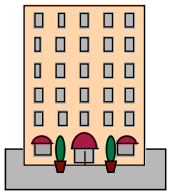
Chapter １２. 配列

１２－１．配列と要素

１２－１－１．配列とは？

配列とは、変数の一種で、同じ名前で幾つものデータを保持することができる変数のことです。数学でベクトルをやった人なら、ベクトル変数のことだと思ってください。ベクトルを知らない人のために、現実の世界に例えてみましょう。通常の変数が一戸建ての家だとすれば、配列はマンションやアパートのように、いくつもの世帯を持つことができる構造物だと考えてください。

一戸建て（普通の変数）には一世帯



マンション（配列変数）には複数の世帯

図12-1　変数と配列変数

なぜ、配列を使うのでしょうか？データが幾つもあったときに、それに対応して変数を用意すると、データの個数分だけ変数が必要になってしまいます。配列を用いて同種のデータを複数保持すれば、わずかな数の配列変数を済むことが多いのです。特に、配列にすれば、繰り返しの構文を使うことによって、複数のデータに同一の処理を適用することができるという利点もあります。たとえば、1000個のデータを同時に扱いたいときに、それをプログラムでどのように扱うか想定してみてください。誰も次のように変数を1000個用意しようとは思わないでしょう。

悪い例：

```
int     x1, x2, x3, x4, x5, x6, x7, x8, x9, ....., x998, x999, x1000;
//      こんなに変数宣言を書くだけで大変です
```

１２－１－２．配列の宣言の仕方

Java言語での配列は、同じ型のデータだけを保持します。プログラム中に導入するためには通常の変数と同じように宣言を行なう必要があります。

▼配列の宣言の書式

```
型名     変数名 [ ];    または
型名     [ ] 変数名;
```

たとえば、配列の宣言だけを行なう場合は、次のように記述します。配列であることを示す角括弧[]は、変数名の前に記述しても、後に記述しても構いません。

```
int          a[ ];          // 整数配列の宣言のみを行なう
double       [ ] b;         // 実数配列の宣言のみを行なう
```

従来のプログラミング言語と決定的に違う点は、宣言だけを行なっても、データを入れる領域が確保されないということです。そのため、宣言では何個のデータを保持するかという配列のサイズを必要としていません。実際にデータを入れる場所を確保するには、次のような書式で記述する必要があります。

▼データ領域を確保する書式

```
変数名 = new 型名[ 配列のサイズ ];
```

配列のサイズとは、データをいれる部屋の個数を示します。整数式で記述します。たとえば、先ほどの２つの配列変数aとbにそれぞれデータ領域を確保してみましょう。

```
a = new int[ 10 ];          // 配列 aに、10個の整数のデータ領域を確保する
b = new double[ 5 ];        // 配列 bに、5個の実数のデータ領域を確保する
```

もちろん、通常の変数と同様に、データ領域を確保しながら宣言することもできます。

▼宣言と共にデータ領域を確保する書式

```
型名 変数名 [ ] = new 型名[ 配列のサイズ ];    または
型名 [ ] 変数名 = new 型名[ 配列のサイズ ];
```

データを入れる場所を確保しつつ、配列を宣言してみましょう。宣言で使われている型名と、データ領域の確保のときに必要な型名とは一致している必要があります。

```
int a [ ] = new int[ 8 ];                  // 8つ整数データを入れることが可能
double [ ] b = new double[ 20 ];           // 20個実数データを入れることが可能
```

これは、次のように分解して記述することもできます。配列変数aについてだけ記述してみました。

```
int a[ ];                  // 変数aを整数の配列として宣言
a = new int[ 8 ];          // ８つのデータを入れる領域を確保する
```

データ領域が確保されますと、データをおく場所ができます。次の図のように、部屋番号が振られます。この番号は、添え字あるいはインデックス（index）と呼ばれています。配列ではそれぞれの部屋を別個に扱うことも、全部のデータを１つとして扱うこともできます。

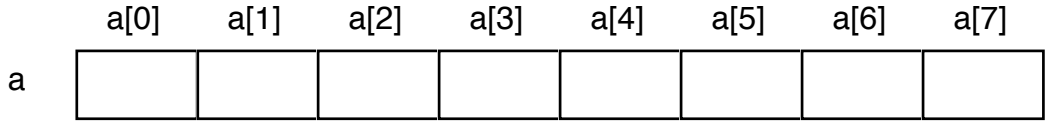


図12-2　変数と配列変数

１２－１－３．配列の要素への参照と代入

それぞれのデータを置いておける部屋を要素（Element）と呼んでいます。一旦配列を宣言した後は、多くの場合、それ以降の操作の対象は配列の要素単位となります。要素の参照の仕方は、次のような書式で行ないます。

▼配列の要素への参照の書式

```
変数名 [  添え字の式  ]
```

たとえば次のように参照するための記述を行なうことができます。添え字の式は整数値になるように記述します。

```
a[ 4 ]
b[ 7 ]
```

添え字の取り得る範囲は、「0〜配列のサイズ-1」になっています。添え字が０から始まることに注意してください。そのため、通常の数え方よりは１つ数が違ってきています。たとえば、配列で４番目の要素とは、通常の数え方では５番目の要素になります。また、添え字が配列の持つ範囲を越えると実行時にエラーが起こり、プログラムが急停止してしまいます。

★要素への代入と参照

要素は参照するための書式を用いて、通常の変数のように扱うことができます。まずは、代入からみてみましょう。aは、整数型の配列、bは実数型の配列と仮定します。

```
a[ 4 ] = 20;          // 配列 aの4番目の要素の値を20にする
b[ 0 ] = 10.4;        // 配列 bの0番目の要素の値を10.4にする
```

一旦要素に値が代入されたら、通常の変数のようにその値を参照することもできます。

```
x = a[ 5 ];           // 配列 aの5番目の要素の値をxに代入する
y = a [ 4 ] * 20;     // 配列 aの4番目の要素の値を20倍してyに代入する
```

あるいは、ある要素を参照して、それが持つ値を別の要素に代入するために用いることができます。

```
a[ 3 ] = a[ 4 ] + 12;     // 配列 aの3番目の要素の値を4番目の要素+12の値にする
b[ 6 ] = a[ 4 ] * 1.5;    // 配列 aの4番目の要素の値×1.5を配列 bの
                          // 6番目の要素に代入する
```

もちろん、再帰代入文も用いることできます。配列の要素の対してインクリメント・デクリメント演算子を用いるプログラマも多いのですが、最初はわかりにくいので分けた方が無難でしょう。

```
a[ 3 ] = a[ 3 ] * 2;      // 配列 aの3番目の要素の値を２倍にする
a[ 5 ] ++;                // 配列 aの5番目の要素の値を+1する
```

★添え字の式

配列の要素が指定された場合、まず要素が何番目であるかを指定する添え字から評価されます。添え字は、整数の式であれば構いません。ですから、変数なども使うことができます。

```
int x = 5;
a[ x ] = 36;              // 5番目の要素に、36を代入
a[ x + 1 ] = 30;          // 6番目の要素に、30を代入
a[ x -2 ] = a[ x ] + 20;  // 3番目の要素に、5番目の要素に20を足したものを代入
```

さらに複雑なことに配列の要素が保持する値を添え字として使うこともできる。

```
a[ 1 ] = 4;  a[ 4 ] = 50;
a[ 0 ] = a[ a[ 1 ] ] +10;     // 0番目の要素に、4番目の要素に10を足したものを代入
```

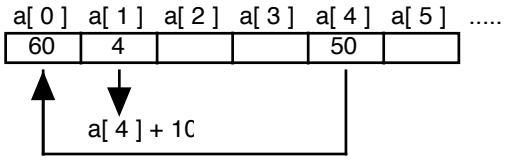


図12-3　配列の要素の値を添え字として使う

１２－１－４．繰返しを使った配列への代入

配列の要素を操作するためには、繰返しの構文は欠かせません。繰返しと配列を併用することにより、長かったプログラムを短くすることができるのです。それだけ、この組合せは操作の記述能力が優れていることを示しています。まずは、配列の要素を表示することと、繰返しを使って配列の各要素に代入することを考えてみましょう。

★配列の要素の初期値とその表示

配列の各要素は、基本型の配列である限り、0クリア（すべて０が代入されている）されているようです。たとえば、整数型の配列の場合は、各要素には0が代入されていると仮定されます。実数型の配列の場合は、0.0が、論理型の配列の場合はfalseが、各要素に代入されているようです。ただし、次の例以降のように、これを仮定せず、各要素には何らかの値を最初に代入しておくべきでしょう。０クリアされている状態を表示してみましょう。

```
int       a [ ] = new  int [ 10 ];
for ( int  i= 0; i <10; i++ ) {  System.out.println( a [ i ] );  }
```

★インデックスの値を使った代入

それぞれの要素がもつ整数値を、要素のインデックスを元に計算してみます。たとえば、0番目の要素は10で始まり、番号が増えるに従って要素の値が10ずつ増えていくように繰返しを使って代入してみます。

```
int       a [ ] = new  int [ 10 ];
for ( int  i= 0; i <10; i++ ) {  a[ i ] = ( i+1) * 10;  }
```

a

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]
10	20	30	40	50	60	70	80	90	100

図12-4　代入結果の配列

★漸化式を使った代入

手前の要素の値を参照して、次の要素の値を代入します。そのような代入の仕方は、数列では漸化式と呼ばれています。たとえば、手前の要素の値とインデックスの値を足し合わせた値を次の要素に代入してみましょう。最初の要素の値は、０とします。

```
int       a [ ] = new  int [ 10 ];
a[ 0 ] = 0;
for ( int  i= 1; i <10; i++ ) {  a[ i ] = a[ i -1 ] + i;  }
```

だんだん足し合わされていきますので、各要素の値は次の図のようにインデックスが示す数までの総和になっています。たとえば、a[6]は、1から6までの数の総和になっていて、値は15になっています。

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]
0	1	3	6	10	15	21	28	36	45

図12-5　代入結果の配列

インデックスと足し合わせるのではなくて、次の要素の値を代入するために、手間の要素ともう一つ手前の要素を足し合わせることを考えてみましょう。最初の２つの要素の値は、0と1にしておきます。次のような形で

記述できるでしょう。

```
int       f[ ] = new int [ 10 ];
f[ 0 ] = 0; f[ 1 ] = 1;
for ( int i= 2; i <10; i++ ) {   f[ i ] = f[ i -1 ] + f[ i -2 ];  }
```

配列の各要素は次のような結果になります。これは数列では有名でFibonatti数列と呼ばれています。実用ではほとんど役に立ちませんが、漸化式で作られる基本的な数列の１つです。

f[0]	f[1]	f[2]	f[3]	f[4]	f[5]	f[6]	f[7]	f[8]	f[9]
0	1	1	2	3	5	8	13	21	34

図12-6　代入結果の配列（Fibonatti数列）

★乱数を用いた代入

実数の章で紹介したMath.randomメソッドを用いて、適当な値を配列の各要素に代入してみましょう。

```
int       myArray [ ] = new  int [ 40 ];
for ( int  i = 0; i < 40; i++ ) {
          myArray[ i ] = (int) (Math.random( ) * 200 );
}
```

Math.randomメソッドは、0.0以上1.0未満の間の実数を乱数として発生させます。上の例ではそれに200を掛けて、0.0以上200.0未満までの実数を発生させるようにしています。そして、その結果を整数への型変換をしていますので、最終的に0から199までのいずれかの整数値が各要素に代入されます。

※負の数も含めて、ある範囲の整数を発生させたい場合はどうしたらいいか考えてみてください。

１２－１－５．宣言時における初期化（初期値代入）

通常の変数と同じように配列を宣言するときに、初期値を代入することができます。この場合、初期値の個数に応じて、データを入れる領域は自動的に確保されます。初期値は、波括弧　{}　で囲み、カンマ,を用いて区切ります。

▼配列への初期値代入の書式

```
型名      変数名 [ ] = { 初期値 , 初期値 , … } ;
```

初期値代入を伴う配列の宣言の例を以下に記述してみましょう。aは整数型の配列なので整数値が記述されています。また、bは実数型の配列なので実数値が記述されています。

```
int       a [ ] = { 10, 34, 82, 95, 3 };
double   b [ ] = { 0.56, 1.3e-5, 13.6e5, 1.45, 10.35, -1.3e-4, 0.003, 98.7 };
```

上の例では、aのサイズは5、bのサイズは8となる。これは次のように領域を確保して配列を宣言し、各要素に値を代入したのと同じになります。変数aの場合だけ記述してみましょう。

```
int       a [ ] = new int[ 5 ];
a[ 0 ] = 10;          a[ 1 ] = 34;          a[ 2 ] = 82;          a[ 3 ] = 95;          a[ 4 ] = 3;
```

初期値代入は、宣言と同時にしか行なうことができません。ですから、次のように記述するとコンパイルでエラーでになります。

```
int  twice [ ] ;
```

```
twice = { 324, 233, 112,  434, 745 };      // 宣言と一緒にやっていないので、エラーになります
```

なお、次のように**new**文を使えば、宣言時でなくても配列に初期値代入することができます。

```
int  advanced [ ];                                  // 別の場所で宣言された配列変数
advanced  =  new int [ ] {  10, 20, 50, 34 };       // new文と共に初期値代入が使えます
```

１２－１－６．初期値代入された配列のサイズを知るには？

配列には、lengthというフィールドがあり、このフィールドを使って初期値代入された配列などのサイズを知ることができます。プログラム中に繰返しなどで配列のサイズを整数の定数で指定する替わりにこれを用いることができます。

▼配列のサイズを求める書式

```
配列名 .length
```

あるいは、初期値代入された配列の場合には、いちいち数えなくとも、この書式を利用すれば、どのようなサイズの配列にも対応したプログラムを記述できます。次の例は初期値代入されたすべての要素を列挙表示します。この例では、配列aのサイズは5ですが、サイズがいくらになっても繰返しはうまく動作します。

```
int       a [  ] = { 10, 34, 82, 95, 3 };
for (  int   i =0;  i < a.length; i++ ) {
          System.out.println( i + "番目の値は" + a [ i ] + "です。" );
}
```

今までは、配列のサイズを整数値で記述してきましたが、これからはこの書式を使って記述することにします。この書式を使えば、プログラムを作成してから、後で配列のサイズだけ変更したい場合に、プログラムを修正する箇所が少なくて済みます。

１２－１－７．配列同志の代入

今までは、主に要素について代入を行なってきましたが、配列変数同志で代入を行なうこともできます。配列変数そのものを指定する場合には、角括弧[ ] を指定する必要はありません。

▼配列そのものを参照する書式

```
配列名
```

たとえば、２つの配列originalとaliasがあり、aliasにoriginalの配列の内容を代入するのには、次のように記述します。

```
int  original [ ]  = { 10, 20, 30, 40, 50 };
int  alias [ ] ;
alias = original ;                          // alisにoriginalの内容を代入しました
```

このように、配列同志の代入を行ないますと、結果として両方の配列は、同じ配列の実体を共有することになります。例えば、以下の式では配列変数aと配列変数bは同じ配列の実体を指しています。

```
int  a [ ]  = { 10, 20, 30, 40, 50 };
int  b [ ]  = a;
```

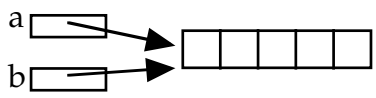


図12-7　共有の図

このような場合、プログラマが把握している分には、大丈夫かも知れませんが、次のように別々に代入をし始めると、一体どこで配列の要素が変更されたのかわかりにくくなります。

```
public class ArrayTest {
        public static void main( String [] arg ) {
                int a [] = { 10, 20, 30 ,40, 50 };
                int b [] = a;

                a[ 4 ] = 60;                    // 別々に要素に値を代入するが結果的には
                b[ 2 ] = 20;                    // 同じ配列に変更を掛けている
                for ( int i=0; i<a.length; i++ ) {
                        System.out.print( "a[ " + i + " ] = " + a[ i ] + "  " );
                }
                System.out.println( "" );
        }
}
```

この例の場合、結果としては、次のような表示がなされます。

```
a[ 0 ] = 10        a[ 1 ] = 20        a[  2 ] = 20        a[ 3 ] = 40        a[ 4 ] = 60
```

同じ内容を重複して持っておきたいときは、別々に配列の実体を用意して、次のように繰返しを使って各要素毎に代入をしておく必要があります（注１）。

```
int a [ ]  = { 10, 20, 30 ,40, 50 };
int b [ ] = new  int[  a.length  ];                    // 同じ個数分だけデータ領域を確保しておく

for ( i=0;  i < a.length; i++ ) {   b[ i ] = a[ i ]; }      // 要素を１つずつコピー
```

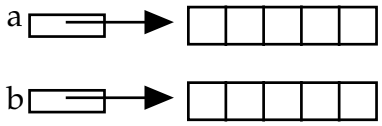


図12-8　コピーの図

コピーした場合は、別々に値を重複して持っていますので、一方の配列の要素の値を変更しても、それが他方に影響することはありません。

※注１　配列にはcloneという名前のメソッドが用意されています。これは、コピーを自動的に行なってくれるメソッドですので、繰返しを用いなくても、次のようにしてコピーをつくることができます。

```
int  b [] = ( int [ ] ) a.clone( );
```

１２－２．配列の典型的な使い方

配列を使って、特定の値が入っている要素の番号を求めたり、すべての要素の総和・平均を求めたり、最大値・最小値を求めたり、特定の値を持つ要素の個数をカウントしたりするのは、配列の典型的な使い方と言えます。また、配列の要素を画面に表示したり、グラフとして表示したりすることも良く行なわれます。ここでは、それらの典型的な例をいくつか紹介していきましょう。

１２－２－１．配列の要素を走査する

★配列のすべての要素の総和と平均を求めるアプリケーション。

以下のプログラムでは、変数*sum*の値を最初は0にしておいて、各要素の値を*sum*に足し込んでいます。また、平均は小数点以下の数も表示したいので、*sum*を一度実数型にして計算結果を実数として表示させています。

```
public class Summetion {
        public static void  main( String  [ ]  arg ) {
                int         sum, a [  ] =  { 5, 2, 565, 222, 111, 344, 22, 99, 348, 222 };
                sum = 0;
                for ( int  i= 0; i <a.length; i++ ) {
                        sum = sum + a[ i ];
                }
                System.out.println( "要素の値の総和は" + sum + "平均は" +
                        ((double) sum / a.length) + "です。" );
        }
}
```

★配列のすべての要素の中で、一番値の大きいものを表示する。

次のプログラムは、変数*max*に一番値が大きい「要素のインデックス」を保持しています。値そのものを保持していないことに注意してください。アルゴリズムとしては、最初の要素を仮に一番大きい値を保持していると仮定し、*max*の値を0にしています。配列を走査していき、それよりも大きな要素が現れたら、それを指すようにしています。走査が終わったら、*max*は一番大きい要素のインデックスを示しています。

```
public class SeekMaximum {
        public  static  void   main(  String  [ ]  args  ) {
                int         max;
                int         a  [ ] = { 5, 2, 565, 222, 111, 344, 22, 99, 348, 222 };

                max = 0;
                for ( int   i= 1; i < a.length ; i++ ) {
                        if ( a[ max ] < a[ i ] )  { max = i; }
                }
                System.out.println( "最大の要素は" + max  + "番目で、その値は"
                        + a [ max ] + "です。" );
        }
}
```

この方法は、「さらに良い条件の人が現れたら、そちらの人に乗り換える」という打算的な恋愛者のポリシーと同じです。そのような恋愛者は、各時点で、今まで現れた一番良い条件の人とつきあっている筈でしょう（理論的には）。

※最小値を求めるためにはどうしたらよいか考えてみてください。

★特定の値が何番目にあるかどうか判定する

あまり効率は良くないのですが、配列の最初から順番に要素を走査して、目的とする値と等しい値を持つ要素のインデックスを求めるプログラムです。変数*target*が目的の値を保持しています。同じ値の要素が現れたら、そこで繰返しを抜けています。もし、同じ値の要素が現れなかったら、配列の最後まで走査したことになりますから、繰返しの後で、条件文で最後まで走査したかどうかをチェックしています。そのために、繰返しで配列の走査に用いている変数*index*を、**for**文の前に宣言しています。こうすると、*index*は**for**文が終了した後も有効になります。

```
public class SeekTarget {
        public  static  void   main(  String  [ ]  args  ) {
                int         target = 22;
                int         a  [ ] = { 5, 2, 565, 222, 111, 344, 22, 99, 348, 222 };
```

```
                int  index = 0;
                for ( ; index < a.length; index++ ) {
                        if ( a[ index ] == target )  { break; }
                }
                if  (  index  < a.length  )  {
                        System.out.println( target  +  "は" + index + "番目にありました" );
                } else {
                        System.out.println( target  +  "はありません！" );
                }
        }
}
```

★特定の値が何個あるか数える

変数*target*が保持する値と同じ値の要素が幾つあるか数えるためのプログラムです。変数*count*が個数を保持しています。最初は0個にしておいて、同じ値の要素が現れる度に*count*の値を１つずつ増やしていきます。繰返しが終わった段階で、*count*には最終的な個数が求まっています。

```
public class ValueCounter {
        public  static  void   main(  String  [ ]  args  ) {
                int         target = 33;
                int         a  [ ] = { 5, 33, 565, 33, 111, 344, 22, 99, 348, 33 };

                int  count = 0;
                for ( int index = 0; index < a.length; index++ ) {
                        if ( a[ index ] == target )  { count++; }
                }
                System.out.println( target  +  "は" + count + "個ありました" );
        }
}
```

※上記プログラムをある値よりも大きい（小さい）要素が何個あるか数えるように、変えてみなさい。

１２－２－２．配列の要素の値を表示する

★すべての要素を表示するアプリケーション

端末画面に配列の要素を表示してみましょう。ただ単に、繰返しとSystem.out.printを使って要素を順番に表示しているだけです。

```
public class DisplayArrayOnConsole {
        public static void  main(  String [ ]  args ) {
                int         a [  ] = { 10, 34, 82, 95, 3, 4 ,5 ,1,32, -32, 1, 3, 12, 45, 11, 32, 19, -6, 23, 21, 33 };
                for  (  int   i =0;  i < a.length; i++ ) {
                        System.out.print( " " + a[ i ]  );
                        if ( i % 8 == 7 ) { System.out.println(  ); }
                }
        }
}
```

要素を8個表示するたびに、System.out.printlnメソッドを使って改行するようにしています。前に説明しましたように、繰返しの中で周期的に何かの操作を行ないたいときは、このような剰余演算と条件分岐を用います。

★すべての要素を表示するアプレット

カレンダー的に数をそのまま表示してみましょう。横に8個ずつ並べます。1つの数を表示するのに、幅30ドット、高さ15ドットを確保しています。y座標は20の位置から表示を始めることとします。

```
import  java.awt.*;
import  java.applet.*;

public class DisplayArray extends Applet {
        public void  paint(  Graphics  gc ) {
                int       a [  ] = { 10, 34, 82, 95, 3, 4 ,5 ,1,32, -32, 1, 3, 12, 45, 11, 32, 19, -6, 23, 21, 33 };
                int       width = 30,  height = 15;
                for (  int  i =0;  i < a.length; i++ ) {
                        gc.drawString( " " + a[ i ],  i % 8 * width , i / 8 * height + 20 );
                }
        }
}
```

要素の値を表示するx座標とy座標を技巧的に計算しています。剰余演算はこのように周期的にサイクルさせたいときに使います。また整数除算は、周期毎に１つ値を増やすような場合に使います。次の周期のときには、割り算の結果が１つ大きな値になりますから、表示されるy座標は次の段になります。

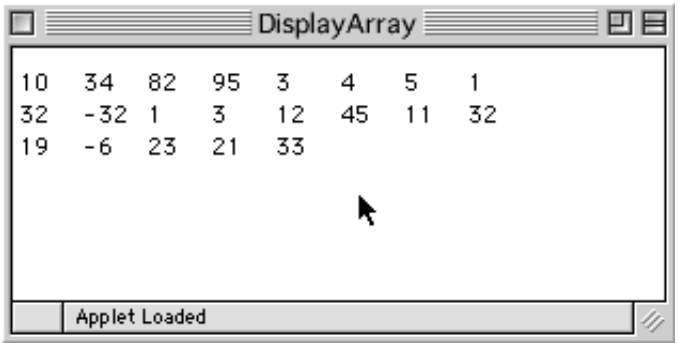


図12-9　表示結果

※逆順に表示したいときはどうしたらいいか考えてみてください。

★すべての要素を棒グラフとして表示するアプレット

横のグラフとして表示しています。１つの棒グラフを表示するのに、高さ5ドットを確保し、棒グラフと棒グラフの縦の間隔を10ドットとしています。値の範囲は、0から50までとし、横方向の長さを2倍して、最大100ドットの大きさで表示されるようにしました。

```
import  java.awt.*;
import  java.applet.*;

public class DisplayGraph extends Applet {
        public void  paint(  Graphics  gc ) {
                int       a  [  ] = { 10, 34, 42, 55, 3, 4 ,5 ,1,32, 5, 1, 3, 12, 45, 11 };
                for (  int  i =0;  i < a.length; i++ ) {
                        gc.fillRect(  10,  i*10 + 10,  a[ i ] * 2, 5 );
                }
        }
}
```

※横のグラフとして表示しましたが、縦方向にするにはどうしたら良いか考えてください。

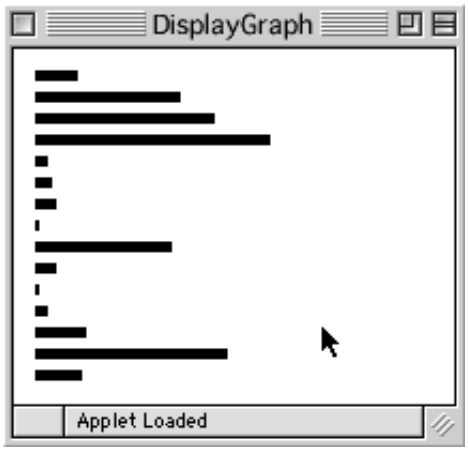


図12-10　表示結果

１２－２－３．頻度を数える

★特定の数の頻度を数える
配列の中にどのような数が入っているか、統計データとして調べたいことがあります。そのときに、ある数がどの程度出現したかを、別の配列に保存するやり方を考えてみましょう。たとえば、整数の配列birthの中に、ある統計調査で調べた誕生月が入っているとします。これを、それぞれの誕生月ごとに何人いるのか頻度を数えてみましょう。整数の配列monthは、それぞれの月について、何人いたかを示します。最初、この配列の要素はすべて０に初期化され（０人であることを示す）、そして、配列birthが走査されるにしたがって、該当する月の要素が+1されていきます。なお、わかりやすくするために、配列monthは、サイズを13にしておき、１～12までのインデックスの要素だけを使っています。すなわち、month[0]は使われていません。アプリケーションで記述しました。

```
public class FrequencyCounter {
        public static void  main( String [ ] args ) {
                int       birth [ ] = { 1, 3, 5, 2, 8, 11, 4, 6, 7, 9, 8, 10, 12, 1, 4, 5, 3, 2, 1, 8, 9, 5, 6, 7, 12, 4, 2};
                int       month [ ] = new int [ 13 ];

                for ( int i = 1; i < month.length; i++ ) { month[ i ] = 0; }

                for ( int index = 0; index < birth.length; index++ ) {
                        month[ birth[ index ] ]++;
                }
                for ( int i = 1; i < month.length; i++ ) {
                        System.out.println( i + "月の誕生日の人は" + month[ i ] + "人いました" );
                }
        }
}
```

★ある範囲の数の出現の頻度を表示する（度数分布表）
整数のデータが、配列dataに格納されているとします。このデータの範囲を区切るデータが配列partitionに入っているとします。partitionの各要素は、その値よりも等しいか小さな値であれば、データが、その範囲に入ると考えられます。ただし、partitionの要素は、小さい順で格納されており、１つ前の要素でそのデータが引っ掛からない場合に限り、次の要素で範囲にあるかどうか調べていくとします。partitionの最後の要素は、データが取りえる一番大きな値を持っており、この値を超えたら、カウントしないことにします。頻度は、配列frequentに数えることにします。この配列は、partitionと同じサイズにします。

たとえば、int [ ] partition = { 12, 56, 70 };だった場合は、0番目の範囲が12以下、１番目の範囲が13以上56以下、２番目の範囲が57以上70以下の３つ範囲に区切られます。データが、28が現れた場合は、１番目の範囲として分類されます。

アプレットとして、度数分布表として表示してみます。まず、度数を数えた後、区切りを数値で表示し（たとえば、上記の例の最初の範囲だったら、<= 12などという表示になります）、横棒で度数を示し、その値を表示しています。

```
import  java.awt.*;
import  java.applet.*;

public class FrequencyTable extends Applet {
        public void  paint(  Graphics  g ) {
                int       data  [  ] = { 10, 34, 42, 55, 33, 64 ,57 ,1,32, 59, 61, 3, 12, 45, 11 };
                int       partition [ ] = { 12, 56, 70 };
                int       frequent [ ] = new  int  [  partition.length ];
                for ( int i = 0; i < frequent.length; i++ ) { frequent[ i ] = 0; }

                for ( int index = 0; index < data.length; index++ ) {
                        for ( int i = 0; i < partition.length; i++ ) {
                                if  (  data[ index ]  <= partition[ i ] )  {  frequent[ i ]++;  break;  }
                        }
                }

                for  (  int   i =0;  i < frequent.length; i++ ) {
                        g.drawString(  "<=" +  partition[ i ] , 10,  i*20 + 15 );
                        g.fillRect(  50,  i*20 + 10,  frequent[ i ] * 2, 5 );
                        g.drawString(  ""+ frequent[ i ] ,  50 + frequent[ i ] * 2 + 5,  i*20 + 15 );
                }
        }
}
```

※partitionの配列の初期値代入をいろいろ変えてみて、どのような場合でも動くことを確認しなさい。定められた値の範囲で、配列dataのデータの個数を増やしてみなさい。また、配列frequentのサイズを１つだけ大きくし、予めpartitionで指定された最大の値よりも大きい値が現れた場合に、それをカウントするように変更してみなさい。

１２－３．グラフィックスと配列

１２－３－１．折れ線を描くメソッド

２つの配列の要素が、１つの点のx座標値、y座標値を持っていたとしましょう。そのような２つの配列を使って、多角形を表示するメソッドが用意されています。たとえば、次のように２つの配列に座標値が代入されているとします。

```
int       xarray [] = {  10, 30, 45, 23, 45, 65, 123, 45, 29, 30 };
int       yarray [] = {  50, 90, 23, 50, 92, 3, 12, 100, 111, 50 };
```

AWTクラスライブラリには、Graphicsクラスに次のような３つのメソッドが用意されています。

drawPolygon(x座標の配列 , y座標の配列 , 点の個数)	多角形を描画する
fillPolygon(x座標の配列 , y座標の配列 , 点の個数)	多角形を描画する（塗りつぶし）
drawPolyline(x座標の配列 , y座標の配列 , 点の個数)	折れ線を描画する

drawPolylineは、折れ線ですので閉じていません。その前の２つのは、閉じた多角形が表示されます。これを使ってみましょう。点の個数は、配列のサイズを使います。どちらの配列でもいいのですが、下の例ではx座標の配列のサイズを用いています。

```
gc.drawPolygon( xarray, yarray, xarray.length );
```

```
gc.fillPolygon( xarray, yarray, xarray.length );
gc.drawPolyline( xarray, yarray, xarray.length );
```

drawPolyineは、次のようなfor文とdrawLineメソッドで同じような折れ線を描くことができます。

```
for ( int i = 1;  i < xarray.length; i++ ) {
        gc.drawLine( xarray[ i-1 ], yarray[ i-1 ], xarray[ i ], yarray[ i ] );
}
```

１２－３－２．Polygonクラスのオブジェクト

AWTクラスライブラリでは、Polygonというクラスも用意されています。これは多角形の点を保持するためのオブジェクト用のクラスです。新しいオブジェクトを作る際には、２つの配列と点の個数を引数として用意します。たとえば、上の２つの配列を用いて、Polygonクラスのオブジェクトを１つ生成してみましょう。

```
Polygon mypolygon = new Polygon(  xarray, yarray, xarray.length  );
```

Polygonクラスのオブジェクトについても、多角形を描くdrawPolygonやfillPolygonがGraphicsクラスに用意されている他に、移動をするためのtranslateというメソッドがあります。これは、x座標の移動量、y座標の移動量を引数に取り、すべての点の座標を変更するためのものです。それでは、先ほどの２つの配列を使って、折れ線を描き、Polygonクラスのオブジェクトを作って、横に移動させ、多角形を表示させるようなアプレットを記述してみましょう。

```
import  java.awt.*;
import  java.applet.*;

public class LinkPoints extends Applet {
        public void paint( Graphics  gc ) {
                int       xarray [ ] = {  10, 30, 45, 23, 45, 65, 123, 45, 29, 30 };
                int       yarray [ ] = {  50, 90, 23, 50, 92, 3, 12, 100, 111, 50 };

                gc.drawPolyline( xarray, yarray, xarray.length );
                Polygon decimal = new Polygon(  xarray, yarray, xarray.length  );
                decimal.translate( 100,  0  );          // 右に100ドット移動する
                gc.drawPolygon( decimal );
                decimal.translate( 100,  0  );          // 更に右に100ドット移動する
                gc.fillPolygon( decimal );
        }
}
```

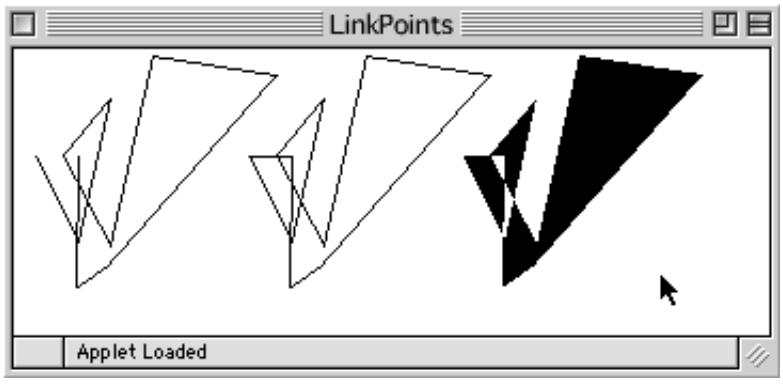


図12-11　表示結果

１２－４．２次元配列

１２－４－１．２次元配列

２次元配列は、配列の配列として定義します。まず、領域を確保するためには、第１次元の配列のサイズを指定します。同時に第２次元の方のサイズも指定することができます。第２次元の方のサイズは省略することもできます。省略された場合は、後で個々の要素に対して、第２次元のサイズを指定してデータ領域を確保する必要があります。

```
int  sample[ ] [ ] =  new  int [ 30 ][ 10 ];          // 第１次元のサイズは30、第２次元のサイズは10
int  later[ ][ ] =  new  int[ 25 ][ ];                // 第１次元のサイズは25だが、第２次元は指定しない
```

たとえば、個々の要素が九九の値（０も含みます）を持つような整数の２次元配列を作ってみましょう。第１次元のサイズが10、第２次元のサイズも10であるような配列を宣言することになります。

```
int  matrix[ ][ ]  = new int[ 10 ][ 10 ];
```

角括弧が２つあることが、２次元配列であることを示しています。この10×10で100個の要素を持つ配列に値を代入してみましょう。第１次元の方のサイズを得るためには、lengthフィールドをそのまま用いますが、第２次元のサイズを求めるために、第１次元の０番目の要素に対して、lengthフィールドを用いています。

```
for ( int i=0; i<matrix.length; i++ ) {
        for ( int  j=0; j<matrix[ 0 ].length; j++ ) {
                matrix[ i ][ j ] = i * j;
        }
}
```

２次元配列を初期化することもできます。初期値を代入する際は、波括弧{ }をネストさせて複数使うことになります。たとえば、次の２次元配列は、第１次元のサイズが３、第２次元のサイズが７に設定されます。

```
int  values [ ] [ ] = {  { 10, 9, 8, 4, 5, 6, 3 },  { 4, 3, 52, 11, -4, 2, 3 }, { 0, 29, 19, 39, 11, 25, 8 } };
```

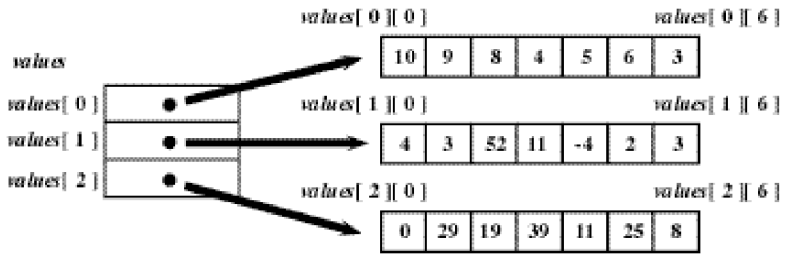


図12-12　２次元配列の構造

１２－４－２．可変長のサイズを持つ２次元配列

ところで、Java言語では第２次元のサイズが、一定でなくても構わないのです。たとえば、次のように初期化を行ないますと、第２次元のサイズが異なるような２次元配列を作ることができます。

```
int  hetero [ ] [ ] = { {10, 40, 10},  {343, 222} ,  { 789 },  { 198, 293, 923, 933 },  { 44, 55 } };
```

このような配列を画面に表示することを考えてみましょう。先ほどは、第１次元の０番目の要素に対してサイズを求めましたが、今度は第２次元のサイズがそれぞれ異なりますので、内側のfor文の継続条件のところでは、それぞれの要素に対してサイズを求めるように記述されています。また、それらの第１次元目の要素に属する第２次元目の要素をすべて表示し終わったら、改行するようにしています。

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

第１次元のサイズだけ先に指定おいて、要素毎に異なるサイズの第２次元のデータ領域を確保しながら、２次元配列を形成することもできます。たとえば、第２次元の要素の数が、第１次元のインデックスに１を足した形なるようなものを作ってみましょう。

```
int  linear [ ] [ ]  =  new  int[  10  ][ ];          // 第１次元の要素の数だけを指定する
for (  int  i=0 ; i < linear.length; i++ ) {
        linear[  i  ] = new  int [ i+ 1 ];            //  インデックス+1の個数分だけ第２次元の要素を用意
}
```

このようにした場合、最初の要素には１個、次の要素には２個、その次の要素には３個というように第２次元目の要素が用意されることになります。

２次元配列と同じようにして、３次元、４次元、あるいは５次元以上の配列を使うこともが可能です。たとえば、次の配列の宣言は一体何次元の配列を作っているのでしょうか？どこまでの次元のデータ領域を確保しているのでしょうか？

```
int  space [ ][ ][ ][ ][ ][ ] = new int [ 10 ][ 10 ][ 10 ][][][];
```

１２－５．課題

12-1.
この章の例題を逐次実行してみなさい。※のついた部分についても考えてみなさい。

12-2.
棒グラフを表示するアプレットの例題などを用いて、0から99の値の範囲で、20個の整数を乱数を用いて発生させ、一番小さい値を持つ棒グラフが赤色（Color.red）で、一番大きい値を持つ棒グラフが青色（Color.blue）で、それ以外の値が緑色（Color.green）で表示されるようなアプレット作成しなさい。クラス名は、GraphPresenterにて。

12-3.
整数型の変数nに、適当な数を代入します。このnが持つ値を、2進数として表示しなさい。2進数への変換は、2で割っていき、毎回割ったときの剰余（0か1になります）を、配列に格納していきます。割った値が0になるまで、この剰余を配列に格納していきます。そして、次に、この配列を逆に辿って、剰余だけを表示していくと、それが、その数の2進数標記になります。クラス名は、BinaryConverterにて。

ヒント：どこまで配列を使っているのか覚えておく変数を用意しておくと良いでしょう。

12-4.
度数分布表の例題で、一定の範囲ごとに区切っていく場合には、整数除算を使ってもっと簡単にできます。すなわち、配列partitionを用意しなくても済みます。いま、出現するデータの値の範囲が1〜100で、10ずつに区切って度数を数えていく場合についてアプレットを記述してみなさい。クラス名は、SimpleFrequencyにて。