# Datorlaborationer, EDAA01 Programmeringsteknik – fördjupningskurs

Datorlaborationerna ger exempel på tillämpningar av det material som behandlas under kursen.

- *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.
- *Laborationerna är obligatoriska.* Det betyder att du måste bli godkänd på alla uppgifterna under ordinarie laborationstid. Om du skulle vara sjuk vid något laborationstillfälle så måste du anmäla detta till mig (`anna.axelsson@cs.lth.se`, 046–222 98 12) före laborationen.

  Om du varit sjuk bör du göra uppgiften på egen hand och redovisa den under påföljande laborationstillfälle. Det kommer också att anordnas en uppsamlingslaboration i slutet av kursen. Detta tillfälle erbjuds dock bara till dem som haft giltigt skäl för frånvaro på någon laboration eller som varit närvarande vid samtliga laborationer men, trots rimliga förberedelser, inte hunnit bli färdiga. Övriga, som inte blir färdiga med laborationerna under kursen, hänvisas till kommande kurstillfällen. Kursen ges två gånger per år, en gång under läsperiod 1-2 och en gång under läsperiod 3.
- *Laborationerna kräver en hel del förberedelser.* I början av varje laboration finns anvisningar om förberedelser under rubrikerna Läsanvisningar och Förberedelser. Läsanvisningarna anger vilka avsnitt i läroboken som ska läsas. 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. Ni får naturligtvis under dessa förberedelser gärna kontakta mig om ni stöter på svårigheter.
- Du måste för varje laboration se till att laborationsledaren noterar dig som godkänd på listan på nästa sida.

Om du hittar någonting i uppgifterna eller andra anvisningar som är felaktigt eller oklart så är jag tacksam om du meddelar detta till `anna.axelsson@cs.lth.se`.

## Innehåll

## Programmeringsteknik — fördjupningskurs, godkända laborationsuppgifter

Skriv ditt namn och din namnteckning nedan:

Namn: .....

Namnteckning: .....

| Godkänd laboration | Datum | Laborationsledarens namnteckning |
|---|---|---|
| 1 | | |
| 2 | | |
| 3 | | |
| 4 | | |
| 5 | | |
| 6 | | |

# Laboration 1 – generisk klass, testning med jUnit

*Mål:* Du ska befästa och utöka dina kunskaper om Java inom de områden som behandlats under de två första föreläsningarna: arv, interface, exceptions, generik och skuggning av metoder (overriding). Du kommer också att få lära dig utföra test av klasser med hjälp av verktyget JUnit, som finns tillgängligt i Eclipse.

## Läsanvisningar

Läs igenom följande avsnitt i boken: 1.1–1.4, 1.6, 2.1 (1.2, 1.3, 2.1–2.4, 3.1–3.2, 3.5, 4.1 i gamla upplagan). Läs också på föreläsningsbilderna från föreläsning 1 och 2, som finns på kursens hemsida, samt utdelat PM om JUnit.

## Förberedelser

### Eclipse

Under laborationerna ska programutvecklingsmiljön Eclipse användas. Ni som inte använt Eclipse kan läsa mer om detta verktyg i ett PM som delats ut och som även finns på kursens hemsida.

På studentdatorerna finns Eclipse installerat. Om du vill jobba med laborationen hemma och inte redan laddat ner Eclipse så finns anvisningar för detta på hemsidan. Nedan ges kortfattade anvisningar för att hämta ett förberett arbetsområde för kursens laborationer och för att starta Eclipse på studentdatorerna.

Det finns ett arbetsområde (workspace) för laborationerna på denna kurs i packad form i filen `edaa01-workspace.zip`. Filen finns att hämta från kursens hemsida. Starta en webbläsare och gå till adressen `http://cs.lth.se/edaa01ht`. Klicka där på länken Laborationer i menyn till vänster. Ladda ner filen `edaa01-workspace.zip` till din hemkatalog och packa upp den.

Starta därefter Eclipse. Efter en stund får du en dialogruta där Eclipse frågar efter vilket ”workspace” (arbetsområde) du vill använda. Leta efter arbetsområdet `edaa01-workspace` och välj detta. I Eclipse-fönstret visas då i projektvyn (”Package Explorer”, längst till vänster) alla projekten i arbetsområdet. Projekten öppnas genom att man klickar på pilen bredvid projektnamnet. I projekten finns det förberett ett eller flera paket. Dessa öppnas genom att man klickar på pilen bredvid deras namn. Enskilda filer i paketen öppnas i editorn genom att man dubbelklickar på deras namn.

### Förberedelser för den första laborationen

Läs igenom den inledande texten nedan under rubriken ”Datorarbete”. Lös uppgift D1 och D2. Läs igenom uppgifterna D3, D4 och D5.

## Datorarbete

Under denna laboration ska en klass `ArraySet<E>` implementeras. Klassen ska implementera ett enkelt interface, `SimpleSet<E>`, för hantering av en mängd av element. Klassen ska också testas med hjälp av verktyget JUnit. Därefter ska ytterligare en klass, som ärver från `ArraySet<E>` implementeras och testas. Interfacet `SimpleSet<E>` finns i paketet `set` i projektkatalogen `lab1` och har följande specifikation:

```
public interface SimpleSet<E> extends Iterable<E> {
   /**
    * Adds the specified element to this set, if it is not already present.
    * post: x is added to the set if it is not already present
    * @param  x the element to be added
    * @return true if the specified element was added
    */
   boolean add(E x);

   /**
    * Removes the specified element from this set if it is present.
    * post: x is removed if it was present
    * @param x the element to remove - if present
    * @return true if the set contained the specified element
    */
   boolean remove(Object x);

   /**
    * Returns true if this set contains the specified element.
    * @param x the element whose presence is to be tested
    * @return true if this set contains the specified element
    */
   boolean contains(Object x);

   /**
    * Returns true if this set contains no elements.
    * @return true if this set contains no elements
    */
   boolean isEmpty();

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

Notera att interfacet ärver från interfacet `Iterable<E>`. Det innebär att klasser som implementerar interfacet `SimpleSet` också måste implementera följande metod (som är den enda metoden i interfacet `Iterable`):

```
   /**
    * Returns an iterator over a set of elements of type E.
    * @return an iterator over a set of elements of type E
    */
   Iterator<E> iterator();
```

D1. I paketet `set` finns det en java-fil med namnet `ArraySet.java`. Öppna denna fil i editorn. Filen innehåller en klass `ArraySet<E>`. I klassrubriken anges att `ArraySet<E>` implementerar interfacet `SimpelSet<E>`. I klassen finns också ett attribut `set` av typ `ArrayList<E>` som ska innehålla de element som sätts in i mängden.

I denna uppgift ska konstruktorerna och alla metoder utom `addAll` implementeras. Alla metoder kan enkelt implementeras genom att man anropar motsvarande metod i klassen `ArrayList<E>`. Därför är det lämpligt att ta fram dokumentationen av denna klass i en webbläsare. Länk till Javas dokumentation finns på kursens hemsida. Observera dock att eftersom dubbletter ej tillåts så måste man ta hänsyn till detta i `add`-metoden.

I projektkatalogen `lab1` finns i paketet `test` en fil med namnet `TestArraySet.java`. Denna innehåller en rad test av klassen `ArraySet`. Studera innehållet i filen.

Testfilen kan exekveras genom att man markerar den med högerknappen, väljer Run As –> JUnit test.

Det är lämpligt att man testar metoderna i `ArraySet` efterhand som man implementerar dem. Normalt skriver man test parallellt med att man implementerar. I denna uppgift är test av samtliga metoder redan färdigskrivna. När man bara implementerat ett fåtal av metoderna i klassen `ArraySet` och vill testa dessa, kan man kommentera bort de test som avser andra metoder. Efterhand som man implementerar fler metoder kan man sedan avkommentera testen. Alternativt kan man alltid köra alla test och bortse från fel som avser ännu inte implementerade metoder.

Implementera, testa, rätta eventuella fel och testa igen tills det blir grönt ljus i alla test av de metoder som implementerats.

D2. Implementera metoden `addAll` i klassen `ArraySet`:

```
/**
 * Adds all of the elements in the specified set, for which it is
 * possible, to this set.
 * post: all elements, for which it is possible, in the
 * specified set are added to this set.
 * @return   true if this set changed as a result of the call
 */
boolean addAll(SimpleSet<? extends E> s);
```

Metoden har som parameter en mängd `s`. Lägg märke till att vi använder interfacenamnet `SimpleSet` för att ange formell typ för denna parameter. Det innebär att vid anrop får den aktuella parametern vara en instans av någon klass som implementerar interfacet, t ex `ArraySet`.

Notera också att den formella typen för `s` inte är `SimpleSet<E>` utan `SimpleSet<? extends E>`. Anledningen är följande: I en mängd av typen `SimpleSet<E>` är det tillåtet att sätta in objekt av typen `E` eller objekt av någon subklass till `E`. Antag t ex att vi deklarerar en mängd

```
ArraySet<Person> personSet = new ArraySet<Person>();
```

Antag vidare att klassen `Student` är subklass till `Person`. Då är det tillåtet att sätta in objekt av typen `Student` i `personSet`:

```
Student  s = new Student(...);
personSet.add(s);
```

I metoden `addAll` har vi därför ingen anledning att kräva att den mängd `s` som är parameter är av typ `SimpleSet<E>`. Det går bra att lägga till alla element som finns i `s` även när dess formella typ är `SimpleSet<T>` där `T` är en subtyp till `E`. Detta anger vi i Java genom att specificera typparametern som `<? extends E>`. `?` är ett s.k. wildcard och deklarationen `SimpleSet<? extends E>` kan utläsas som att typparametern får vara en godtycklig subklass till `E` (inklusive `E`).

Den givna signaturen för metoden `addAll` innebär att följande kod är korrekt:

```
ArraySet<Person> personSet = new ArraySet<Person>();
ArraySet<Student> studentSet = new ArraySet<Student>();
...
personSet.addAll(studentSet);
```

Implementera metoden `addAll`. Du behöver då iterera över elementen i mängden `s`. Du kan använda en s.k. `foreach`-sats eller explicit använda en iterator. Du skapar en iterator för `s` på följande sätt:

```
Iterator<? extends E> itr = s.iterator();
```

Om du i stället vill använda en `foreach`-sats, skriver du:

```
for (E e : s) {
    ...
}
```

Vi vet ju att alla element i `s` är objekt av klassen `E` eller en subklass till `E`.

Färdiga testmetoder för metoden `addAll` finns i `AddAllTest.java`. Testa. Korrigera eventuellt och testa på nytt tills alla test går igenom.

D3. I denna uppgift ska en ny klass, som ärver `ArraySet`, implementeras. Klassen har följande specifikation:

```
public class MaxSet<E extends Comparable<E>> extends ArraySet<E> {
   /**
    * Returns the currently largest element in this set.
    * pre: the set is not empty
    * post: the set is unchanged
    * @return the currently largest element in this set
    * @throws NoSuchElementException if this set is empty
    */
   public E getMax();
}
```

I klassen ska man hålla reda på vilket element som för tillfället är det största elementet i mängden. Användare ska kunna ta reda på vilket detta element är genom att anropa metoden `getMax`.

Eftersom begreppet ”största element” är meningsfullt enbart om elementen i mängden går att jämföra med varandra, har klassen `MaxSet` en typparameter (`E`) med en begränsning (`extends Comparable<E>`). På detta sätt anger man att de element som sätts in i mängder av typen `MaxSet` måste impelementera interfacet `Comparable`. Detta interface (som finns i Javas klassbibliotek) innehåller en metod för att jämföra element enligt följande:

```
public interface Comparable<E> {
   /**
    * Compares this object with the specified object for order.
    * @param x the object to be compared
    * @return a negative integer, zero or a positive integer as this
    * object is less than, equal to, or greater than the specified object
    */
   int compareTo(E x);
}
```

I implementeringen av klassen `MaxSet` kan vi därmed utgå ifrån att vi kan jämföra element av typen `E` med hjälp av metoden `compareTo`.

I paketet `set` finns en fil med namnet `MaxSet.java`. Öppna denna fil i editorn. I klassen finns ett attribut `maxElement` som ska hålla reda på det för tillfället största elementet i mängden.

`getMax()` ska alltså returnera `maxElement` om mängden inte är tom. Om mängden är tom ska den generera (`throw`) ett undantag av typen `NoSuchElementException`. Klassen `NoSuchElementException` finns färdigimplementerad i Java-biblioteket. Implementera `getMax()` enligt denna specifikation.

För att metoden `getMax()` ska uppfylla sin uppgift krävs det naturligtvis att vi i implementeringen av klassen ser till att attributet `maxElement` alltid refererar till det för tillfället största elementet i mängden. De operationer som kan påverka vilket detta element är, är de som förändrar innehållet i mängden d.v.s. operationer för insättning och borttagning. Därför måste dessa metoder skuggas (override) i `MaxSet`. I filen finns dessa metoder med tomma metodkroppar och det är din uppgift att implementera dem. Tänk på att du vid behov kan anropa motsvarande metoder i superklassen. Om du t ex i implementeringen av `add` i `MaxSet` vill anropa metoden `add` i superklassen så gör du detta genom `super.add(...)`.

Observera att du inte ska ändra skyddsnivå på attributet `data` i superklassen `ArraySet`. Den ska fortfarande vara deklarerad `private`.

I filen `TestMaxSet.java` i paketet `test` finns färdigskrivna test av klassen `MaxSet`. Du kan arbeta med implementeringen av klassen och test av metoderna parallellt. Implementera, testa, rätta fel och testa igen tills alla test ger grönt ljus.

D4. Använd klassen `MaxSet` för att implementera följande metod:

```
public static int[] uniqueElements(int[] ints);
```

Metoden ska returnera en vektor bestående av de tal som förekommer i vektorn `ints`. I resultatet ska det alltså inte finnas några dubbletter. Den returnerade vektorn ska vara sorterad i växande ordning. Om `ints` = {7, 5, 3, 5, 2, 2, 7} vid anropet ska resultatet bli vektorn {2, 3, 5, 7}.

Skapa en ny klass `RemoveDuplicates` i paketet `set` där metoden placeras. Du skapar en ny klass genom att markera paketet med högerknappen och välja New –> Class (Det finns flera sätt att skapa nya klasser: man kan använda ikonerna så som beskrivs i PM om Eclipse, eller man kan markera paketet och välja New –> Class från File-menyn).

Testa metoden med en vektor {7, 5, 3, 5, 2, 2, 7} genom att i samma klass skriva en `main`-metod.

D5. Naturligtvis räcker det inte att bara testa metoden `uniqueElements` med ett enda fall. Skriv därför en testklass i jUnit som testar metoden. Det ska finnas testmetoder för relevanta fall, (variera antal element, samma element, olika element ...).

Tips! Hämta inspiration från de färdiga testmetoderna.

Se också dokumentationen (javadoc) av jUnits klasser på nätet. I klassen `Assert` hittar du alla `assert`-metoderna som kan användas för att skriva testmetoder. Där finns bl.a. `assertArrayEquals` som kan användas för att jämföra innehållet i två vektorer.

# Laboration 2 – länkad struktur

*Mål:* Du ska lära dig implementera länkade datastrukturer genom att implementera en klass för köhantering. I samband med det får du träna på att implementera interfacet `Iterator<E>`. Du ska också lära dig att testa en klass genom att skriva testmetoder och använda testverktyget JUnit.

## Läsanvisningar

Läs följande avsnitt i läroboken: 2.2–2.6, 2.9, 4.1–4.3 (4.2–4.5, 4.8, 6.1–6.3 i gamla upplagan). Läs också föreläsningsbilder från föreläsningarna 3 och 4, som behandlar motsvarande avsnitt. Dessa finns på kursens hemsida.

## Förberedelser

Läs igenom den inledande texten under rubriken ”Datorarbete” nedan och lös uppgift D1. Läs igenom och sätt dig in i uppgifterna D2 och D3.

## Datorarbete

I denna uppgift ska du 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`. Detta interface har många metoder. En del av dessa kan implementeras genom att man anropar andra metoder i interfacet. I Javas klassbibliotek finns det en abstrakt klass `AbstractQueue` där detta är genomfört. Vi låter därför vår klass ärva `AbstractQueue`. Därmed återstår det bara att implementera följande metoder:

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

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

/**
 * 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();
```

```
/**
 * 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();
```

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 `size`, `iterator`, `offer`, `poll` och `peek`.

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 någon sådan begränsning inte göras. Metoden ska därför i klassen `FifoQueue` alltid sätta in elementet och returnera `true`.

I `FifoQueue` ska en enkellänkad lista användas för elementen i kön. Noderna i en lista representeras av följande privata nästlade klass (deklarerad i klassen `FifoQueue`):

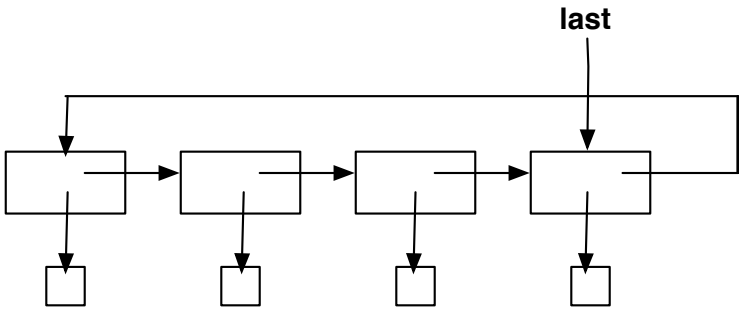
```
private static class QueueNode<E>  {
    E element;          // refererar till elementet
    QueueNode<E>  next; // refererar till efterföljande nod

    /* 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.

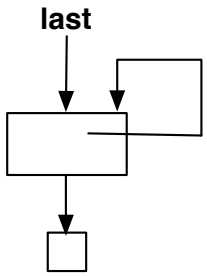
Observera att det bara är implementeringen av listan som är cirkulär. Utifrån sett är det en vanlig lista med början och slut.

D1. I projektet `lab2` finns det i paketet `queue` en fil med namnet `FifoQueue.java`. I filen 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. I filen `TestFifoQueue` i paketet `testqueue` finns det testmetoder 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`.




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.

D2. 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`.

Lägg också i filen `TestFifoQueue` till test som kontrollerar att iteratorn fungerar. Tänk på att även testa din iterator med de olika specialfall som finns. T. ex. ska ett anrop av `hasNext` returnera `false` om kön är tom, medan ett anrop av `next` i detta fall ska generera `NoSuchElementException`.

D3. 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
 * @param q the queue to append
 */
public void append(FifoQueue<E> q);
```

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.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.

Glöm inte att kontrollera att den andra kön är tom efter sammanslagningen.

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

D4. (Frivillig uppgift). Det finns en sorteringsmetod, positionssortering (eng. Radix sort) som sorterar en mängd icke-negativa heltal med ett visst känt maximalt antal siffror mycket snabbt genom att använda köer. (Egentligen är metoden inte begränsad till att sortera tal, den klarar även följder av annat slag t ex strängar.) Algoritmen använder sig (för sortering av heltal) av 10 köer, en för varje siffra 0..9, samt en kö som från början innehåller de osorterade talen och som i slutet av algoritmen innehåller talen sorterade i växande ordning. Den arbetar enligt följande, där vi använder `FifoQueue`-klassen för att representera köerna:

```
Placera talen i en kö numberQ av typ FifoQueue<Integer>
for (int i = 1; i <= d; i++) {  // d är antalet siffror i det längsta talet
    så länge numberQ inte är tom
        tag ut det första talet ur numberQ;
        j = i:e siffran bakifrån i talet;
        placera talet i kön subQ[j];
    for (int j = 0; j < 10; j++)    // konkatenera (den nu tomma) numberQ
        numberQ.append(subQ[j]);    // med de 10 köerna i subQ
}
```

I algoritmen distribuerar man först talen på de tio köerna med avseende på den sista siffran, sedan med avseende på den näst sista, osv. Efter varje sådan distribution konkateneras de tio köerna. Försök övertyga dig om att algoritmen är korrekt genom att studera exemplet som finns på sista sidan i denna laboration. Implementera metoden `radixSort` i filen `RadixSort.java` i paketet `sort`. Metoden har följande rubrik:

```
/** Sorterar talen i vektorn a med positionssortering. d anger maximalt
    antal siffror i talen. */
    public static void radixSort(int[] a, int d);
```

Tips: Siffror ur ett heltal kan extraheras med hjälp av divisionsoperatorn / och modulooperatorn %. När ett heltal $a$ divideras med ett annat heltal $b$ med operatorn /sker heltalsdivision vilket innebär att decimalerna stryks efter divisionen. Om a är ett heltal blir $a/10$ därför det tal som består av alla siffror i $a$ utom den sista. Operatorn % står för rest vid heltalsdivision. Sista siffran i ett tal $a$ får man därför ur $a\%10$. Näst sista siffran i $a$ är den sista i talet $a/10$ dvs vi får den genom att bilda $a/10\%10$. Allmänt gäller att i-te siffran från slutet kan erhållas ur $a/10^{i-1}\%10$.

Filen `TestRadixSort.java` i paketet `testsort` innehåller några test för sorteringsmetoden. Kör testprogrammet. Korrigera tills alla test går igenom. Lägg eventuellt till egna test.

**Anm**: I filen `RadixSort.java` finns följande programsatser för att skapa de tio köerna i subQ:

```
FifoQueue<Integer>[] subQ = (FifoQueue<Integer>[]) new FifoQueue[10];
for (int i = 0; i < 10; i++) {
    subQ[i] = new FifoQueue<Integer>();
}
```

Den första satsen skapar en vektor av typ `FifoQueue<Integer>[]`. I `for`-satsen initieras de tio elementen i vektorn som tomma köer av typen `FifoQueue<Integer>`. Man förväntar sig kanske att den första programsatsen skulle kunna skrivas enklare:

```
FifoQueue<Integer>[] subQ = new FifoQueue<Integer>[10]; //fel!
```

Det är emellertid inte tillåtet att i Java skapa vektorer vars element är av en parametriserad typ. För att kunna skapa vår vektor måste vi därför först skapa en vektor `FifoQueue[]` (utan att ange typen av element) och sedan göra typkonvertering till den typ vi önskar.

### Exempel på användning av RadixSort. (Bilaga till sista uppgiften på Lab 2.)

Nedan ges ett exempel på hur en följd av heltal sorteras med metoden RadixSort. Talen som sorteras är 721, 10, 51, 122, 674, 96, 109, 44, 236, 178, 1, 567, 674. Vi börjar med att distribuera talen med avsende på den sista siffran på 10 köer numrerade 0,1,..9 Resultatet av detta blir att köerna får följande innehåll:

```
0:      10
1:      721 51 1
2:      122
3:
4:      674 44 674
5:
6:      96 236
7:      567
8:      178
9:      109
```

Därefter konkateneras dessa köer (i ordningen 0, 1,.. 9) och vi får den resulterande kön: 10 721 51 1 122 67 44 674 96 236 567 178 109

Det vi nu åstadkommit är att talen är sorterade med avseende på sin sista siffra. Talen i denna kö distribueras nu med avseende på den andra siffran från slutet på 10 köer på samma sätt. Observera att ensiffriga tal då hamnar i kö nummer 0. Den andra siffran från slutet i dessa motsvarar alltså en inledande nolla i talet, vilket också ges av den formel för att räkna fram i:e siffran från slutet som anges i uppgiften:

```
0:      1 109
1:      10
2:      721 122
3:      236
4:      44
5:      51
6:      567
7:      674 674 178
8:
9:      96
```

Dessa köer konkateneras nu med resultatet: 1 109 10 721 122 236 44 51 567 674 674 178 96

Lägg märke till att om vi enbart betraktar de tal som består av de två sista siffrorna så utgör dessa nu en sorterad följd. Eftersom det maximala antalet siffror i talen är tre behövs ett sista tredje pass i vilket talen igen distribueras, nu med avseende på den tredje siffran från slutet. (Nu hamnar alla tal som har en eller två siffror i kö nummer 0):

```
0:      1 10 44 51 96
1:      109 122 178
2:      236
3:
4:
5:      567
6:      674 674
7:      721
8:
9:
```

Efter konkatenering får vi nu det sorterade slutresultatet: 1 10 44 51 96 109 122 178 236 567 674 674 721

# Laboration 3 – rekursion

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

## Läsanvisningar

Läs avsnitt 5.1–5.5 i läroboken (7.1–7.5 i gamla upplagan). Läs också föreläsningsbilder från föreläsningarna 6.

## Förberedelser

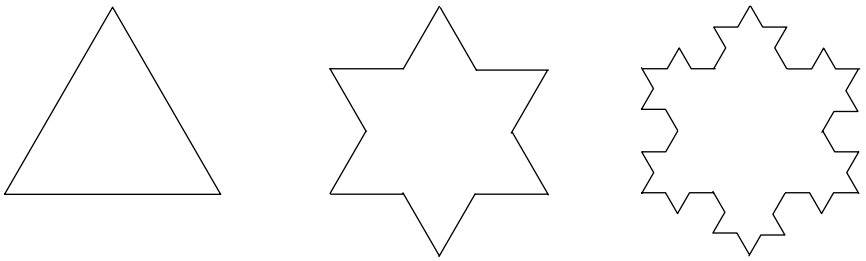
Läs igenom den inledande texten under rubrikerna ”Datorarbete”. Lös uppgift D1, D2, D3 och D4. Läs igenom de övriga uppgifterna.

## 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 2 som visar fraktalen Kochs snöflinga (uppkallad efter den svenska matematikern Helge von Koch).



Figur 2: 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 3. För att få en figur av ordning 2 ersätts varje linje i figuren av ordning 1 med fyra nya linjer osv.

Figur 3: 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 och det finns bara en knapp för att öka en fraktals ordning. Under laborationen kommer gränssnittet att behöva kompletteras så att man kan välja ytterligare en fraktaltyp ur menyn och så att det även finns en knapp för att minska vald fraktals ordning.

D1. I projektet `lab3` 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:

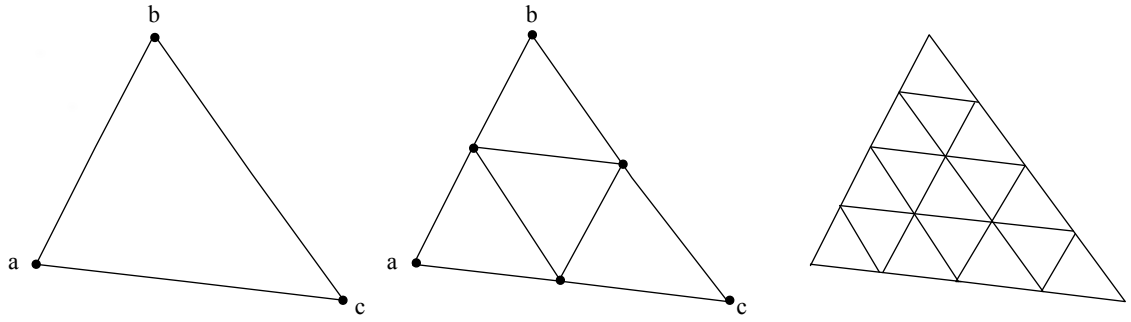
- Fönstret har en meny med namnet Fraktaler och texten ”Kochs triangel ordning 0” syns på fönstret.
- Om du öppnar menyn så syns det ett val: Kochs triangel. Om du väljer detta alternativ ur menyn så händer det ingenting. Det beror på att programmet vid detta val försöker rita Kochs fraktal av ordning 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 häftet 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. 4.




Figur 4: Bergfraktal av ordning 0, 1 och 2.

I projekten `lab3` finns ett paket `mountain`. Där 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 main-metoden i klassen `FractalApplication`. Öka vektorn `fractals` storlek och lägg in ett objekt av din nya fraktalklass i den. När du provkör huvudprogrammet 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.

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. 5.
Förskjutningens storlek bestäms av funktionen `randFunc` som ger ett slumptal enligt en viss fördelning med avvikelsen `dev`:

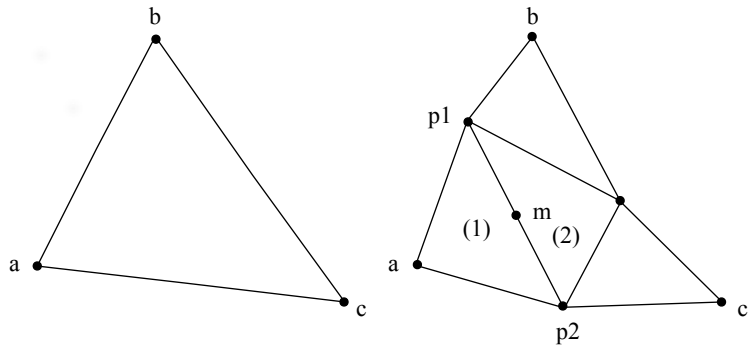
```
public static double randFunc(double dev) {
   double t = dev * Math.sqrt(-2 * Math.log(Math.random()));
   if (Math.random() < 0.5) {
      t = -t;
   }
   return t;
}
```

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 5: 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. 5 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 och mittpunkt.

Skapa i klassen `Mountain` en lista där beräknade mittpunkterna (`Side`-objekt kan lagras. När m beräknas första gången som mittpunkt på sidan p1–p2, lagras ett `Side`-objekt innehållande p1, p2 och m tillsammans i listan. Före varje mittpunktsberäkning kontrollerar man först i listan om mittpunkten redan har beräknats, om så är fallet hämtar man den direkt från listan. För att sökningen i listan ej ska bli alltför långsam är det lämpligt att ta ut elementet ur listan när man använt den redan beräknade mittpunkten.

Som lista kan t.ex. klassen `ArrayList` från Java Collections Framework användas.

# Laboration 4 – binära sökträd

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

## Läsanvisningar

Relevanta avsnitt i läroboken är: 5.1–5.5 och 6.1–6.4 (7.1–7.5 och 8.1–8.4 i gamla upplagan). Dessutom föreläsningsbilder om dessa avsnitt.

## Förberedelser

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

## Datorarbete

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

I projektet `lab4` finns i paketet `bst` 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* som i boken:

```
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 bokens klassrubrik. 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.

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.

D3. Implemetera i klassen `BinarySearchTree` metoden

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

som skriver ut nodernas innehåll i inorder.

D4. Testa metoderna `height`, `add`, `size` och `printTree` genom att implementera en `main`-metod i klassen. Glöm inte att testa att din `add`-metod fungerar som avsett om man försöker sätta in dubbletter.

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`).

Lägg till anrop av `drawTree` i din testkod. 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.

Du kan också testa din klass genom att använda `JUnit`. Uppritningen av trädet med hjälp av `BSTVisulizer` fungerar dock inte inifrån jUnit.

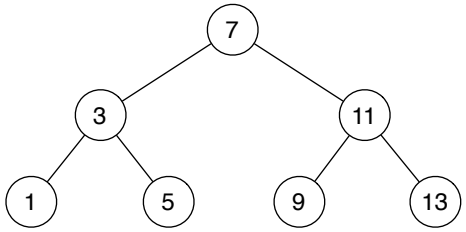
D5. Ett träd kan bli snett (obalanserat) när man gör många insättningar och borttagningar. Ett sätt att undvika detta är att balansera trädet i samband med varje insättning/borttagning enligt den metod som vi gått igenom på föreläsningarna (s.k. AVL-träd). Ett annat sätt kan vara att ”bygga om” trädet 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. 6.

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

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




Figur 6: Binärt sökträd med nycklar 1, 3, 5, 7, 9, 11, 13 byggt enligt algoritmen i texten.

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`:

```
/*
 * 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];
```

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 5 – hashtabell

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

## Läsanvisningar

Läs följande avsnitt i boken: 7.1–7.5 (9.1–9.5 i gamla upplagan). Läs också föreläsningsbilder från föreläsningarna om dessa avsnitt.

## 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”) som skiljer sig något från den i läroboken. Skillnaden består i att de länkade listorna inte representeras med `LinkedList` utan 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;
        ...

}
```

Det kan vara lämpligt att implementera en sak i taget och testa när så är möjligt. I paketet `test` finns en färdig testklass `TestSimpleHashMap`. Dessutom ska du skriva en metod som skriver ut hashtabellens innehåll. Den metoden ska också användas för att testa din klass.

D1. I kursens workspace finns det ett paket `map` i projektet `lab5`. 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 skriver 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 kommentartecknen 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!

# Laboration 6 – använda Javas klassbibliotek, grafiska användargränssnitt

*Mål:* Att ge träning i att använda klasser från klassbiblioteket Java Collections Framework. Att ge träning i att implementera ett enkelt grafiskt användargränssnitt i JavaFX.

## Läsanvisningar

Läs igenom följande avsnitt i boken: 6.1–6.4, 7.2, 7.5 (8.1–8.4, 9.2, 9.5 i gamla upplagan). För grafiska användargränssnitt rekommenderas följande alternativa källor: Oracles dokumentation `http://docs.oracle.com/javase/8/javase-clienttechnologies.htm` (specifikt avsnittet om *User Interface Components*). Klass-specifikationer hittar du här:
`https://docs.oracle.com/javase/8/javafx/api/index.html`.
Dessutom finns det föreläsningsbilder om dessa avsnitt på kursens hemsida.

## Förberedelser

Lös uppgift D1-D3. Läs igenom övriga uppgifter.

## Datorarbete

Under denna laboration ska en klass för hantering av en personlig telefonkatalog implementeras (uppgift D1). Användare ska kunna hantera telefonkatalogen via ett grafiskt användargränssnitt (uppgifterna D2, D4-D5). Telefonkatalogen ska också läsas in från fil då programmet startar och sparas på en fil då programmet avslutas (uppgift D3).

D1. I den här uppgiften ska en klass som representerar telefonkatalogen skrivas. Telefonkatalogen specificeras i interfacet `PhoneBook`:

```
public interface PhoneBook {
    /**
     * Associates the specified number with the specified
     * name in this phone book.
     * post: If the specified name is not present in this phone book,
     *        the specified name is added and associated  with
     *        the specified number. Otherwise the specified
     *        number is added to the set of number associated with name.
     * @param name The name for which a phone number is to be added
     * @param number The number associated with the specified name
     * @return true if the specified name and number was inserted
     */
    boolean put(String name, String number);


    /**
     * Removes the the specified name from this phone book.
     * post: If the specified name is present in this phone book,
     *       it is removed. Otherwise this phone book is
     *       unchanged.
     * @param name The name to be removed
     * @return true if the specified name was present
     */
    boolean remove(String name);
```

```
    /**
     * Removes the the specified number from name in this phone book.
     * post: If the specified name and number is present in this phone book,
     *       it is removed. Otherwise this phone book is
     *       unchanged.
     * @param number The number to be removed
     * @param name The name from which to remove the number
     * @return true if the specified name and number was present
     */
    boolean removeNumber(String name, String number);

    /**
     * Retrieves a set of phone numbers for the specified name. If the
     * specified name is not present in this phone book an empty set is
     * returned.
     * @param name The name whose associated phone numbers are to be returned
     * @return The phone numbers associated with the specified name
     */
    Set<String> findNumbers
    (String name);

    /**
     * Retrieves a set of names associated with the specified phone number.
     * If the specified number is not present in this phone book an empty
     * set is returned.
     * @param number The number for which the set of associated
     * names is to be returned.
     * @return The names associated with the specified number
     */
    Set<String> findNames(String number);

    /**
     * Retrieves the set of all names present in this phone book.
     * The set's iterator will return the names in ascending order
     * @return The set of all names present in this phone book
     */
    Set<String> names();

    /**
     * Returns true if this phone book is empty
     * @return true if this phone book is empty
     */
    default boolean isEmpty() {
        return size() == 0;
    }

    /**
     * Returns the number of names in this phone book
     * @return The number of names in this phone book
     */
    public int size();
}
```

I paketet `lab6/src/phonebook` finns interfacet `PhoneBook`. Lägg till en klass `MapPhoneBook` som implementerar interfacet. I en telefonkatalog associeras namn till ett eller flera telefonnummer. Samma namn kan inte förekomma mer än en gång. I implementeringen är det därför lämpligt att använda någon av de klasser i `java.util` som implementerar interfacet `Map`. Ett namn respektive ett telefonnummer kan representeras av en sträng. Ett antal telefonnummer associerade med ett namn kan representeras av en mängd av telefonnummer. Använd någon av de klasser som implementerar interfacet `Set` för detta.

I implementeringen av klassen kan du till en början utgå ifrån att vi alltid startar med en tom telefonbok. Konstruktorn ska alltså bara skapa ett tomt objekt av någon klass som implementerar interfacet `Map`.

Testa metoderna efterhand som du implementerar dem. En testklass finns i paketet `lab6/src/test`. Öppna klassen `TestPhoneBook` och ta bort kommentarstecknet på den rad där `MapPhoneBook`-objektet skapas i metoden `setUp`. (Den är bortkommenterad för att inte orsaka kompileringsfel innan klassen `MapPhoneBook` existerar.)

D2. Vi ska nu implementera ett grafiskt användargränssnitt. Via detta gränssnitt ska det vara möjligt för användare att hantera en telefonkatalog. Ett påbörjat användargränssnitt finns i paketet `lab6/src/application`. Öppna klassen `PhoneBookApplication` och ta bort kommentarstecknet på den rad där `MapPhoneBook`-objektet skapas samt på raden med motsvarande importsats. När du exekverar programmet öppnas ett fönster med menyer, två knappar samt en yta för att visa information. Exekvera programmet och prova att lägga till någon person i telefonkatalogen.

Studera klassen `PhoneBookApplication`. Under laborationen används JavaFX som ramverk för det grafiska gränssnittet. Alla JavaFX-applikationer utgår från en klass som ärver från `javafx.application.Application`. Via arvet fås bl.a. två metoder, `public void start(Stage primaryStage)` och `public void stop()`, som ramverket anropar när programmet startar respektive avslutas. Som parameter får metoden `start` ett `javafx.stage.Stage`-objekt. Det representerar programmets fönster. Normalt finns koden för att skapa ett *JavaFX scenträd*, d.v.s. objekten för programmets grafiska komponenter, t.ex. knappar, menyer och dialogrutor, i `start()`-metoden.

I den givna koden är det `PhoneBookApplication`, som ärver från Application. För att koden i `start`-metoden inte ska bli alltför lång är skapandet av menyerna flyttat till klassen `PhoneBookMenu` medan klassen `NameListView` hanterar knapparna samt en listvy där namnen i telefonkatalogen visas.

Studera också klasserna `NameListView` och `PhoneBookMenu` och se till att du i stora drag förstår koden.

Klassen `Dialog` innehåller färdiga statiska metoder för att visa några olika slags dialogrutor. Skumma igenom klassen och lägg märke till vilka olika slags dialogrutor som finns att använda. Resultatet från dialog-metoder är ofta ett objekt av typen `Optional`. På detta objekt anropar man metoden `isPresent` för att ta reda på om det finns något resultat (saknas. t.ex. om användaren tryckt på "Cancel"). För att få tag på själva resultatet anropas `get`. Ex:

```
Optional<String> result = Dialogs.oneInputDialog(...);
if (result.isPresent()) {
    String input = result.get();
    ...
}
```

D3. För praktisk användning krävs naturligtvis att telefonkatalogens innehåll sparas när programmet avslutas och att det vid programstart är möjligt att rekonstruera telefonkatalogen.

En möjlighet är att man, då programmet avslutas, ser till att innehållet i katalogen skrivs ut på en textfil. Man måste då lägga till en metod som skapar en fil, går igenom katalogen och skriver ut alla namn med tillhörande nummer. För att kunna rekonstruera innehållet krävs då också att man bestämmer sig för ett visst format på utskriften, t.ex. att man skriver ett namn och alla dess asscocierade nummer på en rad. Då kan man vid start

av programmet öppna denna fil, läsa innehållet rad för rad och på nytt sätta in alla namn med sina associerade nummer i katalogen.

Ett enklare sätt är att göra sina objekt serialiseringsbara genom att använda interfacet `Serializable`. Detta interface saknar metoder och kan karakteriseras som ett markörinterface. När man för en klass anger att den implementerar interfacet ger man tillåtelse att skriva ut objekt av klassen på en fil på ett mycket enkelt sätt. Det som då skrivs ut är en intern representation av objektet som lätt kan läsas in igen för rekonstruktion av objektet.

För att skriva ut ett objekt av en klass som implementerar interfacet `Serializable` kan man följa detta mönster:

```
File file = ...;
try {
    ObjectOutputStream out =
        new ObjectOutputStream(new FileOutputStream(file));
    out.writeObject(nameofObjectToBeWritten);
    out.close();
} catch (Exception e) {
    e.printStackTrace();
    System.exit(1);
}
```

Det är alltså metoden `writeObject` i klassen `ObjectOutputStream` som utför utskriften av objektet.

För att läsa in ett objekt från en fil gör man följande:

```
File file = ...;
try {
    ObjectInputStream in =
        new ObjectInputStream(new FileInputStream(file));
    objectName = (objectType) in.readObject();
    in.close();
} catch (Exception e) {
    e.printStackTrace();
    System.exit(1);
}
```

Det är alltså metoden `readObject` i klassen `ObjectInputStream` som utför inläsning av objektet. Denna metod returnerar ett objekt av typen `Object`. Eftersom vi vet vilken typ av objekt vi skrivit ut på filen kan vi göra en säker typomvandling. Om vi t. ex. vet att vi skrivit ut ett objekt av klassen `MyClass` så kommer alltså inläsningssatsen att bli:

```
myObject = (MyClass) in.readObject();
```

där `myObject` förutsätts vara en variabel av typ `myClass`.

Om man vill kunna skriva ut objekt av egna klasser på det här sättet måste man lägga till `implements Serializable` till klassrubriken. Genom att lägga till `implements Serializable` till klassrubriken i `MapPhoneBook` kan du spara `phoneBook`-objektet enligt ovan. För att Java ska kunna läsa in objektet igen måste klassen se likadan ut (samma typer på attributen och lika många attribut). För att säkerställa det har alla objekt som implementerar `Serializable` ett versionsnummer. Detta ändras normalt varje gång klassen kompileras om, vilket medför att du inte kan läsa in telefonkatalogen efter att programmet kompilerats om. För att behålla samma versionsnummer kan du lägga till attributet `private static final long serialVersionUID = 1L`. Om du ändrar typ, ordning eller lägger till attribut ska du ändra värdet på `serialVersionUID`.

Lägg till kod i klassen `PhoneBookApplication` som ger användaren möjlighet att läsa

telefonkatalogen från fil. (I annat fall ska en tom telefonkatalog skapas). Vid start av programmet ska användaren få frågan om katalogen ska läsas in från fil. Välj någon lämplig metod i klassen `Dialogs` för detta. Användaren ska sedan själv få välja i vilken fil som ska läsas. Använd klassen `FileChooser` för detta. I dokumentationen av `FileChooser` på Javas hemsida finns exempel på hur klassen används.

Lägg också till kod för att på motsvarande sätt skriva ut telefonkatalogen på fil då programmet avslutas.

Placera gärna den nya koden för inläsning respektive utskrift i var sin privat metod och anropa dem i `start`- respektive `stop`-metoden.

Testa!

D4. Lägg till två knappar, ”Remove” och ”Remove number” i klassen `NameListView`. När man klickar på knapparna ska följande hända:

- ”Remove” – Personen vars namn är valt i listVyn ska tas bort. Innan personen tas bort ska användaren tillfrågas om personen verkligen ska tas bort.
- ”Remove number” – Önskat telefonnummer ska tas bort från personen vars namn är valt i listvyn.

Hämta inspiration från koden för knapparna ”Add” och ”Add number”. För att bestämma vad som händer när användaren klickar på en knapp kopplas ett objekt av en klass som implementerar interfacet `EventHandler<ActionEvent>` till knappen. I tidigare Javaversioner var man tvungen att skriva en egen lyssnar-klass eller använda anonyma klasser: Exempel:

```
Button helloButton = new Button("Hello");
button.setOnAction(new EventHandler<ActionEvent>() {
    @Override
    public void handle(ActionEvent event) {
      System.out.print("Hello World !!");
  }
});
```

Fr.o.m. Java 8 kan det även göras med lambda-uttryck. I praktiken innebär det att du inte behöver skriva texten för att skapa den anonyma klassen utan bara innehållet i `handle()`-metoden:

```
Button helloButton = new Button("Hello");
button(event -> System.out.print("Hello World !!"));
```

I fallet med knapparna i klassen `NameListView` är det praktiskt att lägga det som ska göras i en privat metod och sedan anropa den i lambda-uttrycket:

```
Button addButton = new Button("Add");
addButton.setOnAction(e -> add());
```

Se till att den information som visas uppdateras korrekt efter det att man tryckt på en knapp. De nya knapparna ska precis som knappen ”Add number” vara inaktiva när inget namn är valt i listvyn.

Prova att de nya funktionerna fungerar. Se till att ditt program uppträder vettigt även om användaren klickar på ”Cancel” eller klickar på Enter utan att först ha skrivit in någonting i någon inmatningsruta.

D5. I `PhoneBookMenu` finns koden för att skapa de två menyerna. Menyn ”PhoneBook” är färdig och innehåller bara alternativet ”Quit”. I menyn ”Find” finns alternativet ”Show all” som ser till att alla namn i telefonkatalogen visas i listvyn. Lägg till följande tre alternativ i menyn ”Find”.

- ”Find number(s)” – En dialogruta ska visas där användaren kan mata in ett namn. Om namnet finns ska enbart det namnet visas i listvyn, namnet ska vara markerat och information om telefonnummer ska visas. Om namnet ej finns ska användaren meddelas detta på lämpligt sätt.
- ”Find name(s)” – En dialogruta ska visas där användaren kan mata in ett telefonnummer. Namnen till de personer som har detta telefonnummer ska visas i listvyn. Om numret ej finns ska användaren meddelas detta på lämpligt sätt.
- ”Find person(s)” – Ska fungera på samma sätt som ”Find name(s), men med den skillnaden att alla namn som startar med de tecken som matas in ska visas i listvyn.

Menyn byggs upp av klasserna `MenuBar, Menu och MenuItem` från paketet `javafx.scene.control`. För att bestämma vad som händer när användaren väljer ett menyalternativ gör man på samma sätt som i fallet med knappar. Ett objekt av en klass som implementerar interfacet `EventHandler<ActionEvent>` kopplas till varje val i menyn. Ex:

```
menuShowAll.setOnAction(e -> showAll());
```

Prova efterhand som du implementerar de nya funktionerna att programmet fungerar som det ska.