Java 言語エッセンスのエッセンス

内容

はじめに 1
1. 危険なコード 1
1.1. JVM の構造 2
1.2. Java クラスの構造 5
1.3. 不具合が起こる原因と対策 6
（1） 内部クラスから外側クラスの変数参照 6
（2） Web アプリのインスタンス変数更新 6
（3） String の比較 6
（4） ラッパークラス（Byte, Short, Integer...）の比較 7
（5） オブジェクトの大量生成（String の更新等） 8
（6） 小数点付き数値の計算 8
1.4. 不具合対策を実施する工程 9
2. 慣用コード 9

Java 言語エッセンスのエッセンス

はじめに

“Java”には言語以外の関連仕様が大変多く、Web、DataBase、xml、分散処理、認証等々Java 仕様要求(JSRs: Java Specification Requests)の最終リリース[1]だけでも 260 件超(2021/3 末)あります。

Java と言っただけでは何についての話をしているのか分からないほどですが、ここでは初心者向けに Java 言語仕様と Java 仮想マシン(JVM)仕様[2]を基にして「危険なコード」の理由と「慣用コード」を纏めます。

1. 危険なコード

Java は C と比べると安全と言われたこともありましたが、継続的なバージョンアップによるクラスの変更・追加・封印に平行して危険なコードの例がたくさん報告されています[3]。Eclipse 等の IDE に「静的解析ツール」を組込んで不良コードの検出を行うことが多いですが、それだけでは効率が悪く品質も上がりません（以下が理由です)。

- 注意・警告が重要なものも重要でないものも大量・混在して通知され、見極めが困難
- ツールが発する警告の意味・理由が理解できず、本質的な対策が打てない
- エラーではないが運用開始後に挙動不審の動作が発生して原因が分からない

上記リスクの対処には、Java のクラス（コンパイル後のバイトコード）実行環境（JVM）と Java クラスの構造の理解が必要です。

---

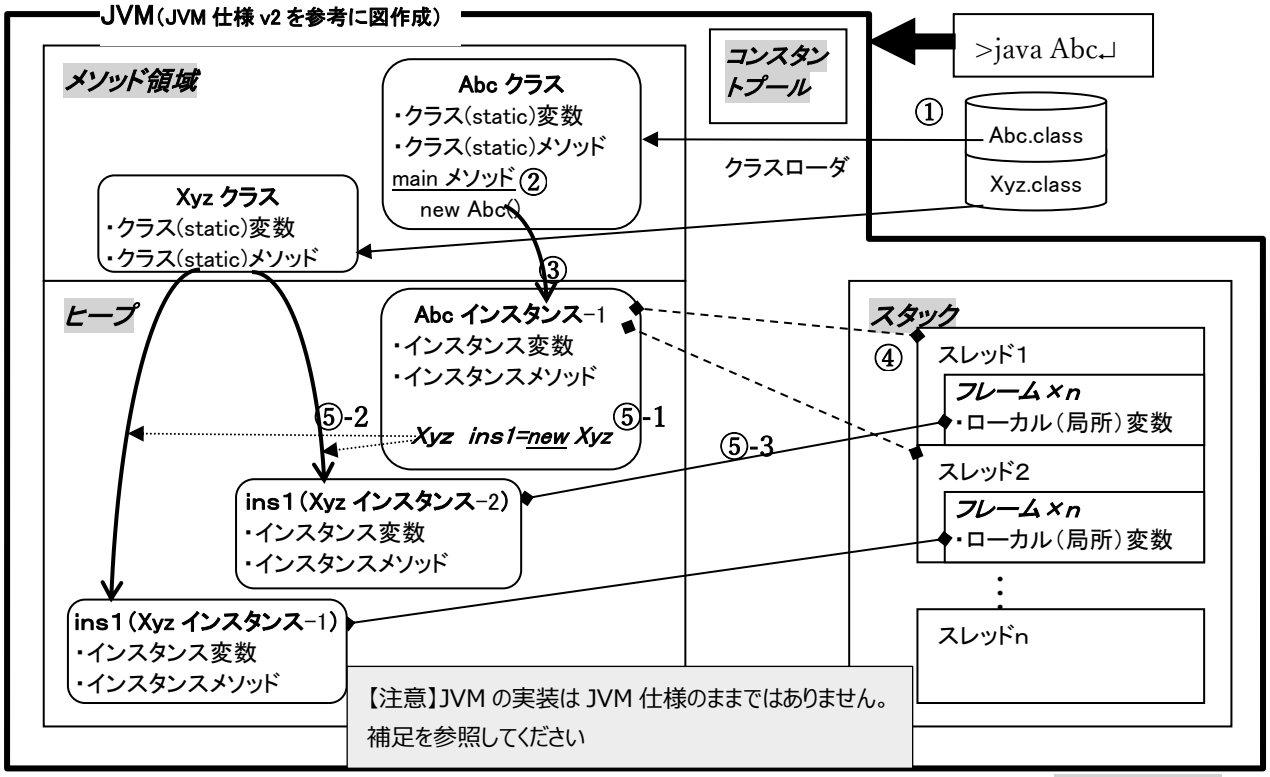
[1] Java 仕様要求 https://jcp.org/en/jsr/stage?listBy=final

[2] 言語仕様、仮想マシン仕様 https://docs.oracle.com/javase/specs/index.html

[3] Java コーディングスタンダード CERT/Oracle 版 https://www.jpcert.or.jp/java-rules/

1.1. JVM の構造

JVM の構造はメモリの用途からみるとメソッド領域、ヒープ、スタックがある。



① パラメータに(Abc)クラスを指定して Java コマンドを実行すると、JVM が起動してクラスローダがクラスファイルをメソッド領域に展開する

② 読み込まれたクラス(Abc)の“main”という名前のメソッド(JVM と同時に起動するクラスには必須のクラスメソッド)に制御が移る（main メソッドが終了すると JVM も終了する）

③ インスタンス変数／インスタンスメソッドを使うために、自分自身（Abc クラス）をインスタンス化（ヒープにコピー）する

④ メソッドの実行はスタックを確保して実行される（命令カウンタ等もここにある)。マルチスレッドで動作するクラスは、各々のスレッド毎にローカル変数が管理される

⑤ 実行中のクラス（Abc）から new 命令が発行されると、該当クラス（Xyz）のインスタンス化及びコンストラクタの実行が起こり、インスタンスに対する参照（ポインタ）は new 命令の左辺で指定されたローカル変数に格納される

⑥ メソッドの実行が終わるか変数=null によりインスタンスが参照されなくなるとガーベイジコレクション(GC:GerbageCollection)の対象になり、メモリが枯渇してきたタイミングで消去される

スレッドというのは、１プロセス（ここでは JVM＝Abc クラスが相当する）内で多重平行で処理を行う機構で、「n スレッド／インスタンス」なります。次の点に注意！**。

★ クラス変数は、インスタンス間とスレッド間で共用になる

★ クラス変数／インスタンス変数は、スレッド間で共用になる

★ ローカル変数はスレッド毎に別の場所で管理される

⇒ローカル変数に格納された new 命令で作ったインスタンスはスレッド固有で管理される

【注意】JVM の仕様と実装について 補足

JVM 仕様は実装の方法までは定義しておらず、どう実装するかは JVM 開発者次第です。ネット上の情報は実装（特に Oracle の HotspotJVM）の用語が頻出していますので、以下に関連する文書の一部を転記します。

● Java SE 16 の JVM 仕様［2.5.実行時データ領域］には以下のように記述されています。

2.5.2. Java 仮想マシンスタック

各 Java 仮想マシンスレッドには、スレッドと同時に作成されたプライベート *Java 仮想マシンスタック*があります。Java 仮想マシンスタックはフレームを格納します（§2.6）。（中略） ローカル変数と部分的な結果を保持し、メソッドの呼び出しと戻りに関与します。Java 仮想マシンスタックは、フレームのプッシュとポップを除いて直接操作されることはないため、フレームがヒープに割り当てられる場合があります。(以下略)

2.5.3. ヒープ

Java 仮想マシンには、すべての Java 仮想マシンスレッド間で共有されるヒープがあります。ヒープは、すべてのクラスインスタンスと配列のメモリが割り当てられるランタイムデータ領域です。ヒープは、仮想マシンの起動時に作成されます。オブジェクトのヒープストレージは、自動ストレージ管理システム（**ガベージコレクター**と呼ばれます）によって**再利用されます**。(以下略)

2.5.4. メソッドエリア

Java 仮想マシンには、すべての Java 仮想マシンスレッド間で共有される***メソッド領域***があります。(中略) 実行時定数プール、フィールドおよびメソッドデータなどのクラスごとの構造、およびクラスとインターフェイスの初期化およびインスタンスの初期化（§ 2.9）で使用される特別なメソッドを含む、メソッドとコンストラクターのコードを格納します。(以下略)

2.5.5. 実行時定数プール

実行時定数プールは、クラス単位またはインターフェイスランタイム表現である constant_pool のテーブル class ファイル（§ 4.4）。これには、コンパイル時に既知の数値リテラルから、実行時に解決する必要のあるメソッドおよびフィールド参照まで、いくつかの種類の定数が含まれています。

● Oracle や他メーカはガベージコレクション(GC:Gervage Collection)の性能を上げるためにヒープの細分化を進めています。実装(HotspotJVM)に関して Oracle の関連サイトには以下の記述があります。

https://blogs.oracle.com/poonam/about-g1-garbage-collector%2c-permanent-generation-and-metaspace

JDK 7：PermGen

JDK7 では、永続世代＜注:≒メソッド領域＞*[“＜注:xxx＞”は本文書での追記です（以下同）]*に存在するデータの一部が Java ヒープ＜注:GC 対象のヒープ＞またはネイティブヒープ＜注:OS 管理=GC 範囲外のメモリ＞のいずれかに移動されました。JDK7 の永続世代から移動されたもののリストは次のとおりです。

- シンボルはネイティブヒープに移動されました
- インターンされた文字列＜注:コンスタントプール＞は Java ヒープに移動されました
- クラスの統計は Java ヒープに移動されました

JDK8：PermGen

永続世代は JDK8 で完全に削除されました。オプション PermSize および MaxPermSize も JDK8 で削除されました。

JDK8：メタスペース

JDK 8 では、クラスのメタデータがネイティブヒープに格納されるようになり、このスペースはメタスペース(Metaspace)と呼ばれます。JDK8 のメタスペースに追加されたいくつかの新しいフラグがあります。

- -XX：MetaspaceSize = <NNN>

  ここで、<NNN>は、クラスをアンロードするためにガベージコレクションを誘導する可能性のあるクラスメタデータに割り当てられたスペースの初期量（初期最高水準点）（バイト単位）です。量は概算です。最初に初期最高水準点に到達した後は、次の最高水準点はガベージコレクタによって管理されます。

- -XX：MaxMetaspaceSize = <NNN>

  ここで、<NNN>は、クラスメタデータに割り当てられるスペースの最大量（バイト単位）です。このフラグを使用して、クラスメタデータに割り当てられるスペースの量を制限できます。この値は概算です。デフォルトでは、制限は設定されていません。

- -XX：MinMetaspaceFreeRatio = <NNN>

  ここで、<NNN>は 、ガベージ コレクションを誘発するクラスメタデータに割り当てられるスペース（最高水準点）の量の増加を回避するための、GC 後のクラスメタデータの空き容量の最小パーセンテージです。

- -XX：MaxMetaspaceFreeRatio = <NNN>

  ここで、<NNN>は 、ガベージ コレクションを誘発するクラスメタデータに割り当てられるスペース（最高水準点）の量の減少を回避するための、GC 後のクラスメタデータの空き容量の最大パーセンテージです。

1.2. Java クラスの構造

Java クラスは主に下図の要素で構成されます。

クラスの構成要素

コンストラクタ			new 演算子により呼び出されインスタンスを初期化して参照を呼び元に返す
メンバ	フィールド(変数)	クラス(staic)変数	メソッド領域に配置 クラスに1つ[共用]*1
		インスタンス変数	初期値またはコンストラクタで初期化後ヒープに配置*1
	クラス(static)メソッド	*処理*	メソッド領域 インスタンス変数は参照できない
		内部(局所)クラス	既存のクラスを上書きして即時インスタンス化する「匿名クラス」でしばしば使われる(Collections.sort()の第 2 パラ等)
		ローカル(局所)変数	スタックに確保される
	インスタンスメソッド	*処理*	ヒープに配置 インスタンス変数を処理する
		内部(局所)クラス	static メソッドと同 但し、インスタンスメソッド内には非 static クラスのみ作れる
		ローカル(局所)変数	スタックに確保される
	内部(メンバ)クラス…非 static ネストクラス …static 〔入れ子の interface は暗黙の裡に static〕		**非 static**:外側クラスへの暗黙的参照が作られる[∴非推奨] **static**:外側クラスの static フィールドだけが参照できる*2

*1:フィールドはメンバーとしてメソッドや文字列定数と同様にアドレス管理される(JVM 仕様第 4 章)。配置場所は推測(仕様不詳…JVM 実装任せ?)

*2:実装例:public static interface Map.Entry<K,V> 外側(この場合 Map)クラスのデータを扱う内部クラス(この場合 Entry)は static で宣言する

クラス定義の内容は「変数」と「処理」で、定義されている場所と static 修飾子の有無で JVM のどこに配置されるかが決まります。

【用語の纏め】

・インスタンス …クラスがコンストラクタ実行済でヒープに展開された状態
・static/静的 xx/クラス xx …メソッド領域に展開された状態(ヒープには存在しない)
・オブジェクト …インスタンス及びヒープに確保された配列

※原語、和訳が使う人・文書ごとで違っていたり混じったりして分かりづらいですが、主に JVM のどこに配置されるかが重要です。メンバの配置先がメソッド領域(static 修飾子付き-クラス単位の管理)なのかヒープ(非 static-インスタンス単位の管理)なのかでアクセスに制限が掛かったり、インスタンス間で共用になったりします。

1.3. 不具合が起こる原因と対策

JVM やクラスの構成を理解していないと動作時に不具合が発生する場合があります。以下に例を挙げます。

（1） 内部クラスから外側クラスの変数参照

static を付けていない内部クラスには自動的に外側クラスへの「参照」を引数に持つコンストラクタが作られます。(詳細は下記 URL-The Java™ Tutorials 参照)

https://docs.oracle.com/javase/tutorial/java/javaOO/nested.html

https://docs.oracle.com/javase/tutorial/reflect/member/methodparameterreflection.html

［問題］

この参照があるため外側クラスが GC で消されず、メモリに残留します。

［対策］

本当に内部クラスである必要があるか真剣に考える。どうしても必要だと判断したら static 修飾子をつけ、ネストクラスにして必要な参照を渡す。

（2） Web アプリのインスタンス変数更新

Servlet/JSP はクラス毎に一つのインスタンスだけが作られ、複数のリクエストに対して複数のスレッドで動作します。クラス変数だけでなく、インスタンス変数の更新が他のリクエスト処理に影響します。

［問題］

インスタンス変数の更新を行っていると想定外の動作をし、リクエスト毎に結果が変わる可能性があります。

［対策］

リクエスト毎の情報はセッション情報に格納してください。

（3） String の比較

ソース中で文字リテラルを使うとコンパイラが重複する文字列を削除し、コンスタントプールから文字列を参照するようにします。このため、以下のように動作します。

String a = “あ”;

String b = “あ”;

String c = new String(“あ”);

(a==b) … true

(a==c) … false

［問題］

同一の文字列を比較しているつもりでも異なる結果となる場合があります。実装の方法を変更した際に意図せず動作が変わってしまう可能性があります。

［対策］

String.equals()で比較を行います。プリミティブ型と同じ書式で初期化できるのでうっかりする人がいますが、String は参照型であることを再確認してください。

参考：String.intern()を使うと実行時にコンスタントプールに追加できます。

（４） ラッパークラス（Byte, Short, Integer...）の比較

Java5 以降のラッパークラス(Autoboxing/Unboxing)は String と同様の問題を起こします。

① Autoboxing/Unboxing を使った計算

```
Integer intCls = 100;
System.out.println("anser=" + (intCls - 1));
```

② Autoboxing/Unboxing を使わない計算

```
Integer intCls = new Integer(100);
System.out.println("anser=" + (intCls.intValue() - 1));
```

※①は②と同一のコードをコンパイラが生成し、ともに以下の出力を得ます

```
anser=99
```

③ ラッパークラスと定数値を比較

```
Integer intCls = 1234;
System.out.println(intCls == 1234);
```

④ ラッパークラス同士を比較

```
Integer intCls_1 = 1234;
Integer intCls_2 = 1234;
System.out.println(intCls_1 == intCls_2);
```

※③は true、④は false を出力します。

これは④が Unboxing されずに参照型同士のポインタ比較になるためです

⑤ Integer：-128～127 の比較

```
Integer intCls_1 = 123;
Integer intCls_2 = 123;
System.out.println("anser=" + (intCls_1 == intCls_2));
```

※④で使っている数値を変えただけですが、⑤は true を出力します

Integer クラスの valueOf(int)メソッドの API ドキュメントには､次のように書かれています。

「このメソッドは、-128 から 127 の範囲の値を常にキャッシュしますが、この範囲に含まれないその他の値をキャッシュすることもあります」

［問題］

Autoboxing、Unboxing が行われた場合、類似のコードに見えても全く異なる動作をする場合があります。

［対策］

equals()で比較を行います。または、明示的にプリミティブ型に変換して比較します。

（上記例では intCls.*intValue()*==1234）

（５） オブジェクトの大量生成（String の更新等）

オブジェクトを大量に作ると、ヒープの割り当てと解放（GC）の負荷が増します。例えば、Stgirng のオブジェクトは値を変えられないので文字列の連結や分割、置換を行う都度新しいインスタンスが作られます。

［問題］

ヒープに空きがなくなると GC が発生します。GC は幾つかの手法から選択できますが、いずれを選んでもアプリの性能劣化は避けられません。参照が残るオブジェクトが大量になり GC でも空きができない場合は java.lang.OutOfMemoryError 例外で JVM が停止します。

［対策］

文字列の加工が必要な場合は java.lang.StringBuilder クラスを、更に複数のスレッドから参照する必要がある（ケースが思いつきませんが）場合は java.lang. StringBuffer を使ってください。

その他、以下に気を付けてください。

・ファイルを扱う場合は使い終わったファイルハンドラのクローズ
・DB を使う場合は、接続が不要になったコネクションのクローズ
・大きな配列やデータの処理が必要な場合は、分割して処理を行い不要になったオブジェクトは解放（参照している変数に null を代入）

（６） 小数点付き数値の計算

Java には浮動小数点付きの数字を扱うための double 型と float 型がありますが、内部表現は二進数（IEEE754 規格）を使っています。小数点以下は $1/2^n$ で表現するので十進数の 0.1 等は扱うことができません。

［問題］

丸めて近似させるのですぐには気が付かない場合があり（0.1 と表示される）ますが、積算すると正確な値になりません。

［対策］

十進数の計算を行う場合は、java.math.BigDecimal を使ってください。

① 二進数で表現した十進 0.1 を表示

```
System.out.println(new java.math.BigDecimal(0.1f));
```

結果＞0.100000001490116119384765625

※この例では BigDecimal の初期値に float 型(0.1f)を指定しているので、float 表現の値が BigDecimal に渡されている

② 十進数の 0.1 を表示

```
System.out.println(new java.math.BigDecimal("0.1"));
```

結果＞0.1

※小数点付き十進数はコード上、文字列でないと表現できません。

リテラル型を付けなかった場合(0.1)は double(0.1d)として扱われます。

```
System.out.println(new java.math.BigDecimal(0.1));
```

結果＞0.1000000000000000055511151231257827021181583404541015625

1.4. 不具合対策を実施する工程

実装の段階でできるのは FindBugs 等による局所的なコードチェックで、クラスの構造に関するもの（内部クラスやマルチスレッド下のインスタンス変数の扱い等）は方式設計の段階での品質保証が必要です。

2. 慣用コード

一般のプロジェクトでは、開発者をスペシャリストだけで構成することは不可能です。効率と品質を上げるためには、ワンパターン化やテンプレートが必要です。また、より効率の良いやり方がプロジェクトの推進過程で見つかることも多いのでプロジェクトが一定量進んだところでコードの再検やリファクタリングを行う時間も必要です。

以下に、プロジェクト色の無い部分だけを集めた各種コーディング例（慣用コード）を示します。

各種コーディング例

種類	コーディング例	留意点等
分岐	＜or 条件＞ if (a==b || x.equls(“y”) || c){*//数値 a,b が同値か文字列 x が”y”か真偽値 c が真の時の処理*} else if (!d) {*//前段の条件が全て偽で、真偽値 d が真では無い時の処理*} else { *//前段までの条件が全て偽のときの処理;* }	途中で if 文が確定すると以降の条件は評価しない（”||”⇒”|”にすると全ての条件を評価＝実行）
	＜and 条件＞ if (a==b && x.equls(“y”) && c){ *//数値 a,b が同値、かつ文字列 x が”y”、かつ真偽値 c が真*} else { *//偽のときの処理;* }	途中で if 文が確定すると以降の条件は評価しない（”&&”⇒”&”にすると全ての条件を評価）
	＜switch＞ int k = Integer. parseInt (str); switch (k) { case 1:System.out.print("one "); break; case 2:System.out.print("too "); break; case 3:System.out.println("many"); }	・条件項目（例の k）に使えるのは char,byte,short,int のいづれかまたは enum ・break を書かないと、下に書いた処理に制御が流れていく

<table>
<tr><td rowspan="3">繰返し</td><td>＜forループ＞
String[] st = {“ abc ”,”def”,”ghi”};
for (int i=0; i < st.length; i++) {
　if (st[i].trim().equals(“abc”)){ continue; }
　if (st[i].trim().equals(“ghi”)){ break; }
　System.out.println(st[i]);
}
【Java5】拡張 f o r
String[] st = {“ abc ”,”def”,”ghi”};
for (String elm : st) {
　System.out.println(elm);
}</td><td>・i の有効範囲は for 文の中だけ
・Java で使う配列等は 0 ｵﾘｼﾞﾝ
・continue でﾙｰﾌﾟを一回抜ける
　ﾌﾞﾛｯｸの先頭に制御が移る
・break でﾙｰﾌﾟを打切り
　ﾌﾞﾛｯｸの外に制御が移る
左記例では”def”だけが出力される

<拡張 f o r >
所謂 for-each 構文
左記例では、変数 elm に配列 st の要素がひとつづ設定されて要素分処理が繰り返される（Map，List でも利用可能）</td></tr>
<tr><td>＜do ループ＞
int i=100;
StringBuffer buf = new StringBuffer(8);
do { //以下の処理は i の値を（逆順で）hex に変換します
　　buf.append(Character.forDigit(i & 0xF, 16));
　　i >>>= 4;
} while (i != 0);
System.out.println buf.reverse().toString(); // 更にひっくり返して文字列として表示</td><td>・ﾙｰﾌﾟを 1 回処理した後に while 条件を判定する
・ﾙｰﾌﾟの制御は for ﾙｰﾌﾟと同様に continue,break で変更可能</td></tr>
<tr><td>＜while ループ＞
int i=0;
while(true) {
　　i++
　　if (i > 100) {break;}
}</td><td>・ﾌﾞﾛｯｸの処理を開始する直前に条件の判定を行い、while 条件を満たさないときはﾙｰﾌﾟに入らない
・条件に true を書くと break が出るまでﾙｰﾌﾟし続ける</td></tr>
</table>

Stream コレクションの パイプ処理	＜Collection に対する繰り返し　JDK1.8 以降＞ ※List, Map, Set, Queue 等のオブジェクトのグループに対してストリーム・パイプラインを生成して『中間操作』⇒『終端操作』という宣言的な表現で処理することができる。 【中間操作:map, 終端操作:reduce】『畳み込み』結合操作により単一の結果を出力 [例 1] List<Map<String, String>> kariMeisai; BigDecimal karikei = kariMeisai.**stream**() .**map**(p->(new BigDecimal(((Map<String, String>)p).get("kariKingaku")))) .**reduce**(BigDecimal.ZERO, BigDecimal::add); 《処理内容》 ・kariMeisai という List（Map が入ってる）から stream をつくる ・stream の 1 要素毎（引数 p）に変換（map）を行う 変換内容： p を Map 型にキャストし、キー”kariKingaku”の値を取り出す ・stream の map 後の要素を、初期値が BigDecimal 型の 0 に対して、BigDecimal クラスの add メソッドを呼び出す（加算する） [例 2]stream を使った、より単純な集計例 List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6); int sum = numbers.stream().reduce(0, Integer::sum); 【中間操作:filter, 終端操作:collect】 [例 1]数字の List から偶数だけの List をつくる List<String>results = numbers.stream .**filter**(i -> i % 2 == 0) .**collect**(java.util.stream.Collectors.toList());	p->(new BigDecimal(((Map<String, String>)p) .get("kariKingaku"))) は、JDK1.8 で追加されたラムダ（λ）式。原始形は、 (p) -> {(new BigDecimal(((Map<String, String>)p) .get("kariKingaku")))} ( )が引数、{ }が処理で、以下省略記法になっている ・引数が 1 つの場合は、( )を省略可 ・処理が 1 ステップの場合は、{ }と return を省略可

	[例 2]collect の結果を List 以外にする ～ .collect(Json::**createArrayBuilder**, // Supplier:インスタンス生成 JsonArrayBuilder::add, // BiConsumer:処理結果の格納 JsonArrayBuilder::add // BiConsumer:処理結果の統合*1)	*1：ストリームは並列処理が可能（parallelStream を使用）で、その方法（メソッド）を指定することができる 注意！）並行処理の場合、実行都度結果が変わる場合がある
例外の捕捉	＜try～catch～fainally＞ try { blowUp(); } catch (Exception e) { e.printStackTrace(); } finally { *//例外が throw されても、されなくても実行する処理;* }	・例中の blowUp()が例外を throw(ｽﾛｰ)した場合、catch のﾌﾞﾛｯｸに入り、引数として“例外”が渡される ・finally は例外が throw されても、されなくても実行される(指定任意) ・ｔｒｙ文に囲まれない個所で例外が throw されると、処理は中断され catch 節が見つかるまでﾒｿｯﾄﾞの呼出し順を遡る ・例外は catch されると、以降の処理は通常のように行われる

<table>
<tr>
<td>出力編集</td>
<td>＜10 進数、小数点あり＞
import java.text.DecimalFormat;
import java.math.BigDecimal;

BigDecimal dv = new BigDecimal("-12345.6");
try {
　　DecimalFormat df = new DecimalFormat("¥u00A4#,##0.0");
　　String st = df.format(dv);
　　}
} catch(IllegalArgumentException e) {
　　//編集ｴﾗｰが発生したときの処理
}</td>
<td>左記例では、-¥1,234.5 という文字列が st に格納される
・¥u00A4は通貨記号を表し、日本では¥が使われる
・負の数字はﾃﾞﾌｫﾙﾄでは”-“が先頭に付く以外は正の数字と同様の編集になる</td>
</tr>
<tr>
<td></td>
<td>＜日付表現形式変換＞
import java.text.SimpleDateFormat;

SimpleDateFormat sdb = new SimpleDateFormat ("yyyyMMdd");
SimpleDateFormat sda = new SimpleDateFormat ("yyyy/MM/dd");
sdb.setLenient(false);
java.text.ParsePosition ppos = new java.text.ParsePosition(0);
otDate = sda.format(sdb.parse(inDate, ppos)); //parse は解析、format で編集</td>
<td>左記例は yyyyMMdd 形式の文字列を yyyy/MM/dd の文字列に編集している（月は大文字の MM と表現することに注意）
・setLenient(false)は存在しない日付や曖昧な入力形式を許さない指定-デフォルトでは 4 月 31 日は 5 月 1 日に変換される
・ParsePosition は解析開始位置を初期値で与え、エラーが発生した場合は、getErrorIndex でエラー位置が返る（正常=-1）
・parse の戻り値及び format の引数型は Date 型</td>
</tr>
</table>

編集戻し	java.text.DecimalFormat df = new java.text.DecimalFormat("#,##0"); Number st; try { 　　st = df.parse("1,234"); } catch(java.text.ParseException e) { 　　*//編集ｴﾗｰが発生（一文字も数字が無い）したときの処理* }	左記例では、st に int の 1234 が格納される（paese ﾒｿｯﾄﾞは桁数等に合わせて int または double の数字を返す） ・通貨記号は削除、負記号は変換後の数値に反映される（但し、数値の途中に含まれている場合は例外を投げる） ・左記例は、NumberFormatから継承したﾒｿｯﾄﾞを使った例で、DecimalFormatには例外を投げないで先頭から数値として有効な部分だけを数値化するparse(数値,変換位置)等もある 注意）先頭から有効な数字だけを変換する-数字以外を含んでも Exception は発生しないので matchs 等によるチェック要
文字列 チェック	String st = "-1234"; if (st.matches("-?¥¥d{0,4}")) { *//-1234 は真になり、処理が実行される* }	"-?¥¥d{0,4}"は、”-“が先頭に有っても無くてもよく(“-?”)、十進数値(¥d)が 0 桁以上、4 桁以下({0,4})の場合真になる ・¥¥d は、”¥d”が十進数値を表し、最初の”¥”は”¥d”を文字列に含ませるためのｴｽｹｰﾌﾟ文字

日付 チェック	＜形式チェック＞ import java.text.SimpleDateFormat; import java.text.ParsePosition; import java.util.GregorianCalendar; import java.util.Calendar; import java.util.Date; Date wDate; if (testDate.length() != 0) { 　SimpleDateFormat sdb = null; 　if (testDate.lastIndexOf('/') != -1) { 　　sdb = new SimpleDateFormat("yyyy/MM/dd"); 　} else { 　　sdb = 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 　　} 　} catch(Exception e) { 　　// 日付形式エラー ⇒ エラー処理 　} } （続く）	“/”の有無で入力日付形式を判定 1 月 32 日⇒2 月 1 日等の変換を行わないように設定 解析開始位置を設定して parse するとエラー箇所が取得できるので、これにより解析が正しくできたか否かが判る SimpleDateFormat.parse の例外を catch することで入力日付の形式に誤りがあることが判る

日付 チェック （続き）	＜範囲チェック＞ 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 }	特定の日付（Date 型）から Calendar を生成することで、年、月、日単位の比較を行うことができる 左記例の wDate は入力データを Date 型に parse したもの

