標準クラス
例外処理
スレッド

第1.0版

作　成　者	NH 中村広二
作　成　日	2019年8月8日
最終更新日	2019年8月8日

標準クラス

Java には多くのクラスライブラリが用意されている。 もとより、クラスライブラリをすべて覚える必要はない。 必要に応じて仕様書（注）を参照すれば良い。 しかしどんなものがあるかくらいは知っている必要がある。 ライブラリに用意されているメソッドを自分で作ってしまうほど、つまらないことはない。 ここでは、特に重要なクラスに限り、そのまたさらに一部の仕様だけ説明する。

（注）インターネットが使えれば、Java の仕様書は容易に参照できる。 例えば Google で「Math java」などとキーワードを入力し、「日本語のページを検索」を選択して I'm feeling Lucky をクリックすると、Math クラスの仕様書（javadoc）が表示される。 この仕様書には Math クラスで使えるメソッドの一覧と使い方（パラメーター群）が記述されている。

Math クラス

Math クラスは、指数関数、対数関数、平方根、および三角関数といった基本的な数値処理を実行するためのメソッドを含んでいる。 Math クラスのメソッドはすべて static メソッドなので、new 文でインスタンスを生成すること無しに使用できる。

```
double a = Math.sqrt(2);          // a には 2 の平方根の値が入る
```

以下の例では、y と z が int 型なら x も int 型となり、double 型なら x も double 型になる。 int 型と double 型とでは、異なるプログラムが実行されるが、オーバーロードの機能によって実行するプログラムが選択される。 max にはこの他に long 型と float 型の演算もある。

```
x = Math.max(y, z);
```

String クラス

String クラスはこれまで何度も出てきた。 String クラスには多数のメソッドがあるが、それらのうち、ごく一部の良く使われるメソッドを示す。

String substring(int start)
　文字列の start の位置(先頭 0)から最後までの文字列を切り出して返す。
String substring(int start, int end)
　文字列の start の位置(先頭 0)から end-1 までの文字列を切り出して返す。
static String valueOf(int a)
　整数値 a を文字列に変換する。
　static なので String.valueOf(a) のように使う。
int indexOf(String str)
　文字列の中で str と一致する文字列が(最初に)存在する位置を返す。
　文字列が存在しない場合は-1 を返す。
int length()
　文字列の長さを返す。
　Java では全文字が 16 ビットなので半角/全角の区別はない。

使用例

```
String s = "ABCDEF";
s = s.substring(2,5)     // s は"CDE"
```

String クラスの生成の仕組み

String クラスでは一度作成した文字列の中身を変更することができない。 前述の使用例の s は"ABCDEF"から"CDE"に変更していると思うかもしれないが、これは一度作った"ABCDEF"を捨て、新たな領域を確保して、そこに"CDE"をセットし、そのアドレスを s にセットしている。 捨てられた"ABCDEF"はガベージコレクションの機能により、自動的にメモリ領域に返却される。

また内容が同じ場合は値を共有する。 例えば以下の場合は、"ABC"の実体は 1 つしか作らず、s1 と s2 が（アドレスとして）同じ値になる。

```
String s1 = "ABC";
String s2 = "ABC";
```

null リテラル

String は参照型で変数には文字列ではなく、文字列へのアドレスが格納されている。 このような参照型の変数には以下の 01: のように null を代入することができる。 null は true や false とともに、Java 言語で定義されている特別な値で（1 章の予約語リスト参照)、null リテラルと呼ばれることがある。

```
01:  String s = null;
02:  System.out.println(s);  // null と出力される
03:  s = s.substring(1,3);
```

01: のようにすると s はメモリ上に場所だけは確保されるが、その場所には実効的なアドレスは格納されず、どこもポイントしていない状態になる。 この状態で 02: のように s を出力すると null と出力される。 s に文字列"null"が代入するときと、出力結果は同じですが、状況は違うので注意する。

またこの状態で 03: のように String クラスのメソッドを実行しようとすると、NullPointerException という例外が発生する。 実際のプログラムではこの例外がよく発生し、初心者の Java プログラマーを悩ませる。
null リテラルは参照型に代入できるので、String 型だけでなく、配列やその他のすべてのクラスから作成したインスタンスに代入できる。 これらに null リテラルを代入すると、インスタンスの利用終了とみなされ、ガベージコレクションの対象になる。 したがって、使い終わったインスタンスに積極的に null を代入すると、メモリの効果的な利用を図る。

StringBuilder クラス

String クラスでは一度作成した文字列の中身の変更ができないので、文字列を変更するような場合には、StringBuilder クラスを使う方が良い。

StringBuilder クラスには、文字列の追加（append)、削除（delete)、指定位置への挿入（insert)、置き換え（replace）など、多数のメソッドが用意されている。

Integer クラス

Java には 32 ビット整数を扱うデータ型として int 型の他に Integer という型（クラス）がある。 Integer クラスは int 型の数値を 1 つだけ変数として持つとともに、int 型の計算に役立つメソッドを備えている。 これは int 型のデータをラップ（wrap：包み込む）していると言う意味でラッパークラスと呼ばれる。 以下に Integer クラスの主要なコンストラクタとメソッドを示す。 Integer クラスのメソッドには static メソッドとインスタンスメソッドとがある。

コンストラクタ

Integer(int i)

整数値 i を Integer 型に変換する。

メソッド

static Integer valueOf(String s)

文字列 s を 10 進数と見なして Integer 型に変換する。

数字に変換できないと NumberFormatException という例外を発行する。

static int parseInt(String s)

文字列 s を 10 進数と見なして int 型に変換する。

数字に変換できないと NumberFormatException という例外を発行する。

int intValue()

Integer 型の数値を int 型に変換する。

以下に使用例を示す。 static メソッドは Integer.xxx と書き、インスタンスメソッドはインスタンス名.yyy と書くことに注意。 01: はコンストラクタの例。 02: をまとめて 03: のように書ける。 また parseInt メソッドを使えば、02: や 03: のような処理が 04: のように一度で書ける。 parse とは構文解析をすると言う意味。

```
01:   Integer obj = New Integer(123);

02:   String s ="123";
      Integer obj = Integer.valueOf(s);
      int i = obj.intValue();

03:   String s ="123";
      int i = Integer.valueOf(s).intValue();

04:   String s ="123";
```

```
int i = Integer.parseInt(s);
```

Autoboxing/Unboxing 機能

これまでは上の例のように int 型と Integer 型はいちいち明示的に変換しなければなりません。 しかし、これは大変煩わしいので、J2SE5.0 ではこれらの型を自動的に変換する機能が追加された。 int 型から Integer 型へ変換する Autoboxing 機能と、その逆の Auto Unboxing 機能。 この機能が導入されたので前項の 01: と 02: の 3 行目は次のように書くことができる。

```
01:          Integer obj = 123;
02 の 3 行目:    int i = obj;
```

もう 1 つ例を挙げる。 Integer 型同士では計算ができないので、これまでは Integer 型の変数の加算は 03: のように int 型に変換してから加算しなければならなかった。

```
01:  Integer a = new Integer(12);
02:  Integer b = new Integer(34);
03:  Integer c = new Integer( a.intValue() + b.intValue() );
```

これに対し Autoboxing/Unboxing 機能が働く J2SE5.0 以降では次のように簡単に記述できる。

```
01:  Integer a = 12;
02:  Integer b = 34;
03:  Integer c = a + b;
```

その他のラッパークラス

ラッパークラスには int 型だけでなく、すべての基本データ型に対するものがある。 それを全部挙げると、

Boolean、Character、Byte、Short、Integer、Long、Float、Double

となる。 これに対応する基本データ型を列挙してみると、

boolean、 char、byte、short、 int、long、float、double

となっている。 比べてみると Character と Integer は省略型だった char と int がフルスペルになり、その他は先頭が大文字になっただけでつづりは同じであることがわかる。 Java では大文字と小文字は区別されますので、これらは当然違ったクラスとして扱われる。

前項で述べた Autoboxing/Unboxing 機能は int 型と Integer 型間の変換以外に、上記のラッパークラスと基本データ型間の変換にも適用されます。

ラッパークラスの用途

次に説明する ArrayList などのコレクションクラスでは、基本データ型のデータは格納できない。 参照型のデータしか格納できない。 そこで int 型の整数値をこれらに格納するときは、そのラッパークラスである Integer 型に変換して格納する。

またラッパークラスには、int 型に変換する parseInt メソッドや 10 進→16 進変換など基数の変換をするメソッドなどそれぞれの数値の処理に役立つメソッド群が用意される。

ArrayList クラス

複数のデータを扱う時には配列が便利。 しかし配列は初期化時に指定したサイズを変更することができる。 また、配列の並びの途中に新たなデータを挿入するのはかなり大変。 このような問題を解決するために Java ではコレクションフレームワークと言う便利なクラス群が用意されている。 これを使うとデータを様々な構成でグルーピングすることができる。

ArrayList は代表的なコレクションフレームワーク。 ArrayList はサイズが変更できる配列といえる。 生成時に一定サイズを確保し、その後、要素の追加に応じてサイズを自動的に拡張する。 ArrayList では、データの追加（add)、取得（get)、検索（indexOf)、削除（remove)、置換（set）などの操作が可能。

ArrayList の生成

配列では生成時にサイズの指定が必要。 以下は配列を生成するコードですが、サイズとして 5 を指定している。

```
String[] str = new String[5];
```

ArrayList は配列と違って生成時に格納サイズを指定する必要がない。 以下は String 型のデータを格納する ArrayList を生成する例。

```
List<String> ar = new ArrayList<String>();
```

ただし ArrayList でもサイズの初期値は指定できる。 以下は初期値として 30 を指定する例。 使うサイズが予測できる場合は、このように初期サイズを指定することで性能を向上させることができる。 この場合 30 を越える数のデータを格納しようとすると、Java の処理系が自動的にサイズを拡張する。 （配列では範囲を越えると ArrayIndexOutOfBoundsException という例外が発生する）

```
List<String> ar = new ArrayList<String>(30);
```

Array リストは配列に比べて処理速度が落ちますしメモリも消費するので、サイズが決まっている場合には ArrayList ではなく配列を使う。

ArrayList には参照型のデータしか格納できない。 int 型など基本データ型のデータは「ラッパークラス」にして格納する必要がある。

ジェネリックス

ここで示した例では＜String＞のように生成時にデータ型を指定する。 実は、このやり方は J2SE5.0 で導入された新機能でジェネリックス（Generics）と呼ばれる。 それ以前は生成時にはデータ型が指定できず、どんな型のデータでも（但し参照型のみ）格納できた。 そして読出時（get メソッド利用時）にキャストする必要があった。 ジェネリックスを使い格納するデータ型を制限することでバグの少ないプログラムを簡明に書くことができるようになった。 J2SE5.0 以降でもジェネリックスを使わないやり方が可能ですが、あまり使われないので説明は省略する。

使い方の例

ArrayList の使い方を例で示する。 ArrayList クラスは java.util パッケージ内にあるので、使うときは java.util.*;を import しておく必要がある。

```
List<String> ar = new ArrayList<String>(); // ar を生成
ar.add("東京");                  // ar の先頭(位置 0)に"東京"を追加
ar.add("大阪");                  // 次の位置(位置 1)に"大阪"を追加
int ix = ar.indexOf("大阪");     // "大阪"の位置を検索, ix は 1 になる
String st = ar.get(ix);          // 位置 ix の要素を取得, st は"大阪"になる
ar.set(ix,"横浜");               // 位置 ix の要素を"横浜"に置き換える
int n = ar.size();               // ar の要素数を取得, n には 2 が入る
System.out.println(ar);          // ar の内容［東京, 横浜］が出力される
```

int 型のデータを格納する例

前述のように int 型のような基本データ型は直接は格納できないので、int のラッパークラスである Integer 型を指定して生成する。 そして、本来は 01: のように Integer インスタンスを生成してそれを格納すべきだが、02: のように書いても Autoboxing 機能が働いてちゃんと処理してくれる。

```
01:   int_ar.add(new Integer(12));
02:   int_ar.add(12);
```

また、読み出し時にも本来は以下の 03: のように intValue メソッドで Integer 型を int 型に変換して格納すべきだが、Auto Unboxing 機能が働くので 04: のように int 型への変換指示を省略することもできる。

```
03:   int i1 = int_ar.get(0).intValue();
04:   int i2 = int_ar.get(1);
```

Vector クラス

ArrayList とほぼ同じ機能を果たすクラスに Vector クラスがある。 Vector クラスは Java の旧版から存在しているクラス。 ArrayList との違いは Vector が同期化の処理を行う。 同期化の処理が行われると、複数のスレッド（多重処理）から同時に操作されても、データの整合性を保つことができる。 これは一見良いことのように思えるが、不要な場合でも必ず同期化を行うため、性能が低下する。 必要に応じて選択する。

その他のコレクションフレームワーク

コレクションフレームワークには ArrayList や Vector の他にも多くのクラスがある。 ここではその中で主なものを紹介する。

LinkedList クラス

LinkedList は ArrayList と同じように、順序データを管理するが、次のような違いがある。 すなわち、ArrayList はデータの追加や削除が列の末尾を除いて得意でないのに対し、LinkedList は列の先頭へのデータの挿入、削除も得意。

逆に ArrayList は位置（index）を指定してのデータの取り出しが高速なのに対し、LinkedList は、これが苦手。 それは、LinkedList はその名の通り Link 構造になっているから。 LinkedList では、データの登録（add、addFirst、addLast)、取得（get、getFirst、getLast)、登録確認（contains)、検索（indexOf)、削除（remove)、置換（set）などの操作が可能。

LinkedList の使い方を例で示す。 LinkedList クラスも java.util パッケージ内にあるので java.util.*;を import しておく必要がある。 LinkedList でも格納できるのは Object 型だけ。

```
01:   LinkedList<String> lk = new LinkedList<String>();       // lk を生成
02:   lk.add("aaa");                  // lk に"aaa"が格納される
03:   lk.add("bbb");                  // "aaa"の次に"bbb"が格納, "aaa""bbb"の順
04:   lk.addFirst("ccc");             // 先頭に"ccc"が格納, "ccc""aaa""bbb"の順
05:   lk.addFirst("ddd");             // 先頭に"ddd"が格納, "ddd""ccc""aaa""bbb"の順
06:   String sf = lk.getFirst();      // sf には先頭の要素"ddd"が代入される
07:   String sl = lk.getLast();       // sl には末尾の要素"bbb"が代入される
```

HashMap クラス

HashMap クラスにはキー（key）と値（value）とがペアで格納できる。 HashMap では、データの登録（put)、値の取得（get)、登録確認（containsKey)、削除（remove）などの操作が可能。 HashMap でも格納できるのは（キー、値ともに）Object 型だけ。

List ではデータに順番(アドレス)を付けて管理してたが、Map ではデータ(value)にはアドレスは付けず、その代わりにキー(key)という文字列を付加して管理する。また Hash とは与えられたデータから整数値を算出する技術のこと。すなわち HashMap という名前は Hash 技術を使った Map 構造のデータを扱うことを示している。Map には、この他に Tree(木)技術を使ってデータを管理する TreeMap というクラスもある。

HashMap の使い方を例で示す。 HashMap クラスも java.util パッケージ内にあるので java.util.*;を import しておく必要がある。 すでに登録されているキーが再度登録されると、新しい値（value）が古い値に上書きされます。 未登録のキーで get しようとすると、null が返る。

```
01:  Map<String,String> hm = new HashMap<String,String>();       // hm を生成
02:  hm.put("one", "onett");
03:  hm.put("two", "twoson");
04:  hm.put("one", "threek");        // 当初の onett は上書きされ消えてしまう
05:  String s = hm.get("two");       // s には twoson が代入される
06:  System.out.println(hm);         // [one=threek, two=twoson] と出力される
```

コレクションインターフェース

コレクションフレームワークは、データの基本的な構成法に対応したインターフェースを定め、それを実装する形でクラスが提供されている。 例えば ArrayList と LinkedList は同じ List インターフェースを実装している。 したがって、これらには List インターフェースが規定した add、get、remove、set などのメソッドを備えていることが保証されている。 主なコレクションインターフェースとそれを実装したクラスは次の通り。

List	順序のある要素を扱う。(ArrayList,LinkedList,Vector など)
Map	キーと値のペアで管理。順序は保証されない。(HashMap など)
Set	重複を許さない要素を扱う。(HashSet,TreeSet など)

イテレータ

繰り返し処理を巧みに行う手段として Iterator（イテレータ）インターフェースが準備されている。
hasNext()メソッドで次の要素の有無を調べ、存在すれば next()メソッドでそれを取り出すことができる。
next()メソッドは取り出した後、要素の位置を 1 つ進める。 以下は list の内容をすべて出力するプログラムの例。

```
01:   Iterator<String> it = list.iterator();
02:   while (it.hasNext()) {
03:     String str = it.next();
04:     System.out.println (str);
05:   }
```

列挙型

関連する定数をまとめ全体に名前をつけて管理することができる。 これを列挙型と言う。 列挙型は enum というキーワードを使って宣言する。 列挙型を宣言する例を挙げる。 列挙型は J2SE5.0 で導入された新しい構文。

```
enum Janken {
  GU,CHOKI,PA   // ジャンケンで使う"手"の名前を列挙
}
```

enum は enumeration の略。 enumeration は辞書を引くと数え上げること、列挙、一覧などと書かれている。
enum はイーナムとかイニューム、エナムなど人によって様々な読み方をする。 列挙型は最上位のレベル（クラスと同レベル）およびクラスの中（メソッドと同レベル）の 2 種類の場所で記述できます。 メソッドの中では記述できない。

列挙型で宣言するのは定数なので、その名前は上の例のように全部を大文字で書くのが普通です（識別子の命名規則の慣用規則参照)。 上で定義した列挙型の利用法の例を示す。

```
Janken j = Janken.GU;    // 代入
System.out.println(j);  // GU と表示される

if (j == Janken.PA){     // 比較

switch(j) {              // switch 文の評価式として使う
  case CHOKI:            // case 文には Janken. は不要

j = Janken.GUU;          // GUU が未定義なのでコンパイルエラーになる
        // コンパイルの段階で誤りを発見できる→誤りの少ないプログラムになる
```

このように、列挙型として宣言すると定数が統一的に管理でき、誤りの少ないプログラムを書くことができる。

書式付きプリント文（printf 文）

書式付きプリント文（printf 文）を使うと印刷の縦位置や小数点位置がきれいに揃った表を簡単に印刷することができる。 この機能は J2SE5.0 で追加された。 printf メソッドは以下のように記述する。

```
printf(書式文字列, データを与える引数…);
```

最初の引数は書式を指定するための書式文字列、それ以降の引数はデータを与える引数。 データを与える引数はデータの数だけ指定できる（可変長引数）。

printf 文の例を示す。

```
System.out.println("商品コード|  単価 | 個数 |    小計   ");
System.out.println("----------+-------+------+-----------");
System.out.printf("%-10s|%7.2f|%6d|%,10.2f¥n","JDK700207",33.3,102,3396.6);
System.out.printf("%-10s|%7.2f|%6d|%,10.2f¥n","AP2546",57.0,28,1596.0);
```

これらの文を実行すると、次のようにきれいに出力される。

```
商品コード|  単価 | 個数 |    小計
----------+-------+------+-----------
JDK700207 |  33.30|   102|  3,396.60
AP2546    |  57.00|    28|  1,596.00
```

書式文字列には、データの前後にどの程度の余白を取るか、小数点以下の桁数をいくつにするか、3 桁ごとにカンマを入れるかなどの書式が指定できる。 書式文字列にはそのまま出力する固定のテキスト（上の例では |）と、その中に埋め込まれた書式指示子からなりたつ。

書式指示子は % で始まり s,f,d などのアルファベットで終わる。 このアルファベットを変換文字と言い、出力する値の形式を指定する。 書式指示子は次のような構造になっている。

```
%［フラグ］［幅］［.詳細］変換文字
```

書式指示子の各要素の主な意味を示す。

フラグ ー 以下のような書式に関するオプションを指定する。
-：左揃え、0：頭に 0 を詰める、+：常に符号を付加、,：3 桁ごとにカンマを付加
上記で「-」以外は数字のみに適用されます。
幅 ー 出力エリアの最小幅を示します。 データの幅がこれより大きければこの値を越えて印刷されます。
詳細 ー 文字列の時は出力する文字数、数字の時は小数点以下の桁数を指定します。
変換文字 ー 出力する値の形式を指定します。 以下はその例です。
c：文字、d：10 進整数、f：浮動小数点数、x：16 進整数、s：文字列

書式指示子の使用例とその印刷結果を示す。 以下で ¥n は改行を表す。

```
int a = 68;
double b = 68.5;
System.out.printf(" 1: %c¥n", a);
System.out.printf(" 2: %d¥n", a);
System.out.printf(" 3: %+5d¥n", a);
System.out.printf(" 4: %7.2f¥n", b);
System.out.printf(" 5: %04x¥n", a);
System.out.printf(" 6: |%s|¥n", "January");
System.out.printf(" 7: |%10s|¥n", "January");
System.out.printf(" 8: |%-10s|¥n", "January");
System.out.printf(" 9: |%.3s|¥n", "January");
System.out.printf("10: |%-10.3s|¥n", "January");
```

印刷結果

```
 1: D
 2: 68
 3:   +68
 4:   68.50
 5: 0044
 6: |January|
 7: |   January|
 8: |January   |
 9: |Jan|
10: |Jan       |
```

例外処理

Java には例外処理という便利な機能がある。 例外はファイルの終了検出、エラーの発生、0 除算の実行など、通常ではない特別の状態のときに発生する。 例外発生時に実行されるのが例外処理。 例外処理を使うことによって、プログラムの構造がとてもすっきりしたものになる。 ここでは例外処理のごく基本的な内容を紹介する。

例外の構文

例外の構造

```
try {
    例外監視の対象となる文(a)
} catch(例外タイプ A) {
    タイプ A の例外を検出したときに実行する文(b)
} catch(例外タイプ B) {
    タイプ B の例外を検出したときに実行する文(c)
} finally {
    例外発生の有無にかかわらず必ず最後に実行する文(d)
}
```

例外処理の基本的な構文を示す。

(a) try ブロック内に例外の発生する可能性のある文を記述する。
(b) 例外タイプを明記した catch ブロック内に、その例外発生時の処理を書く。
(c) catch ブロックは複数書くことができる。
(d) finally ブロックには例外発生の有無にかかわらず最後に実行する処理を書く。

try、catch、finally はこの順序で連続して記述する必要がある。 try と catch だけ、または try と finally だけでも構わない。

例外処理の流れ

(1) 例外が発生すると例外処理状態となり (2) へ行く。
(2) try ブロック内で起きていれば (3) へ、そうでなければ (4) へ行く。
(3) catch ブロックのタイプを上から順に調べる。タイプがマッチすれば (6) へ行く。
　catch ブロック群を最後まで探しても、マッチしなければ (4) へ行く。
　ただし、finally ブロックがあれば、(4) へ行く前にそれを実行する。
(4) 自メソッドを呼び出した処理があれば、そこに戻って (2) から繰り返す。
　自メソッドを呼び出した処理がなければ (main メソッドのとき)、(5) へ行く。
(5) システムの処理が行われ、プログラムは実行停止する。(終わり)
(6) 例外処理状態は解除され、その catch ブロック内の処理を実行して (7) へ行く。
(7) finally ブロックがあれば、それを実行して finally ブロックの次の処理へ進む。
　finally ブロックがなければ、catch ブロック群の直後の処理へ進む。

例外処理の例

実際のプログラムで例外発生時の処理の流れを見てみる。

```
01:  public class Sample {
02:    public static void main(String[] args) {      // メイン処理
03:      System.out.println("main-start");
04:      aaa();                                       // aaa を呼ぶ
05:      System.out.println("main-end");
06:    }

07:    public static void aaa() {                     // aaa メソッド
08:      try {
09:        System.out.println("aaa-start");
10:        bbb();                                     // bbb を呼ぶ
11:        System.out.println("aaa-end");
12:      } catch(ArithmeticException e) {             // 例外の catch
13:        System.out.println("aaa-catch");
14:      } finally {                                  // finally 文
15:        System.out.println("aaa-final");
16:      }
17:    }
```

```
18:     public static void bbb() {                    // bbb メソッド
19:       System.out.println("bbb-start");
20:       int x = 0;
21:       int y = 10/x;                               // 例外発生!(0 除算)
22:       System.out.println("bbb-end");
23:     }
24:   }
```

このプログラムでは次のように処理が行われる。

(1) 03: main-start を出力
(2) 04: メソッド aaa()を呼ぶ。 07 行目に移る
(3) 09: aaa-start を出力
(4) 10: メソッド bbb()を呼ぶ。 18 行目に移る
(5) 19: bbb-start を出力
(6) 21: 0 除算により ArithmeticException 例外が発生する（例外処理状態になる）
(7) try/catch がないので、呼び出し元に戻る（例外処理状態のまま）
(8) 12: 対応する例外の catch があるのでそこに移る（例外処理状態解除)。 11 行目は実行されない
(9) 13: aaa-catch を出力。 catch ブロック終了
(10) 14: finally ブロックがあるのでそこに制御が移る
(11) 15: aaa-final を出力。 finally ブロック終了
(12) メソッド aaa()が終わりなので、呼出元に復帰する
(13) 05: main-end を出力。 main メソッド終了

チェック例外と非チェック例外

Java にはプログラマーが必ず try/catch しなければならない例外がある。 これをチェック例外、または検査例外という。 チェック例外は、例えば以下のように処理メソッドごとに catch すべき例外のタイプまで決まっている。 チェック例外は、try/catch ブロックを作りさえすれば、catch ブロック内に記述する内容は自由。 中括弧の中に何も書かなくても構わない。

ファイルオープン時 … FileNotFoundException 例外
ファイル読み込み時 … IOException 例外

チェック例外の例

以下にチェック例外の例を示す。 FileReader の new 文では FileNotFoundException が発生する可能性がある。この例外はチェック例外なので、try/catch しないとコンパイルエラーになる。

```
public void fileOpen(String filename) {
  try {
    FileReader fr = new FileReader(filename);
  } catch(FileNotFoundException e) {
    System.out.println(e);
  }
}
```

※ 例外処理の強要は、かなり厳しい規則ですが、質の高いプログラム作りに有効。 しかしこの機能は Java より新しい C#では導入されておらず、プログラム言語の必須機能として必ずしもコンセンサスが得られているわけではないのかも知れない。

非チェック（非検査）例外

一方、例外が発生する可能性はあるが、必ずしも例外処理を行う必要がないケースもある。 それが非チェック例外。 非チェック例外には、次の 2 種類がある。

(1) 例外処理を記述してもプログラムレベルでは回復できないような重大な例外
(2) あまねく発生するので、いちいち try/catch するのが効率的でない例外

例えば、メモリ枯渇（OutOfMemoryError）のような重大なエラーは catch しても回復しようがなく、結局プログラムを止めるしかない。 したがって、このようなエラーでは try/catch の記述は強制されない。 これは上記（1）の例。

宣言した範囲を越えて配列をアクセスすると ArrayIndexOutOfBoundsException という例外が発行される。 この例外の発生に備えようとすると、配列を扱うすべての処理を try ブロックで囲まねばならない。 しかも、範囲外のアクセスはもうプログラムのバグなので、例外処理などせずに、さっさと実行を止めた方がよい。 このような理由で、非チェック例外にすべき例外のグループがある。 これは上記（2）の例。

例外の種類と階層構造

Java では例外もクラスを使って管理する。 すなわち、例外が発生すると、そのタイプごとに用意された例外クラスを使って例外を発行する。 例外クラスは継承を基にした階層構造をなしています。 以下に例外クラスの例を示す。

Throwable クラス

例外クラス全体のスーパークラスになっているのが、Throwable というクラス。 Throwable クラスを直接継承しているクラスに、Error クラスと Exception クラスがある。 すなわち、以下のような構造になっている。

Throwable クラス = Exception(例外)クラス ＋ Error(エラー)クラス

例外クラスの親玉が Exception ではなく、Throwable であることに注意すること。 この章の説明内容には Exception クラスだけではなく、Error クラスのことも含んでいる。 したがって、本章のタイトルは例外処理ではなく、Throwable 処理とする方が適切かもしれない。 しかし、通常は Error クラスの処理も含めて例外処理と呼んでいる。 例外という言葉は Throwable クラス相当の広い意味と、Exception クラス相当の狭い意味があると考えるべき。 ちなみに、Throwable は「投げることができる」と言う意味で後述のように（広義の）例外は throw する（投げる）ものであるところから来ている。

Error 系例外

Error クラスを継承している例外を Error 系例外と言う。 Error 系例外は回復不可能な重大なエラーで、非チェック例外の説明で（1）として挙げた例外に相当する。 OutOfMemoryError などのように、名前の最後に Error が付きます。 Error 系例外は、普通は try/catch しない。 Error 系例外には、メモリ不足を示す OutOfMemoryError や、スタックオーバーフローStackOverflowError などがある。

Exception 系例外

Exception クラスを継承している例外を Exception 系例外と言う。 Exception 系の例外は回復可能な軽度のエラー。 Exception 系の例外には、名前の最後に Exception が付く。 この例外には、RuntimeException クラスを継承している非チェック例外と、それ以外のチェック例外とがある。 前者が非チェック例外の説明で (2) として挙げた例外に相当する。 すなわち、Exception 系例外は以下のような構造になっている。

Exception クラス = RuntimeException とそのサブクラス(非チェック例外)
＋ その他の Exception クラスとそのサブクラス(チェック例外)

Exception 系の非チェック例外には、0 除算などで発生する算術例外 ArithmeticException、null が代入された参照を使おうとすると発生する NullPointerException、範囲外のインデックスを使って配列がアクセスされたことを示す ArrayIndexOutOfBoundsException などがある。 また、不適切な文字列を数値に変換しようとしたときに発生する NumberFormatException も、なぜか非チェック例外。

Exception 系のチェック例外には、指定した外部ファイルが見つからない FileNotFoundException や、ファイルの終端 (End Of File) に達したときに出る EOFException、およびこれらのスーパークラスである IOException などがある。 また、呼び出そうとするクラスが存在しないことを示す ClassNotFoundException もチェック例外。

catch ブロックでの例外タイプ指定

catch ブロックは例外タイプごとに記述すると説明したが、必ずしも細かいタイプをいちいち指定する必要はない。 catch ブロックでは例外の上位の階層のクラスを指定できるので、例えば以下のように Exception と指定すれば、Exception クラスを継承しているすべての例外を catch できる。 その代わり、当然ですが詳細なタイプごとの処理はできない。

```
catch(Exception e)
```

catch ブロックで使う変数

catch ブロックの丸括弧内では、例外タイプの他に変数名を指定する。 この変数名は任意の名前で良いが、普通 e を使う。 これは例外クラスのインスタンスを指している。 これを print 文に与えると、その例外クラスを説明するメッセージを出力する。 例えば 0 除算の例外発生時、e には次のようなメッセージが入っている。 e は必要がないなら使わなくてかまわない。

```
java.lang.ArithmeticException: / by zero
```

メソッド外での例外処理

チェック例外はかならず try/catch が必要ですが、その場で catch しなくても、その責任を呼び出し元のメソッドに転嫁することができる。 それにはメソッド定義で throws 宣言をすればよい。 throws 宣言は、その例外発生の可能性をプログラマーが認識していることを表明するもの。 throws 宣言はまた、そのメソッドを呼び出す側で、try/catch して欲しいということも表わしている。

throws の例

以下に throws 宣言の例を示す。 チェック例外のところで例示した fileOpen メソッドは try/catch を必要とする FileReader の new 文を含むが、以下のように、メソッドのシグニチャで throws キーワードを記述し、その後ろに発生する可能性がある例外タイプを書くと、このメソッドでは例外を try/catch する必要はない。

```
public void fileOpen(String filename) throws FileNotFoundException {
  FileReader fr = new FileReader(filename);
}
```

発生する可能性がある例外が複数ある場合は、コンマで区切って列挙する。 また catch ブロックと同じように、上位の階層の例外クラスを使うこともできる。 Exception クラスを使えばすべての例外を throws することができる。

throws の引き継ぎ

throws 宣言により、そのメソッドでは catch が不要になるが、今度はそれを呼び出すメソッドで try/catch する（か、さらに上に throws する）義務が生じる。

独自例外のスロー

プログラマーが独自の例外クラスを定義して、それを発行することができる。 独自の例外クラスは Throwable クラスかそのサブクラスを継承して作る。 普通は Exception クラスを継承する。 以下に独自の例外クラスの定義例を示す。 ただ例外を発生させるためだけなら、以下のように { } 内に何も記述する必要はない。

```
class MyException extends Exception { }
```

この例外の発行は throw 文を使って次のように行う。 throw キーワードは throws キーワードと似ているが、別物。

```
throw new MyException();
```

スレッド

スレッド（thread）とは糸のことで、処理の流れを糸に例えたもの。 通常のプログラムは１つの処理だけが走るが、Java にはマルチスレッドと呼ばれる機能があって、比較的簡単に複数の処理を同時に走らせることができる。 とはいっても、マルチスレッドの仕組みはやはり複雑。 ここでは、マルチスレッドの概要を紹介する。

マルチスレッドの起動

プログラムの起動時には Java のスレッドは１つ。 このスレッドをメインスレッドと言う。 新たなスレッドを動作させるには次のようにする。

(1) まず Thread クラスを継承したクラスを定義する。
(2) そのクラスの run メソッドに別スレッドで動作させたいプログラムを記述する。

```
class NewThread extends Thread {
  public void run() {
    // 別スレッドで実行させたいプログラム
  }
}
```

(3) 次に別のスレッドで、(1) で作ったクラスを new する。
(4) そしてそのインスタンスの start メソッドを実行する。

```
NewThread nt = new NewThread();
nt.start();
// 次の処理
```

すると、(2) で作ったプログラムが実行を開始する。 一方、メインスレッドのプログラムは次の処理へ進むので２本のスレッドが走っていることになる。

別手段によるスレッドの起動

Java では多重継承ができないので、すでに継承をしているクラスは Thread クラスを継承することができない。そのために、次のような手段が提供されている。

(1) Runnable インターフェースを実装したクラスを定義する。
(2) そのクラスの run メソッドに別スレッドで動作させたいプログラムを記述する。

```
class MyRunnable implements Runnable {
  public void run() {
    // 別スレッドで実行させたいプログラム
  }
}
```

(3) 動作中のスレッドで、(1) で作ったクラスを new する。
(4) それを引数にして Thread クラスのインスタンスを生成する。
(5) そのインスタンスの start メソッドを実行する。

```
MyRunnable mr = new MyRunnable();
Thread nt = new Thread(mr);
nt.start();
// 次の処理
```

スレッドとプロセス

並行処理を表わす言葉としてマルチプロセスやマルチタスクがあるが、マルチスレッドはこれらとどう違うのだろうか。 通常は、プロセスやタスクは OS が管理する大がかりな並行処理機構であるのに対し、スレッドはアプリケーション（JavaVM も OS から見ればアプリケーション）が管理する軽量級の並行処理を指す。 つまり 1 つのプロセスの中に複数のスレッドが生成されるという関係になる。

また、プロセスが固有のメモリ空間を持つのに対し、スレッドでは複数のスレッドがメモリ空間を共有している（ただしスタックは各スレッドごとに持つ）。 したがって、スレッドはプロセスに比べて軽快ですが、変数の操作には警戒を要すると言うわけ。

スレッドセーフ

Java ではローカル変数（メソッド内で定義された変数）はスタックに格納される。 上述のようにスタックは各スレッドごとに値を持つので、ローカル変数はスレッドごとの値を持つことができる。 一方、static 変数やインスタンス変数は、複数スレッド間で共有される。

したがって、マルチスレッドで動かすシステムで、後者の変数を扱うときには、他スレッドと競合しないように、必要に応じて他のスレッドからのアクセスを禁止（ロック）しなければならない（ロックについては後述）。 このようにマルチスレッドでも問題なく動作するプログラムの作りを、スレッドセーフと言う。 サーブレットのようにマルチスレッドで動くプログラムは、スレッドセーフに作らねばならない。

並行処理

並行処理といっても、複数の CPU がない限り、本当に同時に処理しているわけではない。 短い時間で処理を振り分け、同時に動いているように見せているだけ。 マルチスレッドは、同時に動いているように見せる処理を実現するのに威力を発揮する。

スレッドの操作

スレッドを操作するメソッドをいくつか説明する。

sleep

sleep メソッドを使うと、スレッドの処理を一時停止させることができる。 停止時間は引数でミリ秒単位で指定する。 sleep メソッドは Thread クラスの static メソッドなので、クラス名.メソッド名で呼び出す。 例えば 3 秒間停止させるときは、次のようにする。

```
Thread.sleep(3000);
```

sleep メソッドは InterruptedException をスローするので、次のように try/catch しなければならない。
catch ブロックの中は何も書かなくてかまわない。

```
try{
  Thread.sleep(3000);
} catch(InterruptedException e) { }
```

interrupt

interrupt メソッドは sleep メソッドなどで停止中のスレッドに割り込み、停止状態を強制的に解除させることができる。 interrupt メソッドは、Thread クラスのインスタンスメソッドなので、次のように、インスタンス名.メソッド名で呼び出す。

```
Mythread mt = new MyThread();
mt.start();
～
mt.interrupt();
```

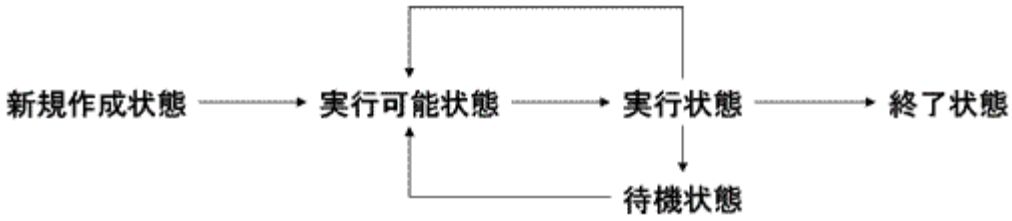
join

join は相手のスレッドが終了するまで待つメソッド。 join はインスタンスメソッドであり、かつ InterruptedException をスローするので、次のように呼び出す。

```
Mythread mt = new MyThread();
mt.start();
～
try {
  mt.join();
} catch(InterruptedException e) { }
// この時点では mt が終了していることが保証されている
```

スレッドのライフサイクル

スレッドは new されると、新規作成状態になり、start メソッドで起動されると実行可能状態になる。 実行可能状態のスレッドを実行状態にするのは、JavaVM のスレッドのスケジューラーの仕事であり、プログラマーは関与することができない。



図解： スレッドのライフサイクル

待機状態

実行状態のスレッドは sleep や後述の wait メソッドが実行されたり、入出力動作で待ちが生じたりすると待機状態になる。 待機状態になった原因が取り除かれてもすぐには実行状態にはならない。 まずは、実行可能状態に移動し、スケジューラーが実行状態にしてくれるのを待つことになる。

強制的な中断

実行状態にあるスレッドが、強制的に実行可能状態に移されることがある。 これもスケジューラーの仕事であり、プログラマーは関与することができない。

終了状態

実行状態のスレッドの実行が進み終了する（プログラムの最後まで行く）と終了状態になる。 終了状態のスレッドは二度と実行可能状態や実行状態になることはできない。

スレッドの優先度

スレッドには優先度がある。 複数のスレッドが、実行可能状態にあるとき、スケジューラーは優先度の高いスレッドから先に実行状態に割り当てようとする。 同じ優先度のタスクが複数あるときは、スケジューラーがそれらを時分割で割り当てようとする。

※ スレッドの強制的な中断や、高い優先度のスレッドの割り当ては、JavaVM が動作する OS の機能により実現されているので、Java 言語の仕様として必ずしも保証されているものではない。

スレッドの同期

共有のデータを複数のスレッドが扱うと、データの競合が発生する可能性がある。 それは以下の処理で、あるスレッドが処理を開始し（2）の状態にあるときに、別のスレッドが（1）の処理を開始するような場合。 これはまさに前述のスレッドセーフでない状況。

（1）共有域のデータを取り込む
（2）取り込んだデータを加工する
（3）加工したデータを共有域に書き込む

ロック

このようなことが起こらないようにするには、あるスレッドが上記の（1）～（3）の処理をしている間は、他のスレッドはこの処理ができないようにロックすれば良い。 このように排他的な操作によって複数のスレッド間で矛盾が起こらないようにすることを、同期をかける（synchronize）と言う。

synchronized キーワード

同期をかけるには、次のようにメソッドに synchronized キーワードを付ける。 するとこのメソッドを含むインスタンスに同期がかかり、他のスレッドはそのインスタンスの synchronized メソッドは実行できなくなる（実行が終わるまで待機状態で待たされます）。 そのインスタンスの synchronized でないメソッドは自由に実行できる。 また、同じクラスから生成されたものでも、別インスタンスのメソッドは、実行できる。

```
synchronized void mathodA() { …          // メソッド単位で設定
```

メソッド全体でなく、メソッドの一部分だけで同期をかけることができる。 それには次のように synchronized ブロックを使う。 この場合同期をかけるオブジェクトを指定する。

```
synchronized (obj) { …                   // ブロック単位で設定
```

同期をかけると、他の処理をロックすることになるので、一般に性能の低下をもたらす。 ロックは必要最小限にする。

デッドロック

同期をかけるときには、デッドロックにならないように注意が必要。 デッドロックとは、2 つのスレッドが互いに相手の同期の開放を待っている状態を指す。 この状態になると、どちらのスレッドも処理が進まず、回復は不可能。 デッドロックは実行時に偶発的に発生する事象であり、コンパイルエラーになったり例外が発生したりすることはない。 ただしデータベースシステムでは同じ状態が続くことを監視して警告を出すことはない。 デッドロックの発生確率は非常に小さいのですが、計算機は非常な高速で処理するので、例え発生確率が 100 万分の 1 でも 1 秒間に 1000 回の処理を行えば、わずか 1000 秒で発生してしまう。 デッドロックは、同期をかける箇所が複数なければ発生しない。 デッドロックを避けるには、同期をかけるオブジェクトの順序を一定にすることがポイント。

スレッド間通信

複数のスレッド間で協調して処理を行う場合、あるスレッドの部分処理が終わるのを、他方のスレッドが待ちたい場合がある。 このような場合に wait/notify/notifyAll メソッドを使う。

wait

wait メソッドは、notify または notifyAll メソッドが呼び出されるまで処理を待機させる。 wait メソッドは Object クラスのメソッドなので、すべてのクラスからそのまま使える。 スレッド間通信で wait メソッドを使うには、synchronized キーワードで同期をとっておかねばならない。 それは wait がこの同期の仕組みを使って待ち合わせをするように作られているから。 また、wait メソッドは InterruptedException をスローするので、次のように try/catch する必要がある。

```
synchronized(this) {
  try{
    wait();
  } catch(InterruptedException e) { }
}
```

wait メソッドは、次のように整数値を引数にとることができる。 こうすると、引数で示した時間後（単位はミリ秒）に実行可能状態なる。 引数がないと、notify/notifyAll が出されるまで、復帰しない。

```
wait(5000)       // 5 秒後に実行可能状態になる
```

notify

notify メソッドは、wait メソッドによって待機状態になっているスレッドを実行可能状態にする。 待機中のメソッドが複数あるときは、そのうちのどれか 1 つだけを、実行可能状態にする。 notify メソッドは Object クラスのメソッドなのですべてのクラスからそのままアクセスできる。 notify メソッドを実行するためには、wait メソッドの場合と同じく、該当オブジェクトのロックを取得していなければならない。 try/catch する必要はない。

```
synchronized(this) {
  notify();
}
```

notifyAll

notify が待機中のメソッドのどれか 1 つだけを実行可能状態にしたのに対し、notifyAll は待機中のすべてのメソッドを実行可能状態にする。 ただし、実際に実行状態になるのはそのうちの、いずれか 1 つだけ。 その他の働きは notify メソッドと同様。

どうやったらオブジェクト指向になるか

Java はオブジェクト指向言語だが、Java を使ったからと言って必ずしもオブジェクト指向が実現できるわけではない。 この様子を以下に例で示す。

【課題】
ある携帯電話会社の料金プランは次の通り。
楽々プラン　基本料金：4000 円　通話料：40 円/分　無料通信分：2000 円
得々プラン　基本料金：6000 円　通話料：25 円/分　無料通信分：4000 円

このとき、以下のユーザの支払額を求めよ。
高橋さん　契約：楽々プラン　通話時間：35 分
鈴木さん　契約：楽々プラン　通話時間：55.5 分
加藤さん　契約：得々プラン　通話時間：200 分

【プログラム 1】

```
public class Payment {
  public static void main(String[] args) {
    charge("高橋", 1, 35);
    charge("鈴木", 1, 55.5);
    charge("加藤", 2, 200);
  }

  static void charge(String name, int plan, double usedMinute) {
    int sum = 0;
    if(plan == 1) {
      double chouka = 40 * usedMinute - 2000;
      if(chouka<0) sum = 4000;
      else sum = (int)(4000 + chouka);
    } else if(plan == 2) {
      double chouka = 25 * usedMinute - 4000;
      if(chouka<0) sum = 6000;
      else sum = (int)(6000 + chouka);
    }
    System.out.println(name + "さんの今月の支払額は" + sum + "円です。");
  }
}
```

簡明でわかりやすい立派なプログラムだと思う。 この課題だけに限れば、このプログラムで十分。
しかし、実際に業務で使うプログラムとしては問題がある。

それは拡張性。 この課題には、今後、次のような拡張が考えられる。

１．着信中心のユーザや、ヘビーユーザ用の新プランの開設
２．通話相手や時間帯による通話料の相違など料金計算ロジックの多様化
３．メールサービスや支払い処理など通話料金計算以外の業務処理への拡張

これに対応できるようにしたのが、以下のプログラム 2。

【プログラム 2】

```
public class Payment {
  public static void main (String[] args) {
    User takahashi = new User("高橋", new RakurakuPlan(35));
    User suzuki = new User("鈴木", new RakurakuPlan(55.5));
    User katou = new User("加藤", new TokutokuPlan(200));
    takahashi.printPayment();
    suzuki.printPayment();
    katou.printPayment();
  }
}

class User {
  String name;
  Plan plan;
  public User(String n, Plan p) {
    name = n;
    plan = p;
  }
  void printPayment() {
    System.out.println(name +"さんの今月の支払額は"+ plan.charge() +"円です");
  }
}

class Plan {
  int base, noCharge, chargePerMinute;
  double usedMinute;
```

```
  public int charge() {
    int sum;
    double chouka = chargePerMinute * usedMinute - noCharge;
    if(chouka<0) sum = base;
    else sum = (int)(base + chouka);
    return sum;
  }
}

class RakurakuPlan extends Plan{
  RakurakuPlan(double used) {
    base = 4000;
    noCharge = 2000;
    chargePerMinute = 40;
    usedMinute = used;
  }
}

class TokutokuPlan extends Plan{
  TokutokuPlan(double used) {
    base = 6000;
    noCharge = 4000;
    chargePerMinute = 25;
    usedMinute = used;
  }
}
```

このプログラムはクラスに分割され、継承やオブジェクトコンポジションが使われている。 20 行から 50 行とサイズも 2 倍以上になった。 しかし、このプログラムでは、先に示した拡張性の問題（の一部）が次のように改善されている。

1. 新プランへの対応は Plan クラスを継承したサブクラスを作ることで可能。
2. 料金計算ロジックの拡張は、このプログラムでは困難。 ここでは示さないが、Plan をクラスからインターフェースにして、実装を下位のクラスに任せると、料金計算ロジックの変更にも対応できる。
3. この構造では、現業務に影響せずに、他の業務処理を追加するのは容易。
   例えばメールの料金計算を追加するなら、MailService を作り User クラスに取り込む（オブジェクトコンポジション）ことで、他の処理とは独立したプログラムとすることができる。

ただし、本当に拡張する可能性があるかを見極めることは重要。 ありもしない拡張に備えてプログラムを複雑にするのはばかげている。

[END]