Übungen zu Kapitel 19

19.1 Integer: Erstellen Sie die Klasse `MyInteger` mit den im Text beschriebenen Konstruktoren, Methoden und Attributen und testen Sie das korrekte Verhalten dieser Klasse an einigen Beispielen.

Dabei ist die private Methode `string2int()` so zu erweitern, dass auch negative Zahlen als `String` mitgegeben werden können und dass bei Ziffernfolgen, die zu große oder zu kleine Zahlen darstellen, eine `NumberFormatException` ausgelöst wird. Diese Ausnahme soll auch bei übergebenem Leerstring (`""`) ausgelöst werden:

```
new MyInteger( "77777777777777" ) → NumberFormatException
new MyInteger( "" )               → NumberFormatException
```

19.2 Autoboxing: Führen Sie das folgende Coding aus und erklären Sie das Phänomen:

```
Integer i1 = 200;
Integer i2 = 200;
System.out.println( i1 <= i2 );
System.out.println( i2 <= i1 );
System.out.println( i1 == i2 );
```

Bei einem Compiler-Fehler sollten Sie die verwendete Java-Version überprüfen.

19.3 BigInteger: Vergleichen Sie die Laufzeit der Addition zwischen primitiven Datentypen mit der zwischen Objekten der Klasse `BigInteger`, indem jeweils in einer Schleife der Wert einer `long`-Variablen bzw. eines `BigInteger`-Objektes um 1 erhöht wird. Wählen Sie die Anzahl der Schleifendurchläufe so groß, dass Laufzeiten sinnvoll gemessen und ausgegeben werden können.

19.4 BigBruch:

Erstellen Sie in Analogie zu `Bruch` eine Klasse `BigBruch`, die für Zähler und Nenner `BigInteger`-Attribute verwendet und daher das Problem des Überlaufs umgeht. Erstellen Sie auch eine Testklasse:

```
import java.math.*;
public class BigBruch {
   private BigInteger zaehler;
   private BigInteger nenner;

   public BigBruch( String z, String n ) {
      zaehler = new BigInteger( z );
      nenner = new BigInteger( n );
      if( nenner.equals( BigInteger.ZERO ) )
         throw new IllegalArgumentException( "Nenner ist 0" );
      kuerze();
   }

   public BigBruch add( BigBruch b )
   public BigBruch negate()
   public BigBruch subtract( BigBruch b )
   public BigBruch multiply( BigBruch b )
   public BigBruch divide( BigBruch b )
   public BigBruch pow( int p )              // Potenz: this-hoch-p
   public BigBruch inv()                     // Kehrwert: 1-durch-this
   public boolean equals( Object obj )
   public int hashCode( Object obj )
   public int compareTo( Object obj )
   public String toString()
}
```

Die Algorithmen der einzelnen Methoden können in der Regel direkt aus der bekannten Klasse übernommen werden, wobei die jeweiligen arithmetischen Operationen entsprechend zu modifizieren sind:

Beim Nenner-Gleich-Null-Test verwenden wir - wie bei Objekten üblich - die Methode `equals()` und vergleichen mit der Konstante `BigInteger.ZERO`. Anstelle dieser Konstanten hätten wir auch den Ausdruck `new BigInteger("0")` verwenden können. Die Methode `kuerze()` ist analog zu der entsprechenden Methode in der "kleinen" Klasse `Bruch` zu definieren:

```
private void kuerze( ) {
   BigInteger gcd = zaehler.gcd( nenner );
   zaehler = zaehler.divide( gcd );
   nenner = nenner.divide( gcd );
   if( nenner.signum() < 0 ) {
      nenner = nenner.negate();
      zaehler = zaehler.negate();
}
```

Die im Testausdruck verwendete Methode `signum()` entspricht der mathematischen Signum-Funktion, die -1, 0 oder 1 zurückgibt, je nachdem, ob das Objekt kleiner, gleich oder größer 0 ist. Die diesem Ausdruck folgende Passage stellt sicher, dass allein am Vorzeichen des Zählers auszumachen ist, ob der Bruch negativ ist.

Es ist hilfreich, einen Konstruktor mit `BigInteger`-Argumenten zu definieren:

```
public BigBruch( BigInteger zaehler, BigInteger nenner ) {
   this.zaehler = zaehler;      this.nenner = nenner;
   if( nenner.equals( BigInteger.ZERO ) )
      throw new IllegalArgumentException( "Nenner ist 0" );
   kuerze();
}
```

Beispielhaft geben wir eine mögliche Definitin der Addition an:

```
public BigBruch add( BigBruch b ) {
  return new BigBruch(
    // neuer zaehler
    zaehler.multiply(b.nenner).add(nenner.multiply(b.zaehler)),
    // neuer nenner
    nenner.multiply(b.nenner)
  );
}
```

19.5 Triviale equals()-Implementierungen: Machen Sie sich bewusst, dass die `equals()`-Implementierung von `Object`

```
public boolean equals( Object obj ) {
   return ( this == obj );
}
```

aber auch die folgende Implementierung die Bedingungen der Bivalenz, Reflexivität, Symmetrie und Transitivität erfüllen:

```
public boolean equals( Object obj ) {
   return true;
}
```

Beide Implementierungen sind nicht sehr sinnvoll und stellen extreme Positionen dar. Die erste betrachtet unterschiedliche Objekte stets inhaltlich verschieden (es werden also nur die Referenzen als gleich betrachtet, die das gleiche Objekt bezeichnen), die zweite identifiziert alle Objekte miteinander.

19.6 equals() in Hierarchien: Gehen Sie davon aus, dass Mitarbeiter ohne Vorname angelegt werden können (in diesem Fall hat das Attribut `vorname` den Wert `null`). Modifizieren Sie die Methode `equals()` in der Klasse `Mitarbeiter` folgendermaßen:

Variante a): Stimmen zwei Mitarbeiter-Instanzen in den Attributen Personalnummer und Name überein und haben beiden keinen Vornamen, dann sind sie gleich; hat nur eine einen Vornamen, dann sind sie unterschiedlich.

Variante b): Stimmen zwei Mitarbeiter-Instanzen in den Attributen Personalnummer und Name überein und hat mindestens eine keinen Vornamen, dann sind sie gleich. Vorsicht: In diesem Fall beschreibt `equals()` keine Äquivalenzrelation. Welche Bedingung ist verletzt?

19.7 equals() in Hierarchien: Machen Sie sich klar, dass die Code-Zeilen

```
if( obj == null ) return false;
if( getClass() != obj.getClass() ) return false;
```

in der `equals()`-Implementierung der Klasse PunktSymbol überflüssig sind. Die beiden Tests müssen nur in der "obersten" Superklasse durchgeführt werden, die `equals()` überschreibt, also in der Regel in der direkt von `Object` abgeleiteten Klasse. Die gegebenenfalls mehrfache rekursive Delegation `super.equals()` stellt sicher, dass das Testcoding vor allen Casts und Attributenvergleichen durchgeführt wird. Aus Performancegründen sollte der Alias-Test jedoch in jeder Klasse formuliert werden.

19.8 equals() in Hierarchien: Erstellen Sie für folgendes Szenario Klassen mit den Methoden `toString()` und `equals()` sowie einer ausführbaren Testklasse: Eine Bank ist eindeutig durch die Bankleitzahl gekennzeichnet und hat zusätzlich einen Namen. Ein Konto ist bestimmt durch die Bank, zu der es gehört, und einer Kontonummer. Zusätzlich besitzt es einen Kontoinhaber mit Namen, Vornamen und Personalausweisnummer (durch dieses Attribut ist ein Kontoinhaber eindeutig bestimmt, dh stimmen Kontoinhaber in der Personalausweisnummer überein, dann sind sie gleich). Verwenden Sie bei der Bankleitzahl ein Attribut vom Typ `String`.

19.9 equals() und Groß-/Kleinschreibung: In einigen Anwendungen wird bei Benutzernamen Groß-/Kleinschreibung nicht unterschieden. Erstellen Sie eine Klasse `Benutzer` mit den angegebenen Methoden, die dies berücksichtigen:

```
public class Benutzer {
   private String s;
   public Benutzer( String s ) {...}
   public boolean equals( Object obj ) {...}
   public int hashCode( ) {...}
   public String toString(){...}    //Ausgabe in Großbuchstaben
}
```

Ein Benutzername besteht aus mindestens 6 und maximal 12 Buchstaben und darf keine Leerzeichen beinhalten. Implementieren Sie diese Bedingung im Konstruktor, der gegebenenfalls eine `IllegalArgumentException` auslöst.

Tipp: Betrachten Sie die Methoden `toUpperCase()` oder `equalsIgnoreCase()` der Klasse `String`. Vorsicht bei der Signatur!

19.10 instanceof anstelle getClass(): In der Literatur und im Internet findet man `equals()`-Implementierungen, bei denen anstelle des vorgestellten Vergleichbarkeitstests

```
if( getClass() != obj.getClass() ) return false;
```

der folgende mit dem Schlüsselwort `instanceof` formulierte Test verwendet wird:

```
if( !( obj instanceof Klassenname ) ) return false;
```

Hier bezeichnet *`Klassenname`* den Namen der Klasse, in der `equals()` definiert wird. Ändern Sie `equals()` im Beispiel `Punkt` und `PunktSymbol`, so dass in beiden Methoden dieser `instanceof`-Test verwendet wird. Zeigen Sie, dass dann die Symmetriebedingung verletzt ist, indem Sie einen Punkt `p` und ein Punktsymbol `s` mit gleichen Ortskoordinaten erzeugen und die folgenden Ausdrücke vergleichen:

```
p.equals( s )                    s.equals( p )
```

Zu weiteren Details über diese Implementierungsalternative siehe [KL02a].

19.11 StringBuffer und equals(): In der Einleitung zu 19.4.1 wurde dargestellt, dass die Semantik von `equals()` in `StringBuffer` nicht mit der in `String` übereinstimmt. Der Versuch, eine Klasse `MyStringBuffer` von `StringBuffer` abzuleiten und `equals()` zu überschreiben, wird scheitern, da `StringBuffer` eine finale Klasse ist. Hier bietet sich folgende Alternative an:

Erstellen Sie eine Klasse `MyStringBuffer` als ***Wrapper*** von `StringBuffer`, deren Instanzen über ein privates `StringBuffer`-Attribut verfügen und damit `StringBuffer`-Objekte repräsentieren. Formulieren Sie entsprechende Konstruktoren und öffentliche Methoden, die - mit Ausnahme von `equals()` - semantisch mit den ursprünglichen Methoden übereinstimmen. Erstellen Sie die Methode `equals()` wie gewünscht und formulieren Sie einige Testfälle.

```
public MyStringBuffer {
   private StringBuffer sb;
   public MyStringBuffer() {
      sb = new StringBuffer();
   }
   ...
   public char charAt( int index ) {
      return sb.charAt( index );
   }
}
```

Da es sehr lästig ist, alle Konstruktoren und Methoden von `StringBuffer` zu formulieren, können Sie sich auf folgende exemplarischen Fälle beschränken:

```
//Konstruktor
public MyStringBuffer( String s )

//Methoden analog zu StringBuffer
public MyStringBuffer append( String s )
public MyStringBuffer append( MyStringBuffer s )
public MyStringBuffer deleteCharAt( int index )
public void setCharAt()

//equals() in gewünschter Semantik
public boolean equals( Object obj )
//hashCode() konsistent zu equals()
public int hashCode()
```

19.12 Triviale Methode hashCode(): Begründen Sie, warum die folgende Methode trivialerweise den ***equals-hashCode-Kontrakt*** erfüllt:

```
public int hashCode() {  return 23;  }
```

Wieviele ***buckets*** besitzt ein auf dieser Methode basierender Container?

19.13 hashCode als Attribut: Falls bei Instanzen von ***immutable*** Klassen der Hash-Code sehr häufig verwendet wird, empfiehlt sich, ein zusätzliches privates `int`-Attribut zu verwenden und den Hash-Code einmalig beim Erzeugen zu berechnen. Implementieren Sie in der folgenden Klasse die angegebenen Methoden und wenden Sie bei `hashCode()` dieses Verfahren an. Führen Sie einige Tests durch.

```
public class Immutable {
   private int jahr;
   private String name;
   private boolean mw;
   public Immutable( int jahr, String name, boolean mw ) {
      this.jahr=jahr; this.name=name; this.mw=mw;
      ...
   }
   public boolean equals( Object obj ) {
      ...
   }
   public int hashCode() {
      ...
```

```
    }
}
```

19.14 Verletzung des equals-hashCode-Kontrakts: Verwenden Sie für diese Aufgabe zunächst die bereits bekannte Klasse `Punkt` mit entsprechenden Konstruktoren und semantisch korrektem `equals()`, jedoch ohne `hashCode()` zu überschreiben. Das ausführbare Programm erzeugt eine `HashSet`, der ein Punkt `p1` hinzugefügt wird. Die anschließende Suche mit dem gleichen Punkt `p2` ist - wie der Output zeigt - erfolglos.

```
import java.util.HashSet;
public class HashTest {
   public static void main( String[] args ) {
      HashSet hs = new HashSet();
      Punkt p1 = new Punkt( 1, 2 );
      hs.add( p1 );
      Punkt p2 = new Punkt( 1, 2 );
      System.out.println( hs.contains( p2 ) );
   }
}
```

Führen Sie zunächst das Programm aus und erklären Sie das Phänomen. Erstellen Sie anschließend in `Punkt` eine Methode `hashCode()`, die immer den Wert 0 zurückgibt und testen Sie erneut. Warum verhält sich das Programm nun korrekt? Formulieren Sie danach eine vernünftige Methode zur Berechnung des Hash-Codes.

19.15 Sortieren eines Arrays: Ändern Sie die Methode `compareTo()` in der Klasse `Mitarbeiter`, so dass diese das folgende Sortierkriterium implementiert.

- Mitarbeiter mit kürzeren Nachnamen sollen grundsätzlich vor Mitarbeitern mit längeren Nachnamen sortiert werden.
- Ist die Länge der Nachnamen gleich, wird in der Priorität Vorname – Nachname – Personalnummer sortiert, wobei das Kriterium "Vorname" in umgekehrter lexikografischer Sortierreihenfolge zu berücksichtigen ist. Dies ist zugegebenermaßen ein eher pathologisches Sortierkriterium.

Testen Sie die Semantik der Methode mittels `Arrays.sort()`.

***19.16 Sortieren mit* `Comparator`**

Um alternative Sortierkriterien zu implementieren verwendet man das Interface `Comparator` und gibt beim Sortieraufruf eine `Comparator`-Instanz als Kriterium mit. Wir modifizieren das Beispiel aus dem Kapitel wie folgt:

```
import java.util.*;
public static void main( String[] args ) {
   Mitarbeiter[] m = new Mitarbeiter[3];
   m[0]=new Mitarbeiter(6, "Meier", "Rolf");
   m[1]=new Mitarbeiter(3, "Meier", "Rolf");
   m[2]=new Mitarbeiter(7, "Lang", "Ulla");
   printArray( m );
   sort( m, new MyComparator() );
   System.out.println("Nach Aufruf von sort( MyComparator ):");
   printArray( m );
}
public static void printArray( Object[] arr ) {
   for( int i=0; i<arr.length; i++ )
      System.out.println( arr[i] );
}
```

In der gleichen Quelldatei fügen wir die Klasse `MyComparator` mit der Definition von `compare()` an:

```
class MyComparator implements Comparator {
  public int compare( Object p, Object q ) {
```

```
      if( p == q ) return 0;
      Mitarbeiter m1 = (Mitarbeiter)p;
      Mitarbeiter m2 = (Mitarbeiter)q;
      int cmp;
      if( (cmp = m1.vorname.compareTo( m2.vorname )) != 0 )
        return cmp;
      if( m1.persNr != m2.persNr )
        return m1.persNr<m2.persNr ? -1 : 1;
      if( (cmp = m1.name.compareTo( m2.name )) != 0 )
        return cmp;
      return 0;
    }
  }
```

Leider lässt sich der Quelltext nicht übersetzen, da ein Zugriff aus der Klasse `MyComparator` auf die privaten Attribute von `Mitarbeiter` nicht erlaubt ist.

Im folgenden sind drei Lösungsalternativen dargestellt. Testen Sie diese, indem Sie den Quelltext entsprechend modifizieren:

Alternative 1: Lesezugriff auf die Attribute

Der Zugriff auf die Attribute von `Mitarbeiter` wird erlaubt, indem man bei den Attributen die Schlüsselworte `private` einfach entfernt. Dies ist natürlich nur ein Notbehelf, als bessere - und aufwendigere - Lösung können die Attribute mit get-Methoden zum Lesen veröffentlicht werden. In diesem Fall sind in `compare()` die Zugriffe auf die Attribute entsprechend anzupassen.

Alternative 2: Comparator als statische verschachtelte Klasse

Wir formulieren `MyComparator` als ***statische verschachtelte Klasse*** (***nested class***), indem wir `MyComparator` innerhalb der Klassendefinition von `Mitarbeiter` platzieren:

```
public class Mitarbeiter {
   //Definition der Attribute und Methoden von Mitarbeiter
   ...
   ...
   static class MyComparator implements Comparator {
      //Definition von compare()
      ...
   }
}
```

Von einer inneren Klasse kann problemlos auf die Attribute und Methoden der umgebenden Klasse zugegriffen werden. Die innere Klasse ist mit `static` zu kennzeichnen, da sonst eine `MyComparator`-Instanz nur über ein bereits existierendes `Mitarbeiter`-Objekt erzeugt werden könnte.

Alternative 3: Comparator als anonyme Innere Klasse

Schließlich kann der Comparator auch als ***anonyme innere Klasse*** formuliert werden, denn der Name der implementierenden Klasse ist völlig unerheblich. Die Definition wird - etwas gewöhnungsbedürftig - zusammen mit einem vorangestellten **`new Comparator()`** anstelle von `new MyComparator()` platziert:

```
public class Mitarbeiter implements Comparable {
  ...
  public static void main( String[] args ) {
    ...
    Arrays.sort( m, new Comparator() {
        //Definition von compare()
        public int compare( Object p, Object q ) {
          ...
```

```
            ...
          }             //Ende der Definition von compare()
        }               //Ende der Implementierung von Comparator
      );                //Ende des Aufrufs sort()
      System.out.println("Nach Aufruf von sort( MyComparator ) :");
      printArray( m );
    }
  }
```

Da dieses Coding leicht unübersichtlich wird, empfiehlt sich - insbesondere wenn das Sortierkriterium mehrfach benötigt wird - die Verwendung von Konstanten

```
public static Comparator MY_COMPARATOR = new Comparator() {
      //Definition von compare()
      ...
   };
```

Dadurch werden zumindest die Sortieraufrufe lesbarer:

```
Arrays.sort( m, MY_COMPARATOR );
```

19.17 Umgekehrte Sortierreihenfolge

Ergänzen Sie die dargestellte Klasse `ReverseComparator`, die `Comparator` implementiert und die Sortierreihenfolge umkehrt. Wird ein Comparator mit parameterlosem Konstruktor

```
new ReverseComparator( )
```

erzeugt, soll dieser die natürliche Ordnung umkehren; wird er mit dem Aufruf

```
new ReverseComparator( Comparator comp )
```

erstellt, soll die in `comp` implementierte Sortierreihenfolge umgekehrt werden.

Tipp: Verwenden Sie ein privates Attribut vom Typ `Comparator`, um die beiden Konstruktoraufrufe zu unterscheiden und den übergebenen Comparator zu registrieren. Um die Methode `compareTo()` für ein Objekt aufrufen zu können, ist ein Cast auf `Comparable` notwendig.

```
import java.util.Comparator;
public class ReverseComparator implements Comparator {
   private Comparator comp = null;
   public ReverseComparator() { }
   public ReverseComparator( Comparator comp ) {
      this.comp = comp;
   }
   public int compare( Object o1, Object o2 ) {
      if( comp==null ) {
          //Umdrehen der Reihenfolge von compareTo()
         ...
      }
      //Umdrehen der Reihenfolge von comp
      ...
   }
}
```

Testen Sie diese Klasse an einigen Beispielen. Wie wird mit dem durch

```
Comparator c = new ReverseComparator( new ReverseComparator() );
```

definierten Kriterium sortiert?