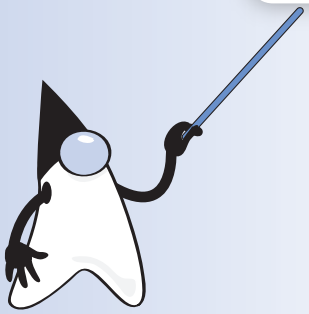
第14章

インタフェース

本章では、インタフェースについて学習します。そのままでは利用できないインタフェースは、クラスを作る際に“実装”することによって利用します。インタフェースの実装は、派生によるクラスの階層関係とは異なる関係をクラス間に与えます。

□インタフェース宣言
□インタフェースの実装
□クラスの派生とインタフェースの実装
□インタフェースの継承



14-1 インタフェース

本節では、参照型の一種であるインタフェースの基本を学習します。インタフェースは、クラスに似ていると同時に、いろいろな点で異なります。しっかりと学習しましょう。

インタフェース

本章では、**インタフェース**（*interface*）を学習します。クラスを『回路の設計図』にたとえるのと同じ感じで表現すると、インタフェースは『リモコンの設計図』となります。

▶ interface とは、『境界面』『共有域』といった意味の語句です。

インタフェース宣言

具体的な例を考えていきましょう。ここでは、ビデオプレーヤ・CD プレーヤ・DVD プレーヤといった、プレーヤ（再生機）を例にとります。どのプレーヤも『再生』や『停止』といった操作ができますね。プレーヤ内での実際の動作は、各プレーヤごとに異なるものの、リモコンに《再生ボタン》と《停止ボタン》がある点は共通です。

共通部のリモコンのイメージを表したのが、**Fig.14-1** a です。

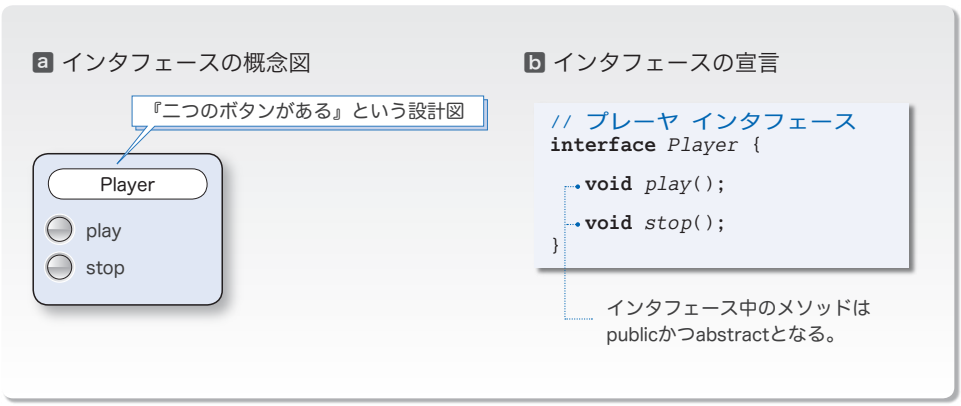


Fig.14-1 プレーヤインタフェース（リモコンの設計図）

『*Player* リモコンは、*play* ボタンと *stop* ボタンの二つから構成される。』という、リモコンの設計図をプログラムとして表したのが、図 b に示す**インタフェース宣言**（*interface declaration*）です。一見クラス宣言と似ていますが、先頭のキーワードが **class** ではなく **interface** となる点で、クラスとは異なります。

また、インタフェース内のすべてのメソッドは、**public** かつ **abstract** です（**public** や **abstract** を付けて宣言しても構いませんが、冗長になるだけです）。

メソッド本体 { ... } の代わりに ; を付けて宣言しなければならない点は、クラスにおける**抽象メソッド**と同じです（p.428）。

インタフェースの実装

それでは、インタフェース内で宣言された抽象メソッドの実体はどこで定義するかというと、そのインタフェースを**実装する**（*implement*）クラスの中です。

インタフェース *Player* を実装するクラス *VideoPlayer* の宣言が、**Fig.14-2** a です。“**implements** *Player*”の部分が、インタフェース *Player* を実装することを示します。この宣言は、派生クラスの宣言と似ていますね。ただし、用いるキーワードは **extends** ではなくて **implements** となります。

▶ 前章では、『抽象メソッド』の実装について学習しました。ここで学習しているのは、『インタフェース』の実装です。

クラス *VideoPlayer* の宣言は、以下のように理解しましょう。

> このクラスでは、*Player* リモコンを実装します。そのために、各ボタンによって呼び出されるメソッドの本体も実装しますよ！

この関係を図示したのが、図 b です。本書では、クラスと見分けがつくように、インタフェースの枠を青色で表します。さらに、あるクラスがインタフェースを実装することを、クラスからインタフェースに向かって**青い点線**で結ぶことによって表現します。

Fig.14-2　インタフェースの実装

クラス *VideoPlayer* は、インタフェース *Player* を実装するとともに、メソッド *play* と *stop* を実装します（メソッドをオーバライドして本体を定義します）。

オーバライドするメソッドは、**public** 宣言しなければなりません。インタフェースのメソッドが **public** であり、それよりもアクセス制限を強めることができないためです。これは、クラスの派生におけるオーバライドの場合と同じです（p.418）。

> **重要**　インタフェースのメソッドは **public** かつ **abstract** である。それを実装するクラスでは、メソッドに **public** 修飾子を与えて実装する。

インタフェース *Player* を実装したビデオプレーヤクラス *VideoPlayer* と CD プレーヤクラス *CDPlayer* を作りましょう。プログラムを **List 14-1** ～ **List 14-3** に示します。

List 14-1 player/Player.java

```
// プレーヤ インタフェース
public interface Player {
   void play();          // ○再生
   void stop();          // ○停止
}
```

List 14-2 player/VideoPlayer.java

```
//===== ビデオプレーヤ =====//
public class VideoPlayer implements Player {
   private int id;                    // 製造番号
   private static int count = 0;      // 現在までに与えた製造番号

   public VideoPlayer() {                                  // コンストラクタ
      id = ++count;
   }

   public void play() {                                    // ○再生
      System.out.println("■ビデオ再生開始！");
   }

   public void stop() {                                    // ○停止
      System.out.println("■ビデオ再生終了！");
   }

   public void printInfo() {                               // 製造番号表示
      System.out.println("本機の製造番号は[" + id + "]です。");
   }
}
```

List 14-3 player/CDPlayer.java

```
//===== ＣＤプレーヤ =====//
public class CDPlayer implements Player {

   public void play() {                                    // ○再生
      System.out.println("□ＣＤ再生開始！");
   }

   public void stop() {                                    // ○停止
      System.out.println("□ＣＤ再生終了！");
   }

   public void cleaning() {                                // クリーニング
      System.out.println("□ヘッドをクリーニングしました。");
   }
}
```

これらのインタフェースとクラスのイメージを表したのが **Fig.14-3** です。*VideoPlayer* のリモコンと、*CDPlayer* のリモコンは、いずれも *Plyaer* のリモコンのボタンである *play* と *stop* を含むものとなっています。『クラス *VideoPlayer* と *CDPlayer* が、インタフェース *Player* を実装している』ことのイメージをつかんでください。

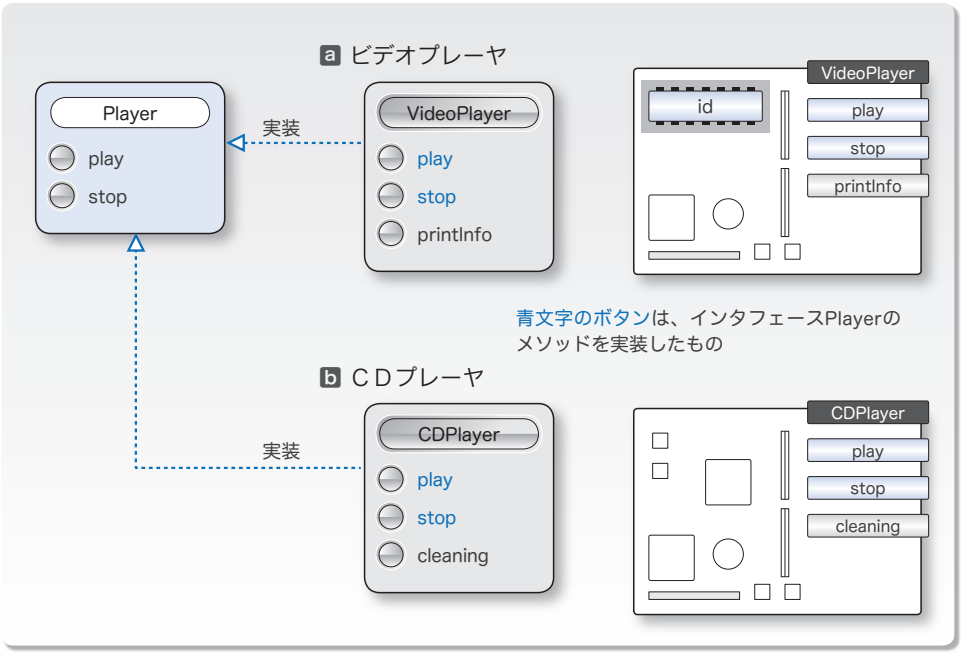


Fig.14-3　インタフェースを実装したクラス

二つのクラスの概略は、次のようになっています。

■ クラス *VideoPlayer*

メソッド *play* は『■ビデオ再生開始！』と表示し、メソッド *stop* は『■ビデオ再生終了！』と表示します。

このクラスのインスタンスには、生成するたびに製造番号として 1，2，3，… を与えます。番号の与え方は、第 10 章で学習した《識別番号》と同じ要領です。

個々のインスタンスに与える製造番号がインスタンス変数 *id* であり、何番まで与えたのかを表すのがクラス変数 *count* です。

メソッド *printInfo* は、製造番号の表示を行います。

■ クラス CDPlayer

このクラスには、フィールドはありません。

メソッド *play* は『□ＣＤ再生開始！』と表示し、メソッド *stop* は『□ＣＤ再生終了！』と表示します。

メソッド *cleaning* は、『□ヘッドをクリーニングしました。』と表示します。

二つのクラスは、インタフェース *Player* のフィールドやメソッドなどの**資産を継承しているのではない**ことに注意しましょう。受け継いでいるのは、*Player* の**メソッドの仕様**（リモコン上のボタンの仕様）のみです。

インタフェースを使いこなす

インタフェースを使いこなすために、いくつかの文法的な決まりや制限などを理解していきましょう。

■ インタフェース型のインスタンスを生成することはできない。

インタフェースは、リモコンの設計図に相当するのでしたね。そのため、実体である回路（インスタンス）を作ることはできません。以下の宣言はエラーとなります。

```
Player c = new Player();            // エラー
```

重要 インタフェース型のインスタンスを生成することはできない。

実体を生成できないという点は、クラスと大きく異なります。

▶ ただし、抽象クラスとは似ています。

■ インタフェース型の変数は、それを実装したクラスのインスタンスを参照できる。

インタフェース型の変数は、そのインタフェースを実装したクラスのインスタンスを参照できることになっています。したがって、以下のようなことができます。

```
Player p1 = new CDPlayer();         // ＯＫ
Player p2 = new VideoPlayer();      // ＯＫ
```

Player リモコンの変数は、それを実装したクラスである *CDPlayer* や *VideoPlayer* のインスタンスを参照できるのです。

重要 インタフェース型の変数は、実装クラスのインスタンスを参照できる。

これは、不思議なことでも不自然なことでもありません。というのも、*CDPlayer* の回路も、*VideoPlayer* の回路も、『*play* ボタン』と『*stop* ボタン』で操作できるからです。

この点では、**スーパークラス型の変数が、サブクラス型のインスタンスを参照できるのと似ています。**

*

実際のプログラムで確認してみましょう。それが **List 14-4** に示すプログラムです。

配列 *a* は、インタフェース *Player* 型を要素型とする配列です。*a*[0] は *VideoPlayer* のインスタンスを参照し、*a*[1] は *CDPlayer* のインスタンスを参照しています。

拡張 **for** 文では、各要素に対して、メソッド *play* とメソッド *stop* を順次呼び出しています（**Fig.14-4**）。

▶ 動的結合（p.410）が行われますから、参照先のインスタンスに所属するメソッドが呼び出されることになります。

List 14-4 player/PlayerTester.java

```
// インタフェースPlayerの利用例

class PlayerTester {

   public static void main(String[] args) {
      Player[] a = new Player[2];
      a[0] = new VideoPlayer();  // ビデオプレーヤ
      a[1] = new CDPlayer();     // ＣＤプレーヤ

      for (Player p : a) {
         p.play();               // 再生
         p.stop();               // 停止
         System.out.println();
      }
   }
}
```

実行結果

```
■ビデオ再生開始！
■ビデオ再生終了！

□ＣＤ再生開始！
□ＣＤ再生終了！
```

図からも分かるように、インタフェース *Player* 型のリモコンがもつボタンは、*play* と *stop* だけです。そのため、a[0] や a[1] を通じて、*printInfo* あるいは *cleaning* のメソッドを呼び出すことはできません。

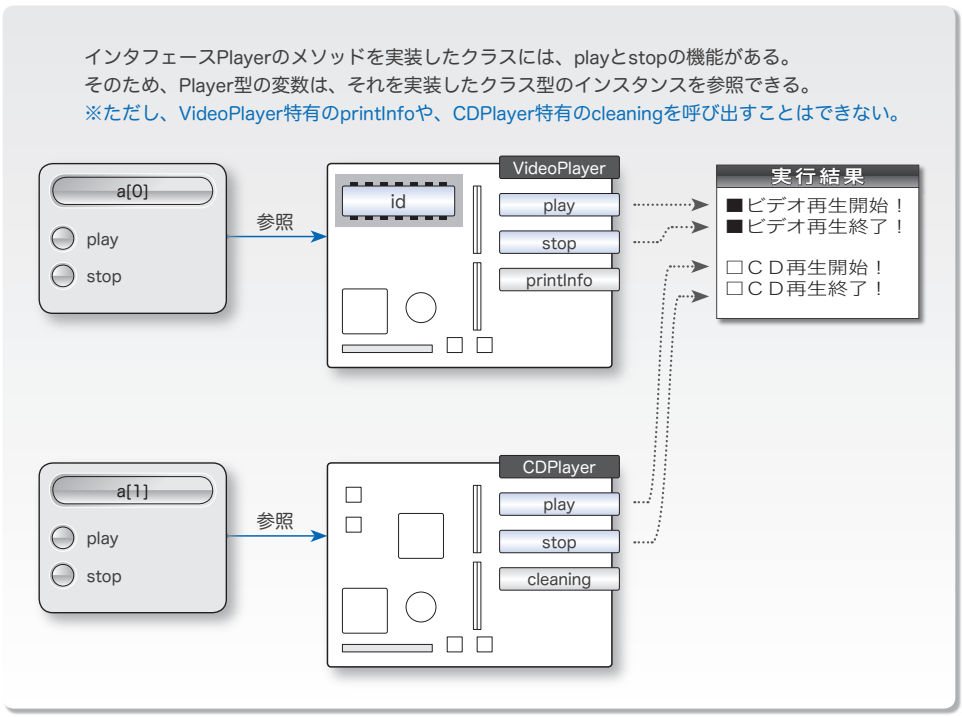


Fig.14-4 インタフェースを実装したクラス

なお、クラス型変数の場合と同様に、インタフェース型の変数に対しても **instanceof** 演算子（p.413）を適用して、参照先のインスタンスの型の判定を行うことができます。