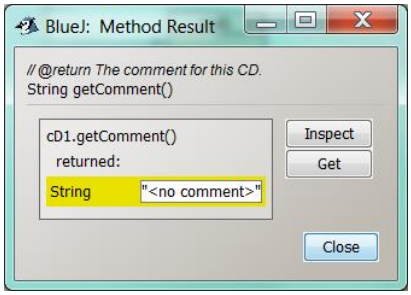
Aufgabe 1: Weitere Aufgaben aus dem Buch

1. Bearbeiten Sie die Aufgaben 8.2, 8.5 und 8.6 zum DoME Beispiel.

8.2:




```
CD: Zeitgeist (65 mins)
    Schiller
    tracks: 13
    super CD :)
```

8.5:

Es können nun Methoden von CD und Item aufgerufen werden.

```
title: Zeitgeist (65 mins)
    nice CD
```

8.6:

```
public class Item
{
   [...]

    /**
     * Return Title
     */
    public String getTitle()
    {
        return title;
    }
}

public class CD extends Item
{
   [...]

    /**
     * Print Artist and Title
     */
    public void printShortDetails()
    {
        System.out.println(artist + ": " + getTitle());
    }
}

Schiller: Zeitgeist


public class DVD extends Item
{
   [...]

    /**
     * Print Director and Title
     */
    public void printShortDetails()
    {
        System.out.println(director + ": " + getTitle());
    }
}
```

Schlümpfe: Die Rache

2. Diskutieren Sie in Gruppen die Aufgaben 8.10 und 8.11.
   done

3. Diskutieren Sie die Aufgabe 8.12 und überprüfen Sie Ihre Antworten gemäss Aufgabe 8.13.

```
s1 = p1;        Error: incompatible types - found Person but expected Student
s1 = p2;        Error: incompatible types - found Person but expected Student
p1 = s1;        OK
t1 = s1;        Error: incompatible types - found Student but expected Teacher
s1 = phd1;      OK
phd1 = s1;      Error: incompatible types - found Student but expected PhDStudent
```

Aufgabe 2: Quicksort

1. Implementieren Sie in der vorgegebenen Klasse QuickSort (siehe ILIAS) eine `main(...)` Methode mit Testfällen.

```
/* Copyright 2008 - Hochschule Luzern */

/**
 * Eine Implementationenen von Quicksort.
 *
 * @author Peter Sollberger
 * @version 1.2, 31. Oktober 2008
 */
public class Quicksort
{
    /**
     * Konstruktor macht nichts.
     */
    public Quicksort()
    {
    }

    /**
     * Quicksort Algorithmus mit rechtem Trennelement.
     * @param a Array zum Sortieren
     * @param left linke Grenze
     * @param right rechte Grenze
     */
    public void quickSort(int a[], int left, int right)
    {
        int up   = left;                // linke Grenze
        int down = right - 1;           // rechte Grenze (ohne Trennelement)
        int t    = a[right];            // rechtes Element als Trennelemt
        boolean allElementChecked = false;

        do
        {
            while (a[up] < t)
            {
                up++;                   // suchen größeres Element von links an
            }
            while ((a[down] > t) && (down > up))
            {
                down--;                 // suchen kleineres Element von rechts an
            }
            if (up < down)
            {
                exchange(a, up, down);  // austauschen
                up++;                   // linke und rechte Grenze verschieben:
                down--;
            }
            else
            {
                allElementChecked = true;
            }
        } while (!allElementChecked);   // Überschneidung

        exchange(a, up, right);         // Trennelement an endgültige Position (a[up])

        if (left < up - 1)
        {                               // mehr als 1 Element in linker Teilfolge?
            quickSort(a, left, up - 1); // linke Hälfte sortieren
        }
        if (up + 1 < right)
        {                               // mehr als 1 Element in rechter Teilfolge?
```

```
            quickSort(a, up + 1, right); // rechte Hälfte sortieren (ohne Trennelement)
        }
    }

    /**
     * Vertausche im übergebenen Array die beiden Elemente an der Position
     * firstIndex und secondIndex.
     */
    private void exchange(int a[], int firstIndex, int secondIndex)
    {
        int tmp;

        tmp            = a[firstIndex];
        a[firstIndex]  = a[secondIndex];
        a[secondIndex] = tmp;
    }

    public void testQuickSort(int[] myArray)
    {
        System.out.print("unsortiert: ");
        for (int i = 0; i < myArray.length; i++) {
            System.out.print(myArray[i] + " ");
        }
        System.out.println();

        // Array sortieren
        quickSort(myArray, 0, myArray.length-1);

        System.out.print("sortiert:   ");
        for (int i = 0; i < myArray.length; i++) {
            System.out.print(myArray[i] + " ");
        }
        System.out.println("\n");
    }

    public void main()
    {
        // init array
        int[] arrayOne   = {1,2,3,4,5,6,7};
        int[] arrayTwo   = {7,6,5,4,3,2,1};
        int[] arrayThree = {3,7,2,4,6,1,5};
        int[] arrayFour  = {0,0,1,1,2,3,4};
        int[] arrayFive  = {4,2,1,5,1,4,7};
        int[] arraySix   = {3,3};
        int[] arraySeven = {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};
        int[] arrayEight = {-2,2,-1,1,3,0};
        int[] arrayNine  = {-1535437,564651866,2,56874,-4567,-10,4567};
        int[] arrayTen   =
{0,1,0,1,0,1,0,1,0,1,0,1,0,1,0,1,0,1,0,1,0,1,0,1,0,1,0,1,0,1,0,1,0,1,0,1,0,1,0,1,0,1,0,1,0,1,0,1,0,1,0,1,0,1,0,1,0,
1};

        testQuickSort(arrayOne);
        testQuickSort(arrayTwo);
        testQuickSort(arrayThree);
        testQuickSort(arrayFour);
        testQuickSort(arrayFive);
        testQuickSort(arraySix);
        testQuickSort(arraySeven);
        testQuickSort(arrayEight);
        testQuickSort(arrayNine);
        testQuickSort(arrayTen);

    }

}
```

```
unsortiert: 3 7 2 4 6 1 5
sortiert:   1 2 3 4 5 6 7

unsortiert: 0 0 1 1 2 3 4
sortiert:   0 0 1 1 2 3 4

unsortiert: -1535437 564651866 2 56874 -4567 -10 4567
sortiert:   -1535437 -4567 -10 2 4567 56874 564651866

unsortiert: 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1
0 1 0 1 0 1
sortiert:   0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1
```

2. Implementieren Sie eine zweite Version von Quicksort, mit der "median-of-three" Verbesserung.

```
/**
 * Quicksort Algorithmus mit meedian-of-three
 * @param a Array zum Sortieren
 * @param left linke Grenze
 * @param right rechte Grenze
 */
public void quickSortMeedianOfThree(int a[], int left, int right)
{
    int up   = left;                // linke Grenze
    int down = right - 1;           // rechte Grenze (ohne Trennelement)

    boolean allElementChecked = false;

    // trennelement bestimmen
    // int t   = a[right];             // rechtes Element als Trennelemt
    int t;
    if (a[left] > a[right]) {
        if (a[left] > a[right/2]) {
            t = a[left];
            exchange(a, left, right);  // Trennelement verschieben
        }
        else {
            t = a[right/2];
            exchange(a, right/2, right);  // Trennelement verschieben
        }
    }
    else
    {
        if (a[right] > a[right/2]) {
            t = a[right];
        }
        else {
            t = a[right/2];
            exchange(a, right/2, right);  // Trennelement verschieben
        }
    }

    do
    {
        while (a[up] < t)
        {
            up++;                   // suchen größeres Element von links an
        }
        while ((a[down] > t) && (down > up))
        {
            down--;                 // suchen kleineres Element von rechts an
        }
        if (up < down)
        {
            exchange(a, up, down);  // austauschen
            up++;                   // linke und rechte Grenze verschieben:
            down--;
        }
        else
        {
            allElementChecked = true;
        }
    } while (!allElementChecked);   // Überschneidung

    exchange(a, up, right);         // Trennelement an endgültige Position (a[up])

    if (left < up - 1)
    {                               // mehr als 1 Element in linker Teilfolge?
        quickSort(a, left, up - 1); // linke Hälfte sortieren
    }
    if (up + 1 < right)
    {                               // mehr als 1 Element in rechter Teilfolge?
        quickSort(a, up + 1, right); // rechte Hälfte sortieren (ohne Trennelement)
    }
}
```

3. Vergleichen Sie die Rekursionstiefen der beiden Implementationen mit verschiedenen Testdaten (z.B. einer bereits sortierten Folge).

bereits sortierte Folge: meedian-of-three wird schneller sein

unsortierte Folge: meedian-of-three ggf. gleich wie „normal“

```
unsortiert: 37 65 58 34 24 60 63 78 84 34 52 64 13 57 10 42 69 27 80 67 14 24 7 60 79 30 49 22 52 10 59 50
54 39 49 55 18 42 30 22 95 95 87 78 54 97 54 79 37 82
sortiert:   Rekursionstiefe: 33
*sortiert:  Rekursionstiefe: 35

unsortiert: 10 10 13 14 18 22 22 24 24 27 30 30 34 34 37 37 39 42 42 49 49 50 52 52 54 54 54 55 57 58 59 60
60 63 64 65 67 69 7 78 78 79 79 80 82 84 87 95 95 97
sortiert:   Rekursionstiefe: 45
*sortiert:  Rekursionstiefe: 35
```

Aufgabe 3: Quicksort Speicherbedarf

Die Methode Quicksort ruft sich rekursiv auf. Bei jedem Aufruf werden die Übergabeparameter auf dem Stack abgelegt, und jeder Methodenaufruf belegt für seine Variablen ebenfalls einen gewissen Platz auf dem Stack. Nennen wir diesen Platzbedarf eine *Quick-Speichereinheit*.

1. Aus wie vielen Bytes besteht eine *Quick-Speichereinheit* beim klassischen Quicksort (Implementation gemäss Folien)?

```
public void quickSort(char[] a, int left, int right)
{
   int up = left;                // linke Grenze
   int down = right-1;           // rechte Grenze (ohne Trennelement)
   char t = a[right];            // rechtes Element als Trennelement
   boolean allChecked = false;   // austauschen beendet
```

Parameter:

char[]	*a*	*pointer => Java 32Bit => 32 Bit = 4 Byte*
int	*left*	*4 Byte*
int	*right*	*4 Byte*

Lokale Vars:

int	*up*	*4 Byte*
int	*down*	*4 Byte*
char	*t*	*2 Byte*
boolean	*allChecked*	*1 Byte*

➔ *23 Byte*

2. Wie verändert sich die *Quick-Speichereinheit* beim "Quick-Insertion-Sort" Verfahren?

```
public void quickInsertionSort(char[] a, int left, int right)
{
    if ((right-left) > M) {  // Klassischer Quicksort anwenden

        ...

        if ((up+1) < right)
            quickInsertionSort(a, (up+1), right);  // rek. Aufruf
    }
    else {  // Insertion Sort verwenden
        char tmp;
        int i, j;
```

Parameter:

Gleich wie bei Aufgabe 1:

char[]	*a*	*pointer => Java 32Bit => 32 Bit = 4 Byte*
int	*left*	*4 Byte*
int	*right*	*4 Byte*

Lokale Vars:

char	*tmp*	*2 Byte*
int	*i*	*4 Byte*
int	*j*	*4 Byte*

➔ *22 Byte*

3. Wie gross (in *Quick-Speichereinheiten*) muss der Stack bei Quicksort im günstigsten und im ungünstigsten Fall sein?

22 resp. 23 Byte

Aufgabe 4: Mergesort (optional)

1. Implementieren Sie den Mergesort Algorithmus.

```
/**
 * Write a description of class Mergesort here.
 *
 * @author HSLU
 * @version 5.6
 */
public class Mergesort
{
    private int[] b;
    /**
     * Nach aussen sichtbare Mergesort Methode.
     * Muss zuerst Zwischenspeicher bereitstellen.
     * @param a Array zum Sortieren
     */
    public void mergesort(int[] a)
    {
        b = new int[a.length];
        mergesortPrivate(a, 0, a.length-1);
    }

    /**
     * Rekursiver Mergesort Algorithmus.
     * Benötigt Hilfsarray b mit der Grösse von a
     * @param a Array zum Sortieren
     * @param left Linke Grenze, zu Beginn 0
     * @param right Rechte Grenze, zu Beginn a.length-1
     */
    private void mergesortPrivate(int a[], int left, int right)
    {
        int i, j, k, m;
        if(right > left) {
            m=(right + left) / 2; // Mitte ermitteln
            mergesortPrivate(a, left, m); // linken Teil sortieren
            mergesortPrivate(a, m + 1, right); // rechten Teil sortieren
            // "Mergen"
            for(i = left; i <= m; i++) { // linker Teil in Hilfsarray
                b[i] = a[i];
            }
            for(j = m; j < right; j++) { // rechter Teil umgekehrt in Hilfsarray
                b[right + m - j] = a[j + 1];
            }
            i = left; j = right; // i für rechte und j für linke Hälfte
            for(k = left; k <= right; k++) { // füge sortiert in a ein
                if(b[i] <= b[j]) {
                    a[k] = b[i];
                    i++;
                }
                else {
                    a[k] = b[j];
                    j--;
                }
            }
        }
    }

    public void testMergesort()
    {
        int[] arrayTest = {10, 10, 13, 14, 18, 22, 22, 24, 24, 27, 30, 30, 34, 34, 37, 37, 39, 42, 42, 49,
49, 50, 52, 52, 54, 54, 54, 55, 57, 58, 59, 60, 60, 63, 64, 65, 67, 69, 7, 78, 78, 79, 79, 80, 82, 84, 87,
95, 95, 97};
        mergesort(arrayTest);
        for (int i = 0; i < arrayTest.length; i++) {
            System.out.print(arrayTest[i] + " ");
        }
    }
}
```

2. Vergleichen Sie die Geschwindigkeit von Mergesort mit dem klassischen Quicksort für verschiedenste Eingabedaten. Wählen Sie grosse, zufällige Folgen, damit Sie Geschwindigkeitsunterschiede messen können. Verwenden Sie dabei die Methode `System.currentTimeMillis()` zur Zeitmessung. Können Sie die Aussage aus der Vorlesung (Geschwindigkeitsvorteil von Quicksort: 10% - 50%) beobachten?

SIZE: 10'000'000

MAXVALUE: 1'000

	Run 1	Run 2	Run 3	Run 4	Run 5	Run 6	Run 7	Run 8	Run 9	Run 10	Average [ms]
Quicksort	895	894	881	890	889	894	898	898	885	886	891
Mergesort	1'898	1'865	1'868	1'871	1'865	1'870	1'866	1'863	1'865	1'867	1'870
	47%	48%	47%	48%	48%	48%	48%	48%	47%	47%	48%

```
import java.util.Random;
/**
 * Sortieralgorithmen vergleichen
 * @author Felix Rohrer
 * @version 1.0
 */
public class TimeTest
{
    // instance variables - replace the example below with your own
    private int[] numbers;
    private final static int SIZE     = 100000;
    private final static int MAXVALUE = 500;

    /**
     * Constructor for objects of class TimeTest
     */
    public TimeTest()
    {
    }

    /**
     * Compare Selectionsort, Quicksort and Mergesort
     */
    public void runTest()
    {
        // init new random numbers
        numbers = new int[SIZE];
        Random generator = new Random();
        for (int i = 0; i < numbers.length; i++) {
            numbers[i] = generator.nextInt(MAXVALUE);
        }

        long startTime;
        long stopTime;
        long elapsedTime;

        // Selectionsort
        Selectionsort sortA = new Selectionsort();
        startTime = System.currentTimeMillis();
        sortA.directSelect(numbers);
        stopTime = System.currentTimeMillis();
        elapsedTime = stopTime - startTime;
        System.out.println("Selectionsort: " + elapsedTime);

        // Quicksort
        Quicksort sortB = new Quicksort();
        startTime = System.currentTimeMillis();
        sortB.quickSort(numbers, 0, numbers.length-1);
        stopTime = System.currentTimeMillis();
        elapsedTime = stopTime - startTime;
        System.out.println("Quicksort: " + elapsedTime);

        // Mergesort
        Mergesort sortC = new Mergesort();
        startTime = System.currentTimeMillis();
        sortC.mergesort(numbers);
        stopTime = System.currentTimeMillis();
        elapsedTime = stopTime - startTime;
        System.out.println("Mergesort: " + elapsedTime);
    }
}
```

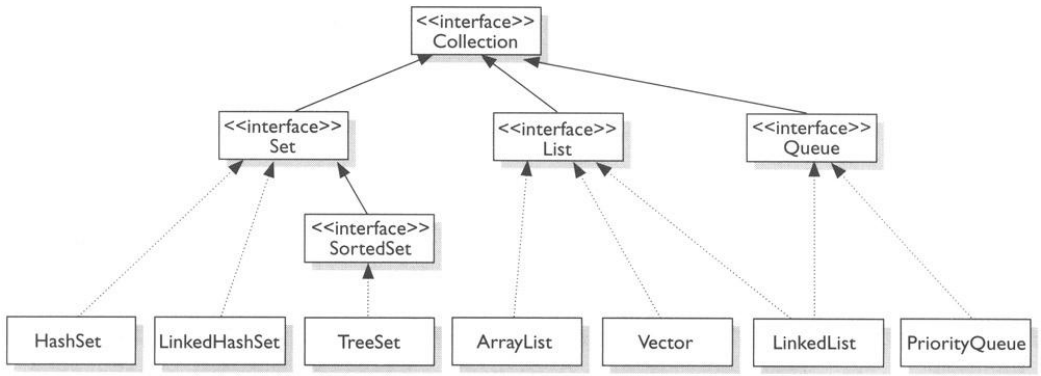
Aufgabe 5: Weiter Aufgaben aus dem Buch (optional)

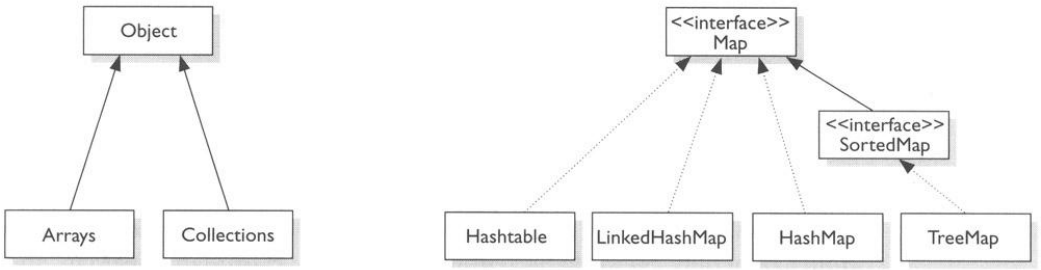
1. Bearbeiten Sie die Aufgaben 8.15, 8.16, 8.17 und 8.19 aus dem Buch.

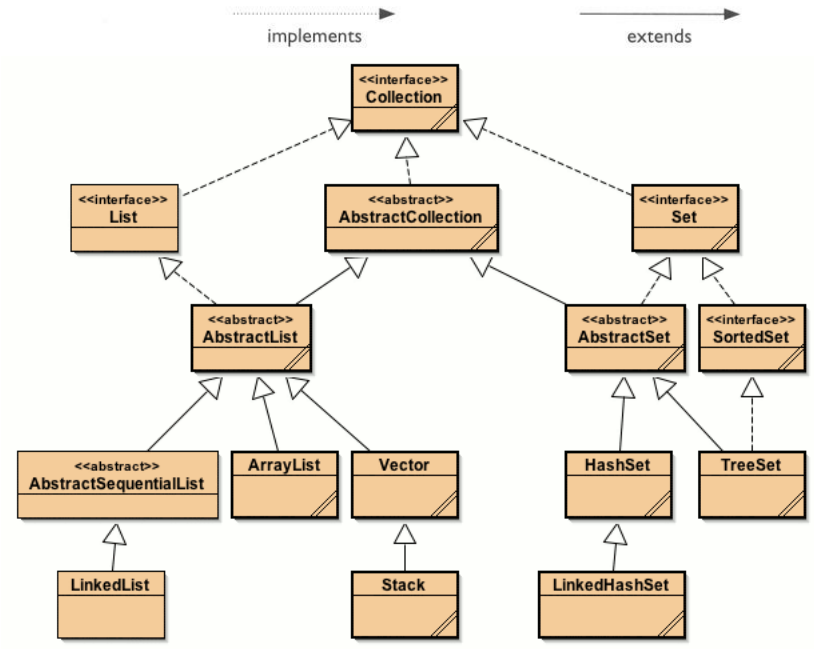
8.15:

http://docs.oracle.com/javase/1.5.0/docs/api/java/util/package-tree.html

http://www.programcreek.com/2009/02/the-interface-and-class-hierarchy-for-collections/

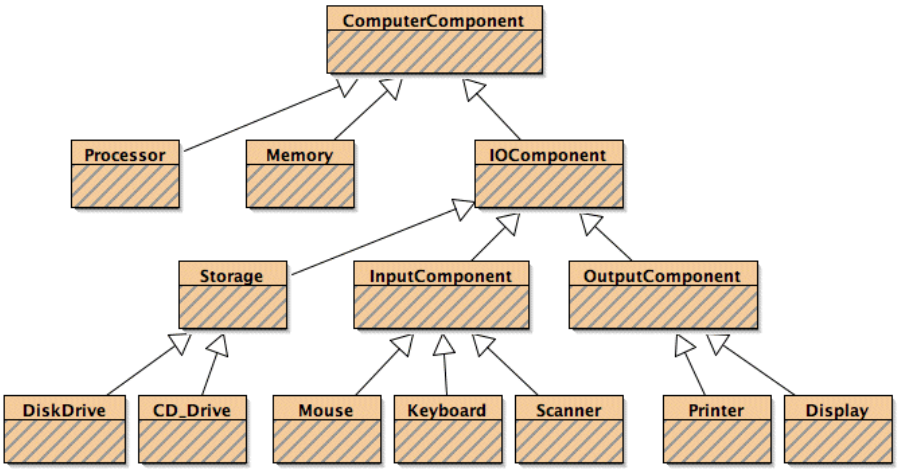






8.16:

8.17:



8.19:

http://docs.oracle.com/javase/1.5.0/docs/api/java/util/package-tree.html