MICHAEL KÖLLING

ジェネリクス：難解な部分を克服する

ジェネリクスにおけるワイルドカード、サブタイピング、型消去

Javaのジェネリック型についての話を続けます。前回の記事では、ジェネリック型について、なぜ便利なのか、何ができるのか、どうやって使うのかなどの説明を始めました。導入部分はとても単純でしたが、説明の最後で汎用コレクションとサブタイピングの問題について触れました。

それを一言で言えば、次のような汎用的なprintListメソッドを書きたいということでした。

```
private void printList(List<Person> list)
```

これを使って、Person型のサブタイプのリスト、たとえばList<Student>やList<Faculty>を出力したいということでした。すなわち、StudentがPersonのサブタイプである場合に、先ほどのメソッドを次のように呼び出したいのです。

```
List<Student> students = getStudentList();
printList(students);
```

Javaでは、これは動作しません。なぜかと言えば、StudentがPersonのサブタイプであるにもかかわらず、List<Student>はList<Person>のサブタイプとは見なされないからです。

何が問題か

では、なぜList<Student>はList<Person>のサブタイプではないのでしょうか。リストの出力だけを考えるなら、何の問題もないように思われます。たとえば、printListメソッドは、すべてのリストの要素のprintメソッド（このメソッドはPersonで定義され、サブタイプで適切に再定義されている可能性があります）を呼び出すことができるでしょう。すべてうまくいくように思えます。

しかし、printListメソッドでもリストを修正する可能性を考えると、問題点が明らかになります。たとえば、このメソッドに次の行が含まれているとします。

```
list.add(new Faculty());
```

list変数の静的な型（メソッドの仮パラメータ）はList<Person>であり、FacultyはPersonのサブタイプであるため、このオブジェクトを追加しても型としては問題はありません。しかし、printListメソッドに渡される実際のリストがStudentのリストである場合、Studentのリストに Facultyオブジェクトを追加することになってしまいます。これは明らかにエラーであり、起こってはいけないことです。

この問題の唯一の解決策は、List<Student>はList<Person>のサブタイプではないものとし、StudentのリストがprintListメソッドに渡されないようにすることです。これで型の安全性は保証されますが、最初の問題は振り出しに戻ってしまいます。それでは、どうすれば汎用的なprintListメソッドを書くことができるでしょうか。

解決策はワイルドカード

この問題を解決するのがワイルドカードです。printListメソッドは、次のように書くことができます。

```
private void printList(List<?> list)
```

リストの要素の型ではなく疑問符を記述している点に注意してください。疑問符はワイルドカードのシンボルで、unknownと呼ばれる型を示しています。これによって、パラメータはunknown型のリストになります。

この構成には明らかなメリットがあり、望みどおりのことを行えるよう

PHOTOGRAPH BY JOHN BLYTHE

になります。printListメソッドは、List<Student>とList<Faculty>の両方をパラメータとして呼び出すことができます。

```
List<Student> students = getStudentList();
List<Faculty> professors = getFacultyList();
printList(students);
printList(professors);
```

すべてのリスト型はこのunknown型のリストのサブタイプと見なされます。そのため、上記のコードは動作します。その代わり、要素の型がわからない場合はリストに追加できないため、先ほどリストに追加しようとした際に生じた型の問題は回避されます。

unknown型からわかること

ワイルドカードはこのような問題を解決するために有効な手段ですが、すべての問題が解決するわけではありません。この点は、リストの要素に対して何ができるかを考えてみるとわかるでしょう。Personスーパークラスにprintaddressdetailsメソッドがあり、printListメソッドの中でそれを使う場合を考えてみます。次の例をご覧ください。

```
private void printList(List<?> list) {
  for (Person p: list) {
    ...
    p.printAddressDetails();
  }
}
```

このコードは動作しません。unknown型を使うメリットは任意の型のリストを渡せることですが、その反面、その型について何もわからないということになります。実際のところ、わかるのは、Objectのサブタイプであることだけです（すべての型はObjectのサブタイプであるからです）。そのため、この要素をPerson型として扱うことはできません。

要素の型について詳しいことがわからなくても、特に問題ない場合もあるでしょう。size()やclear()など、要素の型によらないリスト操作はすべて行うことができます。また、toStringメソッド（System.out.printlnから暗黙的に呼び出される可能性もあります）など、Object型に対する操作も行うことができます。

しかし、型固有のメソッドの呼出しには別の方法が必要になります。ワイルドカードを使用する際、パラメータが厳密にPersonのListであると宣言する代わりに、任意のListであると宣言しました。そうではなく、Personの任意のサブタイプであるListと宣言したいのです。これは、境界付きワイルドカードを使うことで実現できます。

境界付きワイルドカード

総称型パラメータに境界を付けて、実際に使用可能である型を制限できます。次のように書き換えたprintListメソッドをご覧ください。

```
private void printList(List<? extends Person> list)
```

この定義によって、前述の意図どおりにPersonのリストまたはPersonのサブタイプのリストがパラメータとして許可されます（この2つ以外は許可されません）。ワイルドカードを使用しているため、リストへの追加は不可能ですが、すべての要素がPerson型（またはそのサブタイプ）であることはわかります。これによって、各要素をPersonオブジェクトとして扱えるようになり、適切なメソッドを呼び出せるようになります。これでようやく問題が解決しました。

その他の境界型

境界を使うことで有用性が増すのはワイルドカードだけではありません。境界型は、ジェネリック型の宣言や、ワイルドカードのないメソッドでも使うことができます。たとえば、Personとそのサブタイプのみをパラメータとして受け取るジェネリック型、PersonListを定義できます。

```
class PersonList<T extends Person>
```

これは、前号の**Java Magazine**で紹介したArrayListの定義と似ていますが、ここでは、型をインスタンス化する際にPersonのサブタイプのみを使用できます。

```
PersonList<Student> students =
  new PersonList<Students>();
PersonList<Faculty> professors =
  new PersonList<Faculty>();
```

こうすると、PersonListクラスの実装でT型のオブジェクトに対してすべてのPerson型のメソッドを使うことができるようになります。Tを具象化したインスタンスはすべてPerson型のメソッドを持つことが保証されるからです。

ジェネリック・メソッド

ここで、ジェネリックのもう1つの機能であるジェネリック・メソッドについて説明します。先ほどの例では、ジェネリック・クラスを宣言するときにクラス・ヘッダーでジェネリック型のパラメータを使用しましたが、クラス全体をジェネリックにせずに、1つのジェネリック・メソッドを持たせることも可能です。その場合、その1つのメソッドでジェネリック型を扱えます。多くの場合、ジェネリック・メソッドは境界付きジェネリック型とともに使用します。

次の例について考えてみます。ここで書いているのは、リストの要素のうち、指定した制限値よりも小さいものをすべて出力するメソッドです。

```
public <T> void underLimit(List<T> myList, T limit) {
  for (T e : myList) {
    if (e < limit)
      System.out.println(e);
  }
}
```

ここで登場する新しい構文は、ヘッダーのpublicキーワードと戻り値の間にある型パラメータ<T>です。このメソッドは、ジェネリックでないクラスにあるという前提であるため、型パラメータはまだ何も宣言されていません。パラメータ・リストでジェネリック型を使うためには、まずその型を宣言する必要があります。型<T>をヘッダーに記述することには、その意味があります。

しかし、このコードは実行できません。なぜなら、左不等演算子は未指定の型Tには適用できないからです。代わりにcompareToメソッドを使うこともできますが、このメソッドが動作するのはTがComparableのサブタイプである場合のみです。compareToメソッドを使用するためには、先ほどのメソッドを次のように変更します。

```
public <T extends Comparable<T>> void underLimit(
  List<T> myList, T limit) {
  for (T e : myList) {
    if (e.compareTo(limit) < 0)
      System.out.println(e);
  }
}
```

この宣言によって、Comparableのサブタイプである型Tのみを受け入れるようになり、必要なメソッドを使えることが保証されます。

上限と下限

ここまでは、上限だけをワイルドカード・パラメータに指定することでスーパータイプ（上限）を確定させる境界型について説明してきました。たとえば、次のようなものでした。

```
List<? extends Person>
```

これによって、その型をインスタンス化するために使用できるのは、その名前が付いた型またはそのサブタイプのみとなります。つまり、使用する時点で、具象型はPersonを拡張（または実装）している必要があります。Personについての一般的な継承階層を図にするなら、Personとその下にある階層のクラスのみを使えることになります。

下限を宣言した場合、それとは反対方向に型を制限することもできます。

```
List<? super Person>
```

宣言にsuperキーワードを使用した場合、その型はPersonまたはPersonのスーパータイプでなければならないことを明示できます。これを継承階層で考えると、Personまたはその上にある階層の型を使えるということになります。下限を使う機会は上限よりも少ないですが、状況によっては役立つこともあります。

実装

ジェネリック型の使用方法だけでなく、JavaコンパイラやJVMでジェネリック型がどのように実装されているかを少し知っておくことも役に立ちま

す。実装についての話を聞いたことがあるなら、おそらくどこかの段階で型消去という用語も聞いたことがあるでしょう。型消去は実装の効率だけでなく、特定の場合のコードのセマンティックにも影響することがあるため、この用語の意味を知っておくことは重要です

型消去

型消去の中核をなすのは、型パラメータはコンパイル時にのみ存在するという事実です。実行時に、型パラメータは完全に消去されます。型パラメータは、コンパイル時に型チェックを行って型の安全性を保証するためだけに使われる構造です。そのため、型パラメータはJavaバイトコードには引き継がれません。

最初にジェネリクスを理解しようとするときに、ジェネリック・クラスはインスタンス化する際に拡張されるものと考えるとわかりやすいでしょう。たとえば、次の型について考えてみます。

```
class List<T> {
  public void add(T elem);
  ...
}
```

このクラスが具象型List<String>を使ってインスタンス化されると、ソース・テキストの中のすべてのTがStringに置き換えられ、addメソッドのパラメータ型はStringになると考えることができます。List<Integer>では、すべてのTがIntegerに置き換わることになります。他の具象型についても同様です。

この考え方はジェネリクスを理解し始める上では役立つものですが、究極的には誤ったモデルであると言えます。わかりやすいため、役立つ考え方であり、ジェネリクスがどのように動作するかをおおまかに表現する程度であればよいでしょう。しかし、実際にどのような仕組みで動いているかを知っておくことも大切です。仕組みを知っているかどうかで、重大な違いが出てくる場合もあるからです。

ジェネリック型は、ソース・コード、バイナリ・コード、ディスク上、メモリ内のいずれにおいても、実際のインスタンスに展開されることはありません。これは、実際に展開処理が行われるC++のテンプレートなどとは異なります。Javaでは、生成されたコードは境界のない型パラメータの型としてObjectを挿入します。境界のある型についても、境界付きの型として同じようにObjectを挿入します。そのため、プログラムが実行されるまでには、List<String>、List<Integer>、List<Personなどはすべて1つのクラスList<Object>で表されます。クラスが型に安全な方法で使われていることと、型の問題が起こらないことを、この時点までにコンパイラが確認しています。つまり、たくさんの型を使っていても、最終的には1つのクラスになります。

実行時に型パラメータ情報を破棄することには、メリットもデメリットもあります。メリットの1つは、すべてのジェネリック・クラスに対してクラス・ファイルは1つだけ存在すればよいため、時間や領域を節約できることです。何度も保存やコンパイルを行う必要がありません。これは明らかなメリットです。

一方で、型消去によって、開発環境などを作成するツール作成者は作業がしにくくなります。たとえば、型がジェネリック・クラスから派生している場合、実行時にデバッガがオブジェクトの正しい型を見つけるのは困難です。クラス・ファイルには、完全な型情報に関する情報は何も保管されていません。

プログラマーにとってさらに重要なのは、型消去がコードの動作に影響する可能性があるという事実です。以降のセクションでは、Javaシステムの動作を理解する上で型消去を理解することが必要になる例を紹介します。

型パラメータのある型にinstanceofは使えない

ラメータ化した型ではinstanceof演算子を使用できません。先ほどのセクションで定義したList<T>を使おうとしている次の例を考えてみます。

```
if (list instanceof List<Person>) {
  List<Person> pl = (List<Person>) list;
}
```

このコードはまったく問題ないように見えますが、型消去についての先ほどのセクションを見れば、なぜ動作しないかがわかるでしょう。ランタイム・システムでは型情報を保持していないため、型がList<Person>であるかどうかはわかりません（List<Object> であることはわかりますが、それ以上具体的なことはわかりません）。そのため、このチェックを行うことはできないという答えになります。illegal generic type for instanceofというエラーが表示されます。

getClassメソッドを使用する場合も同じ問題が発生します。

```
List<Student> sl = new ArrayList<Student>();
List<Faculty> fl = new ArrayList<Faculty>();
if (sl.getClass() == fl.getClass())
  ...
```

一見すると、if文の条件はfalseになるように思われるかもしれませんが、型消去が行われているため、実際にはtrueと評価されます。ランタイム・システムが関係している限り、いずれのオブジェクトのクラスも ArrayListになります。

ジェネリック・クラスと静的属性

ソース・コードで型消去がもっともわかりやすく現れる場面の1つは、ジェネリック・クラスで静的属性を使う場合です。静的メソッドと静的フィールドは、ジェネリック・クラスのすべてのインスタンスで共有されます。ここでも理由は同じで、ジェネリック・クラスは実際には1つしか存在しないからです。正しいコードを書くには、この点に注意する必要があります。この点の副作用は、ジェネリック・パラメータ型の静的フィールドを宣言できないことです。

```
class MyClass<T> {
  private static T value;  // error
  ...
}
```

このフィールドは、この型から派生するすべての型で共有されるため、特定のインスタンスの型パラメータを参照することはできません。

Javaトリビア：配列と型の安全性

Javaと型の安全性の詳細について興味をお持ちの方には、このちょっとしたJavaトリビアを気に入ってもらえるかもしれません。実は、Javaでの配列の実装には型システムの抜け道があります。これは、Javaが静的に型安全でない珍しい部分の1つです。

これは、本記事ですでに説明したものと同じ問題です。BがAのサブタイプである場合、List<B>はList<A>のサブタイプでしょうか。リストの場合、答えはノーでした。その理由と、List<B>をサブタイプと考えると何がうまくいかないのかについては、本記事ですでに説明しています。しかし、（非常に似た状況で）Javaは配列をサブタイプと見なします。これによって、型の問題が起こる可能性があります。次のコードについて考えてみます。

```
A[] aa;
B[] ba = new B[3];

aa = ba; // 許可される。B[]はA[]のサブタイプ
aa[0] = new B();
aa[1] = new A();  // java.lang.ArrayStoreException: A
```

この例の最後の行は、型エラーが起きることを示しています。ここでは、Aオブジェクトを Bの配列に挿入しようとしています。問題なのは、3行目の代入が許可されることです。この問題は、コンパイル時ではなく、実行時にのみ検出されます。そのため、Javaの静的な型安全性が破られます。ジェネリック・クラスを設計したとき、Javaチームはいっそう慎重を期して、同様の問題をコンパイル時に検出すると決めました。

まとめ

ジェネリック型は基本的にわかりやすい考え方であり、通常は使うのも非常に簡単です。しかし、より洗練されたコード、特にライブラリなどを書こうとする場合、ジェネリクスの高度な仕組みを理解しなければならないさまざまな状況に遭遇するでしょう。

ジェネリック型は基本的にわかりやすい考え方であり、通常は使うのも非常に簡単です。しかし、より洗練されたコード、特にライブラリなどを書こうとする場合、ジェネリクスの高度な仕組みを理解しなければならないさまざまな状況に遭遇するでしょう。

あらゆる考え方を駆使した場合、クラスとメソッドの定義は、経験を積んだプログラマーでも読むのが難しいような非常に複雑なものになることもあります。たとえば、標準ライブラリのCollectionsクラスにある maxメソッドや、Classクラスに定義されているメソッドを確認してみてください。あらゆる仕組みが組み合わせられているため、読み解くのに多少時間がかかるだろうということがわかるでしょう。しかし、それで投げ出してしまってはいけません。このような複雑な仕組みが使われているのはま

れであり、今回説明した考え方やいくらかの実習によって、大半のことは解決できるようになるはずです。さらに重要なことは、自分の力で正しく柔軟なコードを書けるようになるだろうということです。</article>

Michael Kölling：Java Champion、イギリスのケント大学教授。オブジェクト指向やコンピューティング教育について多数の論文を発表しているほか、Java教本2冊を出版。教育用プログラミング環境であるBlueJおよびGreenfootの開発リーダーも務める。ACMのDistinguished Educatorでもある。

learn more

The Java Tutorial on generic types

注目のJDK強化提案

JEP 282 jlink：Javaリンカー

Javaのリンカーと聞いても、大半の読者は不思議に思うでしょう。リンカーの機能はネイティブ言語に関連するビルド・ツールの範囲に含まれ、JavaではJVMのクラス・ローディング機構がこの機能を担っています。中でも、おもにリンカーの機能を実行しているのは、参照先のクラスやメソッドを含むJARを探して現在のJVMメモリ空間にロードするアルゴリズムです。【編集注：このプロセスについての詳細は、Oleg Šelajev氏による本誌の記事「JVMによるライブラリの検索、ロード、実行」のPDFを参照してください】

JEP 282で提案されているのは、従来型のリンカーではなく、ビルド・プロセス内でリンカーが実行される場所（コンパイラの実行が終了してから実行可能ファイルが作成されるまでの間）で動作する汎用的なツールです。このツールは、プラグイン・インタフェースを定義するものになる予定で、このインタフェースによってさまざまなツールをビルド・プロセスに挿入できるようになります。もっともわかりやすいのはオプティマイザでしょう。特に、パフォーマンスの改善やコード・サイズの削減といった、クラス・ベースで処理を行うコンパイラからは見えない部分の最適化を行う機会を与える、プログラム全体を対象としたオプティマイザです。JEPのドキュメントでは、デバッグ情報の削除や、高速にロードするためのリソースの並び替えを行うプラグインのほか、生成ファイルの圧縮を行うプラグインも提案されています。

理論的には、サード・パーティ製のコードも含め、生成されたコードに対するさまざまな改善が可能です。たとえば、計測データの挿入、デバッグ・データの補足、バイトコードの別フォーマットへの変換、クラス内最適化などが考えられます。こういったことはすべて、jlinkテクノロジーで提案されているプラグインで実現可能です。