MICHAEL KÖLLING

進化するJavaインタフェース

Javaの多重継承を理解する

ときおり掲載されているNew to Javaシリーズでは、言語構造の概念的な背景について理解を深められるようなトピックを選択しています。多くの場合、プログラミングの初心者は、概念についての実践的な知識を持っています。つまり、知識をさまざまな状況で使うことはできますが、その根底にある原理を深く理解してはいません。そういった原理を深く理解することは、よいコードを書き、よい構造を作り、与えられた構造をいつ使うのかを適切に判断することにつながります。Javaのインタフェースは、まさにそのようなトピックの1つです。

本記事は、継承の基本を理解していることを前提としています。Javaのインタフェースは、extendsキーワードおよびimplementsキーワードと同じように、継承と密接に関わっています。本記事では、なぜJavaには2つの異なる継承メカニズム（先ほどの2つのキーワードで表されます）があるのか、抽象クラスはどのような目的に適合するのか、インタフェースを使用できるさまざまなタスクにはどのようなものがあるのかについて説明したいと思います。

よくあることですが、このような機能の話は、とてもシンプルでエレガントなアイデアから始まっており、そのアイデアが初期のJavaのバージョンでの概念の定義につながっています。そして、Javaが複雑な現実世界の問題に対処できるよう進化するにつれて、その話も複雑さを増していくことになります。この課題がJava 8でのデフォルト・メソッドの導入につながるわけですが、この導入によって少しばかり水が濁るように感じるかもしれません。

継承についての簡単な背景

継承は、基本的にはわかりやすいものであり、別のクラスを拡張するものとしてクラスを指定できるということです。そのような場合、現在のクラスはサブクラスと呼ばれ、サブクラスによって拡張されるクラスはスーパークラスと呼ばれます。サブクラスのオブジェクトは、スーパークラスとサブクラスの両方のプロパティを持ち、サブクラスまたはスーパークラスのいずれかで定義されているすべてのフィールドとすべてのメソッドを持ちます。ここまでは、何の問題もないでしょう。

写真：JOHN BLYTHE

しかし、継承はプログラミングにおけるスイス・アーミー・ナイフのようなもので、とても多様な目的を実現するために使うことができます。継承は、以前に書いたコードを再利用する、サブタイピングや動的ディスパッチを行う、実装と仕様を分ける、システムの別々のパーツ間で取り決めを行うなど、さまざまなタスクに応用できます。これらはすべて重要なことですが、まったく異なる考え方です。それぞれの違いを理解することは、継承とインタフェースについての感覚を養うために欠かせません。

型継承とコード継承

継承が提供する2つの主な機能は、コードの継承と型の継承です。この2つの考え方は、概念的に区別しておくと有用です。その大きな理由は、標準のJavaの継承は、この2つの考え方を一緒にしたものだからです。Javaでは、クラスを定義するごとに型を定義することになります。たとえば、クラスを作ればその型の変数を作ることができるようになります。

サブクラスを（extendsキーワードを使って）作ると、そのサブクラスはスーパークラスのコードと型の両方を継承します。継承されたメソッドを呼び出すことができ（これを「コード」と呼びます）、スーパークラスのオブジェクトを入れる場所には、サブクラスのオブジェクトを入れることができます（つまり、サブクラスによってサブタイプが作成されます）。

例を見てみます。StudentがPersonのサブクラスだとした場合、Studentクラスのオブジェクトは Student型になりますが、さらにPerson型でもあります。学生も人なのです。その際に、コードと型の両方が継承されます。

Javaで型継承とコード継承をリンクさせるという決断は、言語設計に関わる選択です。この選択がなされたのは、便利なことが多いからです。しかし、これが言語設計の唯一の方法であるわけではありません。他のプログラミング言語には、型を継承することなくコードを継承できるもの（C++のprivate継承など）や、コードを継承せずに型を継承できるもの（これはJavaでもサポートされており、後ほど説明します）もあります。

多重継承

次に登場する考え方は、多重継承です。多重継承とは、1つのクラスが複数のスーパークラスを持ってもよいという考え方です。例を挙げてみます。筆者の大学では、博士課程（PhD）の学生はインストラクターとして働いてもいます。そういう意味では、教員（Faculty）のような存在です（授業のインストラクターを務め、部屋番号や給与番号を持つなどしています）。しかし、彼らは学生でもあります。科目を履修し、学生番号を持つなどしています。これは、多重継承を使ってモデリングできます（図1参照）。

PhDStudentはFacultyとStudentの両方のサブクラスです。こうすることによって、PhDの学生は学生と教員

の両方の属性を持つことができます。概念的には、多重継承は単純なものです。しかし、多重継承に伴う新しい問題が生じるため、実際問題として、多重継承を許す言語は複雑になります。たとえば、両方のスーパークラスが同じ名前のフィールドを持っていたらどうなるでしょうか。同じシグネチャで異なる実装のメソッドがあればどうなるでしょうか。そういった場合に備えて、曖昧さや名前のオーバーロードの問題に関する解決策を定める言語構造が必要になります。しかし、さらに悪い状況になる場合もあります。

ダイヤモンド継承

さらに複雑なシナリオは、ダイヤモンド継承（図2参照）と呼ばれるものです。ダイヤモンド継承は、あるクラス（PhDStudent）に2つのスーパークラス（FacultyとStudent）があり、その両方が同じスーパークラス（Person）を持つ場合です。この場合、継承図は菱形になります。

次の問題について考えてみます。トップレベルのスーパークラス（この場合はPerson）にあるフィールドが存在する場合、最下位のクラス（PhDStudent）にはこのフィールドは1つあるのでしょうか。それとも、2つあるのでしょうか。最下位のクラスは、このフィールドをそれぞれの継承の枝について1回継承しており、合計で2回継承しています。

この問いの答えは、「状況次第」です。問題のフィールドがID番号フィールドの場合、PhDの学生は学生IDと、教員/給与IDの2つを持っているはずです。これらは別々の番号の可能性があります。しかし、もしその人の姓がIDに

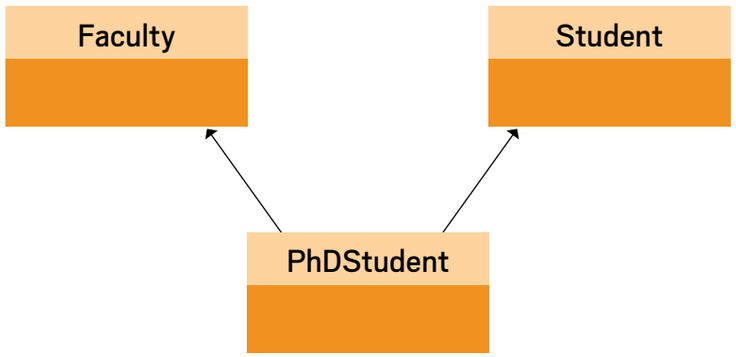

図1：多重継承の例

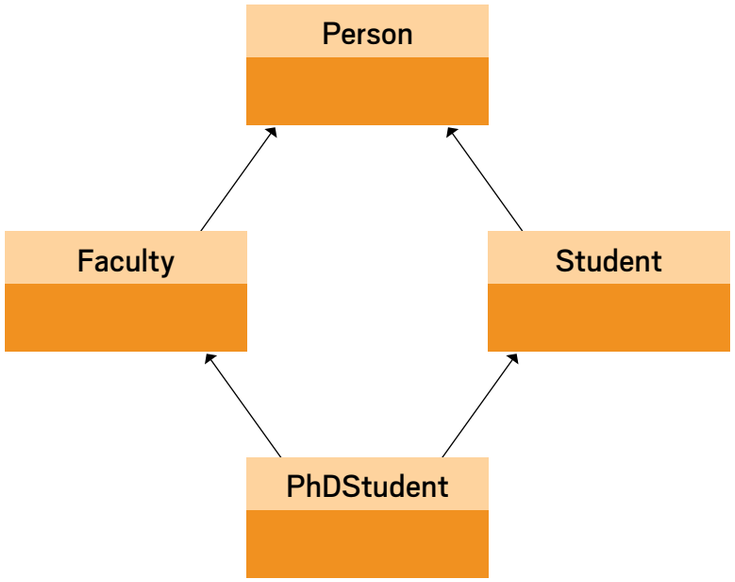

図2：ダイヤモンド継承の例

なる場合、1つだけの方が望ましいでしょう（両方のスーパークラスから継承した場合でも、PhDの学生の姓は1つだけです）。

つまり、この問題はかなりややこしい話になる可能性があります。完全な多重継承を許す言語は、こういった状況に対処するための規則と構造が必要になります。そして、その規則は複雑なものになります。

Javaの設計者は実用的な解決策にたどり着きました。コードの継承は1つだけに限り、型の多重継承は認めるというものです。

解決策は型継承

このような問題を注意深く考えれば、多重継承によって起こる問題はすべてコードの継承、つまりメソッドの実装とフィールドに関連していることに気づくと思います。コードの多重継承はややこしいものですが、型の多重継承では問題は起こりません。さらに考えてみれば、この事実に関して次のようなことも言えます。コードの多重継承は大して重要ではありません。代わりに委譲（別のオブジェクトへの参照を使用すること）を使えるからです。しかし、複数のサブタイプとなることはとても便利な場合が多く、他の方法でエレガントに実現することが容易ではありません。

そういうわけで、Javaの設計者は実用的な解決策にたどり着きました。コードの継承は1つだけに限り、型の多重継承は認めるというものです。

インタフェース

Javaでは、型とコードに対して別の規則を適用できるように、コードを指定せずに型を指定できるようにする必要があります。そのための仕組みがJavaのインタフェースです。

インタフェースは、実装を指定せずにJavaの型（型名とメソッドのシグネチャ）を指定します。フィールドやメソッドの本体は指定しませんが、定数を含めることはできます。修飾子（定数に対してはpublic static final、メソッドに対してはpublic）は省略でき、暗黙的に推測されます。

この仕組みによって、Javaでは2種類の継承ができるようになっています。クラスの継承（extendsを使用）では、型とコードの両方を継承します。インタフェースの継承（implementsを使用）では、型のみを継承します。それによって、多重継承に別々の規則を適用できるようになります。Javaは型（インタフェース）の多重継承を認めますが、クラス（コードを含みます）の継承は1つに限られます。

型の多重継承によるメリット

Javaでは、複数の型を継承でき、本質的に1つのオブジェクトがタイミングによって複数の型を持つように宣言できます。これには、とてもわかりやすいメリットがあります。たとえば、交通シミュレーションのプログラムを書いており、Carクラスのオブジェクトがあるとします。シミュレーションには車以外に、歩行者、トラック、信号などのアクティブなオブジェクトも存在します。そして、プログラムに登場するアクティブな要素をすべて集中的に保持するコレクション、すなわちListがあるものとします。

```
private List<Actor> actors;
```

この場合、Actorを、actメソッドを持つインタフェースとすることができます。

```
public interface Actor
{
   void act();
}
```

Carクラスはこのインタフェースを実装できます。

```
class Car implements Actor
{
   public void act()
   {
      ...
   }
}
```

Carは型（actメソッドのシグネチャを含みます）だけを継承しますが、コードは継承しません。そのため、オブジェクトを生成する前にこのクラス自体で型を実装するコード（actメソッドの実装）を提供する必要があります。

この時点では継承は1つだけです。そのため、これはクラスの継承でも実現できます。しかし、画面に描画されるすべてのオブジェクトのリストも必要になる場合を考えてみます（アクティブな要素には、画面に描画されないも

のもあり、描画されるオブジェクトもアクティブな要素とは限りません。そのため、描画されるすべてのオブジェクトのリストは、すべてのアクティブな要素のリストと同じではありません）。

```
private List<Drawable> drawables;
```

また、どこかの時点でシミュレーションの結果を永続ストレージに保存した方がよいでしょう。その際に保存されるオブジェクトも別のリストかもしれません。保存するためには、Serializable型である必要があります。

```
private List<Serializable> objectsToSave;
```

その場合、Carオブジェクトがこの3つのリストのすべてに含まれるものとすると（車はアクティブで、描画され、保存もされる必要があります）、Carクラスは3つのインタフェースをすべて実装するように定義できます。

```
class Car implements Actor, Drawable, Serializable ...
```

このような状況はよく起こります。複数のスーパータイプを認めることで、1つのオブジェクト（この場合は車）をさまざまな視点から観察し、異なる側面に注目して似たようなオブジェクト同士をグループ化することや、行うことができる動作のサブセットに基づいて扱うことができるようになります。

JavaのGUIイベント処理モデルも同じ考え方に基づいて構築されています。イベント処理は、イベント・リスナーによって行われます。多くの場合、これはメソッドを1つだけ実装するインタフェース（ActionListenerなど）として実現されます。そのため、イベント・リスナーを実装するオブジェクトは、必要に応じてそのリスナー型として見ることができます。

抽象クラス

抽象クラスとインタフェースとの関係について疑問を抱く方も多いため、抽象クラスについても簡単に説明しておきます。抽象クラスは、クラスとインタフェースの中間に位置するもので、型を定義して（クラスと同じように）コードを含めることができるものです。しかし、抽象クラスは、指定はされるものの、実装はされていない抽象メソッドを持つこともできます。ところどころに空白（コードが欠けており、サブクラスで埋める必要がある部分）がある、部分的に実装されたクラスと考えることもできます。

先ほどの例では、Actorインタフェースの代わりに抽象クラスを使うこともできます。actメソッド自体を抽象メソッドにすることもできます（各要素で動作方法が異なり、適当なデフォルトが存在しないためです）が、登場するすべての要素に共通するコードを含めることもできます。

その場合、Actorを抽象クラスとして記述できます。すると、Carクラスでの継承の宣言は次のようになります。

```
class Car extends Actor implements Drawable, Serializable ...
```

しかし、コードを含めたいインタフェースが複数ある場合でも、それらのインタフェースすべてを抽象クラスにすると動作しなくなります。先ほど説明したように、クラスの場合、Javaでは1つしか継承できません（つまり、extendsキーワードの後には1つのクラスしか書くことができません）。多重継承できるのはインタフェースだけです。

しかし、これには解決策があります。Java 8で導入されたデフォルト・メソッドです。詳しくは後ほど説明します。

空インタフェース

ときどき、インタフェースの名前だけ定義されており、メソッドのない空のインタフェースを目にすることがあります。先ほど登場したSerializableもその1つです。Cloneableも同様です。これらのインタフェースは、マーカー・インタフェースと呼ばれています。マーカー・インタフェースは、一定の特性を持つクラスを識別するもので、その目的は、型の実装やプログラムのパーツ間での取り決めの定義よりも、メタデータの提供に密接に関連しています。Javaでは、バージョン5でアノテーションが導入されました。メタデータの提供という意味では、アノテーションの方が優れた方法です。今となっては、Javaでマーカー・インタフェースを使う意味はほとんどありません。興味がある方は、代わりにアノテーションの使用について調べてみてください。

Java 8による新たな夜明け

ここまで、Java 8で導入されたいくつかの新機能にはあえて触れずにきました。その理由は、Java 8では、それより前の言語設計上の決定事項（「コードの継承は1つのみ」など）とは矛盾する機能が追加されていることで、一部の構造間における関係性の説明がとても難しくなってしまうからです。たとえば、インタフェースと抽象クラスの違い

とそれぞれの存在理由についての話も、非常に込み入った話になってしまいます。この後で説明しますが、Java 8でのインタフェースは拡張されて抽象クラスにいっそう類似したものになっています。しかし、依然として微妙な違いは残っています。

問題の説明は、歴史の流れに沿って進めてきており、最初にJava 8より前の状況を説明しました。これから、新しいJava 8の機能についてお話しします。説明はあえてこの順番にしました。歴史を考慮しなければ、現在のような機能の組合せになっていることの正当性を理解できないからです。

もし現時点でJavaチームがJavaをゼロから設計し、下位互換性が問題にならないとするなら、同じような設計にはならないでしょう。しかし、Java言語が優先しているのは理論上の課題ではなく、実際に利用されているシステムです。そして、現実の世界では、以前に行ったことを何も壊すことなく、言語を進化、拡張させる方法を見つけなければなりません。インタフェースのデフォルト・メソッドと静的メソッドは、Java 8での進化を可能にした2つのメカニズムです。

進化するインタフェース

Java 8を開発した際に問題となったことの1つに、インタフェースをどのように進化させるかということがありました。Java 8では、Java言語にラムダ式などの複数の機能が追加されており、Javaライブラリ内における既存のインタフェースの一部を、それらの機能に適応させることが好ましくなりました。しかし、インタフェースを使用している既存のコードを一切壊さずにそのインタフェースを進化させるには、どうすればよいのでしょうか。

既存のライブラリにMagicWandというインタフェースがある場合を考えてみます。

```
public interface MagicWand
{
   void doMagic();
}
```

このインタフェースはすでに使用されており、多くのプロジェクトの多くのクラスによって実装されています。しかし、とてもすばらしい新機能を考えつき、非常に便利なメソッドを新しく追加したいと思いました。

```
public interface MagicWand
{
```

```
    void doMagic();
    void doAdvancedMagic();
}
```

これを実行した場合、このインタフェースを以前に実装したすべてのクラスが壊れます。すべてのクラスで、この新しいメソッドの実装が必要になるからです。そのため、一見しただけでは、どうしようもないように思われます。既存のユーザーのコードを破壊するか（この方法は避けたいものです）、改善が容易でないと見込まれる古いライブラリを使い続けるしかないということです（実際には、試すことができるいくつかのアプローチがあります。たとえば、サブインタフェースによるインタフェースの拡張などです。とはいえ、そのようなアプローチにもそれぞれの問題点がありますが、説明は割愛します）。Java 8には、既存のコードを壊さずに既存のインタフェースに何かを追加する機能という、両方のよいところだけを集めた賢い解決策があります。解決策には、デフォルト・メソッドおよび静的メソッドを使います。次は、この2つについて説明します。

デフォルト・メソッド

デフォルト・メソッドは、メソッド本体、すなわちデフォルトの実装を持つインタフェースのメソッドです。デフォルト・メソッドは、メソッドのシグネチャの先頭にdefault修飾子を付けて定義するもので、完全なメソッドの本体を持ちます。

```
public interface MagicWand
{
    void doMagic();
    default void doAdvancedMagic()
    {
        ... // ここにコードを記述
    }
}
```

このインタフェースを実装するクラスは、（オーバーライドによって）このメソッドでクラス独自の実装を提供することもできますが、このメソッドを完全に無視することもできるようになっています。その場合、インタフェースからデ

フォルトの実装を受け取ることになります。これによって、古いコードを動かし続けながら、新しいコードでは新しい機能を使えるようになります。

静的メソッド

さらに、実装を持つ静的メソッドをインタフェースに含めることもできるようになっています。静的メソッドは、メソッドのシグネチャの先頭に通常のstatic修飾子を付けて定義します。いつものように、インタフェースを書くときはpublic修飾子を省略して構いません。インタフェース内のすべてのメソッドとすべての定数は必ずpublicになるためです。

ダイヤモンド継承問題はどうなるか

おわかりのように、抽象クラスとインタフェースはほとんど同じになっています。構文は異なるものの、いずれも抽象メソッドと実装のあるメソッドを含むことができます。この2つには依然としていくつかの違いがあります（たとえば、抽象クラスはインスタンス・フィールドを持つことができますが、インタフェースではできません）が、この違いは中核的な問題です。すなわち、Java 8のリリース以来、（インタフェースによる）コードを含むことのできる多重継承が存在しているということです。

本記事の冒頭では、どのようにしてJavaの設計者が非常に注意深くコードの多重継承を避けてきたかに注目しました。多重継承を認めてしまうと、複数回の継承や名前の衝突を主な原因とする問題が発生する可能性があるからです。それでは、現在の状況はどうなっているのでしょうか。

いつものように、Java設計者は、これらの問題に対処するために、次のような賢明で現実的な規則を考え出しました。

- 同じ名前の複数の抽象メソッドを継承しても、それらの抽象メソッドは同じメソッドと見なされるため問題ありません。
- インタフェースに定数以外のフィールドを含めることはできないため、難しい問題の1つであるフィールドのダイヤモンド継承も発生しません。
- 静的メソッドと定数（定義上、インタフェースの定数も静的です）の継承も、使用する際にインタフェース名を接頭辞として付けなければならないため、名前が衝突することはなく、問題にはなりません。
- 同じシグネチャと異なる実装を持つ複数のデフォルト・メソッドを別々のインタフェースから継承することは問題になります。しかし、Javaでは、他の言語よりもはるかに現実的な解決策が選択されています。この問題に対応す

るために新たな言語構造を定義するのではなく、コンパイラがエラーを報告するだけとなっています。つまり、これは開発者側の問題です。Javaが伝えるのは、「これをやってはいけません」ということだけです。

まとめ

インタフェースはJavaの強力な機能です。プログラムの別々のパーツ間での取り決めを定義する、動的ディスパッチ用の型を定義する、型の定義と実装を分ける、Javaで多重継承を実現するなど、さまざまな状況で役に立ちます。コードを書く際に頻繁に活用できるため、インタフェースの動作はきちんと押さえておく必要があります。

デフォルト・メソッドなどのJava 8でのインタフェースの新機能がもっとも役に立つのは、ライブラリを書く場合です。アプリケーションのコードで利用する機会はあまりないかもしれません。しかし、Javaライブラリではこれらの新機能が多用されているため、何をしているのかは知っておくようにしましょう。インタフェースを注意深く使うことができれば、コードの質を大幅に改善できます。</article>

[編集注：本記事は、Java Magazine2016年12月発行号に掲載されたものを改訂したものです。]

Michael Kölling：Java Championで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。