39

Lucerne University of
Applied Sciences and Arts

HOCHSCHULE
LUZERN

Technik & Architektur

Programmieren 1

Test 2, HS 2011/12

M. Pouly, P. Sollberger, R. Diehl, H. Diethelm; Version 2

Nachname: Rohrer Vorname: Felix
(Bitte mit Druckbuchstaben schreiben)

Rahmenbedingungen:

1. Zeit: **90 Minuten**
2. Im Test können maximal **90 Punkte** erreicht werden. Jeder Aufgabe ist eine maximal erreichbare Punktezahl zugeordnet.
3. Schreiben Sie Ihren Namen und Vornamen auf dieses Blatt. Blätter ohne Namensangabe werden nicht bewertet.
4. Es handelt sich um einen schriftlichen Test ohne Einsatz des Computers oder elektronischer Hilfsmittel. Sie dürfen keine Unterlagen verwenden.
5. Sollte die Problemstellung Unklarheiten aufweisen, dürfen Sie eigene Annahmen treffen. Führen Sie diese in der Lösung auf.
6. Schreiben Sie verständlich und gut leserlich. Unleserliche Lösungen werden nicht berücksichtigt.
7. Benutzen Sie den Freiraum unter den Aufgaben für Ihre Lösung. Es werden keine Zusatzblätter akzeptiert!
8. Benutzen Sie den Anhang mit den API Spezifikationen.

Für die Korrektur (nicht ausfüllen!)

1	2	3	4	5	6	7	8	9	10	Punkte
9	10	10	5	12	9	5	15	10	5	90

9

Aufgabe 1: Schleifen (9 Punkte)

a) Beantworten Sie folgende Fragen. (3 Punkte)

1. Welche Schleifenart setzen Sie ein, wenn die Schleife unendlich oft durchlaufen werden soll?

   while ✓

   Schreiben Sie den Code für die von Ihnen gewählte Schleifenart, welche unendlich viele Schleifendurchläufe macht.

   while (true) { ✓
   } ...

2. Welche Schleifenart setzen Sie ein, wenn Sie eine Eingabe im Programm überprüfen wollen?

   do-while ✓

   Wie muss das Resultat der Bedingung Ihrer gewählten Schleifenart sein, damit die Schleife nicht mehr durchlaufen wird?

   false ✓

3. Welche Schleifenart setzen Sie ein, wenn die Anzahl Schleifendurchläufe bekannt ist?

   for ✓

   Wie muss das Resultat der Bedingung Ihrer gewählten Schleifenart sein, damit es einen Schleifendurchlauf gibt?

   true ✓

b) Schleifen, und das gilt insbesondere für `for`-Schleifen, können geschachtelt werden, so wie der folgende Codeausschnitt zeigt:

```
for ( int i = 5; i > 0; i-- ) {
   for ( int j = 0; j < i; j+=2 ) {
      System.out.print("+");
   }
   System.out.println();
}
```

Was sieht man auf dem Bildschirm, wenn der obige Code ausgeführt wurde? (3 Punkte)

```
+++
++
++ ✓
+
+
```

c) Ersetzen Sie die <u>äussere Schlaufe</u> aus der Teilaufgabe b) durch eine `do-while` Schlaufe. Die innere Schlaufe dürfen Sie mit `//for ...` abkürzen. (3 Punkte)

```
int i = 5; ✓
do {
   //for ...
   i--; ✓
} while (i>0); ✓
```

Aufgabe 2: Arrays (10 Punkte) 10

a) Gegeben ist die Klasse `KleinerGauss` mit ein paar Lücken.

Die Klasse `KleinerGauss` enthält ein Attribut `gauss` vom Typ Integer Array. Der Konstruktor erzeugt das Array mit der Grösse, die durch den formalen Parameter `n` gegeben wird. Anschliessend initialisiert er die Array Elemente mit der Summe zum jeweiligen Index (i := 1+2+3+...+i).

Ergänzen Sie die leeren Zeilen mit Array Definitionen, bzw. Anweisungen. (5 Punkte) 5

```
public class KleinerGauss
{
    private int[] gauss; ✓
    public void KleinerGauss(int n)
    {
        gauss = new int[n]; ✓
        for (int i = 0; i < n; i++) {
            gauss[i] = i*(i+1)/2; ✓
        }
    }

    private int sumAt(int x)
    {
        return gauss[x]; ✓
    }
}
```

b) Vervollständigen Sie die Methode `daysPerMonth`. Als aktuellen Parameter übergibt man der Methode einen Monat von 1 bis 12, als Resultat liefert die Methode die Anzahl Tage dieses Monats. Schaltjahre müssen nicht berücksichtigt werden. (5 Punkte)

Implementieren Sie die Methode mit Hilfe eines Integer Array. Sie benötigen dazu <u>nur zwei Zeilen</u>! Falsche aktuelle Parameter müssen Sie nicht abfangen.

```
public int daysPerMonth(int month)
{
    int[] days = {31,28,31,30,31,30,31,31,30,31,30,31};
    return days(month-1);
}
```

✓ 5

Aufgabe 3: ArrayList (10 Punkte) 10

Schreiben Sie unter die folgenden Instruktionen die entsprechenden Code Segmente der Klasse `Queue`.
Die Queue muss nach dem FIFO Prinzip (first in – first out) funktionieren
(siehe Anhang API Dokumenation):

a. Deklarieren Sie ein Attribut `queue` vom Typ `ArrayList`, welches nur Objekte der Klasse `String` beinhaltet. (1 Punkt)

```
private ArrayList<String> queue;
```
✓ 1

b. Initialisieren Sie das Attribut `queue`. (1 Punkt)

```
queue = new ArrayList<String>();
```
✓ 1

c. Die Methode `put` soll ein Element nach dem FIFO Prinzip in die Queue legen. (2 Punkte)

```
public void put(String data)
{
    queue.add(data);
}
```
✓ 2

d. Die Methode `get` soll ein Element nach dem FIFO Prinzip aus der Queue entfernen. (2 Punkte)

```
public String get()
{
    return queue.remove(0);
}
```
✓ 2

e. Die Methode `hasElements` soll angeben, ob Elemente in der Queue sind. (1 Punkt)

```
public boolean hasElements()
{
    return !queue.isEmpty();
}
```
✓ 1

f. Die Methode `listElements` soll alle Elemente der Queue auf das Terminal (Bildschirm) ausgeben. Iterieren Sie mit einem Iterator durch die Queue. Die Reihenfolge der Elemente bei der Ausgabe spielt keine Rolle. (3 Punkte)

```
public void listElements()
{
    Iterator<String> itr = queue.iterator();
    while (itr.hasNext()) {
        System.out.println(itr.next());
    }
}
```
✓ 3

Aufgabe 4: Javadoc (5 Punkte)

5

Für eine Ernährungsberatungsfirma soll ein Kalorienrechner programmiert werden, der den durchschnittlichen Kalorienverbrauch (Einheit: kcal) einer Person berechnet.
Die entsprechende Methode hat folgenden Methodenkopf:

```
public int calculateCalories(boolean female, int weight, int size, short age)
```

Das Gewicht soll in Kilogramm, die Körpergrösse in Zentimeter und das Alter in Jahren angegeben werden. Erstellen Sie für diese Methode eine Javadoc Source-Code Dokumentation (in Deutsch) mit

- einer aussagekräftigen (!) Methodenbeschreibung
- Dokumentation des Rückgabewertes
- Dokumentation aller Parameter

Hinweis: Ihr Kommentar muss alle Informationen liefern, so dass die Methode von einem Programmierer benutzt werden kann. (5 Punkte)

```
/** ✓
* Die Methode berechnet den durchschnittlichen Kalorienverbrauch
* von einer Person. ✓
* @param female   Geschlecht: true=weiblich, false=männlich
* @param weight   Gewicht der Person in kg
* @param size     Körpergrösse der Person in cm
* @param age      Alter der Person in Jahren
* @return durchschnittlichen Kalorienverbrauch in kcal ✓
*/
```

Aufgabe 5: HashMap und Generics, Strings (12 Punkte)

12

Bei der Klasse `Directory` wird eine HashMap zum Speichern von Telefonnummern (Klasse `PhoneNumber`) und deren Besitzer (Klasse `Subscriber`) verwendet.

1. Bei einem ersten Versuch meldet der Compiler folgenden Fehler:
   `cannot find symbol - class HashMap`
   Ergänzen Sie untenstehenden Code mit den Anweisungen, damit die aus der Klassenbibliothek verwendeten Klassen verwendet werden können (drei Anweisungen). (3 Punkte)

   3

2. Wird diese Klasse kompiliert, werden folgende Warnungen und Fehler angezeigt:
   `Note: Directory.java uses unchecked or unsove operations.`
   `Note: Recompile with -Xlint:unchecked for details.`
   und auf Zeile 28
   `incompatible types`
   Korrigieren Sie untenstehenden Code unmissverständlich (an vier Stellen). (7 Punkte)

   7

3. Beim Testen der Methode `listSameName(...)` stellen Sie fest, dass nie eine Ausgabe erfolgt (auch wenn der entsprechende Name in der `HashMap` vorhanden ist).
   Korrigieren Sie auch dazu den untenstehenden Code unmissverständlich. (2 Punkte)

   2

```
1   import java.util.HashMap;
2   import java.util.Set;
3   import java.util.Iterator;
4
5   public class Directory
6   {
7       private HashMap<PhoneNumber, Subscriber> directory;
8
9       public Directory()
10      {
11          directory = new HashMap<PhoneNumber, Subscriber>();
12      }
13
14      public void addSubscriber(Subscriber s, PhoneNumber n)
15      {
16          directory.put(n, s);
17      }
18
19      public boolean isNumberUsed(PhoneNumber aNumber)
20      {
21          return directory.containsKey(aNumber);
22      }
23
24      public void listSameName(String aName)
25      {
26          Set<PhoneNumber> numbers = directory.keySet();
27
28          Iterator<PhoneNumber> itr = numbers.iterator();
29
30          while(itr.hasNext()) {
31
32              PhoneNumber k = itr.next();
33              Subscriber  s = directory.get(k);
34              String directoryName = s.getName();
35
36
37              if (directoryName == aName) {     [struck through]
38
39              if (directoryName.equals(aName)) {
40
41                  String txt = k.getNumber() + ": " + directoryName;
42                  System.out.println(txt);
43              }
44          }
45      }
46  }
47
48
```

Aufgabe 6: Klassendesign und Zugriffsmodifizierer (9 Punkte)

9

Für einen Ballonwettbewerb haben Sie die auf der nächsten Seite abgebildete Klasse `Balloon` erhalten.

1. Um was für ein Element handelt es sich bei `idCounter` (Zeile 7)? (1 Punkt)~~Klassenvariable~~Welchen Zugriffsmodifizierer sollten Sie bei `idCounter` verwenden? (1 Punkt)

   Klassenvariable | 2. private ✓

   2

3. Erläutern Sie kurz, was mit dem Element `idCounter` bewirkt wird. (1 Punkt)

   Jedem Objekt von dieser Klasse wird eine eindeutige (aufsteigende) id, nummer zugeteilt. ✓

   1

4. Nennen Sie ein Beispiel im Code, wo das Konzept des Information-Hidings schlecht angewendet wurde. Begründen Sie kurz Ihre Wahl. (2 Punkte)

   Beim Feld owner.
   Dieses sollte private sein. Gesetzt wird es im Konstruktor, abgefragt kann es via getOwnerName() werden. ✓

   2

5. Welche Methode verletzt das Konzept der starken Kohäsion?
   Begründen Sie kurz Ihre Wahl. (2 Punkte)

   setStartTimeAndPrintOwner() ✓
   Diese Methode führt zwei unterschiedliche Arbeiten aus. ✓

   2

6. Zählen Sie die Klassen auf, mit denen die Klasse `Balloon` gekoppelt ist. (2 Punkte)

   - Address
   - Color
   - Date
   - Integer
   - String

   ✓

   2

```
1   public class Balloon
2   {
3       public  Address owner;
4       private Color   color;
5       private Date    startTime;
6       private int     id;
7       static  int     idCounter = 1;
8
9       public Balloon(Address anOwner, Color aColor)
10      {
11          owner = anOwner;
12          color = aColor;
13          id = idCounter++;
14      }
15
16      public Integer getId()
17      {
18          return new Integer(id);
19      }
20
21      public String getOwnerName()
22      {
23          return owner.getName();
24      }
25
26      public void setStartTimeAndPrintOwner()
27      {
28          startTime = new Date();
29          System.out.println(owner.getName());
30      }
31
32      public double calculateSpeed(Date landingTime, double distance)
33      {
34      ...
35      }
36  }
```

Aufgabe 7: Komplexität (5 Punkte)

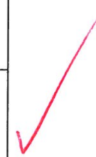
5

Gegeben sind die folgenden Funktionen. Tragen Sie in der Tabelle die Ordnung der Funktionen ein.

1. Geben Sie zu jeder Funktion deren Ordnung an: (3 Punkte)

3

Funktion	Ordnung
$f_1(n) = 1000 + 0.01 \cdot n$	$O(n)$
$f_2(n) = \log_2(n) + 2 \cdot n^2$	$O(n^2)$
$f_3(n) = n \cdot (77 + 4 \cdot n + 0.1 \cdot n^2)$	$O(n^3)$
$f_4(n) = 5000 + 4!$	$O(1)$
$f_5(n) = 3n \cdot \log(n) + 4711 \cdot n$	$O(n \cdot \log(n))$
$f_6(n) = 100 \cdot n + n^3 + 2^{0.01 \cdot n}$	$O(2^n)$



2. Bestimmen Sie bezüglich Laufzeit *die Ordnung* der folgenden Methode doA(…): (2 Punkte)

2

```
public static void doA(int n)
{
    for (int i=0; i < n/2; i++) {      // n/2
        doB();
    }
    for (int j=0; j < 100; j++) {      // 1
        doC();
    }
}
```

→ $O(n)$

Aufgabe 8: Rekursion (15 Punkte)

15

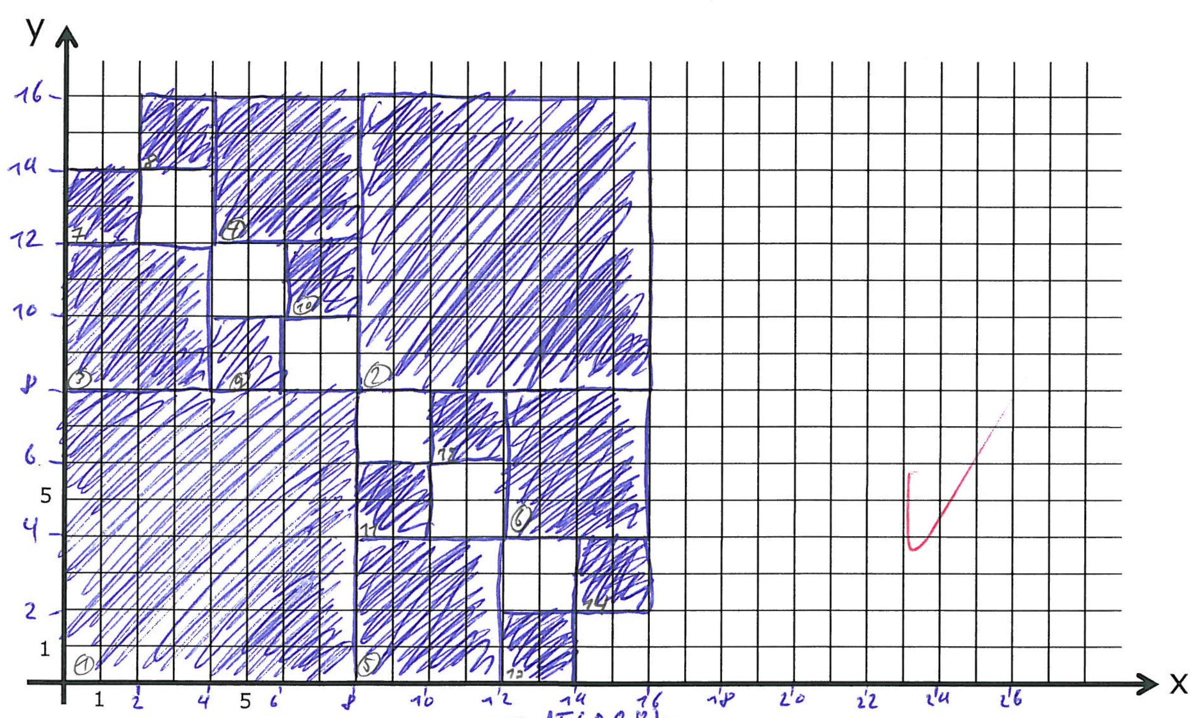
1. Zeichnen Sie unten die Figur genau auf, welche beim Aufruf von "`java Rekursion`" gezeichnet wird. (11 Punkte)
2. Wie oft wird oben die Methode drawFigure(…) insgesamt aufgerufen? Wir empfehlen Ihnen, für die Beantwortung der Frage die Aufrufe "baumförmig" von oben nach unten aufzuzeichnen. (4 Punkte)

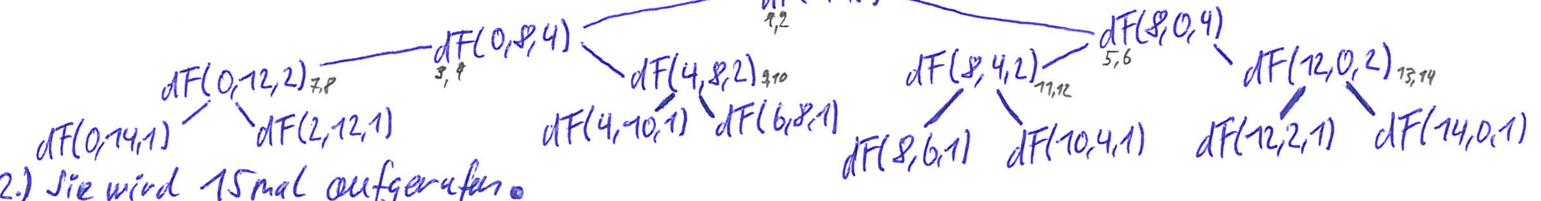
15x

```
public class Rekursion
{
    public static void drawFigure(int x, int y, int s)
    {
        if(s > 1) {
            drawBlackSquare(x, y, s);
            drawBlackSquare(x+s, y+s, s);
            drawFigure(x, y+s, s/2);
            drawFigure(x+s, y, s/2);
        {
    }

    public static void drawBlackSquare(int x, int y, int s)
    {
        // Draws a filled black square with side s at
        // the position x/y (lower-left corner of the square).
        ...
    }

    public static void main(String[] args)
    {
        drawFigure(0, 0, 8);
    }
}
```





Aufgabe 9: Backtracking (10 Punkte)

```
// Backtracking-Algorithmus in Java-Pseudocode

boolean findeLoesung(int index, Lsg loesung, ...)
{
    while(Es gibt noch neue Teil-Lösungsschritte) {
        Wähle einen neuen Teil-Lösungsschritt schritt;
        if(schritt ist gültig) {
            Erweitere loesung um schritt;
            if(loesung noch nicht vollständig) {
                if(findeLoesung(index+1, loesung, ...)) {
                    return ???;  // (1)
                }
                else {
                    Mache schritt rückgängig;
                }
            }
            else {
                return ???;  // (2)
            }
        }
    }
    return ???;  // (3)
}
```

1. Geben Sie an, was anstelle der "Drei Fragezeichen" jeweils stehen muss: (3 Punkte)

(1) true // möglicher Lösungsschritt

(2) true // Lösung vollständig

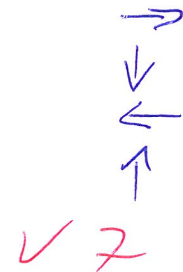
(3) false // keine neue Teil-Lösungsschritte

✓ 3

2. Suchen Sie mit Hilfe des Backtracking-Algorithmus einen Weg vom Feld "Start" hinaus aus dem Labyrinth, d.h. zum Feld "Ziel". Der Algorithmus wählt den nächsten Schritt immer nach folgendem **Schema: nach rechts, nach unten, nach links, nach oben**. (7 Punkte)

   - Geben Sie den besuchten Feldern eine fortlaufende Nummer (beginnend bei 1).
   - Streichen Sie beim Zurückgehen (Backtracking) das entsprechende Feld durch.
   - Schlussendlich führt der Weg entlang der Nummerierung 1, 2, 3, … zum Ziel.

			11 Ziel	10	
1 Start	2			9	8
	3	4	5	6	7
				~~9~~	~~8~~



Aufgabe 10: Sortieren (5 Punkte)

1. Sortieren Sie das folgende Array mittels direktem Vertauschen (bubble sort). Bekanntlich durchläuft dieser Algorithmus das Array mehrfach. Geben Sie den Inhalt der Zellen nach jedem vollständigen Durchlauf an. (2 Punkte)

8	5	6	10	1	3

5	6	8	1	3	10

5	6	1	3	8	10

5	1	3	6	8	10

1	3	5	6	8	10

2

2. Eine grosse Firma unterhält eine virtuelle Kartei mit allen Mitarbeitern. Die Karteikarten sind nach Nachname geordnet. Manchmal (z.B. bei Heirat) muss ein Nachname geändert werden. Der Azubi ändert dann die entsprechende Karteikarte. Er darf sie jedoch nicht gleich am neuen Ort in die Kartei einfügen, denn der Sekretariatsleiter will jeden Abend alle Änderungen persönlich überprüfen. Dann startet der Sekretariatsleiter einen Sortierprozess zur Wiederherstellung der alphabetischen Ordnung. Natürlich betreibt der Sekretariatsleiter diesen hohen Kontrollaufwand nur, weil solche Änderung erfahrungsgemäss eher selten vorkommen.

   a. Welche Komplexität hat der Sortieralgorithmus Quicksort ? (1 Punkt) $O(n \cdot \log(n))$

   b. Aufgrund dieser Komplexität ist Quicksort oftmals die beste Wahl als Sortieralgorithmus. Nicht jedoch in dieser speziellen Anwendung. Warum ist Quicksort hier ungeeignet? Geben Sie eine kurze Antwort, indem Sie sich auf die Komplexität von Quicksort in diesem Spezialfall beziehen. (1 Punkt)
   Quicksort ist bei fast sortierten List ≈ Quadratisch $O(n^2)$.

   c. Begründen Sie, warum direktes Einfügen (insertion sort) hier die bessere Wahl ist. Geben Sie eine kurze Antwort, indem Sie sich auf die Komplexität von insertion sort in diesem Spezialfall beziehen. (1 Punkt)

---- Ende des Tests ----

Insertion sort ist bei fast sortierten Listen linear $O(n)$.

5

Appendix API Specifications

Class ArrayList<E>

Constructor Summary

ArrayList()

Method Summary

boolean	**add**(E o) Appends the specified element to the end of this list.
void	**add**(int index, E element) Inserts the specified element at the specified position in this list.
void	**clear**() Removes all of the elements from this list.
boolean	**contains**(Object elem) Returns true if this list contains the specified element.
E	**get**(int index) Returns the element at the specified position in this list.
boolean	**isEmpty**() Tests if this list has no elements.
Iterator<E>	**iterator**() Returns an iterator over the elements in this set.
E	**remove**(int index) Removes the element at the specified position in this list.
E	**set**(int index, E element) Replaces the element at the specified position in this list with the specified element.
int	**size**() Returns the number of elements in this list.

Interface Iterator<E>

Method Summary

boolean	**hasNext**() Returns true if the iteration has more elements.
E	**next**() Returns the next element in the iteration.
void	**remove**() Removes from the underlying collection the last element returned by the iterator (optional operation).

Class HashMap<V,K>

Constructor Summary
HashMap()

Method Summary	
void	**clear()** Removes all mappings from this map.
Object	**clone()** Returns a shallow copy of this HashMap instance: the keys and values themselves are not cloned.
boolean	**containsKey(**Object key**)** Returns true if this map contains a mapping for the specified key.
boolean	**containsValue(**Object value**)** Returns true if this map maps one or more keys to the specified value.
Set<Map.Entry<K,V>>	**entrySet()** Returns a collection view of the mappings contained in this map.
V	**get(**Object key**)** Returns the value to which the specified key is mapped in this identity hash map, or null if the map contains no mapping for this key.
boolean	**isEmpty()** Returns true if this map contains no key-value mappings.
Set<K>	**keySet()** Returns a set view of the keys contained in this map.
V	**put(**K key, V value**)** Associates the specified value with the specified key in this map.
void	**putAll(**Map<? extends K,? extends V> m**)** Copies all of the mappings from the specified map to this map These mappings will replace any mappings that this map had for any of the keys currently in the specified map.
V	**remove(**Object key**)** Removes the mapping for this key from this map if present.
int	**size()** Returns the number of key-value mappings in this map.
Collection<V>	**values()** Returns a collection view of the values contained in this map.

Interface Set<E>

Method Summary	
boolean	**add**(E e) Adds the specified element to this set if it is not already present (optional operation).
boolean	**addAll**(Collection<? extends E> c) Adds all of the elements in the specified collection to this set if they're not already present (optional operation).
void	**clear**() Removes all of the elements from this set (optional operation).
boolean	**contains**(Object o) Returns true if this set contains the specified element.
boolean	**containsAll**(Collection<?> c) Returns true if this set contains all of the elements of the specified collection.
boolean	**equals**(Object o) Compares the specified object with this set for equality.
int	**hashCode**() Returns the hash code value for this set.
boolean	**isEmpty**() Returns true if this set contains no elements.
`Iterator<E>`	**iterator**() Returns an iterator over the elements in this set.
boolean	**remove**(`Object o`) Removes the specified element from this set if it is present (optional operation).
boolean	**removeAll**(`Collection<?> c`) Removes from this set all of its elements that are contained in the specified collection (optional operation).
boolean	**retainAll**(`Collection<?> c`) Retains only the elements in this set that are contained in the specified collection (optional operation).
int	**size**() Returns the number of elements in this set (its cardinality).
`Object[]`	**toArray**() Returns an array containing all of the elements in this set.