

4-1 オブジェクト指向とは

オブジェクト指向のとらえかた

オブジェクト指向とは**プログラムを書く時の考え方**である。

4章ではオブジェクト指向の3大要素である**継承・多態性(ポリモーフィズム)・カプセル化**を学ぶ。
また、それらを支える**クラスとインスタンス**についても学ぶ。

オブジェクト指向の本質

オブジェクト指向ではクラスを軸にして考える。

クラス(登場人物)に属性(メンバ変数)と操作(メソッド)を持たせ、クラス自体に意味を持たせる。
プログラムをクラスの集合体で表現するのがオブジェクト指向である。

4-2 クラスとインスタンス

クラスとは

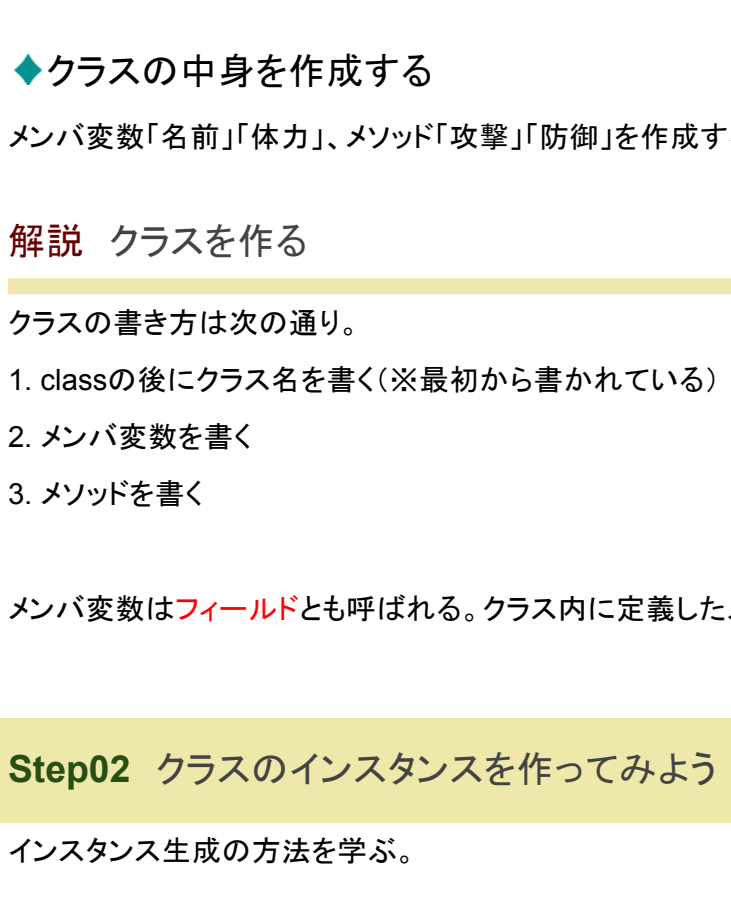
教科書的にクラスとは「関連のある変数とメソッドをまとめたもの」とある。
それだけでは不十分で、Javaで学んだ登場人物・情報保持責任・行動責任の考えを取り入れるべき。

クラスのまとまりを見つける方法

内容

- ◆手順①アプリケーション内で変化する可能性のあるモノを書き出す
「変化する可能性」というのが漠然としすぎてよくわからないはず。Javaで学んだ「登場人物」で捉えたと良い。
- ◆手順②書き出したモノそれぞれに属する値を列挙する
Javaで学んだ「属性」で捉えたと良い。
- ◆手順③書き出したモノそれぞれに対応する動作や処理を列挙する
Javaで学んだ「操作」で捉えたと良い。

Step01 プレイヤクラスを作ってみよう



メンバ変数に名前と体力、メソッドに攻撃と防御を持つプレイヤクラスを作る。

クラス図では次の様になる。

C プレイヤ

+string 名前
+int 体力
+void 攻撃()
+void 防御()

- ◆新規プロジェクトを作成する
スタートウィンドウを表示 > 新しいプロジェクトの作成 > コンソールアプリ(.NET Core) > プロジェクト名を「SampleRPG」と入力 > 作成ボタンをクリック
- ◆クラスを追加する
ソリューションエクスプローラーの「SampleRPG」を右クリック > 追加 > 新しい項目 > 左側の項目から「Visual C#アイテム」を選択 > 中央の項目で「クラス」を選択 > 画面下部の「名前」欄に「Player.cs」を入力 > 「追加」ボタンをクリック

📁ファイル名とクラス名

Javaでは基本的にはファイル名(xxx.java)とクラス名は一致させなければならなかった。
C#では一致させなくても作成できるが、管理が大変なので統一することを推奨する。

◆クラスの中身を作成する

メンバ変数「名前」「体力」、メソッド「攻撃」「防御」を作成する。

解説 クラスを作る

クラスの書き方は次の通り。

1. classの後にクラス名を書く(※最初から書かれている)
2. メンバ変数を書く
3. メソッドを書く

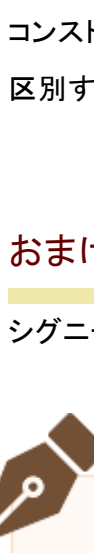
メンバ変数は**フィールド**とも呼ばれる。クラス内に定義したメソッドは**メンバメソッド**と呼ばれる。

Step02 クラスのインスタンスを作ってみよう

インスタンス生成の方法を学ぶ。

解説 クラスのインスタンスを作る

インスタンス生成は**new演算子**で行う。

 構文4-1 new演算子の構文

クラス型の変数 = new クラス名();

使い方はJavaと同様である。
インスタンス経由でメンバ変数とメソッド(メンバメソッド)にアクセスできる。
メンバ変数へアクセス
→クラス型の変数.メンバ変数名
メソッドへアクセス
→クラス型の変数.メソッド名

Step03 コンストラクタで初期値の代入忘れを防ごう

コンストラクタ定義の方法を学ぶ。

解説 コンストラクタでメンバ変数に初期値を代入する

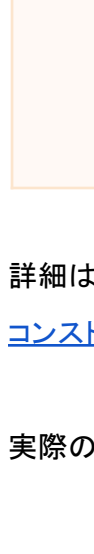
コンストラクタは**インスタンス生成時に必ず呼び出される**。

※Javaで学んだ通り、**直接コンストラクタを呼ぶことはできない**。

[コンストラクター - C# プログラミング ガイド](#)

※リファレンス内でたまにコンストラクタを呼び出す と表現しているから困る

コンストラクタは以下の構文のように定義する。

 構文4-2 コンストラクタの構文

public クラス名()
{
 処理
}

コンストラクタ内でメンバ変数の初期化を行うことが多い。
※そうしなければならぬルールはない。

📁デフォルトコンストラクタ

コンストラクタをひとつも定義しなかった場合はコンパイラが自動で引数なし、処理なしのコンストラクタを追加する。これを**デフォルトコンストラクタ**と呼ぶ。

※コード上ではなく、中間コードMSILに追加される

Step04 引数付きコンストラクタを作ろう


コンストラクタ定義の方法を学ぶ。

解説 引数付きコンストラクタ

コンストラクタには引数を定義することができる。
コンストラクタ内で仮引数を使ってメンバ変数を初期化する場合、仮引数名とメンバ変数名が同じであればthisをつかって区別すれば良い。

おまけ1 シグネチャの異なるコンストラクタを呼び出されるようにしたい

シグネチャの異なるコンストラクタを呼び出されるようにするには以下の構文を使う。

 構文4-3 シグネチャの異なるコンストラクタを呼び出されるようにする構文

public クラス名(シグネチャA) : this(シグネチャB)
{
 処理
}
※シグネチャBを持つコンストラクタが呼び出される

詳細は以下を参照。
[コンストラクターの使用 - C# プログラミング ガイド](#)

EXAMPLE

例4-1 構文4-3の例

class Player
{
 string name;

 public Player() : this("たかし")
 {
 Console.WriteLine(name + "2");
 }

 public Player(string name)
 {
 this.name = name;
 Console.WriteLine(name + "1");
 }
}

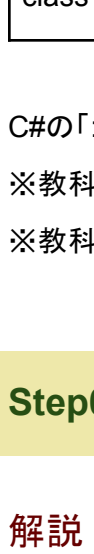
RESULTS

例4-1の実行結果 (new Player())としてインスタンス化した場合)

たかし1
たかし2

おまけ2 スーパークラスのコンストラクタを呼び出されるようにしたい

スーパークラスのコンストラクタを呼び出されるようにするには以下の構文を使う。

 構文4-4 スーパークラスのコンストラクタを呼び出されるようにする構文

public クラス名(シグネチャA) : base(シグネチャB)
{
 処理
}
※スーパークラスのシグネチャBを持つコンストラクタが呼び出される

詳細は以下を参照。
[コンストラクターの使用 - C# プログラミング ガイド](#)

実際の挙動は継承を学んでから試すと良い。

4-3 カプセル化～クラスの中身を隠す仕組み～

カプセル化はなぜ必要？

アクセス修飾子publicではインスタンスのメンバ変数・メソッド全てにアクセスできる。privateにすることでアクセスを限定することができる。この仕組みを**カプセル化**と呼ぶ。

カプセル化によって利用を限定でき、意図しない値が設定されることを防ぐことができる。

Step01 アクセス修飾子でメンバ変数に鍵をかけよう

アクセス修飾子の使い方を学ぶ。

解説 アクセス修飾子

アクセス修飾子にはpublic, protected, privateがあり、Javaの使い方と同様である。

Step02 メンバ変数にアクセスできる手段を作ろう

アクセサについて学ぶ。

解説 メンバメソッド経由でメンバ変数を書き換える

メンバ変数アクセスするにはgetter / setterを使う。getter / setterは**アクセサ**と呼ばれる。

Step03 プロパティで簡潔にまとめる

プロパティについて学ぶ。Javaには無かった機能。

解説 プロパティとは

プロパティはアクセサをより楽に作る仕組み。

プロパティの構文は次の通り。

 構文4-5 プロパティの構文

型 メンバ変数名;
public 型 プロパティ名
{
 set
 {
 メンバ変数名 = value;
 }
 get
 {
 return メンバ変数名;
 }
}
※set, getにはプロパティとは別のアクセス修飾子をつけることができる。
※例えばprivate setとすれば、他クラスからsetアクセサを利用できなくなる。
※詳しくは[アクセサのアクセシビリティの制限 - C# プログラミング ガイド](#)を参照。

※アクセサ(set, get部分)は必ずメンバ変数名を使わないといけないルールはない。リテラルでも可能。

4-4 継承～プログラムの重複を避ける仕組み～

類似したクラスを考える

(Javaと全く同じなので特記事項なし)

Step01 継承を使わずにSkyKartとTurboKartを作る

◆クラスの中身を入力する

(特記事項なし)

Step02 継承を使わずに作った場合の問題点

修正や仕様変更時に、同じような修正・変更を複数のクラスで行わなければならない。

Step03 継承を使ってKartを作り直そう

解説 継承を使ってクラスを作る

継承はクラス名の後に「:」を付け、その後にスーパークラス名を記述する。

表4-1 C#とJavaの継承

C#	Java
class SkyKart : Kart	public class SkyKart extends Kart

C#の「:」を使って継承する方法はクラスでもインターフェースでも同様である。

※教科書ではインターフェースは扱わない

※教科書では抽象クラス・抽象メソッドは扱わない(P167に少しだけ解説あり)

Step04 継承を使ったクラスに減速機能を追加してみよう

解説 基本クラスの修正・変更が派生クラスにも反映される

(特記事項なし)

4-5 ポリモーフィズム～派生クラスのインスタンスを基本クラスの変数に入れる～

ポリモーフィズム(多態性)、オーバーライドを学ぶ。

Step01 メソッドオーバーライドを使う

メソッドのオーバーライドを学ぶ。

解説 メソッドオーバーライド

基本クラスから継承したメソッドを派生クラスで定義しなおすることができる。

前提条件として、以下3点を全て満たす必要がある。

※ただし、抽象メソッドを除く

- ・オーバーライドされる事を許可するクラスにvirtualキーワードがあること
- ・オーバーライドされたメソッドにoverrideキーワードを付けること
- ・メソッド名、シグネチャが一致していること

Step02 ポリモーフィズムを使う

ポリモーフィズムを学ぶ。

解説 ポリモーフィズム

派生クラスのインスタンスは基本クラス型の変数に代入することができる(Javaのザックリ理論)。

その状態で派生クラスでオーバーライドされているメソッドを呼び出すと、基本クラスのメソッドではなく、派生クラスのメソッドが呼び出されることになる。これをポリモーフィズム(多態性)と呼ぶ。

※同じメソッドを呼び出してもインスタンスによって挙動が異なる。

[ポリモーフィズム](#)

📁抽象クラスと抽象メソッド

C#では中身を実装していない抽象メソッドを作ることができる。
抽象メソッドにはabstractキーワードを付ける必要があり、抽象メソッドを1つ以上持つクラスはabstractキーワードを付けた抽象クラスにしなければならない。
抽象クラスはインスタンス化できない。
抽象クラスを継承して抽象メソッドを実装する場合、overrideキーワードでオーバーライドする必要がある。オーバーライドする際にはabstractキーワードは付けない。
[抽象クラスとシールクラス、およびクラスメンバー - C# プログラミング ガイド](#)

Chapter4のまとめ

以下の点を学んだ。

- ・クラス
- ・インスタンス
- ・カプセル化
- ・継承
- ・ポリモーフィズム