多忙な Java 開発者のための Scala ガイド: オブジェクト指向のための関数型プログラミング

2 つの世界の利点を最大限に活用する Scala を学ぶ

Ted Neward
Principal
Neward & Associates

2008年 1月 22日

Java™ プラットフォームは、その登場以来ずっとオブジェクト指向プログラミングの領域のものでしたが、熱烈な Java 言語支持者でさえ、アプリケーション開発における『古いものは新しい』という最近の傾向である、関数型プログラミングに注目し始めています。この新しいシリーズでは、Ted Neward が Scala を紹介します。Scala は JVM に対して関数型の手法とオブジェクト指向型の手法を組み合わせたプログラミング言語です。Scala を紹介する中で、なぜ Scala を学ぶために時間を費やす必要があるのか、その理由 (1 つは並行性です) を説明し、また学んだことをすぐに生かせることを示します。

このシリーズの他の記事を見る

初恋は忘れられないものです。

私の場合、彼女の名前は Tabinda (Bindi) Khan です。それは私の若き日の、正確には 7 年生の時の、幸せな日々でした。彼女は美しく、明晰であり、そして何よりも、10 代の少年のつまらない冗談にも笑ってくれたのです。私達は 7 年生と 8 年生の大部分の間、時々一緒に時間を過ごしました (当時はそれを「going out」と言っていました)。しかし 9 年生の頃になると、私達は徐々に会わなくなりました。それは彼女が、10 代の少年のつまらない冗談を 2 年間も聞き続けて飽きてしまったことを、失礼がないように示したのだと思います。私は (特に、高校の卒業 10 年後の同窓会で彼女と再会したことから) 決して彼女を忘れませんが、もっと重要なこととして、こうした (いくらか誇張されているかもしれない) 大切な思い出を決して忘れないでしょう。

このシリーズについて

このシリーズでは、Ted Neward が皆さんと共に Scala プログラミング言語を深く掘り下げます。developerWorks の、この新しいシリーズでは、Scala が最近もてはやされている理由を調べ、Scala の言語機能の実際の動作を調べます。Scala のコードと Java のコードの比較が重要な場合には両者のコードを並べて示しますが、(これから学ぶように) Scala の機能のうちの多くは、Java プログラミングには直接対応するものがありません。そして Scala の魅

© Copyright IBM Corporation 2008

力の多くがあるのはそこなのです。結局のところ、Java コードで可能ならば、手間をかけて Scala を学ぶ必要はないのです。

Java プログラミング、そしてオブジェクト指向は、多くのプログラマーにとって初恋でした。そして私達はそれを、私が初恋の人 Bindi に対して示すのと同じ敬意と真摯な憧憬をもって扱います。一部の開発者は、メモリー管理と C++ による地獄の業火から Java プログラミングによって救われたと語ります。他の人達は、手続き型による絶望の深みから Java プログラミングによって抜け出すことができたと語ります。さらには、自分たちにとって Java コードでのオブジェクト指向プログラミングこそ「私達がいつも行ってきたやり方」と言う開発者さえいます (そして何と、この言葉は私の父にも、そのまた父にも、都合のよい言葉だったのです)。

しかし、すべての初恋は最終的に時間に打ち負かされ、新たな旅立ちの時が訪れます。気持ちは変わり、物語を演じる役者は成熟しています (そして新しい冗談もいくつか学んでいるはずです)。しかしもっと重要なこととして、私達を取り巻く世界が変化を遂げています。多くの Java 開発者は、Java プログラミングを必要としつつも、開発の世界の中での新しい機会を捉え、それをどう利用できるかを考える時が来ていると感じています。

私は常にあなたを愛しています

ここ 5 年ぐらいの間に、Java 言語に対する不満が大きく高まってきています。一部の人は、その主な原因が Ruby on Rails の成長にあると指摘しますが、私は RoR (Ruby の識者はそう呼びます) は結果であって原因ではないと思います。あるいは、おそらくもっと正確に言えば、Ruby を取り上げている Java 開発者の存在こそが、いつのまにか原因がより深く浸透している影響が現れた結果なのです。

簡単に言えば、Java プログラミングは古くなり始めています。

あるいは、もっと正確に言えば、Java 言語が古くなり始めているのです。

次のことを考えてみてください。Java 言語が初めて誕生したときは、第一期クリントン政権の時代で、インターネットを定常的に使用しているのは本格的な技術専門家のみでしたが、それは家でインターネットに接続するためにはダイアルアップ以外に方法がなかったからです。ブログはまだ発明されておらず、継承こそが再利用のための基本な手法だと誰もが信じていました。また私達は、世界をモデリングするための方法としてオブジェクトが最適であり、ムーアの法則は指数関数的に永遠に有効だと信じていました。

実際のところ、業界の多くの人が特に関心を持ったのががムーアの法則です。しかし 2002年から2003年以降、マイクロプロセッサーの大きな傾向として、複数の「コア」を持つ CPU が作成されるようになってきました。要するに、1 つのチップの中に複数の CPU を入れることです。これによって、CPU のスピードは 18 カ月ごとに 2 倍になる、というムーアの法則は不要になりました。1 つの CPU で標準的なラウンドロビン・サイクルを行うのではなく、2 つの CPU で同時に実行するマルチスレッド環境を利用するということは、コードを確実にスレッドセーフにしない限り使用に耐えない、ということです。

この特定の問題に関して、学術界では大いに研究が行われてきており、その結果、大量の新しい言語が誕生しています。ただし致命的な欠陥として、こうした言語の大部分は独自の仮想マシンまたはインタープリターの上に構築されています。つまりこうした言語は (Ruby が示しているよ

うに)、新しいプラットフォームへの移行が必要なことを示しているのです。並行性の不足は大きな懸念事項であり、一部の新しい言語はそれに対する強力な答えを提供していますが、多くの企業や団体は、ほんの 10 年前に C++ から Java プラットフォームに移行したときのことを覚えています。新しいプラットフォームへの移行はリスクが高く、多くの会社は真剣に移行を検討することはできません。実際のところ、多くの人々は、前回の Java プラットフォームへの移行の傷をまだ癒している最中なのです。

そこに Scala が登場します。

スケーラブルな言語 Scala (SCAlable LAnguage)

なぜ Scala なのか

新しいプログラミング言語の学習は、いつも気の重い作業です。特に、Scala に採用されている関数型の手法のように問題へのアプローチの手法にまったく新しい考え方が必要な言語の場合、さらにはその手法に別の手法が混合されている場合には、特に気が重いものです (Scala ではオブジェクト指向の概念と関数型の概念が混合されています)。Scala を学ぶためには時間がかかります。しかも皆さんは期限に追われた多忙なスケジュールのため、投資の回収ができるのかどうか一目見て判断することはできないかもしれません。私は皆さんに保証します。Scala には数々の魅力的な特徴があり、その多くをこのシリーズの今後の記事で解説していきます。以下に挙げるのは、この言語に取り組むメリットをわかりやすく抜粋して並べたものです。つまり Scala によって以下のことが可能になります。

- **内部 DSL を作成することができます** (Ruby を考えてみてください)。これは識別子に関する Scala の柔軟性のおかげです。
- **非常にスケーラブルで並行処理の行えるデータ・プロセッサーを作成することができます。**これは Scala ではデフォルトで状態が不変なことによるものです。
- **等価な Java コードを削減することができます。**削減される量は、半分あるいは 3 分の 2 程度です。これは、Scala の持つさまざまな構文機能 (クロージャーや暗黙的な定義など) によるものです。
- **並列ハードウェア・アーキテクチャーを活用することができます (マルチコア CPU など)。**これは Scala が関数型の設計を推奨しているためです。
- **大規模なコード・ベースを理解することができます。**これは Scala が一部の型ルールを単純化し、言ってみれば「すべてがオブジェクトである」ことを要求しているためです。

確かに Scala は、プログラミングに関する強力な新しい見方を示しています。JVM にコンパイルでき、そして JVM 上で実行するという事実のおかげで、「実際の作業」での Scala の使い方は非常に容易になっています。

Scala は、関数型とオブジェクト指向を混合した言語であり、以下のいくつか強力な要因が有効に働きます。

- 第 1 に、Scala は Java バイトコードにコンパイルされます。これは Scala をJVM 上で実行できるということです。Java によるオープンソースのリッチなエコシステムを相変わらず活用できることに加えて、マイグレーション作業をまったく必要とせず、Scala を既存の IT 環境の中に統合することができます。
- 第 2 に、Scala は Haskell と ML の関数型の原則に基づいていますが、Java プログラマーが愛する、おなじみのオブジェクト指向の概念も大いに借用しています。その結果、両方の世界の最良の部分を一体に融合することができ、これまで依存してきた慣れ親しんできた部分を犠牲にせずに大きな効果を発揮することができます。
- 最後に、Scala は Martin Odersky によって開発されました。彼は Java コミュニティーでは、おそらく Pizza 言語と GJ 言語で最もよく知られています。GJ 言語は Java 5 の Generics の実動

> プロトタイプとなりました。そのため、Scala には「真剣さ」が伴っています。つまりこの言語は思いつきで作られたものではなく、また思いつきで見捨ててしまうものでもありません。

Scala という名前が示すように、Scala は非常にスケーラブルな言語でもあります。この点については、このシリーズをもう少し進んでから詳しく説明する予定です。

Scala をダウンロードしてインストールする

Scala のバンドルは Scala の Web サイトからダウンロードすることができます。この記事の執筆時点での最新リリースは 2.6.1-final です。このバンドルの入手形式には、Java インストーラー・バージョン、RPM パッケージと Debian パッケージ、ターゲット・ディレクトリーに簡単に解凍できる gzip/bz2/zip バンドル、そしてゼロから作成するためのソース tarball があります。(Debian ユーザーの場合は、「apt-get install」で簡単にインストールできる、バージョン 2.5.0-1 を Debian の Web サイトで入手することができます。ただし 2.6 バージョンには微妙な違いがいくつかあるため、Scala の Web サイトから直接ダウンロードしてインストールする方法をお勧めします)。

適当なターゲット・ディレクトリーに Scala をインストールします。私は Windows® 環境でこの記事を書いているため、私の場合のディレクトリーは C:/Prg/scala-2.6.1-final です。環境変数 `SCALA_HOME` にこのディレクトリーを設定し、コマンドラインから呼び出しやすいように、`SCALA_HOME\bin` を `PATH` に設定します。インストールされたシステムをテストするには、単にコマンド・プロンプトから「`scalac -version`」と起動するだけです。すると Scala version, 2.6.1-final という応答が返ってくるはずです。

関数型の概念

始める前に、なぜ Scala がそのように解釈し、そのように動作するのかを理解するために必要な、関数型の概念をいくつか説明しましょう。Haskell や ML、あるいはもっと最近関数型の世界に登場した F# など、関数型言語を少し経験したことのある人は、このセクションをスキップして次のセクションに進んでください。

関数型言語という名前は、プログラムは数学関数のように振る舞う必要があるという概念から来ています。つまり一連の入力を与えると、関数は必ず同じ出力を返す必要があります。これは、すべての関数は値を返さなければならないことを意味しているだけではなく、関数は本質的に、1 つの呼び出しから次の呼び出しに移る間に関数固有の状態を保持してはならない、ということも意味しています。関数に本来備わっているこのステートレスという概念は、関数型の世界/オブジェクト指向の世界に持ち込まれて当初より不変オブジェクトを意味するようになり、際立った並行型の世界で偉大な救世主として関数型言語が熱烈に歓迎されている大きな理由の 1 つになっています。

最近になって Java プラットフォーム上に独自の空間を作り始めた多くの動的言語とは異なり、Scala は Java コードとまったく同じように静的型付けの言語です。しかし Java プラットフォームとは異なり、Scala は型推論 (type inferencing) を多用します。つまり、特定の値がどんな型なのか、プログラマーの介在なしにコンパイラーがコードを詳しく分析します。型推論では冗長な型コードが少なくてすみます。例えば、ローカル変数を宣言し、それらの変数に値を割り当てるために必要な Java コードを考えてみてください (リスト 1)。

リスト 1. javac は天才です (ため息)

```
class BrainDead {
  public static void main(String[] args) {
    String message = "Why does javac need to be told message is a String?" +
      "What else could it be if I'm assigning a String to it?";
  }
}
```

後ほど説明するように、Scala では、このように手動で型を保持する必要はありません。

その他無数の関数型機能 (パターン・マッチングなど) が Scala 言語の中に取り入れられています。しかし、それらのすべてを記載してしまうと、話を先回りしてしまうことになります。また Scala には、Java プログラミングに現在欠けている、いくつかの機能も追加されています。例えば演算子のオーバーロード (これは調べてみればわかるように、大部分の Java 開発者が想像しているものとはまったく異なります) や、「上方および下方型制限」付きの Generics、そしてビューなどです。何よりもこれらの機能によって、XML の処理や生成など、ある種のタスクを処理する際に Scala はとても大きな力を発揮します。

しかし抽象的な概要はもう十分です。プログラマーはコードを見たがるものです。では Scala ではどんなことができるのかを見て行きましょう。

Scala を知る

最初の Scala プログラムは、コンピューター・サイエンスの神が求める標準的なデモ・プログラム、Hello World です。

リスト 2. Hello.Scala

```
object HelloWorld {
  def main(args: Array[String]): Unit = {
    System.out.println("Hello, Scala!")
  }
}
```

これを `scalac Hello.scala` を使ってコンパイルし、そしてコンパイルされたコードを、Scala のランチャー (`scala HelloWorld`) または従来の Java ランチャーを使って実行します。その際、Scala のコア・ライブラリーを JVM のクラスパスに含めます (`java -classpath %SCALA_HOME%\lib \scala-library.jar;. HelloWorld`)。どちらの方法で実行しても、おなじみの挨拶の画面が表示されるはずです。

リスト 2 のいくつかの要素は、まず確実に皆さんにおなじみでしょう。しかしここでは、いくつかの非常に新しい要素も使われています。例えば、おなじみの `System.out.println` への呼び出しで開始している点は、ベースとなっている Java プラットフォームに対する Scala の忠誠を表しています。Scala は Java プラットフォームの力を Scala プログラムで最大限に利用するために大きな努力をしています。(実際 Scala では、Scala の型を Java クラスから継承することができ、その逆も可能です。これについては後ほど説明します。)

その一方、注意力の鋭い人は、`System.out.println` 呼び出しの最後にセミコロンがないことに気付いたかもしれません。これはタイプミスではありません。Scala では Java プラットフォームとは異なり、もし行の終わりで文が終了することが明白な場合には、文を終了させるためのセミコロ

ンを必要としません。しかしセミコロンは相変わらずサポートされており、必要な場合もあります。例えば、物理的に同じ行に複数の文がある場合などです。ほとんどの場合、新進の Scala プログラマーは単純にセミコロンを省略することができ、Scala コンパイラーはセミコロンが必要な場合には (通常は、非常に目立つエラー・メッセージで) 優しく注意を喚起してくれます。

また、これは些細な点ですが、Scala では、クラス定義を含むファイルがそのクラスの名前を反映している必要がありません。一部の人はこれを、Java プログラミングから変更された新鮮な点と思うでしょう。そう思わない人達は、クラスとファイル名の対応という Java の命名規則をそのまま使い続けることもできます。

今度は、従来の Java (オブジェクト指向) コードから本格的に分岐を始める部分を調べることにしましょう。

関数とフォームがついに統合

まず、Java の熱狂的な愛好者は、HelloWorld が「`class`」の代わりに `object` というキーワードを使って定義されていることに気付くはずです。これは、Scala が Singleton パターンの普及に賛同していることを示しています。つまり `object` キーワードは Scala コンパイラーに対して、これが Singleton オブジェクトであることを伝え、それによって Scala は、`HelloWorld` のインスタンスが必ず 1 つしか存在しないことを保証します。これと同じ理由から、Java プログラミングの場合とは異なり、`main` が静的なメソッドとして定義されていないことに注目してください。実際、Scala は「`static`」を使うことを完全に回避しています。もしアプリケーションが、ある型を持つインスタンスと、何らかの種類の「グローバル」インスタンスの両方を必要とする場合には、Scala アプリケーションは `class` 定義と、そのクラスと同じ名前の `object` 定義の両方を許可します。

次に、`main` の定義に注目してください。`main` は Java コードの場合と同じく、Scala プログラムへのエントリー・ポイントと考えられています。main の定義は Java の場合とは異なるように見えますが、実は同じです。つまり `main` は `String` の配列を引数として取り、何も返しません。しかし Scala での `main` の定義は Java の場合とほんの少し異なるようです。`args` パラメーターは `args: Array[String]` のように定義されます。

Scala では、配列はジェネリックな `Array` クラスのインスタンスとして表現されます。このことはまた、Scala ではパラメーター化された型を示すために不等号括弧 (<>) ではなく大括弧 ([]) を使うことも示しています。さらに、この「`name: type`」というパターンは、一貫性を保持するために Scalar 言語のすべての部分に現れます。

従来の他の関数型言語と同じく、Scala では関数 (この場合はメソッド) は必ず値を返す必要があります。そのため Scala は、`Unit` と呼ばれる、「値ではない」値を返します。実際上は、Java 開発者は (少なくとも当面の間は) `Unit` のことを `void` と同じと考えることができます。

メソッド定義の構文は「`=`」演算子を使うため、なにやら興味深いものに見えます。これはまるで、この演算子の後に続くメソッド本体を識別子 `main` に割り当てているように見えます。実はこれは、実際に行われていることなのです。つまり関数型言語では、関数は (変数や定数と同じく) ファーストクラスの概念であり、従って構文的にもファーストクラスとして扱われます。

クロージャーと言いましたか

関数が持つ性質の中でファーストクラスの概念を表している点の 1 つは、関数は何らかの形で独立した構成体として認識されなければならないという点です。この独立した構成体はクロージャーという別名でも知られ、Java コミュニティーでは最近このクロージャーに関してホットな議論が行われています。Scala では、関数は容易にクロージャーとして認識されます。クロージャーの威力を実際に示す前に、リスト 3 の単純な Scala プログラムを考えてみてください。このプログラムでは、関数 `oncePerSecond()` が 1 秒ごとに 1 回この関数のロジック (ここでは Systems.out への出力) を繰り返しで実行します。

リスト 3. Timer1.scala

```
object Timer
{
  def oncePerSecond(): Unit =
  {
    while (true)
    {
      System.out.println("Time flies when you're having fun(ctionally)...")
      Thread.sleep(1000)
    }
  }

  def main(args: Array[String]): Unit =
  {
    oncePerSecond()
  }
}
```

残念なことに、このコードはあまり関数型とは言えず、便利でもありません。例えば、表示されるメッセージを変更したいとすると、`oncePerSecond` メソッドの本体を変更しなければなりません。従来の Java プログラマーがこれを行うのであれば、表示するメッセージを含めるように、`oncePerSecond` に対して `String` パラメーターを定義するでしょう。しかしそうしたとしても、それは非常に限定的な方法です。他のすべての周期的なタスク (例えばリモート・サーバーに ping を送信するなど) には、それぞれ独自のバージョンの oncePerSecond が必要になり、DRY (Don't Repeat Yourself) の原則に明らかに違反します。リスト 4 に示すように、クロージャーにはこれに代わる柔軟で強力な方法があります。

リスト 4. Timer2.scala

```
object Timer
{
  def oncePerSecond(callback: () => Unit): Unit =
  {
    while (true)
    {
      callback()
      Thread.sleep(1000)
    }
  }

  def timeFlies(): Unit =
  { Console.println("Time flies when you're having fun(ctionally)..."); }

  def main(args: Array[String]): Unit =
  {
    oncePerSecond(timeFlies)
  }
}
```

今度は、少し興味深い様相を見せ始めました。リスト 4 では、関数 `oncePerSecond` はパラメーターを 1 つ取りますが、その型が奇妙です。正式には、`callback` と呼ばれるパラメーターは関数をパラメーターとして取ります。これが成立するのは、渡される関数がパラメーターを取らず (「()」で示されます)、返すもの (「=>」で示されます) がない (関数型の値「Unit」で示されます) 場合のみです。そして次に注目すべき点として、ループ本体の中で、渡されたパラメーター関数オブジェクトを `callback` を使って呼び出しています。

幸い、そうした関数はプログラムの中の他の部分、`timeFlies` という部分にあります。そこで、単純にこの関数を `main` 内部から `oncePerSecond` 関数に渡します。(`timeFlies` が Scala で導入されたクラス `Console` を使っていることにも気付く人がいるかもしれません。Console は `System.out` クラスや新しい `java.io.Console` クラスと同じ基本的な役割を果たします。これは純粋に美的感覚の問題であり、ここでは `System.out` と `Console` のどちらでも機能します。)

匿名関数、その機能は何ですか

さて、この `timeFlies` 関数は無駄なもののように見えます。この関数は結局のところ、`oncePerSecond` 関数に渡される、という目的以外に何の役割も果たしていません。そこで、この関数を正式に定義することを完全にやめることにします (リスト 5)。

リスト 5. Timer3.scala

```
object Timer
{
  def oncePerSecond(callback: () => Unit): Unit =
  {
    while (true)
    {
      callback()
      Thread.sleep(1000)
    }
  }

  def main(args: Array[String]): Unit =
  {
    oncePerSecond(() =>
      Console.println("Time flies... oh, you get the idea."))
  }
}
```

リスト 5 では、main 関数は `oncePerSecond` へのパラメーターとして、任意のコード・ブロックを渡し、Lisp あるいは Scheme のラムダ式のように世界中を探します。実際、ラムダ式も別の種類のクロージャーです。この匿名関数も関数を第一級市民として扱うことの強力さを示しており、匿名関数を使うことで、継承を超えたまったく新しい次元でコードをジェネリックにすることができます。(Strategy パターンのファンは、既にコントロールできずによだれを垂らしているかもしれません。)

実際のところ、これでも `oncePerSecond` は具体的すぎます。つまりコールバックを毎秒呼び出す、という不合理な制約を課しています。これをさらにジェネリックにするためには、渡された関数をどの程度の頻度で呼び出すかを示す、2 つ目のパラメーターを取るようにします (リスト 6)。

リスト 6. Timer4.scala

```
object Timer
{
  def periodicCall(seconds: Int, callback: () => Unit): Unit =
  {
    while (true)
    {
      callback()
      Thread.sleep(seconds * 1000)
    }
  }

  def main(args: Array[String]): Unit =
  {
    periodicCall(1, () =>
      Console.println("Time flies... oh, you get the idea."))
  }
}
```

これは関数型言語では一般的なテーマです。つまり 1 つのことを行う上位レベルの抽象関数を作成し、その関数がパラメーターとしてコード・ブロック (匿名関数) を取るようにし、そしてそのコード・ブロックをその上位レベルの関数の中から呼び出すのです。例えば、オブジェクトのコレクションをウォークスルーする場合を考えてみてください。for ループの中で従来の Java のイテレーター・オブジェクトを使うのではなく、代わりに、1 つのパラメーター (繰り返しの対象となるオブジェクト) を取る関数を持つコレクション・クラスに対して、関数型のライブラリーによって (通常は「iter」または「map」と呼ばれる) 関数が定義されます。そのため、例えば先ほど触れた `Array` クラスは、リスト 7 で定義される関数 `filter` を持ちます。

リスト 7. Array.scala のリストの抜粋

```
class Array[A]
{
    // ...
   def filter  (p : (A) => Boolean) : Array[A] = ... // not shown
}
```

リスト 7 は、`p` が、`A` によって指定される Generics 型のパラメーターを取り、ブール値を返す関数であることを宣言しています。Scala のドキュメンテーションによれば、`filter` は「述部 p を満足するすべての要素で構成された配列を返します。」これはつまり、少しの間 Hello World プログラムに戻り、「G」という文字で始まるすべてのコマンドラインの引数を見つけたい場合には、リスト 8 のように単純に書くことができる、ということです。

リスト 8. Hello, G-men!

```
object HelloWorld
{
  def main(args: Array[String]): Unit = {
    args.filter( (arg:String) => arg.startsWith("G") )
        .foreach( (arg:String) => Console.println("Found " + arg) )
  }
}
```

ここでは、`filter` はパラメーターとして述部、つまり暗黙的にブール値 (`startsWith()` 呼び出しの結果) を返す匿名関数を取り、「`args`」配列の中にあるすべての要素を使ってその述部を呼び出

します。もし述部が真を返したら、その述部は結果の配列に追加されます。「`args`」配列全体をウォークスルーしたら、結果の配列を返します。この結果の配列は「`foreach`」呼び出しのソースとして即座に使われます。「`foreach`」呼び出しはその名前が示すとおりのことをします。つまり `foreach` は配列の中のすべての要素に対して、別の関数を適用します (この場合は単純に各要素を表示します)。

上記の `HelloG.scala` に等価な Java 版がどのようなものかを想像するのは難しくはありません。また、そして Scala 版の方が Java 版よりもはるかに短く、はるかに明確であることも容易にわかります。

まとめ

Scala でのプログラミングは、すぐに興味が持てるほど馴染み深いおなじみのものでもあり、また同時に異なったものでもあります。皆さんがこれまで長年にわたって学び、そして好んで使ってきたものと同じ、Java のコア・オブジェクトで作業できるという点では似ていますが、プログラムを部分に分解して考える必要があるという点で明らかに異なっています。「多忙な Java 開発者のための Scala ガイド」シリーズの第 1 回である今回は、Scala によって何ができるのかを、ほんの少し紹介しました。まだまだ内容は山ほどあります。しかし次回までしばらくの間、関数型を楽しんでください。

著者について

Ted Neward

Ted Neward は、Neward & Associates の代表として、Java や .NET、XML サービスなどのプラットフォームに関するコンサルティング、助言、指導、講演を行っています。彼はワシントン州シアトルの近郊に在住です。

© Copyright IBM Corporation 2008
(www.ibm.com/legal/copytrade.shtml)
商標
(www.ibm.com/developerworks/jp/ibm/trademarks/)