Lösningsförslag: Övning 6

Uppgift 1

a)

i)

```
//return false if reading fails
//return true otherwise
public boolean readFile() {
   //code
}


public boolean method2() {
   //code block C
   boolean success = method3();
   if (!success)
       return false;
   else {
       //code block  D
       return true;
   }
}
```

```
public boolean method3() {
   //code block E
   boolean success = readFile();
   if (!success)
       return false;
   else {
       //code block  F
       return true;
   }
}
public void method1() {
   //code block A;
   boolean success = method2();
   if (!success)
       //code for error processing
   else
       //code block  B;
}
```

Kommentarer:

Poängen ligger i att observera hur felet som uppstår i readFile() och ska hanteras i metod1(), kräver att felhantering även införs i metod2() och metod3() för att felet ska kunna propageras uppåt.

i)

```
//throws  ReadFileFailedException if reading fails
public void readFile() throw ReadFileFailedException {
   //code
}


public void method1() {
   //code block A;
   try {
       method2();
   } catch( ReadFileFailedException e) {
       //code for error processing
   }
   //code block  B;
}
```

```
public void method3() throws ReadFileFailedException{
       //code block E
       readFile();
       //code block  F
}


public void method2() throws ReadFileFailedException {
       //code block C
       method3();
       //code block D
}
```

Kommentarer:

Felhanteringen som tidigare gjordes "manuellt" med **if**-satser kan nu hanteras med exceptions. Det gör att felhanteringen dels blir tydligare (**if**-satser kan ju handla om annat än fel) och dels kan tas bort från kod som inte har med felet att göra. Koden i både method2() och method3() blir nu mycket "renare". Metoderna uppmärksammar att något kan gå fel under körningen genom sina **throws**-deklarationer, men behöver inte hantera felet själva.

Använd endast exceptions för sådant som är just "undantag" från kodens huvudsakliga uppgift, d.v.s. (i första hand) olika typer av fel. Om det inte finns någon exekveringsväg i ett program som är helt fri från exceptions så är det inte frågan om undantag.

Låt aldrig ett **catch**-block stå tomt! Då fångas ett fel utan att det åtgärdas!

Använd lämplig typ av exceptions beroende på kodens abstraktionsnivå. Låt t.ex. inte ett IOException kunna fångas i kod där programmeraren inte vet om att någon input/output inträffar. I ett sådant fall bör detta IOException fångas i den kod som handhar input/output och en mer lämplig typ av exception kastas vidare.

Fånga aldrig ett Exception (alltså basklassen för alla exceptions). Det har en benägenhet att dölja fel. Fånga bara de exceptions du kan förvänta dig.

b)

i) Det fångas eftersom ett NotAPositiveNumberException också är ett NotANumberException (genom arvsrelationen).

ii) Koden kompilerar inte eftersom **catch**(NotAPositiveNumberException napne) inte kan nås ("unreachable code"). Detta på grund av samma anledning som i uppgift i), nämligen att NotAPositiveNumberException fångas i **catch**(NotANumberException nane).

Uppgift 2

a)

```
import java.util.HashMap;
import java.util.Map;
import java.util.Set;
public class FrequencyTable1 {
   public static void main(String[] args) {
      Map<String,IncrementableInteger> lookupTable = new HashMap<String,IncrementableInteger>();
      for (String str : args) {
         IncrementableInteger freq = lookupTable.get(str);
         if (freq == null)
            lookupTable.put(str, new IncrementableInteger(1));
         else
            freq.increment();
      }
      Set<Map.Entry<String,IncrementableInteger>> set = lookupTable.entrySet();
      System.out.println("\nFrequency table:");
      for (Map.Entry<String,IncrementableInteger> e : set)
         System.out.println(e.getKey() + ": " + e.getValue());
   }
}
```

b)

```
import java.util.HashMap;
import java.util.Map;
import java.util.Set;
public class FrequencyTable2 {
   public static void main(String[] args) {
      Map<String,Integer> lookupTable = new HashMap<String,Integer>();
      for (String str : args) {
         Integer freq = lookupTable.get(str);
         if (freq == null)
            lookupTable.put(str, 1);
         else
            lookupTable.put(str, ++freq);
      }
      Set<Map.Entry<String,Integer>> set = lookupTable.entrySet();
      System.out.println("\nFrequency table:");
      for (Map.Entry<String,Integer> e : set)
         System.out.println(e.getKey() + ": " + e.getValue());
   }
}
```

Kommentarer:

En Integer är icke-muterbar (precis som String, Float, Boolean etc.). Ickemuterbarhet har många fördelar, men en nackdel är att ett helt nytt objekt måste skapas när ett nytt värde ska lagras, något som kan dra ned programmets prestanda. Ett frekvensvärde i uppgift b) lagras som en Integer och när ett sådant värde ökas, är det alltså istället ett nytt objekt med det högre värdet som skapas.

Uppgift 3

```
import java.util.Set;
import java.util.TreeSet;
public class ExclusiveUnion {
   public static <T> Set<T> exclusiveUnion(Set<T> s1, Set<T> s2) {
       TreeSet<T> t1 = new TreeSet<T>(s1);
       TreeSet<T> t2 = new TreeSet<T>(s2);
       t1.removeAll(s2);
       t2.removeAll(s1);
       t1.addAll(t2);
       return t1;
   }
}
```

Uppgift 4

Övningen vill visa på vikten av valet av datastruktur, trots att de ofta erbjuder samma funktionalitet genom sina gränssnitt.

En länkad lista (x) är bäst för tillägg av innehåll (add) medan den är betydligt sämre än de två andra datastrukturerna på att söka upp ett lagrat element (contains). Detta beror på att allting i en länkad lista läggs "på rad", med bara direkt tillgång till första (och ev. sista) elementet i denna rad. Att lägga till ett element är därför lätt, men att hitta det kräver att programmet börjar i ena änden av listan och går igenom ett element i taget, i värsta fall kan det därför behöva stega sig igenom alla elementen. Elementen i en länkad lista lagras inte konsekutivt i minnet vilket är fallet för fält. Varje elementaccess kräver därför en adressering som är mer tidskrävande än vid access av fältelement.

Ett träd (y) visar sig ungefär lika bra vid tillägg och sökningar. Trädet lagrar sina element i form av ett binärt sökträd. I förhållande till den länkade listan behöver den (förmodligen) inte gå lika lång väg, den börjar vid trädets rot och letar sig steg för steg ut till rätt gren. Detta behöver den göra vid både tillägg och sökning, och därför tar dessa ungefär lika lång tid. Vid tillägg behöver den ev. också ändra på trädets struktur för att det ska fortsätta vara just ett träd (i värsta fall skulle ett träd annars kunna bli en länkad lista, d.v.s. alla grenar ligger på en rad efter varandra). Denna omstrukturering kallas för *balansering*. För att kunna avgöra var elementen ska lagras måste de kunna jämföras med varandra (d.v.s. implementera Comparable<T> i Java).

En hashtabell (z) ter sig i det här testet som det bästa alternativet, även om den länkade listan slår den för tillägg. Dess snabbhet beror på att den internt använder sig av ett fält (array), men den kan ändå inte garantera att alla element lagras på olika positioner. Flera element som lagras på samma position läggs därför i en länkad lista. Om (teoretiskt) alla element lagrades på en position i fältet skulle alltså en hashtabell degraderas till en enda länkad lista och heller inte prestera bättre. Utmaningen ligger därför i att försöka placera elementen så jämt över fältets positioner som möjligt. En position i ett fält nås som vanligt genom dess index, alltså ett positivt heltal. För att bestämma indexet för ett element utförs ofta en beräkning som involverar elementets inneboende tillstånd (känns det bekant? i Java används metoden hashCode()).

Varför används då träd överhuvudtaget om hashtabellen presterar bättre? Ett träd har ytterligare en egenskap, det håller elementen sorterade! Dessutom är värsta falls-beteendet för hashtabeller linjär, d.v.s. det kan bli lika dyrt att söka som i en länkad lista. Skillnaden mellan förväntad genomsnittstid kontra värsta falls-tid framgår av tabellen nedan där n är antalet befintliga element i strukturen. Vi antar också att de träd vi använder garanterat är balanserade, vilket är fallet med t.ex. TreeSet och TreeMap. Genomsnittsfallet för hashtabeller är något förenklat men är inte en funktion av antalet element utan av fyllnadsgraden hos tabellen.

Struktur	Mean add	Worst add	Mean contains	Worst contains
Länkad lista	1	1	n/2	n
Balanserat träd	$\log_2(n)$	$\log_2(n)$	$\log_2(n)$	$\log_2(n)$
Hashtabell	1	n	1	n

Uppgift 5

Programmet krashar eftersom Bicycle inte är en subklass till Car. Då programmet kommer till det element som är en Bicycle i vehicles blir det ClassCastException; en cykel släpper inte ut koldioxid (har inte metoden getCO2EmissionLevel()).

Då vi använder "råa typer", d.v.s. vi utelämnar den möjliga typparametern finns det ingen möjlighet för kompilatorn att avgöra om vi vid ett senare tillfälle förväntar oss att det endast finns objekt av en given typ i samlingen.

Lösningen blir alltså att inskränka samlingen genom att istället deklarera variabeln vehicles som:

```
Collection<Car> vehicles = new HashSet<Car>;
```

Försöker vi lägga in något annat i samlingen än ett objekt av typen Car, t.ex. en cykel av typen Bicycle, kommer kompilatorn att reagera och ett kompileringsfel uppstår.

```
import java.util.Collection;
import java.util.HashSet;
public class Vehicles {
   public static void main(String[] args) {
      Collection<Car> vehicles = new HashSet<Car>();
      vehicles.add(new Bicycle()); // Illegal, now caught by the compiler.
      vehicles.add(new Car());
      vehicles.add(new RaceCar());
      for (Car obj : vehicles) {
          System.out.println(obj.toString() + " emits " + obj.getCO2EmissionLevel() + " g CO2 / km");
      }
   }
}
```

Uppgift 6

Problemet är att metoderna next och remove inte följer sina specifikationer.

```
import java.util.Date;
import java.util.Iterator;
import java.util.NoSuchElementException;
import java.util.Random;
public class DummyDates implements Iterable<Date> {
   private final static int SIZE = 15;
   private Date[] dates = new Date[SIZE];
   private int actualSize;
   public DummyDates() {
       Random r = new Random();
       // Fill the array....
       for (int i = 0; i < SIZE; i++) {
           dates[i] = new Date(r.nextInt());
       }
       actualSize = SIZE;
   }
   public Iterator<Date> iterator() {
       return new DSIterator();
   }

   private class DSIterator implements Iterator<Date> {
       // Start stepping through the array from the beginning
       private int next = 0;
       private boolean nextCalled = false;
       public boolean hasNext() {
           return (next < actualSize);  // Check if the current element is the last in the array
       }
       /**
        * @pre hasNext() returns true
        */
       public Date next() {
           if (hasNext()) {
               nextCalled = true;
               return dates[next++];
           } else {
               throw new NoSuchElementException();
           }
       }
       /**
        * @pre next must have been called after last call to remove
        */
       public void remove() {
           if (!nextCalled) {
               throw new IllegalStateException();
           }
           next--;
           for (int i = next ; i < actualSize - 1; i++) {
               dates[i] = dates[i + 1];
           }
           dates[SIZE - 1] = null;
           actualSize--;
           nextCalled = false;
       }
   }
}
```