インターフェイス

ビヘイビアの定義

C# は C++ 言語のように多重継承を行うことはできません
常に直列的な継承を行うことで、インスタンスや名前の衝突を避けるという目的があります
しかし、これだけでは完成されたオブジェクト指向プログラムを作成することはできません

そこで、多重継承ではなく**インターフェイス**という技術が使われます
これは、Java 言語でも使われており、Java プログラマにとってはお馴染みのものです

インターフェイスは抽象的にクラスの動作「ビヘイビア」を宣言します
しかし、インターフェイスはその実装を持つことはなく、ただのシグネチャの宣言でとどまります
そして、クラスがインターフェイスを実装することではじめて意味をなします

クラスは1つの基底クラスしかもつことができませんが、インターフェイスはいくつも実装できます
「インターフェイスを継承する」という表現も存在しますが、これは論理的な解釈であり
物理的な動作を見ると「実装する」という表現が適確なのでこの場では「実装する」という表現を使います

クラス宣言では、クラスベースの最初の識別子がクラスならばそのクラスを継承し
その後、カンマ (,) で区切ってインターフェイス型リストを指定します
最初の識別子がインターフェイスならば、インターフェイス型リストであると判断します

インターフェイスの役割は、クラスに対する契約です
インターフェイスを実装するクラスは、インターフェイスで宣言される全てのメンバ
すなわち、宣言されている**ビヘイビアを必ず実装**しなければなりません
宣言されているメソッドなどが実装されていなければ、そのクラスはコンパイルできません

インターフェイスの宣言は **interface** キーワードを用います

[attributes] [modifiers] interface identifier [:base-list] {interface-body}[;]

インターフェイスの定義は、クラスのそれとほとんど変わりません
attributes に属性、modifiers に修飾子、identifier に識別子
そして、base-list に基底インターフェイスなどを指定することもできます
interface-body には、インターフェイスのメンバを定義します

ただし、インターフェイスのメンバは実装のないビヘイビアです
インターフェイスはクラスと交わす契約書にすぎません
また、インターフェイスは常に公開される必要があるため、メンバはアクセス可能性を指定されません

実装のないビヘイビアとは、つまりはシグネチャだけを定義するものです
メソッド、プロパティ、インデクサなどを指定することができます
（まだ説明していませんが、イベントというものも指定できます）

type method(list);
type property { get; set; }
type this [list] { get; set; }

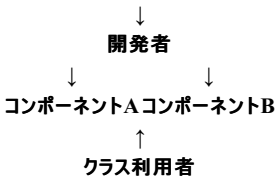
上からメソッド、プロパティ、インデクサのインターフェイスでの宣言構文です
type には型、list にはパラメータリストを指定することができます

```
interface KittyStandard {
 void WriteName();
}

class Kitty : KittyStandard {
 private string name;
 public Kitty(string name) {
  this.name = name;
 }
 public void WriteName() {
  System.Console.WriteLine(name);
 }
}

class Test {
 static void Main() {
  Kitty rena = new Kitty("RENA");
  rena.WriteName();
 }
}
```

このプログラムでは KittyStandard というインターフェイスを定義しています
このインターフェイスを実装するクラスは、必ずインターフェイスで宣言しているビヘイビアを
クラス内で再定義して、その実装を保有しなければコンパイルすることができません

つまり、実装がどのような形であれ KittyStandard 型は WriteName() メソッドを持ちます
プロジェクトの設計者がインターフェイスを設計してその仕様をアプリケーション設計者に指示すれば
異なるコンポーネント間でも、インターフェイスを通じて同じ操作を保証することができるのです

設計者
↓
インターフェイス



クラスの利用者は、例えコンポーネントの開発者が異なっていても
開発者が設計者のインターフェイス仕様を守ってコンポーネントを開発していれば
まったく同じ操作を異なるコンポーネント間で保証されるのです

```
interface KittyStandard {
 void WriteName();
}

enum KittyName { RENA , YUKI , MIMI }

class Kitty_On_Your_Lap : KittyStandard {
 private string name;
 public Kitty_On_Your_Lap(KittyName name) {
  this.name = name.ToString();
 }
 public void WriteName() {
  System.Console.WriteLine(name);
 }
}

class Di_Gi_Charat : KittyStandard {
 private const string name = "Di Gi Charat";
 public void WriteName() {
  System.Console.WriteLine(name);
 }
}

class Test {
 static void Main() {
  KittyStandard kitty = new Kitty_On_Your_Lap(KittyName.RENA);
  kitty.WriteName();

  kitty = new Di_Gi_Charat();
  kitty.WriteName();
 }
}
```

このプログラムを理解できれば、インターフェイスの強力さがわかるでしょう
インターフェイス KittyStandard はそれだけでもインターフェイス型として独立しています
KittyStandard 型は、実装はともかく WriteName() というシグネチャを保証しているのです

KittyStandard インターフェイスを実装するクラスは、常に WriteName() を保証します
そのため、KittyStandard を実装するクラスは KittyStandard 形であるとも考えられ
このプログラムのように異なるクラスを同一の型として扱うことができるのです

このプログラムでは Kitty_On_Your_Lap クラスと Di_Gi_Charat クラスがあります
それぞれ KittyStandard インターフェイスを実装しているため
クラスの関連がなくても KittyStandard 型として一貫した操作を提供しているのです
Main() メソッドの操作を見れば、これは明らかでしょう

名前の隠蔽

インターフェイスを実装するクラスは必ずそのインターフェイスで定義されている
メソッドやプロパティといったビヘイビアを実装することを義務付けられます

そのため、インターフェイス型のメソッドにはインターフェイス型でアクセスするこも
インターフェイスを実装するクラス型でアクセスすることも可能です
しかし、これが場合によっては不要な混乱や問題を発生させることもあります

問題なのは、インターフェイス型とクラス型のメソッドの名前の衝突です
インターフェイス型は、仕様を持った意味のあるシグネチャの定義であり
できることならクラス型ではなくインターフェイス型としてアクセスされることが好ましく
実装するクラスのメソッドとしてアクセスされるのは、時として好ましくありません

そこで、**インターフェイス方の名前を隠蔽**する方法があります
実は、クラスのメンバの宣言はインターフェイスを指定する構文も存在します

type interfacce-name.member...

type にはメンバ型を、interface-name にはインターフェイスを指定します
インターフェイスを指定する場合、アクセス可能性を指定することはできません
インターフェイスで定義されているビヘイビアは常に公開される必要があります

こうして、インターフェイス名でメンバを修飾することでクラスと切り離すことができます
クラスはインターフェイスを実装しますが、クラスからはその名前を隠蔽するのです
このインターフェイスの実装にアクセスするには、インターフェイス型にキャストしなければなりません

```
interface KittyStandard {
 void WriteName();
```

```
}

enum KittyName { RENA , YUKI , MIMI }

class Kitty : KittyStandard {
 private string name;
 public Kitty(KittyName name) {
  this.name = name.ToString();
 }
 public void WriteName() {
  System.Console.WriteLine("Kitty on your lap");
 }
 void KittyStandard.WriteName() {
  System.Console.WriteLine(name);
 }
}

class Test {
 static void Main() {
  Kitty kitty = new Kitty(KittyName.YUKI);
  kitty.WriteName();
  ((KittyStandard)kitty).WriteName();
 }
}
```

このプログラムの Kitty クラスは二つの WriteName() メソッドを持ちます
コンパイルして実行すると次のような結果になりました

Kitty on your lap
YUKI

重要なのは、Kitty クラスの void KittyStandard.WriteName() メソッドです
このメソッドは KittyStandard で修飾することによって名前を隠蔽しているのです

そのため、Main() メソッドで Kitty 型として WriteName() でアクセスした場合
Kitty クラスの WriteName() メソッドを呼び出され
KittyStandard にキャストして WriteName() を実行すると
KittyStandard の WriteName() メソッドが呼び出されます

インターフェイスの結合

インターフェイスは、他のインターフェイスを含むことで結合させることができる
構文はクラスの継承と同じで、他のインターフェイスを継承するような形になります

この機能を用いれば、いくつかのインターフェイスを合成し
必要であればさらに新しい機能を定義することが可能である

```
interface A {
 void WriteA();
}

interface B {
 void WriteB();
}

interface C : A , B {}

class Kitty : C {
 public void WriteA() {
  System.Console.WriteLine("Kitty on your lap");
 }
 public void WriteB() {
  System.Console.WriteLine("Silver Gene");
 }
}

class Test {
 static void Main() {
  Kitty obj = new Kitty();
  obj.WriteA();
  obj.WriteB();
 }
}
```

このプログラムのインターフェイス C は A と B を結合したものです
新しい機能の追加はありませんが、A の定義と B の定義を結合した状態にあります

---

interface

[attributes] [modifiers] interface identifier [:base-list] {interface-body}[;]

インターフェイスを宣言します

attributes - 属性を指定します
modifiers - 修飾子を指定します
identifier - 識別子を指定します
base-list - 他の結合するインターフェイスを指定します
intarface-body - ビヘイビア（操作特性）を定義します
