訛りのない Java 言語を話す

Java に転向したプログラマーのためのナチュラルな Java 表現

Elliotte Rusty Harold
Software Engineer
Cafe au Lait

2010年 1月 12日

この記事では Elliotte Rusty Harold が Java™ 言語とそのコミュニティーでの固有の慣用表現、方言、訛りについて詳しく説明します。この記事のガイダンスに従えば、C/C++ やその他の言語から Java に転向したプログラマーでも、Java 言語のネイティブ・スピーカーの仲間に加われます。

新しいプログラミング言語を学ぶのは、新しい話し言葉を学ぶほど難しくありません。しかしどちらを学ぶにしても、訛りなしでその言語を話せるようになるためにはさらなる努力が必要です。スウェーデン語を話せる人がデンマーク語を学ぶのと同じように、C や C++ の知識があれば、Java 言語を学ぶのはそれほど難しくはありません。Java と C や C++ とは異なる言語ですが、互いに通じる言語だからです。けれども用心していなければ、C 言語や C++ 言語の訛りによって、Java 言語を使うたびに生粋の Java プログラマーでないことが明らかになってしまいます。

C++ プログラマーは Java コードに特定の癖をつけがちで、その癖によって紛れもなく、自分がネイティブ・スピーカーではなく、Java に転向したプログラマーであることが明らかになります。作成したコードはそれでも機能しますが、生粋の Java プログラマーの耳には奇異に聞こえ、その結果、彼らは Java に転向したプログラマーを見下すことになります。C 言語や C++ 言語 (または Basic、Fortran、Scheme など、どの言語でも) から Java 言語に転向する場合は、この言語をナチュラルに使用できるように、元の言語の慣用表現を完全になくし、独特の訛りを正さなければなりません。

この記事では、仮に問題があるとしても、意味的にはそれほど重要でないがために見過ごされがちな Java プログラミングの詳細を取り上げます。これらの問題は単にスタイルと慣例に関連することでしかありません。なかには問題と呼ぶべき正当な理由があるものもあれば、それさえないものもあります。しかしいずれの問題も、現在作成されている Java コードで実際に起きていることです。

何の言語で作成されたコードでしょう？

まずはリスト 1 に記載する、華氏の温度を摂氏の温度に変換するコードを見てください。

リスト 1. これは C 言語のコードでしょうか？

```
float F, C;
float min_tmp, max_tmp, x;

min_tmp = 0;
max_tmp = 300;
x  = 20;

F = min_tmp;
while (F <= max_tmp) {
   C = 5 * (F-32) / 9;
   printf("%f\t%f\n", F, C);
   F = F + x;
}
```

リスト 1 で使用されている言語がわかりますか？明らかに C 言語のようですが、ちょっと待ってください。今度はリスト 2 に、上記のコードが含まれるプログラム全体を記載します。

リスト 2. Java プログラム

```
class Test  {

    public static void main(String argv[]) {
        float F, C;
        float min_tmp, max_tmp, x;

        min_tmp = 0;
        max_tmp = 300;
        x  = 20;

        F = min_tmp;
        while (F <= max_tmp) {
          C = 5 * (F-32) / 9;
          printf("%f\t%f\n", F, C);
          F = F + x;
        }
    }

    private static void printf(String format, Object... args) {
        System.out.printf(format, args);
    }

}
```

信じようと信じまいと、リスト 1 とリスト 2 はどちらも Java 言語で記述されています。この 2 つは単に、C の慣用表現で作成された Java コードというだけのことに過ぎません (公正を期して言うなら、リスト 1 は純粋な C コードであると見なすこともできます)。しかし Java コードにしては、かなり異様なコードです。このなかで使用されている以下の慣用表現が、このプログラムの作成者は C 言語で考えていて、その考えを Java 言語に置き換えているに過ぎないことを示しています。

- 変数が `double` ではなく、`float` になっています。
- すべての変数がメソッドの先頭で宣言されています。
- 宣言の次に初期化が続いています。
- `for` ループの代わりに `while` ループが使われています。
- `println` ではなく `printf` が使われています。
- `main()` メソッドの引数の名前が `argv` となっています。

- 配列の括弧が、型の後ではなく、引数名の後にあります。

これらの慣用表現は、コードがコンパイルされないわけでも、誤った答えを出すわけでもなく、どれも誤ってはいません。上記に指摘した点の 1 つひとつは大きな意味を持ちませんが、すべてを合わせると非常に奇妙なコードとなり、Java プログラマーにとって読みづらいコードとなってしまいます。それは、アメリカ人にとってイングランド北部の方言である Geordie を理解するのが大変なのと同じことです。このような C の慣用表現を使用することが少なくなれば、その分、Java プログラマーが理解しやすいコードになります。このことを念頭に、これから Java コードを作成する上で C プログラマーが自分の正体を明かしてしまう種となる最も一般的な癖を分析し、Java プログラマーの目に快適なコードにするための方法を紹介します。

命名規則

C および C++ プログラマーの出身か、または C# プログラマーの出身かによって、学習したクラスの命名規則は異なります。例えば C# の場合、クラス名は小文字で始まり、メソッドとフィールドの名前は大文字で始まりますが、Java のスタイルではまったく逆です。どちらか一方の命名規則をもっともな理由で正当化することはできません。けれども確かに言えることは、命名規則を混同すると恐ろしく違和感のあるコードになるということです。さらに、バグの原因にもなります。すべてが大文字の名前は定数であるとわかっていれば、これに該当する名前については、他の名前とは別に処理することができます。私はこれまで、宣言された型が命名規則と一致してない箇所を探すという単純な方法によって、プログラムに潜むたくさんのバグを見つけてきました。

`argv` ではなく `args` です

この指摘は最も平凡ながらも、スタイルに競合が発生する些細な点の 1 つです。Java の隠語では、`main()` メソッドの引数の名前は `args` であり、`argv` ではありません。

```
public static void main(String[] args)
```

しかしこのように `argv` の代わりに `args` を使ったからといって、ごくわずかな改善がなされるだけであり、arguments の省略形であることが少しばかりわかりやすくなるだけのことです。もちろん慣用語法にかなった Java コードでは省略形の使用は一般に認められません (「省略しないこと」を参照)。`main()` メソッドの引数の名前として `args` を使用する唯一の理由は、最初の C 言語の解説書を書いた Kernighan と Ritchie の 2人が使っていたからという理由で C プログラマーが `argv` を使用するのと同じで、Gosling と Arnold が `args` を使っているからです。これ以外の理由は何もありませんが、生粋の Java プログラマーは誰もが `args` を使用しているので、訛りのない Java を使いたいのであれば、それに倣ってください。

Java プログラミングでの基本的な命名規則はごく単純なので、覚えておく価値があります。

- クラス名とインターフェース名は大文字で始まります (例: `Frame`)。
- メソッド、フィールド、ローカル変数の名前は、小文字で始まります (例: `read()`)
- クラス名、メソッド名、フィールド名ではキャメル・ケース (各単語の先頭文字を大文字にして連結すること) を使用します (例: `InputStream`、`readFully()`)。
- 定数 (final static フィールド、場合によっては final ローカル変数) は、すべて大文字にして単語間をアンダーバーで結びます (例: `MAX_CONNECTIONS`)。

省略しないこと

`sprintf` や `nmtkns` のような名前は、スーパーコンピューターのメモリーが 32 KB だった時代の遺物です。コンパイラーはメモリーを節約するため、ID を 8 文字以下に制限していました。しかしこの 30 年以上、名前の文字数は問題ではなくなっています。現在、変数やメソッドの名前を完全にスペルアウトしない理由はありません。母音がなくて理解不可能な変数名は他の何よりも、C から転向したハッカーが作ったプログラムであることを示す明らかな証拠となります。

リスト 3. Abbrvtd nms r hrd 2 rd

```
for (int i = 0; i < nr; i++) {
    for (int j = 0; j < nc; j++) {
        t[i][j] = s[i][j];
    }
}
```

リスト 4 を見るとわかるように、キャメル・ケース形式で省略しない名前のほうが、遥かに読みやすくなります。

リスト 4. 省略されていない名前は簡単に読めます

```
for (int row = 0; row < numRows; row++) {
    for (int column = 0; column < numColumns; column++) {
        target[row][column] = source[row][column];
    }
}
```

コードは作成される回数よりも、読まれる回数のほうが多いため、Java 言語は読みやすさのために最適化されています。C プログラマーは難読化されたコードに強く心を引き付けられるものですが、Java プログラマーはそうではありません。Java 言語は簡潔さよりも読みやすさに重点を置いています。

ただし以下のように、罪悪感を持たずに使用できる一般的な略語がいくつかあります。

- maximum の省略形 `max`
- minimum の省略形 `min`
- `InputStream` の省略形 `in`
- `OutputStream` の省略形 `out`
- `catch` 節に含まれる exception の省略形 `e` または `ex` (他の場所では使用不可)
- number の省略形 `num` (`numTokens` や `numHits` のように頭辞辞として使用する場合のみ)
- 極めて局所的に使用する一時変数の省略形 `tmp` (2 つの値を交換する場合など)

上記の他にもいくつかあると思いますが、これらの例外を除き、名前で使用するすべての単語は完全にスペルアウトしてください。

変数の宣言、初期化、および (再) 利用

C の初期の頃のバージョンでは、すべての変数をメソッドの先頭で宣言するように規定していました。これは、RAM が極めて乏しい環境でも実行できるように、コンパイラーでのある程度の最適化を可能にするためでした。このことから、C のメソッドは以下のように、変数を宣言する複数の行で始まる傾向があります。

```
int i, j, k;
double x, y, z;
float cf[], gh[], jk[];
```

その一方、このスタイルにはいくつものマイナス面があります。まず、変数の宣言と変数の使用が切り離されるため、コードを追うのが少し大変になります。さらに、1 つのローカル変数が、おそらく無意識に異なる用途で再利用される可能性がかなり高くなります。この場合、変数にそのまま値が残され、その値がコードの期待していた値でなければ、予期せぬバグが発生してしまいます。このような変数の再利用と、C での短く不可解な変数名という傾向を併せて考えれば、大惨事の原因となることは明白です。

Java 言語では (そして C の最近のバージョンでも)、変数は最初にそれを使用する箇所、あるいはその近くで宣言することができます。この宣言方法を実践して Java コードを作成すれば、コードはより安全になり、バグが発生しにくくなり、そして読みやすくなります。

関連する話として、Java コードは一般に、それぞれの変数をその変数が宣言された時点で初期化します。例えば、C プログラマーは以下のようなコードを作成することがあります。

```
int i;
i = 7;
```

構文的に正しいとしても、Java プログラマーがこのようなコードを作成することはほとんどありません。Java プログラマーであれば以下のように作成します。

```
int i = 7;
```

上記のようにすれば、初期化されていない変数が不用意に使用されて、その結果バグが発生するという事態を避けることができます。よくある唯一の例外は、1 つの変数のスコープを `try` にも、`catch` または `finally` ブロックにも設定しなければならない場合です。`finally` ブロックでクローズする必要のある入力ストリームと出力ストリームを扱うコードでは大抵、この例外が発生します (リスト 5 を参照)。

リスト 5. 例外処理により、変数のスコープを適切に設定しにくい場合があります

```
InputStream in = null;
try {
  in = new FileInputStream("data.txt");
  // read from InputStream
}
finally {
  if (in != null) {
    in.close();
  }
}
```

しかし上記の場合を抜かし、この例外が発生することはほとんどありません。

このスタイルの波及効果をもう 1 つ挙げると、それは、Java プログラマーは通常 1 行につき 1 つの変数しか宣言しないことです。例えば、Java プログラマーは 3 つの変数を以下のように初期化します。

```
int i = 3;
int j = 8;
int k = 9;
```

Java プログラマーが以下のようなコードを作成することはほとんどありません。

```
int i=3, j=8, k=9;
```

この文は構文的には正しいものの、常に Java を使用しているプログラマーは、以下で説明する 1 つの特殊な場合を除き、1 行で複数の変数を初期化することは通常ありません。

古風な C プログラマーであれば、上記のコードを以下のように 4 行で記述することも考えられます。

```
int i, j, k;
i = 3;
j = 8;
k = 9;
```

このように、3 行でまとめた Java スタイルのほうが多少簡潔になっています。Java スタイルは宣言と初期化を結合していることが、その理由です。

変数はループの内側に入れること

よく発生する 1 つの特殊なケースとは、ループの外側での変数宣言です。例えば、リスト 6 の単純な `for` ループは、フィボナッチ数列の最初の 20 の項を計算します。

リスト 6. C プログラマーはループの外側で変数を宣言する傾向があります

```
int high = 1;
int low = 1;
int tmp;
int i;
for (i = 1; i < 20; i++) {
  System.out.println(high);
  tmp = high;
  high = high+ low;
  low = tmp;
}
```

4 つの変数はすべてループの外側で宣言されているため、ループのなかでしか使用されないにも関わらず、変数には過大なスコープが設定されています。このように過大に設定されたスコープは、バグが発生する原因となります。それは、その変数の使用が意図されたスコープ以外でも変数が再利用される可能性があるためです。特に複数の変数が `i` や `tmp` などの共通の名前を持つ場合には、その可能性は尚更大きくなります。変数が再利用されると、1 回使用された値がそのまま残り、以降のコードに予期せぬ影響を及ぼしかねません。

最初の改善策 (C の最近のバージョンでもサポートされています) は、`i` ループ変数宣言を、このループの内側に移動することです (リスト 7 を参照)。

リスト 7. 変数をループの内側に移動する

```
int high = 1;
int low = 1;
int tmp;
for (int i = 1; i < 20; i++) {
  System.out.println(high);
  tmp = high;
  high = high+ low;
  low = tmp;
}
```

これで終わりではありません。経験を積んだ Java プログラマーであれば、`tmp` 変数ループの内側に移動するはずです (リスト 8 を参照)。

リスト 8. 一時変数をループの内側で宣言する

```
int high = 1;
int low = 1;
for (int i = 1; i < 20; i++) {
  System.out.println(high);
  int tmp = high;
  high = high+ low;
  low = tmp;
}
```

スピードに並々ならぬこだわりを持つ学生は、ループ内で必要以上の作業を行うとスピードが落ちてしまうと反論するでしょうが、変数宣言が実行時に行う作業はまったくありません。宣言をループの内側に移動することによって Java プラットフォームのパフォーマンスに影響が及ぶことは何もないというわけです。

熟練の Java プログラマーを含め、多くのプログラマーはここで作業を終わりにすることでしょう。けれどももう 1 つ、あまり使われていない手法ですが、すべての変数をループの内側に含める手法があります。実は、変数をカンマで区切るだけで、`for` ループの初期化フェーズで複数の変数を宣言することができます (リスト 9 を参照)。

リスト 9. すべての変数をループの内側に移動する手法

```
for (int i = 1, high = 1, low = 1; i < 20; i++) {
  System.out.println(high);
  int tmp = high;
  high = high+ low;
  low = tmp;
}
```

これで、単に慣用語法にかなったナチュラルな Java コードというだけでなく、まさにベテラン Java プログラマー流のコードになりました。Java コードでは、このようにローカル変数のスコープを絞り込むことができるため、Java コードで目にする `for` ループは C コードより遥かに多く、逆に `while` ループは C コードよりも遥かに少なくなっています。

変数の使い回しは禁物です

上記の説明から当然言えることは、Java プログラマーがローカル変数を異なる値やオブジェクトに再利用することはめったにないということです。一例として、リスト 10 ではアクション・リスナーを関連付けたいくつかのボタンをセットアップしています。

リスト 10. ローカル変数を使い回しているコード

```
Button b = new Button("Play");
b.addActionListener(new PlayAction());
b = new Button("Pause");
b.addActionListener(new PauseAction());
b = new Button("Rewind");
b.addActionListener(new RewindAction());
b = new Button("FastForward");
b.addActionListener(new FastForwardAction());
b = new Button("Stop");
b.addActionListener(new StopAction());
```

経験豊富な Java プログラマーは、これを 5 つの異なるローカル変数を使用して書き直します (リスト 11 を参照)。

リスト 11. 変数を使い回さないコード

```
Button play = new Button("Play");
play.addActionListener(new PlayAction());
Button pause = new Button("Pause");
pause.addActionListener(new PauseAction());
Button rewind = new Button("Rewind");
rewind.addActionListener(new RewindAction());
Button fastForward = new Button("FastForward");
fastForward.addActionListener(new FastForwardAction());
Button stop = new Button("Stop");
stop.addActionListener(new StopAction());
```

1 つのローカル変数を複数の論理的に異なる値やオブジェクトに再利用すると、バグが発生しやすくなります。基本的に、ローカル変数 (常にオブジェクトを指すわけではありませんが) にはメモリーの問題も、時間の問題もありません。恐れることなく、必要な数だけ異なるローカル変数を使用してください。

メモリーの管理はガーベッジ・コレクターに任せてください

主に C++ 言語を使用していたプログラマーは、メモリーの使用やメモリー・リークを心配し過ぎる傾向があり、その結果としてやってしまいがちな 2 つのことがあります。1 つは、変数を使い終わった時点で null を設定すること、そしてもう 1 つは `finalize()` を呼び出すか、`finalize()` をある種の擬似デストラクターとして使用することです。いずれの措置も、通常は必要ないものです。Java コードでどうしても手動でメモリーを解放しなければならないこともありますが、そのような場合は極めて稀です。大抵はガーベッジ・コレクターを頼りにすれば、メモリーの管理を賢明に、しかもかなり短時間で行うことができます。ほとんどの最適化と同様、最善の経験則は、メモリーの管理は自分で行わず、どうしても必要であると証明できるまで手を付けないことです。

優先されるプリミティブ・データ型を使用すること

Java 言語には 8 つのプリミティブ・データ型がありますが、使用されているのはそのうちの 6 つだけです。C コードとは異なり、Java コードでは `float` をほとんど使用しません。Java コードで `float` 変数やリテラルを目にすることはほとんどなく、それよりも `double` が優先して使用されています。`float` を使用するのは、ストレージ・スペースがかなりの大きさになりそうな、精度が限られた浮動小数点値からなる大きな多次元配列を操作する際のみです。これ以外の場合は、すべて `double` にしてください。

`float` よりも珍しいのは、`short` です。私は Java コードで short 変数を見たことはめったにありません。このデータ型が登場するとしたら、それは唯一 (これは極めて珍しいことであると警告しておきます)、16 ビットの符号付き整数型が含まれる外部定義のデータ・フォーマットが読み取られる場合のみです。この場合、大抵のプログラマーはただ単に `int` と理解します。

private のスコープ設定

リスト 12 に記載するような `equals()` メソッドを見たことはありますか？

リスト 12. C++ プログラマーが作成した `equals()` メソッド

```
public class Foo {

  private double x;

  public double getX() {
    return this.x;
  }

  public boolean equals(Object o) {
    if (o instanceof Foo) {
      Foo f = (Foo) o;
      return this.x == f.getX();
    }
    return false;
  }

}
```

このメソッドは技術的には間違っていませんが、このクラスは古い考えにとらわれた C++ プログラマーが作成したものだと保証してもよいくらいです。その根拠は、private フィールド `x` と public のゲッター・メソッド `getX()` が同じメソッド内の、まさに同じ行で使用されているという点です。C++ では private のスコープをクラスではなくオブジェクトに設定しなければならないため、こうする必要があります。それはすなわち、C++ では、同じクラスのオブジェクトは互いの private メンバー変数を見られないということです。互いの private メンバー変数を見るには、アクセサー・メソッドを使用しなければなりません。一方、Java 言語の場合、private のスコープはオブジェクトではなくクラスに設定されます。そのため、それぞれに `Foo` 型の 2 つのオブジェクトは互いの private フィールドに直接アクセスすることができます。

大抵の場合は関係ありませんが、いくつかの微妙な考慮事項により、Java コードではゲッターによるアクセスよりも直接フィールドへアクセスする方が優先される場合、あるいはその逆の場合があります。フィールドへの直接アクセスは、ゲッターによるアクセスより速いことがありますが、それは稀です。ゲッターによるアクセスは、特にサブクラスが関わってきた場合には、フィールドへの直接アクセスとは多少異なる結果をもたらすこともあります。しかし Java 言語では、フィールドへの直接アクセスとゲッターによるアクセスの両方を決して同じクラスの同じ行にある同じフィールドで使用してはいけません。

記号と構文の慣用表現

このセクションでは、C とは異なる Java の慣用表現をいくつか取り上げます。これらの慣用表現は、場合によっては Java 言語の特定の機能を利用するために必要となります。

配列の括弧は型に続けて配置すること

Java 言語での配列の宣言は、以下に示す C 言語の場合とほとんど同じです。

```
int k[];
double temperature[];
String names[];
```

ただし、Java 言語では配列の括弧を変数名の後に続けるのではなく、以下に示すように型の後に続ける構文を使用することもできます。

```
int[] k;
double[] temperatures;
String[] names;
```

ほとんどの Java プログラマーは後者のスタイルを採用しています。上記では、`k` の配列は `int` 型、`temperatures` の配列は `double` 型、`names` の配列は `String` 型となります。

また、他のローカル変数と同じく、Java プログラマーは配列が宣言された時点で初期化する傾向があります。

```
int[] k = new int[10];
double[] temperatures = new double[75];
String[] names = new String[32];
```

`null == s` ではなく、`s == null` とすること

慎重な C プログラマーは、リテラルは比較演算子の左側に置くように習慣付けています。以下の例を見てください。

```
if (7 == x) doSomething();
```

この習慣の目的は、等号を 2 つ続けた比較演算子の代わりに、以下のように誤って代入演算子である単一の等号を使用しないようにすることです。

```
if (7 = x) doSomething();
```

リテラルを左側に置くと、上記の例はコンパイル時にエラーとなります。リテラルを左側に配置するという手法は、C 言語では堅実なプログラミング・プラクティスで、真のバグを防ぐ効果があります。右側にリテラルを持ってくると、常に `true` が返されるからです。

しかし C 言語と違い、Java 言語には `int` 型と `boolean` 型のそれぞれがあります。代入演算子は `int` 型を返すのに対し、比較演算子は `boolean` 型を返します。したがって、`if (x = 7)` は明らかにコンパイル時にエラーとなります。つまり、ベテランの Java プログラマーであれば使わない、`if (7 == x)` という不自然な形を比較文に使用する理由は一切ないということです。

ストリングはフォーマット設定せずに連結すること

長年 Java 言語には `printf()` 関数がありませんでしたが、ついに Java 5 でこの関数が追加され、今ではよく使用されているようになっています。フォーマット・ストリングは、数値を特定の幅や小数点以下の桁数を指定してフォーマット設定する必要があるという特別な場合には、特に役立

つドメイン固有の言語です。けれども C プログラマーは Java コードで `printf()` を使いすぎる感があります。通常は、単純なストリング連結の代わりとして使用するべきではありません。一例として、以下のコードを見てください。

```
System.out.println("There were " + numErrors + " errors reported.");
```

上記のコードは、以下のようにするよりも適切です。

```
System.out.printf("There were %d errors reported.\n", numErrors);
```

単純な場合には特に、ストリング連結を使用したバージョンのほうが読みやすくなくなります。さらに、バグも発生しにくくなります。なぜなら、フォーマット・ストリング内のプレースホルダーと変数の引数の数または型が一致しなくなる危険がないためです。

プリインクリメントよりもポストインクリメントを優先して使用すること

`i++` と `++i` との違いが非常に重要になる箇所があります。Java プログラマーは、これらの箇所に特別な名前を付けています。それは、「バグ」です。

プリインクリメントとポストインクリメントとの違いに依存するコードは、決して作成しないでください (これは、C にも言えることです)。その理由は単に、理解しにくく、エラーの原因となりやすいためです。この 2 つのメソッドの違いが重要となるコードを作成していることに気付いたら、コードを編成し直して 2 つの文に分け、違いが重要でなくなるようにしてください。

プリインクリメントとポストインクリメントとの違いがそれほど重要でない場合 (例えば、`for` ループのインクリメント・ステップなど)、Java プログラマーは約 4 対 1 の割合で、ポストインクリメントをプリインクリメントに優先して使用しています。`i++` は `++i` よりも遥かによく使用されています。その理由を正当化することはできませんが、これが現状です。`++i` を使用すると、そのコードを読む誰もが、なぜこれが使用されているのかを疑問に思って時間を無駄にすることになります。したがって、プリインクリメントを使用する特別な理由がない限り (プリインクリメントを使用しなければならない理由が出てくることはないはずです)、常にポストインクリメントを使用してください。

エラー処理

エラー処理は Java プログラミングで最も複雑な問題の 1 つであり、エラー処理によって、この言語を完全にマスターした専門家であるかどうかが決まるほどです。実に、エラー処理の話題だけでも 1 つの記事が書けるくらいですが、重要なのは、例外を正しく使用し、エラー・コードが返されることがないようにすることです。

Java に転向したプログラマーが最も起こしがちな過ちは、例外をスローする代わりに、エラーを示す値を返すことです。実際この過ちは、Java 言語固有の API のいくつかにも見られます。これらの API の起源は初期の Java 1.0 に遡ります。その当時、Sun のすべてのプログラマーがこの新しい言語をまだ完全に習得しているわけではありませんでした。一例として、`java.io.File` にある以下の `delete()` メソッドを見てください。

```
public boolean delete()
```

このメソッドは、ファイルまたはディレクトリーが正常に削除されると `true` を返し、そうでなければ `false` を返します。しかし本来このメソッドは、削除が正常に完了した場合には何も返さず、何らかの理由でファイルを削除できない場合に例外をスローするべきです。

```
public void delete() throws IOException
```

メソッドがエラーの値を返すとなると、すべてのメソッド呼び出しがエラー処理コードのなかに囲み込まれてしまいます。これでは、何の問題もなく、すべて順調な場合 (通常はそうなります) にメソッドが辿る通常の実行フローを追いかけて理解するのが困難になってしまいます。その一方、例外によってエラー条件が示されれば、エラー処理の部分は、ファイルの後のほうで別のコード・ブロックのなかに含めることができます。あるいは、問題を処理するのにもっと適切な場所があれば、別のメソッドや別のクラスに移動することもできます。

このことから、話はエラー処理のもう 1 つのアンチパターンにつながります。C または C++ 出身のプログラマーは、例外がスローされた箇所のできるだけ近くで例外を処理しようするものです。そのため極端な例として、リスト 13 のようなコードになってしまいます。

リスト 13. あまりにも早急な例外処理

```
public void readNumberFromFile(String name) {
    FileInputStream in;
    try {
        in = new FileInputStream(name);
    } catch (FileNotFoundException e) {
        System.err.println(e.getMessage());
        return;
    }

    InputStreamReader reader;
    try {
        reader = new InputStreamReader(in, "UTF-8");
    } catch (UnsupportedEncodingException e) {
        System.err.println("This can't happen!");
        return;
    }


    BufferedReader buffer = new BufferedReader(reader);
    String line;
    try {
       line = buffer.readLine();
    } catch (IOException e) {
        System.err.println(e.getMessage());
        return;
    }

    double x;
    try {
        x = Double.parseDouble(line);
    }
    catch (NumberFormatException e) {
        System.err.println(e.getMessage());
        return;
    }

    System.out.println("Read: " + x);
}
```

これでは読みにくいだけでなく、例外処理に置き換わるように設計された `if (errorCondition)` テストよりも入り組んでいます。ナチュラルな Java コードでは、問題の発生箇所からエラー処理を遠ざけ、エラー処理コードと通常の実行フローとを混在させることはしません。上記のコードを改良したリスト 14 のバージョンのほうが、遥かにコードを追って理解するのが容易になります。

リスト 14. 例外のメイン・パスは 1 つのコードにまとめてください

```
public void readNumberFromFile(String name) {
    try {
        FileInputStream in = new FileInputStream(name);
        InputStreamReader reader = new InputStreamReader(in, "UTF-8");
        BufferedReader buffer = new BufferedReader(reader);
        String line = buffer.readLine();
        double x = Double.parseDouble(line);
        System.out.println("Read: " + x);
        in.close();
    }
    catch (NumberFormatException e) {
        System.err.println("Data format error");
    }
    catch (IOException e) {
        System.err.println("Error reading from file: " + name);
    }
}
```

場合によっては、同じ例外を発生させる異なる障害モードを、ネストした `try-catch` ブロックを使って分ける必要がありますが、これは珍しい場合です。一般的な経験則から言うと、メソッドに複数の `try` ブロックに相当するコードがあるとしたら、そのメソッドは大き過ぎるということになるので、いずれにしても複数の小さなメソッドに分割しなければなりません。

最後の注意点として、どの言語の出身であれ、Java プログラミングに馴染みのないプログラマーは、メソッド内のチェック例外は、それがスローされたメソッドでキャッチしなければならないと思い込みがちです。大抵の場合、例外をスローしたメソッドは、例外をキャッチしなければならないメソッドとは異なります。例えば、リスト 15 のようなストリームをコピーするメソッドについて考えてみてください。

リスト 15. あまりにも早急な例外処理

```
public static void copy(InputStream in, OutputStream out) {
  try {
    while (true) {
      int datum = in.read();
      if (datum == -1) break;
      out.write(datum);
    }
    out.flush();
  } catch (IOException ex) {
     System.err.println(ex.getMessage());
  }
}
```

上記のメソッドには、発生する可能性のある `IOException` を適切に処理するための情報が欠けています。例外の原因もわからなければ、その例外が発生したことによる結果もわからないため、このメソッドは唯一残された妥当な対処法として、呼び出し側が気付くまで `IOException` を放置するしかありません。リスト 16 に、このメソッドを作成する正しい方法を記載します。

リスト 16. すべての例外を最初にキャッチできる時点でキャッチしなければならないわけではありません

```
public static void copy(InputStream in, OutputStream out) throws IOException {
  while (true) {
    int datum = in.read();
    if (datum == -1) break;
    out.write(datum);
  }
  out.flush();
}
```

このとおり短くて単純な読みやすいコードになっています。そしてエラー情報に関しては、その情報を処理するのに最適なコードの部分に渡しています。

実際のところ、どれだけ重要な話なのでしょうか？

以上に説明した規則は、どれも重大な問題ではありません。しかしなかには、慣例とする正当な理由がある規則もあります。例えば、変数は最初に使用する時点で宣言すること、例外のスローはその処理内容が不明な場合に行うなどです。その他については、`argv` ではなく `args` を使用する、`++i` ではなく `i++` を使用するなどの、純粋にスタイル上の慣例です。これらの規則に従ったからと言って、実行速度が向上するとは言いません。また、バグの防止に役立つのは、このうちのわずかな規則だけです。けれどもナチュラルに Java 言語を使いこなす Java プログラマーになるためには、このすべての規則が必要となります。

良くも悪くも、訛りなしで話せば (つまり、コードを作成すれば)、他の人々からの尊敬が高まり、あなたの言葉に一層の注意が払われるようになるだけでなく、その言葉により多くの対価が払われるようになるでしょう。さらに、訛りのない Java 言語を話すのは、訛りのないフランス語や中国語、あるいは英語を話すよりもずっと簡単です。Java 言語をいったん学んだら、ネイティブと同じように話す一層の努力をする価値はあります。

著者について

Elliotte Rusty Harold

Elliotte Rusty Harold が初めて Java 言語を学んだのは 1995年のことです。それ以前は、最初に Fortran、次に Applesoft Basic を使用していました。C はおそらく彼の 3 番目の言語で、4 番目は Microphone II だったかもしれません。5 番目は Pascal でしたが、それほど深くこの言語に関わることはありませんでした。6 番目はおそらく IDL (Interactive Data Language)、7 番目は Perl でしょうか。Java は彼が学んだ 8 番目の言語で、他のどの言語より没頭した言語です。けれどもその後も、彼は新しい言語として PHP、AppleScript (Java より古いかもしれません)、XSLT、XQuery、C++、そして最近では Haskell を学んでいます。