

Datorlaborationer, Programmeringsteknik – fördjupningskurs

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

- *Uppgifterna i laborationerna ska lösas 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 kommer att ges två gånger per år, en gång under lp VT1 och en gång under perioderna HT1+HT2.

- *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

1	Laboration 1	3
2	Laboration 2	8
3	Laboration 3	14
4	Laboration 4	18
5	Laboration 5	21
6	Laboration 6	24

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

Mål: Du ska befästa och utöka dina kunskaper om Java inom de områden som behandlats under de två första föreläsningarna och den första övningen: 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 E-husets datorer 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 E-husets datorer.

Det finns ett förberett 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/edaa01vt>. Klicka där på länken Laborationer i menyn till vänster. Ladda ner filen `edaa01-workspace.zip` till din hemkatalog och packa upp den genom att skriva följande kommando i ett terminalfönster:

```
cd
unzip edaa01-workspace.zip
```

Eclipse startas med kommandot

```
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 projekt vyn ("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 `SimpleSet<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 metoderna i klassen 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 testat 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. Lägg till följande metod 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, tex 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`.

- D3. Lägg till lämpliga test för metoden `addAll` i `TestArraySet.java`. Testa. Korrigera eventuellt och testa på nytt tills alla test går igenom.
- D4. 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 implementera 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 tex i implementeringen av `add` i `MaxSet` vill anropa metoden `add` i superklassen så gör du detta genom `super.add(...)`.

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.

- D5. 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 dubletter. 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 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 antingen genom att i samma klass skriva en `main`-metod eller genom att använda `JUnit`.

Laboration 2

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 föreläsningbilder 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 ärvä `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();

```

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.

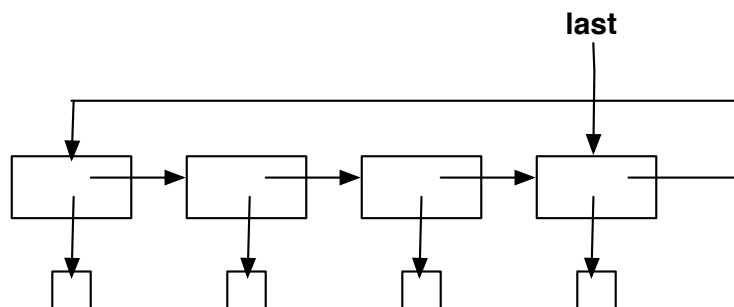
I FifoQueue ska en enkellänkad lista användas för elementen i kön. Noderna i en lista representeras av följande privata nästlade klass (deklarerad i klassen FifoQueue):

```

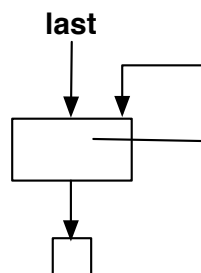
private static class QueueNode<E> {
    E element;           // refererar till elementet
    QueueNode<E> next;   // refererar till efterföljande nod

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

```



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.

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

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 som kontrollerar funktionaliteten hos de operationer som implementeras i denna uppgift. 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*.

- 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;
        ...
        /* Konstruktör */
        private QueueIterator() {...}

        public boolean hasNext() {...}

        public E next() {...}

        public void remove() {
            throw new UnsupportedOperationException();
        }
    }
}
```

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 konstruktör 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 q_1 och q_2 till en kö bestående av alla element i q_1 följda av alla element i q_2 . Om man bara har tillgång till de befintliga metoderna på listan kan man successivt ta ut elementen ur q_2 med metoden `poll` och sätta in dem i q_1 med metoden `offer`. Om det finns n element i q_2 anropas alltså båda metoderna n gånger.

Du ska göra en effektivare lösning genom att i stället utföra konkateneringen i en metod i klassen `FifoQueue`. Då kan du 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 korriger 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 konkatenerar man talen i den enda kö som fortfarande innehåller tal.

neras 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 divisionsoperatoren `/` och modulooperatoren `%`. När ett heltal a divideras med ett annat heltal b med operatoren `/` 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. Operatoren `%` 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 avseende 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

Mål: Att ge träning i att skriva program med rekursiva algoritmer. Att ge viss träning i att implementera ett enkelt grafiskt användargränssnitt.

Läsanvisningar

Läs avsnitt 5.1–5.5 i läroboken (7.1–7.5 i gamla upplagan). För grafiska användargränssnitt rekommenderas följande alternativa källor: Kapitel 16 i kursboken (av Per Holm) på grundkursen för C, D, L och M (upplaga 3:1, 2007) och/eller delar av Appendix C i läroboken av Koffman/Wolfgang. Dessutom finns det föreläsningbilder om dessa avsnitt på kursens hemsida.

Förberedelser

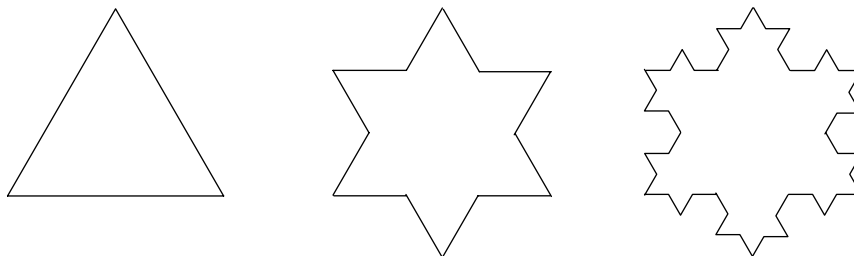
Läs igenom den inledande texten under rubrikerna "Datorarbete". Lös uppgift D1, D2, D3 och D4. Läs igenom de övriga uppgifterna.

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 åstadkommit 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) ritas 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, paneler och knappar. Bekanta dig med klasserna i paketet.

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 en knapp med texten ">" för att öka 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`.

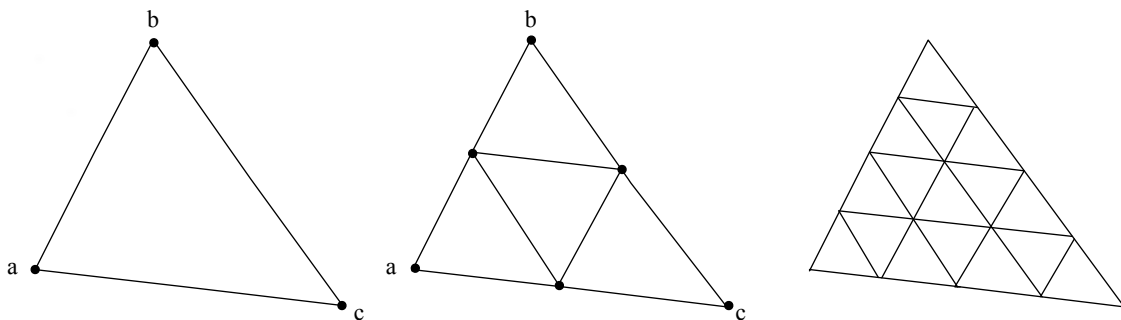
Observera att koden för att rita Kochs snöflinga här i häftet är pseudokod och att du i den riktiga koden även behöver ha med ett objekt av klassen `TurtleGraphics` som parameter för att rita linjer.

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. När man kör programmet `FractalApplication` kan man öka ordningen för den fraktal som ritas genom att klicka på knappen längs ned till vänster i fönstret. Lägg till en motsvarande knapp för att minska fraktalens ordning.

Tips! Studera klasserna `CommandPanel` och `OrderUpButton` i paketet `fractalView`.

- D4. 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.

För att din nya fraktal ska synas i användargränssnittets 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.

- D5. 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`:


```

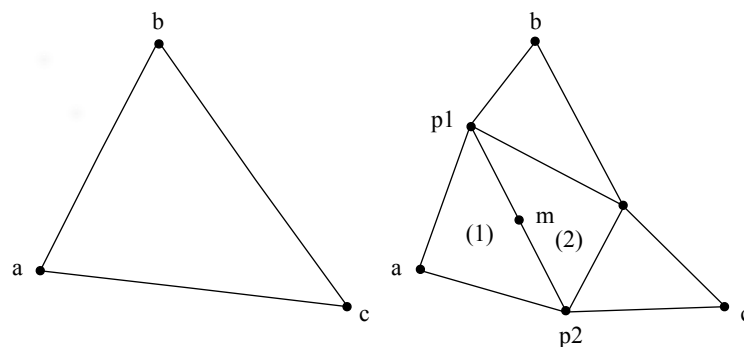
public static double randFunc(double dev) {
    double t = dev * Math.sqrt(-2 * Math.log(Math.random()));
    if (Math.random() < 0.5) {
        t = -t;
    }
    return t;
}

```

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 konstruktör.

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

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



Figur 5: Bergfraktal av ordning 0 och 1. Mittpunkterna förskjuts - y-led innan en triangel delas upp i fyra nya trianglar.

- D6. Kanske blir figuren från förra uppgiften inte helt perfekt. Av den högra figuren i fig. 5 ser vi att en speciell svårighet uppstår genom att triangelarna 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 triangelns 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 LinkedList från Java Collections Framework användas.

Laboration 4

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. Implementera 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 `BSTVisualizer` finns metoden `void drawTree(BinarySearchTree<?> bst)` som ritar ett binärt träd i ett fönster. (Inuti klassen `BSTVisualizer` 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 `BSTVisualizer` fungerar dock inte inifrån `JUnit`.

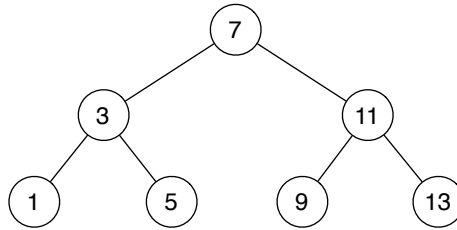
- D5. Ett träd kan bli snett (obalanserat) när man gör många insättningar och borttagningar. Ett sätt att undvika detta är att balansera trädet i samband med varje insättning/borttagning enligt den metod som vi gått igenom på föreläsningarna (s.k. AVL-träd). Ett annat sätt kan vara att "bygga om" trädet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

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 föreläsningbilder 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 `HashMap` 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` där detta görs.

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 metoderna `get` och `put` som ej är implementerade ännu. Kör testen ändå och bortse från de fel som avser ännu inte implementerade metoder. Eller vänta med att köra `JUnit`-testerna tills du implementerat `get`.

- D8. 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.

- D9. Implementera `get(Object object)`. Argumentet måste omvandlas till typen `K`. Om nyckeln inte finns returneras `null`.

Testa!

- D10. 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

Mål: Att ge träning i att använda lämpliga klasser från Javas klassbibliotek för att implementera en telefonkatalog. Att ge träning i att implementera ett enkelt grafiskt användargränssnitt, som ger användare möjlighet att interaktivt utföra operationer på telefonkatalogen.

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: Kapitel 16 i kursboken (av Per Holm) på grundkursen för C, D, L och M (upplaga 3:1, 2007) och/eller delar av Appendix C i läroboken av Koffman/Wolfgang. Dessutom finns det föreläsningsbilder om dessa avsnitt på kursens hemsida.

Förberedelser

Läs igenom texten under rubriken "Datorarbete". Lös uppgiften D1. Läs texten efter uppgift D1 och studera den påbörjade implementeringen av PhoneBookGUI som finns i projektkatalogen lab6. Lös uppgift D2. Läs igenom uppgift D3.

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 som också ska implementeras (uppgifterna D2 och D3). Det finns också två uppgifter där man sparar telefonkatalogen på en fil då programmet avslutas respektive läser in katalogen från fil då det startar.

Klassen som representerar en telefonkatalog har följande specifikation:

```
public class PhoneBook {

    public PhoneBook();

    /**
     * Associates the specified name with the specified
     * number 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 new number is added
     *       to the set of numbers associated with name.
     * @param name The name for which a number is to be added
     * @param number The number to be associated with the specified name
     */
    public void 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.
     */
    public boolean remove(String name);

    /**
     * Retrieves a list of phone numbers for the specified name.
```



```

    * If the specified name is not present in this phone book an empty
    * list is returned.
    * @param name The name whose associated phone numbers are to be returned
    * @return The list of phone numbers associated with the specified name
    */
    public List<String> findNumber(String name);

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

    /**
     * Returns true if this phone book is empty
     * @return true if this phone book is empty
     */
    public boolean isEmpty();

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

```

- D1. I denna uppgift ska klassen PhoneBook implementeras. 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 repektive ett telefonnummer kan representeras av en sträng. Ett antal telefonnummer associerade med ett namn kan representeras av en lista av telefonnummer (strängar). Avgör själv, utgående från specifikationen av PhoneBook vilken klass som kan vara lämplig att använda för att representera telefonkatalogen.

I filen `lab6` finns det en fil `PhoneBook.java`. Den innehåller metodrubriker enligt specifikationen ovan. Implementera klassen. Testa metoderna efterhand som du implementerar dem.

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

Vi ska nu implementera ett grafiskt användargränssnitt. Via detta gränssnitt ska det vara möjligt för användare att hantera en telefonkatalog.

I filen `PhoneBookGUI.java` finns ett sådant gränssnitt påbörjat. Vidare finns det i filen `Main.java` en `main`-metod som skapar ett grafiskt gränssnitt och knyter detta till en instans av klassen `PhoneBook`. Om du exekverar denna öppnas på skärmen ett fönster med menyer, en knapp samt

en textarea. Det händer dock tills vidare ingenting när man väljer menyalternativ eller klickar på knappen.

Öppna filen `PhoneBookGUI.java`. I denna klass (som ärver `JFrame`) finns två attribut, `phoneBook` som refererar till den telefonkatalog som hanteras i programmet och ett som refererar till en textarea på fönstret där resultat av operationer på katalogen eller meddelande till användaren ska skrivas ut.

I konstruktorn får attributet `phoneBook` först sitt värde. Därefter utförs följande sats:

```
setDefaultCloseOperation(WindowConstants.DO_NOTHING_ON_CLOSE);
```

Om man försöker stänga fönstret kommer p.g.a. denna sats ingenting att hända. Tanken är att man i stället ska använda `Quit`-knappen och att man då i programmet kan se till att spara innehållet i katalogen på en fil innan fönstret stängs och programmet avslutas. Tills vidare kan du avsluta programmet genom klicka på den röda knappen (kvadraten) till höger om `Console`-fönstrets flik. Då terminerar körande program.

Följande två rader i filen är:

```
Locale.setDefault(new Locale("en"));  
/* To avoid hardcoded Swedish text on OptionPane dialogs */  
UIManager.put("OptionPane.cancelButtonText", "Cancel");
```

Klassen `Locale` i `java.util` representerar regioner eller språkområden. Vissa operationer i klassbiblioteket använder denna klass för att anpassa utseendet på olika komponenter för olika språk. Man kan påverka vilket språk som används med operationen `setDefault`. I detta fall sätts default om till att vara engelska. Det betyder bl.a. att rubriken på ett dialogfönster kommer att bli "input". Om vi inte ändrat default skulle vi få en svensk rubrik ("indata") i stället. För vissa dialogfönster (som vi kommer att använda och säga mer om senare i denna text) är tyvärr texten på vissa knappar hårdkodad och kan inte påverkas av `Locale`. Den andra satsen ovan ser till att `Avbryt`-knappar på sådana fönster också får engelsk text (`Cancel`). Om man vill ha svensk text så kan man ta bort båda satserna.

Längre fram i konstruktorn skapas en menyrad (`JMenuBar`) som knyts till fönstret. Alla menyer som hör ihop med ett fönster måste placeras på en sådan menyrad. En meny (av typ `JMenu`) med namnet "Edit" skapas och adderas till menyraden. De menyalternativ som ska finnas på denna meny skapas sedan och adderas till `Edit`-menyn. De två alternativen namnges "Add" respektive "Remove" och ska användas när en användare vill lägga till eller ta bort något ur katalogen. Menyalternativen är subclasser till klassen `JMenuItem`. Båda subclasserna (`AddMenu` och `RemoveMenu`) implementerar interfacet `ActionListener`. Därför innehåller de en metod `actionPerformed`. Denna metod ska (i en uppgift nedan) implementeras så att användarens val av respektive menyalternativ hanteras.

Du kan, om du vill, läsa om menyer i läroboken Appendix C, avsnitt 6.

Senare i konstruktorn skapas ytterligare en panel, `southPanel`, och på denna placeras en textarea med en "scrollbar". Avsikten är att resultat av operationer ska visas här. Se vidare i uppgift D3.

- D2. För att kunna hantera även andra operationer på katalogen behövs fler menyer. Lägg till en meny med namnet "Find" på menyraden. Lägg till denna meny menyalternativen "Find number(s)" och "Find name(s)". Skapa även en meny med namnet "View", som har ett menyalternativ "Show all". Skapa samtidigt nya klasser för de nya menyalternativen enligt samma mönster som för klasserna `AddMenu` och `RemoveMenu`. I de klasser som representerar de nya menyalternativen ska också i konstruktorerna läggas till satser som knyter lyssnare till respektive alternativ (enligt samma mönster som i klasserna för de givna menyalternativen).

Testa genom att exekvera Main. Du bör nu se alla menyalternativ på fönstret. Fortfarande händer förstås ingenting när man väljer något menyalternativ eftersom `actionPerformed` ännu inte är implementerad.

- D3. Till varje menyalternativ och till Quit-knappen finns det nu kopplat en instans av en klass som implementerar `ActionHandler`. Om man väljer ett menyalternativ eller klickar på Quit-knappen kommer metoden `actionPerformed` att exekveras. I denna metod ska de åtgärder utföras som föranleds av användarens val.

I denna uppgift ska metoden `actionPerformed` implementeras i samtliga subklasser till `JMenuItem` och i klassen `QuitButton`. I klassen `QuitButton` kan metoden till en början implementeras så att programmet avslutas. I en senare uppgift kommer den sedan att ändras så att innehållet sparas på en fil innan programmet avslutas.

Resultatet av operationer på telefonkatalogen ska visas i textarean på fönstret. För sökoperationer visas resultat av sökning eller ett meddelande om att sökningen misslyckades. För insättningsoperationen och borttagningsoperationen ska ett meddelande visas som talar om huruvida operationen kunde utföras eller ej.

Testa din implementering genom att sätta in och ta bort element via de olika menyalternativen. Resultatet av dessa operationer kan kontrolleras genom att välja andra menyalternativ.

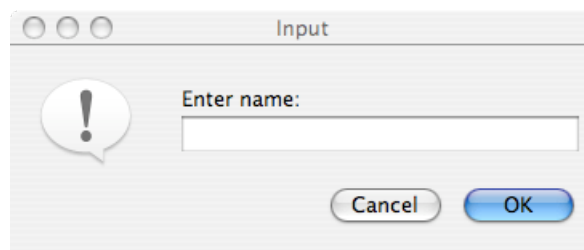
Tips: Operationerna kräver dialog med användaren. Om man väljer menyalternativet "Find number(s)" måste man ta reda på namnet på den person vars nummer man söker. I `javax.swing` finns det en klass, `JDialog`, som kan användas för sådana dialoger. Använder man denna klass måste man själv beskriva vilka komponenter som ska finnas på dialogfönstret och vilken layout som ska användas etc. För enklare dialoger, som i vårt fall, kan man med fördel använda färdiga dialogfönster. Det finns en mängd sådana som kan skapas genom att anropa statiska operationer i klassen `JOptionPane`. Tex finns följande metod

```
/**
 * Shows a question-message dialog requesting input from the user
 * @param message - the Object to display
 */
public static String showInputDialog(Object message);
```

Genom att göra följande anrop:

```
String s = JOptionPane.showInputDialog("Enter name");
```

visas följande dialogfönster:



Användaren kan nu skriva in en text. Om man klickar på knappen "OK" stängs fönstret och den sträng som skrivits in returneras. I exemplet ovan kommer då strängen `s` att referera till den text som skrivits in. Man kan också stänga fönstret genom att trycka på Enter efter det att man skrivit in önskad text eller genom att klicka på knappen längst till

vänster i fönstrets övre ram. Om man väljer att klicka på knappen "Cancel" stängs också fönstret, men då returneras null.

- D4. 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 associerade 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` gör man följande:

```
try {
    ObjectOutputStream out =
        new ObjectOutputStream(new FileOutputStream(fileName));
    out.writeObject(nameofObjectToBeWritten);
} 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:

```
try {
    ObjectInputStream in =
        new ObjectInputStream(new FileInputStream(fileName));
    objectName = (objectType) in.readObject();
} 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`.

Lägg till en metod som läser in hela `Map`-objektet från en fil och en metod som skriver ut `Map`-objektet på en fil.

De klasser i Javas klassbibliotek som implementerar interfacet `Map` implementerar ock-

så interfacet `Serializable`. Om man vill kunna skriva ut objekt av egna klasser på det här sättet måste man lägga till `implements Serializable` till klassrubriken.

- D5. Ändra nu implementeringen av lyssnaren för Quitknappen så att telefonkatalogen sparas enligt ovan. Låt lyssnaren fråga användaren efter filnamn. Använd t. ex. samma teknik som i uppgift D3 (klassen `JOptionPane`). Om du vill kan du istället använda klassen `JFileChooser`. Kanske vill du fråga användaren om katalogen ska sparas på fil eller inte. Använd då `JOptionPane.showConfirmDialog`.

Testa nu Quit. Det är inte säkert att du ser den fil där du sparat objektet direkt i Eclipse. Om du markerar projektet med högerknappen och väljer Refresh från menyn dyker den upp (om du lagt den i projektet).


- D6. Nu måste vi också se till att katalogen läses in vid start av programmet. Ändra i programmet så att telefonkatalogen hämtas (enligt ovan) från en fil som användaren får ange.

Testa att du nu kan köra programmet och att det vid start finns samma namn och nummer som då du senast avslutade det.

Ett alternativ till att fråga användaren efter filnamnet med hjälp av klassen `JOptionPane` eller klassen `JFileChooser` är att låta filnamnet vara argument till programmet. Filnamnet finns då i `args[0]`.

Filnamnet varifrån katalogen läses ska i så fall anges som ett argument till Main-klassen då man kör programmet. I Eclipse kör man ett program med argument på följande sätt:

- Markera klassen Main med högerknappen och välj Run as → Run configurations.
- I det fönster som öppnas kan vi ge ett namn åt den konfiguration vi ska skapa (vi kan ju vilja köra med olika argument). Fyll t.ex. i Main-filnamn i fältet Name (där filnamn ersätts med det namn du gett filen där katalogen skrivits ut). Klicka sedan på fliken Arguments. Fyll i filnamnet i fältet Program arguments. Klicka på knappen Run. Nu körs programmet med filen som argument.

När du skapat en konfiguration på detta sätt kan du sedan lätt köra programmet med samma argument igen. I översta raden med symboler på Eclipsefönstret finns det en grön symbol med en vit pil: . Genom att trycka på denna startas senast körda program. Andra programkonfigurationer får man fram genom att trycka på den lilla svarta pilen till höger om symbolen. Den visar en meny med de senast körda konfigurationerna. Här bör nu din nyskapade konfiguration finnas med.