Java™ プラットフォーム 移行ガイド

バージョン 1.3 から 5.0 へ

このガイドは、開発者が、Java™ アプレット、スタンドアロンアプリケーション、Java™ Web Start アプリケーション、および開発ツールを、Java プラットフォームのバージョン 1.3 からバージョン 5.0 へ移行する際に役立ちます。多くの 1.3 アプリケーションは変更しなくても動作しますが、このガイドで説明するように、互換性の問題が存在します。最初の 3 つの章では、すべてのアプリケーション開発者に関係する問題について説明します。第 4 章では、主にプラットフォーム実装者とツール開発者に関係する問題について説明します。

注 : このガイドでは、「1.4」は「J2SE™ プラットフォームのバージョン 1.4.x」を、「5.0」は「Java プラットフォームのバージョン 5.0」を、それぞれ意味します。

最新リリースにおける最新の問題や既知のバグについては、リリースノートを必ずお読みください。

目次

第 1 章
実行時の問題

以下の問題は、5.0 プラットフォーム上で実行される 1.3 クラスに影響します。

1.1 AWT

AWT GUI コンポーネントライブラリが変更された結果、クロスプラットフォーム動作、パフォーマンス、および軽量 GUI コンポーネントとの相互運用性が改善されました。

1.1.1 MouseEvent.MOUSE_LAST

1.4 で、クラス `java.awt.event.MouseEvent` の static final フィールド `MOUSE_LAST` の値が `507` に変更されました。以前のバージョンでは、この値は `506` でした。

static final 値は、コンパイラによってコンパイル時にハードコードされるため、`MOUSE_LAST` を参照するコードおよび 1.3 の下でコンパイルされたコードは、その古い値を保持しています。そのようなコードは、5.0 で正しく動作するためにはコンパイルし直す必要があります。

1.1.2 AWT フォーカス関連の変更

AWT 1.3 アプリケーション開発者の大部分は 1.4 への移行時に非互換性の問題に直面したという経緯があるので、5.0 における 1.3 アプリケーションのフォーカス動作を確認することをお勧めします。ここでは一般的な問題について概説します。Windows 固有の問題については次節で説明します。これらの問題の詳細や関連するアーキテクチャーの変更点については、「The AWT Focus Subsystem」を参照してください。

1. 5.0 では、すべてのコンポーネントのフォーカストラバース可否が、デフォルトで `true` になっています。以前は、一部のコンポーネント (特にすべての軽量コンポーネント) のフォーカストラバース可否が、デフォルトで `false` になっていました。

   注 : この変更にもかかわらず、すべての AWT コンテナの `DefaultFocusTraversalPolicy` は、以前のリリースのトラバーサル順序をそのまま維持しています。

2. フォーカストラバースが不可能な (つまり、フォーカスが不可能な) コンポーネントへのフォーカス要求は、5.0 では拒否されます。以前は、そのような要求が許可されていました。
3. 5.0 では、ウィンドウが表示されていない場合には、`Window.toFront()` と `Window.toBack()` は何の操作も行いません。以前は、この動作はプラットフォームに依存していました。

4. フォーカストラバーサルキー (ほとんどの場合、これは TAB キーを意味する) が消費されるようになりました。このため、これらのキーのイベント通知を受け取るキーリスナーに依存するプログラムで問題が発生する可能性があります。以前は、AWT コンポーネントはこれらのイベントを認識して、AWT がフォーカストラバーサルを開始する前に、それらを消費する機会を得ていました。フォーカストラバーサルキーが消費されるのを防ぐには、次のコードを使用します。

```
component.setFocusTraversalKeysEnabled(false);
```

ここで、component は、キーイベントを生成するコンポーネントです。これで、フォーカストラバーサルを手動で処理できるようになります。別の方法として、コード内で AWTEventListener または KeyEventDispatcher を使ってすべてのキーイベントを事前待機することもできます。詳細については、バグ 4650902 を参照してください。

5. 1.4 で、java.awt.event.FocusEvent と java.awt.event.WindowEvent に「opposite」フィールドが追加されました。たとえば、WindowEvent の場合、その状態変化に関与する他方のウィンドウが「opposite」になります。イベントの生成元がアクティブになっていた場合、非アクティブになったウィンドウが opposite になります。その逆も成り立ちます。FocusEvent の場合、フォーカス移動に関与する他方のコンポーネントが opposite になります。イベントの生成元がフォーカスを取得していた場合、フォーカスを失ったコンポーネントが opposite になります。その逆も成り立ちます。

ほかのイベントの継承元となる java.util.EventObject の source フィールドは、一時的なものです。FocusEvent.opposite と WindowEvent.opposite は一時的ではないため、それらの直列化と直列化復元は意味を持ちません。このため、1.4.2 では、直列化復元後に WindowEvent.opposite と FocusEvent.opposite が null に設定されます。

この変更に関連するバグ報告は、4759974 です。

6. 5.0 では、任意のコンテナがフォーカストラバーサルポリシーを提供できます。そのポリシーの提供有無を示すのが、Container の新しい FocusTraversalPolicyProvider プロパティーです。以前は、フォーカストラバーサルポリシーを提供できるコンテナは、フォーカスサイクルルートだけでした。

5.0 では、Java プラットフォームに付属するフォーカストラバーサルポリシーが変更され、フォーカストラバーサルポリシープロバイダが提供されるようになりました。具体的には、あるポリシーが順方向 (逆方向) トラバーサルの実行中にフォーカストラバーサルポリシープロバイダを検出した場合、そのプロバイダのコンポーネントを与えられたフォーカスサイクルルートに属するものとして処理するのではなく、そのプロバイダのフォーカストラバーサルポリシーを使って次の (前の) コンポーネントを取得するべきです。その戻り値のコンポーネントが、フォーカストラバーサルポリシープロバイダのフォーカストラバーサルポリシーから返された最初の (最後の) コンポーネントと同一であった場合、そのポリシーの呼び出し時に、フォーカストラバーサルポリシープロバイダのあとの (前の) サイクル内における次の (前の) コンポーネントが取得されるようにするべきです。フォーカスサイクルルートにおける「最初」と「最後」のコンポーネントを算出する際には、必要に応じて (「最初」または「最後」のコンポーネント自身がコンテナであり、かつフォーカストラバーサルポリシープロバイダである場合に)、フォーカストラバーサルポリシープロバイダのフォーカストラバーサルポリシーを使用するべきです。

この変更ではフォーカストラバーサルポリシー内に新しいメソッドを導入する必要がないため、サードパーティー製のフォーカストラバーサルポリシーも引き続き動作します。ただし、それらのポリシーではプロバイダの概念はサポートされません。

新しく記述したフォーカストラバーサルポリシーでプロバイダをサポートするには、5.0 のプラットフォームに付属するポリシーに行われたのものと同様の変更を行う必要があります。

詳細については、フォーカス仕様「The AWT Focus Subsystem」の「Focus Traversal Policy Providers」節を参照してください。

1.1.3 Microsoft Windows における AWT フォーカス関連の変更

1. Window.toBack() は、Z オーダー変更後の最上位のウィンドウにフォーカスを移動します。

2. `requestFocus()` で、どのような場合でもウィンドウ間フォーカス変更を要求できるようになりました。以前は、重量コンポーネントでは要求が許可されていましたが、軽量コンポーネントでは拒否されていました。

1.1.4 Solaris™/Linux における AWT (XToolkit / XAWT)

5.0 で、Solaris™ および Linux プラットフォーム上の AWT が実装し直されました。新しいツールキット実装には次の利点があります。

- Motif および Xt ライブラリに対する依存性の除去
- ほかの GUI ツールキットとの相互運用性の向上
- パフォーマンスと品質の向上

新しいツールキット (XToolkit) は、Linux 上における 5.0 のデフォルトです。Solaris 上では引き続き、MToolkit (Motif ベースのツールキット) が 5.0 のデフォルトとして使用されますが、これも最終的には XToolkit に置き換えられる予定です。

特定のアプレットやアプリケーションに対するツールキットを明示的に設定するには、環境変数またはシステムプロパティーを使用するか、または Java Plug-In コントロールパネルを使用できます。環境変数がシステムプロパティーよりも優先される点に注意してください。

環境変数の設定

VM を起動する前に環境変数を設定できます。

csh

```
setenv AWT_TOOLKIT XToolkit # XToolkit を選択
setenv AWT_TOOLKIT MToolkit # MToolkit を選択
```

ksh/bash

```
export AWT_TOOLKIT=XToolkit
export AWT_TOOLKIT=MToolkit
```

システムプロパティーの設定

別の方法として、コマンド行でシステムプロパティーを使用することもできます。

```
java -Dawt.toolkit=sun.awt.X11.XToolkit MyApp
java -Dawt.toolkit=sun.awt.motif.MToolkit MyApp
```

Java Plug-In コントロールパネルの使用

- Java Plug-In コントロールパネルを起動します。

  ```
  $java_home/bin/ControlPanel
  ```

- 「Java ランタイムパラメータ」フィールドにシステムプロパティーを追加します。このフィールドにアクセスするには、「Java」タブの「Java アプレットのランタイム設定」ボタンまたは「Java アプリケーションのランタイム設定」ボタンをクリックします。

  ```
  -Dawt.toolkit=sun.awt.X11.XToolkit
  -Dawt.toolkit=sun.awt.motif.MToolkit
  ```

アプレット用ツールキットの設定

端末ウィンドウからブラウザを起動する場合、ブラウザを起動する前に端末ウィンドウ内で環境変数を設定できます。

デスクトップアイコンやメニューからブラウザを起動する場合は、ブラウザ用の環境変数を設定する方法がないため、Java Plug-In コントロールパネルを使用します。

1.1.5 AWT ドラッグ＆ドロップ

1. 1.4.0 で、バグ 4395290 を修正するためにドラッグ＆ドロップ動作が変更されました。新しい動作では、DropTarget とそれに登録されている DropTargetListener で dragExit() が呼び出されるのは、ドラッグ操作中にこの DropTarget のドロップ領域の動作可能部分の外側にマウスポインタが出た場合だけになりました。以前は、DropTarget または DropTargetListener 上で `drop()` が呼び出される直前に、これらのメソッドもそれぞれ呼び出されていました。この古い動作はドラッグ＆ドロップ仕様に記載されていましたが、それは Java 2 プラットフォームの API 仕様と矛盾していました。また、この古い動作は不便でもあり、Swing のドラッグ＆ドロップ操作のユーザビリティーに多大な悪影響も与えていました。J2SE 1.4.0 以降、この動作は Java 2 プラットフォームの API 仕様と一致するようになりました。この変更を反映するためにドラッグ＆ドロップ仕様は更新されました。
2. バグ 4426794 と 4435403 を修正するための動作変更が、1.4.0 で導入されました。1.4 以降では、マウスカーソルの下にある最上位のコンポーネントがアクティブなドロップターゲットを持っている場合に、そのコンポーネントにドラッグ通知がディスパッチされます。以前は、マウスカーソルの下にある最上位のコンポーネントに常にドラッグ通知がディスパッチされていました。このコンポーネントにドロップターゲットが関連付けられていなかった場合、その通知は破棄されました。この設計には次の 2 つの欠点がありました。
   - ドラッグ通知は Swing ガラス区画を無視する必要がありました。そうしないと、すべてのドラッグ通知が消費されてしまいます。ところが、その場合、開発者は Swing ガラス区画をドロップターゲットとして使用できません。
   - `JColorChooser` などの複合コンポーネント上にドロップサポートを実装し、その内側の任意の場所へのドロップを可能にするには、開発者は、その複合コンポーネントのすべての下位クラスにドロップターゲットをインストールする必要があり、そうした方法は面倒であり、非効率的でした。
3. 以前は、X11 上でサポートされるドラッグ＆ドロップ (DnD) プロトコルは、Motif DnD プロトコルだけでした。5.0 では、XDND プロトコルもサポートされるようになったほか、Motif DnD プロトコルが Motif ライブラリに依存しないように実装し直されました。新しい Motif DnD プロトコル実装と Motif ライブラリが提供する実装との差異により、機能退化が発生する可能性があります。ただし、Motif ライブラリの実装にはバグが多く、新しい実装は少なくともそれ以上の品質を備えており、かつそのサポートは Motif ライブラリのものよりも充実していると考えられます。

   この変更の詳細については、バグ 4638443 を参照してください。

1.2 Java2D™ グラフィックス

この節では、Java2D™ のグラフィックス、フォントレンダリング、およびイメージ処理に関する問題を列挙します。変更の基盤となるアーキテクチャーの詳細については、次のリソースを参照してください。

- 「High Performance Graphics」白書
- 「Java 2 SDK, v1.4 での Java2D の新機能」
- 「J2SE 5.0 での Java2D の新機能」
- 「Java2D テクノロジのシステムプロパティ」

1.2.1 動作の変更

1. 5.0 のフォントメトリックス情報は、1.3 の場合とは異なります。ANSI コードページフォントを使用しているプログラムの場合、この差異の影響はさほど大きくありませんが、コンポーネント数が増えるにつれて、その影響も次第に増大します。アジア言語のフォントを使用しているプログラムの場合、この差異の影響がすぐに現れる可能性があります。

1.4 で、計算がより正確になりました。以前は、計算時の丸め処理が不正確である場合が頻繁にありました。1.4 では、ライブラリから報告されるサイズが、フォントの実際のサイズにより正確に対応するようになりました。ただし、整数のみを報告する API では、依然として整数値として報告されます。

詳細については、バグ 4711444、4455492、および 4467709 を参照してください。

2. 1.4 で、`GlyphVector.getGlyphOutline` から返される輪郭 (線) と、`GlyphVector.getGlyphVisualBounds` から返される境界が、個々のグリフの原点の周りに別々に配置されるようになりました。以前は、それらの輪郭と境界は、点 (0, 0) の周りに配置されていました。この変更により、`GlyphVector.getGlyphLogicalBounds` の動作と矛盾しない結果が得られるようになります。

1.2.2 イメージ処理

1. 以前は、`Graphics.drawImage()` メソッドに null `Image` パラメータを渡すと、`NullPointerException` が発生していました。5.0 ではこの例外は発生しません。null `Image` を渡すアプリケーションは、Microsoft の仮想マシン上では動作していました。それらのアプリケーションは、新しい動作を備えた 5.0 の仮想マシン上でも動作します。

2. Microsoft Windows プラットフォーム上では、イメージスケーリングのハードウェア高速化にアクセスできます。ただし、レンダリングの質と整合性を保証する目的で、この機能はデフォルトでは無効になっています。ハードウェア高速化されたスケーリングを有効にするには、次の実行時フラグを使用します。ただし、このフラグを使用して配備するのは、アプリケーションが正常に動作することを確認したあとにしてください。

```
java -Dsun.java2d.ddscale=true
```

3. 1.4 で、`VolatileImage` クラスが導入されました。このクラスのイメージは、実行時プラットフォーム上で可能な場合にはハードウェア高速化されます。以前は、いくつかのプラットフォーム上 (特に Windows 上) で、高速化メモリー内に格納されたイメージが、アプリケーションの制御範囲外の理由により「失われる」ことがありました。この新しいイメージ型は、そうしたイメージ損失を回避するために作成されました。

   `VolatileImage` がハードウェア高速化を活用できる場合には、イメージとそのイメージから作成されたコピーの両方に対するレンダリング操作が高速化されます。これらの高速化を利用すれば、単純なアプリケーションでさえも、それまで可能であったより大きなパフォーマンス向上を達成できる可能性があります。

   注 : 1.4 で、Swing がバックバッファーで `VolatileImage` を使用するようになりました。このため、Swing アプリケーションはパフォーマンスの高速化を自動的に実現できます。

   5.0 の Java2D は、イメージの作成方法にかかわらず、イメージコピーの高速化サポートを提供します。Java2D が、あるイメージから別の宛先イメージまたは宛先ウィンドウへのコピーを検出すると、Java2D ライブラリはそのイメージのキャッシュバージョンまたは高速化バージョンを作成します。`VolatileImage`、`Component.createImage()` を使ってイメージが生成された場合でも、さらには「new `BufferedImage()`」を使ってイメージが手動で作成された場合でも、アプリケーションは自動的に高速化を活用します。高速化バージョンの使用は、高速化の実現方法、つまり Windows 向けの DirectX、すべてのプラットフォーム向けの OpenGL のいずれが使用されているかには依存しません。もちろん、その実現方法は、プラットフォームのフラグと実行時のフラグによって決定されます。

1.2.3 グラフィックスの高速化

新しい BufferStrategy クラス

1.4 で、グラフィックスのバッファリングをより簡単にする目的で、BufferStrategy クラスが導入されました。グラフィックスのダブルバッファリングを使用すると、アニメーションやレンダリング更新がよりスムーズになります。Swing のバックバッファーでもこの手法が使用されています。この新しいクラスを使用すれば、その機能をより簡単に実装できるようになります。さらに、この新しいクラスが提供する機能は、従来のイメージ型や新しい VolatileImage 型を使って手動で実現できる機能よりも強力です。

Windows 上での DirectX のサポート

Java2D は 1.4 から、Windows 上の DirectX グラフィックスライブラリを使用して基本的な GUI やグラフィックスオブジェクトの高速化を実現するようになりました。この実装により、Swing バックバッファーなどのオブジェクトがビデオメモリー (VRAM) 内に存在し、そのようなオフスクリーン表面に対する単純なレンダリング操作の高速化を実現できるようになります。DirectX を使用すると、場合によってはグラフィックスアーティファクトが発生する可能性があります。Java2D に用意されているいくつかのコマンド行フラグを使用すれば、ユーザーアプリケーションにおける問題の特定や回避がしやすくなります。

- **`-Dsun.java2d.d3d=false`** - このフラグを指定すると、Java2D が線やその他のエンティティーを描画する際に Direct3D を使用しなくなります。ビデオカードの中には、Direct3D ドライバに不具合がある場合があるため、このフラグを使用すれば、3D ドライバに関するそうした問題を特定できます。
- **`-Dsun.java2d.noddraw=true`** - これは、もっとも強烈なフラグです。これを指定すると、Direct3D による線描画や DirectDraw によるその他のすべての高速化機能を含む、Windows 上のすべてのハードウェア高速化が無効になります。

すべてのプラットフォーム上での OpenGL のサポート

5.0 では、OpenGL を使って Java2D を実行できます。したがって、半透明操作、テキスト平滑化操作、変換操作などの高度なレンダリング機能 (および GUI で使用されるより基本的な線描画 / 塗りつぶし / コピー操作) に対してハードウェア高速化を適用することで、アプリケーションのパフォーマンスを改善できます。OpenGL は、Sun が提供するすべてのプラットフォーム (Windows、Linux、および Solaris) 上でサポートされています。

ただし、ドライバサポートの一貫性に問題があるため、OpenGL レンダリングはどのプラットフォームにおいてもデフォルトでは有効になっていません。高速化を有効にするには、次のコマンド行フラグを使用します。

```
-Dsun.java2d.opengl=true
```

注 : このレンダリングアプローチのパフォーマンスと堅牢性は、使用するハードウェアプラットフォームによって異なります。特定のプラットフォーム上でこれを有効にしてアプリケーションを実行する場合は、必ず事前に入念なテストを行うべきです。不明なプラットフォーム上でこれを有効にした場合、望むような結果が得られない可能性があります。

1.2.4 X11 関連の改善点

Solaris および Linux 上で X11 の下で実行されるプログラムでは、大幅なパフォーマンスの改善が見られます。この節では、変更点を概説するとともに、その活用方法を示します。また、パフォーマンスをさらに調整するために設定可能なフラグについても説明します。

高速化イメージの読み取りパフォーマンスの向上

以前は、高速化イメージに対する半透明操作やスケーリング操作は低速でしたが、その原因は、VRAM からのイメージの読み取りが頻繁に発生することにありました。VRAM からの読み取りは、システムメモリーからの読み取りに比べると格段に低速です。半透明操作では読み取りが頻繁に必要になります。なぜなら、アルファ合成が読み取り、変更、書き込みの一連の操作をターゲットに対して実行するからです。高速化イメージからのスケーリングでは、ソースイメージまたはターゲットイメージからの読み取りを行わないと、操作が正常に実行されません。

1.4 で、Java2D は、イメージの読み取りが頻繁に発生していると、その表面をシステムメモリーに転送します。読み取り頻度が低下すると、Java2D は表面を VRAM に戻します。UNIX® 環境で作業している場合には、`J2D_PIXMAPS` 環境フラグを使ってこのヒューリスティックを上書きできます。詳細については、「Java2D テクノロジのシステムプロパティ」の「Linux と Solaris プラットフォームのシステムプロパティ」を参照してください。

リモート X サーバーのパフォーマンスの向上

UNIX または Linux 環境でグラフィックスを操作する場合、使用するマシンから X サーバーを実行すれば、リモートクライアント上でグラフィックス計算を実行できます。ただし、その場合、オフスクリーンイメージのパフォーマンスは決して高くありませんでした。なぜなら、オフスクリーンイメージがクライアント側に作成されたからです。イメージをターゲットにレンダリングし直す必要性が生じるたびに、そのイメージをリモート X クライアントからディスプレイにコピーする必要がありました。ダブルバッファリング用にオフスクリーンイメージを使用している場合には、画面が再描画されるたびにイメージがネットワーク経由でコピーされていました。

1.4 で、高速化オフスクリーンイメージが利用可能になり、画面に対してローカルなサーバー側にオフスクリーンイメージを作成できるようになりました。Java2D は、X プロトコル要求を使用することにより、ネットワーク上のサーバー側に存在するオフスクリーンイメージへのレンダリング内容とレンダリング方法を、X サーバーに対して指示します。ネットワーク経由で送信されるのは、X プロトコル要求だけです。イメージ自体はサーバー側に存在します。この変更により、Swing のパフォーマンスも改善されました。なぜなら、Swing はダブルバッファリングを使用するからです。Swing アプリケーションは、画面が再表示されるたびにバックバッファーがネットワーク経由でコピーされるのを待つ必要がなくなりました。

ただし、リモート X、DirectDraw のどちらの場合にも当てはまる欠点が、1 つあります。それは、平滑化とアルファ合成が高速化されないという点です。実際、リモート X 上での平滑化操作とアルファ合成操作は通常、1.3 の場合に比べて格段に低速です。なぜなら、X クライアントにイメージをコピーしてこれらの操作のいずれかを行なったあとで、その新しいイメージを再度サーバーにコピーする必要があるからです。Java2D チームは現在、将来のリリースに向けてこの問題の解決策を模索しています。この問題は一般に、Swing アプリケーションでは問題になりません。なぜなら、その大部分がアルファ合成や平滑化を使用していないからです。

ローカル X サーバーのパフォーマンスの向上

1.4 の Java2D は、Solaris および Linux 上のローカルディスプレイ環境では、共有メモリー拡張 (Shared Memory Extension) を使用します。共有メモリー拡張を使用すると、同一マシン上で動作する X サーバーと X クライアントが共有メモリーに一緒にアクセスできるようになるため、データの転送速度が向上します。この変更により、画面へのレンダリング時やイメージ処理時のパフォーマンスが向上します。

DGA が利用できない場合、ツールキットイメージおよび `Component.createImage` または `GraphicsConfiguration.createCompatibleImage` で作成されたイメージは、システムメモリー内ではなくピックスマップ内に格納されます。このため、X プロトコル要求による画面への高速コピーが可能となります。この動作をオーバーライドするには pmoffscreen 実行時フラグを使用します。このフラグについては、「Java2D テクノロジのシステムプロパティ」の「Linux と Solaris プラットフォームのシステムプロパティ」を参照してください。

共有メモリー内に存在しないピックスマップは VRAM 内に格納される可能性があるため、画面へのコピーは高速です。しかし、「高速化イメージの読み取りパフォーマンスの向上」節で前述したように、それらのイメージからの読み取りは非常に低速です。ローカルディスプレイ環境では共有メモリー拡張にアクセスできるようになったため、読み取りが頻繁に発生しているイメージは、共有メモリーピックスマップ内に格納できます。この共有メモリーピックスマップは常にシステムメモリー内に存在します。Java2D は、イメージの読み取りまたはコピー頻度に基づいて、イメージを適切なメモリーに自動的に転送できます。使用するメモリーを制御するには、`J2D_PIXMAPS` 環境変数を設定します。この変数については、「Java2D テクノロジのシステムプロパティ」の「Linux と Solaris プラットフォームのシステムプロパティ」を参照してください。

新しいディスプレイピクセル形式のサポート

1.4.2 では、3 バイト RGB ピクセル形式のディスプレイがサポートされます。このディスプレイ形式は、Linux システムでよく使用されます。

12 ビット PseudoColor ビジュアルのサポートも、1.4.2 で実装されました。

パフォーマンスや品質に関わる改善点が多数追加され、医療業界のアプリケーションで一般的に使用されるグレースケールイメージ形式である、ByteGray と 12 ビット UShortGray がサポートされるようになりました。

1.3 Unicode 補助文字のサポート

「補助文字」とは、その 32 ビット数値 (「コードポイント」) が U+FFFF を超えている Unicode 文字のことです。したがって、Java プログラミング言語の char データ型など、単一の 16 ビットエンティティーとして補助文字を記述することはできません。そのような文字は一般にまれですが、たとえば、中国語や日本語の個人名の一部として使用されます。

Java プラットフォームへの補助文字サポートの導入は、大部分の文字処理アプリケーションが変更なしで実行できるようなアプローチを使って行われました。個々の文字を解釈するアプリケーションも、文字データに補助文字が含まれていない限り、変更なしで実行できます。個々の文字を解釈するアプリケーションの対象データに補助文字が含まれる場合には、`Character` クラスや各種 `CharSequence` サブクラス内の、コードポイントに基づく新しい API を使用できます。

多くの場合、補助文字をすでにサポートしている (一般に、より便利な) 5.0 API を使用すれば、プログラム内で文字変換処理を行う必要がなくなります。たとえば、次のようなコードを使用しているとします。

```
System.out.println("Character " + String.valueOf(char) + " is invalid.");
```

この場合、補助文字をサポートする次の出力書式 API を代わりに使用できます。

```
System.out.printf("Character %c is invalid.%n", codePoint);
```

この、より高レベルな API を使用すれば、コードがより単純になるだけでなく、メッセージのローカライズを困難にする連結を使わずに済み、リソースバンドル内に格納する必要のある文字列の数も、2 個から 1 個に減ります。

詳細 :

- 変更する必要のないアプリケーション - テキストの処理をあらゆる形式 (`char[]`、`java.lang.CharSequence` の実装、`java.text.CharacterIterator` の実装) の `char` シーケンス形式としてのみ行い、そうした文字シーケンスを引数や戻り値に持つ Java API のみを使用するアプリケーション。そのような場合には、Java プラットフォーム API の実装が、ユーザーに代わって補助文字を処理します。

- 変更する必要のないアプリケーション - 個々の文字を解釈したり、個々の文字を Java プラットフォーム API に渡したり、個々の文字を返すメソッドを呼び出したりするが、補助文字は処理しないアプリケーション。たとえば、HTML タグの文字シーケンスを走査し、その各文字を個別に検査するアプリケーションがあるとします。この場合、それらのタグで使用されているのは、「Basic Latin」ブロックに含まれる文字だけであることがわかっているので、UTF-16 に基づく `char` シーケンス内にたとえ補助文字が含まれていたとしても、そのアプリケーション内で補助文字は処理されません。
- 変更する必要のあるアプリケーション - 個々の文字を解釈したり、個々の文字を Java プラットフォーム API に渡したり、個々の文字を返すメソッドを呼び出したりし、それらの文字値に補助文字が含まれている可能性があるようなアプリケーション。
- さらに考慮すべき点 - UTF-8 との間の変換時に標準 UTF-8、Modified UTF-8 のどちらを使用する必要があるか。これにより、適切な Java プラットフォーム機能を使用できます。Modified UTF-8 は、Java プラットフォームの内部 API によって使用されます。標準 UTF-8 は、外部に公開されたすべての API で使用されます。詳細については、「Modified UTF-8」を参照してください。

文字処理アプリケーションを変換する場合 :

- 同等の API を利用できる場合 - 単純な `char` 値ではなく `char` シーケンスを使用するもの - 最良のアプローチは、これらの API を使用するようにアプリケーションを変換することです。
- 同等の API を利用できない場合 - コードポイントに基づく新しい API と `Character.toCodePoint(char high, char low)` API を併用することにより、2 個の UTF-16 コード単位を単一の 32 ビットコードポイントに変換します。`Character.toChars(int codePoint)` API はその逆を行います。つまり、単一のコードポイントを 1 個または 2 個の UTF-16 コード単位に変換し、それらを `char[]` 内に格納します。
- 最大限の単純さ - すべてのテキストをコードポイント表現 (`int[]` など) に変換し、その表現形式で処理を行います。そうすれば、文字変換の心配をする必要もなくなります。
- より高い利便性と最高のパフォーマンス - アプリケーション内で `char` シーケンスを引き続き使用し、必要になった場合にのみコードポイントに変換します。文字シーケンスを使用する Java プラットフォーム API は多数存在します。また、文字シーケンスを使用することは、メモリー領域の節約にもなります。

補助文字、コードポイント、Unicode 表現などのトピックに関するすぐれたチュートリアルについては、「Java プラットフォームにおける補助文字のサポート」を参照してください。追加情報を入手するには、「国際化の機能拡張」を参照してください。

1.4 ネットワーク

以下の変更は、ネットワークアプリケーションに影響します。

1.4.1 URL 接続の処理

1.4 より前のバージョンでは、ファイルタイプが既知で応答コードが 400 以上である場合に、`URLConnection.getInputStream` から `FileNotFoundException` がスローされていました。それ以外の場合は、例外はスローされませんでした。

1.4 では、正しい動作が実装されています。ファイルタイプにかかわらず、すべての `http` エラーに対して `URLConnection.getInputStream` から `IOException` がスローされます。`FileNotFoundException` (`IOException` のサブクラス) がスローされるのは、リソースが見つからなかったことを HTTP 応答が示した場合だけです。言い換えれば、`FileNotFoundException` がスローされるのは、応答コードが 404 または 410 の場合だけです。

その他の変更点を次に示します。

- `HttpURLConnection.getErrorStream` を使ってサーバーから返されたエラーページを読み取れるようになりました。1.4 より前のバージョンでは、getErrorStream() は常に null を返していました。

- `HttpURLConnection.getResponseCode` メソッドが正しく動作するようになりました。

1.4.2 URI の形式

バージョン 1.4.2 以降では、階層 URI (クラス `java.net.URI`) のホスト構成要素のホスト名が単一のドメインラベルだけから構成される場合に、そのホスト名の先頭文字として数字が使用できるようになりました。以前は、「`s://123/p`」などの URI の場合、その authority 構成要素はサーバーに基づく authority として解析されず、レジストリに基づく authority とみなされていました。この変更の一部として、`URI.getHost()` の仕様が改訂されました。更新版の仕様では、「2 つ以上のラベルから構成されるドメイン名の右端のラベルは、英字で始まること」とされています。

1.5 セキュリティー

この節では、セキュリティーに関する変更について説明します。

1.5.1 Java™ Secure Socket Extension (JSSE)

1.4 で、システムプロパティー `com.sun.net.ssl.dhKeyExchangeFix` のデフォルト値が `true` になりました。以前は、JSSE 1.0.2 オプションパッケージのデフォルト値は `false` でした。DH 鍵交換を使用する暗号化方式群は、このプロパティー値の変更の影響を受けます。ただし、そのような方式群が使用されることはほとんどありません。

この変更が施された理由は、元のオプションパッケージにバグがあったからです。そのバグが原因となって、サーバー鍵交換メッセージの一部として DSA 署名が使用される場合にその署名の符号化が正しく行われていませんでした。このバグは、JSSE が鍵交換仕様に準拠していないことを意味していました。、また、それが、JSSE とその他のベンダーによる SSL 実装との間の非互換性の原因でもありました。このシステムプロパティーはこうした状況に対応するために導入されたものであり、そのデフォルト値は、以前のリリースとの互換性を考慮して `false` に設定されました。1.4 では、ほかの SSL/TLS 実装との互換性を考慮して、そのデフォルト値が `true` に変更されました。

1.5.2 ポリシーファイルのエンコーディング用のシステムプロパティー

1.4.2 で、新しいシステムプロパティー `sun.security.policy.utf8` が導入されました。このシステムプロパティーを `true` に設定した場合、ポリシーファイルの読み取り時に UTF-8 が使用されます (1.4.0 と 1.4.1 の動作)。このシステムプロパティーを `false` に設定した場合、ポリシーファイルの読み取り時にデフォルトのエンコーディングが使用されます (1.4.0 より前のバージョンの動作)。このシステムプロパティーを設定しなかった場合のデフォルト値は、`true` です。

J2SE 1.4.0 より前のバージョンでは、セキュリティーポリシーファイルの文字エンコーディング方式は特に指定されておらず、このファイルの読み取り時にはデフォルトの文字エンコーディングが使用されていました。1.4.0 以降では、ポリシーファイルを UTF-8 でエンコーディングすることが要求されるようになりました。この要求により、1 つのポリシーファイルをさまざまなロケールで使用できるようになりましたが、既存のポリシーファイルのうち、デフォルトエンコーディングを使用した文字を含むものが破壊されます。

1.5.3 暗号化 Key オブジェクトの直列化

新しいインタフェース `java.security.KeyRep` が J2SE 5.0 に追加されました。このインタフェースは、暗号化 `Key` オブジェクトの標準直列化表現を示します。既存の直列化された `Key` オブジェクトは引き続き、同一ベンダーの仮想マシン内では直列化や直列化復元が可能ですが、異なるベンダーの仮想マシン間では直列化復元が不可能です。この動作は変更されていません。

KeyRep を直列化表現として使用する新しい Key クラス実装は、単一ベンダーの仮想マシン内ではもちろん、異なるベンダーの仮想マシン間でも直列化および直列化復元できます。

ただし、KeyRep を直列化表現として使用した Key クラスは、5.0 より前のどの Java バージョンでも直列化復元ができません。5.0 で、Sun の暗号化プロバイダ内に実装された Key クラスはすべて変更されました。したがって、それらの直列化表現としては、KeyRep を使用してください。

1.5.4 KerberosKey.serialVersionUID

5.0 で、javax.security.auth.kerberos.KerberosKey クラス内に独自の private serialVersionUID フィールドが定義されました。KerberosKey が実装している java.security.Key インタフェースから以前に継承されていた serialVersionUID フィールドが、この新しいフィールドによって隠蔽されることになります。

この変更によって発生する実行時の問題はありません。以前にコンパイルされた KerberosKey.serialVersionUID を参照するコードは、正しく動作します。ただし、この変更により、ソースの非互換性問題が発生します。というのも、KerberosKey.serialVersionUID を参照するアプリケーションコードは、5.0 ではコンパイルエラーとなるからです。

1.6 直列化

この節では、直列化の非互換性について説明します。

1.6.1 シリアルバージョン UID の変更

入れ子クラスとそれが包含するクラスの両方に対するデフォルトのシリアルバージョン UID の 1.3 における計算値が、5.0 で変更されました。5.0 では、クラスリテラルへの参照に対してバイトコード命令が生成されます。以前は、static メソッドへの参照が生成されていました。この変更の結果、Foo.class などのクラスリテラルを参照する任意の直列化可能クラスは、5.0 VM での実行時には動作しなくなります。

異なるバージョンのコンパイラ (または異なるベンダーによるコンパイラ) における変更にコードが影響されないようにするには、明示的なシリアルバージョン UID を直列化可能クラスに追加します。1.3 の javac コンパイラでコンパイルされたクラスのシリアルバージョン UID を取得するには、serialver ツールを使用します。

詳細については、バグ 4786115 を参照してください。

1.6.2 ストリーム入出力サブクラスで必要となった直列化可能アクセス権

1. ObjectOutputStream のサブクラスを作成しており、そのサブクラス内で putFields() をオーバーライドし、ObjectOutputStream の単一引数 public コンストラクタを呼び出している場合、現在のコンテキスト内で SerializablePermission の enableSubclassImplementation がアクティブになっているかどうかを確認する必要があります。セキュリティー対策として、ObjectOutputStream は、SecurityManager を使ってこのアクセス権の存在有無をチェックします。
2. ObjectOutputStream のサブクラスを作成しており、そのサブクラス内で readFields() をオーバーライドし、ObjectOutputStream の単一引数 public コンストラクタを呼び出している場合、現在のコンテキスト内で SerializablePermission の enableSubclassImplementation がアクティブになっているかどうかを確認する必要があります。セキュリティー対策として、ObjectOutputStream は、SecurityManager を使ってこのアクセス権の存在有無をチェックします。

1.6.3 メソッドの継承

1.3 では、javac バイトコードコンパイラはデフォルトで `-target 1.1` を使用していました。5.0 では、`-target 5.0` がデフォルトです。この変更の 1 つの結果として、クラスがインタフェースから未実装メソッドを継承している場合は、コンパイラが、メソッド宣言の生成とクラスファイル内への挿入を行わなくなりました。これらの挿入されたメソッドは、ほかのすべての非 private メソッドと同じく、デフォルトの `serialVersionUID` を計算する際の処理対象となります。したがって、あるインタフェースを直接実装しているが、そのメソッドのいくつかを実装していないような直列化可能抽象クラスを定義した場合、そのデフォルトの serialVersionUID 値は、1.4 バージョンの javac でコンパイルした場合とそれ以前の javac でコンパイルした場合とで異なります。

`target` オプションの詳細については、javac コンパイラのリファレンスページを参照してください。以前のバージョンの javac によって挿入されるメソッドに関する背景情報については、バグ 4043008 を参照してください。

1.7 Swing

この節では、Swing の GUI コンポーネントに影響する互換性の問題について説明します。

1.7.1 ボタンの色

カスタマイズされた背景色を持つボタンの場合、コードを変更しないと、5.0 の Java Look & Feel テーマ Ocean で意図したとおりにレンダリングされない可能性があります。Ocean はデフォルトで、ボタン上にグラデーションを描画します。グラデーションが不要な場合には、`contentAreaFilled` プロパティーを `true` に設定するか、背景色を `UIResource` ではない `Color` に設定します。ほとんどの場合、この作業は次のように単純です。

```
button.setBackground(Color.RED);
```

何らかの理由により、`UIResource` を使用する場合には、`UIResource` でない `Color` を次のようにして新たに作成できます。

```
button.setBackground(new Color(oldColor));
```

詳細については、バグ 4908404 を参照してください。

1.7.2 DefaultTreeModel

バージョン 1.4 以降では、javax.swing.tree.DefaultTreeModel で null ルートノードが使用できます。以前のバージョンでは、DefaultTreeModel で null ルートを使用できませんでした。

1.7.3 DefaultHighlighter.DefaultPainter

1.4 で、クラス javax.swing.text.DefaultHighlighter の public static フィールド DefaultPainter が、final になりました。これは、以前は final ではありませんでした。

1.7.4 ドラッグ & ドロップ

1.4 の Swing は、ドラッグ & ドロップ (DnD) をサポートします。Swing コンポーネント上で AWT のドラッグ & ドロップサポート (特に `DropTarget`) を使用しているアプリケーションでは、Swing の `DropTarget` との衝突が発生する可能性があります。詳細については、バグ 4485914 を参照してください。

新しいアプリケーションでは常に、カスタマイズされた解決策の代わりに、Swing の組み込み DnD サポートを使用するべきです。なぜなら、Swing の組み込みサポートは、ドラッグジェスチャーの開始やドロップ位置の表示といった詳細を処理するからです。

Swing のドラッグ＆ドロップサポートの詳細については、次を参照してください。

http://java.sun.com/j2se/1.5.0/ja/docs/ja/guide/swing/1.4/dnd.html

完全なチュートリアルについては、次を参照してください。

http://java.sun.com/docs/books/tutorial/uiswing/misc/dnd.html

1.7.5 フォーカスに関する変更

軽量 Swing コンポーネントライブラリが変更され、AWT との整合性と相互運用性が高まりました。1.4 では、すべてのコンポーネントのフォーカストラバース可否が、デフォルトで `true` になっています。以前は、軽量 Swing コンポーネントのフォーカストラバース可否が、デフォルトで `false` になっていました。

1.7.6 JTable のインデックス操作

`JTree` と `JList` では、キーボードアクション発生時には常にリードインデックスが操作されます。たとえば、ある `JList` でリードが行 4 に存在している状態でユーザーが上矢印キーを押すと、そのリードは行 3 に移動し、3 番目の項目が選択されます。これらのコンポーネントでは、リードがフォーカスインデックスとみなされます。これらのコンポーネントは、指定されたインデックスがリードである場合にそのインデックスに対してフォーカスインジケータを描画するように、レンダリング機能に指示します。

5.0 で、`JTable` の動作が、`JList` および `JTree` と整合性のとれたものになりました。以前の `JTable` は、それらとは正反対の動作を行なっており、`JTree` および `JList` がリードを使用するのと同じ方法でアンカーインデックスを使用していました。この不整合に対する修正要求が RFE 番号 4759422 として発行され、最終的に 4303294 の一部としてその修正が実施されました。

この変更は、以前の動作を前提にしていた開発者に影響します。したがって、`JTable` のアンカーを使ってどのセルがフォーカスセルとして表示されるかを判断しているアプリケーションは、正しく動作しない可能性があります。詳細については、バグ 4759422 および 4303294 を参照してください。

1.7.7 XP および GTK の Look & Feel のサポート

1.4.2 で、XP および GTK の Look & Feel のサポートが、Swing に追加されました。`WindowsLookAndFeel` を、クラス名経由で直接的に、または `UIManager.getSystemLookAndFeelClassName()` 経由で間接的に使用している場合、Windows XP 上での実行時に、XP の Look & Feel が自動的に取得されます。Gnome が動作するマシン上では、GTK の Look & Feel が取得されます。

XP と GTK の両方の Look & Feel をサポートしたことで、Swing がこれまで一般的にウィジェットの描画時に使用してきた方法との潜在的な非互換性問題が、さまざまな場面で発生する可能性が出てきました。

- どちらの Look & Feel も、通常はイメージを使用してウィジェットのレンダリングを行います。このため、ボーダーを頻繁に交換しても、その表示が変化しなくなりました。以前は、ボーダーを交換しても、ボーダーが描画されませんでした。
- `JButton` では、背景イメージの描画が、`contentAreaFilled` プロパティーに基づいて条件付きで実行されるようになりました。このため、ある `JButton` 上で `setContentAreaFilled(false)` を呼び出した場合、イメージを持たない平面的なボタンが表示されます。
- 背景はイメージを使って描画されるため、背景色を変更しても表示は変化しません。この動作も、`contentAreaFilled` プロパティーを使えば変更できます。

- Swing では、不透明なコンポーネントは常に、ComponentUI の update メソッド経由で自身の背景を描画します。ところが、GTK の透明なコンポーネントの場合、その背景が描画されるかどうかは、使用されるエンジンによって異なります。GTK の Look & Feel を正しく提供できるように、以前は不透明であったコンポーネントの多くが、ComponentUI のサブクラスの installUI メソッド経由で透明に変更されました。あるコンポーネントで目的の値を設定しなくてもある不透明度で描画されることを期待する場合、問題が発生する可能性があります。また、setOpaque(false) が「背景を塗りつぶすべきでないこと」を意味すると考えた場合にも、問題が発生する可能性があります。

1.8 XML 処理

JAXP 1.1 (Crimson) は 1.3 プラットフォームの一部でした。JAXP 1.3 (Xerces) は 5.0 プラットフォームの一部です。これら 2 つは、共通の API である JAXP によってアクセス可能な、まったく異なる実装です。

Crimson はサイズが小さく高速でしたが、その機能は最終的に、Apache でホストされるオープンソース実装である Xerces に及びませんでした。さらに、JAXP 標準も 1.1 から 1.3 へと進化しました。これらの 2 つの要因が組み合わされて、互換性の問題が発生します。

JAXP 1.1 は 1.4 プラットフォームの一部でもありました。したがって、「JAXP 互換性ガイド (J2SE 5 Platform 用)」では、1.4 から 5.0 への移行方法について説明していますが、1.3 の XML 処理アプリケーションを 5.0 に変換する際の問題についても説明しています。以下で説明する変更の詳細については、このガイドを参照してください。

1.8.1 DOM

5.0 の JAXP は、DOM Level 3 ファミリの API をサポートします。

- DOM インタフェースに新しいメソッドが追加されました。このため、既存のアプリケーションによっては、その新しいインタフェースを使ってコンパイルできない可能性があります。
- また、アプリケーションによっては、実行時に NoSuchMethodException が発生する可能性もあります。この問題を解決する唯一の方法は、5.0 ライブラリを使ってソースをコンパイルし直すことです。
- 空白の処理方法が 2 つのライブラリ間で異なります。このため、可読性が高く、「出力品質の高い」XML を出力することが求められるアプリケーションについては、この相違点に応じた変更を行う必要があります。

1.8.2 SAX

5.0 の JAXP は、SAX 2.0.2 をサポートします。一般に、SAX 2.0.2 はバグ修正リリースであるため、API は変更されていません。ただし、SAX 2.0.2 リリースに実装されているいくつかの明確化によって、次のような互換性の問題が発生する可能性があります。

- アプリケーションで、ErrorHandler、EntityResolver、ContentHandler、および DTDHandler を null に設定できるようになりました。SAX 2.0 は、そうした場合に java.lang.NullPointerException をスローすることを XML プロセッサに要求していました。5.0 に実装されている JAXP パーサーは、大部分の実装と同様に、デフォルト設定を使用して null に対処します。
- DefaultHandler の resolveEntity メソッドと EntityResolver サブクラスから、IOException と SAXException がスローされます。以前は、SAXException のみがスローされていました。resolveEntity を呼び出すコードを変更し、SAXException に加えて IOException も処理できるようにする必要があります。

1.8.3 XSLTC

5.0 では、XSLTC がデフォルトのトランスフォーマになりました。これは、Apache コミュニティーが、XSLTC を XSLT 2.0 開発用のデフォルトプロセッサにすることに決定したからです。以前のバージョンでは、デフォルトのトランスフォーマは Xalan でした。次のような互換性の問題が存在します。

- Xalan には XSLTC にはないバグが含まれており、当然その逆も成り立ちます。Xalan のバグ動作に依存しているアプリケーションコードの実行は、失敗する可能性が高くなります。
- Xalan がサポートする拡張の中には、XSLTC がサポートしないものもあります。それらの拡張は、JAXP 仕様や XSLT 仕様に定義されていないものです。この問題を回避するには、Apache から Xalan のクラスをダウンロードし、それらを使用してください。ただし、今後は、XSLTC でより多くの拡張がサポートされる予定です。
- スタンドアロンの XPath 式 (XSLT スタイルシートの一部になっていない XPath 式) を評価する際に Xalan の XPath API を明示的に使用しているアプリケーションでは、JAXP 1.3 に含まれる標準の XPath API を使ってコーディングし直す必要があります。別の方法として、Apache からダウンロードした Xalan のライブラリを、そのアプリケーションに含めることもできます。

1.8.4 セキュリティーの拡張

セキュリティー問題を解決する目的で、新しいシステムプロパティーやパーサープロパティーが追加されました。

- 新しい「セキュリティー保護された処理」機能を使用すれば、アプリケーションで、SAXParserFactory または `DocumentBuilderFactory` がセキュリティー保護された XML プロセッサを取得するように構成できます。この機能を有効にすると、エンティティーの拡張上限が 64000 に設定されるため、サービス拒否攻撃を防ぐことができます。
- 別の方法として、`entityExpansionLimit` を使用することによっても、エンティティー拡張の合計数を制約できます。
- `disallow-doctype-decl` パーサープロパティーは、受信 XML ドキュメントに DOCTYPE 宣言が含まれることを禁止します。

1.8.5 パッケージ名の変更

5.0 では、org.apache のクラスが `com.sun.org.apache.package.internal` に移されましたが、これは、それらのクラスが開発者がダウンロードした最近のバージョンのクラスと衝突しないようにするためです。

この変更は、標準の JAXP API のみを使用するよう制約されているアプリケーションには影響しません。次のように実装固有の機能に依存しているアプリケーションには影響します。

- プロパティー値を使用して内部実装にアクセスしていた場合、それらの値を変更する必要があります。
- Xalan 実装クラスの内部 API を使用していたアプリケーションでは、その import 文を変更してそれらの API にアクセスできるようにする必要があります。
- その他の Crimson ライブラリの内部 API を使用していたアプリケーションでは、次のいずれかの方法で新しいパッケージ名を取り入れる必要があります。
  a. JAXP の一部となっているサポート対象インタフェースのみを使用するようにアプリケーションをコーディングし直します。
  b. org.apache のクラスを Apache からダウンロードし、それらをクラスパスに含めます。

1.9 その他の実行時に関する変更

次に示す変更はエンドユーザーアプリケーションの開発者に関係するものです。

1.9.1 CORBA

1.4 の CORBA API に加えられた変更は、「CORBA 互換性情報」で参照されている OMG ドキュメントで規定された CORBA 2.3 マッピングに準拠するためのものです。API の変更に関する情報のリンクや J2SE 1.4.0 が準拠するすべての OMG 仕様の一覧を参照してください。

1.9.2 非 ANSI ファイル用のデフォルトエンコーディング (Windows)

J2SE 1.4.2 では、Microsoft Windows の `file.encoding` システムプロパティーは、システムのデフォルトロケールから派生します。このため、非 ANSI コードページロケールのデフォルトとして `utf-16le` エンコーディングを仮定するアプリケーションは失敗します。そのようなアプリケーションでは、代わりに `file.encoding` システムプロパティーを使用するべきです。

この変更は、バグ 4459099 を修正するためのものです。以前は、「Hindi」の場合のように、ユーザーのロケール設定に対応する ANSI コードページがコントロールパネル内に存在しない場合には、`file.encoding` システムプロパティーが `utf-16le` に設定されていました。その場合、すべてのリーダーとライターがデフォルトでそのエンコーディングを使用することになり、その結果、システム上のファイルが `utf-16le` コンバータ経由で読み書きされる際に例外が生成されていました。

1.9.3 HTML のフォーム

1.4 で、HTML のフォームのモデルが、1.3 のモデルから変更されました。以前は、フォームの属性が、その子にあたる文字要素の `attributeset` 内に格納されていました。1.4 では、フォームを表現するための要素が作成されます。この新しい要素は、html ファイルの内容に一致します。このため、フォームのより効果的なモデリングと一貫性のあるフォームの記述が可能となります。この変更は、バグ 4200439 に対処するためのものです。

この変更の影響を受けるのは、フォームが厳密でない方法で処理されることに依存していた開発者だけです。これまでフォームの属性がリーフ要素の属性に含まれていることを期待してきた開発者は、それらの属性をフォーム要素の `AttributeSet` から取得する必要があります。

たとえば、次のような無効な HTML があるとします。

```
<table>
    <form>
</table>
</form>
```

これは、1.4.0 より前の実装では次のように処理されていました。

```
<form>
    <table>
    </table>
</form>
```

ところが、1.4 では代わりに次のように処理されます。

```
<table>
```

```
        <form>
        </form>
    </table>
```

1.9.4 java.vm.info プロパティー (値の追加)

5.0 で導入されたクラス共有機能を反映するために、`java.vm.info` プロパティーに共有モードが指定されるようになりました。このプロパティーの内容は、`java -version` で表示されるテキストに反映されます。`java.vm.info` プロパティー値の末尾までのすべてを解析するコードや、`java -version` の出力を解析するコードはすべて、変更しなければならない可能性があります。詳細については、バグ 4964160 と「クラスデータの共有」を参照してください。

1.9.5 Java 入出力関連の変更

以下の変更は、putFields メソッドまたは readFields メソッドをオーバーライドする ObjectInputStream と ObjectOutputStream のサブクラスに影響します。ただし、直列化インフラストラクチャーの残りの部分もオーバーライドするサブクラスには影響しません。

1. J2SE 1.4.0 以降では、`ObjectOutputStream.putFields` または `ObjectOutputStream.writeUnshared` をオーバーライドするサブクラスが ObjectOutputStream の単一引数 public コンストラクタを直接的または間接的に呼び出す際に、enableSubclassImplementation SerializablePermission が必要となります。
2. 同様に、J2SE 1.4.0 以降では、ObjectInputStream.readFields または ObjectInputStream.readUnshared をオーバーライドするサブクラスが ObjectInputStream の単一引数 public コンストラクタを直接的または間接的に呼び出す際に、enableSubclassImplementation SerializablePermission が必要となります。

1.9.6 Java™ DataBase Connectivity (JDBC™) / BigDecimal API の変更

BigDecimal のメソッドの動作が 1.4 と 5.0 とで異なるため、JDBC ドライバが正常に動作しなくなりました。この問題を解決するには、5.0 版の JDBC ドライバを使用してください。

1.9.7 JDBC の Timestamp と Date の比較

5.0 では、`java.sql.Timestamp` と `java.util.Date` を比較するために `Timestamp` 上で `compareTo` を呼び出すと、`ClassCastException` が発生します。たとえば、次のコードは、1.4.2 では `Timestamp` と `Date` を正常に比較しますが、5.0 では例外が発生して失敗します。

```
aTimeStamp.compareTo(aDate) // 動作しません
```

この変更はコンパイル済みのコードにも影響し、バイナリ互換の問題が発生します。具体的には、以前のリリースの下で動作していたコンパイル済みのコードが、5.0 では失敗します。この問題は今後のリリースで修正される予定です。

詳細については、バグ 5103041 を参照してください。

1.9.8 ログ作成

以前は、`java.util.logging.Level(String name, int value, String resourceBundleName)` コンストラクタでは null の名前引数を使用できましたが、解析用のメソッドでは使用できませんでした。5.0 では、名前が null の場合にこのコンストラクタから `NullPointerException` がスローされるようになりました。これにより、互換性のリスクが軽減されます。というのも、以前は、Level のサブクラスを作成してこのコンストラクタを使用したあと、後続の呼び出しで Level の名前として null を使用した時点で `NullPointerException` が発生していたからです。ただし、`toString()` などの単純な呼び出しは除きます。

詳細については、バグ 4625722 を参照してください。

第 2 章
配備時の問題

この章では、配備時の問題を概説します。詳細については、『Java 配備ガイド』を参照してください。

2.1 アプレット

2.1.1 Java コントロールパネル

5.0 で新しく導入された Java コントロールパネルは、Java Plug-in コントロールパネルと Java Web Start アプリケーションマネージャーとを統合したものであり、単一の構成インタフェースを提供します。互換性の問題に関係するのは、次の節で説明するアプレットキャッシングです。Java コントロールパネルの詳細については、http://java.sun.com/j2se/1.5.0/ja/docs/ja/guide/deployment/deployment-guide/jcp.html を参照してください。

2.1.2 アプレットキャッシングに関する変更

5.0 では、アプレットに独立したキャッシュが用意され、そのキャッシュが複数のブラウザ間で共有されるようになりました。以前は、各ブラウザ内でアプレットがキャッシュされていたため、特定のマシン上で使用されているブラウザごとに (またはブラウザバージョンごとに) キャッシュされたアプレットのコピーが存在する可能性が高くなっていました。

この変更の結果、アプレットキャッシングの大部分の側面が、ブラウザ経由ではなく Java コントロールパネル経由で管理されるようになります。それらの側面を次に列挙します。

- キャッシュの場所
- キャッシュサイズの制限
- キャッシュの圧縮
- キャッシュ管理ツール
- キャッシュ削除ポリシー

このため、5.0 では、これらの値に対するブラウザ設定がアプレットキャッシングに影響しなくなります。

2.1.3 署名付きアプレットの証明書の検証

5.0 では、署名検証時に使用されるルート認証局 (CA) 証明書は、次の場所から取得されます。

- Java™ Runtime Environment (JRE) の cacerts ファイル (常に有効)
- ブラウザのキーストア (デフォルトで有効になっているが、Java コントロールパネルで無効にできる)

以前は、署名検証時にブラウザのルート CA 証明書が使用されていました。

2.1.4 タイムスタンプを含むアプレット署名

5.0 では、配備済みの Java™ Archive (JAR) ファイルを毎年署名し直す必要がなくなりました。代わりに、jarsigner を使用してタイムスタンプを含む署名を生成できるようになりました。その場合、システムや配備プログラム (Java Plug-in など) は、タイムスタンプ情報を取得するための新しい API を使用することにより、JAR ファイルが署名された時点でその署名に使われた証明書が有効であったかどうかを確認することができます。

以前は、jarsigner で生成される署名にはタイムスタンプが含まれていませんでした。その他の情報もまったく利用できなかったため、システムや配備プログラムは一般に、署名に使われた証明書の有効性をチェックすることによって、署名付き JAR ファイルの有効性を確認していました。ところが、そのようなチェックは、署名に使われた証明書の有効期限が切れると失敗していました。そうした期限切れは通常、毎年発生していました。

詳細については、http://java.sun.com/j2se/1.5.0/ja/docs/ja/guide/security/time-of-signing-beta1.html を参照してください。

2.2 ライブラリ

次のパッケージは、Java 5.0 リリースの一部になったため、オプションパッケージとして配布する必要がなくなりました。

- Java Secure Socket Extension (JSSE)
- Java™ Authentication and Authorization Service (JAAS)
- Java™ Cryptography Extension (JCE)
- Java Web Start (JWS)
- Java™ Management Extensions (JMX™)

注 : ターゲット JAR ファイルと現在インストールされている拡張 JAR ファイル間のバージョンの衝突を検出するには、このリリースに含まれる extcheck ユーティリティーを使用してください。

2.3 インストール

2.3.1 Windows オンラインインストーラ

5.0 では、新しい「オンラインインストールオプション」が利用可能になりました。これは高速な接続に適しています。低速な接続においても、このインストーラの起動はすばやく、ダウンロードの合計バイト数も少なくて済むのですが、インストーラがデータのダウンロードをいったん開始すると、1 つの CAB ファイルのダウンロードが完了するまでは進捗インジケータが更新されません。指定されたダウンロードの所要時間に関する情報が何も表示されません。したがって、低速な接続では、基本的にオフラインのインストールオプションを選択することをお勧めします。ダウンロードの合計サイズは大きくなりますが、ブラウザの進捗バーに完了予想時間が表示されます。

2.3.2 名前の変更

5.0 で、ディレクトリ、バンドル、パッケージ、レジストリ、および Linux RPM の名前が変更されました。このため、古いパス名に依存しているスクリプトについては、新しい名前が反映されるように更新する必要があります。新しい命名規則は次のとおりです。

古い名前	新しい名前
j2se	java
j2re	jre
j2sdk	jdk

大文字と小文字の区別は、各プラットフォームの規則に従います。以下では、プラットフォーム固有の情報を示します。詳細については、「J2SE Naming and Versioning」と「J2SE 5.0 Name and Version Change」を参照してください。

Solaris

Java™ Development Kit (JDK™) のパッケージは次の場所にインストールされます。

```
/usr/jdk/jdk<version>
```

すべてのパッケージに使用される接頭辞が、1.3 と 1.4 で使用されていた「SUNWj3」から「SUNWj5」に変更されました。

Linux

JRE と JDK の RPM は次の場所にインストールされます。

```
/usr/java/jre<version>
/usr/java/jdk<version>
```

RPM データベース名とは、RPM クエリーの実行時に「Name」フィールドに表示される値のことです。この値は、「`jre-1.5.0-fcs`」のような完全修飾名を取得するために「Version and Release」フィールドに付加されます。RPM データベースに対してクエリーを実行すると、特定のパッケージが提供する内容を確認できます。JRE と JDK はどちらも、新しい命名規則の一部として「jre」を提供するほか、下位互換性を確保するために「j2re」も提供します。JDK はさらに、「jdk」と下位互換性用の「j2sdk」も提供します。古い名前は標準 EOL ポリシーに従って提供されますが、新しいスクリプトや RPM では新しい規則を使用するべきです。

UNIX

tarball は次の場所に展開されます。

```
./jre<version>
./jdk<version>
```

Microsoft Windows

JRE と JDK は次の場所にインストールされます。

```
%ProgramFiles%\Java\jre<version>
%ProgramFiles%\Java\jdk<version>
```

レジストリキーは変更されていません。レジストリキーでは引き続き、完全な名前である「Java Runtime Environment」と「Java Development Kit」が使用されます。

2.4 仮想マシン (Solaris)

5.0 では、サーバークラスの Solaris/SPARC マシン上ではデフォルトで、クライアント VM ではなくサーバー VM が実行されるようになりました。一般に、サーバー VM を使用すると、スループットは格段に高くなりますが、起動時間が若干長くなります。以前は、Solaris/SPARC のデフォルトの仮想マシン (VM) はクライアント VM でした。しかし、Solaris/SPARC ボックスの多くはサーバーとして使用されており、サーバー上ではサーバー VM のほうが高いパフォーマンスを示します。

注 : 「サーバークラスマシン」の現在の定義は、「2 個以上のプロセッサと 2G バイト以上のメモリーを備えたマシン」となっています。詳細については、「サーバークラスマシンの検出」と「ガベージコレクタのエルゴノミクス」を参照してください。

第 3 章
コンパイル時の問題

ここでは、1.3 コードを 5.0 上で実行するようにコンパイルする際に発生する問題について説明します。

3.1 API の変更

3.1.1 JDBC

1.4 プラットフォームの一部である JDBC 3.0 API では、2 つのインタフェースが新たに導入されたほか、既存のインタフェースにいくつかの新しいメソッドが追加されました。1.3 の下でコンパイルされたドライバとアプリケーションは問題なく動作しますが、それらのソースのコンパイルは、これらの変更のために失敗します。したがって、JDBC インタフェースを実装するドライバとアプリケーションは、正常にコンパイルするためには、更新してこれらの変更を反映する必要があります。要件の完全な一覧については、「JDBC 3.0 Specification」の第 6 章を参照してください。

3.1.2 新しいプロキシクラス

5.0 では、`java.net.Proxy` クラスが追加されたため、`Proxy` という名前を持つクラスが次の 2 つになりました。

- java.lang.reflect.Proxy
- java.net.Proxy

この新しいクラスが導入されたことで、次の条件が満たされた場合に既存コードのコンパイルが失敗するようになります。

- 次の import 宣言が存在する

```
import java.lang.reflect.*;
import java.net.*;
```

- いずれかの Proxy クラスを明確にインポートする import 宣言が存在しない
- コード内で `Proxy` クラスを参照する際に、`java.lang.reflect.Proxy` のような完全修飾名ではなく、単純名が使用されている

この場合、コンパイル時エラーが発生します。なぜなら、参照があいまいであるからです。

このあいまいな参照を解決して java.lang.reflect.Proxy が選択されるようにするには、次の 3 つ目の import 文を追加します。

```
import java.lang.reflect.Proxy;
```

この 3 つ目の import 文があれば、ソースコードのコンパイルが正常終了し、以前のバージョンの場合と同じ動作が実現されます。

3.1.3 ソケット API / SocketImpl のサブクラス

1.4 で、ソケット API の SocketImpl abstract クラスに新しい abstract メソッドが追加されました。新しいメソッドが追加されたため、1.4 より前に作成された SocketImpl のサブクラスはコンパイルに失敗します。なぜなら、新しいメソッドに対する実装が存在しないからです。ただし、バイナリは期待したとおりに動作します。

SocketImpl をサブクラス化しているアプリケーションはごくまれです。しかし、そのようなアプリケーションが実際に存在した場合、この変更に対処するには次の 2 つの方法があります。

- J2SE 1.3.x (またはそれ以前) でコンパイルされたクラスファイルを使用します。
- 新しいメソッドの実装を提供します。次のコードはその単純な実装例を示したものです。connect() の実装は、単純な TCP/IP 実装を前提にしており、タイムアウト値を無視しています。使用しているアプリケーションに適したタイムアウト値を反映するように変更してください。sendUrgentData() の実装では、単に例外をスローしているだけです。supportsUrgentData() をオーバーライドして true を返すようにしていない限り、必要な処理はこれだけです。

```
  /**
   * Creates a socket and connects it to the specified address on
   * the specified port.
   * @param address the address
   * @param timeout the timeout value in milliseconds,
   *        or zero for no timeout.
   * @throws IOException if connection fails
   * @throws IllegalArgumentException if address is null or is a
   *         SocketAddress subclass not supported by this socket
   * @since 1.4
   */
  protected void connect(SocketAddress address, int timeout)
  throws IOException {
     if (address == null
     || !(address instanceof InetSocketAddress))
        throw new IllegalArgumentException(
  "unsupported address type");
     InetSocketAddress addr = (InetSocketAddress) address;
     if (addr.isUnresolved())
        throw new UnknownHostException(addr.getHostName());
     this.port = addr.getPort();
     this.address = addr.getAddress();
     try {
        connect(this.address, port);
        return;
     } catch (IOException e) {
        // すべての処理が失敗した
        close();
        throw e;
     }
  }
  /**
   * Send one byte of urgent data on the socket.
   * The byte to be sent is the low eight bits of the parameter
   * @param data The byte of data to send
```

```
     * @exception IOException if there is an error sending the data.
     * @since 1.4
     */
    protected void sendUrgentData (int data) throws IOException {
        throw new IOException("Unsupported operation");
    }
```

3.2 総称

5.0 では、コレクションクラス、Class クラス、およびその他のコアライブラリを総称化するために、総称型パラメータや総称型引数が既存のクラスとメソッドに追加されました。総称を活用すれば、より簡潔で可読性の高いコードを作成でき、値をキャストしてコンパイラに型を宣言する必要をなくすことで、実行時エラーの原因の 1 つを除去できます。

既存のソースコードのほとんどは、5.0 の総称化されたライブラリを使って正しくコンパイルできますが、コンパイルできないコードもあります。総称化の変更が原因でコンパイルに失敗するようになったコードをコンパイルするための (または単純に警告メッセージが表示されないようにするための) もっとも単純な回避策は、javac コマンド行で -source 1.4 を指定することです。

注 : 最近の IDE の多くは、総称化されたライブラリを十分に活用できるよう、既存のコードを自動変換する機能を備えています。この変換処理は一般に、リファクタリング機能の一部として提供されます。総称やコアライブラリの総称化については、JSR 14 および総称に関するチュートリアル (PDF) を参照してください。

3.3 新しい予約語

1.3 から 5.0 の間で、Java 言語に次の単語が追加されました。したがって、今後は、これらの単語をフィールドやメソッドの識別子として使用することはできません。

- assert (1.4 で追加)
- enum

クラスファイルはこの変更の影響を受けません。しかし、これらの新しいキーワードは予約語であるため、これらを識別子として使用している既存のプログラムは、次の互換性スイッチを使用しない限り、コンパイルに失敗します。

- -source 1.3: すべてのキーワードを無効にし、それらを識別子として使えるようにします。
- -source 1.4: assert キーワードを有効にします。プログラムで表明を使用することはできますが、「assert」を識別子として使用することはできません。その他のキーワードは無効化されるので、識別子として使用できます。
- -source 1.5 (デフォルト): すべてのキーワードを有効にし、それらを識別子として使用できなくします。

注 : 1.3 のソース互換性は、今後段階的にサポートされなくなる予定です。-source 1.5 以外のオプションが指定された場合、5.0 のすべての言語機能が無効になります。それらの機能の詳細については、「Java プログラミング言語」を参照してください。このページは、チュートリアルや白書の索引です。可能な限り簡潔で読みやすく、保守もしやすいコードを作成したい方は、このページを参照することを強くお勧めします。

3.4 コンパイラの変更

3.4.1 デフォルトターゲットの変更

1.3 では、javac バイトコードコンパイラはデフォルトで `-target 1.1` を使用していました。5.0 では、`-target 5.0` がデフォルトになります。この変更の影響を受けるのは、バージョン 1.1 の Java プラットフォーム上で実行するように設計されたアプリケーションです。`target` オプションの詳細については、javac コンパイラのリファレンスページを参照してください。

3.4.2 言語仕様へのより厳密な準拠

1.4 で、J2SE 1.4.0 に含まれる Javac バイトコードコンパイラが、「Java 言語仕様」により厳密に準拠するようになりました。このため、「Java 言語仕様」に厳密に準拠していない既存のコードは、以前のバージョンではコンパイルできていたとしても、今後はコンパイルできない可能性があります。

コンパイラが以前よりも厳密に動作する場合の事例を、以下に 2 つ示します。

- コンパイラは、言語仕様の必要に応じて到達不能な空文を検出するようになりました。コンパイラが拒否するようになった非常に一般的な例を、次に 2 つ示します。

  ```
  return 0;/* 正常終了 */;
  ```

  および

  ```
  {
    return f();
  } catch (Whatever e) {
    throw new Whatever2();
  };
  ```

  どちらの場合も、コンパイラは、余分なセミコロンを到達不能な空文として正しく処理するようになりました。

  注 : 自動生成されたソースコードには、到達不能な空文が含まれている場合があります。

- コンパイラは、無名の名前空間から特定の型をインポートする import 文を拒否するようになりました。以前のバージョンのコンパイラは、言語で許可されていない可能性が高いにもかかわらず、そのような import 宣言を受け入れていました。なぜなら、import 節に含まれる型名は、コンパイラの処理対象外であるからです。言語仕様が明確化された結果、今では「import 文に単純な名前を含めたり、無名の名前空間からインポートしたりすることはできない」と明確に規定されています。

  このため、次の構文は合法ではなくなりました。

  ```
  import SimpleName;
  ```

  無名の名前空間から入れ子クラスをインポートする次の構文も、合法ではなくなりました。

  ```
  import ClassInUnnamedNamespace.Nested;
  ```

  使用しているコード内のこうした問題を解決するには、コード内のすべてのクラスを無名の名前空間から名前付きの名前空間へ移動してください。

第 4 章

ツール開発者とプラットフォーム実装者に影響する変更

以下の変更は一般に、アプリケーション開発者ではなく、ツール開発者と Java プラットフォーム実装者に影響します。

4.1 アプレットのデータストリーム / コンテナの実装

1.4 では、アプレットコンテナクラス (Java Plug-in や `appletviewer` に含まれるクラスなど、`AppletContext` インタフェースを実装するクラス) を変更し、改訂された `AppletContext` API を実装する必要があります。

`AppletContext` の改訂された仕様を利用すれば、アプレット開発者は、ブラウザセッション中にデータやオブジェクトを永続的に使用する目的でストリームできるようになります。この変更により、開発者は static クラスを使ってデータやオブジェクトをキャッシュする必要がなくなりますが、その一方で、アプレットコンテナでバイナリ非互換の問題が発生します。

4.2 クラスファイル / 内部クラス / 計測されるコード

5.0 でクラスファイルの形式が変更されました。このため、クラスファイルを計測してパフォーマンス測定やデバッグを行うプログラムから、無効なクラスが生成されるようになります。

内部クラスに対して生成される名前も変更されました。この変更は、内部クラスをその命名パターンで識別する計測プログラムやその他のプログラムに影響します。

4.3 クラスリテラル評価後のクラスの初期化

5.0 では、`Foo.class` などのクラスリテラルを評価しても、そのクラスは初期化されません。以前は初期化されていました。

この新しい動作は、VM がクラスリテラルを定数プール内でサポートするようになったことの結果です。5.0 より前のコンパイラでコンパイルしたクラスや `-target 1.4` フラグを指定してコンパイルしたクラスの場合、5.0 VM での実行時にも古い動作が残ります。

以前の動作に依存しているコードは、次のように書き直す必要があります。

```
//... Foo.class ... // 古いコード
... forceInit(Foo.class) ... // 新しいコード
/**
* Forces the initialization of the class pertaining to
* the specified <tt>Class</tt> object. This method does
* nothing if the class is already initialized prior to
* invocation.
*
* @param klass the class for which to force initialization
 * @return <tt>klass</tt>
*/
public static <T> Class<T> forceInit(Class<T> klass) {
    try {
       Class.forName(klass.getName(), true,
                     klass.getClassLoader());
    } catch (ClassNotFoundException e) {
              throw new AssertionError(e); // 発生しない
    }
    return klass;
   }
```

詳細については、「Java 言語仕様」の「クラスやインタフェースの初期化」(12.4 節) を参照してください。

注 : 言語仕様は変更されていません。言語仕様にクラスリテラル評価が初期化のトリガーとして記載されていたことは、これまで一度もありません。

4.4 ClassLoader のメソッドの引数

以前は、`String` クラス名引数を取る `ClassLoader` のメソッドに、非バイナリクラス名を指定することが可能でした。この動作は意図的なものではなく、長年にわたるクラス名の仕様に準拠したものでもありませんでした。5.0 では、これらの `ClassLoader` メソッドのパラメータチェックがこの仕様に準拠するように変更されたため、バイナリ名でないクラス名はすべて、ほかの認識されないクラス名と同様に処理されます。

`Class.forName` や `Class.getName` など、クラス名を明示的に要求または返す API は、バイナリ名を参照型として使用します。このため、この変更の影響を受ける開発者はほとんどいません。詳細については、「The Java Language Specification, Second Edition」のバイナリ名の定義を参照してください。また、バグ 4986512 の評価も参照してください。

4.5 API のデバッグとプロファイリング

1. 5.0 で、**Java Virtual Machine Debug Interface** (JVMDI) は非推奨になりました。JVMDI は次のメジャーリリースで削除されます。新しく開発する際には、すべてにおいて JVMTI を使用するべきです。既存ツールについては、JVMTI への移行を開始するべきです。
2. 5.0 で、**Java Virtual Machine Profiling Interface** (JVMPI) は非推奨になりました。JVMPI は次のメジャーリリースで削除されます。新しく開発する際には、すべてにおいて JVMTI を使用するべきです。既存ツールについては、JVMTI への移行を開始するべきです。

   これらの変更の詳細については、「JVMTI のドキュメント」を参照してください。

第 5 章
参照情報

The AWT Focus Subsystem

http://java.sun.com/j2se/1.5.0/ja/docs/ja/api/java/awt/doc-files/FocusSpec.html

バグ報告

http://developer.java.sun.com/developer/bugParade/bugs/bugNumber.html

クラスデータの共有

http://java.sun.com/j2se/1.5.0/ja/docs/ja/guide/vm/class-data-sharing.html

CORBA 互換性情報

http://java.sun.com/j2se/1.4/ja/compatibility-CORBA.html

ガベージコレクタのエルゴノミクス

http://java.sun.com/j2se/1.5.0/ja/docs/ja/guide/vm/gc-ergonomics.html

Generics Tutorial

http://java.sun.com/j2se/1.5/pdf/generics-tutorial.pdf

High Performance Graphics

http://java.sun.com/products/java-media/2D/perf_graphics.html

国際化の機能拡張

http://java.sun.com/j2se/1.5.0/ja/docs/ja/guide/intl/enhancements.html

Java 1.3 ドラッグ & ドロップ API 仕様

http://java.sun.com/j2se/1.3/ja/docs/ja/api/java/awt/dnd/DropTargetListener.html#dragExit(28java.awt.dnd.DropTargetEvent)

Java2D テクノロジのシステムプロパティ

http://java.sun.com/j2se/1.5.0/ja/docs/ja/guide/2d/flags.html

Java 5.0 API 仕様

http://java.sun.com/j2se/1.5.0/ja/docs/ja/api/

J2SE 5.0 Name and Version Change

http://java.sun.com/j2se/j2se-namechange.html

Java Database Connection (JDBC) 3.0 仕様

http://java.sun.com/j2ee/ja/jdbc/download.html

Java 配備ガイド

http://java.sun.com/j2se/1.5.0/ja/docs/ja/guide/deployment/deployment-guide/contents.html

J2SE Naming and Versioning

http://java.sun.com/j2se/naming_versioning_5_0.html

Java プログラミング言語

http://java.sun.com/j2se/1.5.0/ja/docs/ja/guide/language/index.html

The Java Language Specification, Second Edition

http://java.sun.com/docs/books/jls/

Java API for XML Processing (JAXP)

http://java.sun.com/xml/jaxp/index.jsp

JAXP 1.1 (Crimson)

http://xml.apache.org/crimson/

JAXP 互換性ガイド (J2SE 5 Platform 用)

http://java.sun.com/j2se/1.5.0/ja/docs/ja/guide/xml/jaxp/JAXP-Compatibility_150.html

JAXP のセキュリティー保護された処理

http://java.sun.com/j2se/1.5.0/ja/docs/ja/api/javax/xml/XMLConstants.html#FEATURE_SECURE_PROCESSING

JSR 14: 総称に対する JSR (Java Specification Request)

http://jcp.org/en/jsr/detail?id=14

JVMTI のドキュメント

http://java.sun.com/j2se/1.5.0/ja/docs/ja/guide/jvmti/index.html

Modified UTF-8

http://java.sun.com/developer/technicalArticles/Intl/Supplementary/index_ja.html#Modified_UTF-8

Java 2 SDK, v1.4 での Java2D の新機能

http://java.sun.com/j2se/1.5.0/ja/docs/ja/guide/2d/14_features.html

J2SE 5.0 での Java2D の新機能

http://java.sun.com/j2se/1.5.0/ja/docs/ja/guide/2d/new_features.html

フォーマッタ API

http://java.sun.com/j2se/1.5.0/ja/docs/ja/api/java/util/Formatter.html

javac コンパイラを含むツールとユーティリティーのリファレンスページ

http://java.sun.com/j2se/1.5.0/ja/docs/ja/tooldocs/index.html

サーバークラスマシンの検出

http://java.sun.com/j2se/1.5.0/ja/docs/ja/guide/vm/server-class.html

Standard End-of-Life (EOL) Policy

http://java.sun.com/products/archive/eol.policy.html

Java プラットフォームにおける補助文字のサポート (Unicode に関する優れたチュートリアル)
http://java.sun.com/developer/technicalArticles/Intl/Supplementary/index_ja.html

著者： Eric Armstrong

Sun Microsystems, Inc. 4150 Network Circle, Santa Clara, CA 95054 USA **Phone** 1-650-960-1300 or 1-800-555-9SUN **Web** sun.com

Sun Worldwide Sales Offices: Argentina +5411-4317-5600, Australia +61-2-9844-5000, Austria +43-1-60563-0, Belgium +32-2-704-8000, Brazil +55-11-5187-2100, Canada +905-477-6745, Chile +56-2-3724500, Colombia +571-629-2323 Commonwealth of Independent States +7-502-935-8411, Czech Republic +420-2-3300-9311, Denmark +45 4556 5000, Egypt +202-570-9442, Estonia +372-6-308-900, Finland +358-9-525-561, France +33-134-03-00-00, Germany +49-89-46008-0, Greece +30-1-618-8111, Hungary +36-1-489-8900, Iceland +354-563-3010, India–Bangalore +91-80-2298989/2295454; New Delhi +91-11-6106000; Mumbai +91-22-697-8111, Ireland +353-1-8055-666, Israel +972-9-9710500, Italy +39-02-641511, Japan +81-3-5717-5000, Kazakhstan +7-3272-466774, Korea +822-2193-5114, Latvia +371-750-3700, Lithuania +370-729-8468, Luxembourg +352-49 11 33 1, Malaysia +603-21161888, Mexico +52-5-258-6100, The Netherlands +00-31-33-45-15-000, New Zealand–Auckland +64-9-976-6800; Wellington +64-4-462-0780, Norway +47 23 36 96 00, People's Republic of China–Beijing +86-10-6803-5588; Chengdu +86-28-619-9333; Guangzhou +86-20-8755-5900; Shanghai +86-21-6466-1228; Hong Kong +852-2202-6688, Poland +48-22-8747800, Portugal +351-21-4134000, Russia +7-502-935-8411, Saudi Arabia +9661 273 4567, Singapore +65-6438-1888, Slovak Republic +421-2-4342-94-85, South Africa +27 11 256-6300, Spain +34-91-767-6000, Sweden +46-8-631-10-00, Switzerland–German 41-1-908-90-00; French 41-22-999-0444, Taiwan +886-2-8732-9933, Thailand +662-344-6888, Turkey +90-212-335-22-00, United Arab Emirates +9714-3366333, United Kingdom +44-1-276-20444, United States +1-800-555-9SUN or +1-650-960-1300, Venezuela +58-2-905-3800, or online at sun.com/store

Printed in USA 1/05