Java コーディング（文と式、ブロック）

内容

Copyright(C)2021 Future Office Coordinate Service Corporation All Rights Reserved.

Java コーディング（文と式、ブロック）

はじめに

未経験者に対する Java 言語研修ではしばしば市販の独習用書籍を併用しますが、未経験者がこれをいくら読んでも「...で？」という感想を持つのが関の山です。初めて英語の勉強を始めた時に文法や単語の活用について「ナニそれ」状態になるのと同様です。英語の場合は熟語や慣用句・言い回しに慣れてくると表現方法が分かってきます。Java 言語にも自然言語と同じく慣用的なコードがあります。慣用コードは Java の場合 文と式を使って表現するので、まずは文や式とはなにかというところを説明します。また、Java 関連の試験用以外では覚える必要もないと思いますが、言語仕様が多少は読み易くなるように「文」と「式」についての形式的な説明も加えながら進めます。

1. 文と式

Java SE 11 の言語仕様の 14 章[1]には以下のように記述されています。
『プログラムの実行順序は、ステートメント（Statements：以下、文）によって制御されます。文は、その効果のために実行され、値はありません。*--中略--* 一部の文には、構造の一部として式（Expression）が含まれています。』…他の箇所の記述からも肉付けすると...

「文」はブロック(Blocks“{}”)内の処理を制御（分岐、繰返し、例外スロー・キャッチ）し、「式」はメソッド呼び出しや演算子(Operator)の実行結果として値に置き換わり別の式のオペランドになります。また、式はセミコロン「;」で完結すると文の扱いになります。

2. 文の種類（JDK 11）

文はブロックの中に書くことができ、以下の種類があります（一般的な記述形式だけ記載）。

＜凡例＞ 処理：ブロックまたは式、[ ]：省略可能、
{変数修飾}：任意 … @NonNull,@ReadOnly 等コンパイラ向け注釈または final

- ；//空文「;」だけの文はなにもしない
- 式；//「;」で終わらせた式＝式文
- if (式) { 式が真の時の処理 } else { 式が偽の時の処理 } //処理が１文の場合は”{}”不要
- assert 式 ；
- switch (式) {

```
        case 定数式 1: 処理 1;
                    break; //switch 文を抜ける…break しない場合、後続 case に入る
        case 定数式 2: 処理 2;
                    break;
        case 定数式 n: 処理 n;
                    break;
        default: //定数式１～定数式ｎの間に該当がなかったときの処理
    }
```

[1] https://docs.oracle.com/javase/specs/jls/se11/html/jls-14.html

Copyright(C)2021 Future Office Coordinate Service Corporation All Rights Reserved.

・while（ 式 ）{ 処理 } //式が true の間 処理を繰り返す…式に true と書くと永久ループ
・do { 処理 } while（ 式 ）; //処理を実行してから式を判定 …一度は必ず処理を実行する
・for（ [初期化処理] ; [繰返し条件式] ; [ループ毎の更新] ）{ 処理 }
・for（ {変数修飾} 型 変数 : 式 ）{ 処理 } //式の値は配列か Iterable 実装（List 等）
・break [ラベル] ; //ラベルで識別される（なければ近接の）switch、while、do、for を抜ける
・continue[ラベル]; while、do、for の次の繰り返しに移る
・return [式] ;
・throw 式 ;
・synchronized（ 式 ）{ 処理 }
・try { 処理 } [catch（例外型){ 例外時処理 } ...] Finally { 最終処理 }
・try（{変数修飾} 型 リソース変数 = 式 ...）{ 処理 } //式が表すリソースを自動で閉じる

3. 式の値と式文

式は実行することで結果（値）に置き換わります。結果の値は優先順に基づき式の右または左に隣接する演算子に渡され、代入演算子(“=”)があれば左辺に値が設定されて左辺が代入式の値になります[2]。但し、式の実行中に例外(Exceptions)が発生するか void を返すメソッドの呼び出し式では値が存在しません。

void を返すメソッド呼び出し式は値が存在しないため、メソッド呼び出しの終端に「;」を付けて単独の式文として実行しなければなりません（代入を含め演算に使うとエラーになります）。

式文の結果の最後の値は捨てられます（式文内で実行した代入式の結果は保存されます）。

＜式の実行と結果 例＞

・代入演算子[3]：代入演算子は右結合（同一演算子は右から処理していく）のため、a=b=c は a=(b=c) となり、これは c の値を b に割り当て、この式の結果である変数 b の値を a に割り当てます。結果として a, b, c は同値になります。

・等価[4]演算子：等価演算子は左結合（左から処理していく）のため、a==b==c は（a==b)==c となり、a==b の結果で得られる真偽値（boolean 型）と c を比較することになって a, b, c の同値性の確認にはなりません。

・void を返すメソッド呼び出し：System.out.println("これは式文です。");

[2] 代入も式なので値を持ち、代入式の結果は代入後の左辺変数の値になります
[3] 右記 12 種類：= *= /= %= += -= <<= >>= >>>= &= ^= |=
[4] “Equality Operators”…等式演算子、等値演算子と訳している資料も見受けられます

Copyright(C)2021 Future Office Coordinate Service Corporation All Rights Reserved.

4. 式の種類と互換性

式には以下の種類があり、各式は結果の値に互換性がある者同士で置き換えが可能です。

4.1. 式の種類

（1）式名(Expression Names)
変数、引数
※本資料では Java 言語仕様に記載の「Expression」を「式」と訳していますが、Expression は「表現」と訳される場合や、米語辞典によれば numbers, symbols という意味を含みます

（2）一次式
リテラル、オブジェクト作成（new 演算子)、フィールドアクセス、メソッド呼び出し、メソッド参照[5]、配列アクセスなど、他のすべての式を構成する最も単純な種類の式のほとんどが含まれます。括弧で囲まれた式も、構文的に一次式として扱われます。

（3）単項演算子式
演算子 +、 -、++、--、~、!、およびキャスト演算子
例＞ ++a a++ (int)1234567890123456789L

（4）二項演算子式
加減乗除剰余演算子：+、-、*、/、% 例＞ a + b
シフト演算子：<<、>>、>>> 例＞ n >> s n を符号拡張 s ビット右シフト ☞ $n/2^s$ と同
関係演算子（比較演算子）：<、<=、>、>=、instanceof、==、!=、&、^、|、&&、||
単純代入演算子：=
複合代入演算子：*=、/=、%=、+=、-=、<<=、>>=、>>>=、&=、^=、|=

（5）三項演算子式
例＞ a ? b : c
a は真偽値 or 真偽値の結果がでる式で、a が真のとき b 偽のとき c がこの式の結果となる

（6）ラムダ式[6]
例＞ (x, y) -> x+y
x,y のパラメータ（例は型を推測）を受けて x+y の演算結果を式の結果とする
… 上記の式は、(int x, int y) -> { return x+y; }の省略形です
JDK1.8 以前の表現では、以下になります。

```
int func(int x, int y){ return x+y; }
int a = func(b, c);
```

---

[5] JDK1.8 で導入された構文で、anyMethod(System.out*::println*)のように目的のメソッドを :: で接続することでメソッド自体を引数で渡すことができます。

[6] JDK1.8 で導入された関数型インターフェース（この例は JDK 組込みの IntBinaryOperator）で、メソッドを一つだけ持つインスタンスを生成します。裏で↓のように展開されます

```
IntBinaryOperator bo = (int x, int y) -> {return x+y;};
bo.applyAsint(m, n);
```

Copyright(C)2021 Future Office Coordinate Service Corporation All Rights Reserved.

4.2. 式の型と互換性

式（の結果）は「型」を持ちます。例えば、a==b は boolean 型です。
以下の式名 a と二項演算子式 a==b は、boolean リテラル true/false と型が同一で互換です。

```
boolean a = true;
if (a) {System.out.println(a);}
----------
int a = 0,b = 0;
if (a==b) { System.out.println(a==b);}
```

4.3. 型の互換性とキャスト

型にはプリミティブ型と参照型があり、変数の記憶形態が異なるため双方に互換性はありません。

（1）プリミティブ型

プリミティブ型には数値型と boolean 型があり、型毎に決まった長さの記憶域を占めます。
数値型は整数型と浮動小数点型です。

・整数型は byte（8 ビット）、short（16 ビット）、int（32 ビット）、及び long（64 ビット）の符号付き 2 の補数の整数、char は UTF-16 コードを表す 16 ビットの符号なし整数です。
【最小／最大値：Long.MIN_VALUE= -9223372036854775808, Long.MAX_VALUE=9223372036854775807】

・浮動小数点型は float(32 ビットの IEEE 754 浮動小数点数を含)、及び double(64 ビットの IEEE 754 浮動小数点数)。

・boolean 型：true か false の値を持つ（ビット数は言語仕様、JVM 仕様に規定なし→実装次第）

※数値型に含まれる型は相互に変換（キャスト演算）が可能ですが、boolean 型に（から）変換できません。また、数値型に対しては単項演算、二項演算が可能です。

（2）参照型

変数はポインタ[7]を記憶し、変数に格納されたポインタを通してオブジェクトを参照します。
このオブジェクトとは、インスタンス（new 演算子でヒープに展開されたクラス）と配列です。

※インスタンスは継承関係かインターフェースとして実装している型に対してキャストによる型変換が可能です。スーパークラスや実装済インターフェースへの（拡大）型変換ではキャスト演算子も不要です。また、単項演算や二項演算はできません（String は除く）。

例：HashMap は Map インタフェースを実装しているので以下の変換が可能です

```
Map map = new HashMap();          //キャスト演算なしで Map 型に代入可能
HashMap hmap = (HashMap)map;      //サブクラスへはキャストにより型変換ができる
```

（3）String 型

String は参照型ですが取扱いが特殊で、変数の初期値として指定された文字列はコンパイル時にコンスタントプールに格納され、同一の文字列は一つだけ保存されて共用されます。
String は加算演算の対象にすることができ、文字列連結が行われます。

---

[7] C 言語などのアドレスを特定する演算可能なポインタとは異なります

Copyright(C)2021 Future Office Coordinate Service Corporation All Rights Reserved.

5. 式の実行順

式は演算子の優先順と評価(Evaluation)順により処理を進めます。

5.1. 演算子の優先順と評価順

最も優先度の低い演算子は、ラムダ式（->）の矢印であり、その後に代入演算子が続きます。
したがって、すべての式は構文的にラムダ式 および代入演算式に含まれます。
∴文を構成する一連の式の結果は最終的に式文で捨てられるか、変数に代入されるか、メソッド呼び出しの引数に使われるか、return 文の戻り値になります

5.2. オペランドの評価順

二項演算子のオペランドは左から右に評価（式の値として取り出す)し、複合代入演算子の場合は同時に計算前の値の保存も行います。

＜例１：二項演算子＞

```
int i = 2;
int j③ = (i = 3)① * i②;
System.out.println (j) ;
---------------------------
```

① 二項演算子＊の左オペランド一次式()内が実行されｉが３に、式の値も３になります（代入式の結果は代入後の変数の値なので）
② ①の式の実行でｉが３と評価され、3×3 が実行されます
③ 最も優先度が低い代入演算子=によりｊに９が代入されます
従って、9 を出力します。

＜例２：複合代入演算子＞

```
int a = 9;
a① +=③ (a = 3)②; //最初の例
System.out.println (a) ;
int b = 9;
b③ = b① + (b = 3)②; // 2 番目の例
System.out.println (b) ;
---------------------------
```

例 1、例２はオペランドの評価順を説明するために言語仕様書から一部を転記して説明を加えたものですが、式の結果が明解ではないため使うべきではありません。特に…
- 一つの変数の値が式文中で複数回変化する
- 代入演算子が複数回出現している

※コードは共有物として誤解の入り込まない表現〔()を使って手順を明示する等〕にしてください

① 左端のａが評価（取り出し）され、９が保存されます
② 代入演算子+= の右一次式()の中が実行され、ａが３に置き換わります
③ 代入演算子+=が実行され、ａ（①で９が取り出されている）に３が加算されます
従って、最初の例は 12 を出力します
① 二項演算子+ 左オペランドｂが評価（取り出し）され、９が保存されます
② 右オペランドの一次式()が評価され、結果が３になります
③ 最初の代入演算子+ により①+②の結果が再左端のオペランドｂに設定されます
従って、２番目の例も 12 を出力します

Copyright(C)2021 Future Office Coordinate Service Corporation All Rights Reserved.

5.3. 単項演算子（++、--）

項（変数）の前または後ろに ++ を付けるとインクリメント（１加算)、-- を付けるとデクリメント（１減算）になります。他の式と混合して使った場合に、加減記号を後ろに付けるか前に付けるかで評価のタイミングが異なります。

＜例１：後置＞

```
int a,b;a=b=0;
b = 100 +① a++②;
System.out.println(b);
---------------------------
```

① 二項演算子の右オペランドである a が評価され(0)、左辺 100 に加算後 b に代入されます

② 二項演算が終了した後 a に 1 加算されます

従って、100 を出力します。

【注】 a++② +①100 としても、式の結果は 100 になります … ① a+100 ② a+1 の評価順です

＜例２：前置＞

```
int a,b;a=b=0;;
b = 100 +② ++a①;
System.out.println(b);
---------------------------
```

① 二項演算子の右オペランドである++a が実行され a に 1 加算されます

② 左辺 100 に①の a が加算されて b に代入されます

従って、101 を出力します。

5.4. 三項演算子

三項演算子は条件により二値から一つ選択（評価）し、他の式に埋め込むことができます。

＜例＞

```
String species = "bird";
System.out.println(species + "の足の数は,"+ ( species.equals("bird")? 2:4 ) +"本");
---------------------------
```

この実行結果は「bird の足の数は,2 本」と出力します。

Copyright(C)2021 Future Office Coordinate Service Corporation All Rights Reserved.

6. ブロック

式や文はブロックの中に書きます。ブロックは書く場所と修飾子によって種類があり、動作時期が異なります。

6.1. ブロックが書ける場所

Java のクラスの中で処理（ブロック）は以下の場所に書くことができます。

① 静的初期化子(Static Initializers)

クラス宣言内に書かれ、直前に static と書かれているブロックで、クラスの初期化（インスタンス初期化子の実行前）時に 1 回だけ実行されます。this, super 等のインスタンス参照や、インスタンス変数へのアクセスはできません。

② インスタンス初期化子(Instance Initializers)

クラス宣言内に書かれた静的初期化子でないブロックで、インスタンス化の都度実行されます。コンストラクターが選択される前に実行されます。

③ コンストラクター

インタンスの初期化を行います。以下の点がインスタンス初期化子と異なります。

・アクセス修飾子と引数を持ち、複数のコンストラクターがある場合は一つが選択される

・実行後に new 演算子の戻り値としてインスタンスへの参照を返す

④ メソッド

戻り値、メソッド名、引数、例外と共に宣言し、自分以外のクラス（アクセス修飾子が許していれば）と、自クラスのメソッドから呼び出されて結果を返します。返す値が無い場合は戻り値に void と宣言します。

6.2. 処理の記述

アプリケーションの主な処理はメソッドで行い、それ以外のブロックはメソッドが動作する初期環境を用意するために使います。コンストラクターとメソッドは外部から呼び出せますが、アクセス修飾子が private だと自クラス以外からは呼び出すことができません。

外部に見せる必要がないコンストラクター、メソッドは private にします。

Copyright(C)2021 Future Office Coordinate Service Corporation All Rights Reserved.

7. メソッド呼び出し式

メソッドを呼び出す方法は対象のメソッドが static か否かにより形式が異なり、アクセス修飾子によりアクセス可能か否かが決まります(public：公開、protected：同一パッケージまたは継承クラス、private：同一クラス内からのみ、指定なし：同一パッケージのみ-但しインタフェースは公開)。

7.1. メソッド呼び出しの形式

コンパイラーはメソッド呼び出し式が書ける場所で左括弧「(」をみつけるとそのすぐ左側の識別子から名前の種類を判定します。識別子がセパレータ「.」(ドット)を含んでいると、識別子がパッケージ名かクラス名、フィールドアクセス式[8]、または参照型変数です。

左括弧のすぐ左にあるのがメソッド名で、メソッド名のすぐ左に識別子があればそれはクラス名か static 付きのフィールド、または参照型の変数と判定し、クラス名の左に更に識別子があればパッケージ名[9]です。メソッド呼び出し式がエラー無く実行されるとその部分の式は「式の結果」に置き換わり、式の解析が続きます。形式によっては実行するまで型が決まらないので、式の左端から実行を始め式の最後のメソッドの呼び出しに至る前に実行時エラーが発生する場合があります。

メソッド呼び出し式の形式には以下の様式とその組み合わせがあります。

（1）static なし

static 付きでないメソッドは、メソッド名の左に null でない参照型の変数または super、this を指定します。また、自クラス内に宣言がるメソッドであればメソッド名だけで呼び出せます。

（2）static 付き

クラスのメンバー（フィールド、メソッド）は、[パッケージ名 .]クラス名 . 識別子で呼び出すことができます。メソッド呼び出し式の結果はメソッドの戻り値となります。

＜様式 1＞ パッケージ名 . クラス名 . メソッド名()

＜例＞ java.util.stream.Collectors.toList()①

・パッケージ名：java.util.stream

・クラス名：Collectors

・メソッド名：toList()

① Collectors クラスが持つ public **static** <T> Collector<T,?,List<T>> **toList**()を呼び出します

※ パッケージ名は import java.util.stream.Collectors; のように import 文を書けば省略可能です

---

[8] https://docs.oracle.com/javase/specs/jls/se11/html/jls-15.html#jls-15.11

[9] https://docs.oracle.com/javase/specs/jls/se11/html/jls-6.html#jls-6.5.1
https://docs.oracle.com/javase/specs/jls/se11/html/jls-15.html#jls-15.12
参考＞パッケージ名も pacage1.package2.package3 のように任意の数のセパレータ「.」を含む場合があります。このパッケージ名の連結はクラスファイルが格納されているディレクトリ階層に一致し、.jar ファイルの名前にもなります。クラスパス(classpass)にはこの階層の最上位ディレクトリか jar ファイル自体を指定します

Copyright(C)2021 Future Office Coordinate Service Corporation All Rights Reserved.

＜様式 2＞ クラス名 .[フィールドアクセス式 …] . メソッド名()

＜例＞ System.out①.println()②;

・クラス名：System

・フィールド名：out

・メソッド名：println()

① System クラスに public **static** final *PrintStream* **out** という *PrintStream* 型のフィールド宣言があり、System.out は準備済の PrintStream の参照を格納したクラス変数です

② *PrintStream* クラスには public void **println**() というメソッドが宣言されています

＜様式 3＞ クラス名 .[メソッドアクセス式 …] . メソッド名()

＜例＞ Calendar.getInstance()①.getTime()②;

・クラス名：Calendar

・メソッド名：getInstance()

① Calendar に public **static** Calendar **getInstance**() メソッドがあり、現在の日付と時刻で初期化した *Calendar* オブジェクトを返します

② Calendar クラスには public final Date **getTime**() というインスタンス・メソッドも宣言されており、①が返した *Calendar* オブジェクトのメソッドを呼び出します

7.2. メソッド呼び出し式の型

メソッド呼び出しも式なので、結果は値に置き換わります。メソッド呼び出しの結果はメソッド内で return 文とともに指定した値で、型はメソッド宣言で指定した戻り値の型です。

引数の型と戻り値の型は受け取る側と一致しているか、拡大変換が可能な場合のみ成立します。

【メソッド宣言の一般形式】

[アクセス修飾子] 戻り値の型 メソッド名(引数１型 引数１, 引数２型 引数２, …) { 処理 }

※メソッド宣言と呼び出し式は、シグネチャ（メソッド名と引数）と戻り値、例外が一致しているか必要があります（引数、戻り値に関しては拡大変換で受け渡し可能）

（１）プリミティブ型の受け渡し

＜例＞

〔メソッド宣言〕

```
private int sampleMethod(short x, short y) {
    return x * y ;
}
```

〔メソッド呼び出し〕

```
short a = sampleMethod(10, 100); //引数の 10 が int のため、エラーになります
short a = sampleMethod((short)10, (short)100); //戻り値の型が int でエラーになります
short a = (short)sampleMethod((short)10, (short)100); //受取の型にキャストして成功！
long a = sampleMethod((short)10, (short)100); //これも拡大変換で成功
```

Copyright(C)2021 Future Office Coordinate Service Corporation All Rights Reserved.

（２）参照型の受け渡し

＜例＞

〔メソッド宣言〕

```
private List sampleMethod2(Map<String, List> x) {
    return x.get("key") ;
}
```

〔メソッド呼び出し〕

```
List l = new ArrayList<>(Arrays.asList("a", "b"));
Map<String, List> m = new HashMap<String, List>();
m.put("key", l);
ArrayList m1 = sampleMethod2(m); //これは戻り値が List⇒ArrayList になるため失敗
List m2 = sampleMethod2(m); //これは成功
ArrayList m1 = (ArrayList)sampleMethod2(m); //キャストしてこれも成功
```

7.3. メソッド引数の保護

メソッドを切り出すときにはいくつか注意が必要です。

（１）メソッドで扱える変数・引数

メソッドの中ではフィールド（クラス変数、インスタンス変数）、メソッド内で宣言した変数（ローカル変数）及びメソッドの引数 を式名として操作することができます。

（２）変数や引数の更新

引数はスタックを経由してメソッドに渡されるため引数自体は変更できません。但し、参照型の場合は引数で受け取った参照を使って参照先のオブジェクトを更新することができます。これは修飾子に final を付けた変数でも同様で、変数の参照先を別のオブジェクトに変えることはできませんが、参照先のオブジェクト自体を更新することは可能です。

参照先オブジェクトの更新に関わる危険は、メソッドから戻り値として返したフィールドに関しても同様です。メソッド呼び出し式で取得した戻り値を変更すると同様の問題が起こります。

（３）変数や引数の意図しない更新の抑止

意図しない変数や引数(にしたオブジェクト)の更新を抑止する方法には以下のものがあります。

① 引数や戻り値とするオブジェクトはディープコピーして渡す
（apache.commons.lang.SerializationUtils.clone()等で受け渡し用のオブジェクトを作る）

② プリミティブ型かそのラッパークラス、String 等の内容が変えられない型を使う

③ オブジェクトに更新系のメソッドを抑止したラッパークラスを被せて渡す

※①、③に関してはクリティカルな業務やユーティリティクラスに対象を絞らないと、手間がかかる上に性能に影響する可能性があります。適切なクラス設計によりメソッド間、クラス間の共有情報を減らす方が有効です

Copyright(C)2021 Future Office Coordinate Service Corporation All Rights Reserved.

7.4. メソッドの複数の処理結果

Java の構文上、戻り値は一つだけ（または、戻り値無し）です。例えば、エラーチェックの処理を作るときにエラーコードと詳細メッセージの二つを返すことはできません。

このような場合には以下の方法があります。

（1）引数の参照先を更新する

前項の 7.3 メソッド で「引数の参照先が更新できる」注意点を利用した方法です。

```
StringBuilder msg = new StringBuilder();
if (validateTrdate(item.getTrdate(), msg)) ++errcnt;//true が帰ってきたらエラー有

//日付チェックメソッド
private boolean validateTrdate(String testDate, StringBuilder msg) {
    if (testDate == null) {
        msg.append("1-1:取引日付は入力必須です"); //引数にエラーの文言を追加
        return true;                              //戻り値にはエラー = true を返す
    }
    return false;
}
```

（2）情報を保持するクラスを作成して渡す

オブジェクトとして呼び出し元に返す。

【注意】

下記の例はエラー情報用に内部クラスを作成して利用しています。static 修飾子無しの内部クラスの利用は外側クラスが GC から外れてしまう可能性からコードチェッカーで警告になる可能性があります。

下記の例では、ParaErr が外側の SampleClass への参照を持っているため ParaErr か ParaErr の参照を持っている errors を SampleClass の外に渡す(return)と SampleClass のインスタンスが ParaErr の寿命に依存するようになります。受け渡し範囲に注意してください。

```
public class SampleClass{
    public List someMethod(Map<String, Object> reqMap) {
        List<ParaErr> errors = new ArrayList<ParaErr> ();
        int errcnt = 0;
        for (Map.Entry<String, Object> reqEnt : reqMap.entrySet()) {;
            if (reqEnt.getValue() == null){
                ParaErr pe = new ParaErr();
                pe.setName(reqEnt.getKey());
                pe.setMsg(reqEnt.getKey()+"は入力必須です");
                errors.add(pe);
                errcnt++;
            }
        }
        return errors;
    }
    /**
     * パラメータのエラー情報を持つクラスです
     *
     */
```

Copyright(C)2021 Future Office Coordinate Service Corporation All Rights Reserved.

```
    class ParaErr {      //情報保持用の内部クラス 宣言
        private String name;
        private String msg;
        void setName(String name) {
            this.name = name;
        }
        void setMsg(String msg) {
            this.msg = msg;
        }
        String getName() {
            return name;
        }
        String getMsg() {
            return msg;
        }
    }
}
```

Copyright(C)2021 Future Office Coordinate Service Corporation All Rights Reserved.

7.5. 再帰処理の例

階層、特に配列の要素に配列が含まれる等の再帰的なデータ構造を処理する場合は処理も再帰的になります。下記の例は配列を含む Json 形式のデータを処理する例です。ラムダ式の中から再帰していますが、ラムダ式については別項で説明します。

```
import javax.json.JsonArrayBuilder;
import javax.json.JsonNumber;
import javax.json.JsonObject;
import javax.json.JsonObjectBuilder;
import javax.json.JsonReader;
import javax.json.JsonString;
import javax.json.JsonValue;
    JsonObject reqJson = req2Json(request);
    Map<String, Object> reqMap = json2Map(reqJson);

/**
 * JsonValue のデータ取り出し
 * JsonObject を受け取り、Json の配列／階層 ⇒ Map の List／階層に変換する
 * @param value
 * @return
 */
public Map<String, Object> json2Map(final JsonObject jo) {
    Map<String, Object> map = new HashMap<String, Object>();
    Set<Entry<String, JsonValue>> sets = jo.entrySet();
    for (Entry<String, JsonValue> set: sets) {
        JsonValue value = set.getValue();
        switch (value.getValueType()) {
        case ARRAY:
            map.put(set.getKey(), value.asJsonArray().stream()
                    .map(p->json2Map((JsonObject)p))
                    .collect(java.util.stream.Collectors.toList()));
            break;
        case STRING:
            map.put(set.getKey(), JsonString.class.cast(value).getString());
            break;
        case TRUE:
            map.put(set.getKey(), true);
            break;
        case FALSE:
            map.put(set.getKey(), false);
            break;
        case NUMBER:
            map.put(set.getKey(), JsonNumber.class.cast(value).numberValue());//型は Long...
            break;
        case NULL:
            map.put(set.getKey(), null);
            break;
        case OBJECT:
            map.put(set.getKey(), (JsonObject)value);
            break;
        default:
            break;
        }
    }
    return map;
}
```

Copyright(C)2021 Future Office Coordinate Service Corporation All Rights Reserved.

【Json[10]のデータ例】

項目は「フィールド名：値」で表しカンマ「,」で区切る。階層や配列の形式を持つことができます。

凡例 { } オブジェクト-項目の値は中括弧{}で囲むことで配列や子項目を包含できる

[ ] 配列- 配列の要素はカンマで区切る。要素としてオブジェクトを持つことができる

```
{
  ”tekiyou”:”摘要”,
  ”trDate”:”2021/03/19”,
  ”tantouCd”:”1010”,
  ”kariMeisai”:[
                 {
                  ”kariKamokuCd”:”210100”,
                  ”kariKingaku”:”10000000”
                 }
                 {
                  ”kariKamokuCd”:”210300”,
                  ”kariKingaku”:”50000000”
                 }
                 {
                  ”kariKamokuCd”:”210700”
                  ,”kariKingaku”:”30000000”
                 }
               ],
  ”kasiMeisai”:[
                 {
                  ”kasiKamokuCd”:”110100”,
                  ”kasiKingaku”:”90000000”
                 },
                 {
                  ”kasiKamokuCd”:””,
                  ”kasiKingaku”:”0”
                 }
               ],
  ”denNo”:”10000”
}
```

[10] JavaScript Object Notation (JSON)は、標準 ECMA-404 (JSON データ交換フォーマット)および ECMA-262 (ECMAScript 言語仕様、第 3 版)で定義されています
https://developer.mozilla.org/ja/docs/Glossary/JSON

Copyright(C)2021 Future Office Coordinate Service Corporation All Rights Reserved.

7.6. ラムダ式の例

Java 言語のラムダ式はインスタンスを生成して式の一部を置き換えてくれます[11]。

関数のオブジェクト化ではなく前もってインタフェースを用意しておく必要があり、JDK11 には引数と戻り値の組み合わせが異なる 43 種類[12]が同梱されています。使う場面を想定して用意されているため、（仕組みに興味がある人以外は）場面を覚えておくだけで十分です。

（1）ソート-並べ替え順の指定（Comparator）

以下の例は、自インスタンスのサブクラスを含むメソッドを調べてメソッド名で昇順に並べ替える処理の例です。引数の 2 オブジェクトの大小判断を行っているのがラムダ式です。

```
final List<Method> vMeth = new ArrayList<Method>();
{
    Method[] allMeth = this.getClass().getDeclaredMethods();
    for (Method meth: allMeth) {
        String m = meth.getName();
        if ( m.length()>8 && m.substring(0, 8).equals("validate") ){    //バリデータ
            vMeth.add(meth);
        }
    }
    // バリデータをメソッド名順に並び替え
    Collections.sort(vMeth,
        (obj1, obj2) -> {
            String o1 = obj1.getName();
            String o2 = obj2.getName();
            return o1.compareTo(o2);
        }
    );
}
```

Collections.**sort** の呼び出し式は以下のようになっており...

Collection.sort(List<T> list, Comparator<? super T> c)

List：sort 対象、Comparator：大小判定を行うクラス

この例では大小判定を行うクラスをラムダ式に置き換えています。ラムダ式の中は比較対象のオブジェクト 2 つ（obj1, obj2）を引数にして obj1 の名前が obj2 の名前より小さい場合は負の整数、等しい場合はゼロ、大きい場合は正の整数を返すようにしています（都合の良いことに String クラスの compareTo は Unicode のコード比較を行って「負、0、正」の値を返してくれます）。

---

[11] 使い勝手はそんな感じ...Oracle 社のサイトの説明は若干異なります。
https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html

[12] https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/function/package-summary.html

Copyright(C)2021 Future Office Coordinate Service Corporation All Rights Reserved.

7.7. Stream＋ラムダ式の例

前項 7.4 の再帰処理で挙げた例の中に、コレクションクラスに対する操作でラムダ式を使っています。例では JDK1.8 で追加になった Stream インターフェースとラムダ式を組み合わせています。

※コレクションクラスとは、Collection のインターフェースを持つクラスで、同一種類のオブジェクトをグループにして扱います。良く使われるコレクションクラスの例としては、HashSet、ArrayList、ArrayDeque、HashMap 他があります

※Stream はコレクションの中の要素を逐次操作するインタフェースです

【関連部分抜粋】

```
Map<String, Object> map = new HashMap<String, Object>();
        (略)
        map.put(set.getKey(), value.asJsonArray().stream()
                .map(p->json2Map((JsonObject)p))
                .collect(java.util.stream.Collectors.toList()));
```

〔整形後〕

```
Map<String, Object> map = new HashMap<String, Object>();
map.put(set.getKey(), value.asJsonArray().stream()①
        .map②( p -> json2Map( (JsonObject)p )④ )
        .collect③(java.util.stream.Collectors.toList())
        );
-----------------------------------
```

① Json 配列から Stream を取り出します

② Stream の map メソッドは、引数の関数（ラムダ式）で処理した要素で Stream を作り出します

③ Stream の collect メソッドは、渡された要素で新しいコレクションを作りだします

… java.util.stream.Collectors.toList() を指定すると List が作られます

ここまでの操作で、value.asJsonArray の全要素に json2Map の加工をした List が作られます

Stream ⇒ map(関数で加工)>Stream ⇒ List

④ Stream の 1 要素を引数(p)として受け取り、JsonObject 型にキャストして json2Map()に渡します

これは、(p) -> { return json2Map((JsonObject)p) } から "()"、"{}"、return を省略した形式です

※Stream はこの例のようにコレクション⇒中間操作⇒終端操作（新しいコレクション作成）を行う仕組みで、中間操作には map 以外に要素を選別する filter や limit、並べ替えの sorted 他いくつか用意されています。終端操作には allMatch、count 等の要素の中身を調べるものや reduce のように要素の集計を行うもの等があります[13]。

[13] https://docs.oracle.com/javase/jp/8/docs/api/java/util/stream/Stream.html

Copyright(C)2021 Future Office Coordinate Service Corporation All Rights Reserved.

7.8. Stream＋メソッド参照式の例

メソッド参照も JDK1.8 で追加され、Stream と一緒の使い勝手が良くなっています。

```
import javax.json.JsonObject;
import javax.json.JsonObjectBuilder;
/**
     * Map の包含関係を再帰的に Json に変換します。
     * @param resMap
     * @return JsonObject
     */
    @SuppressWarnings("unchecked")
    private JsonObject map2Json(Map<String, Object> resMap) {
        JsonObjectBuilder builder = Json.createObjectBuilder();
        resMap.entrySet().forEach(es->{
            String key = es.getKey();
            Object val = es.getValue();
            if (val instanceof Map) {
                builder.add(key, map2Json((Map<String, Object>)val));
            }else if(val instanceof List){
                builder.add(key,
                        ((List<Map<String, Object>>)val).stream()
                        .map( p->map2Json(p) )
                        .collect①(
                                Json::createArrayBuilder,
                                JsonArrayBuilder::add,
                                JsonArrayBuilder::add
                        ));
            }else if(val instanceof Boolean){
                builder.add(key, (boolean)val);
            }else {
                builder.add(key, Objects.toString(val, ""));
            }
        });
        JsonObject obj = builder.build();
        logger.log(Level.INFO, obj.toString());
        return obj;
    }
```

① Stream の collect メソッドは渡された要素で新しいコレクションを作りだす終端操作で、API の形式が複数あり、この例はコレクションの作成と要素追加メソッドの参照を渡す例です。

<R> R collect（Supplier<R>サプライヤ, //入れ物の初期作成
BiConsumer <R, R super T >アキュムレータ,//要素の追加
BiConsumer <R, R> コンバイナ） //結果の合成[*1]

[*1]：Collection は複数スレッドを使い stream を並列で処理する parallelStream()があり、この場合の並列に出来上がった複数の中間結果を統合するメソッドを指定します。

Copyright(C)2021 Future Office Coordinate Service Corporation All Rights Reserved.

8. 応用

クラスの機能は public なメソッド／フィールドで公開し、それ以外はカプセル化（アクセス修飾子を private）し隠蔽します。メソッドはアプリケーション開発の方式によりますが、所謂 サービスクラス、エンティティクラス、ユーティリティクラス等の[14]用途毎に以下の種類があります。

（1）サービスクラス

フレームワークに組み込んで業務処理を行うメソッド…インスタンスメソッド（固有のデータを処理）

（2）エンティティクラス

セッター、ゲッターなどのデータ交換用のメソッド、方式によっては update, delete 等の DB 更新用のメソッド …インスタンスメソッド（固有のデータを持つ）

（3）ユーティリティクラス

データフォーマット、日付、利率計算等のメソッド …クラス(static)メソッド（定数程度）

8.1. メソッドの切り出しルール

メソッドの実装者は担当する public メソッドの機能を private メソッドに切り出しながら全体の構成を決めます。

どんな時に切り出しが必要かというと...

① サイズ（式、文の量）が大きくなって見通しが悪くなってきた
② 分岐条件のネストが深くなって見通しが悪くなってきた
③ 変数名が違うだけのコードが重複して出現している
④ 2 行以上の同一のコードが複数個所に出現している
⑤ プロジェクトで定めた実装のためのルール（これは、上記の全てのルールに優先します）

※ ①～②と③～④は切り出しの目的が異なります。①、②は分かり易さが主目的で③、④は「一つの事実は一つのコードで表す〔2 行同じなら切り出す〕」ことで改修漏れや矛盾の混入を防ぎます。

切り出し方は...

① 情報の塊で分ける（メソッドごと差し替え可能な程度）
② メソッド間の情報の受け渡しは引数と戻り値だけで行う（フィールドを使わない）

※メソッド間の関係を疎にし、全体の見通しをよくすることを目指します。

[14] アプリケーションのクラス分割やメソッドの割付については、設計手法〔有名なのは、ドメイン駆動設計(DDD:Domain Driven Design)〕や採用したフレームワークで異なります

Copyright(C)2021 Future Office Coordinate Service Corporation All Rights Reserved.

8.2. 実装例

ユースケースとして、「伝票テーブルを読込み、エラーがあったレコードのエラーフラグを true にするとともに、エラーメッセージを設定する。」を考えます。

＜伝票テーブルの構成＞

```
CREATE TABLE denpyo(
        err bool                        --エラー識別
      , trdate CHAR(8) NOT NULL         --取引日付
      , denpyono CHAR(5)                --伝票番号
      , tantou CHAR(4)                  --担当者
      , karikamokucd CHAR(6)[]          --借方科目コード　配列
      , karikingaku VARCHAR(13)[]       --借方金額　　配列
      , kasikamokucd CHAR(6)[]          --貸方科目コード　配列
      , kasikingaku VARCHAR(13)[]       --貸方金額　　配列
      , tekiyo VARCHAR(50)              --摘要
      , sysdate CHAR(8) NOT NULL        --システム処理日
      , errmessage VARCHAR(100)         --エラーメッセージ
 , CONSTRAINT pk_denpyo PRIMARY KEY (denpyono)
);
```

＜処理概要＞

適用する方式はフレームワークに Spring Batch、アプリケーションは ItemProcessor として以下の制御フローを作成します。

A. 単項目チェック
　　取引日付チェック
　　　形式チェック
　　　形式チェックでエラーがなかったとき ＞ 範囲チェック（昨年～処理日）
　　伝票番号…必須
　　担当者…必須
　　借方金額…数値のみで 13 桁以内
　　貸方金額…数値のみで 13 桁以内

B. 単項目チェックでエラーが無かったとき ＞ 関連チェック
　　借方関連チェック…科目コードと金額が科目コード≠""かつ金額≠ 0 のペアであること
　　貸方関連チェック…科目コードと金額が科目コード≠""かつ金額≠ 0 のペアであること
　　貸借関連チェック…借方と貸方の合計金額が一致している かつ 金額 ＞0 であること

C. 単項目チェック及び関連チェックでエラーがなかったとき ＞ 主（更新）処理
　　仕訳テーブルの編集
　　仕訳テーブルに insert

Copyright(C)2021 Future Office Coordinate Service Corporation All Rights Reserved.

＜public メソッド 実装内容＞

クラス宣言とフレームワークから呼び出される public メソッドは以下のようになります。

（クラス名、Dto 名等は仮で、**太文字**部分がキーワードです）

※ここではメソッドの切り出し方の説明をしたいので、フレームワークの使い方等は割愛します

```
@Component
public class DenpyoValidateItemProcessor implements ItemProcessor<DenpyoDto, DenpyoDto> {
    Logger logger = LoggerFactory.getLogger(this.getClass());//ログ出力用インスタンス
      @Override
      public DenpyoDto process(DenpyoDto item) throws Exception {
          logger.debug("item denpyono:" + item.getDenpyono());
          // A. 単項目チェック
          if (validateItems(item) > 0) {
              return item;
          } else { // B. 関連チェック
              if (validateKanren(item) > 0) {
                  return item;
              }
          }
          // C. エラー無 ⇒ 主処理
          // < テーブル編集 & Ｉｎｓｅｒｔ -- 略 -- >
    }
```

●単項目チェック、関連チェック、更新処理の実行条件が分かり易いようにチェックは内部メソッド化し、if 文の判定式をメソッド呼び出し式にしています。この場合の API は以下になります。

```
    /**
     * 単項目チェック
     *
     * @param item
     * @return エラー件数
     */
    private int validateItems(DenpyoDto item) {

    /**
     * 関連チェック
     *
     * @param item
     * @return エラー件数
     */
    private int validateKanren(DenpyoDto item) {
```

Copyright(C)2021 Future Office Coordinate Service Corporation All Rights Reserved.

＜単項目チェックメソッド 実装内容 一部＞

以下は単項目チェックの取引日付に関する部分（形式と範囲）です。

会計日付等は会社・組織毎にカレンダーを持っているのが一般的ですが、この例では Java 組込のクラスを使ってチェックしています。また、メッセージの設定に列挙子(enum)を使っています。

```
/**
 * 単項目チェック
 *
 * @param item
 * @return エラー件数
 */
private int validateItems(DenpyoDto item) {
    // 取引日付チェック
    logger.debug("取引日付 trDate:" + item.getTrdate());
    StringBuilder msg = new StringBuilder();
    int errcnt = 0;
    if (validateTrdate(item.getTrdate(), msg)) {
        ++errcnt;
    }
    // 伝票番号チェック（略）

    // エラーあり＞item にエラー情報を設定…return 後、DB に反映されます。
    if (errcnt > 0) {
        item.setErr(true);
        item.setErrmessage(msg.toString());
    }
    return errcnt;
}
/**
 * 取引日付チェック
 *
 * @param testDate
 * @param msg
 * @return boolean
 */
private boolean validateTrdate(String testDate, StringBuilder msg) {
    if (testDate == null) {
        msg.append(Message.E1_1.getMessage());
        return true;
    }
    Date wDate = null;
    if (testDate.length() != 0) {
        SimpleDateFormat sdb =
            testDate.lastIndexOf('/') != -1 ?
                new SimpleDateFormat("yyyy/MM/dd"): new SimpleDateFormat("yyyyMMdd");

        sdb.setLenient(false); // 存在しない日付や曖昧な入力形式を許さない
        ParsePosition ppos = new ParsePosition(0); // 一桁目から解析
        try {
            wDate = sdb.parse(testDate, ppos); // 日付解析
            if (ppos.getErrorIndex() == -1) { // 正しい日付形式
            } else { // 日付変換エラー
```

Copyright(C)2021 Future Office Coordinate Service Corporation All Rights Reserved.

```
                msg.append(Message.E1_4.getMessage());
                return true;
            }
        } catch (Exception e) {
            // 日付形式エラー ⇒ エラー処理
            msg.append(Message.E1_4.getMessage());
            return true;
        }
    }
    Calendar nowCal = new GregorianCalendar(); // 現在日付・時刻
    Calendar tstCal = new GregorianCalendar();
    Calendar befCal = new GregorianCalendar();
    tstCal.setTime(wDate); // テスト対象の日付(0 時)に設定
    befCal.add(Calendar.YEAR, -1); // 1 年前の今日・時刻
    if (tstCal.after(befCal) && tstCal.before(nowCal)) {
        // 時間レベルで 1 年前の今日・現在時刻<テスト対象日 0 時<今日・現在時刻
    } else if (tstCal.get(Calendar.YEAR) <= befCal.get(Calendar.YEAR)) {
        // テスト対象日は去年の日付 or 古い
        msg.append(Message.E1_2.getMessage());
        return true;
    } else {
        msg.append(Message.E1_3.getMessage());
        return true;
    }
    return false;
}
/**
 * メッセージを管理する enum です
 *
 */
enum Message{
    E1_1("1-1:取引日付は入力必須です")
   ,E1_2("1-2:日付が古すぎます")
   ,E1_3("1-3:未来日付は扱えません")
   ,E1_4("1-4:カレンダーに存在しない日付です")
   ;
    private final String msg;
    Message(String msg){
        this.msg = msg;
    }
    String getMessage(){
        return msg;
    }
}
```

enum は特殊なクラスで、列挙型定数（列挙子-「;」の前に列挙した"E1_1"等の定数）を経由して各定数毎の初期化されたインスタンスが参照できます。このとき、列挙子の後ろの()内に書かれているのがコンストラクタ実行の引数です。

enum にはメソッドを書くことができ、列挙子を経由して呼び出す(*enum 名.列挙子*.メソッド名())ことができます。メソッドを列挙子毎に書く（列挙子の後に {処理}）ことも、上記の例のように共通のメソッドとして書くこともできます。また、values(),valueOf(String name)が暗黙的に宣言されます。

Copyright(C)2021 Future Office Coordinate Service Corporation All Rights Reserved.

＜関連チェックメソッド 実装内容 一部＞

関連チェックの対象になる明細部分は配列になっていて、借方／貸方が対称になっています。したがって、配列の繰り返し、借方／貸方の処理の共通化、借方：貸方の処理の配置を考えます。

例では、以下のメソッドに切り出しています。

① 科目コードと金額のペアチェック×明細行数

② 配列金額の集計

```
/**
 * 関連チェック
 *
 * @param item
 * @return エラー件数
 */
private int validateKanren(DenpyoDto item) {
    int errcnt = 0;
    //借方科目・金額ペアチェック
    if (validateKamokuKingaku①(item.getKarikamokucd(), item.getKarikingaku())) {
        //エラー処理 ( ++errcnt; 他)
    }
    //貸方科目・金額ペアチェック
    if (validateKamokuKingaku①(item.getKasikamokucd(), item.getKasikingaku())) {
        //エラー処理 ( ++errcnt; 他)
    }
    //借方金額合計と貸方金額合計を比較
    if (sumKingaku②(item.getKarikingaku()) != sumKingaku②(item.getKasikingaku())){
        //エラー処理 ( ++errcnt; 他)
    }
    return errcnt;
}
/**
 * 科目コード・金額ペアチェック
 *
 * @param kamoku[]
 * @param kingaku[]
 * @return boolean
 */
private boolean validateKamokuKingaku(String[] kamoku, String[] kingaku) {
    if (kamoku.length != kingaku.length) {
        //エラー処理
        return true;
    }
    for (int i = 0; kamokucd.length > i; i++) {
        if (kamokucd[i]==null || kamokucd[i].equals("") //"||"が"|"だと NullPointer が発生
        ||  kingaku[i]==null  || kingaku[i].equals("")) {
            //エラー処理
            return true;
        }
    }
}
```

Copyright(C)2021 Future Office Coordinate Service Corporation All Rights Reserved.

配列の集計には、「Stream＋ラムダ式、メソッド参照式」を使っています。

Stream を使うと、処理をロジックとして書く必要がないので簡潔でデバッグも楽になります。また、意味が分かれば流用や肉付けも簡単にできます。

```
/**
 * 合計金額算出
 *
 * @param kingaku[]
 * @return long
 */
private long sumKingaku(String[] kingaku) {
    return Arrays.stream(kingaku)                    //金額配列から Stream を作成
                .filter(Objects::nonNull)            //null を除外
                .filter(s -> s.matches("¥¥d+"))  //数値以外を除外
                .mapToLong(Long::parseLong)          //文字列数値変換
                .sum();                              //集計
}
```

【小数点付きの場合】java.math.BigDecimal を使います

```
/**
 * 合計金額算出
 *
 * @param kingaku[]
 * @return BigDecimal
 */
private BigDecimal sumKingaku(String[] kingaku) {
    return Arrays.stream(kingaku)
                .filter(Objects::nonNull)
                .filter(s -> s.matches("¥¥d+¥¥.*¥¥d*"))  //数値と"."*1 以外を除外
                .map(BigDecimal::new)                    //文字列 10 進数変換
                .reduce(BigDecimal.ZERO, BigDecimal::add);//集計
}
```

*1:この正規表現では整数部 1 桁以上、小数点「.」と小数部は有っても無くてもよいになります

以下は全て true になります。

```
"123".matches("¥¥d+¥¥.*¥¥d*");
"123.".matches("¥¥d+¥¥.*¥¥d*");
"123.4".matches("¥¥d+¥¥.*¥¥d*");
```

更に桁区切りのカンマ「,」が有っても無くてもよい場合は、以下の処理を追加します。

```
    return Arrays.stream(kingaku)
                .filter(Objects::nonNull)
                .map(s -> s.replace(",", ""))            //カンマを削除
                .filter(s -> s.matches("¥¥d+¥¥.*¥¥d*"))  //数値と"."*1 以外を除外
                .map(BigDecimal::new)                    //文字列 10 進数変換
                .reduce(BigDecimal.ZERO, BigDecimal::add);//集計
```

Copyright(C)2021 Future Office Coordinate Service Corporation All Rights Reserved.

9. ブロックの組立

アプリケーションの構造は適用するフレームワークで決まります[15]。例えば Spring Batch であれば ItemProcessor、ItemReader、ItemWriter etc 。Web であれば、Controller、Service、Entity/Domain 等の単位でクラスや関連の定義体を作ります。このうち、ユースケース／業務処理を実装する ItemProcessor や Service[16]クラスは処理内容に応じてブロック（メソッド）の組立が複雑になりがちです。

9.1. ブロックの切り分け手順

ブロックの切り出しルールを前項 8.1 に書きましたが、ユースケースの実装ではトップダウンで計画的にブロックの切り分けを行う必要があります。

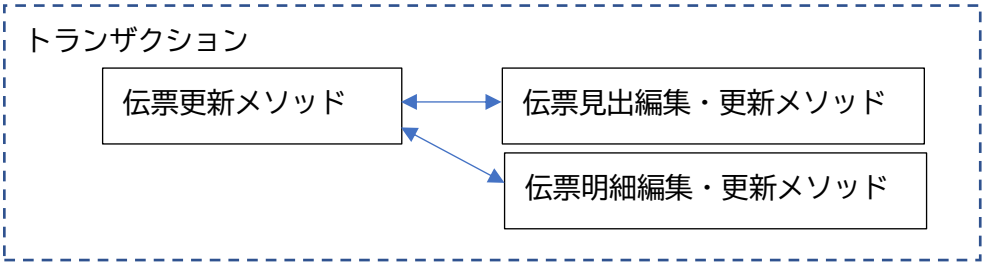
（1）コードの行数

最初にサイズ（コードの行数や命令数）でブロックの内容を決めます。プロジェクトのコーディング規約内で、処理の概観を見通せる（例えば画面スクロール無で全体が見える）サイズにして、その中に必要な処理が全て収まるようにします。収まらない処理はメソッド呼び出し式にして子メソッドに切り出します。ユースケースのボリューム次第では例えば ItemProcessor の呼出し口である process メソッドは、単項目チェック、関連チェック、更新（出力編集）処理、エラー編集処理の各メソッド呼出し式及びエラーの有無による制御だけになるかもしれません。

（2）機能の階層

メソッド呼び出し式への置き換えは、機能単位になるように行います。機能単位とは、単項目チェックであれば、伝票日付の形式チェック、伝票日付のカレンダーチェック、伝票番号の形式チェック…のように意味のある完結した（**メソッドの名前で内容が明確になる**）単位に実装します。

特に、データベースではトランザクションとして完結する単位にすることが必要になります。例えば、伝票見出しテーブルと伝票明細行テーブルを更新する場合は伝票見出の編集・更新メソッドと伝票明細編集・更新メソッドを伝票更新メソッドから呼び出すようにしてトランザクションの単位にします。



---

[15] フレームワークを使わなくてもアプリケーションを構築することは可能ですが、裸の Java を使って担当者毎に好き勝手に作っていては生産性が悪い上に保守が困難になります。少なくともアプリケーションの「作り方」は統一しておく必要があります。

[16] 一般的には ItemReader、ItemWriter etc、Entity はデータ構造、Domain は業務ルールを実装しますが、実際にはフレームワークや方式設計次第です。

Copyright(C)2021 Future Office Coordinate Service Corporation All Rights Reserved.

9.2. 繰返し処理と初期処理

Java に限らず、データ処理を行うアプリケーションはブロック単位でみると繰り返し処理とその前後の前処理／後処理の構成になります。繰返し処理は、バッチでレコードを処理する場合やオンラインでのリクエスト処理[17]（Web コンテナが繰返し処理を行います）、配列を処理する場合など、繰返し処理の入れ子がしばしば出現します。

アプリケーションは Web コンテナやフレームワークによりインスタンスが生成され、アプリケーションとして実装した主な業務処理（下図の網掛け部分）は多重のスレッドから呼び出されます。

このため、シングルスレッドで動作するように環境設定するか、スレッド間でフィールド（変数）の更新・参照が干渉しないようにスレッドセーフなプログラミングが必要になります。

【Java の実行ブロック】

――＜クラスのロード＞―――

・静的初期化子(Static Initializers)

――＜インスタンス化（コンテナ／フレームワークによる new 演算）＞―――

・*コンテナ／フレームワークによる DB コネクション等の確保*

・インスタンス初期化子(Instance Initializers)

・コンストラクター

――＜スレッド-1＞―――

繰り返し

・Web リクエスト or バッチ読込み（ItemReader）

・アプリケーションのメソッド呼び出し式（@Controller、ItemProcessor）

――＜スレッド-n＞―――

・Web リクエスト or バッチ読込み（ItemReader）

・アプリケーションのメソッド呼び出し式（@Controller、ItemProcessor）

――＜終了：GC 待ち＞―――

※初期化子／コンストラクターに対応する終了処理として finalize メソッドがありますが、これはインスタンスが消される（GC）ときに実行されるので何時実行されるかは不定です

以上

[17] サーブレットは複数リクエストを n スレッド／1 インスタンスで多重処理します。

Copyright(C)2021 Future Office Coordinate Service Corporation All Rights Reserved.