歴史的なコレクション・クラス – 配列

Java Collections より抜粋

John Zukowski

2001年 5月 01日

Apress の好意により

John Zukowskiの著書, Java Collections, (Apress) は、Java 2プラットフォームとともに提供されるCollection Libraryについて詳しく説明しています。この著書には、歴史的なコレクション・クラス、Collections Framework、および代替コレクション・ライブラリーに関するセクションが含まれています。ここに抜粋した記事 (「Historical Collection Classes」セクションの第1章) は、配列に関する詳細、つまり、配列を使用して行えること、および配列を使用する際の落とし穴を避ける方法を扱っています。

配列(Array) は、Javaプログラム言語で定義されている唯一のコレクション・サポートです。これは、その要素の集合を、指標 すなわち位置によってアクセス可能な順序で保管するオブジェクトです。配列は`Object` のサブクラスであり、`Serializable` インターフェースと`Cloneable` インターフェースの両方を実装します。ただし、内部動作を知る手掛かりとなるような .javaソース・ファイルはありません。基本的に、サイズと要素の型を指定して配列を作成し、要素を埋めていくことになります。

注: 配列は`Object` をサブクラス化するものであるため、配列変数で同期を取って、その変数の`wait()` および`notify()` メソッドを呼び出すことができます。

配列オブジェクトで何ができるのかを見てみましょう – まず、基本的な用法と宣言について説明し、さらにコピーと複製について説明します。また、配列代入、等価性検査、およびリフレクションについても説明します。

配列の基本

配列の宣言、作成、初期化、およびコピーについて詳しく説明する前に、簡単な配列の例を見てみましょう。Javaアプリケーションを作成する場合、`main()` メソッドは1つの引き数を取ります。この引き数は`public static void main(String args [])` というString配列です。コンパイラーには、使用される引数の名前はどうでもよいわけですが、それは`String` オブジェクトの配列であるということになります。

これでアプリケーションへのコマンド行引数が`String` オブジェクトの配列として与えられ、各要素を見たり印刷したりすることができます。Java内部では、配列はそれ自体のサイズを認識してい

て、常に位置ゼロから指標付けされます。したがって、配列の唯一のインスタンス変数、すなわち長さを調べることにより、その配列の大きさを知ることができます。次のコードは、そのための方法を示しています。

```
public class ArrayArgs {
  public static void main (String args[]) {
    for (int i=0, n=args.length; i<n; i++) {
      System.out.println("Arg " + i +":" + args[i]);
    }
  }
}
```

注: 配列指標の型は`long` にすることはできません。指標として使用できるのは負ではない整数だけであるため、配列の要素の数は実質的に2,147,483,648、すなわち2^{31} までに制限され、指標の範囲は0から2^{31}-1までに制限されます。

配列のサイズはループを経過しても中で変更されないため、`for (int i=0; i<args.length; i++)` のように、毎回のテストごとに長さを調べる必要はありません。実際、ループを経過するたびにカウント・アップするのでなく、`for (int i=args.length-1; i>=0; i -)` のようにカウント・ダウンしゼロ・テストをすると、ほとんどの場合で名目上は処理が高速になります。JDK 1.1および1.2リリースでは、カウント・ダウンする場合とカウント・アップする場合のパフォーマンスの差は相対的に小さいですが、1.3リリースでは、これらの時間的な差はかなり大きくなります。プラットフォームでの速度の差を調べるために、リスト2-1のプログラムを試して、「max int」回ループするための所要時間を計ってみてください。

リスト2-1. ループ・パフォーマンスの計測

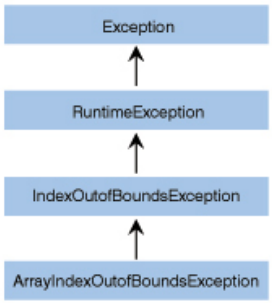
```
public class TimeArray {
  public static void main (String args[]) {
    int something = 2;
    long startTime = System.currentTimeMillis();
    for (int i=0, n=Integer.MAX_VALUE; i<n; i++) {
      something =- something;
    }
    long midTime = System.currentTimeMillis();
    for (int i=Integer.MAX_VALUE-1; i>=0; i-) {
      something = -something;
    }
    long endTime = System.currentTimeMillis();
    System.out.println("Increasing Delta: " + (midTime - startTime));
    System.out.println("Decreasing Delta: " + (endTime - midTime));
  }
}
```

このテスト・プログラムには配列アクセスが含まれないので、実際に計時されるのはforループであって配列アクセスではありません。

注: ほとんどの場合、私の400 MHz Windows NTシステムで計算された数値は、JDK 1.1および1.2では11,000台前半でした。しかしJDK 1.3で`-classic` オプションを指定した場合 (JITなし) には、計時値は約250,000に増加しました。1.3でHotSpot VMを使用した場合でも、数値は19,000から30,000までの間でした。

配列の先頭よりも前または配列の終わりよりも後にアクセスしようとすると、`ArrayIndexOutOfBoundsException` がthrowされます。`ArrayIndexOutOfBoundsException` は`IndexOutOfBoundsException` のサブクラスであり、図2-1に示すようにランタイム例外です。ありがたいことに、これはtry-catchブロック内に配列アクセスを配置する必要がないことを意味します。さらに、配列の境界を越えて検索することはランタイム例外ですので、プログラムのコンパイルはきちんと行われます。プログラムが例外をthrowするのは、アクセスが試みられたときだけです。

図2-1. `ArrayIndexOutOfBoundsException`のクラス階層



注: 配列境界の検査をオフにすることはできません。Javaランタイムのセキュリティー・アーキテクチャーでは、無効なメモリー・スペースにアクセスされないようになっています。

次のコードは、コマンド行の配列要素を読んでいくための不適切な方法を示しています。

```
public class ArrayArgs2 {
  public static void main (String args[]) {
    try {
      int i=0;
      do {
        System.out.println("Arg " + i + ": " + args[i++]);
      } while (true);
    } catch (ArrayIndexOutOfBoundsException ignored) {
    }
  }
}
```

これは、前のArrayArgsの例と機能的には同等ですが、制御フローのために例外処理を使用するプログラミングは不適切です。例外処理は、例外時のためにとっておくべきです。

配列の宣言と作成

配列は要素の集合を指標によってアクセス可能な順序で保管するオブジェクトである、ということを思い出してください。これらの要素は、`int` または`float` などのプリミティブ・データ型であっても、任意の型の`Object` であっても構いません。特定の型の配列を宣言するには、宣言に大括弧 ([ ]) を追加してください。

```
int[] variable;
```

配列宣言の場合の大括弧は、int[] variable、int []variable、およびint variable[] のように3種類の記述が可能です。最初のものは、変数の型がint[] であることを表しています。後の2つは、変数が配列であって、その配列の型が`int` であることを表しています。

注: このようなことを書くと、セマンティクスについて論じているように思えるかもしれません。しかし、複数の変数を宣言する場合、どの形式を使用するのかによって違いが生じます。`int[] var1, var2;` という形式は、`int` 配列である2 つの変数を宣言することになりますが、`int []var1,var2;` または`int var1[], var2;` という形式は、1つの`int` 配列と`int` 型の変数を宣言することになります。

配列を宣言すると、配列を作成してそれに対する参照を保管できるようになります。配列を作成するためには`new` 演算子が使用されます。配列を作成するときには、その長さを指定しなければなりません。この長さは、一度設定すると変更はできません。次のコードで示されているように、長さは定数として指定することも、式として指定することもできます。

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

または

```
int[] createArray(int size) {
  return new int[size];
}
```

注: 長さが負の配列を作成しようとすると、ランタイム例外`NegativeArraySizeException` がthrowされます。しかし、長さゼロの配列は有効です。

配列の宣言と作成を1つのステップに結合することができます。

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

警告: 配列作成の結果`OutOfMemoryError` がthrowされた場合には、すべての次元式がすでに計算されています。次元の記述の中で代入が実行されるときには、このことは重要です。たとえば、式`int variable[] = new int[var1 = var2*var2]` が原因で`OutOfMemoryError` がthrowされる場合、そのエラーがthrowされる前に変数`var1` が設定されます。

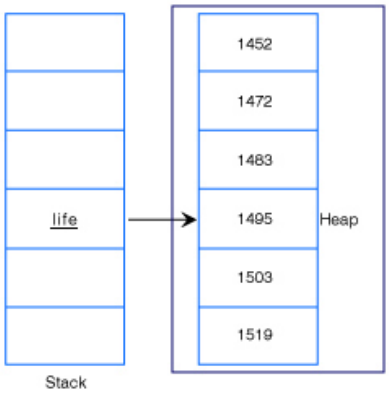
配列が作成されると、それに値を埋め込むことができるようになります。これは通常、for-loopまたは個別の代入文を使用して行われます。たとえば、次のコードは、名前を要素とする３要素の配列を作成し、値を割り当てるものです。

```
String names = new String[3];
names [0] = "Leonardo";
names [1] = "da";
names [2] = "Vinci";
```

プリミティブの配列

プリミティブ型の要素の配列を作成すると、配列はそれらの要素の実際の値を保持します。たとえば図2-2は、変数`life` から参照された6つの整数 (1452、1472、1483、1495、1503、1519) の配列で、スタックおよびヒープ・メモリーがどのようになるのかを示しています。

図2-2. プリミティブの配列のスタックおよびヒープ・メモリー



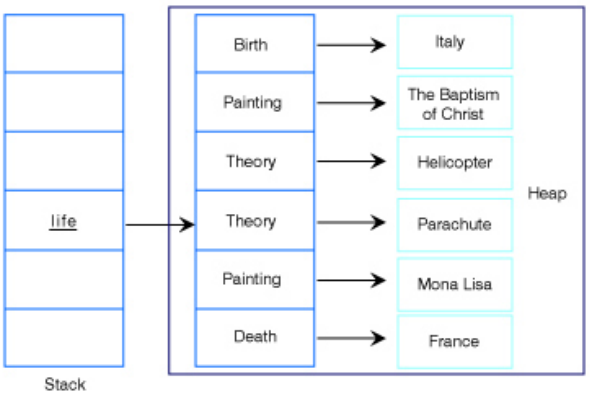
オブジェクトの配列

プリミティブの配列とは異なり、オブジェクトの配列を作成する場合、それらのオブジェクトは実際の配列には保管されません。配列が保管するのは実際のオブジェクトへの参照だけであり、最初はどの参照も、明示的に初期化されないかぎりヌルになります。(初期化については、後で簡単にふれます。)図2-3は、要素が次のものであるときに`Object`型の要素の配列がどのようになるのかを示しています。

- Leonardo da Vinciの生まれた国イタリア
- 彼の作品キリストの洗礼 のイメージ
- ヘリコプターとパラシュートに関する彼の理論 (スケッチ)
- モナリザ のイメージ
- 彼が亡くなった国フランス

図2-3で注意する必要があることは、オブジェクトが配列に入っていないということです。配列に入っているのはオブジェクトへの参照 だけです。

図2-3. オブジェクトの配列のスタックおよびヒープ・メモリー



多次元配列

配列は参照を介して取り扱われるため、ある配列要素から別の配列を参照することができます。ある配列が別の配列を参照すると、多次元配列 が得られます。この場合、追加されたそれぞれの次元に対応する追加の大括弧の組を宣言行に含める必要があります。たとえば、2次元配列である長方形を定義したい場合には、次のような行を使用します。

```
int coordinates[][];
```

1次元配列の場合と同じように、配列がプリミティブの配列である場合には、配列を作成した直後に値を配列に保管することができます。宣言するだけでは不十分です。たとえば、次の2行を使用すると、配列変数が初期化されないため、コンパイル時エラーが発生します。

```
int coordinates[][];
coordinates[0][0] = 2;
```

しかし、これらの2つのソース行の間に (`coordinates = new int [3][4];` のように) 配列を作成すると、最後の行は正しいものになります。

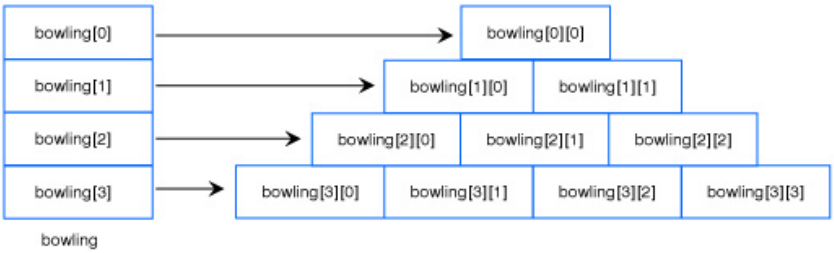
オブジェクトの配列の場合、多次元配列を作成すると、ヌル・オブジェクト参照だけが入った配列が作られます。配列に保管するオブジェクトも作成する必要があります。

多次元配列の最外部の各要素はオブジェクト参照ですので、配列が長方形 (あるいは、3次元配列の場合は直方体) である必要はありません。その内側の各配列は、独自のサイズにすることができます。たとえば、次に示す`float` の2次元配列の作成方法では、内側の配列のサイズはボウリングのピンのセットのようになっていて、最初の行には1つの要素、2番目には2つ、3番目には3つ、4番目には4つの要素が入ります。

```
float bowling[][] = new float[4][];
for (int i=0; i<4; i++) {
  bowling[i] = new float[i+1];
}
```

最後の配列のイメージを把握するために、図2-4を参照してください。

図2-4. 三角形のボウリングのピン状の配列



複数の次元の配列にアクセスするときには、それぞれの次元式が完全に計算された後で、その右にある次の次元式が計算されます。配列アクセス時に例外が発生した場合には、このことを理解しておく必要があります。

注: 構文上、配列変数が単一次元を表す場合、配列変数の後または前に大括弧を置くことができますが (`[index]name` または`name[index]`)、複数次元を表す配列変数の場合は、`name[index1][index2]` のようにその配列変数の後に大括弧を置かなければなりません。統語上、`[index1][index2]name` と`[index1]name[index2]` は正しくなく、コードでこのような指定が検出されるとコンパイル時エラーになります。宣言を行う場合には、大括弧を変数名の前に置いても (`type [][]name`)、後に置いても (`type name[][]`)、あるいは前後に置いても (`type []name[]`) 構いません。

コンピューター・メモリーがリニアであることを忘れないでください。多次元配列にアクセスするときには、実際にはメモリー内の単一次元配列にアクセスしていることになります。保管されているのと同じ順序でメモリーにアクセスできる場合には、アクセス効率は最も高くなります。通常、すべての要素がメモリ中にあるならば、コンピューター・メモリーはすぐに読み出せるので、このことは問題ではありません。ただし、大規模なデータ構造を使用する場合、リニア・アクセスはその威力を最大限に発揮し、不必要なスワッピングを回避します。さらに、多次元配列を1次元配列にパックすることによって、多次元配列をシミュレートすることもできます。画像データでは、これがよく行われます。1次元配列にパックする方法として、行優先順と列優先順の2つの方法があります。行優先順では1つの行が一度に配列に入り、列優先順では配列に列が入ります。図2-5は両者の違いを示しています。

図2-5. 行優先順と列優先順

注:`ImageFilter` クラスの`setPixels()` をはじめとする多くのイメージ処理ルーチンでは、2次元イメージ配列が行優先順に変換され、`Pixel(m, n)` は1次元位置n * scansize + mに変換されます。これにより、イメージ・データは、左から右への方向に、上から順に読み取られます。

配列の初期化

ある配列がはじめて作成されたときに、ランタイム環境は、配列の内容が何らかの既知の値 (未定義でない値) に自動的に初期化されるようにします。初期化されていないインスタンスおよびクラス変数の場合と同様に、配列の内容は、表2-1に示すように、数値の場合はゼロ、文字の場合は\u0000、ブールの場合はfalse、またオブジェクト配列の場合はヌルに初期化されます。

表2-1. 配列の初期値

デフォルト値	配列
0	byte short int long
0.0	float double
\u0000	char
false	boolean
null	Object

配列を宣言するときには要素の初期値を指定することができます。これは、宣言のところにある等号の後の中括弧 {} の中に、コンマで区切ったリストを含めることによって行います。

たとえば、次のコードは、3次元の名前の配列を作成します。

```
String names[] = {"Leonardo", "da", "Vinci"};
```

配列初期化指定子を提供するときには、長さを指定する必要はありません。配列の長さは、コンマで区切ったリスト内の要素の数に基づいて自動的に設定されます。

注: Java言語構文では、{"Leonardo", "da", "Vinci",} のように、配列初期化指定子ブロック内の最後の要素の後にコンマを付けることが許されます。これによって配列の長さが4に変更されることはなく、3のまま維持されます。この柔軟性は、主として、コード生成プログラムのための利便性を考慮したものです。

多次元配列の場合、追加されるそれぞれの次元ごとに、括弧の組を追加してください。たとえば、次のようにすると、年と出来事の6 X 2配列が作成されます。この配列は`Object` 型の要素の配列として宣言されているため、内部のそれぞれの`int` プリミティブ値を保管するために`Integer` ラッパー・クラスを使用する必要があります。配列内のすべての要素は、すべての要素がそのサブクラスだとしても、その配列の宣言された型、または、その型のサブクラス (この例では`Object`) になっていなければなりません。

```
Object events [][] = {
  {new Integer(1452), new Birth("Italy")},
  {new Integer(1472), new Painting("baptismOfChrist.jpg")},
  {new Integer(1483), new Theory("Helicopter")},
  {new Integer(1495), new Theory("Parachute")},
  {new Integer(1503), new Painting("monaLisa.jpg")},
  {new Integer(1519), new Death("France")}
};
```

注: 配列の型がインターフェースの場合、その配列内のすべての要素はそのインターフェースを実装していなければなりません。

2番目のドット・ポイントJavaリリース (Java 1.1) 以降、匿名配列 という概念が導入されました。配列の宣言時にそれを初期化するのは簡単に行えましたが、新しい配列を保管するための別の変数を宣言しないかぎり、コンマで区切ったリストを使用してその配列を再初期化することはできません。匿名配列が使用されるのはそのためです。匿名配列を使用すると、配列を新しい値の集まりに再初期化したり、上記のような配列を保管するためのローカル変数を定義したくないときに、名前の付いていない配列をメソッドに渡したりすることができます。

匿名配列は、通常の配列と同じように宣言されます。ただし、大括弧の中に長さを指定するのではなく、次に示すように、コンマで区切った値のリストを中括弧に入れて大括弧の後に指定してください。

```
new type[] {comma-delimited-list}
```

次の行は、メソッドを呼び出して、`String` オブジェクトの匿名配列のリストをそれに渡す方法を示しています。

```
method(new String[] {"Leonardo", "da", "Vinci"});
```

コード生成プログラムでは匿名配列が頻繁に使用されます。

配列引数と戻り値の引き渡し

配列が引数としてメソッドに渡されるときには、その配列への参照がやりとりされます。これにより、配列の内容を変更したり、メソッドの戻り時に配列に対して行われた変更を呼び出しルーチンに知らせたりすることができるようになります。さらに、参照が渡されるため、メソッド内で作成された配列を戻すことができ、そのとき、メソッドの実行後にガーベッジ・コレクターが配列のメモリーを解放することを心配する必要はありません。

配列のコピーと複製

配列を使用すると、多くのことが行えます。初期設定時の配列のサイズでは不十分になったときには、より大きな新規配列を作成し、元の要素をその新規配列での同じ位置にコピーする必要があります。また、配列を大きくする必要はないが、元の配列をそのまま残したままその要素を変更したい場合には、配列のコピーまたは複製を作成しなければなりません。

`System` クラスの`arraycopy()` メソッドを使用すると、ある配列から別の配列に要素をコピーすることができます。このコピーを行うときには、宛先配列を大きくすることができます。しかし、宛先配列を小さくすると、実行時に`ArrayIndexOutOfBoundsException` がthrowされます。`arraycopy()` メソッドは、次のように5つの引数 (それぞれの配列と開始位置に2つずつ、コピーする要素の数を表すために1つ) を取ります:`public static void arraycopy (Object sourceArray,int sourceOffset,Object destinationArray,int destinationOffset,int numberOfElementsToCopy)`。型の互換性のほかに必要な要件は、宛先配列のメモリーがすでに割り振られていることです。

警告: 異なる配列間で要素のコピーを行うときに、コピー元または宛先引数が配列でなかったり、それらの型に互換性がなかったりすると、ArrayStoreExceptionがthrowされます。配列が非互換であるとは、一方がプリミティブの配列で他方がオブジェクトの配列である場合、またはプリミティブの型が異なる場合、またはオブジェクトの型が代入不能な場合を言います。

リスト2-2では、整数配列を取り、その2倍の大きさの新規配列を作成しています。これは、次の例の`doubleArray()` メソッドによって行うことができます。

リスト2-2. 配列のサイズを2倍にする

```
public class DoubleArray {
  public static void main (String args ]) {
    int array1[] = {1, 2, 3, 4, 5};
    int array2[] = {1, 2, 3, 4, 5, 6, 7, 8, 9};
    System.out.println("Original size: " + array1.length);
    System.out.println("New size: " + doubleArray(array1).length);
    System.out.println("Original size: " + array2.length);
    System.out.println("New size: " + doubleArray(array2).length);
  }
  static int[] doubleArray(int original[]) {
    int length = original.length;
    int newArray[] = new int[length*2];
    System.arraycopy(original, 0, newArray, 0, length);
    return newArray;
  }
}
```

元の配列の長さを取得した後で、古い要素が新規配列の元の位置にコピーされる前に正しいサイズの新規配列が作成されます。配列のリフレクションについて学ぶと、このメソッドを一般化して任意の型の配列のサイズを2倍にすることがるようになります。

このプログラムを実行すると、次の出力が生成されます。

```
Original size: 5
New size: 10
Original size: 9
New size: 18
```

注:`arraycopy()` を使用して配列をコピーするときに、配列のサブセットをその配列内の別のエリアにコピーしたい場合には、元配列と宛先配列が同じであってかまいません。これは、オーバーラップする部分があっても有効です。

配列は`Cloneable` インターフェースを実装しているため、配列の領域をコピーするだけでなく、配列を複製することもできます。複製 をすると、同じサイズおよび型の新規配列が作成され、すべての古い要素が新規配列にコピーされます。この点がコピーとは異なります。コピーの場合には、宛先配列の作成とサイズ指定をユーザー自身が行わなければなりません。プリミティブ型要素の場合には、新規配列には古い要素のコピーが入りますので、古い要素に対して行われた変更はコピーに反映されません。ただし、オブジェクト参照の場合には、参照だけがコピーされます。したがって、両方の配列は同じオブジェクトを指します。そのオブジェクトを変更すると、変更内容は両方の配列に反映します。これは浅いコピー または浅い複製 と呼ばれます。

次のメソッドは1つの整数配列を取り、上記のような配列のコピーを戻します。

```
static int[] cloneArray(int original[]) {
  return (int[])original.clone();
}
```

配列の複製では、CloneNotSupportedException を`throw` する`Object` の`protected` メソッドを、実際に使う`public` メソッドで置き換えます。

配列の不変性

基礎になる配列構造がメソッドの呼び出し元によって変更されることが望ましくない場合には、メソッドから配列の複製を戻すと便利です。次の例のように、配列を`final` として宣言することができます。

```
final static int array[] = {1, 2, 3, 4, 5};
```

しかし、オブジェクト参照に`final` (特に、この場合には配列参照) を宣言しても、オブジェクトの変更は制限されません。制限されるのは、`final` 変数が参照する対象の変更だけです。次の行のようにすると、コンパイル・エラーが発生します。

```
array = new int[] {6, 7, 8, 9};
```

しかし、個々の要素の変更は完全に適格です。

```
array[3] = 6;
```

ヒント: 変更不能な配列をメソッドから「戻す」ためのもう1つの方法として、実際の配列ではなくその`Enumeration` または`Iterator` を戻すやり方があります。どちらのインターフェースを使用する場合にも、配列全体を変更にさらすことなく、あるいは配列全体のコピーを作成することなしに、個々の要素にアクセスして変更することができます。これらのインターフェースについては、Java Collections の後の章でさらに説明します。

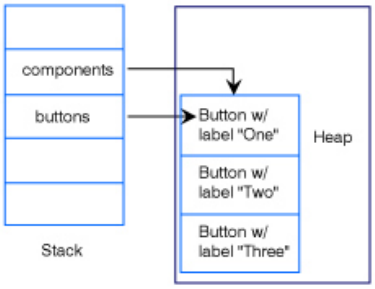
配列の代入

配列の代入は変数の代入と同じように行われます。変数xがyの配列への参照である場合、型zの変数がyに代入可能であれば、xはzへの参照にすることができます。たとえば、yがAWT`Component` クラスで、zがAWT`Button` クラスであるとします。`Button` 変数は`Component` 変数に代入することができるため、`Button` 配列を`Component` 配列に代入することができます。

```
Button buttons[] = {
  new Button("One"),
  new Button("Two"),
  new Button("Three")};
Component components[] = buttons;
```

このような代入を行う場合には、図2-6に示すように、両方の変数のボタンとコンポーネントが、メモリー内の同じヒープ・スペースを参照します。一方の配列の配列要素を変更すると、両方の配列の要素が変更されます。

図2-6. 配列割り当て後の共用メモリー



(前の例でボタン配列をコンポーネント配列変数に代入したように) 配列変数をスーパークラス配列変数に割り当てた後で、別のサブクラス・インスタンスをその配列に入れようとすると、`ArrayStoreException` がthrowされます。前の例を使用して説明すると、コンポーネント配列に`Canvas` オブジェクトを入れようとすると`ArrayStoreException` がthrowされます。コンポーネント配列が`Component` オブジェクトの配列として宣言されていても、コンポーネント配列は`Button` オブジェクトの配列だけを参照するため、`Canvas` オブジェクトを配列に保管することはできません。タイプ・セーフ・コンパイラーの観点からは実際の代入は適格であるため、これはランタイム例外です。

配列の等価性の検査

2つの配列が等価であるかどうかの検査は、求める等価性のタイプに応じて、2つのいずれかの方法で行うことができます。配列変数がメモリー内の同じ位置を指し、したがって同じ配列を指しているかどうかを調べるのか、それとも、2つの配列の要素を比較して等しくなっているかどうかを調べるのか、です。

同じメモリー・スペース内への2つの参照かを検査するには、2つの等号演算子 (==) を使用します。この場合、たとえば前のコンポーネントおよびボタン変数は、一方が他方への参照であるため、等価になります。

```
components == buttons  //true
```

ただし、ある配列を、その配列の複製バージョンと比較する場合、== を使用するかぎり、等価にはなりません。これらの配列は同じ要素を持っていますが、異なるメモリー・スペースに存在するため、異なる配列となります。配列の複製を元の配列と等価にするには、`java.util.Arrays` クラスの`equals()` メソッドを使用しなければなりません。

```
String[] clone = (String[]) strarray.clone();
boolean b1 = Arrays.equals(strarray, clone);   //Yes,they're equal
```

これにより、各要素の等価性が検査されます。引数がオブジェクトの配列である場合には、各オブジェクトの`equals()` メソッドが使用されて、透過性の検査が行われます。`Arrays.equals()` は、複製ではない配列に対しても使用できます。

配列のリフレクション

何らかの理由で、引数またはオブジェクトが配列であるのかどうか不明な場合には、そのオブジェクトの`Class` オブジェクトを検索して照会することができます。`Class` クラスの`isArray()` メ

ソッドを使用すると、これを調べることができます。配列であることが分かった場合には、Classのメソッドを使用して、実際にどの型の配列であるのかを調べることができます。isArray() メソッドが偽を戻すと、getComponentType() メソッドはヌルを戻します。それ以外の場合、要素のClass型が戻されます。配列が多次元配列である場合には、isArray() を再帰的に呼び出すことができます。この場合にもコンポーネントの型は1つだけです。さらに、java.lang.reflect パッケージにあるArray クラスのgetLength() メソッドを使用して、配列の長さを調べることもできます。

リスト2-3は、main() メソッドへの引数がjava.lang.String オブジェクトの配列であって、その長さが、指定されたコマンド行引数の数であることを示しています。

リスト2-3. 配列の型と長さを検査するためのリフレクションの使用

```
public class ArrayReflection {
  public static void main (String args[]) {
    printType(args);
  }
  private static void printType (Object object) {
    Class type = object.getClass();
    if (type.isArray()) {
      Class elementType = type.getComponentType();
      System.out.println("Array of: " + elementType);
      System.out.println(" Length: " + Array.getLength(object));
    }
  }
}
```

注:printType() が、前に定義されたbuttons およびcomponents 変数を使用して呼び出されたとすると、それぞれの配列はjava.awt.Button 型の配列になります。

isArray() およびgetComponentType() メソッドを使用しないで、ある配列に関するClass型を出力しようとすると、[の後に1文字とクラス名 (または、プリミティブの場合にはクラス名なし) が続いたものを含むストリングを受け取ります。たとえば、上のprintType() メソッドで型変数を出力しようとすると、class [Ljava.lang.String; という出力が得られます。

オブジェクトに対してそれが配列であるのかどうか、またどの型の配列であるのかを照会するほかに、java.lang.reflect.Array class を使用して実行時に配列を作成することもできます。これは、サイズの倍増のような配列タスクを実行する汎用ユーティリティー・ルーチンを作成する場合に役立ちます。(これについては、後で簡単に説明します。)

新規配列を作成するためには、Array のnewInstance() メソッドを使用してください。このメソッドには2種類のものがあります。単一次元配列の場合には、単純なほうのバージョンを使用します。これは、new type [length] というステートメントと似た働きをし、オブジェクトとして配列を戻します:public static Object newInstance(Class type, int length)。たとえば、次のように指定すると、5つの整数を入れることのできる配列が作成されます。

```
int array[] = (int[])Array.newInstance(int.class, 5);
```

注: プリミティブの場合にClass オブジェクトを指定するには、プリミティブ型名の最後に .classを追加してください。Integer.TYPEのように、ラッパー・クラスのTYPE変数を使用することもできます。

2番目の種類の`newInstance()`メソッドでは、整数の配列として次元を指定する必要があります:`public static Object newInstance(Class type,int dimensions [])`。最も単純な、単一次元配列の作成の場合には、1つの要素だけを指定して配列を作成することになります。つまり、5つの整数の同じ配列を作成する場合、5という整数値を渡す代わりに、次のように単一要素5を持つ配列を作成して、`newInstance()`メソッドに渡す必要があります。

```
int dimensions[] = {5};
int array[] = (int[])Array.newInstance(int.class, dimensions);
```

長方形の配列を作成したいだけであれば、各配列の長さで次元配列を埋めることができます。たとえば次の例は、整数の3 X 4配列を作成するのと同じ意味を持ちます。

```
int dimensions[] = {3, 4};
int array[][] = (int[][])Array.newInstance(int.class, dimensions);
```

ただし、長方形以外の配列を作成したい場合には、`newInstance()`メソッドを何回か呼び出す必要があります。最初の呼び出しは、外側の配列の長さを定義し、奇妙なクラス引数を持ちます(`float`の配列の場合には []class)。それに続く各呼び出しでは、内側の各配列の長さを定義します。たとえば、次に示す`float`の配列の作成方法では、内側の配列のサイズはボウリングのピンのセットのようになっていて、最初の行には1つ、2番目には2つ、3番目には3つ、4番目には4つの要素が入ります。このイメージを把握するためには、前に図2-4で示した三角形の配列を思い出してください。

```
float bowling[][] = (float[][])Array.newInstance(float[].class, 4);
for (int i=0; i<4; i++) {
  bowling[i] = (float[])Array.newInstance(float.class, i+1);
}
```

実行時に配列を作成すると、その配列の要素を取得および設定することもできるようになります。通常これは、ユーザーのキーボードの大括弧キーが使えない場合、あるいはダイナミック・プログラミング環境 (この環境では、プログラムの作成時には配列名が不明です) で作業している場合以外には行われません。表2-2に示すように、`Array`クラスには、配列要素を取得および設定するための一連のgetterメソッドとsetterメソッドがあります。どのメソッドを使用するかは、ユーザーが使用している配列の型に応じて異なります。

表2-2. 配列のgetterメソッドとsetterメソッド

getterメソッド	setterメソッド
get(Object array, int index)	set(Object array, int index, Object value)
getBoolean(Object array, int index)	setBoolean(Object array, int index, boolean value)
getByte(Object array, int index)	setByte(Object array, int index, byte value)
getChar(Object array, int index)	setChar(Object array, int index, char value)
getDouble(Object array, int index)	setDouble(Object array, int index, double value)
getFloat(Object array, int index)	setFloat(Object array, int index, float value)
getInt(Object array, int index)	setInt(Object array, int index, int value)
getLong(Object array, int index)	setLong(Object array, int index, long value)
getShort(Object array, int index)	setShort(Object array, int index, short value)

注:`get()` メソッドと`set()` メソッドは、いつでも使用できます。配列がプリミティブの配列である場合、`get()` メソッドの戻り値または`set()` メソッドへの値引数は、`int` 配列の場合における`Integer` のような、プリミティブ型に対応するラッパー・クラスにラップされます。

リスト2-4は、配列を作成し、埋め込み、またその配列に関する情報を表示するための方法を示す完全な例です。大括弧が使用されているのは`main()` メソッドの宣言だけです。

リスト2-4. 配列の作成、埋め込み、および表示のためのリフレクションの使用

```
import java.lang.reflect.Array;
import java.util.Random;
public class ArrayCreate {
  public static void main (String args[]) {
    Object array = Array.newInstance(int.class, 3);
    printType(array);
    fillArray(array);
    displayArray(array);
  }
  private static void printType (Object object) {
    Class type = object.getClass();
    if (type.isArray()) {
      Class elementType = type.getComponentType();
      System.out.println("Array of: " + elementType);
      System.out.println("Array size: " + Array.getLength(object));
    }
  }
  private static void fillArray(Object array) {
    int length = Array.getLength(array);
    Random generator = new Random(System.currentTimeMillis());
    for (int i=0; i<length; i++) {
      int random = generator.nextInt();
      Array.setInt(array, i, random);
    }
  }
  private static void displayArray(Object array) {
    int length = Array.getLength(array);
    for (int i=0; i<length; i++) {
      int value = Array.getInt(array, i);
      System.out.println("Position: " + i +", value: " + value);
    }
  }
}
```

これを実行した場合の出力は、次のものと似たようになります (ただし、乱数が異なります)。

```
Array of: int
Array size: 3
Position: 0, value: -54541791
Position: 1, value: -972349058
Position: 2, value: 1224789416
```

前に示した、配列のサイズを倍増させるメソッドの作成の例に戻りましょう。配列の型を取得する方法が分かりましたので、任意の型の配列のサイズを倍増させるメソッドを作成することができます。このメソッドは、長さと型を取得する前に、配列があることを確認します。その後で、新規インスタンスのサイズを2倍にしてから、元の要素のセットをコピーします。

```
static Object doubleArray(Object original) {
  Object returnValue = null;
  Class type = original.getClass();
  if (type.isArray()) {
    int length = Array.getLength(original);
    Class elementType = type.getComponentType();
    returnValue = Array.newInstance(elementType, length*2);
    System.arraycopy(original, 0, returnValue, 0, length);
  }
  return returnValue;
}
```

文字配列

Java配列についてまとめる前に、もう1つ注意すべきことがあります。CおよびC++ とは異なり、Javaの文字配列はストリングではありません。`String` コンストラクター (これは`char` オブジェクトの配列を取ります) と`String` の`toCharArray()` メソッドを使用することにより、`String` と`char[]` の間の変換は簡単に行えますが、両者はまったく異なるものです。

バイト配列の場合は少し異なります。バイト配列もストリングではありませんが、Javaにおけるストリングはユニコード・ベースであり、また16ビット幅であるため、`byte[]` と`String` の間で変換を行うためには、若干の作業が必要です。エンコード・スキームがどのようなものであるのかをStringコンストラクターに指示する必要があります。表2-3は、1.3プラットフォームで使用可能な基本的なエンコード・スキームを表しています。拡張セットのリストについては、http://java.sun.com/j2se/1.3/docs/guide/intl/encoding.doc.html にあるオンライン・リストを参照してください。これらは、JDKのバージョンによって異なります。

表2-3. バイトと文字間の基本的なエンコード・スキーム

名前	説明
ASCII	情報交換用米国標準コード
Cp1252	Windows Latin-1
ISO8859_1	ISO 8859-1、ローマ字アルファベット1
UnicodeBig	16ビット・ユニコード変換フォーマット、ビッグ・エンディアン・バイト順、バイト順マーク付き
UnicodeBigUnmarked	16ビット・ユニコード変換フォーマット、ビッグ・エンディアン・バイト順

UnicodeLittle	16ビット・ユニコード変換フォーマット、リトル・エンディアン・バイト順、バイト順マーク付き
UnicodeLittleUnmarked	16ビット・ユニコード変換フォーマット、リトル・エンディアン・バイト順
UTF16	16ビット・ユニコード変換フォーマット、必須の初期バイト順マークで指定されたバイト順
UTF8	8ビット・ユニコード変換フォーマット

エンコード方式を指定する場合には、try-catchブロック内に`String` コンストラクターへの呼び出しを置く必要があります。これは、指定されたエンコード・スキームが無効である場合に`UnsupportedEncodingException` がthrowされることがあるからです。

ASCII文字だけを使用している場合、このことはあまり気にする必要がありません。エンコード・スキーム引数を指定しないで`String` コンストラクターに`byte []` を渡すと、プラットフォームのデフォルト・エンコード方式が使用されますが、それで十分です。もちろん、安全を期して常にASCIIをスキームとして渡すことができます。

注: ユーザーのプラットフォームで使用されるデフォルトを検査するには、「file.encoding」システム・プロパティーを調べてください。

要約

Javaにおける配列は、一見簡単に使えそうですが、その機能を完全に利用するには、多くの注意が必要です。基本的な配列の宣言と使用は単純であると考えて差し支えありませんが、プリミティブの配列とオブジェクトの配列、および多次元配列を使用する場合には、さまざまな事柄を考慮する必要があります。配列の初期化は難しくないはずですが、匿名配列におけるthrowなどは、多少入り組んでいます。

配列が作成された後で、それを最大限に活用するためには、若干の配慮が必要です。メソッドに配列を渡す場合は、参照によって渡されますので、特別の注意が必要です。元の配列サイズでは足りなくなってしまった場合には、追加スペースを備えたコピーを作成する必要があります。配列の複製を行うと、`final` キーワードの特異性を気にせずにコピーを渡すことができます。他の変数に配列を代入するときには、`ArrayStoreException` が発生しないように注意してください。実行時にこの例外に対処するのは大変です。配列の等価性検査には、メモリー域が同じになっているかどうかの検査が必要な場合と、要素の値が同じになっているかどうかの検査が必要な場合とがあります。Javaリフレクションという魔術により、オブジェクトが配列であっても、手を加えることができます。この章の最後で学んだことは、バイト配列とストリングの間の変換を行う方法、および内容が異なる言語なので、バイト配列はデフォルトではストリングでないということでした。

関連トピック

- Java Collections に関する情報を読み、同書を発注してください。Apress Webサイトで他の出版物やイベントに関する情報を参照してください。
- 愛用のプラットフォーム向けのIBM developer kit をダウンロードしてください。
- 公式のCollections Frameworkの概要を参照してください。
- Annotated Outline of Collections Framework をご覧ください。