen eftersom vi kommer att behöva det `Map`-attribut du införde där i laboration 1.

Lägg till följande metod i klassen `GeneralWordCounter`:

```
public Set<__________________________> getWords() {
    return ________________;
}
```

Den första streckade luckan ska ersättas med ett lämpligt typargument. För den andra luckan behövs en mängd av ord-antal-par. Mängden och listan har samma typargument.

**Tips:** förberedelseuppgifterna ger dig ledtrådar till luckornas innehåll.

D2. Vi ska nu skapa en klass som ska fylla rollen av Controller i Model-View-Controller-mönstret. I JavaFX utgår vi då från klassen `Application`, som vi skapar en subklass till. Välj ”New”→”Class” i menyn (eller tryck på knappen ”New Java Class”) och fyll i följande i dialogrutan:

- Name: (välj själv ett klassnamn, exempelvis `BookReaderController`)
- Superclass: `javafx.application.Application`
- Se till att rutorna för ”public static void main...” och ”inherited abstract methods” är markerade

När du klickat på ”Finish” får du en klass med tom `main`-metod och en tom `start`-metod. Kontrollera detta.

D3. Ett JavaFX-program fungerar lite annorlunda än vanligt, och vi behöver därför komplettera med några rader för att få ett tomt fönster att utgå ifrån. Fyll i metoderna `start` och `main` som följer:

```
public class BookReaderController extends Application {

    @Override
    public void start(Stage primaryStage) throws Exception {
        BorderPane root = new BorderPane();

        Scene scene = new Scene(root);
        primaryStage.setTitle("BookReader");
        primaryStage.setScene(scene);
        primaryStage.show();
    }

    public static void main(String[] args) {
        Application.launch(args);
    }
}
```

Några kommentarer:

- I JavaFX används `main`-metoden endast för att starta JavaFX. Det är därefter JavaFX som har kontrollen, och vår applikation anropas vid behov. Exempelvis skapas automatiskt ett objekt av vår `BookReaderController`, och därefter anropas metoden `start` på det nya objektet.

  Detta är alltså den enda rad som ska stå i `main`. Du ska inte lägga till något mer i `main`-metoden under denna laboration.
- I metoden `start` skapas innehållet för programmets fönster. Det görs genom att skapa ett `Scene`-objekt. Den sista raden i `start` ser till att fönstret visas på skärmen.
- Det behövs även ett objekt som bestämmer hur de olika delarna av användargränssnittet placeras i fönstret. Just nu finns ju inget sådant alls, men vi behöver ändå ett `BorderPane`-objekt, som kommer att få bli hemvist för de övriga delarna av användargränssnittet.

Pröva att köra programmet, och kontrollera att ett tomt fönster med rätt titel visas. Nu är vi strax redo att börja bygga upp användargränssnittet, det vill säga vyn i Model-View-Controller-mönstret. Först ska vi emellertid komplettera programmet med inläsningen av boken.

D4. Inför satser i `start` som skapar ett `GeneralWordCounter`-objekt och använder det för att räkna ord i *Nils Holgerssons underbara resa*, så som gjordes i laboration 1. (Det går bra att kopiera satserna för inläsningen från programmet `Holgersson` i den laborationen.)

Därefter behöver vi lägga till satser som gör följande:

- Läs ut en mängd (`Set`) med alla ord och deras antal. Använd metoden `getWords` som du just lagt till i `GeneralWordCounter`.
- Skapa en `ObservableList` med orden. Inför en lokal variabel för listan, eftersom du kommer att behöva den till fler saker senare. Det finns en bekväm metod i JavaFX för att skapa en `ObservableList` utifrån en mängd, som du strax ska få se.
- Skapa ett `ListView`-objekt för att visa listan i fönstret.

Inför följande rader på lämpligt ställe i metoden `start` (vi antar här att `counter` refererar till ett `GeneralWordCounter`-objekt):

```
ObservableList<__________________________> words
    = FXCollections.observableArrayList(counter.getWords());
ListView<__________________________> listView
    = new ListView<__________________________>(words);

root.setCenter(listView);
```

Återigen behöver de streckade luckorna ersättas med ett lämpligt typargument. Den sista raden låter vår `ListView` fylla ut hela vår `BorderPane`. (Den senare bestämmer ju hur användargränssnittets delar placeras i fönstret.)

Provkör programmet. Du ska nu få upp ett fönster med orden och deras antal (i någon ordning). Pröva att scrolla och ändra storleken på fönstret och se att `ListView`-objektet följer med.

D5. Vi vill nu även få plats med en rad i fönstrets nederkant, där knapparna och textfältet kan få plats. För detta ändamål ska vi använda ett `HBox`-objekt. Lägg till Java-satser i `start`-metoden för att:

- Skapa ett `HBox`-objekt.
- Skapa två knappar med etiketterna ”Alphabetic” respektive ”Frequency” (eller något liknande).
- Lägg till knapparna i din `HBox`.
- Lägg din `HBox` i fönstrets nederkant (vi antar att din `HBox` heter `hbox` här):

  ```
  root.setBottom(hbox);
  ```

Provkör programmet. Kontrollera att fönstret innehåller ordlistan och två knappar. Pröva gärna att trycka på knapparna.

D6. Nu ska vi få något att hända när man trycker på knapparna. Lägg till en rad i ditt program så att en enkel textsträng skrivs ut när man trycker på en av knapparna. Använd ett lambdauttryck med en `System.out.println`-sats.

Kör programmet och kontrollera att du får det förväntade resultatet när du trycker på knappen.

**Tips:** Lambdauttrycket ersätter en metod. Därför måste det ha lika många parametrar som metoden ifråga, även om parametrarna inte används i uttrycket.

D7. Ändra ditt program så att ordlistan sorteras alfabetiskt när man trycker på ”Alphabetic”, samt på antal förekomster när man trycker på ”Frequency”.

Vår lista är en `ObservableList`, och därför kan förändringarna förmedlas automatiskt till `ListView`. Kör programmet och verifiera att listans ordning förändras på rätt sätt när du trycker på knapparna.

D8. Lägg nu till två element i vyn, ett textfält och en knapp, för att göra det möjligt att söka upp ett ord i listan. Om det inskrivna ordet finns i listan scrollas listan så att ordet blir synligt, annars görs ingenting.

**Tips:** Det är praktiskt att använda ett lambdauttryck även här, eftersom du då har tillgång till alla lokala variabler i `start`. För att söka upp rätt rad i listan behöver du säkert använ-

da mer än en Java-sats. Ett lambdauttryck med flera Java-satser kan kapslas in i ett block, så här:

```
obj.someMethod(e -> {
                    ... ;
                    ... ;
                    ... ;
                });
```

D9. Lös två uppgifter ur avsnittet ”valbara uppgifter” nedan. Välj själv två (eller fler) uppgifter som du tycker verkar intressanta. Uppgifterna är oberoende och kan göras i valfri ordning. De bygger på att du själv söker i JavaFX-dokumentationen på nätet.

D10. Fundera igenom följande, och diskutera med din handledare.

- JavaFX (liksom många motsvarigheter, som Android och Swing) bygger på en princip som kallas för *Hollywoodprincipen*: ”Don’t call us – we’ll call you!”

  Kan du se i ditt program vad som menas med detta?
- Varför passar lambdauttryck särskilt bra ihop med denna princip?
- När kontrolldelen av ditt program ändrar ordningen i ordlistan uppdateras användargränssnittet (`TextView`) automatiskt. Kan du förstå något om hur det hänger ihop?

  **Tips:** det har med skillnaden mellan `List` och `ObservableList`, som du undersökte i förberedelseuppgifterna, att göra.

## Valbara uppgifter

V1. (En rad.) Gör så att knappen ”Find” aktiveras (trycks) automatiskt när man trycker Return. Det finns en användbar metod i klassen `Button` för detta.

V2. (En rad.) När man ändrar storlek på fönstret förändras inte knappraden. Ändra programmet så att textfältet växer och krymper i takt med att fönstrets storlek ändras.

Metoden `setHgrow` i klassen `HBox` är användbar för detta.

V3. (En eller ett par rader.) I ditt användargränssnitt kan man skriva in ett ord att söka efter. Om användaren råkar inleda eller avsluta ordet med ett eller fler mellanslag fungerar inte sökningen. Om användaren på samma sätt råkar skriva in versaler fungerar inte heller sökningen.

Ändra programmet så att sökningen fungerar, även om det inmatade ordet börjar/slutar på mellanslag eller innehåller versaler. Du har nytta av ett par lämpliga metoder i klassen `String`.

V4. (En eller ett par rader.) När man söker upp ett ord scrollas vår `ListView` till rätt plats (i uppgift D8 ovan), men det vore ännu bättre om det sökta ordet markeras också (på samma sätt som om man klickar i listan). Ändra programmet så att så sker.

I `ListView` hanteras markeringen av ett särskilt `SelectionModel`-objekt, som du kan hämta med metoden `getSelection`. Genom att anropa en metod på det objektet kan du få en viss rad att markeras.[7]

[7] `https://docs.oracle.com/javase/8/javafx/api/javafx/scene/control/SelectionModel.html#select-int-`

V5. (Några få rader.) När man söker efter ett ord som inte finns i boken vore det bra med en ruta som meddelar användaren detta. Använd klassen `Alert` i JavaFX för att visa en sådan ruta.

V6. (Några få rader.) För de två sorteringsknapparna passar det bra att använda s.k. radioknappar. En sådan markeras när den är intryckt, och bara en knapp i samma grupp kan vara intryckt i taget. Läs om klassen `RadioButton` i JavaFX och använd den för sorteringsknapparna.

V7. (Några få rader, och kanske mindre förändringar i befintlig kod.) Programmet vore mer användbart om man kunde välja vilken textfil som ska analyseras. Lägg till en knapp som låter användaren välja en fil att analysera. Använd klassen `FileChooser` i JavaFX.

# Laboration 4 – rekursion

*Mål:* Att ge träning i att skriva program med rekursiva algoritmer.

## Förberedelser

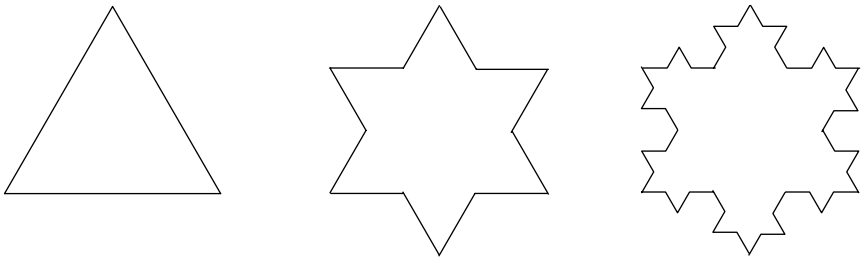
Läs igenom den inledande texten under rubrikerna ”Datorarbete”. Lös uppgift D1, D2, D3 och D4. Läs igenom de övriga uppgifterna.

## Datorarbete

Uppgiften i denna laboration är att skriva ett program som ritar fraktala figurer. Fraktal, en term som myntades av Mandelbrot 1975, är benämningen på bilder som i motsats till t.ex räta linjer, cirklar och trianglar är starkt sönderbrutna. De är uppbyggda av olika element med samma struktur.

Det finns många exempel på fraktaler i naturen såsom berglandskap, kustlinjer och virvelbilning i vattenfall. Inom matematiken använder man fraktaler för att beskriva sådana verkliga fenomen.

Studera fig 1 som visar fraktalen Kochs snöflinga (uppkallad efter den svenska matematikern Helge von Koch).



Figur 1: Kochs fraktal av ordning 0, 1 och 2.

Varje ny figur har åstadkommits genom att varje linje ersatts med en figur bestående av fyra nya linjer.

För att rita fraktalen Kochs snöflinga utgår man från en liksidig triangel. För att få en figur av ordning 1 ersätter man var och en av de tre linjerna med fyra nya linjer enligt fig 2. För att få en figur av ordning 2 ersätts varje linje i figuren av ordning 1 med fyra nya linjer osv.

Figur 2: En linje ersätts med fyra nya linjer.

En linje med längden length och riktningen alpha (vinkeln mellan linjen och x-axeln) ersätts alltså med fyra nya linjer som har följande längd och riktning:

- length/3, alpha
- length/3, alpha - 60°
- length/3, alpha + 60°
- length/3, alpha

Följande metod och tillhörande rekursiva hjälpmetod (i pseudokod) ritar Kochs snöflinga av en godtycklig ordning:

```
public void draw(int order, double length) {
    fractalLine(order, length, 0);
    fractalLine(order, length, 120);
    fractalLine(order, length, 240);
}

private void fractalLine(int order, double length, double alpha) {
    if (order == 0) {
        "rita en linje med längden length och riktningen alpha"
    } else {
        fractalLine(order-1, length/3, alpha);
        fractalLine(order-1, length/3, alpha-60);
        fractalLine(order-1, length/3, alpha+60);
        fractalLine(order-1, length/3, alpha);
    }
}
```

För att kunna rita olika fraktaler och fraktaler av olika ordning (grad av sönderbrytning) under laborationen finns det ett grafiskt användargränssnitt. Avsikten är att man som användare av det färdiga programmet skall kunna välja vilken fraktal man vill se ur en meny och kunna påverka fraktalens ordning genom att klicka på knappar. Användargränssnittet är i stora delar färdigt. Dock finns det bara en enda typ av fraktal att välja i menyn. Under laborationen kommer gränssnittet att behöva kompletteras så att man kan välja ytterligare en fraktaltyp.

D1. I projektet för laborationen finns tre paket: `fractal`, `koch` och `mountain`. I paketet `fractal` finns klasser för det grafiska användargränssnittet. Där finns bland annat en abstrakt klass `Fractal` som ska vara superklass till de egna ”fraktalklasser” du skapar. Vidare finns klassen `TurtleGraphics` med metoder för att rita linjer i användargränssnittets fönster. (Klassen påminner en hel del om den klass `Turtle` som behandlats i grundkursen.) De övriga klasserna i paketet beskriver användargränssnittets fönster med dess meny och knappar.

Huvudprogrammet finns i klassen `FractalApplication`. Kör detta program. Då öppnas ett fönster på skärmen:

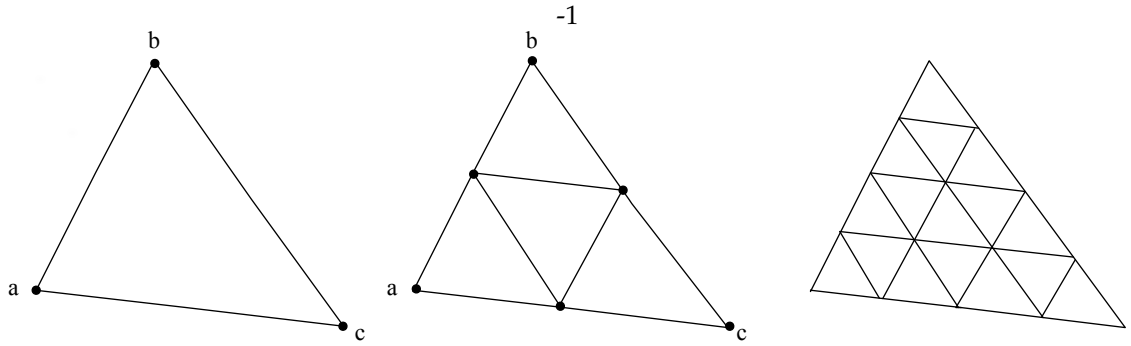
- Fönstret har en meny med namnet Fraktaler och texten ”Kochs triangel ordning 0” syns på fönstret.
- Om du öppnar menyn så syns det ett val: Kochs triangel. Om du väljer detta alternativ ur menyn så händer det ingenting. Det beror på att programmet vid detta val försöker rita Kochs fraktal av ordning 0, men denna metod gör ingenting förrän du själv kompletterat koden (uppgift D2).
- Fönstret har också knappar med texten ”>” resp. ”<” för att öka resp. minska den valda fraktalens ordning och rita den på nytt. Klicka på knappen ”>”. Texten i fönstret ändras då till ”Kochs triangel ordning 1” men fortfarande ser man ingen fraktal av de skäl som nämnts ovan.

D2. I paketet `koch` finns en påbörjad klass `Koch` med metoder för att rita Kochs snöflinga. Fyll i de rader som saknas i metoden `fractalLine`.

Observera att koden för att rita Kochs snöflinga här i texten är pseudokod och att du i den riktiga koden även behöver ha med ett objekt av klassen `TurtleGraphics` som parameter för att rita linjer.

Kör huvudprogrammet. Nu ska du se Kochs fraktal av ordning 0 på fönstret då programmet startar och du skall kunna se samma fraktal av högre ordning genom att använda knappen ”>”.

D3. I denna uppgift ska du lägga till ännu en fraktal till ditt tidigare program. Denna fraktal ska åskådliggöra ett bergsmassiv. En figur av ordning 0 utgörs av en triangel (gärna något sned). För att få nästa ordning ersätts varje triangel av fyra nya trianglar enligt fig 3.




Figur 3: Bergfraktal av ordning 0, 1 och 2.

I paketet `mountain` ska du lägga till en klass (liknande `Koch` i paketet `koch`) med metoder för att rita bergsfraktalen. Lämpliga parametrar till konstruktorn kan vara de tre startpunkterna. Till din hjälp finns den färdiga klassen `Point`, som beskriver en punkt.

*OBS! Bergsfraktalen ritas på liknande sått som Kochs snöflinga, men det finns ett par viktiga skillnader. Kochs snöflinga byggs upp av tre linjer. I varje rekursiv nivå ersätts en linje av fyra nya. En linje har en längd och en riktning. Bergsfraktalen består av* en *triangel. I varje rekursiv nivå ersätts en triangel av* fyra *nya trianglar. En triangel beskrivs av* tre *punkter.*

För att din nya fraktal ska synas i användargränsnittets meny och kunna ritas upp behöver du bara ändra i `start`-metoden i klassen `FractalApplication`. Öka vektorn `fractals` storlek och lägg in ett objekt av din nya fraktalklass i den. När du provkör programmet kommer du att se att det i menyn dyker upp ett alternativ till med det namn som metoden `getTitle()` i den nya fraktalklassen returnerar. Välj detta alternativ för att testa ritning av bergsmassiv. Tips! Lägg in det nya fraktalobjektet först i vektorn så kommer det att visas när programmet startar. Du kommer nog att provköra det en hel del gånger.

D4. Fraktalen i föregående uppgift blir för regelbunden för att likna ett bergsmassiv. Inför uppdelningen av en triangel i fyra nya, mindre trianglar ska därför mittpunkten förskjutas i y-led. Se fig. 4.

Förskjutningens storlek bestäms av funktionen `randFunc` som ger ett slumptal enligt en viss fördelning med avvikelsen `dev`:

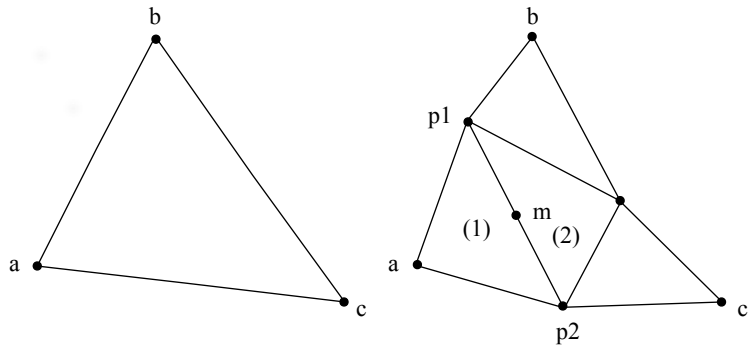
```
public static double randFunc(double dev) {
   double t = dev * Math.sqrt(-2 * Math.log(Math.random()));
   if (Math.random() < 0.5) {
      t = -t;
   }
   return t;
}
```

För varje nivå ska parametern `dev` till `randFunc` halveras. Om man glömmer att halvera denna parameter blir figuren för taggig. Låt gärna startvärdet på `dev` vara parameter till `Mountains` kontruktor.

De förskjutna mittpunkterna kommer att tillsammans med triangelns ursprungliga hörn att utgöra hörn i de fyra nya trianglarna.

Metoden `randFunc` är färdig att använda och finns i klassen `RandomUtilities`.

Berget kommer nu att ha lite mer oregelbundna former och se mer naturligt ut. Men det kommer att finnas vita fält här och var i figuren (löses i nästa deluppgift).




Figur 4: Bergfraktal av ordning 0 och 1. Mittpunkterna förskjuts - y-led innan en triangel delas upp i fyra nya trianglar.

D5. Av den högra figuren i fig. 4 ser vi att en speciell svårighet uppstår genom att trianglarna har vissa sidor gemensamma. När triangel (1) ska delas in i fyra mindre trianglar så ska mittpunkten m förskjutas. När senare triangel (2) ska delas in får inte m förskjutas en gång till. Så här kan man göra för att klara av denna svårighet:

Implementera först en klass `Side` som håller reda på en triangelsidas ändpunkter.

Skapa i klassen `Mountain` en map av typen `HashMap<Side, Point>` där `Side`-objekt kan lagras tillsammans med sin beräknade mittpunkt.

När en mittpunkt ska beräknas söker man först i mappen efter en sida med ändpunkterna p1, p2. Om en sådan sida finns använder man den redan beräknade mittpunkten. I annat fall beräknar man mittpunkten på samma sätt som tidigare och lagrar sidan med p1, p2 i mappen tillsammans med den beräknade mittpunkten.

Eftersom en sida bara används högst två gånger (första gången då mittpunkten beräknas och andra gången då den hittas i mappen) kan man ta bort sid-mittpunktsparet från mappen när man använt den. Sökningen blir snabbare då.

I mappen används klassen `Side` som nyckel. För att det ska fungera måste man där skugga metoderna `equals` och `hashCode` i klassen `Side`. Dessa två metoder används för att hitta nyckeln i mappen. (Hur mappen är implementerad och hur dessa två metoder används kommer att behandlas senare i kursen. När du tidigare har använt klassen `HashMap` har någon av Javas standardklasser används som nyckel. I dessa klasser är redan `equals` och `hashCode` skuggade.)

Metoden `hashCode` kan se ut så här:

```
public int hashCode() {
    return p1.hashCode() + p2.hashCode();
}
```

Inuti metoden `equals` är det sidornas ändpunkter som ska jämföras. Tänk på att man inte säkert vet ordningen på sidans ändpunkter.

# Laboration 5 – binära sökträd

*Mål:* Att ge träning i att implementera rekursiva algoritmer, speciellt för träd.

## Förberedelser

Läs igenom texten under rubriken ”Datorarbete”. Lös uppgifterna D1, D2, D3 och D4. Läs igenom övriga uppgifter.

## Datorarbete

Under denna laboration kommer delar av en klass för hantering av binära sökträd att implementeras.

I projektet för laborationen finns ett paket `bst` med en fil `BinarySearchTree.java`. Här finns en påbörjad implementering för hantering av binära sökträd.

Lägg märke till att klassens rubrik är:

```
public class BinarySearchTree<E extends Comparable<? super E>>
```

och *inte*:

```
public class BinarySearchTree<E extends Comparable<E>>
```

Den klassrubrik som används i laborationen gör klassen mera generell. Antag t.ex. att vi har följande klasser:

```
public class Person implements Comparable<Person> {
    ... attribut och metoder, däribland compareTo...
}
public class Student extends Person {
    ...
}
```

Objekt av typen `Student` går då att jämföra med varandra eftersom metoden `compareTo` finns (i superklassen `Person`). Det går däremot *inte* att deklarera ett binärt sökträd av typen `BinarySearchTree<Student>` om vi använder den andra klassrubriken. Det beror på att denna klassrubrik kräver att typen `E` är en klass som implementerar interfacet `Comparable<E>`. Detta villkor uppfylls inte av `Student`-klassen. Den implementerar ju inte interfacet `Comparable<Student>` utan interfacet `Comparable<Person>`. Genom att i klassrubriken i stället ange att `E` ska vara en klass som implementerar interfacet `Comparable<? super E>` anger vi att kravet på `E` är att den själv eller någon av dess superklasser implementerar `Comparable`-interfacet. Då går det bra att deklarera och skapa binära sökträd av typen `BinarySearchTree<Student>` .

Noderna i trädet representeras av en statisk nästlad klass `BinaryNode`.

D1. Börja med att i klassen `BinarySearchTree` implementera metoden

```
public int height();
```

som beräknar trädets höjd med rekursiv teknik.

Tips! Det är lämpligt att skriva en rekursiv privat metod som anropas i den publika metoden.

D2. I denna uppgift ska en metod med följande rubrik implementeras i klassen `BinarySearchTree`:

```
public boolean add(E x);
```

Metoden ska lägga in elementet `x` i trädet om det inte redan finns. Metoden ska returnera `true` om insättningen kunde utföras, annars `false`. Implementeringen ska vara rekursiv.

Implementera också metoden

```
public int size();
```

som returnerar antal noder i trädet.

Tips! Metoden `size` blir kort (en rad bör räcka).

D3. Implemetera i klassen `BinarySearchTree` metoden

```
public void printTree();
```

som skriver ut nodernas innehåll i inorder.

D4. Testa metoderna `height`, `add` och `size` genom att använda `JUnit`. Glöm inte att testa att din `add`-metod fungerar om man försöker sätta in dubbletter. Glöm inte heller att testa metoderna `height` och `size` i ett tomt träd.

D5. I klassen `BSTVisulizer` finns metoden `void drawTree(BinarySearchTree<?> bst)` som ritar ett binärt träd i ett fönster. (Inuti klassen `BSTVisulizer` används metoden `height` från uppgift D1 samt klasser i paketet `drawing`).

Skriv en `main`-metod i klassen `BinarySearchTree` som ritar trädet (anropa `drawTree`) samt skriver ut det (anropa `printTree`). Prova att skapa några träd av olika form, t. ex. ett skevt träd innehållande talen 1, 2, 3, 4 och 5 resp. träd med mer optimal form.

D6. Ett träd kan bli snett (obalanserat) när man gör många insättningar och borttagningar. Ett sätt att undvika detta är att balansera trädet i samband med varje insättning/borttagning enligt den metod som vi gått igenom på föreläsningarna (s.k. AVL-träd). Ett annat sätt kan vara att ”bygga om” trädet då och då när det blivit alltför snett förutsatt att detta inte sker alltför ofta. En algoritm som bygger om trädet till ett träd som har maximalt antal noder på alla nivåer utom den som ligger längst bort från roten ska implementeras i denna uppgift.

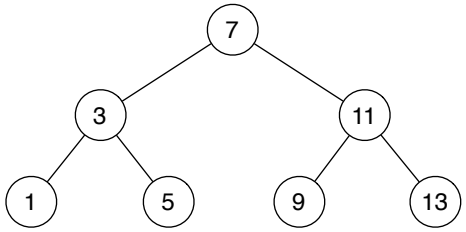
Om man placerar alla element från trädet i växande ordning i en vektor är det sedan enkelt att bygga ett träd där antalet noder i vänster respektive höger underträd aldrig skiljer sig med mer än ett och som därför är balanserat. Algoritmen är följande: Skapa en nod som innehåller mittelelementet i vektorn. Bygg (rekursivt) ett träd som innehåller elementen till vänster om mittelementet och ett träd som innehåller elementen till höger om mittelementet. Låt dessa båda träd bli vänster respektive höger barn till roten.

Antag t.ex. att vektorn innehåller heltalen 1, 3, 5, 7, 9, 11, 13. Trädet som byggs får då det utseende som visas i fig. 1.

En metod med följande rubrik ska implementeras i klassen `BinarySearchTree`:

```
/**
 * Builds a balanced tree from the elements in the tree.
 */
public void rebuild();
```

Metoden ska implementeras så att den går igenom trädet i inorder och bildar en vektor med innehållet i växande ordning. Sedan ska trädet byggas enligt algoritmen ovan. Följande rekursiva hjälpmetoder ska implementeras och användas inuti `rebuild`:




Figur 1: Binärt sökträd med nycklar 1, 3, 5, 7, 9, 11, 13 byggt enligt algoritmen i texten.

```
/*
 * Adds all elements from the tree rooted at n in inorder to the array a
 * starting at a[index].
 * Returns the index of the last inserted element + 1 (the first empty
 * position in a).
 */
private int toArray(BinaryNode<E> n, E[] a, int index);

/*
 * Builds a complete tree from the elements a[first]..a[last].
 * Elements in the array a are assumed to be in ascending order.
 * Returns the root of tree.
 */
private BinaryNode<E> buildTree(E[] a, int first, int last);
```

Inuti metoden `rebuild` ska du deklarera och skapa en vektor av typen `E[]`. Eftersom man inte kan skapa en vektor där elementen är av parametriserad typ får man göra så här:

```
E[] a = (E[]) new Comparable[size];
```

Tips! Att få `toArray` att returnera korrekt värde kräver lite eftertanke. Tänk på att det i de rekursiva anropen för subträden kan läggas till element i vektorn `a`. Tänk också på att dessa anrop returnerar värden som ska tas om hand.

Testa genom att skriva en main-metod som bygger ett snett träd genom successiva add-anrop och som sedan anropar `rebuild()`. Låt sedan main-metoden rita trädet och kontrollera att det blivit ett balanserat träd.

# Laboration 6 – hashtabell

*Mål:* Att ge förståelse för den abstrakta datatypen `Map` och datastrukturen `hashtabell`.

## Förberedelser

Läs igenom texten under rubriken ”Datorarbete”. Lös uppgifterna D1 – D6.

## Datorarbete

I denna uppgift ska du göra en implementering av en *öppen hashtabell* (”separate chaining”). Listorna ska konstrueras från grunden med enkellänkade listor.

Din klass ska implementera gränssnittet `map.Map` som innehåller en delmängd av de metoder som finns i gränssnittet `java.util.Map`. Dokumentationen för `java.util.Map` på nätet beskriver metoderna.

```
package map;

interface Map<K,V> {
    static interface Entry<K,V> {
        K getKey();
        V getValue();
        V setValue(V value);
    }
    V get(Object arg0);
    boolean isEmpty();
    V put(K arg0, V arg1);
    V remove(Object arg0);
    int size();
}
```

Den nästlade klass som ska implementera `Map.Entry<K,V>` ska ha ett attribut som är en länk till nästa element i listan.

```
 private static class Entry<K,V> implements Map.Entry<K,V> {
        private K key;
        private V value;
        private Entry<K,V> next;
        ...

}
```

Dessutom ska du skriva en metod som skriver ut hashtabellens innehåll.

Det kan vara lämpligt att implementera en sak i taget och testa när så är möjligt. I paketet `test` finns en färdig testklass `TestSimpleHashMap`.

D1. I projektet för laborationen finns ett paket `map`. Skapa i detta paket klassen

```
public class SimpleHashMap<K,V> implements Map<K,V>
```

Så här kan man göra i Eclipse:

- Markera paketet med höger musknapp och välj New–>Class.
- Fyll i namnet på klassen (`SimpleHashMap<K,V>`) i fältet `Name`.
- Klicka på Add-knappen vid textfältet Interfaces. Då öppnas ett nytt dialogfönster. Fyll i interfacets namn (`map.Map`). Under “matching items” kommer medan du skri-

ver förslag på interface som matchar ditt namn. Markera här interfacet (`map.Map`) och klicka på OK. Klicka på Finish i det första dialogfönstret (New Java Class).

Den nya klassen öppnas normalt i editorn och om du inte har ändrat inställningarna i Eclipse så bör nu din klass `SimpleHashMap` innehålla ”stubbar” för de metoder som föreskrivs av interfacet (`map.Map`). Om det inte innehåller stubbar kan du skapa dem genom att på menyn Source välja alternativet Override/Implement methods.

Nu återstår det att lägga in den nästlade klassen

```
private static class Entry<K,V> implements Map.Entry<K,V>
```

Om du vill använda dialogen även för detta ska du

- Markera klassen `SimpleHashMap` och välja New–>Class igen.
- I dialogen föreslås nu `SimpleHashMap` som Enclosing type (eftersom vi markerade den klassen).
- Kryssa i rutan vid Enclosing type.
- Fyll i namnet på den nästlade klassen (`Entry<K,V>`) i fältet Name.
- Markera att klassen ska vara statisk genom att kryssa i rutan static.
- Även den nästlade klassen ska implementera ett interface. I princip skulle samma teknik som beskrivits ovan för den omgivande klassen nu kunna användas. Det verkar dock som om Eclipse har vissa svårigheter med typparametrar för interface som implementeras av inre klasser. Därför föreslås att detta tillägg görs manuellt. Klicka alltså på Finish.

Skriv i filen in att den nästlade klassen implementerar interfacet `Map.Entry<K,V>`. Välj därefter från menyn Source alternativet Override/Implement methods. Du får då stubbar för de metoder som interfacet föreskriver.

D2. Implementera konstruktorn och metoderna i den nästlade klassen `Entry`. Skugga också metoden `toString()` som ska returnera nyckel och värde med "=" emellan.

D3. Bland attributen i `SimpleHashMap` ska det finnas en vektor (`table`) med `Entry`-element. Lägg in detta och andra lämpliga attribut och implementera följande två konstruktorer (som skapar vektorn):

```
/** Constructs an empty hashmap with the default initial capacity (16)
   and the default load factor (0.75). */
SimpleHashMap();

/** Constructs an empty hashmap with the specified initial capacity
    and the default load factor (0.75). */
SimpleHashMap(int capacity);
```

Man får inte använda en parametriserad typ när man skapar en vektor i Java. Gör därför så här:

```
(Entry<K,V>[]) new Entry[capacity];
```

D4. För att kunna kontrollera att informationen lagras på rätt sätt ska du i klassen `SimpleHashMap` skriva en metod `String show()` som ger en sträng med innehållet på varje position i tabellen på egen rad.

```
0        key=value key=value etc.
1        key=value key=value etc.
...
```

D5. Implementera `size()` och `isEmpty()`.

D6. För att enkelt kunna implementera de övriga metoderna är det lämpligt att ha två privata hjälpmetoder:

```
private int index(K key)
private Entry<K,V> find(int index, K key)
```

`index(key)` ska returnera det index som ska användas för nyckeln `key`.

`find(index, key)` ska returnera det `Entry`-par som har nyckeln `key` i listan som finns på position `index` i tabellen. Om det inte finns något sådant ska metoden returnera `null`.

D7. Implementera `put(K key, V value)`. Om det fanns ett gammalt värde ska detta returneras. Annars returneras `null`. Tänk på att fyllnadsgraden inte ska överstiga 0.75 och öka kapaciteten om så är fallet. Det är lämpligt att skriva en privat metod `rehash` för detta.

D8. Implementera `get(Object object)`. Argumentet måste omvandlas till typen `K`. Om nyckeln inte finns returneras `null`.

D9. Nu går det bra att testa. Öppna klassen `TestSimpleHashMap` och ta bort kommentarstecknen på de rader där `SimpleHashMap`-objekten skapas i metoden `setUp`. (De är bortkommenterade för att inte orsaka kompileringsfel innan klassen `SimpleHashMap` existerar.) En del tester i `TestSimpleHashMap` använder metoden `remove` som ej är implementerad ännu. Kör testen ändå och bortse från de fel som avser ännu inte implementerade metoder.

Tips! Om det inte fungerar som det ska så kommentera bort koden med anropet av `rehash`. Om programmet då går igenom testerna har du isolerat felet till koden för rehashingen.

D10. Testerna i JUnit testar en hel del, men inte allt. Det är svårt att skriva ett fullständigt test av hashtabellen utan att förutsätta för mycket om hur den är implementerad. Skriv därför en main-metod där du skapar ett `SimpleHashMap`-objekt, och lägger in slumpmässigt valda element samt skriver ut innehållet med hjälp av metoden `show`. Om både nyckel och värde i varje par är samma `Integer`-värde och kapaciteten 10 blir det lätt att kontrollera resultatet. Använd både positiva och negativa tal. Öka antal element och kontrollera att ökningen av kapacitet (rehashing) fungerar som den ska. Kontrollera att listorna inte blir orimligt långa.

D11. Implementera `remove(Object key)`. När man ska implementera `remove` bestämmer man först i vilken lista som nyckeln borde finnas. Följande fall måste hanteras:

1. Listan är `null`.
2. `key` finns i det första elementet i listan.
3. `key` finns senare i listan.
4. `key` finns inte i listan.

Testa!