Java 研修　WisdomsSoft

http://www.wisdomsoft.jp/1.html

石井裕太

2020 年 6 月 21 日

目次

第 1 章　Java と Java 言語　v

1.1　Java 仮想マシン　v

1.2　オブジェクト指向　vi

1.3　プログラミング言語 Java　vii

1.4　開発環境とコンパイラ　viii

第 2 章　初めての Java 言語　ix

2.1　初めての Java 言語　ix

2.2　コメント　x

2.3　字句変換と Unicode　x

2.4　トークン　xi

2.5　文　xi

2.6　変数と型　xii

2.7　リテラル　xiv

2.8　式と演算　xvi

第 3 章　配列　xix

3.1　配列型の変数　xix

3.2　文字列と文字配列　xx

3.3　配列初期化子　xx

3.4　配列のメンバ　xxi

3.5　配列の配列　xxi

3.6　配列宣言の混合表記　xxi

第 4 章　例外　xxiii

4.1　例外の発生　xxiii

4.2　例外のキャッチ　xxiii

4.3 明示的例外 xxv
4.4 チェック例外 xxv

参考文献 xxvii

参考文献 xxvii

第 1 章

Java と Java 言語

1.1　Java 仮想マシン

1.1.1　なぜ、今 Java なのか

Java...Sun Microsystems 社が開発

- Java... システムとして
- Java 言語... 言語として

Java 言語で開発されたソフトウェアは、Java がサポートされている全ての環境で実行可能

- C,C++　など　コンパイラ言語
- BASIC,Perl... インタプリタ言語

1.1.2　どこでも動く Java の秘密

コンパイルの流れ：ソースファイル→コンパイラ→オブジェクトファイル→リンカ→実行可能ファイル

インタプリタの流れ：ソースファイル→インタプリタ→実行

Java：ソースファイル→コンパイラ→ Java バイトコード→仮想マシン→実行

1.2　オブジェクト指向

1.2.1　オブジェクトの役割

高水準言語は、手続き型と非手続き型に分類される
- 手続き型
命令手順を記述して問題を解決するアプローチ C,Pascal,FORTRAN など
- 非手続き型
手順ではなく必要な情報を与えることで問題を解決するアプローチ LISP,Prolog など

Java は、非手続き型に属するオブジェクト指向型に分類される。これは、プログラムの流れではなく、オブジェクトの役割に注目している

継承と抽象化

これまでのプログラマは、プログラムを組むときに流れを意識

- 順番に命令が実行される逐次
- 条件によって命令が選択される分岐
- 条件が満たされるまで繰り返す反復

CPU はこれらの計算を延々と繰り返している。一方、オブジェクト指向はこのような流れよりもプログラムを部品化して独立させることに注目する。

正しく部品化されたプログラムは他のプログラムコードの影響を受けない。このような状態をプログラムの独立性として評価される。

オブジェクト指向プログラミングでは、プログラムの部品をオブジェクト（物）として考える。実行手順ではなく、役割を与えられたオブジェクトを部品として開発し、最終的にオブジェクトを結びつけ、それぞれのオブジェクトが関係し合うことによって目的が達成される。

- 継承　　既存プログラムが提供する提供する機能を引き継ぎ、拡張することができる機能。抽象→具体的な実装という流れでプログラムを設計できる。
- 多様性　　目的を達成する達成するための方法にバラエティを持たせる手段。
- カプセル化　プログラムのオペレーションを帰省すること。外部に公開する必要のない内部事情のコードを隠蔽し、プログラムを利用する人間に必要なコードだけを公開できる。

1.3 プログラミング言語 Java

1.3.1 Java の誕生と歴史

家電製品市場のアプリケーション開発のため、プラットフォームに依存しない開発システムが求められた。まず、Oak が開発される。しかし、成功の目を見ることなく、製品名を Java に変更。

Java の開発環境 JDK(Java Development Kit) は Sun Microsystems の Web サイトから無償でダウンロードでき、誰もが自由に使える。

JDK1.0

- アプレットなどを中心に、クライアント側のネットワーク関連ソフトウェアや、小規模な GUI プログラミング、コマンドラインで利用する変換プログラムや圧縮、暗号などの研究プログラムに利用可能。
- 当時は完成度が不十分で、標準ライブラリが提供する昨日も貧弱

その後、サーヴレットや JDBC と呼ばれる API の普及により、サーバー側のネットワーク関連ソフトウェアやデータベース関連ソフトウェアが注目を集める。セキュリティシステムも堅牢で、国際プログラムを得意とし、ビジネスの世界で支持されている。

JDK1.2

- Swing と呼ばれる GUI(Graphical User Interface) コンポーネントが登場し、クライアント側のソフトウェア開発で注目される。
- Java の動作は遅かったが、マシンの高速化と Java 技術の工場により十分採用できる範囲になった。
- 携帯コンピュータや携帯電話も発達し、これらに実装するアプリケーションとしても Java が流行った。

JDK1.3,JDK1.4

- マルチメディアの分野でも Java が実用可能な範囲になってきた。
- XML, サウンド、イメージなどの高度な入出力をサポートする API が新たに実装。
- 今後はゲーム開発、音楽編集、動画関連ソフトウェアにも期待

現在も Java は安定しているとは言えず、改良と拡張を繰り返して Java は進化し続けている。

1.3.2　Java を学ぶ

オブジェクト指向型言語である Java はプログラムの流れでなく、部品としての役割を重視して目的のソフトウェアを構築していく。しかしオブジェクト指向プログラミングには高度な設計力が必要となる。

Java 言語のようなオブジェクト指向型の開発は、C 言語のような手続き型とは異なり、巨大なソリューションの小さな部品を開発するという性質をもつ。このためオブジェクト指向は大規模な開発に向いている。大規模な開発には、設計図が必要

1.4　開発環境とコンパイラ

1.4.1　Java SDK

Java プラットフォーム専用のアプリケーションを開発するためには、Java バイトコードを生成しなければならない。Java にとっては Java バイトコードこそが機械語であり実行データ。

Java バイトコードを生成するためには、C/C++ 言語などによる開発と同様、高水準言語を用いて開発する方法が用いられる。Java バイトコードを生成するための専用の高水準言語が Java 言語。Java 言語のコンパイラは JDK に付属している。

第 2 章

初めての Java 言語

2.1　初めての Java 言語

2.1.1　最初のプログラム

Java 言語は C 言語のような手続き型言語とは異なり、ルーチンや関数といったプログラムコードの集積ではなく、何らかの役割を持った小さな部品単位で開発し、それらを組み立ててシステムを構築する。

このときの部品をオブジェクト指向ではクラスと呼んでいる。Java のあらゆるコードはクラスの中に属し、様々なクラスの関係によって 1 つのシステムが実現する。プログラムの流れは、クラスの動作として記述し、Java 言語ではクラスの動作のことをメソッドと呼ぶ。

オブジェクト指向プログラミングは手順ではなく役割単位で設計を行うため、実世界のような複雑な構造をもつオブジェクトを表現できる。

2.1.2　文字を表示する

```
System.out.print("表示する文字列");
System.out.println("表示する文字列");
```

Java の標準クラスライブラリが様々な機能を提供することで、システムを隠蔽する。文字表示は System クラスの機能を使って文字を表示する

2.2　コメント

2.2.1　注釈を残す

プログラムを改変する必要があるとき、オープンソースなど共同作業の場合など、開発者以外の人がソースコードを読む際の理解の手助けにする

```
/* 注 釈 テ キ ス ト */
// 注 釈 テ キ ス ト
```

2.3　字句変換と Unicode

2.3.1　Unicode エスケープ

コンピュータの世界では文字も二進数として表現されている。国際的にどのコンピュータでも文字データは統一して扱うことができるように規格が定められた。これがいわゆる文字コードである。

1 バイトコード

- ASCII コード (American Standard Code for Infomation Interchange) ANSI(American National Standard Institute) 米国規格協会が制定

しかし、日本語、中国語など文字の種類が多い言語には対応できない。そこで各国独自の文字コードが誕生。

- JIS
- Shift JIS
- 日本語 EUC

など

文字コードの乱立が問題となり、Microsoft や IBM などの米国企業が中心となって提唱したのが Unicode。ISO(International Organization for Standardizatio 国際標準化機構) で国際規格 ISO/IEC 10646 の一部として標準化されている。ASCII と異なり、全ての文字を 2 バイトで表現。

Java はどこでも動くのは、マシン環境だけでなく、世界の様々な国の言語環境も Unicode で記述することによって解決している。Java 言語は ASCII だけでプログラムできるように設計されている。

Unicode エスケープ

\u****
”\u732B\u304C\u597D\u304D” → 猫 が 好 き

2.4 トークン

2.4.1 Java 言語の基本構造

コンパイラは、文字の並びを頭から順に解析して機械語 (Java の場合はバイトコード) へと変換しているにすぎない。

1. まず Unicode エスケープを検索して Unicode 文字に変換する。
2. 次に行末記号の分析が開始される。
3. Unicode エスケープと行末文字を処理した最終的な結果を入力要素と呼ぶ。
4. ソースプログラムは入力要素へと還元され、さらに入力要素はトークンへと分解される。
5. トークン：空白とコメントを除いた入力要素、ソースプログラムの最小単位

2.5 文

2.5.1 文を完成させる

トークンとは、英文に置ける英単語だが、英単語だけでは意味は伝わらない。複数のトークンを正しく組み合わせて何をやりたいのかを伝えなければならない。

- ブロック
- 空文
- 式文
- ローカル変数宣言文
- if 文
- switch 文
- while 文
- do 文
- for 文
- break 文
- continue 文

- return 文
- throw 文
- synchronized 文
- try 文
- 到達不能文

空文

;

何もせずに常に正常終了させることが保証されている。

2.6　変数と型

2.6.1　情報を保存する

プログラミングでは、与えられた情報を決められた手順で分析、加工して出力するということが基本原理。その過程で必要な情報を保存することは極めて重要。保存といっても、いちいち記録ディスクに保存しているとプログラムは遅くて使い物にならない。

通常、プログラムが稼働中に必要な情報を保存するときは、主記憶装置に対して入出力が行われる。Java はどこでも動くという思想から、プログラマが実装システムの記録装置を意識する必要が内容に設計されているので、主記憶装置の扱いを意識する必要はない。

プログラム言語ではデータの保存には**変数**を用いる。変数は主記憶装置の記録場所の代名詞。プログラマはこの変数に情報を保存したり、保存した情報を引き出したりすることができる。

情報の保存のもう一つの大きな問題として、保存されている情報が計算用の数値なのか、表示する文字なのかの識別方法はどうなるか。

機械語レベルでは開発者の自己責任。どちらとして扱ってもいいが、人間のミスによりバグが発生しうる。

Java は保存された情報の種類の管理を行ってくれる。この管理を行うための情報を**型**と呼ぶ。

Java 言語の設計者たちは仕様書の中で「Java は強く型づけされた言語である」と表現している。

2.6.2 ローカル変数宣言文

変数の利用には、メソッド内で**ローカル変数宣言文**を使う。ローカル変数はメソッド内でしか宣言することができず、メソッドが終了すると破棄される。

2.6.3 識別子

ローカル変数宣言文では、肩を表すキーワードと変数の名前を設定しなければならない。Java 言語ではこのような、何らかの実態を認識するための名前を識別子と呼ぶ。識別子は自由に設定できるが、Java 言語が定める命名規則に従わなければならない。Java 言語では、識別子は Java 文字と Java 数字の並びであると定義している。

Java 文字

ASCII 文字のアルファベット A-Z(\u0041-\u005A), a-z(\u0061-ŭ007A)、ASCII のアンダーライン_ (\u005F) とドル記号$(\u0024) のことを表す。

Java 数字

ASCII の 0-9(\u0030-\u0039) までの数字を含む Unicode 文字。

ただし、識別子の先頭 1 文字は Java 文字でなくてはならない。

2.6.4 確実な代入

Java 言語ではローカル変数宣言文で宣言された変数は、その値がアクセスされるまでに必ず値が保持されていなければならない。これを**確実な代入**と呼ぶ。コンパイラは変数のアクセスを認識すると、厳密な手順の解析を行って確実な代入状態であるか調べる。値が保持されていることが保証できない場合、コンパイル・エラーを発生させる。

2.6.5 変数宣言子

変数宣言

```
型　変数宣言子, 変数宣言子...;
```

2.6.6 名前つけ規約

Java 言語ではソースの可読性を高めるために変数の識別子に対して名前つけ規約を定めている。

型	1 文字識別子
byte	b
byte 以外の整数型	i,j,k
char	c
float	f
double	d
long	l
String	s

変数は、意味のある名前を省略した、単語とならない短い小文字の並びであるべきとされる。

- temp:ごく一時的な利用に限られた変数
- val:値を保持する

原則として 1 文字の変数は推奨されない。しかし、一時てきな利用や汎用的な利用をされる変数であれば、変数型を連想させる文字を採用するような規約が定められている。

2.7　リテラル

2.7.1　整数リテラル

リテラル：そのままの値のトークン,10,”Kitty”などリテラルの値はコンパイル時に確定し、実行中に変化することはない。

とはいうものの、実は変数と同時にリテラルにも型が存在する。意味は変数リテラルかだけの違いであり、型の考え方は全く同じ。リテラルを変数に保存する場合、型が一致しなければならない。

小数点を含まない数字は**整数リテラル**と呼ばれ、整数型の変数に代入することができる。iValue = 10 という文では、整数リテラルは int 型として認識されている。

2^{31} 以上の数値を表現したい時は int 型では不十分。long 型の整数リテラルを記述するためには、数字の末尾に l または L を付加する。(小文字の l は 1 と誤認しやすいので L を使うべきと考えられている。) long 型では 2^{63} までの値を表現することが可能。

2.7.2 浮動小数点数リテラル

float や double 型の定数は**浮動小数点数リテラル**と呼ばれ、整数部、小数点、小数部、指数部、型接尾辞で構成される。

整数部 . 小数部 E 指数部 型接尾辞

浮動小数点数リテラルは、整数リテラルとは異なり、10 進数でのみ表記できる。

浮動小数点数リテラルもまた、型接尾辞を指定することで float 型なのか double 型なのかを区別することができる。float 型は f または F、double 型は d または D を末尾につける。接尾辞がないときは double 型であると解釈される。

2.7.3 文字と文字列リテラル

Java では文字が Unicode として扱われる。C 言語の char 型変数は ASCII 文字を表すための 1 バイトだったが、Java 言語の char 型は 2 バイトであることに注意。

char 型で表現できる 1 つの Unicode 文字を**文字リテラル**と呼ぶ。文字と文字列は異なることに注意。

- 文字... 一つの Unicode
- 文字列... 複数の文字の並び

文字リテラルは' で囲む。

char 型に整数リテラルを代入できる理由は、char 型が整数型に分類されることから理由づけされる。print() および println() メソッドは char 型のデータを受け取るとそれを文字として表示する性質がある。逆も可能で、

```
int iValue = '猫';
```

iValue には Unicode 文字' 猫' を示すコード 0x732B が格納される。

2.7.4 エスケープ・シーケンス

文字や文字列リテラルは、引用符の間で改行することはできない。改行を表現したい場合、文字列リテラルの中に二重引用符や水平タブを使いたい場合も同様。

このように文字としては表現が難しい「'」「"」のような記号、表示できない文字などを表現したい場合は**エスケープ・シーケンス**を使う。エスケープ・シーケンスはバックスラッシュ \(\u005C) とそれに続く ASCII 文字で構成される。エスケープ・シーケンスはこれで 1 つの文字と解釈されるため、文字リテラルとして指定することも可能。Unicode

表 2.1　エスケープ・シーケンス

エスケープ・シーケンス	Unicode	意味
\b	0008	バックスペース
\t	0009	水平タブ
\n	000A	改行
\f	000C	フォーム・フィード
\r	000D	復帰
\”	0022	二重引用符
\’	0027	引用符
\\	005C	バックスラッシュ
\XXX	0000 ~ 00FF	00~FF までの 8 進数で表現されたコード

エスケープとは異なり、コンパイルの軸変換サイクルで処理されるわけではない。コンパイラがエスケープ・シーケンスを処理する頃にはすでに \u は変換済みであることに注意。

2.7.5　審議リテラル

boolean 型には true と false の二つの値が入る。この true と false を**審議リテラル**と呼ぶ。

2.7.6　null リテラル

変数が保持する情報が存在しないことを示す null 型が存在。null 型は唯一、null という値を保持し、ASCII 文字を用いた **null リテラル**は常に null 型となる。null は数値型の変数に代入することができず、代入できるのは何らかのオブジェクトの位置を表す参照型と呼ばれるタイプの変数のみ。

2.8　式と演算

2.8.1　数の計算

コンピュータ＝電子計算機、計算機の本領は計算。

計算を行うには式文を記述する。計算式は左側から右側に向かって評価され、その結果を変数に代入する。評価とは式を実行することを表し、評価によって導き出された値を結果と呼ぶ。

```
iValue = 10;
```

このとき、記号=を**演算子**と呼び、iValue と 10 のような演算対象を**オペランド**と呼ぶ。

第 3 章

配列

3.1　配列型の変数

3.1.1　変数のコレクション

大量の変数をまとめて管理するために**配列**を使う。

配列を用いれば、同じ型の同じ変数名で、複数の記憶領域を確保することができる。配列変数を宣言するには、フィールド宣言やローカル変数宣言文で、変数名の後に [] を指定する。

配列の宣言

```
int ary[];
```

配列変数を宣言しただけでは記憶領域は確保されない。配列変数はプリミティブ型ではなく参照型。変数 ary は数値型のリストを保存するための記憶領域を参照しているだけなので、宣言するだけでは null を指しているだけ。

配列の実体を生成するには、**配列生成子**を用いる。

配列生成子

```
new 型名[式];
```

配列生成子は、指定した型名の配列をブラケット内に指定した数値だけ作成する。

```
int ary[] = int[5];
```

この文は数値型の配列を 5 つ作成している。メモリ概念としては、少なくとも数値型の変数を 5 つ宣言した場合と同じ記憶領域が確保される。配列内に存在する 5 つの変数を**構成要素**と呼ぶ。そして、配列の実体の型を構成要素型と呼ぶ。

3.1.2　main() メソッドのパラメータ

main() メソッドの宣言では、必ず String 型のパラメータ args を宣言していた。この変数の直後にブラケットを指定していたのは、このパラメータが文字列型の配列だったから。

main() メソッドを呼び出すのは Java 仮想マシンを実装するシステムであり、プログラムではない。Java 仮想マシンは、main() メソッドにコマンドライン引数を渡す。Java 仮想マシンを（Java コマンド）を起動するときに渡された引数は、main() メソッドの args パラメータに渡される。コマンドを通してプログラムに引数を渡せるため、コンソールプログラムを開発する場合、この情報は重要になる。

T7_7_2 のプログラムには欠点が存在し、引数を渡さずに起動した場合は null を参照してしまうため例外が発生してしまう。複数の引数を渡した場合も、支所の引数しか表示することができない。与えられた引数に柔軟に対応するには、配列の構成要素数を調べなければならない。

3.2　文字列と文字配列

3.2.1　String 型

C 言語では文字列とは文字列 char 型の配列だと考えることができる。しかし Java の場合は文字列と文字配列は異なるものと定められている。

3.3　配列初期化子

3.3.1　配列の初期化

配列宣言時に、**配列初期化子**を用いることで宣言と同時に初期化できる。

```
int iArray[] = {1,10,100,1000};
```

コード 7_3_1

配列初期化子で指定する各要素の変数初期化子はリテラルである必要はない。変数やクラス、インスタンス生成式を指定することも可能。C 言語では配列の初期化にリテラルしか使えなかったが、Java ではこの制約はない。

3.3.2 配列生成式における初期化

配列生成式はインスタンスを生成すると同時に各要素を初期化する方法を提供している。

コード 7_3_2

3.4 配列のメンバ

3.4.1 配列の長さを得る

Java では配列も参照型として扱われており、暗黙的に Object クラスを継承した型として制御されている。そのため、Object クラスで定義されているメンバを継承している他、配列の構成要素の数を格納する length フィールドを公開している。

コード 7_4_1

3.5 配列の配列

3.5.1 配列型の構成要素

多次元配列... 構成要素が配列の配列を考える。

コード 7_5_1

3.5.2 配列の配列を初期化する

多重配列を配列初期化子で初期化するには、配列初期化子も配列の深さに合わせて入れ子にする必要がある。配列の配列を初期化する作業は面倒なので、宣言時に各配列の値が決定しているのであれば、配列初期化子を用いてインスタンス化することができる。

3.6 配列宣言の混合表記

3.6.1 型にブラケットを指定する

Java では配列は 1 つの型であると考えることができる。つまり、配列型 int[] は一つの型だと考えることができる。つまり、

```
int iArray[];
int[] iArray;
```

は同じ、整数型の配列を参照する iArray 変数が宣言される。

第 4 章

例外

4.1　例外の発生

4.1.1　有事の備え

プログラムのバグでもっとも厄介なのは、実行時のエラーである。

実行時のエラーやバグというのは、例えば配列サイズ以上の存在しない構成要素を参照する、インスタンス生成時にメモリが不足するなど、コンパイル時点では予測することのできないプログラム、またはコンピュータや仮装マシンのエラー。このような実行時のエラーは復帰不能な致命的なもの。

コード 8_1_1

このプログラムでは 0 除算により仮想マシンは異常な処理を検出する。その結果、java.lang.ArithmeticException という例外が発生したことが実行結果から知ることができる。java.lang.ArithmeticException は立派なクラスであり、例外が発生するとこのようにそのクラスに応じて適切な例外クラスがインスタンス化され、それが例外としてスローされる。例外はクラスとして構造的な情報を提供することができるので、この例外クラスを通すことによって、開発者は何が原因で例外が発生したのかを知ることができる。

4.2　例外のキャッチ

4.2.1　try 文

開発者は、例外のスローを受け取り例外処理を行うコードに制御を導くことができる。これを例外を**キャッチ**すると表現する。例外をキャッチするには **try 文**を使う。

```
try {
    //例外が発生する可能性のあるプログラム
```

h

表 4.1　一般的な例外

例外クラス	発生理由
ArithmeticException	算術計算で例外的処理が発生した (0 の除算など)
ArrayIndexOutOfBoundsException	不正なインデックスを使って配列がアクセスされた
ArrayStoreException	不正な型のオブジェクトをオブジェクトの配列に格納しよう
ClassCastException	あるオブジェクトを継承関係にないクラスにキャストしよう
IndexOutOfBoundsException	ある種のインデックス (配列、文字列、ベクタなど) が範囲外
NegativeArraySizeException	負のサイズを持った配列をアプリケーションが作成しようと
NullPointerException	Null を参照した

```
}
catch(例外型　識別子) {
    //例外に対する処理
}
```

例外型とは、例外が発生した時に生成された、例外の発生理由などの情報を格納したオブジェクトのこと。Java では、どこかで発生した実行時の不正な処理の原因を伝達するため、クラスのインスタンスを投げる。catch ブロックでは、このオブジェクトをパラメータとして受け取り、この情報を基に適切な処理を施す。

発生した例外が catch で指定されている例外型と一致しなければ、catch ブロックは実行されない。try 文に対し catch は複数指定することが可能なので、目的の例外処理ごとにブロックを記述することができる。catch で適切な処理を施しブロックから抜け出すと、プログラムの制御は try 文から抜け出す。

コード 8_2_1

例外の種類

コード 8_2_2

4.2.2　fainally ブロック

try 文を実行した結果、例外が発生するしないに関わらず、何らかの処理を最終的に実行したい場合は fainally というブロックを定義する。finally は try 文の一部で catch に続いて記述するブロックである。try 文は正常に try ブロックが終了しても、例外がスローされても必ず finally ブロックを実行する。

コード 8_2_4

try 文が実行されると、最終的に必ず finally ブロックが実行される。finally ブロックが指定されている場合、catch 文を省略しても問題はない。

コード 8_2_5

コード 8_2_5 では、catch ブロックを省略している。

4.3 明示的例外

4.3.1 例外をスローする

これまでの、Java 仮想マシンが不正な処理を検出した時にスローされる実行時例外だった。しかし、実行時に検出される不正なプログラムというのは必ずしも構文的な問題とは限らない。クラスの実装に反した操作が実行された場合も例外によって開発者の過ちを伝える必要があると考えられる。例えば、正数しか与えてはならないメソッドの引数に負数を指定した場合などは例外を発生させるべき。

プログラムが意図的に例外をスローするには **throw 文**を用いる。throw 文が実行された時点で例外がスローされるため、プログラムの制御はその時点で catch の検索に移行する。

4.4 チェック例外

4.4.1 スロー宣言

特定のブロック内で発生した例外をその場で取り扱うのならば、それは内部仕様の問題。クラスの利用者がその事実を知る必要はない。しかし、メソッドが何らかの例外をスローし、それを取り扱わなければ、メソッドを呼び出すクラス利用者がスローされた例外を取り扱う必要がある。ところが、必ずしもメソッドがスローした例外を、クラスの利用者がキャッチする必要はない。そのため、Java コンパイラは明示的にスローされた例外がキャッチされない場合はエラーを出す。

コード 8_4_1 このコードは構文上のミスはないように見えるが、スローされる可能性がある例外が確実にキャッチされる保証がないという理由からエラーが発生する。

コード 8_4_2 ThrowException() メソッドは Exception 型の例題を呼ぶだし元にスローすることができるが、ThrowException を呼び出している main() メソッドが Exception 例外を取り扱っていないため、さらにどの呼び出し元に Exception 例外をスローする可能性があるため、コンパイルエラーとなる。

参考文献

[1] 赤坂玲音, 読本 Java 2011/11/14 毎日コミュニケーションズ
http://www.wisdomsoft.jp/1.html