NumberFormat の構文解析問題の解決

標準構文解析によって起こり得るデータ損失を防ぐ

Joe Sam Shirah
Principal and developer
conceptGO

2006年 10月 17日

Java™ SE (Standard Edition) API の NumberFormat クラスでは、数値を表す定形式テキストをプログラムで構文解析できます。追加設定なしですぐにローカライズを行える Java SE API は、すべての Java プログラマーにとって便利なツールですが、残念ながら、基礎となる DecimalFormat クラスによって予告もなしに符号やデータを損失する場合があります。この記事では、Joe Sam Shirah がこの問題を説明し、問題を正しく処理するためのコードを紹介します。

駆け出しのプログラマーは、数字のテキスト表現は、プログラムが数学演算を実行できる数値変数とは明確に異なるということにすぐに気付きます。例えば "123" は、実際の数値 123 または 16 進数 0x7B と同じではありません。プログラムがテキストから数値を得るには、アルゴリズムや変換ルーチンを使用する必要があります。とくに、テキストが桁区切りや小数点 (例えば、米国の数字フォーマットではカンマと小数点) を使ってフォーマット設定されている場合、それはなおさらのことです。テキストから数値への変換は主に対話型プログラムの懸念事項ですが、HTML と XML、そしてデータをテキストとして扱うその他のファイル・フォーマットや通信フォーマットにも必要になることがよくあります。

Java SE API は Integer.parseInt() および Double.parseDouble() などの変換メソッドを提供しますが、これらのメソッドではフォーム内の引数が Java 言語仕様 (「参考文献」を参照) のリテラルに対して定義されている必要があります。この記事では主に整数と倍精度に注目するため、フォーマットを構成するのは当然、以下の文字だけになります。

- 先頭の負符号 (ASCII 値 45、または 16 進数 0x2D)
- 数値 0 ~ 9 (ASCII 値 48 ~ 57、または 16 進数 0x30 ~ 0x39)
- 浮動小数点数の場合、ドットまたはピリオドで表された小数点 (ASCII 値 46、または 16 進数 0x2E)

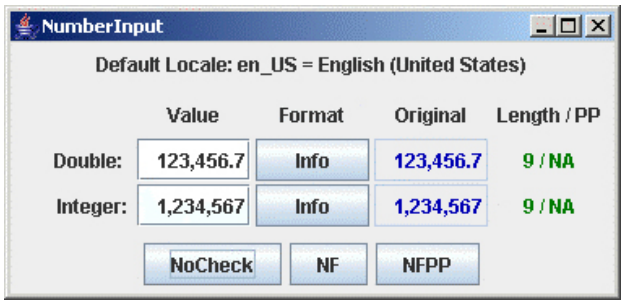
上記はプログラマーとコードには妥当な要件ですが、ユーザーはそれぞれの地域の文化に共通のフォーマットで数字を入力したり、表示させたいと思うものです。Java SE API の java.text.NumberFormat クラスには、ほとんどのプログラマーがロケール固有の定形式テキストを

数値に構文解析するために使用する便利なメソッド、parse(String source) が含まれています。ですが残念なことに、このメソッドは予期しない、しかも正確ではない結果をもたらすことがあります。この記事では、NumberFormat の論理的根拠を説明し、その機能性を検討してクラスの正しい構文解析方法を突き止めるとともに、このクラスを確実に使用するためのガイドラインを紹介します。

検証を行わない構文解析

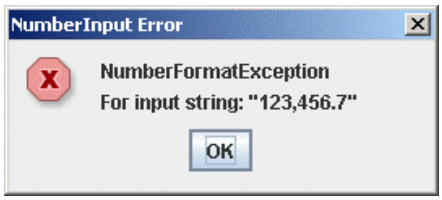
図 1 に示したこの記事のサンプル・プログラム (「ダウンロード」を参照)、NumberInput は、テキスト入力を数値に変換するいくつかのメソッドを検討するための Swing アプリケーションです。このプログラムは入力フィールドの他、デフォルトのロケール名、本来の入力値、値の元の長さ、そして構文解析された位置を表す数 (該当する場合) を表示します。プログラムは起動時に、入力フィールドのそれぞれに倍精度値の 123456.7 と整数値の 1234567 を読み込みます。両方の値は、デフォルト・ロケールに合わせて、現地のユーザーが期待するようなフォーマットで設定されます。ここではアメリカ用として、倍精度値は「123,456.7」、整数値は「1,234,567」と表示されています。

図 1. NumberInput の初期表示



「NoCheck」ボタンをクリックすると、プログラムはDouble.parseDouble() と Integer.parseInt() を使用して、一切の検証を行わずに構文解析だけを行います。その他のメソッドを呼び出す前に、actionPerformed() の入力ストリングからは先頭と末尾のスペースが除去されることに注意してください。図 2 に、結果を示します。

図 2. Double.parseDouble() がスローする NumberFormatExceptionfor の定形式テキスト



このエラーの原因は、アメリカのロケールに合わせてカンマが桁区切りとして使用されていることです。カンマをなくして入力を「123456.7」にすると、プログラムは倍精度値を許容します。

それでは負の値の場合はどうでしょう。プログラムは先頭に入力した符号 (さらに、カンマを除去) は許容しますが、末尾の符号については NumberFormatException for input string: "123456.7-" という結果になります。

整数の構文解析でも、上記と同じ理由から同様の動作になります。「NoCheck」ボタンに対して呼び出されるコードは、NumberInput の noCheckInput() メソッドに含まれています。このコードは JTextField jtD からの入力には Double.parseDouble() を使用し、JTextField jtI からの入力には Integer.parseInt() を使用します。このような結果はルールどおりに従ったもので、まったくの初心者レベルだけではなく、ほとんどの Java プログラマーにとって期待通りの動作であるはずです。

NumberFormat による救いの手

入力ミスやその他のユーザー入力エラーはまったく別として、表示するフォーマット化された数値テキストと、これと同じフィールドから受け取る数値テキストの入力との間には常にやっかいな関係が存在します。おそらくプログラマーがこの問題に対するソリューションとして選ぶのは、「カンマと先頭の負符号を使わずに数字を入力してください」というようなメッセージを表示することでしょう。データ入力スタッフにとってこの方法は大した問題になりませんが、ユーザーが通常求めるのは、フォーマット化された数字が表示された定様式表示画面で、区切り文字や桁をそのまま残して数字を直接入力することです。たいていの場合、アメリカのプログラマーは気が進まないながらも問題解決の最初の一歩として、カンマを除去し、末尾の負符号を入力値の先頭に移動させるルーチンを作成します。このように作成されたプログラムの多くは、実際に長い間生かされています。ある意味、これはプログラマーの最初の国際化 (I18N) およびローカライズ (L10N) への進出でもありました (「参考文献」を参照)。問題は、このようなコードがプログラムを効率的にローカライズするのは、唯一あるいは限られた数のロケールに対してのみであるという点です。

Java プログラムはあらゆる対応プラットフォームで実行可能であるとうたわれており、多くの人はこれが、どの国でもどの言語でも同じように適用されると解釈しています。Java SE SDK はこのような期待を現実にするための API を提供していますが、上記で説明した方法で作成されたプログラムは、想定されている範囲を超えて使用されると、すぐに支障をきたしはじめます。例えば 123456.7 という値は、国によって「123.456,7」、「123456,7」、「123'456,7」など、さまざまにフォーマット化または入力されるためです。すべてのロケールで同じ桁区切りと小数点 (繰り返しますが、アメリカの場合はそれぞれ「,」と「.」) を使用することを前提としたプログラムでは当然、対応できません。この問題を見込んで、API には java.text.NumberFormat が組み込まれています。このクラスが提供する見かけは単純なメソッド、parse() および format() は、記号のフォーマット設定に関する知識を組み込んで自動的にロケールを認識します。実際、NumberInput は NumberFormat を使用して、入力フィールドに表示された値をフォーマット設定しています。

Java の Locale オブジェクトは言語と地域または国の特定の組み合わせを表し、識別します。このオブジェクト自体はローカライズされた動作を提供しません。クラスが、クラス自体をローカライズする必要があります。ただし、Java プラットフォームでは一貫したロケールのセットをサポートするため、標準クラスの多くが、一貫してローカライズされた動作を実装します。これらのクラスのメソッドには通常、2 つのバージョンがあります。Locale 引数を使用するバージョンと、デフォルトを前提とするバージョンです。デフォルト・ロケールはプログラムの起動時に自動的に決定されるか、あるいはJava ランタイムに渡された引数によってオーバーライドされます。

NumberFormat は抽象クラスですが、フォーマットを事前定義およびローカライズした具体的な実装を取得する静的ファクトリー・メソッド、getXXXInstance() を提供します。基礎となる実装は通常、java.text.DecimalFormat のインスタンスです。この記事でのコードと説明では、倍精度

値のフォーマット設定と構文解析には NumberFormat.getNumberInstance() から戻されたデフォルト、整数値のフォーマット設定と構文解析には NumberFormat.getIntegerInstance() から戻されたデフォルトを使用します。

注目に値するのは、構文解析を完全にローカライズするのに必要なコードがどれだけ少ないかという点です。この手順は以下のとおりです。

1. NumberFormat インスタンスを取得します。
2. String を Number に構文解析します。
3. 該当する数値を取得します。

作業がこれだけ少ないことは大きな利点になるため、すべての Java プログラマーはフォーマット設定された数値変換の処理に NumberFormat を使用するべきです。各種のロケールで試してみるには、以下のコマンド・ラインを使用して NumberInput アプリケーションを起動してください。ここで、lc は ISO-639 の言語コード、cc は ISO-3166 の国コードです。

```
java -Duser.language=lc -Duser.region=cc NumberInput
```

JDK 1.4 では、user.region の代わりに user.country システム・プロパティーを使用できます。Java プラットフォームでサポートされるロケールを調べるには、JDK 資料 (「参考文献」を参照) の「Internationalization」セクションに記載されている「Supported Locals」を参照してください。java.util.Locale の静的メソッド getAvailableLocales() を使用すると、プログラムが実行時にロケール・サポートを判別できます。

リスト 1 に、 NumberInput の NFInput() メソッドに関するコードを示します。このコードは、「NF」ボタンをクリックすると呼び出されます。このメソッドは、NumberFormat.parse(String) を使用して検証と変換を行います。

リスト 1. NFInput() が使用する NumberFormat.parse(String)

```
...
NumberFormat nfDLocal =
   NumberFormat.getNumberInstance(),
             nfILocal =
   NumberFormat.getIntegerInstance();
...

  public void NFInput( String sDouble, String sInt )
  { // "standard" NumberFormat parsing
    double d;
    int    i;
    Number n;

    try
    {
      n = nfDLocal.parse( sDouble );
      d = n.doubleValue();
      ...
      n = nfILocal.parse( sInt );
      i = n.intValue();
      ...
    }
    catch( ParseException pe )
    {
      ...
    }
```

```
  } // end NFInput
```

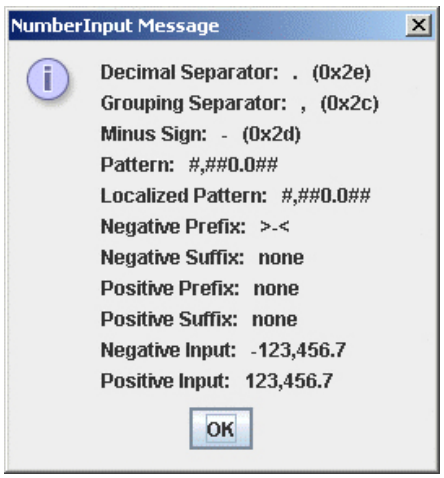
NumberFormat インスタンスの実装

ここで、NumberFormat に getXXXInstance() が要求されたときの動作と、DecimalFormatSymbols クラスの役割の両方を簡単に説明しておきましょう。この説明は、J2SE 1.4 に付属の参照用実装ソース・コードに基づいているため、内容が変更されることがあります。

基本的なイベントの流れは、まず、NumberFormat が関連ロケールに基づく適切なパターンを内部 ListResourceBundle で参照し、そのパターンで作成した DecimalFormat オブジェクトを戻します。明示的な負のパターンがない場合、正のパターンとの組み合わせで先頭の負符号が想定されます。プロセス中に、ロケールに対応した DecimalFormatSymbols オブジェクトが作成され、DecimalFormat インスタンスがこのオブジェクトの参照を取得します。NumberFormat.getXXXInstance() メソッドはファクトリー・パターンに基づいているため、その他の実装や今後の参照用実装では別のクラスを戻す可能性があります。そのため、カスタム・コードは、関連付けられた DecimalFormatSymbols インスタンスにアクセスする前に、DecimalFormat が確実に戻されるようにする必要があります。

DecimalFormatSymbols オブジェクトには、該当する小数点、桁区切り、負符号などの情報が含まれています。NumberInput はこの情報を収集し、「Info」ボタンがクリックされたときに収集した情報をダイアログに表示します。図 3 に、en_US ロケールを使用した場合の例を示します。この情報は、ローカライズされたフォーマットの数字を構文解析して検証する際に重大な意味を持ちます。

図 3. DecimalFormatSymbols データ



「NF」ボタンをクリックしてみてください。カンマやその他のローカル桁区切りが使用されている値でも、正常に受け入れられることがわかります。負符号も現行パターンに従って配置されていれば受け入れられます。それでは、負符号が別の場所にあったとしたらどうなるでしょう。それが、他のいくつかの問題とともに次のセクションで取り上げる話題で、この記事の大きな展開となります。

UnexpectedResults.equals(bigTrouble)

Java の国際化対応に関する大多数の記事は、NumberFormat のフォーマット化機能に重点を置き、この記事のここまでの内容を形を変えて説明してから構文解析の話題を締めくくっています。残念ながら、クラス (実際は、NumberFormat.getXXXInstance() から戻される具体的な DecimalFormat サブクラス) でのテストや実験は、構文解析の意外な欠点を浮き彫りにします。フィールドに共通の多数の条件により、NumberFormat.parse(String) はプログラマーに何も知らせずに潔くデータを切り捨て、符号をなくします。このような動作を取らせるのは、以下の条件です (明記していない限り、en_US ロケールが使用されています)。

- **小数点の前に連続的または不規則に挿入された複数の桁区切りは無視されます。** 例えば、「123,,,456.7」と「123,45,6.7」が受け入れられると、どちらも 123456.7 を戻します。 よくよく考えた末、この動作は技術的にはエラーになりますが失われるデータはないため、対処に値するソリューションはないという結論にたどり着きました。この動作は認識しておく必要はありますが、NumberInput アプリケーションではこれを修正しません。この記事では以降、このことについては触れません。
- **小数点の後の桁区切りは切り捨てられます。** 「123,456.7,85」は 123456.7 として受け入れられます。
- **複数の小数点は切り捨てられます。** 「123,456..7」は 123456.0 として受け入れられ、「12.3.456.7」は 12.3 として受け入れられます。
- **先頭に負符号 (負の接頭辞) があるパターンの場合、組み込み負符号を含め、数字以外の文字がある場所で切り捨てが行われます。** 「123,4r56.7」は 1234.0 として受け入れられ、「12-3,456.7」は 12.0 (正の値) として受け入れられます。
- **末尾に負符号 (負の接尾辞) があるパターンの場合、組み込み負符号を除き、数字以外の文字がある場所で切り捨てが行われます。組み込み負符号は受け入れられますが、追加データは切り捨てられます。** サウジアラビアのロケール (ar_SA) の場合、「123,4r56.7」は 1234.0 として受け入れられ、「12-3,456.7」は -12.0 (負の値) として受け入れられます。
- **パターンが負の入力については先頭に負符号を指定している場合、末尾の負符号は無視されます。** 「123,456.7-」は 123456.7 (正の値) として受け入れられ、「-123.456,7-」(オランダのロケール nl_NL) は -123456.7 (負の値) として受け入れられます。

図 4 と図 5 に、「NF」ボタンがクリックされたときのこれらの動作の例を示します。元の入力が何を意図していたかを解釈するのは困難ですが、1234.0 という倍精度値の結果が期待されていたのでも、ユーザーが整数値の入力から最後の 2 桁を切り捨てようとしたのでもないことは確かです。この場合も例外はスローされず、入力の一部が無視されたことについての通知はありません。

図 4. NumberFormat.parse(String) での予期しない結果

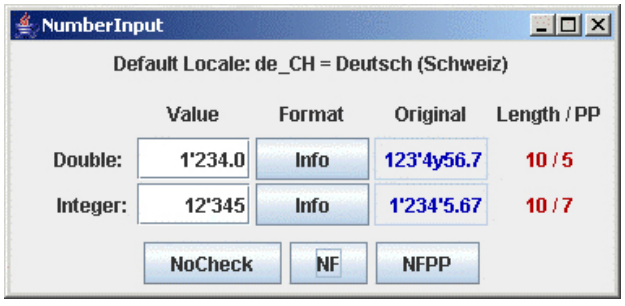
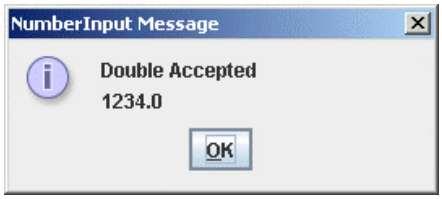


図 5. NumberFormat.parse(String) が受け入れる不完全な値



このような結果は、JDK 1.4 および 5.0 で何度テストしても同じことで、NumberFormat および DecimalFormat クラスの実装に費やした作業量を考えると理解に苦しみます。その一方で、引数をメソッドに渡して結果を調べるより、コードのほうがわかりやすいというわけでもありません。原因を解く真実かつ唯一の手掛かりは、NumberFormat.parse(String source) に関する JDK 資料に記載されています。そこには詳細は説明されていませんが、「このメソッドは、特定のストリングのテキスト全体を使用しない場合がある」と書かれています。

このような異常なことを前提とするのはやっかいで、一見すると、「自分なりの方法で入力する」というプログラミング手法に立ち返ったほうがいいように思えるかもしれません。「ゴミを入れれば、ゴミしか出てこない」というのはコンピューターの世界での決まり文句ですが、これが意味しているのは、プログラムがデータの正確性を保証するのは不可能であるため、プログラマーがその義務として、入力されたすべてのものが有効であることをできるだけ確実にしなければならないということです。入力ストリングの一部から可能な場合に限り数字を戻すというのはバグではなく、むしろ NumberFormat.parse(String) の設計そのもののようです。残念ながら、この動作にはデータがすでに検証済みであるという暗黙の前提が含まれています。入力が無効であるかどうかをプログラマーが判別できないという結末は、ユーザーそしてデータ自体との暗黙の契約を破ることになります。

数年前にこの問題が判明した際、私は最初の対応として、フロントエンド・プリプロセッサーに相当するものを parse(String) メソッドに作成しました。これは上手くいきましたが、その代価は、一部が重複するコードを追加しなければならなかったことと、データの処理時間が長くなったことです。幸いにも、既存の NumberFormat メソッドを慎重に使用すれば、この問題を解決できることがわかりました。

NumberFormat.parse(String, ParsePosition) による検証

parse(String source, ParsePosition parsePosition) メソッドは、例外をスローしないという点で特殊です。このメソッドは通常、単一のストリングから複数の数字を構文解析する場合に使用します。ただし、メソッドが戻った時点での ParsePosition.getIndex() からの値は、入力ストリングで最後に構文解析された位置に 1 を足した値になります。つまり、コードが常にゼロに設定されたインデックスで始まっていれば、処理の完了時に、インデックス値は解析された文字数と同じになります。このメソッドを検証に使用する場合の鍵は、更新されたインデックスを元の入力ストリングの長さと比較することです。

混乱を避けるため触れておきますが、ParsePosition には getErrorIndex() メソッドもあります。ここで説明している条件ではエラーが検出されないため、このメソッドは当然役に立ちません。さらに、このメソッドを使用する場合は、毎回構文解析の操作を行う前にエラー・インデックスを -1 にリセットしておかないと誤った結果になります。

NumberInput アプリケーションでは、「NF」または「NFPP」ボタンをクリックすると、「Length/PP」列の下に ParsePosition のインデックスが表示されます。元の値の長さがゼロより大きく、インデックス値と同じである場合は、両方の値が緑色で表示されます。それ以外の場合、値は赤で表示されます。この操作は、特定の検証メソッドとは別に行われます。図 4 をもう一度見てください。「NF」ボタンに関連付けられた NFInput() メソッドがデータを受け入れたのにも関わらず、この図では値はエラーを示す赤で表示されています。

最後の検証バージョンでは、「NFPP」ボタンをクリックして NFPPInput() メソッドを呼び出します。このメソッドは parse(String, ParsePosition) を使用して入力を検証し、数値を取得します。図 6 と図 7 では、図 4 での無効な入力が、NFPPInput() で検出されています。私が行ったテストでは、このメソッドは NumberFormat.parse(String) では見逃していたすべての条件を適切に処理しました。

図 6. 無効な倍精度値入力の検出

図 7. 無効な整数値入力の検出

parse(String, ParsePosition) で正しい結果を得るためには、以下のガイドラインに従う必要があります。

- **このメソッドは決して例外をスローしないということを覚えておいてください。** 説明をわかりやすくするため、ここに記載したコードは Accelitable/Unaccelitable ダイアログだけを表示しています。一般目的で使用する場合、スローする ParseExcelition はより一般的な期待に合わせたものにしなければなりません。
- **parse(String, ParsePosition) を呼び出す前には必ず、ParsePosition のインデックスをゼロにリセットしてください。** リセットが必要なのは、このメソッドでは入力ストリング内の ParsePosition のインデックスで構文解析を開始するためです。
- **倍精度値の構文解析には NumberFormat.getNumberInstance() を使用し、整数値の構文解析には NumberFormat.getIntegerInstance() を使用します。** 整数値に整数インスタンスを使用 (あるいは、数字インスタンスに setParseIntegerOnly(true) を適用) しないと、メソッドは小数点を飛ばして入力ストリングの終わりまで構文解析します。その結果、長さとインデックスが一致し、無効な入力を受け入れることになってしまいます。
- **長さとインデックス値が同じであるかを比較するだけでなく、構文解析後の Number がヌルではないか、または空の入力ストリング ("" またはゼロの長さ) がないかどうかを調べてください。** 入力フィールドをクリアすると、ストリングは空になります。この場合、長さとイン

デックス値はどちらもゼロになって一致します。また、構文解析メソッドは、空のストリング入力にはヌルを戻します。この動作は、NumberFormat.parse(String source) を使用した場合の空ストリングの結果 (「構文解析不可の数字」の ParseExcelition がスローされます) とは異なります。parse(String source, ParsePosition parsePosition) は決して例外をスローすることがないことを忘れないでください。NumberInput では、リスト 2 のコード・スニペットを使用してエラーの可能性を処理します。

リスト 2. エラー条件のチェック

```
if( sDouble.length() != pp.getIndex() ||
    n == null )
{ /* error */ }
```

入力の処理手順をまとめると、以下のようになります。

1. 適切な NumberFormat を取得して、ParsePosition 変数を定義します。
2. ParsePosition のインデックスをゼロに設定します。
3. parse(String source, ParsePosition parsePosition) を使用して入力値を構文解析します。
4. 入力の長さと ParsePosition のインデックス値が一致しない場合、または構文解析された Number がヌルの場合はエラー操作を実行します。
5. それ以外の場合、値は検証に合格しました。

リスト 3 に、関連コードを示します。

リスト 3. NFPPInput() メソッド

```
...
NumberFormat  nfDLocal =
   NumberFormat.getNumberInstance(),
              nfILocal =
   NumberFormat.getIntegerInstance();

ParsePosition pp;
...

  public void NFPPInput( String sDouble,
                         String sInt )
  { // validate NumberFormat with ParsePosition
    Number n;
    double d;
    int    i;

    pp.setIndex( 0 );
    n = nfDLocal.parse( sDouble, pp );

    if( sDouble.length() != pp.getIndex() ||
        n == null )
    {
      showErrorMsg(
        "Double Input Not Acceptable\n" +
         "\"" + sDouble + "\"");
    }
    else
    {
      d = n.doubleValue();
      jtD.setText( nfDLocal.format( d ) );
      showInfoMsg( "Double Accepted \n" + d );
    }
```

```
   pp.setIndex( 0 );
   n = nfILocal.parse( sInt, pp );
   if( sInt.length() != pp.getIndex()  ||
       n == null )
   {
     showErrorMsg(
       "Int Input Not Acceptable \n" +
        "\"" + sInt + "\"");
   }
   else
   {
     i = n.intValue();
     jtI.setText( nfILocal.format( i ) );
     showInfoMsg( "Int Accepted \n" + i );
   }
 } // end NFPPInput
```

まとめ

Java SE API には多大な作業が費やされて、バイトコード・レベルだけでなく、国際化およびローカライズ化されたアプリケーションも含めて「一度作成すれば、場所を選ばずに実行」できるようになっています。NumberFormat および DecimalFormat クラスは、世界クラスのアプリケーションを作成することを目指す Java プログラマーには、なくてはならないものです。ただし、この記事で説明したように、開発者は現状の parse(String source) メソッドを使う限り、完璧な入力を前提としなければなりません (現実の世界ではまれなことです)。この記事に記載した情報とコードは、parse(String source, ParsePosition parsePosition) を使用して入力が無効であるかどうかを判断し、正しい結果を得るための代替手段になります。

ダウンロード

内容	ファイル名	サイズ
NumberInput source code and classes	j-numberformat.zip	8KB

著者について

Joe Sam Shirah

Joe Sam Shirah は、conceptGO の代表兼開発者です。クライアントの要求を満足させるかたわら、developerWorks および Sun 開発者向けサイトに多数のチュートリアルを発表しています。Java Community Award の受賞者である彼は、developerWorks の Java Filter Forum のモデレーターでもあり、jGuru の JDBC、I18N、そして Java400 FAQs を管理しています。