Javaの理論と実践: Generics、了解！

Genericの使い方を学ぶ上での落とし穴を知り、回避する

Brian Goetz
Principal Consultant
Quiotix

2005年 1月 25日

JDK 5.0で追加されたgenericタイプは、Java言語でのタイプ・セーフにとって大きな改善です。ところがgenericは、初めて使う人にとっては間違えやすく、極端に言えば奇妙に見えるかも知れません。今回のJavaの理論と実践では、Brian Goetzがgenericを初めて使う人が陥りがちな落とし穴について解説します。

このシリーズの他の記事を見る

Genericタイプ（またはgeneric）は、構文においても、想定されるユース・ケース（例えばコンテナー・クラスなど）においても、表面的にはC++のテンプレートに似ています。ところが、似ているのは、あくまでも表面だけです。Javaでのgenericは、ほとんど完全にコンパイラーの中で実装されます。コンパイラーがタイプ・チェックとタイプ推論(type inference)を行い、その後で通常の、非genericなバイトコードを生成するのです。この実装手法は消去（erasure）と言われますが、驚くべき、そして時には混乱を招きそうな結果をもたらします。（消去（erasure）では、コンパイラーがgenericタイプ情報を使ってタイプ・セーフを確保し、その後で、バイトコード生成前にそのタイプ・セーフを消し去ります。）genericはJavaでのタイプ・セーフにとって大きな前進ですが、その使い方を学ぶためには、まず確実に、頭を掻きむしる必要があるでしょう（あるいは呪いの言葉も必要かも知れません）。

注意: この記事は、JDK 5.0でのgenericの基本に慣れていることを想定しています。

Genericはcovariantではない

コレクションを配列の抽象化であると考えると分かりやすい、と思う人がいるかも知れませんが、配列はコレクションにはない、特別な特性があります。Javaでの配列はcovariantです。つまり、もし`Integer`が`Number`を継承すれば（実際、継承します）、`Integer`も`Number`であるだけではなく、`Integer[]`も`Number[]`であり、`Number[]`が呼ばれるところでは、自由に`Integer[]`を渡したり割り当てたりすることができます。（より正式に言えば、もし`Number`が`Integer`のスーパータイプであるならば、`Number[]`は`Integer[]`のスーパータイプです。）皆さんはgeneircタイプでも同じだろう、と思うかも知れません。つまり、`List<Number>`は`List<Integer>`のスーパータイプであ

り、List<Number>が想定されているところではList<Integer>を渡すことができる、と思うかも知れません。ところが、そうは行かないのです。

そう行かないのには、立派な理由があるのです。つまり、そうしてしまうと、genericが提供するはずのタイプ・セーフを壊してしまうことになります。List<Number>にList<Integer>を割り当てられる、と仮定してみてください。そうだとすると、次のコードを使って、Integerではないものをを入れることができてしまいます。

```
List<Integer> li = new ArrayList<Integer>();
List<Number> ln = li; // illegal
ln.add(new Float(3.1415));
```

lnはList<Number>なので、そこにFloatを加えても全く問題ないように思えます。ところが、もしlnがliと別名をつけられていたら、liの定義（整数のリストである、という定義）で暗黙的に約束されている、タイプ・セーフを破ってしまうことになります。これが、genericタイプがcovariantになり得ない理由なのです。

Covariantにまつわる、さらなるトラブル

配列はcovariantですがgenericはcovariantではない、という事実の結果としてもう一つ、タイプ引き数が制限のないワイルドカードでない限り、genericタイプの配列をインスタンス化することができません。（List<String>[3] は不正です。new List<?>[3] は問題ありません）genericタイプの配列宣言が許されるとしたら何が起きるかを見てみましょう。

```
List<String>[] lsa = new List<String>[10]; // illegal
Object[] oa = lsa;  // OK because List<String> is a subtype of Object
List<Integer> li = new ArrayList<Integer>();
li.add(new Integer(3));
oa[0] = li;
String s = lsa[0].get(0);
```

List<String>であるべきはずのものにList<Integer>を押し込んだので、最後の行はClassCastExceptionを投げます。配列covarianceを使うとgenericのタイプ・セーフを破壊できてしまうので、genericタイプの配列のインスタンス化は許可されていません（ただし、タイプ引き数がList<?>のような制限のないワイルドカードであるタイプは除きます）。

コンストラクションの遅れ

Erasure（という仕組み）のために、List<Integer>とList<String>は同じクラスであり、（C++の場合とは異なり）コンパイラーがList<V>をコンパイルする時には、一つのクラスしか生成しません。ですからコンパイラーがList<V>クラスをコンパイルする時には、コンパイラーはvがどんなタイプを表すかを知りません。ですから、表現されるクラスを知っていればできたはずのことが、List<V>のクラス定義にあるタイプ・パラメーター（List<V>のv）を使ったのではできない、という場合があります。

ランタイムはList<String>とList<Integer>を区別できないので（ランタイムでは、どちらも単にListです）、genericタイプ・パラメーターで識別されるタイプを持つ変数をコンストラクトすると、問題を起こしがちです。このように、ランタイムにはタイプ情報が無いため、genericコンテナー・クラスや、ディフェンシブ・コピー(defensive copy)をしようとするgenericクラスにとっては問題になります。

Fooというgenericクラスを考えてみてください。

```
class Foo<T> { public void doSomething(T param) { ... }
}
```

doSomething()メソッドが、入る時にparam引き数のディフェンシブ・コピーをしたい、としましょう。あまり選択肢はありません。皆さんは次のようにdoSomething()を実装したいと思うでしょう。

```
public void doSomething(T param) { T copy = new T(param);// illegal
}
```

ところが、タイプ・パラメーターを使ってコンストラクターにアクセスすることはできません。なぜなら、コンパイル時にはどんなクラスがコンストラクトされているのかが分からず、従ってどんなコンストラクターが使えるかが分からないからです。genericを使って、「Tはコピー・コンストラクターを持つ必要がある」といった制約を表現する方法はありません（あるいは、引き数のないコンストラクターであっっても同じです）。ですから、genericタイプ・パラメーターで表現されるクラスに対するコンストラクターにはアクセスできないのです。

ではclone()はどうでしょう。TがCloneableを拡張するようにFooが定義されていると仮定しましょう。

```
class Foo<T extends Cloneable> { public void doSomething(T param) {
    T copy = (T) param.clone();  // illegal }
}
```

残念ながら、やはりparam.clone()を呼ぶことはできません。なぜでしょう？clone()はObjectに対して保護されたアクセスがあり、clone()を呼ぶためには、clone()をパブリックとすべくオーバーライドしたクラスへの参照を通して呼ぶ必要があるのです。ところが、Tがclone()をパブリックとして再宣言することは分かりません。ですからクローン作成も、やはり不可なのです。

ワイルドカード参照をコンストラクトする

分かりました。つまり、コンパイル時にクラスが全く分からないタイプへの参照をコピーすることはできない、ということです。ではワイルドカード・タイプはどうでしょう？ タイプがSet<?>であるパラメーターのディフェンシブ・コピーをしたいとしましょう。皆さんは、Setにはコピー・コンストラクターがあることを知っているでしょう。また、setの内容のタイプが分からない時には、生のSetタイプよりもSet<?>を使った方が良い（そうした方が変換未チェック警告（unchecked conversion warnings）の出方が少ない）、と言われてきたでしょう。ですから、次を試したくなるかも知れません。

```
class Foo {
  public void doSomething(Set<?> set) {
    Set<?> copy = new HashSet<?>(set);  // illegal
  }
}
```

残念ながら、（皆さんはそうしたコンストラクターが存在することを知っているかも知れませんが）ワイルドカード・タイプ引き数でgenericコンストラクターを呼び出すことはできないのです。ただし、次は可能です。

```
class Foo {
  public void doSomething(Set<?> set) {
    Set<?> copy = new HashSet<Object>(set);  }
}
```

この構成体は、あまり目立たないかも知れませんがタイプ・セーフであり、`new HashSet<?>(set)`がするであろうと思われることをするのです。

配列をコンストラクトする

`ArrayList<V>`はどのように実装すればよいのでしょう？`ArrayList`クラスは`V`の配列を操作するはずなので、皆さんは、`ArrayList<V>`に対するコンストラクターが`V`の配列を作る、と思われるかも知れません。

```
class ArrayList<V> {
  private V[] backingArray;
  public ArrayList() {
    backingArray = new V[DEFAULT_SIZE]; // illegal
  }
}
```

しかし、このコードではうまく行きません。タイプ・パラメーターで表現されたタイプの配列をインスタンス化することはできないのです。コンパイラーには、タイプ`V`が実際に何を表すのかが分かりません。ですから`V`の配列をインスタンス化することはできないのです。

Collectionsクラスでは、汚い方法、つまりCollectionsクラスをコンパイルする時には変換未チェック警告を生成する、というトリックを使って、この問題を回避しています。`ArrayList`を本当に実装するコンストラクターは次のようなものです。

```
class ArrayList<V> {
  private V[] backingArray;
  public ArrayList() {
    backingArray = (V[]) new Object[DEFAULT_SIZE]; }
}
```

`backingArray`がアクセスされた時に、このコードが、`ArrayStoreException`を生成しないのはなぜでしょう？ 結局、`Object`配列を`String`配列に割り当てることはできないのです。つまりgenericはerasureによって実装されるので、そしてObjectは`V`のerasureなので、`backingArray`のタイプは、実は`Object[]` なのです。これはつまり、クラスはとにかく`backingArray`が`Object`の配列であると想定するのですが、コンパイラーが追加のタイプ・チェックを行い、必ずタイプ`V`のオブジェクトのみを含むようにする、ということです。ですからこの手法は動作しますが、美しくはありませんし、真似すべきものでもありません（generic化Collectionsフレームワークを作った人達ですら、そう言っています・・・参考文献を見てください）

別の方法としては、`backingArray`を`Object`の配列として宣言し、`backingArray`が使われるところでは常に`backingArray`を`V[]` にキャストする、というものでしょう。これでも（先の手法と同様）変換未チェック警告が出ますが、明言されていない想定（例えば`backingArray`は`ArrayList`の実装を別扱いすべきではない、というような事実）が、より明確になります。

未踏の道

一番良いのは、実装が（ランタイム時に）Tの値を知ることができるように、クラス・リテラル（Foo.class）をコンストラクターに渡すことでしょう。この手法をとらなかった理由は、新たなgeneric化コレクションクラスが以前のバージョンのCollectionsフレームワークと互換性がなくなる、という後方互換性のためです。

この手法を使ったとしたら、ArrayListは次のようになります。

```
public class ArrayList<V> implements List<V> {
  private V[] backingArray;
  private Class<V> elementType;
  public ArrayList(Class<V> elementType) {
    this.elementType = elementType;
    backingArray = (V[]) Array.newInstance(elementType, DEFAULT_LENGTH);
  }
}
```

でも、ちょっと待ってください。これでもやはり、Array.newInstance()を呼ぶ時には醜悪な未チェック・キャストがあるではありませんか。なぜか？ これもまた後方互換性のためです。Array.newInstance()のシグニチャーは下記です。

```
public static Object newInstance(Class<?> componentType, int length)
```

下記のようにタイプ・セーフではありません。

```
public static<T> T[] newInstance(Class<T> componentType, int length)
```

なぜArrayはこのような方法でgeneric化されたのでしょう。これもまた、ご不満かも知れませんが、後方互換性を保つためなのです。int[]のようなプリミティブ・タイプの配列を作るためには、適当なラッパー・クラスからTYPEフィールドを持つArray.newInstance()を呼びます（intの場合では、Integer.TYPEをクラス・リテラルとして渡します）。Class<?>ではなくClass<T>パラメーターでArray.newInstance()をgeneric化した方が、参照タイプとしては、よりタイプ・セーフなのですが、そうするとArray.newInstance()を使ってプリミティブ配列のインスタンスを作ることができなくなります。恐らく将来は、参照タイプとして別バージョンのnewInstance()が提供され、両方の使い方ができるようになるでしょう

皆さんにはここで、あるパターンが見えてきたのではないでしょうか。genericに関連した問題や妥協点の多くは、generic自体の問題ではなく、既存コードとの後方互換性を保つ必要から生じる副作用なのです。

既存のクラスをgeneric化する

既存のライブラリー・クラスがgenericとスムーズに動作するように変換するのは、それほど簡単なことではありません。毎度のことですが、後方互換性を保とうとすると、タダではすまないのです。後方互換性のためにクラス・ライブラリーのgeneric化が制限される例として、既に2つを挙げました。

もう一つの例として、後方互換性の問題がなければ異なった方法でgeneric化されたであろうメソッドが、Collections.toArray(Object[])です。toArray()に渡される配列には、2つの役割

があります。提供される配列に収まるほどコレクションが小さければ、その内容は単純に、その配列の中に置かれます。そうでない場合には、（反映を使って）同じタイプの新しい配列が作られ、結果を受け取ります。もしCollectionsフレームワークが全く初めから書き直されたとすると、`Collections.toArray()` への引き数は恐らく配列ではなく、クラス・リテラルでしょう。

```
interface Collection<E> { public T[] toArray(Class<T super E> elementClass);
}
```

Collectionsフレームワークは良いクラス設計として広くエミュレートされているので、後方互換性に制限を受ける領域について指摘しておくのは無駄ではないでしょう。そうすれば、そうした領域が盲目的にエミュレートされるのを防げると思います。

generic化Collections APIの要素の一つで、最初に混乱しやすいのは、`containsAll()`と`removeAll()`、それに`retainAll()`のシグニチャーです。皆さんは`remove()`や`removeAll()` のシグニチャーを次のようだと思うか知れません。

```
interface Collection<E> { public boolean remove(E e);  // not really
  public void removeAll(Collection<? extends E> c);  // not really
}
```

ところが実際は、下記なのです。

```
interface Collection<E> { public boolean remove(Object o);
  public void removeAll(Collection<?> c);
}
```

なぜこうなのでしょう。ここでも、答えは後方互換性の中にあるのです。`x.remove(o)` のインターフェース契約の意味は、「もし`o`が`x`の中に含まれていたら、それを削除せよ。そうでなければ、何もするな」なのです。`x`がgenericなコレクションである場合には、`o`は`x`のタイプ・パラメーターとタイプ互換である必要はありません。引き数がタイプ互換（`Collection<? extends E>`）である場合にのみ呼べるように`removeAll()` をgeneric化したとすると、generic化の前には正しいものであった一部のコード・シーケンスは、不正なものになってしまいます。下記はその一例です。

```
// a collection of Integers
Collection c = new HashSet();
// a collection of Objects
Collection r = new HashSet();
c.removeAll(r);
```

上記のコード断片を明白にgeneric化（`c`を`Collection<Integer>`に、そして`r`を`Collection<Object>`に）したとすると、`removeAll()` のシグニチャーが引き数としてno-op（ノー・オペレーション）ではなく`Collection<? extends E>`を要求する場合には、上記のコードはコンパイルできません。クラス・ライブラリーをgeneric化する大きな目的の一つは、既存コードの意味体系を壊したり変更したりしないということです。ですから`remove()`や`removeAll()`、`retainAll()`、それに`containsAll()` などは、全く初めからgeneric用に再設計されたものよりも、弱いタイプ制約で定義されている必要があったのです。

Generic以前に設計された既存クラスには、「明白な」generic化手法に抵抗する意味体系を持っているものもあるかも知れません。そうした場合には、恐らくここで説明したような妥協を行う必要があるでしょう。しかし全く初めからgenericなクラスを再設計する場合には、エミュレートが不適切に行われないように、Javaライブラリー・クラスの、どのイディオムが後方互換性の結果なのかを理解することが重要です。

Erasureの意味合い

Genericはほとんど完全にJavaコンパイラーの中で実装され、ランタイムの中では実装されないので、genericタイプに関するタイプ情報はほとんど全て、バイトコードが生成される時までに「消去」されてしまっています。言い方を変えれば、コンパイラーが生成するコードは、手動で書いたコード、つまりプログラムのタイプ・セーフをチェックした後、generics やキャスト、等々を使わず手で書いたものとほとんど同じなのです。C++の場合とは異なり、`List<Integer>`や`List<String>`は同じクラスです（ただし、両者は別々のタイプであり、両者とも`List<?>`のサブタイプです。JDK 5.0では、この区別は以前のバージョンよりも重要です）。

Erasureが持つ意味合いの一つは、クラスは`Comparable<String>`と`Comparable<Number>`の両方を実装することはできない、ということです。これは、実際には両者とも、同じ`compareTo()`メソッドを規定する同じインターフェースであるためです。`String`と`Number`の両方にとって同等な`DecimalString`クラスを宣言した方が妥当、と思えるかも知れませんが、Javaコンパイラーにとっては、同じメソッドを2回宣言しているように見えることになります。

```
public class DecimalString implements Comparable<Number>,
Comparable<String> { ... } // nope
```

Erasureの意味合いとして、もう一つは、genericタイプのパラメーターを持つキャストまたは`instanceof`を使っても意味がない、ということです。次のコードは、コードのタイプ・セーフを全く改善しません。

```
public <T> T naiveCast(T t, Object o) { return (T) o; }
```

コンパイラーは、キャストが安全なものかどうかは分からないため、単に変換未チェック警告を出力します。実際、`naiveCast()`メソッドは何のキャストも行いません。`T`は単にそのerasure（`Object`）で置き換えられ、渡されたオブジェクトは`Object`にキャストされます。これは意図された動作ではありません。

Erasureはまた、先に説明したコンストラクションの問題、つまりコンパイラーはどのコンストラクターを呼ぶべきか分からないのでgenericタイプのオブジェクトを作るべきではない、という問題にも責任を持ちます。genericクラスが、genericタイプのパラメーターで規定されるタイプを持つオブジェクトを作る必要がある時には、インスタンスが反映で作られるように、そのクラスのコンストラクターはクラス・リテラル（`Foo.class`）を取って保存する必要があります。

まとめ

Javaでのタイプ・セーフにとって、genericは大きな前進です。ただし、generic機能の設計やクラス・ライブラリーのgeneric化には妥協が必要です。仮想マシンの命令セットを拡張してgenericをサポートすることはできません。そんなことをするとJVMのベンダーがJVMを更新す

ることが非常に困難になってしまう可能性があります。そこで、完全にコンパイラーの中で実装できる、erasureによる手法が採用されたのです。同じように、Javaクラス・ライブラリーをgeneric化する際に後方互換性を維持しようとしたために、クラス・ライブラリーをgeneric化する方法に対して多くの制約が生じました。おかげで、間違いやすく、また腹立たしい構造体（`Array.newInstance()` など）を使う必要が出てきています。これらはgeneric自体の問題ではなく、Java言語の進化と互換性の現実なのです。genericを学び、使うことが間違いやすく、また不満を抱かせるのも、やむを得ないのかも知れません。

著者について

Brian Goetz

Brian Goetz は18 年間以上に渡って、専門的ソフトウェア開発者として働いています。彼はカリフォルニア州ロスアルトスにあるソフトウェア開発コンサルティング会社、Quiotixの主席コンサルタントであり、またいくつかのJCP Expert Groupの一員でもあります。2005年の末にはAddison-Wesleyから、Brianによる著、Java Concurrency In Practiceが出版される予定です。Brian著による有力業界紙に掲載済みおよび掲載予定の記事のリストを参照してください。