

Chapter 13. 配列

13.1. 配列と要素

■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。



図13-1 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
```

■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ると、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。

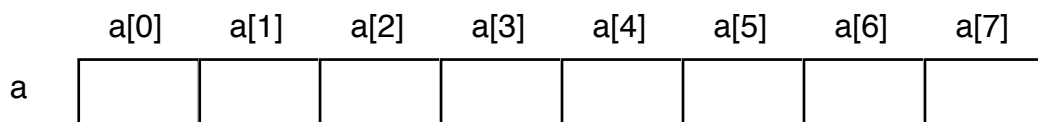


図13-2 変数と配列変数

■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[ 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ができます。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ムも多いのですが、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
```

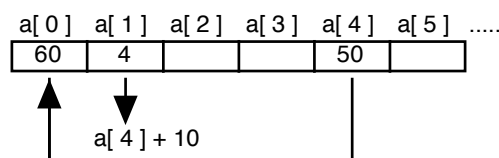


図13-3 配列の要素の値を添え字として使う

■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、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。

★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13-4 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13-5 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13-6 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。

■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
```

この制限はちょっと残念ですね。そこでJDK1.1からは言語が改良されて、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るようになりました。

```
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
```

■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代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。

■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には、次のように記述します。

```
int original [ ] = { 10, 20, 30, 40, 50 };
int alias [ ];
alias = original;                                // alias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;
```

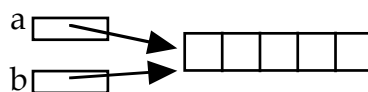


図13-7 共有の図

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

```

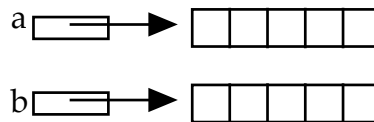


図13-8 コピーの図

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( );
```

13.2. 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。

■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 Summation {
    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" );
    }
}
```

■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 == 0 ) { 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な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。

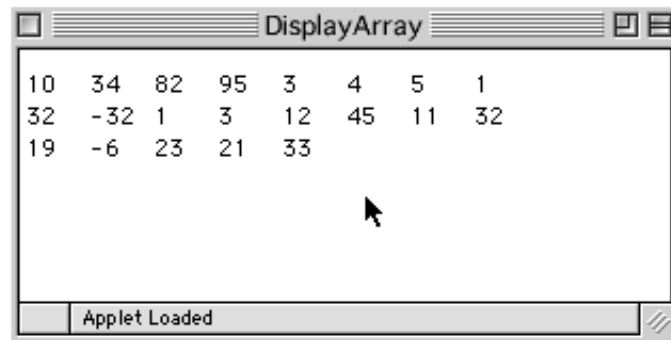


図13-9 表示結果

※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。

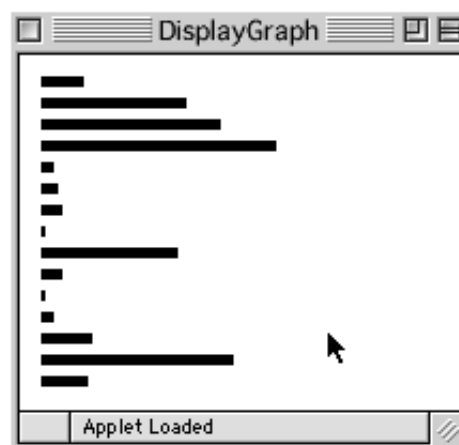


図13-10 表示結果

★Polygonクラスのオブジェクト

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 );
```

drawPolyline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 ] );
}
```

また、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 );
    }
}
```

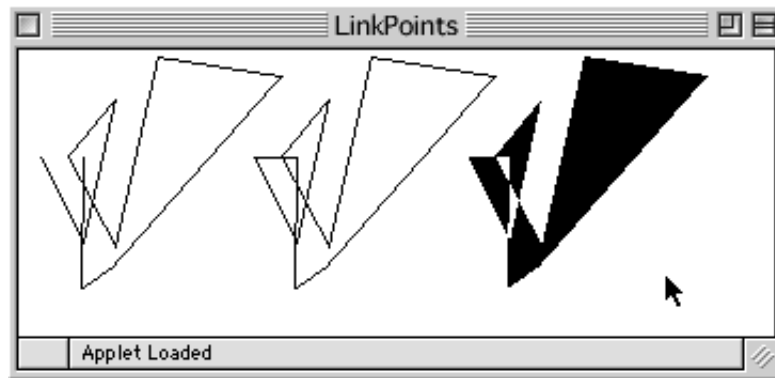


図13-11 表示結果

★配列で描いた線を覚えておく

前に、マウス操作で線を描画するようなアプレットを作りましたが、一本線を引く度に前に引いた線が消えてしまいます。配列を使って、前に引いた線を10本までは覚えておくようにしましょう。startx、starty、endx、およびendyがそれぞれ始点と終点のx、y座標値を覚えておくための配列になっています。変数currentは、今まで線を引いた本数を表しているのですが、同時に新しい線の座標値を覚えるための配列のインデックスとしても使われています。paintメソッドでは、今まで引いた本数分、配列の要素に覚えた座標値を元に線を描いています。なお、10本まで線を引いたら、currentを0にして御破算にしています。

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

public class LineDrawer extends Applet implements MouseListener, MouseMotionListener {
    int    startx [] = new int[ 10 ], starty [] = new int[ 10 ];
    int    endx [] = new int[ 10 ], endy[] = new int[ 10 ];
    int    current = 0;

    public void init( ) {
        addMouseListener( this );
        addMouseMotionListener( this );
    }

    public void paint( Graphics gc ) {
        for ( int i=0; i<current; i++ ) {
            gc.drawLine( startx[ i ], starty[ i ], endx[ i ], endy[ i ] );
        }
    }

    public void mousePressed( MouseEvent e ) {
        if ( current >= startx.length ) { current = 0; }
        startx[ current ] = e.getX( ); starty[ current ] = e.getY( );
        endx[ current ] = e.getX( ); endy[ current ] = e.getY( );
        current++;
        repaint();
    }

    public void mouseDragged( MouseEvent e ) {
        endx[ current-1 ] = e.getX( ); endy[ current-1 ] = e.getY( );
        repaint();
    }

    public void mouseClicked( MouseEvent e ) {}
}
```

```

    public void mouseReleased( MouseEvent e ) {}
    public void mouseEntered( MouseEvent e ) {}
    public void mouseExited( MouseEvent e ) {}
    public void mouseMoved( MouseEvent e ) {}
}

```

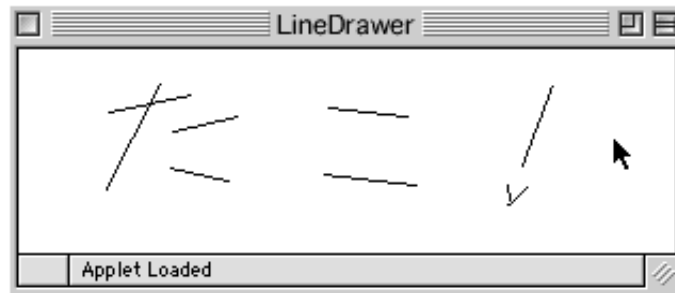


図13-12 10本まで線を描けるアプレットの実行例

13.3. 2次元配列とオブジェクトの配列

■ 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 } };

```

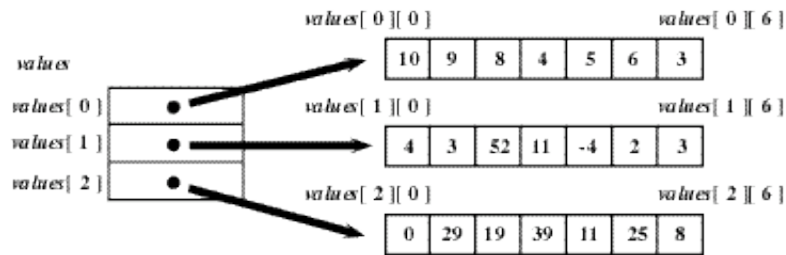


図13-13 2次元配列の構造

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 ][][];
```

■オブジェクトの配列

オブジェクトの配列を宣言するときは、次のように型名の代わりにクラス名を用います。上の方の書式は、オブジェクトの配列を宣言するだけです。下の方の書式は、オブジェクトの配列の実体を用意するものです。

▼オブジェクト配列の宣言の書式

```
クラス名 変数 [ ];
クラス名 変数 [ ] = new クラス名[ 配列のサイズ ];
```

たとえば、次の2つの配列は、それぞれカラーとボタン用の配列となります。

```
Color colors [ ] = new Color[ 20 ]; // 20個のカラーオブジェクト用の配列
Button operations [ ] = new Button[ 10 ]; // 10個のボタンオブジェクト用の配列
```

後は、通常の配列のように用いることができます。配列の要素を指定する場合は、通常のオブジェクトを参照している変数のように用いることができます。

```
colors[ 3 ] = new Color( 10, 30, 40 ); // 3番目の要素にカラーオブジェクトを生成
gc.setColor( colors[ 3 ] );
operations[ 4 ] = new Button( "Proceed" ); // 4番目の要素にボタンオブジェクトを生成
operations[ 4 ].addActionListener( this );
operations[ 4 ].setCommand( "proceed" );
add( operations[ 4 ] );
```

上の例をみてもわかるように、オブジェクトの配列は、オブジェクトそのものを生成したものではありません。**new**文では配列を1つ生成にただけに過ぎません。ですから、配列の要素となる個々のオブジェクトは、その都度繰返しなどを使って**new**文を使ってそれぞれを生成していく必要があります。これが、通常の整数などの配列と異なる点です。

★ボタンの配列を作る

ボタンの配列を使ってみましょう。次のアプレットは、カラーの配列とボタンの配列とを用いています。ボタンが押されたら、そのボタンに対応する色を用いて四角形を塗りつぶすようなプログラムになっています。変数`carrray`は、カラーの配列です。変数`barray`がボタンの配列になっています。変数`current`が、現在選択されている色の番号（インデックス）を保持しています。

`init`メソッドでは、配列のそれぞれの要素に、ボタンを**new**文で生成して割り当てています。配列の要素である個々のオブジェクトはこのように1つずつ生成しなければならないことに注意してください。ボタンには、例えばColor 10というようなラベルがつけられます。各ボタンには、インデックスと同じ番号がコマンドとして設定されています。ボタンが押されたときに呼ばれる`actionPerformed`メソッドでは、押されたボタンのコマンドを解析して、それを整数に変換しています。文字列を整数に変換する`Integer.parseInt`メソッドは、後の章で詳しく説明します。その整数が変数`current`に代入されて、色の番号を選択するために用いられています。

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

public class ButtonArray extends Applet implements ActionListener {
    Color carrray [ ] = { Color.red, Color.green, Color.blue, Color.magenta,
        Color.cyan, Color.yellow, Color.pink, Color.orange,
        Color.white, Color.black, Color.gray, Color.darkGray,
        Color.lightGray };
    Button barray [ ] = new Button[ carrray.length ];
    int current = 0;

    public void init() {
        for ( int i=0; i<barray.length; i++ ) {
            barray[ i ] = new Button( "Color "+ i );
            barray[ i ].addActionListener( this );
        }
    }
}
```

```

        bararray[ i ].setActionCommand( ""+ i );
        add( bararray[ i ] );
    }

    public void paint( Graphics gc ) {
        gc.setColor( carray[ current ] );
        gc.fillRect( 100, 100, 100, 100 );
    }

    public void actionPerformed((ActionEvent e) {
        current = Integer.parseInt( e.getActionCommand( ) );
        repaint();
    }
}

```

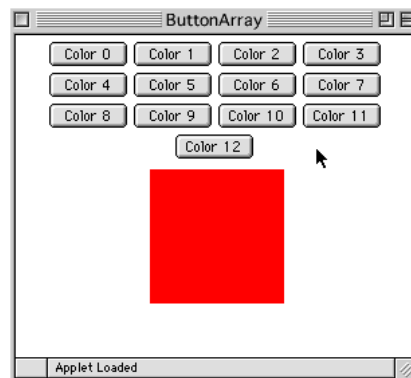


図13-14 ボタン配列アプレットの実行例

13.4. まとめと課題

■まとめ

★配列

- ・配列を宣言した後に、その実体を確保するには2通りあります
 1. **new**文でデータ領域を確保する
 2. 初期値代入で値を指定して領域を確保する
- ・個別のデータを処理するときは要素を参照します
- ・要素の参照の際にはインデックスを整数の式で指定します
- ・配列のサイズを得たいときには、lengthというフィールドを使います

★配列と繰返し

- ・配列は繰返しを使って、各要素を参照したり代入したりします
- ・次のような典型的な処理を行なえます
 - 総和を求める、平均を求める、最大・最小値を求める、
 - 特定の値のインデックスを求める、特定の値をカウントする

★2次元配列

- ・1次元と2次元のサイズをそれぞれ指定できます
- ・2次元配列の初期値代入では波括弧 {} をネストさせます
- ・1次元目の各要素では、2次元目のサイズは同一でなくても構いません

★オブジェクト配列

- ・**new**文では配列用のデータ領域は確保されても、個々のオブジェクトが生成される訳ではない
- ・個々のオブジェクトは、別の**new**文で1つ1つ生成していく必要があります

■課題

13-1.

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。

13-2.

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。

13-3.

10本の線を描くアプレットの例題を拡張して、ボタンを幾つか用意して、それ以降描画される線の色を変えるようにしなさい。

ヒント：色を覚えておくための配列を用意しておきます。