Appendix

Java言語応用

紙幅の都合で取り上げられなかった話題を集めました。必要に応じて参照してください。

App.1 情報隠蔽

コード規模の拡大に対応するため情報隠蔽という考え方があります。情報隠蔽を支える言語機能として、スコープ、アクセス制御、ネストしたクラスを説明します。

App.1-1 情報隠蔽の必要性

規模の大きなプログラミングでは、一度に考えることをいかに減らすかが重要になります。コードの一部分を改変するために、プログラム全体を気にする必要があると、頭がいくらあっても足りないからです。

実装詳細を外部から隠すことを情報隠蔽と呼びます。開発者の工夫による情報隠蔽も可能ですが、言語の補助があると更に強力な情報隠蔽をできます(注1)。あるコードから呼び出せないメソッドや使えないクラスを定義できるからです。開発中「使えないイコール考えなくて良い」にできます。結果として下記の利点を享受できます。

- **開発者が考えることを少なくできる**
- **変わらない部分を保証できる**

App.1-2 スコープ

プログラミング言語の「スコープ(scope)」は識別子の使える範囲を意味します。

スコープの制限機能がないと、すべてのコードからすべての識別子を使えます。変数を例にするといわゆるグローバル変数と呼ばれる状態になります。この状態はコードサイズが一定サイズを超えると問題になります。問題の1つは変数名を使い回せない弊害です。1度どこかで使った変数名を使えなくなるので常に新しい変数名を考える必要があります。もう1つの問題は意図しない変数の使用です。知らない所で変数の値が変わってしまうバグは、発見も修正も困難です。

Javaのスコープと該当章をまとめます(**表1.1**)。

(注1) メソッドやクラスの分割、前章で説明したパッケージも情報隠蔽に使えます。命名の工夫も1例です。たとえば慣例的に_(アンダースコア文字)で始める命名は狭い可視性を意図します。

表1.1 Javaのスコープ

識別子の種類	スコープの説明	本書の章
トップレベルの型(クラスなど)	同一パッケージ内の全域	6章 クラス
インポートした型	import行以後の型宣言内の全域	18章 パッケージ
staticインポートしたクラスメンバ	import行以後の型宣言内の全域	18章 パッケージ
型宣言(クラス宣言など)内のメンバ(フィールド、メソッド、ネストしたクラスなど)	その型宣言内の全域	6章 クラス
「メソッド、コンストラクタ、ラムダ式」のパラメータ変数	該当「メソッド、コンストラクタ、ラムダ式」の全域	6章 クラス
「メソッド、コンストラクタ、ラムダ式」内で宣言した変数や型名	該当「メソッド、コンストラクタ、ラムダ式」内の宣言行以後の全域	4章 変数とオブジェクト
ブロック内で宣言した変数や型名	宣言行以後のブロック内	4章 変数とオブジェクト
for文内で宣言した変数	for文内。for文の他の変数宣言式を含む	13章 Javaプログラムの実行と制御構造
try-with-resources文内で宣言した変数	try-with-resources文のtry節内。他のリソース変数の変数宣言式を含む	14章 例外処理
try文のcatch節内で宣言した変数	catch節内	14章 例外処理
パターン変数	パターンマッチ時に実行されるコード内	12章 文、式、演算子
パッケージ名(javaパッケージ以外)	モジュール間の公開設定次第	App.2 モジュール
ジェネリック型の型変数	そのジェネリック型の宣言内の全域。型パラメータ列含む	19章 ジェネリック型
「ジェネリックメソッドとジェネリックコンストラクタ」の型変数	該当「ジェネリックメソッドとジェネリックコンストラクタ」の全域。型パラメータ列含む	19章 ジェネリック型

App.1-3 アクセス制御

アクセス制御は、クラス名などの型名およびクラスの構成要素(メンバとコンストラクタ)の可視性を制御できる言語機能です。適切にアクセス制御を使うと堅牢なコードになります(注2)。

以後、型宣言をクラス宣言に限定して説明します。レコードクラス、enum型、インタフェース、アノテーション型の宣言でも同じ説明が成り立ちます。

C O L U M N

情報隠蔽のレベル

アクセス制御を使う情報隠蔽は、開発者が読もうと思えばコードを読めるレベルの情報隠蔽です。更に進んだ情報隠蔽は実装コード自体の隠蔽です。

この隠蔽のための実装技法として、基底型(インタフェースなど)と実装クラスを分離して、基底型のみを公開する方法が一例です(「**11-1-6 メソッドの返り値の型をインタフェース型にする意義**」参照)。

情報隠蔽のレベルは上げれば良いという単純な話ではありません。隠蔽の結果、トラブル発生時の調査の障壁になる場合もあるからです。

(注2) アクセス制御を使いオブジェクトの状態(データ)の適正さを守る技法をカプセル化と呼びます。情報隠蔽が可視性を主眼にした技法なのに対し、カプセル化はデータの制約(constraint)や不変条件(invariant)に主眼があります。

App.1-3-1　アクセス制御の構文

クラス自身およびクラス内の構成要素のアクセス制御に使う修飾子をまとめます（**表1.2**、**表1.3**）。

表1.2　トップレベルのクラスのアクセス制御

修飾子	意味
public	グローバル可視。同一パッケージおよび他パッケージからアクセス可能
なし	パッケージ可視。同一パッケージ内からのみアクセス可能

表1.3　クラス内の構成要素のアクセス制御

修飾子	意味
public	グローバル可視。同一パッケージおよび他パッケージからアクセス可能
protected	そのクラス、派生クラス、および同一パッケージ内の他クラスからアクセス可能
なし	パッケージ可視。そのクラスおよび同一パッケージ内の他クラスからアクセス可能
private	そのクラスからのみアクセス可能

アクセス制御に違反したコードはコンパイルエラーになります。たとえばprivate修飾子のついたメンバは該当クラス内のコードからしか使えません。他クラスのコードから使うとコンパイルエラーになります（**リスト1.1**）。

リスト1.1　private修飾子の使用

```
class My {
    private void method() { /* 本体省略 */ }
}

// クラス外からアクセス不可（コンパイルエラー）
class Other {
    void callMyMethod(My my) {
        my.method();
    }
}
|  Error:
|  method() has private access in My
```

■アクセス制御の使い分け

アクセス制御の使い分けの原理原則は「可能な限り狭い可視性にする」です。より具体的な指針を示します。

- フィールドは基本的にprivateにする
- 特定のクラス内でしか使わない下請け処理メソッドをprivateにする
- テンプレートメソッドパターンで派生クラスにオーバーライドさせたいメソッドをprotectedにする（「**17-4-6 テンプレートメソッドパターン**」参照）
- パッケージ内で他から呼ぶメソッドをパッケージ可視にする
- パッケージ外から呼ぶ必要があるメソッドをpublicにする

■オブジェクト可視性との違い

クラスのアクセス制御でしばしば誤解される点を書いておきます。

クラスのアクセス制御で可視性が変わるのはクラス名などの型名です。オブジェクトではありません。クラス名をアクセスできないコードからでも、該当クラスのオブジェクトを参照可能です。ただし、そのオブジェクトを参照する変数の型を、アクセス制御が許す基底型にする必要があります。

具体例は標準ライブラリにも存在します。たとえばList.ofメソッドが1例です。List.ofメソッドが返すオブジェクトの具象クラスのアクセス制御は、java.utilパッケージ内のパッケージ可視です(注3)。このため、java.util以外のパッケージのコードからこのオブジェクトの具象クラス名は使えません。しかしオブジェクトは使えます。「**8章 コレクションと配列**」などで見たように、オブジェクトを参照する変数の型をListインタフェースにします。Listインタフェースはパブリック可視で他パッケージから問題なく使えます。

オブジェクト自体の可視性を制御する構文は存在しません。型名のアクセス制御や変数名のスコープなどの可視性を通じた間接制御のみです。

■同一クラスの別オブジェクトからのprivateメンバアクセス

同じクラスの別オブジェクトからprivateメンバにアクセスできます。privateの語感の直感に反します。**リスト1.2**のmy2オブジェクトに対するメソッド呼び出しで、my1オブジェクトのprivateフィールドにアクセスしています。

リスト1.2　同一クラスの別オブジェクトからはprivateフィールドにアクセス可能

```
class My {
    private final String field; // privateフィールド
    My(String field) {
        this.field = field;
    }
    void method(My other) {
        System.out.println(other.field);
    }
}
```

```
// 使用例
// 変数my2の参照先オブジェクトから変数my1の参照先オブジェクトのprivateフィールドは可視
jshell> var my1 = new My("my1 field");
jshell> var my2 = new My("my2 field");
```

（注3）　java.util.ImmutableCollectionsクラスのメンバです。

```
jshell> my2.method(my1)
my1 field
```

App.1-4 ネストしたクラス

App.1-4-1 ネストしたクラス

ネストしたクラスとは、クラスの中で宣言するクラスです。ネストしたクラスと区別するため、他のクラスの中にないクラスをトップレベルクラスと呼びます。

ネストしたクラスがある時、外側のクラスを「エンクロージングクラス」と呼びます。ネストしたクラスの中に更に別のネストしたクラスの宣言も可能です。ネストの階層の数に制限はありません。ネストしたクラスとエンクロージングクラスは相対的な役割です。エンクロージングクラスは必ずしもトップレベルクラスとは限りません。

ネストしたクラスは次の4種類に分類できます（**リスト1.3**）。

- staticなネストしたクラス
- 非staticなネストしたクラス
- ローカルクラス
- 匿名クラス

リスト1.3　ネストしたクラス

```
class MyHost { // ネストしたクラスから見たときのエンクロージングクラス
    static class Nested { // staticなネストしたクラス
        /* 本体省略 */
    }
    class Inner { // 非staticなネストしたドクラス
        /* 本体省略 */
    }
    void method() {
        class Local { // ローカルクラス
            /* 本体省略 */
        }

        var obj = new Object() { // 匿名クラス
            /* 本体省略 */
        };
    }
}
```

最初の2つはエンクロージングクラスのメンバとして宣言するクラスです。コード的にはフィールドやメソッドと同列の形でクラスを宣言します。メンバクラスという呼び方もあります。

本書はネストしたクラスで用語を統一します。

ローカルクラスと匿名クラスはメソッド内などで宣言して使うクラスです。後ほど説明します。

■ネストしたクラスの用途

ネストしたクラスを使う主な用途は次のとおりです。

- **ソースファイルの増殖をおさえたい場合（名前の節約）**
- **エンクロージングクラス内部だけでオブジェクトを使いたい場合（下請け処理）**
- **ネストしたクラスの実装をエンクロージングクラス内に隠蔽したい場合（実装クラスの隠蔽）**

具体例を含めて説明します。

■名前の節約

クラス内でメンバにつけるstatic修飾子の働きは一貫しています。static修飾子のついたメンバはクラスに属し、つかないメンバはオブジェクトに属します。

static修飾子のあるネストしたクラスの動作もこの原則どおりです。具体例を**リスト25.4**に示します。概念的にはMyHost.Myという名前のクラスを宣言したかのように解釈できます。一貫してこの名前を使うコードを書けば、トップレベルで宣言したクラスと同じように使えます。違いはアクセス制御の選択肢の多さぐらいです。

リスト1.4　staticなネストしたクラス

```
class MyHost {
    static class My {
        void method() {
            System.out.println("MyHost.Myのメソッド");
        }
    }
}
```

```
// 使用例
// 変数の型を省略可能ですが説明のために明示します
jshell> MyHost.My my = new MyHost.My()
jshell> my.method()
MyHost.Myのメソッド
```

エンクロージングクラスとネストしたクラスの両方にpublic修飾子を付与すると、ネストしたクラスはパッケージを超えて使えます。

非privateのstaticなネストしたクラスは、実質的にトップレベルのクラス名の節約につながります。結果としてソースファイルの増殖を抑制できます。

■下請け処理

staticなネストしたクラスをprivateにすると、エンクロージングクラスのコード内からのみ使えるクラスになります。クラス内の下請け処理などに使えます。なおトップレベルクラスをプライベート可視にする手段は存在しません。

具体例を**リスト1.5**に示します。MyHostクラスのコード内でMyHelperクラスを使えます。MyHelperクラスはプライベート可視なので他のクラスのコードからは使えません。

リスト1.5　プライベート可視のネストしたクラス（下請け処理）

```
class MyHost {
    // ヘルパークラス
    private static class MyHelper {
        void helperMethod() {
            System.out.println("MyHelperクラスのメソッド");
        }
    }

    void method() {
        // ヘルパークラスのオブジェクト使用
        var helper = new MyHelper();
        helper.helperMethod();
    }
}
```

```
// 使用例
jshell> var my = new MyHost()
jshell> my.method()
MyHelperクラスのメソッド
```

■実装クラスの隠蔽

アクセス制御の対象は型名の可視性で、そのオブジェクトの可視性ではない点を説明しました。

プライベート可視のネストしたクラスのもう1つの典型利用例が実装クラスの隠蔽です。オブジェクトをクラスの外に渡しつつ実装クラスを隠蔽します。オブジェクトをクラスの外に渡す方法は主に2つあります。メソッドの返り値として返すか、メソッド呼び出しの実引数に使うかです。型名がプライベート可視なので必然的に外部に渡す時の型は、インタフェースか基底クラスになります。プライベート可視のクラスのオブジェクトをメソッドの引数として渡す例を**リスト1.6**に示します。

リスト1.6　プライベート可視のネストしたクラス（実装クラスの隠蔽）

```
class MyHost {
    // プライベート可視のネストしたクラス
    private static class StringLengthComparator implements Comparator<String> {
        @Override
        public int compare(String s1, String s2) {
            return s1.length() - s2.length();
        }
    }

    void method() {
        // StringLengthComparatorオブジェクトを外部に渡す
        // 外部からStringLengthComparatorクラスのクラス名は不可視
        // 外部（sortedメソッド内）で見える型名はComparatorインタフェース
        List<String> sortedList = Stream.of("abc", "xyz", "za", "defghi")
                                        .sorted(new StringLengthComparator()).toList();
        System.out.println(sortedList);
    }
}
```

```
// 使用例
jshell> var my = new MyHost()
jshell> my.method()
[za, abc, xyz, defghi]
```

後ほどリスト1.6をラムダ式で書き換える例を紹介します。本目的の多くはラムダ式でも記述可能だからです。

■ネストしたクラスとエンクロージングクラスのアクセス制御

ネストしたクラスは、エンクロージングクラスのメンバとして扱われます。このためエンクロージングクラスのprivateフィールドやprivateメソッドにアクセスできます。逆も同じで、エンクロージングクラスはネストしたクラスのprivateフィールドやprivateメソッドにアクセスできます（**リスト1.7**と**リスト1.8**）。

リスト1.7　エンクロージングクラスからネストしたクラスのprivateフィールドにアクセス

```
class MyHost { // エンクロージングクラス
    // ネストしたクラス
    private static class MyHelper {
        private static final String cField = "MyHelperのクラスフィールド";
        private final String iField = "MyHelperのインスタンスフィールド";
    }

    void method() {
        System.out.println(MyHelper.cField);

        var myHelper = new MyHelper();
        System.out.println(myHelper.iField);
    }
}
```

```
// 使用例
jshell> var myHost = new MyHost()
jshell> myHost.method()
MyHelperのクラスフィールド
MyHelperのインスタンスフィールド
```

リスト1.8　ネストしたクラスからエンクロージングクラスのprivateフィールドにアクセス

```
class MyHost { // エンクロージングクラス
    // ネストしたクラス
    static class My {
        void method() {
            System.out.println(MyHost.cField);

            var myHost = new MyHost();
            System.out.println(myHost.iField);
        }
    }

    private static final String cField = "MyHostのクラスフィールド";
    private final String iField = "MyHostのインスタンスフィールド";
}
```

```
// 使用例
jshell> var my = new MyHost.My()
jshell> my.method()
MyHostのクラスフィールド
MyHostのインスタンスフィールド
```

ネストしたクラスとエンクロージングクラスのお互いのクラスフィールドには、「クラス名.フィールド名」でアクセスできます。インスタンスフィールドにアクセスするには、「(オブジェクトの)参照変数名.フィールド名」でアクセスできます。わかりやすさのためにフィールドで説明しましたが、メソッドの相互の呼び出しについても同様です。

App.1-4-2　非staticなネストしたクラス

非staticなネストしたクラスは、内部クラス(inner class)の1つです。内部クラスは、非staticなネストしたクラスの総称です(注4)。

内部クラスの非staticの意味は、エンクロージングクラスのオブジェクト(エンクロージングオブジェクト)との次の結びつきです。

(注4)　言語仕様上は「非staticなネストしたクラス」「インスタンスメソッド内で宣言したローカルクラスもしくは匿名クラス(どちらも暗黙的に非static)」の総称が内部クラスです。

• 内部クラスのオブジェクトは、エンクロージングオブジェクトへの参照を暗黙的に持つ

static有無の違いはこれだけです。内部クラス固有の上記の性質を必要としない限り、ネストしたクラスにstatic修飾子をつけてください。内部クラスはオブジェクト間に余分な依存をもたらすからです。

内部クラスとエンクロージングクラスの結びつきの具体例を示します(**リスト1.9**)。

リスト1.9　内部クラスとエンクロージングクラスの結びつき

```
// エンクロージングクラス
class MyHost {
    // static修飾子のないネストしたクラス(内部クラス)
    private class MyHelper {
        private final String sameNameField = "MyHelperの同名インスタンスフィールド";

        private void helperMethod() {
            // 内部オブジェクトのインスタンスメソッドから
            // エンクロージングオブジェクトのインスタンスフィールドが可視
            System.out.println(MyHost.this.iField);

            // 同名のフィールドの場合
            System.out.println(sameNameField); // 内部オブジェクトのフィールド。this.を記述しても同じ
            System.out.println(MyHost.this.sameNameField); // エンクロージングオブジェクトのフィールド
        }
    }

    private final String iField;
    private final String sameNameField = "MyHostの同名インスタンスフィールド";

    MyHost(String iField) {
        this.iField = iField;
    }

    void method() {
        // 内部クラスのオブジェクト(内部オブジェクト)を生成
        var helper = new MyHelper();
        helper.helperMethod();
    }
}
```

```
// 使用例
jshell> var my = new MyHost("MyHostのインスタンスフィールド")
jshell> my.method()
MyHostのインスタンスフィールド
MyHelperの同名インスタンスフィールド
MyHostの同名インスタンスフィールド
```

リスト1.9のmethod内で、内部クラス(MyHelperクラス)のオブジェクトを生成します。以後、

これを内部オブジェクトと呼びます。この内部オブジェクトは、エンクロージングオブジェクトへの参照を暗黙的に持ちます。

内部オブジェクトのインスタンスメソッドの中から、エンクロージングオブジェクトのインスタンスメンバにアクセス可能です。アクセスの記法は エンクロージングクラス名.this.フィールド名（もしくはメソッド名）です。リスト1.9のMyHost.this.iFieldが該当コードです。リスト1.7との微妙な違いを確認してください。

内部オブジェクトからエンクロージングオブジェクトのインスタンスメンバがそのまま見えるということは、同名のフィールドやメソッドがある場合に名前が被ることを意味します。この場合、名前の隠蔽が起きます。隠蔽時、内部オブジェクトのメンバ名を優先します。

■非privateな内部クラス

内部クラスを非privateにすると、エンクロージングクラスの外側で内部オブジェクトの生成が可能になります。エンクロージングオブジェクトの参照にドット文字とnewを続ける構文になります。**リスト1.10**のmyHost.new MyHelper()が該当コードです。実開発で見る機会は少ないですが、知らないと読めない構文なので紹介します。

リスト1.10　非privateな内部クラス

```
class MyHost {
    // static修飾子のないネストしたクラス（非privateな内部クラス）
    class MyHelper {
        void helperMethod() {
            System.out.println(MyHost.this.iField);
        }
    }

    private final String iField;
    MyHost(String iField) {
        this.iField = iField;
    }
}
```

```
// 使用例
jshell> var myHost = new MyHost("エンクロージングオブジェクトのフィールド")

// 内部オブジェクト生成
jshell> var myHelper = myHost.new MyHelper()

// 内部オブジェクトのメソッド呼び出し
jshell> myHelper.helperMethod()
エンクロージングオブジェクトのフィールド
```


App.1-4-3　その他のネストした型

■ネストしたインタフェース、レコードクラス、enum型

ネストして宣言されたインタフェース、レコードクラス、enum型は、修飾子staticがあろうとなかろうと常にstaticです。慣習的にstatic修飾子を省略します。常にstaticなので内部クラスのような仕組みは存在しません。

■インタフェース内のネストした型

インタフェース内のネストした型は、修飾子の有無に関わらず常にpublicかつstaticです。

App.1-4-4　ローカルクラス

ローカルクラスは、メソッド内、コンストラクタ内、初期化ブロック内、if節などのブロック内で宣言するクラスです。便宜上、説明をブロックで代表します。

ローカルクラス名は、宣言したブロックの外からはアクセスできません。アクセス制御用の修飾子は意味がないため付与できません。ローカルクラスにstatic修飾子を書くとコンパイルエラーになります。

インスタンスメソッドやコンストラクタ内などで宣言したローカルクラスは、エンクロージングクラスのインスタンスメンバにアクセスできます。実例を**リスト1.11**に示します。My.this.fieldでエンクロージングオブジェクトのフィールドにアクセスしています。この動きから、ローカル内部クラスと呼んで区別する場合があります。

クラスメソッドまたはstatic初期化ブロック内で宣言したローカルクラスは、エンクロージングオブジェクトのフィールドに上記のようなアクセスはできません。static修飾子はありませんが、概念上、staticなクラスと見なしても問題ありません。

リスト1.11　インスタンスメソッド内で宣言したローカルクラス

```
class My {
    void method() {
        // ローカルクラスの宣言
        class LocalClass {
            private final String lField = "ローカルクラスのフィールド";
            void localMethod() {
                System.out.println("ローカルクラスのメソッド呼び出し");
                // ローカルクラスのオブジェクトは
                // エンクロージングオブジェクトのフィールドにアクセス可能
                System.out.println(My.this.field);
            }
        }

        // ローカルクラスのオブジェクト使用
```

```
        var lc = new LocalClass();
        lc.localMethod();
        System.out.println(lc.lField);
    }

    private final String field = "エンクロージングクラスのフィールド";
}
```

```
// 使用例
jshell> var my = new My()
jshell> my.method()
ローカルクラスのメソッド呼び出し
エンクロージングクラスのフィールド
ローカルクラスのフィールド
```

ローカルクラスを使う主な用途は次のとおりです。

- **ブロック内部だけでオブジェクトを使いたい場合**
- **実装クラスをブロック内に隠蔽したい場合**

1番目の例はリスト1.11で包含できるので省略します。2番目の目的の例を示します（**リスト1.12**）。コードの意図はリスト1.6と同じです。StringLengthComparatorのクラス名はブロック内のみ可視で、ブロックの外側ではComparatorインタフェース型の変数でオブジェクトを参照します。

リスト1.12　ローカルクラスの使用例（実装クラスの隠蔽）

```
class My {
    void method() {
        // ローカルクラス
        // このクラス名はブロック内のみ有効
        class StringLengthComparator implements Comparator<String> {
            @Override
            public int compare(String s1, String s2) {
                return s1.length() - s2.length();
            }
        }
        // 外部からStringLengthComparatorクラスのクラス名は不可視
        // 外部（sortedメソッド内）で見える型名はComparatorインタフェース
        List<String> sortedList = Stream.of("abc", "xyz", "za", "defghi").
            sorted(new StringLengthComparator()).toList();
        System.out.println(sortedList);
    }
}
```



```
// 使用例
jshell> var my = new My()
jshell> my.method()
[za, abc, xyz, defghi]
```

■ローカルクラスとローカル変数

便宜上、ローカルクラスを宣言したエンクロージングクラスのメソッドを、エンクロージングメソッドと呼びます。

エンクロージングメソッド内の実質的finalなローカル変数を、ローカルクラス内で使えます。再代入のあるローカル変数を使うとコンパイルエラーになります。この規則「実質的finalなローカル変数のみ使用可能」は、ラムダ式および後で説明する匿名クラスと同様です。

通常、メソッドの実行を完了すると、メソッド内のローカル変数は消滅します。しかし、ローカルクラスのオブジェクトがエンクロージングメソッドのローカル変数を使う場合、ローカルクラスのオブジェクトは該当ローカル変数の値を保持します。エンクロージングメソッドを抜けた後であってもです。このあたりの挙動は「**9-3-4 ラムダ式自体をreturn文で返す**」で説明したラムダ式と同じ動きです。

■ローカルインタフェース、ローカルレコードクラス、ローカルenum型

言語仕様上、ローカルクラスと同じようにローカルインタフェース、ローカルレコードクラス、ローカルenum型を使えます。

これらは暗黙的かつ強制的にstatic扱いです。static修飾子を記述するとコンパイルエラーになります。常にstatic扱いなのでローカル内部クラスのような仕組みは存在しません。

エンクロージングメソッドがインスタンスメソッドの時、ローカルクラスのみが非static相当で、ローカルインタフェースなどはstaticです。見た目と動作に一貫性がなくなります。たとえばstaticなローカルインタフェースは、エンクロージングクラスのインスタンメンバやエンクロージングメソッドの実質的finalローカル変数を使えません。非staticなローカルクラスはこれらを使えます。ただ、この動作の違いはそれほど大きな問題にはなりません。コンパイル時に検出できるからです。

App.1-4-5　匿名クラス

匿名クラスはクラス名なしで宣言できるクラスです。クラス宣言と同時にオブジェクトを生成する場合に限り使えます。次の構文で匿名クラスのオブジェクトを生成します。

```
new 基底型(実引数) {
    メソッド宣言とフィールド宣言の差分実装
}
```

クラスもしくはインタフェースを基底型としてnew式に渡し、基底型との差分実装を書き足

します。差分実装を書き足して定義したクラスが匿名クラスです。

匿名クラスのnew式を記述できるのはオブジェクトの参照を使う場所です。具体的には代入式の右辺やメソッド呼び出しの実引数やメソッドの返り値などです。

匿名クラスの宣言場所により、static相当の匿名クラスと非static相当の匿名内部クラスがあります。違いはローカルクラスの場合と同じなので説明を省略します。匿名クラスにstatic修飾子やアクセス制御用修飾子を記述できない事情も同じです。

匿名クラスの使用例を**リスト1.13**に示します。sortedメソッドの実引数に匿名クラスのオブジェクトを渡しています。コードの意図はリスト1.6やリスト1.12と同じです。匿名クラスを使うコードの多くは、変数に代入せず、メソッドの実引数やメソッドの返り値にそのままオブジェクト参照を使います。

リスト1.13　リスト1.12を匿名クラスで書き換えた例

```
class My {
    void method() {
        List<String> sortedList = Stream.of("abc", "xyz", "za", "defghi").
            // 匿名クラスのオブジェクトをsortedメソッドの実引数に渡す
            sorted(new Comparator<String>() {
                @Override
                public int compare(String s1, String s2) {
                    return s1.length() - s2.length();
                }
            }).toList();
        System.out.println(sortedList);
    }
}
```

```
// 使用例
jshell> var my = new My()
jshell> my.method()
[za, abc, xyz, defghi]
```

匿名クラスを使う基準は次のようになります。

- **コンストラクタが不要（必要であれば初期化ブロックを使用可能）**
- **オブジェクト作成が1箇所だけ**

■匿名クラスを活用したオブジェクト初期化

「**8-8-1 コレクションオブジェクトの初期化記法**」でコレクションオブジェクトを生成と同時に初期化するコード例を紹介しました。**リスト1.14**に再掲します。

リスト1.14　Listオブジェクトを生成と同時に初期化（リスト8.22の再掲）

```
var list = new ArrayList<String>() {
          { // 初期化ブロック
              add("abc");
              add("def");
              add("ghi");
          }
    }
```

リスト1.14のnew式の評価値は匿名クラスのオブジェクトです。基底型がArrayListで、初期化ブロックを差分実装として書き足しています。初期化ブロックはオブジェクト生成時に実行されます。結果として、オブジェクトを生成しつつ同時に初期化処理をしています。

■匿名クラスとラムダ式

リスト1.13はラムダ式で書き直せます。省略記法を使わないコードを**リスト1.15**、省略記法で書いたコードを**リスト1.16**に示します。

リスト1.15　リスト1.13をラムダ式で書き換え

```
class My {
    void method() {
        // 省略記法を使わないラムダ式
        List<String> sortedList = Stream.of("abc", "xyz", "za", "defghi").
            sorted((String s1, String s2) -> {
                return s1.length() - s2.length();
            }).toList();
        System.out.println(sortedList);
    }
}
```

```
// 使用例
jshell> var my = new My()
jshell> my.method()
[za, abc, xyz, defghi]
```

リスト1.16　省略記法を活用したラムダ式

```
class My {
    void method() {
        List<String> sortedList = Stream.of("abc", "xyz", "za", "defghi")
            .sorted((s1, s2) -> s1.length() - s2.length())
            .toList();
        System.out.println(sortedList);
    }
}
```

リスト1.13とリスト1.15を見比べると匿名クラスとラムダ式は記法が異なるだけに見えます。しかし内部実装は別です。記法の違いとは考えず別物と考えてください。内部実装の違いから、ラムダ式に下記2つの固有な性質があります。

- **常にメソッド1つ**
- **ラムダ式の中で使うthis参照の参照先はエンクロージングオブジェクト（「9-3-3 ラムダ式の詳細」参照）**

世の中には、コールバック実装を匿名クラスで実装する慣例のフレームワークがあります。コールバック実装側に状態管理が不要かつ実装メソッドが1つであれば、多くの場合ラムダ式でも記述可能です。

App.2  モジュール

複数パッケージをまとめて配布できる仕組みをモジュールと呼びます。作った成果物の配布と他チームの作った成果物の再利用の観点からモジュールを説明します。

App.2-1 モジュール

モジュールは複数パッケージをまとめる仕組みです。モジュールの位置づけはJavaプログラムの配布形態と関連します。まずJavaプログラムの配布形態の現状を説明します。

App.2-1-1 アプリとライブラリ

便宜上、配布Javaプログラムをアプリとライブラリに分類します。起動エントリポイントを持ちそこから実行する想定のプログラムをアプリと呼称します。他のアプリまたはライブラリに組み込んで使う意図のプログラムをライブラリと呼称します。

App.2-1-2 ライブラリ配布の実際

標準ライブラリ以外のライブラリ（サードパーティライブラリと呼びます）を使う場合、開発者が該当ライブラリを入手する必要があります(注1)。開発に必要なのはライブラリのクラスファイルです。ソースファイルはあってもなくてもかまいません。

ある規模以上のJava開発の場合、ライブラリ入手の手動実施は少数派でしょう。必要なライブラリを自動ダウンロードできるツールが存在しているからです。代表的なツールはMavenとGradleです。ライブラリが別のライブラリに依存している推移的な依存関係もこれらのツールで解決できます。

MavenとGradleでは、配布物をアーティファクトという単位で管理します。配布物の管理サーバをアーティファクトレポジトリと呼びます。ライブラリ開発者はこれらのツールの流儀でライブラリを配布します。

配布効率と実行効率から複数クラスファイルを1つのjarファイルにまとめて配布するのが通例です。jarファイルをクラスパスに指定すると、コンパイル時や実行時にjarファイル内のクラスファイルを探索できます。

（注1） 標準ライブラリはJavaの開発環境に含まれているので追加の入手作業は不要です。

App.2-1-3　アプリ配布の実際

Javaプログラムの動作に必要なのはクラスファイルです。すべての依存クラスファイルがローカルのクラスパスで見つかれば、Javaプログラムを実行可能です。

理論上は、アプリ利用者がMavenやGradleなどのツールで依存ライブラリを入手してアプリを使う方法も可能です。しかしこの配布手段はあまり採用されません。アプリ利用者の負担が大きいからです。通常、依存ライブラリ一式を含んだ形でアプリを配布します。

App.2-1-4　モジュールの狙い

MavenとGradleはJavaの仕様の外側で進化したツールです。デファクトスタンダードのツールですがJava仕様に沿った手段ではありません。

ライブラリ依存の解決を狙うJava公式の機能がモジュールです。

MavenとGradleの依存性解決の単位であるアーティファクトはjarファイルの集合です。jarファイルは複数パッケージのクラスファイルの集合です。この意味では比較的大きな単位で依存を管理します。一方、モジュールはパッケージ単位、場合によってはクラス単位の依存を管理できます。理論上、より細かい依存管理が可能です。

App.2-1-5　モジュールの実際

MavenとGradleには前述したアーティファクトレポジトリを筆頭にエコシステムがすでに存在しています。ライブラリの検索からダウンロードまでの作業フローが確立しています。このためモジュールは既存ツールを置き換えるものではなく共存していく流れになっています。

App.2-2　モジュールの使い方

モジュールはパッケージに比べるとやや暗黙の存在です。package宣言文のような明示的な記述もなく、ディレクトリ名にも使いません。

モジュールを定義するにはmodule-info.javaというファイルを作ります。次の構文のファイルです。

```
[open] module モジュール名 {
    モジュール指示;
}
```


予約語のmoduleに続いて記載するモジュール名は任意の識別子です。パッケージ名同様.（ドット文字）で区切って名前を一意にする努力義務があります。パッケージ名とモジュール名は別の名前空間です。このためパッケージ名と同名でもかまいません。パッケージ名と同名もしくはパッケージ名の部分文字列のモジュール名にするのが慣例です。

module-info.javaに記述するモジュール指示の詳細を**表2.1**に示します。記述必須のモジュール指示は存在しません。必要な指示のみ記述すれば十分です。ただしexports指示は実質的には必須です。exports指示のないmodule-info.javaを作ると、他モジュールから何の機能も使えないモジュールになるからです。

表2.1　モジュール指示

モジュール指示	説明
exports 指定パッケージ名 [to 他モジュール名]	モジュール内の指定パッケージを他モジュールへ公開。to以後を省略した場合はすべての他モジュールへ公開
requires 他モジュール名	他モジュールへの依存を明示
requires static 他モジュール名	他モジュールへの依存を明示（コンパイル時のみ依存）
requires transitive 他モジュール名	他モジュールへの依存を明示（他モジュールの依存先にも依存）
opens 指定パッケージ名 [to 他モジュール名]	モジュール内の指定パッケージを実行時のみ公開。具体的には他モジュールからリフレクション可能になる(注2)
provides 指定型名 with 具象型名	モジュール内の具象型をServiceLoader経由で公開。本書は詳細を省略します
uses 指定型名	依存先モジュールの指定型をServiceLoader経由で使用。本書は詳細を省略します

module-info.javaファイルをソースファイルのトップディレクトリに配置して、他のソースファイ

C O L U M N

複数バージョン共存問題

既存の依存性解決ツールの問題として、バイナリ互換性のない複数バージョンの共存問題があります。次のような問題です。

アプリがライブラリAとライブラリBに依存、ライブラリBが内部でライブラリAに依存しているとします。2つのライブラリAが同一バージョンであれば問題ありません。どちらの依存から解決しても実行時にロードするのは同じクラスファイルだからです。

2つのライブラリAのバージョンが別でかつバイナリ互換性がない場合は実行時にエラーもしくは不正動作をします。どちらのバージョンのクラスファイルをロードするかはクラスパス次第だからです。

残念ながらこの問題はモジュールを使っても解決できません。1つのクラスローダが同一の完全修飾名のクラスを1つしかロードできないのが原因だからです。バイナリ互換性を失う場合、ライブラリ側で完全修飾名を変更するのが推奨されています。この推奨に従っていないライブラリを使う場合、アプリ側でライブラリの完全修飾名を変更する細工が必要です。

（注2）　moduleの前にopenを記述すると、モジュール内の全パッケージに対してopens指示をした場合と同様の動作になります。

ルと一緒にコンパイルします。こうするとコンパイル時に依存先モジュールのチェックが走ります。

module-info.javaのクラスファイル（module-info.classファイル）を配布物に含めると、その配布物がモジュール対応になります。配布物を使用する側のツールが、コンパイル時および実行時にmodule-info.classを見て依存をチェックします。

module-info.javaファイルの作成は必須ではありません。ファイルが存在しない場合、全パッケージを公開、他の全モジュール依存を意味します。疑似コードで書くと**リスト2.1**相当の動作になります。

リスト2.1　module-info.javaファイルが存在しない場合に相当する疑似コード

```
open module モジュール名 {
    exports *;
    requires *;
}
```

モジュールを使う現実的な利点は下記です。下記利点に限定してモジュールの使い方を説明します。

- **パッケージ単位のアクセス制御**
- **Java実行環境を内包したアプリ配布**

App.2-3　パッケージ単位の可視性制御

App.2-3-1　解決したい課題

トップレベルの型名のアクセス制御はグローバル可視とパッケージ可視しかありません（「**App.1-3 アクセス制御**」参照）。必然的に他のパッケージから使いたい型のアクセス制御はpublicになります。

一定規模以上のプログラムは、パッケージを分割して開発するのが普通です。これはライブラリであっても同様です。この場合、ライブラリ内でパッケージ間にまたがって使う機能をpublicにします。しかしこのpublic機能をライブラリの外側には隠したい場合があります。ライブラリが外部に公開したい機能と必ずしも一致するとは限らないからです。

App.2-3-2　モジュールを使う解決法

モジュールのexports指示を使うと、配布物から外部に公開したい機能を指定できます。公開単位はパッケージです。モジュールの使用前と使用後の実例を比較します。

App.2-3-3　モジュール使用前の例

アプリの開発ディレクトリを**図2.1**、ライブラリの開発ディレクトリを**図2.2**のファイル構成にします。モジュール使用前はmodule-info.javaファイルとmodule-info.classファイルがない状態を想定してください。

図2.1　アプリの開発ディレクトリ

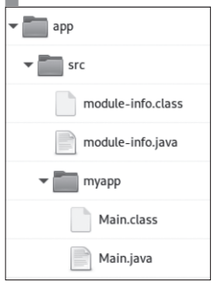
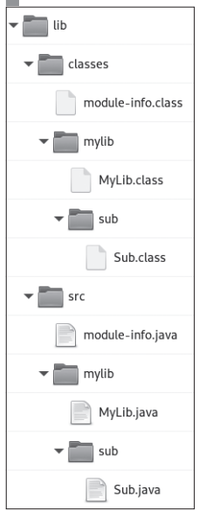


図2.2　ライブラリの開発ディレクトリ



アプリの開発ディレクトリの説明を**表2.2**に示します。クラスファイルをソースファイルと同じディレクトリに出力します(注3)。

表2.2　図2.1（アプリ開発）のファイル構成

ディレクトリ名	説明
app/	ライブラリを使う想定のアプリの開発ディレクトリ
app/src/	アプリのソースコードのトップディレクトリ
app/src/myapp/	パッケージmyappのディレクトリ

ライブラリの開発ディレクトリの説明を**表2.3**に示します。ライブラリ開発チームは、アプリ開発チームと別に存在すると想定してください。jarファイル作成の都合のため、クラスファイルの出力先を分けています。

（注3）　実開発ではクラスファイルの出力先ディレクトリを分けるのが普通です。

表2.3　図2.2（ライブラリ開発）のファイル構成

ディレクトリ名	説明
lib/	ライブラリのアプリの開発ディレクトリ
lib/src/	ライブラリのソースコードのトップディレクトリ
lib/src/mylib/	パッケージmylibのディレクトリ
lib/src/mylib/sub/	パッケージmylib.subのディレクトリ
lib/classes/	ライブラリのクラスファイルの出力ディレクトリ

App.2-3-4　ソースコードの例

図2.2のライブラリ側のコード例を示します（**リスト2.2、リスト2.3**）(注4)。

リスト2.2　lib/src/mylib/MyLib.java

```
package mylib;
import mylib.sub.Sub;

// ライブラリ外に公開する想定のクラス
public class MyLib {
    private MyLib() {}

    public static void method() {
        Sub.method("MyLib");
    }
}
```

リスト2.3　lib/src/mylib/sub/Sub.java

```
package mylib.sub;

// ライブラリ内のみで使う想定のクラス
// 上位のmylibパッケージだけに公開したいがpublic指定が必要
public class Sub {
    private Sub() {}

    public static void method(String caller) {
        System.out.println("MyLib.Sub.method: 呼び出し元は" + caller);
    }
}
```

ライブラリ開発チームの意図が下記だとします。

- mylibパッケージをライブラリの外部に公開する想定
- mylib.subパッケージをライブラリの外部から使わない想定
- mylibパッケージからmylib.subパッケージのSubクラスを使う想定。このためにSubクラスをpublic指定

（注4）　説明の簡略化のためクラスメソッドのみ使います。インスタンスメソッドでも事情は同じです。

ライブラリ開発チームの意図を守るかどうかは、アプリ開発チームの善意のみです。アプリ側のコード例を示します（**リスト2.4**）。使おうと思えばアプリ側のコードからmylib.sub.Subクラスを使用できてしまいます。

リスト2.4　app/src/myapp/Main.java

```
package myapp;
import mylib.MyLib;
import mylib.sub.Sub; // モジュールによる制御がないとアクセス可能

public class Main {
    public static void main(String... args) {
        MyLib.method();
        Sub.method("Main");
    }
}
```

App.2-3-5　コマンド例

ここまでの想定で使うコマンド例を示します。コマンドそのものではなく出力に注目してください。

コマンド例の理解に必要な最小限のコマンドラインオプションを**表2.4**にまとめます。

表2.4　コマンド例を理解するための説明

コマンド	オプション	意味
javac	-d ディレクトリ名	クラスファイルの出力先ディレクトリの指定
javac	-cp ディレクトリ名またはjarファイル名	クラスファイルを探すディレクトリまたはjarファイルの指定（複数指定可）
javac	-p ディレクトリ名またはjarファイル名	モジュールを探すディレクトリまたはjarファイルの指定（複数指定可）
java	-cp ディレクトリ名またはjarファイル名	クラスファイルを探すディレクトリまたはjarファイルの指定（複数指定可）
java	-p ディレクトリ名またはjarファイル名	モジュールを探すディレクトリまたはjarファイルの指定（複数指定可）
java	-m モジュール名:メインクラス名	実行メインクラスの指定（正確には初期モジュールの指定）
jar	--create	jarファイル作成モード
jar	-d	jarファイルの中身の検証モード
jar	--file ファイル名	jarファイル名を指定
jar	-C ディレクトリ名 .	指定ディレクトリに移動後、そのディレクトリの全クラスファイルをまとめる

下記がライブラリ開発チームのコマンド例です。javacコマンドでクラスファイル（MyLib.class）を生成します。jarコマンドでクラスファイルをjarファイル（mylib.jar）にまとめます。

```
$ cd lib/src
$ javac -d ../classes mylib/MyLib.java
$ cd ..
$ jar --create --file ../mylib.jar -C classes .
```

下記がアプリ開発チームのコマンド例です。上記ファイルを直接相対パスで参照しています。チームが別組織であれば上記のjarファイルをMavenやGradleなどのツールで取得したと仮定してください。javacコマンドでコンパイル、javaコマンドで実行する例です。

```
$ cd app/src
$ javac -cp ../../mylib.jar myapp/Main.java
$ java -cp ../../mylib.jar:. myapp/Main
```

App.2-3-6　モジュール使用後の例

ライブラリ開発チームの意図は、ライブラリ外部に対するmylib.subパッケージ非公開でした。モジュールを使い、mylib.subパッケージをアプリ側コードで使えないようにします。

ライブラリ開発チームは**リスト2.5**のmodule-info.javaを作成します。exports指示でmylibパッケージのみをライブラリ外部に公開設定します。

リスト2.5　lib/src/module-info.java

```
module mylib.module {
    exports mylib;
}
```

モジュール名の命名は開発者の自由です。パッケージ名と同名にする場合も多いのですが、区別のためにmylib.moduleという命名にしました。この命名は一般的ではないので注意してください。

ライブラリ開発チームのコマンド例を示します。モジュール使用前との違いはmodule-info.javaをコンパイルする点です。jarファイルがmodule-info.classファイルを含みます(注5)。

```
$ cd lib/src
$ javac -d ../classes mylib/MyLib.java module-info.java
$ cd ..
$ jar --create --file ../mylib.jar -C classes .
```

アプリ開発チームのコマンド例を示します。上記のjarファイルを参照して下記コマンドでコンパイルします。myapp/Main.javaファイルはリスト2.4のままだとします。

```
$ cd app/src
$ javac -cp ../../mylib.jar myapp/Main.java module-info.java
myapp/Main.java:3: error: package mylib.sub is not visible
```

(注5)　module-info.classを含むjarファイルを、モジュール対応jarファイルと呼びます。

上記コンパイルはエラーになります。このエラー発生は意図した結果です。ライブラリ開発チームの意思でmylib.subパッケージの外部公開を禁止したからです。リスト2.4のMain.javaからmylib.Subクラス依存コードを削除すると、下記のようにコンパイルと実行が可能になります。

```
$ cd app/src
$ javac -cp ../../mylib.jar myapp/Main.java
$ java -cp ../../mylib.jar:. myapp/Main
```

App.2-3-7　アプリのモジュール対応

前項の例ではアプリ側にmodule-info.javaファイルを作成していません。コマンド実行時にクラスパス指定を使い続ける限り、アプリ側のmodule-info.java作成は必須ではありません。

アプリ側にmodule-info.javaファイルを作ると、アプリ側もモジュール対応になります(**リスト2.6**)。

リスト2.6　app/src/module-info.java

```
module myapp.module {
    requires mylib.module;
}
```

アプリをモジュール対応した場合、アプリ側のjavacコマンド実行時に、クラスパス(-cp)ではなくモジュールパスを-pで指定する必要があります。クラスパス指定のままではコンパイルエラーになります。

```
$ cd app/src
$ javac -p ../../ myapp/Main.java module-info.java
```

実行時も-pでモジュールパスを指定します。-mオプションでメインクラスを指定します。指定方法は「モジュール名/メインクラスの完全修飾名」です。クラスパスの場合はjarファイルのファイルパスを指定しますが、モジュールパスの場合はjarファイルが存在するディレクトリを指定できます。なおjarファイルのファイルパスをモジュールパスに指定しても動作します。

```
$ cd app/src
$ java -p ../../:. -m myapp.module/myapp.Main
```

App.2-3-8　モジュール対応・非対応の混在

モジュール対応･非対応に応じてコマンドラインオプションが変わる説明をしました。モジュール対応・非対応の混在について整理します。

モジュール非対応アプリからモジュール対応ライブラリの使用には特別な対処が不要です。クラスパスにライブラリのjarファイルを指定すれば動作します。

モジュール対応アプリからモジュール非対応ライブラリの使用には少し注意が必要です(注6)。ライブラリのモジュール名を知る必要があるからです。モジュール非対応ライブラリのモジュール名の調査方法の1例としてjarコマンドの使用例を紹介します。下記コマンド実行例の mylib automatic はモジュール非対応ライブラリが自動的にmylibというモジュール名になったことを意味しています。

```
$ jar -d --file mylib.jar
No module descriptor found. Derived automatic module.

mylib automatic
requires java.base mandated
contains mylib
contains mylib.sub
```

C O L U M N

jmod形式ファイル

モジュールをまとめるファイル形式のjmod形式を紹介します。jmodファイルは複数クラスファイルをまとめるjarと同じ役割に加えモジュール間の依存関係の情報も保持します。

jarファイルの場合、クラスパスに並べた順番の中で単に最初に見つかったクラスファイルを使います。jmodファイルの場合、モジュールパス上のjmodファイルの依存関係を解決してからコンパイルをできます。

本文のjarファイルを扱うコマンド例をjmodファイルに置き換える例を紹介します。

```
$ jar --create --file ../mylib.jar -C classes .
の代わりに
$ jmod create --class-path classes ../mylib.jmod

$ javac -p ../../mylib.jar myapp/Main.java module-info.java
の代わりに
$ javac -p ../.. myapp/Main.java module-info.java
```

（注6）　原則はツールに任せてください。アプリをモジュール対応する必要性の検討もしてください。

App.2-4　Java実行環境を内包したアプリ配布

App.2-4-1　解決したい課題

現状の多くのJavaアプリは、Javaアプリ利用者のコンピュータにJava実行環境、具体的にはjavaコマンドや標準ライブラリがインストールされていることを想定しています。このためJavaアプリ利用者は、Javaアプリのインストール作業と合わせてJava実行環境のインストール作業も必要になります。このインストール作業は、ただアプリを使いたいだけの人には負担です(注7)。

App.2-4-2　Java実行環境を内包した配布物

Java実行環境を内包した配布物を作ると上記課題の負担を軽減できます。Javaの標準ライブラリ自体がモジュール化され配布サイズも小さくできるようになっています。

Java実行環境込みの配布物を生成するにはjlinkコマンドを使います。前節の説明で使ったアプリ配布物を生成するコマンド例を示します。jlinkコマンドの-pオプションでモジュールパスを指定します。--add-modulesオプションで配布物に含めるモジュールを指定します。ライブラリのモジュール指定は不要です。依存モジュールを自動検出するからです。

```
$ cd app/src
$ jlink -p ../../:. --add-modules myapp.module --output myapp-dist
```

outputオプションで指定したディレクトリに配布に必要なファイルが生成されます。次のように実行可能です。

```
$ cd app/src/myapp-dist
$ bin/java -m myapp.module/myapp.Main
```

より配布サイズを小さくできるオプション例を示します。個別オプションの意味はヘルプを参照してください。

```
$ jlink -p ../../:. --add-modules myapp.module --no-header-files --no-man-pages --strip-debug
--launcher app=myapp.module/myapp.Main --output myapp-dist
```

（注7）　Dockerなどのコンテナ技術の利用も1つの解決法です。コンテナを使う場合でもJava実行環境のサイズを小さくする価値はあります。

App.3 リフレクション

Javaプログラム実行中にオブジェクトの型情報、たとえばどんなメソッドがあるかなど、を取得できるリフレクションの説明をします。リフレクションを使うと実行時に未知のクラスをロードしたり、クラスの情報を調べたりできます。章の最後にリフレクションの応用例を紹介します。

App.3-1 リフレクション

リフレクションはプログラム実行中に型情報を取得し、型そのものを操作対象にできる仕組みです。型とは、クラス、インタフェース、配列型、基本型、アノテーション型（「**App.4 アノテーション**」）を指します。

型を操作対象にすると次のようなことができます。

- **クラスのロード**
- **クラスの型階層の列挙**
- **クラスや配列からオブジェクト生成**
- **クラスやインタフェースの構成要素（フィールドやメソッドやコンストラクタなど）の型や名前の取得**
- **フィールド値の読み書きやメソッド呼び出し**
- **アノテーション型の情報の読み出し**
- **プロクシクラスによる処理の差し込み**

App.3-2 Classクラス

リフレクションを実現する仕掛けの中心はClassクラスです。名前が紛らわしいですがjava.lang.Classという名前のクラスです。

Classクラスを直感的に理解するには、StringやStringBuilderなど個別クラスごとにそれぞれ1つオブジェクトが自動的に存在するクラスを考えてください。つまりStringクラスに対応するただ1つのClassオブジェクト、StringBuilderクラスに対応するただ1つのClassオブジェクトなどです。後述するクラスローダを考慮するとこの説明は正確ではありませんが、ひとまずこの理解でかまいません。

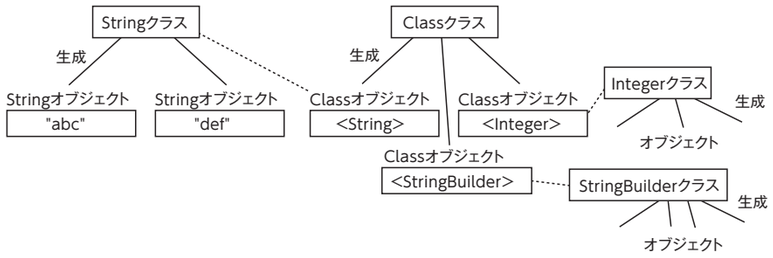
App.3-2-1 Classオブジェクト

Classクラスを雛型としてClassオブジェクトが生成されます。しかしnew式でClassオブジェクトは生成できません。Classオブジェクトは何らかのクラスをロードすると自動的に生成されます。たとえばStringクラスを使うコード（Stringオブジェクトや Stringクラスの使用）を動かすと、裏でStringクラスに対応するClassオブジェクトが自動生成されます(注1)。

以後、Foo型に対応するClassオブジェクトをClass<Foo>オブジェクトと書きます。たとえばStringクラスであればClass<String>オブジェクトと記載します。

Stringクラスを雛型として複数のStringオブジェクトが存在するように、Classクラスを雛型として複数のClassオブジェクトが存在します。Class<String>オブジェクトやClass<StringBuilder>オブジェクトなどです（**図3.1**）。なお、Classクラスに対応するClass<Class>オブジェクトも存在します。

図3.1　ClassクラスとClassオブジェクト



■Classオブジェクトの取得手段

Classオブジェクトの参照の取得方法を示します。

- **クラスリテラル**
- **Objectクラスの getClass メソッド**
- **Classクラスの forNameクラスメソッド**
- **その他（Classクラスや ClassLoaderクラスのメソッド経由）**

Classオブジェクトが既に存在すればそのClassオブジェクトの参照を得られます。存在しない場合にClassオブジェクトの参照を得ようとすると、クラスローダが対象クラスをロードしてClassオブジェクトを生成します。クラスローダとは名前のとおりクラスをロードする役目を担うオブジェクトです。Classオブジェクトの取得は、内部的にはクラスローダが保持するClass

（注1）　mainメソッドの引数にString型を使うので、Stringに対応するClassオブジェクトは実行時に必ず自動生成されます。

オブジェクト群からの検索になります。

クラスリテラルは、String.classのように型名に.classを後置する記法です。型名にはクラス名、インタフェース名、基本型名を記述可能です。すべて.classの後置記法でクラスリテラルになります。クラスリテラルは1つのリテラル値として評価されます。評価結果はClassオブジェクトです。

すべてのオブジェクトに対してgetClassメソッドを呼べます。メソッドの返り値はClassオブジェクトです。getClassメソッドはObjectクラスのfinal修飾子のついたメソッドです(「**17-2-6 Objectクラス**」参照)。

ClassクラスのforNameクラスメソッドは、型名の完全修飾名を文字列で渡してClassオブジェクトを取得できます。forNameメソッドにクラスローダオブジェクト(java.lang.ClassLoaderオブジェクト)を渡すメソッドも存在します。クラスローダオブジェクトを渡す例は後ほど紹介します。

上記3つの手段でClass<String>オブジェクトの参照を取得するコードを**リスト3.1**に示します。Class<String>オブジェクトは1つのクラスローダに1つしか存在しないので、リスト3.1の3つの参照は同一になります。

リスト3.1　Class<String>オブジェクトの取得

```
// Class型の変数名には伝統的にclazzまたはklassをよく使います
// classは予約語で変数名に使えないからです

// 下記3つはStringクラスに対応する同一のClassオブジェクト
jshell> Class<String> clazz1 = String.class
jshell> Class<?> clazz2 = "".getClass()
jshell> Class<?> clazz3 = Class.forName("java.lang.String")

// 同じオブジェクトの場合、System.identityHashCodeが同じ値になります
// 出力数値は環境により異なります
jshell> int hcode1 = System.identityHashCode(clazz1)
hcode1 ==> 1676062616
jshell> int hcode2 = System.identityHashCode(clazz2)
hcode2 ==> 1676062616
jshell> int hcode3 = System.identityHashCode(clazz3)
hcode3 ==> 1676062616

// ==演算の結果が真
jshell> boolean result = clazz1 == clazz2
result ==> true
jshell> boolean result = clazz2 == clazz3
result ==> true
```

自作クラスのClassオブジェクトも取得可能です。レコードクラス、enum型、インタフェース、アノテーション型のClassオブジェクト取得例を含めて紹介します(**リスト3.2**)。

リスト3.2　Classオブジェクトの種類

```
// クラスのClassオブジェクト
jshell> class MyClass {}
jshell> Class<MyClass> clazz = MyClass.class

// レコードクラスのClassオブジェクト
jshell> record MyRecord() {}
jshell> Class<MyRecord> clazz = MyRecord.class
jshell> boolean result = clazz.isRecord()
result ==> true

// enum型のClassオブジェクト
jshell> enum MyEnum {}
jshell> Class<MyEnum> clazz = MyEnum.class
jshell> boolean result = clazz.isEnum()
result ==> true

// インタフェースのClassオブジェクト
jshell> interface MyInterface {}
jshell> Class<MyInterface> clazz = MyInterface.class
jshell> boolean result = clazz.isInterface()
result ==> true

// アノテーション型のClassオブジェクト
jshell> @interface MyAnnotation {}
jshell> Class<MyAnnotation> clazz = MyAnnotation.class
jshell> boolean result = clazz.isAnnotation()
result ==> true
```

■基本型のClassオブジェクト

int型などの基本型にも対応するClassオブジェクトが存在します(**リスト3.3**)。int型に対応するClassオブジェクトの型は概念上はClass<int>です。ジェネリック型の制約から参照する変数の型はClass<Integer>になっています。本書は概念上の理屈のままClass<int>と記述します。

intに対応するClass<int>オブジェクトとIntegerクラスに対応するClass<Integer>オブジェクトは異なるオブジェクトです。Class<int>オブジェクトをClassクラスのforNameメソッドで取得する手段は存在しません。

リスト3.3　基本型のClassオブジェクト

```
// 下記2つはint型に対応する同一のClassオブジェクト
jshell> Class<Integer> clazz1 = int.class
jshell> Class<Integer> clazz2 = Integer.TYPE
jshell> boolean result = clazz1 == clazz2
result ==> true
```

```
// 下記はIntegerクラスのClassオブジェクト
jshell> Class<Integer> clazz3 = Integer.class

// Class<int>オブジェクトとClass<Integer>オブジェクトは別
jshell> boolean result = clazz1 == clazz3
result ==> false
```

■配列型のClassオブジェクト

配列型に対応するClassオブジェクトの参照の取得例を**リスト3.4**に示します。配列の要素型ごとに異なるClassオブジェクトになります。forNameメソッドに渡す文字列には特殊な規則があります。たとえばint型配列であれば"[I"、String型配列であれば"[Ljava.lang.String;"です。この文字列はClassオブジェクトのgetNameメソッドで得られます。

リスト3.4　配列型のClassオブジェクト

```
// String型要素の配列
jshell> String[] strArr = {}

// 下記3つは要素型がStringの配列型に対応する同一のClassオブジェクト
jshell> Class<String[]> clazz1 = String[].class
jshell> Class<?> clazz2 = strArr.getClass()
jshell> Class<?> clazz3 = Class.forName("[Ljava.lang.String;")

// 配列の要素型に対応するClassオブジェクトはClass<String>
jshell> Class<?> elementType = clazz1.getComponentType()
jshell> boolean result = elementType == String.class
result ==> true

// forNameメソッドに渡す文字列の取得
jshell> String name = clazz1.getName()
name ==> "[Ljava.lang.String;"
```

要素が基本型の配列、2次元配列それぞれのClassオブジェクトの例を示します(**リスト3.5**、**リスト3.6**)。

リスト3.5　要素が基本型の配列に対応するClassオブジェクト

```
jshell> Class<int[]> clazz4 = int[].class

// forNameメソッドに渡す文字列の取得
jshell> String name3 = clazz4.getName()
name3 ==> "[I"

// 配列の要素型に対応するClassオブジェクトはClass<int>
jshell> Class<?> elementType = clazz4.getComponentType()
jshell> boolean result = elementType == int.class
result ==> true
```

リスト3.6　2次元配列に対応するClassオブジェクト

```
jshell> Class<?> clazz5 = String[][].class

// forNameメソッドに渡す文字列の取得
jshell> String name2 = clazz5.getName()
name2 ==> "[[Ljava.lang.String;"

// 配列の要素型に対応するClassオブジェクトはClass<String[]>
jshell> Class<?> elementType1 = clazz5.getComponentType()
jshell> boolean result = elementType1 == String[].class
result ==> true

// 配列の要素の要素に対応するClassオブジェクトはClass<String>
jshell> Class<?> elementType2 = elementType1.getComponentType()
jshell> boolean result = elementType2 == String.class
result ==> true
```

■ジェネリック型のClassオブジェクト

ジェネリック型は型引数の型が異なっても同じClassオブジェクトになります(**リスト3.7**)。正確には型変数の境界の型でClassオブジェクトが決まります(**リスト3.8**)。内部的にジェネリック型は境界の型ごとにクラスが存在するからです(「**19章 ジェネリック型**」参照)。

リスト3.7　ジェネリック型に対応するClassオブジェクト

```
jshell> Class<ArrayList> clazz1 = ArrayList.class
jshell> Class<?> clazz2 = Class.forName("java.util.ArrayList")

// 型引数がString
jshell> var slist = new ArrayList<String>()
jshell> Class<?> clazz3 = slist.getClass()

// 型引数がInteger
jshell> var ilist = new ArrayList<Integer>()
jshell> Class<?> clazz4 = ilist.getClass()

// 上記4つは同一のClassオブジェクト
jshell> boolean result = clazz1 == clazz2
result ==> true

jshell> boolean result = clazz2 == clazz3
result ==> true

jshell> boolean result = clazz3 == clazz4
result ==> true
```

リスト3.8　境界のある型引数のジェネリック型に対応するClassオブジェクト

```
jshell> class MyClass1<E> {}  // 暗黙的に E extends Object
jshell> class MyClass2<E extends String> {}

jshell> Class<?> clazz1 = MyClass1.class
jshell> Class<?> clazz2 = MyClass2.class

// 型引数の境界が異なる場合、別のClassオブジェクト
jshell> boolean result = clazz1 == clazz2
result ==> false
```

Listインタフェースに対応するClassオブジェクトと ArrayListクラスに対応するClassオブジェクトは、異なるオブジェクトです（**リスト3.9**）。

リスト3.9　Class<List>とClass<ArrayList>は別オブジェクト

```
jshell> Class<?> clazz1 = List.class
clazz1 ==> interface java.util.List

jshell> Class<?> clazz2 = ArrayList.class
clazz2 ==> class java.util.ArrayList

jshell> boolean result = clazz1 == clazz2
result ==> false
```

Classオブジェクトはオブジェクトの型で決まります。オブジェクトを参照する変数の型は無関係です（**リスト3.10**）。

リスト3.10　Classオブジェクトはオブジェクトの型で決まる

```
jshell> List<String> list1 = new ArrayList<>()
jshell> ArrayList<String> list2 = new ArrayList<>()

jshell> boolean result = list1.getClass() == list2.getClass()
result ==> true
```

その他、Javaのコードで変数から参照できる対象（ネストしたクラス、ラムダ式、メソッド参照など）はどれもClassオブジェクトを取得可能です。実例の紹介は省略します。

App.3-2-2　型情報の取得

型の様々な情報をClassオブジェクトから取得可能です。クラスやインタフェースの構成要素を取得する代表的なメソッドを示します（**表3.1**）。

表3.1　Classオブジェクトの代表的なメソッド（メソッドのthrows節は省略）

メソッド	説明
String getName ()	型の文字列を返す
Field [] getFields ()	すべてのpublicフィールド情報を返す（継承したフィールドも含む）
Field getField (String name)	指定した名前のpublicフィールドを返す（継承したフィールドも含む）
Field [] getDeclaredFields ()	すべてのフィールド情報を返す（継承したフィールドを含まない）
Field getDeclaredField (String name)	指定した名前のフィールドを返す（継承したフィールドを含まない）
Method [] getMethods ()	すべてのpublicメソッド情報を返す（継承したメソッドも含む）
Method getMethod (String name,Class<?>... parameterTypes)	指定したシグネチャのpublicメソッドを返す（継承したメソッドも含む）
Method [] getDeclaredMethods ()	すべてのメソッド情報を返す（継承したメソッドを含まない）
Method getDeclaredMethod (String name,Class<?>... parameterTypes)	指定したシグネチャのメソッドを返す（継承したメソッドを含まない）
Constructor<?> [] getConstructors ()	すべてのpublicコンストラクタ情報を返す
Constructor<T> getConstructor (Class<?>... parameterTypes)	指定したシグネチャのpublicコンストラクタを返す
Constructor<?> [] getDeclaredConstructors ()	すべてのコンストラクタ情報を返す
Constructor<T> getDeclaredConstructor (Class<?>... parameterTypes)	指定したシグネチャのコンストラクタを返す
RecordComponent[] getRecordComponents()	レコードコンポーネントを返す
T[] getEnumConstants()	enum定数を返す
Class<?> [] getClasses ()	ネストしたpublicクラスおよびpublicインタフェースを返す（継承したクラス、インタフェースも含む）
Class<?> [] getDeclaredClasses ()	ネストしたクラスおよびインタフェースを返す（継承したクラス、インタフェースを含まない）

表3.1のMethod、Field、Constructorなどは名前のとおりメソッド、フィールド、コンストラクタを表現するオブジェクトです。これらのオブジェクトの使用例は後ほど説明します。

Methodオブジェクトと Fieldオブジェクトは、インスタンスメンバとクラスメンバでクラスが分かれていません。代わりに、Methodオブジェクトや Fieldオブジェクトの getModifiersメソッドの返り値で判定します。getModifiersメソッドの返り値は java.lang.reflect.Modifierの定数のビット和です（**リスト3.11**）。getModifiersメソッドの返り値に Modifier.STATICビットが立っていれば、そのメンバはクラスメンバです。

リスト3.11　java.lang.reflect.Modifierの定数（Modifier.javaを整形して抜粋）

```
public class Modifier {
    public static final int PUBLIC           = 0x00000001;
    public static final int PRIVATE          = 0x00000002;
    public static final int PROTECTED        = 0x00000004;
    public static final int STATIC           = 0x00000008;
    public static final int FINAL            = 0x00000010;
    public static final int SYNCHRONIZED     = 0x00000020;
    public static final int VOLATILE         = 0x00000040;
    public static final int TRANSIENT        = 0x00000080;
    public static final int NATIVE           = 0x00000100;
    public static final int INTERFACE        = 0x00000200;
    public static final int ABSTRACT         = 0x00000400;
```

```
    public static final int STRICT          = 0x00000800;
}
```

■型階層の情報

型階層に関係する情報を取得するメソッドを紹介します（**表3.2**）。使用例を**リスト3.12**に示します。Objectクラス、インタフェース型、基本型に対応するClassオブジェクトには継承元がないので、getSuperclassメソッドがnullを返します。配列型に対応するClassオブジェクトのgetSuperclassはClass<Object>オブジェクトを返します。

表3.2　Classオブジェクトから型階層を取得するメソッド

メソッド	説明
Class<? super T> getSuperclass ()	拡張継承の基底クラスのClassオブジェクトを返す
Class<?>[] getInterfaces ()	継承元インタフェースのClassオブジェクトを返す
Class<?>[] getPermittedSubclasses()	シールクラスの派生クラスのClassオブジェクトを返す

リスト3.12　型階層の情報

```
// Class<String>のgetSuperclassメソッドはClass<Object>を返す
jshell> Class<String> clazz1 = String.class
jshell> Class<?> clazz2 = clazz1.getSuperclass()
clazz2 ==> class java.lang.Object

// Stringクラスの継承元インタフェースのClassオブジェクト列挙
jshell> Arrays.stream(clazz1.getInterfaces()).forEach((Class<?> e) -> System.out.println(e))
interface java.io.Serializable
interface java.lang.Comparable
interface java.lang.CharSequence
interface java.lang.constant.Constable
interface java.lang.constant.ConstantDesc
```

App.3-3　オブジェクト生成と操作

App.3-3-1　オブジェクト生成

リフレクションを使ってオブジェクトを生成できます。これはnew式を使わないオブジェクト生成手段になります。new式との違いを生かした応用例は後ほど説明します。

次の手段でオブジェクトを生成できます。

- **Constructorクラスのnewlnstanceメソッド呼び出し**
- **Arrayクラスのnewlnstanceメソッド呼び出し**

Classクラスの forNameメソッドを組み合わせると、ソースコードにまったく記述がないクラスのオブジェクトを生成できます。Constructorクラスのnewlnstanceメソッドを使いStringBuilderオブジェクトを生成する例を示します（**リスト3.13**）。

リスト3.13　リフレクションによるオブジェクト生成

```
import java.lang.reflect.*;

public class Main {
    public static void main(String... args) {
        if (args.length == 0) {
            return;
        }

        try {
            Class<?> clazz = Class.forName(args[0]);
            // String.classはStringBuilderのコンストラクタの引数の型
            Constructor<?> constructor = clazz.getConstructor(String.class);
            // StringBuilderオブジェクトの生成
            var sb = constructor.newInstance("abc");
            System.out.println(sb);
        } catch (ReflectiveOperationException e) {
            e.printStackTrace();
        }
    }
}
```

```
// 実行方法
$ java Main.java java.lang.StringBuilder
```

リスト3.13に識別子としてのStringBuilderが存在しない点に注目してください。StringBuilderに関係する情報はすべて文字列なので、実行中にファイルから"java.lang.StringBuilder"という文字列を読み込むなど、プログラムの外部から与えられます。

この例だけではコードをいたずらに複雑化した黒魔術に見えるかもしれません。しかしこの技法を応用すると、コードを書く時ではなく実行時に実装クラスを決める柔軟性を得られます。

App.3-3-2　メソッド呼び出し

リフレクションでメソッドを呼ぶにはjava.lang.reflect.Methodクラスを使います。Classオブジェクトから Methodオブジェクトを取得する方法は表3.1を参照してください。

Methodオブジェクトに対してinvokeメソッドを呼んでメソッド呼び出しができます。invokeメソッドの定義を示します。

```
// Methodクラスのinvokeメソッドの定義（throws節は省略）
Object invoke(Object obj, Object... args)
```

invokeメソッドの第1引数でメソッド呼び出しのレシーバオブジェクトを指定します。通常のメソッド呼び出し時のobj.method()のobjに相当するオブジェクトを渡します。クラスメソッドに対応するMethodオブジェクトの場合、第1引数は無視されます。通常、nullを渡します。第2引数以降の可変長引数でメソッドの実引数を渡します。通常のメソッド呼び出しがobj.method("abc")であれば、"abc"に相当する部分です。

invokeメソッドでStringBuilderのメソッドを呼び出す例を**リスト3.14**に示します。

リスト3.14　リフレクションによるメソッド呼び出し

```
jshell> import java.lang.reflect.*

// Class<StringBuilder>オブジェクト取得
jshell> Class<?> clazz = Class.forName("java.lang.StringBuilder")

// 引数なしコンストラクタに対応するConstructorオブジェクト取得
jshell> Constructor<?> constructor = clazz.getConstructor()

// Methodオブジェクト取得
jshell> Method appendMethod = clazz.getMethod("append", String.class)
jshell> Method lengthMethod = clazz.getMethod("length")

// StringBuilderオブジェクト生成
jshell> var sb = constructor.newInstance()

// StringBuilderオブジェクトのメソッド呼び出し
jshell> appendMethod.invoke(sb, "abc")
jshell> appendMethod.invoke(sb, "def")

jshell> sb
sb ==> abcdef
jshell> int length = (int)lengthMethod.invoke(sb)
length ==> 6
```

リスト3.14はStringBuilderのappendメソッドなどの呼び出し方を知っている前提のコードです。**表3.3**のMethodオブジェクトのメソッドを使うと未知のメソッドの型を調査できます。検証コードであれば変数の型を推論に任せて省略して問題ありません（**リスト3.15**）。

表3.3　Methodオブジェクトの代表的なメソッド（メソッドのthrows節は省略）

メソッド	説明
Class<?> getReturnType()	メソッドの返り値の型を取得
Class<?>[] getParameterTypes()	メソッドの引数の型を取得
Class<?>[] getExceptionTypes()	メソッドのthrows節の例外型を取得

リスト3.15　Methodオブジェクトでメソッド調査

```
// 調査対象のメソッド
// int codePointCount(int beginIndex, int endIndex)
jshell> var clazz = StringBuilder.class
jshell> var method = Stream.of(clazz.getMethods()).filter(m -> m.getName().
equals("codePointCount")).findAny().get()

jshell> var result = method.getReturnType()
result ==> int

jshell> var result = method.getParameterTypes()
result ==> Class[2] { int, int }
```

App.3-3-3　フィールド操作

リフレクションでフィールド値を読み書きするにはjava.lang.reflect.Fieldクラスを使います。ClassオブジェクトからFieldオブジェクトを取得する方法は表3.1を参照してください。

Fieldオブジェクトの代表的なメソッドの定義を示します（**表3.4**）。

表3.4　Fieldオブジェクトの代表的なメソッド（メソッドのthrows節は省略）

メソッド	説明
Class<?> getType ()	フィールドの型を取得
Object get (Object obj)	フィールドの値を取得。第1引数はレシーバオブジェクト。値が基本型の場合はgetIntなどを使う
void set (Object obj, Object value)	フィールドに値を設定。第1引数はレシーバオブジェクト。第2引数は設定値。値が基本型の場合はsetIntなどを使う。finalフィールドの設定はできません

Methodオブジェクト同様、Fieldオブジェクト経由でクラスフィールドを操作する場合、引数に渡すレシーバオブジェクトは無視されます。

Fieldオブジェクト経由で対象オブジェクトのフィールド値を書き換える例を示します（**リスト3.16**）。finalフィールドの書き換えはできませんが、privateフィールドの書き換えは可能です。setAccessibleメソッドでアクセス制御を変更可能だからです。危険なコードなので通常行うことはありませんが、リフレクションの可能性の1つとして紹介します。

リスト3.16　対象オブジェクトのフィールド値の書き換え

```
class MyClass {
    private String stringField = ""; // 書き換え対象のフィールド
    void method() {
        System.out.println("value = " + this.stringField);
    }
}
```

```
// 使用例
jshell> var my = new MyClass()
jshell> Class<?> clazz = MyClass.class
jshell> Field field = clazz.getDeclaredField("stringField")

// privateフィールドの書き換えは実行時例外
jshell> field.set(my, "abc")
|  Exception java.lang.IllegalAccessException

// 強引にprivateフィールドを書き換え
jshell> field.setAccessible(true)
jshell> field.set(my, "abc")
jshell> my.method()
value = abc
```

App.3-4　リフレクションの応用

App.3-4-1　JavaBeans

JavaBeans（以後Bean）は、元々GUIの部品（コンポーネント）をツールの支援で容易に作成可能にする技術として登場しました。Javaは静的型付け言語ですが、Beanはリフレクション活用とメソッド名の命名規約を前提に、動的な型付け言語に近い方向性の技術になっています。

Javaの利用用途がGUIプログラムからサーバ用途に広がる中で、Beanは当初の想定であるGUIの部品より広く使われるようになりました。サーバ利用では、オブジェクトの作成容易性に加えてオブジェクトのライフサイクル管理をBeanコンテナ（後述）に任せる目的が強くなりました。

Beanを特徴づける技術要素は下記です。

- **リフレクションによるオブジェクト生成**
- **プロパティ**
- **カスタマイズ機能を想定した設計**
- **イベント処理**
- **永続化機能**

Beanを特徴づけるのが、リフレクションによるオブジェクト生成とプロパティ操作です。この2つを説明します。

■Beanの生成

あるクラスがBeanクラスか否かを決定づける要因はクラス自身にはありません。Beanオブジェクトを操作するツールやフレームワークの要請で決まるからです。このようなツールやフレームワークを「Beanコンテナ」と呼びます。

BeanコンテナはBeanクラスのオブジェクト生成と破棄の責任を持ちます。依存なしで生成できるようにするため、Beanクラスは引数なしコンストラクタを持つ必要があります。Beanコンテナはリフレクションを使いBeanオブジェクトを生成します。Beanコンテナによりライフサイクルを管理されたオブジェクトをBeanオブジェクトあるいは単にBeanと呼びます。

■Beanのプロパティ

Beanはプロパティを持ちます。Beanのプロパティアクセスには次の規則があります。

- **プロパティ名がfooの時、プロパティ値を取得するメソッド名はgetFoo**
- **プロパティ名がfooの時、プロパティ値を設定するメソッド名はsetFoo**

多くのBeanクラスはプロパティと1対1に対応するフィールドを持ちます。かつ通常はフィールドのアクセス制御をprivateにします。しかしフィールドの存在は必須ではありません。Beanの要件は上記命名のメソッドのみです。

具体例で説明します。Beanコンテナで管理されたBeanがあり、変数objがこのBeanを参照しているとします。obj.fooというfooプロパティ読み出しコードに対して、Beanコンテナはリフレクションを使いgetFooメソッド呼び出しをします。obj.fooへの代入コードに対してはsetFooメソッド呼び出しをします。こうして、動的型言語のようにあまり型を気にせずobj.fooを読み書きできるコードを成立させます。

App.3-4-2　DI（Dependency Injection）

DI（Dependency Injection）と呼ばれる実装技法があります。リフレクションを活用した技法です。具体例で説明します。**リスト3.17**のクラスを見てください。

リスト3.17　Listインタフェースに依存したクラス（DIによる書き換え前）

```
import java.util.List;

class MyStringList {
    private final List<String> list;

    public MyStringList(List<String> list) {
        this.list = list;
    }
```

```
    public void append(String s) {
        this.list.add(s);
    }

    public void dump() {
        this.list.forEach(System.out::println);
    }
}
```

MyStringListクラスのコンストラクタは引数にListオブジェクトを受け取ります。このためMyStringListオブジェクトを使うにはどこかにListオブジェクトの生成コードが必要です。

機能や役割をオブジェクトに分割したプログラムには、オブジェクト生成コードが相対的に増えていきます。更に、あるオブジェクトの生成に依存するオブジェクトを事前に生成する必要があります。上記例でMyStringListオブジェクト生成の前にListオブジェクト生成が必要なようにです。

このように依存関係のあるオブジェクト群の生成処理は面倒で複雑な作業です。このため、オブジェクト生成を適切に管理する技法がいくつか生まれています。デザインパターンの世界ではファクトリパターンやビルダーパターンなどが有名です。

■DIの役割

面倒で複雑な作業は隠蔽するのが定石です。オブジェクト生成処理を分離する技法の1つがDIです。DIコンテナという形でオブジェクト生成処理を役割として分離します。

オブジェクト生成の役割を担うDIコンテナは、結果としてオブジェクト間の依存性の解決を行います。このことを外部から依存性を注入（解決）すると称します。

従来のファクトリパターンが、個々のクラスごとにオブジェクト生成を隠蔽したのに対し、DIコンテナは依存関係のあるオブジェクト群の生成を一括して隠蔽します。いわば、巨大な汎用ファクトリになります。オブジェクト生成を引き受けたDIコンテナは同時にオブジェクトのライフサイクル管理を担います。

リスト3.17のMyStringListクラスの依存性を解決する簡易なDIコンテナを**リスト3.18**に示します。当然ですが、本物のDIコンテナは特定のクラスのために書かれるものではありません。リスト3.18は説明のためのコードと理解してください。

リスト3.18　簡易なDIコンテナ例

```
import java.lang.reflect.Constructor;

class MyDI {
    // クラス、引数型、引数オブジェクト型を文字列で受け取り、オブジェクトを生成する
    public static Object resolv(String targetType, String argType, String argObjectType) throws Exception {
        // 生成オブジェクトのクラス
        Class<?> targetClass = Class.forName(targetType);
        Constructor<?> targetConstructor = targetClass.getConstructor(Class.forName(argType));

        // 引数オブジェクトの型
        Class<?> argClass = Class.forName(argObjectType);
        Constructor<?> argConstructor = argClass.getConstructor();

        return targetConstructor.newInstance(argConstructor.newInstance());
    }
}

public class Main {
    public static void main(String... args) throws Exception {
        // DIでMyStringListオブジェクト（リスト3.17）を生成
        MyStringList sl = (MyStringList)MyDI.resolv("MyStringList", "java.util.List", "java.util.ArrayList");
        sl.append("one");
        sl.append("two");
        sl.dump();
    }
}
```

MyDIクラスに"java.util.ArrayList"を文字列としてハードコードしています。しかし識別子としては記述していない点に注目してください。このため、この情報を外部ファイルやアノテーションあるいは命名規約などで指定可能です。

App.3-4-3　クラスローダとホットデプロイ

■クロスローダの役割

Javaのクラスをロードするのはクラスローダ（java.lang.ClassLoader）の役割です。普通にJavaプログラムを起動した場合、クラスローダオブジェクトは自動的に生成され、開発者がその存在を気にする必要はありません。

1度ロードしたクラスはクラスローダが保持し続けます。このため普通の手段ではクラスファイルの再ロードはできません。

■ホットデプロイ

開発時、プログラムを起動したまま、コードを書き換えたクラスのクラスファイルを再ロードしたい時があります。Webアプリ開発などでいわゆるホットデプロイとして知られる機能です。

Javaプロセスは複数のクラスローダを持てます(注2)。プログラム実行中に新しいクラスローダオブジェクトを明示的に生成するとクラスの再ロードを実現できます。クラスローダごとにクラスのロードを行えるからです。

(注2)　起動時点でブートストラップクラスローダとシステムクラスロードの2つのクラスローダが存在します。

■クラスローダ生成の具体例

クラスローダオブジェクトの生成とクラス再ロードの具体例を示します。**リスト3.19**は、javax.tools.JavaCompilerを使い、コードの中からソースファイルのコンパイルを行います。

リスト3.19　簡易なホットデプロイ例（MyCompiler.java）

```
import java.lang.reflect.Method;
import java.lang.reflect.Constructor;
import java.io.Console;
import java.net.URL;
import java.net.URLClassLoader;
import java.nio.file.Path;
import javax.tools.JavaCompiler;
import javax.tools.ToolProvider;

public class MyCompiler {
    public static void main(String... args) throws Exception {
        if (args.length < 2) {
            System.err.println("usage: java " + MyCompiler.class.getSimpleName() + " クラス名 メソッド名");
            System.exit(-1);
        }

        String targetClass = args[0];
        String targetMethod = args[1];

        JavaCompiler compiler = ToolProvider.getSystemJavaCompiler();
        if (compiler == null) {
            System.err.println("Compiler is not found");
            System.exit(-1);
        }

        Console con = System.console();
        while (true) {
            // 引数で指定したクラスのソースファイルをコンパイル
            int ret = compiler.run(null, null, null, new String[] { targetClass + ".java" } );
            if (ret == 0) {
                // クラスローダ生成
                ClassLoader loader = URLClassLoader.newInstance(new URL[]{ MyCompiler.class.getResource(".") }, null);
                // 指定したクラスローダでクラスを再ロード
                Class<?> clazz = Class.forName(targetClass, true, loader);
                Constructor<?> cnstructor = clazz.getConstructor();
                Method method = clazz.getMethod(targetMethod);
                method.invoke(cnstructor.newInstance());
            } else {
                System.err.println("Compile failed");
                break;
            }
            con.printf("Press enter key to continue");
            con.readLine();
        }
    }
}
```

MyCompilerの実行時、コマンドライン引数でクラス名とメソッド名を渡します。たとえば**リスト3.20**のようなソースファイルを用意します。クラスもメソッドもpublicにしてください。メソッドは引数なしにしてください。

リスト3.20　MyCompilerからロードされるクラス

```
public class My {
    public void method() {
        System.out.println("Myクラスのmethodメソッドversion1");
    }
}
```

MyCompilerを次のように実行します。

```
$ java MyCompiler.java My method
Myクラスのmethodメソッドversion1
Press enter key to continue
```

MyCompilerがMy.javaをコンパイルしてMy.classファイルを生成後、My.classファイルをロードしてmethodを実行します。ここまでは単なるリフレクションの機能で、わざわざクラスローダオブジェクトを新規に生成した意味はありません。

MyCompilerがキー入力待ちで停止している間に、My.javaファイルを**リスト3.21**のように書き換えたとします。

リスト3.21　MyCompilerからロードされるクラスの書き換え例

```
public class My {
    public void method() {
        System.out.println("Myクラスのmethodメソッドversion2");
    }
}
```

エンターキーを押してMyCompilerプログラムの停止を解除します。次のように書き換えたメッセージを表示できます。内部的にMyクラスを再コンパイルして再ロードしているからです。

```
$ java MyCompiler.java My method
Myクラスのmethodメソッドversion1
Press enter key to continue
Myクラスのmethodメソッドversion2
```

```
Press enter key to continue
```

App.3-4-4　プロクシクラス

プロクシクラスは既存クラスをラップするクラスです。java.lang.reflect.Proxyクラスを使うと、差分コードを書くだけでプロクシクラスを作れます。

プロクシクラスを使うと既存メソッドを変更せずに、既存メソッド呼び出しの前後などに独自処理を書き足せます。メソッドを呼ぶ側から透過に使えるのが肝です。

具体例を**リスト3.22**に示します。MyClassがラップ対象のクラスです。MyInterfaceをインタフェース継承させています。プロクシオブジェクトを参照する変数の型として必要だからです。

Proxy.newProxyInstanceメソッドの返り値でプロクシオブジェクトを得られます。このメソッド呼び出しに必要な情報は、ラップ対象のクラスの型、インタフェースの型、プロクシ用途のクラスの型です。それぞれMyClass、MyInterface、MyProxyが該当します。引数の細かな説明は省略します。

プロクシオブジェクトの概念的な意味は、MyClassオブジェクトをMyProxyオブジェクトでラップしたものです。MyProxyオブジェクトそのものではありません。

プロクシオブジェクトはMyInterfaceインタフェースを継承します。このためプロクシオブジェクトに対してmethodの呼び出しが可能です。

リスト3.22　プロクシクラスの使用（Main.java）

```
import java.lang.reflect.Proxy;
import java.lang.reflect.Method;
import java.lang.reflect.InvocationHandler;

public class Main {
    public static void main(String... args) {
        // プロクシオブジェクトの取得
        MyInterface my = (MyInterface)Proxy.newProxyInstance(Main.class.getClassLoader(), // クラスローダ
        new Class<?>[] { MyInterface.class }, // インタフェースのClassオブジェクト
        new MyProxy(new MyClass()));  // InvocationHandlerオブジェクト

        boolean ret = my.method("abc"); // プロクシのinvoke経由でmethodメソッド呼び出し
        System.out.println("ret = " + ret); // MyProxyクラスのinvokeメソッドの返り値
    }
}

// プロクシ対象のインタフェース
interface MyInterface {
    boolean method(String value);
}

// プロクシ対象のクラス
class MyClass implements MyInterface {
    // プロクシ（ラップ）対象のメソッド
    @Override
    public boolean method(String arg) {
        System.out.println("元メソッドが引数%sで呼ばれた".formatted(arg));
        return true;
    }
}

// プロクシクラス
class MyProxy implements InvocationHandler {
    // プロクシ（ラップ）対象のオブジェクトを保持
    private final Object target;

    public MyProxy(Object target) {
        this.target = target;
    }

    @Override
    public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
        System.out.println("インターセプタ(Before)");
        // 元オブジェクトの元メソッド呼び出し
        Object ret = method.invoke(target, args);
        System.out.println("インターセプタ(After)");
        return ret;
    }
}
```

プロクシオブジェクトのmethodを呼ぶと、内部的にMyProxyオブジェクトのinvokeメソッドを呼び出します。第1引数のObject proxyはプロクシオブジェクトです。

第2引数で元オブジェクト（MyClassオブジェクト）の元メソッド（method）のMethodオブジェクトが渡ってきます。リフレクションで元メソッドを呼び出し可能です。

元オブジェクトのメソッド呼び出し以外の処理も記述可能です。MyProxyオブジェクトのinvokeメソッド内の処理は開発者の自由だからです。通例として、メソッド呼び出しの前後に挟み込む処理を、フック処理やインターセプタ処理と呼びます。既存コードに手を入れることなくこのように処理を差し挟む手法はAOP（Aspect Oriented Programming）としても知られています。AOPの簡易な実現方法の1つがプロクシクラスの使用です。リスト3.22を実行すると次のようになります。

```
$ java Main.java
インターセプタ(Before)
元メソッドが引数abcで呼ばれた
インターセプタ(After)
ret = true
```

App.4 アノテーション

アノテーションは、クラスやメソッドやフィールドなど、プログラムの構成要素に付与できる付加情報です。アノテーションは開発者が独自に定義できる修飾子として働き、宣言的なプログラミング技法をJavaにもたらします。アノテーションの使い方と応用例を説明します。

App.4-1 アノテーション

アノテーションはプログラムの構成要素に付与するメタ情報です。「**11章　インタフェース**」で紹介した@Overrideや@FunctionalInterfaceがアノテーションの具体例です。

文法上、アノテーションはpublicやfinalのような修飾子の1つとして働きます。しかし他の修飾子とは違いアノテーションは予約語ではありません。クラスやインタフェースのように開発者自身がアノテーション型を宣言します。アノテーション型の宣言方法は後ほど説明します。

アノテーションを修飾子として記述することを「アノテーションの適用」と呼びます。アノテーションを適用できるプログラム要素を下記に示します。

- **型宣言（クラス、インタフェース、レコードクラス、enum型、アノテーション型）**
- **変数宣言**
- **メソッド宣言**
- **コンストラクタ宣言**
- **レコードコンポーネント**
- **パッケージ宣言**
- **モジュール宣言**
- **（ジェネリック型の）型パラメータ宣言**
- **（使用時の）型**

アノテーションには、そのアノテーションを処理するプログラムが必要です。コンパイラやコード解析ツールなどが処理するアノテーションもあれば、フレームワークがコンパイル時または実行時に処理するアノテーションもあります。

たとえば@Overrideアノテーションや@FunctionalInterfaceアノテーションであれば、コンパイラが読み取って処理します。アノテーションを使い不正なコードをコンパイルエラーにします。

App.4-1-1 アノテーションの適用

ソースコード上、修飾子を書ける場所にはアノテーションを書けると考えて問題ありません。慣例として、アノテーションは最初の修飾子として書きます（**リスト4.1**）。

リスト4.1　アノテーションの適用

```
// 下記のように書いてもOKですが
public @Override String method(String s) { /* 省略 */ }

// このように書くのが慣例です（行を分けるかは好みの問題です）
@Override
public String method(String s) { /* 省略 */ }
```

App.4-1-2 アノテーションの応用例

アノテーションの応用例を示します。

- **構文補助**
- **宣言的プログラミング**

■構文補助

構文補助を目的としたアノテーションの代表例は@Overrideアノテーションです。構文補助によりJavaの文法の独自拡張に近いことができます。

別の例として、型の独自拡張があります。使用時の型にアノテーションを適用します。このようなアノテーションをタイプアノテーションと呼びます。たとえば@ReadOnlyアノテーションを定義して次のように使うと、理屈上は読み込み専用のStringBuilder型という新しい型を創出できる可能性があります。ただし、この可能性を現実にするには、@ReadOnlyアノテーションを処理するプログラムが別途必要です(注1)。

```
@ReadOnly StringBuilder sb = new @ReadOnly StringBuilder("abc");
```

■宣言的プログラミング

機能や役割によるコードの分割はプログラミングの王道です。分割したコードがどう動くかはコードを見ればわかります。しかしそれを誰が何のために使うかはコードを見るだけではわからない場合があります。役割を示す手がかりが型名や変数名などの名前しかないからです。

従来はこのような手がかりをコメントで書き記していました。しかしコメントは不完全です。単なる自由な文字列に過ぎないからです。決まった構文でコードの役割を記述できる仕組みがあ

（注1）　@ReadOnlyアノテーションをコードから読み取るコンパイラや実行時に検出するフレームワークなどの処理系が必要です。本書執筆時点で標準的な処理系は存在しません。

ると便利です。アノテーションを活用すると、コードの役割の手がかりを形式的に記述できます。

アノテーションで記述した役割はプログラムで読み取り可能です。これによりコメントの延長以上の力を持ちます。ツールやフレームワークがアノテーションで記述した役割を読み取り、コードの間をつなぐコードを自動生成できるからです。

プログラミングの歴史の中で、手続きを記述する手続き型プログラミングと別に「宣言型プログラミング」という流れがあります。宣言型プログラミングは手順ではなく目的を記述します。アノテーションで記述したコードの役割を目的の記述と見ると、アノテーションはJavaに宣言型プログラミングの道を開きます。

App.4-2 標準アノテーション

既存アノテーション型の使い方を紹介します。アノテーション型の自作方法は後ほど説明します。Javaの標準ライブラリに含まれる主なアノテーション型を示します（**表4.1**）。

表4.1 標準ライブラリに含まれる主なアノテーション型

アノテーション名	説明
Override	オーバーライドしたメソッドであることを明示（「11章 インタフェース」参照）
FunctionalInterface	関数型インタフェースを明示（「11章 インタフェース」参照）
Deprecated	非推奨であることを明示
SuppressWarnings	コンパイラの警告メッセージを抑制
SafeVarargs	ジェネリック型の可変長引数の警告メッセージを抑制（説明省略）
Native	フィールドがネイティブコード由来であることを明示（説明省略）

App.4-2-1 @Deprecatedアノテーション

@Deprecatedアノテーションは、使用を推奨しない対象要素を明示します。典型的な対象はメソッドやコンストラクタですが、型宣言やフィールドなど他の要素にも適用可能です。後方互換性のために残してあるクラスやメソッドに付与して、開発者に新しい機能への移行をうながします。

標準クラスの中に@Deprecatedアノテーションのついた要素がいくつかあります。たとえばjava.lang.Integerクラスの次のコンストラクタには@Deprecatedアノテーションがついています（**リスト4.2**）。

リスト4.2 @Deprecatedアノテーションの例

```
// Integer.javaから抜粋
@Deprecated(since="9", forRemoval = true)
public Integer(int value) {
    this.value = value;
}
```


このコンストラクタを使うコードは警告がでます（**リスト4.3**）。JShellではなくコンパイルした場合、コンパイル時に同じ警告がでます。

リスト4.3 @Deprecatedアノテーションの警告

```
jshell> var num = new Integer(123)
|  Warning:
|  Integer(int) in java.lang.Integer has been deprecated and marked for removal
```

コンパイルエラーではなく警告なのでコードのコンパイルは成功します。実行にも問題はありません。しかし、原則はコード修正を勧めます。なお、リスト4.2のようなforRemoval = trueは削除予定を意味します。将来のJavaでコンパイルエラーになる可能性があります。

クラス作者はメソッドなどを非推奨に変更した場合、修正方法を示す義務があります。たとえば修正方法をjavadocコメントに明記します。上記のコンストラクタの場合、Integer.valueOfメソッドの使用を推奨しています（「**16-3 数値クラス（数値ラッパークラス）**」参照）。

App.4-2-2 @SuppressWarningsアノテーション

@SuppressWarningsアノテーションはコンパイラの警告を抑制する指示です。警告の種別を文字列で指定します。前節の警告を抑制するには**リスト4.4**のように書きます。

リスト4.4 警告を抑制する例

```
public class Main {
    public static void main(String... args) {
        @SuppressWarnings("removal")
        var num = new Integer(123);
    }
}

// 上記コードのコンパイル
$ javac -Xlint:all Main.java
```

@SuppressWarningsアノテーションに指定できる文字列はjavaコンパイラに依存します。指定可能な文字列は javac -help-lint の出力で探せます。

@SuppressWarningsアノテーションによるコンパイラ警告の抑制は、バグを隠す危険があります。しかし実開発ではいろいろな事情で警告の修正を後回しにせざるを得ない場合があります。この時、警告の放置より @SuppressWarningsアノテーションによる抑制のほうがマシという考えがあります。一度警告を放置してしまうと、開発チーム全体がそれを是として警告を誰も気にしなくなる危険があります。それはいつか大きなバグにつながります。@SuppressWarningsアノテーションも濫用すれば同じですが、コードに明示される分、後ろめたい気持ちが残ります。将来対応する意思表示を示すのは警告の無視よりは健全です。

App.4-3 アノテーション型宣言

アノテーション型の自作方法を説明します。

文法的にはアノテーション型は型の一種です。ちょうどObjectクラスやStringクラスに、対応するObject.javaやString.javaのソースファイルが存在するように、@Overrideアノテーションには Override.javaというソースファイルが存在します。クラス同様、アノテーション型宣言のファイルもjavacでコンパイルします。ネストしたアノテーション宣言も可能です(注2)。

自作したアノテーションを修飾子のように適用できます。宣言と適用を区別するため、宣言するのをアノテーション型、適用するのをアノテーションと表現します(注3)。

アノテーションはアノテーション型のインスタンスです。Javaの他のオブジェクト同様、アノテーションを参照型変数で参照可能です。ただしアノテーションを変数で参照するコードはやや特殊な使い方になります。今はインスタンスという概念を忘れて修飾子の1つとしてアノテーションを考えてください。アノテーションを変数で参照する使い方は後ほど紹介します。

アノテーション型の宣言例としてOverride.javaのソースファイルを見てみます(**リスト4.5**)。

リスト4.5　Override.javaから抜粋

```
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.SOURCE)
public @interface Override {
}
```

@Overrideアノテーション型の宣言自身に、TargetとRetentionの2つのアノテーションがついています。アノテーション型宣言に適用するアノテーションをメタアノテーションと呼びます。メタアノテーションについては節を改めて説明します。

アノテーション型の宣言の構文は次になります。

```
// アノテーション型の宣言の文法
[修飾子] @interface アノテーション名 {
    [メンバ宣言(主にアノテーション要素)]
}
```

アノテーション名は開発者が決めます。アノテーション名はクラス名およびインタフェース名と同じ名前空間に属します。つまり完全修飾名の一致するクラスやインタフェースが存在する場合、コンパイルエラーになります。

トップレベルのアノテーション型宣言に付与可能な修飾子はpublic修飾子またはメタアノテーションです。ネストしたアノテーション型宣言には他のアクセス制御用の修飾子の付与が可能です。アクセス制御用の修飾子の意味はクラスやインタフェースの場合と同じです。

アノテーション型のメンバに宣言できるのは次の4つです。

- **アノテーション要素**
- **staticフィールド**
- **staticなネストしたクラス**
- **staticなネストしたインタフェース**

アノテーション要素以外の3つはインタフェースのメンバと同じ意味と働きです。説明は省略します。

App.4-3-1　アノテーション要素

アノテーション要素を持つ例として@SuppressWarningsアノテーション型宣言を紹介します(**リスト4.6**)。なおアノテーション要素を持たないアノテーション型をマーカーアノテーションと呼びます。@Overrideアノテーションはマーカーアノテーションの1例です。

リスト4.6　SuppressWarnings.javaから抜粋(メタアノテーションは省略)

```
public @interface SuppressWarnings {
    String[] value();
}
```

アノテーション要素は、文法上、本体のないメソッド定義です。形式的には抽象メソッドと同じです。つまり、リスト4.6のアノテーション要素はvalueという名前で返り値の型がString[]のメソッドです。

形式は抽象メソッドですが、使用の観点ではアノテーション要素はメソッドよりフィールドに近い存在です。この意味は実際にアノテーション適用例を見る中で明らかになります。

アノテーション要素名は開発者が自由につけられる名前です。ただ、valueという名前だけは特別な省略記法が用意されています。アノテーション要素が1つの時は名前をvalueにするのが慣例です。valueの特別な省略記法の意味は後述します。

アノテーション要素は複数記述できます(**リスト4.7**)。

リスト4.7　複数のアノテーション要素の例

```
public @interface MyAnnotation {
    String foo();
    int bar();
    Class baz();
```

(注2)　使う機会はあまりありません。またローカルアノテーションの宣言はできません。

(注3)　Java言語仕様の用語は、アノテーション型ではなくアノテーションインタフェースです。インタフェースの一種の扱いです。しかし言語機能としてインタフェースとアノテーションを別物と考えるほうが理解は容易と考え、本書は別物として説明します。

```
    String[] qux();
}
```

抽象メソッドの返り値の型がアノテーション要素の型になります。たとえばリスト4.7の場合、要素fooの型がString、要素barの型がintです。アノテーション要素の型は次に限定されます。

- **基本型**
- **String型**
- **enum型**
- **アノテーション型**
- **Class型**
- **上記を要素型とする配列型（配列の配列は不可）**

アノテーション要素には、通常の抽象メソッド宣言にはない次の制約があります。

- **引数は禁止（文法上は空の引数）**
- **throws節は禁止**
- **ジェネリックメソッドは禁止**

制約違反でコンパイルエラーになる例を示します（**リスト4.8**）。

リスト4.8　コンパイルエラーになるアノテーション要素

```
public @interface InvalidAnnotation {
    Integer foo();  // 返り値の型は決まった型しか許可されない
    int bar(int i); // 引数は禁止
    String baz() throws java.io.IOException; // throws節は禁止
    <T> String qux();  // ジェネリックメソッドは禁止
}
```

App.4-3-2　メタアノテーション

アノテーション型宣言自体を修飾するアノテーションをメタアノテーションと呼びます。標準のメタアノテーション型を**表4.2**に示します。アノテーション理解の助けになる@Targetアノテーションと@Retentionアノテーションについて説明します。

表4.2　標準メタアノテーション型

アノテーション名	説明
Target	アノテーションの対象要素を指示
Retention	アノテーションの存在期間を指示
Documented	アノテーションを文書化対象するjavadocへの指示
Inherited	（インタフェースや基底クラスに）適用されたアノテーションが自動的に派生クラスにも適用されることの指示
Repeatable	同じ対象要素に同じアノテーションを複数指定可能

■@Targetアノテーション

@Targetアノテーションは、対象アノテーションを適用可能なプログラムの構成要素を指示します。@Targetアノテーションの要素はElementTypeのenum定数です（**リスト4.9**）。複数のenum定数を同時に指定可能な配列型です(注4)。

リスト4.9　@Targetアノテーション

```
// ElementType.javaから抜粋
public enum ElementType {
    TYPE,
    FIELD,
    METHOD,
    PARAMETER,
    CONSTRUCTOR,
    LOCAL_VARIABLE,
    ANNOTATION_TYPE,
    PACKAGE,
    TYPE_PARAMETER,
    TYPE_USE,
    MODULE,
    RECORD_COMPONENT;
}

// Target.javaから抜粋
@Documented
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.ANNOTATION_TYPE)
public @interface Target {
    ElementType[] value();
}
```

たとえば@Overrideアノテーションの宣言には @Target(ElementType.METHOD) がついています（リスト4.5）。@Overrideアノテーションがメソッドだけに適用可能である旨を意味しています。一方、@SuppressWarningsアノテーションは**リスト4.10**のように多くの要素に適用可能です。

（注4）　ElementTypeの各値は、本章の冒頭に挙げた、アノテーションを適用可能なプログラム構成要素の一覧に対応します。

リスト4.10　SuppressWarnings.javaから抜粋

```
@Target({TYPE, FIELD, METHOD, PARAMETER, CONSTRUCTOR, LOCAL_VARIABLE})
@Retention(RetentionPolicy.SOURCE)
public @interface SuppressWarnings {
    String[] value();
}
```

リスト4.9を見てわかるように、@Targetアノテーション型宣言自身に、@Target(ElementType.ANNOTATION_TYPE)が適用されています。@Targetアノテーションが、アノテーション型宣言に付与可能という意味です。

■@Retentionアノテーション

@Retentionアノテーションは実質的にアノテーション型宣言に適用必須のアノテーションです(注5)。対象のアノテーション型をどう使うかに影響するからです。

@Retentionアノテーションの要素はRetentionPolicyのenum定数です（**リスト4.11**）。

リスト4.11　@Retentionアノテーション

```
// RetentionPolicy.javaから抜粋
public enum RetentionPolicy {
    SOURCE,
    CLASS,
    RUNTIME
}

// Retention.javaから抜粋
@Documented
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.ANNOTATION_TYPE)
public @interface Retention {
    RetentionPolicy value();
}
```

RetentionPolicyのenum定数は**表4.3**に示す意味を持ちます。

表4.3　@Retentionアノテーションの意味

RetentionPolicyのenum定数	説明
SOURCE	アノテーションはソースファイルにのみ残る
CLASS	アノテーションはソースファイルとクラスファイルにのみ残る
RUNTIME	アノテーションはソースファイルとクラスファイルに残り、かつ実行時にも残る

（注5）　Retentionアノテーションの適用を省略したアノテーション型は暗黙的にRetentionPolicy.CLASS動作になります。このデフォルト動作に頼らず、明示的な指定を推奨します。

RUNTIME指定したアノテーションは実行時にリフレクションで読み取り可能です。具体例は後述する応用例で示します。コンパイル時に読み取るだけでよいアノテーションはSOURCEを指定します。たとえば@Overrideアノテーションはコンパイル時に使うだけです。このためSOURCEの指定になっています。

App.4-4　アノテーションの適用

アノテーションの適用例は@Overrideアノテーションなどで説明済みです。適用時に@Override()のように括弧をつけても問題ありません。アノテーション要素がない場合、括弧の省略が通例です。

アノテーション要素があるアノテーションの適用例を示します（**リスト4.12**）。

リスト4.12　リスト4.7のアノテーション適用例

```
@MyAnnotation(foo = "abc", bar = 42, baz = String.class, qux = {"xyz", "zzz"})
void method() { /* 省略 */ }
```

アノテーション記載後の括弧内に「アノテーション要素名 = アノテーション要素値」を記述します。アノテーション要素値の型は、アノテーション要素の型と一致させます。要素型が配列型の場合、中括弧{}で値を指定します。

直感的には、アノテーションの適用でアノテーション型インスタンスを生成します。リスト4.12で言えば、値が"abc"や42のフィールド値を持つアノテーションを生成したように考えて問題ありません。

アノテーション要素の名前がvalueの時のみ、value = を省略可能です。アノテーション要素valueのある具体例として@SuppressWarningsアノテーションを例に、省略記法と省略しない記法の両方を列挙します（**リスト4.13**）。

リスト4.13　@SuppressWarningsアノテーションの適用例

```
// value=を省略した記法
@SuppressWarnings({"deprecation", "divzero"})
void method() { /* 省略 */ }

// value=を省略しない記法
@SuppressWarnings(value = {"deprecation", "divzero"})
void method() { /* 省略 */ }

// アノテーション要素の型が配列型の場合、指定する値が1つであれば中括弧を省略可能
// value=と中括弧を省略した記法
@SuppressWarnings("deprecation")
void method() { /* 省略 */ }
```

アノテーション適用時にすべてのアノテーション要素の値を指定する必要があります。**リスト4.14**は@MyAnnotationアノテーションの要素bazの値を指定していないためコンパイルエラーになります。

リスト4.14　リスト4.7のアノテーション適用でコンパイルエラー

```
// 要素bazのアノテーション値の指定が必要
@MyAnnotation(foo = "abc", bar = 42, qux = {"xyz", "zzz"})
void method() { /* 省略 */ }
```

■アノテーション要素のデフォルト値

アノテーション型宣言時に予約語defaultを使い、要素のデフォルト値指定が可能です。@MyAnnotationアノテーション型の要素にデフォルト値を指定する例を示します（**リスト4.15**）。

リスト4.15　デフォルト値を指定したアノテーション型宣言

```
public @interface MyAnnotation {
    String foo() default "abc";
    int bar() default 42;
    Class baz() default String.class;
    String[] qux() default {"xyz", "zzz"};
}
```

アノテーション適用時に値を指定しなければ、アノテーション要素がデフォルト値になります。すべてのアノテーション要素がデフォルト値を持っていればマーカーアノテーション同様、括弧を省略して記述可能です。アノテーション適用時の値指定の必要性を減らすために、可能な限り、適切なデフォルト値設定を推奨します。

App.4-5　アノテーションの応用

本章の最後にアノテーションの応用を具体例で示します。アノテーションを使い任意の外部コードを呼び出す例です。外部から与えるコードをプラグインコードと呼ぶことにします。

プログラムの構造は「**11-6-1　コールバックパターン**」と同じです。インタフェースを定義する代わりにアノテーションを使います。インタフェースとアノテーションの比較は最後に考察します。

最初に@MyFilterアノテーション型を宣言します（**リスト4.16**）。プラグインコードのメソッドに適用するのでTargetをMETHODにします。実行時にリフレクションでアノテーションを読み取るのでRetentionをRUNTIMEにします。

リスト4.16　@MyFilterアノテーション型の宣言

```
import java.lang.annotation.*;

@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface MyFilter {
}
```

文字列を受け取り、大文字に変換した文字列を返すメソッドを実装して、@MyFilterアノテーションを適用します（**リスト4.17**）。「11-6-1　コールバックパターン」のリスト11.30と比較してください。意図的に名前を揃えたので、違いはMyFilterインタフェースを継承するか@MyFilterアノテーションを付与するかだけです。

リスト4.17　@MyFilterアノテーションの適用（CapitalizeFilter.java）

```
public class CapitalizeFilter {
    @MyFilter
    public String doJob(String input) {
        return input.toUpperCase();
    }
}
```

App.4-5-1　プラグイン

リスト4.17をプラグインコードとして呼び出すコードを**リスト4.18**に示します。

リスト4.18　プラグインコードの呼び出し側（基本構造はリスト 11.30 と同じ）

```
import java.util.List;
import java.util.ArrayList;
import java.io.Console;
import java.lang.reflect.Method;
import java.lang.reflect.Constructor;

// MyEcho4オブジェクトを使うコード
public class Main {
    public static void main(String... args) {
        try {
            var echo = new MyEcho4(args); // コマンドライン引数でクラス名を指定
            echo.execute();
        } catch (ReflectiveOperationException e) {
            e.printStackTrace();
        }
    }
}
```

```
class MyEcho4 {
    // プラグインコードを表現するレコードクラス（オブジェクト参照とメソッドのペア）
    private record Filter(Object obj, Method method) {}

    // プラグインコードを保持
    private final List<Filter> filters = new ArrayList<>();

    public MyEcho4(String... classNames) throws ReflectiveOperationException {
        // プラグインコードのロード
        for (String className : classNames) {
            Class<?> clazz = Class.forName(className); // リフレクションで探す
            for (Method method : clazz.getMethods()) {
                if (method.isAnnotationPresent(MyFilter.class)) { // アノテーションを探す
                    filters.add(new Filter(clazz.getDeclaredConstructor().newInstance(), method));
                }
            }
        }
    }

    public void execute() throws ReflectiveOperationException {
        Console console = System.console();
        while (true) {
            System.out.println("Input any text");
            String msg = console.readLine();

            String output = msg;
            for (Filter filter : filters) {
                // プラグインコードの呼び出し(注6)
                output = (String)filter.method().invoke(filter.obj(), output);
            }
            System.out.println("You said, " + output);
        }
    }
}
```

プラグインの骨子は、クラス名からクラスをロードして@MyFilterアノテーションのついたメソッドを探す部分です。リスト4.18のMyEcho4のコンストラクタの中で行っています。クラス名からClass.forNameでClassオブジェクトをロードする部分は「**App.3 リフレクション**」を参照してください。クラスのすべてのメソッドの中から@MyFilterアノテーションのついたメソッドを探します。isAnnotationPresentメソッドを使って探しています。@MyFilterアノテーションのついたメソッドをfiltersリストに格納します。

（注6）　このコードを理解しづらい人は、filter.method()とfilter.obj()の返り値をそれぞれ一時変数に代入してみてください。やっていることはMethodオブジェクトのinvokeメソッド呼び出しです。

プラグインコードのメソッド呼び出しはリフレクションのinvokeメソッド経由です。メソッドの呼び出し方の違いを除くと、リスト11.30（MyEcho2クラス）と同じ構造です。

プラグインするクラスは実行時のコマンドライン引数で指定します。リスト4.18の実行例は下記になります。事前にCapitalizeFilterクラスのコンパイルが必要です。

```
$ javac CapitalizeFilter.java
$ java Main.java CapitalizeFilter
Input any text
hello
You said, HELLO
```

■プラグインの制御

プラグインコードの呼び出し順序を制御できるようにアノテーションを改造してみます。Positionというenum定数をアノテーション要素に定義します（**リスト4.19**）。

リスト4.19　呼び出しの順序制御を可能にする@MyFilterアノテーション

```
public @interface MyFilter {
    enum Position {
        BEFORE,
        AFTER,
    }
    Position value();
}
```

@MyFilterアノテーションを適用する側でアノテーション要素の値を指定します（**リスト4.20**）。

リスト4.20　順序制御可能にする@MyFilterアノテーション適用

```
public class CapitalizeFilter {
    @MyFilter(MyFilter.Position.BEFORE)
    public String doJob(String in) {
        return in.toUpperCase();
    }
}
```

MyEcho4クラスの変更部分は**リスト4.21**のようになります。isAnnotationPresentメソッドの存在チェックの代わりにgetAnnotationメソッドで@MyFilterアノテーションの参照を取得します。アノテーション要素値の取得はvalueメソッド呼び出しで行います。アノテーション要素値を見てリスト先頭への追加とリスト後尾への追加を切り分けます。

リスト4.21　リスト4.18の変更部分

```
for (Method method : clazz.getMethods()) {
    MyFilter myfilter = method.getAnnotation(MyFilter.class); // アノテーションの読み取り
    if (myfilter != null) {
        switch (myfilter.value()) {
            case BEFORE -> {
                filters.add(0, new Filter(clazz.getDeclaredConstructor().newInstance(), method));
            }
            case AFTER -> {
                filters.add(new Filter(clazz.getDeclaredConstructor().newInstance(), method));
            }
            case null -> throw new IllegalArgumentException();
        }
    }
}
```

本章はここまでアノテーションをオブジェクトのように扱わず、修飾子の1つの視点で説明してきました。一方、リスト4.21のようにアノテーションを読み取るコードの場合、異なる視点が必要です。アノテーション型の変数でアノテーションを参照し、アノテーション要素の値の取得がメソッド呼び出しになるからです。アノテーションがオブジェクトの一種で、アノテーション型がその型になる視点で読む必要があります。

App.4-5-2　アノテーションとPOJO

インタフェースを使わず、アノテーションで同じような仕組みを構築できました。

インタフェース版とアノテーション版を比較してみます。アノテーション版の欠点は型安全性に劣る部分です。CapitalizeFilterクラスのメソッドの型を変更すると実行時エラーになります。リフレクション時にメソッドの引数や返り値の型を検証すれば解消可能ですが、少々面倒なのは否めません。

インタフェース版であればコンパイル時に検証できます。文字列を受け取り文字列を返すメソッドをインタフェースで定義すれば、実装クラスはこの仕様に従わざるを得ないからです。通常、実行時よりコンパイル時に問題に気づけるほうが正義です。この観点で言えばインタフェース版のほうが優れた手段です。

アノテーション版の利点は、プラグインするコードから他への依存性を減らせる点です。どこかに依存のあるコードは、依存先の変更による影響を受ける可能性があります。依存制御を突き詰めると、何にも依存しないコードがもっとも堅牢である、という結論になります。他のクラスやインタフェースと継承関係を持たないクラスのオブジェクトをPOJO（Plain Old Java Object）と呼びます(注7)。アノテーション版のCapitalizeFilterクラスはPOJOクラスです。POJOクラスは

(注7)　POJOでなくせる依存は継承関係の依存であって、それ以外の依存は別の論点になります。POJO自体は厳密に定義された用語ではありません。継承まみれのクラスに対するアンチテーゼとして生まれた用語です。

単体テストが容易であったり、実行環境への配置が簡易であるという利点を持ちます。

コード全体を俯瞰するとアノテーションとPOJOコードの利用は複雑性を増します。裏にPOJOコードの間をつなぐ間接コードが必要だからです。しかしこれらの間接コードの複雑さをフレームワークやライブラリに隠蔽できれば、利用者は拡張ポイントのコードをただPOJOで書くだけでよくなります。これはソフトウェアの進化の1つの方向性です。

App.5 順序制御

マルチスレッドやノンブロッキング処理などの並行処理をすると、意図しない順序で処理が進みデータが不整になったり処理が終わらない可能性があります。このような問題を解決する順序制御について説明します。

App.5-1 スレッドの順序制御

複数スレッドの実行順序を制御したい場合があります。たとえばあるスレッドの実行結果を別のスレッドが使いたい場合などです。待ちたい処理は通信の受信結果だったり時間のかかる計算処理の結果だったりします。本書のサンプルコードではこのような処理をThread.sleep処理で模倣します。

App.5-1-1 joinメソッドによる順序制御

Threadオブジェクトのjoinメソッドを使うと対象スレッドの終了を待機できます。もっとも低水準の順序制御です。スレッドの終了を待つだけであれば順序制御に使えます。具体例は「**20章　スレッド**」のコード例を参照してください。

App.5-2 通知を使う順序制御

通知処理は、ある条件が成立するまでスレッド群を待機させ、条件成立後に待機スレッド群を起こす機能です。通知を使うと複数スレッドの順序制御を可能です。

通知処理では、条件が成立するまで待つスレッドと、条件を成立させ待機スレッドを起こすスレッドが存在します。

スレッド間をつなぐ役割のためにオブジェクトを1つ用意する必要があります。この用途にはどんなオブジェクトでも使えます。通知に使うメソッドがObjectクラスのメソッドだからです。便宜上、この役割に使うオブジェクトを通知用オブジェクトと呼びます。

通知用オブジェクトのwaitメソッドを呼んだスレッドは待機状態になります。別スレッドが同じオブジェクトを使ってnotifyメソッドもしくはnotifyAllメソッドを呼ぶまで待機中スレッドは停止します。タイムアウト値を指定したwaitの場合、時間経過後にnotify呼び出しがなくても待機状態から抜けます。

別スレッドがnotifyメソッドを呼ぶと待機中スレッドのうち1つだけを起こします。どのスレッドを起こすかを制御する手段はありません。notifyAllメソッドを呼ぶと待機中スレッドをすべて起こします。本書は2つを合わせてnotify系メソッドと記載します。

App.5-2-1 通知処理のコードの構造

通知処理のコードの構造は明確に決まっています。**リスト5.1**のようになります(注1)。

リスト5.1　通知処理のコードの構造

```
// targetObjは任意の共有オブジェクトという想定

// 待機スレッド：通知を待つ側の処理
synchronized(targetObj) {
    while (!条件) {
        targetObj.wait(); // ここで待機（待機中はプロセッサを使わない）
    }
    条件が成立した時の処理
}

// 通知スレッド：起こす側の処理
synchronized(targetObj) {
    条件を成立させる処理
```

C O L U M N

スリープによる順序制御

知識なしに思いつく順序制御の手法の1つがスリープによる待機です。タイムアウトのあるネットワーク処理を待機するコードで書きがちです。スリープを使う順序制御は不可能ではありませんが推奨しません。理由は次の2つです。

- 待っている対象の処理が想定より早く完了すると順序制御に失敗する
- 待っている対象の処理が想定より遅く完了すると不必要に長く待つ

後者よりは前者のほうがマシなのでスリープ時間は必然的に長めになります。しかしこれはプログラム全体の性能劣化につながります。実開発の既存コードにスリープ処理を見つけた場合は要注意です。苦肉の策で書かれた可能性が高いからです。ハードウェア性能の向上で処理速度が変わると不正動作になる可能性があります。危険な地雷コードなので慎重に除去してください。

(注1)　リスト 28.1のwhile文を不要だと思った人もいるかもしれません。しかしwhile文のチェックは必須です。条件が成立しなくてもwaitメソッドが返る可能性があるからです。

```
    targetObj.notifyAll(); // 待機スレッドを起こす
}
```

通知処理は内部でモニタロックを使います。このためwaitメソッドとnotify系メソッド呼び出しをsynchronizedコードで囲む必要があります。リスト5.1に示す例では通知用オブジェクトをtargetObjにしています。このオブジェクトのモニタロックを使います。

リスト5.1の2つのスレッド処理は、普通にコードを読むと実行できないようにも見えます。待機スレッドが先にモニタロックを獲得すると通知スレッドがsynchronizedコードを実行できないように見えるからです。動くからくりは、waitメソッドが内部で自動的にモニタロックを解放する点にあります。仕組みを説明します。待機スレッドが先にsynchronizedコードに入ったとします。待機スレッドはtargetObjのモニタロックを獲得しますが、targetObjのwaitメソッドを呼ぶと内部でtargetObjのモニタロックを解放します。この結果、通知スレッドがモニタロックを獲得できます。別の見方をすると、これがwaitメソッド呼び出しをsynchronizedコード内で行う必要性の理由です。

通知スレッドがtargetObjのnotifyAllメソッドを呼ぶと、待機スレッドがwaitから戻ります。この時、待機スレッドはwaitメソッド内で自動的にtargetObjのモニタロックを獲得しようとします。しかし、通知スレッドがモニタロック獲得中なのですぐには獲得できません。通知スレッドがsynchronizedコードを抜けてモニタロックを解放すると、待機スレッドがモニタロックを獲得してwaitメソッドから返ります。

通知スレッドが先にsynchronizedコードに入ると、待機スレッドがない状態でnotifyAllメソッドを呼ぶだけです。notifyした状態は残らないので、その後に別のスレッドがwaitすると、新たに起こす処理をするスレッドが現れるまで待機します。

App.5-2-2　通知処理の具体例

通知処理の典型例を示します。待つ側のスレッド(群)が必要なデータを待ち、起こす側が必要なデータを揃える、というのがシナリオです。しばしば待つ側を消費者(consumer)、起こす側を生産者(producer)としてモデル化します。

リスト5.2　通知処理の具体例

```
import java.util.*;
import java.util.concurrent.*;
import java.io.Console;

public class MyProduceConsume {
    public static void main(String... args) {
        try {
            var queue = new LinkedList<String>(); // 共有データ兼通知オブジェクト
            // プラットフォームスレッドと仮想スレッドのいずれかを有効化
            //ThreadFactory thFactory = Thread.ofPlatform().factory();
            ThreadFactory thFactory = Thread.ofVirtual().factory();
            ExecutorService executor = Executors.newFixedThreadPool(8, thFactory);

            Future<?> producer = executor.submit(new Producer(queue));
            Future<?> consumer = executor.submit(new Consumer(queue));

            producer.get();
            consumer.get();
            executor.shutdown();
            executor.awaitTermination(10, TimeUnit.SECONDS);
        } catch (InterruptedException | ExecutionException e) {
            e.printStackTrace();
        }
    }
}

class Producer implements Runnable {
    private final Queue<String> queue;

    Producer(Queue<String> queue) {
        this.queue = queue;
    }

    @Override
    public void run() { // 通知スレッドのエントリポイント
        Console console = System.console();
        while (true) {
            System.out.println("input any string");
            String s = console.readLine(); // ユーザの入力待ち
            synchronized(queue) {
                queue.add(s);      // 待機を抜ける条件を成立
                queue.notifyAll(); // 待機スレッドを起こす
            }
        }
    }
}

class Consumer implements Runnable {
    private final Queue<String> queue;

    Consumer(Queue<String> queue) {
        this.queue = queue;
    }

    @Override
    public void run() { // 待機スレッドのエントリポイント
        try {
```

```
            while (true) {
                synchronized(queue) {
                    while (queue.isEmpty()) { // 待機を抜ける条件
                        System.out.println("waiting...");
                        queue.wait();         // ここで待機
                    }
                    String s = queue.remove();
                    System.out.println(s + " is consumed");
                }
            }
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

リスト5.2の動作を説明します。ProducerクラスとConsumerクラスのオブジェクトを生成して別々のスレッドで実行します。2つのスレッドはキュー（Queueオブジェクト）を共有します。Producerスレッドはコンソールでユーザ入力を待ちます。ユーザ入力があると入力文字列をキューに追加して、Consumerスレッドを起こします。起こされるまでConsumerスレッドはwaitメソッドで待機します。起こされたConsumerスレッドはキューから文字列を取り出します。

App.5-2-3　タスクキューモデル

リスト5.2の待機スレッドは1つでした。複数の待機スレッドで並行処理を行う場合があります。ネットワーク系のサーバプログラムでは、クライアントからの接続を受けつける1つのマスタースレッドと、実際にデータ送受信をする多数のワーカースレッドの構成が1つのパターンです。

タスクキューを用意して、ワーカースレッドをキューに対して待機させます。マスタースレッドは接続を受け入れるとタスクキューにタスク（またはタスクに対する入力データ）を押し込んでワーカースレッドを起こします。ワーカースレッドのどれか1つがタスクをキューから取得して、残りのワーカースレッドは再び待機状態に戻ります。マスタースレッドも次の接続受け入れ処理に戻ります。このようなモデルを「タスクキューモデル」と呼びます。タスクキューモデルは、並行処理に有効なモデルです。

App.5-3　BlockingQueueとBlockingDequeを使う待機処理

java.util.concurrentパッケージにあるBlockingQueueやBlockingDequeを使うと通知処理を使わずにタスクキューモデルを実装可能です。

BlockingQueueとBlockingDequeはどちらもインタフェースです。BlockingQueueとBlockingDequeの違いはデータ構造の違いです。違いは「**8章 コレクションと配列**」の「スタック、キュー、デック」を参照してください。ここでは説明をBlockingQueueに限定します。

あるスレッドがタスクやデータをBlockingQueueオブジェクトに要素として追加します。そのタスクやデータを待つ別スレッドがBlockingQueueオブジェクトから要素を取り出します。追加や取得にタイムアウト値の指定も可能です。BlockingQueueの実装クラスの1例としてArrayBlockingQueueオブジェクトを使う例を示します（**リスト5.3**）。動作はリスト5.2と同じです。Producerスレッドがコンソールでユーザ入力を待ちます。ユーザ入力があるとBlockingQueueのputメソッド経由でConsumerスレッドを起こします。takeメソッド呼び出しでブロックしていたConsumerスレッドの処理が再開します。

リスト5.3　BlockingQueueを使う待機処理

```
import java.util.*;
import java.util.concurrent.*;
import java.io.Console;

public class MyProduceConsumeWithBlockingQueue {
    public static void main(String... args) {
        try {
            var queue = new ArrayBlockingQueue<String>(8); // 共有キュー

            // プラットフォームスレッドと仮想スレッドのいずれかを有効化
            //ThreadFactory thFactory = Thread.ofPlatform().factory();
            ThreadFactory thFactory = Thread.ofVirtual().factory();
            ExecutorService executor = Executors.newFixedThreadPool(8, thFactory);

            Future<?> producer = executor.submit(new Producer(queue));
            Future<?> consumer = executor.submit(new Consumer(queue));

            producer.get();
            consumer.get();
            executor.shutdown();
            executor.awaitTermination(10, TimeUnit.SECONDS);
        } catch (InterruptedException | ExecutionException e) {
            e.printStackTrace();
        }
    }
}

class Producer implements Runnable {
    private final BlockingQueue<String> queue;

    Producer(BlockingQueue<String> queue) {
        this.queue = queue;
    }
```

```
    @Override
    public void run() { // 通知スレッドのエントリポイント
        Console console = System.console();
        try {
            while (true) {
                System.out.println("input any string");
                String s = console.readLine(); // ユーザの入力待ち
                queue.put(s); // 待機スレッドを起こす
            }
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}

class Consumer implements Runnable {
    private final BlockingQueue<String> queue;

    Consumer(BlockingQueue<String> queue) {
        this.queue = queue;
    }

    @Override
    public void run() { // 待機スレッドのエントリポイント
        try {
            while (true) {
                String s = queue.take();// ここで待機
                System.out.println(s + " is consumed");
            }
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

App.5-4 Futureオブジェクトを使う待機処理

Futureオブジェクトを「**20章 スレッド**」で説明しました。Futureオブジェクトからスレッド実行結果を取得できます。実行結果を取得できるまで待機するのでスレッド実行の順序制御に使えます。

リスト5.4はリスト5.3と同じです。ret1を返すスレッドの実行中、待機スレッドはFutureオブジェクトのgetメソッド呼び出しでブロックします。ret1を返すスレッドの実行が完了すると、待機スレッドの処理が再開します。

リスト5.4 Futureオブジェクトを使う待機処理

```
import java.util.concurrent.*;

public class Main {
    private static boolean done = false;
    public static void main(String... args) {
        try {
            // プラットフォームスレッドと仮想スレッドのいずれかを有効化
            //ThreadFactory thFactory = Thread.ofPlatform().factory();
            ThreadFactory thFactory = Thread.ofVirtual().factory();
            ExecutorService executor = Executors.newFixedThreadPool(8, thFactory);

            Future<?> ret1 = executor.submit(() -> {
                emulateLongTask(2000);
                Main.done = true;
            });

            Future<?> ret2 = executor.submit(() -> {
                try {
                    ret1.get(); // 順序制御のための待機（タイムアウト設定も可能）
                } catch (InterruptedException | ExecutionException e) {
                    e.printStackTrace();
                }
                System.out.println("done is " + Main.done);
            });

            ret2.get();
            executor.shutdown();
            // サブスレッド終了を待機
            executor.awaitTermination(10, TimeUnit.SECONDS);

        } catch (InterruptedException | ExecutionException e) {
            e.printStackTrace();
        }
    }

    // Thread.sleepで待機処理を模倣（以後のサンプルコードでも使います）
    private static void emulateLongTask(Duration duration) {
        try {
            Thread.sleep(duration);
        } catch (InterruptedException e) { /* 無視 */ }
    }
}
```

App.5-5 CompletableFutureオブジェクトを使う順序制御

順序制御にはFutureよりCompletableFutureのほうが便利です。

最初にCompletableFutureオブジェクトを直接生成するコード例を紹介します。new式で直接オブジェクトを生成しない書き方は次に説明します。**リスト5.5**は今までのコードに近い処理をCompletableFutureで書き直したコードです。ただし、どこにも待機処理に見えるコードはありません。これがCompletableFutureを使うコードの特徴です。

リスト5.5　CompletableFutureオブジェクト

```
import java.util.concurrent.*;

public class Main {
    public static void main(String... args) throws InterruptedException {
        // プラットフォームスレッドと仮想スレッドのいずれかを有効化
        //ThreadFactory thFactory = Thread.ofPlatform().factory();
        ThreadFactory thFactory = Thread.ofVirtual().factory();
        ExecutorService executor = Executors.newFixedThreadPool(8, thFactory);

        var cfuture = new CompletableFuture<String>();
        cfuture.thenAccept((ret) -> {
            // Workerスレッドがcompleteした時に呼ばれる処理
            System.out.println("ret is " + ret); // ret is doneを表示
        });
        executor.submit(new Worker(cfuture));
        executor.shutdown();
        executor.awaitTermination(10, TimeUnit.SECONDS);
    }
}

// スレッドのエントリポイント
class Worker implements Runnable {
    private final CompletableFuture<String> cfuture;

    public Worker(CompletableFuture<String> cfuture) {
        this.cfuture = cfuture;
    }

    @Override
    public void run() {
        System.out.println("run " + Thread.currentThread());
        emulateLongTask(2000); // リスト5.4参照
        this.cfuture.complete("done"); // 実行完了
    }

    private void emulateLongTask(int msec) {
        try {
            Thread.sleep(msec);
        } catch (InterruptedException e) {}
    }
}
```


待機処理はCompletableFutureオブジェクトのthenAcceptメソッドに隠蔽されています。CompletableFutureオブジェクトのcompleteメソッドを呼び出すと、thenAcceptの引数に渡したラムダ式を実行します。complete呼び出し時に渡した実引数が、ラムダ式の実引数として渡ります。リスト5.5であれば"done"の文字列がラムダ式の引数のretに渡る形です。

CompletableFutureオブジェクトは内部に状態を持ちます。completeメソッド呼び出しの意味はcomplete状態への変更です。状態がcomplete状態になるとthenAcceptメソッドに渡したラムダ式を呼びます。処理の呼び出し関係に混乱する人は「**11章　インタフェース**」の「関数型インタフェースを使うコールバックパターン」を読み返してください。コードの構造は同じです。thenAcceptの引数に渡したラムダ式は裏で積まれるだけで、実行タイミングは別である点を理解してください。

App.5-5-1　CompletableFutureのファクトリメソッド

CompletableFutureオブジェクトのファクトリメソッドのsupplyAsyncとrunAsyncを紹介します（**リスト5.6**）。2つの違いは引数に渡す関数型インタフェースのみです。

リスト5.6　CompletableFutureのファクトリメソッド

```
// CompletableFuture.javaから編集して抜粋
public static <U> CompletableFuture<U> supplyAsync(Supplier<U> supplier)
public static CompletableFuture<Void> runAsync(Runnable runnable)
```

引数に渡すSupplierやRunnableオブジェクトの実行開始前にCompletableFutureオブジェクトは生成され、ファクトリメソッドの実行が完了します。この動作はFutureオブジェクトを返すメソッドと同じです。

CompletableFuture.runAsyncメソッドを使う例を示します（**リスト5.7**）。リスト5.6の場合、明示的にCompletableFutureのcompleteメソッドを呼び出しました。リスト5.7の場合、runAsyncメソッドの引数に渡したラムダ式の実行が終了するとCompletableFutureオブジェクトがcomplete状態になります。complete状態になるとthenRunメソッドの引数のラムダ式を実行します。実行順序の視点で見ると、runAsyncの引数のラムダ式の実行とthenRunの引数のラムダ式の実行の順序を制御できています。

リスト5.7　CompletableFuture.runAsyncメソッド

```
import java.util.concurrent.*;

public class Main {
    private static boolean done = false;
    public static void main(String... args) {
        CompletableFuture<?> ret1 = CompletableFuture.runAsync(() -> {
            emulateLongTask(2000);
            Main.done = true;
```

```
        });

        CompletableFuture<?> ret2 = ret1.thenRun(() -> {
            System.out.println("done is " + Main.done); // done is trueを表示
        });

        // サブスレッド終了を待機
        ret2.join();
    }
}
```

App.5-5-2　CompletableFutureのthenメソッド

リスト5.5でthenAcceptメソッド、リスト5.7でthenRunメソッドを使いました。追加でthenApplyメソッドも含めて説明します。これらの違いは引数のみです（**リスト5.8**）。他にもthenがつくメソッドがあります。本書は総称してthenメソッドと記載します。

リスト5.8　CompletableFutureのthenメソッド

```
CompletableFuture<U> thenApply(Function<? super T,? extends U> fn)
CompletableFuture<Void> thenAccept(Consumer<? super T> action)
CompletableFuture<Void> thenRun(Runnable action)
```

thenメソッドの引数に渡す処理をactionと記述します。サンプルコードではラムダ式を渡します。メソッド参照でも関数型インタフェース型のオブジェクトでも記述可能です。

紙幅の都合ですべてのthenメソッドの説明は省略します。代わりに原則のみ説明します。thenメソッドの原則はCompletableFutureがcomplete状態になった時のactionの実行です。これはリスト5.5とリスト5.7で説明しました。もう1つのthenメソッドの原則が返り値です。thenメソッドの返り値はCompletableFutureオブジェクトです。このCompletableFutureオブジェクトは引数のactionの実行状態を表現します。この特徴により次節の連鎖処理を自然に記述できます。

App.5-5-3　CompletableFutureの連鎖処理

CompletableFutureを使うとタスク実行の継続処理を簡易に記述できます。タスク処理をthenメソッドでつなぐ実装パターンを**リスト5.9**に示します。実行するとタスク1、タスク2、タスク3を順に実行します。thenApplyメソッドのactionの返り値が次のthenApplyメソッドのactionの引数になる点を確認してください。

リスト5.9　CompletableFutureの連鎖処理

```
import java.util.concurrent.*;

public class Main {
    public static void main(String... args) throws InterruptedException, ExecutionException {
        CompletableFuture<String> cf = CompletableFuture.supplyAsync(() -> {
            System.out.println("タスク1");
            emulateLongTask(2000);
            return "ret1";
        }).thenApply((s) -> {
            System.out.println("タスク2: 引数 " + s); // sは"ret1"
            emulateLongTask(1000);
            return "ret2";
        }).thenApply((s) -> {
            System.out.println("タスク3: 引数 " + s); // sは"ret2"
            emulateLongTask(1000);
            return "ret3";
        });
        String s = cf.get(); // 最後のタスクの終了を待機
        System.out.println("タスク: 結果 " + s); // sは"ret3"
    }
}
```

CompletableFuture.supplyAsyncメソッドおよびすべてのthenメソッドの返り値はCompletableFutureオブジェクトです。これらのCompletableFutureオブジェクトは引数に渡したactionの実行開始前に生成されます。

リスト5.9の変数cfの参照先オブジェクトはCompletableFuture.supplyAsyncメソッドの返り値ではありません。連鎖の最後のthenApplyメソッドの返り値です。コードの見た目から誤解しやすいので注意してください。

ストリーム処理が繰り返し処理の実装パターンであるように、CompletableFutureを使うと並行処理の実行結果を待つ処理をつなぐ実装パターンを実現できます。

App.5-5-4　CompletableFutureの例外処理（try文）

thenメソッドのaction内で発生した例外について説明します。本節で説明する例外は実行時例外に限定します。検査例外は捕捉して実行時例外に例外翻訳する想定です(注2)。

CompletableFutureの連鎖処理のactionで発生した実行時例外は普通に外側のtry文のcatch節で捕捉できます（**リスト5.10**）。例外が発生すると後続のactionを実行しません。リスト5.10を実行すると「タスク3」は出力しません。cf.get()による最後のタスクの終了の待機処理も実行しません。

（注2）　「**14章 例外処理**」の「ラムダ式と例外」で説明した理屈です。

リスト5.10　CompletableFutureの例外処理（try文）

```
import java.util.concurrent.*;

public class Main {
    public static void main(String... args) throws InterruptedException, ExecutionException {
        try {
            CompletableFuture<String> cf = CompletableFuture.supplyAsync(() -> {
                System.out.println("タスク1");
                emulateLongTask(2000);
                return "ret1";
            }).thenApply((s) -> {
                System.out.println("タスク2: 引数 " + s);
                emulateLongTask(1000);
                throw new RuntimeException("例外発生の模倣");
            }).thenApply((s) -> {
                // このタスクは実行しない
                System.out.println("タスク3: 引数 " + s);
                emulateLongTask(1000);
                return "ret3";
            });

            // 下記を実行しない
            String s = cf.get(); // 最後のタスクの終了を待機
            System.out.println("タスク: 結果 " + s);
        } catch (RuntimeException e) {
            e.printStackTrace();
        }
    }
}
```

App.5-5-5　CompletableFutureの例外処理（exceptionallyメソッド）

CompletableFutureのexceptionallyメソッドを使うとtry文を使わない例外処理を記述できます。exceptionallyメソッドの定義を**リスト5.11**に示します。exceptionallyメソッドには類似メソッドがあります。本書はそれらを総称してexceptionallyメソッドと記載します。類似メソッドの説明は省略して原則のみ説明します。詳細はAPIドキュメントを参照してください。

リスト5.11　CompletableFutureのexceptionallyメソッド

```
CompletableFuture<T> exceptionally(Function<Throwable, ? extends T> fn)
```

便宜上、exceptionallyメソッドの引数に渡す処理もactionと記載します。exceptionallyメソッドの返り値はCompletableFutureオブジェクトです。thenメソッド同様、actionの実行と無関係に返ります。

thenメソッドのactionの中で実行時例外が発生すると、exceptionallyメソッドに渡したactionを実行します。例外が発生したthenメソッドとexceptionallyメソッドの間にあるthenメソッドのactionはスキップします。exceptionallyメソッドに渡したactionを普通に終了すると再び後続のthenメソッドのactionを実行します。exceptionallyメソッドのactionから実行時例外を再送すると、後続のthenメソッドのaction実行をスキップし、後続のexceptionallyメソッドのaction実行に移ります。

連鎖処理の最後のactionが例外送出で終了した場合は実行時例外が送出されます。この例外を捕捉したい場合は外側のtry文で捕捉します。exceptionallyメソッドを使う具体例を**リスト5.12**に示します。

リスト5.12　CompletableFutureの例外処理（exceptionallyメソッド）

```
import java.util.concurrent.*;

public class Main {
    public static void main(String... args) throws InterruptedException, ExecutionException {
        CompletableFuture<String> cf = CompletableFuture.supplyAsync(() -> {
            System.out.println("タスク1");
            emulateLongTask(2000);
            return "ret1";
        }).thenApply((s) -> {
            System.out.println("タスク2: 引数 " + s);
            emulateLongTask(1000);
            throw new RuntimeException("例外発生の模倣");
        }).thenApply((s) -> {
            System.out.println("実行されないタスク3: 引数 " + s);
            emulateLongTask(1000);
            return "ret3";
        }).exceptionally((ex) -> {
            System.out.println("例外処理タスク");
            emulateLongTask(1000);
            return "ex-ret"; // 例外処理タスクをreturn文で正常終了
        }).thenApply((s) -> {
            // 上記の例外処理の後にこのタスクを実行
```

C O L U M N

マルチスレッドと順序制御

スレッド1つの実行に限定すると順序制御自体は実は簡単です。ブロッキング処理で書いた処理は書いたとおりの順序で動くからです。順序制御が必要になるのは複数スレッドに別れた処理の順序制御です。

CompletableFutureが真価を発揮するのはノンブロッキング処理を使う場合です。ノンブロッキング処理の順序制御のコードは複雑になりがちだからです。実例は「**App.6 ファイルとネットワーク**」で紹介します。

```
            System.out.println("タスク4: 引数 " + s);
            emulateLongTask(1000);
            return "ret4";
        });

        String s = cf.get(); // 最後のタスクの終了を待機
        System.out.println("タスク: 結果 " + s);
    }
}
```

```
// 実行
$ java Main.java
タスク1
タスク2: 引数 ret1
例外処理タスク
タスク4: 引数 ex-ret
タスク: 結果 ret4
```

App.5-5-6　whenCompleteメソッドとhandleメソッド

whenCompleteメソッドもしくはhandleメソッドを使うと正常時と例外発生時のどちらでも実行するactionを記述できます。2つのメソッドの定義を**リスト5.13**に示します。2つの違いはactionがBiConsumerかBiFunctionかの違いです。whenCompleteはthenAcceptとexceptionallyを合わせたもの、handleはthenApplyとexceptionallyを合わせたもの、と理解してください。

リスト5.13　whenCompleteメソッドとhandleメソッド

```
CompletableFuture<T> whenComplete(BiConsumer<? super T, ? super Throwable> action)
CompletableFuture<U> handle(BiFunction<? super T, Throwable, ? extends U> fn)
```

whenCompleteメソッドとhandleメソッドには類似メソッドがあります。本書では類似メソッドを総称してwhenメソッドとhandleメソッドと記述します。類似メソッドの説明は省略します。

例外なしの場合、whenメソッドとhandleメソッドの動作はそれぞれthenAcceptおよびthenApplyと同じです。

例外発生時のwhenメソッドとhandleメソッドの動作はexceptionallyメソッドと同じです。actionをretunr文で正常終了すると後続の連鎖処理を続けます。actionを例外送出で抜けると後続のexceptionallyメソッドまたはwhenメソッドまたはhandleメソッドのactionを実行します。

handleメソッドを使う具体例を**リスト5.14**に示します。handleメソッドのactionには引数が2つあります。最初の引数は連鎖処理の直前のactionの正常終了時の結果です。2つ目の引数は例外です。正常終了と例外発生は排他なので、2つの引数のうちどちらかは必ずnullになります。直前のactionがRunnableやConsumerなど値を返さない関数型インタフェースの場合、正常終了時でも結果がnullになります。

リスト5.14　CompletableFuture<String>のhandleメソッド

```
import java.util.concurrent.*;

public class Main {
    public static void main(String... args) throws InterruptedException, ExecutionException {
        CompletableFuture<String> cf = CompletableFuture.supplyAsync(() -> {
            System.out.println("タスク1");
            emulateLongTask(2000);
            return "ret1";
        }).handle((s, ex) -> {
            System.out.println("タスク2: 引数 %s, 例外 %s".formatted(s, ex));
            emulateLongTask(1000);
            throw new RuntimeException("例外発生の模倣");
        }).handle((s, ex) -> {
            System.out.println("タスク3: 引数 %s, 例外 %s".formatted(s, ex));
            emulateLongTask(1000);
            return "ret3";
        }).handle((s, ex) -> {
            System.out.println("タスク4: 引数 %s, 例外 %s".formatted(s, ex));
            emulateLongTask(1000);
            return "ret4";
        });

        String s = cf.get(); // 最後のタスクの終了を待機
        System.out.println("タスク: 結果 " + s);
    }
}
```

```
// 実行
$ java Main.java
タスク1
タスク2: 引数 ret1, 例外 null
タスク3: 引数 null, 例外 java.util.concurrent.CompletionException: java.lang.RuntimeException: 例外発生の模倣
タスク4: 引数 ret3, 例外 null
タスク: 結果 ret4
```

App.5-5-7　複数CompletableFutureの並行実行

複数タスクを同時実行して、その完了を待ちたい場合があります。たとえば同時実行可能な通信処理があり、すべての実行完了を待ってから処理したい場合などです。あるいは最初に完了した通信処理の結果のみ使えればよい場合もあります。

このために使えるメソッドがCompletableFuture.allOfメソッドとCompletableFuture.anyOf

メソッドです(**リスト5.15**)。メソッドの引数に複数CompletableFutureオブジェクトを渡します。メソッドの返り値はCompletableFutureオブジェクトです。他のメソッド同様、引数に渡したaction実行前にCompletableFutureオブジェクトは返ります。

リスト5.15　複数CompletableFutureの並行実行

```
public static CompletableFuture<Void> allOf(CompletableFuture<?>... cfs)
public static CompletableFuture<Object> anyOf(CompletableFuture<?>... cfs)
```

allOfは引数すべてのCompletableFutureオブジェクトがcomplete状態になると返り値のCompletableFutureオブジェクトがcomplete状態になります。anyOfは引数のどれかのCompletableFutureオブジェクトがcomplete状態になると返り値のCompletableFutureオブジェクトがcomplete状態になります。

CompletableFuture.allOfメソッドを使う具体例を**リスト5.16**に示します。

リスト5.16　CompletableFuture.allOfメソッド

```
import java.util.stream.Stream;
import java.util.concurrent.*;

public class Main {
    public static void main(String... args) {
        var inputs = Stream.of(1000, 10000, 2000); // 1秒、10秒、2秒

        @SuppressWarnings("unchecked") // 配列生成の型警告を抑制
        CompletableFuture<String>[] tasks = inputs.map(msec -> CompletableFuture.runAsync(() -> {
            emulateLongTask(msec);
        })).toArray(CompletableFuture[]::new);

        CompletableFuture<?> cf = CompletableFuture.allOf(tasks);
        cf.join(); // タスクの終了を待機(10秒後)
    }
}
```

CompletableFuture.allOfメソッドの可変長引数にCompletableFutureオブジェクトの配列を渡します。リスト5.16の3つのタスクはそれぞれ1秒、10秒、2秒かかります。CompletableFuture.runAsyncメソッド呼び出しでタスク実行を開始します。タスク実行開始より前にCompletableFutureオブジェクトが返るので、リスト5.16のCompletableFuture.allOfメソッド呼び出し時点で3タスクは実行開始済みの可能性があります。開始前か開始済みかはタイミング次第です。CompletableFuture.allOfメソッドの返り値のCompletableFutureオブジェクトもすぐに返ります。このCompletableFutureオブジェクトがcomplete状態になるのはすべてのタスクの完了後です。リスト5.16の場合、最長の10秒タスクの完了後です。

リスト5.16のCompletableFuture.allOfをCompletableFuture.anyOfに書き換えると、タスクのどれかの完了でanyOfメソッドの返り値のCompletableFutureオブジェクトがcomplete状態になります。リスト5.16であれば1秒タスク完了時です。残りの2タスクの実行は裏で継続します(注3)。

App.5-5-8　タスクそのものとタスク結果のCompletableFutureの違い

ラムダ式やメソッド参照を何かのメソッドの引数に渡すコードとCompletableFutureオブジェクトを引数に渡すコードは混乱しがちです。どちらもタスクの実行を保留する点で類似しているからです。CompletableFuture.allOfメソッドおよびanyOfメソッドの引数に渡すのはCompletableFutureオブジェクトです。引数にラムダ式などのタスクをそのまま渡すのは間違いです。**リスト5.17**のコードはコンパイルエラーになります。

リスト5.17　コンパイルエラーになるコード

```
Runnable task1 = () -> System.out.println("task1")
Runnable task2 = () -> System.out.println("task2")
CompletableFuture<?> cf = CompletableFuture.allOf(task1, task2);
```

リスト5.17を書き直したコードが**リスト5.18**です。

リスト5.18　リスト5.17の修正例

```
CompletableFuture<?> cf = CompletableFuture.allOf(CompletableFuture.runAsync(() -> task1.run()),
                                                  CompletableFuture.runAsync(() -> task2.run()));
```

CompletableFutureを使うコードの場合、**リスト5.19**のようにCompletableFutureを返すメソッド設計にすると全体として統一感のあるコードにできます。

リスト5.19　CompletableFutureオブジェクトを返却するメソッド

```
CompletableFuture<?> runTask() {
    return CompletableFuture.runAsync(() -> {
        emulateLongTask(2000);
    });
}
```

(注3)　allOfをanyOfに書き換えると1秒タスク完了でmainメソッドを抜けてしまいプロセスが終了します。残り2タスク完了を待つにはcf.join()の後に待機処理を入れてください。

App.6 ファイルとネットワーク

ファイルやネットワークを扱う方法を説明します。Webアプリなどサーバ開発で本章の内容を直接使う機会は多くありません。フレームワークなどが詳細を隠蔽するからです。しかし内部的な仕組みを理解しておくと問題解決に役立ちます。

App.6-1 JavaのI/O処理の概要

I/OはInput/Outputの略です。IO処理を日本語にすると入出力処理です。Javaプログラムが外部とデータ送受信する処理を意味します。本章はI/O処理の代表であるファイルとネットワークを扱います。

ファイルとネットワークを扱う標準ライブラリのパッケージは、**表6.1**に示す4つです。java.nioパッケージは後発なので、部分的にjava.ioおよびjava.netと機能が被っています。

表6.1　JavaのI/O処理関連の標準ライブラリ

パッケージ名	主な提供機能
java.io	I/O処理をI/Oストリームとして抽象化した機能
java.net	ネットワーク操作のソケットAPI
java.nio	I/O処理をチャネルとして抽象化したAPI。ファイルのパス操作の基本API
java.net.http	HTTPクライアントAPI。「22章　Web技術」参照

本書はjava.nioパッケージのチャネルAPIと、チャネルAPIと一緒に使うjava.netパッケージを中心に説明します。java.ioパッケージのI/Oストリームの使い方は他の書籍をご覧ください。

本章で扱うクラスやインタフェースは複数のパッケージにまたがっています。本章で型を言及する際、最初のみ完全修飾名を記載しその後は単純名の記載にします。

App.6-1-1　バイト列と文字コード

I/O処理で送受信するデータはバイト単位の処理が原則です（バイトについては「**15-2　文字とバイト**」参照）。受信したバイト列が文字列であれば文字列として解釈、構造化された文字列であればその構造を解釈、バイト列が画像や音声のデータであればそれぞれに応じた解釈をします。このような解釈をパース処理やデコード処理と呼びます。送信時は逆で内部データをバイト列に変換して送信します。

なおプログラムの性質によってはバイト列のまま処理したほうが適切な場合もあります。たとえば受信した文字列のバイト列をそのまま出力するプログラムであればわざわざ内部で文字列に変換しないほうが効率的です。

外部と取り決めさえあれば、JavaのStringオブジェクトの内部バイト列をそのまま外部に送信しても問題ありません。いわゆるUTF-16エンコーディングの入出力動作になります。ただし現在ではUTF-16のデータ入出力は主流ではありません。外部で扱う文字列のバイト列表現はUTF-8が中心だからです。本章は文字列の入出力バイト列表現にUTF-8を使う想定とします。UTF-8以外の文字コードの扱いは「**App.7 国際化**」を参照してください。

App.6-1-2　I/O処理の例外

ファイルやネットワークはプログラムの外部世界なので、予期せぬ異常の発生を避けられません。異常発生時、標準ライブラリの大半のメソッドは例外を投げる仕様になっています。

I/O処理で発生する例外は、原則java.io.IOException例外またはその派生例外で統一されています。IOExceptionは検査例外です。「**14章　例外処理**」で説明したように検査例外には扱いにくい面があります。扱いやすくするため、実行時例外への例外翻訳の技法があります。この際の例外翻訳に使える実行時例外としてjava.io.UncheckedIOExceptionが存在します。IOException例外を実行時例外に例外翻訳する場合、UncheckedIOException例外あるいは派生例外を使うとコードの統一性を得られます。

本章のサンプルコードはIOException例外を捕捉してe.printStackTrace()をするだけです。実開発では適切な例外処理をしてください。

App.6-1-3　I/O処理とリソース解放処理

I/O処理で使う多くのオブジェクトにclose処理が必要です。close処理が必要なクラスはjava.io.Closableインタフェースを実装しています。ClosableインタフェースはAutoCloseableインタフェースを継承しているのでtry-with-resource文で解放処理を自動化できます。明示的なcloseメソッド呼び出しをせず、try-with-resource文を使うのが定石です。

■I/O処理とストリーム処理

「**10章 ストリーム処理**」で説明したストリーム処理とI/O処理を組み合わせて使う場合があります。

StreamインタフェースはAutoClosableインタフェースを継承しています。I/O処理以外で使うStreamオブジェクトには、通常close呼び出し不要です。しかし、I/O処理と組み合わせて使うStreamオブジェクトには、close処理が必要になる場合があります。close処理が必要なStreamオブジェクトの場合、try-with-resource文でclose処理を自動化してください。何をデータソースにした場合にclose処理が必要かを知るには、APIドキュメントの確認が必要です。

App.6-2　ファイル処理

java.nio.FileChannelオブジェクトを使うと、ファイルに対するデータ読み書きができます。

FileChannelオブジェクトは直接new式で生成しません。FileChannel.openメソッドもしくは後述するFilesユーティリティクラスのメソッドで生成します。

FileChannelオブジェクトの基本操作はopen、read、write、closeの4つ組です。openでファイル読み書き開始、readで読み込み、writeで書き込み、closeで読み書き終了です。

■FileChannelの利用例

FileChannelオブジェクトを使ったコードを**リスト6.1**に示します。なおこのコード自体でできることはFiles.copyメソッド（後述）と等価です。説明のためのコードと理解してください。

リスト6.1　FileCopy.java

```
// 後述のコード例では紙幅の節約のためオンデマンドインポートにします
import java.util.EnumSet;
import java.io.IOException;
import java.nio.ByteBuffer;
import java.nio.file.Path;
import java.nio.file.StandardOpenOption;
import java.nio.channels.FileChannel;
import java.nio.charset.StandardCharsets;

public class FileCopy {
    public static void main(String... args) {
        // コマンドライン引数チェック。下記のコード例では省略します
        if (args.length <= 1) {
            System.out.println("コマンドライン引数でコピー元ファイル名とコピー先ファイル名を指定してください");
            System.exit(0);
        }

        var srcFname = Path.of(args[0]);
        var destFname = Path.of(args[1]);

        try (FileChannel srcFile = FileChannel.open(srcFname, StandardOpenOption.READ);
             FileChannel destFile = FileChannel.open(destFname, EnumSet.of(StandardOpenOption.CREATE_
NEW, StandardOpenOption.WRITE))) {
            ByteBuffer buf = ByteBuffer.allocate(4096);
            int rlen;
            while ((rlen = srcFile.read(buf)) != -1) {  // ファイルから読み込み
                buf.flip();       // ByteBufferの有効データ領域をリセット（positionを先頭に戻す）
                destFile.write(buf);
                buf.clear();      // ByteBufferをクリア（positionを先頭に戻す）
            }
        } catch (IOException e) { // 包括的に例外を捕捉。以後のコードも同様
            e.printStackTrace();
        }
    }
}
```

リスト6.1はコマンドライン引数でコピー元ファイルとコピー先ファイルの2つのファイルパス名を指定する前提です。Path.ofメソッドでファイルパス名からjava.nio.file.Pathオブジェクトに変換します。この意味は後述します。

FileChannel.openメソッドの引数にPathオブジェクトとモードを渡します。モードはjava.nio.file.StandardOpenOptionの定数で指定します。読み込み可能、書き込み可能、ファイルが存在しなければ新規生成する、などの指定が可能です。リスト6.1の場合、コピー元ファイルを読み込み専用、コピー先ファイルを書き込み専用で開いています。StandardOpenOption.CREATE_NEWオプションは必須ではありません。この指定があると同名のファイルがすでに存在する場合にオープンが失敗します。コピーで誤った既存ファイルを上書きしないために指定しています。

リスト6.1を読解するにはFileChannelオブジェクトとByteBufferオブジェクトを知る必要があります。それぞれ説明します。

App.6-2-1　FileChannelオブジェクト

FileChannelオブジェクトの理解にはファイルポジションの理解が助けになります。

ファイルポジションは、ファイル読み書き中の現在位置を示す値です。ファイルの先頭から数え始める0から始まるオフセット値です。ファイルの中身をメモリ上の配列と等価に見立てると、ファイルポジションは配列のインデックス値と同じ役割をする数値です。

FileChannelオブジェクトは、オープンしたファイルのファイルポジション値を保持します。FileChannelオブジェクト生成直後のファイルポジション値は0です。ファイルに対してreadやwriteをすると自動的にファイルポジションが進みます。たとえば10バイトのreadをするとファイルポジション値は10になります。この状態で20バイトのreadを行うと、ファイルの先頭10バイト目から20バイト分読み出し、ファイルポジション値は30になります。

先頭から順にreadし続けるとファイルポジション値は増え続けます。ファイル末尾に達するとreadはメソッドの返り値として-1を返します。ファイルポジション値はそれ以上増えません。

writeメソッドによる書き込みでも自動的にファイルポジションが進みます。ファイルポジションはreadとwriteで共通です。ファイルオープン直後、つまりファイルポジション値0の状態から10バイトをwriteすると、ファイルの先頭に10バイトを書き込みます。既存ファイルへの書き込みであれば上書きで書き込みます。次にreadで20バイトを読み込むとファイルの先頭10バイト目から20バイト分を読み込み、ファイルポジション値は30に変化します。ファイル末尾でwriteするとファイルポジション値が増え続けます。この時、ファイルサイズも増え続けます。

ファイルポジション値は明示的にも変更可能です。ファイルの同じ位置からの再読み込みや明示的な読み飛ばしのために使えます。ファイルポジション値の取得および変更はpositionメソッドで行えます（**リスト6.2**）。

リスト6.2　ファイルポジション値の操作メソッド（FileChannel.javaから抜粋）

```
long position() throws IOException;                          // 現在のファイルポジションの取得
FileChannel position(long newPosition) throws IOException; // ファイルポジションの変更
```

App.6-2-2　ByteBufferオブジェクト

ByteBufferオブジェクトの直感的な理解は、ポジション機能付きバイト配列です。FileChannelオブジェクトで読み書きするデータのメモリ側の管理オブジェクトです。

ByteBufferオブジェクト生成時にバイト数でサイズを決めます。作成後のサイズ拡張はできません。

FileChannelのreadメソッドを呼ぶと、ファイルから読んだバイト列をByteBufferオブジェクトにコピーします。writeメソッドを呼ぶと、ByteBufferオブジェクトのバイト列をファイルに書き出します。readとwriteのメソッド定義は下記のようになっています。

```
// FileChannelオブジェクトのデータ読み書きメソッド
public int read(ByteBuffer dst) throws IOException;
public int write(ByteBuffer src) throws IOException;
```

ByteBufferオブジェクトは内部にpositionとlimitの2値を持ちます。positionはByteBufferの読み書き中の現在位置です。limitは有効データの終端を示します。ByteBufferのposition値からlimit値の間の範囲が有効データ扱いになります。生成直後のposition値は先頭でlimit値は終端です。

FileChannelからのread時、読んだバイト列をByteBufferにputします(注1)。putするごとにByteBufferのposition値が増えます。この動作により、ファイルから読み進めたバイト列を、連続putで順次ByteBufferに格納できます。FileChannelへのwrite時はByteBuferからバイト列をgetしてファイルに書き込みます。get時も同様にposition値が増えます。連続getでByteBufferからバイト列を取得してファイルに書き込めます。

通常、ファイルから読み出したバイト列を何かに利用します。たとえば画面出力や内容の解釈などです。FileChannelからのread直後のByteBufferのpositionは、読み込み済みバイト列の終端を指しています。ここでの定石コードがByteBufferのflipメソッド呼び出しです。flipメソッドは現在のposision値にlimitを移動し、positionを先頭に移動します。position値とlimit値の間の範囲が有効データ扱いなので、flipの結果、読み込み済みバイト列が有効データになります。flip後、ByteBufferからgetすると先頭から有効データのバイト列を順次取得できます。

もう1つのByteBufferの定石コードがclearメソッド呼び出しです。clearメソッドはByteBufferのpositionとlimitを初期状態に戻します。positionが先頭、limitが終端に戻ります。ByteBufferを再利用する時に使います。

ファイル読み書きで説明しましたが後述するネットワーク通信の読み書きでも原理は同じです。

ここまでの知識を使ってリスト6.1を読解すると次のようになります。whileループの条件式がFileChannelオブジェクトのread処理を繰り返します。返り値が-1になるまでread呼び出しを繰り返すと、結果としてファイル全体の読み込みを実現できます。ファイルからreadしたバイト列はByteBufferにputされます。使う視点で見ればファイルからByteBufferへのデータコピーです。while文の中でByteBufferの中身を出力ファイル先にwriteします。write呼び出しの前後のflipとclearの意味は上記を読み直してください。

■ファイルの文字列読み書き

ファイルの中身がUTF-8の文字列の場合、ファイルから読んだバイト列を**リスト6.3**のように文字列に変換できます。StandardCharsets.UTF_8.decodeメソッドでバイト列からUTF-8文字列に変換します。変換後の文字列をSystem.out.printlnで画面出力します。

リスト6.3　リスト6.1のtry文を抜粋して文字列読み書きコードに変更

```
try (FileChannel file = FileChannel.open(fname, StandardOpenOption.READ)) {
    ByteBuffer buf = ByteBuffer.allocate(4096);
    int rlen;
    while ((rlen = file.read(buf)) != -1) {  // ファイルから読み込み
        buf.flip();       // ByteBufferの有効データ領域をリセット（positionを先頭に戻す）
        // バイト列から文字列へ変換（多バイト文字では文字化けリスクあり）
        String s = StandardCharsets.UTF_8.decode(buf).toString();
        System.out.println(s);
        buf.clear();      // ByteBufferをクリア（positionを先頭に戻す）
    }
} catch (IOException e) { // 包括的に例外を捕捉。必要に応じて細分化
    e.printStackTrace();
}
```

ファイルの中身が1バイト文字（いわゆるASCIIコード相当。英数字文字）のみであればリスト6.3は問題ありません。しかし日本語の文字など多バイト文字の場合、文字化けが発生する可能性があります。多バイト文字の途中でバイト列が終端する可能性があるからです。

文字化け問題を修正するには多バイト文字のバイト境界を正しく扱う必要があります。もっとも容易な方法は行単位での読み込みです。行の終端では多バイト文字も必ず終端しているからで

（注1）　read・writeとget・putが逆の文脈になるので注意してください。ファイルからreadしたバイト列をByteBufferにputする関係になります。writeとgetも同様です。メソッドの動詞は原則としてメソッドを呼ぶ側の視点の動詞を使います。

す。行単位の読み込みは次節のFiles.linesメソッドの利用が便利です。

■Filesクラスのデータ読み書き操作

java.nio.file.Filesクラスはファイル操作のためのユーティリティクラスです。ここではFilesクラスの代表的なデータ読み書き用のクラスメソッドを紹介します（**表6.2**）。

表6.2　Filesクラスの代表的なデータ読み書き操作のクラスメソッド（throws節およびオプショナルな可変長引数を省略）

クラスメソッド定義	説明
List<String> readAllLines (Path path)	ファイルの中身をすべて読み込んで行単位文字列で返す
byte[] readAllBytes (Path path)	ファイルの中身をすべて読み込んでバイト列で返す
Stream<String> lines (Path path)	ファイルの中身をすべて読み込んで行単位文字列で返す
Path write (Path path, byte[] bytes)	指定したバイト列をファイルに書き込む
Path write (Path path, Iterable<? extends CharSequence> lines)	指定した行単位文字列をファイルに書き込む
SeekableByteChannel newByteChannel (Path path)	ファイル読み書き用のChannelを返す

Files.linesメソッドでファイルから行単位で文字列を読むコード例を示します（**リスト6.4**）。ファイルから読んだ文字列をSystem.out.printlnで画面出力します。

リスト6.4　ファイルから行単位で文字列読み込み

```
import java.io.IOException;
import java.util.stream.Stream;
import java.nio.file.Files;
import java.nio.file.Path;

public class Main {
    public static void main(String... args) {
        // コマンドライン引数チェック
        if (args.length <= 0) {
            System.out.println("コマンドライン引数で読み込みファイル名を指定してください");
            System.exit(0);
        }

        var fname = Path.of(args[0]);

        try (Stream<String> lines = Files.lines(fname)) {
            lines.forEach(System.out::println);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```


Files.linesメソッドの返り値はStreamオブジェクトです。Streamオブジェクトのclose処理が必要な点に注意してください。forEachの引数にメソッド参照のSystem.out::printlnを渡しています。行単位でこのメソッド呼び出しをします。

App.6-2-3　Pathオブジェクト

Pathオブジェクトは、ファイルのパスを表現するオブジェクトです。文字列で表現したファイル名やファイルパス名からPathオブジェクトを生成します。ファイルパス名は相対パス記述も可能です。

ほぼすべてのファイル関連メソッドは、対象ファイルをPathオブジェクトで指定します。Path.ofメソッドもしくはjava.nio.file.Pathsクラスのファクトリメソッドで Pathオブジェクトを生成できます（**リスト6.5**）。ファイルパスを表現するだけなので対象ファイルの存在とは無関係にPathオブジェクトを生成可能です。

リスト6.5　Pathオブジェクトの生成例

```
// 下記2つはどちらでも動きます
jshell> var path = Path.of("/etc/ssh/ssh_config")
jshell> var path = Path.of("/etc", "ssh", "ssh_config")

jshell> boolean isAbsolute = path.isAbsolute()
isAbsolute ==> true

jshell> Path fpath = path.getFileName()
fpath ==> ssh_config

jshell> Path dpath = path.getParent()
dpath ==> /etc/ssh
```

Pathオブジェクトそのものはファイルパスの表現のみでファイルに対する操作は提供しません。

■Filesクラスのファイルパス操作

ファイル名変更やファイル削除など、ファイル自体に対する操作をFilesクラスのクラスメソッドで実行できます。これらの代表的なメソッドをまとめます（**表6.3**と**表6.4**）。

表6.3　Filesクラスの代表的なパス操作クラスメソッド（throws節およびオプショナルな可変長引数を省略）

クラスメソッド定義	説明
Path createFile（Path path）	ファイルの新規作成
Path createDirectory（Path dir）	ディレクトリの新規作成
Path createDirectories（Path dir）	ディレクトリの新規作成 （途中のサブディレクトリも同時に作成）
Path createTempFile(String prefix, String suffix)	OS標準の一時ディレクトリに一時ファイルのファイル名を作成
Path createSymbolicLink（Path link, Path target）	シンボリックリンクファイルを作成
Path createLink（Path link, Path existing）	ハードリンクファイルを作成
boolean deleteIfExists（Path path）	ファイルを削除
Path copy（Path source, Path target）	ファイルをコピー
Path move（Path source, Path target）	ファイルを移動（リネーム）
long size（Path path）	ファイルのサイズを取得
FileTime getLastModifiedTime（Path path）	ファイルの最終更新時刻を取得
Path setLastModifiedTime（Path path, FileTime time）	ファイルの最終更新時刻を変更
Set<PosixFilePermission> getPosixFilePermissions（Path path）	ファイルのパーミッションを取得
Path setPosixFilePermissions（Path path, Set<PosixFilePermission> perms）	ファイルのパーミッションを変更
UserPrincipal getOwner（Path path）	ファイルの所有ユーザを取得
Path setOwner（Path path, UserPrincipal owner）	ファイルの所有ユーザを変更
boolean exists（Path path）	ファイルの存在チェック
boolean isDirectory（Path path）	パスがディレクトリかをチェック
boolean isRegularFile（Path path）	パスがファイルかをチェック

表6.4　Filesクラスの代表的なディレクトリ操作クラスメソッド（throws節およびオプショナルな可変長引数を省略）

クラスメソッド定義	説明
Path walkFileTree（Path start, FileVisitor<? super Path> visitor）	ディレクトリ配下のファイルごとにvisitor処理する
DirectoryStream<Path> newDirectoryStream（Path dir）	ディレクトリ配下のファイルパスのイテレータを返す
Stream<Path> list（Path dir）	ディレクトリ内のファイル一覧のパスStreamを返す
Stream<Path> walk（Path start, int maxDepth）	ディレクトリ配下のファイル一覧のパスStreamを返す
Stream<Path> find（Path start, int maxDepth, BiPredicate<Path, BasicFileAttributes> matcher）	ディレクトリ配下で条件に合致するファイル一覧のパスStreamを返す

Files.walkメソッドを使って指定ディレクトリ内のファイルを表示するコード例を示します（**リスト6.6**）。

リスト6.6　ディレクトリ内のファイル名を表示

```
// 紙幅の都合でオンデマンドインポートにします。以後も同様
import java.io.IOException;
import java.nio.file.*;

public class Main {
    public static void main(String... args) {
        if (args.length <= 0) {
            System.out.println("コマンドライン引数でディレクトリ名を指定してください");
            System.exit(0);
        }

        var dpath = Paths.get(args[0]);
        try {
            Files.walk(dpath).filter(Files::isRegularFile).forEach(System.out::println);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

App.6-2-4　ファイルの変更監視

java.nio.file.WatchServiceを使うと指定したディレクトリ内のファイルの変更を監視できます（**リスト6.7**）。

リスト6.7　ファイルの変更監視

```
import java.io.IOException;
import java.nio.file.*;
import static java.nio.file.StandardWatchEventKinds.*;

public class Main {
    public static void main(String... args) {
        if (args.length <= 0) {
            System.out.println("コマンドライン引数でディレクトリ名を指定してください");
            System.exit(0);
        }

        var dpath = Paths.get(args[0]);

        try (WatchService watchService = FileSystems.getDefault().newWatchService()) {
            // ENTRY_CREATEなどはStandardWatchEventKindsの定数
            dpath.register(watchService, ENTRY_CREATE, ENTRY_DELETE, ENTRY_MODIFY);

            while (true) {
                // 変化があるまで呼び出しが停止する（takeの代わりにpollだと停止しない）
                WatchKey key = watchService.take();

                key.pollEvents().stream().forEach((WatchEvent<?> event) -> {
                    System.out.println(event.kind());  // StandardWatchEventKindsの定数を出力
                    System.out.println((Path)event.context()); // 変更のあったファイル名を出力
                });
                if (!key.reset()) {
                    break;
                }
            }
```

```
        } catch (IOException | InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

App.6-2-5　標準入出力

標準入出力とは、Unix系OSのプロセスが起動時にオープン状態で受け取る「ファイル」のことです。この文脈の「ファイル」はディスク上のファイルに限りません。Unix系OSはプロセスの入出力全般を「ファイル」として抽象化するからです。たとえば、端末上で起動したプロセスの場合、標準入力に端末入力（キーボードからの文字入力）、標準出力と標準エラー出力に端末出力（画面への文字出力）を割り当てるのがデフォルト動作です。

標準入出力はjava.lang.Systemクラスのクラスフィールドで扱えます（**表6.5**）。これらはjava.ioパッケージの機能なので本章は説明を割愛します。

表6.5　標準入出力のフィールド

フィールド	型	意味
System.in	java.io.InputStream	標準入力
System.out	java.io.PrintStream	標準出力
System.err	java.io.PrintStream	標準エラー出力

System.outは今まで説明なしに使ってきました。printやprintlnのメソッド利用は半ばイディオムになっているからです。参考までに標準入力から読み込んだバイト列を標準出力に書き出す例を**リスト6.8**に示します。

リスト6.8　Systemクラスを使う標準入出力の読み書き

```
byte[] buf = new byte[4096];
while (System.in.read(buf) != -1) {
    System.out.write(buf);
}
```

標準入出力はUnix系OSに依存した機能なので、プラットフォーム独立のJava思想と理屈上は合いません。しかし、最近のOSのほとんどは標準入出力機能を持つので、用途に応じて使うと便利です。

■java.io.Consoleクラス

Systemクラスのconsoleメソッドでjava.io.Consoleオブジェクトを取得できます（**リスト6.9**）。Consoleオブジェクトを使うとより簡易に標準入出力を扱えます。端末があるか否かはプラットフォームに依存するため、consoleメソッドがnullを返しうる点に注意してください。

リスト6.9　Consoleクラスを使う標準入出力の読み書き

```
Console console = System.console(); // Consoleオブジェクトを取得
console.printf("Please input¥n");   // 標準出力に文字列表示
String line = console.readLine();   // ユーザの入力文字列を待つ
```

App.6-3　ネットワーク処理

App.6-3-1　ネットワークプログラミング概論

伝統的にネットワークのデータ入出力にソケットと呼ぶ抽象化を使います。通信先の相手と1対1につながる仮想的な線を考え、その端点にデータを読み書きする抽象化です。この端点を「ソケット」と呼びます。Javaでは、ソケット操作をI/Oストリームもしくはチャネルで扱います(注2)。

■ネットワークプログラミングの基本

ネットワークプログラミングの基本操作は、ソケットに対するデータの読み書きです。ソケットからデータを読み込むと通信相手から送られてきたデータを読み取れます。ソケットにデータを書き込むと、通信相手にデータを送信します。書いたデータを相手が読むかは相手次第、読もうとしたデータが届いているかは相手次第、という部分がファイルやメモリの読み書きとは異なります。

ソケットを使うにはまず通信相手との接続確立が必要です。なお、本書はネットワークの本ではないのでTCP/IPなどのレイヤの説明は割愛します。

通信確立の概略をWebの世界の言葉で説明します。Webでは http://www.apache.org のように通信相手先を指定します。この表記中の www.apache.org に当たる部分がホスト名（FQDN：Fully Qualified Domain Name）です。実際の通信時にはホスト名からIPアドレスに変換します。この変換を名前解決と呼びます。名前解決は多くの場合、DNSと呼ぶ仕組みで行います。名前解決の詳細はネットワークの本に譲ります。

接続相手を識別する第一歩が相手先のIPアドレスです。もう1つ、接続相手の識別のためにポート番号を使います。直感的な理解は、IPアドレスでコンピュータを特定し、ポート番号でそのコンピュータ上の実行プロセスを特定します(注3)。IPアドレスとポートのペアをエンドポイントと表記します。

（注2）　ソケットはネットワークの複雑なやりとりを「データを書くこと」と「データを読むこと」のみに抽象化したものです。
（注3）　1つのプロセスが複数のポートを利用可能なので、プロセスとポート番号の対応は1対多です。

■サーバとクライアント

ネットワークの世界ではサーバとクライアントと呼ぶ区分けをよく使います。ソケットにそのような区分けがあるのではなく、使い方に違いがあります。接続を待ち受けるソケットと自分から接続しに行くソケットの2種類の使い方です。前者を受動的ソケット、後者を能動的ソケットと呼ぶことにします。通常、サーバプログラムは受動的ソケットを使い、クライアントプログラムは能動的ソケットを使います。

サーバは待ち受けポート番号を決めて、受動的ソケットをオープンします。クライアントはサーバのIPアドレスと待ち受けポート番号に接続できるように能動的ソケットをオープンします。この時、通常は能動的ソケットはローカルのポート番号を自動採番します。

能動的ソケットが受動的ソケットに接続できると、受動的ソケットは、受け入れソケットを自動生成します。受け入れソケットには新しいポート番号を使います。このポート番号はたいてい自動採番です。実際にデータをやりとりするソケットは、サーバ側の受け入れソケットとクライアント側の能動的ソケットのペアです。

サーバ側の受動的ソケットはそのまま残っています。こうして受動的ソケットを持つサーバプロセスは、複数のクライアントからの接続を受け入れられます。

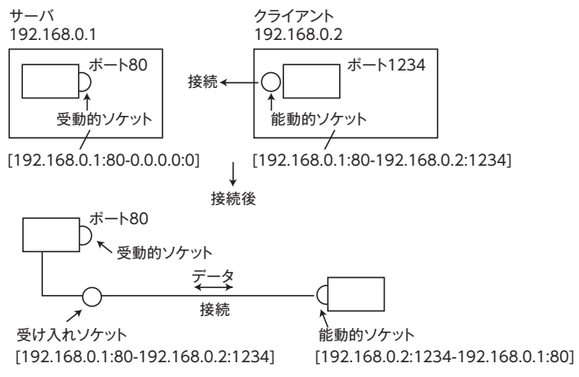
■ソケットと4つ組

ソケットは自分自身のエンドポイント、接続相手のエンドポイントのペアからなります。エンドポイントはIPアドレスとポート番号のペアです。ソケットは次のように4つ組で表記できます。

```
// ソケットの4つ組表記
[ローカルIPアドレス、ローカルポート番号、リモートIPアドレス、リモートポート番号]
```

サーバの受け入れソケットと、クライアントの能動的ソケットは対称関係の4つ組になります（**図6.1**）。通信両端のプロセスは、この対称関係のソケットにお互いが読み書きすることで、接続相手と通信できます。

図6.1 ソケットの4つ組



サーバとクライアントの接続開始までの動作は非対称ですが、いったん接続を確立すると、2つの通信プロセスの関係は対称になります。ソケットに対するデータの読み書きの中身や順序を規定するのは、上位の通信プロトコルです。WebであればHTTPやWebSocketなどが相当します。

通信終了時はソケットをクローズします。クローズはどちらからでも行えます。クローズ時の処理順序を決めるのも上位の通信プロトコルです。

App.6-3-2 クライアント処理

接続を開始する側のソケットを使うネットワーク処理を便宜上クライアント処理と呼びます。クライアント処理のコード例を**リスト6.10**に示します。

リスト6.10 クライアント処理

```
import java.io.IOException;
import java.net.InetSocketAddress;
import java.nio.ByteBuffer;
import java.nio.channels.SocketChannel;
import java.nio.charset.StandardCharsets;

public class Client {
    public static void main(String... args) {
        var host = "localhost"; // 接続先のホスト名
        int port = 8000; // 接続先のポート番号
        // 接続先のアドレス
        var address = new InetSocketAddress(host, port);

        try (SocketChannel channel = SocketChannel.open(address)) {
```

```
            // 送信バイト列用のByteBuffer確保
            ByteBuffer wbuf = ByteBuffer.allocate(4096);
            wbuf.put("GET / HTTP/1.1¥r¥n".getBytes());
            wbuf.put("Host: localhost¥r¥n".getBytes());
            wbuf.put("Connection: close¥r¥n¥r¥n".getBytes());
            wbuf.flip();
            // ソケットへ送信（書き込みバイト数チェック省略）
            channel.write(wbuf);

            // 受信バイト列用のByteBuffer確保
            ByteBuffer rbuf = ByteBuffer.allocate(4096);
            int rlen;
            // ソケットから受信
            while ((rlen = channel.read(rbuf)) != -1) {
                rbuf.flip();
                // 文字化けリスクあり
                System.out.println(StandardCharsets.UTF_8.decode(rbuf));
                rbuf.clear();
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

クライアント処理は最初に能動的ソケットを生成します。能動的ソケットはjava.nio.channels.SocketChannelオブジェクトをSocketChannel.openメソッドで生成します。SocketChannel.openメソッドの引数にjava.net.InetSocketAddressオブジェクトを渡してサーバへの接続を試みます(注4)。接続に成功するとSocketChannelオブジェクトが返ります。接続に失敗すると例外が発生します。

InetSocketAddressオブジェクトはエンドポイントを表現するオブジェクトです。オブジェクト生成時にホスト名とポート番号を渡します。ホスト名ではなくIPアドレスの文字列を渡してもかまいません。引数にホスト名を渡した場合、SocketChannelの接続時に名前解決を行います。名前解決ができない場合、UnresolvedAddressException例外が発生します。

接続後のSocketChannelオブジェクトに対してreadとwriteを使いデータ送受信処理をできます。使い方そのものはファイルの読み書きと同じです。ファイル読み書きとの違いは通信失敗の可能性の高さです。通信は失敗する可能性が高いので失敗時の対応を決めておくのが必須です。

リスト6.10で省略している処理があります。SocketChannelオブジェクトのwriteメソッドの返り値を確認していない点です。writeメソッドの返り値は送信できたバイト長です。通信の場合、送信しようとしたバイト長と送信できたバイト長が一致する保証はありません(注5)。すべてを送信できるまでwrite処理を繰り返すのが正しいコードです。ただし再試行の回数の上限設定は必須です。なお通信の性質上、送信できたことと相手に届いたことは別です。

リスト6.10は受信処理の終端を、readメソッドの返り値が-1になることで判定しています。これはサーバが通信をクローズした場合の返り値です(注6)。なお、これが受信処理の完了判定の唯一の方法ではありません。バイト長を通知してもらい、受信バイト長がそれに達するまでreadを繰り返す方法もあります。たとえばHTTPであればContent-Lengthレスポンスヘッダでバイト長を通知します。他にもバイト列の終端バイトのパターンを事前に決めておく方法もあります。このあたりの決めごとはTCP/IPより上位の通信プロトコルの仕様で決まります。本書はこれ以上の深入りをしません。

（注4） SocketChannel.openメソッドを引数なしでも呼べます。この場合、未接続状態のSocketChannelオブジェクトが返ります。未接続状態のSocketChannelオブジェクトのconnectメソッドを呼ぶと接続を試みます。connectメソッドの引数にInetSocketAddressオブジェクトを渡します。

App.6-3-3 サーバ処理

受動的な待ち受けソケットを作成し、能動的ソケットからの接続を受け入れるネットワーク処理を便宜上サーバ処理と呼びます。サーバ処理のコード例を**リスト6.11**に示します。

リスト6.11 サーバ処理

```
import java.io.IOException;
import java.net.InetSocketAddress;
import java.nio.ByteBuffer;
import java.nio.channels.*;
import java.nio.charset.StandardCharsets;

public class Server {
    public static void main(String... args) {
        int port = 8000; // 待ち受けポート番号

        // ServerSocketChannelは待ち受けソケット
        try (ServerSocketChannel serverSocketChannel = ServerSocketChannel.open()) {
            serverSocketChannel.bind(new InetSocketAddress(port));
            while (true) {
                // channelは受け入れソケット
                try (SocketChannel channel = serverSocketChannel.accept()) {
                    // 受信バイト列用のByteBuffer確保
                    // サイズが足りない場合の処理は省略
                    ByteBuffer rbuf = ByteBuffer.allocate(4096);
                    // ソケットから受信
                    int rlen = channel.read(rbuf);
```

（注5） 理論上、ファイル書き込みでもバイト長の不一致問題は発生しえます。ファイル書き込みの場合の不一致はハードウェア故障疑いもあるので自動再試行はお勧めしません。

（注6） HTTPの場合、クライアント側がConnection:closeリクエストヘッダを送ると、サーバ側がレスポンス送信後に通信をクローズする仕様です。通信全般やソケット全般の仕様ではなくHTTP固有の仕様です。

```
                    rbuf.flip();
                    // 文字化けリスクあり
                    System.out.println(StandardCharsets.UTF_8.decode(rbuf));
                    rbuf.clear();

                    // 送信バイト列用のByteBuffer確保
                    ByteBuffer wbuf = ByteBuffer.allocate(4096);
                    wbuf.put("HTTP/1.1 200 OK¥r¥n".getBytes());
                    wbuf.put("Content-Length: 5¥r¥n¥r¥n".getBytes());
                    wbuf.put("hello".getBytes());
                    wbuf.flip();
                    // ソケットへ送信（書き込みバイト数チェック省略）
                    channel.write(wbuf);
                }
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

サーバ処理は最初に受動的ソケットを生成します。java.nio.channels.ServerSocketChannelクラスを使います。ServerSocketChannelオブジェクト生成後にbindメソッドで待ち受けポート番号を指定します。bindメソッドの引数はInetSocketAddressオブジェクトです。ポート番号のみで生成したInetSocketAddressオブジェクトが、ローカルIPアドレスに対する待ち受けエンドポイントになります。

ServerSocketChannelオブジェクトのacceptメソッドを呼ぶと、待ち受けソケットは接続待ち状態になります。クライアントから接続があるまでacceptメソッドは待機します。この動作を処理がブロックすると言います。プログラムが1スレッドであれば、事実上、プログラムは何もせずに止まったままになります。クライアントからの接続を受け付けると、acceptメソッドは新たなSocketChannelオブジェクトを返します。このSocketChannelオブジェクトが受け入れソケットです。受け入れソケットに対して読み書きすると、クライアントと通信できます。

既に説明したように、受け入れソケットは、クライアント側の能動的ソケットと完全に対称な関係です。どちらからでもデータを送信可能です。どちらからでもソケットをクローズ可能です。これらの順序を決定するのは上位のプロトコルの決まりで、決まりに従うのはネットワークプログラムを作成する開発者の責任です。

受け入れソケットに対する送受信処理はクライアント側コード（リスト6.10）と区別がないことを確認してください。ただし読み書きの順序は対称的になります。リスト6.10が送信してから受信しているのに対し、リスト6.11は受信してから送信しています。

■並行処理サーバ（マルチスレッド版）

ServerSocketChannelオブジェクトのacceptメソッドは、接続を確立すると受け入れソケットを生成して返します。受け入れソケット生成後も待ち受けソケットオブジェクトは残ります。

ServerSocketChannelオブジェクトに対してacceptメソッドを再び呼ぶと、次の新しい通信相手からの接続を受け入れる待機状態になります。この仕組みにより、リスト6.11は次々に新しいクライアントからの接続を受け入れ可能です。しかし、クライアント相手の送受信処理が終わるまで新しい接続の受け入れをできません。readとwriteはそれぞれデータ送受信中に処理がブロックするからです。

多くのサーバプログラムは、複数クライアントからの接続を受け付けつつ、複数クライアントとの送受信を同時処理します。このような並行処理を行う方法の1つはマルチスレッドを使う手法です。クライアントからの接続を受け入れ後、データ送受信処理を別スレッドに任せる実装です。こうすることで、メインスレッドはすぐに次のaccept処理に戻れます。

具体例を**リスト6.12**に示します。リスト6.11とwhileループ内の処理を比較してください。acceptメソッドの返り値で受け入れソケットを取得後、ExecutorServiceオブジェクトのsubmitメソッドを使い、別スレッドに受け入れソケット処理を委譲します。submitメソッドはブロックしないので次のacceptメソッドをすぐに実行します。

リスト6.12　並行処理サーバ（マルチスレッド版）

```
import java.io.IOException;
import java.util.concurrent.ThreadFactory;
import java.net.InetSocketAddress;
import java.nio.ByteBuffer;
import java.nio.channels.*;
import java.nio.charset.StandardCharsets;
import java.util.concurrent.Executors;
import java.util.concurrent.ExecutorService;

public class Server {
    public static void main(String... args) {
        // プラットフォームスレッドと仮想スレッドのいずれかを有効化
        //ThreadFactory thFactory = Thread.ofPlatform().factory();
        ThreadFactory thFactory = Thread.ofVirtual().factory();
        ExecutorService executor = Executors.newFixedThreadPool(8, thFactory);
        int port = 8000;

        try (ServerSocketChannel serverSocketChannel = ServerSocketChannel.open()) {
            serverSocketChannel.bind(new InetSocketAddress(port));
            while (true) {
                // channelは受け入れソケット
                SocketChannel channel = serverSocketChannel.accept();
                executor.submit(() -> Server.runServer(channel));
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
```

```
    // スレッドのエントリポイント
    private static void runServer(SocketChannel channel) {
        try (channel) {
            // 受信バイト列用のByteBuffer確保
            // サイズが足りない場合の処理は省略
            ByteBuffer rbuf = ByteBuffer.allocate(4096);
            int rlen = channel.read(rbuf);
            rbuf.flip();
            // 文字化けリスクあり(本文参照)
            System.out.println(StandardCharsets.UTF_8.decode(rbuf));
            rbuf.clear();

            // 送信バイト列用のByteBuffer確保
            ByteBuffer wbuf = ByteBuffer.allocate(4096);
            wbuf.put("HTTP/1.1 200 OK¥r¥n".getBytes());
            wbuf.put("Content-Length: 5¥r¥n¥r¥n".getBytes());
            wbuf.put("hello".getBytes());
            wbuf.flip();
            channel.write(wbuf); // 書き込みバイト数のチェック省略
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

App.6-3-4　タイムアウト処理

■Selector版

ServerSocketChannelオブジェクトのaccept処理、SocketChannelオブジェクトのconnect処理、SocketChannelへの送受信処理、これらは相手側が反応しない限り永遠に待ち続けます(注7)。現実のネットワークプログラムでは相手の反応は不明です。相手の都合でプログラムが停止するのは困るので、タイムアウト処理が必要です。

タイムアウト処理をする1つの方法はjava.nio.channels.Selectorオブジェクトの使用です。Selectorオブジェクトの考え方は次のとおりです。

処理前にSocketChannelオブジェクトをSelectorオブジェクトに登録します。登録するとSelectorのselectメソッドでチャネルの状態を確認できます。チャネルの状態は下記の4種類です。

- **受動的ソケットが受け入れ可能**(accept可能)
- **能動的ソケットが接続可能**(connect可能)

(注7)　writeはブロックしないこともあります。メモリ上のバッファに書き込むだけだからです。

- **データ受信可能**(read可能)
- **データ送信可能**(write可能)

selectメソッドで状態を確認して可能な操作だけを行います。こうすればreadやwriteのメソッド呼び出しが停止しません。結果的に停止しない動作を実現できます。なおselectメソッドの実行はブロックします。しかしタイムアウトを設定できるので無限の待機を回避できます。

Selectorを使う簡単なHTTPクライアントのコードを示します(**リスト6.13**)。Selectorに登録するSocketChannelオブジェクトをノンブロッキングモードにする必要があります。channel.configureBlocking(false)が該当コードです。connect処理のタイムアウトは紙幅の都合で省略しました。

リスト6.13　ノンブロッキングソケットを使う簡易HTTPクライアント

```
import java.io.IOException;
import java.net.InetSocketAddress;
import java.nio.ByteBuffer;
import java.nio.channels.*;
import java.nio.charset.StandardCharsets;

public class Client {
    // selectメソッドのタイムアウト値(ミリ秒)
    private static final long POLL_TIMEOUT = 2000L;
    // HTTPクライアントの状態管理に使うenum定数
    private enum HttpState {
        SEND_REQUEST,  // リクエスト送信
        RECV_RESPONSE, // レスポンス受信
    };

    public static void main(String... args) {
        var host = "localhost"; // 接続先のホスト名
        int port = 8000; // 接続先のポート番号
        var address = new InetSocketAddress(host, port);

        try (Selector selector = Selector.open();
             SocketChannel channel = SocketChannel.open(address)) {
            channel.configureBlocking(false); // ノンブロッキングモード

            var httpState = HttpState.SEND_REQUEST;

            loop:
            while (true) {
                switch (httpState) {
                    case SEND_REQUEST -> {
                        channel.register(selector, SelectionKey.OP_WRITE);
                    }
```

```
                    case RECV_RESPONSE -> {
                        channel.register(selector, SelectionKey.OP_READ);
                    }
                    case null -> {
                        assert(false);
                    }
                }

                // selectメソッド呼び出しは待機（タイムアウト処理あり）
                if (selector.select(Client.POLL_TIMEOUT) > 0) {
                    for (SelectionKey key : selector.selectedKeys()) {
                        if (httpState == HttpState.SEND_REQUEST && key.isWritable()) {
                            sendRequest(channel);
                            httpState = HttpState.RECV_RESPONSE;
                        } else if (httpState == HttpState.RECV_RESPONSE && key.isReadable()) {
                            recvResponse(channel);
                            break loop;
                        } else {
                            assert(false);
                        }
                    }
                    selector.selectedKeys().clear();
                } else {
                    System.out.println("タイムアウト発生");
                    break loop;
                }
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    // データ送信処理
    private static void sendRequest(SocketChannel channel) throws IOException {
        ByteBuffer wbuf = ByteBuffer.allocate(4096);
        wbuf.put("GET / HTTP/1.1¥r¥n".getBytes());
        wbuf.put("Host: localhost¥r¥n".getBytes());
        wbuf.put("Connection: close¥r¥n¥r¥n".getBytes());
        wbuf.flip();
        channel.write(wbuf); // 書き込みバイト数チェック省略
    }

    // データ受信処理
    private static void recvResponse(SocketChannel channel) throws IOException {
        ByteBuffer rbuf = ByteBuffer.allocate(4096);
        int rlen;
        while ((rlen = channel.read(rbuf)) != -1) {
            rbuf.flip();
            // 文字化けリスクあり
            System.out.println(StandardCharsets.UTF_8.decode(rbuf));
            rbuf.clear();
        }
    }
}
```

リスト6.13を読解します。whileループ内でselectメソッド呼び出しを行い、送信処理と受信処理を呼び分けています。リクエスト送信時（HttpState.SEND_REQUEST時）はソケット書き込み可能かをチェック、レスポンス受信時（HttpState.RECV_RESPONSE時）はソケット読み込み可能かをチェックします。書き込み可能と読み込み可能はそれぞれSelectionKey.OP_WRITEとSelectionKey.OP_READの定数をchannel.registerで指示します。なお、データ送信可能は内部的な送信バッファの空きを示しているだけです。通信相手がデータ受信可能かは不明です。writeによる書き込みで相手側にデータが届く保証はありません。

■AsynchronousSocketChannel版

Selectorオブジェクトを使うタイムアウト処理は低水準の方法です。もう少し詳細を隠蔽したタイムアウト処理にAsynchronousSocketChannelを使えます。

SocketChannelの代わりにAsynchronousSocketChannelを使います。待ち受けソケットに対しては、ServerSocketChannelに対応するAsynchronousServerSocketChannelが存在します。説明をAsynchronousSocketChannelに限定します。AsynchronousSocketChannelの使い方はSocketChannelとほぼ同じです。違いはメソッド呼び出しがブロックしない点です。代わりにメソッドがFutureオブジェクトを返します(注8)。Futureオブジェクトの詳細は「**20章 スレッド**」を参照してください。

Futureを返すメソッドは、処理が裏で積まれ、処理実行前にFutureを返します。実行状態や実行結果をFutureから取得できます。Futureからの結果取得時にタイムアウトを設定可能なので結果的に通信処理にタイムアウトを組み込めます。

「**App.5-5 CompletableFutureオブジェクトを使う順序制御**」で説明したようにCompletableFutureを使うと処理の順序制御が可能です。AsynchronousSocketChannelとCompletableFutureを使うコード例を示します（**リスト6.14**）。ノンブロッキング処理を使うコードは実行処理順とコードの記載順がばらばらになりがちです。CompletableFutureを使うと2つがそろうコードを記述できます。

（注8） Futureを返す代わりに引数にコールバック処理を渡す方式も存在します。説明は省略します。

リスト6.14　AsynchronousSocketChannelとCompletableFutureを使うタイムアウト処理

```
import java.io.IOException;
import java.net.InetSocketAddress;
import java.nio.ByteBuffer;
import java.nio.channels.AsynchronousSocketChannel;
import java.util.concurrent.*;
import java.nio.charset.StandardCharsets;

public class Client {
    private static final int POLL_TIMEOUT = 2; // タイムアウト値(秒)

    public static void main(String[] args) {
        var host = "localhost"; // 接続先のホスト名
        int port = 8000; // 接続先のポート番号

        try (AsynchronousSocketChannel channel = AsynchronousSocketChannel.open()) {
            var address = new InetSocketAddress(host, port);
            CompletableFuture<Void> cfuture = CompletableFuture.runAsync(() -> {
                try {
                    channel.connect(address).get(Client.POLL_TIMEOUT, TimeUnit.SECONDS);
                } catch (InterruptedException | ExecutionException | TimeoutException e) {
                    throw new CompletionException(e);
                }
            }).thenRun(() -> {
                try {
                    // データ送信処理
                    ByteBuffer wbuf = ByteBuffer.allocate(4096);
                    wbuf.put("GET / HTTP/1.1¥r¥n".getBytes());
                    wbuf.put("Host: localhost¥r¥n".getBytes());
                    wbuf.put("Connection: close¥r¥n¥r¥n".getBytes());
                    wbuf.flip();
                    channel.write(wbuf).get(Client.POLL_TIMEOUT, TimeUnit.SECONDS);
                } catch (InterruptedException | ExecutionException | TimeoutException e) {
                    throw new CompletionException(e);
                }
            }).thenRun(() -> {
                try {
                    // データ受信処理
                    ByteBuffer rbuf = ByteBuffer.allocate(4096);
                    int rlen;
                    while ((rlen = channel.read(rbuf).get(Client.POLL_TIMEOUT, TimeUnit.SECONDS)) != -1) {
                        rbuf.flip();
                        // 文字化けリスクあり
                        System.out.println(StandardCharsets.UTF_8.decode(rbuf));
                        rbuf.clear();
                    }
                } catch (InterruptedException | ExecutionException | TimeoutException e) {
                    throw new CompletionException(e);
                }
            }).exceptionally(e -> {
                if (e.getCause() instanceof TimeoutException) {
                    System.out.println("タイムアウト");
                    return null;
                } else {
                    throw new RuntimeException(e);
                }
            });
            // 終了を待機
            cfuture.join();
        } catch (IOException | RuntimeException e) {
            e.printStackTrace();
        }
    }
}
```

App.7 国際化

「15章 文字と文字列」で説明を割愛したUnicodeのBMP範囲外の文字やUnicode以外の文字の扱いを説明します。その後、表示メッセージの多言語化と日付時刻処理を説明します。

App.7-1 文字とコードポイント

App.7-1-1 コードポイント

文字とコードポイントという2つの混乱しやすい概念があります。これらはUnicodeの歴史事情に起因します。

Javaが発表された当時、Unicodeは1文字を16ビット長で表現していました。現在のUnicodeは21ビット長の範囲の文字を扱うように進化しています。扱いやすさのため通常は32ビット長で1文字を表現します。この表現をUTF-32と呼びます。

Javaが文字表現に採用している文字コード(エンコーディング方式)はUTF-16です。UTF-16は16ビットを1単位とする文字コードです。Javaはこの単位に合わせて16ビット長のchar型を定義しています。

Unicodeが16ビット長だった時代は、UTF-16のコード値とUnicode文字とJavaの文字(char型)の3つが完全に1対1に対応していました。なお16ビット長に収まる範囲の文字集合をBMPと呼びます。

21ビット長のUnicodeに対応するため、現在のUTF-16はBMP範囲外の文字を16ビット2単位で表現します。この2単位をサロゲートペアと呼びます。このUTF-16の変更によりUnicode文字とJava文字(char型)は1対1に対応しなくなりました。char値は16ビット長のままだからです。

Javaはこの状況に対処するため、16ビット長のchar型に対応する概念を「文字」、UTF-32のUnicode文字に対応する概念を「コードポイント」と呼び分けるようにしました。Javaはコードポイント値の表現に32ビット長のint型を使います。

BMP範囲外の文字を含めるとUnicodeの文字は実質的に3つの表現を持ちます。

- **UTF-32値: コードポイント値。32ビット値1つで表現**(注1)**。int型を使用**
- **UTF-16値: Java標準の文字コード値。BMP範囲内は16ビット値1つで1文字を表現。BMP範囲外は16ビット値2つで1文字を表現。char型を使用**
- **UTF-8値: 外部(ファイルや通信)の一般的バイト列表現。文字ごとにバイト長が異なる。byte型配列を使用**

(注1) UTF-32とUTF-16サロゲートペアは、ビット長は同じ32ビットですが別のビット表現です。UTF-32値はUnicodeの21ビット長の上位ビットを単純に0埋めして32ビット長にした値です。UTF-16はこの32ビット値を単純に2つに分割した値ではなく特殊な変換をして16ビット値2つに分けます。16ビット長の既存UTF-16と混在可能にするための変換です。変換方法の詳細は他書に譲ります。

■本章で扱う文字一覧

本章の説明に使う文字を一覧にします(表7.1)。UTF-16値の表記はエスケープ文字の記法を使います(「**15章 文字と文字列**」参照)。UTF-32値の表記はUnicodeの慣習にならいU+で始まる表記を使います。UTF-8値は数値のリテラル表記を使います。どの数値も16進数表記です。

表7.1 本章で扱う文字一覧

文字	説明	UTF-16値	UTF-32値	UTF-8値
a	1バイトで表現可能な文字の例	¥u0061	U+0061	0x61
あ	2バイトで表現可能な文字の例	¥u3042	U+3042	0xe3 0x81 0x82
👍	BMP外の文字の例。漢字や絵文字など	¥ud83d ¥udc4d	U+1F44D	0xf0 0x9f 0x91 0x8d
が	「が」	¥u304c	U+304C	0xe3 0x81 0x8b
か	「か」。結合文字の濁点と合わせて「が」になる	¥u304b	U+304B	0xe3 0x81 0x8c
濁点(結合文字)	結合用文字の例	¥u3099	U+3099	0xe3 0x82 0x99
令	CJK統合漢字の「令」(注2)	¥u4ee4	U+4EE4	0xe3 0x81 0x8c
令	CJK統合漢字外の「令」	¥uf9a8	U+F9A8	0xef 0xa6 0xa8
濱	異体字セレクタと組み合わせて使える文字の例	¥u6ff1	U+6FF1	0xe6 0xbf 0xb1
異体字セレクタ	異体字セレクタ	¥udb40 ¥udd00	U+E0100	0xf3 0xa0 0x84 0x80

App.7-1-2 Stringオブジェクトとコードポイント

StringオブジェクトとStringBuilderオブジェクトはBMP範囲外の文字も扱えます。BMP外の文字はコードポイント単位で扱います。このため一部のメソッドはchar利用、一部のメソッドはint利用になっています。

たとえばStringオブジェクトのindexOfメソッドは次の定義です。引数のchはint型です。定義上はコードポイント値です。

```
int indexOf(int ch)
```

BMP範囲内の文字であればchar値を引数に渡せます。char値がint型に拡大変換されるからです。BMP範囲内の文字のコードポイント値のint値と、UTF-16のコード値のchar値は数値として同値なので動作も正常です。BMP範囲内の文字を扱う限り、char型とint型のどちらを使ってもかまいません。

(注2) CJKはChinese、Japanese、Koreanの略語です。それぞれの言語の同一字形の漢字を1つにまとめた歴史があります。

Stringオブジェクトをコードポイントで扱うメソッドとchar型前提の対比メソッドを**表7.2**に示します。

表7.2　Stringオブジェクトとコードポイント

動作	コードポイントを扱うメソッド	対比メソッド（char型前提のメソッド）
指定インデックスのコードポイント取得。引数に渡すインデックス値は下記offsetByCodePointsで取得	codePointAt	charAt
コードポイント数	codePointCount	length
コードポイント単位のストリーム生成	codePoints	chars
コードポイント単位のインデックス取得	offsetByCodePoints	なし

App.7-1-3　BMP範囲外の文字

BMP範囲外の文字のUTF-16値、コードポイント値（UTF-32値）、UTF-8値の相互変換例を示します（**表7.3**）(注3)。演算規則が決まっているので手作業でも変換可能です。

表7.3　BMP範囲外のコード値の相互変換

変換元エンコーディング方式	変換先エンコーディング方式	Java実行例
UTF-16	UTF-32	Character.toCodePoint('¥uUTF-16値', '¥uUTF-16値')
UTF-16	UTF-8	"¥uUTF-16値".getBytes()
UTF-32	UTF-16	Character.highSurrogate(UTF-32値) Character.lowSurrogate(UTF-32値)
UTF-32	UTF-8	new String(Character.toChars(UTF-32値)).getBytes()
UTF-8	UTF-16	new String(new byte[]{UTF-8バイト列}).chars().toArray()
UTF-8	UTF-32	new String(new byte[]{UTF-8バイト列}).codePoints().toArray()

BMP範囲外の文字の実例として、絵文字の👍をコードポイントで扱う例を**リスト7.1**に示します。char型で扱おうとすると1文字に2値必要になる点を確認してください。コードポイントのint値であれば1文字1数値で扱えます。

リスト7.1　BMP範囲外の文字

```
jshell> String s = new String(Character.toChars(0x1F44D))
s ==> "👍"

// 文字列長。サロゲートペアのUTF-16値なのでchar値を2つ使う
jshell> int chLen = s.length()
chLen ==> 2

// コードポイント長
jshell> int cpLen = s.codePointCount(0, s.length())
cpLen ==> 1

// ストリーム処理を使うコードポイント値の列挙
jshell> s.codePoints().mapToObj(Integer::toHexString).forEach(System.out::println)
1f44d

// コードポイント値の列挙（ループ版）
jshell> for (int i = 0; i < s.length(); i += s.offsetByCodePoints(i, 1)) {
   ...>     System.out.println(Integer.toHexString(s.codePointAt(i)));
   ...> }
1f44d
```

リスト7.1の文字をUTF-16値およびUTF-8のバイト列と相互変換する例を**リスト7.2**に示します。

リスト7.2　BMP範囲外の文字のエンコーディング変換

```
jshell> String s = new String(Character.toChars(0x1F44D))
s ==> "👍"

// StringオブジェクトからUTF-16コード値のchar配列に変換
// 1文字あたり2つのchar値になる（サロゲートペア）
jshell> s.chars().forEach(c -> System.out.printf("%04x¥n", c))
d83d
dc4d

// UTF-16のchar配列からStringオブジェクト生成
jshell> String s = new String(new char[] { '¥ud83d', '¥udc4d' })
s2 ==> "👍"

// StringオブジェクトからUTF-8のバイト列に変換
// UTF-16サロゲートペアをUTF-8バイト列にすると1文字あたり4バイト
jshell> byte[] bytes = s.getBytes()
jshell> int byteLen = bytes.length
byteLen ==> 4

// UTF-8バイト列の16進数表現
jshell> IntStream.range(0, bytes.length).
   ...> map(i -> Byte.toUnsignedInt(bytes[i])).
   ...> forEach(c -> System.out.printf("%02x¥n", c))
f0
9f
91
8d

// UTF-8バイト列からStringオブジェクト生成
jshell> String s = new String(new byte[]{ (byte)0xf0, (byte)0x9f, (byte)0x91, (byte)0x8d })
s3 ==> "👍"
```

（注3）　BMP範囲内の場合、UTF-16値とUTF-32値は同値です。UTF-8バイト列との相互変換方法は「15章　文字と文字列」を参照してください。

App.7-1-4　Unicodeのその他の話題

■結合文字

Unicodeに結合文字という仕組みがあります。結合文字は直前の文字と合体して結合文字列になります。

結合文字の一例が日本語の濁点です(注4)。結合文字として働く濁点のコードポイント(U+3099)が存在します。この濁点の前にたとえば「か」の文字があると2つを結合して「が」を意味します。なお結合文字ではない濁点(U+309B)も存在します。通常の文中で「日本語の濁点は”と書きます」に使う濁点は結合文字ではない濁点を使います。

普通の「が」と結合用濁点と合わせた「が」を示します(**リスト7.3**)。コードポイントが異なるのでequalsメソッドの比較結果は偽です。次節の正規化もしくは後述するコレーション比較で同一判定可能になります。

リスト7.3　普通の「が」と結合文字列の「が」

```
jshell> String ga = "が"
jshell> String ga2 = "か¥u3099"
// equalsメソッドの比較結果は偽
jshell> boolean result = ga.equals(ga2)
result ==> false
```

■正規化

Unicodeに正規化という概念があります。説明が長くなるので本書では正規化という用語がある点とjava.text.Normalizerクラスを使う点の言及に留めます。

正規化の一例として「令」を扱うソースコードを示します(**リスト7.4**)。歴史的事情により「令」には異なるコードポイント値を持つ2つのUnicode文字が存在します。正規化によりこの2つを同一に扱えます。前節で言及した「か+結合用濁点」と「が」も、正規化すると同一に扱えます(**リスト7.5**)。

リスト7.4　異なるコードポイント値の「令」の正規化

```
jshell> char rei1 = '¥u4ee4'
rei1 ==> '令'
jshell> char rei2  = '¥uf9a8'
rei2 ==> '令'
// 正規化
jshell> String s = Integer.toHexString(Normalizer.normalize("¥uf9a8", Normalizer.Form.NFC).charAt(0))
s ==> "4ee4"
```

(注4)　説明が煩雑になるので日本語の濁点に話を限定します。日本語以外を考慮すると多数の結合文字が存在します。それらの説明は専門書に譲ります。

リスト7.5　結合文字列の「が」の正規化

```
jshell> String ga = Integer.toHexString(Normalizer.normalize("¥u304b¥u3099", Normalizer.Form.
NFC).charAt(0))
ga ==> "304c"  // 普通の「が」のコードポイント
```

■異体字セレクタ

Unicodeに異体字セレクタという仕組みがあります(注5)。異体字セレクタという特殊なコードポイント値が複数定義されています。ある文字の後ろに異体字セレクタを続けると2つを合わせてその文字の異体字を意味します。たとえば「濱」(U+6FF1)の後続に異体字セレクタ(U+E0100)を続けると、2つのコードポイントで「濱」の異体字を意味します。

意味論はともかく2つのコードポイントで1文字になる点に着目してください。具体例は次の書記素の説明で使います。

■書記素クラスタ

結合文字や異体字セレクタの存在のため、コードポイントも1値ずつで扱えなくなりました。表記上1つに見える単位を扱うために書記素クラスタ(grapheme cluster)という概念を使います。本書執筆時点のUnicode規格において、多くの人が文字と感じるものと対応するのが書記素クラスタです。

java.text.BreakIteratorオブジェクトで書記素クラスタを扱えます。具体例を**リスト7.6**に示します。異体字セレクタを使うコードポイント2値を1つの単位として扱えます。Unicodeの理想に従うと、Javaの文字処理プログラムは、char型やint型を使わず常に書記素クラスタ単位で処理すべき考えになります。本書執筆時点ではやや過剰設計なので開発チームで方針を決めてください。

リスト7.6　書記素クラスタ

```
// 異体字セレクタ対応文字
jshell> String hama = new String(new char[] { '¥u6ff1' })
hama ==> "濱"

// 異体字セレクタ対応文字と異体字セレクタ
jshell> String hama2 = new String(new int[] { '¥u6ff1', 0xe0100 }, 0, 2)
hama2 ==> "濱"

// コードポイント長は2
jshell> int numCodePoints = hama2.codePointCount(0, hama2.length())
numCodePoints ==> 2
```

(注5)　説文字の見た目を字形と呼びます。実用上、字形はフォントのグリフ(見た目)です。字形の違いだけで説明しきれない(と考える人がいる)見た目の違いを異体字と呼びます。とは言え異体字の定義はあまり自明ではありません。

```
// BreakIteratorオブジェクトの利用
jshell> import java.text.BreakIterator
jshell> BreakIterator bi = BreakIterator.getCharacterInstance()
jshell> bi.setText(hama2)

// 書記素長は1（下記forループは書記素ごとにまわる処理）
jshell> for (int start = bi.first(), end = bi.next(); end != BreakIterator.DONE; start = end,
end = bi.next()) {
   ...>     System.out.println("%d, %d, %s".formatted(start, end, hama2.substring(start, end)));
   ...> }
0, 3, 濱
```

■コレーション（照合）

国際化関連の用語の1つに「コレーション（collation）」があります。日本語で「照合」と訳します。本書は照合の用語を使います。照合とは文字の同定（同じ文字の判定）や順序（ソート規則）のための規則です。

文字のコード値やコードポイントだけでは照合に不十分な場合があります。わかりやすい例は英語アルファベットの大文字と小文字の関係です。'a' と 'A' は異なる文字コード値を持ちます。異なる文字なのでこれ自体は自明です。

文字コードだけでは、大文字と小文字を同じ文字として扱ったり、大文字小文字を区別しないソートをできません。これらのためには、コード値と別に大文字小文字関係の定義が必要です。このようにコード値と別に持つ定義を照合規則（collation algorithm）と呼びます。

現実的には、照合が大きく意味を持つのはヨーロッパ圏の言語です。ヨーロッパ各国のアルファベットには、英語の大文字小文字の関係のような規則がいろいろとあるからです。

日本語で類似するものは、ひらがなとカタカナを区別せずにソートしたい場合などです。また濁点や半濁点にも似た関係があります。たとえば、「は」「ば」「ぱ」を同じソート順にしたい場合などです。具体例を**リスト7.7**に示します。

リスト7.7　「は」「ば」「ぱ」を同列扱いにするソート

```
jshell> import java.text.Collator
jshell> Collator jaColl = Collator.getInstance(Locale.JAPAN)

jshell> Stream.of("はん", "ぱい", "ハイ").sorted().forEach(System.out::println)
はん
ぱい
ハイ

jshell> Stream.of("はん", "ぱい", "ハイ").sorted(jaColl).forEach(System.out::println)
ハイ
ぱい
はん
```


照合で結合文字の同一判定も可能です（**リスト7.8**）。

リスト7.8　照合を使う結合文字の同一判定

```
jshell> String ga = "が"
jshell> String ga2 = "¥u304b¥u3099"

jshell> import java.text.Collator
jshell> Collator jaColl = Collator.getInstance(Locale.JAPAN)
jshell> boolean result = jaColl.equals(ga, ga2)
result ==> true
```

■漢字の照合規則

漢字にも照合規則があります（**リスト7.9**）。しかし万人が納得できる漢字の並び順は世の中に存在しません。実開発での利用は推奨できません。実開発で氏名の並べ替え処理が必要な場合、読み仮名を入力してもらい読み仮名で並び替えるのが通例です。

リスト7.9　漢字のソート

```
// 文字コードを使うソート
jshell> Stream.of("一", "二", "壱", "弐").sorted().forEach(System.out::println)
一
二
壱
弐

// コレーションを使うソート（リスト7.8のjaCollを利用）
jshell> Stream.of("一", "二", "壱", "弐").sorted(jaColl).forEach(System.out::println)
一
壱
二
弐
```

App.7-1-5　Unicode以外の文字コード

Unicode以外の文字コードをJavaで扱うには下記の2つの方法があります。

- **対象文字コードでエンコーディングしたバイト列をUTF-16文字列と相互変換。Javaコードではstringとして扱う**
- **対象文字コードでエンコーディングしたバイト列をバイト列のまま扱う。Javaコードではbyte型配列で扱う**

後者は少し特殊な領域になるので前者を説明します。なお入力バイト列をそのまま出力するようなプログラムであれば、バイト列のまま扱うほうが合理的です。

ある文字コードのバイト列をUTF-16文字列と相互変換するには、java.nio.charset.Charsetオブジェクトを使います。

Charsetクラスは抽象基底クラスです。Charsetオブジェクトを直接new式で生成できません。Charset.forNameメソッドでCharsetオブジェクトを生成します(注6)。

Stringオブジェクトの文字列をShift-JISのバイト列に変換する例を**リスト7.10**に示します。StringオブジェクトのgetBytesメソッドの引数にCharsetオブジェクトを渡します。Charsetオブジェクトの代わりに"Shift-JIS"文字列を渡しても動作します。

リスト7.10　Stringオブジェクトの文字列からShift-JISのバイト列に変換

```
jshell> import java.nio.charset.Charset
jshell> Charset sjisCharset = Charset.forName("Shift-JIS")

// 下記の結果のバイト列を16進数の正値にすると 0x82 0xa0 0x82 0xa2 です
jshell> byte[] bytes = "あい".getBytes(sjisCharset)
bytes ==> byte[4] { -126, -96, -126, -94 }

// 下記でも動きます
jshell> byte[] bytes = "あい".getBytes("Shift-JIS")
bytes ==> byte[4] { -126, -96, -126, -94 }
```

Shift-JISのバイト列からStringオブジェクトを生成する例を**リスト7.11**に示します。Stringのnew式の第2引数にCharsetオブジェクトを渡します。リスト7.10同様、引数に"Shift-JIS"文字列を渡しても動作します。

リスト7.11　Shift-JISのバイト列からStringオブジェクト生成

```
jshell> byte[] bytes = new byte[]{ (byte)0x82, (byte)0xa0, (byte)0x82, (byte)0xa2 }

jshell> String s = new String(bytes, Charset.forName("Shift-JIS"))
s ==> "あい"
jshell> String s = new String(bytes, "Shift-JIS")
s ==> "あい"
```

App.7-2　表示文字列の多言語化

App.7-2-1　リソースバンドル

プログラムの表示文字列を多言語化するために、リソースバンドルと呼ぶ仕組みがあります。

(注6)　一部、StandardCharsetsクラス内に事前定義されたCharsetオブジェクトがあります。

仕組みは次のとおりです。まず出力文字列ごとに一意なキーを振ります。一意であればよいのでキーの決め方は自由です。通常、キーはなんらかの文字列です。区別のため、表示に使う文字列を表示文字列と記載します。

ソースコードには表示文字列を直接記述せず、代わりにキーを記述します。実行時はキーから表示文字列を検索します。対応表をソースコードの外部に配置すると、ソースコードを変更することなく複数言語の文字列出力を切り替え可能になります。

対応表の持ち方には自由度があります。もっとも簡易な持ち方はテキストファイルです。このテキストファイルをプロパティファイルと呼びます。プロパティファイルの具体例は後ほど説明します。

■ロケール

キーと表示文字列の対応表をロケール(locale)ごとに用意します。ロケールは言語と地域を決める仕組みと理解してください。java.util.Localeオブジェクトで扱います。Javaの標準ライブラリの多くの機能が、ロケールの切り替えで言語や地域に適した内部動作に切り替わります。本章は説明の簡略化のため標準で準備済みのLocale.JAPANESEオブジェクトを使います。

サーバなどで複数ユーザを扱う場合、ユーザに使用言語や地域を設定してもらい内部的にユーザに対応するLocaleオブジェクトおよび後述するResourceBundleオブジェクトを生成します。単体アプリであれば環境変数によるロケール指定も実用的です。

■リソースバンドルの利用

リソースバンドルを使うソースコードは次のようになります。

- **ロケールに応じたResourceBundleオブジェクトを生成**
- **ResourceBundleオブジェクトのgetStringメソッドにキーを渡して、ロケールに応じた表示文字列を取得**

ResourceBundleクラスは抽象基底クラスです。new式でResourceBundleオブジェクトは生成できません。生成手段はいくつかありますが、もっとも簡易な手段はResourceBundle.getBundleメソッドの利用です。getBundleメソッドの第1引数にリソースバンドルのベース名を指定します。第2引数にLocaleオブジェクトを指定します。第2引数を省略するとデフォルトロケールを使います。リソースバンドルを使う例を示します(**リスト7.12**)。

リスト7.12　リソースバンドルの利用

```
import java.util.Locale;
import java.util.ResourceBundle;

public class Main {
    public static void main(String... args) {
        ResourceBundle resourceBundle = ResourceBundle.getBundle("appname");
```

```
        System.out.println(resourceBundle.getString("Foo.One"));
    }
}
```

生成したResourceBundleオブジェクトのgetStringメソッドで、キーから表示文字列を取得します。リスト7.12の場合、"Foo.One"がキーです。キーを識別しやすくするため、Foo.Oneのように.(ドット)で名前を区切って階層管理する慣習があります。慣習なのでドット文字は必須ではありません。getStringメソッドはプロパティファイルからキーを検索して表示文字列を返します。キーが見つからない場合、MissingResourceException実行時例外が発生します。

■プロパティファイル

プロパティファイルのファイルパスはリソースバンドルのベース名とロケール名から決まります。たとえば、ベース名appname、環境変数LANG=ja_JPで実行すると次の順序でファイル名を検索します。

プロパティファイルの検索順序

① appname_ja_JP.properties
② appname_ja.properties
③ appname.properties

プロパティファイルの中にkey=valueのようにキーと翻訳文字列を=で結んだ行を書き連ねます。リスト7.12に対応する例は下記のようになります。

```
Foo.One=いち
```

この中身のプロパティファイルを作ってリスト7.12を実行すると「いち」を出力します。

App.7-2-2　書式の指定

ロケールに応じて表示文字列の書式を変更できます。代表的な書式指定の対象は数値や日付です。日付の書式指定は後述します。

数値の書式指定にjava.text.NumberFormatオブジェクトを使えます(**リスト7.13**)。NumberFormatクラスは抽象基底クラスです。

NumberFormatオブジェクトの生成手段はいくつかあります。new式では生成できません。リスト7.13はNumberFormat.getInstanceメソッドでオブジェクトを取得しています。

NumberFormatオブジェクトのformatメソッドに数値を渡すと3桁ごとにカンマで区切った表記の文字列が返ります。

リスト7.13　数値の書式指定(NumberFormatオブジェクト)

```
jshell> import java.text.NumberFormat
jshell>  NumberFormat numFmt = NumberFormat.getInstance(Locale.JAPAN)
jshell> String s = numFmt.format(1234567)
s ==> "1,234,567"
```

String.formatメソッドでも数値の書式を指定可能です(**リスト7.14**)。書式指定をString.formatで一貫する方針は1つの考え方です。

リスト7.14　数値の書式指定(String.formatメソッド)

```
jshell> String s = String.format("%,d", 1234567)
s ==> "1,234,567"
```

App.7-3　日付と時刻処理

App.7-3-1　Date & Time API

日付時刻処理にjava.timeパッケージを使えます。通称Date & Time APIと呼びます。旧来からあったjava.util.Dateとjava.util.Calendarを置き換えるAPIです。

代表的なクラスを**表7.4**に示します。Date & Time APIのクラスはすべて不変クラスです。

表7.4　Date & Time APIの代表的なクラス

クラス	説明
Instant	時点。Unixエポック起点の時刻の秒数
Duration	期間。日や月の概念がなく、ある時点からある時点の間の秒数
Period	期間。日や月の概念がある期間
ZoneId	タイムゾーン(名前ベース)
ZoneOffset	タイムゾーン(値ベース)
LocalDate	タイムゾーンに依存しない日。年号取得やうるう年判定など日処理に使える
LocalTime	タイムゾーンに依存しない時刻。ただの時刻処理に使える
LocalDateTime	タイムゾーンなしの日時。ただの日時処理に使える。記録日時に使うのは国際化の観点で危険
ZonedDateTime	タイムゾーンありの日時。記録日時に使える
OffsetTime	簡易版タイムゾーンありの時刻。外部処理用の記録日時に使う(SQLやログなど)
OffsetDateTime	簡易版タイムゾーンありの日時。外部処理用の記録日時に使う(SQLやログなど)
TemporalAdjusters	日付演算(月末、直近の日曜日など)ユーティリティクラス
JapaneseChronology	和暦のためのクラス

目的別にどのオブジェクトを使うとよいかを説明します。

■発生時刻の記録

何かの発生時刻の記録のために現在時刻を必要とする場合があります。通常、Unixエポック

時からの経過時間を記録します。タイムゾーンに依存しないのでUTC時間と呼ぶ場合もあります(注7)。以後、秒単位にこだわらないUTC時間の用語を使います。

秒は粒度が荒いので、多くの場合、UTC時間をミリ秒やマイクロ秒で記録します。なおナノ秒まで粒度を細かくしても値が同一になる可能性をゼロにできないので注意してください。値のユニーク性の保証が欲しい場合は時刻以外の仕組みが必要です。値のユニーク性は本章の論点から外れるので説明は割愛します。

現在時刻のUTC時間はInstant.nowメソッドで取得可能です。Systemクラスのクラスメソッドでも数値として取得できます(「**13章 Javaプログラムの実行と制御構造**」参照)。

発生時刻をデータベースやログなど外部に永続化したい場合があります。データベースであれば数値もしくはデータベース標準の時刻型で記録するのが通例です。ログであれば人間が読む可能性を考え、文字列での記録も普通です。人間の読みやすさの優先順位が低ければ数値のまま記録します。発生地のタイムゾーンを記録するかはソフトウェアの要求次第です。

■カレンダー日時の記録

ユーザが入力した日時情報を便宜上カレンダー日時と呼びます。通常、システム設定またはユーザの個人設定でタイムゾーンを決めます。タイムゾーン込みの日時を扱うにはZonedDateTimeオブジェクトを使います(**リスト7.15**)。

リスト7.15　ZonedDateTimeオブジェクトの使用

```
jshell> import java.time.*   // 後続のJShellコードでも使用

// ユーザがJST(日本標準時)で使っている想定
jshell> var jst = ZoneId.of("JST", ZoneId.SHORT_IDS)
jst ==> Asia/Tokyo

// 引数の2024や1や31をユーザが指定した前提
// 記録時はタイムゾーン込みの日時の記録を推奨
jshell> var dt = ZonedDateTime.of(2024, 1, 31, 1/*hour*/, 10/*min*/, 15/*sec*/, 0/*nanosec*/, jst)
dt ==> 2024-01-31T01:10:15+09:00[Asia/Tokyo]
```

ZonedDateTimeオブジェクトとInstantオブジェクトを相互に変換可能です(**リスト7.16**)。発生時刻の観点では同一情報だからです。しかしInstantに変換するとタイムゾーン情報を失います。別途タイムゾーン情報を与えない限り、InstantからZonedDateTimeに戻せません。

リスト7.16　ZonedDateTimeとInstantの相互変換(リスト7.15の継続)

```
// ZonedDateTimeからInstantへ変換
jshell> Instant instant = dt.toInstant()
instant ==> 2024-01-30T16:10:15Z

// InstantからZonedDateTimeへ変換
jshell> ZonedDateTime dt2 = instant.atZone(jst)
dt2 ==> 2024-01-31T01:10:15+09:00[Asia/Tokyo]

// InstantからZonedDateTimeへ変換(別解)
jshell> ZonedDateTime dt3 = ZonedDateTime.ofInstant(instant, jst)
dt3 ==> 2024-01-31T01:10:15+09:00[Asia/Tokyo]
```

カレンダー日時を外部に記録する場合、下記2つの方法があります。多くの場合はUTC時間の記録で十分です(注8)。

- **UTC時間のみ記録。人間のために表示する時、表示設定のタイムゾーンで記録日時をZonedDateTimeに変換**
- **タイムゾーンとUTC時間を記録**

誕生日など日付を記録する場合に注意点があります。日時情報ではなく日付情報として記録してください。日付と日時を等価に扱おうとして、0時から23時59分の期間で日付を記録するのは間違いです。別のタイムゾーンでカレンダーを見た時に誕生日の日付が2日間にまたがってしまいます。

■LocalDateTime

LocalDateTimeオブジェクトはタイムゾーンのない日時情報です。タイムゾーンを指定するとLocalDateTimeオブジェクトからZonedDateTimeオブジェクトに変換できます。

プログラム実行中はLocalDateTimeで日時を扱い、記録時にZonedDateTimeに変換するのは1つの考え方です。LocalDateTimeとZonedDateTimeの混在はバグのもとだからです。通常、プログラム実行中のカレンダー日時はユーザの入力値だけで扱うほうが簡単です(注9)。

■経過時間の演算

ある日時から何時間後または何時間前などの日時を求める処理を、便宜上経過時間の演算と呼びます。経過時間の単位が秒、分、時間、日のどれかである限りこの演算は簡単です。ただの数値の加減算だからです。しかし自前で加減算コードを書かずにDurationオブジェクトの利用を推奨します。具体例を**リスト7.17**に示します。

(注7)　Unixエポック時は1970年1月1日午前0時です。Unixエポック時からの経過秒数をUnix時間と呼びます。Unix時間とUTC時間は実用上は同値と考えて問題ありません。

(注8)　将来タイムゾーンが変わった場合でも指定日時の意味を保持したい場合、タイムゾーンも記録します。この考えの背景は書籍「ソフトウェア設計のトレードオフと誤り」を参照してください。

(注9)　プログラム実行中に利用者のタイムゾーン変更に追随したいならタイムゾーンを常に使う必要があります。この場合、LocalDateTimeは実質的に使えません。

リスト7.17　経過時間の演算（Durationオブジェクト使用）

```
jshell> import java.time.*
jshell> var localDt = LocalDateTime.parse("2024-01-01T00:00:00")
// 上記日時の1日後かつ2時間後かつ3分後
jshell> var duration = Duration.ofDays(1).plusHours(2).plusMinutes(3)
jshell> var localDt2 = localDt.plus(duration)
localDt2 ==> 2024-01-02T02:03
```

■カレンダー日時の演算

ある日時から何ヶ月後または何ヶ月前などの日時を求める処理を、便宜上カレンダー日時の演算と呼びます。1月31日の1ヶ月後を想像するとわかるように、これは単純な加減算ではありません。カレンダー日時の演算にはPeriodオブジェクト、複雑な演算にはTemporalAdjustersユーティリティクラスを使ってください。具体例を**リスト7.18**に示します。

リスト7.18　カレンダー日時の演算（Periodオブジェクト使用）

```
jshell> var localDt = LocalDateTime.parse("2024-01-31T00:00:00")
jshell> var period = Period.ofMonths(1)

// 2024年1月31日の1ヶ月後は2月29日（Periodの定義）
jshell> var localDt2 = localDt.plus(period)
localDt2 ==> 2024-02-29T00:00

// 複雑な計算はTemporalAdjustersを使う
jshell> var localDt3 = localDt.with(TemporalAdjusters.next(DayOfWeek.SATURDAY))
localDt3 ==> 2024-02-03T00:00
```

■和暦の元号

和暦の元号を使うにはjava.time.chrono.JapaneseDateオブジェクトを使います（**リスト7.19**）。

リスト7.19　和暦の元号

```
jshell> import java.time.chrono.JapaneseDate
jshell> import java.time.format.TextStyle
jshell> import java.time.temporal.ChronoField

jshell> var jpDt = JapaneseDate.of(2024, 1, 1)
jshell> String reiwa = jpDt.getEra().getDisplayName(TextStyle.FULL, Locale.JAPANESE)
reiwa ==> "令和"

// 2024年は令和6年
jshell> int reiwaYear = jpDt.get(ChronoField.YEAR_OF_ERA)
reiwaYear ==> 6

jshell> import java.time.chrono.JapaneseEra

jshell> var reiwaDt = JapaneseDate.of(java.time.chrono.JapaneseEra.REIWA, 6, 1, 1)

// 令和6年は2024年
jshell> int year = reiwaDt.get(ChronoField.YEAR)
year ==> 2024
```

■日付の書式指定

日付の書式指定にはjava.time.format.DateTimeFormatterオブジェクトを使えます(注10)。DateTimeFormatterオブジェクトは原則としてファクトリメソッドで生成します。生成方法はいくつかあります。**リスト7.20**はDateTimeFormatter.ofPatternメソッドを使っています。

リスト7.20　日付書式指定（DateTimeFormatterオブジェクト使用）

```
jshell> import java.time.format.DateTimeFormatter
jshell> var dtFmt = DateTimeFormatter.ofPattern("yyyy/MM/dd HH:mm:ss")
jshell> var localDt = LocalDateTime.parse("2024-01-01T00:00:00")
jshell> String s = dtFmt.format(localDt)
s ==> "2024/01/01 00:00:00"
```

DateTimeFormatterオブジェクトとLocaleオブジェクトの組み合わせも可能です。**リスト7.21**は日本語の日付表示文字列を作る例です。

リスト7.21　日本語の日付書式指定

```
jshell> String s = DateTimeFormatter.ofLocalizedDateTime(FormatStyle.LONG).
   ...> withZone(ZoneId.of("JST", ZoneId.SHORT_IDS)).withLocale(Locale.JAPANESE).
   ...> format(localDt)
s ==> "2024年1月1日 0:00:00 JST"
```

（注10）String.formatメソッドでも日付の書式処理を可能です。DateTimeFormatterオブジェクトのほうが可読性に優れます。

App.8 自動テスト

自動テストの設計や実装方法を学ぶことで、実装したプログラムの正しさを検証しつつ、品質を高めることができます。テストは品質観点だけでなく、プログラムの実態の理解を深めたりなど副次的な効果も期待できます。積極的に習得してください。

App.8-1 自動テストとは

開発において、テストは当たり前に実施する工程の1つです。テストを実施することで、作成したプログラムコードが設計どおりかを検証できます。テストによりバグを検出できるため、プログラムをより堅牢にできます。

従来は、テスト工程のほとんどを手動で実施していました。近年、時代の進化とともにテストの多くを自動化できます。また、自動化できるテスト範囲が広がっただけでなく、あらゆる技術でテストの簡素化などを実現しています。自動化されたテスト工程や、その実施作業を自動テストと呼びます。Javaでは、これらのテストをプログラムで実装し、自動テストを実現します。

App.8-1-1　テストの自動化

自動テストの実現には、テスト内容をコンピュータに実施させるプログラムが必要です。
このテスト用のプログラムをテストコードと呼びます。
テストコードの質が悪いと、自動テストの利点を得にくい状態となります。
適切なテストの設計やテストコードの実装を習得し、自動テストの利点を活かしてください。

App.8-1-2　テストの自動化による価値

自動テストは、さまざまな価値を持っています。主に下記の価値が挙げられます。

- **効率性**: **手動テストよりも高速にテストを実施できる**
- **再現性**: **常に同じテストを実施できる**
- **カバレッジの可視性**: **テストの網羅度合いを可視化できる**

効率性が優れていると、テストに要する時間やコストを小さくできます。テスト実施が容易になると、プログラム実装とテストを並行して進められます。並行して進めることで、プログラムのバグを早期に発見し、修正できます。

再現性が優れていると、新機能の追加や既存機能の修正時、既存機能に影響を与えていないかを確認できます。手動テストでは、担当者による微細な違いが生じる可能性があります。
主な例としては、テスト環境構築の誤り、テスト手順の誤り、パラメータの入力誤りなどが挙げられます。自動テストは、これら環境を常に同じ状態にしやすいため、人為的ミスの解消に役立ちます。

カバレッジの可視性が優れていると、テストすべきプログラムがどの程度検証されているか確認できます。カバレッジ情報は、さまざまなツールにより可視化できます。
可視化により、テストされていないプログラム箇所を容易に発見でき、テスト漏れを未然に防げます。

App.8-2 テストと設計

自動テストの実現において、テストの本質の理解は重要です。

テストはシステムのプログラムの正しさを検証する工程です。ここでの正しさとは、プログラムの実装が基となる設計に従っているかです。テストと設計は、密接な関係であると言えます。

設計が悪いとテストの価値を活かせません。これは、プログラムが悪い設計に従っているかを検証するためです。テスト結果が正常と判定されても、「実装したプログラムは、悪い設計どおりです。」という意思表示でしかないためです。設計が悪い状態では、バグが多く存在する可能性があります。良い設計を意識し、テストの価値を最大化することが重要です。

C O L U M N

カバレッジの誤解

カバレッジの世界では、カバレッジ率やテスト網羅率と呼ばれる指標があります。カバレッジ率とは、プログラムに対し、どの程度テストが実施されたか示します。

「カバレッジ率が高いほど、プログラム品質も高い。」と勘違いする人を見かけます。カバレッジ率は、プログラム品質と関わりがありますが、直接的な関係ではありません。

テストが不十分かどうかの判定に役立ててください。つまり、「カバレッジ率が低いとテストが不十分である。」ということです。「カバレッジ率が高いとテストが十分である。」とはなりません。カバレッジ率が低い場合は、テストケース(※1)の追加を検討してください。

カバレッジ率は、プログラムやテストコードの実装次第で意図的に操作できます。
これらの点に注意し、カバレッジ率を扱ってください。

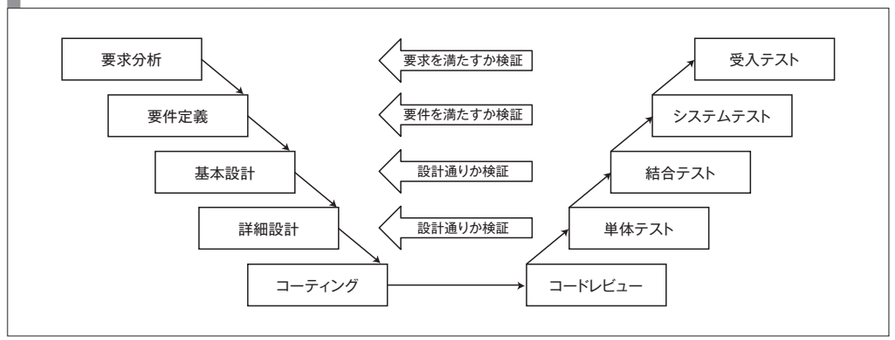
※1　テスト関連用語は、「頻出するテスト用語」を参照してください。

App.8-2-1　自動テストの見極め

すべてのテストを自動テストとして扱えば良い訳ではありません。テストにも自動テストにすべきものとそうでないものが存在します。

テスト工程を細分化していくと、目的別にテストを分解できます。ウォータフォールと呼ばれる開発手法では、テスト工程を単体テスト、結合テスト、総合テストと分解することが多いです。同様に設計工程も基本設計、詳細設計に分解でき、基本設計の前段には要件定義があります。テストと設計の関係性を示します（**図8.1**）。

図8.1　ウォータフォールの開発工程例



ここで理解して欲しいことは、ウォータフォールのような開発手法の良し悪しでなく、設計の考え方に基づいたテスト工程の分類です。自動テストの実態は、テストコードと呼ばれるプログラムです。実装に近いテスト工程ほど実装しやすく、自動テストとして扱いやすくなります。

すべてのテスト工程を自動テストとして扱うことは困難です。どのテスト工程やテスト項目を自動テストにすべきかの判断能力も必要です。

■テストコードの対象例

工程以外にも下記の機能に対し、テストコードを実装すると価値を創出しやすいです。開発チームやプロジェクトで異なるため、一例として押さえてください。

- **ビジネスロジックのコア機能や、その周辺機能**
- **過去の修正で意図しない動作をし、挙動を変えたくない機能**
- **文書などの情報が不十分な機能**

コア機能やその周辺機能は、最優先で品質を気にすべきです。そのため、テスト回数も他より多く実施する傾向があります。回数が増えるほど、効率性や再現性で高い価値を発揮します。

過去、コードの改修で意図しない影響を与えてしまった機能もテストコードを書く有力な候補です。テストの再現性を高めることで、再発防止策となります。

文書などの知見情報が少ないコードに対して、テストコードが有用な代替となる場合があります。テストコードから機能に対する具体的な入出力を読み取れるからです。その結果、知見が少なくても、機能の想定する動作を理解できます。

自動テストにすることで得られる価値や支払う対価を見極め、自動テストを上手に使いこなしてください。本章を通して、得られる価値やそのためのコストを理解できるようになるはずです。

App.8-2-2　単体テスト

自動テストとして特に扱いやすいテスト工程は、単体テストです(注1)。

単体テストの特徴として、下記が挙げられます。これらの特徴は、自動テストと相性が良いです。

- **単位ごとの実装を検証できる**
- **テスト時間を短くできる**
- **複数のテストを順番を問わず、並列的に処理できる**

ここで示す単位とはクラスだと解釈してください。より具体的には、クラスフィールドやクラスメソッド、インスタンスフィールド、インスタンスメソッドを検証します。

後述するテストコードから特徴や価値を確かめていきましょう。

App.8-2-3　頻出するテスト用語

Javaのテストで頻出となるテスト用語の意味を**表8.1**にまとめます。

表8.1　テスト用語

テスト用語	意味
テストケース	特定の条件下でシステムが正しく動作する手順またはシナリオか検証する
テストスイート	関連するテストケースの集合
テストランナー	テストケースを実行するツールやフレームワーク、エンジン
テストメソッド	テストケースが実装されたメソッド
テストクラス	テストスイートが実装されたクラス

App.8-3　テストコードの概要

テストコードは、本番環境でのソフトウェアの動作を模倣するように設計します。この設計により、本番環境でのソフトウェアのバグやエラーの早期発見や修正ができ、本番環境での障害発生を抑制できます。

（注1）　開発次第では、結合テストや総合テストも自動テストとして扱えます。

しかし、テストコードでソフトウェアのすべてのバグやエラーに対処することは困難です。ソフトウェアが作って終わりではないように、ソフトウェアの更新に合わせてテストコードも更新すべきです。ソフトウェアの更新に伴い、テストケースの追加、削除、変更が発生します。状況に合わせてテストケースを実装し、ソフトウェアの品質を高めます。

テストコードを実装できるようになると、初めはあらゆるテストの自動化を試みたいと思うのが、技術者の性です。手動テストが自動テストに変わる様は、効果がわかりやすく、実装自体も楽しめます。

しかし、自動テストにできるテストをすべて、自動化することは良い判断とは言えません。テストコードもソフトウェアの実装と同様、実態はプログラムです。ソフトウェア同様、プログラムの増大に伴い、複雑性が増加し、保守性や効率性が悪化します。「**App.8-2-1 自動テストの見極め**」の観点を意識しながら、テストコードの実装方法を習得してください。

App.8-3-1　テスト向けフレームワーク

Javaには、テストコードを実装するのに強力なフレームワークが複数あります。広く使われるフレームワークは、JUnit(注2)です。JUnitはフレームワークの機能を担うライブラリを提供しています。

さらに、JUnitを使いやすくするAssertJライブラリがあります。JUnitとAssertJを組み合わせて使うことで、テストコードの可読性を向上できます。

本書で採用する各ライブラリのバージョンを示します(**表8.2**)。

表8.2　本書で採用するライブラリのバージョン

JUnit	5.10.2
AssertJ	3.24.2

テストコードの実行は、コマンドラインを例に説明します。多くの場合はビルドツールを使います(注3)。

App.8-3-2　テストフレームワークの特徴

JUnitは、単体テストで役立つ機能を提供しています。

- **タグ付けとフィルタリング：テストのグルーピングやフィルタリングが可能**
- **ダイナミックテスト：テスト実行にテストケースを自動生成**
- **拡張モデル：テストメソッドやコンストラクターのパラメータを動的解決やカスタムロジックの実行**

（注2）　JUnitとAssertJは、サードパーティ製ライブラリです。Java標準ライブラリと違って、ライブラリとなる.jarファイルを取得し、クラスパスに設定する必要があります。
（注3）　主要なビルドツールにMavenやGradleがあります。

JUnitは他にも役立つ機能がありますが、本書のスタンスとしては基本的な文法や考え方を説明します。応用的な使い方を知りたい方は、公式ドキュメントや関連書籍を参照してください。本章を読み終えた頃には、基本部分を理解し、応用部分を理解しやすくなっています。

App.8-3-3　テストフレームワークの慣習

Javaでは、テストコードを実装する上で、広く知られている慣習があります。主な慣習をまとめると次のようになります。

- **テストコードとソースコードの分離**
- **テストクラスやテストメソッドの命名**
- **1テストメソッド 1検証**
- **AAAパターン**
- **モックの使用**
- **カバレッジ生成**
- **継続的なテスト実行**

上記は代表的な慣習です。Java特有のものから、言語を問わず広く使われるものもあります。慣習はルールではありません。プロジェクトやチームによっては、特有の事情があり慣習に反することがあります。慣習を意識することも大事ですが、それだけに捉われず、柔軟に対応してください。

■テストコードとソースコードの分離

テスト対象となるプログラムとテストコードを別ファイルに分ける慣習があります。以降、テスト対象となるプログラムをソースコードと呼びます。Mavenなどのビルドツールを活用する場合は、テストコードを所定のフォルダに格納します。ソースコードとテストコードを分けることで、コード管理がしやすくなります。テストコードはシステムを構成するプログラムではありません。システムの成果物に誤ってテストコードが混入する事態を防ぎやすくなります。

■テストクラスやテストメソッドの命名

JUnitでは、テストクラスやテストメソッドに命名規則がありません。しかし、テストクラスやテストメソッドが何をテストしているかを明確にすべきです。

テストクラスは、テスト対象のクラス名の接頭辞または、接尾辞にTestを付けた名前にします。

（例）クラス名：Sample、テストクラス名：TestSample または、SampleTest

本書では、接尾辞にTestを付けて説明しています。したがって、テストコードのファイル名もテストクラス名に拡張子を付けたファイル名にします。

（例）ソースコード：Sample.java、テストコード：TestSample.java または、SampleTest.java==

システムによっては、クラス名ではなく機能名やシナリオ名を使うことも有用です。

（例）機能名：PaymentProcessingTest、シナリオ名：AddingItemsToCartScenarioTest

テストメソッドは、テスト対象のメソッド名の接頭辞または、接尾辞にtestを付けた名前にします。

（例）メソッド名：findMax、テストメソッド名：testFindMax または、findMaxTest
　　現代の傾向では、テストメソッド名を自然言語に近い名前にします。
（例）shouldNotSendReceiptWhenCreditCardPaymentFails や、givenEmptyString_whenCalculateLength_thenZero[注4]

■1テストメソッド1検証

テストメソッドが何をテストしているか明確にするため、1つのテストメソッドは1つの検証のみという考え方があります。これにより、テストメソッドの意図を理解しやすくなり、可読性や効率性に寄与します。

1つのテストメソッドに複数の検証が含まれている場合、テストケースの失敗時の原因調査と解消がしづらくなります。失敗の原因がどの検証かの特定や、前検証の影響がないかなどの調査に時間がかかるためです。

テストコード自体のバグによりテストケースが失敗している可能性も十分ありえます。
1テストメソッド1検証という考え方に基づくことで、テストコードのバグも抑制しやすくなります。

■AAAパターン

AAAパターンとは、テストをArrange、Act、Assertionと呼ばれる3つのセクションで構成しアプローチする手法です。

Arrangeは、テストの準備作業を担います。テストに必要となる入力データや環境変数、その他環境のセットアップが該当します。

Actは、テストの実行作業を担います。テスト対象となるメソッドの呼び出しと結果の取得が該当します。

Assertは、テスト結果の検証作業を担います。取得した結果と期待値を比較した検証が該当します。

これらの構成を用いることで、テストコードの構造が明確になり、可読性や効率性が向上します。

（注4）　このテストメソッド名は、BDD（Behavior-Driven Development）と呼ばれる開発手法でよく使われます。Given-When-Thenフォーマットに従い命名しています。

■モックの使用

外部システムとの依存関係を持つクラスやメソッドは、テストコードの実装が容易ではありません。
このパターンでは、多くの場合モックを使います。
どのように使うかは、後述する「App.8-6　モック」にて説明します。

C O L U M N

カバレッジデータの代表例

テストの網羅率として、未実行命令（Missed Instructions）と未実行分岐（Missed Branches）があります。

未実行命令は、テスト中に実行されなかったコードの命令を示します。より具体的には、メソッドがテストにより実行されたかを数値化できます。ここでのメソッドとは、条件分岐がないメソッドを指します。

網羅率 = テスト済みのメソッド数 / メソッド数

上記式より、未実行命令を基にした網羅率が低い場合は、テスト漏れのメソッドが存在することを表します。

未実行分岐は、テスト中に実行されなかったコードの条件分岐を示します。より具体的には、if文などの条件分岐を実行したかを数値化できます。

網羅率 = テスト済みの条件分岐数 / 条件分岐数

上記式より、未実行分岐を基にした網羅率が低い場合は、テスト漏れの条件分岐が存在することを表します。

（例）if-else文におけるテスト

```
if（条件）{
    // 処理
} else {
    // 処理
}
```

条件を満たすテストケースのみ実行した場合、ifブロック内の処理のみ実行します。elseブロックの処理は実行しません。このパターンでは、条件分岐が2つ存在します。ifブロックのみテストしているため、テスト済みの条件分岐は1つになります。よって、網羅率 = 1/2 = 50% より、半分の条件分岐がテストしていないことを示せます。

■カバレッジ生成

テスト実行時、ソースコードがどの程度実行されたかを示す網羅度合いやテスト実行時間などを生成できます。カバレッジツールの広く使われるライブラリは、JaCoCoです。手動でカバレッジを生成すると手間が大きいため、ビルドツールを使ってカバレッジを生成します。

カバレッジを生成することで、テスト度合いが不十分かを判断できます。しかし、テスト度合いが十分かの判断には向きません。指標や基準値として扱う場合には、十分注意してください。

App.8-3-4　テストコードの構造

基本的な操作を行うユーティリティクラスが提供されています。このユーティリティクラスを使って、テストコードを作成します。

■Assertionsクラス

Assertionsクラスは、使用頻度が一番高いユーティリティクラスです。テストケースにて、テスト結果を検証する役割を担います。AAAパターンのAssertionである検証に該当します。

主に下記のAssertionsクラスが使われています。同名のクラスなので、どちらか一方のみを使います。

- org.junit.jupiter.api.Assertions　JUnitが提供
- org.assertj.core.api.Assertions　AssertJが提供

JUnit提供のAssersionsクラスの使用例を示します（**リスト8.1**）。

リスト8.1　JUnit提供のAssertionsクラスの使用例

```
// 変数 actual はテスト結果の実績値、変数 expected はテスト結果の期待値
Assertions.assertEquals(expected, actual);
```

assertEqualsクラスメソッドに、テスト結果の期待値であるexpectedと実績値であるactualを指定します。指定されたテスト結果の実績値が期待値どおりかを検証します。期待値どおりの場合はテストケースは正常と判定され、期待値と異なる場合はテストケースは異常と判定されます。また、実績値の取得処理は、AAAパターンのActである実行に該当します。JUnitが提供するAssertionsクラスの代表的なクラスメソッドを**表8.3**にまとめます。

表8.3　JUnit提供Asseriionsクラスの代表的なクラスメソッド

メソッド名	用途
assertEquals	期待値、実績値が等しいかを検証
assertNotEquals	2つの値が異なるかを検証
assertTrue	条件がtrueであるかを検証
assertFalse	条件がfalseであるかを検証
assertNull	オブジェクトがnullであるかを検証
assertNotNull	オブジェクトがnullでないかを検証
assertThrows	指定された例外が投げられたかを検証
assertTimeout	指定時間内に完了するかを検証
assertArrayEquals	2つの配列が等しいかを検証
assertIterableEquals	2つのIteratableオブジェクトが等しいかを検証
fail	意図的にテストケースを失敗

AssertJ提供のAssersionsクラスの使用例を示します（**リスト8.2**）。

リスト8.2　AssertJ提供のAssertionsクラスの使用例

```
// 変数 actual はテスト結果の実績値を保持、変数 expected はテスト結果の期待値を保持
Assertions.assertThat(actual).isEqualTo(expected);
```

JUnit提供のassertEqualsクラスメソッドと同様の検証処理です。AssertJ提供のクラスメソッドでは、メソッドチェーンによる記述方式で検証します。検証内容自体に違いはありませんが、AssertJ提供のクラスメソッドのほうが、より自然言語に近いコードになります。そのため、より可読性に優れたテストコードを作成できます。

多くの場合は、AssertJとJUnitを組み合わせてテストコードを作成します。AssertJが提供するAssertionsクラスの代表的なクラスメソッドを**表8.4**にまとめます。

表8.4　AssertJ提供Assertionsクラスの代表的なクラスメソッド

メソッド名	用途
assertThat	オブジェクトに対し検証ができる状態の生成
assertThatThrownBy	コードブロックで指定された例外が投げられたかを検証
assertThatExceptionOfType	assertThatThrownByと同様
assertThatCode	assertThatThrownByと同様
fail	意図的にテストケースを失敗

assertThatクラスメソッドは、検証用メソッドです。メソッドチェーンとあわせて複数の検証を簡潔に作成できます（**表8.5**）。

表8.5　assertThatクラスメソッドとメソッドチェーンでよく使うメソッド

メソッド名	意味
isEqualTo	期待値とオブジェクトが等しいかを検証
isNotEqualTo	期待値とオブジェクトが等しくないかを検証
isNull	オブジェクトがnullかを検証
isNotNull	オブジェクトがnullでないかを検証
isTrue	trueであるかを検証
isFalse	falseであるかを検証
isEmpty	コレクションや文字列が空であるかを検証
isNotEmpty	コレクションや文字列が空でないかを検証
contains	コレクションが指定した要素を含んでいるかを検証

テスト対象となるメソッドの返り値や投げられる例外などの観点に合わせた検証方法が提供されています。検証となる処理を作成することで、テスト処理の本体を実現できます。

■Assertionsクラスの使用例

テスト対象メソッドの返り値に合わせて、テストコードを作成します。そのため、テスト対象となるメソッドがクラスメソッド、インスタンスメソッドどちらでも変わりはありません。この段階では、テストコードの記述方法は知らなくても大丈夫です。テスト処理の根幹となる検証処理の記述の違いに着目してください。

クラスメソッドの検証の例を示します(**リスト8.3**)。

リスト8.3　クラスメソッドの検証

```
// クラスメソッドとしてGreet.helloメソッドが実装されている
// このメソッドは、引数に<name>を受け取り、返り値として文字列"Hello <name>!"を返す
String actual = Greet.hello("Duke");                  // テスト結果の実績値
String expected = "Hello Duke!";                      // テスト結果の期待値

Assertions.assertEquals(expected, actual);          // JUnit提供
Assertions.assertThat(actual).isEqualTo(expected);  // AssertJ提供
```

クラスメソッドをテストする場合は、クラスメソッドを呼び出します。返り値を実績値として、期待値と比較し検証します。

インスタンスメソッドの検証の例を示します(**リスト8.4**)。

リスト8.4　インスタンスメソッドの検証

```
// インスタンスメソッドとしてGreet.helloメソッドが実装されている
// コンストラクタに<name>情報を指定し、保持できる
// このメソッドは、引数なし、返り値として文字列"Hello <name>!"を返す
Greet greet = new Greet("Duke");
String actual = greet.hello();                        // テスト結果の実績値
String expected = "Hello Duke!";                      // テスト結果の期待値
```



```
Assertions.assertEquals(expected, actual);          // JUnit提供
Assertions.assertThat(actual).isEqualTo(expected);  // AssertJ提供
```

インスタンスメソッドをテストする場合は、テスト対象クラスのインスタンスを生成します。インスタンスからメソッドを呼び出し、返り値を実績値として期待値と検証(比較)します。

このように、テストの根幹となる検証処理は、クラスメソッドやインスタンスメソッド共に同様の記述です。

■基本型のテスト

テスト対象メソッドの返り値が基本型の場合は、2つの値を比較し検証します。

リスト8.5　基本型の検証例

```
// expected と actual は int型変数
Assertions.assertEquals(expected, actual);          // JUnit提供
Assertions.assertThat(actual).isEqualTo(expected);  // AssertJ提供
```

boolean型は、専用の検証用メソッドが用意されています。
上記の検証用メソッドでも検証可能ですが、可読性の観点から後述の記載を推奨します。

リスト8.6　boolean型の検証例

```
// 変数 actual は boolean型変数
Assertions.assertTrue(actual);          // JUnit提供
Assertions.assertFalse(actual);         // JUnit提供
Assertions.assertThat(actual).isTrue();  // AssertJ提供
Assertions.assertThat(actual).isFalse(); // AssertJ提供
```

■単体オブジェクトのテスト

テスト対象のメソッドの返り値が単体のオブジェクトの場合は、基本型のテストと同様に検証します。

リスト8.7　単体オブジェクトの検証例

```
// 変数 expected と actual は単体のオブジェクト
Assertions.assertEquals(expected, actual);
Assertions.assertThat(actual).isEqualTo(expected);
```

この検証処理の実態は、オブジェクトに実装されているequalsメソッドが実行されています。そのため基本型のように比較演算子が使えなくても、比較による検証が可能となります。

■コレクションのテスト

テスト対象のメソッドの返り値がコレクションの場合は、テストすべき観点が複数あります。コレクションが保持するオブジェクトの個数やその内容などです。

リスト8.8　コレクションの検証例

```
// actual は 下記の文字列を持つ ArrayList<String>オブジェクト変数
// sun, mercury, venus, earth, mars, jupiter, saturn, uranus, neptune

/* JUnit提供 */
Assertions.assertFalse(actual.isEmpty());
Assertions.assertEquals(9, actual.size());
Assertions.assertTrue(actual.contains("earth"));

/* AssertJ提供 */
Assertions.assertThat(actual)
          .isNotEmpty()
          .hasSize(9)
          .contains("earth");
```

上記処理は、下記の内容で検証しています。

① コレクションが空でない
② コレクションのサイズが９である。すなわち、９つの文字列を持っている
③ コレクションの中に文字列 "earth" が存在する

検証自体は、JUnit提供メソッドとAssertJ提供メソッドで違いがありません。AssertJ提供メソッドは、メソッドチェーンで検証処理を繋げられるため、JUnit提供メソッドよりも読みやすいコードになっています。また、コレクションに対応した検証メソッドが提供されています。自然言語に近いメソッド名になっていることも、可読性が優れている特徴の１つです。

■例外処理のテスト

テスト対象のメソッドが特定の条件下で、正しく例外を投げるかを検証できます。特定の条件下を設定し、テスト対象のメソッドを実行します。実行した結果から、特定の例外が投げられているか検証します。

int型の除算を計算するクラスメソッドを例として説明します。このクラスメソッドは、下記の条件下で例外を投げます。

- **条件**：**ゼロ除算**
- **例外**：ArithmeticException
- **例外時のメッセージ**：**ゼロ除算が発生しました。**

この条件下で、期待する例外やメッセージが投げられるか検証するテストコードの例を示します（**リスト8.9**）。

リスト8.9　例外処理の検証例

```
// Calculator クラスは int型の除算結果を返す divide クラスメソッドを持つ
// ゼロ除算時に ArithmeticException が投げられる

/* JUnit提供 */
Exception exception = Assertions.assertThrows(ArithmeticException.class, () -> Calculator.divide(1, 0));
Assertions.assertEquals("ゼロ除算が発生しました。", exception.getMessage());

/* AssertJ提供 */
Assertions.assertThatThrownBy(() -> Calculator.divide(1, 0))
          .isInstanceOf(ArithmeticException.class)
          .hasMessageContaining("ゼロ除算が発生しました。");
```

```
Calculator.divide(1, 0)
```

テスト対象となるクラスメソッドがゼロ除算である１÷０を計算するように、条件を設定しています。

```
() -> Calculator.divide(1, 0)
```

JUnitとAssertJでは、テスト対象となるコードブロックを指定します。コードブロックとは、特定の操作を行う処理の集合を指します。コードブロックとしてメソッドを指定するには、ラムダ式や匿名クラスを使って記述します。

この例では、ラムダ式を使って指定しています(注5)。

```
Assertions.assertThrows(ArithmeticException.class, () -> Calculator.divide(1, 0));
Assertions.assertEquals("ゼロ除算が発生しました。", exception.getMessage());
```

JUnitでは、投げられた例外の検証にassertThrowsクラスメソッドを使います。第１引数は、期待する例外を指定します。第２引数は、コードブロックを指定します。コードブロックは、テスト対象となる処理です。第２引数で指定された処理結果の例外が、第１引数に指定された例外クラスに代入できることを検証しています。instanceof演算と同値の処理です。

```
<発生した例外> instanceof <指定した例外>
```

assertThrowsクラスメソッドは、例外オブジェクトを返り値とします。例外オブジェクトのgetMessageメソッドを使って、例外メッセージを取得し検証できます。

(注5)　コードブロックと表現している背景は、テスト対象がメソッドなどの処理自体に重点が置かれているためです。処理の結果ではありません。

```
Assertions.assertThatThrownBy(() -> Calculator.divide(1, 0))
        .isInstanceOf(ArithmeticException.class)
        .hasMessageContaining("ゼロ除算が発生しました。");
```

AssertJでは、投げられた例外やメッセージの検証にassertThatThrownByクラスメソッドを使います(注6)。第1引数は、コードブロックを指定します。検証内容は、メソッドチェーンで指定します。

投げられた例外の検証にisInstanceOfメソッドを使います。期待する例外を引数に指定します。

また、例外メッセージの検証は、hasMessageContainingメソッドを使います。期待する例外メッセージを引数に指定します。

■時系列処理のテスト

テストの対象は、処理の結果ばかりではありません。下記の観点も重要です。

- **テスト対象処理の実行時間**
- **テスト対象処理の前後関係**
- **テスト対象オブジェクトの状態遷移**

テスト対象処理の実行時間を観点としたテストは、「**App.8-4-8　タイムアウトの設定**」を参照してください。テスト対象処理の前後関係を観点としたテストは、「**App.8-6-4　モックの構造**」の「InOrderクラス」を参照してください。

テスト対象オブジェクトの状態遷移を観点としたテストは、時間や処理の流れで適切な状態に変化しているか検証します。検証処理自体は、前述のいずれかの検証メソッドを使って実装できます。

状態遷移を検証するテストコードの例を示します(**リスト8.10**)。

リスト8.10　状態遷移の検証例

```
// 変数 task は、何らかの処理を実行するオブジェクト
// getState メソッドを呼び出し、下記のいずれかの処理状態を取得できる
// State列挙型に定義[PENDING: 初期状態, RUNNING: 処理中, COMPLETED: 処理完了]

/* JUnit提供 */
Assertions.assertEquals(State.PENDING, task.getState());
// タスクの開始処理
Assertions.assertEquals(State.RUNNING, task.getState());
// タスク完了処理
Assertions.assertEquals(State.COMPLETED, task.getState());

/* AssertJ提供 */
Assertions.assertThat(task.getState()).isEqualTo(State.PENDING);
// タスクの開始処理
Assertions.assertThat(task.getState()).isEqualTo(State.RUNNING);
// タスク完了処理
Assertions.assertThat(task.getState()).isEqualTo(State.COMPLETED);
```

リスト8.10のテストコードは、処理の進行に応じて適切に状態が設定されているか検証しています。複数のメソッドがオブジェクトに影響を与える場面において、各メソッドが期待どおりに機能しているかをテストできます。

(注6)　assertThatExceptionOfType、assertThatCode、catchThrowable。これらのクラスメソッドでも例外処理を検証できます。

App.8-4　テスト実行の制御

テスト処理となる箇所の宣言や設定を担うアノテーションが提供されています。このアノテーションを使って、テストメソッドをフレームワークに認識させます。また、テストの設定をアノテーションで宣言することで、テスト処理のコードに対する影響を最小限にできます。

よく使うアノテーションを**表8.6**にまとめます。

表8.6　よく使うアノテーション

アノテーション	意味
@Test	テストメソッドの定義
@BeforeEach	各テストメソッドの実行前に実行するメソッドの定義
@AfterEach	各テストメソッドの実行後に実行するメソッドの定義
@BeforeAll	テストクラスの中で1番最初に実行するメソッドの定義
@AfterAll	テストクラスの中で1番最後に実行するメソッドの定義
@DisplayName	テストメソッドに別名を定義。テストレポートへの表示用
@Disabled	テストメソッドを一時的に無効化
@ParameterizedTest	テストメソッドのパラメータ化。異なる入力値でテストメソッドの複数回実行を実現
@Nested	ネストされたテストクラスの定義。テストケースを階層化
@Tag	テストにタグを指定
@Timeout	テストメソッドにタイムアウト値を設定

用途別にアノテーションを適切に使い分けることで、テストケースのみならず、テストスイート全体を最適なテスト構造にします。

App.8-4-1　テストメソッドの宣言

テストメソッドの宣言には、@Testアノテーションを使います。メソッドに@Testアノテーションを付与するだけで、テストフレームワーク側でテストメソッドとして認識されます。そのため、テスト実行時は@Testアノテーションが付与されたメソッドが自動で実行されます。

テストメソッドの宣言コードを示します(**リスト8.11**)。

リスト8.11　テストメソッドの宣言

```
@Test
void testExample() {
    // テスト処理
}
```

App.8-4-2　ライフサイクルメソッドの宣言

ライフサイクルメソッドは、テストメソッドの実行前後に実行するメソッドを指します。テストメソッドのテスト環境や設定の初期化や削除などで役立つメソッドです。AAAパターンのArrangeである準備に該当します。準備だけでなく、後片付けも可能です。捉え方次第では、次のテストのための準備とも言えます。

ライフサイクルメソッドも特定のアノテーションを付与するだけの簡単な操作です。テストコード実装で頻出するアノテーションですので、よく理解しておきましょう。また、ライフサイクルメソッド名は、自由に命名できます。

■前処理メソッドの宣言

前処理メソッドの宣言には、@BeforeEachアノテーションを使います。前処理メソッドは、セットアップメソッド（Setup Method）とも呼びます。名前のとおり、テストのためにセットアップ処理を実行する目的で使うメソッドになります。

テスト実行時は、テストメソッドの実行前に前処理メソッドを実行します。

前処理メソッドの主な目的は、テストケースの初期設定です。
主な使用例を下記に挙げます。

- **テスト対象のオブジェクトを生成**
- **モックの生成　※「App.8-6-4　モックの構造」の「Mockitoクラス」にて後述**
- **共有リソースの初期化　　※重い初期化処理は、「初期化メソッドの定義」にて後述**
- **テストデータの生成**
- **オブジェクトなどの状態のリセット**

上記の処理は、テストケース間の干渉を抑制します。前処理メソッドの宣言コードを示します（**リスト8.12**）。

リスト8.12　前処理メソッドの宣言

```
@BeforeEach
void setUp() {
    // 初期設定の処理
}
```


■後処理メソッドの宣言

後処理メソッドの宣言には、@AfterEachアノテーションを使います。後処理メソッドは、ティアダウンメソッド（Teardown Method）とも呼びます。名前のとおり、テストのためにティアダウン処理を実行する目的で使うメソッドになります。

テスト実行時は、テストメソッドの実行後に後処理メソッドを実行します。

後処理メソッドの主な目的は、テストケースの後片付けです。
主な例を下記に挙げます。

- **モックのリセット　※「App.8-6-4　モックの構造」の「モックのリセット」にて後述**
- **共有リソースのリセット**
- **プロパティのリセット**
- **スレッドの終了**
- **リソースの解放**

上記の処理は、テストケース間の干渉を抑制します。後処理メソッドの宣言コードを示します（**リスト8.13**）。

リスト8.13　後処理メソッドの宣言

```
@AfterEach
void tearDown() {
    // 後片付けの処理
}
```

■初期化メソッドの宣言

初期化メソッドの宣言には、@BeforeAllアノテーションを使います。初期化メソッドは、イニシャライゼーションメソッド（Initialization Method）とも呼びます。名前のとおり、テストのために初期化処理を実行する目的で使うメソッドになります。

前処理メソッドとの主な違いは、初期設定の対象です。前処理メソッドの対象は、テストケースです。すなわち、テストメソッドです。初期化メソッドの対象は、テストスイートです。すなわち、テストクラスです。テストクラス間で共有するリソースの初期設定や重い初期設定などは初期化メソッドに実装します。

また、前処理メソッドは、テストクラスのインスタンスメソッドとして定義しますが、
初期化メソッドは、テストクラスのクラスメソッドとして定義します。

テスト実行時は、1番最初に初期化メソッドが実行されます。

初期化メソッドの主な目的は、テストスイートの初期設定です。主な例を下記に挙げます。

- データベースや外部サービスなどの外部リソースの接続
- 共有リソースの初期化
- テスト用ロガーの生成
- プロパティの設定

初期化メソッドの宣言コードを示します（**リスト8.14**）。

リスト8.14　初期化メソッドの宣言

```
@BeforeAll
static void init() {
    // 初期設定の処理
}
```

■終了処理メソッドの宣言

終了処理メソッドの宣言には、@AfterAllアノテーションを使います。終了処理メソッドは、クリーンアップメソッド（Cleanup Method）とも呼びます。名前のとおり、テストのために後片付け処理を実行する目的で使うメソッドになります。

後処理メソッドとの主な違いは、後片付けの対象です。後処理メソッドの対象は、テストケースです。すなわち、テストメソッドです。終了処理メソッドの対象は、テストスイートです。すなわち、テストクラスです。テストクラス間で共有するリソースのリセット・削除や重い初期設定などは終了処理メソッドに実装します。

また、後処理メソッドは、テストクラスのインスタンスメソッドとして定義しますが、終了処理メソッドは、テストクラスのクラスメソッドとして定義します。

テスト実行時は、1番最後に終了処理メソッドが実行されます。

終了処理メソッドの主な目的は、テストスイートの後片付けです。主な例を下記に挙げます。

- データベースや外部サービスなどの外部リソースの切断
- 共有リソースの解放
- 使用済みテスト環境の削除

終了処理メソッドの宣言コードを示します（**リスト8.15**）。

リスト8.15　終了処理メソッドの宣言

```
@AfterAll
static void cleanUp() {
    // 後片付けの処理
}
```


App.8-4-3　テストケース名の設定

テストケース名の設定は、@DisplayNameアノテーションを使います。テストケースの内容は、テストメソッド名やテスト処理を読むことで理解できます。テストケース名の設定は、テストケース内容のさらなる理解に役立ちます。

テストケース名の設定により、テストの意図を明確に示せます（**リスト8.16**）。

リスト8.16　テストケース名の設定例

```
@Test
@DisplayName("入力値をユーザIDとしてデータベースからユーザ情報が取得できることを確認")
// テストメソッドの実装
```

テストケース名は、カバレッジレポートに記載されます。テスト失敗時、原因調査やテスト内容理解に役立ちます。

App.8-4-4　テストケースの無効化

テストケースの無効化には、@Disabledアノテーションを使います。テストケースの無効化は、既に実装されたテストケースを一時的に実行したくない時に役立ちます。

主に下記のケースでは、誤ってテストケースが実行されないように、テストケースを無効化することがあります。

- 未実装のテストケース
- テストケースに対応するバグの修正中
- 環境依存のテストケース
- 長い時間を要するテストケース

@Disabledアノテーションはデバッグ用途です。一時的なテストケースの無効化以外では推奨されません。常に無効化するテストケースは、そのテストケースが本当に必要か検討してください。

また、@Disabledアノテーションの削除忘れにより、本来テストすべきテストケースが実行されない事態が発生します。品質に問題を抱えたままのシステムを世に放つことになりかねないため、扱いには十分注意してください。

テストケースの無効化例を示します（**リスト8.17**）。

リスト8.17　テストケースの無効化例

```
@Test
@Disabled("このテストは一時的に無効化されています")
// テストメソッドの実装
```

App.8-4-5　テストメソッドのパラメータ化

テストメソッドのパラメータ化は、@ParameterizedTestアノテーションを使います。テストメソッドをパラメータ化することで、複数の異なる入力値に対するテストケースを容易に作成できます。

複数の異なる入力値を設定するアノテーションも提供されています。これらのアノテーションを**表8.7**にまとめます。

表8.7　入力値を設定するアノテーション

アノテーション	説明
@ValueSource	short, byte, int, long, float, double, char, boolean, java.lang.String, java.lang.Classの型を入力値として設定
@NullSource	nullを入力値として設定
@EmptySource	空の入力値を設定
@NullAndEmptySource	@NullSourceと@EmptySourceの組み合わせ
@EnumSource	列挙型から特定の値を入力値として設定
@MethodSource	特定のメソッドから返される値を入力値として設定
@CsvSource	CSV形式の文字列から値を入力値として設定
@CsvFileSource	CSVファイルから値を入力値として設定
@ArgumentsSource	カスタムプロバイダを使用して複雑な引数を入力値として設定

@ParameterizedTestアノテーションと入力値の設定用アノテーションはセットで覚えてください。

テストメソッドのパラメータ化とパラメータのソースの設定例を示します（**リスト8.18**）。

リスト8.18　正規表現を満たさないパラメータの設定例

```
@ParameterizedTest
@ValueSource(strings = {"1234567", "ABCDEFG", "123456789", "ABCDEFGHI", "1234abcd", ""}) // 正規表現を満たさないパラメータのソースを設定
void testInvalidRegexPatterns(String input) {
    // 変数 REGEX は、正規表現[0-9A-Z]{8}が設定されている
    assertFalse(Pattern.matches(REGEX, input));
}
```

リスト8.18は、正規表現[0-9A-Z]{8}(注7)を満たさない文字列をパラメータのソースとして設定しています。このパラメータのソースごとにテストケースを実行します。

このように、複数の入力値ごとにテストケースを実行できます。データ駆動テストや境界値分析を始めとするテスト技法に適しています。

（注7）　正規表現[0-9A-Z]{8}は、0から9までの数字とAからZまでの大文字で組み合わされた8文字の文字列を表す

App.8-4-6　テストケースのネスト化

テストケースのネスト化は、@Nestedアノテーションを使います。テストケースのネスト化は、複雑なテストケースの管理に役立ちます。

@Nestedアノテーションはテストクラスの内部クラスに付与します。この内部クラスをネストされたテストクラスやネステッドテストクラスと呼びます。テストクラスとネストされたクラスをより区別しやすくするため、テストクラスをトップレベルテストクラスとも呼びます。

ネストされたテストクラスを入れ子状態とした場合は、その親となるテストクラスのコンテキストが適用されます。親クラスのメソッドの実行順序に影響を受けます。実行順序の詳細は、「**App.8-4-9 テストのライフサイクル**」の「ネストされたテストクラスのライフサイクル」を参照してください。

ネストされたテストクラスの設定例を示します（**リスト8.19**）。

リスト8.19　テストケースのシステムモデル別グルーピング例

```
class ExampleTest {
    @Nested
    class ModelStdTests {
        // 複数のテストケースを実装
    }

    @Nested
    class ModelProTests {
        // 複数のテストケースを実装
    }
}
```

リスト8.19は、テストケースをシステムのモデル別にクラスとしてまとめています。ネスト化した内部クラス名は、自由に命名できます。

App.8-4-7　タグの設定

テストケースやテストスイートのタグ付けには、@Tagアノテーションを使います。タグ付けすることで、テストスイートの整理や選択的なテスト実行ができます。

テストスイートやテストケースのタグの設定例を示します（**リスト8.20**）。

リスト8.20　タグの設定例

```
@Tag("integration")
class IntegrationTest {
    @Test
    @Tag("database")
    void testDatabaseConnection() {
```

```
        // データベース接続のテスト
    }

    @Test
    @Tag("network")
    void testNetworkCall() {
        // ネットワーク呼び出しのテスト
    }
}
```

ビルドツールを使う場合は、設定ファイルにテスト実行するタグを記載し、特定のタグのみテストを実行します。ビルドツールを使わない場合は、コマンドラインで--include-tagオプションに実行するタグを指定します。

packageA内のテストコードにおいて、"integration"タグが設定されたテストを実行するコマンドの例を示します（**リスト8.21**）。

```
$ java -jar junit-platform-console-standalone-5.10.1.jar ¥
    --include-tag=integration ¥
    --class-path=./src/test/java/packageA ¥
    --scan-class-path
```

App.8-4-8　タイムアウトの設定

テストスイートやテストケースのタイムアウト設定には、@Timeoutアノテーションを使います。リソースが限られている環境や長時間実行すると問題が生じるような状況に役立ちます。

@Timeoutアノテーションの引数にタイムアウト時間を指定します。

指定できる時間の単位は、ナノ秒単位から日単位までです（**表8.8**）。デフォルトは、秒単位として扱われます。

java.util.concurrent.TimeUnit列挙型を使っていますが、それ以外の列挙型も指定できます。

表8.8　時間単位別の指定例

項目	説明
TimeUnit.NANOSECONDS	ナノ秒単位（1秒の10億分の1）
TimeUnit.MICROSECONDS	マイクロ秒単位（1秒の100万分の1）
TimeUnit.MILLISECONDS	ミリ秒単位（1秒の1000分の1）
TimeUnit.SECONDS	秒単位
TimeUnit.MINUTES	分単位
TimeUnit.HOURS	時間単位
TimeUnit.DAYS	日単位

タイムアウト時間を設定したテストコードの例を示します（**リスト8.22**）。

リスト8.22　タイムアウト時間の設定

```
@Test
@Timeout(value = 30, unit = TimeUnit.SECONDS) // 30秒でタイムアウト
void テストメソッド名() {
    // テスト対象の処理を実行
}
```

割り込み可能なブロッキング操作を持つ処理は、割り込みが発生する可能性があります。

割り込みが発生した場合は、InterruptedException例外が投げられます。

該当するテストメソッドには、明示的に例外をthrows句で宣言してください（**リスト8.23**）[注8]。

割り込み可能なブロッキング操作の主な例は、下記があります。

- **スレッドのスリープ**：Thread.sleepメソッド
- **待機状態のオブジェクト**：Object.waitメソッド
- **ブロッキングキュー操作**：BlockingQueue.takeメソッド、BlockingQueue.putメソッド

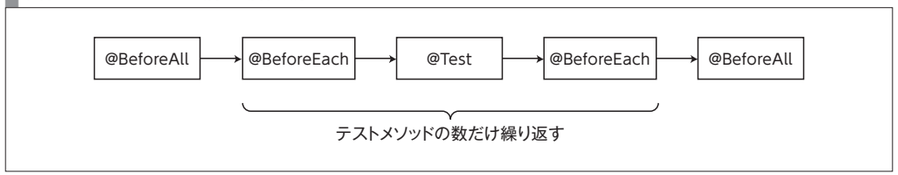
リスト8.23　ブロッキング処理におけるタイムアウト時間の設定

```
@Test
@Timeout(value = 30, unit = TimeUnit.SECONDS)
void テストメソッド名() throws InterruptedException {
    // テスト対象の処理を実行
}
```

App.8-4-9　テストのライフサイクル

前述のアノテーションを付与したメソッドの実行順序を整理します（**図8.2**）。

図8.2　アノテーション別の動作順序



各アノテーションの実行順序と使用例は、下記のとおりです。

① 初期化メソッド（@BeforeAll）：テストスイート内で共有して使用するデータベースの接続

（注8）　テストフレームワークや実行環境で自動的にハンドリングする場合もあります。その場合は、例外の明示的な宣言は不要です。

② 前処理メソッド（@BeforeEach）：テーブルに登録するテストデータの生成
③ テストメソッド（@Test）：テーブルへテストデータの登録処理の実行と結果の検証
④ 後処理メソッド（@AfterEach）：テーブル情報のリセット
⑤ 終了処理メソッド（@AfterAll）：テストスイート内で共有して使用するデータベースの切断

手順の②から④は、テストケースの数だけ繰り返します。

■ネストされたテストクラスのライフサイクル

ネストされたクラスが存在する場合は、テストの実行順序に注意が必要です。下記の観点を押さえておくと理解しやすいです。説明の便宜上、テストクラスをトップレベルテストクラス、ネストされたテストクラスをネステッドテストクラスと称します。

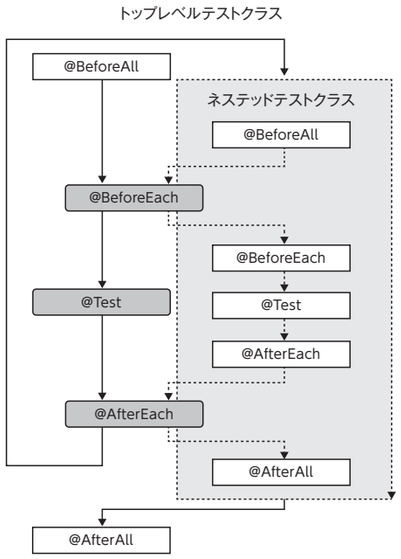
- **テストケース実行時の優先順位：トップレベルテストクラス > ネステッドテストクラス**
- **トップレベルテストクラスのフィールドやメソッドは、ネステッドテストクラスにも引き継がれる**

トップレベルテストクラスとネステッドテストクラスそれぞれに、テストメソッド、初期化メソッド、前処理メソッド、後処理メソッド、終了処理メソッドが実装されていると仮定します。この場合のメソッドの実行順序は、下記のとおりです。

① [トップレベルテストクラス]　初期化メソッド（@BeforeAll）
② [トップレベルテストクラス]　前処理メソッド（@BeforeEach）
③ [トップレベルテストクラス]　テストメソッド（@Test）
④ [トップレベルテストクラス]　後処理メソッド（@AfterEach）
⑤ [ネステッドテストクラス]　初期化メソッド（@BeforeAll）
⑥ [トップレベルテストクラス]　前処理メソッド（@BeforeEach）
⑦ [ネステッドテストクラス]　前処理メソッド（@BeforeEach）
⑧ [ネステッドテストクラス]　テストメソッド（@Test）
⑨ [ネステッドテストクラス]　後処理メソッド（@AfterEach）
⑩ [トップレベルテストクラス]　後処理メソッド（@AfterEach）
⑪ [ネステッドテストクラス]　終了処理メソッド（@AfterAll）
⑫ [トップレベルテストクラス]　終了処理メソッド（@AfterAll）

手順②から④は、トップレベルテストクラスのテストケースの数だけ繰り返します。
手順⑥から⑩は、ネステッドテストクラスのテストケースの数だけ繰り返します。
手順⑤から⑪は、ネステッドテストクラスの数だけ繰り返します。
これらの実行順序を図示化します（**図8.3**）。

図8.3　トップレベルクラスとネステッドテストクラスにおけるアノテーションの動作順序



App.8-5　テストコードの実装例

ここまでで説明したクラスやアノテーションを使って、テストコードを実装します。
より実践的なコードを読み進めながら、使いこなす練習としてください。

App.8-5-1　テスト対象機能

オンラインショッピングを実現するアプリケーションを想定します。アプリケーションのProductServiceクラスは、商品の登録・更新・削除・検索機能を実装しています。商品の情報は、データベースに登録されており、データベース上のデータを取得・更新・削除します。

ここでは、商品の検索機能を担うProductServiceクラスのsearchインスタンスメソッドをテスト対象とします。

App.8-5-2　テストシナリオ

キーワード検索により、一致するキーワードに基づく情報が取得できるかや、一致しないキーワードからは情報が取得できないことを検証します。

○キーワードに一致する商品が存在する場合
使用するキーワード：laptop, phone, tablet
検索結果：商品が１つ以上取得できる

searchメソッド視点では、引数にキーワードを受け取ります。
データベースからキーワードに一致する商品情報を取得し、メソッドの返り値として商品情報を返します。

○キーワードに一致する商品が存在しない場合
使用するキーワード：空文字列
検索結果：商品が取得できない

searchメソッド視点では、引数にキーワードを受け取ります。データベースからキーワードに一致する商品情報を取得できず、メソッドの返り値として空の商品情報を返します。

App.8-5-3　テストコードの実装

テストシナリオに基づくテストコードを示します（**リスト8.24**）。

リスト8.24　テストシナリオに基づくテストコード

```
import static org.assertj.core.api.Assertions.assertThat;
import org.junit.jupiter.api.AfterEach;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.params.ParameterizedTest;
import org.junit.jupiter.params.provider.EmptySource;
import org.junit.jupiter.params.provider.ValueSource;
import java.util.List;

class ProductServiceTest {
    private ProductService svc;

    @BeforeEach
    void setUp() {
        svc = new ProductService();
    }

    @AfterEach
    void tearDown() {
        if (svc != null) {
            svc.close();
        }
    }

    @ParameterizedTest(name = "For example, existing product(s) when searching by word: ¥"{0}¥"")
    @ValueSource(strings = {"laptop", "phone", "tablet"})
    void searchWithValidKeywordShouldReturnProductList(String keyword) {
        List<Product> result = svc.search(keyword);
        assertThat(result).isNotEmpty();
    }

    @ParameterizedTest(name = "For example, non existing product when searching by word: ¥"{0}¥"")
    @EmptySource
    void searchWithEmptyKeywordShouldReturnEmptyList(String keyword) {
        List<Product> result = svc.search(keyword);
        assertThat(result).isEmpty();
    }
}
```

ProductServiceTestは、テストシナリオに従ったテストクラスです。テスト対象となるProductServiceクラスのsearchインスタンスメソッドを実行するために、テストクラスのインスタンスフィールドにProductServiceオブジェクトの参照を持ちます。テストクラスのフィールドにテスト対象となるオブジェクトの参照を持ち、複数のテストメソッドを跨いで使います。

前処理メソッドのsetUpメソッドにて、ProductServiceオブジェクトを生成しています。本処理によって、各テストメソッドは常に初期化されたProductServiceオブジェクトを使えます。

後処理メソッドのtearDownメソッドにて、ProductServiceオブジェクトを削除し、リソースを解放しています。ここでは、ProductServiceクラスがリソース解放を行うcloseインスタンスメソッドを実装しています。データベースの接続情報などのリソース管理処理も持つオブジェクトは、このようにテストメソッドの実行後に解放することが多いです。

searchWithValidKeywordShouldReturnProductListメソッドは、1つ目のテストメソッドです。テストシナリオのキーワードに一致する商品が存在する場合のテストケースを実装しています。

@ParameterizedTestアノテーションで、テストデータを引数に持つテストメソッドを定義しています。このアノテーションの引数にて、nameフィールドを指定しています。nameフィールドに指定した文字列は、テストログに記録されます。記録されるテストログの例は、「**App.8-5-4 テストコードの実行**」を参照してください。

@ValueSourceアノテーションで、テストデータとなる文字列を定義しています。このアノテーションの引数に指定した"laptop"、"phone"、"tablet"それぞれの文字列ごとにテストメソッドが実行されます。

テストメソッドは、引数にテストデータとなる文字列を受け取ります。テスト対象となるProductServiceクラスのsearchインスタンスメソッドにテストデータを引数に指定し実行しま

す。実行結果をresult変数に代入しています。

AssertJ提供のassertThatクラスメソッドで、実行結果を検証しています。ここでは、キーワードに一致する商品が取得できている想定のため、isNotEmptyメソッドで商品情報が空でないことを検証しています。テスト用データベースに登録したテストデータがわかる場合は、要素の数や特定の要素が含まれているかを追加で検証することもできます。

searchWithEmptyKeywordShouldReturnEmptyListメソッドは、2つ目のテストメソッドです。テストシナリオのキーワードに一致する商品が存在しない場合のテストケースを実装しています。

1つ目のテストメソッドとの違いは、指定したテストデータです。@EmptySourceアノテーションにより空文字列をテストデータとしています。テスト対象となるProductServiceクラスのsearchインスタンスメソッドの返り値は、商品情報が取得できていない想定のため、isEmptyメソッドでリストが空であることを検証しています。searchインスタンスメソッドが商品情報を取得できない場合に例外を投げるように実装されている場合は、適切な例外が投げられているかを検証できます。

App.8-5-4　テストコードの実行

テストコードを実行する方法はいくつかあります。ここでは、主要なスタイルを説明します。

■コンソール上でのテスト実行

JUnitやAssertJを使ったテストコードの実行は、コンソールランチャーが必要です。コンソールランチャーは、コンソールからJUnitプラットフォームを起動するスタンドアロンアプリケーションです。

C O L U M N

staticインポートを使った呼び出し

クラス名を省略してクラスメソッドやクラスフィールドを直接呼び出すことができます。

通常のインポート
```
import org.assertj.core.api.Assertions;
Assertions.assertThat(actual).isEqualTo(expected);
```

staticインポート
```
import static org.assertj.core.api.Assertions.assertThat;
assertThat(actual).isEqualTo(expected);
```

テストフレームワークの多くはクラスメソッドやクラスフィールドとして実装されています。そのため、staticインポートと相性が良く、テストコードの実装で広く使われています。

コンソールランチャーのアーカイブファイル(.jar)をダウンロードし、javaコマンドで起動します。コンソールランチャーによるテストの実行例を示します(**リスト8.25**)。

```
$ ls
junit-platform-console-standalone-1.10.2.jar
assertj-core-3.24.2.jar
ProductServiceTest.java
ProductServiceTest.class
...省略...

(Linux/Mac)
$ java -jar junit-platform-console-standalone-1.10.2.jar execute --class-path=".:assertj-core-
3.24.2.jar" --scan-class-path
(Windows)
$ java -jar junit-platform-console-standalone-1.10.2.jar execute --class-path=".;assertj-core-
3.24.2.jar" --scan-class-path
.
+-- JUnit Jupiter [OK]
| '-- ProductServiceTest [OK]
|   +-- searchWithEmptyKeywordShouldReturnEmptyList(String) [OK]
|   | '-- For example, non existing product when searching by word: "" [OK]
|   '-- searchWithValidKeywordShouldReturnProductList(String) [OK]
|     +-- For example, existing product(s) when searching by word: "laptop" [OK]
|     +-- For example, existing product(s) when searching by word: "phone" [OK]
|     '-- For example, existing product(s) when searching by word: "tablet" [OK]
+-- JUnit Vintage [OK]
'-- JUnit Platform Suite [OK]

Test run finished after 107 ms
[         6 containers found      ]
[         0 containers skipped    ]
[         6 containers started    ]
[         0 containers aborted    ]
[         6 containers successful ]
[         0 containers failed     ]
[         4 tests found           ]
[         0 tests skipped         ]
[         4 tests started         ]
[         0 tests aborted         ]
[         4 tests successful      ]
[         0 tests failed          ]
```

コンパイル済みのテストコードとコンソールランチャーアプリケーションがあれば、テスト実行できます。テストログからも実装したテストメソッドがそれぞれ実行され、テスト結果が正常かを確認できます。テスト結果に異常が発生した場合は、テストログにテストに失敗した情報が出力されます。

コンソールランチャーからのテスト実行は、他にも実行方法があります。

```
$ java -cp 'classes:testlib/*' org.junit.platform.console.ConsoleLauncher execute --scan-class-path
```

直接ConsoleLauncherクラスを実行する方法でも、テスト実行が可能です。classesは、コンパイルされたJavaのクラスファイルが格納されたディレクトリを表します。testlibは、アーカイブファイル（.jar）が格納されたディレクトリを表します。このように、クラスパスとして指定するファイルをまとめて扱う場合に役立ちます。

また、--scan-class-pathオプションでディレクトリ配下のテストクラスをすべて実行しています。複数のテストクラスが存在し、特定のテストクラスのみ実行したい場合は、--select-classオプションを使います。本オプションの使用例を示します。

```
$ java -jar junit-platform-console-standalone-1.10.2.jar execute --class-path . --select-class
ProductServiceTest
```

その他にもさまざまなオプションが提供されています。詳細は公式ドキュメントを参照してください。

■ビルドツール上でのテスト実行

JUnitでは、主要なビルドツール(注9)がサポートされています。これらのビルドツールでテスト実行する場合は、下記の2ステップが必要です。

① 依存関係の追加
② テスト実行

依存関係の追加は、Mavenの場合はpom.xmlに記載し、Gradleの場合はbuild.gradleに記載します。ここでは、Mavenにおける依存関係の追加例を示します（**リスト8.27**）。

リスト8.27　依存関係の追加例

```
<!-- ...省略... -->
<dependencies>
    <!-- ...省略... -->
    <dependency>
        <groupId>org.junit.jupiter</groupId>
        <artifactId>junit-jupiter</artifactId>
        <version>5.10.2</version>
        <scope>test</scope>
    </dependency>
    <!-- ...省略... -->
</dependencies>
<build>
    <plugins>
        <plugin>
            <artifactId>maven-surefire-plugin</artifactId>
            <version>3.1.2</version>
        </plugin>
        <plugin>
            <artifactId>maven-failsafe-plugin</artifactId>
            <version>3.1.2</version>
        </plugin>
    </plugins>
</build>
<!-- ...省略... -->
```

依存関係にjunit-jupiterを記載しています。JUnitのアグリゲータを使って、まとめて機能を依存関係に指定できます（**表8.9**）。

表8.9　アグリゲータが含む機能

依存関係	用途	含まれるパッケージ
コンパイル依存	テストコードで使用したクラスやアノテーションの依存解決	junit-jupiter-api, junit-jupiter-params
ランタイム依存	テスト実行のためのテストエンジンの依存解決	junit-jupiter-engine

テスト実行には、Mavenの標準コマンドであるmvnを使います。ラッパーコマンドであるmvnwでも同様の操作です。サブコマンドであるtestを引数に指定することで、テスト実行のみ指定できます。テスト実行例を示します。

```
$ mvn test
...省略...
[INFO] -------------------------------------------------------
[INFO]  T E S T S
[INFO] -------------------------------------------------------
[INFO] Running com.example.ProductServiceTest
[INFO] Tests run: 4, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: xx.xx s -- in com.example.
ProductServiceTest
[INFO]
[INFO] Results:
[INFO]
[INFO] Tests run: 4, Failures: 0, Errors: 0, Skipped: 0
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  <テスト実行時間>
[INFO] Finished at: <テスト実行完了日時>
[INFO] ------------------------------------------------------------------------
```

（注9）　サポートされているビルドツールは、Maven、Gradle、Antです。

■IDE上でのテスト実行

主要なIDE(注10)は、JUnitの実行をサポートしています。

IDE上でテスト実行を可能にするためにIDEのプラグインが提供されています。プラグインをインストールすることで、GUI画面でテストコードを実行できます。

実装したテストメソッドやテストクラスのみの実行がしやすく、修正がしやすいことが利点です。

- **[IDE] テストの実装と部分的なテスト実行**
- **[ビルドツール] すべてのテスト実行と他ツールとの連携**

上記のように場面に応じて、テスト実行を行う環境を使い分けると良いです。

App.8-6 モック

「**App.8-3 テストコードの概要**」では、テスト対象別に基本的な操作を説明しました。基本知識を土台として、ここからはより実践的なテストコードの実装方法を説明します。特に複雑な依存関係を持つソフトウェアのテストにおいて、より洗練されたアプローチを取り入れています。その中心となるのがモック（Mock）です。

ソフトウェア開発において、外部システムや重たいリソース、未実装の機能に依存する部分のテストは、常に挑戦的です。これらの依存関係があると、テストの実行時間の増加や、テスト環境構築の難化、さらにはテストの再現性低下を引き起こしやすくなります。モックは、これら問題の軽減または解決に有効な手段です。

App.8-6-1 モックとは

Javaにおいて、モックとはテスト対象のコードが依存しているオブジェクトの代理オブジェクトです。モックを使用することで、外部システムやリソース、または未実装の機能を模倣します。これにより、テスト対象のコードが期待どおりに動作するか検証できます。

モックの活用により、下記の恩恵を得られます。

- **依存性の分離：テスト対象のコードから外部依存性を分離して、テストの信頼性と再現性を向上**
- **テストの高速化：外部システムへのアクセスの模倣により、テスト実行時間を大幅短縮**
- **柔軟な振る舞いの設定：モックに対して、メソッドの返り値や例外の発生などの任意の振る舞いを動的に設定**
- **振る舞いの検証：モックが期待どおりに呼び出されたかどうかを検証**
- **統合しやすさ：JUnitなどのテストフレームワークと統合可能**

（注10） サポートされているIDEは、IntelliJ IDEA、Eclipse、NetBeans、Visual Studio Codeです。

これらの特徴から、テスト実行時間の短縮や、テスト環境構築の単純化、テストの再現性の向上が期待できます。

■テスト実行時間の短縮

モックは、外部システムとの通信処理や、CPU等のリソースを消費する処理を模倣できます。外部システムとの通信処理には、外部のAPI通信やデータベース通信などが挙げられます。これらの処理は、通信周りの待ち時間を含む処理時間が発生します。

通信処理の模倣は、事前に定義された処理結果を返すため、通信の処理時間を短縮できます。また、リソース消費が大きい処理の模倣は、リソース管理やテスト環境の要件における観点より、処理時間を短縮できます。

モックにより、模倣の対象となる処理時間がなくなり、テスト実行時間がより短くなります。

■テスト環境構築の単純化

外部システムの模倣により、実際の外部システムを用意する必要がなくなります。たとえば、データベースへのアクセスが発生するテストでは、データベースサーバーを用意し、テストデータを登録する必要があります。モックを使えば、データベースアクセスのロジックを模倣し、テストデータをコード内で定義し利用できます。その結果、データベースサーバの構築やテストデータの登録作業が不要になり、テスト環境構築の作業がより単純化できます。

また、リソース消費が大きい処理の模倣により、求められるテスト環境のリソースもより小さくできます。各テストのリソース消費が大きい場合は、テスト環境全体のリソースが枯渇する可能性があります。リソースが枯渇すると、各テストの実行時間が長くなり、テスト全体の実行時間が増加します。そのため、テストで使用するリソースを小さくすることで、これらの問題を回避しやすくなります。

また、未実装の機能に依存するテストの場合、その機能のインタフェースのみを模倣し、テストを実施できます。これにより、開発プロセスが柔軟になり、機能の実装待ちによるテストの停滞や、機能の動作要件の準備がなくなります。

モックにより、テスト環境の構築が単純化され、テスト環境の準備における労力を小さくできます。

■テストの再現性の向上

外部システム、リソース、未実装の機能に依存するテストは、テスト環境の状況や外部要因により結果が変わる可能性があります。これらの依存要素をコントロール下に置くことで、一貫したテスト条件を作り出せます。一貫したテスト条件は、テストの再現性に直接的に寄与します。同じテストを繰り返し実行しても、常に同じ結果が取得できるためです。

テストの再現性が向上することで、問題の特定や修正が容易になり、開発プロセス全体の効率化にも寄与します。

モックにより、テストの信頼性を高め、より安定したソフトウェア開発を実現できます。

App.8-6-2　モックテスト向けフレームワーク

モックテストを行うためには、専用のフレームワークが有用です。モックテスト向けに開発されたフレームワークをモッキングフレームワークあるいは、モックフレームワークと呼びます。

Javaで広く使われるモッキングフレームワークは、Mockito(注11)です。本書で採用するMockitoのバージョンを示します(**表8.10**)。

表8.10　本書で採用するライブラリのバージョン

Mockito	5.11.0

Mockitoライブラリを使ってテストコードを実装します。Mockitoは、JUnitやAssertJと併用して使われます。JUnitやAssertJを使ったテストコードの説明と同様に、テストコードの実装や実行を説明します。

App.8-6-3　モッキングフレームワークの慣習

モッキングフレームワークを効果的に使うには、その慣習やベストプラクティスの理解が重要です。モッキングフレームワークの広く知られている慣習について説明します。

■モックの命名

モック名は、対象となるクラス名やインタフェース名の接頭辞または接尾辞に、mockまたはmockedを付けます。

(例) クラス名：Service、**モック名**：mockedService

本書では、接頭辞にmockedを付けた名前で説明しています。

■限定的な使用

すべての依存関係をモックに置き換えると、テストコードの可読性が悪化します。その結果、テストの目的が不明瞭になりやすく、意図しないバグの誘発や、メンテナンスの労力が肥大化します。テストの目的に直接関わる部分だけをモックにし、必要最小限に留めます。

たとえば、データベースの呼び出しをモックする場合、データアクセスレイヤーだけをモックにし、ビジネスロジックの層は実際のコードを使います。実装方法は、「スパイの利用」にて説明します。

(注11)　Mockitoは、サードパーティ製ライブラリです。

■モックのリセット

複数のテストケースで同じモックを使うことがあります。テストケース間で状態の干渉を避けるため、モックのリセットが推奨されます。リセット方法は、「**App.8-6-4　モックの構造**」の「モックのリセット」にて説明します。

App.8-6-4　モックの構造

モックの作成や定義をするユーティリティクラスが提供されています。このユーティリティクラスを使って、テストコードを作成します。

■Mockitoクラス

org.mockito.Mockitoクラスは、使用頻度が一番高いユーティリティクラスです。モックの作成からテスト対象オブジェクトとのやり取りの検証まで、幅広い役割を担います。

Mockitoクラスの代表的なクラスメソッドを示します(**表8.11**)。

表8.11　Mockitoクラスの代表的なクラスメソッド

メソッド名	意味
mock	指定のクラスまたはインタフェースのモックを作成
when	メソッドの戻りや例外を指定
verify	モックのメソッドが指定された回数呼ばれたか検証
times	メソッド呼び出しが指定の回数だけ発生したか検証
never	メソッド呼び出しが発生していないことを検証
atLeast	メソッド呼び出しの指定の回数以上発生したか指定
atMost	メソッド呼び出しの指定の回数以下発生したか指定
spy	実オブジェクトの一部メソッドのみモック化

■Mockitoクラスの使用例

テスト対象が依存する機能に合わせて、モックを作成します。モックを作成し、作成したモックに振る舞いの模倣を定義することで、テストの焦点を明確にします。

テスト対象が依存するクラスやインタフェースのモックの作成例を示します(**リスト8.29**)。

リスト8.29　Greetクラスまたは、インタフェースのモック作成例

```
// Greetは、挨拶文を生成するクラス
Greet mockedGreet = Mockito.mock(Greet.class);
```

作成したモックに、実際のオブジェクトの動作を模倣するように設定します。特定の条件下における返り値または例外を設定します。特定の条件下は、具体的な呼び出しメソッドとその引数を示します。

Mockitoクラスのwhenクラスメソッドで特定の条件を設定します。whenクラスメソッドは、モックを返り値とします。そのモックのthenReturnメソッドで返り値、thenThrowメソッドで

例外を設定できます。

モックに模倣した動作を設定する例を示します（**リスト8.30**）。

リスト8.30　モックの模倣した動作の設定例

```
// Greetクラスのhelloメソッドの模倣

// 引数に"User"文字列が指定された場合は、"Hello User!"を返り値とする
Mockito.when(mockedGreet.hello("User")).thenReturn("Hello User!");

// 引数に空文字列が指定された場合は、IllegalArgumentException例外を返り値とする
Mockito.when(mockedGreet.hello(null)).thenThrow(new IllegalArgumentException());
```

Mockitoクラスは、引数の指定に役立つクラスメソッドも提供しています。このクラスメソッドを引数マッチャと呼びます。代表的な引数マッチャを示します（**表8.12**）。

表8.12　代表的な引数マッチャ

メソッド名	説明
any	任意の型かつ任意の値
anyBoolean	任意のbooleanまたは、null値を除く任意のBoolean
anyByte	任意のbyteまたは、null値を除く任意のByte
anyChar	任意のcharまたは、null値を除く任意のChar
anyInt	任意のintまたは、null値を除く任意のInt
anyDouble	任意のdoubleまたは、null値を除く任意のDouble
anyFloat	任意のfloatまたは、null値を除く任意のFloat
anyLong	任意のlongまたは、null値を除く任意のLong
anyShort	任意のshortまたは、null値を除く任意のShort
anyString	null値を除く任意のString
anyIterable	null値を除く任意のIterable
anyList	null値を除く任意のList
anyCollection	null値を除く任意のCollection
anyMap	null値を除く任意のMap
anySet	null値を除く任意のSet
isNull	null値
notNull	null値でない任意の値
isA	引数に指定されたクラスのオブジェクト
eq	引数に指定された値と等しい値
startsWith	引数に指定されたプレフィックスで始まるString
endsWith	引数に指定されたサフィックスで終わるString

引数マッチャを使用することで、モックの振る舞いを柔軟に設定できます。引数マッチャの使用例を示します（**リスト8.31**）。

リスト8.31　引数マッチャの使用例

```
// 引数が任意の文字列の場合は、"Hello Test User!"を返す
Mockito.when(mockedGreet.hello(Mockito.anyString())).thenReturn("Hello Test User!");
// 引数がnullの場合は、IllegalArgumentException例外を投げる
Mockito.when(mockedGreet.hello(Mockito.isNull())).thenThrow(new IllegalArgumentException());
```


■メソッド呼び出しの検証

テスト対象が適切に外部依存の操作を呼び出しているかは、重要なテスト観点です。

Mockitoクラスのverifyクラスメソッドで、モックのメソッド呼び出しを検証します。検証対象となるモックのメソッドと引数を定義し、モックのどのメソッドがどのような引数で呼び出されたか正確に検証できます。

モックのメソッド呼び出しの検証例を示します（**リスト8.32**）。

リスト8.32　メソッド呼び出しの検証例

```
// mockedGreetはGreetクラスのモック
// mockedGreet.helloメソッド呼び出しの検証
Mockito.verify(mockedGreet).hello(Mockito.anyString());
```

Greetクラスのモックにおいて、helloメソッドが引数に任意の文字列を指定されて呼び出されたかを検証しています。検証対象となるメソッドと引数は、メソッドチェーンにて指定できます。

また、メソッド呼び出しの回数を指定できるクラスメソッドを使って、メソッド呼び出しの回数が想定どおりか検証できます。

リスト8.32では、メソッド呼び出しの回数を指定していません。メソッド呼び出しが1回のみとデフォルトで指定されているため、記述を省略しています。

リスト8.32でのメソッド呼び出しの検証例に呼び出し回数も考慮した検証例を示します（**リスト8.33**）。

リスト8.33　メソッド呼び出し回数を考慮した検証例

```
// mockedGreet.helloメソッドの呼び出し回数を検証します。

// 呼び出しが発生していない。呼び出し回数が0回
Mockito.verify(mockedGreet, Mockito.never()).hello(Mockito.anyString());

// 呼び出し回数が2回のみ
Mockito.verify(mockedGreet, Mockito.times(2)).hello(Mockito.anyString());

// 呼び出し回数が1回以上
Mockito.verify(mockedGreet, Mockito.atLeastOnce()).hello(Mockito.anyString());

// 呼び出し回数が3回以下
Mockito.verify(mockedGreet, Mockito.atMost(3)).hello(Mockito.anyString());
```

■スパイの利用

スパイは、モックとは異なり、一部の実装を保持したオブジェクトです。実際のメソッド呼び出しを許可し、一部の振る舞いを変更できます。

Mockitoクラスのspyクラスメソッドを使って、既存のオブジェクトをスパイ化できます。スパイ化されたオブジェクトは、実際のメソッド呼び出しを許可し、一部のメソッドの振る舞い

を変更できます。スパイの利用は、一部のメソッドのみモックとして扱う場合に適しています。

スパイの利用例を示します（**リスト8.34**）。

リスト8.34　スパイの利用例

```
// Greet.helloメソッドは既に実装済みとします。

// スパイの作成
Greet greet = new Greet();
Greet spiedGreet = Mockito.spy(greet);

// 一部のメソッドの振る舞いを変更
Mockito.when(spiedGreet.hello("Anonymous")).thenReturn("Who are you?");

// メソッド呼び出し
spiedGreet.hello("Test User");  // => "Hello Test User!"
spiedGreet.hello("Anonymous");  // => "Who are you?"

// メソッド呼び出しを検証
Mockito.verify(spiedGreet).hello("Test User");
Mockito.verify(spiedGreet).hello("Anonymous");
```

多くの場合、スパイのオブジェクト名は、対象クラスの接頭辞または接尾辞に、spyまたはspiedを付与します。モックと同様の命名慣習です。

Greetオブジェクトのhelloメソッドを呼び出す際にスパイ化の特徴が表れています。引数が"Test User"の場合は、既に実装されているhelloメソッドを呼び出します。引数が"Anonymous"の場合は、スパイにより設定された"Who are you?"を返り値とします。

このように、モックと違い、既に実装されたメソッドをそのまま使用できます。上記の例では、同一のメソッドに対する設定です。現場では、特定のメソッドをモック化し、その他のメソッドを既存のまま活かします。

WebAPIと通信し取得したデータを加工する処理があり、WebAPIとの通信を担うメソッド、データ加工を担うメソッドで構成したと仮定します。前者のメソッドをスパイにより、処理をモック化します。後者メソッドをそのまま使います。そうすることで、テスト対象を後者メソッドに絞り込むことができます。WebAPIの都合を考慮せずにテストを行うことが可能です。

このように、外部依存する処理のみモック化したい場合、スパイの活用は有効です。

■モックのリセット

モックを初期状態に戻すことで、テストケース間での影響を避けることができます。

Mockitoクラスのresetクラスメソッドを使用することで、モックをリセットできます。リセット後は、モックの振る舞いが初期状態に戻ります。

モックのリセット例を示します（**リスト8.35**）。

リスト8.35　モックのリセット例

```
// 2回メソッドを呼ぶ
mockedGreet.hello("Test User");
mockedGreet.hello("Anonymous");

// モックのリセット
Mockito.reset(mockedGreet);

// 1回もメソッドが呼び出されていないことを検証
Mockito.verify(mockedGreet, Mockito.never()).hello(Mockito.anyString());
```

リセット処理実行後は、モックのメソッド呼び出しが0に初期化されます。また、モックのみならずスパイも同様です。

■InOrderクラス

org.mockito.InOrderクラスは、メソッド呼び出しの順序検証に役立つクラスです。

メソッドの呼び出し順序が適切に実行されているか検証するテストはよくあります。たとえば、デッドロックが発生する可能性のある処理では、メソッドの呼び出し順序は重要です。

InOrderクラスのverifyクラスメソッドを使って、モックに対してメソッド呼び出しの順序を検証できます。メソッド呼び出しの順序検証の例を示します（**リスト8.36**）。

リスト8.36　メソッド呼び出しの順序検証の例

```
// モックの作成
List<String> mockedList = mock(List.class);
List<String> mockedList2 = mock(List.class);

// モックのメソッド呼び出し
mockedList.add("called at 1st");
mockedList2.add("called at 2nd");
mockedList.add("called at 3rd");

// 順序検証オブジェクトの作成
InOrder inOrder = inOrder(mockedList, mockedList2);

// 順序検証
inOrder.verify(mockedList).add("called at 1st");
inOrder.verify(mockedList2).add("called at 2nd");
inOrder.verify(mockedList).add("called at 3rd");
```

順序検証の処理は、上から順にメソッドが呼び出されたかを検証します。順序に誤りがある場合は、検知次第アサーションが発生します。