MICHAEL KÖLLING

ジェネリクスについて

ジェネリクスによる型保証と可読性の向上

New to Javaシリーズの本記事では、ジェネリクスについて取り上げます。

Javaのプログラミング経験が少しあれば、ジェネリクスを見たことも、おそらく使ったこともあるでしょう（コレクションを利用する際にほぼ必ず登場し、コレクションを利用しなければ、凝ったことは何もできないからです）。C++からJavaに移った人であれば、パラメータ化型あるいはテンプレートという名称で、ジェネリクスと同じ概念をすでにご存じのはずです（C++におけるテンプレートはすべての面でジェネリクスと同等というわけではありませんが、この2つは似通っています）。

Javaプログラミングを始めたばかりの人は、仕組みや実行可能な操作をよく理解せずにジェネリクスを利用しています。このギャップこそ本記事で取り上げる内容です。

Javaにおけるジェネリクスの概念は、原則としては単純でわかりやすいのですが、細部が少々複雑です。特殊なケースについて説明すべき点が多く、JavaコンパイラやJVMでのジェネリクスの実装方法を調べるのも興味深いものです。それらの知識を得ることで、さらに驚くようなジェネリクスの特性を理解し、活用できるようになります。

以降は2つのパートに分けて説明します。まず本号では、ジェネリック型の原理と基本的な考え方について説明します。ジェネリクスの定義と使い方を確認して、基礎となる全体的な理解を深めます。次号のJava Magazineでは、さらに細かな内容や高度な使い方、実装について確認します。2つの記事を読めば、ジェネリクスによって優れたコードを記述するための方法を十分に理解できるようになります。

ジェネリクスの歴史

2004年にJava 5がリリースされる前は、Javaにはジェネリック型がありませんでした。代わりに、異種コレクションと呼ばれる要素が存在しました。当時、以下のような異種リストを記述していました。

```
List students = new ArrayList();
```

（この歴史を学ぶことは大事なことです。現在でもJavaでこのようなコードを記述できるからです（ただし推奨されません）。また、維持管理対象のレガシー・コードにこのようなコレクションが含まれていることもあります。）

この例では、Studentオブジェクトのリストを作成しようとしています。サブタイピングという手法を使っており、変数の宣言型がListインタフェース（ArrayListのスーパータイプ）で、このArrayListが実際に作成されるオブジェクト型になっています。サブタイピングは、柔軟性が向上するという意味では良いアプローチです。以下のように、このリストに学生オブジェクトを追加できます。

```
students.add(new Student("Fred"));
```

次に、このリストから学生オブジェクトを取得するときに、ごく自然に考えれば以下のようなコードになります。

```
Student s = students.get(0);
```

しかし、この記述は通用しません。Javaでは、List型が任意の型の要素を保持できるようにするために、addメソッドがObject型のパラメータを受け取り、getメソッドも同様にObject型を返すように定義されていました。したがって、先ほどのコード例は、Object型の式をStudent型の変数に代入しようとしているため、エラーになります。

かつてのJavaでは、キャストによってこの問題を解決していました。

```
Student s = (Student) students.get(0);
```

写真：
JOHN BLYTHE

このキャストをいちいち記述しなければならないのが、小さな悩みの種でした。プログラマーは通常、特定のリストに格納されるオブジェクトの型が何かをわかっています。それなのになぜ、コンパイラも同じようにその情報を記録できないのでしょうか。

しかし、小さな悩みの種以上の問題もあります。リストの要素型がObjectになると宣言されているため、実際は同じリストにあらゆる型のオブジェクトを追加できます。

```
students.add(new Integer(42));
students.add("a string");
```

このように同じリストが異なる型のオブジェクトを保持できるため、このリストは異種と呼ばれています。リストに複数の要素型を格納できるのです。

互いに関係のない異なる要素型を含むリストを作成することに有用性はほぼありませんが、誤ってそうすることは簡単です。異種リストの問題は、その誤りを実行時まで検出できないことにあります。この学生リストに間違った要素型をうっかり挿入してしまう。そのような事態を防ぐものがありません。さらに良くないことに、リストから間違った型の要素を取得してStudentへキャストしたとしても、コードのコンパイルは通ります。このエラーは実行時、キャストに失敗したときに初めて表面化します。その際には、実行時の型エラーが生成されます。

この実行時エラーは、遅れて（たとえばアプリケーションを顧客に納品した後の実行時に）発生することも問題ですが、検出されたエラーの元の場所が、実際にミスを犯した場所からかけ離れている場合があることも問題です。つまり、要素を取得した時点で問題の通知を受け、実際のエラーが要素の挿入時に発生しているとして、その2つがプログラムのまったく別の場所にある場合があります。

Java言語にジェネリクスが導入された際に少し変更された点について理解しておきます。**それは、クラスと型の関係です。**

Javaと型保証

Javaはこれまで常に型保証言語となるように作られてきました。型のエラーをコンパイル時に検出して、実行時の型エラーが発生しないようにすべきだという考え方です。この目的が完全に達成されたことはありませんが、それでもJava言語の設計者が懸命に目指してきた目標です。しかし、キャストはこの目標を破ります。キャストを使うたびに、型保証に穴が開きます。型チェッカーに対して、見て見ぬふりをし、ただプログラマーを信じるように指示することになります。しかし、プログラマーが正しくコーディングするという保証はどこにもありません。

多くの場合、キャストを使っているコードは書き直すことができます。もっと優れたオブジェクト指向技術を使ってキャストを回避し、型保証を維持することができます。しかし、コレクションには通常とは異なる問題があり、キャストなしにコレクションを使う方法はなく、Javaの哲学に反していました。このように、コレクションのような重要なプログラミング要素を扱う際に型保証を維持できないというのが、本当の悩みの種でした。そのような経緯で、Java言語チームは2004年に、Java言語にジェネリクスを追加することでこの問題を解決しました。

型の損失（Type Loss）

異種コレクションにおける上記の問題は、型の損失という用語で表されます。ある型のコレクションを作成するために、すべてのコレクションの要素型がObjectとして定義されていました。たとえば、addメソッドは以下のように定義できます。

```
public void add(Object element)
```

この定義は、要素（たとえばStudent型のオブジェクト）をコレクション内に配置する際には問題ないのですが、後でその要素を取得する際に問題になります。その要素がStudent型であるということをランタイム・システムは把握しますが、コンパイラは記録しません。後でgetメソッドを使って要素を取得するときに、コンパイラがわかることと言えば、要素型がObjectであることだけです。コンパイラは実際の要素型に関する情報を失います。そのような理由から型の損失という用語が使われます。

ジェネリクスの概要

型の損失の問題を解決するためには、コンパイラに対して要素型に関する十分な情報を提供する必要がありました。そのために、クラス定義またはインタフェース定義に対して型パラメータが追加されました。ArrayListの定義について考えてみます（例のArrayListは完全でなく、簡略化しています）。ジェネリクスが導入される前は、以下のように定義されていました。

```
class ArrayList {
   public void add(Object element);
   public Object get(int index);
}
```

ここでの要素型はObjectです。次に、ジェネリック・パラメータを使った場合、以下のような定義になります。

```
class ArrayList<E> {
   public void add(E element);
   public E get(int index);
}
```

山括弧で囲まれたEは型パラメータです。この場所に、リストの要素型を指定できます。Student型の要素に対するArrayListオブジェクトを以下のように作成することはもうありません。

```
new ArrayList()
```

以下のような記述に変わります。

```
new ArrayList<Student>()
```

メソッドのパラメータと同様に、定義に仮パラメータ仕様（E）を記述し、実際の使用時には実パラメータ（Student）を記述します。ただし、メソッドのパラメータとは異なり、実パラメータは値ではなく型になります。

ArrayList<Student>（通常の読み方は「StudentのArrayList」）を作成することによって、仕様で型パラメータEが使われている他の場所も、実際の型パラメータであるStudentに置き換えられます。したがって、addメソッドのパラメータ型やgetメソッドの戻り型もStudentになります。この仕組みは非常に便利です。要素として追加できるのがStudentオブジェクトだけになり、後で取得する際にはStudentオブジェクトを取得することになるため、キャストが不要になります。

ジェネリクスの導入時に、ダイヤモンド表記と呼ばれる便利な簡略表記が加わりました。この表記によって、可読性を向上させつつも、不必要に冗長な表現を避けることができます。

型の抽象化

Java言語にジェネリクスが導入された際に少し変更された点について理解しておきます。それは、クラスと型の関係です。ジェネリクスの登場より前は、クラスごとに1つの型を定義しました。たとえば、Hexagonクラスを定義した場合、変数定義やパラメータ定義で使うためのHexagonという型が自動的に作成されます。非常に単純な1対1関係です。

しかし、ジェネリック・クラスが導入されたことで、この関係が変わりました。ジェネリック・クラスは1つの型を定義するのではなく、型の集合を定義します。たとえば、ArrayList<E>クラスは、ArrayList<Student>型、ArrayList<Integer>型、ArrayList<String>型、ArrayList<ArrayList<String>>型などの任意の型を定義します。その任意の型は、型パラメータEを具象型に置き換えることによって指定できます。

言い換えれば、ジェネリクスによって型の抽象化ができるようになっています。非常に強力な新機能です。

利点

ジェネリック・クラスを扱う利点の1つが正確性の向上であることは明白です。誤った型のオブジェクトをリストに挿入できなくなります。以前は、誤った要素の挿入操作をテスト中にしか検出できませんでした（そしてテストを完璧にできることはありません）。しかし、そのような操作がコンパイル時に検出され、型の正確性が保証されるようになっています。さらに、エラーがある場合には、実際のエラーの場所とはかけ離れた、要素

の取得時点ではなく、誤った挿入操作の時点でレポートされるようになります。

他にも利点があります。可読性です。コレクションの要素型を明示することで、プログラムの読み手にとっても有用な情報を提供できます。コレクションに対して意図している要素型を明示することで、維持管理担当のプログラマーの作業が楽になります。

ダイヤモンド表記

ジェネリクスの導入時に、ダイヤモンド表記と呼ばれる便利な簡略表記が加わりました。この表記によって、可読性を向上させつつも、不必要に冗長な表現を避けることができます。

例として、変数を宣言して、新規作成したオブジェクトによってその変数を初期化するという、よくある表現について見てみます。

ジェネリクスを使うことで、コードの安全性と可読性を向上させることができます。

```
ArrayList<String> myList =
  new ArrayList<String>();
```

ジェネリック型は、とりわけジェネリック・パラメータが複数ある場合に、以下のように長い表現になることがあります。

```
HashSet<Integer, String> mySet =
  new HashSet<Integer, String>();
```

型パラメータ自体がジェネリックであれば、さらに状況は悪化します。

```
HashSet<Integer, ArrayList<String>> mySet =
  new HashSet<Integer, ArrayList<String>>();
```

それぞれの例で、長々とした同じジェネリック型が、左側の変数宣言部分と右側のオブジェクト作成部分に2回登場します。このような状況で、2つ目の型の記述を省略して、以下のように記述できます。

```
HashSet<Integer, String> mySet = new HashSet<>();
```

この例では、右側のジェネリック・パラメータが省略されています（山括弧だけが残りダイヤモンドの形になっているため、ダイヤモンド表記と呼ばれます）。ダイヤモンド表記はこのような状況で使うことができ、右側のジェネリック・パラメータが左側のジェネリック・パラメータと同じであることを示します。ただ入力の手間を省き、コードの可読性を向上させるためのものです。その場合、型を完全な形式で記述した場合とまったく同じセマンティックになります。

これまでのまとめ

これまでは理解しやすい内容でした。ジェネリクスを使うことで、コードの安全性と可読性を向上させることができます。単純なジェネリック・クラスの記述は非常にわかりやすく、ジェネリック型のオブジェクトの作成も難しくありません。以上で、ListインタフェースのJavadocページやCollectionインタフェースのJavadocページなど、単純なジェネリック型のドキュメントを読めるようにもなりました。

しかし、これで話が終わるわけではありません。これまで、ジェネリクスに伴う問題や、その問題を解決するメカニズムが少し難しいことについては取り上げませんでした。それこそ本当に興味深いところです。まずは、問題について確認します。

ジェネリクスとサブタイピング

小さな継承階層について考えます。大学の「人」をモデリングするために、Student（学生）およびFaculty（教員）というクラス、そして共通するPerson（人）スーパークラスを用意します（StudentとFacultyは共にPersonのサブクラスです）。ここまでは良いでしょう。

さらに、それぞれのクラスのリストについて型を作成すると、List<Student>、List<Faculty>、List<Person>となります。

学生リストと教員リストは、学生オブジェクトと教員オブジェクトを保持して処理するプログラム内に配置されます。Personリスト型は、たとえば以下のように、教員と学生の両方を扱うメソッドの仮パラメータとして使うと便利です。

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

これで、学生リストと教員リストの両方を実パラメータとして指定してprintListを呼び出せるようにしたいという考え方です。この方法は、これらのリストの型がList<Person>のサブタイプであれば機能します。しかし、はたしてそうなるでしょうか。

つまり、StudentがPersonのサブタイプである場合、List<Student>はList<Person>のサブタイプになるでしょうか。

直感的には「そうだ」と言えるかもしれませんが、残念ながら実際は異なります。

printListメソッドが内容を出力するだけでなく、パラメータとして渡されたリストの変更も行うと想像してみれば、この問題について理解できます。このメソッドがFaculty型のオブジェクトをリストに挿入すると考えてみます（リストがパラメータ内でList<Person>として宣言されており、FacultyがPersonのサブタイプであるため、完全に正当な操作です）。しかし、以前にこのメソッドに渡された実際のリストがList<Student>であったかもしれません。その場合、この学生リストに突如Facultyオブジェクトを挿入すれば、問題となることは明白です。

この問題の唯一の回避策は、サブタイプのリストとスーパータイプのリストが、それら自体でサブタイプとスーパータイプの関係を成すと見なさないことです。つまり、List<Student>はList<Person>のサブタイプではありません。

まとめ

一般化されたprintListメソッドを定義しようとした上記の例のように、ジェネリック型を使ったサブタイピングが必要になる場合も多くあります。本記事で説明した要素を使う場合は成功しないことを確認しましたが、「不可能だ」と言うだけでは不十分です。そのようなコードを記述できるようにする必要があります。

その解決策として、ジェネリクスには境界型およびワイルドカードという追加要素があります。これらの概念は強力ですが、難しい特殊なケースもあります。これらの概念については、次号のJava Magazineでパート2として説明します。

それまでは、Javaライブラリ内に存在するジェネリック・クラス、特にコレクションについて学習し、本記事で確認した表記に慣れてください。次回、ジェネリクスをさらに掘り下げていきます。**</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

[オラクルによるJavaのジェネリクスに関するチュートリアル]

[Javaのジェネリクスに関するWikipediaの記事]