


欲しいものが
きつと見つかります



[> 今すぐチェック](#)
プライバーについて

Java可変長引数

JDK1.5から、引数の個数が不定（可変）なメソッドを定義し、呼び出せるようになった。[2007-06-30]
この機能を利用して[printf\(\)](#)が作られたし、[リフレクションの引数指定](#)も変更された。

- 例 [2007-06-30]
 - 定義方法 [2007-06-30]
 - 呼出方法 [2008-06-22]
 - 配列による呼び出し [\[/2008-06-27\]](#)
 - 同様な形式のオーバーロード [2008-06-22]
 - 配列から可変長引数への変更 [\[/2008-07-03\]](#)
 - [リフレクション](#)
 - [isVarArgs\(\)](#)
 - [main\(\)](#)の追加仕様
 - [@SafeVarargs](#)
 - Sunの[可変引数](#)
- [他言語の可変引数](#)

定義例と呼び出し例

引数の型の直後に「...」（ピリオド3つ）を付けると、そのメソッドを呼び出す側はその型の引数をいくつでも書けるようになる。[2007-06-30]
ただし、この可変引数は各メソッドにつき1つだけ、そのメソッドの最後の引数にだけ、指定可能。

定義例：

```
void method(String... args) {  
}
```

```
void method2(String str, int... ns) {  
}
```

呼び出し例：

```
method("abc");  
method("abc", "def");  
method("abc", "def", "ghi");  
method();           ...引数無しも可
```

```
method2("foo");  
method2("zzz", 123);  
method2("goo", 987, 654);
```

定義側の使用法

可変長引数を定義した側（呼び出された側）は、その引数は[配列](#)のようにして扱う。[2007-06-30]

定義例：

```
void method(String... args) {  
    System.out.println(args.length);  
    for (String s : args) {  
        System.out.println(s);  
    }  
}
```

定義の（コンパイルの）実態：

実際、コンパイルすると、可変長引数の実態は[配列](#)になる。

```
void method(String[] args) {  
    System.out.println(args.length);  
    for (int i = 0; i < args.length; i++) {  
        String s = args[i];  
        System.out.println(s);  
    }  
}
```

なので、コンパイル後のメソッドとしては区別が付かない（同時に定義することは出来ないが）。
違いは、[リフレクションのisVarArgs\(\)](#)の返り値のみ。

→[リフレクションで可変長引数メソッドを取得する方法](#)

呼び出し側の実態

Javaの可変長引数の定義の実体は配列なので、これらは、以下のようにコンパイルされる。[2008-06-22]

呼び出し例：

```
method("abc");  
method("abc", "def");  
method("abc", "def", "ghi");  
method();
```

呼び出しの（コンパイルの）実態：

```
method(new String[] { "abc" });  
method(new String[] { "abc", "def" });  
method(new String[] { "abc", "def", "ghi" });  
method(new String[] {} );
```

配列による呼び出し

可変長引数の実体が配列なので、配列のまま呼ぶことも出来る。[2008-06-22]

```
String[] args = { "123", "456", "789" };  
method(args);
```

ただし、Object型の可変長引数の場合、Objectの配列自体を1つのデータとして渡したい場合が出てくるかもしれない。[2008-06-27]
その場合は、明示的にキャストしてやればよい。

```
void method(Object... args) {  
    ~  
  
    method(new Object[] { "abc", "def" });           //(1)…abc,defという2つのデータが渡る  
    method((Object) new Object[] { "abc", "def" });  //(2)…{abc,def}という1つのデータ（配列）が渡る  
    method((Object[]) new Object[] { "abc", "def" }); //(3)…abc,defという2つのデータが渡る \(1\)と同様  
    method(new Object[] { "abc" }, new Object[] { "def" }); //(4)…{abc},{def}という2つのデータ（配列）が渡る  
  
    Object[] objs = new Object[] { "abc", "def" };  
    method(objs);           //\(1\)と同様  
    method((Object)objs);   //\(2\)と同様  
    method((Object[])objs); //\(3\)と同様  
  
    Object obj = new Object[] { "abc", "def" };  
    method(obj);           //\(2\)と同様  
    method((Object)obj);   //\(2\)と同様  
    method((Object[])obj); //\(3\)と同様
```

```
void caller_obj(Object arg) {  
    method(arg);           //\(2\)と同様  
}
```

```
void caller_arr(Object[] arg) {  
    method(arg);           //\(3\)と同様  
}
```

Javaでは基本的にコンパイル時のクラス指定でなく [実行時のインスタンスのクラスに応じて動作が決まる](#)ので、キャスト（コンパイル時のソース上のクラス）によってコンパイル時に動作が決定されるのはちょっと違和感があるな…。

Object以外の場合は区別がつかうので、このようなキャストは不要。

（例えばmethod(String... args)の場合、引数（データ）はStringのみであり、String[]はデータとしては有り得ない。

Objectだとデータは何でもいいので、Object[]も含んでしまうところがポイント。しかも「Object...」はけっこう使われそうだし。Object[]をデータとすることは少ないような気もするけど）

同じ形になるオーバーロード

呼び出し側の引数の並びが同じ形になるメソッドを定義（オーバーロード）することも出来る。[2008-06-22]

```
void method(String arg1) {           //…\(1\)  
}  
void method(String arg1, String arg2) { //…\(2\)  
}  
void method(String... args) {        //…\(3\)  
}
```

「method("abc")」という呼び出しは、[\(1\)](#)にも[\(3\)](#)にも合致する。
「method("abc","def")」という呼び出しは[\(2\)](#)にも[\(3\)](#)にも合致する。
しかしこういう場合は可変長引数でない方が呼び出される。

つまり、「method("abc")」という呼び出しは[\(1\)](#)が呼ばれ、
「method("abc","def")」という呼び出しは[\(2\)](#)が呼ばれる。

しかしこれは、あくまで呼び出し側のクラスのコンパイル時点で決定されるもの。

定義側で新しいオーバーロードが増えたり減ったりした場合に、自動的に（呼び出し側のリコンパイル無しで）呼び出されるメソッドが変更されるわけではない。

考えてみれば当然の話。

可変引数の実体が配列なので、コンパイル時点で完全一致するオーバーロードが無い場合に、コンパイラーは可変引数用に（引数を配列にした形で）コンパイルする。
実行時に呼び出される側のオーバーロードが増えていたとしても、呼び出し側は配列の形式になっているのだから、可変長引数のメソッドが呼ばれるしか無い。

逆に、配列を引数にとるオーバーロードを作ることは出来ない。

```
void method(String[] args) {           //…\(4\)  
}
```

[\(3\)](#)も[\(4\)](#)も、実体はどちらも配列なので、区別が付かない。
「重複したメソッドを定義しようとしている」というコンパイルエラーになる。

配列から可変長引数への変更

元々のメソッドが配列を引数にとるようになっていた場合、簡単に可変長引数に変更することが出来る。[2008-06-22]

```
void method(String[] args) {  
    System.out.println(args.length);  
    ~  
}
```

↓

```
void method(String... args) {           ←引数の定義を変えるのみ  
    System.out.println(args.length);  
    ~  
}
```

Javaの可変長引数では 実体は配列になるので、呼び出し側はリコンパイルせずにそのまま使える。（元から配列を使った呼び出しなので）

ただし、呼び出し側がnullを指定していた場合、それをリコンパイルすると「呼び出しが曖昧である」という警告になる。[2008-07-03]
なぜなら、「配列がnull（引数が無い）」という意味だったのに、「nullが1つというデータ」とも受け取れるから。

この場合は、明示的にキャストするか、引数を無くしてしまうのがよい。

```
method(null);
```

↓


```
method((String[])null);  
あるいは  
method();
```

```
method((String)null);  ...引数が1つ(値はnull)という呼び出しになる
```


と思ったが、可変長引数は引数が1つも無い場合は“要素が0個の配列”になるのであってnullになるわけじゃないから、仕様上全く同じというわけじゃないなあ。
つまり、配列から可変長引数に変えた場合は、引数がnullの場合の動作も変更前と同じになるように考慮しないといけない、という事かな。

[Java目次へ戻る](#) / [新機能へ戻る](#) / [技術メモへ戻る](#)

[メールの送信先：ひしだま](#)



欲しいものが
きつと見つかります



[> 今すぐチェック](#)
プライバシーについて