継承

クラスを拡張する

オブジェクト指向プログラムの重要な機能の一つである「継承」は
プログラムの拡張や移植といった保守を容易にしてくれる極めて高度な技術です
今回は、この継承の基本を詳しく解説します

継承とは、あるクラスを基底とした新しいクラスを生成することです
新たに生成されたクラスは、基底のクラスの能力をデフォルトで持っているのです
class キーワードの書式を再び思い出してください

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

このうちの base-list に拡張する基底のクラス名を指定します
すると、このクラスは指定した基底クラスの能力を自動的に継承するようになります

このとき、基底となるクラスを**基底クラス**と呼び
基底クラスを継承した新しいクラスを**派生クラス**と呼びます
派生クラスは暗黙的に**基底クラスのインスタンスを持つ**と考えられ
派生クラスのインスタンスは暗黙的に基底クラスのメンバにアクセスすることができるのです

これは、何らかの共通点を持つ異なる物質（オブジェクト）を表す時に使用できます
RPG ゲームを作る時、「戦士」や「魔法使い」というような異なるジョブ（職業）を設定する場合でも
お互いに共通する基底クラスを生成すれば、後はそれを拡張するだけで
作業量を大幅に削減して、多くのジョブクラスを生成することができるのです
この場合は、各ジョブ特有のパラメータや動作は派生クラスで新たに設定することになるでしょう

```
class Kitty {
 public string name;
 public void WriteName() {
  System.Console.WriteLine(name);
 }
}

class TokyoMM : Kitty {
 public string ability;
 public void Attack() {
  WriteName();
  System.Console.WriteLine(ability);
 }
 static void Main() {
  TokyoMM ichigo = new TokyoMM();
  ichigo.name = "いちご";
  ichigo.ability = "リボーン・ストロベリー・チェーック!!";
  ichigo.Attack();
 }
}
```

このプログラムのTokyoMM クラスは Kitty クラスを基底クラスとした派生クラスです
Main() メソッドでは TokyoMM クラスのインスタンスを生成し
オブジェクト ichigo に各値を設定して Attack() メソッドを呼び出します

ここで注目するべき点があります
TokyoMM クラスには name メンバ変数も WriteName() メソッドも定義していません
しかし、TokyoMM クラスの Attack() メソッド内で WriteName() を呼び出していますし
Main() メソッドで name メンバ変数に文字列を代入しようとしています

これらのメンバは基底クラスである Kitty クラスのメンバであることがわかります
そして、TokyoMM クラスは Kitty の派生クラスなので
デフォルトの状態でこれらの機能、name メンバ変数や WriteName() メソッドを持つのです
このプログラムを実行した結果、次の文字列がコンソールに表示されます

いちご
リボーン・ストロベリー・チェーック!!

プログラムがこちらの意図どうりに動作したことが証明されました
この結果からも「基底クラスを拡張する」という継承の具体的な概念を理解していただけたでしょう

C++ 言語では、「多重継承」と呼ばれる複雑なクラス階層を持つことがありました
多重継承は、複数の基底クラスを持つ派生クラスを生成することです
C# で記述するならば、次のような形になります

class A {}
class B {}
class C : A , B {}

これは、派生クラス C が A と B クラスを継承していることを表しますが
C# では**多重継承はできない**ため、コンパイラはエラーを出します
なぜ多重継承をできないかというと、多重継承は時に複雑な重複を生み出してしまいます

class A {}
class B : A {}
class C : A {}

class D : B , C

この多重継承には大きな問題があります
クラス D は B と C クラスを継承していますが、B と C クラスは A クラスを継承しています
そのため、D クラスは B クラスと C クラスが持つ二つの A クラスのインスタンスを持ってしまうのです

C++ 言語は A クラスを仮想化することでこの重複による曖昧さを解決しましたが
Java 言語が多重継承を禁止し、より簡素な「インターフェイス」と呼ばれる方法を選択し
これによって複雑な多重継承を言語に実装しなくても
十分にオブジェクト指向プログラムができることを証明しています

C# のこの原理に基いて、多重継承を禁止しています
インターフェイスの技術については、後ほど詳しく紹介することになるでしょう

派生クラスの名前の衝突

継承を行うと、ある問題が発生します
派生クラスのメンバが、基底クラスのメンバ名と衝突した場合はどうなるのでしょうか

オブジェクト指向の思想上、基底クラスはカプセル化され
継承した派生クラスでは基底クラスのソースを直接見なくてもプログラムできる必要があり
名前の衝突をプログラマが気にすることなくプログラムできなければなりません

そこで、派生クラスで基底クラスと同一のメンバ名が宣言された場合
派生クラスからは基底クラスと同じメンバを「隠蔽」することになります

```
class Kitty {
 public string str = "Kitty on your lap";
}

class TokyoMM : Kitty {
 public string str = "Tokyo mew mew";
 static void Main() {
  TokyoMM obj = new TokyoMM();
  System.Console.WriteLine(obj.str);
 }
}
```

このプログラムを見ると Kitty クラスを継承した TokyoMM クラスのメンバで
str メンバ変数が基底クラスの str メンバ変数と衝突しています
これを実行した結果、やはり TokyoMM の str メンバ変数にアクセスされます
この結果から、Kitty クラスの str メンバ変数が隠蔽されたことを確認できます

しかし、C# は**隠蔽にも非常にうるさい**プログラム言語です
プログラマが意図せずに隠蔽してしまったことによるバグを避けるために
派生クラスが基底クラスのメンバを隠蔽するには **new 修飾子** を指定します
そうしなければ、コンパイル時「警告」を発生させるようになっています

```
class Kitty {
 public string str = "Kitty on your lap";
}

class TokyoMM : Kitty {
 new public string str = "Tokyo mew mew";
 static void Main() {
  TokyoMM obj = new TokyoMM();
  System.Console.WriteLine(obj.str);
 }
}
```

TokyoMM クラスの str メンバ変数には new 修飾子が指定されています
これによって、コンパイラはプログラマが明示的にメンバを隠蔽していることを知り
メンバが隠蔽されているという警告を発生させなくなります

同様の概念でメソッドを隠蔽することが可能です
ただし、これはオブジェクト指向のポリモーフィズムに深い関わりがあります
メソッドの隠蔽については「オーバーライド」を参照してください

基底クラスの参照

インスタンスメソッドが this ポインタとしてインスタンスへの参照を持つのと同様に
派生クラスは基底クラスへのインスタンス **base** ポインタを持っています
派生クラスで基底クラスのメンバを隠蔽した場合、このポインタを使って
派生クラスのメソッドから基底クラスのメンバにアクセスすることができるようになります

base . member

base は常にそのクラスの基底クラスの参照を格納しています
基底クラスの隠蔽したメンバ変数やメソッドには、base を介してアクセスします

```
class Kitty {
 public string str = "Kitty on your lap";
}

class TokyoMM : Kitty {
```

```
  new public string str = "Tokyo mew mew";
  public void Write() {
   System.Console.WriteLine(base.str);
   System.Console.WriteLine(str);
  }
  static void Main() {
   TokyoMM obj = new TokyoMM();
   obj.Write();
  }
 }
```

このプログラムを実行した結果、次の文字列が表示されました

Kitty on your lap
Tokyo mew mew

この結果から、base.str が基底クラスの str メンバ変数にアクセスできていることが確認できます

---

new

new 修飾子は修飾したメンバが基底クラスのメンバを隠蔽していることを明示します
基底クラスのメンバを隠蔽し、かつ省略した場合はコンパイル警告が発生します

base

基底クラスのインスタンスへの参照です
派生クラスのインスタンスメンバで使用することができます

前のページへ　戻る　次のページへ