第15章

例外

プログラムを設計するにあたっては、
実行時に想定外の事態が発生する可能性があることを
考慮に入れておく必要があります。
そして、そのような場合にも異常終了や誤動作しないよう
備えておかなければなりません。
Java には想定外の事態に対処する「例外」が備わっています。
本章をしっかり学習し、どのような状況でも安定して動く
高品質のプログラムを開発できるようになりましょう。

CONTENTS

15.1 エラーの種類と対応策

15.1.1 不具合のないプログラムを目指す

最近やっと Java に慣れてきて、エラーがあまり出ないようになりました！

それはよかった。でも重要なプログラムを作る場合は「あまり」ではダメだよ。

Java が誕生して 20 年以上が経ち、今では金融機関や官公庁などの社会基盤を支える情報システムを作る際にも Java が使われるようになりました。しかし、これらの大規模なシステムに不具合が生じると、時に大きな社会問題や損害賠償につながる可能性があります。

ある程度の経験を積めば「とりあえず動作するプログラムを作ること」は難しくありません。本当に難しいのは、「想定外の事態やユーザーの誤操作などがあっても、エラーを起こさず正常に動作するプログラムを作ること」なのです。

不具合のないプログラムを目指す

動くコードは書けて当たり前。不具合対策こそが、腕の見せどころ。

キビシイなぁ…。

何、言ってるのよ。途中で止まっちゃうゲームなんて、誰も遊んでくれないわよ！

15.1.2 3種の不具合と対処法

プログラムが想定どおりに動かないことを総じて不具合と言いますが、Javaプログラムの場合は大きく3つに分類できます。

①文法エラー（syntax error）

文法の誤りによりコンパイルに失敗します。代表例はセミコロン忘れ、変数名の間違い、privateメソッドを外部から呼び出す、などです（図15-1）。

```
>javac SyntaxError.java

SyntaxError.java:4: ';' がありません。

        }
        ^

エラー 1 個
```

図 15-1　文法エラーの例

②実行時エラー（runtime error）

実行している最中に何らかの異常事態が発生し、動作が継続できなくなるエラーです。Javaの文法としては問題がないためコンパイルは成功し、実行もできますが、実行中にエラーメッセージが表示されて強制終了します。代表例は配列の範囲外要素へのアクセス、0での割り算、存在しないファイルのオープン、などです（図15-2）。

```
>javac RuntimeError.java
>java RuntimeError
プログラムを開始します。処理を3つ実行します。
処理1を完了。
処理2を完了。

Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 17
at RuntimeError.main(RuntimeError.java:11)
```

図 15-2　実行時エラーの例

③論理エラー（logic error）

Java の文法に問題はなく、強制終了もしません。しかし、プログラムの実行結果が想定していた内容と違っています。代表例は、電卓ソフトを作ったが計算結果がおかしい、などです（図 15-3）。

```
>javac LogicalError.java
>java LogicalError
プログラムを開始します。
3+5を計算します。
計算完了：　答えは35
プログラムを正常終了します。
```

図 15-3　論理エラーの例

開発者は、これら 3 種類の不具合に対して、それぞれ異なる対策を行う必要があります。その不具合の検出と解決についてまとめたものが次の図 15-4 です。

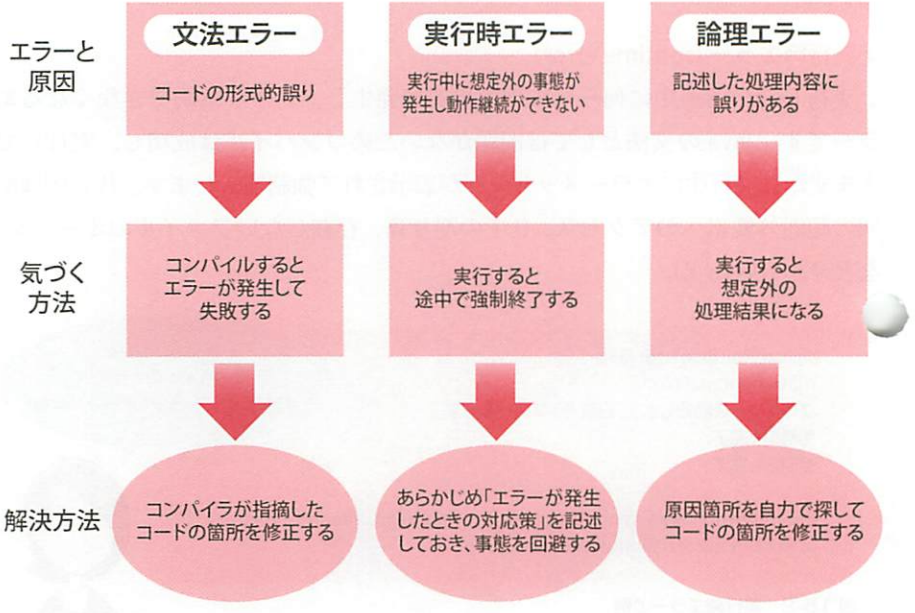


図 15-4　各不具合に対する解決方法

文法エラーと論理エラーは、対策が似ていますね。

いいところに気づいたね。では、残る実行時エラーについて、少し深く考えてみよう。

15.1.3 例外的状況

文法エラーと論理エラーは「開発者の過失」であって、**開発者が開発時にテストをしっかり行い、コードを修正することで、本番での発生を予防できるもの**です。しかし、実行時エラーはそうはいきません。

そもそも実行時エラーは、「プログラム実行中に想定外の事態が発生したこと」によって起こります。この「想定外の事態」のことを**例外的状況**(exceptional situation)または単に**例外**(exception)といいます。例外的状況には次のようなものがあります。

■パソコンのメモリが足りなくなった

開発用のコンピュータには十分な容量のメモリがあったが、本番用コンピュータのメモリが少なく、動作中にメモリが足りなくなってしまった。

■存在すべきファイルが見つからない

動作中に data.txt というファイルを読み込んで動くプログラムを開発したが、利用者が誤ってファイルを削除してしまった。

■ null が入っている変数を利用しようとした

ユーザーが想定外の操作を行ったことが原因で、本来は変数に入るはずのない値(null など)が入り、その変数を使用するメソッドを呼び出してしまった。

これらすべての状況に共通するのは、**プログラマがソースコードを作成する時点では例外的状況の発生を予防できない**ということです。

「nullが入っている変数を利用する」という例外的状況は、事前にif文でチェックすれば「予防できる」と思いますけど？

朝香さんはプログラムに含まれる、すべてのメソッド呼び出しでnullチェックを行うのかい？

あ…理論的には可能でも現実的ではないですね。

プログラマが「例外的状況の発生を防ぐこと」は困難です。しかし、プログラマは無力ではありません。「もし例外が発生したときに、どのように対処するか」という対策を用意しておくことが可能です。たとえば、「もし目的のファイルが見つからなければ、ユーザーに代わりのファイル名を入力してもらう」など、異常事態に備えた代替策を準備しておけばよいのです。このように例外的状況に備えて対策を準備し、その状況に陥った際に対策を実施することを**例外処理**(exception handling)と呼びます。

15.2 例外処理の流れ

15.2.1 従来型例外処理の問題点

パソコンでプログラムを動かしていたら強制終了したことがありました。きっと作者は例外処理をしていなかったのね。

そうかもしれないね。でも、昔は例外処理をきちんと書くことが、とても大変だったんだ。

Java が生まれる以前から例外処理はプログラマにとって取り組むべき重要な課題でした。たとえば Java の祖先にあたる C 言語の場合、「ファイルに『hello!』と書き込む」だけの簡単なプログラムは図 15-5 のように書きます。

```
/* ファイルを開く(失敗したら戻り値は定数NULL)*/
FILE fp = fopen("c:¥test.txt");                    本来の処理
if(fp == NULL) {
  printf("エラーです。終了します。");
  exit(1);                                         例外処理
}

/* ファイルに文字列を書き込む*/
fputs("hello!", fp);                               本来の処理

/* ファイルを閉じる(失敗したら戻り値は定数EOF)*/
int c = fclose(fp);                                本来の処理
if(fp == EOF) {
  printf("エラーです。終了します。");
  exit(1);                                         例外処理
}
```

図 15-5　C 言語での例外処理の例

実はこのプログラムの「本来の処理」は、①ファイルを開く、②ファイルに文字を書き込む、③ファイルを閉じる、この３つだけです。しかし、命令を呼び出すたびに「もしもファイルが開けなかったら…」あるいは「もしもファイルに書けなかったら…」などの例外的状況を１つひとつチェックしているため、プログラムがわかりにくくなっています。そのため、C 言語のような古いプログラミング言語の例外処理には次のような問題点があります。

- 「本来の処理」が、どの行なのかわかりづらい。
- 命令を呼び出すたびに１つひとつチェックしなければならない。
- めんどうなので例外処理をサボって書かないおそれがある。

本来たった３行のプログラムがこんなに長くなってしまうなんて…。本格的なプログラムならもっと酷いことになりそう…。
それで、つい例外処理をサボったプログラムになってしまうのね。

このような問題が発生するのは、従来型のプログラミング言語が**例外的状況発生の検知と対応に関する全責任をプログラマに求めている**からです。

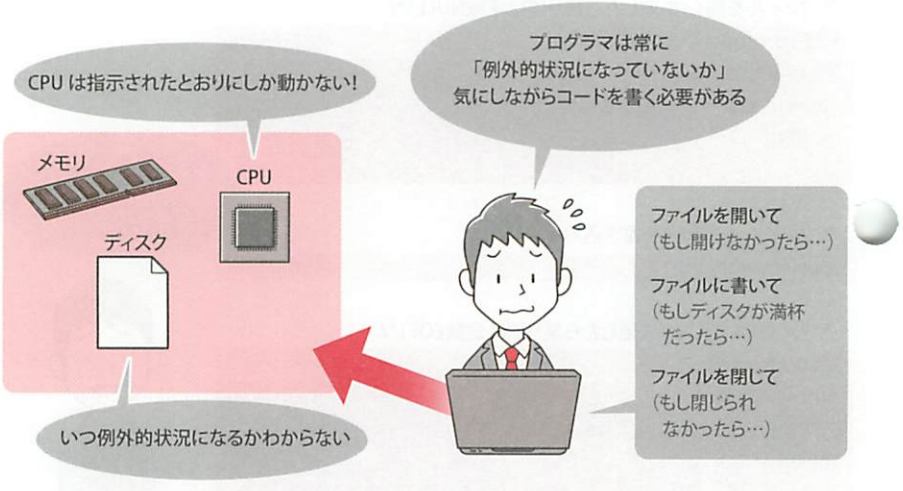


図 15-6　従来型のプログラミング言語での例外処理ではプログラマの責任は重大だ

まじめなプログラマは責任を果たすために 1 つひとつの命令からの戻り値をチェックするめんどうを引き受けて苦しみますし、不まじめなプログラマは責任を放棄して例外処理をサボってしまうのです。

15.2.2 新しい例外処理の方法

このような従来型の例外処理の問題点を解決するために、Java をはじめとする新しいプログラミング言語では、例外処理専用の文法としくみが備わっています。

先ほどの「ファイルに『hello!』と書き込む」を Java で書くと次の図 15-7 のようになります。やっていることは先ほどの C 言語の例と同じで、①ファイルを開く、②ファイルに書く、③ファイルを閉じる、の 3 つです。

```
try {
    FileWriter fw = new FileWriter("c:¥¥data.txt");
    fw.write("hello!");                          本来の処理
    fw.close();

} catch(IOException e) {
    System.out.println("エラーです。終了します。");
    System.exit(1);                              例外処理
}
```

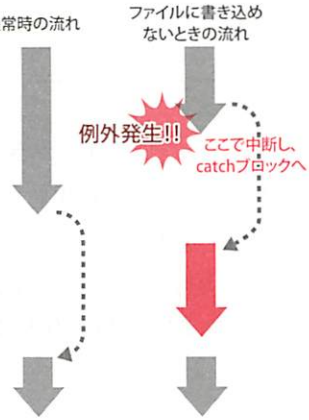


図 15-7　Java での例外処理の例では本来の処理と例外処理がハッキリ分かれている

このプログラムでは「java.io.FileWriter」という、まだ紹介していないクラスを使っているが、「ファイルに書き込むための API」だと理解してほしい。

この例にあるように、Java では例外処理に try と catch という 2 つのブロックを使用します（合わせて try-catch 文と呼びます）。

try と catch、2 つのブロックのうち通常、実行されるのは try ブロックだけで、

catch ブロックの処理は動きません。ただし、try ブロック内を実行中に例外的状況が発生したことを JVM が検知すると、処理は直ちに catch ブロックに移行します。つまり、catch ブロックの中には「例外的な状況が発生したときに実行される処理」を記述しておくのです。

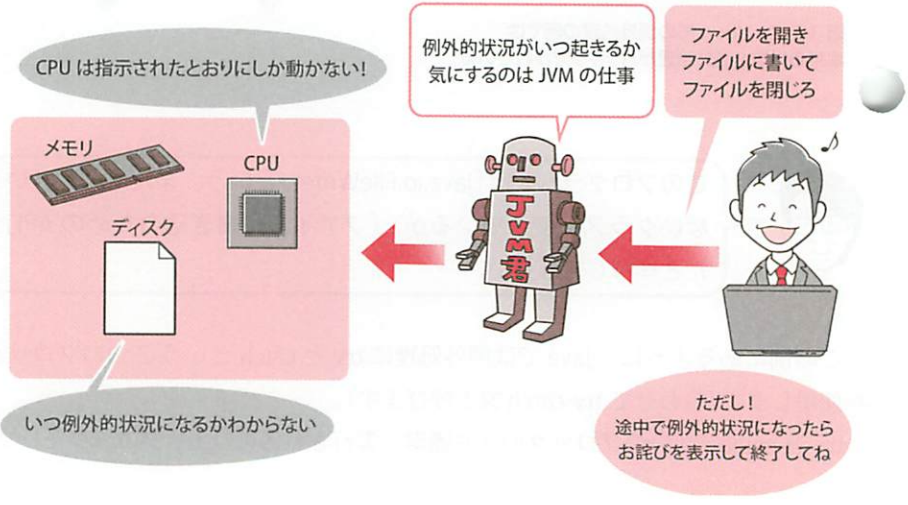
Java における例外処理の基本パターン

```
try {
    通常実行される文
} catch( … ) {
    例外発生時に実行される文
}
```

見方を変えれば、try ブロックとは「この部分では例外的状況が発生する可能性があるから、その検出を試みながら実行しろ」というプログラマから JVM への指示ともいえます。

命令を実行するたびに「例外的な状況が発生しているか？」をチェックするめんどうな作業は JVM に任せることで、プログラマが負う責任は軽減されます。

図 15-8　JVM が例外を検知したら処理を切り替えてくれる



15.3 例外クラスとその種類

15.3.1 例外を表すクラス

先輩。図 15-7 (p.569) のコードにある catch の直後に書かれた「IOException」って何ですか？

それを理解するために、例外クラスについて紹介しよう。

一口に例外的状況といっても、「ファイルがない」「メモリが足りない」「変数が null」など、さまざまな状況があります。それらを同じものとして扱うと、「発生した例外的状況に応じた処理を行う」ことができません。そこで Java では、発生した例外を区別できるように、それぞれの例外的状況を表すクラスが複数、準備されています。

「例外的状況をクラスにした」という意味がわかりません…。

大丈夫、落ち着いて第 II 部を思い出そう。

第 II 部で学んだように、オブジェクト指向は「現実世界の何か」をオブジェクトとして Java 仮想世界で再現したものでした。多くのオブジェクトは、ヒトやモノなど「現実世界で形があるもの」から作り出されることが一般的です。

しかし「現実世界で形がないもの」からオブジェクトを作ることもあります。たとえば、イベント運営会社の「イベント情報管理プログラム」を開発する場合、本来は形があるものと見なさない「イベント」を Event クラス（フィールドとして開催日や主催者を持つ）として作るでしょう。

同様に、「ファイルがなくて困っている状況」や、「nullが入っていて困っている状況」など現実世界の例外的状況（想定外の事態）をクラスにしたものが例外クラスです。

ちなみにjava.lang.IOExceptionは「ファイルの読み書きなどの入出力ができなくて困っている状況」のためのクラスであり、ほかにも多くの例外クラスがAPIとして定義されています。

図15-7は「ファイルが読み書きできない例外的状況に陥ったら、catchブロックの中身を動かせ」という指示なんですね。

そのとおりだ。詳しくは15.4節で説明しよう。

15.3.2 例外の種類

APIで提供されている例外クラスは、次の図15-9のような継承階層を構成しています。

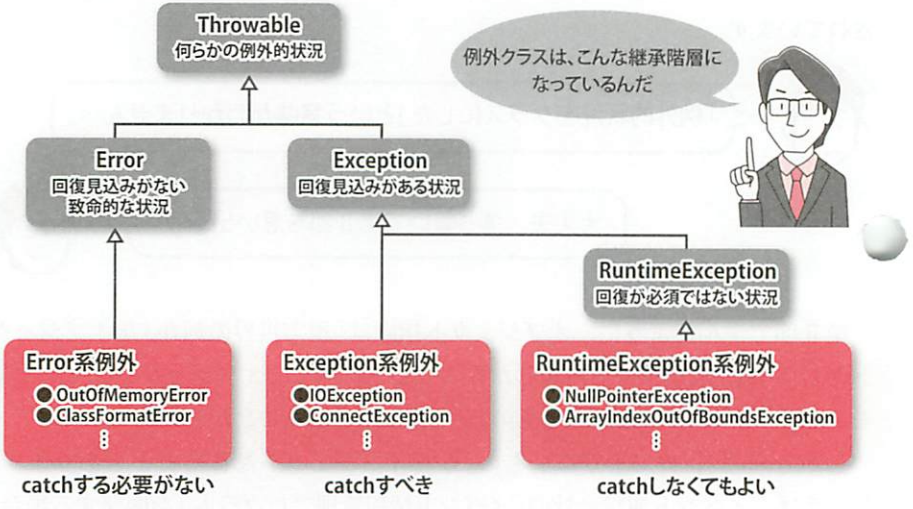


図15-9　例外クラスの継承階層

① Error 系例外

java.lang.Error の子孫で**回復の見込みがない致命的な状況**を表すクラスです。代表的なものに OutOfMemoryError（メモリ不足）や ClassFormatError（クラスファイルが壊れている）があります。通常、このような状況をキャッチしても打つ手はないため、キャッチする必要はありません。

② Exception 系例外

java.lang.Exception の子孫（RuntimeException の子孫を除く）で、**その発生を十分に想定して対処を考える必要がある例外的状況**を表すクラスです。たとえば、IOException（ファイルなどが読み書きできない）や ConnectException（ネットワークに接続できない）といった状況は、ファイルやネットワークを利用する際に当然、想定しておくべき事態です。

③ RuntimeException 系例外

java.lang.RuntimeException クラスの子孫で、**必ずしも常に発生を想定すべきとまではいえない例外的状況**を表すクラスです。たとえば、NullPointerException（変数が null である）や ArrayIndexOutOfBoundsException（配列の添え字が不正）のように、いちいち想定していると「きりがない」ものが多く含まれます。

15.3.3 チェック例外

API には多くの例外クラスが定義されていますけど、これだけたくさんの例外的状況が起こる可能性がある、ということですよね…。

めんどうくさいなぁ…。ボクはサボって try-catch とか書かないと思います。

そうは問屋、もとい JVM が卸さないんだよ。

先ほどの3種類ある例外クラスの中で、特に注目してほしいのは②のException系例外です。これは「その発生を十分に想定して対処を考えておく必要がある状況」を表しているのですから、**いざ例外が発生したときに何も対処されないということはあってはならない**はずです。

そのためJavaでは、**Exception系の例外が発生しそうな命令を呼び出す場合、try-catch文を用いて「例外が発生したときの代替処理」を用意しておかないとコンパイルエラー**になります。

リスト15-1　例外処理を用意していないと…

Main.java

```
1  import java.io.*;
2  public class Main {
3    public static void main(String[] args) {
4      // FileWriterのコンストラクタは、IOExceptionを発生させる
5      // 可能性があります。しかしtry-catchでは囲みません
6      //  （失敗時にどうするか、考えていない）。
7      FileWriter fw = new FileWriter("data.txt");
8    }
9  }
```

コンパイル結果

```
Main.java:7: 例外 java.io.IOException は報告されません。スローするにはキャッチまたは、スロー宣言をしなければなりません。
    FileWriter fw = new FileWriter("data.txt");
                    ^
```

そこで次のようにtry-catch文を用いてIOExceptionの発生に備えれば、コンパイルエラーはなくなります。

リスト15-2　try-catch文でException系例外の発生に備える

Main.java

```
1  import java.io.*;
```

```
2  public class Main {
3    public static void main(String[] args) {
4      try {
5        FileWriter fw = new FileWriter("data.txt");
6
7
8      } catch (IOException e) {
9        System.out.println("エラーが発生しました。");
10     }
11   }
12 }
```

FileWriter のコンストラクタは、IOException を発生させる可能性がある。

例外的状況になったときに備えて記述された代替処理

このように、Exception 系例外は、例外発生時の対策が用意されているかをコンパイルの時点でチェックされるため、**チェック例外**（checked exception）とも呼ばれます。

必ず try-catch 文を書かないと実行できないなんで、めんどうですよぉ…。

誰かさんみたいに例外処理を書かない人がいるから「サボれないしくみ」になっているのね。

3 つの例外クラスのグループとキャッチの強制

- Error 系例外　　try-catch 文でキャッチする必要はない。
- Exception 系例外　　try-catch 文でキャッチしないとコンパイルエラー。
- RuntimeException 系例外　　try-catch 文でキャッチするかは任意。

15.3.4 発生する例外の調べ方

でも、どの命令を呼んだら、どんなエラーが発生するかなんて想像もつかないし、書きようがないじゃないか。

確かにそうよね…。リスト 15-1 では当然のように「FileWriter のコンストラクタを呼ぶと IOException を発生させる可能性がある」って書いてあったけど…。

ここまで、FileWriter を用いてファイルにデータを書き込むプログラムを例に解説してきました。しかし、API に含まれるクラスには、他にも「呼び出すと何らかの例外を発生させる可能性があるメソッド」が数多くあります。

特にチェック例外が起きる可能性のあるメソッドを呼び出す場合は try-catch 文で囲まなければならないので、**「どのメソッドを呼び出したら、どのような例外が発生する可能性があるか」をあらかじめ知っておく必要があります。**

実は「どのクラスの、どのメソッドが、どのような例外を発生させる可能性があるか」という情報は、API リファレンスに掲載されています。

FileWriter

public **FileWriter**(String fileName)
throws IOException

ファイル名を指定して FileWriter オブジェクトを構築します。

図 15-10　FileWriter クラスのコンストラクタ

メソッドやコンストラクタを呼び出した際に Exception 系の例外が発生する可能性がある場合、**引数リストの後に「throws 例外クラス名」と表記されます。**図 15-10 は、API リファレンスから FileWriter クラスの解説を抜粋したものですが、「throws IOException」との記載があるので、「FileWriter のコンストラクタを呼び出す（インスタンスを生成する）ときには、IOException をキャッチする try-catch 文が必要になる」と理解すればよいのです。

15.4 例外の発生と例外インスタンス

15.4.1 例外インスタンスの受け渡し

IOException が何かはわかりましたが「catch(IOException e)」の e って何ですか？

その解説をしていなかったね。try-catch 文の構文を振り返りながら e の役割を説明していこう。

ここで再び図 15-7（p.569）のコードにおける catch ブロック部分に注目してください。

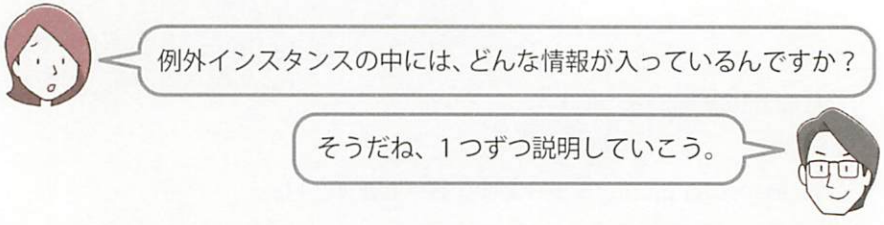
```
try {
  ⋮
} catch (IOException e) {
  ⋮
  System.out.println("エラーです。終了します。");
  System.exit(1);
}
```

15.2 節で説明したように、try ブロック実行中は JVM が例外的状況の発生を監視しながらプログラムを実行します。そして、いざ例外が発生すると、JVM は処理を catch ブロックに移行します。このとき JVM は、「プログラムの中のどこで・どのような例外が起きたのか」という**例外的状況の詳細情報が詰め込まれた IOException インスタンスを catch 文で指定された変数 e に代入します。**

図 15-11　例外インスタンスの受け渡し

catch ブロックの中では、この変数 e に格納された詳細情報を取り出して、適切なエラー処理（画面にエラーメッセージとして表示するなど）を行うことができます。

15.4.2 例外インスタンスの利用



例外インスタンスに格納されている詳細情報は、その例外の種類によって異なります。しかし、すべての例外は「例外的情報の解説文」と「スタックトレース」の情報を必ず持っており、それぞれ以下のメソッドで取得と表示ができます。

表 15-2　例外インスタンスが必ず備えているメソッド

メソッド	意味
String getMessage()	例外的状況の解説文（いわゆるエラーメッセージ）を取得する。
void printStackTrace()	スタックトレースの内容を画面に出力する。

図 15-11 のコードでも e.getMessage() がありますね。

そうだね。この場合、もし例外が発生すると画面に「エラー：c:¥data.txt（アクセスが拒否されました）」などが表示されるよ。

スタックトレースとは、「JVM がプログラムのメソッドを、どのような順序で呼び出し、どこで例外が発生したか」という経緯が記録された情報です。「e.printStackTrace();」という文を catch ブロック内に記述すれば、その内容を画面に表示できます。

ところでみなさんは、「スタックトレースがどのような情報であるか」をすでにご存じのはずです。

```
java.io.IOException: data.txt (アクセスが拒否されました。)
  at java.io.FileInputStream.open(Native Method)
  at java.io.FileInputStream.<init>(FileInputStream.java:120)
  at java.io.FileInputStream.<init>(FileInputStream.java:79)
  at java.io.FileReader.<init>(FileReader.java:41)
  at Main.main(Main.java:6)
```

実行時エラーが発生したときのエラー画面って、発生した例外のスタックトレースの内容だったんですね。

そうだよ。例外が発生しても try-catch 文でキャッチされなかった場合、JVM がプログラムを強制停止してスタックトレースの内容を画面に表示していたんだ。なお、スタックトレースの詳しい読み方は付録 C を参照してほしい。

15.5 さまざまなcatch構文

15.5.1 try-catch構文の基本形

例外の種類やしくみについて基礎は理解できたね。ここからはtry-catch文の詳細な構文を解説しながら、例外の捉え方を紹介しよう。

それでは例外処理の基本構文を復習した上で、さまざまな構文のバリエーションを見ていきましょう。まずは基本構文です。

try-catchの基本構文

```
try {
  本来の処理
} catch ( 例外クラス　変数名 ) {
  例外が発生した場合の処理
}
```

なお、例外インスタンスを受け取るための変数名は自由ですが、慣習としてeやexが使用されます。

15.5.2 2種類以上の例外をキャッチする

前節で紹介したtry-catchの基本構文でキャッチできるのは1種類の例外だけでした。しかし、図15-12のようにcatchブロックを複数記述することもできます。

JVMは発生した例外の型に対応するcatchブロックを上から順に検索し、最初にキャッチできたcatchブロックに処理を移します。

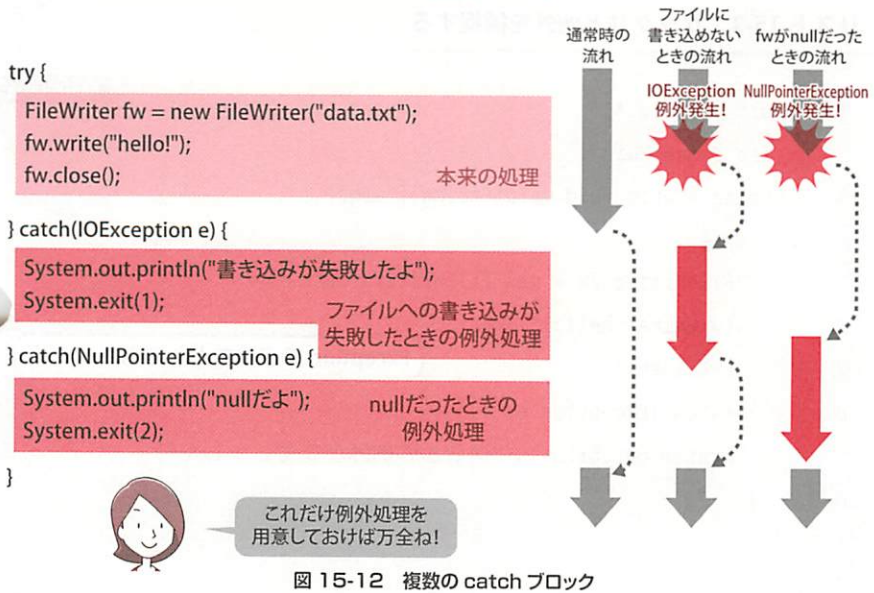


図 15-12　複数の catch ブロック

IOExceptionをキャッチしたときは…、NullPointerExceptionをキャッチしたときは…、というif文みたいですね。

なお、図15-12のコードではcatchブロックを2つ記述していますが、IOExceptionとNullPointerExceptionのどちらを捕まえても同じ処理をする場合、「catch(IOException | NullPointerException e) { … }」という記述でcatchブロックを1つにまとめることが可能です。

15.5.3 ザックリと例外をキャッチする方法

catchブロックに指定する例外クラスは、第13章の多態性で学んだ「ザックリ捉えた型」でも構いません。

例外クラスの継承階層図(p.572の図15-9)によれば、IOExceptionもNullPointerExceptionも、ザックリ捉えればどちらもExceptionです。よって次のように記述することで、どちらの例外が発生しても1つのcatchブロックでキャッ

チできます。

リスト15-3 ザックリと例外を捕捉する

Main.java

```
1  import java.io.*;
2  public class Main {
3    public static void main(String[] args) {
4      try {
5        FileWriter fw = new FileWriter("data.txt");
6        fw.write("hello!");
7        fw.close();
8      } catch (Exception e) {
9        System.out.println("何らかの例外が発生しました");
10     }
11   }
12 }
```

Exceptionの子孫をどれでもキャッチ

これなら発生する例外の種類が増えてもcatchブロックを増やさなくていいから楽ですね。

しかし、どのような種類の例外が発生しても同じように処理するから、大ざっぱな例外処理になってしまうね。

15.5.4 後片付け処理への対応

実はリスト15-3には致命的なバグがあるのですが、どこが問題なのかわかりますか？ 答えは「7行目のファイルを閉じる処理が動かないことがある（ファイルが開いたままになることがある）」です。

ファイルは「開いたら閉じる」のが決まりです。あるプログラムがファイルを開いている間は、ほかのプログラムからそのファイルを使えないことがあるので「閉じ忘れ」はあってはならないのですが、リスト15-3では、それが起こってしまいます。

たとえば6行目の「fw.write("hello!");」を実行したときに、偶然ディスクの容量が満杯になりIOExceptionが発生したとしましょう。するとプログラムの処理はcatchブロックに移動してしまうため7行目のfw.close()は実行されず、ファイルは開いたままになってしまいます。

close()は「例外が起きても、起きなくても」必ず実行しなければならない処理ですよね。

こうすればいいんじゃない？

リスト 15-4　朝香さんが作成したプログラム

Main.java

```
1  import java.io.*;
2  public class Main {
3    public static void main(String[] args) {
4      FileWriter fw = null;
5      try {
6        fw = new FileWriter("data.txt");
7        fw.write("hello!");
8      } catch (IOException e) {
9        System.out.println("エラーです");
10     }
11     fw.close();
12   }
13 }
```

try-catch の後で close する

このプログラムなら問題ないように思えます。しかし、もしtryブロックの中でNullPointerExceptionなどが発生した場合、例外はキャッチされないので、ファイルを閉じないままプログラムは強制終了してしまいます。なお、リスト15-4の11行目ではコンパイルエラーが発生します。fw.close()はIOExceptionを送出する可能性があるため、11行目を単独でtry-catchで囲まなければなりません。

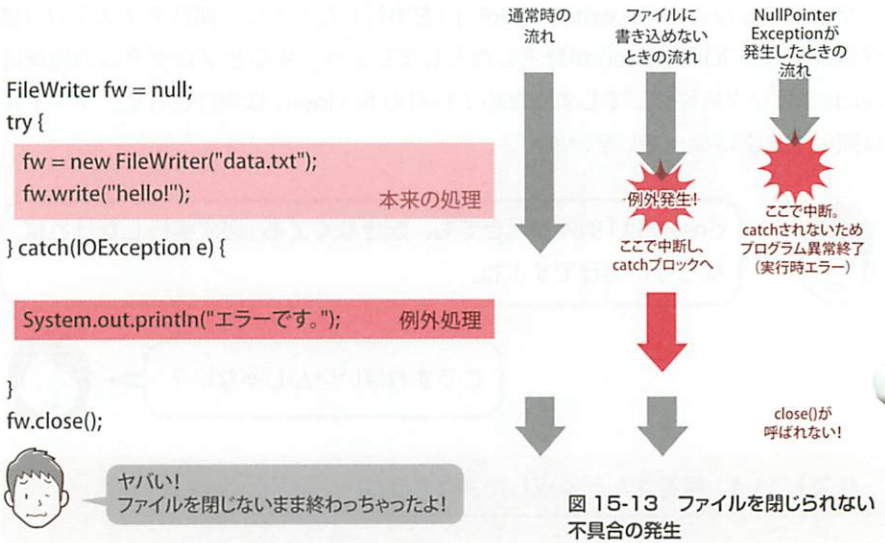


図 15-13　ファイルを閉じられない不具合の発生

この例で取り上げた「fw.close();」のような後片付け処理は、「例外が発生しても、またはしなくても、たとえ強制終了になるときでも、必ず実行しなければならない処理」です。そして、そのような処理をJVMに確実に実行させるために、次のfinallyブロックがあります。

例外発生の如何を問わず必ず処理を実行する

```
try {
  本来の処理
} catch ( 例外クラス　変数名 ) {
  例外が発生した場合の処理
} finally {
  例外があってもなくても必ず実行する処理
}
```

try-catch-finally 構文では、一度 JVM が try ブロックの実行を開始したら、必ず最後に finally ブロックの内容も実行されることが保証されています。

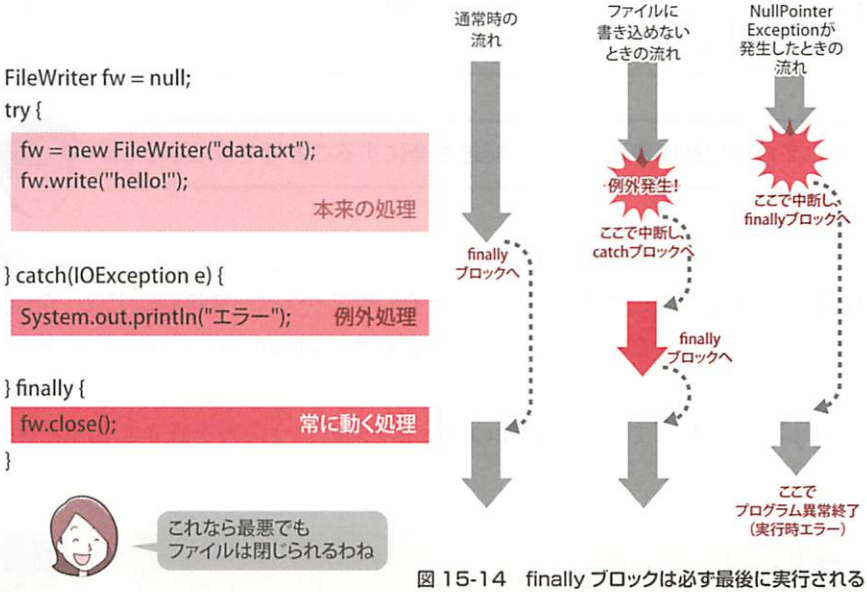


図 15-14　finally ブロックは必ず最後に実行される

開いたファイルを閉じる、開いたデータベースやネットワークとの接続を閉じるなど、「後片付け処理」には必ず finally を使います。

必ず finally を使うべき状況

後片付け処理は、必ず finally ブロックに記述する（ファイル・データベース接続・ネットワーク接続など）。

15.5.5 自動的に close() が呼ばれる try-catch 文

finally での後片付けが大事なことはわかるけど、やっぱめんどうだなぁ…。なんかこう、JVM が気をきかせて自動で後片付けしてくれないんですか？

さすがに自動は無理だけど、記述を楽にすることはできるよ。

Java7 以降では、try の直後に丸カッコで囲まれた複数の文を記述することが可能になりました。ここで開かれたファイルやデータベース接続などは、finally ブロックを記述しなくても Java によって自動的に close() メソッドが呼び出されます。

この構文を活用して、図 15-14 のコードを次のように書き直すことができます。

```
try (
  FileWriter fw = new FileWriter("data.txt");
) {
  fw.write("hello!");
} catch(IOException e) {
  :
}
```

try-catch 文を抜ける際に、自動的に close() が呼び出される。finally ブロックの記述は不要

なお、JVM によって自動的にクローズされるのは、java.lang.AutoClosable インタフェースを実装している型に限られます。ファイル操作やデータベース接続、ネットワーク接続に用いる API クラスの多くは、AutoClosable を実装しているため、この節で紹介した方法での簡潔な記述が可能です。

15.6 例外の伝播

15.6.1 main メソッドで例外をキャッチしないと…

ここまでは 1 つのメソッドに注目して例外処理を解説してきたね。では、main メソッドから呼び出した先のメソッドで例外が発生したらどうなると思う？

うーん、単に、呼び出されたメソッドの中で異常終了するだけなんじゃないかなぁ。

この章の冒頭で解説したように、例外が発生したにも関わらずキャッチしないと実行時エラーとなり、プログラムは強制終了してしまいます。「例外が起きても何もできない＝お手上げ」となり、JVM はスタックトレースを画面に表示して（しかたなく）強制終了するのです。それでは、main メソッドではなく、呼び出した先のメソッドで例外が発生したときはどのような動作になるのでしょうか。次のプログラムを例に考えてみましょう。

- main メソッドの中では sub() メソッドを呼んでいる。
- sub() メソッドの中では subsub() メソッドを呼んでいる。
- subsub() メソッドでは、処理中に何らかの例外が発生することがある。

subsub() メソッド実行時に例外が発生すると、以下のようなことが JVM の中で起きます（図 15-15）。

① まず subsub() メソッドで例外をキャッチしていなければ（try-catch 文がなければ）、「subsub メソッドとしては、この例外的状況に対してお手上げ」となり、呼び出し元の sub メソッドに対応が委ねられます。
② sub() メソッドでもキャッチしなければ、例外の対応は main メソッドに委ねられます。

③ mainメソッドで例外をキャッチしなければ強制終了します。

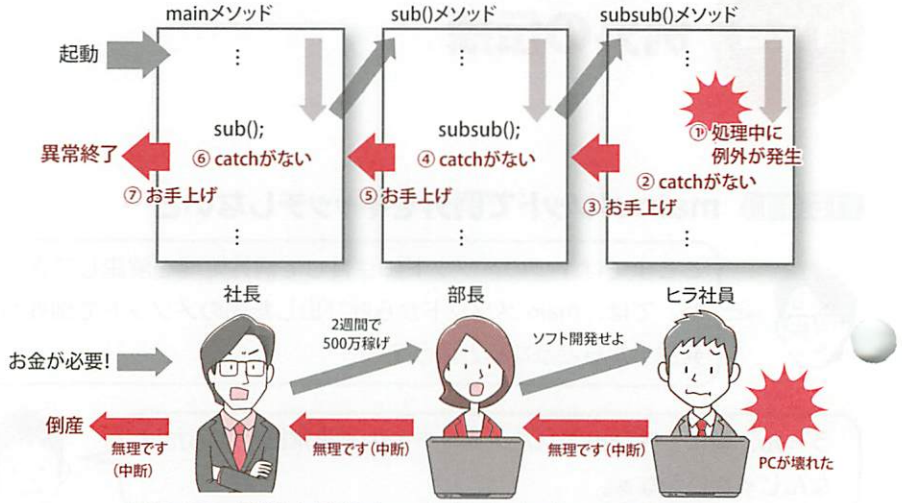


図 15-15　発生した例外はキャッチしないとプログラムが強制終了する

このように例外はキャッチされない限り、メソッドの呼び出し元まで処理を「たらい回し」にされてしまいます。この現象を**例外の伝播**と呼びます。もちろん呼び出し元のmainメソッドやsub()メソッドにcatchブロックが準備されていれば、例外の伝播はそこで止まります(図15-16)。

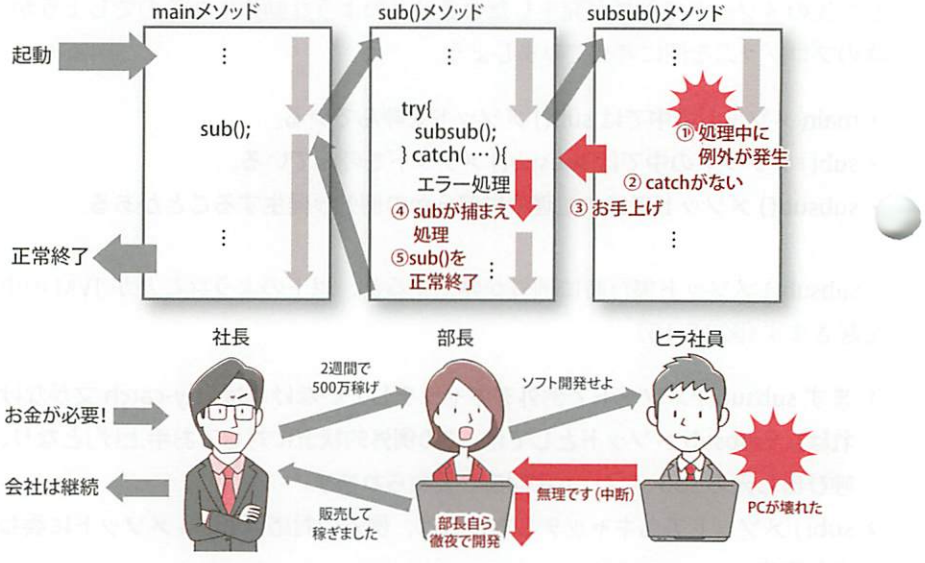


図 15-16　発生した例外は呼び出し元メソッドに処理を委ねる

15.6.2 チェック例外の伝播とスロー宣言

例外の伝播は発生した例外が各メソッドでキャッチされず「お手上げ」になるため起こります。しかし、Exception 系例外（チェック例外）は try-catch 文によるキャッチが必須（15.3.3 項）なため「お手上げ」になることはなく、基本的に例外の伝播は起こりません。

しかし、メソッドを宣言する際にスロー宣言を行うことで、発生するチェック例外を呼び出し元へと伝播させることが許可されます。

スロー宣言による例外伝播の許可

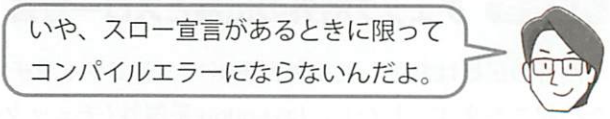
アクセス修飾　戻り値　メソッド名（引数リスト）
　　throws 例外クラス 1, 例外クラス 2,… {
　メソッドの処理内容
}

たとえば次のように記述します。

```
public static void subsub() throws IOException {
  // IOExceptionが発生する可能性があるが、
  // try-catch文がなくてもOK
  FileWriter fw = new FileWriter("data.txt");
}
```

スロー宣言

あれ？ この subsub() メソッドって、実行中にチェック例外が起こるはずなのに try-catch がないですよ。「サボり禁止」でコンパイルエラーになりますよね。



このメソッドの中ではFileWriterをインスタンス化しており、チェック例外IOExceptionが発生する可能性があります。しかし、この例のように**スロー宣言を行っていればtry-catch文がなくてもコンパイルエラーになりません。**

スロー宣言によるチェック例外の伝播

メソッドを定義する際、自らキャッチしないチェック例外をthrowsで宣言することができる。このときメソッド内でtry-catch文によるキャッチをしなくてもコンパイルエラーにならない。

なぜスロー宣言をするとコンパイルエラーにならないのでしょうか。それは、スロー宣言とは、そのメソッドが「私はメソッド内でチェック例外が発生しても処理しませんが、**私の呼び出し元が処理します**」と表明する宣言だからです。

その一方、スロー宣言が含まれるメソッドを呼び出す側は「このメソッドを呼び出すと、呼び出し先で発生した例外が処理されずに**自分に伝播してくる可能性がある**」ことを覚悟しなければなりません。

スロー宣言が及ぼす影響

影響①　呼び出される側のメソッドb()は、メソッド内部での～Exceptionのキャッチが**義務ではなくなる**。

影響②　呼び出す側のメソッドa()は、「～Exceptionを伝播してくる可能性があるb()」の呼び出しをtry-catch文で囲む**義務が生まれる**。

※ throws ～ Exceptionというスロー宣言を伴うメソッドb()を、メソッドa()から呼び出す場合。

チェック例外に対する処理方法についてまとめておきましょう。すべてのメソッドは「チェック例外をどう処理するか」について次の２つの方針のどちらかを採用し、その方針ごとに課せられる義務を果たさなければなりません。

例外処理方針① チェック例外を自分で処理

【この方針の意味】

「私は自分で例外的状況を解決します。例外が発生してもお手上げはせず、呼び出し元に迷惑をかけません」

【この方針を採用することで課せられる義務】

発生する可能性がある、すべてのチェック例外を try-catch 文で処理すること。

例外処理方針② チェック例外を処理せず、呼び出し元に委ねる

【この方針の意味】

「私は自分で例外的状況を解決できません。例外が発生したら、呼び出し元に処理を任せます」

【この方針を採用することで課せられる義務】

メソッド定義にスロー宣言を加え、例外の種類を表明すること。

例外をもみ消さない

```
try {
 :
} catch ( Exception e ) {
}
```

このコードは「発生した例外を捕まえながら、自分では何の処理もせず、上にも報告しない」いわば「不祥事のもみ消し」のようなことをやっています。チェック例外は「何らかの対処がされるべきだから発生している」のですから、もみ消しが重大な不具合につながることは容易に想像できます。ですから空の catch ブロックは極力避けるようにしましょう。もし理由があって「あえて何もしない」場合には、catch ブロックの中にコメントで理由を残しておくとよいでしょう。

15.7 例外を発生させる

15.7.1 例外的状況を JVM に報告する

前節までで「例外の発生にどう備えるか」という解説は終わりだ。
最後に自分たちで「例外を発生させる方法」を紹介しよう。

図 15-8（p.570）で解説したように、「例外的状況が発生したかどうか」は JVM が監視します。そして JVM は例外的状況を検知すると処理を catch ブロックに移すのでしたね。実は、この監視をしている JVM に対して、私たち自身が「～ Exception という例外的状況になりました」と報告をすることができます。例外的状況が発生したことを報告するためには、次の構文を使います。

例外的状況の報告（例外を投げる）

```
throw 例外インスタンス;
```

※一般的には「throw new 例外クラス名 (" エラーメッセージ ") ;」となる。

JVM に報告するには、例外の詳細情報を詰め込んだ
例外インスタンスを用いるんですね。

そうだよ。throw キーワードを使って、
例外インスタンスを監視中の JVM に「投げつける」んだ。

「こんな例外的状況になったよ！ 今すぐ代替策の実施へ移行してください」と例外インスタンスを投げているイメージですね。

監視中の JVM に例外的状況を報告することを、「**例外を投げる**」または「**例外を送出する**」と表現することもあります。例外が投げられると JVM はそれを検知し、即座に catch ブロックの実行や伝播に処理を移します。

実際に例外を投げているプログラムの例がリスト 15-5 です。

リスト 15-5　例外インスタンスを自分で投げる

Person.java

```
1  public class Person {
2    int age;
3    public void setAge(int age) {
4      if (age < 0) {      // ここで引数をチェック
5        throw new IllegalArgumentException
             ("年齢は正の数を指定すべきです。指定値=" + age);
6      }
7      this.age = age;     // 問題ないなら、フィールドに値をセット
8    }
9  }
```

Main.java

```
1  public class Main {
2    public static void main(String[] args) {
3      Person p = new Person();
4      p.setAge(-128);
5    }
6  }
```

誤った値のセットを試みる→例外発生

実行結果

```
Exception in thread "main" java.lang.IllegalArgumentException:年齢は正の数
を指定すべきです。指定値=-128
  at Person.setAge(Person.java:5)
  at Main.main(Main.java:4)
```

Personクラスのsetage()メソッドでは、引数をフィールドage(年齢)に代入します。しかしageが負の値になることを防ぐため、代入の前に引数を検査しています。もし引数に問題がある場合は、IllegalArgumentExceptionインスタンスを投げることで「引数が異常で処理を継続できない」という例外的状況に陥ったことをJVMに報告します。

setAge()メソッドで発生した例外は、呼び出し元のmainメソッドに伝播します。しかし、ここでは例外をキャッチしていないので、最終的にJVMがプログラムを強制停止します。

スロー宣言で使うthrowsと例外的状況の報告に使うthrowは、似ているけど、まったく違うものだから気をつけよう。

15.7.2 オリジナル例外クラスの定義

これまで見てきたように、APIにはIOExceptionやIllegalArgumentExceptionなどの多くの例外クラスが備わっています。それら既存の例外クラスを使えば、多くのプログラムは問題なく作成できるでしょう。

しかし「独自の例外的状況」を表す「オリジナルの例外クラス」を使いたくなることもあります。たとえば音楽プレーヤソフトを開発しているのであれば、「対応していない形式のファイルを再生しようとした」などの例外的状況を表すクラス(たとえば、UnsupportedMusicFileException)が欲しくなるかもしれません。

そのような場合は、**既存の例外クラスを継承してオリジナルの例外クラスを作る**ことができます。

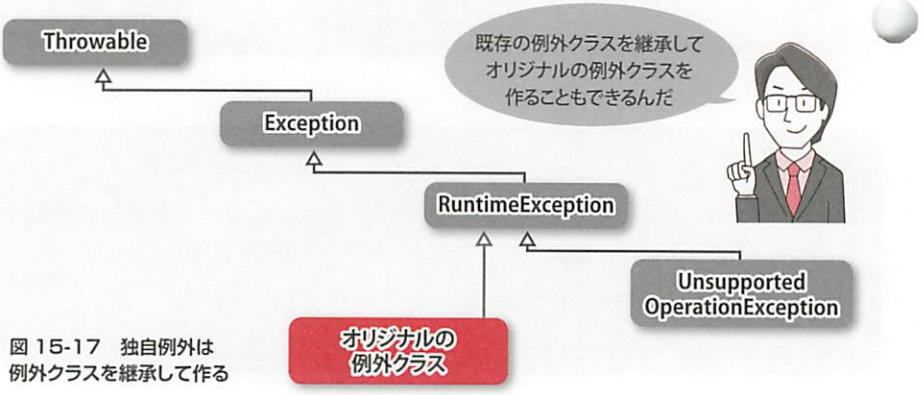


図15-17　独自例外は例外クラスを継承して作る

継承元になる例外クラスは、チェック例外を表す Exception や、非チェック例外を表す RuntimeException の他、IOException など実際に何かの状況を表している例外クラスでも構いません。しかし、Throwable や Error の子クラスとしてオリジナル例外を定義することはほとんどないでしょう。

リスト 15-6　オリジナル例外を定義する

UnsupportedMusicFileException.java

```
1  public class UnsupportedMusicFileException extends Exception {
2    // エラーメッセージを受け取るコンストラクタ
3    public UnsupportedMusicFileException(String msg) {
4      super(msg);
5    }
6  }
```

チェック例外にする

リスト 15-7　オリジナル例外を利用する

Main.java

```
1  public class Main {
2    public static void main(String[] args) {
3      try {
4        // 試験的に例外を発生させる
5        throw new UnsupportedMusicFileException
             ("未対応のファイルです");
6      } catch (Exception e) {
7        e.printStackTrace();
8      }
9    }
10 }
```

本格的で大規模なプログラムを開発するときは、プログラムで想定されるさまざまな例外的状況を思い浮かべ、オリジナルの例外クラスとして作成することで、きめ細かい実行時エラーへの対処が可能になります。

15.8 第15章のまとめ

この章では、次のようなことを学びました。

エラー

- 「文法エラー」、「実行時エラー」、「論理エラー」の3種がある。
- 例外処理を行うことで、実行時エラーに対処できる。

例外の種類

- APIには、さまざまな例外的状況を表す例外クラスが用意されている。
- 例外クラスは「Error系」、「Exception系」、「RuntimeException系」に大別できる。
- 例外クラスを継承してオリジナルの例外クラスを定義できる。

例外処理

- try-catch文を使用すると、tryブロック内で例外が発生したときにcatchブロックに処理が移る。
- 後片付けの処理は、必ず実行されるfinallyブロックに記述する。
- Exception系例外が起こる可能性がある場合は、try-catch文が必須である。
- スロー宣言を行うことで、例外の処理を呼び出し元に委ねることができる。
- throw文を使うことで、開発者自ら例外を発生させることができる。

15.9 練習問題

練習 15-1

次のようなプログラムを作成・実行し、実行時エラーを発生させてください。

① String 型変数 s を宣言し、null を代入する。
② s.length() の内容を表示しようとする

練習 15-2

練習 15-1 で作成したコードを修正し、try-catch 文を用いて例外処理してください。その際に例外処理では次の処理を行ってください。

①「NullPointerException 例外を catch しました」と表示する。
②「ーースタックトレース（ここから）ーー」と表示する。
③スタックトレースを表示する。
④「ーースタックトレース（ここまで）ーー」と表示する。

練習 15-3

Integer.parseInt() メソッドを実行し、文字列 " 三 " の変換結果を int 型変数 i に代入するコードを記述してください。その際に、parseInt() メソッドがどのような例外を発生させる可能性があるかを API リファレンスで調べ、正しく例外処理を記述してください。

練習 15-4

起動直後に IOException を送出して異常終了するようなプログラムを作成してください（ヒント：main() メソッドが「お手上げ」すれば、例外発生時にプログラムが異常終了します）。

15.10 練習問題の解答

練習 15-1 の解答

Main.java

```
1  public class Main {
2    public static void main(String[] args) {
3      String s = null;
4      System.out.println(s.length());
5    }
6  }
```

練習 15-2 の解答

Main.java

```
1  public class Main {
2    public static void main(String[] args) {
3      try {
4        String s = null;
5        System.out.println(s.length());
6      } catch (NullPointerException e) {
7        System.out.println
             ("NullPointerException例外をcatchしました");
8        System.out.println("――スタックトレース（ここから）――");
9        e.printStackTrace();
10       System.out.println("――スタックトレース（ここまで）――");
11     }
12   }
13 }
```

練習 15-3 の解答

Main.java

```
public class Main {
  public static void main(String[] args) {
    try {
      // APIリファレンスから送出例外を調べる
      int i = Integer.parseInt("三");
    } catch (NumberFormatException e) {
      System.out.println
          ("例外NumberFormatExceptionをcatchしました");
    }
  }
}
```

練習 15-4 の解答

Main.java

```
import java.io.IOException;
public class Main {
  public static void main(String[] args) throws IOException {
    System.out.println("プログラムが起動しました。");
    throw new IOException();
  }
}
```