第７章

集　合

7–1 集合とは

《もの》の集まりを表すのが**集合**です。本節では、集合の基本を学習します。

集合と要素

集合（*set*）とは、**客観的に範囲が規定された《もの》の集まり**であり、その集合中の個々の《もの》が**要素**（*element*）です。

Fig.7-1 に『九州の県』の集合を示します。《福岡県》、《佐賀県》などが『九州の県』の集合の要素です。

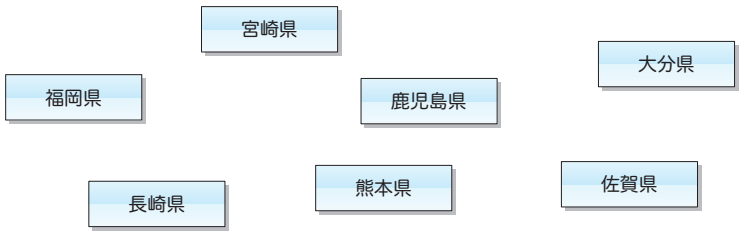

Fig.7-1　九州の県の集合

集合 X の要素が 1 と 5 であることは、以下のように表記します。

$$X = \{1, 5\}$$

ただし、集合内の要素には**順序がありません**ので、

$$X = \{5, 1\}$$

とも表記できます。

また、一般に、

$$N = \{1, 2, 3, 4, \dots\}$$ 自然数の集合

と自然数の集合を N で表し、

$$Z = \left\{ \begin{array}{l} 1,\ 2,\ 3,\ 4, \dots \\ \emptyset \\ -1, -2, -3, -4, \dots \end{array} \right\}$$ 整数の集合

と整数全体の集合を Z で表します。

▶ N は natural number（自然数）の頭文字、Z はドイツ語の Zahl（数）の頭文字に由来します。

集合の要素は、それ以上分解できない要素、すなわち**アトム**（*atom*）であっても、集合であっても構いません。

ただし、すべての要素は**互いに異なるものでなければなりません**。すなわち、{1, 5, 1} といった集合はあり得ません。

▶ 要素が互いに異ならない集合を**多重集合**と呼んで区別します。

a が集合 X の要素であることを、『a は X に**入っている**』、あるいは『a は X に**属する**』といいます。その表記が、

$a \in X$　または　$X \ni a$	a は X に入っている

です。一方、b が集合 X の要素でなければ、

$b \notin X$　または　$X \not\ni b$	b は X に入っていない

と書き表します。

*

二つの集合 X と Y が同じ要素から構成されるとき、『**X と Y は等しい**』といい、

$X = Y$　または　$Y = X$	X と Y は等しい

と表記します。一方、同じ要素で構成されない場合は、『**X と Y は等しくない**』といい、

$X \neq Y$　または　$Y \neq X$	X と Y は等しくない

と表記します。

*

整数の集合のように、要素数が無限大の集合は**無限集合**と呼ばれます。それに対して、要素数が有限の集合は**有限集合**です。

有限集合 X の要素数が n であるとき、

$\lvert X \rvert = n$	X の要素数は n

と表します。ただし、X が無限集合であれば、

$\lvert X \rvert = \infty$	X は無限集合

と書き表します。

さて、集合は《もの》の集まりであると最初に説明しました。日本語の「**集まり**」には複数のものを表すイメージがありますが、集合の要素数は 1 個であっても構いません。すなわち $\lvert X \rvert = 1$ である X も集合です。

さらに、$\lvert X \rvert = \emptyset$ である X も集合とみなされます。このような、要素が 1 個もない集合を**空集合**（*empty set*）と呼び、ϕ で表します。

部分集合と真部分集合

他の集合に含まれる集合は、部分集合あるいは真部分集合です。

部分集合

集合Aのすべての要素が集合Bの要素となっているとき、AはBの**部分集合**（*subset*）であり、『AはBに**含まれる**』といいます（**Fig.7-2**）。

この関係は$A \subset B$あるいは$B \supset A$と表します。

AとBが等しければ、互いに部分集合であり$A \subset B$かつ$B \subset A$です。

例　$A = \{1, 3\}$で$B = \{1, 3, 5\}$のとき$A \subset B$。

例　$A = \{1, 3, 5\}$で$B = \{1, 3, 5\}$のとき$A \subset B$かつ$B \subset A$。

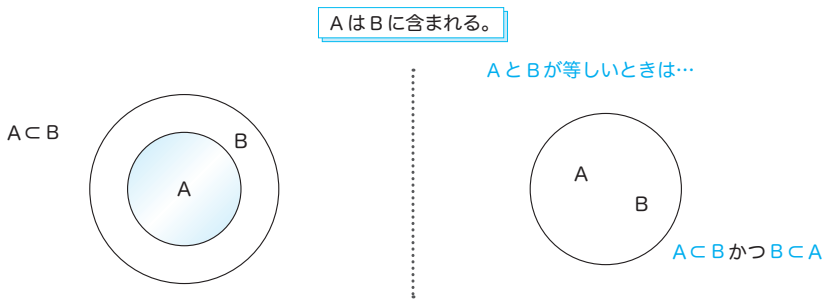

Fig.7-2　部分集合

真部分集合

集合Aのすべての要素が集合Bの要素であって、集合Aと集合Bが等しくないとき、すなわち$A \subset B$かつ$A \neq B$であるとき、AはBの**真部分集合**（*proper subset*）です。

この関係は$A \subsetneq B$あるいは$B \supsetneq A$と表します。

例　$A = \{1, 3\}$で$B = \{1, 3, 5\}$のとき
AはBの部分集合であり、AはBの真部分集合である。

例　$A = \{1, 3, 5\}$で$B = \{1, 3, 5\}$のとき
AはBの部分集合であるが、AはBの真部分集合ではない。

▶　部分集合の関係をA⊆Bと表記して、真部分集合の関係をA⊂Bと表記する流儀もあります。

集合の演算

集合に対する基本的な演算は、和・積・差を求める演算です。

和

集合Aと集合Bの少なくとも一方に属している要素の集合をAとBの**和集合**と呼び、A∪Bと書き表します。

これを図で表したのが**Fig.7-3**です。

例　A={1, 3, 5}でB={1, 4, 6}のとき
A∪Bは{1, 3, 4, 5, 6}です。

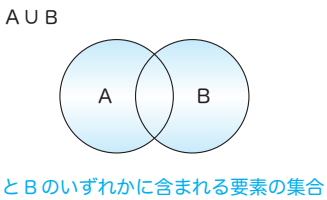

Fig.7-3　集合の和

積

集合Aと集合Bの両方に属している要素の集合をAとBの**積集合**と呼び、A∩Bと書き表します。

これを図で表したのが**Fig.7-4**です。

例　A={1, 3, 5}でB={1, 4, 6}のとき
A∩Bは{1}です。

例　A={3, 5}でB={1, 4, 6}のとき
A∩Bはφ（空集合）です。

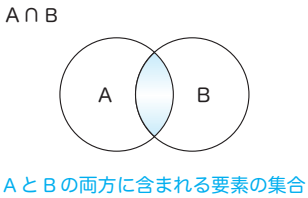

Fig.7-4　集合の積

差

集合Aの要素であって集合Bに属さない要素の集合を**差集合**と呼び、A－Bと書き表します。

これを図で表したのが**Fig.7-5**です。

例　A={1, 3, 5}でB={1, 4, 6}のとき
A　－　Bは{3, 5}です。

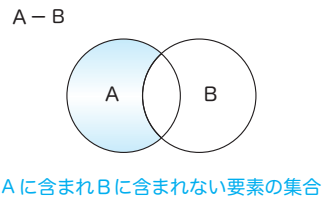

Fig.7-5　集合の差

7–2 配列による集合

同じ型のデータの集合は、配列によって表せます。

配列による集合

すべての要素が同じ型の集合は、配列によって容易に実現できます。たとえば、整数の集合{1，2，3，4，5，6，7，8}は、以下のように要素数8の **int** 型配列に格納できます。

▶ ものの《集まり》が表現できればよいため、配列内での要素の順序は**任意**です。

8	1	4	2	7	6	3	5

もっとも、配列の全要素を用いて集合を表そうとすると、集合の要素数と配列の要素数とを常に一致させるための工夫が必要です。配列本体と、“現在いくつの要素が集合に入っているのか”を表す変数とを組み合わせるのが現実的です。

そこで、**int** 型を要素とする集合を、**Fig.7-6** に示すクラス *IntSet* によって実現することにします。

```
class IntSet {
    int max;     // 集合の容量
    int num;     // 集合の要素数
    int[] set;   // 集合本体
}
```

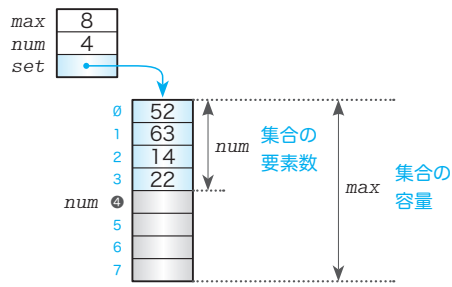

Fig.7-6 int型の集合を実現するクラスIntSet

クラス *IntSet* は、三つのフィールドで構成されます。

▪ *max*

集合の容量、すなわち配列の要素数を表すフィールドです。ここに示す図の場合であれば、*max* の値は8です。

▪ *num*

集合の要素数です。集合の要素は、配列の先頭側（インデックスの小さいほうの要素）に格納します。すなわち、*set*[0] ～ *set*[*num* - 1] の *num* 個の要素が、集合の要素です。この図の場合、*num* の値は4です。なお、配列が空集合であれば、この値は0となります。

List 7-1 [A] Chap07/IntSet.java

```
// int型の集合

public class IntSet {
   private int max;          // 集合の容量
   private int num;          // 集合の要素数
   private int[] set;        // 集合本体

   //--- コンストラクタ ---//
   public IntSet(int capacity) {
      num = 0;
      max = capacity;
      try {
         set = new int[max];            // 集合本体用の配列を生成
      }
      catch (OutOfMemoryError e) {      // 配列の生成に失敗
         max = 0;
      }
   }

   //--- 集合の容量 ---//
   public int capacity() {
      return max;
   }

   //--- 集合の要素数 ---//
   public int size() {
      return num;
   }
```


▪ *set*

集合を格納するための配列です（厳密には、配列の本体を参照する配列変数です）。

▶ 配列用記憶域の確保は、コンストラクタで行います。

クラス *IntSet* のプログラムを **List 7-1** に示します。

コンストラクタ：IntSet

コンストラクタは、集合本体用の配列を生成するなどの準備処理を行います。

生成時の集合は空（データが 1 個もない状態）ですから、*num* の値を 0 にします。

そして、仮引数 *capacity* に受け取った《集合の容量》を *max* にコピーして、要素数が *max* となるように、配列 *set* の本体を生成します。

集合の容量を調べる：capacity

容量（集合が含むことのできる最大の要素数）を返すメソッドです。フィールド *max* の値をそのまま返します。

集合の要素数を調べる：size

集合の要素数を返すメソッドです。フィールド *num* の値をそのまま返します。

要素の探索：indexOf

集合本体の配列 *set* に、値 *n* の要素が入っているかどうかを調べるメソッドです。利用するアルゴリズムは、3-2 節で学習した線形探索です。

Fig.7-7 に示すように、配列の先頭から走査し、探索成功時は見つけた要素のインデックスを返し、失敗時は **-1** を返します（図の場合は 4 を返却します）。

▶ 探索の対象は、*set*[0] ～ *set*[*num* - 1] の *num* 個の要素です。

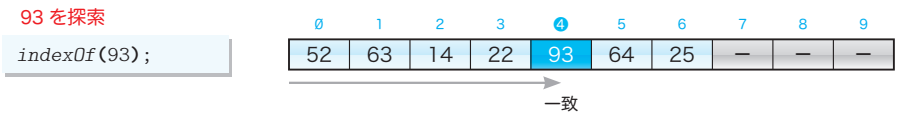

Fig.7-7　集合からの探索

要素が入っているか：contains

集合本体の配列 *set* に、値 *n* の要素が入っているかどうかを調べるメソッドです。

内部でメソッド *indexOf* を呼び出して、その結果に基づいて、入っていれば **true** を、入っていなければ **false** を返します。

要素の追加：add

集合に要素 *n* を追加するメソッドです。追加を行うのは、配列が満杯でなくて、かつ集合に *n* が入っていないときのみです。

追加の手続きは単純です。**Fig.7-8** に示すように、末尾要素の次の要素である *set*[*num*] に *n* を代入して、その後 *num* をインクリメントするだけです。

▶ メソッドの返却型は **boolean** 型です。要素の追加を行わなかった場合は **false** を返し、追加を行った場合は **true** を返します。

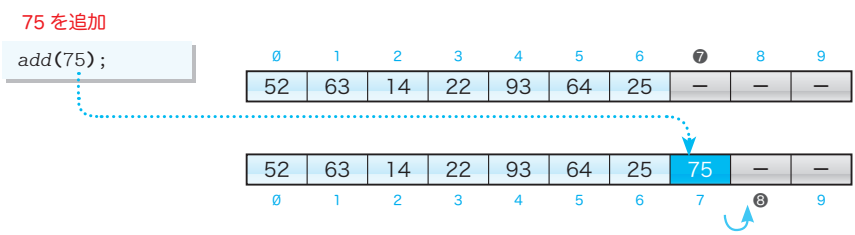

Fig.7-8　集合への追加

要素の削除：remove

集合から要素 *n* を削除するメソッドです。削除を行うのは、集合内に *n* が含まれているときのみです。

List 7-1 [B] Chap07/IntSet.java

```
//--- 集合からnを探索してインデックス（見つからなければ-1）を返す ---//
public int indexOf(int n) {
    for (int i = 0; i < num; i++)
        if (set[i] == n)
            return i;                   // 含まれる
    return -1;                          // 含まれない
}

//--- 集合にnが入っているか ---//
public boolean contains(int n) {
    return (indexOf(n) != -1) ? true : false;
}

//--- 集合にnを追加 ---//
public boolean add(int n) {
    if (num >= max || contains(n) == true) // 満杯 or nが含まれている
        return false;
    else {
        set[num++] = n;                 // 末尾に追加
        return true;
    }
}

//--- 集合からnを削除 ---//
public boolean remove(int n) {
    int idx;                // nが格納されている要素のインデックス

    if (num <= 0 || (idx = indexOf(n)) == -1) // 空 or nが含まれていない
        return false;
    else {
        set[idx] = set[--num];          // 末尾の要素を削除位置に移動
        return true;
    }
}
```

➡

▶ メソッドの返却型は **boolean** 型です。要素の削除を行わなかった場合は **false** を返し、削除を行った場合は **true** を返します。

Fig.7-9 に示すのは、集合から 22 の削除を行う手続きです。

まず 22 が入っている要素のインデックス 3 をメソッド *indexOf* で調べて *idx* に代入します。その後、要素数 *num* を 7 から 6 へとデクリメントし、末尾要素 *set*[*num*] すなわち *set*[6] の値を、*set*[*idx*] すなわち *set*[3] にコピーすると、作業は完了です。

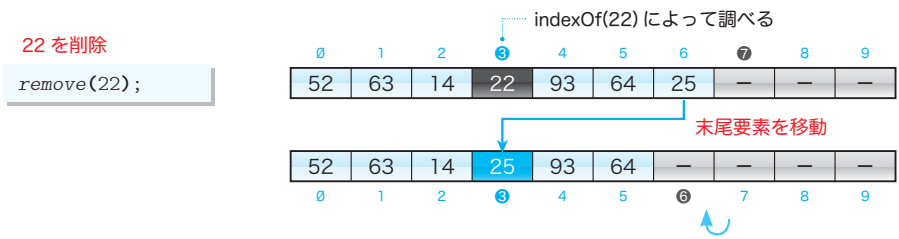

Fig.7-9 集合からの削除

他の集合にコピー：copyTo

集合を他の集合にコピーするメソッドです。コピー元は自分自身（**this**）の集合で、コピー先は引数で指定された集合 *s* です。

集合の要素数 *num* がコピー先集合 *s* の容量 *s.max* を超える場合は、コピー先集合の容量である *s.max* 個の要素だけを先頭からコピーします（**Fig.7-10**）。

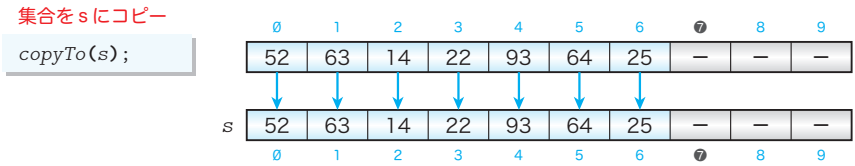

Fig.7-10　集合のコピー

他の集合をコピー：copyFrom

集合をコピーするメソッドです。メソッド *copyTo* とはコピー元とコピー先が逆です。

集合 *s* の要素数 *s.num* がコピー先の自分自身の集合の容量 *max* を超える場合は、集合の容量である *max* 個の要素だけを先頭からコピーします。

他の集合と等しいかどうかを調べる：equalTo

自分自身の集合が、集合 *s* と等しいかどうかを判定するメソッドです。等しければ **true** を、等しくなければ **false** を返します。等価性の判定は以下のように行います。

▪ 要素数が等しくない場合

集合は等しくないと判断します。

▪ 要素数が等しい場合（プログラム網かけ部）

i を 0，1，…，*num* − 1 とインクリメントして、*set*[*i*] が集合 *s* に含まれるかどうかを調べます。含まれない要素を見つけたら、集合は等しくないと判定します（**break** 文によって内側の **for** 文を中断した後に、**return** 文によって **false** を返します）。

中断されることなく外側の **for** 文が完了すれば、集合は等しいと判定できますから、**true** を返します。

二つの集合の和をコピー：unionOf

集合 *s1* と集合 *s2* の和集合を、自分自身の集合にコピーするメソッドです。

メソッド *copyFrom* によって集合 *s1* を自分自身にコピーして、それから *s2* の全要素を一つずつ追加します。

List 7-1 [C]　　Chap07/IntSet.java

```
    //--- 集合sにコピー ---//
    public void copyTo(IntSet s)  {
        int n = (s.max < num) ? s.max : num;    // コピーする要素数
        for (int i = 0; i < n; i++)
            s.set[i] = set[i];
        s.num = n;
    }

    //--- 集合sをコピー ---//
    public void copyFrom(IntSet s)   {
        int n = (max < s.num) ? max : s.num;    // コピーする要素数
        for (int i = 0; i < n; i++)
            set[i] = s.set[i];
        num = n;
    }

    //--- 集合sと等しいか ---//
    public boolean equalTo(IntSet s) {
        if (num != s.num)                       // 要素数が等しくなければ
            return false;                       // 集合も等しくない

        for (int i = 0; i < num; i++) {
            int j = 0;
            for ( ; j < s.num; j++)
                if (set[i] == s.set[j])
                    break;
            if (j == s.num)                     // set[i]はsに含まれない
                return false;
        }
        return true;
    }

    //--- 集合s1とs2の和集合をコピー ---//
    public void unionOf(IntSet s1, IntSet s2) {
        copyFrom(s1);                           // 集合s1をコピー
        for (int i = 0; i < s2.num; i++)        // 集合s2の要素を追加
            add(s2.set[i]);
    }

    //--- "{ a b c }"形式の文字列表現に変換 ---//
    public String toString() {
        StringBuffer temp = new StringBuffer("{ ");
        for (int i = 0; i < num; i++)
            temp.append(set[i] + " ");
        temp.append("}");
        return temp.toString();
    }
}
```

文字列表現への変換：toString

文字列表現を返すメソッドです。

集合の要素が1，5，7であれば、文字列"{ 1 5 7 }"を作って返却します。

▶ toStringメソッドについては、**Column 7-1**（次ページ）で学習します。

Column 7-1 toStringメソッド

クラス *IntSet* では、以下に示す形式の toString メソッドを定義しています。

```
public String toString() { /*…*/ }
```

同じ形式のメソッドは、第3章のハッシュ用のデータクラス *Data* など、本書のいくつかのプログラムでも定義しています。

いずれも、クラスインスタンスの状態（データの内容）を、簡潔な文字情報として表現した文字列を作って返却するメソッドです。このメソッドをクラスに定義することは、一種の《**定石**》です（著者である私が独自に考えついたものではありません）。そのため、Java でプログラムを開発するためには、必ず理解しておかなければなりません。

そもそも、toString というのは、**java.lang** パッケージに所属する **Object** クラスで以下のように定義されたメソッドであり、"クラス名@ハッシュ値" という文字列を返却します。

```
public class Object {
   // 中略
   public String toString() {
      return getClass().getName() + "@" + Integer.toHexString(hashCode());
   }
   // 中略
}
```

＊

Java のすべてのクラスは、直接的あるいは間接的に **Object** クラスから派生しています（**Column 2-10**：p.70）。そのため、クラスで toString メソッドを定義するということは、**Object** クラスの toString メソッドを**オーバライドする**（上位クラスから継承したメソッドに対して、新たな定義を与える）ことを意味します。

また、クラスで toString メソッドを定義しなければ（そのクラスの上位クラスで独自にオーバライドされていない限り）、上に示した **Object** クラスの toString メソッドが、そのまま継承されることになります。

さて、Java でメソッドをオーバライドする際は、**アクセス制限を強めることができない**ことになっています。したがって、いかなるクラスであっても、toString メソッドは、**public** メソッドとして定義しなければならないことに注意しましょう（すなわち、**public** を省略したり、**private** 属性や **protected** 属性を与えたりすることはできません）。

＊

さて、クラスで toString メソッドをオーバライドする際は、クラスの特性やインスタンスの状態を表す、適切な文字列を返却するように定義します。

toString メソッドをオーバライドするプログラム例を **List 7C-1** に示します。このプログラムは、クラス *A*、クラス *B*、それらをテストするためのクラス *ToStringTester* で構成されています。

▪ クラスA

toString メソッドをオーバライドしていません。そのため、**Object** クラスの toString メソッドをそのまま継承します。

▪ クラスB

toString クラスメソッドをオーバライドしています。文字列 "B[99]" を返却します（"99" の部分は、フィールド *x* に設定されている値です）。

List 7C-1 Chap07/ToStringTester.java

```
// toStringメソッドの働きを確認

class A {
   // toStringは定義しない
}

class B {
   int x;
   B(int x) { this.x = x; }   // コンストラクタ
   // toStringをオーバライド
   public String toString() { return "B[" + x + "]"; }
}

public class ToStringTester {

   public static void main(String[] args) {
      A a1 = new A();
      A a2 = new A();
      B b1 = new B(18);
      B b2 = new B(55);

      System.out.println("a1 = " + a1.toString()); ← 1
      System.out.println("a2 = " + a2); ← 2
      System.out.println("b1 = " + b1.toString()); ← 3
      System.out.println("b2 = " + b2); ← 4
   }
}
```

実行結果一例

```
a1 = A@ca0b6
a2 = A@10b30a7
b1 = B[18]
b2 = B[55]
```

▪ クラス ToStringTester

二つのクラスとその toString メソッドをテストするためのクラスです。クラス *A* のインスタンスを 2 個、クラス *B* のインスタンスを 2 個生成します。

1 文字列リテラル `"a1 = "` と、インスタンス *a1* に対して toString メソッドが呼び出された結果返される文字列が連結されますので、`"a1 = A@ca0b6"` と表示されます（`"A"` は *a1* のクラスの名前で、`"ca0b6"` はインスタンス *a1* に対して内部的に与えられているハッシュ値です）。

2 *a2* を表示します。このとき、**暗黙の内に toString メソッドが呼び出されます**。というのも、『文字列が必要となる文脈にクラス型変数が置かれていれば、そのクラス型変数に対して、自動的に toString メソッドが呼び出される』という規則があるからです。したがって、文字列リテラル `"a2 = "` と、*a2* に対して暗黙裏に呼び出された toString が返却した文字列とが連結されますので、`"a2 = A@10b30a7"` と表示されます。

3 文字列リテラル `"b1 = "` と、インスタンス *b1* に対して toString メソッドが呼び出された結果返される文字列とが連結されますので、`"b1 = B[18]"` と表示されます。

4 文字列リテラル `"b2 = "` と、インスタンス *b2* に対して暗黙裏に toString メソッドが呼び出されることによって得られる文字列が連結されますので、`"b2 = B[55]"` と表示されます。

toString メソッドの明示的な呼出し（プログラム網かけ部）**は省略できる**ことが分かりました。このメソッドについて、簡潔にまとめると、以下の教訓となります：

インスタンスの状態を簡潔な文字列表現で返却するメソッドは、`public String toString()` の形式で定義するとよい。というのも、そのメソッドは、文字列が必要な文脈で自動的に呼び出されるからである。

int 型集合クラス *IntSet* の利用例を **List 7-2** に示します。

List 7-2 Chap07/IntSetTester.java

```
// int型集合クラスIntSetの利用例

public class IntSetTester {

   public static void main(String[] args) {
      IntSet s1 = new IntSet(20);
      IntSet s2 = new IntSet(20);
      IntSet s3 = new IntSet(20);

      s1.add(10);         // s1 = {10}
      s1.add(15);         // s1 = {10, 15}
      s1.add(20);         // s1 = {10, 15, 20}
      s1.add(25);         // s1 = {10, 15, 20, 25}

      s1.copyTo(s2);      // s2 = {10, 15, 20, 25}
      s2.add(12);         // s2 = {10, 15, 20, 25, 12}
      s2.remove(25);      // s2 = {10, 15, 20, 12}

      s3.copyFrom(s2);    // s3 = {10, 15, 20, 12}

      System.out.println("s1 = " + s1);
      System.out.println("s2 = " + s2);
      System.out.println("s3 = " + s3);

      System.out.println("集合s1に15は" +
                         (s1.contains(15) ? "含まれる" : "含まれない"));

      System.out.println("集合s2に25は" +
                         (s2.contains(25) ? "含まれる" : "含まれない"));

      System.out.println("集合s1とs2は" +
                         (s1.equalTo(s2) ? "等しい" : "等しくない"));

      System.out.println("集合s2とs3は" +
                         (s2.equalTo(s3) ? "等しい" : "等しくない"));

      s3.unionOf(s1, s2);      // s3 ← s1 ∪ s2

      System.out.println("集合s1とs2の和集合は" + s3);
   }
}
```

実行結果

```
s1 = { 10 15 20 25 }
s2 = { 10 15 20 12 }
s3 = { 10 15 20 12 }
集合s1に15は含まれる
集合s2に25は含まれない
集合s1とs2は等しくない
集合s2とs3は等しい
集合s1とs2の和集合は
{ 10 15 20 25 12 }
```

演習 7–1

クラス *IntSet* に、集合が空であるかどうかを判定するメソッド、満杯であるかどうかを判定するメソッド、全要素を削除するメソッドを追加せよ。

```
boolean isEmpty()          // 集合は空であるか
boolean isFull()           // 集合は満杯か
void clear()               // 集合を空にする（全要素を削除）
```

演習 7–2

クラス *IntSet* に対して、集合 *s* の全要素を追加するメソッド、集合 *s* に入っている要素のみを残して入っていない要素を削除するメソッド、集合 *s* に入っている要素を削除するメソッドを追加せよ。

```
boolean add(IntSet s)          // 集合sとの和集合にする
boolean retain(IntSet s)       // 集合sとの積集合にする
boolean remove(IntSet s)       // 集合sとの差集合にする
```

メソッドの実行によって、集合が変更された場合は **true** を、そうでなければ **false** を返すこと。

演習 7–3

クラス *IntSet* に対して、集合 *s* の部分集合であるかどうかを判定するメソッド、集合 *s* の真部分集合であるかどうかを判定するメソッドを作成せよ。

```
boolean isSubsetOf(IntSet s)          // 集合sの部分集合か
boolean isProperSubsetOf(IntSet s)    // 集合sの真部分集合か
```

判定結果が成立すれば **true** を、そうでなければ **false** を返すこと。

演習 7–4

クラス *IntSet* に対して、集合 *s1* と集合 *s2* の積集合をコピーするメソッド、集合 *s1* と集合 *s2* の差集合をコピーするメソッドを作成せよ。

```
void intersectionOf(IntSet s1, IntSet s2)      // s1とs2の積集合をコピー
void differenceOf(IntSet s1, IntSet s2)        // s1とs2の差集合をコピー
```

演習 7–5

クラス *IntSet* では、要素の並びが不定である。配列内の要素を常に昇順にソートしておくように変更したクラス *IntSortedSet* を作成せよ。

そうすると、要素の探索は２分探索によって行えるし、要素の追加や削除も２分探索によって得られた位置に対して行える。また、他の配列と等しいかどうかの判断も効率よく行える。