In diesem Kapitel:
- BitSet
- Observer / Observable
- Random
- StringTokenizer
- Timer *und* TimerTask
- Math *und* StrictMath

Verschiedene Utilities

Computers are useless - they can only give you answers.
- Pablo Picasso

Im Package `java.util` finden Sie einige weitere Hilfsprogramme, Klassen und Interfaces. Die meisten dieser Konstrukte gehören zum Collection Framework. Daneben befinden sich auch Hilfsprogramme, die für spezielle Aufgaben entwickelt wurden.

- `BitSet` - ein Bit Vektor mit dynamischer Grösse
- `Observer / Observable` - dieses Interface/Klassen Paar erlaubt es einem Objekt `Observable` zu sein, indem ihm ein oder mehrere `Observer` Objekt zugeordnet werden, welche informiert werden, falls sich beim `Observable` Objekt etwas ändert.
- `Random` - eine Klasse, mit deren Hilfe Sequenzen von Pseudozahlen generiert werden.
- `StringTokenizer` - eine Klasse, welche eine Zeichenkette in Tokens zerlegt. Das Trennzeichen kann selber definiert werden.
- `Timer / TimerTask` - eine Möglichkeit Aufgaben zeitlich gezielt zu starten (Scheduler)

Neben diesen java.util Klassen schauen wir uns auch noch die zwei Klassen aus dem Package `java.lang` an:

- `Math` - eine Klasse, welche die grundlegenden mathematischen Funktionen zur Verfügung stellt, wie beispielsweise trigonometrische Funktionen, Potenzieren, Logarithmen und so weiter.
- `StrictMath` - eine Klasse, welche die selben Methoden wie `Math` definiert, aber garantiert, dass alle Berechnungen in strikter Gleitkommaarithmetik ausgeführt werden. Dies garantiert absolut reproduzierbare Ergebnisse.

1.1. BitSet

Die BitSet Klasse stellt Bit Vektoren zur Verfügung, welche dynamisch wachsen können. Ein BitSet besteht aus 2^{31} `true` oder `false` Bits, welche anfänglich alle `false` sind. Die Bits werden indexiert, von `0` bis `Integer.MAX_VALUE`. Sie können individuell gesetzt, gelöscht oder gelesen werden. `BitSet` benötigt lediglich soviel Platz, dass das höchste Bit, welches gesetzt wurde, gespeichert werden kann. Alle weiteren Bits sind implizit auf `false` gesetzt.

Die Klasse besitzt zwei Konstruktoren:

`public BitSet(int size)`
: kreiert ein neues `BitSet` mit einer Speicherkapazität, die ausreicht, um `o` bis `size-1` Bits aufzunehmen. Alle Bits sind beim Initialisieren `false`.

`public BitSet()`
: kreiert ein neues `BitSet` mit einer Standardspeicherkapazität Alle Bits werden mit `false` initialisiert.

Um die einzelnen Bits zu manipulieren können Sie eine der drei folgenden Methoden einsetzen:

`public void set(int index)`
: setzt das Bit an der `index`-ten Stelle auf `true`.

`public boolean get(int index)`
: liefert den Wert des `index`-ten Bits.

`public void clear(int index)`
: setzt das Bit an der `index`-ten Stelle auf `false`.

Zusätzliche Methoden verändern die Bits auf Grund von Verknüpfungen, logischen Operationen:

`public void and(BitSet other)`
: verknüpft dieses `BitSet` mit den Elementen des BitSets `other`. Der Wert an er Stelle i ist `true`, falls sowohl beim `BitSet`, als auch im `BitSet other` an dieser Stelle `true` steht.

`public void andNot(BitSet other)`
: löscht die Bits im `BitSet`, falls sie im `BitSet other` gesetzt sind. Ein Bit ist genau dann `true`, falls es im `BitSet true` ist und im `BitSet other false`.

`public void or(BitSet other)`
: verknüpft jedes `Bit` des BitSets mit einer ODER Verknüpfung mit dem entsprechenden Bit des `BitSet other`. Ein Bit ist also genau dann `true`, falls es im `BitSet` oder im `BitSet other true` ist.

`public void xor(BitSet other)`
: logische XOR Verknüpfung. Ein Bit im `BitSet` ist genau dann `true`, falls es vor der Verknüpfung anders als im `BitSet other` war.

Zusätzlich besitzt `BitSet` einige weitere Methoden:

`public int size()`
: liefert die Anzahl Bits, welche aktuell im `BitSet` gespeichert werden. Falls Sie ein Bit setzen, welches oberhalb von `size()` ist, wird das `BitSet` erweitert.

`public int length()`
: liefert den Index des höchsten Bits im `BitSet` plus 1.

`public int hashCode()`
: liefert einen brauchbaren Hash-Code für dieses `BitSet` auf Grund der Werte seiner Bits. Ein Problem hat dieser Hash-Code: falls Sie die Werte der Bits verändern und mit Maps arbeiten, wird auf Grund der veränderten Bits ein neuerHash-Code berechnet dh. Sie werden Ihr `BitSet` nicht mehr finden, da der Hash-Code sich verändert hat.

`public boolean equals(Object other)`
: liefert `true`, falls alle Bits im (Objekt) `BitSet` den selben Wert wie im referenzierten `BitSet` haben.

Zu beachten ist, dass eine Negation aller Bits mit einer vordefinierten Methode nicht vorgesehen, weil nicht möglich ist, auf Grund der Speicherung des BitSets. Sie können aber diese Funktion leicht selber realisieren: falls Sie die ersten n Bits im BitSet negieren wollen, könnten Sie beispielsweise eine xor Verknüpfung mit einem BitSet durchführen, in dem alle Bits true sind. Das eigentliche Problem ist die dynamische Länge, deswegen gibt es keine allgemeine Methode für die Operation.

Hier ein Beispiel, welche ein `BitSet` einsetzt, um die Zeichen zu markieren, welche in einer Zeichenkette vorkommen. Jede Position entspricht einem Unicode Zeichen: Position 0 ist \u0000, die 97-ste Position entspricht 'a' und so weiter. Das BitSet kann ausgedruckt werden, um die Buchstaben anzuzeigen.

```
package beispiel;

import java.util.*;

public class WhichChars {
  private BitSet used = new BitSet();

  public WhichChars(String str) {
    for (int i=0; i < str.length(); i++)
      used.set(str.charAt(i) );           // setzen des Bits
  }

  public String toString() {
    String desc = "[";
    int length = used.length();
    for (int i=0; i<length; i++) {
      if (used.get(i) )
        desc += (char)i;
    }
    return desc + "]";
  }
```

```
}

package beispiel;

public class BitSetBeispiel {

  public BitSetBeispiel() {
    String str= "Testen 1 2 3 ";
    WhichChars w = new WhichChars(str);
    System.out.println("Original Zeichenkette : "+str);
    System.out.println("Analysierte Zeichenkette (Unicode sortierte
Zeichenfolge]: "+w);
  }
  public static void main(String[] args) {
    BitSetBeispiel bitSetBeispiel1 = new BitSetBeispiel();
  }
}
```

Beispiel:

```
Original Zeichenkette : Testen 1 2 3
Analysierte Zeichenkette (Unicode sortierte Zeichenfolge]: [ 123Tenst]
```

Selbsttestaufgabe 1

Bei der obigen Verschlüsselung der Unicode Zeichen gibt es im oberen Bereich Probleme, weil viele Bit Kombinationen unbenutzt bleiben. Sie können dies testen, indem Sie einfach alle Unicode Zeichen ausgeben, beispielsweise für 64'000 Bit Kombinationen. Sie sehen dort, dass die unterschiedlichen Zeichensets (Latin, Katakana, ...) nicht bümdig aneinander gereiht sind, sondern "Löcher" zwischen den einzelnen Sets bestehen. Eine Variante, das Problem zu beheben wäre, die gültigen Zeichen in eine Hash-Map abzuspeichern, ein `Character` Objekt pro gültige Bitdarstellung. Zum Abfragen müssen Sie in dieser Variante allerdings einen Iterator einsetzen.

1.2. Observer / Observable

Dieses Konstrukt, `Observer/Observable`, stellt ein Protokoll für eine beliebige Anzahl `Observer` Objekte zur Verfügung, welche Änderungen und Ereignisse in einer beliebigen Anzahl `Observable` Objekten beobachten.

Ein `Observable` Objekt erweitert die `Observable` Klasse, welche Methoden zur Verfügung stellt, um eine Liste von `Observer` Objekten mit Informationen über Änderungen bei den `Observable` Objekten zu versorgen.

Alle Objekte der "Interessenten" Liste müssen das `Observer` Interface implementieren.

Falls ein `Observable` Objekt sich verändert, werden die `Observer` darüber informiert. Das `Observable` Objekt führt die `notifyObservers()` Methode aus, welche die `update()` Methode der `Observer` Objekte aufruft.

```
package addobserver;

import java.util.Observable;

class FarbObservable extends Observable {
    public static String colorName(int i) {
        switch (i) {
        case 1: return ("rot");
        case 0: return ("weiss");
        case 2: return ("blaue");
        default: return (null);
            }
    }
    public void changeColor(int i) {
        setChanged();
        notifyObservers(new Integer(i));
        clearChanged(); // nicht nötig; notifyObservers() räumt auf
    }
}
package addobserver;

import java.util.Observable;
import java.util.Observer;

public class Main {
    final static int white = 0;
    final static int red = 1;
    final static int blue = 2;

    public static void main(String[] args) {
        FarbObservable colors = new FarbObservable();
        Statistician counter = new Statistician(3);

        // Observers zuordnen
        colors.addObserver(counter);
        colors.addObserver(new Echoer());
        System.out.println("Anzahl Observers: " + colors.countObservers());

        //  Observable verändern
```

```
        colors.changeColor(blue);
        colors.changeColor(white);
        colors.changeColor(red);
        colors.changeColor(blue);

        counter.report();

        // einen Observer entfernen
        colors.deleteObserver(counter);
        System.out.println("Anzahl Observers: " + colors.countObservers());
        // entferne alle Observers
        colors.deleteObservers();
        System.out.println("Anzahl Observers: " + colors.countObservers());
    }
}

package addobserver;

import java.util.Observer;
import java.util.Observable;

// Observer zählt die Anzahl Zugriffe auf die Integer 'arg'
class Statistician implements Observer {
    private int[] counts;
    Statistician(int array_size) {
        counts = new int[array_size];
    }
    public void update(Observable o, Object arg) {
        Integer int_obj = (Integer)arg;
        if (int_obj.intValue() < counts.length)
            ++counts[int_obj.intValue()];
    }
    public void report() {
        System.out.println("Änderungen: ");
        for (int i = 0; i < counts.length; i++)
            System.out.println(FarbObservable.colorName(i) + ": " +
counts[i]);
    }
}

// Observer zeigt die einzelnen Änderungen
class Echoer implements Observer {
    int current_color;
    public void update(Observable o, Object arg) {
        int new_color = ((Integer)arg).intValue();
        System.out.println("Von " +
                           FarbObservable.colorName(current_color) +
                           " auf " +
                           FarbObservable.colorName(new_color));
        current_color = new_color;
    }
}
```

Das `Observer` Interface besitzt folgende Methode, und nur diese:

`public void update(Observable obs, Object arg)`
: Diese Methode wird ausgeführt, falls das `Observable` Objekt obj sich verändert oder ein Ereignis signalisiert.

Der `Observer/Observable` Mechanismus ist sehr allgemein einsetzbar. Sie können bei der Definition Ihrer `Observable` Klasse selber festlegen, unter welchen Umständen welche `Observer` Objekte mittels `update()` informiert werden sollen. Das `Observable` Objekt unterhält ein "verändert" Flag, mit dem die Methoden feststellen können, ob sich etwas verändert hat.

`protected void setChanged()`
: markiert das Objekt als "geändert" - die Methode `hasChanged()` liefert in diesem Fall `true`, informiert aber keinen Observer.

`protected void clearChanged()`
: zeigt an, dass ein Objekt nicht länger als "verändert" bezeichnet wird oder dass alle Observer über die letzte Änderung informiert wurden - `hasChanged()` liefert nun `false`.

`public boolean hasChanged()`
: liefert den aktuellen Wert des "changed" / "verändert" Flags.

Falls eine Veränderung eintritt, sollte das `Observable` Objekt seine `setChanged()` Methode aufrufen und die `Observer` informieren.

`public void notifyObservers(Object arg)`
: informiert alle `Observer` Objekte in der Liste, dass sich etwas verändert hat. Anschliessend wird das "changed" Flag gelöscht. Pro `Observer` in der Liste wird seine `update()` Methode aufgerufen mit dem `Observable` Objekt als ersten Parameter und `arg` als zweites.

`public void notifyObservers()`
: äquivalen zu `notifyObservers(null)`

Die folgenden Methoden der `Observable` Klasse unterhalten die Liste der `Observer` Objekte:

`public void addObserver(Observer o)`
: fügt den Observer `o` der Observer-Liste hinzu.

`public void deleteObserver(Observer o)`
: löscht den Observer `o` aus der Observer-Liste.

`public void deleteObservers()`
: löscht alle Observer Objekte aus der Observer-Liste.

`public int countObservers()`
: liefert die Anzahl Observer in der Observer-Liste.

Die Methoden der Observable Klasse verwenden synchronisierte Methoden, falls auf Objekte zugegriffen werden soll. Ein Thread könnte beispielsweise einen Observer eintragen wollen, während ein anderer einen entfernt und ein dritter eine Zustandsänderung mitteilen möchte.

Die `update()` Methode verwendet aus Sicherheitsgründen keine Synchronisation: die Gefahr eines Deadlocks wäre zu gross!

Die Standardimplementation der `notifyObservers()` Methode verwendet einen synchronisierten Snapshot der aktuellen Observer-Liste bevor die `update()` Methode ausgeführt wird. Das hat beispielsweise zur Folge, dass ein Objekt über eine Änderung noch informiert werden könnte, selbst wenn es aus der Liste ausgetragen wird (weil notify...() mit einem Snapshot arbeitet, der bereits wieder veraltet ist). An solche Seiteneffekte müssen Sie bei der Implementation denken, falls so etwas auftreten könnte.

Die `notifyObserver()` Methode benützt den aufrufenden Thread, um die `update()` Methode für jeden `Observer` auszuführen. Die Reihenfolge, in der die `Observer` informiert werden, ist nicht festgelegt. Falls Sie ein spezielleres Modell für den notify-Mechanismus benötigen, müssen / können Sie dies in einer erweiternden Klasse selber definieren.

Hier ein weiteres Beispiel für einen möglichen Einsatz des `Observer / Observable` Paares:

```
package programmskizze;

import java.util.*;

public class Users extends Observable {
  private Map loggedIn = new HashMap();

  public void login(String name, String password) throws BadUserException {
    if (!passwordValid(name, password) ) throw new BadUserException(name);

    UserState state = new BadUserException(name);
    loggedIn.putAll(name,state);
    setChanged();
    notifyObservers(state);
  }

  public void logout(UserState state) {
    loggedIn.remove(state.name());
    setChanged();
    notifyObservers(state);
  }

  // ...

}
```

Ein User Objekt speichert die Benutzerinformationen in einem Map: die eingeloggten Benutzer werden eingetragen und ein UserState pro eingeloggter Benutzer unterhalten. Falls jemand einlogged oder auslogged wird allen Observer Objekten die UserState Information zurVerfügung gestellt. Die notifyObservers() Methode sendet Messages nur dann, falls sich der Zustand ändert. Im Programm muss also mit der Methode setChanged() signalisiert

werden, dass sich etwas bei den Users verändert hat. Sonst kann notifyObservers() nicht aktiv werden.

Und so könnte ein Benutzer, ein Observer, sich laufend über die eingeloggten Benutzer auf dem Laufenden halten, indem er die Methode update() implementiert:

```
package programmskizze;

import java.util.*;

public class Watcher implements Observer {
  Users watching;

  public Watcher(Users user) {
    watching = user;
    watching.addObserver(this);
  }

  public void update(Observable user, Object whichState) {
    if (user != watching) throw new IllegalArgumentException();

    UserState state = (UserState)whichState;
    if(watching.loggedIn(state) ) // Users logged in
      addUser(state);
    else
      removeUser(state);
  }
}
```

Ein Watch Objekt beobachtet ein bestimmtes Users Objekt. Immer wenn ein Benutzer einlogged oder auslogged, wird das Watch Objekt informiert.

Der `Observer / Observable` Mechanismus ist analog zu `wait/notify` für Threads. Aber es gibt feine Unterschiede:

- der Thread Mechanismus garantiert den synchronisierten Zugriff.
- der Observer Mechanismus gestattet es beliebige Beziehungen zwischen zwei Teilnehmern festzulegen, unabhängig vom Threading Modell.

Beide Modelle besitzen Informationsproduzenten (Observable und den Aufrufer von notify()) und Informationskonsumenten (Observer und den Aufruf von wait()), aber beide Mechanismen dienen speziellen Anwendungszwecken. Verwenden Sie wait/notify, falls Sie ein bestimmtes Threading Modell implementieren; verwenden Sie Observer / Observable wenn Sie allgemeinere Mechanismen implementieren möchten.

Selbsttestaufgabe 2

Implementieren Sie ein einfaches Producer-Consumer System (wie im Abschnitt Threads) mittels Observer / Observable. Der Einfachheit halber sei das System folgendermassen aufgebaut:
es wird Ware in ein Lager eingelagert und von Konsumenten konsumiert.
Das Lager / der einlagernde Produzent ist die Observable; der Kunde / der Konsument ist der Observer (er kauft, sobald die Ware an Lager ist).

1.3. Random

Die `Random` Klasse kreiert Objekte, welche unabhängige Sequenzen von Pseudozufallszahlen generieren. Falls Ihnen die Folge egal ist und Sie lediglich einen double Zufallswert wollen, dann können Sie auch mit `java.lang.Math.random` arbeiten. Diese Methode kreiert ein `Random` Objekt und liefert Pseudozahlen aus diesem `Random` Objekt.Mit der `Random` Klasse können Sie einfach die Generierung der Zufallszahlen genauer kontrollieren.

`public Random()`
kreiert einen neuen Zufallsgenerator. Der Startwert basiert auf der Systemzeit.

`public Random(long seed)`
kreiert einen Zufallsgenerator mit einem spezifischen Startwert `seed` (dem Korn). Zwei Zufallsgeneratoren mit dem selben `seed` Wert liefern die selben Zufallsfolgen. Sie haben also die Möglichkeit Zufallszahlen zu reproduzieren.

`public void setSeed(long seed)`
setzt die Wurzel des Zufallsgenerator auf den Wert `seed`. Diese Methode können Sie öfters aufrufen und somit die selbe Pseudozufallszahlensequenz mehrfach erzeugen.

`public int nextBoolean()`
liefert einen zufälligen Wahrheitswert.

`public int nextInt()`
liefert eine ganze Zufallszahl, uniform verteilt über den gesamten Integer Wertebereich (Integer.MIN_VALUE bis Integer.MAX_VALUE). Der Zusatz next... garantiert, dass Sie nicht immer wieder vorne starten und die selbe Zahl erhalten.

`public int nextInt(int ceiling)`
funktioniert analog zu `nextInt()` mit der Erweiterung, dass die Zahl kleiner als der Deckenwert ceiling und positiv ist. Falls ceiling negativ ist, wird eine `IllegalArgumentException` geworfen.

`public long nextLong()`
liefert eine Zahl zwischen Long.MIN_VALUE und Long.MAX_VALUE, inklusive.

`public void nextBytes(byte[ ] buf)`
füllt das Byte-Array mit Zufallsbytes.

`public float nextFloat()`
liefert eine Zufallszahl, Gleitkommazahl im Bereich 0.0f (inklusive) bis 1.0f (exklusive).

`public double nextDouble()`
liefert wie oben eine Double Zahl zwischen 0 und 1.

`public double nextGaussian()`
liefert eine Pseudozufallszahl mit einer Gaussverteilung zur Auswahl. Mittelwert ist 0.0 und Standardabweichung 1.0.

Alle next... Methoden verwenden synchronisierte Zugriffe, auch die setSeed() Methode, damit keine Zugriffskonflikte auftreten.

Die Random Klasse spezifiziert den Algorithmus für die Generierung der Zufallszahlen, gestattet es Ihnen aber, eigene Algorithmen einzusetzen. Der Basisalgorithmus ist in der Methode next() definiert du wird für alle anderen Methoden eingesetzt. Daher können Sie den Algorithmus ersetzen, indem Sie die next() Methode überschreiben

Hier als Beispiel einen einfachen Roulette-Simulator:

```
package random;

// Roulette spielen

import java.util.Random;
class Roulette {
    Random generator = new Random();
    // Rad drehen und Wert zurücklieferen (00, 0, 1-36)
    String spin() {
        int rand = generator.nextInt();
        int num = Math.abs(rand % 38);

        switch (num) {
        case 37: return ("00");
        case 36: return ("0");
        default: return (Integer.toString(num + 1)); // 1- 36 inclusive
        }
    }
    //
    void changeDealer() {
        generator.setSeed(System.currentTimeMillis());
    }
}
public class Main {
    public static void main(String[] args) {
        Roulette r = new Roulette();

        // dreh 20 Mal
        for (int i = 0; i < 20; i++)
            System.out.println(i + ": " + r.spin());

        //
        r.changeDealer();

            // dreh 20 Mal
        for (int i = 0; i < 20; i++)
            System.out.println(i + ": " + r.spin());
    }
}
```

Selbsttestaufgabe 3

Generieren Sie ein Multi-User Würfelspiel. Ihr Gegner ist der Computer.

1.4. StringTokenizer

Die StringTokenizer Klasse zerlegt eine Zeichenkette in Ihre Bestandteile, mit Hilfe von Delimitern. Die Zeichenkette wird in eine Sequenz, eine Enumeration aufgeteilt. Daher implementiert der StringTokenizer auch das Enumeration Interface. Sie sehen, die Legacy Collections sind tief im System im Einsatz.

StringTokenizer stellt Methoden zur Verfügung, welche sich von jenen der Enumeration wesentlich unterscheiden. Zudem wird intern ein SnapShot der Zeichenkette erstellt, um Zugriffskonflikte zu vermeiden. Es könnte ja sein, dass die Zeichenkette gerade verändert wird und Sie eine Zerlegung in Tokens vornehmen wollen.

Das folgende Programmskelett zerlegt eine Zeichenkette in die einzelnen Wörter. Delimiter ist dabei das Leerzeichen oder das Komma.

```
package einfachesbeispiel;

import java.util.*;

public class Beispiel {

  public Beispiel() {
    String str = "vo Luzärn gägä Weggis zue, ... und so weiter, %& % &/ @
*#";
    StringTokenizer tokens = new StringTokenizer(str, " ,");
    while (tokens.hasMoreTokens() )
      System.out.println(tokens.nextToken());
  }
  public static void main(String[] args) {
  Beispiel bsp = new Beispiel();
  }
}
```

mit der Ausgabe:

```
vo
Luzärn
gägä
Weggis
zue
...
und
so
weiter
%&
%
&/
@
*#
```

Die einzelnen Trennzeichen sind als Parameter in einer Zeichenkette beim Konstruktor angegeben.

Die Klasse besitzt mehrere Methoden, mit deren Hilfe Sie kontrollieren können, was als Wort zu betrachten ist.

`public StringTokenizer(String str, String delim, boolean returnTokens)`
konstruiert einen `StringTokenizer` auf der Zeichenkette `str` und verwendet die Zeichen in der Zeichenkette `delim` als Delimiter. Die Boole'sche Variable `returnTokens` steuert die Berücksichtigung der Delimiter: falls diese Variable `true` ist, werden auch die Delimiter als Worte betrachtet und zurückgegeben.

`public StringTokenizer(String str, String delim)`
dieser Konstruktor ist äquivalent zu `StringTokenizer(str, delim, false)`. Die Delimiter werden also nicht berücksichtigt.

`public StringTokenizer(String str)`
dieser Konstruktor ist äquivalent zu `StringTokenizer(str, " \t\n\r\f")`: die Delimiter sind die sogn.'Whitespace' Zeichen.

`public boolean hasMoreTokens()`
liefert `true`, falls weitere Tokens existieren.

`public String nextToken()`
liefert den nächsten Token des Strings. Falls keine weiteren Tokens mehr existieren, wird eine `NoSuchElementException` geworfen.

`public String nextToken(String delim)`
verändert den Delimiter auf die Zeichen in `delim` und liefert das nächste Wort gemäss diesem neuen Delimiter. Sie können das Delimiter Set nicht ändern ohne ein Wort zu lesen!
Falls keine weiteren Tokens existieren, wird eine `NoSuchException` geworfen.

`public int countTokens()`
liefert die Anzahl Tokens, welche noch vorhanden sind (in der restlichen Zeichenkette). Diese Zahl ist also gleich der Anzahl `nextToken()` Aufrufe, die Sie noch absetzen könnten, bevor eine Exception geworfen würde. Diese Methode, die Anzahl Worte zu bestimmen, ist viel schneller als eine selbstgebastelte Methode, die die Worte bestimmt und zählt, weil in dieser Methode die Worte nicht gebildet werden.

Die Methoden des StringTokenizer, welche aus der Enumeration übernommen wurden, entsprechen `hasMoreElement()` und `nextElement()` - im StringTokenizer ist dies `hasMoreTokens()` und `nextTokens()`.

Falls Sie noch umfassendere Kontrolle über den Zerlegungsprozess benötigen, können Sie den "StreamTokenizer" einsetzen. Damit Sie diesen auf eine Zeichenkette anwenden können, müssen Sie die Zeichenkette mit einem StringReader wrappen. Aber in der Regel reicht die Funktionalität des StringTokenizers auch schon.

Selbsttestaufgabe 4

1) Zerlegen Sie eine Zeichenkette aus Gleitkommazahlen und addieren Sie deren Werte.
2) Setzen Sie einen Tokenizer ein, um die Anzahl Worte einer Textdatei zu bestimmen.

1.5. Timer und TimerTask

Die `Timer` Klasse kann eingesetzt werden, um Tasks, die zu einem späteren Zeitpunkt ausgeführt werden sollen, aufzusetzen. Jedes `Timer` Objekt besitzt einen dazugehörigen Thread, welcher zu dem im `TimerTask` Objekt vordefinierten Zeitpunkt aufwacht.

Das folgende Programmfragment zeigt den Speicher der Virtual Machine periodisch an:

```
public JVMMemory() {
    long start = System.currentTimeMillis();
    System.out.println("Kreieren des Timers");
    Timer timer = new Timer(true);
    System.out.println("Scheduling der Task");
    timer.scheduleAtFixedRate(new MemoryWatchTask(), 0, 1000);
  }
```

Dieses Programm kreiert ein neues `Timer` Objekt, welches für das Scheduling und Ausführen der `MemoryWatchTask` zuständig ist. Der `true` Parameter im `Timer` Konstruktor gibt an, dass ein Daemon Thread verwendet werden soll. Dadurch wird auch die Speicherabfrage beendet, sobald die anderen Threads beendet sind. Falls der Parameter auf `false` wäre, würde dieser Thread auch weiterlaufen, nachdem das Hauptprogramm beendet ist, der Garbage Collector aber noch nicht aufgeräumt hat.

Der Scheduling Teil zeigt, dass zu fixen Abständen (1000 Millisekunden) ab dem Startzeitpunkt (0) die `MemoryWatchTask()` ausgeführt werden soll.

Vereinfacht gesagt heisst das, dass etwa alle Sekunden eine Anzeige des Speichers erfolgen sollte.

```
package timertask;

import java.util.TimerTask;
import java.util.Date;

public class MemoryWatchTask extends TimerTask{
  public void run() {
    System.out.println(new Date()+": ");
    Runtime rt = Runtime.getRuntime();
    System.out.print(rt.freeMemory()+" frei,");
    System.out.print(rt.totalMemory()+ " total");
    System.out.println();
  }
}
```

Das Programm `MemoryWatch` erweitert die abstrakte `TimerTask` Klasse, um eine Klasse zu definieren, welche den Speicherbedarf abfragt und anzeigt. Die Klasse `TimerTask` implementiert das `Runnable` Interface und im Speziellen die `run()` Methode, die Methode, welche vom `Timer` Objekt aufgerufen wird. Die Ausführung dieser Aufgabe erfolgt einmal pro Sekunde, gemäss der Festlegung im ersten Aufruf.

Das vollständige Programm könnte folgendermassen aussehen:

```
package timertask;

import java.util.*;
import java.util.TimerTask;
import java.util.Date;

public class JVMMemory {
  public static void main(String[] args) {
    long start = System.currentTimeMillis();
    System.out.println("Kreieren des Timers");
    Timer timer = new Timer(true);
    System.out.println("Scheduling der Task");
    System.out.println(new Date()+": ");
    Runtime rt = Runtime.getRuntime();
    System.out.print(rt.freeMemory()+" frei,");
    System.out.print(rt.totalMemory()+ " total");
    System.out.println("\n-------------------------------------");
    timer.scheduleAtFixedRate(new MemoryWatchTask(), 0, 1000);
    try {
      Thread.sleep(1000);
    } catch(Exception e) { }
  }
}
```

Der TaskTimer besitzt folgende Methoden:

`public abstract void run()`
definiert die Aktion(en), welche durch die `TimerTask` ausgeführt werden sollen.

`public boolean cancel()`
beendet diesen `TimerTask` , so dass er nie mehr ausgführt wird. Falls die Task wiederholt ausgeführt werden sollte, liefert `cancel() true`. Die selbe Rückgabe erfolgt, falls die Task nur einmal ausgeführt werden soll, aber noch nicht ausgeführt wurde.
Sonst wird `false` zurückgegeben. Die Methode liefert also `true`, falls sie die Task daran gehindert hat eine gescheduled Aktion auszuführen.

`public long scheduledExecutionTime()`
liefert die letzte geplante Ausführzeit (eventuell die gerade laufende) Der Rückgabewert ist die Zeit in Millisekunden. Oft wird diese Methode innerhalb `run()` verwendet, um festzustellen, um festzustellen, ob die Aufgabe noch ausgeführt werden kann.

Sie können entweder eine Task oder einen Timer abbrechen. Falls Sie eine Task abbrechen wird diese in Zukunft nicht mehr ausgeführt. Beim Abbrechen eines Timer Objekts werden die durch dieses Objekt in Zukunft nicht mehr ausgeführt.

Jedes Timer Objekt verwendet einen Thread, um die Ausführung der Tasks zu steuern. Sie können im Timer Konstruktor steuern, ob es sich bei diesem Thread um einen normalen, oder um einen Daemon Thread handeln soll. Wir haben ein Beispiel dazu bereits oben gesehen.

Hier die Zusammenstellung der Konstruktoren:

public Timer(boolean isDaemon)
kreiert einen neuen Timer, dessen Thread im Daemon Zustand isDaemon ist, also ein Daemon Thread, falls isDaemon true ist, ein normaler Thread sonst.

public Timer()
dieser Konstruktor ist äquivalent zu Timer(false).

Falls Sie den Default Konstruktor verwenden, erhalten Sie einen Benutzerthread. Falls der Timer nicht mehr erreichbar ist, also alle Referenzen darauf gelöscht sind und somit keine Aufgabe mehr ausgeführt werden muss, wird der Thread beendet.

Dieses Verhalten ist also vom Garbage Collector abhängig und Sie sollten sich nicht zu sehr darauf stützen, weil Sie nie genau wissen können wann und ob dieser (Garbage Collector) Thread aktiv wird.

Insgesamt kennt man drei Arten, Aufgaben zeitlich festzulegen (zu schedulen):

- ein *einmaliges* Scheduling bedeutet, dass die Aufgabe einmal und nur einmal ausgeführt wird.

- ein *fixed-delay* Scheduling bedeutet, dass die Aufgabe erst nach einer bestimmten Wartezeit ausgeführt wird. Die Aufgabe wird periodisch ausgeführt, aber das erste Mal erst nach einer Verzögerungszeit. Dabei können Ungenauigkeiten bei der Ausführungszeit auftreten, weil eventuell gerade der Garbage Collector oder andere Threads mit hoher Priorität aktiv sein können.

- ein *fixed-rate* Scheduling bedeutet, dass die Aufgaben in fixen Intervallen ausgeführt werden. Dabei tritt eine Finesse auf: das Zeitintervall wird ab Startzeit, also ab der ersten Ausführung der Aufgabe berechnet, nicht etwa als Zeitdifferenz der einzelnen Aufgaben. Es könnte also passieren, dass die Intervalle zwischen der Ausführung der Aufgaben unterschiedlich ist. Diese Technik hat also klare Vorteile zu anderen Methoden, weil Sie davon ausgehen können, dass die Aufgaben einigermassen regelmässig ausgeführt werden. Man verwendet diese Scheduling Art beispielsweise für Alarme oder Timer.

`public void schedule(TimerTask task, Date time)`
führt eine Aufgabe einmal zu einer bestimmten Zeit aus.

`public void schedule(TimerTask task, long delay)`
führt die Task einmal nach der Wartezeit `delay` (in Millisekunden) aus.

`public void schedule(TimerTask task, Date firstTime, long period)`
führt die Task nach dem erstenmal Ausführen regelmässig nach `period` Millisekunden aus.

`public void schedule(TimerTask task, long delay, long period)`
führt die Task nach dem erstenmaligen Abwarten von `delay` Millisekunden das erstemal aus, anschliessend regelmässig nach `period` Millisekunden [*fixed-delay Scheduling*].

`public void scheduleAtFixedRate(TimerTask task, Date firstTime, long period)`

führt die Task nach dem erstmaligen Ausführen zum Zeitpunkt `firstTime` regelmässig nach `period` Millisekunden aus.

`public void scheduleAtFixedRate(TimerTask task, long delay, long period)`

führt die Task nach dem erstmaligen Ausführen zum Zeitpunkt `firstTime` regelmässig nach `period` Millisekunden aus [*fixed-rate Scheduling*].

Falls Sie irgend einen Zeitpunkt angeben, der in der Vergangenheit liegt, wird die Aufgabe sofort ausgeführt. Alle Zeitintervalle werden in Millisekunden angegeben. Falls ein Overflow resultieren würde (die Anzahl Millisekunden plus die aktuelle Zeit zu einer Zahl führen würde, welche grösser als die maximal darstellbare Zahl ist), dann wird eine `IllegalArgumentException` geworfen.

Jedes `TimerTask` Objekt kann lediglich durch ein `Timer` Objekt verwaltet werden. Ein einmal getoppter `Timer` kann keine neuen Tasks verwalten. Falls Sie dies nicht beachten, wird eine `IllegalStateException` geworfen.

`Timer` Threads unterliegen den plattformabhängigen Beschränkungen. Sie können die Priorität der Threads in der `run()` Methode verändern und setzen. Allerdings muss der steuernde Thread eine höhere Priorität haben.

Sie können nicht abfragen welche Tasks durch einen bestimmten Timer gesteuert werden.

Selbsttestaufgabe 5

Modifizieren Sie das Beispielprogramm so, dass ein von Ihnen gewählter Scheduling - Typ verwendet wird. Die Übung dient dem Festigen des Stoffes über Scheduling.

Dieser Abschnitt zeigte Ihnen, wie Sie mit Hilfe höherer Konstrukte als den (primitiven / Basis-) Operationen `wait() / notify() / notifyAll()` mit Threads arbeiten können und sicher weniger Probleme generieren, als mit Threads direkt.

In der Regel wird man neben diesen Konstrukten auch mit Threading-Bibliotheken arbeiten, welche Ihnen ähnliche Konstrukte wie `Timer / TimerTask` anbieten.

1.6. Math und StrictMath

Die `Math` Klasse besteht aus statischen Konstanten und Methoden für allgemeine mathematischen Manipulationen, welche die normale Gleitkommarechnung verwenden.

Die `StrictMath` Klasse definiert die selben Konstanten und Methoden, verwendet aber immer Gleitkommadarstellungen. `StrictMath` ist also immer als `strictfp` definiert. `Math` braucht dies nicht zu sein. Beide Klassen definieren `double` Konstanten:

`E` repräsentiert *e* (2.718...) und `PI` repräsentiert π (3.1415...).

In der folgendes Tabelle sind alle Angaben zu Winkeln als Radius und alle Parameter und Rückgabewerte als `double` definiert, ausser es wird etwas anderes angezeigt.

Funktion	**Wert / Semantik**
`sin(a)`	Sinusfunktion
`cos(a)`	Cosinusfunktion
`tan(a)`	Tangensfunktion
`asin(v)`	Arcus Sinusfunktion
`acos(v)`	arccosinus(v) mit v in [0.0, 1.0]
`atan(v)`	arctangent(v), liefert Werte in [-PI/2, PI/2]
`atan2(v)`	arctangens(v), Werte in [-PI, PI]
`toRadians(d)`	Umwandlung von Winkeln in Radius
`toDegrees(r)`	Umwandlung von Radius in Winkel
`exp(x)`	e^x
`pow(y,x)`	y^x
`log(x)`	ln x (natürlicher Logarithmus)
`sqrt(x)`	Wurzel aus x>0
`ceil(x)`	kleinste ganze Zahl >= x
`floor(x)`	grösste ganze Zahl <= x
`rint(x)`	x gerundet auf die nächste ganze Zahl
`round(x)`	(int)floor(x +0.5) für float x
`abs(x)`	absoluter Wert von x
`max(x,y)`	grössere von x, y (beliebiger num. Datentyp)
`min(x,y)`	kleinerer von x,y (bel. num. Datentyp)

Die statische Methode `IEEEremainder()` berechnet den Rest gemäss IEEE-754. Der Modulo (remainder) Operator `%` gehorcht der Regel:

```
(x/y) * y + x % y == x
```

Die statische Methode `random()` generiert eine Zufallszahl r , mit 0<= r <=1. Falls Sie genauere Kontrolle über die Zufallszahl haben möchten, können Sie die Klasse `Random` einsetzen.

Selbsttestaufgabe 6

Implementieren Sie die mathematischen Funktionen in Ihrem Swing / AWT Taschenrechner.

VERSCHIEDENE UTILITIES