C#入門: 初心者から実践エンジニアへの第一歩

はじめに

プログラミングの世界へようこそ！本書は、C#というプログラミング言語を通して、ソフトウェア開発の基礎から実践までを学ぶための入門書です。C#は、マイクロソフトが開発した強力でありながら学びやすい言語で、Webアプリケーション、デスクトップアプリ、モバイルアプリ、ゲーム開発など幅広い分野で活用されています。

本書の対象読者

この本は以下のような方々に向けて書かれています：

- プログラミングを初めて学ぶ方
- 他の言語を知っているが、C#を新たに学びたい方
- プログラミングの基礎を学んだが、実践的なスキルを身につけたい方

特に前提知識は必要ありませんが、コンピュータの基本的な操作ができることを想定しています。

本書の特徴

- **段階的な学習構造**: 基礎から応用へと、無理なく着実にスキルを積み上げられるよう構成しています
- **実践的なコード例**: 理論だけでなく、すぐに活用できる実用的なコード例を多数掲載
- **ベテランの知恵**: 現場のプロフェッショナルからのアドバイスを「ベテランの知恵袋」として紹介
- **若手エンジニアの疑問**: よくある疑問とその回答を「若手の疑問解決」コーナーで解説
- **視覚的な学習補助**: 複雑な概念を図表で視覚的に説明
- **プロジェクト事例**: 実際のプロジェクトに基づいた事例とその解説
- **チェックリスト**: 学習の定着を確認するためのチェックリストを各章に用意

それでは、C#の世界への扉を開きましょう！

目次

1. C#の世界への第一歩
2. C#の基本構文と型システム
3. 条件分岐と繰り返し処理
4. メソッドとクラスの基礎
5. オブジェクト指向プログラミングの実践
6. コレクションとLINQ
7. 例外処理とデバッグ
8. ファイルとデータの取り扱い
9. 実践プロジェクト: コンソールアプリケーション
10. 応用への道: GUIアプリケーション入門
11. 学習を続けるために

第1章: C#の世界への第一歩

C#とは何か

C#（シーシャープ）は、マイクロソフトによって開発されたモダンなプログラミング言語です。2000年に初めて発表されて以来、継続的に進化を続け、現在では世界中の多くの開発者に愛用されています。

C#の主な特徴は以下の通りです：

- **汎用性**: Webアプリケーション、デスクトップアプリ、モバイルアプリ、ゲーム開発など、様々な用途に使用できます
- **型安全性**: コンパイル時に型チェックが行われ、多くのエラーを事前に検出できます
- **オブジェクト指向**: クラスやインターフェースを使ったオブジェクト指向プログラミングをサポートしています
- **.NET Framework/.NET Core/.NET 5以降との連携**: マイクロソフトの強力なフレームワークと組み合わせて利用できます
- **豊富なライブラリ**: 標準ライブラリとNuGetパッケージを通じて、膨大な機能が利用可能です

> **ベテランの知恵袋**: C#を学ぶということは、単に文法を覚えるだけではありません。思考の枠組みを身につけることです。最初は難しく感じるかもしれませんが、基本的な概念を理解すれば、応用は自然と身についていきます。焦らず、一歩ずつ進みましょう。

開発環境のセットアップ

C#でプログラミングを始めるには、開発環境を整える必要があります。初心者に最もおすすめなのは、Visual Studioを使うことです。

Visual Studioのインストール

1. Visual Studioの公式サイトにアクセス
2. 「Community」エディション（個人開発者や学習用に無料）をダウンロード
3. インストーラーを実行し、「.NET デスクトップ開発」のワークロードを選択
4. インストールを実行（インターネット接続とディスク容量を確認してください）

初めてのプロジェクト作成

Visual Studioをインストールしたら、最初のプロジェクトを作成しましょう：

1. Visual Studioを起動
2. 「新しいプロジェクトの作成」をクリック
3. 「コンソールアプリ (.NET Core)」を選択
4. プロジェクト名を入力（例: "HelloWorld"）
5. 「作成」をクリック

これで最初のC#プロジェクトが作成されました。

> **若手の疑問解決**: 「.NET Framework」と「.NET Core」と「.NET 5以降」の違いは何ですか？
>
> .NET Frameworkは古いWindows専用のフレームワークで、.NET Coreはクロスプラットフォーム対応の新しいフレームワークです。.NET 5以降は、.NET Coreの後継で、今後の主要な開発プラットフォームとなります。新規プロジェクトでは、基本的に.NET 5以降を選択することをお勧めします。

最初のプログラム: Hello World!

プログラミング言語を学ぶ最初のステップとして、古くから「Hello World」と表示するプログラムを作成する伝統があります。C#でも、この伝統に従ってみましょう。

```
using System;

namespace HelloWorld
{
    class Program
    {
        static void Main(string[] args)
        {
            // 画面に文字列を表示する
            Console.WriteLine("Hello World! C#の世界へようこそ！");

            // キー入力を待つ（プログラムがすぐに終了しないようにするため）
            Console.ReadKey();
        }
    }
}
```

このコードを実行すると、コンソール画面に「Hello World! C#の世界へようこそ！」と表示されます。

コードの解説

- `using System;`: System名前空間を使用することを宣言しています。これにより、ConsoleクラスなどのSystemに含まれる機能を利用できます。
- `namespace HelloWorld`: プログラムの名前空間を定義しています。名前空間は、コードを整理するための仕組みです。
- `class Program`: プログラムのメインクラスを定義しています。
- `static void Main(string[] args)`: プログラムのエントリーポイント（開始点）となるメソッドです。
- `Console.WriteLine()`: コンソールに文字列を表示し、改行するメソッドです。
- `Console.ReadKey()`: キー入力を待機するメソッドです。

> **プロジェクト事例**: あるWebサービス開発プロジェクトでは、チーム全員が同じコーディング規約に従うために、最初にHello Worldプログラムを作成し、コメントの書き方やコードの整形ルールを確認するという習慣がありました。基本のスタイルを揃えることで、後々のコードレビューがスムーズになりました。

プログラムの実行方法

Visual Studioでプログラムを実行するには、以下の3つの方法があります：

1. **F5キーを押す**: デバッグモードでプログラムを実行します
2. **Ctrl+F5キーを押す**: デバッグなしでプログラムを実行します
3. **実行ボタン（緑の三角形）をクリック**: ツールバーの実行ボタンをクリックします

> **ベテランの知恵袋**: デバッグなしで実行（Ctrl+F5）すると、プログラムが終了しても窓が閉じないため、初心者の学習には便利です。デバッグ実行（F5）は、エラーを追跡したいときに使います。

C#プログラムの基本構造

C#プログラムは、主に以下の要素から構成されています：

1. **名前空間（namespace）**: コードを論理的に整理する仕組み
2. **クラス（class）**: オブジェクト指向プログラミングの基本単位
3. **メソッド（method）**: 処理を行う機能の集まり
4. **文（statement）**: 実行される個々の命令

この構造を視覚的に理解するために、以下の図を見てみましょう：

```
namespace MyApplication
{
    class Program            ← クラス
    {
        static void Main()   ← メソッド
        {
            Console.WriteLine("Hello");  ← 文
        }
    }
}
```

これらの概念は、今後の章で詳しく説明していきます。今はプログラムの基本的な骨組みとして理解しておいてください。

章末チェックリスト

- [ ] C#の主な特徴を3つ以上説明できる
- [ ] Visual Studioをインストールし、新しいプロジェクトを作成できる
- [ ] Hello Worldプログラムを作成し、実行できる
- [ ] C#プログラムの基本構造（名前空間、クラス、メソッド、文）を理解している
- [ ] F5とCtrl+F5の違いを説明できる

まとめと次のステップ

この章では、C#の基本的な概念と開発環境のセットアップ、最初のプログラムの作成方法を学びました。ここまでの内容を理解できれば、C#プログラミングへの第一歩を踏み出したことになります。

次の章では、C#の基本的な文法や型システムについて学び、より複雑なプログラムを書くための基礎を固めていきます。変数、データ型、演算子など、プログラミングの基本的な要素を理解していきましょう。

第2章: C#の基本構文と型システム

変数とデータ型

プログラミングにおいて、変数はデータを一時的に保存するための「箱」のようなものです。C#は「静的型付け言語」と呼ばれ、変数には特定のデータ型が割り当てられます。

基本的なデータ型

C#には多くのデータ型がありますが、最も基本的なものは以下の通りです：

データ型	説明	例
`int`	整数（-2,147,483,648 から 2,147,483,647）	`int age = 25;`
`double`	倍精度浮動小数点数	`double price = 29.99;`
`float`	単精度浮動小数点数	`float temperature = 36.5f;`
`decimal`	高精度の小数（金融計算などに使用）	`decimal amount = 1200.50m;`
`char`	単一の文字	`char grade = 'A';`
`string`	文字列	`string name = "田中太郎";`
`bool`	真偽値（trueまたはfalse）	`bool isActive = true;`

変数の宣言と初期化

C#で変数を使用するには、まず宣言する必要があります：

```
// 変数の宣言
int age;

// 変数の初期化
age = 25;

// 宣言と初期化を同時に行う
string name = "鈴木花子";
```

> **若手の疑問解決**: 「変数名はどのように決めればいいですか？」
>
> 変数名は、その変数が何を表しているのか明確にわかるものにしましょう。例えば、ユーザーの年齢を格納する変数は `age` や `userAge` などが適切です。一文字の変数名（`a`, `b` など）は避け、ローカル変数は先頭小文字のキャメルケース（`userName`）、定数は大文字のスネークケース（`MAX_SIZE`）とするのが一般的です。

型推論とvar

C#では、初期値から型を自動的に推論する機能もあります：

```
// varを使用した型推論
var count = 10;         // intと推論される
var message = "こんにちは";  // stringと推論される
var isValid = true;     // boolと推論される
```

`var` を使用すると、コンパイラが右辺の値から適切な型を判断します。これにより、コードをより簡潔に書けますが、可読性を考慮して使用する必要があります。

> **ベテランの知恵袋**: `var` は便利ですが、乱用は避けましょう。特に初心者のうちは、明示的に型を書くことで、どのような型の変数を扱っているのか意識することが大切です。複雑な型名（ジェネリックコレクションなど）を扱う場合や、右辺から型が明白な場合に `var` を使うとよいでしょう。

定数

値が変更されないことを保証したい場合は、変数ではなく定数を使用します：

```
// 定数の宣言
const double PI = 3.14159;
const string APP_NAME = "My First C# App";

// 定数は後から値を変更できない
// PI = 3.14; // コンパイルエラー
```

定数は `const` キーワードを使用して宣言し、宣言時に必ず初期化する必要があります。

演算子

C#には様々な演算子があり、計算や比較などの操作を行うことができます。

算術演算子

```
int a = 10;
int b = 3;

int sum = a + b;        // 13（加算）
int difference = a - b; // 7（減算）
int product = a * b;    // 30（乗算）
int quotient = a / b;   // 3（整数除算：小数部は切り捨て）
int remainder = a % b;  // 1（剰余：割り算の余り）

// インクリメント/デクリメント
int c = 5;
c++;  // cの値が1増えて6になる
c--;  // cの値が1減って5に戻る
```

比較演算子

```
int x = 5;
int y = 10;

bool isEqual = (x == y);        // false（等しい）
bool isNotEqual = (x != y);     // true（等しくない）
bool isGreater = (x > y);       // false（より大きい）
bool isLess = (x < y);          // true（より小さい）
bool isGreaterOrEqual = (x >= y); // false（以上）
bool isLessOrEqual = (x <= y);    // true（以下）
```

論理演算子

```
bool condition1 = true;
bool condition2 = false;

bool andResult = condition1 && condition2;  // false（AND：両方trueならtrue）
```

```
bool orResult = condition1 || condition2;   // true (OR: どちらかがtrueならtrue)
bool notResult = !condition1;               // false (NOT: 真偽を反転)
```

> **失敗から学ぶ**: あるプロジェクトで、整数除算の結果を小数で得たかったにもかかわらず、`int / int` のように記述してしまい、小数部が常に切り捨てられるというバグが発生しました。例えば、`5 / 2` の結果が期待していた `2.5` ではなく `2` になっていました。このバグを修正するには、少なくとも一方の値を浮動小数点数に変換する必要がありました：`5.0 / 2` や `(double)5 / 2` のように。

文字列の操作

C#では、文字列を様々な方法で操作することができます。

文字列の連結

```
string firstName = "太郎";
string lastName = "山田";

// +演算子による連結
string fullName = lastName + " " + firstName;  // "山田 太郎"

// string.Formatによる連結
string greeting = string.Format("こんにちは、{0}さん。あなたは{1}歳ですね。", fullName, 30);

// 文字列補間（C# 6.0以降）
int age = 30;
string message = $"こんにちは、{fullName}さん。あなたは{age}歳ですね。";
```

文字列の主なメソッド

```
string text = "C#プログラミングを学ぼう";

// 長さの取得
int length = text.Length;  // 14

// 部分文字列の取得
string substring = text.Substring(0, 2);  // "C#"

// 文字列の置換
string replaced = text.Replace("C#", "Java");  // "Javaプログラミングを学ぼう"

// 大文字/小文字変換
string upper = "hello".ToUpper();  // "HELLO"
string lower = "WORLD".ToLower();  // "world"

// 文字列の検索
bool contains = text.Contains("プログラミング");  // true
int index = text.IndexOf("プログラミング");       // 2
bool startsWith = text.StartsWith("C#");        // true
bool endsWith = text.EndsWith("学ぼう");         // true

// 空白の削除
string trimmed = "  Hello  ".Trim();  // "Hello"
```

> **プロジェクト事例**: あるログ解析システムでは、大量のログファイルから特定のパターンを検索する必要がありました。文字列操作メソッドを効果的に組み合わせることで、数百万行のログから必要な情報を抽出し、エラーの傾向分析が可能になりました。

型変換

C#では、異なるデータ型間で値を変換する方法がいくつかあります。

暗黙的な型変換（自動的に行われる安全な変換）

```
int intValue = 100;
long longValue = intValue;  // intからlongへの暗黙的な変換

float floatValue = 10.5f;
double doubleValue = floatValue;  // floatからdoubleへの暗黙的な変換
```

明示的な型変換（キャスト）

```
double doubleValue = 10.5;
int intValue = (int)doubleValue;  // doubleからintへの明示的な変換（小数部は切り捨て）

// 注意：大きい値を小さい型に変換すると、データが失われる可能性があります
long longValue = 10000000000;
int anotherInt = (int)longValue;  // オーバーフローが発生する可能性あり
```

Convert クラスの使用

```
string numberAsString = "123";
int number = Convert.ToInt32(numberAsString);  // 文字列から整数への変換

bool boolValue = true;
string boolAsString = Convert.ToString(boolValue);  // 真偽値から文字列への変換
```

Parse メソッドの使用

```
string intString = "456";
int parsedInt = int.Parse(intString);  // 文字列から整数への変換

string doubleString = "123.45";
double parsedDouble = double.Parse(doubleString);  // 文字列から倍精度浮動小数点数への変換
```

> **若手の疑問解決**: 「Parse メソッドと Convert クラスの違いは何ですか？」
>
> Parse メソッドは文字列からのみ変換できますが、null を渡すと例外が発生します。一方、Convert クラスはより広範な型変換をサポートし、null を扱うことができます（null の場合は型のデフォルト値を返します）。状況に応じて使い分けると良いでしょう。

C#の型システムの特徴

C#の型システムには、いくつかの重要な特徴があります：

値型とリファレンス型

C#のデータ型は、大きく分けて「値型」と「リファレンス型」の2種類に分類されます：

1. **値型**: 変数に直接値が格納される型（int, double, boolなど）
2. **リファレンス型**: 変数には実際のデータへの参照（メモリ上のアドレス）が格納される型（string, arrayなど）

この違いは、変数の代入時や関数に渡す際の挙動に影響します。

```
// 値型の例
int a = 10;
int b = a;  // aの値のコピーがbに格納される
a = 20;     // aを変更しても、bは影響を受けない（b = 10のまま）

// リファレンス型の例
int[] arrayA = { 1, 2, 3 };
int[] arrayB = arrayA;  // arrayAへの参照がarrayBに格納される
arrayA[0] = 99;         // arrayAを変更すると、arrayBも影響を受ける（両方とも[99, 2, 3]になる）
```

Nullable型

C#では通常、値型にはnull値を割り当てることができませんが、Nullable型を使用することで可能になります：

```
// 通常の値型はnullを割り当てられない
// int normalInt = null;  // コンパイルエラー

// Nullable型ではnullを割り当て可能
int? nullableInt = null;
double? nullableDouble = null;

// 値の確認
if (nullableInt.HasValue)
{
    Console.WriteLine($"値があります: {nullableInt.Value}");
}
else
{
    Console.WriteLine("値はnullです");
}

// Nullになりうる値に対するデフォルト値の提供
int result = nullableInt ?? 0;  // nullableIntがnullならば0、そうでなければnullableIntの値
```

章末チェックリスト

- [ ] 基本的なデータ型（int, double, string, boolなど）を使って変数を宣言できる
- [ ] 算術演算子、比較演算子、論理演算子を使った式を書ける
- [ ] 文字列の連結や操作（Substring, Replace, IndexOfなど）ができる
- [ ] 型変換の方法（暗黙的変換、キャスト、Convertクラス、Parseメソッド）を理解している
- [ ] 値型とリファレンス型の違いを説明できる

- [ ] Nullable型の使い方を理解している

まとめと次のステップ

この章では、C#のデータ型、変数、演算子、型変換など、プログラミングの基本的な要素について学びました。これらは、あらゆるプログラムの基礎となる重要な概念です。

次の章では、条件分岐と繰り返し処理について学びます。これにより、プログラムが状況に応じて異なる処理を行ったり、同じ処理を繰り返し実行したりすることが可能になります。

第3章: 条件分岐と繰り返し処理

プログラミングでは、条件に応じて異なる処理を行ったり、同じ処理を繰り返し実行したりする必要があります。C#には、そのための構文が用意されています。

if文による条件分岐

if文は、指定した条件が真（true）の場合にのみ、特定のコードブロックを実行します。

```
int age = 18;

// 基本的なif文
if (age >= 20)
{
    Console.WriteLine("成人です");
}

// if-else文
if (age >= 20)
{
    Console.WriteLine("成人です");
}
else
{
    Console.WriteLine("未成年です");
}

// if-else if-else文
if (age >= 20)
{
    Console.WriteLine("成人です");
}
else if (age >= 18)
{
    Console.WriteLine("18歳以上の未成年です");
}
else
{
    Console.WriteLine("18歳未満です");
}
```

条件の組み合わせ

複数の条件を組み合わせることもできます：

```
int age = 25;
bool hasLicense = true;

// AND条件：両方の条件がtrueの場合
if (age >= 18 && hasLicense)
{
    Console.WriteLine("車を運転できます");
}

// OR条件：どちらかの条件がtrueの場合
if (age < 6 || age >= 65)
{
    Console.WriteLine("入場料は無料です");
}

// 複雑な条件の組み合わせ
bool isStudent = true;
if ((age >= 18 && hasLicense) || (isStudent && age >= 16))
{
    Console.WriteLine("特別プログラムに参加できます");
}
```

入れ子のif文

if文の中に別のif文を入れることもできます：

```
int score = 85;

if (score >= 60)
{
    Console.WriteLine("合格です");

    if (score >= 80)
    {
        Console.WriteLine("すばらしい成績です");
    }
    else
    {
        Console.WriteLine("もう少し頑張りましょう");
    }
}
else
{
    Console.WriteLine("不合格です");
}
```

> **ベテランの知恵袋**: 入れ子のif文が多くなりすぎると、コードの可読性が低下します。これを「アロー型コード」と呼ぶこともあります。条件を組み合わせたり、早期リターンパターンを使うことで、入れ子を減らし、コードをシンプルに保ちましょう。

switch文による条件分岐

複数の選択肢から条件に合うものを選ぶ場合、`if-else if`の連続よりも`switch`文の方が見やすくなることがあります。

```
int day = 3;
string dayName;

// switch文の基本形
switch (day)
{
    case 1:
        dayName = "月曜日";
        break;
    case 2:
        dayName = "火曜日";
        break;
    case 3:
        dayName = "水曜日";
        break;
    case 4:
        dayName = "木曜日";
        break;
    case 5:
        dayName = "金曜日";
        break;
    case 6:
        dayName = "土曜日";
        break;
    case 7:
        dayName = "日曜日";
        break;
    default:
        dayName = "不明な日";
        break;
}

Console.WriteLine($"今日は{dayName}です");
```

各`case`の後には、その条件が一致した場合に実行するコードを記述します。`break`文は、そのケースの処理が終わったらswitch文を抜けることを意味します。`default`は、どのケースにも一致しなかった場合の処理です。

C# 7.0以降の拡張されたswitch文

C# 7.0以降では、switch文がより柔軟になりました：

```
object item = 123;
string typeDescription;

switch (item)
{
    case int i when i > 100:
        typeDescription = "100より大きい整数";
        break;
    case int i:
        typeDescription = "整数";
```

```
            break;
        case string s when s.Length > 10:
            typeDescription = "長い文字列";
            break;
        case string s:
            typeDescription = "文字列";
            break;
        case null:
            typeDescription = "null値";
            break;
        default:
            typeDescription = "その他の型";
            break;
}

Console.WriteLine($"itemは{typeDescription}です");
```

この例では、型のチェックと条件式（`when`節）を組み合わせています。

> **若手の疑問解決**: 「`if`文と`switch`文、どちらを使うべきですか？」
>
> 基本的には、選択肢が多い場合や値に基づいて分岐する場合は`switch`文、複雑な条件や範囲での比較には`if`文が適しています。コードの読みやすさを最優先に考えて選びましょう。

条件演算子（三項演算子）

単純な条件分岐なら、条件演算子（三項演算子とも呼ばれる）を使って、より簡潔に書くことができます：

```
// if-else文
int age = 20;
string status;

if (age >= 20)
{
    status = "成人";
}
else
{
    status = "未成年";
}

// 条件演算子
status = age >= 20 ? "成人" : "未成年";
```

条件演算子の構文は`条件 ? 真の場合の値 : 偽の場合の値`です。

> **ベテランの知恵袋**: 条件演算子は簡潔で便利ですが、ネストさせたり複雑な式を書くと可読性が低下します。簡単な条件分岐にのみ使用し、複雑な場合は通常の`if`文を使いましょう。

while文による繰り返し処理

`while`文は、指定した条件が真（true）である間、コードブロックを繰り返し実行します：

```
// 基本的なwhile文
int count = 1;
while (count <= 5)
{
    Console.WriteLine($"カウント: {count}");
    count++;  // カウントを増やす
}
// 出力:
// カウント: 1
// カウント: 2
// カウント: 3
// カウント: 4
// カウント: 5

// 条件が最初から偽の場合、一度も実行されない
int x = 10;
while (x < 5)
{
    Console.WriteLine("この文は実行されません");
    x++;
}
```

do-while文

`do-while`文は、最低1回はコードブロックを実行し、その後で条件をチェックします：

```
// 基本的なdo-while文
int count = 1;
do
{
    Console.WriteLine($"カウント: {count}");
    count++;
} while (count <= 5);

// 条件が最初から偽でも、一度は実行される
int x = 10;
do
{
    Console.WriteLine("この文は一度だけ実行されます");
    x++;
} while (x < 5);
```

> **プロジェクト事例**: あるユーザー登録システムでは、ユーザーが有効なパスワードを入力するまで繰り返し入力を求める必要がありました。do-while文を使うことで、最低1回はパスワード入力画面を表示し、入力されたパスワードが条件を満たさない場合のみ再度入力を求めるという自然な流れを実現できました。

for文による繰り返し処理

`for`文は、カウンター変数を使った繰り返しに適しています：

```
// 基本的なfor文
for (int i = 1; i <= 5; i++)
```

```
{
    Console.WriteLine($"カウント: {i}");
}
// 出力:
// カウント: 1
// カウント: 2
// カウント: 3
// カウント: 4
// カウント: 5

// 逆順のカウント
for (int i = 5; i >= 1; i--)
{
    Console.WriteLine($"逆順カウント: {i}");
}
// 出力:
// 逆順カウント: 5
// 逆順カウント: 4
// 逆順カウント: 3
// 逆順カウント: 2
// 逆順カウント: 1

// ステップ数の指定
for (int i = 0; i <= 10; i += 2)
{
    Console.WriteLine($"偶数: {i}");
}
// 出力:
// 偶数: 0
// 偶数: 2
// 偶数: 4
// 偶数: 6
// 偶数: 8
// 偶数: 10
```

`for` 文の構文は `for (初期化; 条件; 増分)` です。初期化は最初に1回だけ実行され、条件は各反復の前にチェックされ、増分は各反復の後に実行されます。

foreach文によるコレクション処理

`foreach` 文は、配列やコレクションの各要素に対して処理を行うのに最適です：

```
// 文字列配列の各要素を処理
string[] fruits = { "りんご", "バナナ", "オレンジ", "ぶどう" };

foreach (string fruit in fruits)
{
    Console.WriteLine($"フルーツ: {fruit}");
}
// 出力:
// フルーツ: りんご
// フルーツ: バナナ
// フルーツ: オレンジ
// フルーツ: ぶどう
```

```
// 整数配列の各要素を2倍にする
int[] numbers = { 1, 2, 3, 4, 5 };
int[] doubled = new int[numbers.Length];
int index = 0;

foreach (int number in numbers)
{
    doubled[index] = number * 2;
    index++;
}

// 結果の表示
foreach (int number in doubled)
{
    Console.WriteLine($"2倍の値: {number}");
}
// 出力:
// 2倍の値: 2
// 2倍の値: 4
// 2倍の値: 6
// 2倍の値: 8
// 2倍の値: 10
```

> **若手の疑問解決**: 「`for`と`foreach`、どちらを使うべきですか？」
>
> `foreach`は配列やコレクションのすべての要素を順番に処理する場合に最適です。インデックスが不要で、単純に全要素を処理するだけなら`foreach`の方が簡潔で読みやすいコードになります。一方、インデックスを使って要素にアクセスしたり、特定の増分パターンで反復したり、複数のコレクションを同時に処理したりする場合は`for`文が適しています。

ループ制御ステートメント

ループの実行を制御するために、`break`と`continue`という2つの重要なステートメントがあります：

break文

`break`文は、ループを即座に終了し、ループの次の文から実行を続けます：

```
// 特定の条件でループを抜ける
for (int i = 1; i <= 10; i++)
{
    if (i == 5)
    {
        Console.WriteLine("5で終了します");
        break;  // ループを抜ける
    }
    Console.WriteLine($"数値: {i}");
}
// 出力:
// 数値: 1
// 数値: 2
// 数値: 3
// 数値: 4
// 5で終了します
```

```
// 無限ループからの脱出
int count = 1;
while (true)  // 無限ループ
{
    Console.WriteLine($"カウント: {count}");

    if (count >= 5)
    {
        break;  // 条件を満たしたらループを抜ける
    }

    count++;
}
```

continue文

`continue` 文は、現在の反復をスキップし、ループの次の反復に進みます：

```
// 偶数のみを処理する
for (int i = 1; i <= 10; i++)
{
    if (i % 2 != 0)  // 奇数の場合
    {
        continue;  // 次の反復に進む
    }

    Console.WriteLine($"偶数: {i}");
}
// 出力:
// 偶数: 2
// 偶数: 4
// 偶数: 6
// 偶数: 8
// 偶数: 10
```

> **失敗から学ぶ**: あるプロジェクトでは、ユーザーデータを処理するループ内で例外が発生した場合、そのユーザーのデータをスキップして次のユーザーの処理を続けるようにしていました。しかし、`continue` ではなく `break` を使ってしまったため、最初のエラーでループ全体が終了し、残りのユーザーが処理されないという問題が発生しました。ループ制御ステートメントの意味を正確に理解することの重要性を学びました。

入れ子のループ

ループの中に別のループを入れることも可能です。これを「入れ子のループ」と呼びます：

```
// 九九の表を作成
for (int i = 1; i <= 9; i++)
{
    for (int j = 1; j <= 9; j++)
    {
        Console.Write($"{i * j,3}");  // 3桁の幅で整形
    }
    Console.WriteLine();  // 行の終わりで改行
}
```

```
// 出力:
//   1  2  3  4  5  6  7  8  9
//   2  4  6  8 10 12 14 16 18
//   3  6  9 12 15 18 21 24 27
//   ...（以下省略）
```

この例では、外側のループが行を、内側のループが列を制御しています。

> **ベテランの知恵袋**: 入れ子のループは、行列や表などの2次元データを処理する際に便利ですが、処理速度に注意が必要です。たとえば、10,000回のループを持つ2つの入れ子になったループは、合計で1億回（10,000 × 10,000）の処理を行うことになります。パフォーマンスが重要な場合は、必要最小限の反復回数にとどめましょう。

パターンマッチング（C# 7.0以降）

C# 7.0以降では、`is`演算子と`switch`文でパターンマッチングが可能になりました：

```
// isパターンマッチング
object obj = "Hello";

if (obj is string s)
{
    // sは文字列として使用可能
    Console.WriteLine($"文字列の長さ: {s.Length}");
}

// switchパターンマッチング
object item = 123;

switch (item)
{
    case int i:
        Console.WriteLine($"整数値: {i}");
        break;
    case string s:
        Console.WriteLine($"文字列値: {s}");
        break;
    case bool b:
        Console.WriteLine($"真偽値: {b}");
        break;
    default:
        Console.WriteLine("その他の型");
        break;
}
```

> **若手の疑問解決**: 「パターンマッチングは具体的にどんな場面で役立ちますか？」
>
> パターンマッチングは、様々な型のオブジェクトを扱うときや、条件に基づいて異なる処理を行う必要がある場合に特に役立ちます。例えば、ユーザー入力の検証、異なる形式のデータ処理、ポリモーフィズムの代替としての使用などが挙げられます。従来の型チェックやキャストよりも簡潔で読みやすいコードになります。

章末チェックリスト

- [ ] if文、else if文、else文を使って条件分岐を実装できる
- [ ] switch文を使って複数の条件に基づく分岐を実装できる
- [ ] 条件演算子（三項演算子）を使って簡潔な条件式を書ける
- [ ] while文とdo-while文の違いを理解し、適切に使い分けられる
- [ ] for文を使ってカウンターベースのループを実装できる
- [ ] foreach文を使ってコレクションの各要素を処理できる
- [ ] break文とcontinue文を使ってループの制御ができる
- [ ] 入れ子のループを使って2次元データを処理できる
- [ ] パターンマッチングの基本的な使い方を理解している

まとめと次のステップ

この章では、C#における条件分岐と繰り返し処理の基本について学びました。これらの制御構造を理解し使いこなすことで、様々な状況に対応できるプログラムを書くことができるようになります。

次の章では、メソッドとクラスの基礎について学びます。メソッドを使うことでコードを再利用可能な部品に分割し、クラスを使うことでデータと操作をひとつのユニットにまとめることができるようになります。これらは、より大規模で構造化されたプログラムを書くための基盤となる概念です。

第4章: メソッドとクラスの基礎

プログラムが大きくなるにつれて、コードを整理し再利用できるようにする必要があります。C#では、メソッドとクラスがその役割を果たします。

メソッドの基礎

メソッドは、特定のタスクを実行するコードのブロックです。メソッドを使うと、同じコードを何度も書く代わりに、一度定義したコードを必要なときに呼び出すことができます。

メソッドの定義と呼び出し

```
// メソッドの定義
static void Greet()
{
    Console.WriteLine("こんにちは！");
}

// メソッドの呼び出し
Greet();  // 出力: こんにちは！
Greet();  // 出力: こんにちは！
```

パラメータと戻り値

メソッドは、パラメータ（引数）を受け取り、値を返すことができます：

```
// パラメータを持つメソッド
static void GreetPerson(string name)
{
    Console.WriteLine($"こんにちは、{name}さん！");
```

```
}

// 戻り値を持つメソッド
static int Add(int a, int b)
{
    return a + b;
}

// パラメータと戻り値を持つメソッドの呼び出し
GreetPerson("田中");  // 出力: こんにちは、田中さん！
int sum = Add(5, 3);  // sum = 8
Console.WriteLine($"合計: {sum}");
```

> **若手の疑問解決**: 「メソッド名はどのように付ければよいですか？」
>
> メソッド名は、そのメソッドが何をするのかを明確に表すべきです。C#の慣習では、メソッド名はパスカルケース（最初の文字と各単語の最初の文字を大文字にする）で書きます。動詞または動詞+名詞の形式（例: `CalculateTotal`, `SaveData`, `GetUserInfo`）にすると、メソッドの目的が明確になります。

オプションパラメータとデフォルト値

C#では、パラメータにデフォルト値を設定することができます：

```
// オプションパラメータを持つメソッド
static void DisplayInfo(string name, int age = 30, string country = "日本")
{
    Console.WriteLine($"名前: {name}, 年齢: {age}, 国: {country}");
}

// 様々な呼び出し方法
DisplayInfo("鈴木");                    // 出力: 名前: 鈴木, 年齢: 30, 国: 日本
DisplayInfo("佐藤", 25);                // 出力: 名前: 佐藤, 年齢: 25, 国: 日本
DisplayInfo("田中", 40, "アメリカ");     // 出力: 名前: 田中, 年齢: 40, 国: アメリカ
```

名前付き引数

引数の順序を変えて指定したい場合は、名前付き引数を使用できます：

```
// 名前付き引数の使用
DisplayInfo(name: "山田", country: "カナダ", age: 35);  // 出力: 名前: 山田, 年齢: 35, 国: カナダ
DisplayInfo(country: "イギリス", name: "伊藤");          // 出力: 名前: 伊藤, 年齢: 30, 国: イギリス
```

> **プロジェクト事例**: あるWebアプリケーションプロジェクトでは、ユーザー設定を読み込むメソッドがありました。このメソッドには20以上のパラメータがあり、その多くはオプションでした。名前付き引数を使用することで、必要なパラメータだけを明示的に指定でき、コードの可読性が大幅に向上しました。

メソッドのオーバーロード

同じ名前で異なるパラメータを持つメソッドを複数定義することができます：

```
// メソッドのオーバーロード
static int Add(int a, int b)
{
    return a + b;
}

static double Add(double a, double b)
{
    return a + b;
}

static string Add(string a, string b)
{
    return a + b;
}

// オーバーロードされたメソッドの呼び出し
int sum1 = Add(5, 3);                    // int版のAddが呼び出される
double sum2 = Add(2.5, 3.7);             // double版のAddが呼び出される
string combined = Add("Hello, ", "World!");  // string版のAddが呼び出される
```

コンパイラは、引数の型に基づいて適切なメソッドを選択します。

値渡しと参照渡し

C#では、デフォルトでは値型は「値渡し」、参照型は「参照渡し」で渡されます：

```
// 値渡しの例
static void DoubleValue(int x)
{
    x = x * 2;  // この変更は呼び出し元には影響しない
}

int number = 5;
DoubleValue(number);
Console.WriteLine(number);  // 出力: 5 (変更されていない)

// 参照渡しの例 (配列は参照型)
static void DoubleArray(int[] arr)
{
    for (int i = 0; i < arr.Length; i++)
    {
        arr[i] = arr[i] * 2;  // この変更は呼び出し元にも影響する
    }
}

int[] numbers = { 1, 2, 3 };
DoubleArray(numbers);
Console.WriteLine(string.Join(", ", numbers));  // 出力: 2, 4, 6 (変更されている)
```

明示的に参照渡しや出力パラメータを使用することもできます：

```
// refキーワードによる参照渡し
static void DoubleValueByRef(ref int x)
```

```
{
    x = x * 2;  // この変更は呼び出し元にも影響する
}

int number = 5;
DoubleValueByRef(ref number);
Console.WriteLine(number);  // 出力: 10 (変更されている)

// outキーワードによる出力パラメータ
static void GetMinMax(int[] numbers, out int min, out int max)
{
    min = numbers[0];
    max = numbers[0];

    for (int i = 1; i < numbers.Length; i++)
    {
        if (numbers[i] < min) min = numbers[i];
        if (numbers[i] > max) max = numbers[i];
    }
}

int[] values = { 3, 7, 2, 15, 9 };
GetMinMax(values, out int minimum, out int maximum);
Console.WriteLine($"最小値: {minimum}, 最大値: {maximum}");  // 出力: 最小値: 2, 最大値: 15
```

> **ベテランの知恵袋**: `ref` と `out` パラメータは強力ですが、コードを理解しにくくする可能性があります。基本的には複数の値を返す必要がある場合はタプルやカスタムクラスを返すことを検討し、`ref` や `out` は必要な場合にのみ使用することをお勧めします。特に `ref` パラメータは、その変数が変更される可能性があることを明示的に示すため、必要なケースに限定して使いましょう。

クラスの基礎

クラスは、データ（フィールド）と機能（メソッド）をひとつのユニットにまとめたものです。オブジェクト指向プログラミングの基本的な構成要素です。

クラスの定義と使用

```
// クラスの定義
class Person
{
    // フィールド（データ）
    public string Name;
    public int Age;

    // メソッド（機能）
    public void Introduce()
    {
        Console.WriteLine($"こんにちは、私は{Name}です。{Age}歳です。");
    }
}

// クラスの使用
Person person1 = new Person();  // Personクラスのインスタンス（オブジェクト）を作成
person1.Name = "田中太郎";      // フィールドに値を設定
person1.Age = 30;
```

```
person1.Introduce();  // メソッドを呼び出す
// 出力: こんにちは、私は田中太郎です。30歳です。

// 別のインスタンスを作成
Person person2 = new Person();
person2.Name = "鈴木花子";
person2.Age = 25;

person2.Introduce();
// 出力: こんにちは、私は鈴木花子です。25歳です。
```

コンストラクタ

コンストラクタは、オブジェクトが作成されるときに自動的に呼び出される特別なメソッドです：

```
class Person
{
    public string Name;
    public int Age;

    // コンストラクタ
    public Person(string name, int age)
    {
        Name = name;
        Age = age;
    }

    public void Introduce()
    {
        Console.WriteLine($"こんにちは、私は{Name}です。{Age}歳です。");
    }
}

// コンストラクタを使ってオブジェクトを初期化
Person person1 = new Person("山田一郎", 35);
person1.Introduce();  // 出力: こんにちは、私は山田一郎です。35歳です。
```

複数のコンストラクタ（オーバーロード）

クラスは複数のコンストラクタを持つことができます：

```
class Person
{
    public string Name;
    public int Age;

    // デフォルトコンストラクタ
    public Person()
    {
        Name = "名無し";
        Age = 20;
    }
```

```
    // パラメータ付きコンストラクタ
    public Person(string name, int age)
    {
        Name = name;
        Age = age;
    }

    // 名前のみのコンストラクタ
    public Person(string name)
    {
        Name = name;
        Age = 20;  // デフォルト値
    }

    public void Introduce()
    {
        Console.WriteLine($"こんにちは、私は{Name}です。{Age}歳です。");
    }
}

// 様々なコンストラクタの使用
Person person1 = new Person();              // デフォルトコンストラクタ
Person person2 = new Person("佐藤二郎");     // 名前のみのコンストラクタ
Person person3 = new Person("伊藤三郎", 40); // パラメータ付きコンストラクタ

person1.Introduce();  // 出力: こんにちは、私は名無しです。20歳です。
person2.Introduce();  // 出力: こんにちは、私は佐藤二郎です。20歳です。
person3.Introduce();  // 出力: こんにちは、私は伊藤三郎です。40歳です。
```

プロパティ

プロパティは、フィールドへのアクセスを制御するための機能です：

```
class Person
{
    // プライベートフィールド
    private string _name;
    private int _age;

    // プロパティ
    public string Name
    {
        get { return _name; }
        set { _name = value; }
    }

    // 検証ロジックを含むプロパティ
    public int Age
    {
        get { return _age; }
        set
        {
            if (value < 0)
            {
                throw new ArgumentException("年齢は0以上である必要があります");
            }
```

```
            _age = value;
        }
    }

    // 自動実装プロパティ（C# 3.0以降）
    public string Country { get; set; } = "日本";

    // 読み取り専用プロパティ
    public bool IsAdult
    {
        get { return _age >= 20; }
    }

    // コンストラクタ
    public Person(string name, int age)
    {
        _name = name;
        if (age < 0)
        {
            throw new ArgumentException("年齢は0以上である必要があります");
        }
        _age = age;
    }

    public void Introduce()
    {
        Console.WriteLine($"こんにちは、私は{Name}です。{Age}歳です。");
        Console.WriteLine($"国籍: {Country}");
        Console.WriteLine($"成人: {(IsAdult ? "はい" : "いいえ")}");
    }
}

// プロパティの使用
Person person = new Person("高橋四郎", 22);
Console.WriteLine(person.Name);  // プロパティから値を取得
person.Name = "高橋四郎Jr.";     // プロパティに値を設定
person.Country = "カナダ";        // 自動実装プロパティに値を設定

person.Introduce();
// 出力:
// こんにちは、私は高橋四郎Jr.です。22歳です。
// 国籍: カナダ
// 成人: はい

// エラーケース
// Person invalidPerson = new Person("テスト", -5);  // 例外がスローされる
```

> **若手の疑問解決**: 「フィールドを直接公開せずにプロパティを使う利点は何ですか？」
>
> プロパティを使用する主な利点は以下の通りです：
>
> 1. **カプセル化**: 内部データへのアクセスを制御できます
> 2. **検証**: 設定される値が有効かどうかをチェックできます
> 3. **計算**: 実際のフィールド値から計算された値を返せます
> 4. **互換性**: 後からアクセス方法を変更しても、外部のコードに影響を与えません

> 5. **デバッグ**: 値の変更時にブレークポイントを設定しやすくなります

静的メンバー

静的（static）メンバーは、クラスのインスタンスではなくクラス自体に属します：

```
class MathHelper
{
    // 静的フィールド
    public static double PI = 3.14159;

    // 静的メソッド
    public static double CalculateCircleArea(double radius)
    {
        return PI * radius * radius;
    }
}

// 静的メンバーの使用
double area = MathHelper.CalculateCircleArea(5);
Console.WriteLine($"円の面積: {area}");
Console.WriteLine($"円周率: {MathHelper.PI}");
```

静的メンバーは、インスタンスを作成せずに直接クラス名を通じてアクセスします。

> **ベテランの知恵袋**: 静的メンバーはグローバル変数や関数に似ていますが、名前空間によって整理されています。ユーティリティ関数や定数など、インスタンス固有のデータに依存しない機能には適していますが、多用するとテストしにくいコードになる可能性があるので注意が必要です。

静的クラス

クラス自体を静的にすることもできます。静的クラスはインスタンスを作成できず、すべてのメンバーも静的である必要があります：

```
// 静的クラス
static class Logger
{
    // 静的メソッド
    public static void LogInfo(string message)
    {
        Console.WriteLine($"情報: {message}");
    }

    public static void LogError(string message)
    {
        Console.WriteLine($"エラー: {message}");
    }
}

// 静的クラスの使用
Logger.LogInfo("アプリケーションが起動しました");
Logger.LogError("ファイルが見つかりません");
```

静的クラスは通常、ユーティリティ機能やヘルパー機能のグループ化に使用されます。

部分クラス（partial class）

大きなクラスを複数のファイルに分割することができます：

```
// ファイル1: Person.cs
partial class Person
{
    private string _name;

    public string Name
    {
        get { return _name; }
        set { _name = value; }
    }
}

// ファイル2: Person.Methods.cs
partial class Person
{
    public void Introduce()
    {
        Console.WriteLine($"こんにちは、私は{Name}です。");
    }
}
```

部分クラスは、コードジェネレーターを使用する場合や、大規模なクラスを整理する場合に便利です。

> **プロジェクト事例**: あるWPFアプリケーション開発では、デザイナーが作成したUI部分（XAML）と開発者が実装したロジック部分（C#）を分離するために部分クラスが活用されました。これにより、UIデザインとビジネスロジックの担当者が同じファイルを編集することなく並行して作業ができ、プロジェクトの効率が向上しました。

オブジェクト指向プログラミングの基本概念

クラスを扱う際に理解しておきたい基本的な概念をいくつか紹介します。

カプセル化

カプセル化は、データとその操作を一つのユニットにまとめ、不必要な詳細を隠すことです：

```
class BankAccount
{
    // プライベートフィールド - 直接アクセスできない
    private decimal _balance;
    private string _accountNumber;

    // パブリックプロパティ - 制御されたアクセスを提供
    public string AccountHolder { get; set; }

    public decimal Balance
    {
        get { return _balance; }
        private set { _balance = value; }  // privateセッター
    }
```

```
    // コンストラクタ
    public BankAccount(string accountHolder, string accountNumber)
    {
        AccountHolder = accountHolder;
        _accountNumber = accountNumber;
        _balance = 0;
    }

    // パブリックメソッド
    public void Deposit(decimal amount)
    {
        if (amount <= 0)
        {
            throw new ArgumentException("入金額は正の値である必要があります");
        }

        _balance += amount;
    }

    public bool Withdraw(decimal amount)
    {
        if (amount <= 0)
        {
            throw new ArgumentException("出金額は正の値である必要があります");
        }

        if (_balance >= amount)
        {
            _balance -= amount;
            return true;
        }

        return false;  // 残高不足
    }

    // 口座情報の表示
    public void DisplayAccountInfo()
    {
        // 口座番号の一部を隠す
        string maskedNumber = "xxxx-xxxx-xxxx-" +
_accountNumber.Substring(_accountNumber.Length - 4);

        Console.WriteLine($"口座名義: {AccountHolder}");
        Console.WriteLine($"口座番号: {maskedNumber}");
        Console.WriteLine($"残高: {Balance}円");
    }
}

// 使用例
BankAccount account = new BankAccount("山田太郎", "1234-5678-9012-3456");
account.Deposit(10000);
account.DisplayAccountInfo();

bool withdrawResult = account.Withdraw(3000);
Console.WriteLine($"出金結果: {(withdrawResult ? "成功" : "失敗")}");
account.DisplayAccountInfo();
```

```
// これは動作しない（privateフィールドにアクセスできない）
// account._balance = 1000000;  // コンパイルエラー

// これは動作しない（privateセッターにアクセスできない）
// account.Balance = 1000000;  // コンパイルエラー
```

このクラスでは、残高（`_balance`）と口座番号（`_accountNumber`）を直接アクセスできないように隠し、入金（`Deposit`）と出金（`Withdraw`）の操作を通じてのみ残高を変更できるようにしています。

> **ベテランの知恵袋**: カプセル化は「情報隠蔽」とも呼ばれますが、その目的は単にデータを隠すことではなく、クラスの内部実装の詳細をユーザーから分離することです。これにより、内部の実装を変更しても外部のコードに影響を与えずに済みます。また、データの整合性を保つためのルールを集中管理できます。

アクセス修飾子

C#には、クラスのメンバーに対するアクセスレベルを制御するためのアクセス修飾子があります：

修飾子	説明
`public`	どこからでもアクセスできる
`private`	同じクラス内からのみアクセスできる
`protected`	同じクラスと派生クラスからアクセスできる
`internal`	同じアセンブリ（プロジェクト）内からアクセスできる
`protected internal`	同じアセンブリ内、または任意のアセンブリ内の派生クラスからアクセスできる
`private protected`	同じアセンブリ内の派生クラスからのみアクセスできる（C# 7.2以降）

```
class AccessModifierExample
{
    public int PublicField;               // どこからでもアクセス可能
    private int PrivateField;             // このクラス内からのみアクセス可能
    protected int ProtectedField;         // このクラスと派生クラスからアクセス可能
    internal int InternalField;           // 同じアセンブリ内からアクセス可能
    protected internal int ProtectedInternalField;  // 同じアセンブリ内または派生クラスからアクセス可能
    private protected int PrivateProtectedField;    // 同じアセンブリ内の派生クラスからアクセス可能
}
```

> **若手の疑問解決**: 「デフォルトのアクセス修飾子はどれですか？」
>
> クラスメンバー（フィールド、メソッド、プロパティなど）のデフォルトのアクセス修飾子は`private`です。一方、クラス自体のデフォルトは`internal`です。基本的には、必要最小限のアクセスレベルを選択し、明示的に指定することをお勧めします。

namespaceとusing

名前空間（namespace）は、関連するクラスをグループ化し、名前の衝突を防ぐための仕組みです：

```
// 名前空間の定義
namespace MyCompany.MyProject.Utilities
{
    public class Logger
    {
        public static void Log(string message)
        {
            Console.WriteLine($"ログ: {message}");
        }
    }
}

// 別のファイルでの使用
using MyCompany.MyProject.Utilities;  // 名前空間のインポート

class Program
{
    static void Main()
    {
        Logger.Log("テストメッセージ");  // インポートした名前空間のクラスを使用
    }
}

// エイリアスを使用
using MyLogger = MyCompany.MyProject.Utilities.Logger;

class AnotherProgram
{
    static void Main()
    {
        MyLogger.Log("別のテストメッセージ");  // エイリアスを使用
    }
}
```

> **失敗から学ぶ**: あるプロジェクトでは、複数のチームが別々のライブラリを開発していましたが、名前空間の設計が不十分だったため、同じ名前のクラスが複数存在するという問題が発生しました。これにより、どちらのクラスを使用すべきかの混乱が生じ、バグの原因となりました。名前空間は、会社名、プロジェクト名、機能領域などの階層に基づいて慎重に設計することの重要性を学びました。

章末チェックリスト

- [ ] メソッドを定義し、パラメータと戻り値を適切に使用できる
- [ ] オプションパラメータと名前付き引数を理解し使用できる
- [ ] メソッドのオーバーロードを実装できる
- [ ] 値渡しと参照渡し（ref, out）の違いを理解している
- [ ] クラスを定義し、フィールドとメソッドを適切に配置できる
- [ ] コンストラクタを使ってオブジェクトを初期化できる
- [ ] プロパティを使ってフィールドへのアクセスを制御できる
- [ ] 静的メンバーと静的クラスを適切に使用できる
- [ ] カプセル化の概念を理解し、適切なアクセス修飾子を選択できる
- [ ] 名前空間を定義し、usingディレクティブを使って他の名前空間のクラスを利用できる

まとめと次のステップ

この章では、メソッドとクラスの基礎について学びました。メソッドを使うことでコードを再利用可能な部品に分割し、クラスを使うことでデータと操作をひとつのユニットにまとめることができます。また、カプセル化やアクセス修飾子といったオブジェクト指向プログラミングの基本概念も理解しました。

次の章では、オブジェクト指向プログラミングの実践として、継承やポリモーフィズム、インターフェイスなどの高度な概念について学びます。これらを理解することで、より柔軟で拡張性の高いコードを書けるようになります。

第5章: オブジェクト指向プログラミングの実践

前章ではクラスの基本的な概念を学びました。この章では、オブジェクト指向プログラミング（OOP）のより高度な概念を探求し、C#でそれらをどのように実装するかを学びます。

継承の基本

継承は、既存のクラス（基底クラスまたは親クラス）から新しいクラス（派生クラスまたは子クラス）を作成する仕組みです。派生クラスは基底クラスのすべてのメンバーを継承し、独自のメンバーを追加したり、基底クラスのメンバーを変更したりできます。

```
// 基底クラス
public class Animal
{
    public string Name { get; set; }

    public Animal(string name)
    {
        Name = name;
    }

    public virtual void MakeSound()
    {
        Console.WriteLine("動物が鳴いています");
    }

    public void Sleep()
    {
        Console.WriteLine($"{Name}は眠っています");
    }
}

// 派生クラス
public class Dog : Animal
{
    public string Breed { get; set; }

    // 基底クラスのコンストラクタを呼び出す
    public Dog(string name, string breed) : base(name)
    {
        Breed = breed;
    }
```

```
    // 基底クラスのメソッドをオーバーライド
    public override void MakeSound()
    {
        Console.WriteLine("ワンワン！");
    }

    // 派生クラス独自のメソッド
    public void Fetch()
    {
        Console.WriteLine($"{Name}はボールを取ってきました");
    }
}

// 使用例
Animal genericAnimal = new Animal("生き物");
genericAnimal.MakeSound();  // 出力: 動物が鳴いています
genericAnimal.Sleep();      // 出力: 生き物は眠っています

Dog dog = new Dog("ポチ", "柴犬");
dog.MakeSound();  // 出力: ワンワン！
dog.Sleep();      // 出力: ポチは眠っています（基底クラスから継承）
dog.Fetch();      // 出力: ポチはボールを取ってきました

// 基底クラスの変数で派生クラスのインスタンスを参照
Animal animalDog = new Dog("ジョン", "ゴールデンレトリバー");
animalDog.MakeSound();  // 出力: ワンワン！（ポリモーフィズム）
// animalDog.Fetch();   // コンパイルエラー（Animal型にはFetchメソッドがない）
```

> **若手の疑問解決**: 「`virtual` と `override` キーワードは何のためにあるのですか？」
>
> `virtual` キーワードは、基底クラスのメソッドが派生クラスでオーバーライド（再定義）可能であることを示します。`override` キーワードは、派生クラスで基底クラスの仮想メソッドを実際に再定義することを示します。この仕組みにより、派生クラスは基底クラスの振る舞いを変更することができます。これはポリモーフィズム（多態性）と呼ばれる重要な概念の基盤です。

is-a関係

継承は「is-a」関係を表します。つまり、「犬は動物である」という関係です。このような関係がある場合に継承を使用します。

```
// さらに派生クラスを追加
public class Cat : Animal
{
    public Cat(string name) : base(name) { }

    public override void MakeSound()
    {
        Console.WriteLine("ニャー！");
    }

    public void Climb()
    {
        Console.WriteLine($"{Name}は木に登りました");
    }
```

```
}

// 継承を使用した多様性の例
public void PetAnimal(Animal animal)
{
    Console.WriteLine($"{animal.Name}を撫でています");
    animal.MakeSound();
}

// 異なる動物に同じメソッドを使用
Dog dog = new Dog("ポチ", "柴犬");
Cat cat = new Cat("タマ");

PetAnimal(dog);  // 出力: ポチを撫でています / ワンワン！
PetAnimal(cat);  // 出力: タマを撫でています / ニャー！
```

シールドクラスとメソッド

継承やオーバーライドを禁止したい場合は、`sealed` キーワードを使用します：

```
// このクラスは継承できない
public sealed class SpecialAnimal : Animal
{
    public SpecialAnimal(string name) : base(name) { }

    public override void MakeSound()
    {
        Console.WriteLine("特別な鳴き声");
    }
}

// コンパイルエラー：シールドクラスを継承できない
// public class SuperSpecialAnimal : SpecialAnimal { }

// メソッドのみをシールドする例
public class Bird : Animal
{
    public Bird(string name) : base(name) { }

    // このメソッドは派生クラスでオーバーライドできない
    public sealed override void MakeSound()
    {
        Console.WriteLine("チュンチュン");
    }
}

public class Sparrow : Bird
{
    public Sparrow(string name) : base(name) { }

    // コンパイルエラー：シールドメソッドをオーバーライドできない
    // public override void MakeSound() { }
}
```

> **ベテランの知恵袋**: 継承は強力なツールですが、過度に使用すると複雑なクラス階層が生まれ、コードの理解と保守が難しくなります。「継承よりコンポジション（構成）を優先する」という原則を覚えておくと良いでしょう。単純に機能を再利用したい場合は、継承ではなくオブジェクトを構成要素として使用することを検討してください。

抽象クラスとメソッド

抽象クラスは、直接インスタンス化できないクラスです。抽象メソッドは、実装を持たず、派生クラスでオーバーライドすることを強制するメソッドです。

```
// 抽象クラス
public abstract class Shape
{
    // 通常のプロパティ
    public string Color { get; set; }

    // コンストラクタ
    public Shape(string color)
    {
        Color = color;
    }

    // 通常のメソッド
    public void DisplayColor()
    {
        Console.WriteLine($"色: {Color}");
    }

    // 抽象メソッド（実装なし）
    public abstract double CalculateArea();

    // 抽象メソッド
    public abstract double CalculatePerimeter();
}

// 派生クラス
public class Circle : Shape
{
    public double Radius { get; set; }

    public Circle(string color, double radius) : base(color)
    {
        Radius = radius;
    }

    // 抽象メソッドの実装
    public override double CalculateArea()
    {
        return Math.PI * Radius * Radius;
    }

    // 抽象メソッドの実装
    public override double CalculatePerimeter()
    {
        return 2 * Math.PI * Radius;
```

```
    }
}

public class Rectangle : Shape
{
    public double Width { get; set; }
    public double Height { get; set; }

    public Rectangle(string color, double width, double height) : base(color)
    {
        Width = width;
        Height = height;
    }

    public override double CalculateArea()
    {
        return Width * Height;
    }

    public override double CalculatePerimeter()
    {
        return 2 * (Width + Height);
    }
}

// 使用例
// Shape shape = new Shape("赤");  // コンパイルエラー：抽象クラスはインスタンス化できない

Circle circle = new Circle("青", 5);
circle.DisplayColor();  // 出力: 色: 青
Console.WriteLine($"円の面積: {circle.CalculateArea()}");
Console.WriteLine($"円の周囲長: {circle.CalculatePerimeter()}");

Rectangle rectangle = new Rectangle("緑", 4, 6);
rectangle.DisplayColor();  // 出力: 色: 緑
Console.WriteLine($"長方形の面積: {rectangle.CalculateArea()}");
Console.WriteLine($"長方形の周囲長: {rectangle.CalculatePerimeter()}");

// 抽象クラス型の変数で派生クラスのインスタンスを参照
Shape shape = new Circle("赤", 3);
Console.WriteLine($"形の面積: {shape.CalculateArea()}");  // 出力: 形の面積: 28.27...
```

抽象クラスは、関連するクラスの共通の振る舞いを定義するテンプレートのような役割を果たします。

> **プロジェクト事例**: グラフィックエディタアプリケーションの開発では、様々な図形（円、四角形、三角形など）を描画する必要がありました。これらの図形には共通のプロパティ（位置、色など）や操作（移動、拡大縮小など）がありました。抽象クラス `Shape` を作成し、各図形クラスがこれを継承することで、コードの重複を減らし、新しい図形の追加も容易になりました。

インターフェース

インターフェースは、クラスが実装すべきメソッドやプロパティの「契約」を定義します。C#では、クラスは複数のインターフェースを実装できますが、継承できる基底クラスは1つだけです（単一継承）。

```
// インターフェースの定義
public interface IMovable
{
    // メソッドの宣言（実装はなし）
    void Move(double x, double y);

    // プロパティの宣言
    double Speed { get; set; }
}

// 別のインターフェース
public interface IResizable
{
    void Resize(double factor);
    double Size { get; set; }
}

// インターフェースを実装するクラス
public class Car : IMovable
{
    // インターフェースで定義されたプロパティの実装
    public double Speed { get; set; }

    // インターフェースで定義されたメソッドの実装
    public void Move(double x, double y)
    {
        Console.WriteLine($"車が位置({x}, {y})に移動しました。速度: {Speed}km/h");
    }

    // クラス独自のメソッド
    public void Honk()
    {
        Console.WriteLine("ビビビー！");
    }
}

// 複数のインターフェースを実装するクラス
public class GameCharacter : IMovable, IResizable
{
    public double Speed { get; set; }
    public double Size { get; set; }

    public void Move(double x, double y)
    {
        Console.WriteLine($"キャラクターが位置({x}, {y})に移動しました。速度: {Speed}");
    }

    public void Resize(double factor)
    {
        Size *= factor;
        Console.WriteLine($"キャラクターのサイズが{Size}に変更されました");
    }
}

// 使用例
Car car = new Car { Speed = 60 };
```

```
car.Move(100, 200);
car.Honk();

GameCharacter character = new GameCharacter { Speed = 5, Size = 1.0 };
character.Move(10, 20);
character.Resize(1.5);

// インターフェース型の変数
IMovable movable1 = new Car { Speed = 80 };
IMovable movable2 = new GameCharacter { Speed = 3 };

movable1.Move(50, 60);  // 車の移動メソッドが呼ばれる
movable2.Move(30, 40);  // キャラクターの移動メソッドが呼ばれる

// このようなメソッドは、IMovableを実装するどんなクラスでも受け取れる
public void MoveToDestination(IMovable entity, double x, double y)
{
    entity.Speed = 10;  // 速度設定
    entity.Move(x, y);  // 移動
}

MoveToDestination(car, 300, 400);
MoveToDestination(character, 70, 80);
```

> **若手の疑問解決**: 「抽象クラスとインターフェースの違いは何ですか？どちらを使うべきですか？」
>
> 主な違いは以下の通りです：
>
> 1. クラスは複数のインターフェースを実装できますが、継承できる抽象クラスは1つだけです
> 2. 抽象クラスはフィールドやコンストラクタを持ち、実装済みのメソッドも含められますが、インターフェースは通常メソッドやプロパティの宣言のみです（C# 8.0からはデフォルト実装も可能）
> 3. 抽象クラスはアクセス修飾子（public, private等）を使用できますが、インターフェースのメンバーは常にpublicです
>
> 使い分けのガイドライン：
>
> - 「is-a」関係（「〜は〜である」という関係）には抽象クラスを使用する
> - 「can-do」関係（「〜ができる」という関係）にはインターフェースを使用する
> - 共通の実装を提供したい場合は抽象クラスを使用する
> - クラスに複数の機能を提供したい場合はインターフェースを使用する

ポリモーフィズム（多態性）

ポリモーフィズムは、同じインターフェースを使用して異なる基底クラスの異なる実装にアクセスする能力です。これにより、コードの柔軟性と再利用性が向上します。

```
// 基底クラス
public class Animal
{
    public string Name { get; set; }

    public Animal(string name)
    {
        Name = name;
```

```
    }

    public virtual void MakeSound()
    {
        Console.WriteLine("何らかの音を出しています");
    }
}

// 派生クラス
public class Dog : Animal
{
    public Dog(string name) : base(name) { }

    public override void MakeSound()
    {
        Console.WriteLine("ワンワン！");
    }
}

public class Cat : Animal
{
    public Cat(string name) : base(name) { }

    public override void MakeSound()
    {
        Console.WriteLine("ニャー！");
    }
}

public class Cow : Animal
{
    public Cow(string name) : base(name) { }

    public override void MakeSound()
    {
        Console.WriteLine("モー！");
    }
}

// ポリモーフィズムの例
public void AnimalSounds(List<Animal> animals)
{
    foreach (Animal animal in animals)
    {
        Console.Write($"{animal.Name}の鳴き声: ");
        animal.MakeSound();  // それぞれの動物に応じた鳴き声メソッドが呼ばれる
    }
}

// 使用例
List<Animal> animals = new List<Animal>
{
    new Dog("ポチ"),
    new Cat("タマ"),
    new Cow("モーモー"),
    new Animal("謎の生物")
};
```

```
AnimalSounds(animals);
// 出力:
// ポチの鳴き声: ワンワン！
// タマの鳴き声: ニャー！
// モーモーの鳴き声: モー！
// 謎の生物の鳴き声: 何らかの音を出しています
```

上記の例では、`AnimalSounds`メソッドは`Animal`型のリストを受け取りますが、実際には`Dog`、`Cat`、`Cow`などの派生クラスのインスタンスを含むことができます。各オブジェクトの実際の型に応じて、適切な`MakeSound`メソッドが呼び出されます。

> **ベテランの知恵袋**: ポリモーフィズムを活用すると、新しい種類のオブジェクトを追加する際に既存のコードを変更する必要がなくなります。例えば、新しい動物クラス`Pig`を追加しても、`AnimalSounds`メソッドを変更する必要はありません。これは「開放/閉鎖の原則」（拡張に対して開かれ、修正に対して閉じている）と呼ばれるオブジェクト指向設計の重要な原則です。

抽象クラスとインターフェースの組み合わせ

実際のアプリケーションでは、抽象クラスとインターフェースを組み合わせて使用することが多いです：

```
// インターフェース
public interface IPrintable
{
    void Print();
}

public interface ISavable
{
    void Save(string filename);
}

// 抽象クラス
public abstract class Document
{
    public string Title { get; set; }
    public string Author { get; set; }

    // コンストラクタ
    public Document(string title, string author)
    {
        Title = title;
        Author = author;
    }

    // 共通メソッド
    public void DisplayInfo()
    {
        Console.WriteLine($"タイトル: {Title}");
        Console.WriteLine($"著者: {Author}");
    }

    // 抽象メソッド
    public abstract void Render();
}
```

```
// 具象クラス - 抽象クラスを継承し、インターフェースを実装
public class TextDocument : Document, IPrintable, ISavable
{
    public string Content { get; set; }

    public TextDocument(string title, string author, string content)
        : base(title, author)
    {
        Content = content;
    }

    // Document抽象クラスのメソッドを実装
    public override void Render()
    {
        Console.WriteLine("テキストドキュメントを表示:");
        Console.WriteLine(Content);
    }

    // IPrintableインターフェースのメソッドを実装
    public void Print()
    {
        Console.WriteLine($"テキストドキュメント '{Title}' を印刷中...");
    }

    // ISavableインターフェースのメソッドを実装
    public void Save(string filename)
    {
        Console.WriteLine($"テキストドキュメントを '{filename}' に保存中...");
    }
}

public class PDFDocument : Document, IPrintable, ISavable
{
    public int PageCount { get; set; }

    public PDFDocument(string title, string author, int pageCount)
        : base(title, author)
    {
        PageCount = pageCount;
    }

    public override void Render()
    {
        Console.WriteLine($"PDFドキュメント '{Title}' を表示しています（{PageCount}ページ）");
    }

    public void Print()
    {
        Console.WriteLine($"PDFドキュメント '{Title}' を印刷中...");
    }

    public void Save(string filename)
    {
        Console.WriteLine($"PDFドキュメントを '{filename}' に保存中...");
    }
```

```
}

// 使用例
void ProcessDocument(Document doc)
{
    doc.DisplayInfo();  // 抽象クラスの共通メソッド
    doc.Render();       // オーバーライドされたメソッド

    // インターフェースのチェックと使用
    if (doc is IPrintable printable)
    {
        printable.Print();
    }

    if (doc is ISavable savable)
    {
        savable.Save("document.dat");
    }
}

TextDocument textDoc = new TextDocument(
    "プログラミング入門",
    "山田太郎",
    "C#は素晴らしい言語です。"
);

PDFDocument pdfDoc = new PDFDocument(
    "C#完全ガイド",
    "鈴木花子",
    300
);

ProcessDocument(textDoc);
Console.WriteLine();
ProcessDocument(pdfDoc);
```

> **プロジェクト事例**: あるドキュメント管理システムでは、テキスト、PDF、画像、スプレッドシートなど様々な種類のドキュメントを扱う必要がありました。抽象クラス `Document` で共通のプロパティやメソッドを定義し、各ドキュメント型に特化した機能をインターフェース（`IPrintable`、`IEditable`、`ISearchable` など）で定義することで、柔軟で拡張しやすいシステムを構築できました。

C# 8.0以降のインターフェースのデフォルト実装

C# 8.0からは、インターフェースにデフォルト実装を提供できるようになりました：

```
// デフォルト実装を持つインターフェース
public interface ILogger
{
    // デフォルト実装を持つメソッド
    void Log(string message)
    {
        Console.WriteLine($"デフォルトログ: {message}");
    }
```

```
    // デフォルト実装を持つプロパティ
    string LogLevel { get; set; }

    // デフォルト実装がないメソッド（従来通り）
    void LogError(string message);
}

// 最小限の実装
public class SimpleLogger : ILogger
{
    public string LogLevel { get; set; } = "Info";

    // LogErrorのみ実装（Logはデフォルト実装を使用）
    public void LogError(string message)
    {
        Console.WriteLine($"エラー: {message}");
    }
}

// デフォルト実装をオーバーライド
public class ConsoleLogger : ILogger
{
    public string LogLevel { get; set; } = "Debug";

    // デフォルト実装をオーバーライド
    public void Log(string message)
    {
        Console.WriteLine($"{DateTime.Now} [{LogLevel}]: {message}");
    }

    public void LogError(string message)
    {
        LogLevel = "Error";
        Log(message);
    }
}

// 使用例
ILogger simpleLogger = new SimpleLogger();
simpleLogger.Log("これはデフォルト実装を使用します");  // デフォルト実装
simpleLogger.LogError("エラーメッセージ");             // カスタム実装

ILogger consoleLogger = new ConsoleLogger();
consoleLogger.Log("カスタムログ");                     // オーバーライドされた実装
consoleLogger.LogError("致命的なエラー");              // カスタム実装
```

この機能により、インターフェースにデフォルトの振る舞いを提供しつつ、必要に応じてカスタマイズできるようになりました。

> **若手の疑問解決**: 「インターフェースにデフォルト実装を提供できるなら、抽象クラスと何が違うのですか？」
>
> 確かに機能的な重複が増えましたが、いくつかの重要な違いがあります：
>
> 1. クラスは依然として単一継承のみ（1つの抽象クラスしか継承できない）ですが、複数のインターフェースを実装できます

> 2. インターフェースはフィールドやコンストラクタを持つことができません
> 3. インターフェースのデフォルト実装はメンバーをオーバーライドすることなく利用できます
>
> デフォルト実装は主に、既存のインターフェースに新しいメソッドを追加する際に互換性を保つために導入されました。新しいメソッドを追加する際に、すべての実装クラスを更新する必要がなくなります。

オブジェクト指向設計の原則（SOLID）

オブジェクト指向プログラミングを実践する際には、SOLIDと呼ばれる5つの設計原則を覚えておくと役立ちます：

1. **単一責任の原則**（Single Responsibility Principle）
   - クラスは単一の責任を持つべき
   - 変更の理由が1つだけになるようにする
2. **開放/閉鎖の原則**（Open/Closed Principle）
   - クラスは拡張に対して開かれ、修正に対して閉じているべき
   - 新しい機能を追加する際に既存のコードを変更しないようにする
3. **リスコフの置換原則**（Liskov Substitution Principle）
   - 派生クラスは基底クラスと置き換え可能であるべき
   - 派生クラスで基底クラスの契約を破らないようにする
4. **インターフェース分離の原則**（Interface Segregation Principle）
   - クライアントは自分が使用しないメソッドに依存すべきではない
   - 大きなインターフェースは小さく特化したインターフェースに分割する
5. **依存性逆転の原則**（Dependency Inversion Principle）
   - 高レベルのモジュールは低レベルのモジュールに依存すべきではない
   - 両方が抽象に依存すべき

```
// SOLIDの原則に従った例

// 単一責任の原則：ユーザー情報と認証を別々のクラスに分離
public class User
{
    public string Username { get; set; }
    public string Email { get; set; }
}

public class AuthenticationService
{
    public bool ValidateCredentials(string username, string password)
    {
        // 認証ロジック
        return true;
    }
}

// 開放/閉鎖の原則：新しい種類の支払い方法を追加しやすい設計
public interface IPaymentMethod
{
    void ProcessPayment(decimal amount);
}
```

```
public class CreditCardPayment : IPaymentMethod
{
    public void ProcessPayment(decimal amount)
    {
        Console.WriteLine($"クレジットカードで{amount}円支払い処理");
    }
}

public class PayPalPayment : IPaymentMethod
{
    public void ProcessPayment(decimal amount)
    {
        Console.WriteLine($"PayPalで{amount}円支払い処理");
    }
}

// 新しい支払い方法を追加しても、PaymentProcessorを変更する必要はない
public class PaymentProcessor
{
    public void ProcessOrder(decimal amount, IPaymentMethod paymentMethod)
    {
        paymentMethod.ProcessPayment(amount);
    }
}

// リスコフの置換原則：基底クラスと派生クラスが置き換え可能
public class Rectangle
{
    public virtual double Width { get; set; }
    public virtual double Height { get; set; }

    public double CalculateArea()
    {
        return Width * Height;
    }
}

public class Square : Rectangle
{
    private double _side;

    public override double Width
    {
        get { return _side; }
        set { _side = value; }
    }

    public override double Height
    {
        get { return _side; }
        set { _side = value; }
    }
}

// インターフェース分離の原則：インターフェースを機能ごとに分割
public interface IReadable
{
```

```
    string Read(string filename);
}

public interface IWritable
{
    void Write(string filename, string content);
}

public class TextFileHandler : IReadable, IWritable
{
    public string Read(string filename)
    {
        return $"{filename}の内容を読み込み中...";
    }

    public void Write(string filename, string content)
    {
        Console.WriteLine($"{filename}に内容を書き込み中...");
    }
}

public class ReadOnlyFile : IReadable
{
    public string Read(string filename)
    {
        return $"{filename}の内容を読み込み中...";
    }
}

// 依存性逆転の原則：高レベルモジュールが低レベルの実装ではなく抽象に依存
public interface ILogger
{
    void Log(string message);
}

public class ConsoleLogger : ILogger
{
    public void Log(string message)
    {
        Console.WriteLine($"ログ: {message}");
    }
}

public class FileLogger : ILogger
{
    public void Log(string message)
    {
        Console.WriteLine($"ファイルにログ出力: {message}");
    }
}

public class OrderService
{
    private readonly ILogger _logger;

    // 具体的なロガーではなく、インターフェースに依存
    public OrderService(ILogger logger)
```

```
    {
        _logger = logger;
    }

    public void PlaceOrder()
    {
        // 注文処理
        _logger.Log("注文が完了しました");
    }
}

// 使用例
OrderService service1 = new OrderService(new ConsoleLogger());
OrderService service2 = new OrderService(new FileLogger());
```

> **ベテランの知恵袋**: SOLIDの原則は、ガイドラインであってルールではありません。これらの原則を機械的に適用するよりも、プロジェクトの複雑さや要件に基づいて、どこまで適用するかを判断することが重要です。小規模なアプリケーションでは、過度に複雑な設計を避けるために、部分的に適用するだけで十分な場合もあります。

章末チェックリスト

- [ ] 継承を使用して基底クラスから派生クラスを作成できる
- [ ] virtualメソッドとoverrideメソッドの仕組みを理解している
- [ ] 抽象クラスと抽象メソッドを作成し、派生クラスで実装できる
- [ ] インターフェースを定義し、クラスに実装できる
- [ ] ポリモーフィズムを活用して、異なるクラスのオブジェクトを同じ方法で扱える
- [ ] C# 8.0以降のインターフェースのデフォルト実装を理解している
- [ ] SOLIDの原則を説明し、コードに適用できる
- [ ] 抽象クラスとインターフェースの使い分けの基準を理解している
- [ ] シールドクラスとシールドメソッドの用途を理解している
- [ ] 複数のインターフェースを実装するクラスを作成できる

まとめと次のステップ

この章では、継承、ポリモーフィズム、抽象クラス、インターフェースなど、オブジェクト指向プログラミングの高度な概念について学びました。これらの概念を理解し活用することで、より柔軟で保守性の高いプログラムを設計できるようになります。

次の章では、コレクションとLINQについて学びます。コレクションは複数のオブジェクトを管理するための仕組みであり、LINQはそれらのデータに対して強力なクエリ操作を提供します。これにより、データ操作がさらに直感的かつ効率的になります。

第6章: コレクションとLINQ

プログラミングでは、複数のデータを扱うことが非常に一般的です。C#には、データのコレクションを効率的に管理するためのクラスと、それらを操作するための強力なクエリ機能（LINQ）が用意されています。

配列の基本

配列は、同じ型の複数の要素を格納するための最も基本的なデータ構造です：

```
// 配列の宣言と初期化
int[] numbers = new int[5];  // 5つの整数を格納できる配列

// 個々の要素に値を代入
numbers[0] = 10;
numbers[1] = 20;
numbers[2] = 30;
numbers[3] = 40;
numbers[4] = 50;

// 初期化と同時に値を設定
string[] fruits = new string[] { "りんご", "バナナ", "オレンジ" };

// さらに簡潔な書き方
string[] colors = { "赤", "青", "緑" };

// 配列の要素にアクセス
Console.WriteLine(numbers[2]);  // 出力: 30
Console.WriteLine(fruits[0]);   // 出力: りんご

// 配列の長さ
Console.WriteLine(numbers.Length);  // 出力: 5

// 配列の全要素を反復処理
foreach (int number in numbers)
{
    Console.WriteLine(number);
}

// forループを使った反復処理
for (int i = 0; i < colors.Length; i++)
{
    Console.WriteLine($"色 {i+1}: {colors[i]}");
}
```

多次元配列

C#では、多次元配列も作成できます：

```
// 2次元配列
int[,] matrix = new int[3, 4];  // 3行4列の配列

// 2次元配列の初期化
int[,] grid = {
    { 1, 2, 3, 4 },
    { 5, 6, 7, 8 },
    { 9, 10, 11, 12 }
};

// 要素にアクセス
Console.WriteLine(grid[1, 2]);  // 出力: 7（2行目、3列目の要素）

// 2次元配列の全要素を反復処理
```

```
for (int i = 0; i < grid.GetLength(0); i++)
{
    for (int j = 0; j < grid.GetLength(1); j++)
    {
        Console.Write($"{grid[i, j],3}");  // 3桁の幅で整形
    }
    Console.WriteLine();
}
```

ジャグ配列（配列の配列）

ジャグ配列は、各行の長さが異なる多次元配列です：

```
// ジャグ配列の宣言
int[][] jaggedArray = new int[3][];

// 各行の初期化
jaggedArray[0] = new int[] { 1, 2 };          // 1行目: 2要素
jaggedArray[1] = new int[] { 3, 4, 5, 6 };    // 2行目: 4要素
jaggedArray[2] = new int[] { 7, 8, 9 };       // 3行目: 3要素

// 要素にアクセス
Console.WriteLine(jaggedArray[1][2]);  // 出力: 5

// ジャグ配列の全要素を反復処理
for (int i = 0; i < jaggedArray.Length; i++)
{
    for (int j = 0; j < jaggedArray[i].Length; j++)
    {
        Console.Write($"{jaggedArray[i][j]} ");
    }
    Console.WriteLine();
}
```

> **若手の疑問解決**: 「2次元配列とジャグ配列、どちらを使うべきですか？」
>
> 各行が同じ長さのデータ（例：チェス盤、画像ピクセルなど）を表す場合は2次元配列が適しています。各行の長さが異なる可能性がある場合（例：不規則な形のデータ、可変長のリスト群など）はジャグ配列が適しています。また、パフォーマンスの観点からは、ジャグ配列の方がメモリアクセスが効率的な場合があります。

コレクションの基本

.NET Framework/.NET Coreには、さまざまなコレクションクラスが用意されています。配列とは異なり、ほとんどのコレクションは動的にサイズを変更できます。

List<T>

`List<T>` は、最も一般的に使用される動的配列です：

```
// Listの作成
List<int> numbers = new List<int>();

// 要素の追加
```

```
numbers.Add(10);
numbers.Add(20);
numbers.Add(30);

// 初期化と同時に要素を追加
List<string> names = new List<string> { "田中", "鈴木", "佐藤" };

// インデックスによるアクセス
Console.WriteLine(numbers[1]);  // 出力: 20
Console.WriteLine(names[0]);    // 出力: 田中

// 要素の挿入
numbers.Insert(1, 15);  // インデックス1の位置に15を挿入
// numbers: [10, 15, 20, 30]

// 要素の削除
numbers.Remove(15);     // 値15を削除
numbers.RemoveAt(0);    // インデックス0の要素を削除
// numbers: [20, 30]

// リストの要素数
Console.WriteLine(numbers.Count);  // 出力: 2

// リストに特定の要素が含まれているか確認
bool containsNumber = numbers.Contains(20);  // true

// リスト内の要素の検索
int index = names.IndexOf("鈴木");  // 1

// リストの全要素を反復処理
foreach (string name in names)
{
    Console.WriteLine(name);
}
```

Dictionary<TKey, TValue>

`Dictionary<TKey, TValue>` は、キーと値のペアを格納するコレクションです：

```
// Dictionaryの作成
Dictionary<string, int> ages = new Dictionary<string, int>();

// 要素の追加
ages.Add("田中", 30);
ages.Add("鈴木", 25);
ages.Add("佐藤", 40);

// 初期化と同時に要素を追加
Dictionary<string, string> capitals = new Dictionary<string, string>
{
    { "日本", "東京" },
    { "アメリカ", "ワシントンD.C." },
    { "イギリス", "ロンドン" }
};

// より簡潔な初期化構文（C# 6.0以降）
```

```
Dictionary<string, int> scores = new Dictionary<string, int>
{
    ["田中"] = 85,
    ["鈴木"] = 92,
    ["佐藤"] = 78
};

// キーを使って値にアクセス
Console.WriteLine(ages["鈴木"]);       // 出力: 25
Console.WriteLine(capitals["日本"]);  // 出力: 東京

// キーの存在確認
if (ages.ContainsKey("伊藤"))
{
    Console.WriteLine(ages["伊藤"]);
}
else
{
    Console.WriteLine("キー「伊藤」は存在しません");
}

// TryGetValueの使用
if (capitals.TryGetValue("フランス", out string capital))
{
    Console.WriteLine($"フランスの首都は{capital}です");
}
else
{
    Console.WriteLine("フランスの情報はありません");
}

// 要素の追加/更新
scores["佐藤"] = 80;  // 既存の値を更新
scores["伊藤"] = 95;  // 新しいキーと値を追加

// 要素の削除
ages.Remove("田中");

// すべてのキーと値のペアを反復処理
foreach (KeyValuePair<string, int> pair in scores)
{
    Console.WriteLine($"{pair.Key}: {pair.Value}点");
}

// キーだけを反復処理
foreach (string name in scores.Keys)
{
    Console.WriteLine(name);
}

// 値だけを反復処理
foreach (int score in scores.Values)
{
    Console.WriteLine(score);
}
```

> **ベテランの知恵袋**: Dictionaryは内部的にハッシュテーブルを使用しているため、大量のデータでも高速に検索できます。ただし、キーはユニークである必要があります。同じキーを持つ項目を複数格納したい場合は、`Dictionary<TKey, List<TValue>>`のような構造を使用するか、`Lookup<TKey, TValue>`や`GroupBy`操作を検討してください。

HashSet<T>

`HashSet<T>`は、重複のない要素のコレクションです：

```
// HashSetの作成
HashSet<int> uniqueNumbers = new HashSet<int>();

// 要素の追加
uniqueNumbers.Add(1);
uniqueNumbers.Add(2);
uniqueNumbers.Add(3);
uniqueNumbers.Add(2);  // 重複するので追加されない

// 初期化と同時に要素を追加
HashSet<string> fruits = new HashSet<string> { "りんご", "バナナ", "オレンジ", "りんご" };

// 要素数
Console.WriteLine(uniqueNumbers.Count);  // 出力: 3（重複は数えない）
Console.WriteLine(fruits.Count);         // 出力: 3（重複は数えない）

// 要素の存在確認
bool contains = uniqueNumbers.Contains(2);  // true

// 要素の削除
uniqueNumbers.Remove(3);

// 集合演算
HashSet<int> setA = new HashSet<int> { 1, 2, 3, 4, 5 };
HashSet<int> setB = new HashSet<int> { 3, 4, 5, 6, 7 };

// 和集合（セットAとセットBのすべての要素）
setA.UnionWith(setB);
// setA: { 1, 2, 3, 4, 5, 6, 7 }

// 差集合の例（別のセットを使って示します）
HashSet<int> setC = new HashSet<int> { 1, 2, 3, 4, 5 };
HashSet<int> setD = new HashSet<int> { 3, 4, 5, 6, 7 };

// 差集合（セットCにあってセットDにない要素）
setC.ExceptWith(setD);
// setC: { 1, 2 }

// 交差（両方のセットに共通する要素）
HashSet<int> setE = new HashSet<int> { 1, 2, 3, 4, 5 };
HashSet<int> setF = new HashSet<int> { 3, 4, 5, 6, 7 };

setE.IntersectWith(setF);
// setE: { 3, 4, 5 }

// 対称差（どちらか一方にのみ存在する要素）
HashSet<int> setG = new HashSet<int> { 1, 2, 3, 4, 5 };
```

```
HashSet<int> setH = new HashSet<int> { 3, 4, 5, 6, 7 };

setG.SymmetricExceptWith(setH);
// setG: { 1, 2, 6, 7 }

// すべての要素を反復処理
foreach (int number in uniqueNumbers)
{
    Console.WriteLine(number);
}
```

`HashSet<T>` は、要素の重複を許さず、集合演算を効率的に行いたい場合に特に役立ちます。

> **プロジェクト事例**: あるデータ分析アプリケーションでは、複数のデータソースから収集した顧客IDを処理する必要がありました。`HashSet<string>` を使用することで、重複するIDを自動的に排除し、一意な顧客リストを簡単に作成できました。また、異なるデータソース間の共通顧客や固有の顧客を特定するために、集合演算（交差、差集合など）が非常に役立ちました。

Queue<T>（キュー）

`Queue<T>` はFIFO（First-In-First-Out、先入れ先出し）のデータ構造です：

```
// Queueの作成
Queue<string> messageQueue = new Queue<string>();

// 要素の追加（エンキュー）
messageQueue.Enqueue("メッセージ1");
messageQueue.Enqueue("メッセージ2");
messageQueue.Enqueue("メッセージ3");

// キューの先頭要素を取得（削除せず）
string peek = messageQueue.Peek();  // "メッセージ1"
Console.WriteLine($"次の処理対象: {peek}");

// 要素の取り出し（デキュー）
string message1 = messageQueue.Dequeue();  // "メッセージ1"
Console.WriteLine($"処理中: {message1}");

// キューの要素数
Console.WriteLine($"待機中メッセージ数: {messageQueue.Count}");  // 2

// すべての要素を反復処理（取り出さない）
foreach (string message in messageQueue)
{
    Console.WriteLine($"キュー内: {message}");
}

// すべての要素を取り出す
while (messageQueue.Count > 0)
{
    string message = messageQueue.Dequeue();
    Console.WriteLine($"処理: {message}");
}
```

Stack<T>（スタック）

`Stack<T>` はLIFO（Last-In-First-Out、後入れ先出し）のデータ構造です：

```
// Stackの作成
Stack<string> history = new Stack<string>();

// 要素の追加（プッシュ）
history.Push("ページ1");
history.Push("ページ2");
history.Push("ページ3");

// スタックの先頭要素を取得（削除せず）
string currentPage = history.Peek();  // "ページ3"
Console.WriteLine($"現在のページ: {currentPage}");

// 要素の取り出し（ポップ）
string previousPage = history.Pop();  // "ページ3"
Console.WriteLine($"前のページ: {previousPage}");

// スタックの要素数
Console.WriteLine($"履歴数: {history.Count}");  // 2

// すべての要素を反復処理（取り出さない）
foreach (string page in history)
{
    Console.WriteLine($"履歴内: {page}");
}

// すべての要素を取り出す
while (history.Count > 0)
{
    string page = history.Pop();
    Console.WriteLine($"ページに戻る: {page}");
}
```

> **若手の疑問解決**: 「キューとスタックはどのような場面で使いますか？」
>
> キュー（Queue）は、処理や実行を順番に行いたい場合に役立ちます。例えば：
>
> - プリンターの印刷ジョブ管理
> - メッセージ処理システム
> - タスクスケジューリング
> - 幅優先探索アルゴリズム
>
> スタック（Stack）は、最近行った操作を逆順に追跡したり、再帰的な処理を非再帰的に実装する場合に役立ちます。例えば：
>
> - ウェブブラウザの履歴管理
> - 「元に戻す」機能の実装
> - 式の評価
> - 深さ優先探索アルゴリズム

ジェネリックコレクションとジェネリックプログラミング

C#のコレクションのほとんどは「ジェネリック」です。これは、特定の型のオブジェクトのみを格納するようにコレクションを設定できるということです。ジェネリックを使用することで、型安全性が向上し、パフォーマンスも向上します。

自作のジェネリッククラス

自分でジェネリッククラスを作成することもできます：

```
// ジェネリッククラスの定義
public class Pair<T, U>
{
    public T First { get; set; }
    public U Second { get; set; }

    public Pair(T first, U second)
    {
        First = first;
        Second = second;
    }

    public override string ToString()
    {
        return $"({First}, {Second})";
    }
}

// ジェネリッククラスの使用
Pair<int, string> score = new Pair<int, string>(95, "素晴らしい");
Console.WriteLine(score);  // 出力: (95, 素晴らしい)

Pair<string, bool> status = new Pair<string, bool>("処理完了", true);
Console.WriteLine(status);  // 出力: (処理完了, True)
```

ジェネリックメソッド

特定の型に依存しない一般的なアルゴリズムを実装するジェネリックメソッドも作成できます：

```
// ジェネリックメソッドの定義
public static void Swap<T>(ref T a, ref T b)
{
    T temp = a;
    a = b;
    b = temp;
}

// ジェネリックメソッドの使用
int x = 5, y = 10;
Console.WriteLine($"交換前: x = {x}, y = {y}");
Swap(ref x, ref y);
Console.WriteLine($"交換後: x = {x}, y = {y}");
// 出力:
// 交換前: x = 5, y = 10
// 交換後: x = 10, y = 5

string s1 = "こんにちは", s2 = "さようなら";
```

```
Console.WriteLine($"交換前: s1 = {s1}, s2 = {s2}");
Swap(ref s1, ref s2);
Console.WriteLine($"交換後: s1 = {s1}, s2 = {s2}");
// 出力:
// 交換前: s1 = こんにちは, s2 = さようなら
// 交換後: s1 = さようなら, s2 = こんにちは
```

ジェネリック制約

ジェネリック型パラメータに制約を追加して、特定の条件を満たす型だけを使用できるようにすることもできます：

```
// インターフェース定義
public interface IPrintable
{
    void Print();
}

// ジェネリック制約を持つクラス
public class Printer<T> where T : IPrintable
{
    public void PrintItem(T item)
    {
        item.Print();
    }
}

// インターフェースを実装するクラス
public class Document : IPrintable
{
    public string Title { get; set; }

    public Document(string title)
    {
        Title = title;
    }

    public void Print()
    {
        Console.WriteLine($"文書「{Title}」を印刷中...");
    }
}

// 使用例
Document doc = new Document("レポート");
Printer<Document> printer = new Printer<Document>();
printer.PrintItem(doc);  // 出力: 文書「レポート」を印刷中...

// これはコンパイルエラーになる
// Printer<string> invalidPrinter = new Printer<string>();  // stringはIPrintableを実装していない
```

> **ベテランの知恵袋**: ジェネリック制約は、型パラメータに期待する機能や振る舞いを明確にするのに役立ちます。適切な制約を使用することで、コンパイル時に型の互換性エラーを検出できます。また、制約によって型パラメータの持つメソッドやプロパティにアクセスできるようになるため、より

> 柔軟なコードが書けます。一般的な制約には、`where T : class`（参照型のみ）、`where T : struct`（値型のみ）、`where T : new()`（パラメータなしのコンストラクタを持つ型のみ）などがあります。

LINQ (Language Integrated Query)

LINQ（Language-Integrated Query）は、C#に統合されたクエリ機能で、コレクションに対する強力な操作を提供します。

LINQの基本

```
// サンプルデータ
List<int> numbers = new List<int> { 5, 10, 3, 8, 1, 7, 4, 9, 2, 6 };

// 基本的なフィルタリング - 偶数のみを選択
IEnumerable<int> evenNumbers = numbers.Where(n => n % 2 == 0);
Console.WriteLine("偶数: " + string.Join(", ", evenNumbers));
// 出力: 偶数: 10, 8, 4, 2, 6

// 射影 - 各要素を2倍にする
IEnumerable<int> doubledNumbers = numbers.Select(n => n * 2);
Console.WriteLine("2倍の値: " + string.Join(", ", doubledNumbers));
// 出力: 2倍の値: 10, 20, 6, 16, 2, 14, 8, 18, 4, 12

// 順序付け - 昇順
IEnumerable<int> sortedNumbers = numbers.OrderBy(n => n);
Console.WriteLine("昇順: " + string.Join(", ", sortedNumbers));
// 出力: 昇順: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10

// 順序付け - 降順
IEnumerable<int> descendingNumbers = numbers.OrderByDescending(n => n);
Console.WriteLine("降順: " + string.Join(", ", descendingNumbers));
// 出力: 降順: 10, 9, 8, 7, 6, 5, 4, 3, 2, 1

// 最初の要素
int firstNumber = numbers.First();
Console.WriteLine("最初の要素: " + firstNumber);
// 出力: 最初の要素: 5

// 条件を満たす最初の要素
int firstEven = numbers.First(n => n % 2 == 0);
Console.WriteLine("最初の偶数: " + firstEven);
// 出力: 最初の偶数: 10

// 要素が存在しない場合はデフォルト値を返す
int firstNegative = numbers.FirstOrDefault(n => n < 0);
Console.WriteLine("最初の負数（なければデフォルト値）: " + firstNegative);
// 出力: 最初の負数（なければデフォルト値）: 0

// 集計 - 合計
int sum = numbers.Sum();
Console.WriteLine("合計: " + sum);
// 出力: 合計: 55

// 集計 - 平均
```

```
double average = numbers.Average();
Console.WriteLine("平均: " + average);
// 出力: 平均: 5.5

// 集計 - 最大値
int max = numbers.Max();
Console.WriteLine("最大値: " + max);
// 出力: 最大値: 10

// 集計 - 最小値
int min = numbers.Min();
Console.WriteLine("最小値: " + min);
// 出力: 最小値: 1

// 条件を満たす要素数
int countOfEvens = numbers.Count(n => n % 2 == 0);
Console.WriteLine("偶数の数: " + countOfEvens);
// 出力: 偶数の数: 5

// 条件を満たす要素があるかどうか
bool hasGreaterThan9 = numbers.Any(n => n > 9);
Console.WriteLine("9より大きい数があるか: " + hasGreaterThan9);
// 出力: 9より大きい数があるか: True

// すべての要素が条件を満たすかどうか
bool allPositive = numbers.All(n => n > 0);
Console.WriteLine("すべて正の数か: " + allPositive);
// 出力: すべて正の数か: True
```

クエリ構文

LINQには、メソッド構文とクエリ構文の2つの書き方があります。上記の例はメソッド構文ですが、SQL風のクエリ構文も使用できます：

```
// クエリ構文の例
var evenNumbersQuery = from n in numbers
                       where n % 2 == 0
                       select n;

Console.WriteLine("クエリ構文による偶数: " + string.Join(", ", evenNumbersQuery));
// 出力: クエリ構文による偶数: 10, 8, 4, 2, 6

var sortedQuery = from n in numbers
                  orderby n
                  select n;

Console.WriteLine("クエリ構文による昇順: " + string.Join(", ", sortedQuery));
// 出力: クエリ構文による昇順: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10

// より複雑なクエリ
var complexQuery = from n in numbers
                   where n > 3
                   orderby n descending
                   select n * n;
```

```
Console.WriteLine("複雑なクエリ結果: " + string.Join(", ", complexQuery));
// 出力: 複雑なクエリ結果: 100, 81, 64, 49, 36, 25
```

複雑なオブジェクトに対するLINQ

LINQは、単純なコレクションだけでなく、複雑なオブジェクトのコレクションに対しても使用できます：

```
// サンプルクラス
public class Student
{
    public int Id { get; set; }
    public string Name { get; set; }
    public int Age { get; set; }
    public List<string> Courses { get; set; }
}

// サンプルデータ
List<Student> students = new List<Student>
{
    new Student { Id = 1, Name = "田中太郎", Age = 20, Courses = new List<string> { "数学", "物理", "化学" } },
    new Student { Id = 2, Name = "鈴木花子", Age = 22, Courses = new List<string> { "文学", "歴史", "哲学" } },
    new Student { Id = 3, Name = "佐藤健", Age = 19, Courses = new List<string> { "経済学", "統計学" } },
    new Student { Id = 4, Name = "伊藤誠", Age = 21, Courses = new List<string> { "コンピュータサイエンス", "数学", "電子工学" } },
    new Student { Id = 5, Name = "山本美咲", Age = 20, Courses = new List<string> { "生物学", "化学", "物理" } }
};

// 年齢が20以上の学生の名前を取得
var adultStudentNames = students.Where(s => s.Age >= 20).Select(s => s.Name);
Console.WriteLine("20歳以上の学生: " + string.Join(", ", adultStudentNames));
// 出力: 20歳以上の学生: 田中太郎, 鈴木花子, 伊藤誠, 山本美咲

// 年齢順に並べ替えて学生のIDと名前を取得
var sortedStudents = students.OrderBy(s => s.Age).Select(s => new { s.Id, s.Name });
foreach (var student in sortedStudents)
{
    Console.WriteLine($"ID: {student.Id}, 名前: {student.Name}");
}
// 出力:
// ID: 3, 名前: 佐藤健
// ID: 1, 名前: 田中太郎
// ID: 5, 名前: 山本美咲
// ID: 4, 名前: 伊藤誠
// ID: 2, 名前: 鈴木花子

// 特定の科目を履修している学生を検索
var mathStudents = students.Where(s => s.Courses.Contains("数学")).Select(s => s.Name);
Console.WriteLine("数学を履修している学生: " + string.Join(", ", mathStudents));
// 出力: 数学を履修している学生: 田中太郎, 伊藤誠
```

```
// 科目ごとの学生リストを作成（グループ化）
var courseGroups = students.SelectMany(s => s.Courses, (student, course) => new {
StudentName = student.Name, Course = course })
                           .GroupBy(x => x.Course)
                           .Select(g => new { Course = g.Key, Students = g.Select(x =>
x.StudentName).ToList() });

foreach (var group in courseGroups)
{
    Console.WriteLine($"科目: {group.Course}");
    Console.WriteLine($"  受講者: {string.Join(", ", group.Students)}");
}
// 出力:
// 科目: 数学
//   受講者: 田中太郎, 伊藤誠
// 科目: 物理
//   受講者: 田中太郎, 山本美咲
// ...（以下省略）
```

> **プロジェクト事例**: あるEコマースシステムでは、大量の注文データから様々な分析情報を抽出する必要がありました。LINQを使用することで、時間帯別の注文数、商品カテゴリ別の売上、地域別の顧客分布など、様々な集計を簡潔かつ読みやすいコードで実装できました。特に、複数のデータソースを結合（join）して分析する場合に、SQLのような直感的な構文で複雑な処理を記述できる点が大きなメリットでした。

遅延評価と即時評価

LINQは基本的に「遅延評価」（lazy evaluation）を採用しています。これは、クエリが実際に結果を必要とするまで実行されないことを意味します：

```
// 遅延評価の例
List<int> numbers = new List<int> { 1, 2, 3, 4, 5 };

// クエリを定義（この時点では実行されない）
var evenNumbers = numbers.Where(n => n % 2 == 0);

// リストに要素を追加
numbers.Add(6);
numbers.Add(7);
numbers.Add(8);

// クエリを使用（この時点で実行される）
Console.WriteLine("偶数: " + string.Join(", ", evenNumbers));
// 出力: 偶数: 2, 4, 6, 8 （追加された6と8も含まれる）

// 即時評価の例
List<int> numbers2 = new List<int> { 1, 2, 3, 4, 5 };

// ToList()を使って即時評価
var evenNumbersList = numbers2.Where(n => n % 2 == 0).ToList();

// リストに要素を追加
numbers2.Add(6);
numbers2.Add(7);
numbers2.Add(8);
```

```
// 結果を使用
Console.WriteLine("偶数（即時評価）: " + string.Join(", ", evenNumbersList));
// 出力: 偶数（即時評価）: 2, 4 （追加された6と8は含まれない）
```

即時評価を強制するには、`ToList()`、`ToArray()`、`ToDictionary()`などのメソッドを使用します。

> **ベテランの知恵袋**: 遅延評価は必要になるまで計算を遅らせるため、パフォーマンス上のメリットがあります。特に大きなデータセットに対して複数のフィルタリング操作を行う場合、中間結果のために余分なメモリを使用しません。ただし、同じクエリを複数回実行する場合や、元のデータソースが変更される可能性がある場合は、`ToList()`などで即時評価を行い、結果を固定するとよいでしょう。

章末チェックリスト

- [ ] 配列の宣言、初期化、アクセスの方法を理解している
- [ ] 多次元配列とジャグ配列の違いを説明できる
- [ ] Listの基本的な操作（追加、削除、アクセスなど）ができる
- [ ] Dictionary<TKey, TValue>を使ってキーと値のペアを管理できる
- [ ] HashSetの特徴と集合演算を理解している
- [ ] QueueとStackの違いと使用場面を説明できる
- [ ] ジェネリッククラスとジェネリックメソッドを定義できる
- [ ] LINQの基本的なメソッド（Where, Select, OrderBy, Groupなど）を使いこなせる
- [ ] LINQのメソッド構文とクエリ構文の両方を書ける
- [ ] 遅延評価と即時評価の違いを理解している

まとめと次のステップ

この章では、C#のコレクションとLINQについて学びました。配列や様々なコレクションクラスを使って複数のデータを効率的に管理する方法と、LINQを使ってそれらのデータを柔軟に操作する方法を理解しました。これらの知識は、実際のアプリケーション開発において非常に重要です。

次の章では、例外処理とデバッグについて学びます。プログラムの実行中に発生する問題をどのように検出し、適切に対処するかを理解することで、より堅牢なプログラムを作成できるようになります。

第7章: 例外処理とデバッグ

プログラムの実行中には、予期しないエラーが発生することがあります。これらのエラーを適切に処理し、プログラムのクラッシュを防ぐために、C#には例外処理の仕組みがあります。また、問題を特定し修正するためのデバッグ技術も重要です。

例外の基本

例外（Exception）は、プログラムの実行中に発生するエラーや異常な状態を表すオブジェクトです。例外が発生すると、通常のプログラムの流れが中断され、例外処理のコードが実行されます。

```
// 例外が発生する可能性のあるコード
int[] numbers = { 1, 2, 3 };
try
```

```
{
    // 範囲外のインデックスにアクセス（例外が発生する）
    Console.WriteLine(numbers[10]);
}
catch (IndexOutOfRangeException ex)
{
    // 例外が発生した場合の処理
    Console.WriteLine("配列の範囲外にアクセスしました: " + ex.Message);
}
```

try-catch-finally

C#での例外処理は、主に `try` 、 `catch` 、 `finally` ブロックを使用します：

```
try
{
    // 例外が発生する可能性のあるコード
    Console.Write("数値を入力してください: ");
    string input = Console.ReadLine();
    int number = int.Parse(input);
    Console.WriteLine($"入力された数値の2倍: {number * 2}");
}
catch (FormatException ex)
{
    // 数値形式が不正な場合
    Console.WriteLine("数値形式が不正です: " + ex.Message);
}
catch (OverflowException ex)
{
    // 数値が大きすぎる場合
    Console.WriteLine("数値が大きすぎます: " + ex.Message);
}
catch (Exception ex)
{
    // その他の例外
    Console.WriteLine("予期しないエラーが発生しました: " + ex.Message);
}
finally
{
    // 例外の発生有無にかかわらず必ず実行されるコード
    Console.WriteLine("処理を終了します");
}
```

`finally` ブロックは、例外が発生してもしなくても必ず実行されるコードを記述するのに使用します。リソースの解放など、確実に実行したい処理を記述します。

例外のスロー

自分で例外をスローすることもできます：

```
public static void CheckAge(int age)
{
    if (age < 0)
    {
```

```
            throw new ArgumentException("年齢は0以上である必要があります");
        }

        if (age < 20)
        {
            throw new InvalidOperationException("20歳未満は利用できません");
        }

        Console.WriteLine("年齢確認が完了しました");
    }

    // 使用例
    try
    {
        CheckAge(-5);
    }
    catch (ArgumentException ex)
    {
        Console.WriteLine("引数エラー: " + ex.Message);
    }
    catch (InvalidOperationException ex)
    {
        Console.WriteLine("操作エラー: " + ex.Message);
    }
```

カスタム例外

特定のアプリケーションに固有の例外をサポートするために、独自の例外クラスを作成することもできます：

```
    // カスタム例外の定義
    public class UserNotFoundException : Exception
    {
        public string Username { get; }

        public UserNotFoundException(string username)
            : base($"ユーザー '{username}' が見つかりませんでした")
        {
            Username = username;
        }

        public UserNotFoundException(string username, Exception innerException)
            : base($"ユーザー '{username}' が見つかりませんでした", innerException)
        {
            Username = username;
        }
    }

    // カスタム例外の使用
    public class UserService
    {
        private readonly List<string> _registeredUsers = new List<string> { "admin",
    "user1", "user2" };

        public bool ValidateUser(string username)
```

```
    {
        if (string.IsNullOrEmpty(username))
        {
            throw new ArgumentNullException(nameof(username), "ユーザー名が指定されていません");
        }

        if (!_registeredUsers.Contains(username))
        {
            throw new UserNotFoundException(username);
        }

        return true;
    }
}

// 使用例
UserService service = new UserService();

try
{
    service.ValidateUser("guest");
}
catch (UserNotFoundException ex)
{
    Console.WriteLine(ex.Message);
    Console.WriteLine($"問題のユーザー名: {ex.Username}");
}
catch (ArgumentNullException ex)
{
    Console.WriteLine("引数エラー: " + ex.Message);
}
```

> **ベテランの知恵袋**: カスタム例外を作成する際は、次のガイドラインに従うと良いでしょう：
>
> 1. 例外クラス名は「Exception」で終わるようにする
> 2. 直接または間接的に `Exception` クラスを継承する
> 3. 最低でも3つのコンストラクタ（引数なし、メッセージ付き、内部例外付き）を実装する
> 4. シリアライズ可能にする（`[Serializable]` 属性を付ける）
> 5. 例外に関連する追加情報を格納するためのプロパティを提供する

try-catch-whenの条件付きキャッチ

C# 6.0からは、例外フィルターを使用して特定の条件下でのみ例外をキャッチできるようになりました：

```
try
{
    ProcessData("data.txt");
}
catch (IOException ex) when (ex.Message.Contains("disk"))
{
    Console.WriteLine("ディスクエラーが発生しました: " + ex.Message);
}
catch (IOException ex) when (ex.Message.Contains("network"))
{
```

```
    Console.WriteLine("ネットワークエラーが発生しました: " + ex.Message);
}
catch (IOException ex)
{
    Console.WriteLine("一般的なIOエラーが発生しました: " + ex.Message);
}

// 例外のロギングと再スロー
try
{
    ProcessData("data.txt");
}
catch (Exception ex) when (LogException(ex))
{
    // このブロックは実行されない（LogExceptionはfalseを返す）
}

bool LogException(Exception ex)
{
    Console.WriteLine($"例外がログに記録されました: {ex.Message}");
    return false;  // falseを返して例外をキャッチしない
}
```

using文とIDisposable

リソースを確実に解放するための`using`文も、例外処理と密接に関連しています：

```
// ファイルを開いて操作する例
public void WriteToFile(string path, string content)
{
    // usingブロックを抜けると自動的にStreamWriterが破棄される
    using (StreamWriter writer = new StreamWriter(path))
    {
        writer.WriteLine(content);
    }  // ここでwriter.Dispose()が自動的に呼ばれる
}

// C# 8.0からのusingステートメント
public void WriteToFile2(string path, string content)
{
    using StreamWriter writer = new StreamWriter(path);
    writer.WriteLine(content);
    // メソッドの終了時にwriter.Dispose()が自動的に呼ばれる
}

// 複数のリソースを扱う例
public void CopyFile(string sourcePath, string destinationPath)
{
    using (StreamReader reader = new StreamReader(sourcePath))
    using (StreamWriter writer = new StreamWriter(destinationPath))
    {
        string line;
        while ((line = reader.ReadLine()) != null)
        {
            writer.WriteLine(line);
        }
```

```
    }  // readerとwriterの両方がここで破棄される
}
```

`using`文は、`IDisposable`インターフェイスを実装するオブジェクトと共に使用され、例外が発生してもリソースが確実に解放されるようにします。

> **プロジェクト事例**: あるファイル処理システムでは、大量のファイルを並行して処理する必要がありました。各ファイルの処理中に様々な例外が発生する可能性があり、それぞれに適切に対応する必要がありました。try-catch-finallyブロックを使った例外処理と、usingステートメントを使ったリソース管理の組み合わせにより、一部のファイルで問題が発生しても処理を継続し、すべてのファイルハンドルが確実に閉じられるようになりました。これにより、システムの安定性と回復力が大幅に向上しました。

デバッグの基本

プログラムの問題を特定して修正するためのデバッグ技術は、開発者にとって不可欠なスキルです。Visual Studioには、強力なデバッグ機能が組み込まれています。

ブレークポイントの設定

ブレークポイントを設定すると、プログラムの実行が一時停止し、その時点での変数の値などを確認できます：

1. コードの左側のマージンをクリックする
2. または、行にカーソルを置いて F9 キーを押す
3. コード行の左側に赤い丸（ブレークポイント）が表示される

デバッグ実行

デバッグモードでプログラムを実行するには：

1. F5 キーを押す（デバッグ開始）
2. プログラムはブレークポイントで停止する

デバッグウィンドウ

デバッグ中に使用できる主なウィンドウ：

- **ローカル**: 現在のスコープ内の変数とその値
- **ウォッチ**: 監視したい特定の変数や式
- **イミディエイト**: 式の評価やコマンドの実行
- **コールスタック**: 現在の実行位置に至るメソッド呼び出しの履歴

ステップ実行

デバッグ中に使用できる主なコマンド：

- **F10（ステップオーバー）**: 現在の行を実行し、次の行で停止（メソッド呼び出しの中には入らない）
- **F11（ステップイン）**: メソッド呼び出しの中に入って実行
- **Shift + F11（ステップアウト）**: 現在のメソッドを最後まで実行し、呼び出し元に戻る
- **F5（続行）**: 次のブレークポイントまで実行を続ける

デバッグ用のコード

デバッグを支援するコードを記述することもできます：

```
// Debug.WriteLine - デバッグ実行時のみ出力
Debug.WriteLine("デバッグ情報: 値 = " + value);

// Debug.Assert - 条件が偽の場合にブレーク
Debug.Assert(value > 0, "値は正でなければなりません");

// Trace.WriteLine - デバッグとリリース両方で出力
Trace.WriteLine("情報: 処理を開始します");

// 条件付きコンパイル
#if DEBUG
    Console.WriteLine("これはデバッグビルドでのみ実行されます");
#endif
```

> **若手の疑問解決**: 「デバッグしやすいコードを書くコツは？」
>
> 1. **メソッドを小さく保つ**: 各メソッドは1つのことだけを行うようにする
> 2. **適切な命名**: 変数やメソッドの名前から目的がわかるようにする
> 3. **早期リターン**: 異常条件を早めにチェックし、異常があれば早めに関数から抜ける
> 4. **ガード節**: メソッドの先頭で引数の検証を行う
> 5. **ログ出力**: 重要なポイントでログを出力する
> 6. **状態を明示的に管理**: 不変条件を保持し、副作用を最小限に抑える
> 7. **例外処理**: 例外は適切な粒度でキャッチし、具体的なメッセージを含める

パフォーマンスの最適化

デバッグだけでなく、パフォーマンス問題の特定と解決も重要なスキルです。

パフォーマンス測定

コードの実行時間を測定する簡単な方法：

```
using System.Diagnostics;

public void MeasurePerformance()
{
    // 実行時間を測定するストップウォッチ
    Stopwatch stopwatch = new Stopwatch();
    stopwatch.Start();

    // 測定対象のコード
    for (int i = 0; i < 1000000; i++)
    {
        // 何らかの処理
        Math.Sqrt(i);
    }

    stopwatch.Stop();
```

```
    // 経過時間を表示
    Console.WriteLine($"処理時間: {stopwatch.ElapsedMilliseconds}ミリ秒");
    Console.WriteLine($"処理時間（詳細）: {stopwatch.Elapsed}");
}
```

一般的なパフォーマンス最適化テクニック

```
// 1. 文字列連結にはStringBuilderを使用する
StringBuilder sb = new StringBuilder();
for (int i = 0; i < 1000; i++)
{
    sb.Append(i.ToString());
    sb.Append(", ");
}
string result = sb.ToString();

// 2. コレクションの容量を事前に確保する
List<int> numbers = new List<int>(1000);  // 1000要素分の容量を確保

// 3. 不要なオブジェクト生成を避ける
// 悪い例
for (int i = 0; i < 1000; i++)
{
    string temp = "値: " + i;  // 毎回新しい文字列オブジェクトが生成される
    Process(temp);
}

// 良い例
for (int i = 0; i < 1000; i++)
{
    Process("値: " + i);  // 文字列は一時的にのみ使用
}

// 4. LINQクエリの実行タイミングに注意する
IEnumerable<int> evenNumbers = numbers.Where(n => n % 2 == 0);
// numbers.Add(6);  // evenNumbersを使用するたびに、このフィルタリングも再適用される

// 結果を固定するなら
List<int> evenNumbersList = numbers.Where(n => n % 2 == 0).ToList();
numbers.Add(6);  // evenNumbersListには影響しない
```

> **ベテランの知恵袋**: パフォーマンスの最適化は、測定に基づいて行うことが重要です。「早まった最適化は諸悪の根源」という格言があるように、実際にボトルネックとなっている部分を特定してから最適化するべきです。また、読みやすさと保守性を犠牲にするような最適化は、それによるパフォーマンス向上が十分に大きい場合にのみ検討してください。

ロギング

問題のトラブルシューティングや監視のために、ログを記録することは重要です：

```
// 単純なコンソールログ
public void SimpleLogging()
{
    try
```

```
    {
        Console.WriteLine("処理を開始します");
        // 何らかの処理
        int result = 10 / 0;  // 例外が発生する
    }
    catch (Exception ex)
    {
        Console.WriteLine($"エラーが発生しました: {ex.Message}");
        Console.WriteLine($"スタックトレース: {ex.StackTrace}");
    }
    finally
    {
        Console.WriteLine("処理を終了します");
    }
}

// ファイルへのログ出力
public void FileLogging()
{
    string logPath = "application.log";

    try
    {
        File.AppendAllText(logPath, $"{DateTime.Now}: 処理を開始します\n");
        // 何らかの処理
    }
    catch (Exception ex)
    {
        File.AppendAllText(logPath, $"{DateTime.Now}: エラー - {ex.Message}\n");
        File.AppendAllText(logPath, $"{DateTime.Now}: スタックトレース -
{ex.StackTrace}\n");
    }
    finally
    {
        File.AppendAllText(logPath, $"{DateTime.Now}: 処理を終了します\n");
    }
}
```

実際のアプリケーションでは、NLogやlog4netなどのロギングライブラリを使用することが一般的です。

> **失敗から学ぶ**: あるプロジェクトでは、デプロイ後のアプリケーションで断続的に問題が発生していましたが、デバッガを直接接続できない環境でした。問題の追跡と対応のためのロギングが不十分だったため、問題の根本原因を特定するのに非常に時間がかかりました。この経験から、本番環境用のログは以下の点を考慮して設計するようになりました：
>
> 1. ログレベル（情報、警告、エラーなど）を適切に使い分ける
> 2. コンテキスト情報（ユーザーID、セッションID、関連するデータなど）を含める
> 3. 例外の詳細情報とスタックトレースを記録する
> 4. ログの出力先を柔軟に設定できるようにする（ファイル、データベース、監視システムなど）
> 5. 機密情報がログに含まれないよう注意する

章末チェックリスト

- [ ] try-catch-finallyブロックを使って例外を適切に処理できる

- [ ] 適切な例外をスローしてエラー状態を伝えることができる
- [ ] カスタム例外クラスを作成できる
- [ ] リソースの解放を確実に行うためにusingステートメントを使用できる
- [ ] ブレークポイントを設定してプログラムを一時停止できる
- [ ] デバッグウィンドウを使って変数の値や実行状態を確認できる
- [ ] ステップ実行を使ってコードの流れを追うことができる
- [ ] パフォーマンスの測定と最適化の基本テクニックを理解している
- [ ] ログを記録してトラブルシューティングを支援できる
- [ ] デバッグしやすいコードを書くための原則を理解している

まとめと次のステップ

この章では、例外処理とデバッグの重要性と基本テクニックについて学びました。予期しないエラーを適切に処理し、問題を効率的に特定・修正するスキルは、プロフェッショナルな開発者にとって不可欠です。

次の章では、ファイルとデータの取り扱いについて学びます。ファイルの読み書きやデータの永続化など、実際のアプリケーションでよく必要とされる機能を実装する方法を理解していきましょう。

第8章: ファイルとデータの取り扱い

実用的なプログラムでは、ファイルからデータを読み込んだり、ファイルにデータを保存したりする機能が必要になることが多いです。この章では、C#でファイルやデータを扱う方法を学びます。

ファイル操作の基本

C#には、ファイル操作を行うための様々なクラスが用意されています。まずは、基本的なファイル操作から見ていきましょう。

File クラスとFileInfo クラス

`File` クラスと `FileInfo` クラスは、ファイルの作成、コピー、削除、移動、および開くための静的メソッドを提供します：

```
// Fileクラスを使用した基本的なファイル操作
public void BasicFileOperations()
{
    string path = "sample.txt";

    // ファイルへの書き込み
    File.WriteAllText(path, "これはサンプルファイルです。");

    // 既存のファイルに追記
    File.AppendAllText(path, "\n新しい行を追加します。");

    // ファイルから読み込み
    string content = File.ReadAllText(path);
    Console.WriteLine("ファイルの内容:");
    Console.WriteLine(content);
```

```
    // ファイルを行ごとに読み込み
    string[] lines = File.ReadAllLines(path);
    Console.WriteLine("行数: " + lines.Length);

    // ファイルのコピー
    File.Copy(path, "sample_copy.txt", overwrite: true);

    // ファイルの移動
    File.Move("sample_copy.txt", "sample_moved.txt");

    // ファイルの存在確認
    bool exists = File.Exists("sample_moved.txt");
    Console.WriteLine("ファイルは存在しますか？ " + exists);

    // ファイルの属性取得
    DateTime creationTime = File.GetCreationTime(path);
    DateTime lastAccess = File.GetLastAccessTime(path);
    long fileSize = new FileInfo(path).Length;

    Console.WriteLine($"作成日時: {creationTime}");
    Console.WriteLine($"最終アクセス日時: {lastAccess}");
    Console.WriteLine($"ファイルサイズ: {fileSize}バイト");

    // ファイルの削除
    File.Delete("sample_moved.txt");
}
```

Directory クラスとDirectoryInfo クラス

`Directory` クラスと `DirectoryInfo` クラスは、ディレクトリの作成、移動、列挙などの操作を提供します：

```
// Directoryクラスを使用した基本的なディレクトリ操作
public void BasicDirectoryOperations()
{
    // ディレクトリの作成
    Directory.CreateDirectory("MyFolder");

    // サブディレクトリの作成
    Directory.CreateDirectory("MyFolder/SubFolder");

    // ファイルをディレクトリに作成
    File.WriteAllText("MyFolder/file1.txt", "ファイル1");
    File.WriteAllText("MyFolder/file2.txt", "ファイル2");
    File.WriteAllText("MyFolder/SubFolder/file3.txt", "ファイル3");

    // ディレクトリ内のファイル一覧の取得
    string[] files = Directory.GetFiles("MyFolder");
    Console.WriteLine("MyFolderのファイル:");
    foreach (string file in files)
    {
        Console.WriteLine("  " + Path.GetFileName(file));
    }

    // ディレクトリ内のサブディレクトリ一覧の取得
```

```
    string[] directories = Directory.GetDirectories("MyFolder");
    Console.WriteLine("MyFolderのサブディレクトリ:");
    foreach (string dir in directories)
    {
        Console.WriteLine("  " + Path.GetFileName(dir));
    }

    // 再帰的にすべてのファイルを取得
    string[] allFiles = Directory.GetFiles("MyFolder", "*",
SearchOption.AllDirectories);
    Console.WriteLine("すべてのファイル（サブディレクトリを含む）:");
    foreach (string file in allFiles)
    {
        Console.WriteLine("  " + file);
    }

    // ディレクトリの存在確認
    bool exists = Directory.Exists("MyFolder/SubFolder");
    Console.WriteLine("サブディレクトリは存在しますか？ " + exists);

    // ディレクトリの移動
    Directory.Move("MyFolder/SubFolder", "MyFolder/RenamedFolder");

    // ディレクトリの削除（空でない場合は再帰的に削除）
    Directory.Delete("MyFolder", recursive: true);
}
```

Path クラス

`Path` クラスは、パス文字列の操作や情報の取得に役立ちます：

```
// Pathクラスを使用したパス操作
public void PathOperations()
{
    string filePath = "C:\\MyFolder\\SubFolder\\myfile.txt";

    // パスの部分を取得
    string directoryName = Path.GetDirectoryName(filePath);  // C:\MyFolder\SubFolder
    string fileName = Path.GetFileName(filePath);            // myfile.txt
    string fileNameWithoutExt = Path.GetFileNameWithoutExtension(filePath);  // myfile
    string extension = Path.GetExtension(filePath);          // .txt

    Console.WriteLine($"ディレクトリ名: {directoryName}");
    Console.WriteLine($"ファイル名: {fileName}");
    Console.WriteLine($"拡張子を除くファイル名: {fileNameWithoutExt}");
    Console.WriteLine($"拡張子: {extension}");

    // パスの結合
    string combined = Path.Combine("C:\\MyFolder", "SubFolder", "myfile.txt");
    Console.WriteLine($"結合されたパス: {combined}");

    // 一時ファイルのパスを取得
    string tempFile = Path.GetTempFileName();
    Console.WriteLine($"一時ファイル: {tempFile}");

    // 絶対パスに変換
```

```
    string absolutePath = Path.GetFullPath("relative/path/file.txt");
    Console.WriteLine($"絶対パス: {absolutePath}");

    // パスの正規化
    string normalizedPath = Path.GetFullPath("C:\\MyFolder\\..\\MyFolder\\myfile.txt");
    Console.WriteLine($"正規化されたパス: {normalizedPath}");  // C:\MyFolder\myfile.txt

    // 無効なパス文字をチェック
    char[] invalidChars = Path.GetInvalidPathChars();
    Console.WriteLine($"無効なパス文字数: {invalidChars.Length}");
}
```

> **ベテランの知恵袋**: ファイルパスを操作する際は、常に`Path`クラスを使用しましょう。文字列結合（`+`演算子）でパスを組み立てると、プラットフォーム間の違い（Windowsは「\」、MacやLinuxは「/」をパス区切り文字として使用）やパス区切り文字の重複などの問題が発生する可能性があります。`Path.Combine`を使えば、これらの問題を避けることができます。

ストリームを使ったファイル操作

より柔軟なファイル操作を行うには、ストリームを使用します。ストリームは、データを連続した流れとして扱い、大きなファイルでも効率的に処理できます。

FileStream

`FileStream`は、ファイルからのバイト単位の読み取りや書き込みを行うためのクラスです：

```
// FileStreamを使用したファイル操作
public void FileStreamOperations()
{
    string filePath = "binary_data.bin";

    // ファイルへの書き込み
    using (FileStream fs = new FileStream(filePath, FileMode.Create))
    {
        // バイトデータの作成
        byte[] data = new byte[100];
        for (int i = 0; i < 100; i++)
        {
            data[i] = (byte)i;
        }

        // データの書き込み
        fs.Write(data, 0, data.Length);
    }

    // ファイルからの読み取り
    using (FileStream fs = new FileStream(filePath, FileMode.Open))
    {
        byte[] buffer = new byte[10];
        int bytesRead;
        int position = 0;

        Console.WriteLine("ファイルの内容（最初の10バイトごと）:");

        while ((bytesRead = fs.Read(buffer, 0, buffer.Length)) > 0)
```

```
        {
            Console.Write($"位置 {position}: ");
            for (int i = 0; i < bytesRead; i++)
            {
                Console.Write($"{buffer[i]} ");
            }
            Console.WriteLine();

            position += bytesRead;
        }

        // ファイルサイズの取得
        long fileSize = fs.Length;
        Console.WriteLine($"ファイルサイズ: {fileSize}バイト");

        // ファイル内の位置を移動
        fs.Seek(0, SeekOrigin.Begin);  // 先頭に移動

        // 先頭から10バイトだけ読み取り
        byte[] firstTenBytes = new byte[10];
        fs.Read(firstTenBytes, 0, 10);

        Console.WriteLine("先頭の10バイト:");
        foreach (byte b in firstTenBytes)
        {
            Console.Write($"{b} ");
        }
        Console.WriteLine();
    }
}
```

StreamReader と StreamWriter

テキストファイルを扱う場合は、`StreamReader` と `StreamWriter` を使用すると便利です：

```
// StreamReaderとStreamWriterを使用したテキストファイル操作
public void TextStreamOperations()
{
    string filePath = "text_data.txt";

    // テキストファイルへの書き込み
    using (StreamWriter writer = new StreamWriter(filePath))
    {
        writer.WriteLine("これは1行目です。");
        writer.WriteLine("これは2行目です。");
        writer.WriteLine("これは3行目です。");

        // 改行なしでテキストを書き込む
        writer.Write("改行なしのテキスト1");
        writer.Write("改行なしのテキスト2");

        // 改行を追加
        writer.WriteLine();

        // 書式指定文字列の使用
        writer.WriteLine("名前: {0}, 年齢: {1}", "山田太郎", 30);
```

```
    }

    // テキストファイルからの読み取り
    using (StreamReader reader = new StreamReader(filePath))
    {
        // ファイル全体を一度に読み取る
        // string content = reader.ReadToEnd();
        // Console.WriteLine("ファイルの内容（一括）:");
        // Console.WriteLine(content);

        // 読み取り位置を先頭に戻す
        // reader.BaseStream.Seek(0, SeekOrigin.Begin);
        // reader.DiscardBufferedData();

        // ファイルを1行ずつ読み取る
        Console.WriteLine("ファイルの内容（1行ずつ）:");

        string line;
        int lineNumber = 1;

        while ((line = reader.ReadLine()) != null)
        {
            Console.WriteLine($"行 {lineNumber}: {line}");
            lineNumber++;
        }
    }

    // ファイルに追記
    using (StreamWriter writer = new StreamWriter(filePath, append: true))
    {
        writer.WriteLine("この行は追記されました。");
    }

    // 追記後の内容を確認
    Console.WriteLine("\n追記後のファイル内容:");
    Console.WriteLine(File.ReadAllText(filePath));
}
```

BinaryReader と BinaryWriter

バイナリデータを扱う場合は、`BinaryReader` と `BinaryWriter` を使用すると便利です：

```
// BinaryReaderとBinaryWriterを使用したバイナリファイル操作
public void BinaryStreamOperations()
{
    string filePath = "binary_data.dat";

    // バイナリファイルへの書き込み
    using (BinaryWriter writer = new BinaryWriter(File.Open(filePath,
FileMode.Create)))
    {
        // 様々な型のデータを書き込む
        writer.Write(42);        // int
        writer.Write(3.14159);   // double
        writer.Write("こんにちは");  // string
        writer.Write(true);      // bool
```

```
            writer.Write((byte)255); // byte

            // 配列の書き込み
            int[] numbers = { 10, 20, 30, 40, 50 };
            writer.Write(numbers.Length);  // 配列の長さを最初に書き込む
            foreach (int number in numbers)
            {
                writer.Write(number);
            }
        }

        // バイナリファイルからの読み取り
        using (BinaryReader reader = new BinaryReader(File.Open(filePath, FileMode.Open)))
        {
            // データを順番に読み取る（書き込んだ順序と同じ順序で読み取る必要がある）
            int intValue = reader.ReadInt32();
            double doubleValue = reader.ReadDouble();
            string stringValue = reader.ReadString();
            bool boolValue = reader.ReadBoolean();
            byte byteValue = reader.ReadByte();

            Console.WriteLine($"読み取った値: {intValue}, {doubleValue}, {stringValue},
{boolValue}, {byteValue}");

            // 配列の読み取り
            int arrayLength = reader.ReadInt32();
            int[] readNumbers = new int[arrayLength];

            for (int i = 0; i < arrayLength; i++)
            {
                readNumbers[i] = reader.ReadInt32();
            }

            Console.WriteLine("読み取った配列: " + string.Join(", ", readNumbers));
        }
    }
```

> **若手の疑問解決**: 「テキストファイルとバイナリファイルはどう使い分ければよいですか？」
>
> テキストファイルは人間が読める形式でデータを保存するため、設定ファイル、ログファイル、簡単なデータ保存などに適しています。一方、バイナリファイルは、画像、音声、動画などの非テキストデータや、大量の数値データを効率的に保存したい場合に適しています。バイナリ形式はファイルサイズが小さく読み書きが高速ですが、特殊なツールがないと内容を直接確認できないというデメリットがあります。アプリケーションの要件に応じて適切な形式を選びましょう。

シリアライゼーション

オブジェクトを保存したり転送したりするために、シリアライゼーション（直列化）という技術があります。シリアライゼーションを使用すると、複雑なオブジェクトグラフを簡単にファイルに保存したり、ネットワーク経由で送信したりできます。

バイナリシリアライゼーション

```
using System.Runtime.Serialization.Formatters.Binary;
```

```
// シリアライズ可能なクラス
[Serializable]
public class Person
{
    public string Name { get; set; }
    public int Age { get; set; }
    public DateTime BirthDate { get; set; }
    public List<string> Hobbies { get; set; }

    // シリアライズしたくないフィールド
    [NonSerialized]
    private string secretCode;

    public Person(string name, int age)
    {
        Name = name;
        Age = age;
        secretCode = "12345";
    }

    public override string ToString()
    {
        return $"名前: {Name}, 年齢: {Age}, 誕生日: {BirthDate.ToShortDateString()}, " +
               $"趣味: {string.Join(", ", Hobbies ?? new List<string>())}";
    }
}

// バイナリシリアライゼーションの例
public void BinarySerializationExample()
{
    string filePath = "person.dat";

    // オブジェクトの作成
    Person person = new Person("山田太郎", 30)
    {
        BirthDate = new DateTime(1990, 5, 15),
        Hobbies = new List<string> { "読書", "旅行", "プログラミング" }
    };

    Console.WriteLine("シリアライズ前のデータ:");
    Console.WriteLine(person);

    // オブジェクトのシリアライズ
    using (FileStream fs = new FileStream(filePath, FileMode.Create))
    {
        BinaryFormatter formatter = new BinaryFormatter();
        formatter.Serialize(fs, person);
    }

    Console.WriteLine($"オブジェクトをファイル {filePath} にシリアライズしました");

    // オブジェクトのデシリアライズ
    Person deserializedPerson;
    using (FileStream fs = new FileStream(filePath, FileMode.Open))
    {
        BinaryFormatter formatter = new BinaryFormatter();
        deserializedPerson = (Person)formatter.Deserialize(fs);
```

```
    }

    Console.WriteLine("デシリアライズしたデータ:");
    Console.WriteLine(deserializedPerson);
}
```

> **ベテランの知恵袋**: `BinaryFormatter`によるバイナリシリアライゼーションは、.NET Coreで非推奨（deprecated）になりました。これはセキュリティ上の問題があるためです。新しいコードでは、JSON、XML、またはProtobuf（Google Protocol Buffers）などの代替シリアライゼーション形式を使用することをお勧めします。

JSONシリアライゼーション

JSONは、人間が読みやすく、幅広いプラットフォームでサポートされているデータ形式です：

```
using System.Text.Json;

// JSONシリアライゼーションの例
public void JsonSerializationExample()
{
    string filePath = "person.json";

    // オブジェクトの作成
    Person person = new Person("鈴木花子", 25)
    {
        BirthDate = new DateTime(1995, 10, 20),
        Hobbies = new List<string> { "料理", "ガーデニング", "写真" }
    };

    // JSONシリアライズのオプション
    JsonSerializerOptions options = new JsonSerializerOptions
    {
        WriteIndented = true,  // 整形されたJSONを出力
        Encoder = System.Text.Encodings.Web.JavaScriptEncoder.UnsafeRelaxedJsonEscaping
// 日本語などのUnicode文字を適切に処理
    };

    // オブジェクトのシリアライズ
    string jsonString = JsonSerializer.Serialize(person, options);
    File.WriteAllText(filePath, jsonString);

    Console.WriteLine("JSONシリアライズ結果:");
    Console.WriteLine(jsonString);

    // JSONからオブジェクトへのデシリアライズ
    string readJson = File.ReadAllText(filePath);
    Person deserializedPerson = JsonSerializer.Deserialize<Person>(readJson);

    Console.WriteLine("デシリアライズしたデータ:");
    Console.WriteLine(deserializedPerson);
}
```

XMLシリアライゼーション

XML形式でのシリアライゼーションも広くサポートされています：

```
using System.Xml.Serialization;

// XML対応のクラス（バイナリシリアライゼーションとは異なり、XMLシリアライゼーションでは
[Serializable]属性は不要です）
public class Product
{
    public int Id { get; set; }
    public string Name { get; set; }
    public decimal Price { get; set; }

    // XMLシリアライゼーションで無視するプロパティ
    [XmlIgnore]
    public string InternalCode { get; set; }

    // パラメータなしのコンストラクタが必要（XMLシリアライゼーションの要件）
    public Product() { }

    public Product(int id, string name, decimal price)
    {
        Id = id;
        Name = name;
        Price = price;
    }
}

// XMLシリアライゼーションの例
public void XmlSerializationExample()
{
    string filePath = "product.xml";

    // 単一オブジェクトのシリアライズ
    Product product = new Product(1, "ノートパソコン", 89800)
    {
        InternalCode = "PC001"  // この値はシリアライズされません
    };

    XmlSerializer serializer = new XmlSerializer(typeof(Product));

    using (StreamWriter writer = new StreamWriter(filePath))
    {
        serializer.Serialize(writer, product);
    }

    Console.WriteLine($"製品情報をXMLファイル {filePath} に保存しました");

    // XMLの内容を表示
    Console.WriteLine("XMLの内容:");
    Console.WriteLine(File.ReadAllText(filePath));

    // XMLからオブジェクトへのデシリアライズ
    Product deserializedProduct;
    using (StreamReader reader = new StreamReader(filePath))
    {
        deserializedProduct = (Product)serializer.Deserialize(reader);
    }
```

```
    Console.WriteLine("デシリアライズした製品情報:");
    Console.WriteLine($"ID: {deserializedProduct.Id}, 名前: {deserializedProduct.Name},
価格: {deserializedProduct.Price}円");

    // オブジェクトのコレクションのシリアライズ
    List<Product> products = new List<Product>
    {
        new Product(1, "ノートパソコン", 89800),
        new Product(2, "スマートフォン", 79800),
        new Product(3, "タブレット", 49800)
    };

    // コレクション用のXMLシリアライザ
    XmlSerializer listSerializer = new XmlSerializer(typeof(List<Product>));

    using (StreamWriter writer = new StreamWriter("products.xml"))
    {
        listSerializer.Serialize(writer, products);
    }

    Console.WriteLine("製品リストをXMLに保存しました");
}
```

> **プロジェクト事例**: あるデスクトップアプリケーションでは、アプリケーションの設定や状態をJSON形式で保存し、次回起動時に復元する必要がありました。System.Text.Jsonライブラリを使用することで、ユーザーの好みや最後に開いたファイル、ウィンドウの位置などの情報を簡単に保存できました。JSONはテキスト形式なので、問題が発生した場合にもファイルを直接確認して修正できるという利点がありました。

データベース接続

多くのアプリケーションでは、データベースを使用してデータを永続化します。C#では、ADO.NETを使用してデータベースに接続できます。ここでは、SQLiteという軽量データベースを例に説明します。

まず、NuGetパッケージマネージャーを使用して、「System.Data.SQLite」パッケージをインストールします。

```
using System.Data.SQLite;

// SQLiteデータベースの基本操作
public void SqliteExample()
{
    // データベースファイルのパス
    string dbPath = "sample.db";

    // データベース接続文字列
    string connectionString = $"Data Source={dbPath};Version=3;";

    // データベースファイルが存在しない場合は作成
    if (!File.Exists(dbPath))
    {
        SQLiteConnection.CreateFile(dbPath);
    }

    // データベースに接続
```

```
    using (SQLiteConnection connection = new SQLiteConnection(connectionString))
    {
        connection.Open();

        // テーブルの作成
        string createTableQuery = @"
            CREATE TABLE IF NOT EXISTS Users (
                Id INTEGER PRIMARY KEY AUTOINCREMENT,
                Name TEXT NOT NULL,
                Email TEXT UNIQUE,
                Age INTEGER
            )";

        using (SQLiteCommand command = new SQLiteCommand(createTableQuery, connection))
        {
            command.ExecuteNonQuery();
        }

        // データの挿入
        string insertQuery = @"
            INSERT INTO Users (Name, Email, Age)
            VALUES (@Name, @Email, @Age)";

        using (SQLiteCommand command = new SQLiteCommand(insertQuery, connection))
        {
            // パラメータの設定
            command.Parameters.AddWithValue("@Name", "山田太郎");
            command.Parameters.AddWithValue("@Email", "yamada@example.com");
            command.Parameters.AddWithValue("@Age", 30);

            int rowsAffected = command.ExecuteNonQuery();
            Console.WriteLine($"{rowsAffected}行挿入されました");
        }

        // 複数のデータを挿入
        List<(string Name, string Email, int Age)> users = new List<(string, string,
int)>
        {
            ("鈴木花子", "suzuki@example.com", 25),
            ("佐藤健", "sato@example.com", 35),
            ("田中誠", "tanaka@example.com", 28)
        };

        foreach (var user in users)
        {
            using (SQLiteCommand command = new SQLiteCommand(insertQuery, connection))
            {
                command.Parameters.AddWithValue("@Name", user.Name);
                command.Parameters.AddWithValue("@Email", user.Email);
                command.Parameters.AddWithValue("@Age", user.Age);

                command.ExecuteNonQuery();
            }
        }

        // データの読み取り
        string selectQuery = "SELECT * FROM Users";
```

```
using (SQLiteCommand command = new SQLiteCommand(selectQuery, connection))
{
    using (SQLiteDataReader reader = command.ExecuteReader())
    {
        Console.WriteLine("ユーザー一覧:");
        Console.WriteLine("ID | 名前 | メール | 年齢");
        Console.WriteLine("------------------");

        while (reader.Read())
        {
            int id = reader.GetInt32(0);
            string name = reader.GetString(1);
            string email = reader.GetString(2);
            int age = reader.GetInt32(3);

            Console.WriteLine($"{id} | {name} | {email} | {age}");
        }
    }
}

// データの更新
string updateQuery = "UPDATE Users SET Age = @Age WHERE Name = @Name";
using (SQLiteCommand command = new SQLiteCommand(updateQuery, connection))
{
    command.Parameters.AddWithValue("@Age", 31);
    command.Parameters.AddWithValue("@Name", "山田太郎");

    int rowsAffected = command.ExecuteNonQuery();
    Console.WriteLine($"{rowsAffected}行更新されました");
}

// 条件付きデータの取得
string filteredQuery = "SELECT * FROM Users WHERE Age > @MinAge";
using (SQLiteCommand command = new SQLiteCommand(filteredQuery, connection))
{
    command.Parameters.AddWithValue("@MinAge", 30);

    using (SQLiteDataReader reader = command.ExecuteReader())
    {
        Console.WriteLine("30歳以上のユーザー:");

        while (reader.Read())
        {
            int id = reader.GetInt32(0);
            string name = reader.GetString(1);
            int age = reader.GetInt32(3);

            Console.WriteLine($"{id} | {name} | {age}歳");
        }
    }
}

// データの削除
string deleteQuery = "DELETE FROM Users WHERE Name = @Name";
using (SQLiteCommand command = new SQLiteCommand(deleteQuery, connection))
{
    command.Parameters.AddWithValue("@Name", "田中誠");
```

```
            int rowsAffected = command.ExecuteNonQuery();
            Console.WriteLine($"{rowsAffected}行削除されました");
        }
    }
}
```

> **ベテランの知恵袋**: データベース操作を行う際には、以下の点に注意すると良いでしょう：
>
> 1. SQLインジェクション攻撃を防ぐため、常にパラメータ化クエリを使用する
> 2. `using` ステートメントを使用して、接続やコマンドのリソースを確実に解放する
> 3. 長時間の処理では接続を開きっぱなしにせず、必要なときだけ開いて閉じる
> 4. トランザクションを使用して、複数の操作を一つの論理的な単位として扱う
> 5. 例外処理を適切に行い、データベースエラーを適切に処理する

Entity Frameworkの紹介

ADO.NETを直接使用する代わりに、オブジェクトリレーショナルマッピング（ORM）フレームワークである「Entity Framework」を使用することもできます。Entity Frameworkを使用すると、データベースのテーブルをC#のクラスとしてモデル化し、SQLを直接書く必要なく操作できます。

以下はEntity Framework Core（EF Core）の基本的な使い方の例です。まず、NuGetパッケージマネージャーを使用して、「Microsoft.EntityFrameworkCore.Sqlite」パッケージをインストールします。

```
using Microsoft.EntityFrameworkCore;
using System.ComponentModel.DataAnnotations;

// エンティティクラス
public class User
{
    [Key]
    public int Id { get; set; }

    [Required]
    [MaxLength(100)]
    public string Name { get; set; }

    [EmailAddress]
    public string Email { get; set; }

    public int Age { get; set; }

    // ナビゲーションプロパティ（一対多のリレーションシップ）
    public List<Post> Posts { get; set; }
}

public class Post
{
    [Key]
    public int Id { get; set; }

    [Required]
    public string Title { get; set; }
```

```
    public string Content { get; set; }

    public DateTime CreatedAt { get; set; }

    // 外部キー
    public int UserId { get; set; }

    // ナビゲーションプロパティ
    public User User { get; set; }
}

// データベースコンテキスト
public class BlogDbContext : DbContext
{
    public DbSet<User> Users { get; set; }
    public DbSet<Post> Posts { get; set; }

    protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
    {
        optionsBuilder.UseSqlite("Data Source=blog.db");
    }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        // ユーザーとポストの関係を設定
        modelBuilder.Entity<Post>()
            .HasOne(p => p.User)
            .WithMany(u => u.Posts)
            .HasForeignKey(p => p.UserId);

        // シード（初期）データの投入
        modelBuilder.Entity<User>().HasData(
            new User { Id = 1, Name = "管理者", Email = "admin@example.com", Age = 30 }
        );
    }
}

// Entity Frameworkの使用例
public void EntityFrameworkExample()
{
    // データベースが存在しない場合は作成
    using (var context = new BlogDbContext())
    {
        context.Database.EnsureCreated();

        // ユーザーの追加
        var user = new User
        {
            Name = "山田太郎",
            Email = "yamada@example.com",
            Age = 28
        };

        context.Users.Add(user);
        context.SaveChanges();

        Console.WriteLine($"ユーザーID: {user.Id} が作成されました");
```

```
        // 投稿の追加
        var post1 = new Post
        {
            Title = "初めての投稿",
            Content = "Entity Frameworkを使ってみました。",
            CreatedAt = DateTime.Now,
            UserId = user.Id
        };

        var post2 = new Post
        {
            Title = "2つ目の投稿",
            Content = "ORMの使い方が分かってきました。",
            CreatedAt = DateTime.Now,
            UserId = user.Id
        };

        context.Posts.AddRange(post1, post2);
        context.SaveChanges();

        Console.WriteLine("投稿が追加されました");

        // データの読み取り（単純なクエリ）
        var users = context.Users.ToList();
        Console.WriteLine($"ユーザー数: {users.Count}");

        // 条件付きクエリ
        var youngUsers = context.Users.Where(u => u.Age < 30).ToList();
        Console.WriteLine($"30歳未満のユーザー数: {youngUsers.Count}");

        // ナビゲーションプロパティを含むクエリ（遅延ロード）
        var userWithPosts = context.Users.Include(u => u.Posts)
                                         .FirstOrDefault(u => u.Name == "山田太郎");

        if (userWithPosts != null)
        {
            Console.WriteLine($"{userWithPosts.Name}の投稿一覧
({userWithPosts.Posts.Count}件) :");
            foreach (var post in userWithPosts.Posts)
            {
                Console.WriteLine($"- {post.Title} ({post.CreatedAt})");
            }
        }

        // データの更新
        var userToUpdate = context.Users.Find(user.Id);
        if (userToUpdate != null)
        {
            userToUpdate.Age = 29;
            context.SaveChanges();
            Console.WriteLine("ユーザー情報が更新されました");
        }

        // LINQクエリ
        var query = from u in context.Users
                    join p in context.Posts on u.Id equals p.UserId
```

```
                    where u.Age >= 25
                    orderby p.CreatedAt descending
                    select new { UserName = u.Name, PostTitle = p.Title, PostDate =
p.CreatedAt };

        Console.WriteLine("最新の投稿一覧:");
        foreach (var item in query.Take(5))
        {
            Console.WriteLine($"{item.UserName} - {item.PostTitle} ({item.PostDate})");
        }

        // データの削除
        var postToDelete = context.Posts.FirstOrDefault(p => p.Title == "初めての投稿");
        if (postToDelete != null)
        {
            context.Posts.Remove(postToDelete);
            context.SaveChanges();
            Console.WriteLine("投稿が削除されました");
        }
    }
}
```

Entity Frameworkを使用すると、データベース操作が大幅に簡素化され、より直感的なコードを書くことができます。特に大規模なアプリケーションでは、SQLを直接扱う複雑さを低減できます。

章末チェックリスト

- [ ] ファイルの基本操作（作成、読み書き、コピー、削除）ができる
- [ ] Pathクラスを使ってパス文字列を適切に操作できる
- [ ] FileStreamを使った低レベルのファイル操作ができる
- [ ] StreamReaderとStreamWriterを使ってテキストファイルを処理できる
- [ ] BinaryReaderとBinaryWriterを使ってバイナリデータを処理できる
- [ ] オブジェクトのシリアライゼーション（JSON, XML, バイナリ）ができる
- [ ] データベースへの接続とSQLコマンドの実行ができる
- [ ] Entity Frameworkの基本的な使い方を理解している
- [ ] ファイル操作やデータベース操作で適切に例外処理を行える
- [ ] リソースの解放（using文など）を適切に行える

まとめと次のステップ

この章では、ファイルやデータの操作方法について学びました。テキストファイルやバイナリファイルの読み書き、オブジェクトのシリアライゼーション、データベース操作など、実用的なアプリケーションで必要となる重要な技術を理解しました。

次の章では、これまでに学んだ知識を活用して、実際のコンソールアプリケーションを開発します。簡単なプロジェクトを通して、C#プログラミングのさまざまな側面を統合的に理解していきましょう。

第9章: 実践プロジェクト: コンソールアプリケーション

この章では、これまでに学んだC#の知識を活用して、実際に動作するコンソールアプリケーションを作成します。具体的には、シンプルなタスク管理アプリケーションを開発し、クラス設計、ファイル操作、例外処理など、様々な要素を実践的に学びます。

プロジェクトの概要

今回作成するアプリケーションは「コンソールタスクマネージャー」です。主な機能は以下の通りです：

1. タスクの追加
2. タスク一覧の表示
3. タスクの完了/未完了のマーク
4. タスクの削除
5. タスクのJSONファイルへの保存
6. タスクのJSONファイルからの読み込み

プロジェクトの準備

まず、新しいコンソールアプリケーションプロジェクトを作成します：

1. Visual Studioを起動
2. 「新しいプロジェクトの作成」を選択
3. 「コンソールアプリ」（.NET Core）を選択
4. プロジェクト名を「ConsoleTaskManager」に設定
5. 「作成」をクリック

基本的なクラス構造

まず、タスクを表現するクラスを作成します：

```
using System;
using System.Collections.Generic;
using System.Text.Json.Serialization;

namespace ConsoleTaskManager
{
    // タスクのステータスを表す列挙型
    public enum TaskStatus
    {
        Pending,     // 未完了
        Completed,   // 完了
        Delayed      // 延期
    }

    // タスクを表すクラス
    public class Task
    {
        // プロパティ
        public int Id { get; set; }
        public string Title { get; set; }
        public string Description { get; set; }
        public DateTime DueDate { get; set; }
        public TaskStatus Status { get; set; }
        public DateTime CreatedAt { get; private set; }
```

```
        // JSONシリアライズ時に無視するプロパティ
        [JsonIgnore]
        public bool IsOverdue => Status == TaskStatus.Pending && DueDate <
DateTime.Today;

        // コンストラクタ
        public Task()
        {
            CreatedAt = DateTime.Now;
            Status = TaskStatus.Pending;
        }

        public Task(string title, string description, DateTime dueDate)
        {
            Title = title;
            Description = description;
            DueDate = dueDate;
            Status = TaskStatus.Pending;
            CreatedAt = DateTime.Now;
        }

        // メソッド
        public void MarkAsCompleted()
        {
            Status = TaskStatus.Completed;
        }

        public void MarkAsPending()
        {
            Status = TaskStatus.Pending;
        }

        public void Delay(DateTime newDueDate)
        {
            if (newDueDate <= DueDate)
            {
                throw new ArgumentException("新しい期日は現在の期日より後である必要がありま
す");
            }

            DueDate = newDueDate;
            Status = TaskStatus.Delayed;
        }

        // タスクの情報を文字列で取得
        public override string ToString()
        {
            string statusText = Status.ToString();

            if (IsOverdue)
            {
                statusText = "期限超過";
            }

            return $"[{Id}] {Title} ({statusText}) - 期限:
{DueDate.ToShortDateString()}";
```

```
        }
    }

    // タスクコレクションを管理するクラス
    public class TaskManager
    {
        private List<Task> _tasks;
        private int _nextId;

        public IReadOnlyList<Task> Tasks => _tasks.AsReadOnly();

        // コンストラクタ
        public TaskManager()
        {
            _tasks = new List<Task>();
            _nextId = 1;
        }

        // タスクの追加
        public Task AddTask(string title, string description, DateTime dueDate)
        {
            if (string.IsNullOrWhiteSpace(title))
            {
                throw new ArgumentException("タイトルは必須です", nameof(title));
            }

            Task newTask = new Task(title, description, dueDate)
            {
                Id = _nextId++
            };

            _tasks.Add(newTask);
            return newTask;
        }

        // IDによるタスクの取得
        public Task GetTaskById(int id)
        {
            return _tasks.FirstOrDefault(t => t.Id == id);
        }

        // タスクの削除
        public bool RemoveTask(int id)
        {
            Task task = GetTaskById(id);
            if (task != null)
            {
                return _tasks.Remove(task);
            }
            return false;
        }

        // タスクを完了としてマーク
        public bool MarkTaskAsCompleted(int id)
        {
            Task task = GetTaskById(id);
            if (task != null)
```

```
            {
                task.MarkAsCompleted();
                return true;
            }
            return false;
        }

        // タスクのステータスによるフィルタリング
        public List<Task> GetTasksByStatus(TaskStatus status)
        {
            return _tasks.Where(t => t.Status == status).ToList();
        }

        // 期限切れのタスクの取得
        public List<Task> GetOverdueTasks()
        {
            return _tasks.Where(t => t.IsOverdue).ToList();
        }

        // すべてのタスクのクリア
        public void ClearAllTasks()
        {
            _tasks.Clear();
            _nextId = 1;
        }
    }
}
```

データの永続化

タスクデータをJSONファイルに保存し、それを読み込む機能を追加します：

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Text.Json;

namespace ConsoleTaskManager
{
    // タスクのシリアライズと永続化を担当するクラス
    public class TaskStorage
    {
        private readonly string _filePath;

        // JSONシリアライズのオプション
        private readonly JsonSerializerOptions _options = new JsonSerializerOptions
        {
            WriteIndented = true,
            PropertyNamingPolicy = JsonNamingPolicy.CamelCase
        };

        public TaskStorage(string filePath)
        {
            _filePath = filePath;
        }
```

```
        // タスクリストをJSONファイルに保存
        public void SaveTasks(IEnumerable<Task> tasks)
        {
            try
            {
                string json = JsonSerializer.Serialize(tasks, _options);
                File.WriteAllText(_filePath, json);
            }
            catch (Exception ex)
            {
                throw new ApplicationException("タスクの保存中にエラーが発生しました", ex);
            }
        }

        // JSONファイルからタスクリストを読み込み
        public List<Task> LoadTasks()
        {
            if (!File.Exists(_filePath))
            {
                return new List<Task>();
            }

            try
            {
                string json = File.ReadAllText(_filePath);
                var tasks = JsonSerializer.Deserialize<List<Task>>(json, _options);
                return tasks ?? new List<Task>();
            }
            catch (Exception ex)
            {
                throw new ApplicationException("タスクの読み込み中にエラーが発生しました",
ex);
            }
        }
    }
}
```

ユーザーインターフェースの作成

コンソールベースのユーザーインターフェースを実装します：

```
using System;
using System.Collections.Generic;

namespace ConsoleTaskManager
{
    // コンソールUIを担当するクラス
    public class ConsoleUI
    {
        private readonly TaskManager _taskManager;
        private readonly TaskStorage _taskStorage;

        public ConsoleUI(TaskManager taskManager, TaskStorage taskStorage)
        {
```

```
        _taskManager = taskManager;
        _taskStorage = taskStorage;
    }

    // メインメニューの表示
    public void ShowMainMenu()
    {
        bool exit = false;

        while (!exit)
        {
            Console.Clear();
            Console.ForegroundColor = ConsoleColor.Cyan;
            Console.WriteLine("===== コンソールタスクマネージャー =====");
            Console.ResetColor();
            Console.WriteLine();
            Console.WriteLine("1. タスク一覧を表示");
            Console.WriteLine("2. 新しいタスクを追加");
            Console.WriteLine("3. タスクを完了としてマーク");
            Console.WriteLine("4. タスクを削除");
            Console.WriteLine("5. タスクを保存");
            Console.WriteLine("6. タスクを読み込み");
            Console.WriteLine("0. 終了");
            Console.WriteLine();
            Console.Write("選択肢を入力してください (0-6): ");

            if (int.TryParse(Console.ReadLine(), out int choice))
            {
                Console.WriteLine();

                switch (choice)
                {
                    case 1:
                        ShowAllTasks();
                        break;
                    case 2:
                        AddNewTask();
                        break;
                    case 3:
                        MarkTaskAsCompleted();
                        break;
                    case 4:
                        RemoveTask();
                        break;
                    case 5:
                        SaveTasks();
                        break;
                    case 6:
                        LoadTasks();
                        break;
                    case 0:
                        exit = true;
                        Console.WriteLine("アプリケーションを終了します...");
                        break;
                    default:
                        ShowError("無効な選択です。0から6の数字を入力してください。");
                        break;
```

```
                    }

                    if (!exit)
                    {
                        Console.WriteLine("\nメインメニューに戻るには何かキーを押してくださ
い...");
                        Console.ReadKey();
                    }
                }
                else
                {
                    ShowError("無効な入力です。数字を入力してください。");
                    Console.ReadKey();
                }
            }
        }

        // すべてのタスクを表示
        private void ShowAllTasks()
        {
            var tasks = _taskManager.Tasks;

            if (tasks.Count == 0)
            {
                Console.WriteLine("タスクがありません。");
                return;
            }

            Console.WriteLine("===== タスク一覧 =====");

            foreach (var task in tasks)
            {
                // ステータスに応じた色分け
                if (task.Status == TaskStatus.Completed)
                {
                    Console.ForegroundColor = ConsoleColor.Green;
                }
                else if (task.IsOverdue)
                {
                    Console.ForegroundColor = ConsoleColor.Red;
                }
                else if (task.Status == TaskStatus.Delayed)
                {
                    Console.ForegroundColor = ConsoleColor.Yellow;
                }

                Console.WriteLine(task);
                Console.ResetColor();

                if (!string.IsNullOrWhiteSpace(task.Description))
                {
                    Console.WriteLine($"  説明: {task.Description}");
                }

                Console.WriteLine();
            }
        }
```

```
        // 新しいタスクを追加
        private void AddNewTask()
        {
            Console.WriteLine("===== 新しいタスクの追加 =====");

            Console.Write("タイトル: ");
            string title = Console.ReadLine();

            if (string.IsNullOrWhiteSpace(title))
            {
                ShowError("タイトルは必須です。");
                return;
            }

            Console.Write("説明 (オプション): ");
            string description = Console.ReadLine();

            Console.Write("期限 (yyyy/MM/dd): ");
            if (!DateTime.TryParse(Console.ReadLine(), out DateTime dueDate))
            {
                ShowError("無効な日付形式です。");
                return;
            }

            try
            {
                var task = _taskManager.AddTask(title, description, dueDate);
                Console.ForegroundColor = ConsoleColor.Green;
                Console.WriteLine($"タスク「{task.Title}」が追加されました。ID:
{task.Id}");
                Console.ResetColor();
            }
            catch (Exception ex)
            {
                ShowError($"タスクの追加に失敗しました: {ex.Message}");
            }
        }

        // タスクを完了としてマーク
        private void MarkTaskAsCompleted()
        {
            Console.WriteLine("===== タスクを完了としてマーク =====");

            Console.Write("完了としてマークするタスクのIDを入力してください: ");
            if (!int.TryParse(Console.ReadLine(), out int taskId))
            {
                ShowError("無効なID形式です。");
                return;
            }

            bool result = _taskManager.MarkTaskAsCompleted(taskId);

            if (result)
            {
                Console.ForegroundColor = ConsoleColor.Green;
                Console.WriteLine($"タスク (ID: {taskId}) が完了としてマークされました。");
```

```
            Console.ResetColor();
        }
        else
        {
            ShowError($"ID {taskId} のタスクが見つかりませんでした。");
        }
    }

    // タスクを削除
    private void RemoveTask()
    {
        Console.WriteLine("===== タスクの削除 =====");

        Console.Write("削除するタスクのIDを入力してください: ");
        if (!int.TryParse(Console.ReadLine(), out int taskId))
        {
            ShowError("無効なID形式です。");
            return;
        }

        Console.Write($"タスク (ID: {taskId}) を削除してもよろしいですか？ (y/n): ");
        string confirmation = Console.ReadLine()?.ToLower();

        if (confirmation == "y" || confirmation == "yes")
        {
            bool result = _taskManager.RemoveTask(taskId);

            if (result)
            {
                Console.ForegroundColor = ConsoleColor.Green;
                Console.WriteLine($"タスク (ID: {taskId}) が削除されました。");
                Console.ResetColor();
            }
            else
            {
                ShowError($"ID {taskId} のタスクが見つかりませんでした。");
            }
        }
        else
        {
            Console.WriteLine("タスクの削除がキャンセルされました。");
        }
    }

    // タスクを保存
    private void SaveTasks()
    {
        try
        {
            _taskStorage.SaveTasks(_taskManager.Tasks);
            Console.ForegroundColor = ConsoleColor.Green;
            Console.WriteLine("タスクが正常に保存されました。");
            Console.ResetColor();
        }
        catch (Exception ex)
        {
            ShowError($"タスクの保存に失敗しました: {ex.Message}");
```

```
            }
        }

        // タスクを読み込み
        private void LoadTasks()
        {
            try
            {
                var tasks = _taskStorage.LoadTasks();
                _taskManager.ClearAllTasks();

                foreach (var task in tasks)
                {
                    _taskManager.AddTask(task.Title, task.Description, task.DueDate);
                }

                Console.ForegroundColor = ConsoleColor.Green;
                Console.WriteLine($"{tasks.Count} 個のタスクが読み込まれました。");
                Console.ResetColor();
            }
            catch (Exception ex)
            {
                ShowError($"タスクの読み込みに失敗しました: {ex.Message}");
            }
        }

        // エラーメッセージの表示
        private void ShowError(string message)
        {
            Console.ForegroundColor = ConsoleColor.Red;
            Console.WriteLine($"エラー: {message}");
            Console.ResetColor();
        }
    }
}
```

メインプログラム

最後に、アプリケーションのエントリーポイントとなるメインプログラムを実装します：

```
using System;
using System.IO;

namespace ConsoleTaskManager
{
    class Program
    {
        static void Main(string[] args)
        {
            try
            {
                // タスクデータの保存先
                string appDataPath = Path.Combine(

Environment.GetFolderPath(Environment.SpecialFolder.ApplicationData),
```

```
                    "ConsoleTaskManager"
                );

                // ディレクトリが存在しない場合は作成
                if (!Directory.Exists(appDataPath))
                {
                    Directory.CreateDirectory(appDataPath);
                }

                string tasksFilePath = Path.Combine(appDataPath, "tasks.json");

                // 各コンポーネントの初期化
                TaskManager taskManager = new TaskManager();
                TaskStorage taskStorage = new TaskStorage(tasksFilePath);
                ConsoleUI ui = new ConsoleUI(taskManager, taskStorage);

                // UIの表示
                ui.ShowMainMenu();
            }
            catch (Exception ex)
            {
                Console.ForegroundColor = ConsoleColor.Red;
                Console.WriteLine($"予期しないエラーが発生しました: {ex.Message}");
                Console.WriteLine(ex.StackTrace);
                Console.ResetColor();
                Console.WriteLine("\nアプリケーションを終了するには何かキーを押してくださ
い...");
                Console.ReadKey();
            }
        }
    }
}
```

プロジェクトの構造

以上のファイルを次のような構造で配置します：

```
ConsoleTaskManager/
├── Program.cs            // メインエントリーポイント
├── Task.cs               // タスクとステータスの定義
├── TaskManager.cs        // タスクコレクションの管理
├── TaskStorage.cs        // JSONシリアライゼーション
└── ConsoleUI.cs          // ユーザーインターフェース
```

> **ベテランの知恵袋**: 実際のプロジェクトでは、上記のクラスを別々のファイルに分けると、コードの管理が容易になります。また、より大規模なアプリケーションでは、機能ごとにフォルダを分けたり、インターフェースを使って依存関係を管理するなど、さらに構造化することをお勧めします。

プロジェクトの拡張アイデア

このプロジェクトをさらに発展させるためのアイデアをいくつか紹介します：

1. **タスクの優先度**: タスクに優先度（高、中、低など）を追加する

2. **タスクのカテゴリ**: タスクをカテゴリ別に分類する機能を追加する
3. **検索機能**: タイトルや説明でタスクを検索できるようにする
4. **通知機能**: 期限が近いタスクや期限切れタスクに関する通知を表示する
5. **サブタスク**: タスクにサブタスクを追加できるようにする
6. **複数のユーザー**: ユーザー認証とユーザーごとのタスク管理を実装する
7. **データベース連携**: JSONファイルの代わりにSQLiteデータベースを使用する

> **プロジェクト事例**: あるチームでは、このようなシンプルなタスク管理ツールから始めて、徐々に機能を拡張していきました。最終的には、チーム全体で使えるWeb版のタスク管理システムに発展し、プロジェクト管理の効率が大幅に向上しました。小さく始めて、必要に応じて拡張していくというアプローチは、ソフトウェア開発において非常に有効です。

コードレビューのポイント

このプロジェクトのコードを見直す際に注目すべきポイントをいくつか挙げます：

1. **責任の分離**: 異なるクラスが明確な責任を持っている（タスク管理、ストレージ、UI）
2. **例外処理**: 適切な場所で例外をキャッチし、ユーザーフレンドリーなメッセージを表示している
3. **入力検証**: ユーザー入力を適切に検証し、無効な入力を処理している
4. **データの永続化**: JSONを使用してデータを保存・読み込みしている
5. **UI/ロジックの分離**: ビジネスロジックとユーザーインターフェースが分離されている

> **若手の疑問解決**: 「なぜクラスを分けることが重要なのですか？」
>
> クラスを分けることで、コードの「単一責任の原則」（Single Responsibility Principle）を実現できます。各クラスが明確な役割を持つことで、コードの理解、テスト、保守が容易になります。また、将来的な変更（例えば、コンソールUIをGUIに変更する、JSONストレージをデータベースに変更するなど）も、関連する部分だけを修正すれば済むようになります。

章末チェックリスト

- [ ] クラス設計と責任の分離の原則を理解している
- [ ] プロパティとフィールドを適切に使い分けられる
- [ ] 列挙型を使って状態を表現できる
- [ ] JSONシリアライゼーションでデータを永続化できる
- [ ] コンソールUIを構築し、ユーザー入力を処理できる
- [ ] 入力検証と例外処理を適切に行える
- [ ] アプリケーションの異なる部分を適切に連携させることができる
- [ ] ファイルシステムを使ってデータを管理できる
- [ ] コードをわかりやすく構造化できる
- [ ] 実用的なコンソールアプリケーションを開発できる

まとめと次のステップ

この章では、これまでに学んだC#の知識を活用して、実用的なコンソールタスク管理アプリケーションを開発しました。クラス設計、ファイル操作、ユーザーインターフェース、例外処理など、様々な要素を組み合わせてアプリケーションを構築しました。

次の章では、グラフィカルユーザーインターフェース（GUI）アプリケーションの開発に向けた最初のステップを学びます。Windows Forms（WinForms）やWindows Presentation Foundation（WPF）を使用

した基本的なGUIアプリケーションの作成について理解していきましょう。

第10章: 応用への道: GUIアプリケーション入門

コンソールアプリケーションは多くの場面で有用ですが、一般的なエンドユーザー向けのアプリケーションでは、グラフィカルユーザーインターフェース（GUI）が使われることがほとんどです。この章では、C#を使ったGUIアプリケーション開発の基礎を学びます。

GUIフレームワークの概要

C#には、GUIアプリケーションを開発するためのいくつかのフレームワークがあります：

1. **Windows Forms (WinForms)**: 比較的シンプルで、従来のWindowsデスクトップアプリケーションを作成できる
2. **Windows Presentation Foundation (WPF)**: より高度なグラフィックス機能とXAMLベースのデザインシステムを備えている
3. **Universal Windows Platform (UWP)**: Windows 10向けの最新プラットフォーム
4. **Xamarin.Forms / .NET MAUI**: クロスプラットフォームのモバイル／デスクトップアプリケーション開発用
5. **Avalonia UI**: クロスプラットフォームのデスクトップアプリケーション開発用

この章では、比較的シンプルなWindows Formsを使って基本的なGUIアプリケーション開発を学びます。

Windows Formsプロジェクトの作成

はじめに、Windows Formsアプリケーションのプロジェクトを作成します：

1. Visual Studioを起動
2. 「新しいプロジェクトの作成」を選択
3. 「Windows Forms App (.NET)」を選択
4. プロジェクト名を「SimpleNotepad」に設定
5. 「作成」をクリック

プロジェクトが作成されると、Form1.csファイルがデザイナーで開かれます。これが、アプリケーションのメインウィンドウになります。

フォームデザイン

Windows Formsデザイナーを使って、視覚的にユーザーインターフェースを設計できます。以下の手順で簡単なメモ帳アプリケーションを作成してみましょう：

1. **フォームのプロパティ設定**:
   - プロパティウィンドウで、Form1の以下のプロパティを設定します：
     - Text: "シンプルメモ帳"
     - Size: Width = 800, Height = 600
     - StartPosition: CenterScreen
2. **コントロールの追加**:
   - ツールボックスから以下のコントロールをフォームにドラッグ＆ドロップします：
     - MenuStrip（メニューバー）

    - TextBox（テキスト入力エリア）
    - StatusStrip（ステータスバー）
3. **TextBoxの設定**:
    - TextBoxを選択し、以下のプロパティを設定します：
        - Name: "textBoxContent"
        - Dock: Fill（フォーム全体に広がるように）
        - Multiline: True（複数行入力可能に）
        - ScrollBars: Vertical（垂直スクロールバーを表示）
        - Font: 任意のフォントとサイズを設定
4. **MenuStripの設定**:
    - MenuStripをクリックし、以下のメニュー項目を追加します：
        - ファイル
            - 新規
            - 開く
            - 保存
            - 区切り線
            - 終了
        - 編集
            - 切り取り
            - コピー
            - 貼り付け
        - ヘルプ
            - バージョン情報
5. **StatusStripの設定**:
    - StatusStripをクリックし、ステータスラベルを追加します：
        - Name: "statusLabel"
        - Text: "準備完了"

イベントハンドラの実装

次に、各コントロールにイベントハンドラを追加して、ユーザーアクションに応答できるようにします：

1. **メニュー項目のイベントハンドラ**:
    - 「ファイル」→「新規」メニュー項目をダブルクリックし、イベントハンドラを作成します。以下のコードを追加します：

```
private void 新規ToolStripMenuItem_Click(object sender, EventArgs e)
{
    // 未保存の変更がある場合は確認
    if (textBoxContent.Modified)
    {
        DialogResult result = MessageBox.Show(
            "変更を保存しますか？",
            "確認",
            MessageBoxButtons.YesNoCancel,
            MessageBoxIcon.Question);

        if (result == DialogResult.Yes)
        {
```

```
            保存ToolStripMenuItem_Click(sender, e);
        }
        else if (result == DialogResult.Cancel)
        {
            return;
        }
    }

    // テキストボックスをクリア
    textBoxContent.Clear();
    textBoxContent.Modified = false;
    statusLabel.Text = "新しいファイルを作成しました";
}
```

2. **「開く」メニュー項目のイベントハンドラ**:

```
private void 開くToolStripMenuItem_Click(object sender, EventArgs e)
{
    // ファイル選択ダイアログの表示
    OpenFileDialog openFileDialog = new OpenFileDialog
    {
        Filter = "テキストファイル (*.txt)|*.txt|すべてのファイル (*.*)|*.*",
        Title = "ファイルを開く"
    };

    if (openFileDialog.ShowDialog() == DialogResult.OK)
    {
        try
        {
            // ファイルからテキストを読み込む
            string filePath = openFileDialog.FileName;
            textBoxContent.Text = File.ReadAllText(filePath);

            // 現在のファイルパスをTagプロパティに保存
            this.Tag = filePath;

            textBoxContent.Modified = false;
            statusLabel.Text = $"ファイル '{Path.GetFileName(filePath)}' を開きました";
        }
        catch (Exception ex)
        {
            MessageBox.Show(
                $"ファイルを開けませんでした: {ex.Message}",
                "エラー",
                MessageBoxButtons.OK,
                MessageBoxIcon.Error);
        }
    }
}
```

3. **「保存」メニュー項目のイベントハンドラ**:

```
private void 保存ToolStripMenuItem_Click(object sender, EventArgs e)
{
```

```
    // 現在のファイルパスがない場合は「名前を付けて保存」と同じ動作
    if (this.Tag == null)
    {
        SaveFileDialog saveFileDialog = new SaveFileDialog
        {
            Filter = "テキストファイル (*.txt)|*.txt|すべてのファイル (*.*)|*.*",
            Title = "名前を付けて保存"
        };

        if (saveFileDialog.ShowDialog() == DialogResult.OK)
        {
            this.Tag = saveFileDialog.FileName;
        }
        else
        {
            return;  // キャンセルされた場合
        }
    }

    try
    {
        // ファイルにテキストを書き込む
        string filePath = this.Tag.ToString();
        File.WriteAllText(filePath, textBoxContent.Text);

        textBoxContent.Modified = false;
        statusLabel.Text = $"ファイル '{Path.GetFileName(filePath)}' を保存しました";
    }
    catch (Exception ex)
    {
        MessageBox.Show(
            $"ファイルを保存できませんでした: {ex.Message}",
            "エラー",
            MessageBoxButtons.OK,
            MessageBoxIcon.Error);
    }
}
```

4. **「終了」メニュー項目のイベントハンドラ**:

```
private void 終了ToolStripMenuItem_Click(object sender, EventArgs e)
{
    // フォームを閉じる（FormClosingイベントが発生）
    this.Close();
}
```

5. **フォームのClosingイベントハンドラ**:
   - Form1を選択し、Propertiesウィンドウでイベントビュー（稲妻アイコン）をクリックします。
   - FormClosingイベントをダブルクリックし、以下のコードを追加します：

```
private void Form1_FormClosing(object sender, FormClosingEventArgs e)
{
    // 未保存の変更がある場合は確認
    if (textBoxContent.Modified)
```

```
    {
        DialogResult result = MessageBox.Show(
            "変更を保存しますか？",
            "確認",
            MessageBoxButtons.YesNoCancel,
            MessageBoxIcon.Question);

        if (result == DialogResult.Yes)
        {
            保存ToolStripMenuItem_Click(sender, e);
        }
        else if (result == DialogResult.Cancel)
        {
            e.Cancel = true;  // 閉じる操作をキャンセル
        }
    }
}
```

6. **編集メニュー項目のイベントハンドラ**:

```
private void 切り取りToolStripMenuItem_Click(object sender, EventArgs e)
{
    textBoxContent.Cut();
    statusLabel.Text = "選択したテキストを切り取りました";
}

private void コピーToolStripMenuItem_Click(object sender, EventArgs e)
{
    textBoxContent.Copy();
    statusLabel.Text = "選択したテキストをコピーしました";
}

private void 貼り付けToolStripMenuItem_Click(object sender, EventArgs e)
{
    textBoxContent.Paste();
    statusLabel.Text = "クリップボードの内容を貼り付けました";
}
```

7. **「バージョン情報」メニュー項目のイベントハンドラ**:

```
private void バージョン情報ToolStripMenuItem_Click(object sender, EventArgs e)
{
    MessageBox.Show(
        "シンプルメモ帳 v1.0\nC#入門学習用のサンプルアプリケーションです。",
        "バージョン情報",
        MessageBoxButtons.OK,
        MessageBoxIcon.Information);
}
```

完全なコードサンプル

最終的なForm1.csファイルは以下のようになります（デザイナーが生成するコードは省略しています）：

```
using System;
using System.IO;
using System.Windows.Forms;

namespace SimpleNotepad
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();

            // 初期設定
            this.Text = "シンプルメモ帳 - 無題";
        }

        // 「新規」メニュー項目のイベントハンドラ
        private void 新規ToolStripMenuItem_Click(object sender, EventArgs e)
        {
            // 未保存の変更がある場合は確認
            if (textBoxContent.Modified)
            {
                DialogResult result = MessageBox.Show(
                    "変更を保存しますか？",
                    "確認",
                    MessageBoxButtons.YesNoCancel,
                    MessageBoxIcon.Question);

                if (result == DialogResult.Yes)
                {
                    保存ToolStripMenuItem_Click(sender, e);
                }
                else if (result == DialogResult.Cancel)
                {
                    return;
                }
            }

            // テキストボックスをクリア
            textBoxContent.Clear();
            textBoxContent.Modified = false;
            this.Tag = null;
            this.Text = "シンプルメモ帳 - 無題";
            statusLabel.Text = "新しいファイルを作成しました";
        }

        // 「開く」メニュー項目のイベントハンドラ
        private void 開くToolStripMenuItem_Click(object sender, EventArgs e)
        {
            // 未保存の変更がある場合は確認
            if (textBoxContent.Modified)
            {
                DialogResult result = MessageBox.Show(
                    "変更を保存しますか？",
                    "確認",
                    MessageBoxButtons.YesNoCancel,
```

```
                    MessageBoxIcon.Question);

                if (result == DialogResult.Yes)
                {
                    保存ToolStripMenuItem_Click(sender, e);
                }
                else if (result == DialogResult.Cancel)
                {
                    return;
                }
            }

            // ファイル選択ダイアログの表示
            OpenFileDialog openFileDialog = new OpenFileDialog
            {
                Filter = "テキストファイル (*.txt)|*.txt|すべてのファイル (*.*)|*.*",
                Title = "ファイルを開く"
            };

            if (openFileDialog.ShowDialog() == DialogResult.OK)
            {
                try
                {
                    // ファイルからテキストを読み込む
                    string filePath = openFileDialog.FileName;
                    textBoxContent.Text = File.ReadAllText(filePath);

                    // 現在のファイルパスをTagプロパティに保存
                    this.Tag = filePath;
                    this.Text = $"シンプルメモ帳 - {Path.GetFileName(filePath)}";

                    textBoxContent.Modified = false;
                    statusLabel.Text = $"ファイル '{Path.GetFileName(filePath)}' を開きま
した";
                }
                catch (Exception ex)
                {
                    MessageBox.Show(
                        $"ファイルを開けませんでした: {ex.Message}",
                        "エラー",
                        MessageBoxButtons.OK,
                        MessageBoxIcon.Error);
                }
            }
        }

        // 「保存」メニュー項目のイベントハンドラ
        private void 保存ToolStripMenuItem_Click(object sender, EventArgs e)
        {
            // 現在のファイルパスがない場合は「名前を付けて保存」と同じ動作
            if (this.Tag == null)
            {
                SaveFileDialog saveFileDialog = new SaveFileDialog
                {
                    Filter = "テキストファイル (*.txt)|*.txt|すべてのファイル (*.*)|*.*",
                    Title = "名前を付けて保存"
                };
```

```
            if (saveFileDialog.ShowDialog() == DialogResult.OK)
            {
                this.Tag = saveFileDialog.FileName;
            }
            else
            {
                return;  // キャンセルされた場合
            }
        }

        try
        {
            // ファイルにテキストを書き込む
            string filePath = this.Tag.ToString();
            File.WriteAllText(filePath, textBoxContent.Text);

            textBoxContent.Modified = false;
            this.Text = $"シンプルメモ帳 - {Path.GetFileName(filePath)}";
            statusLabel.Text = $"ファイル '{Path.GetFileName(filePath)}' を保存しました";
        }
        catch (Exception ex)
        {
            MessageBox.Show(
                $"ファイルを保存できませんでした: {ex.Message}",
                "エラー",
                MessageBoxButtons.OK,
                MessageBoxIcon.Error);
        }
    }

    // 「終了」メニュー項目のイベントハンドラ
    private void 終了ToolStripMenuItem_Click(object sender, EventArgs e)
    {
        // フォームを閉じる（FormClosingイベントが発生）
        this.Close();
    }

    // フォームのClosingイベントハンドラ
    private void Form1_FormClosing(object sender, FormClosingEventArgs e)
    {
        // 未保存の変更がある場合は確認
        if (textBoxContent.Modified)
        {
            DialogResult result = MessageBox.Show(
                "変更を保存しますか？",
                "確認",
                MessageBoxButtons.YesNoCancel,
                MessageBoxIcon.Question);

            if (result == DialogResult.Yes)
            {
                保存ToolStripMenuItem_Click(sender, e);
            }
            else if (result == DialogResult.Cancel)
            {
```

```
                    e.Cancel = true;  // 閉じる操作をキャンセル
                }
            }
        }

        // 「切り取り」メニュー項目のイベントハンドラ
        private void 切り取りToolStripMenuItem_Click(object sender, EventArgs e)
        {
            textBoxContent.Cut();
            statusLabel.Text = "選択したテキストを切り取りました";
        }

        // 「コピー」メニュー項目のイベントハンドラ
        private void コピーToolStripMenuItem_Click(object sender, EventArgs e)
        {
            textBoxContent.Copy();
            statusLabel.Text = "選択したテキストをコピーしました";
        }

        // 「貼り付け」メニュー項目のイベントハンドラ
        private void 貼り付けToolStripMenuItem_Click(object sender, EventArgs e)
        {
            textBoxContent.Paste();
            statusLabel.Text = "クリップボードの内容を貼り付けました";
        }

        // 「バージョン情報」メニュー項目のイベントハンドラ
        private void バージョン情報ToolStripMenuItem_Click(object sender, EventArgs e)
        {
            MessageBox.Show(
                "シンプルメモ帳 v1.0\nC#入門学習用のサンプルアプリケーションです。",
                "バージョン情報",
                MessageBoxButtons.OK,
                MessageBoxIcon.Information);
        }

        // テキストボックスの内容が変更された時のイベントハンドラ
        private void textBoxContent_TextChanged(object sender, EventArgs e)
        {
            // 文字数と行数をステータスバーに表示
            int charCount = textBoxContent.Text.Length;
            int lineCount = textBoxContent.Lines.Length;
            statusLabel.Text = $"文字数: {charCount}, 行数: {lineCount}";
        }
    }
}
```

> **ベテランの知恵袋**: Windows Formsアプリケーションでは、変数名やイベントハンドラ名に日本語を使用することも可能ですが、一般的には英語名を使用することが推奨されます。特に複数人で開発する場合や、国際的なプロジェクトでは、英語名の方が汎用性があります。このサンプルでは学習目的で日本語名を使用していますが、実際のプロジェクトでは英語名を検討するとよいでしょう。

アプリケーションの実行とテスト

プロジェクトをビルドして実行し、以下の機能をテストします：

1. 新しいテキストの入力
2. ファイルへの保存
3. 新規ファイルの作成
4. 既存ファイルを開く
5. テキストの切り取り、コピー、貼り付け
6. 変更を保存せずにアプリケーションを閉じようとする

より洗練されたGUIアプリケーションへ

この簡単なメモ帳アプリケーションは、GUIアプリケーション開発の基本を示しています。より洗練されたアプリケーションにするための拡張アイデアをいくつか紹介します：

1. **検索機能**: テキスト内の特定の文字列を検索する機能
2. **置換機能**: 文字列を別の文字列に置き換える機能
3. **フォント設定**: ユーザーがテキストのフォントやサイズを変更できる機能
4. **印刷機能**: テキストを印刷する機能
5. **複数のタブ**: 複数のファイルを同時に編集できるタブ機能
6. **構文ハイライト**: プログラミング言語のキーワードを色分けする機能
7. **自動保存**: 定期的に自動保存する機能
8. **取り消し・やり直し**: 操作を取り消したり、やり直したりする機能

> **若手の疑問解決**: 「Windows FormsとWPFはどう使い分ければいいですか？」
>
> Windows Formsは比較的シンプルで学習コストが低く、従来型のデスクトップアプリケーションを素早く作成できます。一方、WPFはより柔軟なデザイン（XAMLによる宣言的UI）と高度なグラフィックス機能を提供し、MVVMなどの近代的なアーキテクチャパターンとの親和性が高いです。以下のような基準で選ぶとよいでしょう：
>
> Windows Formsを選ぶケース:
>
> - シンプルなビジネスアプリケーション
> - 学習コストを最小限にしたい場合
> - 短期間で開発する必要がある場合
> - 既存のWindows Formsアプリケーションの保守・拡張
>
> WPFを選ぶケース:
>
> - リッチなユーザーインターフェースが必要な場合
> - カスタマイズ性の高いUIが必要な場合
> - MVVMパターンでの開発を予定している場合
> - アニメーションや3Dグラフィックスを使用する場合

MVVMパターンの紹介

より複雑なGUIアプリケーション開発では、MVVM（Model-View-ViewModel）などのデザインパターンがよく使われます。MVVMは、ユーザーインターフェース（View）とビジネスロジック（Model）を分離し、その間を仲介するViewModel層を提供します。

MVVMパターンの基本構造：

1. **Model**: データとビジネスロジックを表現

2. **View**: ユーザーインターフェース
3. **ViewModel**: ViewとModelの間のブリッジ。Viewに表示するデータとコマンドを提供

WPFでは、データバインディング、コマンド、ビヘイビアなどの機能を使ってMVVMパターンを効果的に実装できます。以下は、MVVMパターンに基づいたシンプルなWPFアプリケーションの例です：

```
// Model
public class NoteModel
{
    public string Content { get; set; }
    public string FilePath { get; set; }
    public bool IsModified { get; set; }

    public void SaveToFile()
    {
        if (!string.IsNullOrEmpty(FilePath))
        {
            File.WriteAllText(FilePath, Content);
            IsModified = false;
        }
    }

    public void LoadFromFile(string path)
    {
        if (File.Exists(path))
        {
            Content = File.ReadAllText(path);
            FilePath = path;
            IsModified = false;
        }
    }
}

// ViewModel
public class NoteViewModel : INotifyPropertyChanged
{
    private NoteModel _model;

    public NoteViewModel()
    {
        _model = new NoteModel();
        SaveCommand = new RelayCommand(Save, CanSave);
        OpenCommand = new RelayCommand(Open, CanExecute);
    }

    public string Content
    {
        get { return _model.Content; }
        set
        {
            if (_model.Content != value)
            {
                _model.Content = value;
                _model.IsModified = true;
                OnPropertyChanged(nameof(Content));
                OnPropertyChanged(nameof(WindowTitle));
```

```
                SaveCommand.RaiseCanExecuteChanged();
            }
        }
    }

    public string WindowTitle
    {
        get
        {
            string fileName = string.IsNullOrEmpty(_model.FilePath) ? "無題" :
Path.GetFileName(_model.FilePath);
            string modifiedMark = _model.IsModified ? "*" : "";
            return $"シンプルメモ帳 - {fileName}{modifiedMark}";
        }
    }

    // コマンド
    public RelayCommand SaveCommand { get; private set; }
    public RelayCommand OpenCommand { get; private set; }

    private bool CanExecute()
    {
        return true;
    }

    private bool CanSave()
    {
        return _model.IsModified;
    }

    private void Save()
    {
        if (string.IsNullOrEmpty(_model.FilePath))
        {
            SaveFileDialog dialog = new SaveFileDialog
            {
                Filter = "テキストファイル (*.txt)|*.txt|すべてのファイル (*.*)|*.*",
                Title = "名前を付けて保存"
            };

            if (dialog.ShowDialog() == true)
            {
                _model.FilePath = dialog.FileName;
            }
            else
            {
                return;
            }
        }

        _model.SaveToFile();
        OnPropertyChanged(nameof(WindowTitle));
        SaveCommand.RaiseCanExecuteChanged();
    }

    private void Open()
    {
```

```
        OpenFileDialog dialog = new OpenFileDialog
        {
            Filter = "テキストファイル (*.txt)|*.txt|すべてのファイル (*.*)|*.*",
            Title = "ファイルを開く"
        };

        if (dialog.ShowDialog() == true)
        {
            _model.LoadFromFile(dialog.FileName);
            OnPropertyChanged(nameof(Content));
            OnPropertyChanged(nameof(WindowTitle));
            SaveCommand.RaiseCanExecuteChanged();
        }
    }

    // INotifyPropertyChangedの実装
    public event PropertyChangedEventHandler PropertyChanged;

    protected virtual void OnPropertyChanged(string propertyName)
    {
        PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(propertyName));
    }
}

// RelayCommandの実装（簡略版）
public class RelayCommand : ICommand
{
    private readonly Action _execute;
    private readonly Func<bool> _canExecute;

    public RelayCommand(Action execute, Func<bool> canExecute = null)
    {
        _execute = execute ?? throw new ArgumentNullException(nameof(execute));
        _canExecute = canExecute;
    }

    public bool CanExecute(object parameter)
    {
        return _canExecute?.Invoke() ?? true;
    }

    public void Execute(object parameter)
    {
        _execute();
    }

    public event EventHandler CanExecuteChanged;

    public void RaiseCanExecuteChanged()
    {
        CanExecuteChanged?.Invoke(this, EventArgs.Empty);
    }
}
```

XAMLファイル（MainWindow.xaml）：

```xaml
<Window x:Class="WpfNotepad.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="{Binding WindowTitle}"
        Height="450" Width="800">
    <Grid>
        <Grid.RowDefinitions>
            <RowDefinition Height="Auto"/>
            <RowDefinition Height="*"/>
            <RowDefinition Height="Auto"/>
        </Grid.RowDefinitions>

        <Menu Grid.Row="0">
            <MenuItem Header="ファイル">
                <MenuItem Header="開く" Command="{Binding OpenCommand}"/>
                <MenuItem Header="保存" Command="{Binding SaveCommand}"/>
                <Separator/>
                <MenuItem Header="終了" Click="ExitMenuItem_Click"/>
            </MenuItem>
        </Menu>

        <TextBox Grid.Row="1"
                 Text="{Binding Content, UpdateSourceTrigger=PropertyChanged}"
                 AcceptsReturn="True"
                 VerticalScrollBarVisibility="Auto"
                 HorizontalScrollBarVisibility="Auto"/>

        <StatusBar Grid.Row="2">
            <TextBlock Text="準備完了"/>
        </StatusBar>
    </Grid>
</Window>
```

この例ではMVVMパターンの基本的な構造を示していますが、実際のアプリケーションではより複雑になり、MVVMフレームワーク（Prism、MVVMLight、ReactiveUIなど）を使用することが一般的です。

> **プロジェクト事例**: あるビジネスアプリケーションでは、当初はWindows Formsで開発されていましたが、UIの複雑化に伴いコードのメンテナンスが困難になりました。そこでWPFとMVVMパターンを採用してリファクタリングを行い、UIとビジネスロジックを明確に分離することで、テストの容易性と保守性が大幅に向上しました。また、デザイナーとプログラマーが並行して作業できるようになり、開発効率も上がりました。

その他のGUIフレームワーク

C#でのGUIアプリケーション開発には、Windows FormsとWPF以外にも選択肢があります：

1. **Universal Windows Platform (UWP)**:
   - Windows 10向けのモダンなUI
   - Windowsストアへの配布が可能
   - タッチ対応やタブレットモードなどの新機能をサポート
2. **.NET MAUI (Multi-platform App UI)**:
   - Windows、macOS、Android、iOS向けの統一フレームワーク
   - 単一のコードベースで複数のプラットフォームに対応

- Xamarin.Formsの後継

3. **Avalonia UI**:
   - クロスプラットフォームのWPFライクなフレームワーク
   - Windows、macOS、Linux向けのデスクトップアプリケーションを開発可能
   - XAMLに基づくUIデザイン

> **ベテランの知恵袋**: GUIフレームワークを選ぶ際には、以下の点を考慮するとよいでしょう：
>
> 1. 対象プラットフォーム（Windows専用か、クロスプラットフォームか）
> 2. チームの経験とスキルセット
> 3. プロジェクトの規模と複雑さ
> 4. パフォーマンス要件
> 5. デザインの柔軟性
> 6. 長期的なサポートとメンテナンス
>
> また、小規模なプロジェクトから始めて、フレームワークの特性を理解してから大規模なプロジェクトに適用するというアプローチも有効です。

章末チェックリスト

- [ ] Windows Formsの基本的なコントロール（TextBox、Button、Menuなど）を使用できる
- [ ] イベントハンドラを追加して、ユーザーアクションに応答できる
- [ ] ダイアログ（MessageBox、OpenFileDialog、SaveFileDialogなど）を表示できる
- [ ] メニューとステータスバーを設定できる
- [ ] ファイル操作（読み込み・保存）をGUIから行える
- [ ] 基本的なGUIアプリケーションを設計・実装できる
- [ ] MVVMパターンの基本概念を理解している
- [ ] 様々なGUIフレームワークの特徴と用途を説明できる
- [ ] データバインディングの基本を理解している
- [ ] GUIアプリケーションの拡張方法を理解している

まとめと次のステップ

この章では、C#を使ったGUIアプリケーション開発の基礎を学びました。Windows Formsを使ったシンプルなメモ帳アプリケーションの実装を通じて、イベント処理、ダイアログの表示、ファイル操作などの基本的なGUIプログラミングを理解しました。また、WPFとMVVMパターンの紹介を通じて、より高度なGUIアプリケーション開発の方向性も見てきました。

次の章では、C#学習の次のステップについて考え、継続的な学習のためのリソースや方向性を探ります。

第11章: 学習を続けるために

C#プログラミングの基礎を学んできましたが、プログラミングスキルの向上は継続的な学習と実践によって実現します。この章では、C#学習の次のステップや継続的な成長のための方法について考えます。

学習の道筋

C#スキルを向上させるための一般的な道筋は以下の通りです：

1. **基礎の強化**
   - 本書で学んだ基本概念の復習と実践
   - 小規模なプロジェクトを通じた理解の定着
2. **応用技術の習得**
   - 非同期プログラミング（async/await）
   - LINQの高度な使用法
   - リフレクションとアトリビュート
   - ジェネリックプログラミングの深堀り
3. **専門分野の探求**
   - Webアプリケーション開発（ASP.NET Core）
   - デスクトップアプリケーション開発（WPF、UWP、.NET MAUI）
   - モバイルアプリケーション開発（Xamarin、.NET MAUI）
   - ゲーム開発（Unity）
   - クラウドアプリケーション開発（Azure）
   - 機械学習（ML.NET）
4. **設計とアーキテクチャの学習**
   - デザインパターン
   - ソフトウェアアーキテクチャ
   - テスト駆動開発（TDD）
   - クリーンコード原則

次のステップに向けた学習トピック

以下に、本書の内容から一歩進んだ学習トピックをいくつか紹介します：

非同期プログラミング

`async/await` を使った非同期プログラミングは、応答性の高いアプリケーションを作成するために重要です：

```
// 非同期メソッドの例
public async Task<string> DownloadDataAsync(string url)
{
    using (HttpClient client = new HttpClient())
    {
        string result = await client.GetStringAsync(url);
        return result;
    }
}

// 非同期メソッドの呼び出し
private async void DownloadButton_Click(object sender, EventArgs e)
{
    try
    {
        statusLabel.Text = "ダウンロード中...";
        string data = await DownloadDataAsync("https://example.com/api/data");
        textBox.Text = data;
        statusLabel.Text = "ダウンロード完了";
```

```
    }
    catch (Exception ex)
    {
        statusLabel.Text = "エラー: " + ex.Message;
    }
}
```

高度なLINQ

LINQをより深く理解し、複雑なデータ操作を行う方法：

```
// 複雑なLINQクエリの例
var query = from c in customers
            join o in orders on c.CustomerId equals o.CustomerId into customerOrders
            from co in customerOrders.DefaultIfEmpty() // 左外部結合
            group co by new { c.CustomerId, c.Name } into g
            where g.Sum(o => o?.Total ?? 0) > 1000
            orderby g.Sum(o => o?.Total ?? 0) descending
            select new
            {
                CustomerId = g.Key.CustomerId,
                CustomerName = g.Key.Name,
                TotalOrders = g.Count(o => o != null),
                TotalSpent = g.Sum(o => o?.Total ?? 0)
            };
```

リフレクションとメタプログラミング

実行時にタイプ情報にアクセスして操作する方法：

```
// リフレクションの例
public void InspectObject(object obj)
{
    Type type = obj.GetType();
    Console.WriteLine($"タイプ名: {type.Name}");

    Console.WriteLine("プロパティ一覧:");
    foreach (var prop in type.GetProperties())
    {
        object value = prop.GetValue(obj);
        Console.WriteLine($"  {prop.Name}: {value} ({prop.PropertyType.Name})");
    }

    Console.WriteLine("メソッド一覧:");
    foreach (var method in type.GetMethods())
    {
        Console.WriteLine($"  {method.Name}({string.Join(", ",
method.GetParameters().Select(p => $"{p.ParameterType.Name} {p.Name}"))})");
    }
}

// カスタム属性の定義と使用
[AttributeUsage(AttributeTargets.Class | AttributeTargets.Method)]
public class AuditAttribute : Attribute
```

```
{
    public string Author { get; }
    public DateTime Created { get; }

    public AuditAttribute(string author)
    {
        Author = author;
        Created = DateTime.Now;
    }
}

// 属性の使用
[Audit("山田太郎")]
public class AuditableClass
{
    [Audit("鈴木花子")]
    public void AuditableMethod()
    {
        // 何らかの処理
    }
}

// 属性情報の取得
public void GetAuditInfo(Type type)
{
    var attributes = type.GetCustomAttributes(typeof(AuditAttribute), false);

    if (attributes.Length > 0)
    {
        var audit = (AuditAttribute)attributes[0];
        Console.WriteLine($"クラス {type.Name} は {audit.Author} によって作成されました。作
成日時: {audit.Created}");
    }

    foreach (var method in type.GetMethods())
    {
        var methodAttributes = method.GetCustomAttributes(typeof(AuditAttribute),
false);

        if (methodAttributes.Length > 0)
        {
            var audit = (AuditAttribute)methodAttributes[0];
            Console.WriteLine($"メソッド {method.Name} は {audit.Author} によって作成されま
した。作成日時: {audit.Created}");
        }
    }
}
```

高度なオブジェクト指向設計

デザインパターンの適用と実践的なOOP設計：

```
// シングルトンパターンの例
public class Logger
{
    private static Logger _instance;
```

```
    private static readonly object _lock = new object();

    // 外部からのインスタンス化を防ぐためのプライベートコンストラクタ
    private Logger()
    {
        // 初期化処理
    }

    public static Logger Instance
    {
        get
        {
            if (_instance == null)
            {
                lock (_lock)
                {
                    if (_instance == null)
                    {
                        _instance = new Logger();
                    }
                }
            }
            return _instance;
        }
    }

    public void Log(string message)
    {
        Console.WriteLine($"{DateTime.Now}: {message}");
    }
}

// ファクトリーパターンの例
public abstract class Document
{
    public abstract void Open();
    public abstract void Save();
    public abstract void Close();
}

public class PdfDocument : Document
{
    public override void Open() { /* PDFを開く処理 */ }
    public override void Save() { /* PDFを保存する処理 */ }
    public override void Close() { /* PDFを閉じる処理 */ }
}

public class WordDocument : Document
{
    public override void Open() { /* Wordを開く処理 */ }
    public override void Save() { /* Wordを保存する処理 */ }
    public override void Close() { /* Wordを閉じる処理 */ }
}

public class DocumentFactory
{
    public static Document CreateDocument(string fileExtension)
```

```
    {
        switch (fileExtension.ToLower())
        {
            case "pdf":
                return new PdfDocument();
            case "doc":
            case "docx":
                return new WordDocument();
            default:
                throw new ArgumentException($"サポートされていないファイル形式:
{fileExtension}");
        }
    }
}
```

依存性注入（DI）

疎結合で保守性の高いアプリケーション設計のためのテクニック：

```
// インターフェース
public interface ILogger
{
    void Log(string message);
}

public interface IEmailService
{
    void SendEmail(string to, string subject, string body);
}

// 実装
public class ConsoleLogger : ILogger
{
    public void Log(string message)
    {
        Console.WriteLine($"ログ: {message}");
    }
}

public class SmtpEmailService : IEmailService
{
    private readonly ILogger _logger;

    // コンストラクタインジェクション
    public SmtpEmailService(ILogger logger)
    {
        _logger = logger;
    }

    public void SendEmail(string to, string subject, string body)
    {
        // メール送信ロジック
        _logger.Log($"メール送信: 宛先={to}, 件名={subject}");
    }
}
```

```
// サービスクラス
public class UserService
{
    private readonly IEmailService _emailService;

    public UserService(IEmailService emailService)
    {
        _emailService = emailService;
    }

    public void RegisterUser(string email, string name)
    {
        // ユーザー登録ロジック

        // ウェルカムメールの送信
        _emailService.SendEmail(
            email,
            "ご登録ありがとうございます",
            $"{name}様、ご登録ありがとうございます。"
        );
    }
}

// 利用例
public class Program
{
    public static void Main()
    {
        // 依存関係を手動で設定
        ILogger logger = new ConsoleLogger();
        IEmailService emailService = new SmtpEmailService(logger);
        UserService userService = new UserService(emailService);

        userService.RegisterUser("user@example.com", "山田太郎");
    }
}
```

.NET Coreや.NET 5以降では、組み込みの依存性注入コンテナを使うこともできます：

```
public class Startup
{
    public void ConfigureServices(IServiceCollection services)
    {
        // サービスの登録
        services.AddSingleton<ILogger, ConsoleLogger>();
        services.AddScoped<IEmailService, SmtpEmailService>();
        services.AddTransient<UserService>();
    }
}
```

テスト駆動開発（TDD）

テストファーストで信頼性の高いコードを書くアプローチ：

```
// テスト対象のクラス
public class Calculator
{
    public int Add(int a, int b)
    {
        return a + b;
    }

    public int Subtract(int a, int b)
    {
        return a - b;
    }

    public int Multiply(int a, int b)
    {
        return a * b;
    }

    public int Divide(int a, int b)
    {
        if (b == 0)
        {
            throw new DivideByZeroException("0で除算することはできません");
        }
        return a / b;
    }
}

// NUnitを使用したテストクラス
[TestFixture]
public class CalculatorTests
{
    private Calculator _calculator;

    [SetUp]
    public void Setup()
    {
        _calculator = new Calculator();
    }

    [Test]
    public void Add_TwoPositiveNumbers_ReturnsCorrectSum()
    {
        // Arrange
        int a = 5;
        int b = 3;
        int expected = 8;

        // Act
        int result = _calculator.Add(a, b);

        // Assert
        Assert.AreEqual(expected, result);
    }

    [Test]
```

```
    public void Subtract_TwoNumbers_ReturnsCorrectDifference()
    {
        // Arrange
        int a = 5;
        int b = 3;
        int expected = 2;

        // Act
        int result = _calculator.Subtract(a, b);

        // Assert
        Assert.AreEqual(expected, result);
    }

    [Test]
    public void Multiply_TwoNumbers_ReturnsCorrectProduct()
    {
        // Arrange
        int a = 5;
        int b = 3;
        int expected = 15;

        // Act
        int result = _calculator.Multiply(a, b);

        // Assert
        Assert.AreEqual(expected, result);
    }

    [Test]
    public void Divide_TwoNumbers_ReturnsCorrectQuotient()
    {
        // Arrange
        int a = 6;
        int b = 3;
        int expected = 2;

        // Act
        int result = _calculator.Divide(a, b);

        // Assert
        Assert.AreEqual(expected, result);
    }

    [Test]
    public void Divide_ByZero_ThrowsDivideByZeroException()
    {
        // Arrange
        int a = 6;
        int b = 0;

        // Act & Assert
        Assert.Throws<DivideByZeroException>(() => _calculator.Divide(a, b));
    }
}
```

継続的な学習のためのリソース

C#と.NETの学習を継続するためのリソースをいくつか紹介します：

公式ドキュメント

- Microsoft Learn: C# ガイド
- .NET ドキュメント
- ASP.NET Core ドキュメント

Microsoft Learnは、初心者から上級者までを対象とした質の高い学習コンテンツを提供しています。対話型の学習モジュールもあり、実践的に学ぶことができます。

書籍

- 「C#プログラミング入門」（適切な日本語の書籍タイトル）
- 「実践C#」（適切な日本語の書籍タイトル）
- 「デザインパターン入門」（適切な日本語の書籍タイトル）

オンラインコース

- Udemy、Pluralsight、Coursera、edXなどのプラットフォームには、C#と.NETに関する多くのコースがあります。

コミュニティとフォーラム

- Stack Overflow - プログラミングに関する質問と回答のプラットフォーム
- Reddit r/csharp - C#に関するディスカッションフォーラム
- Microsoft Q&A - マイクロソフト製品に関する質問と回答のプラットフォーム

ブログとニュースサイト

- .NET Blog - マイクロソフトの.NETチームによる公式ブログ
- C# Corner - C#と.NETに関する記事、チュートリアル、ニュースを提供

オープンソースプロジェクト

オープンソースプロジェクトのソースコードを読むことは、実践的なコーディングスキルを学ぶ素晴らしい方法です：

- .NET Runtime - .NETのコアランタイム
- ASP.NET Core - ウェブアプリケーションとサービスのフレームワーク
- Roslyn - C#コンパイラプラットフォーム

GitHub上のプロジェクトに貢献することで、実際のプロジェクトでの経験を積むこともできます。

実践を通じた学習

学んだことを実践するためのいくつかのアイデアを紹介します：

個人プロジェクト

- **家計簿アプリケーション**: 収入と支出を記録し、レポートを生成する

- **タスク管理アプリ**: タスクの作成、編集、削除、フィルタリング機能を提供
- **ブログシステム**: 投稿の作成、コメント、カテゴリ分け、検索機能を実装
- **フィットネストラッカー**: 運動記録、目標設定、進捗グラフなどを実装
- **ゲーム**: 簡単なパズルゲームやカードゲームを作成

コーディングチャレンジ

- LeetCode
- HackerRank
- Exercism
- Project Euler

これらのプラットフォームでは、アルゴリズムやデータ構造に関する問題を解くことができます。問題を解くことで、プログラミングスキルを向上させることができます。

ハッカソンや勉強会への参加

地域のハッカソンや勉強会に参加することで、他の開発者と交流し、新しい技術やアイデアに触れることができます。

> **ベテランの知恵袋**: プログラミングスキルを向上させる最も効果的な方法は、実際にコードを書くことです。「読むだけ」では身につかないことが多いので、学んだことを常に実践しましょう。また、他の人のコードを読むことも重要です。優れたコードを読むことで、良い書き方や設計のパターンを学ぶことができます。最後に、知識を共有することも学習の素晴らしい方法です。他の人に教えることで、自分自身の理解も深まります。

専門分野の探求

C#は様々な分野で活用されています。自分の興味や目標に合わせて特定の分野を深く探求することで、専門性を高めることができます：

ウェブ開発

ASP.NET Core、Blazor、Razor Pagesなどを使ったウェブアプリケーション開発：

```
// ASP.NET Core MVC コントローラの例
public class ProductsController : Controller
{
    private readonly IProductRepository _repository;

    public ProductsController(IProductRepository repository)
    {
        _repository = repository;
    }

    public async Task<IActionResult> Index()
    {
        var products = await _repository.GetAllAsync();
        return View(products);
    }

    public IActionResult Create()
    {
```

```
            return View();
        }

        [HttpPost]
        [ValidateAntiForgeryToken]
        public async Task<IActionResult> Create(ProductViewModel model)
        {
            if (ModelState.IsValid)
            {
                await _repository.AddAsync(model.ToProduct());
                return RedirectToAction(nameof(Index));
            }
            return View(model);
        }
    }
```

デスクトップアプリケーション開発

WPF、Windows Forms、UWP、.NET MAUIなどを使ったデスクトップアプリケーション開発：

```
// .NET MAUI アプリケーションの例
public partial class MainPage : ContentPage
{
    public MainPage()
    {
        InitializeComponent();
        BindingContext = new MainViewModel();
    }

    private async void OnSaveButtonClicked(object sender, EventArgs e)
    {
        await DisplayAlert("保存", "データが保存されました", "OK");
    }
}

public class MainViewModel : INotifyPropertyChanged
{
    private string _name;

    public string Name
    {
        get => _name;
        set
        {
            if (_name != value)
            {
                _name = value;
                OnPropertyChanged();
            }
        }
    }

    public ICommand SaveCommand { get; }

    public MainViewModel()
    {
```

```
        SaveCommand = new Command(Save);
    }

    private void Save()
    {
        // 保存ロジック
    }

    public event PropertyChangedEventHandler PropertyChanged;

    protected virtual void OnPropertyChanged([CallerMemberName] string propertyName =
null)
    {
        PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(propertyName));
    }
}
```

クラウド開発

Azure FunctionsやAzure Web Appsなどを使ったクラウドアプリケーション開発：

```
// Azure Function の例
public static class HttpTriggerFunction
{
    [FunctionName("HttpTrigger")]
    public static async Task<IActionResult> Run(
        [HttpTrigger(AuthorizationLevel.Function, "get", "post", Route = null)]
HttpRequest req,
        ILogger log)
    {
        log.LogInformation("C# HTTP trigger function processed a request.");

        string name = req.Query["name"];

        string requestBody = await new StreamReader(req.Body).ReadToEndAsync();
        dynamic data = JsonConvert.DeserializeObject(requestBody);
        name = name ?? data?.name;

        return name != null
            ? (ActionResult)new OkObjectResult($"Hello, {name}")
            : new BadRequestObjectResult("Please pass a name on the query string or in
the request body");
    }
}
```

ゲーム開発

UnityエンジンとC#を使ったゲーム開発：

```
// Unity スクリプトの例
public class PlayerController : MonoBehaviour
{
    public float speed = 5f;
    private Rigidbody rb;
```

```
    void Start()
    {
        rb = GetComponent<Rigidbody>();
    }

    void Update()
    {
        float horizontalInput = Input.GetAxis("Horizontal");
        float verticalInput = Input.GetAxis("Vertical");

        Vector3 movement = new Vector3(horizontalInput, 0, verticalInput) * speed *
Time.deltaTime;
        transform.Translate(movement);

        if (Input.GetKeyDown(KeyCode.Space))
        {
            Jump();
        }
    }

    void Jump()
    {
        rb.AddForce(Vector3.up * 5f, ForceMode.Impulse);
    }

    void OnCollisionEnter(Collision collision)
    {
        if (collision.gameObject.CompareTag("Enemy"))
        {
            GameManager.Instance.LoseLife();
        }
    }
}
```

機械学習

ML.NETを使った機械学習アプリケーション開発：

```
// ML.NET の例
public static class SentimentAnalysisModel
{
    public static void Train()
    {
        // データのロード
        var data = new MLContext().Data.LoadFromTextFile<SentimentData>
("sentiment.csv", hasHeader: true, separatorChar: ',');

        // パイプラインの作成
        var pipeline = new MLContext().Transforms.Text.FeaturizeText("Features",
nameof(SentimentData.Text))
            .Append(new
MLContext().BinaryClassification.Trainers.SdcaLogisticRegression());

        // モデルのトレーニング
        var model = pipeline.Fit(data);
```

```
        // モデルの保存
        new MLContext().Model.Save(model, data.Schema, "sentiment_model.zip");
    }

    public static bool Predict(string text)
    {
        // モデルのロード
        var model = new MLContext().Model.Load("sentiment_model.zip", out var schema);

        // 予測エンジンの作成
        var predictor = new MLContext().Model.CreatePredictionEngine<SentimentData,
SentimentPrediction>(model);

        // 予測
        var prediction = predictor.Predict(new SentimentData { Text = text });

        return prediction.Prediction;
    }
}

public class SentimentData
{
    [LoadColumn(0)]
    public bool Sentiment { get; set; }

    [LoadColumn(1)]
    public string Text { get; set; }
}

public class SentimentPrediction
{
    [ColumnName("PredictedLabel")]
    public bool Prediction { get; set; }

    public float Score { get; set; }
}
```

キャリアパスとしてのC#開発

C#開発者として考えられるキャリアパスをいくつか紹介します：

1. **ジュニア開発者**
   - 基本的なC#プログラミングスキルと簡単なアプリケーション開発
   - チームメンバーの指導のもとでの実装タスク
2. **ミドルレベル開発者**
   - より複雑なアプリケーション開発と設計
   - コードレビューやメンタリングの提供
   - 特定の分野（ウェブ、デスクトップ、クラウドなど）での専門性
3. **シニア開発者**
   - アーキテクチャ設計と技術選定
   - 複雑な問題の解決とベストプラクティスの確立
   - チームリーダーシップとメンタリング

4. **テクニカルリード / アーキテクト**
   - システム全体のアーキテクチャ設計
   - 複数のチームにまたがる技術的な決定
   - 新技術の評価と導入
5. **開発マネージャー**
   - 開発チームのマネジメント
   - プロジェクト計画と進捗管理
   - リソース配分と予算管理
6. **専門コンサルタント**
   - 特定の技術領域（Azure、.NET最適化など）での専門的なアドバイス提供
   - クライアントへの技術ソリューションの提案
7. **起業家 / フリーランス**
   - 独自のサービスやプロダクトの開発
   - クライアントプロジェクトの請負

> **プロジェクト事例**: ある開発者は、C#の基礎から始めて、小規模なデスクトップアプリケーションの開発に携わりました。その後、ASP.NET CoreとAzureの学習に取り組み、企業向けのウェブアプリケーション開発に移行しました。数年間の実務経験を積んだ後、特にマイクロサービスアーキテクチャに興味を持ち、その分野で専門性を高めました。現在は、複数のチームにまたがる大規模なマイクロサービスプロジェクトのアーキテクトとして活躍しています。このように、継続的な学習と新しい挑戦によって、キャリアを着実に発展させることができます。

まとめ

本書では、C#プログラミングの基礎から応用までを幅広く学んできました。変数、制御構造、クラスといった基本的な概念から始まり、ファイル操作、データベース連携、GUIアプリケーション開発などの実践的なトピックまで、C#開発の多くの側面をカバーしました。

プログラミングは継続的な学習と実践が必要な分野です。本書で学んだ基礎をもとに、さらに知識を深め、実際のプロジェクトで経験を積んでいくことで、C#開発者としてのスキルを着実に向上させていくことができるでしょう。

最後に、プログラミングの旅を楽しんでください。問題解決の喜び、創造の楽しさ、継続的な成長の満足感を味わいながら、C#というパワフルな言語を使って素晴らしいソフトウェアを作り出していってください。

付録：C#の主要バージョン履歴

C#は継続的に進化しており、各バージョンで新しい機能や改善が追加されています。以下に主要なバージョンとその特徴をまとめます：

- **C# 1.0** (2002年)
  - 基本的なクラスと型
  - プロパティ、イベント、デリゲート
  - 演算子オーバーロード
- **C# 2.0** (2005年)
  - ジェネリック型とメソッド
  - 部分クラス
  - 匿名メソッド

    - イテレータ
    - Nullable型
- **C# 3.0** (2007年)
    - LINQ (Language Integrated Query)
    - ラムダ式
    - 暗黙的に型指定されたローカル変数（var）
    - オブジェクト初期化子
    - 匿名型
    - 拡張メソッド
- **C# 4.0** (2010年)
    - 動的バインディング（dynamic型）
    - 名前付き引数とオプションパラメータ
    - ジェネリック型の共変性と反変性
- **C# 5.0** (2012年)
    - 非同期プログラミング（async/await）
    - 呼び出し情報属性
- **C# 6.0** (2015年)
    - 文字列補間
    - Null条件演算子（?.）
    - nameofオペレーター
    - 式形式のメンバー
    - プロパティ初期化子
    - using static
- **C# 7.0-7.3** (2017-2018年)
    - タプル
    - パターンマッチング
    - ローカル関数
    - out変数
    - 参照戻り値と参照ローカル変数
    - 非同期Main
    - インライン変数宣言
    - パターンマッチングの拡張
- **C# 8.0** (2019年)
    - インターフェースのデフォルト実装
    - Nullable参照型
    - 非同期ストリーム（async/await foreach）
    - rangeとindex
    - パターンマッチングの拡張
    - using宣言
    - 静的ローカル関数
- **C# 9.0** (2020年)
    - レコード型
    - イニットのみのプロパティ
    - トップレベルステートメント
    - パターンマッチングの拡張

    - newの型推論
    - ターゲットとする型
    - コワリアント戻り値
- **C# 10.0** (2021年)
    - グローバルusing
    - ファイルスコープの名前空間
    - レコード構造体
    - null許容の参照フィールドに対する必須初期化子
    - ラムダ式の改善
    - 文字列補間の改善
- **C# 11.0** (2022年)
    - 必須のメンバー
    - 静的な抽象メンバー
    - ref フィールドとscoped ref
    - raw文字列リテラル
    - 引数null検査
    - パターンマッチングの拡張

継続的に新しいバージョンが開発されており、C#はより表現力豊かで強力な言語へと進化しています。最新の機能や改善点を学ぶことで、より効率的で読みやすいコードを書けるようになります。

これでC#入門ガイドは終了です。この先の学習と開発の旅の成功をお祈りしています！