Lösungen

Aufgabe 1:

Die Variable `y` enthält den Wert 1.0. Entsprechend den Prioritäten der beteiligten Operatoren „/“ und „=“ wird erst die Division und anschließend die Zuweisung durchgeführt. Die Division hat zwei ganzzahlige Operanden, daher liefert sie das ganzzahlige Ergebnis 1. Der Wert 1 wird nun zunächst (implizit) in die Gleitkommazahl 1.0 umgewandelt und anschließend der Variablen `y` zugewiesen. Zu dem Thema *Typumwandlung* erfahren Sie später mehr in dem entsprechenden Exkurs (S. 106).

Aufgabe 2:

```
public static void main(String[] args) {

   int x = 3;
   int y = 5;
                     // vorher: 3, 5
   System.out.println("vorher: " + x + ", " + y);
   int h = x;
   x = y;
   y = h;
                     // nachher: 5, 3
   System.out.println("nachher: " + x + ", " + y);
}
```

Aufgabe 3:

```
public static void main(String[] args) {

   double a = 0.5;
   double b = -3.0;
   double c = 2.5;

   double x1, x2;

   x1 = (-b + Math.sqrt(b*b-4*a*c)) / (2*a);
   x2 = (-b - Math.sqrt(b*b-4*a*c)) / (2*a);

   System.out.println(x1 + "," + x2); // 5.0, 1.0
}
```

Aufgabe 4:

```
public static void main(String[] args) {

   for (int i = 2; i < 1000; i++) {
      boolean isprim = true;

      for (int j = 2; j <= Math.sqrt(i); j++) {
         if (i % j == 0) {
            isprim = false;
            break;
         }
      }

       if (isprim) System.out.println(i);
   }
}
```

Aufgabe 5:

```
public static void main(String[] args) {

    int von, bis, h, summe = 0;

    // 1. Zahlen einlesen
    Scanner sc = new Scanner(System.in);

    System.out.print("Zahl1: ");
     von = sc.nextInt();
    System.out.print("Zahl2: ");
     bis = sc.nextInt();

    // 2. Zahlen ggf. vertauschen
    if (von > bis) {
       h = von;
       von = bis;
       bis = h;
    }

    // 3. Zahlen aufaddieren
    while (von <= bis) {
       summe = summe + von;
       von = von + 1;
    }
```

```
        // 4. Summe ausgeben
        System.out.println("Summe: " + summe);
    }
```

Aufgabe 6:

```
    public static double wurzel(double a) {

       double x = 1.0; // irgend ein Startwert...
       double oldx;

        do {
           oldx = x;
           x = 0.5 * (x + a/x);
       }
        while (x != oldx);

        return x;
    }

    public static void main(String[] args) {
       System.out.println(wurzel(2)); // 1.4142135623730
    }
```

Aufgabe 7:

Nachfolgende Lösung zeigt den sog. Euklid'schen Algorithmus zur Bestimmung des größten gemeinsamen Teilers zweier Zahlen:

```
    public static int ggt(int x, int y) {

       while (y != 0) {
           int h = y;
           y = x % y;
           x = h;
       }
       return x;
    }

    public static void main(String[] args) {
       System.out.println(ggt(15,21)); // 3
    }
```

Übrigens gibt es für den Euklid'schen Algorithmus auch eine sehr elegante rekursive Lösung:

```
public static int ggtRek(int x, int y) {
   if (y == 0) return x;
   else        return ggtRek(y, x % y);
}
```

Aufgabe 8:

```
public static int kgv(int x, int y) {
   return x * y / ggt(x, y);
}

public static void main(String[] args) {
   System.out.println(kgv(6,21)); // 42
}
```

Aufgabe 9:

```
public static int quer(int x) {

    int quer = 0;

    while (x > 0) {
       quer = quer + x % 10;
       x = x / 10;
    }

    return quer;
}

public static void main(String[] args) {
   int zahl = 1234;
   System.out.println("Quersumme aus " + zahl +
                      " ist " + quer(zahl));
   // Quersumme aus 1234 ist 10
}
```

Aufgabe 10:

Iterative Lösung:

```
public static int fib(int x) {

   if (x == 1) return 1;

   int fib = 0;
   int last2 = 0;
   int last1 = 1;
   for (int i = 1; i < x; i++) {
      fib = last1 + last2;
      last2 = last1;
      last1 = fib;
   }

   return fib;
}
```

Rekursive Lösung:

```
public static int fib(int x) {
   if (x == 0 || x == 1) return x;
   return fib(x - 1) + fib(x - 2);
}
```

Aufgabe 11:

```
public static int pascal(int z, int s) {

   if (z < 0 && s < 0) return 1;
   if (z < 0 || s < 0) return 0;

   return pascal(z-1, s-1) + pascal(z-1, s);
}

public static void main(String[] args) {

   int n = 6;

   for (int i = 0; i < n; i++) {
```

```
            for (int j = 0; j < (n - i - 1); j++)
               System.out.printf("%4s", "");

            for (int j = 0; j < i + 1; j++)
               System.out.printf("%8d", pascal(i,j));

            System.out.println();
         }
      }
```

Beachten Sie, dass die `pascal()`-Funktion zwar sehr nett aussieht, aber nicht besonders performant arbeitet. Sie macht mehr rekursive Funktionsaufrufe, als notwendig wären. Sie haben bestimmt eine bessere Lösung!

Aufgabe 12:

```
public static void binaer(int x) {
   int q = x / 2;
   int r = x % 2;
   if (q > 0) binaer(q);
   System.out.print(r);
}

public static void main(String[] args) {
   binaer(44); // 101100
}
```

Aufgabe 13:

Hier werden gleich zwei Lösungsalternativen vorgestellt. Beide Ansätze verwenden die folgende Funktion zur Ermittlung der jeweils nächsten Primzahl. Der entsprechende Algorithmus ist aus dem Lösungsvorschlag von Aufgabe 3 abgeleitet.

```
public static int getNextPrim(int i) {

   for (; i < 1000; i++) {
      boolean isprim = true;

      for (int j = 2; j <= Math.sqrt(i); j++) {

         if (i % j == 0) {
            isprim = false;
            break;
```

```
            }
        }

        if (isprim) return i;
    }

    return 0;    // keine weitere Primzahl gefunden...
}
```

Lösungsalternative 1:

Wir legen die Länge des Arrays auf 500 fest, da außer der 2 keine gerade Primzahl existiert. Wir wissen also, dass mindestens die Hälfte aller Zahlen von 1 bis 1000 keine Primzahlen sind und wir somit für diese Zahlen auch keinen Speicherplatz reservieren müssen.

```
public static void main(String[] args) {

    int[] arr = new int[500];
    int n = 0;// Anzahl bislang gefundener Primzahlen

    // Primzahlen ermitteln
    for (int i = 2; i < 1000; i++) {
        i = getNextPrim(i);
        if (i != 0) arr[n++] = i;
        else break;
    }

    // Primzahlen ausgeben
    for (int i = 0; i < n; i++) {
        System.out.println(arr[i]);
    }
}
```

Lösungsalternative 2:

Unsere zweite Lösungsalternative ist die interessantere von beiden. Wir verwenden hier eine rekursive Funktion `getPrimList()`. Sie ermittelt je Funktionsinstanz die nächste Primzahl ab der Zahl `i`. Erst nachdem die letzte Primzahl ermittelt wurde, wird der Array mit der exakt notwendigen Länge angelegt. Wir müssen daher nicht abschätzen, wie viele Primzahlen wir ermitteln werden, sondern wir kennen die exakte Anzahl. Beim Abbau

des Funktionsstacks werden die ermittelten Primzahlen in umgekehrter Reihenfolge dann sukzessive in den Array geschrieben.

```
public static int[] getPrimList(int i, int n) {

   for (; i < 1000; i++) {

      // nächste Primzahl ermitteln
      i = getNextPrim(i);

      // Abbruch nach letzter Primzahl
      if (i == 0) break;

      // rekursiver Aufruf!
      int[] arr = getPrimList(i+1,n+1);

      // in umgekehrter Reihenfolge wird der Array
      arr[n] = i;

      // mit den ermittelten Primzahlen befüllt.
      return arr;
   }

   return new int[n];
      // erst wenn die letzte Primzahl ermittelt
      // wurde, wird der Array mit der passenden
      // Länge angelegt.
}

public static void main(String[] args) {

   int[] arr = getPrimList(2, 0);

   for (int i = 0; i < arr.length; i++) {
      System.out.println(arr[i]);
   }
}
```

Dieser Algorithmus ist nahe verwandt mit den *Backtracking-Algorithmen*. Sie sind oft rekursiv implementiert. Bei immer tieferer Verschachtelung wird die Lösung nach und nach ermittelt, beim Abbau des Funktionsstacks wird die gefundene Lösung festgehalten. Typische Probleme, die mit Backtracking-Algorithmen gelöst werden können, sind das Kürzeste-Wege-Problem, das Acht-Damen-Problem oder das Finden des Weges aus einem Labyrinth.

Aufgabe 14:

```
boolean isDone = false;

while (/* Bedingung */ && !isDone) { // "!" heißt "NOT"
   // Anweisungen, wenn Bedingung wahr (if-Zweig)
   isDone = true;
}
while (!/* Bedingung */ && !isDone) {
   // Anweisungen, wenn Bedingung falsch (else-Zweig)
   isDone = true;
}
```

Die Bool'sche Variable `isDone` stellt fest, ob der `if`-Zweig oder der `else`-Zweig bereits abgearbeitet wurden. Sie verhindert zum Einen, dass eine der beiden Schleifen mehr als einmal ausgeführt wird und stellt zum Anderen sicher, dass der `else`-Zweig nur dann abgearbeitet wird, wenn der `if`-Zweig zuvor noch nicht ausgeführt wurde. Es ist wichtig, eine zusätzliche Variable wie `isDone` dafür zu haben, da sich durch die Abarbeitung des `if`-Zweiges die Parameter der eigentlichen Bedingung der „`if`-Anweisung" derart geändert haben könnten, dass auch eine Abarbeitung des `else`-Zweigs herbeigeführt werden würde.