Software Engineering Institute

Java セキュアコーディング 並行処理編

Fred Long
Dhruv Mohindra
Robert Seacord
David Svoboda

2011 年 8 月（原著「Java Concurrency Guidelines」公開 2010 年 5 月）

TECHNICAL REPORT
CMU/SEI-2010-TR-015
ESC-TR-2010-015

CERT® Program

http://www.cert.org/

JPCERT CC®
一般社団法人 JPCERT コーディネーションセンター（JPCERT/CC）訳
https://www.jpcert.or.jp

Carnegie Mellon

Notification and Disclaimers

- *This non-SEI-sanctioned translation of “Java Concurrency Guidelines”, CMU/SEI-2010-TR-015, Copyright 2010 Carnegie Mellon University was prepared by JPCERT/CC with special permission from the Software Engineering Institute.*
- *This translation of Carnegie Mellon University copyrighted material is not an offical SEI-sanctioned translation.*
- *Neither Carnegie Mellon University nor the Software Engineering Institute directly or indirectly endorse this non-SEI-sanctioned translation. Accuracy and interpretation of this translation are the responsibility of JPCERT/CC. The SEI has not participated in this translation.*
- *CERT is a registered mark of Carnegie Mellon University.*
- *Copyright 2010 Carnegie Mellon University.*
- *NO WARRANTY. THIS CARNEGIE MELLON UNIVERSITY AND SOFTWARE ENGINEERING INSTITUTE MATERIAL IS FURNISHED ON AN "AS-IS" BASIS. CARNEGIE MELLON UNIVERSITY MAKES NO WARRANTIES OF ANY KIND, EITHER EXPRESSED OR IMPLIED, AS TO ANY MATTER INCLUDING, BUT NOT LIMITED TO, WARRANTY OF FITNESS FOR PURPOSE OR MERCHANTABILITY, EXCLUSIVITY, OR RESULTS OBTAINED FROM USE OF THE MATERIAL. CARNEGIE MELLON UNIVERSITY DOES NOT MAKE ANY WARRANTY OF ANY KIND WITH RESPECT TO FREEDOM FROM PATENT, TRADEMARK, OR COPYRIGHT INFRINGEMENT.*

著作権・免責事項

- この ”Java Concurenncy Guidelines”（CMU/SEI-2010-TR-015, コピーライト 2010年 カーネギーメロン大学）の翻訳は、SEI（Software Engineering Institute）の監訳ではなく、SEI の特別な許可の元、JPCERT/CCが行った翻訳である
- この翻訳は、カーネギーメロン大学が著作権を所有するマテリアルの翻訳であるが、SEI による公式な翻訳ではない
- カーネギーメロン大学と Software Engineering Institute は、直接的あるいは間接的にも、SEIによって承認されていないこの翻訳を支持しない。この翻訳の正確性と解釈の責任はJPCERT/CCに帰属する。SEIはこの翻訳に関与していない。
- CERT は カーネギーメロン大学の登録商標である
- コピーライト 2010年　カーネギーメロン大学
- 無保証。このカーネギーメロン大学とSoftware Engineering Institute の文書は、「現状有姿のまま」提供されるものである。カーネギーメロン大学は、特定目的適合性、商品性、独占性、および使用結果についての保証を含む、いかなる明示または暗黙の保証も提供しない。カーネギーメロン大学は、特許権、商標権、または著作権侵害の問題が生じても、いかなる責任も負うことはない。

この文書の利用について

- 内部利用

  この文書は、利用者組織の内部において複製し、二次的著作物を作成することができる。ただし、複製物および二次的著作物のすべてに、カーネギーメロン大学の著作権表示および無保証の表示をすること。

- 外部利用

  利用者組織外または商業用にこの文書を使用したい場合は、

  JPCERT コーディネーションセンター（JPCERT/CC） に相談すること。

  メールアドレス： office@jpcert.or.jp

翻訳について

本書は経済産業省の委託事業として、一般社団法人 JPCERT コーディネーションセンター（JPCERT/CC）が、米国 CERT® Coordination Center より 2010 年 5 月に発行された「Java Concurrency Guidelines」を翻訳したものである。

本翻訳では、原著者の許可の下、原著に存在したいくつかの誤りを修正している。また原著公開後に内容に変更が発生している点については注釈を追記した。

技術用語の翻訳にあたり、原著である英語版の参考文献の中から日本語翻訳版が存在する書籍を参考にした。これらの書籍は、末巻の「翻訳参考図書」に記載している。

目次

図の一覧

表の一覧

謝辞

Siddarth Adukia、Lokesh Agarwal、Ron Bandes、Kalpana Chatnani、Jose Sandoval Chaverri、Tim Halloran (SureLogic)、Thomas Hawtin、Fei He、Ryan Hofler、Sam Kaplan、Georgios Katsis、Lothar Kimmeringer、Bastian Marquis、Michael Kross、Christopher Leonavicius、Bocong Liu、Efstathios Mertikas、David Neville、Justin Pincar、Michael Rosenman、Eric Schwelm、Tamir Sen、Philip Shirey、Jagadish Shrinivasavadhani、Robin Steiger、John Truelove、Theti Tsiampali、Tim Wilson、および Weam Abu Zaki を含むガイドラインの開発に自らの時間と労力を寄付して頂いた皆様に感謝します。

また、この技術報告書および元となった Wiki の両方に対して注意深いレビューを実施して頂いた以下の人々にも、感謝申し上げます:Hans Boehm、Joseph Bowbeer、Klaus Havelund、David Holmes、Bart Jacobs、Niklas Matthies、Bill Michell、Philip Miller、Nick Morrott、Attila Mravik、Tim Peierls、Alex Snaps、Kenneth A. Williams。

更に、我々の編集者である Pamela Curtis と Pennie Walters にも感謝します。

この研究は、米国国防総省(DoD)および米国国土安全保障省(DHS) 国家サイバーセキュリティ部門(NCSD)の支援を受け実施されました。

この報告書について

この報告書のベースとなるセキュアコーディングスタンダード

この報告書のベースとなっている『CERT Oracle Java セキュアコーディングスタンダード』は、カーネギーメロン®大学ソフトウェア工学研究所の CERT®プログラムと Oracle の共同作業の成果である。同スタンダードは、コミュニティーの共同作業により www.securecoding.cert.org 上の CERT セキュアコーディング Wiki で整備された。この報告書（2010 年 5 月公開）は、CERT Oracle Java セキュアコーディングスタンダードの中で並行処理に関するガイドラインのサブセットを含んでいるが、これらは同スタンダードの整備と公開過程において、更なる修正が加わる可能性がある。並行処理に関するガイドラインは以下のカテゴリーに分類される。

- 可視性とアトミック性(VNA)
- ロック(LCK)
- スレッド API(THI)
- スレッドプール(TPS)
- スレッドの安全性に関する雑則(TSM)

これらのガイドラインに関する利用者からのフィードバックは歓迎されている。Wiki 上でコメントを行うには、直接サイトにアクセスして Wiki アカウント登録の手続きを行えば可能である。

ガイドラインの優先度

各ガイドラインでは、故障モード(failure mode)、影響(effect)、および致命度解析(criticality analysis)に基づくメトリクスを用いて割り当てた優先順位が定義されている(FMECA) [IEC 2006]。各ガイドラインに対して、以下の各項目に対する値を割り当てている。

- 深刻度 - ガイドラインが無視される場合の結果はどの程度深刻か？
  1 = 低(DoS 攻撃、異常終了)
  2 = 中(データの完全性違反、意図的ではない情報公開))
  3 = 高(任意のコード実行、権限昇格)

® CERT とカーネギーメロンはカーネギーメロン大学により米国特許商標庁で登録されている

- 可能性 - ガイドラインを無視した結果、欠陥が作り込まれた場合、その結果がどの程度まで悪用可能な脆弱性につながるか？

  1 = 低

  2 = 中

  3 = 高

- 修正コスト - ガイドラインに準拠するには、どの程度のコストを要するか？

  1 = 高(手動検出と手動修正)

  2 = 中(自動検出と手動修正)

  3 = 低(自動検出と自動修正)

続いて、上記の三つの値を各ガイドラインにおいて掛け合わせる。最終的な値(1～27)は、ガイドラインの適用優先順位をつけるために使用できるメトリクスを提供する。各ガイドラインは、優先順位の観点から以下の三つのレベルに分類される。

レベル-3(優先順位: 1～4)

レベル-2(優先順位: 6～9)

レベル-1(優先順位: 12～27)

その結果、図 1 で示されるように、レベル-1、レベル-2、あるいは完全準拠であるレベル-3に適合していることを該当レベルのガイドラインを実装することにより主張できる。

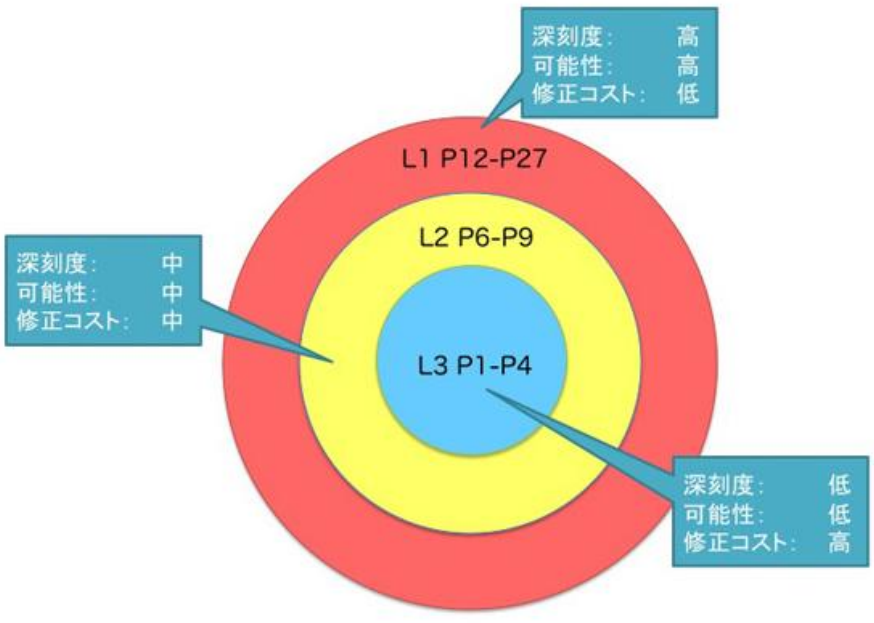

図 1: ガイドラインの優先順位

このメトリクスは主として既存ソフトウェアの脅威緩和プロジェクトでの適用を目的に設計されている。新規開発プロジェクトにおいては、すべてのスタンダードに準拠することが推奨される。

概要

Java プログラミング言語による安全なコーディングを行うために、信頼できるコーディング規約は必要不可欠である。コーディング規約は、プログラマの精通度あるいは好みではなく、プロジェクトや組織の要求により決定したガイドライン一式に従うようにプログラマを奨励する。一旦コーディング規約が確立されたならば、これらの規約はソースコードを手動あるいは自動化されたプロセスを用いて評価するためのメトリクスとして使用できる。

CERT Oracle Java セキュアコーディングスタンダードは、Java プログラミング言語による安全なコーディングのためのガイドラインを提供している。これらのガイドラインの目的は、悪用可能な脆弱性につながりうる安全でないコーディングの習慣および未定義な動作を除去することである。このセキュアコーディングスタンダードを適用することで、堅牢でかつ攻撃耐性のあるより高品質なシステム構築につながるだろう。

この報告書は、並行処理に関する Java ガイドラインを記載している。

1. はじめに

スレッド間で共有可能なメモリは、*共有メモリ*または*ヒープメモリ*と呼ばれる。*変数*という用語は、この報告書で使われているように、フィールドと配列要素の両方を指す[Gosling 2005]。 スレッド間で共有される変数は、*共有変数*と呼ばれる。すべてのインスタンスフィールド、静的フィールド、および配列要素は、ヒープメモリに配置される共有変数である。ローカル変数やメソッドの引数、および例外ハンドラの引数は、決してスレッド間で共有されず、Java メモリモデル(JMM)の影響を受けない。

現代の共有メモリ型マルチプロセッサアーキテクチャは、各プロセッサでメインメモリの内容を定期的に保存するキャッシュを複数レベルで保持している(図 2)。

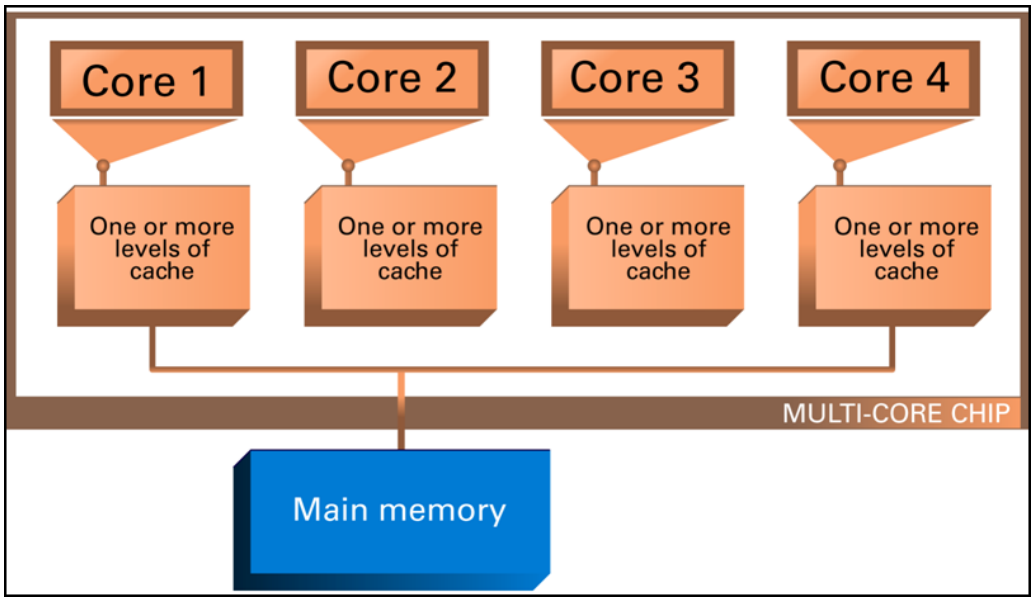

図 2: 現代の共有メモリ型マルチプロセッサのアーキテクチャ

共有変数の値はキャッシュされ、すぐにはメインメモリに書き込まれないかもしれないため、共有変数への書込み結果の可視性は問題となりやすい。他のスレッドが陳腐化した(最新でない)変数値を読み取るおそれがある。

更なる懸念点としては、コードの同時並行的な実行は一般的にインターリーブされ、各ステートメントの実行は、パフォーマンス最適化のため、コンパイラやランタイムシステムにより並び替えが行われるかもしれない。したがって、ソースコードを見ただけでは、実行順序がどうなるかは明確ではない。実行順序の並び替えの可能性を考慮していないことは、データ競合の良く見られる原因の一つである。

下記の例において、a と b は(共有)グローバル変数またはインスタンスフィールドであり、r1 と r2 は他スレッドからアクセスが不可能なローカル変数であると仮定する。

はじめに、a = 0、b = 0 とする (表 1)。

表 1: スレッドによる代入例 その 1

スレッド 1	スレッド 2
a = 10;	b = 20;
r1 = b;	r2 = a;

スレッド 1 における二つの代入(a = 10; および r1 = b;)には関連性がないので、コンパイラやランタイムシステムは並び替えを行っても良い。同様に、スレッド 2 においても並び替えは自由に行われる。また、直感に反するようだが、Java メモリモデルでは実行順で後の書込みによる値の先読みを許している。

取りうる実行順序と代入の一例を表 2 に示す。

表 2: 実行順序と代入例 1

実行順 (時間)	スレッド番号	代入	代入される値	注意事項
1	t_1	a = 10;	10	
2	t_2	b = 20;	20	
3	t_1	r1 = b;	0	b の初期値、即ち 0 を読み取る
4	t_2	r2 = a;	0	a の初期値、即ち 0 を読み取る

この実行順の例においては、r1 と r2 は、更新後の値(20 と 10)を読み取ることを期待されているにもかかわらず、各々変数 b と a の更新前の値を読み取る。以下の表には、別の実行順序と代入例を示している。

表 3: 実行順序と代入例 2

実行順 (時間)	スレッド番号	代入	代入される値	注意事項
1	t_1	r1 = b;	20	書込み(ステップ 4)後の値(20)を読み取る
2	t_2	r2 = a;	10	書込み(ステップ 3)後の値(10)を読み取る
3	t_1	a = 10;	10	
4	t_2	b = 20;	20	

上記の実行順では、r1 と r2 は、ステップ 4 と 3 の実行前に、その実行結果の b と a の値を読み取る。

起こりうる並び替えのパターンを制限すれば、ソースコードが正しいかどうかを判断しやすくなる。しかし、たとえ各ステートメントがスレッドに記述されている順序で実行されたとしても、キャッシングにより最新の値がメインメモリに反映されないことも考えられる。

プログラマは *Java 言語仕様*で定義された JMM に基づき並行処理を実装できる。JMM は動作の見地から規定されており、変数の読取りや書込み、モニタのロックやアンロック（固有ロックの取得や解放）、およびスレッドの開始や結合といった動作を含んでいる。JMM では、プログラム内の全動作において*事前発生*(happens-before)と呼ばれる半順序関係を定義している。たとえば、動作 B を実行するスレッドが動作 A の結果を見られることを保証するためには、「A は B の前に起こる」というような happens-before 関係が定義されねばならない。

Java 言語仕様 の 17.4.5 節「事前発生の順序」[Gosling 2005] によると、

1. *モニタのアンロックは、後に続くすべての該当モニタへのロックよりも事前発生する。*
2. *volatile フィールドへの書込みは、後に続くすべての該当フィールドの読取りよりも事前発生する。*
3. *スレッドの start() 呼出しは、開始されるスレッド中の任意の動作よりも事前発生する。*
4. *あるスレッド中のすべての動作は、そのスレッドへの join() から正常に戻った他のスレッドよりも事前発生する。*
5. *オブジェクトのデフォルト初期化は、プログラムの他の任意の動作よりも事前発生する。*
6. *スレッドによる他スレッドへの割込みの呼出しは、割込まれたスレッドによる割込みの検知よりも事前発生する。*
7. *オブジェクトのコンストラクタの終了は、該当オブジェクトのファイナライザの開始よりも事前発生する。*

二つの操作間に事前発生関係(happens-before relationship)が存在しない場合、Java 仮想マシン(JVM)は自由に並び替えを行う。ある変数が一つのスレッドにより書き込まれ、少なくとも一つの他のスレッドが読み取り、これらの読取りと書込みに事前発生関係が存在しない場合、データ競合が発生する。正しく同期化されたプログラムではデータ競合は発生しない。JMM は、正しく同期化されたプログラムの*逐次一貫性* を保証する。逐次一貫性とは、あたかもすべてのスレッドによる共有変数への読み書き操作が、ある一定の順序に沿って実行され、かつ、各スレッドの操作が該当するプログラムに記述された順序通りに実行さ

れたかのように、どのような実行結果にも差異が発生しないことを意味する[Tanenbaum 2002] 。これを次のように言いかえることができる。

1. 各スレッドが実行する読取りと書込みの操作を各スレッドが実行する順序に並び替える (スレッド順序)。
2. 事前発生関係に基づいて操作をインターリーブし、一つの実行順序を形成する。
3. 実行が逐次一貫性を有するには、すべての読込み操作は、プログラム順序中、その時点で最新の書込みデータを返さなければならない。

プログラムが逐次一貫性を有するとき、すべてのスレッドによる共有変数の読取りと書込みが同じ順序に基づいて行われる。

命令とメモリアクセスの実際の実行順序は、次の制約の範囲内で変更されうる。
・ スレッドの各操作が、プログラム全体の実行順序（プログラム順序）に沿っている
・ すべての値は、Java メモリモデルの規約に沿って読み取られている

これらの一連の仕様によって、プログラマは作成するプログラムのセマンティクスを理解することが可能となり、コンパイラ開発者や仮想マシン実装者は多様な最適化を行うことができる[Arnold 2006]。

Java 言語には並列処理に関する様々な言語要素が含まれており、プログラマがマルチスレッドプログラムのセマンティクスを理解するのに役立つ。

1.1.1. volatile キーワード

共有変数を volatile 修飾して宣言することで、可視性を確保し、また、変数へのアクセス順序の並び替えを制限できる。volatile 変数へのアクセスは、変数値のインクリメントなど複合的な操作のアトミック性を保証するものではない。したがって、複合的な操作のアトミック性が保証されなければならない場合は、volatile 修飾だけでは十分ではない。(詳細は、ガイドライン「VNA02-J. 共有変数への複合操作のアトミック性を確保する」を参照)

変数を volatile 宣言することで、voaltile 変数への書込みが、常に後続の読取りを行うスレッドに対して可視となる事前発生関係が確立される。volatile 変数への書込みの前に実行される各ステートメントもまたその volatile 変数へのあらゆる読取りの前に発生する。

いくつかのステートメントを実行する二つのスレッドについて考察する (図 3)。

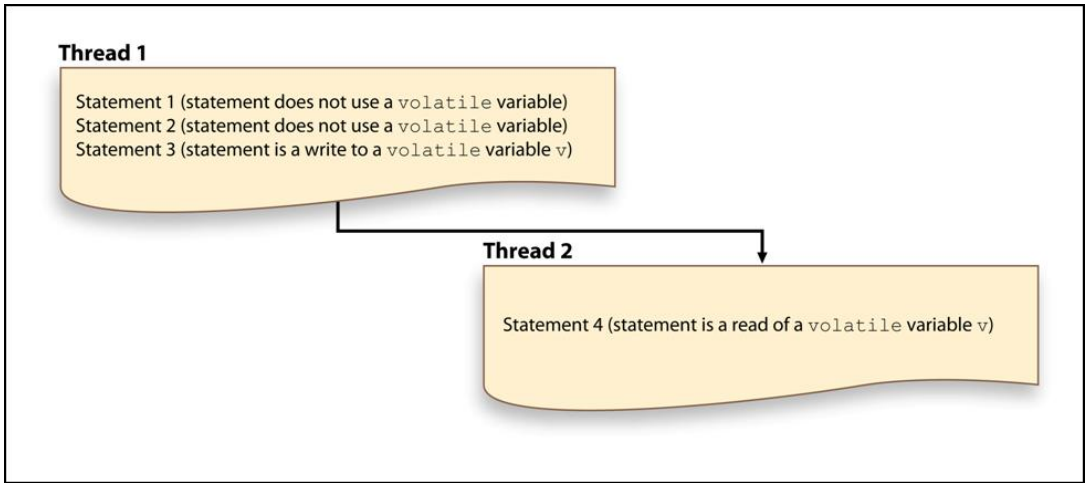

図 3: 二つのスレッドとその実行ステートメントの例

Thread 1 と Thread 2 は事前発生関係を持ち、Thread 2 は Thread 1 が完了するまでは開始されない。

上記の例では、Thread 1 の Statement 3 が書き込む volatile 変数 v を、Thread 2 の Statement 4 が読み取る。この読取り結果は、Statement 3 の最新の書込み結果が反映されている。

volatile 変数の読込みと書込み操作は並び替えることができず、また、非 volatile 変数の操作と並び替えることもできない。Thread 2 が volatile 変数を読み取る時点で、Thread 1 の volatile 変数への書込み以前の書込みはすべて可視である。このように volatile 変数の読み書きは比較的強い保証が適用されるため、パフォーマンス上のオーバーヘッドは同期化を行う場合とほぼ同じである。

前述の例において、Statement 1 と Statement 2 がソースコードの記述順通りに実行される保証はない。両ステートメント間に事前発生関係は存在しないため、コンパイラは自由に並び替えることが許されている。

volatile 変数と非 volatile 変数間で発生しうる並び替えを表 4 にまとめる。load と store は、それぞれ読取りと書込みと同義である[Lea 2008]。

表 4: volatile 変数および非 volatile 変数間の並び替え

並び替えが可能	*二番目の操作*			
最初の操作	非 volatile 変数の load	非 volatile 変数の store	volatile 変数の load	volatile 変数の store
非 volatile 変数の load	可	可	可	不可
非 volatile 変数の store	可	可	可	不可
volatile 変数の load	不可	不可	不可	不可
volatile 変数の store	可	可	不可	不可

1.1.2. 同期化（Synchronization）

正しく同期化されたプログラムは、取りうる逐次実行順序にデータ競合を含まない。以下の例は、使用している非 volatile 変数 x と volatile 変数 y が正しく同期化されていない。

表 5: スレッドによる代入例 その 2

スレッド 1	スレッド 2
x = 1	r1 = y
y = 2	r2 = x

上記の例について、二通りの逐次実行順序を表 6 および 表 7 に示す。

表 6: 実行順序 #1

ステップ (時間)	スレッド#	ステートメント	コメント
1	t_1	x = 1	非 volatile 変数への書込み
2	t_1	y = 2	volatile 変数への書込み
3	t_2	r1 = y	volatile 変数の読取り
4	t_2	r2 = x	非 volatile 変数の読取り

表 7: 実行順序 #2

ステップ (時間)	スレッド#	ステートメント	コメント
1	t_2	r1 = y	volatile 変数の読取り
2	t_2	r2 = x	非 volatile 変数の読取り
3	t_1	x = 1	非 volatile 変数への書込み
4	t_1	y = 2	volatile 変数への書込み

最初の実行順序では、事前発生関係がステップ間に存在し、ステップ 1 と 2 は常にステップ 3 と 4 の前に実行される。しかし、二番目の実行順序では、どのステップ間にも 事前発生関係が存在しない。よって、このスレッド代入例（*表 5: スレッドによる代入例 その 2*）は、事前発生関係を持たない逐次実行順序を取りうるため、データ競合を含んでいる。

可視性を正しく確保することで、共有データにアクセスする複数のスレッドが互いの処理結果を得ることができるが、各スレッドの共有データへのアクセス順序は定まらない。同期化を正しく行うことで、各スレッドが共有データへのアクセスを適切な順序で行うことを保証できる。たとえば、以下のコードでは、逐次実行順序は一つとなり、スレッド 2 の動作の前にスレッド 1 の動作すべてが実行される。

```
class Assign {
  public synchronized void doSomething() {
    // スレッド 1 の動作を実行
    x = 1;
    y = 2;
    // スレッド 2 の動作を実行
    r1 = y;
    r2 = x;
  }
}
```

同期化する場合、変数 y を volatile 宣言する必要はない。同期化には、ロックの確保、各操作の実行、およびロックの解放が含まれている。上記の例では、doSomething()メソッドがオブジェクト(Assign)の固有ロックを取得している。ブロック同期を用いて以下のようにコーディングしてもよい。

```
class Assign {
  public void doSomething() {
    synchronized (this) {
      // スレッド 1 の動作を実行
      x = 1;
      y = 2;
      // スレッド 2 の動作を実行
      r1 = y;
      r2 = x;
    }
  }
}
```

両方の例で使われている固有ロックは同じである。

1.1.3. java.util.concurrent パッケージ

1.1.3.1 アトミック変数クラス

volatile 変数は可視性を保証するために有効であるが、アトミック性を保証するためには不十分である。同期化はこのギャップを埋めるものではあるが、コンテキストスイッチのオーバーヘッドを招いたり、ロック競合の原因となることがある。java.util.concurrent.atomic パッケージのクラスは、多くの実環境で競合を減らし、かつ、アトミック性を保証する仕組みを提供する。Goetz らは以下のように述べている[Goetz 2006] 。

> *低～中レベルの競合状態では、アトミック変数がより良いスケーラビリティを提供する。高レベルの競合状態では，ロックがより良い競合状態の回避を提供する。*

アトミック変数クラスの実装は、最近のプロセッサの設計を生かして、プログラマが一般的に必要とする機能を提供している。たとえば、AtomicInteger.incrementAndGet()メソッドを使うと、変数をアトミックにインクリメントすることができる。最近のプロセッサが提供するコンペア・アンド・スワップ命令により更に細かい制御が可能で、以下のような高レベルのメソッド呼出しから直接利用することもできる。

java.util.concurrent.atomic.Atomic*.compareAndSet()

注：アスタリスク(*) には、Integer、Long、または Boolean などがあてはまる。

1.1.3.2 Executor フレームワーク

java.util.concurrent パッケージは、複数タスクを並行実行する仕組みを提供する Executor フレームワークを含んでいる。*タスク*とは、Runnable や Callable インタフェースを実装するクラスによりカプセル化された作業の論理ユニットのことである。Executor フレームワークは、低レベルのスケジューリングや細かなスレッド管理からタスクの依頼を分離することを可能にする。このフレームワークは、システムが同時に処理できる分量以上のリクエストを受けた場合でも、一定の機能を提供し続けることができるスレッドプールの仕組みを提供する。

Executor インターフェースはフレームワークの核となるインターフェースであり、スレッドプールを終了したり各タスクの返り値(Future インターフェース)を取得することができる ExecutorService インターフェースによって拡張されている。ExecutorService インターフェースは更に、各タスクを定期的にあるいは多少の遅延時間後に実行可能な ScheduledExecutorService インターフェースによって拡張されている。Executors クラスは、いくつかのファクトリメソッドやユーティリティメソッドを持ち、これらのメソッドは Executor、ExecutorService、および他の関連するインターフェースから一般的に利用される既製構成のスレッドプールを提供する。たとえば、Executors.newFixedThreadPool() メソッドは、サイズが無制限のタスク待ち受けキューと、並行して実行可能なタスクの上限数が定まった固定長のプールとで構成されたスレッドプールを返す。スレッドプールの実装は、ThreadPoolExecutor クラスにより提供される。このクラスをインスタンス化し、タスクの実行ポリシーをカスタマイズすることができる。

java.util.concurrent パッケージのユーティリティは、同期化や volatile 変数など従来の同期プリミティブよりも好まれる。その理由は、java.util.concurrent ユーティリティが、根底の詳細を抽象化した簡潔で誤用しにくい API を持ち、スケーラビリティが確保し易く、ポリシーの適用によって利用を強制できるからである。

1.1.3.3 明示的ロック

java.util.concurrent パッケージは、固有ロックにはない機能を提供する ReentrantLock クラスを含んでいる。たとえば、ReentrantLock.tryLock()メソッドは他のスレッドが既にロックを保持している場合はロックの取得待ちでブロックしない。ReentrantLock の取得および解放は、固有ロックの取得および解放と同一のセマンティックを持つ。

2. 可視性とアトミック性のガイドライン

2.1. VNA00-J. 共有プリミティブ型変数の可視性を確保する

あるスレッドで共有プリミティブ型変数を読み取っても、その値には他のスレッドによる最新の書込みが反映されていないかもしれない。その結果、スレッドは陳腐化した(最新ではない) 共有変数の値を得ることになるかもしれない。最新の更新を反映して可視性を確保するには、変数を volatile 宣言するか、読取りと書込みを同期化する必要がある。

共有変数を volatile 宣言することでスレッドセーフに可視性が保証できるのは、以下の条件を満たす場合のみである。

- ある変数への書込みが、変更前の値に依存しない。
- ある変数への書込みが、他の変数の読取りと書込みを伴うアトミックでない複合操作の結果に依存しない。(詳細は、ガイドライン「VNA02-J. 共有変数への複合操作のアトミック性を確保する」を参照。)

ただ一つのスレッドだけが変数の値を更新することが確実な場合は、一点目の条件は除外できる[Goetz 2006]。しかし、特定のスレッドがただ一つであることに依存したコーディングは間違いをおこしやすく、メンテナンスしづらい。当ガイドラインはこういったコーディングを認めるが、推奨はしない。

コードを同期化することでその振舞いが理解し易くなり、単に volatile キーワードを使用するよりも多くの場合は安全性を高めることになる。しかし、同期化は、パフォーマンス上のオーバーヘッドが大きく、また、過度に使用するとスレッド間の競合およびデッドロックにつながりうる。

変数を volatile 宣言したり、コードを正しく同期化することで、64 ビットのプリミティブ型変数(long および double)へのアクセスをアトミックに行うことを保証できる。(これらの変数を複数のスレッドで共有する場合の詳細は、「VNA05-J. 64 ビット値の読み書きのアトミック性を確保する」を参照。)

2.1.1. 違反コード (volatile 宣言されていない変数)

以下の違反コード例では、shutdown() メソッドを使用して、run() メソッドでチェックされる非 volatile なフラグである done をセットしている。

```
final class ControlledStop implements Runnable {
  private boolean done = false;
  @Override public void run() {
    while (!done) {
      try {
        // ...
        Thread.currentThread().sleep(1000); //  何らかの処理を行う
      } catch(InterruptedException ie) {
        Thread.currentThread().interrupt(); //  割込みステータスをリセット
      }
    }
  }

  public void shutdown() {
    done = true;
  }
}
```

あるスレッドで shutdown()メソッドを呼び出してフラグをセットしても、他のスレッドではその変更結果を得ることができないかもしれない。その結果、他のスレッドは done フラグが false のままであるとし、sleep()メソッドを誤って呼び出してしまうおそれがある。また、done の値が同じスレッドによって変更されない場合、コンパイラはコードを最適化してもよく、結果として無限ループにつながる。

2.1.2. 適合コード (volatile 変数)

以下の適合コードでは、done フラグを volatile 変数として宣言し、done フラグへの書込みが他のスレッドからも可視であることを確実にしている。

```
final class ControlledStop implements Runnable {
  private volatile boolean done = false;
  @Override public void run() {
    while (!done) {
      try {
        // ...
        Thread.currentThread().sleep(1000); //  何らかの処理を行う
      } catch(InterruptedException ie) {
        Thread.currentThread().interrupt(); //割込みステータスをリセット
      }
    }
  }

  public void shutdown() {
    done = true;
  }
}
```

2.1.3. 適合コード (AtomicBoolean クラス)

以下の適合コードでは、done フラグは AtomicBoolean クラスとして宣言される。アトミック型変数は、書込みが他のスレッドに可視であることを保証する。

```
final class ControlledStop implements Runnable {
  private final AtomicBoolean done = new AtomicBoolean(false);
  @Override public void run() {
    while (!done.get()) {
      try {
        // ...
        Thread.currentThread().sleep(1000); //  何らかの処理を行う
      } catch(InterruptedException ie) {
        Thread.currentThread().interrupt(); //  割込みステータスをリセット
      }
    }
  }

  public void shutdown() {
    done.set(true);
  }
}
```

2.1.4. 適合コード (synchronized メソッド)

以下の適合コードでは、オブジェクトの固有ロックを使用することで、フラグの更新が他のスレッドに可視となる。

```
final class ControlledStop implements Runnable {
  private boolean done = false;
  @Override public void run() {
    while (!isDone()) {
      try {
        // ...
        Thread.currentThread().sleep(1000); //何らかの処理を行う
      }     catch(InterruptedException ie) {
        Thread.currentThread().interrupt(); //割込みステータスをリセット
      }
    }
  }

  public synchronized boolean isDone() {
    return done;
  }

  public synchronized void shutdown() {
    done = true;
  }
}
```

上記は適合コードではあるが、固有ロックを使用することでスレッド間のやり取りがブロックされ、ロックの競合が生じるかもしれない。一方、volatile 共有変数は、スレッド間でやり取りをブロックしない。また、過度な同期化はデッドロックにもつながりやすい。

変数の更新後の値が変更前の値に依存するなど volatile キーワードや java.util.concurrent.atomic.Atomic* フィールドの使用が不適切な場合、同期化はより安全な選択肢である。詳細はガイドライン「VNA02-J. 共有変数への複合操作のアトミック性を確保する」を参照。

ガイドライン「LCK00-J. 信頼できないコードから使用されるクラスの同期化には private final ロックオブジェクトを使用する」を遵守し、信頼できない呼出し元がロックオブジェクトにアクセスできないようにすることで同期化に伴う脅威を低減できる。

2.1.5. 例外

VNA00-EX1: Class オブジェクト（java.lang.Class）は、仮想マシンで生成され、使用される前に常に初期化が行われるため、別途可視化する必要はない。

2.1.6. リスク評価

共有プリミティブ型変数の可視性を確保できない場合、スレッドが変数の陳腐化した値を参照するおそれがある。

ガイドライン	深刻度	可能性	修正コスト	優先度	レベル
VNA00-J	中	中	中	P8	L2

2.1.7. 参考文献

[Arnold 2006]	Section 14.10.3, “The Happens-Before Relationship”
[Bloch 2008]	Item 66: “Synchronize access to shared mutable data”
[Gosling 2005]	Chapter 17, Threads and Locks: Section 17.4.5, “Happens-Before Order” Section 17.4.3, “Programs and Program Order” Section 17.4.8, “Executions and Causality Requirements”
[MITRE 2010]	CWE ID 667, “Insufficient Locking” CWE ID 413, “Insufficient Resource Locking” CWE ID 567, “Unsynchronized Access to Shared Data”

2.2. VNA01-J. 不変オブジェクトへの共有参照の可視性を確保する

不変オブジェクトの共有参照は、その値が更新されるとただちに複数スレッドにわたって可視となる、と誤解されていることが多い。たとえば、不変オブジェクトだけを参照するフィールドを含むクラスはそれ自体が不変であり、それゆえにスレッドセーフであると誤った認識を持ってしまう。

*プログラミング言語 Java 第 4 版*の 14.10.2 節「Final Fields and Security」[Arnold 2006]に、以下のように記述されている

> *問題は、共有オブジェクトは不変であるが、共有オブジェクトへのアクセスのために使用される参照それ自体も共有されており、しばしば可変であるということである。したがって、正しく同期化するためには、その共有参照へのアクセスを同期化する必要があるが、プログラマが必要性を認識していないため、そのように作成されない場合が多い。*

不変および可変オブジェクトへの参照は、いずれもすべてのスレッドに可視化されなければならない。不変オブジェクトは、複数のスレッド間で安全に共有することができる。しかし、可変オブジェクトの場合は、その参照が可視となるタイミングで、オブジェクトの構築が不完全な状態かもしれない。ガイドライン「TSM03-J. 初期化が不完全なオブジェクトを公開しない」では、オブジェクト構築および可視性に特化した問題について記述している。

2.2.1. 違反コード

以下の違反コードは、不変の Helper クラスおよび可変の Foo クラスで構成されている。

```
// 不変の Helper クラス
public final class Helper {
  private final int n;
  public Helper(int n) {
    this.n = n;
  }
  // ...
}
// 可変の Foo クラス
final class Foo {
  private Helper helper;
  public Helper getHelper() {
    return helper;
  }
  public void setHelper(int num) {
    helper = new Helper(num);
  }
}
```

getHelper()メソッドは、可変の helper フィールドを公開する。Helper クラスは不変であるため、初期化後に変更されない。また、Helper クラスは不変であるため、その参照が可視となる前に適切に構築される。これはガイドライン「TSM03-J. 初期化が不完全なオブジェクトを公開しない」に適合している。しかし、他のスレッドが、Foo クラスの helper フィールドの陳腐化した参照を読み取ることはありうる。

2.2.2. 適合コード (synchronized メソッド)

以下の適合コードでは、Foo クラスでメソッドを同期化させることにより、各スレッドが陳腐化した Helper オブジェクト参照を得ることがなくなる。Helper クラスに変更はない。

```
final class Foo {
  private Helper helper;

  public synchronized Helper getHelper() {
    return helper;
  }

  public synchronized void setHelper(int num) {
    helper = new Helper(num);
  }
}
```

2.2.3. 適合コード (volatile 変数)

不変のメンバーオブジェクトへの参照は、volatile 宣言により可視化することができる。Helper クラスに変更はない。

```
final class Foo {
  private volatile Helper helper;

  public Helper getHelper() {
    return helper;
  }

  public void setHelper(int num) {
    helper = new Helper(num);
  }
}
```

2.2.4. 適合コード (java.util.concurrent パッケージ)

以下の適合コードでは、不変な Helper オブジェクトを、アトミックな操作を提供する AtomicReference でラップしている。Helper クラスに変更はない。

```
final class Foo {
  private final AtomicReference<Helper> helperRef =
    new AtomicReference<Helper>();

  public Helper getHelper() {
    return helperRef.get();
  }

  public void setHelper(int num) {
    helperRef.set(new Helper(num));
  }
}
```

2.2.5. リスク評価

不変なオブジェクトだけを含むクラスは不変であるという仮定は誤りであり、スレッドの安全性に関する深刻な問題を引き起こす可能性がある。

ガイドライン	深刻度	可能性	修正コスト	優先度	レベル
VNA01-J	低	中	中	P4	L3

2.2.6. 参考文献

[Arnold 2006]	Section 14.10.2, “Final Fields and Security”
[Goetz 2006]	Section 3.4.2, “Example: Using Volatile to Publish Immutable Objects”
[Sun 2009b]	

2.3. VNA02-J. 共有変数への複合操作のアトミック性を確保する

複合操作とは、複数の独立した操作から構成される操作のことである。前置増分または後置増分の演算子(++)、前置減分または後置減分の演算子(--)、あるいは複合代入演算子を含む数式は、複合操作を構成する。複合代入式では、*=、/=、%=、+=、-=、<<=、>>=、>>>=、^=、および |= のような演算子が使用される [Gosling 2005]。共有変数への複合的な操作は、アトミックに実行し、データ競合や競合状態を防がなければならない。

スレッドセーフなクラスに属するアトミックなメソッドをまとめて呼び出した場合のアトミック性については、ガイドライン「VNA03-J. アトミックなメソッドをまとめた呼出しがアトミックであると仮定しない」を参照。

また、*Java 言語仕様*では、64 ビット値の読取りおよび書込みがアトミックでなくてもよいと規定している。詳細は、ガイドライン「VNA05-J. 64 ビット値の読み書きのアトミック性を確保する」を参照。

2.3.1. 違反コード (論理反転)

以下の違反コードでは、共有の boolean flag 変数を宣言し、flag の現在値を反転する toggle() メソッドを提供している。

```
final class Flag {
  private boolean flag = true;
  public void toggle() { // スレッドセーフではないメソッド
    flag = !flag;
  }

  public boolean getFlag() { // スレッドセーフではないメソッド
    return flag;
  }
}
```

flag の値が読み取られ、反転され、そして書き戻されるので、コードを実行するとデータ競合が発生するかもしれない。

toggle()メソッドを呼び出す二つのスレッドを例に考えてみよう。flag を二回反転した時に期待される結果は、flag がオリジナルの値に戻されるということである。しかし、以下のシナリオでは、flag が正しくない状態となる。

時間	flag の値	スレッド	動作
1	true	t_1	flagの現在値 true を一時変数に読み取る
2	true	t_2	flagの現在値 true(無変化) を一時変数に読み取る
3	true	t_1	一時変数を　falseに切り替える
4	true	t_2	一時変数を　falseに切り替える
5	false	t_1	一時変数の値を　flagに書き込む
6	false	t_2	一時変数の値を　flagに書き込む

その結果、t_2 による呼出しは flag に反映されない。即ち、あたかも toggle()メソッドが二回ではなく一回だけ呼ばれたかのようにプログラムは振る舞う。

2.3.2. 違反コード (ビット単位反転)

同様に、toggle()メソッドは、flag の現在値を反転するために複合代入演算子 ^=を使用することができる。

```
final class Flag {
   private boolean flag = true;
   public void toggle() { // スレッドセーフではないメソッド
      flag ^= true; // flag = !flag; と同様
   }
   public boolean getFlag() { // スレッドセーフではないメソッド
      return flag;
   }
}
```

上記のコードもスレッドセーフではない。^= がアトミックではない複合操作なので、データ競合が存在する。

2.3.3. 違反コード (volatile 変数)

flag を volatile 修飾し宣言してもスレッドセーフにはならない。

```
final class Flag {
   private volatile boolean flag = true;
   public void toggle() { // スレッドセーフではないメソッド
      flag ^= true;
   }
   public boolean getFlag() { // スレッドセーフなメソッド
      return flag;
   }
}
```

ある変数を volatile 宣言しても、その変数への複合操作のアトミック性は保証されないため、このコードはスレッドセーフではない。

2.3.4. 適合コード (synchronized メソッド)

以下の適合コードでは、toggle()メソッドおよび getFlag()メソッドの両方を synchronized 修飾している。

```
final class Flag {
  private boolean flag = true;
  public synchronized void toggle() {
    flag ^= true; // flag = !flag; と同様
  }

  public synchronized boolean getFlag() {
    return flag;
  }
}
```

この例では this インスタンスの固有ロックを使用することにより flag フィールドの読取りと書込みを保護している。上記適合コードでは、変更内容が全スレッドに可視となる。結果的に、実行順序を二つに絞り込むことができる。以下に一つの例を示す。

時間	flag の値	スレッド	動作
1	true	t_1	flagの現在値 trueを一時変数に読み取る
2	true	t_1	一時変数を falseに切り替える
3	false	t_1	一時変数の値を flagに書き込む
4	false	t_2	flagの現在値 falseを一時変数に読み取る
5	false	t_2	一時変数を trueに切り替える
6	true	t_2	一時変数の値を flagに書き込む

もう一方の実行順序では、上記例と同様の操作を含むが、t_2 が t_1 の前に開始して終了する。

また、ガイドライン「LCK00-J. 信頼できないコードから使用されるクラスの同期化には private final ロックオブジェクトを使用する」を遵守し、信頼できない呼出し元がロックオブジェクトにアクセスできないようにすることで同期化に伴う脅威を低減できる。

2.3.5. 適合コード (volatile 変数の読取り，同期書込み)

以下の適合コードでは、getFlag()メソッドを同期化しておらず、flag を volatile 宣言している。getFlag()メソッドにおける flag の読取りはアトミックな操作であり、volatile 修飾により可視性が確保されるので、この解決方法はガイドラインに適合している。toggle()メソッドではアトミックでない操作を実行するので、同期化する必要がある。

```
final class Flag {
  private volatile boolean flag = true;
  public synchronized void toggle() {
    flag ^= true; // flag = !flag; と同様
  }
  public boolean getFlag() {
    return flag;
  }
}
```

ゲッターメソッド getFlag() が同期化を行わず、volatile 宣言されたフィールドの値を返す以外の操作を実行する場合、このアプローチは役に立たない。また、読取り処理のパフォーマンスが特に重要な場合を除き、このアプローチは通常の同期化と比較して目立った優位性はないかもしれない [Goetz 2006]。

関連してガイドライン「VNA06-J. オブジェクトへの参照を volatile 宣言することでメンバーの可視性が保証されると想定しない」においても、volatile 変数の読取りおよび同期書込みパターンについて記述している。

2.3.6. 適合コード (リードライトロック)

以下の適合コードでは、アトミック性および可視性を確保するためにリードライトロックを使用している。

```
final class Flag {
  private boolean flag = true;
  private final ReadWriteLock lock = new ReentrantReadWriteLock();
  private final Lock readLock = lock.readLock();
  private final Lock writeLock = lock.writeLock();
  public synchronized void toggle() {
    writeLock.lock();
    try {
      flag ^= true; // flag = !flag; と同様
    } finally {
      writeLock.unlock();
    }
  }
  public boolean getFlag() {
    readLock.lock();
    try {
      return flag;
    } finally {
      readLock.unlock();
    }
  }
}
```

リードライトロックでは、共有オブジェクトが複数のリーダーあるいは単一のライターのどちらかによってアクセスされることを許されているが、その両方の同時アクセスは許されない。Goetz は次のように述べている [Goetz 2006] 。

> *リードライトロックは、マルチプロセッサシステム上で読込み中心のデータ構造に頻繁にアクセスする場合に性能の向上につながるが、その他の条件では、その複雑さのせいで排他的ロックよりも性能が若干劣ることもある。*

アプリケーションのプロファイリングを行うことで、リードライトロックが適しているかどうかを判定できる。

2.3.7. 適合コード (AtomicBoolean クラス)

以下の適合コードでは、flag を AtomicBoolean 型として宣言している。

```
import java.util.concurrent.atomic.AtomicBoolean;

final class Flag {
   private AtomicBoolean flag = new AtomicBoolean(true);
   public void toggle() {
      boolean temp;
      do {
         temp = flag.get();
      } while (!flag.compareAndSet(temp, !temp));
   }
   public AtomicBoolean getFlag() {
      return flag;
   }
}
```

flag 変数の更新は、AtomicBoolean クラスの compareAndSet()メソッドを使用して行われる。すべての更新は、他のスレッドに可視となる。

2.3.8. 違反コード (プリミティブ型変数の加算)

以下の違反コードでは、複数のスレッドが setValues()メソッドを使用して、フィールド a および b に値をセットすることができる。このクラスは整数オーバーフローが発生するかどうかをチェックしていないため、Adder クラスの利用者は、加算時にオーバーフローを起こさない setValues()メソッドの引数を指定する必要がある。

```
final class Adder {
   private int a;
   private int b;
   public int getSum() {
      return a + b;
   }

   public void setValues(int a, int b) {
      this.a = a;
      this.b = b;
   }
}
```

getSum()メソッドには競合状態が存在する。たとえば、変数 a および b の値としてそれぞれ、0 および Integer.MAX_VALUE が代入された状態で、一方のスレッド（スレッド 1）が getSum()メソッドを呼び出すと同時に他方のスレッド（スレッド 2）が setValues(Integer.MAX_VALUE, 0)を呼び出す場合、getSum()メソッドは 0 または Integer.MAX_VALUE を返すかもしれないし、あるいは整数オーバーフローを引き起こして

結果の値がラップするかもしれない。スレッド 2 が変数 a に Integer.MAX_VALUE をセットした直後でかつ変数 b に 0 をセットする前にスレッド 1 が変数 a と b の値を読み取るとオーバーフローが発生することになる。

複合操作では複数の変数の読取りと書込みが行われるため、変数を volatile 宣言しても問題は解決しない。

2.3.9. 違反コード (AtomicInteger の加算)

以下の違反コード例では、変数 a および b がアトミックに操作可能な AtomicInteger と置き換えられている。

```
final class Adder {
  private final AtomicInteger a = new AtomicInteger();
  private final AtomicInteger b = new AtomicInteger();

  public int getSum() {
    return a.get() + b.get();
  }

  public void setValues(int a, int b) {
    this.a.set(a);
    this.b.set(b);
  }
}
```

二つの int フィールドを単純に AtomicInteger へ置き換えただけでは、競合状態は解消しない。複合操作である a.get() + b.get() はアトミックではない。

2.3.10. 適合コード (アトミックな加算)

以下の適合コードでは、setValues()メソッドおよび getSum()メソッドを同期させることで、アトミック性を確保している。

```
final class Adder {
  private int a;
  private int b;

  public synchronized int getSum() {
    return a + b;
  }

  public synchronized void setValues(int a, int b) {
    this.a = a;
    this.b = b;
  }
}
```

同期化したこれらのメソッド内での処理は、同じオブジェクトの固有ロックを使用する他の同期化メソッドに対してアトミックに実行される。したがって、たとえば getSum() メソッドに整数オーバーフローのチェックを追加しても競合状態は発生しない。

2.3.11. リスク評価

共有変数への操作がアトミックでない場合、予期せぬ結果が生ずることがありうる。たとえば、他ユーザに関する情報が取得され、情報漏えいにつながるおそれがある。

ガイドライン	深刻度	可能性	修正コスト	優先度	レベル
VNA02-J	中	中	中	P8	L2

2.3.12. 参考文献

[Bloch 2008]	Item 66: Synchronize access to shared mutable data
[Goetz 2006]	Section 2.3, “Locking”
[Gosling 2005]	Chapter 17, “Threads and Locks” Section 17.4.5, “Happens-Before Order” Section 17.4.3, “Programs and Program Order” Section 17.4.8, “Executions and Causality Requirements”
[Lea 2000a]	Section 2.2.7, The Java Memory Model Section 2.1.1.1, Objects and Locks
[MITRE 2010]	CWE ID 667, “Insufficient Locking” CWE ID 413, “Insufficient Resource Locking” CWE ID 366, “Race Condition within a Thread” CWE ID 567, “Unsynchronized Access to Shared Data”
[Sun 2009b]	Class AtomicInteger
[Sun 2008a]	Java Concurrency Tutorial

2.4. VNA03-J. アトミックなメソッドをまとめた呼出しがアトミックであると仮定しない

一貫したロックポリシーは、共有データが複数のスレッドに同時にアクセスされたり編集されたりしないことを保証する。二つ以上の操作を一つのアトミックな操作として実行する場合、固有ロックを使った同期化や java.util.concurrent ユーティリティを使用するなど、一貫したロックポリシーを適用する必要がある。

複数のオブジェクトが関係する不変項（オブジェクトが取りうる状態の制約条件）の操作において、個別にアトミックな操作に追加的なロックが必要ないと誤認することが考えられる。同様に、スレッドセーフな Collection クラスを使用していれば、そのコレクションの要素を含む不変項を操作するために明示的な同期処理が必要ないと誤認することもありうる。スレッドセーフなクラスは、それ自身の個々のメソッドのアトミック性しか保証できない。そのようなメソッド呼出しをまとめてメソッド化する場合、同期化を追加で行う必要がある。

たとえば、標準のスレッドセーフ API が、特定の人物の記録を Hashtable 上で検索し、対応する給与支払簿情報を更新するという二つの機能を有した単一のメソッドを提供していないケースを考えてみよう。この場合、この二つのメソッド呼出しは、アトミックに実行されねばならない。

列挙やイテレータを使用する場合にも、該当するコレクションオブジェクトを明示的に同期化（クライアントサイドロック）するか、private final 宣言したロックオブジェクトを用いて明示的に同期化する必要がある。

共有変数に対する複合操作もアトミックではない。詳細は、ガイドライン「VNA02-J. 共有変数への複合操作のアトミック性を確保する」を参照。

ガイドライン「VNA04-J. メソッドチェーン呼出しのアトミック性を確保する」では、このガイドラインの特種なケースについて記述している。

2.4.1. 違反コード (AtomicReference クラス)

以下の違反コードでは、BigInteger オブジェクトをスレッドセーフな AtomicReference オブジェクトでラップしている。

```
final class Adder {
  private final AtomicReference<BigInteger> first;
  private final AtomicReference<BigInteger> second;

  public Adder(BigInteger f, BigInteger s) {
    first = new AtomicReference<BigInteger>(f);
    second = new AtomicReference<BigInteger>(s);
  }

  public void update(BigInteger f, BigInteger s) { // スレッドセーフではないメソッド
    first.set(f);
    second.set(s);
  }

  public BigInteger add() { // スレッドセーフではないメソッド
    return first.get().add(second.get());
  }
}
```

AtomicReference はアトミックに更新することができるオブジェクト参照である。しかし、複数の AtomicReference を組み合わせた操作はアトミックではない。上記の違反コードでは、あるスレッドが add() を実行中に他のスレッドが update() を呼び出すかもしれない。そうすると、add() メソッドは、first の新しい値を second の古い値に加算してしまい、結果として間違った計算結果になるかもしれない。

2.4.2. 適合コード (synchronized メソッド)

以下の適合コードでは、アトミック性を保証するために update()メソッドおよび add()メソッドを synchronized 宣言をしている。

```
final class Adder {
  // ...

  public synchronized void update(BigInteger f, BigInteger s){
    first.set(f);
    second.set(s);
  }

  public synchronized BigInteger add() {
    return first.get().add(second.get());
  }
}
```

2.4.3. 違反コード (synchronizedList メソッド)

以下の違反コードではスレッドセーフではない java.util.ArrayList<E>コレクションを使用しており、Collections.synchronizedList を ArrayList のための同期ラッパーとして用いてい

る。ArrayList に対する繰返し処理にイテレータではなく配列を用いて、ConcurrentModificationException を回避している。

```
final class IPHolder {
  private final List<InetAddress> ips =
    Collections.synchronizedList(new ArrayList<InetAddress>());

  public void addAndPrintIPAddresses(InetAddress address) {
    ips.add(address);
    InetAddress[] addressCopy = (InetAddress[]) ips.toArray(new InetAddress[0]);
    // 配列 addressCopy を通して繰返し処理を行う ...
  }
}
```

コレクションメソッド add()および toArray()は、それぞれアトミックである。しかし、それらが連続して(たとえば、addAndPrintIPAddresses()メソッド内のように)呼ばれる場合、組み合わされた操作がアトミックである保証はない。addAndPrintIPAddresses()メソッドには競合状態が存在し、あるスレッドがリストへ追加し終わる前に他のスレッドがリストを変更できてしまう。結果的に、addressCopy 配列は想定よりも多くの IP アドレスを含んでしまうかもしれない。

2.4.4. 適合コード (synchronized ブロック)

使用されているリストを同期化することで競合状態を取り除くことができる。以下の適合コードでは、配列リストへの参照すべてを synchronized ブロック内にまとめている。

```
final class IPHolder {
  private final List<InetAddress> ips =
    Collections.synchronizedList(new ArrayList<InetAddress>());

  public void addAndPrintIPAddresses(InetAddress address) {
    synchronized (ips) {
      ips.add(address);
      InetAddress[] addressCopy = (InetAddress[]) ips.toArray(new InetAddress[0]);
      // 配列 addressCopy を通して繰返し処理を行う ...
    }
  }
}
```

この対応手法はクライアントサイドロックとも呼ばれている[Goetz 2006]。その理由は、他のクラスにアクセスしうるオブジェクトの固有ロックをクラスが取得するからである。ただし、クライアントサイドロックが常に適切な対応手法とは限らない。 詳細は、ガイドライン「LCK11-J. 一貫したロック方式が適用されないクラスには、クライアントサイドロックを利用しない」を参照。

なお、この適合コードはガイドライン「LCK04-J. アクセス可能なコレクションのコレクションビューを同期化に使用しない」には違反していない。コレクションビュー (synchronizedList)を同期しているが、基となるコレクションにはアクセスできず、いかなるコードからも変更できないからである。

2.4.5. 違反コード (synchronizedMap メソッド)

以下の違反コードでは、スレッドセーフではない KeyedCounter クラスを定義している。HashMap を synchronizedMap でラップしているが、インクリメント操作はアトミックではない[Lee 2009]。

```
final class KeyedCounter {
  private final Map<String, Integer> map =
    Collections.synchronizedMap(new HashMap<String, Integer>());

  public void increment(String key) {
    Integer old = map.get(key);
    int oldValue = (old == null) ? 0 : old.intValue();
    if (oldValue == Integer.MAX_VALUE) {
      throw new ArithmeticException("Out of range");
    }
    map.put(key, oldValue + 1);
  }

  public Integer getCount(String key) {
    return map.get(key);
  }
}
```

2.4.6. 適合コード (同期化)

以下の適合コードでは、アトミック性を確保するため、private 宣言されたロックオブジェクトを使用し increment()メソッドおよび getCount()メソッドの各ステートメントを同期化している。

```
final class KeyedCounter {
  private final Map<String, Integer> map = new HashMap<String, Integer>();
  private final Object lock = new Object();

  public void increment(String key) {
    synchronized (lock) {
      Integer old = map.get(key);
      int oldValue = (old == null) ? 0 : old.intValue();
      if (oldValue == Integer.MAX_VALUE) {
        throw new ArithmeticException("Out of range");
      }
      map.put(key, oldValue + 1);
    }
  }

  public Integer getCount(String key) {
    synchronized (lock) {
      return map.get(key);
    }
  }
}
```

この適合コードでは Collections.synchronizedMap()を使用していないが、ロックオブジェクトを用いマップへの操作を同期化している。ガイドライン「LCK04-J. アクセス可能なコレクションのコレクションビューを同期化に使用しない」では、synchronizedMap オブジェクトの同期化に関する詳細な情報を提供している。

2.4.7. 適合コード (ConcurrentHashMap クラス)

前述の適合コードは複数のスレッドで安全に使用することができるが、過度の同期化のため規模拡張は容易ではなく、結果的に競合やデッドロックを引き起こすことも考えられる。

以下の適合コードで使用される ConcurrentHashMap クラスは、アトミックな操作を行ういくつかのユーティリティメソッドを提供しており、スケーラビリティが重要な場合に適している[Lee 2009]。

```
final class KeyedCounter {
  private final ConcurrentMap<String, AtomicInteger> map =
    new ConcurrentHashMap<String, AtomicInteger>();

  public void increment(String key) {
    AtomicInteger value = new AtomicInteger();
    AtomicInteger old = map.putIfAbsent(key, value);
    if (old != null) {
      value = old;
    }

    if (value.get() == Integer.MAX_VALUE) {
      throw new ArithmeticException("Out of range");
    }

    value.incrementAndGet(); // 値をアトミックにインクリメントする
  }

  public Integer getCount(String key) {
    AtomicInteger value = map.get(key);
    return (value == null) ? null : value.get();
  }

  // 他のアクセッサ ...
}
```

Goetz らの著作である「Java 並行処理プログラミング」の 5.2.1 節「ConcurrentHashMap」によると[Goetz 2006]

> *ConcurrentHashMap とその他の並行処理コレクションクラスは、ConcurrentModificationException を投げないイテレータによりイテレーション中のロックを不要とするなど同期化コレクションクラスを更に改善している。*
> *ConcurrentHashMap が返すイテレータは、即時断念しない弱い一貫性を保つ。弱い一貫性を保つイテレータは、並行する更新を許し、イテレータが構築された時点のコレクション要素を対象として走査する。更に、保証はされないが、イテレータが構築された後に行われた更新をも反映しうる。*

なお、性能上の理由から ConcurrentHashMap.size()や ConcurrentHashMap.isEmpty()は、近似値を返すことが許されているため、これらの返り値を使って正確な結果を求めるコードにならないことに注意。

2.4.8. リスク評価

マルチスレッドアプリケーションにおいて、単一のアトミック動作として実行されるべき複数操作のアトミック性を確保できない場合、競合状態の発生につながりうる。

ガイドライン	深刻度	可能性	修正コスト	優先度	レベル
VNA03-J	低	中	中	P4	L3

2.4.9. 参考文献

[Goetz 2006]	Section 4.4.1, “Client-side Locking” Section 5.2.1, “ConcurrentHashMap”
[Lee 2009]	“Map & Compound Operation”
[Oaks 2004]	Section 8.2, “Synchronization and Collection Classes”
[Sun 2009c]	

2.5. VNA04-J. メソッドチェーン呼出しのアトミック性を確保する

メソッドチェーン（method chaining）は、単一のステートメントで同一オブジェクト上の複数メソッドの呼出しを可能にする便利な仕組みである。メソッドチェーンの実装は、this 参照を返す一連のメソッドで構成されている。チェーンの中で先行して呼び出されるメソッドの返り値に対して次のメソッドを呼び出し、連続したメソッド呼出しを可能としている。

メソッドチェーン内で使用されるメソッドがアトミックだとしても、それらのメソッドで構成されているチェーンはアトミックではない。したがって、ガイドライン「VNA03-J. アトミックなメソッドをまとめた呼出しがアトミックであると仮定しない」で示すように、呼出し元が適切なロックを提供しない限り、メソッドチェーン呼出しにつながるメソッドを並行呼出しすべきではない。

2.5.1. 違反コード

メソッドチェーンは、オブジェクトの構築とオプションフィールドのセットをまとめて行うのに適したデザインパターンである。メソッドチェーンを実装するクラスは、各々が this 参照を返すいくつかのセッターメソッドを提供する。しかし、複数スレッドに並行アクセスされた場合、あるスレッドは整合性を欠いた値がセットされた共有フィールドを参照するかもしれない。以下の違反コードは、スレッドセーフではない JavaBeans パターンを示している。

```
final class USCurrency {
  // 両替の単位(オプションフィールド)
  private int quarters = 0;
  private int dimes = 0;
  private int nickels = 0;
  private int pennies = 0;

  public USCurrency() {}
  // セッターメソッド
  public USCurrency setQuarters(int quantity) {
    quarters = quantity;
    return this;
  }
  public USCurrency setDimes(int quantity) {
    dimes = quantity;
    return this;
  }
  public USCurrency setNickels(int quantity) {
    nickels = quantity;
    return this;
  }
  public USCurrency setPennies(int quantity) {
    pennies = quantity;
    return this;
  }
}

// クライアント側ソースコード
private final USCurrency currency = new USCurrency();
// ...

new Thread(new Runnable() {
  @Override public void run() {
    currency.setQuarters(1).setDimes(1);
  }
}).start();

new Thread(new Runnable() {
  @Override public void run() {
    currency.setQuarters(2).setDimes(2);
  }
}).start();
```

JavaBeans パターンでは、引数指定のないコンストラクタと一連のセッターメソッドを使用してオブジェクトを構築する。このパターンはスレッドセーフではなく、オブジェクトが同時変更された場合にオブジェクトが整合性を欠いた状態になりうる。上記の違反コードでは、クライアントは USCurrency オブジェクトを構築し、メソッドチェーンを使用する二つのスレッドを開始して USCurrency のオプションフィールドに値をセットしている。

しかし、quarters が 2 で dimes が 1 や、quarters が 1 で dimes が 2 のような整合性を欠いた状態の USCurrency インスンタスとなる可能性がある。

2.5.2. 適合コード

以下の適合コードでは、オブジェクト構築におけるスレッドの安全性およびアトミック性を確保するために、Bloch [Bloch 2008] の Builder パターンの変形型を使用している。

```
final class USCurrency {
  private final int quarters;
  private final int dimes;
  private final int nickels;
  private final int pennies;

  public USCurrency(Builder builder) {
    this.quarters = builder.quarters;
    this.dimes = builder.dimes;
    this.nickels = builder.nickels;
    this.pennies = builder.pennies;
  }

  // static なクラスメンバー
  public static class Builder {
    private int quarters = 0;
    private int dimes = 0;
    private int nickels = 0;
    private int pennies = 0;

    public static Builder newInstance() {
      return new Builder();
    }

    private Builder() {}
    // セッターメソッド
    public Builder setQuarters(int quantity) {
      this.quarters = quantity;
      return this;
    }
    public Builder setDimes(int quantity) {
      this.dimes = quantity;
      return this;
    }
    public Builder setNickels(int quantity) {
      this.nickels = quantity;
      return this;
    }
    public Builder setPennies(int quantity) {
      this.pennies = quantity;
      return this;
    }
```

```
      public USCurrency build() {
         return new USCurrency(this);
      }
   }
}

// クライアント側ソースコード:
private volatile USCurrency currency;
// ...

new Thread(new Runnable() {
   @Override public void run() {
      currency = USCurrency.Builder.newInstance().setQuarters(1).setDimes(1).build();
   }
}).start();

new Thread(new Runnable() {
   @Override public void run() {
      currency = USCurrency.Builder.newInstance().setQuarters(2).setDimes(2).build();
   }
}).start();
```

ファクトリメソッドである Builder.newInstance()を実行し Builder インスタンスを取得している。オプションパラメータは、ビルダーのセッターメソッドを使用して設定される。オブジェクトの構築は build()メソッドの呼出しで完結する。このデザインパターンにより、USCurrency クラスは不変となり、結果的にスレッドセーフとなる。

currency フィールドには新規の不変オブジェクト(immutable object)が割り当てられるため final 宣言できないことに注意。しかし、currency フィールドは、ガイドライン「VNA01-J. 不変オブジェクトへの共有参照の可視性を確保する」に従い volatile 宣言されている。

2.5.3. リスク評価

マルチスレッド環境で追加的なロックを行わずにメソッドチェーン呼出しを使用すると予期せぬプログラムの動作につながる。

ガイドライン	深刻度	可能性	修正コスト	優先度	レベル
VNA04-J	低	中	中	P4	L3

2.5.4. 参考文献

[Bloch 2008]	Item 2:“Consider a builder when faced with many constructor parameters”
[Sun 2009b]	

2.6. VNA05-J. 64 ビット値の読み書きのアトミック性を確保する

*Java 言語仕様*は、64 ビットの long 型および double 型の値を二つの 32 ビット値として扱うことを認めている。たとえば、64 ビット値の書込み操作は、二つの独立した 32 ビット値の演算として実行されるかもしれない。

*Java 言語仕様*の 17.7 節「double と long の非アトミックな扱い」によれば [Gosling 2005]

> *... こうした振る舞いは実装依存である。つまり、Java 仮想マシンは long 値や double 値の書込みをアトミックな動作として実行するか、あるいは、二つの動作として実行するかを自由に決定することが許されている。プログラミング言語 Java メモリモデルでは、非 volatile な long 値や double 値への単一の書込みは、それぞれ 32 ビットずつの二つの書込みとして扱われる。結果的に、ある 64 ビット値の書込みの最初の 32 ビットと、他の書込みによる次の 32 ビットの組み合わせをスレッドが参照しうる。*

これが原因で、スレッドセーフであることが要求されるコードで確定していない値が読み取られてしまうかもしれない。

2.6.1. 違反コード

以下の違反コードでは、あるスレッドが assignValue()メソッドを繰り返し呼び出し、別のスレッドが printLong()メソッドを繰り返し呼び出すとき、printLong()メソッドでは 0 でも引数 j の値でもない i の値を出力するかもしれない。

```
class LongContainer {
  private long i = 0;

  void assignValue(long j) {
    i = j;
  }

  void printLong() {
    System.out.println("i = " + i);
  }
}
```

i が double 型で宣言される場合も、同様の問題が発生しうる。

2.6.2. 適合コード (volatile 変数)

以下の適合コードでは、i を volatile 宣言している。long 型および double 型の volatile 変数値の読み書きは、常にアトミックである。

```
class LongContainer {
   private volatile long i = 0;
   void assignValue(long j) {
   i = j;
   }
   void printLong() {
      System.out.println("i = " + i);
   }
}
```

なお、assignValue()メソッドへ渡す引数が、volatile 変数値であるか、あるいは明示的に渡された整数値のいずれかであることは重要である。さもないと、引数の読取りが、脆弱性につながりうる。

volatile 修飾子のセマンティックスは、値のインクリメントのような読取り、変更、書込みを伴う一連の複合操作のアトミック性を保証するものではない。ガイドライン「VNA02-J. 共有変数への複合操作のアトミック性を確保する」を参照。

2.6.3. 例外

VNA05-EX1: 64 ビットの long 型および double 型の値へのすべての読み書き操作が同期化されたブロック内で実行される場合、読み書きのアトミック性は保証される。ただし、クラス内の同期化されていないメソッドが 64 ビット値を公開せず、その値が他のコードからアクセスできない (直接的あるいは間接的に)ことが必要な条件である。 (詳細については、ガイドライン「VNA02-J. 共有変数への複合操作のアトミック性を確保する」を参照。)

VNA05-EX2: このガイドラインは、64 ビットの long 型および double 型の値がアトミックに読み書きされることが保証されているシステムでは無視できる。

2.6.4. リスク評価

マルチスレッドアプリケーションでは 64 ビット値への操作のアトミック性が確保されていないと、確定していない値を読み書きしてしまう場合がある。仕様上の必須要件ではないが、多くの JVM は、64 ビット値の読み書きをアトミックに行う。

ガイドライン	深刻度	可能性	修正コスト	優先度	レベル
VNA05-J	低	低	中	P2	L3

2.6.5. 参考文献

[Goetz 2004c]	
[Goetz 2006]	Section 3.1.2, “Non-Atomic 64-Bit Operations”
[Gosling 2005]	Section 17.7, “Non-Atomic Treatment of double and long”
[MITRE 2010]	CWE ID 667, “Insufficient Locking”

2.7. VNA06-J. オブジェクトへの参照を volatile 宣言することでメンバーの可視性が保証されると想定しない [1]

Java 言語仕様の 8.3.1.4 節「volatile 宣言されたフィールド」[Gosling 2005]によれば

> *Java メモリモデル(§17)は、volatile 宣言されたフィールドに対して、すべてのスレッドが整合性ある変数の値を参照することを確実にする。*

この約束が当てはまるのは、プリミティブ型のフィールドと不変なメンバーオブジェクトのみであることに注意。スレッドセーフではない可変オブジェクトへの参照が volatile 宣言されていても、可視性は保証されない。このようなオブジェクトのメンバフィールドに対する書込みが他のスレッドによってなされたとしても可視とならない。オブジェクトが不変でない限り、オブジェクトを volatile 宣言しても、同期化を伴わなければ、オブジェクトの状態の可視性は確保できない。オブジェクトが可変でありかつスレッドセーフでない場合、構築が不完全なオブジェクト、あるいは(一時的に)矛盾した状態のオブジェクトが読み取られるかもしれない[Goetz 2006c]。

厳密には、オブジェクトが不変でないと安全に使えないというわけではない。メンバーオブジェクトの設計がスレッドセーフであると断定できる場合、オブジェクトの参照を保持するフィールドを volatile 宣言してもよい。しかし、各要素を volatile 宣言するというこのアプローチは保守性の低下を招くので避けるべきである。

2.7.1. 違反コード (配列)

以下の違反コードでは、volatile 宣言された配列オブジェクトを示している。

```
final class Foo {
  private volatile int[] arr = new int[20];

  public int getFirst() {
    return arr[0];
  }

  public void setFirst(int n) {
    arr[0] = n;
  }

  // ...
}
```

[1] 2011 年 7 月現在、このガイドラインは削除されている。

volatile キーワードは単に配列オブジェクトの参照を可視化するだけであり、配列内に格納された実データには影響を及ぼさないため、あるスレッドが setFirst()メソッドを呼び出して配列要素にセットした値は、getFirst()メソッドを呼ぶ他のスレッドからは可視ではないかもしれない。

この問題は、setFirst()メソッドを呼ぶスレッドと、getFirst()メソッドを呼ぶスレッドの間に事前発生関係が存在しないために生じる。事前発生関係は、ある volatile 変数に書き込むスレッドと、書込み後にその変数を読み取るスレッドの間に存在する。しかし、上記のコードは、volatile 変数の読み書きのどちらも実行していない。

2.7.2. 適合コード (AtomicIntegerArray クラス)

配列要素への書込みがアトミックに行われ、かつ書き込んだ値が他のスレッドに可視とするため、以下の適合コードでは、java.util.concurrent.atomic パッケージで定義される AtomicIntegerArray クラスを使用している。

```
final class Foo {
  private final AtomicIntegerArray atomicArray = new AtomicIntegerArray(20);

  public int getFirst() {
    return atomicArray.get(0);
  }

  public void setFirst(int n) {
    atomicArray.set(0, 10);
  }

  // ...
}
```

AtomicIntegerArray は、atomicArray.set()メソッドを呼ぶスレッドと、次に atomicArray.get()メソッドを呼ぶスレッドの間の事前発生関係を保証する。

2.7.3. 適合コード (同期化)

アクセッサメソッドへのアクセスを同期させることで、volatile 宣言されている配列の volatile ではない要素への操作結果の可視性を確保することができる。以下のコードは、volatile 宣言されていない配列オブジェクトの参照を使用しているがスレッドセーフである。

```
final class Foo {
  private int[] arr = new int[20];
  public synchronized int getFirst() {
    return arr[0];
  }

  public synchronized void setFirst(int n) {
    arr[0] = n;
  }
}
```

同期化により、setFirst()メソッドを呼ぶスレッドと、後続の getFirst()メソッドを呼ぶスレッド間の事前発生関係が確立され可視性が保証される。

2.7.4. 違反コード (可変オブジェクト)

以下の違反コードでは、Properties のインスタンスフィールドを volatile 宣言している。Properties オブジェクトのインスタンスは put()メソッドにより変更できるため、フィールド properties は可変となる。

```
final class Foo {
  private volatile Properties properties;

  public Foo() {
    properties = new Properties();
    // properties への値のロード
  }

  public String get(String s) {
    return properties.getProperty(s);
  }

  public void put(String key, String value) {
    // 挿入前の値の検証
    if (!value.matches("[¥¥w]*")) {
      throw new IllegalArgumentException();
    }
    properties.setProperty(key, value);
  }
```

put()メソッド内の操作が Properties オブジェクトの状態を変更するため、get()メソッドと put()メソッドを交互に呼び出すと Properties オブジェクトから整合性を欠いた値を取得し

てしまうかもしれない。オブジェクトを volatile 宣言しても、このようなデータ競合は解決されない。

なお、put()メソッドには、検証ロジックが含まれているが、検証の対象は共有された Properties インスタンスではなく不変の引数 value なので TOCTOU 競合は存在しない。

2.7.5. 違反コード (volatile 変数の読取り，同期書込み)

以下の違反コードでは、Goetz がまとめた volatile 変数の読取りと同期された書込みを組み合わせて使用する手法を適用しようとしている[Goetz 2006c]。properties フィールドは、その読み書きを同期させるために volatile 宣言されている。put()メソッドも、そのステートメントがアトミックに実行されるように同期化されている。

```
final class Foo {
  private volatile Properties properties;

  public Foo() {
    properties = new Properties();
    // properties への値のロード
  }

  public String get(String s) {
    return properties.getProperty(s);
  }

  public synchronized void put(String key, String value) {
    // 挿入前の値の検証
    if (!value.matches("[¥¥w]*")) {
      throw new IllegalArgumentException();
    }
    properties.setProperty(key, value);
  }
}
```

この volatile 変数の読取りと同期された書込みを組み合わせた手法は、インクリメントのような複合操作のアトミック性を同期化により確保しつつ、より高速なアトミックな読取りを実現する。しかし、volatile 宣言されたオブジェクト参照の可視性はオブジェクトのメンバーには拡張されないため、この手法は可変オブジェクトには有効ではない。したがって、properties への書込みと読込みの間に事前発生関係は存在しない。

この手法は、ガイドライン「VNA02-J. 共有変数への複合操作のアトミック性を確保する」でも触れられている。

2.7.6. 適合コード (同期化)

以下の適合コードでは、可視性を保証するためにメソッドを同期化している。

```
final class Foo {
  private final Properties properties;

  public Foo() {
    properties = new Properties();
    // properties への値のロード
  }

  public synchronized String get(String s) {
    return properties.getProperty(s);
  }

  public synchronized void put(String key, String value) {
    // 挿入前の値の検証
    if (!value.matches("[¥¥w]*")) {
      throw new IllegalArgumentException();
    }
    properties.setProperty(key, value);
  }
}
```

メソッドが同期化されているので、properties フィールドは volatile でなくてもよい。properties フィールドは final 宣言されているため、その参照は不完全な初期化状態で公開されない(ガイドライン「TSM03-J. 初期化が不完全なオブジェクトを公開しない」を参照)。

2.7.7. 違反コード (可変サブオブジェクト)

以下の違反コードでは、volatile 宣言された format フィールドを用いて、可変オブジェクトである java.text.DateFormat への参照を保持している。

```
final class DateHandler {
  private static volatile DateFormat format =
    DateFormat.getDateInstance(DateFormat.MEDIUM);

  public static Date parse(String str) throws ParseException {
    return format.parse(str);
  }
}
```

DateFormat クラスはスレッドセーフではないので[Sun 2009c]、parse()メソッドは引数 str に対応しない Date の値を返すかもしれない。

2.7.8. 適合コード (呼出し毎のインスタンス化、防御的コピー)

以下の適合コードでは、parse()メソッドが呼び出されるたびに新規に生成した DateFormat インスタンスを返す [Sun 2009c]。

```
final class DateHandler {
  public static Date parse(String str) throws ParseException {
    return DateFormat.getDateInstance(DateFormat.MEDIUM).parse(str);
  }
}
```

上記の解決方法では、クラス内に可変な状態が含まれていないので、ガイドライン「OBJ05-J. Defensively copy private mutable class members before returning their references [2]」には違反しない。

2.7.9. 適合コード (同期化)

以下の適合コードでは、parse()メソッド内のステートメントを同期化しており、DateHandler はスレッドセーフである [Sun 2009c]。

```
final class DateHandler {
  private static DateFormat format=
    DateFormat.getDateInstance(DateFormat.MEDIUM);

  public static Date parse(String str) throws ParseException {
    synchronized (format) {
      return format.parse(str);
    }
  }
}
```

2.7.10. 適合コード (ThreadLocal ストレージ)

以下の適合コードでは、ThreadLocal オブジェクトを使用して、スレッド毎に個別の DateFormat インスタンスを生成している。

```
final class DateHandler {
  private static final ThreadLocal<DateFormat> format =
    new ThreadLocal<DateFormat>() {
    h@Override protected DateFormat initialValue() {
      return DateFormat.getDateInstance(DateFormat.MEDIUM);
    }
  };
  // ...
}
```

2 このガイドラインは、https://www.securecoding.cert.org/confluence/display/java/に記述されている。

2.7.11. リスク評価

フィールドを volatile 宣言すれば参照されるオブジェクトのメンバーが可視になるという間違った思い込みをすると、スレッドは最新ではない陳腐化した値を読み取るおそれがある。

ガイドライン	深刻度	可能性	修正コスト	優先度	レベル
VNA06-J	中	中	中	P8	L2

2.7.12. 参考文献

[Goetz 2006c]	Pattern #2:“one-time safe publication”
[Gosling 2005]	
[Miller 2009]	Mutable Statics
[Sun 2009c]	Class java.text.DateFormat

3. ロック(LCK)ガイドライン

3.1. LCK00-J. 信頼できないコードから使用されるクラスの同期化には private final ロックオブジェクトを使用する

synchronized キーワードを使うと、実行中のスレッドがロックを保持している間は他のスレッドがロックを取得できないような、排他的なロックを取得することができる。共有された可変変数へのアクセスを同期化するには、メソッド同期およびブロック同期の二通りの方法がある。

synchronized 宣言したメソッドは、synchronized ブロックで this 参照を同期化しているコードと同様にオブジェクトの固有ロックを使用する。しかし、不適切に同期化されたコードは、競合状態やデッドロックを引き起こしやすい。攻撃者は、アクセス可能なクラスの固有ロックを取得して、これを無期限に保持することで、競合状態やデッドロックを発生させ、サービスを妨害することが可能である。

この脆弱性は、クラス内で private final 宣言された java.lang.Object（Object クラス）を使用することにより防ぐことができる。使用されるオブジェクト（Object クラスのインスタンス）は、クラスのメソッド内の synchronized ブロックでロックを行う目的で明示的に宣言される必要がある。この場合同期で使われる固有ロックは、private 宣言された Object クラスのインスタンスである。したがって、クラスメソッドと他の悪意のあるクラスのメソッドの間にロック競合は生じない。Bloch は、この手法を「private 宣言したロックオブジェクト」と呼んでいる [Bloch 2001]。

static 宣言にも同様の問題が潜在的に存在する。静的メソッドを synchronized 宣言すると、そのメソッド内のステートメントが実行される前に Class オブジェクト（java.lang.Class）の固有ロックが取得され、メソッドが完了した時点でリリースされる。クラスあるいはサブクラスのオブジェクトにアクセス可能な信用できないコードもクラスのロックを獲得するために getClass()メソッドを使用できてしまう。対策として、static 宣言されたデータを private static final 宣言した Object クラスでロックして保護できる。また、クラスのアクセスを package-private に限定することで、信頼できない呼出しに対する保護を更に強化することもできる。

この「private static final 宣言したロックオブジェクト」を使用する手法は、継承用に設計されたクラスにも適している。スーパークラスのスレッドが自身の固有ロックを必要とする場合、サブクラスのスレッドはその操作を妨害することができてしまう。たとえば、サブ

クラスが無関係な操作にスーパークラスオブジェクトの固有ロックを使用した場合、深刻なロック競合およびデッドロックを引き起こすかもしれない。スーパークラスのロック方式をサブクラスのそれから分離することにより、双方がロックオブジェクトを共有する可能性はなくなる。また、複数のロックオブジェクトを利用することで、相互干渉を最小限にした細かい粒度のロックが可能となり、アプリケーションの性能改善につながる。

あるクラスが信頼できないコードによる以下の操作を阻止できない場合、そのクラスのオブジェクト自身の固有ロックではなく private final 宣言されたロックオブジェクトを使用すべきである。

- 該当クラスあるいはそのスーパークラスをサブクラス化する(信頼できるコードは、クラスのサブクラス化が許されている)
- 該当クラス、そのスーパークラスあるいはサブクラスのオブジェクトを生成する
- 該当クラス、そのスーパークラスあるいはサブクラスのオブジェクトにアクセスするか取得する

あるクラスが共有データへの操作を同期化するために private final 宣言されたロックを使用する場合、そのサブクラスも同様に private final 宣言されたロックを使用しなければならない。また、スーパークラスがロックポリシーを明文化せずにスーパークラス自身の固有ロックを同期化に利用する場合でも、サブクラスのロックポリシーが明文化されていない限りは、サブクラスでの同期化に同様のロック手法を実装しないかもしれない。スーパークラスのロックポリシーでクライアントサイドロックにも対応する旨が文書化されている場合、サブクラスの実装においてはオブジェクトの固有ロックか private 宣言されたロックのいずれかを選択できる。いずれが選ばれるにせよ、サブクラス自体のロックポリシーが文書化されなければならない。関連情報は、ガイドライン「TSM00-J. スレッドセーフなメソッドを、スレッドセーフでないメソッドでオーバーライドしない」を参照。

前述の制限をまったく満たしていない場合、オブジェクトの固有ロックは信頼できない。一方、これらの制限を満たしている場合、オブジェクトは private final 宣言されたロックオブジェクトの使用によるセキュリティ上のメリットはあまり得られないので、オブジェクトの固有ロックを使用して同期化してもよい。しかし、メソッドが非アトミック（アトミック性が要求されない）操作から構成され、それらが同期化を必要としないかあるいは複数のロックオブジェクトを使用して更にきめ細かい排他制御が実現できる場合には、private final 宣言したロックオブジェクトによるブロック同期を行うことが望ましい。非アトミック操作は、同期化が要求される部分から分離して、synchronized ブロックの外部で実行することが可能である。この理由に加えて保守性の観点からも、private final 宣言されたロックオブジェクトによるブロック同期が一般に推奨されている。

3.1.1. 違反コード (synchronized メソッド)

以下の違反コードでは、SomeObject クラスのインスタンスが信頼できないコードから利用できる。

```
public class SomeObject {
  public synchronized void changeValue() { // オブジェクトのモニタをロックする
    // ...
  }
}

// 信用できないコード
SomeObject someObject = new SomeObject();
synchronized (someObject) {
  while (true) {
    Thread.sleep(Integer.MAX_VALUE); // 無期限に someObject を遅延させる
  }
}
```

信頼できないコードは、synchronized 宣言された changeValue()メソッドの呼出しに必要となるオブジェクトの固有ロックを取得した後、そのロックをリリースせずに他からのロックの取得（changeValue()の呼出し）を妨害するため、処理を無期限に遅延している。信頼できないコードでは、攻撃目的で故意にガイドライン「LCK09-J. ロックを保持したままブロックする操作を実行しない」に違反している。

3.1.2. 違反コード (final 宣言されていない public ロックオブジェクト)

以下の違反コードでは、SomeObject の固有ロックではなく、final 宣言されていない public オブジェクトをロックしている。

```
public class SomeObject {
  public Object lock = new Object();

  public void changeValue() {
    synchronized (lock) {
      // ...
    }
  }
}
```

しかし、信頼できないコードがアクセス可能なロックオブジェクトの値を変更し、同期を妨害することが可能である。

3.1.3. 違反コード (final 宣言されていない、変更可能な private ロックオブジェクト)

以下の違反コードでは、private 宣言のみで final 宣言されていないロックオブジェクトで同期を行っている。

```
public class SomeObject {
  private volatile Object lock = new Object();

  public void changeValue() {
    synchronized (lock) {
      // ...
    }
  }

  public void setLock(Object lockValue) {
    lock = lockValue;
  }
}
```

どんなスレッドでも、setLock()メソッドのようなアクセッサを介して、ロックオブジェクトを変更できてしまう。ロックオブジェクトが変更されると、本来は二つのスレッドが同一オブジェクトをロックすべきところ、それぞれ異なるオブジェクトをロックしてしまい、結果的に二つのクリティカルセクションが安全に実行されない可能性がある。たとえば、一つのスレッドがそのクリティカルセクションを処理中に、該当するロックオブジェクトが変更されてしまう場合、他のスレッドは変更された異なるオブジェクトをロックするだろう。

ロックを変更するメソッドを提供しないクラスは、信頼できない操作から安全であるが、プログラマによる不注意な保守作業を招きやすい。アクセッサメソッドを排除することは、保守性の観点から推奨される解決方法ではなく、また、アクセッサメソッドが必要とされる他の理由もあるだろう。

3.1.4. 違反コード (public final ロックオブジェクト)

以下の違反コードでは、public final 宣言したロックオブジェクトを使用している。

```
public class SomeObject {
  public final Object lock = new Object();
  public void changeValue() {
    synchronized (lock) {
      // ...
    }
  }
}
// 信頼できないコード
SomeObject someObject = new SomeObject();
someObject.lock.wait()
```

SomeObject クラスのインスタンスの生成、あるいは、既に生成済のインスタンスにアクセスできる信頼できないコードが、パブリックにアクセス可能な lock の wait()メソッドを呼び出すと changeValue()メソッドのロックが直ちにリリースされてしまう。

また、changeVlaue()メソッド中で lock.wait()メソッドを呼び出すが条件述語によるチェックを行わない場合、意図しないタイミングで wait()メソッドを終了させるような悪意のある通知に脆弱となる。詳細は、ガイドライン「THI03-J. wait()および await()メソッドは、常にループ内部で呼び出す」を参照。

3.1.5. 適合コード (private final ロックオブジェクト)

信頼できないコードとやり取りするスレッドセーフな public クラスは、private final 宣言されたロックオブジェクトを使用しなければならない。固有ロックを使用しているクラスは、ロックオブジェクトを用いたブロック同期を実装するために修正が必要である。以下の適合コードでは、changeValue()メソッドの呼出しで、クラスの外からはアクセスできない private final 宣言された Object クラスのインスタンスへのロックを取得している。

```
public class SomeObject {
  private final Object lock = new Object(); // private final 宣言されたロックオブジェクト

  public void changeValue() {
    synchronized (lock) { // Locks on the private Object
      // ...
    }
  }
}
```

private final 宣言されたロックオブジェクトは、ブロック同期でのみ使用することができる。ブロック同期はメソッド同期よりも推奨される。なぜならば、同期を必要としない操作は同期化されたブロック外に移動しやすく、ロック競合および処理がブロックされる機会を減少できるからである。final 宣言されたフィールドの可視性は確保されるので、lock を volatile 宣言する必要がない点に注意。多様な粒度のロック要件に応える方法としては、セッターメソッドを使用してロックを都度変更するのではなく、private final 宣言した複数のロックオブジェクトを使用することが推奨される。

3.1.6. 違反コード (static)

以下の違反コードでは、SomeObject の Class オブジェクトは、信頼できないコードからアクセス可能である。

```
public class SomeObject {
  // changeValue は、Class オブジェクトのモニタをロックする
  public static synchronized void changeValue() {
    // ...
  }
}

// 信用できないコード
synchronized (SomeObject.class) {
  while (true) {
    Thread.sleep(Integer.MAX_VALUE); // 無期限に someObject を遅延させる
  }
}
```

信頼できないコードは、synchronized 宣言された changeValue()メソッドで使われる Class オブジェクトの固有ロックの取得を試み、これに成功すると他の changeValue()メソッド呼出しがロックを取得することを妨害するための無期限ループに入る。

適合コードを作成するには、ガイドライン「LCK05-J. 信頼できないコードが変更できる static フィールドへのアクセスは同期する」に準拠しなければならない。しかし、上記の信頼できないコードで、攻撃者は故意にガイドライン「LCK09-J. ロックを保持したままブロックする操作を実行しない」に違反しており、ブロッキングの危険性は残ったままである。

3.1.7. 適合コード (static)

オブジェクトの固有ロックを用いており、かつ、信頼できないコードとやり取りを行うスレッドセーフな public クラスは、private final static 宣言されたロックオブジェクト使用してブロック同期を行うよう修正する。

```
public class SomeObject {
  private static final Object lock = new Object(); // private final ロックオブジェクト

  public static void changeValue() {
    synchronized (lock) { // private 宣言された Object へのロック
      // ...
    }
  }
}
```

上記の適合コードでは、changeValue()メソッドは、信頼できないコードからアクセスできない static private 宣言された Object クラスのロックを取得している。

3.1.8. 例外

LCK00-EX1: 下記条件をすべて満たすクラスはこのガイドラインに適合しなくてもよい。

- 呼出し元（クライアント）に対し、信頼できないコードにクラスのオブジェクトを渡すことを禁止する旨が十分に文書化されている。
- このガイドラインに直接的あるいは間接的に違反している信頼できない他のクラスのオブジェクトのメソッド呼出しを行わない。
- 同期ポリシーが適切に文書化されている。

下記条件をすべて満たすクライアントはこのガイドラインに適合しないクラスを使用してもよい。

- クラスのオブジェクトを信頼できないコードに渡さない。
- このガイドラインに直接的あるいは間接的に違反する信頼できないクラスを使用しない。

LCK00-EX2: スーパークラスでクライアントサイドロックとクラスのオブジェクト固有ロックによる同期化に対応する旨が文書化されており、サブクラスでも同様にクライアントサイドロックに対応し、その旨がサブクラスのポリシーとして文書化されている。

LCK00-EX3: package-private なクラスは、そのアクセス制限によって信頼できない呼出し元から保護されるので、このガイドラインに適合しなくてもよい。しかし、同一パッケージ内の信頼できるコードがロックオブジェクトを不注意に再使用あるいは変更することがないよう明示的に文書化されるべきである。

3.1.9. リスク評価

信頼できないコードがクラスのオブジェクトにアクセス可能な場合、サービス妨害攻撃の危険性がある。

ガイドライン	深刻度	可能性	修正コスト	優先度	レベル
LCK00-J	低	中	中	P4	L3

3.1.10. 参考文献

[Bloch 2001]	Item 52:“Document Thread Safety”

3.2. LCK01-J. 再利用されるオブジェクトを同期化に使用しない

同期プリミティブの誤用は、並行処理に関する問題の温床となっている。再利用されるオブジェクトによる同期化は、デッドロックや予期しない結果につながる場合がある。

3.2.1. 違反コード (クラス Boolean ロックオブジェクト)

以下の違反コードでは、Boolean クラスのロックオブジェクトで同期化している。

```
private final Boolean initialized = Boolean.FALSE;

public void doSomething() {
    synchronized (initialized) {
        // ...
   }
}
```

Boolean は、二値(true または false)しかとることができないため、ロック目的で使用するには適さない。同じ値を持つすべての Boolean リテラルは、JVM 環境下でただ一つの Boolean クラスのインスタンス（true または false）を共有する。上記の例では、initialized が false 値に対応するインスタンスを参照している。他のコードが不注意に Boolean リテラルの false 値による同期を行うと、ロックインスタンスは再利用され、システムは応答しなくなるかデッドロックに陥るかもしれない。

3.2.2. 違反コード (ボクシングしたプリミティブ型変数)

以下の違反コードでは、int 型変数をボクシング（boxing）した Integer オブジェクトをロックしている。

```
int lock = 0;
private final Integer Lock = lock; // ボクシングしたプリミティブな Lock は共有される

public void doSomething() {
   synchronized (Lock) {
      // ...
   }
}
```

プリミティブ変数をボクシングしたオブジェクトは、整数値の範囲内で同じインスタンスが再利用される場合があるため、Boolean と同様の理由でロックオブジェクトとして使用するには問題がある。プリミティブ変数の値が 1 バイトで表現可能な場合、その値に該当するラッパーオブジェクトのインスタンスが再利用される。プリミティブ型 int をボクシングしたクラス Integer のオブジェクトの使用は安全ではないが、new オペレータを使用し

て構築したクラス Integer のオブジェクトインスタンスは常に一意であり再利用されない。一般的にボクシングした値を含んでいるオブジェクトへのロックを保持することは安全面で問題がある。

3.2.3. 適合コード (クラス Integer ロックオブジェクト)

以下の適合コードでは、ボクシングしていない Integer のロックを推奨している。doSomething()メソッドでは、Integer のインスタンスである Lock の固有ロックを使用して同期化している。

```
int lock = 0;
private final Integer Lock = new Integer(lock);

public void doSomething() {
  synchronized (Lock) {
    // ...
  }
}
```

明示的にインスタンスを構築した場合、Integer オブジェクトの参照は一意であり、その固有ロックも他の Integer オブジェクトや同じ整数値を保持するボクシングされたものと共有されることはない。これは解決方法の一つではあるが、ボクシングされた整数をロックオブジェクトとして利用することの問題点を開発者が正しく認識していない場合、コードの保守上の問題が発生するおそれがある。より望ましい解決方法は、前述のセクション「3.1.5. 適合コード (private final ロックオブジェクト)」で述べられているような private final 宣言されたロックオブジェクトでの同期化である。

3.2.4. 違反コード (intern した String オブジェクト)

以下の違反コードでは、intern した String オブジェクトをロックしている。

```
private final String lock = new String("LOCK").intern();

public void doSomething() {
  synchronized (lock) {
    // ...
  }
}
```

Java API クラス java.lang.String の文書によれば[Sun 2009c]

> *intern()メソッドが呼び出されたときに、equals(Object)メソッドによってこの String オブジェクトに等しいと判定される文字列がプールにすでに存在した場合は、プール内の該*

当する文字列が返される。そうでない場合は、この String オブジェクトがプールに追加され、この String オブジェクトへの参照が返される。

したがって、intern された String オブジェクトは、JVM においてはグローバル変数のように振る舞う。上記の違反コードで例示したように、このクラスのすべてのインスタンスがそれ自身の lock フィールドを保持していても、各フィールドは共通の String 定数を参照している。String 定数へのロックに関しては、Boolean 定数へのロックと同じ問題が存在する。

更に、他パッケージの信頼できないコードからのアクセスが可能な場合、この脆弱性を悪用される可能性がある。(詳細は、ガイドライン「LCK00-J. 信頼できないコードから使用されるクラスの同期化には private final ロックオブジェクトを使用する」を参照。)

3.2.5. 違反コード (String リテラル)

以下の違反コードでは、final 宣言された String リテラルをロックしている。

```
// This bug was found in jetty-6.1.3 BoundedThreadPool
private final String lock = "LOCK";

// ...
synchronized (lock) {
// ...
}
// ...
```

String リテラルは定数であり、intern される（プール内で共有される）。したがって、上記の違反コードでは、前節の違反コードと同じ問題を抱えている。

3.2.6. 適合コード (String インスタンス)

以下の適合コードでは、intern されていない String インスタンスをロックしている。

```
private final String lock = new String("LOCK");

public void doSomething() {
   synchronized (lock) {
      // ...
   }
}
```

String インスタンスは String リテラルとは異なる。String インスタンスは一意の参照を持ち、他の String インスタンスあるいはリテラルとは共有されないそれ自身の固有ロックを持つ。より良いアプローチは、次に示す適合コードで説明するように、private final 宣言されたロックオブジェクトで同期化することである。

3.2.7. 適合コード (private final ロックオブジェクト)

以下の適合コードでは、private final 宣言されたロックオブジェクトで同期化している。この方法は、java.lang.Object インスタンスが有用なケースの一つである。

```
private final Object lock = new Object();

public void doSomething() {
  synchronized (lock) {
    // ...
  }
}
```

Object クラスをロックとして使用することの詳細は、ガイドライン「LCK00-J. 信頼できないコードから使用されるクラスの同期化には private final ロックオブジェクトを使用する」を参照。

3.2.8. リスク評価

並行処理に関する脆弱性の多くは、不適切なロックオブジェクトを使用することから発生する。良く考慮せずに同期化用のオブジェクトを選ぶのではなく、使用するロックオブジェクトの特性に十分配慮することが重要である。

ガイドライン	深刻度	可能性	修正コスト	優先度	レベル
LCK01-J	中	中	中	P8	L2

3.2.9. 参考文献

[Findbugs 2008]	
[Miller 2009]	Locking
[Pugh 2008]	“Synchronization”
[Sun 2009c]	Class String, Collections
[Sun 2008a]	Wrapper Implementations

3.3. LCK02-J. getClass()メソッドが返す Class オブジェクトを同期化に使用しない

Object.getClass()メソッドの返り値による同期化は、予期せぬ振舞いの原因となりうる。このような同期を実装するクラスがサブクラス化された場合、サブクラスは常にサブクラス自身にロックを行うが、それはスーパークラスとは完全に異なる Class オブジェクトへのロックである。

*Java 言語仕様*のセクション 4.3.2「クラス Object」は、メソッド同期がどのように機能するかを、以下の通り、記述している[Gosling 2005]。

> *synchronized 宣言されたクラスメソッドは、そのクラスの Class オブジェクトをロックして同期化を行う。*

このことは、getClass()メソッドを使用するサブクラスが、スーパークラスの Class オブジェクトのみを対象に同期化できることを意味しない。プログラマが意図する動作かは分からないが、この場合サブクラスは自身の Class オブジェクトをロックすることになる。プログラマが意図した動作であれば、明確に文書化されるかアノテートされるべきである。本ガイドラインに違反する（getClass()メソッドの返り値を同期化に使用する）メソッドを持つクラスのサブクラスがこのメソッドをオーバーライドしない場合、サブクラスにおいて同メソッドを使いスーパークラスと同様のロックが利用できるとの誤った認識を持つかもしれない。

クラスリテラルを使用して同期化する場合、信頼できないコードにロックオブジェクトへのアクセスを許すべきではない。クラスが package-private であれば、他のパッケージからの呼出し元はクラスオブジェクトにアクセスできないので、固有ロックオブジェクトとしての信頼性は確保されている。詳細については、ガイドライン「LCK00-J. 信頼できないコードから使用されるクラスの同期化には private final ロックオブジェクトを使用する」を参照。

3.3.1. 違反コード (getClass()メソッドが返すロックオブジェクト)

以下の違反コードでは、Base クラスの parse()メソッドは、getClass()メソッドが返す Class オブジェクトで同期を行い日付を解析する。Derived クラスは parse()メソッドを継承している。しかし、この継承されたメソッドの同期は、Base クラスの Class オブジェクトではなく、getClass()メソッドからの返り値である Derived クラスの Class オブジェクトにより行われる。

更に Derived クラスには、Base クラスの Class オブジェクトのロックを行う doSomethingAndParse()メソッドが追加されているが、これは開発者が Base クラスの parse()メソッド同様のロックポリシーに基づき Base クラスの Class オブジェクトのロックを行う必要があると誤解したためである。その結果、Derived クラスには二つの異なるロック方式が混在してしまいスレッドセーフではない。

```
class Base {
  static DateFormat format =
    DateFormat.getDateInstance(DateFormat.MEDIUM);
  public Date parse(String str) throws ParseException {
    synchronized (getClass()) {
      return format.parse(str);
    }
  }
}

class Derived extends Base {
  public Date doSomethingAndParse(String str) throws ParseException {
    synchronized(Base.class) {
      // ...
      return format.parse(str);
    }
  }
}
```

3.3.2. 適合コード (クラス名の限定修飾)

以下の適合コードでは、ロックを提供しているクラス（Base）を完全限定名で指定している。

```
class Base {
  static DateFormat format =
    DateFormat.getDateInstance(DateFormat.MEDIUM);
  public Date parse(String str) throws ParseException {
    synchronized (Base.class) {
      return format.parse(str);
    }
  }
}

// ...
```

上記のコード例では、Derived オブジェクトから呼ばれていても、常に Base.class オブジェクトを同期化している。

3.3.3. 適合コード (Class.forName()メソッド)

以下の適合コードでは、Class.forName()メソッドを使って、Base クラスの Class オブジェクトへの同期化を行っている。

```
class Base {
  static DateFormat format =
    DateFormat.getDateInstance(DateFormat.MEDIUM);
  public Date parse(String str) throws ParseException {
    synchronized (Class.forName("Base")) {
      return format.parse(str);
    }
  }
}

// ...
```

Class.forName()を使用してクラスをロードする際に、信頼できない入力値が引数として渡されないことが重要である。

3.3.4. 違反コード (getClass()メソッドが返すロックオブジェクト、内部クラス)

以下の違反コードでは、Base クラスの parse()メソッド内で getClass()メソッドが返す Class オブジェクトによる同期を行っている。また、Base クラスは、getClass()メソッドの返り値を誤って同期化に使用している doSomethingAndParse()メソッドを含む内部クラス Helper を含んでいる。

```
class Base {
  static DateFormat format =
    DateFormat.getDateInstance(DateFormat.MEDIUM);
  public Date parse(String str) throws ParseException {
    synchronized (getClass()) {
      return format.parse(str);
    }
  }

  public Date doSomething(String str) throws ParseException {
    return new Helper().doSomethingAndParse(str);
  }
  private class Helper {
    public Date doSomethingAndParse(String str) throws ParseException {
      synchronized(getClass()) { // getClass()メソッドの返り値を同期化に使用
        // ...
        return format.parse(str);
      }
    }
  }
}
```

Helper クラスでの getClass()メソッドの呼出しでは、Base クラスの Class オブジェクトではなく Helper クラスの Class オブジェクトが返される。したがって、Base.parse()メソッドを呼ぶスレッドと、Base.doSomething()メソッドを呼ぶスレッドでは異なるオブジェクトへのロックを行うことになる。内部クラスは、外部クラスに含まれる形で記述されていることもあり、内部クラスにおける並行処理に関するエラーは見落とされがちである。レビュー担当者は、二つのクラスが同一のロック方式を有していると誤認するかもしれない。

3.3.5. 適合コード (クラス名の限定修飾)

以下の適合コードでは、parse()メソッドおよび doSomethingAndParse()メソッドの双方で Base クラスリテラルを使用して同期化している。

```
class Base {
  // ...

  public Date parse(String str) throws ParseException {
    synchronized (Base.class) {
      return format.parse(str);
    }
  }

  private class Helper {
    public Date doSomethingAndParse(String str) throws ParseException {
      synchronized(Base.class) { // Base クラスリテラルを同期化に使用
        // ...
        return format.parse(str);
      }
    }
  }
}
```

したがって、Base および Helper クラスの両方が、Base クラスの固有ロックにより同期を行っている。同様に、Class.forname()メソッドをクラスリテラルの代わりに使用することも可能である。

3.3.6. リスク評価

getClass()メソッドが返す Class オブジェクトを使用した同期化は、予期せぬ結果につながる場合がある。

ガイドライン	深刻度	可能性	修正コスト	優先度	レベル
LCK02-J	中	中	中	P8	L2

3.3.7. 参考文献

[Findbugs 2008]	
[Miller 2009]	Locking
[Pugh 2008]	“Synchronization”
[Sun 2009b]	

3.4. LCK03-J. 高水準な並行処理オブジェクトの固有ロックを同期化に使用しない

java.util.concurrent.locks パッケージのインターフェースである Lock または Condition のいずれか一方、あるいは両方を実装するオブジェクトの固有ロックを使用することは不適切である。たとえコードが正しく機能するように見えても、これらのクラスの固有ロックを使用することは推奨されない。この問題は、固有ロックから java.util.concurrent の動的ロックユーティリィティを使用するようにコードを修正する場合に発見されやすい。

3.4.1. 違反コード (ReentrantLock ロックオブジェクト)

以下の違反コードの doSomething()メソッドでは、ReentrantLock によりカプセル化された再入可能な排他ロックを使用せず、ReentrantLock インスタンスの固有ロックで同期化している。

```
private final Lock lock = new ReentrantLock();

public void doSomething() {
  synchronized(lock) {
    // ...
  }
}
```

3.4.2. 適合コード (lock()と unlock() メソッドの使用)

以下の適合コードでは、ReentrantLock オブジェクトの固有ロックではなく、Lock インターフェースが提供する lock()メソッドおよび unlock()メソッドを使用している。

```
private final Lock lock = new ReentrantLock();

public void doSomething() {
  lock.lock();
  try {
    // ...
  } finally {
    lock.unlock();
  }
}
```

なお、java.util.concurrent パッケージの動的ロックユーティリィティが提供する高度な機能が必要ではない場合、Executor フレームワーク、あるいは同期化およびアトミック変数クラスのような他の並列処理用の同期プリミティブを使用する方がよい。

3.4.3. リスク評価

高水準な並行処理ユーティリィティの固有ロックによる同期化は、二種類の異なるロックポリシーが混在することになりうるので、予期せぬ結果を引き起こす場合がある。

ガイドライン	深刻度	可能性	修正コスト	優先度	レベル
LCK03-J	中	中	中	P8	L2

3.4.4. 参考文献

[Findbugs 2008]	
[Miller 2009]	Locking
[Pugh 2008]	“Synchronization”
[Sun 2009b]	
[Sun 2008a]	Wrapper Implementations

3.5. LCK04-J. アクセス可能なコレクションのコレクションビューを同期化に使用しない

java.util.Collections インターフェースの文書[Sun 2009b]では、コレクションビューを利用して繰返し処理を行う際に、コレクションオブジェクトへの同期が失敗する場合について、以下のように警告している。

> *コレクションビューを使用して繰返し処理を行う場合、ユーザは返されるマップを使用して同期化を行う必要がある。…これに従わない場合、動作は不定となる。*

ロックオブジェクトとして、基となるコレクションの代わりにコレクションビューを使用するクラスは、意図せずに二種類の異なるロック方式を使用することになるかもしれない。この場合、基となるコレクションが複数のスレッドからアクセス可能であれば、そのクラスはスレッドセーフではない。

3.5.1. 違反コード (コレクションビュー)

以下の違反コードでは、二つのビューを作成している。一つは map フィールドに定義された Collections.synchronizedMap で同期化された空の HashMap ビューであり、もう一つは set フィールドに定義された map のキーセットのビューである。以下の例では、set ビューを同期化している[Sun 2008a]。

```
// map は package-private
final Map<Integer, String> map =
   Collections.synchronizedMap(new HashMap<Integer, String>());
private final Set<Integer> set = map.keySet();

public void doSomething() {
   synchronized(set) { // 誤って set を同期化している
      for (Integer k : set) {
         // ...
      }
   }
}
```

以下の図 4 で示されているように、上記の例において、HashMap は Map の基となるコレクションを提供しており、Map は Set の基となるコレクションを提供している。

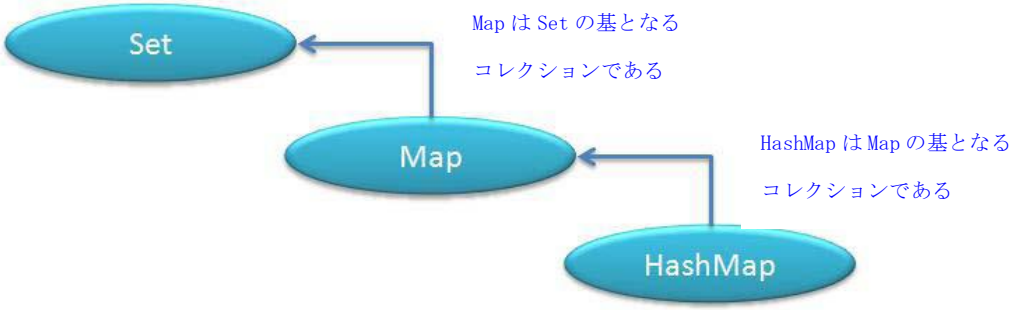

図 4: 違反コード例におけるコレクションビューと基となるコレクションの関係

この違反コードにおいて、HashMap はアクセス不可能であるが、map ビューはアクセス可能（package-private）である。set ビューを map ビューの代わりに同期化するので、他のスレッドが map の内容を更新してイテレータ k を無効にすることができる。

3.5.2. 適合コード (コレクションロックオブジェクト)

以下の適合コードでは、set ビューの代わりに map ビューを同期化している。

```
// map は package-private
final Map<Integer, String> map =
   Collections.synchronizedMap(new HashMap<Integer, String>());
private final Set<Integer> set = map.keySet();
public void doSomething() {
   synchronized(map) { // set にではなく map に同期化している
      for (Integer k : set) {
         // ...
      }
   }
}
```

繰返し処理中は map は変更不可能なので、上記のコードはルールに適合している。

3.5.3. リスク評価

コレクションオブジェクトの代わりにコレクションビューへの同期化を行うと、予期しない結果を引き起こす場合がある。

ガイドライン	深刻度	可能性	修正コスト	優先度	レベル
LCK04-J	低	中	中	P8	L2

3.5.4. 参考文献

[Sun 2009b]	Class Collections
[Sun 2008a]	Wrapper Implementations

3.6. LCK05-J. 信頼できないコードが変更できる static フィールドへのアクセスは同期する

static 宣言されたフィールドを更新するメソッドが信頼できないコードから呼出し可能である場合、そのフィールドへのアクセスを同期化しなければならない。なぜなら、信頼できないクライアントが、フィールドへのアクセスのために同期を適切に行う保証はないためである。static 宣言されたフィールドはすべてのクライアントに共有されるので、信頼できないクライアントが規約に違反して、不適切なロックを用いてアクセスするおそれがある。

Bloch によれば[Bloch 2008]

> *メソッドが static 宣言されたフィールドを変更する場合、そのメソッドを利用するスレッドが通常はただ一つであったとしても、このフィールドへのアクセスは同期化しなければならない。なぜなら、そのようなメソッドに外的な手法で同期を取ろうとしても、すべてのクライアントが同様の手法で同期を行うことを保証することは不可能である。*

攻撃者は通常(規約)文書は無視するであろうから、設計意図が文書化されていても信頼できないコードに対して有効ではない。

3.6.1. 違反コード

以下の違反コードでは、static 宣言された counter フィールドへのアクセスを同期化していない。

```
/** このクラスはスレッドセーフではない */
public final class CountHits {
  private static int counter;

  public void incrementCounter() {
    counter++;
  }
}
```

上記のクラス定義は、ガイドライン「VNA02-J. 共有変数への複合操作のアトミック性を確保する」には違反しない。なぜなら、同ガイドラインは、スレッドセーフを約束するクラスにのみ適用されるが、このクラスはスレッドセーフであることを必要としていない。しかし、このクラスにはパブリックにアクセス可能な incrementCounter()メソッドにより更新される static 宣言された counter フィールドが定義されている。よって、信頼できないコードが故意に同期を取らずにフィールドへアクセスすることが可能な場合、このクラスを信頼できるクライアントコードが安全に使用することはできない。

3.6.2. 適合コード

以下の適合コードでは、counter フィールドを保護するために private static final 宣言されたロックを使用している。したがって、いかなる外部的な同期化にも依存しない。この解決方法も、ガイドライン「LCK00-J. 信頼できないコードから使用されるクラスの同期化には private final ロックオブジェクトを使用する」に準拠している。

```
/** このクラスはスレッドセーフである */
public final class CountHits {
  private static int counter;
  private static final Object lock = new Object();

  public void incrementCounter() {
    synchronized (lock) {
      counter++;
    }
  }
}
```

3.6.3. リスク評価

信用できないコードにより更新できる static 宣言されたフィールドへのアクセスを内部的に同期しないと、信頼できないコードにより同期ポリシーが無視される可能性がある。

ガイドライン	深刻度	可能性	修正コスト	優先度	レベル
LCK05-J	低	中	中	P4	L3

3.6.4. 参考文献

[Bloch 2008]	Item 67: “Avoid excessive synchronization”
[Sun 2009b]	

3.7. LCK06-J. static 共有データの保護にインスタンスロックを使用しない

static 宣言した共有データは、インスタンスロックを用いて保護するべきではない。なぜなら、そのクラスのインスタンスを複数生成した場合には、保護されないからである。それゆえ、static 宣言したロックオブジェクトを使用しない限り、共有オブジェクトへの並行アクセスは安全ではない。クラスが信頼できないコードとやり取りする場合には、ガイドライン「LCK00-J. 信頼できないコードから使用されるクラスの同期化には private final ロックオブジェクトを使用する」に従い、ロックを private final 宣言しなければならない。

3.7.1. 違反コード (static 宣言されていないロックオブジェクト)

以下の違反コードでは、static 宣言した counter フィールドへのアクセスを保護するために、static 宣言していないロックオブジェクトを使用している。Runnable タスクを二つ開始すれば、ロックオブジェクトのインスタンスを二つ生成し、各々別々のインスタンスをロックすることになる。

```
public final class CountBoxes implements Runnable {
  private static volatile int counter;
  // ...
  private final Object lock = new Object();

  @Override public void run() {
    synchronized(lock) {
      counter++;
      // ...
    }
  }

  public static void main(String[] args) {
    for(int i = 0; i < 2; i++) {
      new Thread(new CountBoxes()).start();
    }
  }
}
```

また、上記の例では、volatile 宣言されたフィールドへのインクリメント操作がアトミックではないのに同期が不適切であるため、スレッドが不整合な counter 値を参照しうる(ガイドライン「VNA02-J. 共有変数への複合操作のアトミック性を確保する」を参照)。

3.7.2. 違反コード (synchorized メソッド)

以下の違反コードでは、static 宣言した counter フィールドへのアクセスを保護するためにメソッドを同期化している。

```
public final class CountBoxes implements Runnable {
  private static volatile int counter;
  // ...

  public synchronized void run() {
    counter++;
    // ...
  }
  // ...
}
```

上記のケースでは、クラス自体にではなくクラスの各インスタンスに固有ロックを関連付けている。したがって、異なる Runnable インスタンスを使用して構築されたスレッドは、counter の整合性を欠いた値を参照してしまうかもしれない。

3.7.3. 適合コード (static ロックオブジェクト)

以下の適合コードでは、ロックオブジェクトを static 宣言し、インクリメント操作のアトミック性を確保している。

```
public class CountBoxes implements Runnable {
  private static int counter;
  // ...
  private static final Object lock = new Object();
  public void run() {
    synchronized(lock) {
      counter++;
      // ...
    }
  }
  // ...
}
```

この適合コードのように同期化を行う場合、counter 変数を volatile 宣言する必要はない。

3.7.4. リスク評価

static 宣言された共有データを保護するためにインスタンスロックを使用すると、予期せぬ結果につながりうる。

ガイドライン	深刻度	可能性	修正コスト	優先度	レベル
LCK06-J	中	中	中	P8	L2

3.7.5. 参考文献

[Sun 2009b]	

3.8. LCK07-J. 同一順序でロックを要求および解放し、デッドロックを回避する

マルチスレッド Java プログラムにおけるデータ破壊を回避するために、共有データを並行する読み書きから保護する必要がある。各オブジェクトは、synchronized メソッド、synchronized ブロック、あるいは java.util.concurrent の動的なロックオブジェクトなどを使用することができる。しかし、ロックを過度に使用するとデッドロックにつながりうる。

Java 自体はデッドロックの発生を妨げない、もしくはその検出を要求しない[Gosling 2005]。デッドロックは、複数のスレッドがロックの要求および解放を異なる順序で行った場合に発生しうる。したがって、デッドロックはロックの取得および解放を同じ順序で行うことにより回避することができる。

更に言うならば、同期化を行うのは必要不可欠な場合に制限するべきである。たとえば、paint()、dispose()、stop()および destroy()の各メソッドは、アプレット中では決して同期化してはならない。なぜなら、これらのメソッドは常に専用のスレッドにより呼び出されて使用されるためである。また、Thread.stop()および Thread.destroy()メソッドは、非推奨である。詳細は、ガイドライン「THI05-J. スレッドの終了に Thread.stop()メソッドを使用しない」を参照。

本ガイドラインは、制限のあるリソースを利用するプログラムにも適用される。たとえば、デッドロックは複数のスレッドが互いにデータベース接続のようなリソースが解放されるのを待つ状況下で発生しうる。これらの問題は、各待ち状態のスレッドが、リソースの獲得に成功するまでランダムな時間間隔でリソースの要求を再試行させることにより解決できる。

3.8.1. 違反コード (異なるロック順序)

以下の違反コードでは、過度の同期化のためにデッドロックが発生しやすい。balanceAmount フィールドは、特定の BankAccount オブジェクトで利用可能な総残高を表わしている。ユーザは、ある口座から別の口座へ指定した金額をアトミックに転送することができる。

```
final class BankAccount {
  private double balanceAmount; // 銀行口座の総額
  BankAccount(double balance) {
    this.balanceAmount = balance;
  }

  // このインスタンスから引数 ba の BankAccount インスタンスに amount を預金する
  private void depositAmount(BankAccount ba, double amount) {
    synchronized (this) {
      synchronized(ba) {
        if(amount > balanceAmount) {
          throw new IllegalArgumentException("Transfer cannot be completed");
        }
        ba.balanceAmount += amount;
        this.balanceAmount -= amount;
      }
    }
  }
  public static void initiateTransfer(final BankAccount first,
    final BankAccount second, final double amount) {

    Thread transfer = new Thread(new Runnable() {
      public void run() {
        first.depositAmount(second, amount);
      }
    });
    transfer.start();
  }
}
```

このクラスのオブジェクトは、デッドロックを誘発しやすい。攻撃者が、二つの口座にアクセス可能な場合、それぞれ異なる BankAccount インスタンス a および b から送金を開始する二つのスレッドを生成することができる。以下のコードについて考察してみよう。

```
BankAccount a = new BankAccount(5000);
BankAccount b = new BankAccount(6000);
initiateTransfer(a, b, 1000); // スレッド 1 を開始
initiateTransfer(b, a, 1000); // スレッド 2 を開始
```

各送金操作は、それぞれのスレッド内で実行される。最初のスレッド（スレッド１）では、口座 b に amount で指定した金額を預金した後に同じ金額を口座 a から引き落すことにより、口座 a から口座 b への送金処理をアトミックに行っている。二番目のスレッド（スレッド２）では、逆の操作を行っている。即ち、口座 b から口座 a への送金である。それぞれのスレッドで depositAmount()メソッドが実行されると、最初のスレッドはオブジェクト a へのロックを取得する。このとき、二番目のスレッドは、オブジェクト b へのロックを、最初のスレッドより以前に取得する可能性がある。続いて、最初のスレッドは、b へのロックを要求するが、既に二番目のスレッドによって保持されているかもしれない。二番目のス

レッドも、a へのロックを要求するが、既に最初のスレッドによって保持されている。このように、どちらのスレッドも先に進むことができないデッドロックが成立しうる。

上記の違反コードは、プラットフォーム別のスケジューリングの仕様次第で、実際にはデッドロックが発生するかもしれないし、発生しないかもしれない。二つのスレッドがそれぞれ異なる順序で二つの等しいロックを要求し、それぞれのスレッドがもう一方のスレッドによる送金処理の完了を妨げるロックを取得する時、デッドロックが発生することになる。二つのスレッドが二つの等しいロックを要求するが、一方のスレッドがもう一方のスレッドが開始する前に自身の送金処理を完了する場合、デッドロックは発生しない。同様に、二つのスレッドが二つの等しいロックを同じ順序で要求する場合(両方の送金処理で、送金元の口座と送金先の口座が等しい場合に該当)、あるいは送金元および送金先の口座がまったく異なる二つの送金処理が並行に実行された場合、デッドロックは発生しない。

3.8.2. 適合コード (private static final ロックオブジェクト)

口座振替を実行する前に private static final 宣言されたロックオブジェクトを用いて同期化することによりデッドロックを回避することができる。

```
final class BankAccount {
  private double balanceAmount; //  銀行口座の総額
  private static final Object lock = new Object();

  BankAccount(double balance) {
    this.balanceAmount = balance;
  }

  // このインスタンスから引数 ba の BankAccount インスタンスに amount を預金する
  private void depositAmount(BankAccount ba, double amount) {
    synchronized (lock) {
      if (amount > balanceAmount) {
        throw new IllegalArgumentException("Transfer cannot be completed");
      }
      ba.balanceAmount += amount;
      this.balanceAmount -= amount;
    }
  }
  public static void initiateTransfer(final BankAccount first,
    final BankAccount second, final double amount) {

    Thread transfer = new Thread(new Runnable() {
      @Override public void run() {
        first.depositAmount(second, amount);
      }
    });
    transfer.start();
  }
}
```

上記コードでは、二つの異なる BankAccount オブジェクトを持つ二つのスレッドが、並行してそれぞれもう一方の口座に送金しようとする場合でもデッドロックは発生しない。一方のスレッドの処理が実行される前に、もう一方のスレッドは private 宣言されたロックを取得し送金処理の完了後にロックを解放する。

上記の解決方法では private static 宣言されたロックを用い、同時に一つの送金処理しか実行できないようにシステムに制約をかけているので、性能面では問題となりうる。四つの異なる口座間の二つの送金処理であっても並行して実行することができない。この性能面での問題は、BankAccount オブジェクト数が増えるにしたがって飛躍的に大きくなるため規模の拡大に対応できない。

3.8.3. 適合コード (正しく順序付けられたロック)

以下の適合コードでは、確実に複数のロックを同じ順序で取得し、解放している。この例では、BankAccount オブジェクトへの順序付けが可能であることが必要となる。BankAccount クラスに java.lang.Comparable インターフェースを実装し、compareTo()メソッドをオーバーライドすることで順序付けを可能としている。

```
final class BankAccount implements Comparable<BankAccount> {
  private double balanceAmount; // 銀行口座の総額
  private final Object lock;

  private final long id; // BankAccount 毎に一意
  private static long NextID = 0; // 次の未使用 ID

  BankAccount(double balance) {
    this.balanceAmount = balance;
    this.lock = new Object();
    this.id = this.NextID++;
  }

  @Override public int compareTo(BankAccount ba) {
    return (this.id > ba.id) ? 1 : (this.id < ba.id) ? -1 : 0;
  }

  // このインスタンスから引数 ba の BankAccount インスタンスに amount を預金する
  public void depositAmount(BankAccount ba, double amount) {
    BankAccount former, latter;
    if (compareTo(ba) < 0) {
      former = this;
      latter = ba;
    } else {
      former = ba;
      latter = this;
    }
    synchronized (former) {
      synchronized (latter) {
        if (amount > balanceAmount) {
          throw new IllegalArgumentException("Transfer cannot be completed");
        }
        ba.balanceAmount += amount;
        this.balanceAmount -= amount;
      }
    }
  }
  public static void initiateTransfer(final BankAccount first,
    final BankAccount second, final double amount) {
    Thread transfer = new Thread(new Runnable() {
      @Override public void run() {
        first.depositAmount(second, amount);
      }
    });
    transfer.start();
  }
}
```

送金処理が発生する度に二つの BankAccount オブジェクトを順序付けるので、一番目のオブジェクトへのロックは二番目のオブジェクトへのロックよりも先に取得される。したが

って、二つのスレッドで同じ二つの口座間の送金処理を試みれば、両方のスレッドとも二番目の口座よりも先に一番目の口座オブジェクトのロックを取得しようとする。その結果、一方のスレッドは、もう一方のスレッドが送金処理を開始する前に、両方の口座オブジェクトのロックを取得し、送金を完了した後に両方のロックを解放する。

この適合コードにおいては、送金対象となる口座が重複していない限り、複数の送金処理を並行して実行できる。

3.8.4. 適合コード (ReentrantLock クラス)

以下の適合コードでは、各 BankAccount オブジェクトは、それぞれ一つの java.util.concurrent.locks.ReentrantLock を保持している。depositAmount()メソッドは、両方の口座のロック取得、失敗した場合のロック解放、処理の再試行が行えるように設計されている。

```
final class BankAccount {
  private double balanceAmount; // 銀行口座の総額
  private final Lock lock = new ReentrantLock();
  private final Random number = new Random(123L);
  BankAccount(double balance) {
    this.balanceAmount = balance;
  }

  // このインスタンスから引数 ba の BankAccount インスタンスに amount を預金する
  private void depositAmount(BankAccount ba, double amount) throws InterruptedException {
    while (true) {
      if (this.lock.tryLock()) {
        try {
          if (ba.lock.tryLock()) {
            try {
              if (amount > balanceAmount) {
                throw new IllegalArgumentException("Transfer cannot be completed");
              }
              ba.balanceAmount += amount;
              this.balanceAmount -= amount;
              break;
            } finally {
              ba.lock.unlock();
            }
          }
        } finally {
          this.lock.unlock();
        }
      }
      int n = number.nextInt(1000);
      int TIME = 1000 + n; // ライブロックを防ぐための 1 秒+ランダムな遅延
      Thread.sleep(TIME);
    }
  }
  public static void initiateTransfer(final BankAccount first,
    final BankAccount second, final double amount) {

    Thread transfer = new Thread(new Runnable() {
      public void run() {
        try {
          first.depositAmount(second, amount);
        } catch (InterruptedException e) {
          Thread.currentThread().interrupt(); // 割込みステータスのリセット
        }
      }
    });
    transfer.start();
  }
}
```

上記の適合コードでは、無期限にロックを保持するメソッドが存在しないので、デッドロックは発生しない。実行中のオブジェクト自身が持つロックを取得できるが、もう一方のロックが取得できない場合、スレッドは取得済みのロックを解放し、一定時間スリープした後に再度ロックの取得を試みる。

ReentrantLock を使用するコードは、従来の固有ロックを使用して同期されたコードと似た振舞いをするが、ReentrantLock はいくつかの他の機能を提供する。たとえば、tryLock()メソッドは、他のスレッドが既にロックを保持している場合、待機状態で処理をブロックしない。また、単一スレッドによる排他的な書込みのためのロックや、複数のスレッドによる並行的な読取りを実現するために java.util.concurrent.locks.ReentrantReadWriteLock クラスを使用することができる。

3.8.5. 違反コード (異なるロック順序、再帰)

以下の不変な WebRequest クラスは、サーバーが受け取るウェブリクエストをカプセル化している。

```
// 不変な WebRequest クラス
public final class WebRequest {
  private final long bandwidth;
  private final long responseTime;

  public WebRequest(long bandwidth, long responseTime) {
    this.bandwidth = bandwidth;
    this.responseTime = responseTime;
  }

  public long getBandwidth() {
    return bandwidth;
  }

  public long getResponseTime() {
    return responseTime;
  }
}
```

各リクエストは、応答時間とリクエストを処理するのに使用したネットワーク帯域幅を示す値を持つ。

以下の違反コードでは、ウェブリクエストを監視し、入力リクエストを処理するのに必要となった帯域幅と応答時間の平均を求める計算用ルーチンを提供している。

```
public final class WebRequestAnalyzer {
  private final Vector<WebRequest> requests = new Vector<WebRequest>();
  public boolean addWebRequest(WebRequest request) {
    return requests.add(new WebRequest(request.getBandwidth(),
                        request.getResponseTime()));
  }
  public double getAverageBandwidth() {
    if (requests.size() == 0) {
      throw new IllegalStateException("The vector is empty!");
    }
    return calculateAverageBandwidth(0, 0);
  }


  public double getAverageResponseTime() {
    if (requests.size() == 0) {
      throw new IllegalStateException("The vector is empty!");
    }
    return calculateAverageResponseTime(requests.size() - 1, 0);
  }

  private double calculateAverageBandwidth(int i, long bandwidth) {
    if (i == requests.size()) {
      return bandwidth / requests.size();
    }
    synchronized (requests.elementAt(i)) {
      bandwidth += requests.get(i).getBandwidth();
      // ロックを昇順で獲得する
      return calculateAverageBandwidth(++i, bandwidth);
    }
  }

  private double calculateAverageResponseTime(int i, long responseTime) {
    if (i <= -1) {
      return responseTime / requests.size();
    }
.   synchronized (requests.elementAt(i)) {
      responseTime += requests.get(i).getResponseTime();
      // ロックを降順で獲得する
      return calculateAverageResponseTime(--i, responseTime);
    }
  }
}
```

このアプリケーションで使用される WebRequestAnalyzer クラスは、セッターメソッド addWebRequest()を持ち、Vector クラスの requests を使用してウェブリクエストのリスト管理を行う。すべてのスレッドは、getAverageBandwidth()および getAverageResponseTime()メソッドを呼び出し、全ウェブリクエストの帯域幅の平均、あるいは応答時間の平均を取得できる。

これらのメソッドは、Vector の個々の要素(ウェブリクエスト)へのロックを保持することにより、粒度の細かいロックを行っている。このロックにより、計算処理の実行中であっても新規のリクエストが追加されることが可能となっている。したがって、メソッドにより得られる統計値は正確である。

しかし、これらの二つのメソッドの同期ブロック内での再帰呼出しが、それぞれ異なる順番で個々要素の固有ロックを取得するので、上記違反コードではデッドロックが発生しうる。すなわち、calculateAverageBandwidth()メソッドでは、インデックス 0 から requests.size() – 1 まで、昇順でロックを要求するが、一方で calculateAverageResponseTime()メソッドは、インデックス requests.size() – 1 から 0 まで降順でロックを要求している。どちらのメソッドにおいても、再帰呼出し中に獲得済みのロックは解放しない。二つのスレッドの一方が calculateAverageBandwidth()メソッドを呼び出し、この呼出しが終了する前にもう一方が calculateAverageResponseTime()メソッドを呼び出した場合に、デッドロックが発生する。

たとえば、Vector 上にリクエストが 20 件あり、一つのスレッドが getAverageBandwidth()メソッドを呼ぶ場合、そのスレッドは WebRequest 0(Vector 中の最初の要素)の固有ロックを取得する。その間に、別スレッドが getAverageResponseTime()を呼ぶ場合、WebRequest 19 の固有ロックを取得する(Vector 中の最後の要素)。したがって、いずれのスレッドもすべてのロックを取得して計算を続行することができずにデッドロックが発生する。

addWebRequest()メソッドは calculateAverageResponseTime()との関係において競合状態が存在することにも注意。Vector に繰返し処理を行っている間に新しい要素を Vector に追加することは可能だが、追加される前に算出された結果を無効にしてしまう。この競合状態は、要素の挿入前に Vector (Vector 上に少なくとも一つの要素が存在する場合に)の最後の要素にロックを行うことにより防ぐことができる。

3.8.6. 適合コード

以下の適合コードでは、二つの平均計算メソッドは、ロックを同じ順序で獲得、解放しながら、Vector 内の最初のウェブリクエストから処理を開始する。

```
public final class WebRequestAnalyzer {
  private final Vector<WebRequest> requests = new Vector<WebRequest>();
  public boolean addWebRequest(WebRequest request) {
    return requests.add(new WebRequest(request.getBandwidth(),
                        request.getResponseTime()));
  }

  public double getAverageBandwidth() {
    if (requests.size() == 0) {
      throw new IllegalStateException("The vector is empty!");
    }
    return calculateAverageBandwidth(0, 0);
  }
  public double getAverageResponseTime() {
    if (requests.size() == 0) {
      throw new IllegalStateException("The vector is empty!");
    }
    return calculateAverageResponseTime(0, 0);
  }

  private double calculateAverageBandwidth(int i, long bandwidth) {
    if (i == requests.size()) {
      return bandwidth / requests.size();
    }
    synchronized (requests.elementAt(i)) { // ロックを昇順で獲得する
      bandwidth += requests.get(i).getBandwidth();
      return calculateAverageBandwidth(++i, bandwidth);
    }
  }
  private double calculateAverageResponseTime(int i, long responseTime) {
    if (i == requests.size()) {
      return responseTime / requests.size();
    }
    synchronized (requests.elementAt(i)) {
      responseTime += requests.get(i).getResponseTime();
      // ロックを昇順で獲得する
      return calculateAverageResponseTime(++i, responseTime);
    }
  }
}
```

したがって、あるスレッドが平均帯域幅か平均応答時間を計算している間、他のスレッドによる干渉やデッドロックの発生などの影響を受けない。なぜなら、スレッドはまず始めに先頭のウェブリクエストのロックを取得しなければならないが、同ロックは先行するスレッドの計算が完了しない限り解放されないからである。

この適合コードでは、addWebRequest()メソッドで Vector の最終要素をロックする必要はない。なぜなら、(1)すべてのメソッドにおいてロックを昇順で獲得しており、(2) 常に得られる計算結果に Vector の更新内容が反映されるからである。

3.8.7. リスク評価

誤った順序でロックの取得、解放を行うと、デッドロックにつながるおそれがある。

ガイドライン	深刻度	可能性	修正コスト	優先度	レベル
LCK07-J	低	高	高	P3	L3

3.8.8. 参考文献

[Gosling 2005]	Chapter 17, “Threads and Locks”
[Halloway 2000]	
[MITRE 2010]	CWE ID 412, “Unrestricted Lock on Critical Resource”

3.9. LCK08-J. 例外発生時にロックを確実に解放する

例外条件の発生時にロックが解放されない場合、デッドロックにつながりうる。Java API [Sun 2009b]によれば、

> *ReentrantLock は、最後にロックを取得したままでロックを解放していないスレッドにより「所有」される。lock を呼び出すスレッドは、ロックが別のスレッドに所有されていない時にロックを取得できる。*

したがって、ロックを所有するスレッドによりロックが解放されていない限り、他のスレッドが同じロックを取得できない。メソッドおよびブロックへの同期において使用するオブジェクト固有のロックは、スレッドの異常終了のような例外条件発生により自動的に解放される。

3.9.1. 違反コード (チェックされた例外)

以下の違反コードでは、ReentrantLock を使用してリソースを保護するが、ファイルオープン処理中に例外が発生した場合にロックを解放していない。例外がスローされる場合、実行文の制御は catch ブロックに移り、unlock()メソッドの呼出しは実行されない。

```
public final class Client {
  public void doSomething(File file) {
    final Lock lock = new ReentrantLock();
    try {
      lock.lock();
      InputStream in = new FileInputStream(file);
      // オープンされたファイルを操作する
      lock.unlock();
    } catch (FileNotFoundException fnf) {
      // 例外を取り扱う
    }
  }
}
```

このコードにおいて、doSomething()メソッドからリターンしても、自動的にロックは解放されない。

上記の違反コードでは、入力ストリームのクローズもしていないため、ガイドライン「FIO04-J. Close resources when they are no longer needed [3]」にも違反している。

3 このガイドラインは、https://www.securecoding.cert.org/confluence/display/java/に記述されている。

3.9.2. 適合コード (finally ブロック)

以下の適合コードでは、ロックを獲得した直後に try ブロック中で例外がスローされうる操作をカプセル化している。ロックを try ブロックの直前で獲得しており、finally ブロックの実行においてもロックが保持されていることを確実にしている。finally ブロック内で Lock.unlock()メソッドを呼び出すことにより、例外が発生したか否かにかかわらず、ロックは確実に解放される。

```
public final class Client {
  public void doSomething(File file) {
    final Lock lock = new ReentrantLock();
    InputStream in = null;
    lock.lock();
    try {
      in = new FileInputStream(file);
      // オープンされたファイルを操作する
    } catch(FileNotFoundException fnf) {
      // ハンドラへの転送
    } finally {
      lock.unlock();

      if(in != null) {
        try {
          in.close();
        } catch (IOException e) {
          // ハンドラへの転送
        }
      }
    }
  }
}
```

3.9.3. 適合コード (Execute-Around 手法)

Execute-Around 手法は、リソース割当ておよびクリーンアップ操作を行うための全般的な仕組を提供するので、クライアント側では要求された機能に特化して処理することが可能となる。この手法は、クライアント側のソースコードの保守性を向上させるとともにリソース管理のための安全な仕組みを提供する。

以下の適合コードでは、クラス Client の doSomething()メソッドは、LockAction インターフェースの doSomethingWithFile()メソッドを実装することで、ロックの獲得および解放、あるいはファイルのオープンおよびクローズ操作を実装せずに、必要な機能だけを実装し提供する。ReentrantLockAction クラスでは、リソース管理のための動作をすべてカプセル化している。

```
public interface LockAction {
  void doSomethingWithFile(InputStream in);
}

public final class ReentrantLockAction {
  public static void doSomething(File file, LockAction action) {
    Lock lock = new ReentrantLock();
    InputStream in = null;
    lock.lock();
    try {
      in = new FileInputStream(file);
      action.doSomethingWithFile(in);
    } catch (FileNotFoundException fnf) {
      // ハンドラへの転送
    } finally {
      lock.unlock();
      if (in != null) {
        try {
          in.close();
        } catch (IOException e) {
          // ハンドラへの転送
        }
      }
    }
  }
}

public final class Client {
  public void doSomething(File file) {
    ReentrantLockAction.doSomething(file, new LockAction() {
      public void doSomethingWithFile(InputStream in) {
        // オープンされたファイルを操作する
      }
    });
  }
}
```

3.9.4. 違反コード (チェックされない例外)

以下の違反コードでは、スレッドセーフではない java.util.Date インスタンスを保護するために ReentrantLock を使用している。なお、関連して doSomethingSafely()メソッドが、ガイドライン「ERR01-J. Do not allow exceptions to expose sensitive information [4]」に適合するには Throwable をキャッチしなければならない。

4 このガイドラインは、https://www.securecoding.cert.org/confluence/display/java/に記述されている。

```
final class DateHandler {
  private final Date date = new Date();
  final Lock lock = new ReentrantLock();
  public void doSomethingSafely(String str) {
    try {
      doSomething(str);
    } catch(Throwable t) {
      // ハンドラへの転送
    }
  }
  public void doSomething(String str) {
    lock.lock();
    String dateString = date.toString();
    if (str.equals(dateString)) {
      // ...
    }
    lock.unlock();
  }
}
```

str が空かどうかを doSomething()メソッドではチェックしていないため、実行時例外が発生し、結果的に獲得したロックが解放されないおそれがある。

3.9.5. 適合コード (finally ブロック)

以下の適合コードでは、例外をスローする可能性のあるすべての操作を try ブロックで括り、対応する finally ブロック内でロックを解放している。

```
final class DateHandler {
  private final Date date = new Date();
  final Lock lock = new ReentrantLock();
  public void doSomethingSafely(String str) {
    try {
      doSomething(str);
    } catch(Throwable t) {
      // ハンドラへの転送
    }
  }
  public void doSomething(String str) {
    lock.lock();
    try {
      String dateString = date.toString();
      if (str != null && str.equals(dateString)) {
        // ...
      }
    } finally {
      lock.unlock();
    }
  }
}
```

上記コードでは、ロックは実行時例外イベントが発生する場合でも解放される。また、doSomething()メソッドでは、NullPointerException が決してスローされないようにしている。

3.9.6. リスク評価

例外条件発生時にロックを解放できないと、スレッド飢餓状態およびデッドロックにつながるおそれがある。

ガイドライン	深刻度	可能性	修正コスト	優先度	レベル
LCK08-J	低	高	低	P9	L2

3.9.7. 参考文献

[Sun 2009b]	Class ReentrantLock

3.10. LCK09-J. ロックを保持したままブロックする操作を実行しない

ロックを保持している間に時間が掛かる処理やブロックする操作を呼び出すと、深刻なシステムのパフォーマンス低下やリソース飢餓状態を引き起こす可能性がある。更に、相互依存関係にあるスレッドが無制限にブロックするような場合、デッドロックが生じることもありうる。ブロックする操作には、ネットワーク、ファイル、およびコンソールなどの入出力(たとえば Console.readLine()メソッド)およびオブジェクトの直列化（シリアライゼーション）が含まれる。スレッドの無制限な実行の遅延もブロックする操作に含まれる。

JVM が信頼性の低いネットワークファイルシステムとやり取りする場合、ファイル入出力時に深刻な性能問題が発生しうる。このような状況下では、ロックを保持している間はネットワーク経由のファイル入出力は回避すべきである。ログ出力のような、出力用ストリームのロック獲得、あるいは、入出力操作の完了待ちが必要になるファイル操作においては、全体の処理速度を向上するために専用のスレッドを用いることができる。このとき、ログの出力リクエストは、ファイル入出力と比較してオーバーヘッドがほとんどないキューへの put() 操作により追加できる。[Goetz 2006]。

3.10.1. 違反コード (待機スレッド)

以下の違反コードでは、引数 time を受け取るユーティリィティメソッドを定義している。

```
public synchronized void doSomething(long time)
      throws InterruptedException {
   // ...
   Thread.sleep(time);
}
```

このメソッドは同期されているため、このメソッドを実行中のスレッドが保留状態となる場合、他のスレッドはこの同期メソッドを使用することができない。Thread.sleep()メソッドには、ガイドライン「THI00-J. sleep()、yield()および getState()の各メソッドが同期セマンティックスを持つと想定しない」の中で詳述するように、同期処理に関するセマンティクスが存在しないので、処理中のオブジェクト固有のロックは解放されない。

3.10.2. 適合コード (固有ロック)

以下の適合コードでは､引数として time の代わりに timeout を取る doSomething()メソッドを定義している｡Thread.sleep()メソッドではなく Object.wait()メソッドを使用することで、通知によりスレッドを起動するタイムアウト値をセットできる。

```
public synchronized void doSomething(long timeout)
      throws InterruptedException {
   while (<条件が一致しない場合>) {
      wait(timeout); // 処理中のモニタを直ちに解放する
   }
}
```

このコードでは Object.wait()メソッドを利用しているため、処理中のオブジェクト固有のロックは、待ち状態に移行後直ちに解放される。タイムアウト後、スレッドはオブジェクトモニタを再獲得した後に実行を再開する。

Java API クラス Object の文書によれば [Sun 2009b]

> *wait メソッドにより実行中のスレッドが待機状態となるとき、this オブジェクト固有のロックのみが解放される。同スレッドが他のオブジェクト固有のロックを保持する場合、これらのロックはスレッドが待機中、解放されずに保持され続ける。wait メソッドは、this オブジェクトのモニターを所有するスレッドからのみ呼び出せる。*

他のオブジェクトへのロックを保持するスレッドが待ち状態へ移行する場合は、これらのロックを適切に解放しなければならない。待機と通知に関する補足的なガイドラインについては、「THI03-J. wait()および await()メソッドは、常にループ内部で呼び出す」および「THI04-J. 一つではなく、すべての待ち状態スレッドへ通知する 」に記述されている。

3.10.3. 違反コード (ネットワーク入出力)

以下の違反コードでは、サーバーからクライアントに Page オブジェクトを送る sendPage()メソッドを示している。複数のスレッドから同時アクセスが要求される場合でも、配列 pageBuff が安全にアクセスされるようメソッドを同期している。

```
//別途定義されている Page クラスは、保持する Page 名を getName()メソッドにより返す。
Page[] pageBuff = new Page[MAX_PAGE_SIZE];

public synchronized boolean sendPage(Socket socket, String pageName)
      throws IOException {
   // Page 書込み用の出力ストリームを取得する
   ObjectOutputStream out = new ObjectOutputStream(socket.getOutputStream());

   // クライアントから要求された Page を検索する(この操作は同期化を要求する)
   Page targetPage = null;
   for (Page p : pageBuff) {
      if (p.getName().compareTo(pageName) == 0) {
         targetPage = p;
      }
   }

   // 要求された Page が存在しない場合
   if (targetPage == null) {
      return false;
   }

   // クライアントに Page を送信する(同期化は必要ではない)
   out.writeObject(targetPage);

   out.flush();
   out.close();
   return true;
}
```

遅延が大きいネットワークや不安定なネットワークなどの状況下、同期化された sendPage()メソッド内の writeObject()メソッドを呼び出すと、遅延やデッドロックにつながるおそれがある。

3.10.4. 適合コード

以下の適合コードでは、プロセスを一連のステップに分割している。

1. 同期化を必要とするデータ構造への操作を行う。
2. 送信するオブジェクトのコピーを作成する。
3. 同期化を必要としない別メソッドでネットワーク接続呼出しを行う。

以下の適合コードでは、同期化されていない sendPage()メソッドは、同期化された getPage()メソッドを呼び出して、要求された Page を pageBuff 配列から所得する。Page の取得後、sendPage()メソッドは、同期化されていない deliverPage()メソッドを呼び出して、クライアントに Page を送信している。

```
public boolean sendPage(Socket socket, String pageName) { //  同期化しない。
   Page targetPage = getPage(pageName);
   if (targetPage == null)
      return false;
   return deliverPage(socket, targetPage);
}

private synchronized Page getPage(String pageName) { //  同期化必要
   Page targetPage = null;
   for (Page p : pageBuff) {
      if (p.getName().equals(pageName)) {
         targetPage = p;
      }
   }
   return targetPage;
}

// エラー発生時には false を返し、成功時には true を返す
public boolean deliverPage(Socket socket, Page page) {
   ObjectOutputStream out = null;
   boolean result = true;
   try {
      // Page 書込み用の出力ストリームを取得する
      out = new ObjectOutputStream(socket.getOutputStream());

      // クライアントに Page を送信する
      out.writeObject(page);
   } catch (IOException io) {
      result = false;
   } finally {
      if (out != null) {
         try {
            out.flush();
            out.close();
         } catch (IOException e) {
            result = false;
         }
      }
   }
   return result;
}
```

3.10.5. 例外

LCK09-EX1: 呼出し元に適切な終了メカニズムを提供するクラスは、このガイドラインに従わなくてもよい。詳細は、ガイドライン「THI06-J. ブロックしているスレッドやタスクが確実に終了できること 」を参照。

LCK09-EX2: 複数のロックを必要とするメソッドは、いくつかのロックを保持した状態で、残りの必要なロックが利用可能になるのを待つ場合がある。このようなケースにおいては、プログラマはデッドロックを回避するために他の適用可能なガイドラインに従わなければならない。ガイドライン「LCK07-J. 同一順序でロックを要求および解放し、デッドロックを回避する」を参照。

3.10.6. リスク評価

同期化されたブロック内で行われるブロックする操作あるいは長い時間が必要な操作により、システムがデッドロックしたり無反応になってしまうおそれがある。

ガイドライン	深刻度	可能性	修正コスト	優先度	レベル
LCK09-J	低	中	高	P2	L3

3.10.7. 参考文献

[Gosling 2005]	Chapter 17, “Threads and Locks”
[Grosso 2001]	Chapter 10, “Serialization”
[Rotem-Gal-Oz 2008]	“Falacies of Distributed Computing Explained”
[Sun 2009b]	Class Object

3.11. LCK10-J. ダブルチェックロック手法を誤用しない

コンストラクタを使用してメンバオブジェクトを初期化する代わりに、インスタンスが実際に必要となるまでメンバオブジェクトの構築を延期する*遅延初期化*を行うことができる。遅延初期化は、クラスおよびインスタンスの誤った循環初期化処理の防止、およびその他の最適化においても有用である [Bloch 2005a]。

遅延初期化にはクラスメソッドあるいはインスタンスメソッドが用いられるが、どちらを使うかはメンバオブジェクトが静的かどうかによって決まる。以下コード中のメソッド getHelper()は、インスタンスが生成済であるかどうかをチェックし、未生成の場合にインスタンスを生成する。インスタンスが既に生成済みの場合、単にその内容を返す。

```
// 遅延初期化を実行する正しいシングルスレッドのバージョン
final class Foo {
  private Helper helper = null;
  public Helper getHelper() {
    if (helper == null) {
      helper = new Helper();
    }
    return helper;
  }
  // ...
}
```

マルチスレッドのアプリケーションでは、複数のスレッドがメンバオブジェクトの余計なインスタンスを作成しないように、初期化処理を同期化しなければならない。

```
// 遅延初期化を実行する正しいマルチスレッドのバージョン
final class Foo {
  private Helper helper = null;
  public synchronized Helper getHelper() {
    if (helper == null) {
      helper = new Helper();
    }
    return helper;
  }
  // ...
}
```

上記コード例は、それぞれが想定する状況下での利用において正しく動作するが、ダブルチェックロック手法の適用により実行性能を改善できる。ダブルチェックロック手法は、同期化する対象を実行機会が限定される新しいインスタンス生成に限定し、かつ構築済みのインスタンスを返すなど実行頻度が高い処理よりも先行して実行させることで、実行性能を改善する。

初期化が不完全な状態のオブジェクトの公開は、ダブルチェックロック手法の誤用例の一つである。

3.11.1. 違反コード

ダブルチェックロック手法は、メソッド同期ではなくブロック同期を用い、同期を行う前に付加的な null チェックを行う。以下の違反コードでは、ダブルチェックロックを誤用している。

```
// 「ダブルチェックロック」
final class Foo {
  private Helper helper = null;
  public Helper getHelper() {
    if (helper == null) {
      synchronized (this) {
        if (helper == null) {
          helper = new Helper();
        }
      }
    }
    return helper;
  }
  // 他のメソッドやメンバを定義...
}
```

Pugh によれば [Pugh 2004]

> *……Helper オブジェクトを初期化する書込みと helper フィールドへの書込みは、意図しない順序で実行されうる。getHelper() メソッドを呼び出すスレッドでは、helper オブジェクトへの null ではない参照を得ることはできるが、helper オブジェクトの各フィールドに関しては、コンストラクタによりセットされる値ではなく各フィールドの既定値が参照される可能性がある。コンパイラがこれらの書込みの順序替えをしない場合でも、マルチプロセッサ環境下では、プロセッサまたはメモリシステムがこれらの書込みを順序替えするかもしれない。*

ガイドライン「TSM03-J. 初期化が不完全なオブジェクトを公開しない」も参照。

3.11.2. 適合コード (volatile 変数)

以下の適合コードでは、helper フィールドを volatile 修飾して宣言している。

```
// volatile 変数の同期セマンティックスに基づき機能する
// JDK 1.4 以前のバージョンでは機能しない
final class Foo {
  private volatile Helper helper = null;
  public Helper getHelper() {
    if (helper == null) {
      synchronized (this) {
        if (helper == null) {
          helper = new Helper(); // helper が null の場合、新しいインスタンスを生成する
        }
      }
    }
    return helper; // helper が non-null の場合、自身のインスタンスを返す
  }
}
```

Helper オブジェクトを初期化するスレッドとインスタンスを返す別のスレッドとの間に、事前発生関係が確立される [Pugh 2004, Manson 2004]。

3.11.3. 適合コード (static イニシャライザ)

以下の適合コードでは、helper フィールドの初期化を静的変数の宣言で行っている。

```
final class Foo {
  private static final Helper helper = new Helper();
  public static Helper getHelper() {
    return helper;
  }
}
```

static 宣言と同時に初期化される変数、あるいは static イニシャライザ(クラスロード時に一度だけ実行される static 宣言されたコードブロック)により初期化される変数は、他のスレッドに可視となる前に、その構築を完了することが保証されている。

3.11.4. 適合コード (Initialize-On-Demand Holder クラスパターン)

以下の適合コードでは、static 宣言された Holder 内部クラスで静的変数を宣言することにより、遅延初期化処理が組み込まれる Initialize-On-Demand Holder クラスパターンを使用している。

```
final class Foo {
  // 遅延初期化処理
  private static class Holder {
    static Helper helper = new Helper();
  }
  public static Helper getInstance() {
    return Holder.helper;
  }
}
```

getInstance()メソッドが呼び出されるまで、static 宣言した helper フィールドの初期化処理は遅延される。 static 宣言したフィールドに遅延初期化を行う場合、この手法はダブルチェックロックよりも優れた選択である [Bloch 2008]。しかし、この手法をインスタンスフィールドの遅延初期化を行うために使用することはできない [Bloch 2001]。

3.11.5. 適合コード (ThreadLocal ストレージ)

以下の適合コード (オリジナルは Terekhov が提示[Pugh 2004]) では、Helper インスタンスの遅延生成に ThreadLocal オブジェクトを使用している。

```
final class Foo {
  private final ThreadLocal<Foo> perThreadInstance = new ThreadLocal<Foo>();
  private Helper helper = null;

  public Helper getHelper() {
    if (perThreadInstance.get() == null) {
      createHelper();
    }
    return helper;
  }

  private synchronized void createHelper() {
    if (helper == null) {
      helper = new Helper();
    }
    // non-null の値であれば set()メソッドの引数として使用可能
    perThreadInstance.set(this);
  }
}
```

3.11.6. 適合コード (不変クラス)

以下の適合コードでは、Helper クラスは不変であり、可視となる前に完全な状態で確実に構築される。よって、ダブルチェックロック手法を適用する場合に、不完全な初期化状態のオブジェクトが公開されないよう確認する必要はない。

```
public final class Helper {
  private final int n;

  public Helper(int n) {
    this.n = n;
  }

  // 他のフィールドやメソッドは、すべて final 宣言される
}

final class Foo {
  private Helper helper = null;
  public Helper getHelper() {
    if (helper == null) {
      synchronized (this) {
        if (helper == null) {
          helper = new Helper(42); // helper が null の場合、新しいインスタンスを生成する
        }
      }
    }
    return helper; // helper が non-null の場合、自身のインスタンスを返す
  }
}
```

3.11.7. 例外

LCK10-EX1: 32 ビットのプリミティブ変数(たとえば int 型または float 型)を利用する場合は、このガイドラインに適合する必要がない[Pugh 2004]。しかし、64 ビットのプリミティブ変数への同期していない読み書きのアトミック性は保証されていないので、long 型や double 型の変数の利用には注意する必要がある(ガイドライン「VNA05-J. 64 ビット値の読み書きのアトミック性を確保する」を参照)。

3.11.8. リスク評価

ダブルチェックロックの誤用は、不適切な同期処理の原因となる。

ガイドライン	深刻度	可能性	修正コスト	優先度	レベル
LCK10-J	低	中	中	P4	L3

3.11.9. 参考文献

[Bloch 2001]	Item 48: “Synchronize access to shared mutable data”
[Bloch 2008]	Item 71: before: “Use lay initialization judiciously”
[Gosling 2005]	Section 12.4, “Initialization of Classes and Interfaces”
[MITRE 2010]	CWE ID 609 “Double-Checked Locking”
[Pugh 2004]	
[Sun 2009b]	

3.12. LCK11-J. 一貫したロック方式が適用されないクラスには、クライアントサイドロックを利用しない

Goetz らによれば [Goetz 2006]

> *クライアントサイドロックとは、オブジェクト X を使用するクライアントコードを保護するために、オブジェクト X がその状態を保護するために使用するロックを用いることを意味する。よって、クライアントは、クライアントサイドロックを使用するために、オブジェクト X が使用するロックを知っていなければならない。*

スレッドセーフなクラスに一貫したロック方式が実装され、その内容が明確に文書化されている場合、クライアントサイドロックを利用できるが、Goetz らは、その誤用を警告している [Goetz 2006]。

> *クラスを拡張してアトミックな操作を新たに加えるやり方は、ロック関連のコードを複数のクラスに分散させるので脆弱であるが、クライアントサイドロックはそれに輪をかけて脆弱である。あるクラス C のロック関連のコードが、C とまったく関係のないクラスに置かれるからである。クライアントサイドロックに用いるクラスのロック方式が変わりうる場合、特に注意する必要がある。*

クライアントサイドロックに使用可能なクラスは、その適切な用法を明示的に文書化するべきである。たとえば、java.util.concurrent.ConcurrentHashMap<K,V>クラスは、文書上で以下のように記述されており、クライアントサイドロックに使用するべきでない [Sun 2009b]。

> *…すべての操作はスレッドセーフだが、取得操作にロックは利用されておらず、また、すべてのアクセスをブロックするようなテーブル全体のロックはサポートされていない。プログラムがこのような同期仕様ではなくそのスレッド安全性に依存する場合、このクラスは Hashtable と相互運用が可能である。*

クライアントサイドロックの適用は、使用するクラスに関する文書で推奨されている場合に限定するのが望ましい。たとえば、java.util.Collections クラスの synchronizedList()ラッパーメソッドの文書では、以下のように述べている [Sun 2009b]。

> *確実に直列アクセスを実現するには、基になるリストへのアクセスはすべて、返されたリストを介して行う必要がある。返されたリストの繰返し処理を行う場合、ユーザは同期化を行う必要がある。これを行わない場合、動作は保証されない。*

なお、基となるリストが信用できないクライアントからアクセスできない状況下であれば、上記内容はガイドライン「LCK04-J. アクセス可能なコレクションのコレクションビューを同期化に使用しない」と矛盾しない。

3.12.1. 違反コード (固有ロック)

この違反コードでは、以下のスレッドセーフな Book クラスを使用しているが、このクラスを直接修正できない。たとえば、このソースコードをレビュー用に入手できないか、クラスが拡張不可能な汎用ライブラリの一部である場合、直接修正することはできない。

```
final class Book {
  // 将来的には private final 宣言したロックを使用するようにロックポリシーを変更するかもしれない
  private final String title;
  private Calendar dateIssued;
  private Calendar dateDue;

  Book(String title) {
    this.title = title;
  }
  public synchronized void issue(int days) {
    dateIssued = Calendar.getInstance() ;
    dateDue = Calendar.getInstance();
    dateDue.add(dateIssued.DATE, days);
  }

  public synchronized Calendar getDueDate() {
    return dateDue;
  }
}
```

上記のクラスでは、一貫して特定のロック方式が利用されるとは限らない (すなわち、ロック方式を予告なしに変更する権利が留保されている)。更に、呼出し元がクライアントサイドロックを安全に使用できるとは明示されていない。BookWrapper クライアントクラスは、renew()メソッド内で Book インスタンスを用いた同期化を行い、クライアントサイドロックを使用している。

```
// クライアント側ソースコード
public class BookWrapper {
  private final Book book;

  BookWrapper(Book book) {
    this.book = book;
  }

  public void issue(int days) {
    book.issue(days);
  }

  public Calendar getDueDate() {
    return book.getDueDate();
  }

  public void renew() {
    synchronized(book) {
      if (book.getDueDate().before(Calendar.getInstance())) {
        throw new IllegalStateException("Book overdue");
      } else {
        book.issue(14); // 14 日間貸し出しを延長
      }
    }
  }
}
```

Book クラスの同期化の実装が将来変更された場合、BookWrapper クラスのロック方式は不適切になるかもしれない。たとえば、Book クラスが private final 宣言したロックオブジェクトを使用するように変更された場合(ガイドライン「LCK00-J. 信頼できないコードから使用されるクラスの同期化には private final ロックオブジェクトを使用する」で推奨されている)、BookWrapper クラスのロック方式は破られてしまう。BookWrapper.getDueDate() メソッドを呼び出すスレッドが、新しいロックポリシーに従いスレッドセーフな Book クラスを操作すると、BookWrapper クラスのロック方式は破られる。しかし、renew()メソッドを呼び出すスレッドは、常に Book インスタンスの固有ロックで同期することになる。この場合、実装上二種類の異なるロックが使用されている。

3.12.2. 適合コード (private final ロックオブジェクト)

以下の適合コードでは、private final 宣言されたロックオブジェクトを使用して BookWrapper クラスのメソッドを同期している。

```
public final class BookWrapper {
  private final Book book;
  private final Object lock = new Object();

  BookWrapper(Book book) {
    this.book = book;
  }

  public void issue(int days) {
    synchronized(lock) {
      book.issue(days);
    }
  }

  public Calendar getDueDate() {
    synchronized(lock) {
      return book.getDueDate();
    }
  }

  public void renew() {
    synchronized(lock) {
      if (book.getDueDate().before(Calendar.getInstance())) {
        throw new IllegalStateException("Book overdue");
      } else {
        book.issue(14); // 14 日間貸し出しを延長
      }
    }
  }
}
```

BookWrapper クラスのロック方式は、上記の適合コードでは、Book インスタンスのロックポリシーには依存していない。

3.12.3. 違反コード (クラス継承とアクセス可能なメンバーのロック)

Goetz らは、スレッドセーフなクラスに対する機能追加を目的としたクラス拡張の脆さについて以下のように記述している [Goetz 2006]。

> *サブクラス化による拡張は、同期ポリシーの実装が複数の別々のソースファイルに分散されるため、コードを直接スーパークラスに追記して拡張する方法に比べて脆弱である。もし、元のクラスが同期ポリシーを変更して、その状態変数を保護するために別のロックを選んだ場合、その影響を受けサブクラスは機能しなくなる。なぜなら、ベースクラスの状態変数への並行アクセスを制御するために使うロックが、正しいロックではなくなってしまうからである。*

以下の違反コードでは、PrintableIPAddressList クラスはスレッドセーフな IPAddressList クラスを継承している。PrintableIPAddressList は、addAndPrintIPAddresses()メソッド内で IPAddressList.ips をロックしている。このコードは、サブクラスが、スーパークラスに所有およびロックされているオブジェクトを使用するクライアントサイドロックの一例である。

```
// このクラスは、将来的にはそのロックポリシーを変更するかもしれない。
// たとえば、新規のアトミックでないメソッドを追加する場合
class IPAddressList {
  private final List<InetAddress> ips =
    Collections.synchronizedList(new ArrayList<InetAddress>());
  public List<InetAddress> getList() {
    return ips; // package-private な可視性のための防御的なコピーは必要ではない
  }

  public void addIPAddress(InetAddress address) {
    ips.add(address);
  }
}

class PrintableIPAddressList extends IPAddressList {
  public void addAndPrintIPAddresses(InetAddress address) {
    synchronized(getList()) {
      addIPAddress(address);
      InetAddress[] ia = (InetAddress[]) getList().toArray(new InetAddress[0]);
      // ...
    }
  }
}
```

ガイドライン「LCK00-J. 信頼できないコードから使用されるクラスの同期化には private final ロックオブジェクトを使用する」で推奨されるように、IPAddressList クラスを、private final 宣言したロックオブジェクトを用いたブロック同期を行うように変更した場合、PrintableIPAddressList サブクラス自身に変更はなくても、ロック方式は不適切になってしまう。また、Collections.synchronizedList()のようなラッパーが使用される場合、クラス拡張を行うためにラップされたクラスを判定することは難しい [Goetz 2006]。

3.12.4. 適合コード (コンポジション)

以下の適合コードでは、IPAddressList クラスのオブジェクトをラップして、オブジェクトの状態操作を可能とする同期化されたメソッドを提供している。

コンポジションは、最小のオーバーヘッドとともにカプセル化の利点を提供する。

```
// Class IPAddressList への変更はない
class PrintableIPAddressList {
  private final IPAddressList ips;
  public PrintableIPAddressList(IPAddressList list) {
    this.ips = list;
  }
  public synchronized void addIPAddress(InetAddress address) {
    ips.addIPAddress(address);
  }

  public synchronized void addAndPrintIPAddresses(InetAddress address) {
    addIPAddress(address);
    InetAddress[] ia = (InetAddress[]) ips.getList().toArray(new InetAddress[0]);
    // ...
  }
}
```

この場合、コンポジションにより、PrintableIPAddressList クラスが、基となるリストクラスのロックとは無関係にそれ自身の固有ロックを使用することを可能にしている。PrintableIPAddressList ラッパーが、同期用のロックオブジェクトとして自身を使用することにより、基となるコレクションのメソッドへの直接アクセスを防いでいるので、これらのコレクションがスレッドセーフである必要はない。このアプローチでは、基となるクラスが将来的にロックポリシーを変更しても、一貫性あるロックを提供できる [Goetz 2006]。

3.12.5. リスク評価

スレッドセーフなクラスが一貫して自身のロック方式に適合しない場合、クライアントサイドロックを使用すると、データの不整合およびデッドロックの原因となりうる。

ガイドライン	深刻度	可能性	修正コスト	優先度	レベル
LCK11-J	低	中	中	P4	L3

3.12.6. 参考文献

[Goetz 2006]	Section 4.4.1, “Client-side Locking” Section 4.4.2, “Composition” Section 5.2.1, “ConcurrentHashMap”
[Lee 2009]	“Map & Compound Operation”
[Oaks 2004]	Section 8.2, “Synchronization and Collection Classes”
[Sun 2009b]	Class Vector, Class WeakReference, Class ConcurrentHashMap<K,V>

4. スレッド API (THI) ガイドライン

4.1. THI00-J. sleep()、yield()および getState()の各メソッドが同期セマンティックスを持つと想定しない [5]

*Java 言語仕様*の 17.9 節「Sleep and Yield」によれば [Gosling 2005]

> *Thread.sleep と Thread.yield のいずれも同期セマンティックスを保持していないという点に注意する必要がある。特にコンパイラは Thread.sleep や Thread.yield の呼出し前にレジスタにキャッシュされている内容を共有メモリに書き出す必要がなく、また、Thread.sleep や Thread.yield の呼出し後にレジスタにキャッシュされている値を再読込みする必要もない。*

スレッド実行の停止および一時的な休止によって、以下のようなことが起こるものと誤って想定すると、予期せぬ振舞いにつながる。

- レジスタにキャッシュされた内容の書き出し
- 値の再読込み
- 実行再開時の事前発生関係の確立

4.1.1. 違反コード (sleep()メソッド)

以下の違反コードでは、スレッドの実行終了のためのフラグとして、volatile 宣言されていない boolean 型変数である done を使用している。異なるスレッドが shutdown()メソッドを呼ぶことにより、変数 done に true をセットしている。

```
final class ControlledStop implements Runnable {
  private boolean done = false;
  @Override public void run() {
    while (!done) {
      try {
        Thread.sleep(1000);
      } catch (InterruptedException e) {
        Thread.currentThread().interrupt(); // 割込みステータスのリセットを行う
      }
    }
  }
  public void shutdown() {
    this.done = true;
  }
}
```

[5] 2011 年 7 月現在、このガイドラインは削除されている。

しかし、コンパイラは、this.done フィールドを一度読み取った後、ループを繰り返す間、キャッシュした値を再利用することが許されている。したがって、異なるスレッドが this.done の値を変更するために shutdown()メソッドを呼び出しても、while ループは終了しないかもしれない[Gosling 2005]。このエラーは、Thread.sleep()メソッドの呼出しによりキャッシュの内容が再読込みされ反映されるだろうとプログラマが誤って想定することにより発生する。

4.1.2. 適合コード (volatile 変数)

以下の適合コードでは、複数スレッドに対してフラグの状態変更の可視性を確保するために volatile 宣言している。

```
final class ControlledStop implements Runnable {
  private volatile boolean done = false;

  // ...
}
```

フラグを volatile 宣言することで、このスレッドと変数 done に true をセットする他のスレッドとの間に事前発生関係が確立される。

4.1.3. 適合コード (Thread.interrupt()メソッド)

sleep()メソッドを呼び出しているメソッドには、スレッド割込みの使用が最適である。スレッド割込みは、sleep()メソッドにより一時的に停止しているスレッドを直ちに起動して割込みを処理する。

```
final class ControlledStop implements Runnable {
  @Override public void run() {
    while (!Thread.interrupted()) {
      try {
        Thread.sleep(1000);
      } catch (InterruptedException e) {
        Thread.currentThread().interrupt();
      }
    }
  }

  public void shutdown() {
    Thread.currentThread().interrupt();
  }
}
```

4.1.4. 違反コード (getState()メソッド)

以下の違反コードでは、doSomething()メソッド内でスレッドを開始している。このスレッドは、volatile 宣言したフラグをチェックし、割込みによる通知があるまで処理をブロックする。stop()メソッドは、待機状態でブロックしているスレッドへ通知を行うとともにフラグに true をセットし、スレッドを終了させる。

```
public class Waiter {

  private Thread thread;
  private volatile boolean flag;
  private final Object lock = new Object();

  public void doSomething() {
    thread = new Thread(new Runnable() {
      @Override public void run() {
        synchronized(lock) {
          while (!flag) {
            try {
              lock.wait();
              // ...
            } catch (InterruptedException e) {
              // ハンドラへの転送
            }
          }
        }
      }
    });
    thread.start();
  }

  public boolean stop() {
    if (thread != null) {
      if (thread.getState() == Thread.State.WAITING) {
        flag = true;
        synchronized (lock) {
          lock.notifyAll();
        }
        return true;
      }
    }
    return false;
  }
}
```

残念ながら stop()メソッドで通知を送信する前に、スレッドがブロックされていて、かつ、終了していないかどうかチェックするために、誤って Thread.getState()メソッドを使用している。スレッドが待機状態でブロックしているかどうかチェックするような、同期処理制御のために Thread.getState()メソッドを使用するのは不適当である。なぜなら、JVM がスピンウェイトによりブロッキングを実装している場合、ブロックされたスレッドが常に

WAITING か TIMED_WAITING の状態になるとは限らない [Goetz 2006]。この違反コードでは、stop()メソッドでスレッドが WAITING 状態にならない限り、スレッドを終了できない。

4.1.5. 適合コード

以下の適合コードでは、スレッドの状態が WAITING かどうか判定するためのチェック処理を削除している。wait()メソッドによりブロックされてないスレッドに対して notifyAll()メソッドを呼び出しても弊害はないため、このチェックを削除できる。

```
public class Waiter {
  // ...

  public boolean stop() {
    if (thread != null) {
      flag = true;
      synchronized (lock) {
        lock.notifyAll();
      }
      return true;
    }
    return false;
  }
}
```

4.1.6. リスク評価

同期処理制御を Thread クラスの sleep()、yield()および getState()の各メソッドに頼っていると、予期せぬ振舞いを引き起こす場合がある。

ガイドライン	深刻度	可能性	修正コスト	優先度	レベル
THI00-J	低	中	中	P4	L3

4.1.7. 参考文献

[Gosling 2005]	Section 17.9, “Sleep and Yield”

4.2. THI01-J. ThreadGroup クラスのメソッドを使用しない

Java の各スレッドは、生成と同時に一つのスレッドグループに割り当てられる。スレッドグループは、java.lang.ThreadGroup クラスにより実装されている。スレッドグループ名を明示的に指定しない場合、JVM がデフォルトのグループである main を割り当てる [Sun 2008a]。 ThreadGroup クラスのメソッドは、あるスレッドグループに属するすべてのスレッドを一度に操作するのに便利である。たとえば、ThreadGroup.interrupt()メソッドにより、スレッドグループ内のスレッドをすべて中断できる。また、スレッドグループは、各スレッドをグループにまとめ、異なるグループに属するスレッド同士がお互い干渉しないようにするなど、多層的なセキュリティ強化にも使用できる[Oaks 2004]。

スレッドグループはスレッドを整理するのに役立つが、ThreadGroup クラスのメソッドの多く(たとえば allowThreadSuspension()、resume()、stop()、suspend()など)は非推奨であり、プログラマがスレッドグループを有効に利用できるケースは非常に少ない。その上、他の非推奨とされていない多くのメソッドも、旧式で望ましい機能性をほとんど提示できない。皮肉にも、ThreadGroup クラスのメソッドの中にはスレッドセーフでないものさえ存在する [Bloch 2001]。

セキュアではないが非推奨とされていないメソッドとしては、以下のものが含まれる。

- ThreadGroup.activeCount()
  Java API の activeCount()メソッドの説明によれば [Sun 2009b]

  > *スレッドグループ内のアクティブスレッドのおよその数を返す。*
  >
  > このメソッドは、（ThreadGroup.enumerate()などで）スレッドを列挙する前に必要な配列のサイズを決定するために使用される。スレッドは開始されていない状態でも、スレッドグループ内に存在しているので、アクティブスレッドとしてカウントされる。その上、アクティブスレッド数のカウントは、特定のシステムスレッドの存在にも影響を受ける [Sun 2009b]。したがって、*activeCount()* メソッドの結果は、スレッドグループ内で実行中のタスクの正確な数を示さないこともある。

- ThreadGroup.enumerate()
  Java API の ThreadGroup クラスの文書によれば [Sun 2009b]

  > *[enumerate()メソッドは] スレッドグループとそのサブグループ内の各アクティブスレッドを指定された配列にコピーする。アプリケーションは、望ましい*

> *配列のサイズを得るために activeCount メソッドを使用すべきである。配列が小さすぎてすべてのスレッドを保持できない場合、配列に入らないスレッドは通知なしに無視される。*

スレッドを終了するために ThreadGroup API を使用することにも落とし穴がある。stop() メソッドが非推奨であり、スレッドの停止は、他の方法で行う必要がある。プログラミング言語 Java によれば [Arnold 2006]

> *一つの方法としては、他のスレッドに join することで他のスレッドがいつ終了したかを知った上で、終了処理を開始する方法である。しかし、単に ThreadGroup を検査するのでは、同じグループに終了しないライブラリスレッドが含まれる場合、join から戻れないため、アプリケーションは、自身が生成したスレッドのリストを保持する必要があるかもしれない。*

Executor フレームワークはスレッドの論理的なグループを管理する、より優れた API を提供しており、スレッド終了処理および例外を扱うための安全な仕組みを提示している [Bloch 2008]。

4.2.1. 違反コード

以下の違反コードは、controller と名付けられたスレッドを保持する NetworkHandler クラスを含んでおり、controller スレッドは、ワーカースレッドに新しいリクエストの処理を委譲する。controller スレッドは、競合状態を意図的に発生させるために run()メソッド呼出しで、三つのスレッドを連続して開始させ、それぞれのリクエストを処理する。すべてのスレッドは、スレッドグループ Chief に属する。

```
final class HandleRequest implements Runnable {
  public void run() {
    // 実行したい内容を記述する
  }
}

public final class NetworkHandler implements Runnable {
  private static ThreadGroup tg = new ThreadGroup("Chief");
  @Override public void run() {
    new Thread(tg, new HandleRequest(), "thread1").start(); // スレッド 1 を開始する
    new Thread(tg, new HandleRequest(), "thread2").start(); // スレッド 2 を開始する
    new Thread(tg, new HandleRequest(), "thread3").start(); // スレッド 3 を開始する
  }

  public static void printActiveCount(int point) {
    System.out.println("Active Threads in Thread Group " + tg.getName() +
      " at point(" + point + "):" + " " + tg.activeCount());
  }

  public static void printEnumeratedThreads(Thread[] ta, int len) {
    System.out.println("Enumerating all threads...");
    for(int i = 0; i < len; i++) {
      System.out.println("Thread " + i + " = " + ta[i].getName());
    }
  }

  public static void main(String[] args) throws InterruptedException {
    // controller スレッドを開始する
    Thread thread = new Thread(tg, new NetworkHandler(), "controller");
    thread.start();

    Thread[] ta = new Thread[tg.activeCount()]; // アクティブスレッド数を取得する(安全ではない)

    printActiveCount(1); // P1
    Thread.sleep(1000); // TOCTOU 条件を例示するために遅延する(競合ウィンドウ)
    printActiveCount(2); // P2: 新規スレッドを開始するのでスレッド数は変化する
    // P1 の時点で得たスレッド数(既に変わってしまっている)を誤って使用している
    int n = tg.enumerate(ta);
    printEnumeratedThreads(ta, n); // 新規に開始したスレッドを暗黙裡に無視している
    // (P1 と P2 の中間)

    // 以下のコードでは、生きているスレッドをまったく含んでいない場合はスレッドグループを破壊する
    for (Thread thr : ta) {
      thr.interrupt();
      while(thr.isAlive());
    }
    tg.destroy();
  }
}
```

スレッド数のカウント結果の取得とスレッドのリストへの列挙は、合わせてアトミックな操作として構成されておらず、上記の実装においては time-of-check-to-time-of-use (TOCTOU)脆弱性が存在する。main()メソッドにおいて、activeCount()メソッドの呼出し後と enumerate()メソッドの呼出しの間に新規リクエストが発生した場合、グループ内のスレッド総数は増加するが、スレッドを列挙したリストである ta には、はじめに割り当てたスレッドの数、即ち、二つのスレッド参照(main および controller)のみを含むことになる。したがって、プログラムは、Chief スレッドグループ内で新規に開始したスレッドを取り扱うことはできない。

また、全スレッドを含んでいない状態の配列 ta を継続して使用することは安全ではない。たとえば、スレッドグループおよびそのサブグループを破壊するために destroy()メソッドを呼び出す処理は、期待通りに動作しないであろう。destroy()メソッド呼び出しの前提条件は、スレッドグループ内に実行中のスレッドが存在しないということである。上記コードでは、スレッドグループ内の全スレッドを中断することにより、この前提条件を達成しようしている。しかし、destroy()メソッドを呼出す時点では、スレッドグループは空ではなく、java.lang.IllegalThreadStateException がスローされる。

4.2.2. 適合コード

以下の適合コードでは、三つのタスクをグループ化するために ThreadGroup ではなく、固定長のスレッドプールを使用している。java.util.concurrent.ExecutorService インターフェースは、スレッドプールを管理するメソッドを提供するが、アクティブなスレッド数をカウントするメソッドや、それらを列挙するメソッドは提供されない。しかし、論理的なグループ化により、全体的なグループの振舞いの制御が可能になる。たとえば、特定のスレッドプールに属するスレッドはすべて、shutdownPool()メソッドを呼び出すことにより終了させることができる。

```
public final class NetworkHandler {
  private final ExecutorService executor;

  NetworkHandler(int poolSize) {
    this.executor = Executors.newFixedThreadPool(poolSize);
  }
  public void startThreads() {
    for(int i = 0; i < 3; i++) {
      executor.execute(new HandleRequest());
    }
  }
  public void shutdownPool() {
    executor.shutdown();
  }

  public static void main(String[] args) {
    NetworkHandler nh = new NetworkHandler(3);
    nh.startThreads();
    nh.shutdownPool();
  }
}
```

Java SE 5.0 以前は、他のスレッドが未捕捉例外（uncaught exception）をキャッチする直接的な方法が存在しなかったので、ThreadGroup クラスを継承する必要があった。アプリケーションで UncaughtExceptionHandler を実装する場合、ThreadGroup のサブクラス化によってのみこれを制御することが可能であったためである。最近のバージョンでは、Thread クラスに定義されたインターフェースを使用することにより、UncaughtExceptionHandler をスレッド単位で実装できるため、ThreadGroup クラスにはほとんどユニークな機能性が残っていない [Goetz 2006, Bloch 2008]。

スレッドプールにおける未捕捉例外ハンドラの使用の詳細に関しては、ガイドライン「TPS03-J. スレッドプールで実行されるタスクの異常終了を通知する」を参照。

4.2.3. リスク評価

ThreadGroup API の使用は、競合状態、メモリリークおよび一貫性を欠いたオブジェクト状態という結果をもたらす可能性がある。

ガイドライン	深刻度	可能性	修正コスト	優先度	レベル
THI01-J	低	中	中	P4	L3

4.2.4. 参考文献

[Arnold 2006]	Section 23.3.3, “Shutdown Strategies”
[Bloch 2001]	“Item 53: Avoid thread groups”
[Bloch 2008]	“Item 73: Avoid thread groups”
[Goetz 2006]	Section 7.3.1, “Uncaught Exception Handlers”
[Oaks 2004]	Section 13.1, “ThreadGroups”
[Sun 2009b]	Methods activeCount and enumerate, Classes ThreadGroup and Thread
[Sun 2008a]	
[Sun 2008b]	Bug ID: 4089701 and 4229558

4.3. THI02-J. Thread.run()メソッドを直接呼び出さない [6]

スレッドが確実に正しく開始されることは重要である。スレッドの開始で犯しやすい誤りとして、実行を開始したコードが正しく動作しているように見えても、実際は意図しないスレッドでコードが実行されてしまうことがある。

Thread.start()メソッドは、各スレッドの run()メソッドを実行する。Thread オブジェクトから run()メソッドを直接呼び出すことは誤りである。直接呼び出した場合、run()メソッドのステートメントは、新規に生成されたスレッドではなく呼出し元のスレッド内で実行される。また、Thread オブジェクトが、Runnable オブジェクトから生成されるのではなく、run()メソッドをオーバーライドしていない Thread のサブクラスのインスタンス化により生成される場合、サブクラスの run()メソッドの呼出しで Thread.run()メソッドが呼び出されるが、何の処理も実行されない。

4.3.1. 違反コード

以下の違反コードでは、実行中のスレッドが明示的に run()メソッドを呼び出している。

```
public final class Foo implements Runnable {
  @Override public void run() {
    // ...
  }
  public static void main(String[] args) {
    Foo foo = new Foo();
    new Thread(foo).run();
  }
}
```

run()メソッドは新しいスレッドを開始しないので、新しいスレッド上で start()メソッドは実行されない。その結果、run()メソッドのステートメントは、新しいスレッドではなく、呼出し元スレッドにおいて実行される。

4.3.2. 適合コード

以下の適合コードでは、新しいスレッドを開始するために start()メソッドを正しく使用している。start()メソッドは、内部的に新規スレッドで run()メソッドを呼び出す。

6 2011 年 7 月現在、このガイドラインの識別子は「THI00-J」に変更されている。

```
public final class Foo implements Runnable {
  @Override public void run() {
    // ...
  }
  public static void main(String[] args) {
    Foo foo = new Foo();
    new Thread(foo).start();
  }
}
```

4.3.3. 例外

THI02-EX1:run()メソッドを呼び出して、単体試験を実施することは考えられる。ただし、複数のスレッドで使用するクラスをテストするためにこの方法を使用することはできない点に注意すること。

Runnable オブジェクトを引数に構築された Thread オブジェクトに対して Thread.run()メソッドを実行する場合、Thread オブジェクトを Runnable にキャストすることで、アナライザによる検出を回避できる。

```
Thread thread = new Thread(new Runnable() {
   @Override public void run() {
      // ...

});

((Runnable) thread).run(); // 例外: 新規スレッドは開始しない
```

Runnable にキャストして run()メソッドを呼び出すことで、Thread.run()メソッドの明示的な呼出しを意図的に行っていることがはっきりする。この際、run()メソッドの呼出しに関する説明コメントを追記することが強く推奨される。

4.3.4. リスク評価

スレッドを正しく開始できないと、予期せぬ振舞いを引き起こす場合がある。

ガイドライン	深刻度	可能性	修正コスト	優先度	レベル
THI02-J	低	中	中	P4	L3

4.3.5. 参考文献

[Sun 2009b]	Interface Runnable and class Thread

4.4. THI03-J. wait()および await()メソッドは、常にループ内部で呼び出す

Object.wait()メソッドは、一時的にロックの所有権を放棄するので、ロックを要求している他のスレッドが処理を進めることができる。Object.wait()メソッドは、synchronized ブロックあるいは synchronized メソッドから呼び出す必要があり、また、待機状態スレッドを再開させるには、notify()メソッドにより通知を行わなければならない。更に、wait()メソッドの呼出しは、条件述語（呼出しの事前条件）が真であるかどうかをチェックするループ処理内で行われるべきである。条件述語は、ループ処理を行う条件を反転したものであることに注意すること。たとえば、ベクターから要素を取り除く際の条件述語は「!isEmpty()」であるが、while ループの条件式は「isEmpty()」である。ベクターが空の場合に wait()メソッドを呼び出す正しい方法を、以下に示す。

```
public void consumeElement() throws InterruptedException {
  synchronized (vector) {
    while (vector.isEmpty()) {
      vector.wait();
    }

    // 条件が真である間は処理を行う
  }
}
```

通知メカニズムによって待機スレッドへの通知が行われ、待機スレッドは条件述語のチェックを行う。別スレッドでの notify()あるいは notifyAll()メソッドの呼出しでは、どの待機スレッドが再開するかを正確に特定できない。よって、通知を受信したスレッド中、再開する条件を満たしたスレッドのみが再開されるよう、条件述語ステートメントを記述する。処理の実行前に行われる入力ストリームからのデータ読取りのように、条件が真になるまで、スレッドをブロックする必要がある場合にも条件述語が有効である。

wait/notify を使用する場合、安全性（safety）と生存性（liveness）への配慮は重要である。安全性とは、すべてのオブジェクトが、マルチスレッド環境下で首尾一貫した状態を保持することである[Lea 2000a]。生存性とは、すべての操作あるいはメソッドの呼出しが、中断せずに完了することである。

生存性を保証するために、while ループ条件は wait()メソッドを呼び出す前にテストする必要がある。これは、他のスレッドが、先行して条件述語を満たして通知を送信する場合に備えて行うものである。通知が送信された後に wait()メソッドを呼び出すと、無期限のブロッキングという結果になりうる。

安全性を保証するためには、wait()メソッドの呼出し後にも while ループ条件をテストする必要がある。wait()メソッドの目的は、通知を受信するまで無期限にブロックを行うことであるが、以下の脆弱性を回避するためにループ内に記述する必要がある[Bloch 2001]。

- 中間スレッド - あるスレッドが通知を送信してから、通知を受信したスレッドが実行を再開するまでの間に、第三のスレッドが共有オブジェクトのロックを獲得してしまう可能性がある。このスレッドは、オブジェクトの状態を変更し矛盾した状態とすることができる。これは、time-of-check-to-time-of-use (TOCTOU)脆弱性である。
- 悪意ある通知 - 条件述語が偽の時に、任意に送信される通知を受信するかもしれない。これは、wait()メソッドの呼出しが通知により無効になることを意味する。
- 通知の誤配信 - notifyAll()メソッドからの通知により、無関係なスレッドが条件述語が真の状態で再開される可能性もある。結果的に、本来は休止状態を維持しているべきスレッドが、実行を再開するかもしれない。
- 見せかけの起動（spurious wake-up）- JVM の実装の中には、通知がなくとも待機状態のスレッドを開始してしまう、見せかけの起動が発生しうるものが存在する[Sun 2009b]。

これらの理由から、条件述語は、wait()メソッドを呼び出した後にもチェックする必要がある。wait()メソッドの呼出し前後に条件述語をチェックするためには、while ループが最適な選択である。

同様に、Condition インターフェースの await()メソッドもループ内部で呼び出す必要がある。Java API では、Condition インターフェースについて、以下のように記述している[Sun 2009b]。

> *プラットフォームへの依存性を配慮し Condition の待機中「見せかけの起動」が許されている。Condition は、常にループ上で待機させて、待機対象の条件述語を確認する形で使用するべきなので、大抵のアプリケーションプログラムでは、見せかけの起動による実質的な影響はほとんどない。JVM の実装において見せかけの起動が発生しないようにするのは自由だが、アプリケーションプログラマは、見せかけの起動が発生しうることを前提にループ上で常に待機する対応を取ることが推奨される。*

これから新たに作成するソースコードであれば、wait/notify の仕組みの代わりに、java.util.concurrent パッケージの並行処理用ユーティリティを使用するべきである。しかし、レガシーコードは wait/notify の仕組みに依存しているかもしれない。

4.4.1. 違反コード

以下の違反コードでは、wait()メソッドを if ブロック内で呼び出しており、通知を受信した後の事後条件のチェックが行われていない。通知が意図されていないあるいは悪意がある場合、条件述語が成立していない状態でスレッドを再開させてしまう可能性がある。

```
synchronized (object) {
  if (<条件が成立しない場合>) {
    object.wait();
  }
// 条件が成立したときに実行される
}
```

4.4.2. 適合コード

以下の適合コードでは、while ループの内部で wait()メソッドを呼ぶことにより、wait()メソッド呼出しの前後で条件をチェックしている。

```
synchronized (object) {
  while (<条件が成立しない場合>) {
    object.wait();
  }
  // 条件が成立したときに実行される
}
```

同様に、java.util.concurrent.locks.Condition インターフェースの await()メソッドの呼出しもループの内部で行う必要がある。

4.4.3. リスク評価

生存性および安全性を保証するため、wait()メソッドと await()メソッドは、常に while ループの内部で呼び出す必要がある。

ガイドライン	深刻度	可能性	修正コスト	優先度	レベル
THI03-J	低	低	中	P2	L3

4.4.4. 参考文献

	Item 50: “Never invoke wait outside a loop”
[Goetz 2006]	Section 14.2, “Using Condition Queues”
[Lea 2000a]	Section 3.2.2, “Monitor Mechanics” Section 1.3.2, “Liveness”
[Sun 2009b]	Class Object

4.5. THI04-J. 一つではなく、すべての待ち状態スレッドへ通知する [7]

wait()メソッドを呼び出すスレッドは、その条件述語が真になったとき、待機状態から復帰して実行を再開することになる。待機状態のスレッドは、ガイドライン「THI03-J. wait()および await()メソッドは、常にループ内部で呼び出す」に従って、通知を受信したら条件述語をテストして、結果が偽の場合には再度待機状態に戻る必要がある。

java.lang.Object クラスの notify()および notifyAll()メソッドは、待機中のスレッドを再開するために使用される。これらのメソッドは、待機状態のスレッドと同じオブジェクトの固有ロックを保持するスレッドにより、呼び出される必要がある。これらのメソッドを呼び出すスレッドが、待機状態のスレッドと同じオブジェクトの固有ロックを保持していない場合、IllegalMonitorStateException 例外がスローされる。notifyAll()メソッドは、全スレッドを待機状態から復帰させ、条件述語が真であるスレッドの実行再開を可能にする。待機状態に入る前に特定のロックを保持していたとしても、再開されたスレッドの中で一つのスレッドだけが、通知受信時にそのロックを再度獲得する。その結果、他のスレッドは再度待機することになる。notify()メソッドは、一つのスレッドを待機状態から復帰させるが、どのスレッドへ通知が行われるかは保証されない。一つの復帰したスレッドの条件述語がスレッドの処理続行を許していない場合、このスレッドは、通知側の意図に反して、再度待機するかもしれない。

notify()メソッドは、下記条件がすべて満たされた場合のみ使用すべきである。

- 各待機スレッドにおいて、条件述語の内容は等しい。
- 全スレッドは、復帰後に同じ一連の操作を実行する。これは、notify()メソッドの呼出しによって復帰し処理を再開するスレッドが、どのスレッドでもよいことを意味する。
- 一つのスレッドのみが通知により復帰することが要求されている。

処理内容が等しく、ステートレスなサービスやユーティリィティを提供するスレッドは、これらの条件を満たしている。

java.util.concurrent.locks パッケージの Condition インターフェースは、await()メソッドの呼出しでブロックしているスレッドを復帰させるために signal()および signalAll()のメソッドを提供している。これらのメソッドを利用するには、Lock オブジェクトと関係付けられた Condition オブジェクトが必要となる。なお、Lock オブジェクトと wait()/notify()メソッドの組み合わせは可能ではある。しかし、Lock オブジェクトを用いて同期化を行うコードは、

[7] 2011 年 7 月現在、このガイドラインの識別子は「THI02-J」に変更されている。

それ自身の固有ロックは使用せずに、代わりに Lock オブジェクトと関係付けされた一つ以上の Condition オブジェクトを使用する。これらの Condition オブジェクトは、Lock オブジェクトによって適用されるロックポリシーに沿って相互に影響しあう。したがって、Object.wait()、Object.notify()および Object.notifyAll()の各メソッドの代わりに Condition.await()、Condition.signal()および Condition.signalAll()が使用できる。

複数のスレッドが同一の Condition オブジェクトを待ち受けている状況で signal()メソッドを安全に使用するには、下記の状況を満たす必要がある。

- 各待機スレッドは、同一の Condition オブジェクトを使用している。
- 全スレッドは、復帰後に同じ一連の操作を実行する。これは、signal()メソッドの呼出しによって、復帰し処理を再開するスレッドが、どのスレッドでもよいことを意味する。
- 一つのスレッドのみがシグナルにより復帰することが要求されている。

あるいは、下記の条件を満たす場合、signal()メソッドを使用できる。

- 各スレッドが、それぞれユニークな Condition オブジェクトを使用する。
- 各 Condition オブジェクトが、一つの共通 Lock オブジェクトと関連付けられている。

適切に使用すれば、signal()メソッドは signalAll()メソッドよりも性能面で優れている。

4.5.1. 違反コード (notify())

以下の違反コードでは、複数のスレッドにより実行される一連の処理ステップで構成された複雑なプロセスを処理している。各スレッドは、time フィールドの値に対応する処理ステップを実行する。各スレッドは、time フィールドが対応するステップを示す内容に更新されるのを待った後、担当ステップを実行する。次に time をインクリメントして、後続のステップの実行権限を有するスレッドへ通知を行う。

```
public final class ProcessStep implements Runnable {
  private static final Object lock = new Object();
  private static int time = 0;
  private final int step; // time フィールドがこの値に達したら操作を実行する

  public ProcessStep(int step) {
    this.step = step;
  }

  @Override public void run() {
    try {
      synchronized (lock) {
        while (time != step) {
          lock.wait();
        }
        // 操作を実行する
        time++;
        lock.notify();
      }
    } catch (InterruptedException ie) {
      Thread.currentThread().interrupt(); // 割込みステータスをリセットする
    }
  }

  public static void main(String[] args) {
    for (int i = 4; i >= 0; i--) {
      new Thread(new ProcessStep(i)).start();
    }
  }
}
```

上記のコードは、生存性要件を満たしていない。各スレッドはそれぞれ異なる step の値を実行前提条件としているので、各スレッドの条件述語は異なっている。Object.notify()メソッドは、一度に一つのスレッドしか復帰させないので、復帰したスレッドが幸運にも次に実行予定のステップを担当するスレッドでない限り、プログラムはデッドロックに陥る。

4.5.2. 適合コード (notifyAll())

以下の適合コードでは、各スレッドは担当するステップを処理後、続いて、待機スレッドへ通知を行うために notifyAll()メソッドを呼んでいる。条件述語が真となるスレッドは、その時点で自身の担当ステップを実行することが可能であるが、条件述語の判定結果が偽(ループ条件式の判定結果としては真である)のスレッドは、待機状態に戻る。

前述の違反コードの run()メソッドだけを、以下のように修正する。

```
@Override public void run() {
  try {
    synchronized (lock) {
      while (time != step) {
        lock.wait();
      }
      // 操作を実行する
      time++;
      lock.notifyAll(); // notify()メソッドの代わりに notifyAll()メソッドを使用している
    }
  } catch (InterruptedException ie) {
    Thread.currentThread().interrupt(); // 割込みステータスをリセットする
  }
}
```

4.5.3. 違反コード (Condition インターフェース)

以下の違反コードは、notify()メソッドを使用した違反コードに類似しているが、待機および通知のために Condition インターフェースを使用している。

```
public class ProcessStep implements Runnable {
  private static final Lock lock = new ReentrantLock();
  private static final Condition condition = lock.newCondition();
  private static int time = 0;
  private final int step; // time フィールドがこの値に達したら操作を実行する

  public ProcessStep(int step) {
    this.step = step;
  }

  @Override public void run() {
    lock.lock();
    try {
      while (time != step) {
        condition.await();
      }
      // 操作を実行する
      time++;
      condition.signal();
    } catch (InterruptedException ie) {
      Thread.currentThread().interrupt(); // 割込みステータスをリセットする
    } finally {
      lock.unlock();
    }
  }
  public static void main(String[] args) {
    for (int i = 4; i >= 0; i--) {
      new Thread(new ProcessStep(i)).start();
    }
  }
}
```

Object.notify()メソッドと同様に、signal()メソッドの呼出しでどのスレッドが復帰するかは分からない。

4.5.4. 適合コード (signalAll())

以下の適合コードでは、すべての待機スレッドへの通知を行う signalAll()メソッドを使用している。各スレッドはそれぞれ await()メソッドから戻る前に Condition オブジェクトに関連付けされたロックを再び取得する。復帰したスレッドは、このロックを保持していることが保証されている [Sun 2009b]。条件述語が真となる状態のスレッドはそのタスクを実行することが可能であるが、条件述語が偽である他のスレッドはすべて待機状態に戻る。

前述の違反コードの run()メソッドのみが、以下のように修正されている。

```
@Override public void run() {
  lock.lock();
  try {
    while (time != step) {
      condition.await();
    }
    // 操作を実行する
    time++;
    condition.signalAll();
  } catch (InterruptedException ie) {
    Thread.currentThread().interrupt(); // 割込みステータスをリセットする
  } finally {
    lock.unlock();
  }
}
```

4.5.5. 適合コード (スレッド毎にユニークな Condition オブジェクト)

以下の適合コードでは、各スレッドに独自の Condition オブジェクトを割り当てている。すべての Condition オブジェクトは、全スレッドからアクセスすることが可能である。

```
// コンストラクタが例外を返すので、このクラスでは final 宣言している
public final class ProcessStep implements Runnable {
  private static final Lock lock = new ReentrantLock();
  private static int time = 0;
  private final int step; // time フィールドがこの値に達したら操作を実行する
  private static final int MAX_STEPS = 5;
  private static final Condition[] conditions = new Condition[MAX_STEPS];

  public ProcessStep(int step) {
    if (step <= MAX_STEPS) {
      this.step = step;
      conditions[step] = lock.newCondition();
    } else {
      throw new IllegalArgumentException("Too many threads");
    }
  }

  @Override public void run() {
    lock.lock();
    try {
      while (time != step) {
        conditions[step].await();
      }
      // 操作を実行する
      time++;
      if (step + 1 < conditions.length) {
        conditions[step + 1].signal();
      }
    } catch (InterruptedException ie) {
      Thread.currentThread().interrupt(); // 割込みステータスをリセットする
    } finally {
      lock.unlock();
    }
  }

  public static void main(String[] args) {
    for (int i = MAX_STEPS - 1; i >= 0; i--) {
      ProcessStep ps = new ProcessStep(i);
      new Thread(ps).start();
    }
  }
}
```

signal()メソッドが使用されているが、条件述語が特定の Condition オブジェクトに対応したスレッドだけを復帰できる。

信頼できないコードがこのクラスのインスタンスでスレッドを作成することができない場合に限り、この適合コードは安全である。

4.5.6. リスク評価

すべての待機スレッドではなく、一つのスレッドだけに通知を行うことは、システムの生存性を脅かすかもしれない。

ガイドライン	深刻度	可能性	修正コスト	優先度	レベル
THI04-J	低	低	中	P2	L3

4.5.7. 参考文献

[Bloch 2001]	Item 50: “Never invoke wait outside a loop”
[Goetz 2006]	Section 14.2.4, “Notification”
[Gosling 2005]	Chapter 17, “Threads and Locks”
[Sun 2009b]	java.util.concurrent.locks.Condition interface

4.6. THI05-J. スレッドの終了に Thread.stop()メソッドを使用しない

スレッドが正常終了する場合、スレッドはクラスの不変項（オブジェクトが取りうる状態の制約条件）を維持する。しかし、タスクの完了、リクエストの取消し、あるいは、プログラムや JVM を早急に終了する必要がある場合、プログラマはスレッドを不意に終了しようとする。

スレッドの中断、再開および終了をサポートするために、いくつかのスレッド管理用 API が導入されたが、後に、設計上の欠陥があり非推奨とされた。たとえば、Thread.stop()メソッドを呼び出すとスレッド停止につながる ThreadDeath 例外が直ちにスローされる。

Thread.stop()メソッドの呼出しによりスレッドが獲得したロックはすべて解放されるが、その結果オブジェクトが整合性を欠いた状態で残されるかもしれない。オブジェクトが整合性を欠いた状態になることを防ぐために、スレッドは ThreadDeath 例外の捕捉と finally ブロックを使用することができる。しかし、スレッド実行中のいかなる時点においても ThreadDeath 例外が発生しうるので、すべての synchronized メソッドおよび synchronized ブロックを注意深く精査する必要がある。また、catch や finally ブロックの実行中に発生しうる ThreadDeath 例外からコードを保護しなければならない [Sun 1999a]。

非推奨メソッドに関する詳細な情報は、ガイドライン「MET02-J. Do not use deprecated or obsolete methods [8]」に示されている。

8 このガイドラインは、https://www.securecoding.cert.org/confluence/display/java/に記述されている。

4.6.1. 違反コード (非推奨の Thread.stop() メソッド)

以下の違反コードは、ベクターを擬似乱数で埋めるスレッドを示している。指定された時間が経過した後、スレッドを強制的に停止している。

```
public final class Container implements Runnable {
  private final Vector<Integer> vector = new Vector<Integer>(1000);

  public Vector<Integer> getVector() {
    return vector;
  }
  @Override public synchronized void run() {
    Random number = new Random(123L);
    int i = vector.capacity();
    while (i > 0) {
      vector.add(number.nextInt(100));
      i--;
    }
  }

  public static void main(String[] args) throws InterruptedException {
    Thread thread = new Thread(new Container());
    thread.start();
    Thread.sleep(5000);
    thread.stop();
  }
}
```

Vector クラスはスレッドセーフであるので、その共有インスタンスが複数のスレッドにより操作されたとしても、インスタンスは一貫した状態に保たれている。たとえば、Vector.size()メソッドは、ベクターに並行して変更要求が行われても、ベクター内の要素の正しい数を常に返す。これは、ベクターインスタンスの状態を一時的に矛盾させる変更を含む操作を行っている間、他のスレッドからのアクセスを防ぐために自身の固有ロックを使用しているからである。

しかし、Thread.stop()メソッドにより実行中のスレッドは停止し、ThreadDeath 例外がスローされる。また、獲得していたロックはすべて解放される [Sun 2009b]。スレッドが停止された時にベクターに新しい整数の追加処理中である場合、矛盾した状態のベクターが他のスレッドからアクセス可能になるかもしれない。ベクターの要素数は、要素の追加の後にインクリメントされるため、Vector.size()が誤った要素数を返すという結果となりうる。

4.6.2. 適合コード (volatile 変数)

以下の適合コードでは、スレッドを終了するために volatile 宣言したフラグを使用している。shutdown()メソッドにより、done フラグに true をセットしている。スレッドの run()メソッドが done フラグの値を監視し、true になった時点で終了する。

```
public final class Container implements Runnable {
  private final Vector<Integer> vector = new Vector<Integer>(1000);
  private volatile boolean done = false;
  public Vector<Integer> getVector() {
    return vector;
  }
  public void shutdown() {
    done = true;
  }

  @Override public synchronized void run() {
    Random number = new Random(123L);
    int i = vector.capacity();
    while (!done && i > 0) {
      vector.add(number.nextInt(100));
      i--;
    }
  }

  public static void main(String[] args) throws InterruptedException {
    Container container = new Container();
    Thread thread = new Thread(container);
    thread.start();
    Thread.sleep(5000);
    container.shutdown();
  }
}
```

4.6.3. 適合コード (割込み可能)

以下の適合コードでは、スレッドを終了するために、Thread.interrupt()メソッドを main() メソッドから呼び出している。Thread.interrupt()メソッドの呼出しにより、内部的な割込み状態フラグがセットされる。スレッドは、Thread.interrupted()メソッドを使用して、フラグの状態を評価する。スレッドが中断されていた場合には true が返り、割込み状態フラグを初期化する。

```
public final class Container implements Runnable {
  private final Vector<Integer> vector = new Vector<Integer>(1000);
  public Vector<Integer> getVector() {
    return vector;
  }

  @Override public synchronized void run() {
    Random number = new Random(123L);
    int i = vector.capacity();
    while (!Thread.interrupted() && i > 0) {
      vector.add(number.nextInt(100));
      i--;
    }
  }

  public static void main(String[] args) throws InterruptedException {
    Container c = new Container();
    Thread thread = new Thread(c);
    thread.start();
    Thread.sleep(5000);
    thread.interrupt();
  }
}
```

スレッドは、処理の取消しや終了以外のタスクを実行するために、割込みを使用するかもしれない。よって、その割込みポリシーが明確にされていない場合、不注意なスレッドへの割込みは失敗に終わるかもしれず行うべきではない。

4.6.4. 適合コード (stopThread 実行時アクセス権)

セキュリティポリシーファイルから既定の java.lang.RuntimePermission クラスの stopThread の使用許可を削除することによって、Thread.stop()メソッドによるスレッドの停止を防ぐことができる。ただし、このメソッドに依存した信頼できるコードに対して、このアプローチは推奨できない。なぜなら、stopThread の使用許可を削除した結果として生じうる例外を適切に扱うように設計されていないかもしれない。このようなケースでは、

このガイドラインで説明している他の適合コードに対応する代替的なアプローチを実装することが望ましい。

4.6.5. リスク評価

スレッドの強制停止は、矛盾したオブジェクトの状態を招くおそれがある。要求通りにクリーンアップ処理が実行されない場合、重要なリソースがリークする可能性がある。

ガイドライン	深刻度	可能性	修正コスト	優先度	レベル
THI05-J	低	中	中	P4	L3

4.6.6. 参考文献

[Arnold 2006]	Section 14.12.1, “Don’t stop” Section 23.3.3, “Shutdown Strategies”
[Darwin 2004]	Section 24.3, “Stopping a Thread”
[Goetz 2006]	Chapter 7, “Cancellation and shutdown”
[Oaks 2004]	Section 2.4, “Two Approaches to Stopping a Thread”
[Sun 2009b]	Class Thread, method stop, interface ExecutorService
[Sun 2008c]	Concurrency Utilities, More information: Java Thread Primitive Deprecation
[Sun 99]	

4.7. THI06-J. ブロックしているスレッドやタスクが確実に終了できること [9]

ネットワークあるいはファイル入出力などの操作によりブロックするスレッドおよびタスクは、サービス妨害の脆弱性を防ぐためにも、明示的な終了の仕組みを呼出し元に提供する必要がある。

4.7.1. 違反コード (ブロックする入出力、volatile 変数)

以下の違反コードは、ガイドライン「THI05-J. スレッドの終了に Thread.stop()メソッドを使用しない」を参考に volatile 宣言した done フラグを使用してスレッドが終了可能である。しかし、readLine()メソッドの呼出しがネットワーク入出力でブロックする場合、フラグに値をセットしただけではスレッドは終了しない。

[9] 2011 年 7 月現在、このガイドラインの識別子は「THI04-J」に変更されている。

```
public final class SocketReader implements Runnable { // スレッドセーフなクラス
  private final Socket socket;
  private final BufferedReader in;
  private volatile boolean done = false;
  private final Object lock = new Object();

  public SocketReader(String host, int port) throws IOException {
    this.socket = new Socket(host, port);
    this.in = new BufferedReader(new InputStreamReader(this.socket.getInputStream()));
  }
  //一度に一つのスレッドだけがソケットを使用できる
  @Override public void run() {
    try {
      synchronized (lock) {
        readData();
      }
    } catch (IOException ie) {
      // ハンドラへの転送
    }
  }

  public void readData() throws IOException {
    String string;
    while (!done && (string = in.readLine()) != null) {
      // ストリームの終り(null)までスレッドをブロックする
    }
  }

  public void shutdown() {
    done = true;
  }

  public static void main(String[] args) throws IOException, InterruptedException {
    SocketReader reader = new SocketReader("somehost", 25);
    Thread thread = new Thread(reader);
    thread.start();
    Thread.sleep(1000);
    reader.shutdown(); // スレッドを終了する
  }
}
```

4.7.2. 違反コード (ブロックする入出力、割込み可能)

以下の違反コードは前述の違反コードに似ているが、スレッドを終了するためにスレッド割込みを使用している。しかし、java.net.Socket を使用している間は、ネットワーク入出力はスレッド割込みに反応しない。以下のコードでは、前述の違反コードから変更された readData()および main()メソッドを示している。

```
public final class SocketReader implements Runnable { // スレッドセーフなクラス
  // ...
  public void readData() throws IOException {
    String string;
    while (!Thread.interrupted() && (string = in.readLine()) != null) {
      // ストリームの終り(null)までスレッドをブロックする
    }
  }
  public static void main(String[] args) throws IOException, InterruptedException {
    SocketReader reader = new SocketReader("somehost", 25);
    Thread thread = new Thread(reader);
    thread.start();
    Thread.sleep(1000);
    thread.interrupt(); // スレッドへの割込みを行う
  }
}
```

4.7.3. 適合コード (ソケット接続のクローズ)

以下の適合コードでは、shutdown()メソッドでソケットをクローズすることによりブロックされているスレッドを再開する。ソケットをクローズすると readLine()メソッドは SocketException をスローし、スレッドの処理は進行する。スレッドを直ちに正しく停止する必要がある場合、ソケットを接続状態のまま放置できない。

```
public final class SocketReader implements Runnable {
  // ...
  public void readData() throws IOException {
    String string;
    try {
      while ((string = in.readLine()) != null) {
        // ストリームの終り(null)までスレッドをブロックする
      }
    } finally {
      shutdown();
    }
  }
  public void shutdown() throws IOException {
    socket.close();
  }

  public static void main(String[] args) throws IOException, InterruptedException {
    SocketReader reader = new SocketReader("somehost", 25);
    Thread thread = new Thread(reader);
    thread.start();
    Thread.sleep(1000);
    reader.shutdown();
  }
}
```

main()メソッドから shutdown()メソッドが呼ばれると、readData()メソッド内の finally ブロックで再度 shutdown()メソッドが呼ばれ、二度目のソケットクローズが実行される。ソケットが既にクローズされている場合、二度目の呼出しとなるが、プログラムの動作に影響が無い。

非同期入出力を行う場合であれば java.nio.channels.Selector クラスの close()あるいは wakeup()メソッドを呼び出して、ブロック状態のスレッドを復帰させることもできる。

ブロック状態から抜け出した後、更に処理を実行する必要がある場合は、boolean フラグを使用することができる。フラグの使用に対応するコードを追加する場合、スレッドが while ループから抜け出せるように、shutdown()メソッドにおいて、フラグに false をセットするべきである。

4.7.4. 適合コード (割込み可能チャンネル)

以下の適合コードでは、Socket 接続の代わりに、割込み可能チャンネル java.nio.channels.SocketChannel を使用している。スレッドがデータ読取り中に Thread.interrupt()メソッドによってネットワーク入出力の実行割込みが発生した場合、そのスレッドは ClosedByInterruptException を受信し、チャンネルは直ちに閉じられる。また、スレッドの割込み状態も更新される。

```
public final class SocketReader implements Runnable {
  private final SocketChannel sc;
  private final Object lock = new Object();
  public SocketReader(String host, int port) throws IOException {
    sc = SocketChannel.open(new InetSocketAddress(host, port));
  }
  @Override public void run() {
    ByteBuffer buf = ByteBuffer.allocate(1024);
    try {
      synchronized (lock) {
        while (!Thread.interrupted()) {
          sc.read(buf);
          // ...
        }
      }
    } catch (IOException ie) {
      // ハンドラへの転送
    }
  }

  public static void main(String[] args) throws IOException, InterruptedException {
    SocketReader reader = new SocketReader("somehost", 25);
    Thread thread = new Thread(reader);
    thread.start();
    Thread.sleep(1000);
    thread.interrupt();
  }
}
```

この手法により実行中のスレッドを中断できる。このコードでは、スレッドが Thread.interrupted()メソッドにより割込みステータスを取得し、割込みが発生していたら、スレッドを終了する。読取りがブロックする操作であるにもかかわらず、SocketChannel の使用により、割込み受信後直ちに while ループの条件がテストされる。同様に java.nio.channels.Selector を使用してブロックしているスレッドの interrupt()メソッドを呼び出すことでも、スレッドをブロックした状態から復帰できる。

4.7.5. 違反コード (データベース接続)

以下の違反コードは、スレッド毎に一つの JDBC(Java Database Connectivity)接続を作成するスレッドセーフな DBConnector クラスの例である。各 JDBC 接続はそれぞれ一つのスレッドに属しており、他のスレッドとの共有は行われない。JDBC 接続は複数のスレッドによる共有には向いていないため、これは一般的な使用例である。

```
public final class DBConnector implements Runnable {
  private final String query;
  DBConnector(String query) {
    this.query = query;
  }
  @Override public void run() {
    Connection connection;
    try {
      // ユーザ名とパスワードは、簡素化するためにハードコーディングしている
      connection = DriverManager.getConnection(
        "jdbc:driver:name",
        "username",
        "password"
      );
      Statement stmt = connection.createStatement();
      ResultSet rs = stmt.executeQuery(query);
      // ...
    } catch (SQLException e) {
      // ハンドラへの転送
    }
    // ...
  }

  public static void main(String[] args) throws InterruptedException {
    DBConnector connector = new DBConnector("suitable query");
    Thread thread = new Thread(connector);
    thread.start();
    Thread.sleep(5000);
    thread.interrupt();
  }
}
```

データベース接続はソケットと同じく本質的に割込み可能ではない。よって、上記コード例において、結合操作のように実行に時間がかかるクエリでスレッドがブロックされる場合、クライアントが接続リソースをクローズしてタスクをキャンセルすることができない。

4.7.6. 適合コード (Statement.cancel()メソッド)

以下の適合コードでは、データベースへの接続ステートメントを ThreadLocal クラスでラップし、initialValue()メソッドを呼ぶスレッドが一意の接続インスタンスを取得可能としている。このアプローチの利点は、他のスレッドあるいはクライアントが要求がある時に、長期間にわたるクエリを中断することができるように、cancelStatement()メソッドを提供できることである。cancelStatement()メソッドは、Statement.cancel()メソッドを呼び出している。

```
public final class DBConnector implements Runnable {
  private final String query;
  private volatile Statement stmt;
  DBConnector(String query) {
    this.query = query;
    if (getConnection() != null) {
      try {
        stmt = getConnection().createStatement();
      } catch (SQLException e) {
        // ハンドラへの転送
      }
    }
  }
  private static final ThreadLocal<Connection> connectionHolder =
    new ThreadLocal<Connection>() {
    Connection connection = null;

    @Override public Connection initialValue() {
      try {
        // ...
        connection = DriverManager.getConnection(
          "jdbc:driver:name",
          "username",
          "password"
        );
      } catch (SQLException e) {
        // ハンドラへの転送
      }
      return connection;
    }
  };

  public Connection getConnection() {
    return connectionHolder.get();
  }

  public boolean cancelStatement() { // クライアントによるステートメントのキャンセルを許可する
    if (stmt != null) {
      try {
        stmt.cancel();
        return true;
      } catch (SQLException e) {
        // ハンドラへの転送
      }
    }
    return false;
  }

  @Override public void run() {
    try {
      if(stmt == null || (stmt.getConnection() != getConnection())) {
        throw new IllegalStateException();
      }
```

```
        ResultSet rs = stmt.executeQuery(query);
        // ...
      } catch (SQLException e) {
        // ハンドラへの転送
      }
      // ...
  }

  public static void main(String[] args) throws InterruptedException {
      DBConnector connector = new DBConnector("suitable query");
      Thread thread = new Thread(connector);
      thread.start();
      Thread.sleep(5000);
      connector.cancelStatement();
  }
}
```

もしデータベース管理システム(DBMS)およびドライバが両方ともキャンセル処理に対応していれば、Statement.cancel()メソッドによってクエリをキャンセルできる。そうでなければ、このガイドラインに従うことはできない。

Java API、Statement インターフェースのドキュメントによれば [Sun 2009b]

> *デフォルトでは、Statement オブジェクトごとに一つの ResultSet オブジェクトだけが同時にオープンできる。したがって、一つの ResultSet オブジェクトの読取りと並行して、別の ResultSet 読取りが行われる場合、それぞれ異なった Statement オブジェクトによって生成されていなければならない。*

上記の適合コードでは、一つの ResultSet オブジェクトだけが、一つのインスタンスに属する Statement オブジェクトに関連付けられる。したがって、一つのスレッドだけが、クエリの実行結果にアクセスすることができる。

4.7.7. リスク評価

スレッド終了のための仕組みを提供できないと無反応な状態やサービス妨害につながる原因となりうる。

ガイドライン	深刻度	可能性	修正コスト	優先度	レベル
THI06-J	低	中	中	P4	L3

4.7.8. 参考文献

[Arnold 2006]	Section 14.12.1, “Don't stop” Section 23.3.3, “Shutdown Strategies”
[Darwin 2004]	Section 24.3, “Stopping a Thread”
[Goetz 2006]	Chapter 7, “Cancellation and shutdown”
[Oaks 2004]	Section 2.4, “Two Approaches to Stopping a Thread”
[Sun 2009b]	Class Thread, method stop, interface ExecutorService
[Sun 2008c]	Concurrency Utilities, More information: “Java Thread Primitive Deprecation”

5. スレッドプール (TPS) ガイドライン

5.1. TPS00-J. スレッドプールを使用して大量トラフィック発生による急激なサービス低下を防ぐ

多くのプログラムは、連続した入力リクエストを適切に処理する必要がある。各リクエスト毎に新規スレッドを作成する Thread-Per-Message デザインパターンは、最もシンプルな並行処理パターンである [Lea 2000a]。このパターンは、時間を要するか、入出力に拘束されるか、セッション単位か、あるいは独立性の高いタスクなどを順次実行する場合に好まれる。

しかし、このパターンには、スレッド生成およびスケジューリング、タスク管理、リソース割当てと割当て解除のオーバーヘッド、および頻繁なコンテキストスイッチングなどといった問題点がある[Lea 2000a]。たとえば、攻撃者は大量のリクエストをシステムへ一斉送信することにより、サービスを妨害することができる。攻撃を受けたシステムは、パフォーマンスが徐々に低下するのではなく、応答できなくなり、サービス停止状態に陥る。また、安全性を重視して作られたコンポーネントが、断続的に発生するエラーの対処にすべてのリソースを使ってしまい、他のコンポーネントがリソースの飢餓状態で動作できなくなることもありうる。

スレッドプールを利用することで、殺到する入力リクエストに対してすべてのサービスを停止するのではなく、許容される範囲のリクエスト数であれば継続してサービスを提供することが可能となる。スレッドプールは、初期化および処理の実行を並行して行うことが可能なワーカースレッドの最大数を制御することにより、前述の問題に対処する。スレッドプールを提供するオブジェクトは、Runnable あるいは Callable<T> タスクを受入れ、リソースが利用可能になるまでタスクを一時キューに格納する。スレッドプール内のスレッドの再利用や、スレッドの追加削除が効率的に行えるようになり、スレッドライフサイクル管理上のオーバーヘッドは最小限となる。

5.1.1. 違反コード

以下の違反コードでは、Thread-Per-Message デザインパターンを適用している。RequestHandler クラスは public static 宣言されたファクトリメソッドを提供し、呼出し側によるインスタンスの取得を可能にしている。各リクエストをスレッド毎に処理するため handleRequest()メソッドは連続して呼び出される。

```
class Helper {
  public void handle(Socket socket) {
    //...
  }
}

final class RequestHandler {
  private final Helper helper = new Helper();
  private final ServerSocket server;

  private RequestHandler(int port) throws IOException {
    server = new ServerSocket(port);
  }
  public static RequestHandler newInstance() throws IOException {

    return new RequestHandler(0); // 次の利用可能なポートを選択する
  }
  public void handleRequest() {
    new Thread(new Runnable() {
      public void run() {
        try {
          helper.handle(server.accept());
        } catch (IOException e) {
          // ハンドラへの転送
        }
      }
    }).start();
  }

}
```

Thread-Per-Message のパターンでは、急激なサービス低下を防ぐことはできない。一定のリソースを消費し尽くし、処理を正常に継続できない状態に至るまで、スレッドは作成され続けてしまう。たとえば、リクエストに応じた数のスレッドを作成できたとしても、システム上でオープン可能なファイル記述子の数には上限がある。また、システムのメモリが希少である場合、システムは突然不安定になり、サービス停止状態になるかもしれない。

5.1.2. 適合コード

以下の適合コードでは、並行に実行するスレッドの数の上限値を設定する固定スレッドプールを使用している。スレッドプールに依頼したタスクは、内部キューに格納される。これにより、システムがすべてのリクエストを処理しようとして過負荷状態に陥ることを回避し、一度に対応するクライアントの数を限定することにより急激なサービス低下を防いでいる [Sun 2008a]。

```
// Helper クラスに変更はない

final class RequestHandler {
  private final Helper helper = new Helper();
  private final ServerSocket server;
  private final ExecutorService exec;
  private RequestHandler(int port, int poolSize) throws IOException {
    server = new ServerSocket(port);
    exec = Executors.newFixedThreadPool(poolSize);
  }
  public static RequestHandler newInstance(int poolSize) throws IOException {
    return new RequestHandler(0, poolSize);
  }
  public void handleRequest() {
    Future<?> future = exec.submit(new Runnable() {
      @Override public void run() {
        try {
          helper.handle(server.accept());
        } catch (IOException e) {
          // ハンドラへの転送
        }
      }
    });
  }
  // ... スレッドプールの終了やタスクの取消し等の他のメソッドを定義する...
}
```

Executor インターフェースの Java API 文書によると [Sun 2009b]

> *[Executor インターフェースは、]依頼された Runnable タスクを実行するオブジェクトである。このインターフェースは、タスクの依頼を各タスクの実装の詳細 (スレッドの使用やスケジューリングなど)から分離する方法を提供している。通常、Executor は、明示的にスレッドを作成する代わりに使用される。*

上記の適合コードで使用している ExecutorService インターフェースは、java.util.concurrent.Executor インターフェースから派生している。ExecutorService.submit() メソッドは、その呼出し元が Future<V>クラスのオブジェクトを取得することを可能にする。このオブジェクトは、呼出し時点で結果が定まらない非同期処理の結果をカプセル化するとともに、呼出し元がタスクの取消しのような追加的な処理を実行することを可能にする。

サイズ制限のない newFixedThreadPool を選択することが、必ずしも最適とは限らないため、Java API 文書を参照し、以下のいずれかから、実装要件に適切な選択を行うべきである。[Sun 2009b]。

- newFixedThreadPool()
- newCachedThreadPool()
- newSingleThreadExecutor()
- newScheduledThreadPool()

5.1.3. リスク評価

膨大な数のリクエストを処理するために単純過ぎる並行処理方式を適用した場合、深刻なパフォーマンスの低下、デッドロック、あるいはシステムリソースの消耗によりサービスが停止するといった結果を招く可能性がある。

ガイドライン	深刻度	可能性	修正コスト	優先度	レベル
TPS00-J	低	中	高	P2	L3

5.1.4. 参考文献

[Goetz 2006]	Chapter 8, “Applying Thread Pools”
[Lea 2000a]	Section 4.1.3, “Thread-Per-Message” Section 4.1.4, “Worker Threads”
[MITRE 2010]	CWE ID 405, “Asymmetric Resource Consumption (Amplification)” CWE ID 410, “Insufficient Resource Pool”
[Sun 2009b]	Interface Executor
[Sun 2008a]	Thread Pools

5.2. TPS01-J. サイズ制限のあるスレッドプールで相互に依存するタスクを実行しない

プログラマは、サイズ制限のあるスレッドプールを使用して、スレッドプール内で同時に実行可能なスレッド数の上限を指定することができる。ただし、他のタスクの完了に依存するタスクは、サイズ制限のあるスレッドプール内で実行されるべきでない。

スレッド飢餓状態のデッドロック(thread starvation deadlock)とは、プール内で実行中のスレッドすべてが、内部キュー内で開始されないまま待機しているタスクによりブロックされる場合に発生する。実行中のタスクがスレッドプールに他のタスクを依頼し、その完了を待つが、スレッドプールにタスクを格納する空きがないとスレッド飢餓状態のデッドロックとなる。

プログラムの実行に必要なスレッドの数が少なくてすむ場合、問題なく機能しているように見えるので、この問題は見落とされがちである。プールサイズを拡張することにより、この問題を軽減できる場合もあるが、多くの場合、適切なサイズを決定するのは容易ではない。

同様に、二つの実行中のタスクが終了するために、それぞれもう一方の処理の完了をお互いが必要としている場合、スレッドプール内のそれらのスレッドは終了できず、結果的に再利用できないかもしれない。また、サブタスク内でのブロックする操作により、キューが際限なく伸びてしまうこともありうる [Goetz 2006]。

5.2.1. 違反コード (サブタスク間に相互依存関係が存在)

以下の違反コードは、スレッド飢餓状態のデッドロックを引き起こす可能性がある。ValidationService クラスは､ユーザが指定したフィールドがバックエンドのデータベースに存在するかどうかチェックするなどの入力値検査タスクを実行する。

fieldAggregator()メソッドは、可変個の String 引数を受入れ、各引数に対応するタスクを生成して並列処理する。それぞれのタスクは、ValidateInput クラスを使用して、入力値検査を行う。

次に ValidateInput クラスは、SanitizeInput クラスを使用して各リクエスト用のサブタスクを作成し、入力値のサニタイズ（無害化）を行う。これらのすべてのタスクは、同じスレッドプール内で実行される。fieldAggregator()メソッドは、すべてのタスクの実行が終了す

るまでブロックし、すべてのタスクの結果が揃った段階で、これらの結果を含む StringBuilder オブジェクトを呼出し元に返す。

```
public final class ValidationService {
  private final ExecutorService pool;

  public ValidationService(int poolSize) {
    pool = Executors.newFixedThreadPool(poolSize);
  }
  public void shutdown() {
    pool.shutdown();
  }
  public StringBuilder fieldAggregator(String... inputs)
      throws InterruptedException, ExecutionException {
    StringBuilder sb = new StringBuilder();
    Future<String>[] results = new Future[inputs.length]; // 結果を格納する
    for (int i = 0; i < inputs.length; i++) { // タスクをスレッドプールに依頼する
      results[i] = pool.submit(new ValidateInput<String>(inputs[i], pool));
    }

    for (int i = 0; i < inputs.length; i++) { // 結果を集める
      sb.append(results[i].get());
    }
    return sb;
  }
}

public final class ValidateInput<V> implements Callable<V> {
  private final V input;
  private final ExecutorService pool;

  ValidateInput(V input, ExecutorService pool) {
    this.input = input;
    this.pool = pool;
  }

  @Override public V call() throws Exception {
    // 検証に失敗した場合、ここで例外をスローする
    Future<V> future = pool.submit(new SanitizeInput<V>(input)); // サブタスクの生成
    return (V)future.get();
  }
}

public final class SanitizeInput<V> implements Callable<V> {
  private final V input;
  SanitizeInput(V input) {
    this.input = input;
  }

  @Override public V call() throws Exception {
    // 入力情報をサニタイズして返している
    return (V)input;
  }
}
```

たとえば、プールサイズを 6 とする場合、ValidationService.fieldAggregator()メソッドが、検査対象として六つの引数を与えられて呼び出されると、六つのタスクがスレッドプールへ依頼される。更に、各タスクは、入力をサニタイズするために、それぞれ対応するサブタスクを依頼する。この時 SanitizeInput サブタスクを処理するスレッドが実行される必要があるが、スレッドプールの六つのスレッドはすべてブロックされているので、SanitizeInput サブタスクを開始することができない。また、スレッドプールがアクティブなタスクを含んでいる場合、shutdown()メソッドではスレッドプールを終了できない。

シングルスレッド構成の Executor においても、呼出し元がいくつかのサブタスクを生成して結果を待つ場合、スレッド飢餓状態のデッドロックが発生しうる。

5.2.2. 適合コード (タスク間に相互依存が存在しない)

以下の適合コードでは、SanitizeInput タスクが別々のスレッドにおいてではなく、ValidateInput タスクと同じスレッドで実行されるように、ValidateInput<V>クラスを修正している。結果的に、ValidateInput および SanitizeInput タスクは互いが完了するのを待つ必要はなく独立している。また、SanitizeInput クラスは、Callable インターフェースを実装しないように修正されている。

```
public final class ValidationService {
  // ...
  public StringBuilder fieldAggregator(String... inputs)
      throws InterruptedException, ExecutionException {
    // ...
    for (int i = 0; i < inputs.length; i++) {
      // スレッドプールにはそのまま渡さない。
      results[i] = pool.submit(new ValidateInput<String>(inputs[i]));
    }
    // ...
  }
}

// 同じスレッドプールは使用しない
public final class ValidateInput<V> implements Callable<V> {
  private final V input;
  ValidateInput(V input) {
    this.input = input;
  }

  @Override public V call() throws Exception {
    // 検証に失敗した場合、ここで例外をスローする
    return (V) new SanitizeInput().sanitize(input);
  }
}

public final class SanitizeInput<V> { // もはや Callable タスクではない
  public SanitizeInput() {}

  public V sanitize(V input) {
    // 入力情報をサニタイズして返している
    return input;
  }
}
```

スレッド飢餓状態は、スレッドプールのサイズを大きくすることにより軽減することができる。しかし、信頼できない呼出し元が、大量の入力を行うことにより、システムに過大な負荷を与えるかもしれない(ガイドライン「TPS00-J. スレッドプールを使用して大量トラフィック発生による急激なサービス低下を防ぐ」を参照)。

なお、各スレッドは必要なリソースが利用可能になるまでブロックするので、同時に利用可能なデータベース接続数や ResultSet オブジェクト数といったスレッドプールサイズの上限に影響する制約事項に注意する必要がある。

private static 宣言した ThreadLocal 変数を、各スレッドでローカルな状態を保持するために使用することもできる。スレッドプールを使用する場合、ThreadLocal 変数の生存期間は、

対応するタスクの実行中のみに制限すべきである[Goetz 2006]。また、これらの ThreadLocal 変数はタスク間の通信目的で使用されるべきではない。更にスレッドプール内での ThreadLocal 変数の使用に関しては、いくつかの制約がある(ガイドライン「TPS04-J. スレッドプール使用時に ThreadLocal 変数が再初期化済みであることを確実にする」を参照)。

5.2.3. 違反コード (サブタスク)

以下の違反コードは、共有スレッドプールで実行される一連のサブタスクを含んでいる[Gafter 2006]。BrowserManager クラスは perUser()メソッドを呼んでおり、このメソッドは perProfile()メソッドを呼び出すタスクを開始する。perProfile()メソッドは、perTab()メソッドを呼び出すタスクを開始し、引き続き、perTab()メソッドが doSomething()メソッドを呼び出すタスクを開始する。BrowserManager クラスは、これらの一連のタスクが終了するのを待つ。doSomething()メソッドが任意の順序で呼び出され、実行されたメソッド数は変数 count に記録される。

```
public final class BrowserManager {
  private final ExecutorService pool = Executors.newFixedThreadPool(10);
  private final int numberOfTimes;
  private static AtomicInteger count = new AtomicInteger(); // count = 0

  public BrowserManager(int n) {
    numberOfTimes = n;
  }

  public void perUser() {
    methodInvoker(numberOfTimes, "perProfile");
    pool.shutdown();
  }

  public void perProfile() {
    methodInvoker(numberOfTimes, "perTab");
  }

  public void perTab() {
    methodInvoker(numberOfTimes, "doSomething");
  }

  public void doSomething() {
    System.out.println(count.getAndIncrement());
  }

  public void methodInvoker(int n, final String method) {
    final BrowserManager manager = this;
    Callable<Object> callable = new Callable<Object>() {
      @Override public Object call() throws Exception {
        Method meth = manager.getClass().getMethod(method);
        return meth.invoke(manager);
      }
    };

    Collection<Callable<Object>> collection = Collections.nCopies(n, callable);
    try {
      Collection<Future<Object>> futures = pool.invokeAll(collection);
    } catch (InterruptedException e) {
      // ハンドラへの転送
      Thread.currentThread().interrupt(); // 割込みステータスのリセットを行う
    }
    // ...
  }

  public static void main(String[] args) {
    BrowserManager manager = new BrowserManager(5);
    manager.perUser();
  }
}
```

残念ながら、このプログラムはスレッド飢餓状態のデッドロックを引き起こす可能性がある。たとえば、五つの perUser タスクの各々が、五つの perProfile タスクを生成すると、続いて各 perProfile タスクが perTab タスクを生成するので、スレッドプールは枯渇してしまい、perTab()メソッドは、doSomething()メソッドを呼び出すために新たなスレッドを割当てできなくなる。

5.2.4. 適合コード (CallerRunsPolicy クラス)

以下の適合コードでは、実行するタスクを選択しスケジューリングすることでスレッド飢餓状態のデッドロックを回避している。ThreadPoolExecutor に CallerRunsPolicy をセットして、SynchronousQueue を使用している [Gafter 2006]。このポリシーでは、スレッドプールが利用可能なスレッドを使い尽くした場合、後続するタスクの実行はそれらのタスクを依頼したスレッド内で実行されるようになる。

```
public final class BrowserManager {
  private final static ThreadPoolExecutor pool =
    new ThreadPoolExecutor(0, 10, 60L, TimeUnit.SECONDS,
    new SynchronousQueue<Runnable>());
  private final int numberOfTimes;
  private static AtomicInteger count = new AtomicInteger(); // count = 0

  static {
    pool.setRejectedExecutionHandler(
      new ThreadPoolExecutor.CallerRunsPolicy());
  }

  // ...
}
```

Goetz らによると [Goetz 2006]

> *SynchronousQueue は実際にはキューではなく、スレッド間でタスクを引き渡すための仕組みである。SynchronousQueue にタスクを依頼するためには、他のスレッドが引渡しを待っている必要がある。待っているスレッドがなく、現在のプールサイズが最大サイズ未満なら、ThreadPoolExecutor は新しいスレッドを作成する。この条件に該当しない場合、タスクは飽和ポリシーにしたがって拒絶される。*

Java API によれば [Sun 2009b]、CallerRunsPolicy クラスは,

> *execute メソッドで拒否されたタスクを呼出し元スレッドで直接実行させるハンドラである。executor がシャットダウンされている場合、タスクは破棄される。*

上記の適合コードでは、スレッドプールが満杯の場合、引渡し先のタスクを持つタスクは SynchronousQueue に追加される。たとえば、perProfile()に対応するタスクが引渡しを待ち受けているので、perTab()に対応するタスクは SynchronousQueue に追加される。スレッドプールが満杯になると、飽和ポリシー(saturation policy)に従い、新たなタスクは拒否される。CallerRunsPolicy が設定されているため、拒否されたタスクはすべて最初のタスクを開始したメインスレッドで実行される。perTab()に対応するスレッドすべてが実行を完了すると、perUser()のタスクが引渡しを待ち受けているため perProfile()に対応する次のタスク群が SynchronousQueue に追加される。

大量のリクエストを処理する場合でも、CallerRunsPolicy によって作業がスレッドプールから待ち行列に分散されるので、急激なサービス低下を防ぐことができる。依頼されたタスクが他のタスクの完了待ち以外の理由でブロックしなければ、飽和ポリシーは実行中のスレッドが複数のタスクを順番に扱うことを保証する。ただし、もしタスクがネットワーク入出力のような他の理由でブロックされるのであれば、飽和ポリシーでもスレッド飢餓状態のデッドロックを防ぐことはないだろう。なお、SynchronousQueue はタスクを後の実行のために保存するものではないので、キューが無限に伸びてしまうことはない。すべてのタスクは実行中のスレッドあるいはスレッドプール内のスレッドにより処理される。

上記の適合コードは、スレッドスケジューラが予測できない振舞いをした場合、タスクを最適な形でスケジューリングすることはできないかもしれない。しかし、スレッド飢餓状態デッドロックは回避される。

5.2.5. リスク評価

スレッドプール内での相互に依存するタスクを実行すると、サービス停止状態につながることがある。

ガイドライン	深刻度	可能性	修正コスト	優先度	レベル
TPS01-J	低	中	中	P4	L3

5.2.6 参考文献

[Gafter 2006]	A Thread Pool Puzzler
[Goetz 2006]	Section 8.3.2, “Managing queued tasks” Section 8.3.3, “Saturation Policies” Section 5.3.3, “Deques and work stealing”
[Sun 2009b]	

5.3. TPS02-J. スレッドプールに依頼されるタスクが確実に割込み可能であること

スレッドプールの終了やスレッドプール内の個々のタスクの取消しが必要な場合、Thread.interrupt()による割込みに対応しないタスクの依頼を行うべきではない。

Java API インターフェース[Sun 2009b]によると、java.util.concurrent.ExecutorService.shutdownNow()メソッドは、

> *実行中のアクティブなタスクすべての停止を試み、待機中のタスクの処理を停止し、実行を待機していたタスクのリストを返す。*
> *実行中のアクティブなタスク処理を停止するために最善の努力をすること以上の保証はない。たとえば、一般的に Thread.interrupt() によって取消しが実装されるが、この割込みに応答できないタスクは終了しないかもしれない。*

同様に Future.cancel()メソッドを使用してスレッドプール内の個々のタスクを取り消そうとする場合も、タスクは割込みに対応している必要がある。

5.3.1. 違反コード (スレッドプールの終了)

以下の違反コードでは、PoolService クラス内で宣言されたスレッドプールに、SocketReader クラスをタスクとして依頼している。

```
public final class SocketReader implements Runnable { //  スレッドセーフなクラス
  private final Socket socket;
  private final BufferedReader in;
  private final Object lock = new Object();

  public SocketReader(String host, int port) throws IOException {
    this.socket = new Socket(host, port);
    this.in = new BufferedReader(new InputStreamReader(this.socket.getInputStream()));
  }
  //一度に一つのスレッドだけがソケットを使用できる
  @Override public void run() {
    try {
      synchronized (lock) {
        readData();
      }
    } catch (IOException ie) {
      // ハンドラへの転送
    }
  }

  public void readData() throws IOException {
    String string;
    try {
      while ((string = in.readLine()) != null) {
        // ストリームが null になるまでブロックを行う
      }
    } finally {
      shutdown();
    }
  }
  public void shutdown() throws IOException {
    socket.close();
  }
}

public final class PoolService {
  private final ExecutorService pool;

  public PoolService(int poolSize) {
    pool = Executors.newFixedThreadPool(poolSize);
  }
  public void doSomething() throws InterruptedException, IOException {
    pool.submit(new SocketReader("somehost", 8080));
    // ...
    List<Runnable> awaitingTasks = pool.shutdownNow();
  }

  public static void main(String[] args) throws InterruptedException, IOException {
    PoolService service = new PoolService(5);
    service.doSomething();
  }
}
```

この例のタスクは Thread.interrupt()メソッドを用いた割込みに対応していないので、shutdownNow()メソッドがスレッドプールを終了するという保証はない。shutdownNow()メソッドを使用しても、実行中のタスクがすべて終了するまで待機状態となり、問題は解決しない。

同様にスレッドを終了するタイミングを判定するために Thread.interrupted()メソッド以外の仕組みを使用するタスクは、shutdown()あるいは shutdownNow()メソッドに応答しないだろう。たとえば、終了することが安全かどうか判定するために揮発性フラグをチェックするタスクは、これらのメソッドに応答しないだろう。フラグを使ったスレッド終了に関する詳細は、ガイドライン「THI05-J. スレッドの終了に Thread.stop()メソッドを使用しない」で説明している。

5.3.2. 適合コード (割込み可能なタスクの依頼)

以下の適合コードでは、割込みに対応した SocketReader クラスを定義している。SocketReader クラスはインスタンス化され、スレッドプールに依頼されている。

```
public final class SocketReader implements Runnable {
  private final SocketChannel sc;
  private final Object lock = new Object();
  public SocketReader(String host, int port) throws IOException {
    sc = SocketChannel.open(new InetSocketAddress(host, port));
  }
  @Override public void run() {
    ByteBuffer buf = ByteBuffer.allocate(1024);
    try {
      synchronized (lock) {
        while (!Thread.interrupted()) {
          sc.read(buf);
          // ...
        }
      }
    } catch (IOException ie) {
      // ハンドラへの転送
    }
  }
}

public final class PoolService {
  // ...
}
```

5.3.3. 例外

TPS02-EX1: ブロックせずに短時間で完了するタスクは、このガイドラインにしたがう必要はない。

5.3.4. リスク評価

割込み可能でないタスクを依頼すると、スレッドプールの終了が不可能となり、サービス停止状態を引き起こすかもしれない。

ガイドライン	深刻度	可能性	修正コスト	優先度	レベル
TPS02-J	低	中	中	P4	L3

5.3.5. 参考文献

[Goetz 2006]	Chapter 7, “Cancellation and shutdown”
[Sun 2009b]	Interface ExecutorService

5.4. TPS03-J. スレッドプールで実行されるタスクの異常終了を通知する

長期間にわたって実行されるタスクは、異常終了時にアプリケーションへ通知を行う仕組みを提供するべきである。そのような仕組みがなくても、プール中のスレッドは再利用されるのでリソースのリークは発生しないが、障害診断が非常に困難になる。

アプリケーションレベルで例外を扱う最良の方法は、例外ハンドラの使用である。診断動作、クリーンアップ処理、JVM の終了、あるいは障害情報の記録などを例外ハンドラで行うことができる。

5.4.1. 違反コード (タスクの異常終了)

以下の違反コードは、スレッドプールをカプセル化している PoolService クラスおよび Runnable を実装する Task クラスで構成されている。Task.run()メソッドは、NullPointerException のような実行時例外をスローする可能性がある。

```
final class PoolService {
  private final ExecutorService pool = Executors.newFixedThreadPool(10);
  public void doSomething() {
    pool.execute(new Task());
  }
}

final class Task implements Runnable {
  @Override public void run() {
    // ...
    throw new NullPointerException();
    // ...
  }
}
```

実行時例外によってタスクが予期せずに終了する場合、アプリケーションには何の通知も行われない。また、復旧の仕組みもないため、Task が NullPointerException をスローしても、無視されることになる。

5.4.2. 適合コード (ThreadPoolExecutor フック)

java.util.concurrent.ThreadPoolExecutor クラスの afterExecute()フックメソッドをオーバーライドすることによりタスク別の復旧あるいはクリーンアップ動作を実行することができる。このフックは、タスクがその run()メソッドの全ステートメントの実行により成功裡に終了するか、例外のため停止する場合に呼び出される。(JVM の実装によっては、java.lang.Error が捕捉されない場合がある。詳細は、Bug ID 6450211 を参照 [Sun 2008b]。)

このアプローチの実装には ThreadPoolExecutor の afterExecute()フックメソッドを以下のようにオーバーライドして使用する。

```
final class PoolService {
  // 各引数値は、サンプルコードを見易くするために、ハードコーディングしている
  ExecutorService pool = new CustomThreadPoolExecutor(10, 10, 10, TimeUnit.SECONDS,
    new ArrayBlockingQueue<Runnable>(10));
  // ...
}

class CustomThreadPoolExecutor extends ThreadPoolExecutor {
  // ... コンストラクタ ...

  @Override
  public void afterExecute(Runnable r, Throwable t) {
    super.afterExecute(r, t);
    if (t != null) {
      // 例外が発生したので、ハンドラへ転送
    }
    // ... タスクに特化したクリーンアップ動作を実行
  }

  @Override
  public void terminated() {
    super.terminated();
    // ... クリーンアップの最終動作を実行
  }
}
```

terminated()フックメソッドは、すべてのタスクの実行が終了し、Executor が正常終了した後に呼ばれることになる。このフックをオーバーライドして、（ちょうど finally ブロックで行うように）スレッドプールが獲得したリソースを解放することができる。

5.4.3. 適合コード (未捕捉例外ハンドラ)

以下の適合コードでは、ファクトリクラスがスレッドプールに代わって、未捕捉例外ハンドラをセットしている。ThreadFactory クラスのインスタンスが、スレッドプールの生成時に、引数として渡されている。ファクトリは、新しいスレッドを作成し未捕捉例外ハンドラをセットする。Task クラスは、前述の違反コードからの変更はない。

```
final class PoolService {
  private static final ThreadFactory factory = new
  ExceptionThreadFactory(new MyExceptionHandler());
  private static final ExecutorService pool =
    Executors.newFixedThreadPool(10, factory);

  public void doSomething() {
    pool.execute(new Task()); // Task は Runnable を実装している
  }
  public static class ExceptionThreadFactory implements ThreadFactory {
    private static final ThreadFactory defaultFactory =
      Executors.defaultThreadFactory();
    private final Thread.UncaughtExceptionHandler handler;

    public ExceptionThreadFactory(Thread.UncaughtExceptionHandler handler) {
      this.handler = handler;
    }

    @Override public Thread newThread(Runnable run) {
      Thread thread = defaultFactory.newThread(run);
      thread.setUncaughtExceptionHandler(handler);
      return thread;
    }
  }
  public static class MyExceptionHandler extends ExceptionReporter
      implements Thread.UncaughtExceptionHandler {
    // ...

    @Override public void uncaughtException(Thread thread, Throwable t) {
      // 復旧あるいはログ用のコードを記述
    }
  }
}
```

スレッドプールへのタスクの依頼に execute()メソッドではなく ExecutorService.submit()メソッドを使用することで Future オブジェクトを得ることができる。ただし、ExecutorService.submit()メソッドを呼び出す場合、未捕捉例外ハンドラが呼ばれないことに注意。これは、スローされた例外が、返り値のステータスの一部として ExecutionException クラスにラップされており、Future.get()メソッドにより再度スローできるからである [Goetz 2006]。

5.4.4. 適合コード (Future<V>クラスと submit()メソッド)

以下の適合コードでは、Future オブジェクトを獲得するために、ExecutorService.submit()メソッドを呼び出してタスクを依頼している。Future オブジェクトを使用して、タスクに例外を再送させ、その例外を取り扱うことができるようにしている。

```
final class PoolService {
  private final ExecutorService pool = Executors.newFixedThreadPool(10);

  public void doSomething() {
    Future<?> future = pool.submit(new Task());

    // ...

    try {
      future.get();
    } catch (InterruptedException e) {
      Thread.currentThread().interrupt(); // 割込みステータスのリセット
    } catch (ExecutionException e) {
      Throwable exception = e.getCause();
      // 例外の原因を報告するための処理を記述
    }
  }
}
```

また、Future クラスの取得時に発生しうるあらゆる例外を doSomething()メソッド内で必要に応じて処理できる。

5.4.5. 例外

TPS03-EX1: Runnable あるいは Callable を実装するすべてのタスクのコードにおいて、例外条件が生じる可能性がないことが確認されている場合、このガイドラインに適合していなくてもよい。ただし、復旧のための初期化処理あるいは例外条件記録のために、タスクに固有か、あるいは、グローバルな例外ハンドラを導入することは、一般的に良い習慣である。

5.4.6. リスク評価

スレッドプール中のタスクが、例外条件の結果、異常終了したことを報告する仕組みを提供しないと、問題の原因追求がより困難になりうる。

ガイドライン	深刻度	可能性	修正コスト	優先度	レベル
TPS03-J	低	中	中	P4	L3

5.4.7. 参考文献

[Goetz 2006]	Chapter 7.3, “ Handling abnormal thread termination”
[Sun 2009b]	Interfaces ExecutorService, ThreadFactory and class Thread

5.5. TPS04-J. スレッドプール使用時にThreadLocal変数が再初期化済みであることを確実にする

java.lang.ThreadLocal<T>クラスは、スレッドローカル変数を提供する。Java API によると[Sun 2009b]

> *これらの変数は、アクセッサ get と set を持ち、これらのメソッドを使用するスレッドそれぞれに固有の初期化された変数のコピーを維持する点で、通常の変数と異なる。ThreadLocal インスタンスは、private static フィールドとして状態をスレッドに関連付ける目的で利用される (ユーザーID、トランザクション ID など)。*

スレッドプール内で複数スレッドから呼び出されるオブジェクトのクラスで ThreadLocal オブジェクトを使用する場合は注意する必要がある。スレッドプールそのものは、スレッド生成のオーバーヘッドが大きすぎる場合やスレッドの無制限な生成によるシステムの信頼性低下対策として有効である。また、プールに登録されるすべてのスレッドは、読み込むオブジェクトがデフォルトの初期状態であることを期待する。しかし、スレッドにより ThreadLocal オブジェクトが既に更新されている場合、そのスレッドが再利用される時は、最後に更新された ThreadLocal オブジェクトの状態を読み取ることになる[Arnold 2006]。

5.5.1. 違反コード

以下の違反コードは、曜日(Day)の列挙(enumeration)および二つのクラス(Diary と DiaryPool)により構成されている。Diary クラスでは、各スレッドの実行時の日付など、そのスレッド毎に情報を格納するために ThreadLocal 変数を使用している。実行時の日付の初期値は、月曜日である(この内容は、後で setDay()メソッドを呼び出すことにより変更可能である)。このクラスは、スレッド固有のタスクを実行する threadSpecificTask()メソッドも含んでいる。

DiaryPool クラスは、それぞれ一つのスレッドを開始する doSomething1()および doSomething2()メソッドにより構成されている。doSomething1()メソッドは、日付の初期値(デフォルト値)を金曜日に変更し、threadSpecificTask()メソッドを呼び出している。一方 doSomething2()メソッドは、日付の初期値(月曜日)に依存したまま、threadSpecificTask()メソッドを呼び出している。

main()メソッドでは、doSomething1()メソッドを使用して一つのスレッドを生成し、doSomething2()メソッドの呼出しで更に二つのスレッドを生成している。

```
public enum Day {
  MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY;
}

public final class Diary {
  private static final ThreadLocal<Day> days =
    new ThreadLocal<Day>() {
    // 日付を月曜日に初期化する
    protected Day initialValue() {
      return Day.MONDAY;
    }
  };

  private static Day currentDay() {
    return days.get();
  }

  public static void setDay(Day newDay) {
    days.set(newDay);
  }
  // スレッドに特化したタスクを実行する
  public void threadSpecificTask() {
    // タスクの処理を行う ...
  }
}

public final class DiaryPool {
  final int NoOfThreads = 2; // プール内の最大スレッド数を定義
  final Executor exec;
  final Diary diary;

  DiaryPool() {
    exec = (Executor) Executors.newFixedThreadPool(NoOfThreads);
    diary = new Diary();
  }

  public void doSomething1() {
    exec.execute(new Runnable() {
      @Override public void run() {
        Diary.setDay(Day.FRIDAY);
        diary.threadSpecificTask();
      }
    });
  }

  public void doSomething2() {
    exec.execute(new Runnable() {
      @Override public void run() {
        diary.threadSpecificTask();
      }
    });
  }

  public static void main(String[] args) {
    DiaryPool dp = new DiaryPool();
```

```
    dp.doSomething1(); // スレッド 1 は、実行時の曜日として金曜日を指定
    dp.doSomething2(); // スレッド 2 は、実行時の曜日として月曜日を指定
    dp.doSomething2(); // スレッド 3 は、実行時の曜日として月曜日を指定
  }
}
```

DiaryPool クラスは、固定数のスレッドを再利用するスレッドプールを作成する。NoOfThreads が示すスレッド数が、同時にアクティブとなるスレッドの最大数である。スレッドがすべてアクティブな時に追加のタスクが依頼された場合、スレッドが利用可能になるまで、それらのタスクはキュー内で待機状態となる。スレッドが再利用される場合、スレッドローカルな状態は保持されたままである。

以下の表では、起こりうるタスクの順序を示している。

時間	タスク	プールスレッド	タスクを依頼したメソッド	曜日
1	t_1	1	doSomething1()	金曜日
2	t_2	2	doSomething2()	月曜日
3	t_3	1	doSomething2()	金曜日

上記の実行順序では、doSomething2()メソッドを使用して開始した二つのタスク(t_2およびt_3)は、月曜日を実行時の曜日として認識すると思うかもしれない。しかし、プールされたスレッド 1 が再利用されるので、t_3は、金曜日を実行時の曜日として認識することになる。

5.5.2. 違反コード (スレッドプールサイズの増加)

以下の違反コードでは、問題を緩和しようとして、スレッドプールのサイズを 2 から 3 に変更している。

```
public final class DiaryPool {
  final int NoOfThreads = 3;
  // ...
}
```

スレッドプールのサイズを増加させることにより前述の問題は解決するが、より多くのタスクがプールに依頼される場合には、このスレッドプールサイズの変更では不十分となるので、スケーラビリティに優れた解決方法とはいえない。

5.5.3. 適合コード (try-finally ブロック)

以下の適合コードでは、Diary クラスに removeDay()メソッドを追加して、DiaryPool クラスの doSomething1()メソッドの try-finally ブロックから removeDay()を呼び出している。finally ブロックは、スレッドローカルな days オブジェクトから、実行中のスレッドの値を取り除いてオブジェクトを初期化する。

```
public final class Diary {
  // ...
  public static void removeDay() {
    days.remove();
  }
}

public final class DiaryPool {
  // ...

  public void doSomething1() {
    exec.execute(new Runnable() {
      @Override public void run() {
        try {
          Diary.setDay(Day.FRIDAY);
          diary.threadSpecificTask();
        } finally {
          Diary.removeDay(); // Diary.setDay(Day.MONDAY)メソッドも使用可能である
        }
      }
    });
  }
  // ...
}
```

スレッドローカル変数が再び同一スレッドにより読み取られる時、先にスレッドが明示的に変数の値をセットしていなければ、initialValue()メソッドにより再度初期化される。 [Sun 2009b]。上記の解決方法では、スレッドローカル変数の操作に関する責任がクライアント側(DiaryPool)に移されている。この手法は Diary クラスの修正が不可能な場合に良い解決策である。

5.5.4. 適合コード (beforeExecute()メソッド)

以下の適合コードでは、ThreadPoolExecutor クラスを継承した CustomThreadPoolExecutor クラスを使用し、beforeExecute()メソッドをオーバーライドしている。この beforeExecute()メソッドは、Runnable タスクが指定したスレッド内で実行される前に呼び出される。beforeExecute()メソッドにより、タスク r をスレッド t で実行する前に、スレッドローカル変数が再初期化される。

```
class CustomThreadPoolExecutor extends ThreadPoolExecutor {
  public CustomThreadPoolExecutor(int corePoolSize, int maximumPoolSize,
    long keepAliveTime, TimeUnit unit, BlockingQueue<Runnable> workQueue) {
    super(corePoolSize, maximumPoolSize, keepAliveTime, unit, workQueue);
  }
  @Override
  public void beforeExecute(Thread t, Runnable r) {
    if (t == null || r == null) {
      throw new NullPointerException();
    }
    Diary.setDay(Day.MONDAY);
    super.beforeExecute(t, r);
  }
}

public final class DiaryPool {
  // ...
  DiaryPool() {
    exec = new CustomThreadPoolExecutor(NoOfThreads, NoOfThreads,
      10, TimeUnit.SECONDS, new ArrayBlockingQueue<Runnable>(10));
    diary = new Diary();
  }
  // ...
}
```

5.5.5. 例外

TPS04-EX1: 初期化処理後に状態が変更されない ThreadLocal オブジェクトを再初期化する必要はない。たとえば、ThreadLocal 変数にデータベース接続オブジェクトを設定した場合、初期設定以降その値を変更する必要はないかもしれない。

5.5.6. リスク評価

スレッドプール内の異なるスレッドから、ThreadLocal オブジェクトの再初期化を行わずに実行されたオブジェクトは、再利用時に予期せぬ状態にあるかもしれない。

ガイドライン	深刻度	可能性	修正コスト	優先度	レベル
TPS04-J	中	中	高	P4	L3

5.5.7. 参考文献

[Arnold 2006]	Section 14.13, “ThreadLocal Variables”
[Sun 2009b]	class java.lang.ThreadLocal<T>

6. スレッドの安全性に関する雑則(TSM)ガイドライン

6.1. TSM00-J. スレッドセーフなメソッドを、スレッドセーフでないメソッドでオーバーライドしない

スレッドセーフなメソッドをスレッドセーフでないメソッドでオーバーライドしたサブクラスのインスタンスに対してクライアントが不注意な操作を行った場合、同期処理が正しく行われない可能性がある。安全に並行処理を行えないサブクラスのメソッドで、synchronized メソッドがオーバーライドされると、並行処理の安全性が損なわれてしまうことになる。

スレッドセーフでないメソッドでスレッドセーフなメソッドをオーバーライドすること自体はエラーではない。しかし、分析が困難なエラーを引き起こす可能性が高いので、このガイドラインでは、スレッドセーフでないメソッドによるスレッドセーフなメソッドのオーバーライドを禁止している。

継承されることを前提に設計されたクラスのロック方式は、文書化されるべきである。サブクラスはこの情報を基に適切なロック方式を実装することができる (ガイドライン「LCK00-J. 信頼できないコードから使用されるクラスの同期化には private final ロックオブジェクトを使用する」を参照)。

6.1.1. 違反コード (synchronized メソッド)

以下の違反コードでは、Derived サブクラスの同期化されていないメソッドで Base クラスの同期化された doSomething()メソッドをオーバーライドしている。

```
class Base {
  public synchronized void doSomething() {
    // ...
  }
}

class Derived extends Base {
  @Override public void doSomething() {
    // ...
  }
}
```

Base クラスの doSomething()メソッドは、複数のスレッドにより安全に実行することが可能であるが、Derived サブクラスのインスタンスの doSomething()メソッドの実行は安全ではない。

Base クラスのインスタンスを利用可能なスレッドが、そのサブクラスである Derived のインスタンスも利用できる場合、このエラーは分析が困難になりうる。クライアント側でサブクラスのインスタンスのメソッドをスレッドセーフではないと知らずに使用してしまうかもしれない。

6.1.2. 適合コード (synchronized メソッド)

以下の適合コードでは、サブクラスの doSomething()メソッドを同期化している。

```
class Base {
  public synchronized void doSomething() {
    // ...
  }
}

class Derived extends Base {
  @Override public synchronized void doSomething() {
    // ...
  }
}
```

上記の適合コードは、クラスへのアクセス権が package-private であるため、ガイドライン「LCK00-J. 信頼できないコードから使用されるクラスの同期化には private final ロックオブジェクトを使用する」には違反しない。信用できないコードが該当するパッケージにアクセスできないのであれば、この対策は許容できる。

6.1.3. 適合コード (private final ロックオブジェクト)

以下の適合コードでは、Base クラスの同期化された doSomething()メソッドを private final 宣言されたロックオブジェクトで同期化するメソッドでオーバーライドして、Derived クラスがスレッドセーフであることを確保している。

```
class Base {

  public synchronized void doSomething() {
    // ...
  }
}

class Derived extends Base {
  private final Object lock = new Object();

  @Override public void doSomething() {
    synchronized (lock) {
      // ...
    }
  }
}
```

Derived クラスにおいて一貫したロックポリシーが保持されているので、上記のコードはこのガイドラインに適合している。

6.1.4. 違反コード (private ロック)

以下の違反コードでは、ガイドライン「LCK00-J. 信頼できないコードから使用されるクラスの同期化には private final ロックオブジェクトを使用する」にしたがって、Base クラスにおいて private final 宣言されたロックを使用する doSomething()メソッドを定義している。

```
class Base {
  private final Object lock = new Object();

  public void doSomething() {
    synchronized (lock) {
      // ...
    }
  }
}

class Derived extends Base {
  @Override public void doSomething() {
    try {
      super.doSomething();
    } finally {
      logger.log(Level.FINE, "Did something");
    }
  }
}
```

上記のコードでは、複数のスレッドが同時に呼び出しを行った場合、各タスクの実行順序とは異なる順序でログのエントリを出力してしまう可能性がある。この例ではログが順序通りに出力されることが求められるため、Derived クラスの doSomething()メソッドはスレッドセーフではなく複数のスレッドにより安全に使用することはできない。

6.1.5. 適合コード (private ロック)

以下の適合コードでは、private final 宣言されたロックオブジェクトを使用することにより、サブクラスの doSomething()メソッドを同期化している。

```
class Base {
  private final Object lock = new Object();

  public void doSomething() {
    synchronized (lock) {
      // ...
    }
  }
}

class Derived extends Base {
  private final Object lock = new Object();

  @Override public void doSomething() {
    synchronized (lock) {
      try {
        super.doSomething();
      } finally {
        logger.log(Level.FINE, "Did something");
      }
    }
  }
}
```

Base と Derived の各オブジェクトが、双方のクラスからアクセスできない個別のロックを保持していることに注意。Derived クラスのスレッド安全性を Base クラスから独立して保証している。

6.1.6. リスク評価

スレッドセーフでないメソッドでスレッドセーフなメソッドをオーバーライドすると、予期せぬ振舞いにつながるかもしれない。

ガイドライン	深刻度	可能性	修正コスト	優先度	レベル
TSM00-J	低	中	中	P4	L3

6.1.7. 参考文献

[Sun 2009b]	
[Sun 2008b]	Sun bug database, Bug ID 4294756

6.2. TSM01-J. オブジェクトの構築時に this 参照を逸出させない

*Java 言語仕様*の 15.8.3 節「this」によれば [Gosling 2005]

> *this キーワードが一次式として用いられた場合、インスタンスメソッドの呼出し(§15.12)が行われたオブジェクト、または構築中のオブジェクトへの参照値を表す。this の型は、this キーワードが記述されているクラス C である。実行時の場合、実際のオブジェクトが参照するクラスは、クラス C やクラス C のサブクラスである可能性がある。*

this 参照の処理スコープを超えて利用される状態を逸出(escape)と呼ぶ。よくある this 参照の逸出例としては、以下のようなものが含まれる。

- オブジェクトの構築に伴いクラスのコンストラクタから呼び出されるメソッドで、かつ、private ではなく、オーバーライド可能なメソッドから this を返す。(詳細は、ガイドライン「MET05-J. Ensure that constructors do not call overridable methods [10]」を参照)
- 可変クラスの private ではないメソッドから this を返すメソッドを使用して、呼出し元がオブジェクトの状態を間接的に操作する。これは、通常メソッドチェーンの実装で生じる。詳細は、ガイドライン「VNA04-J. メソッドチェーン呼出しのアトミック性を確保する」を参照。
- オブジェクトの構築に伴いクラスのコンストラクタから呼び出されるよそ者メソッド(alien method)に、引数として this を渡す。
- 内部クラスの使用。内部クラスは static 宣言されない限り、その外部クラスのインスタンスへの参照を保持することになる。
- オブジェクトが構築されるコンストラクタで public static 宣言された変数に this 参照を代入することにより公開する。
- オブジェクトの構築が中断され、オーバーライドした final 宣言されていないクラスのファイナライザから初期化が不完全なインスタンスの this 参照を得る。同じことが、コンストラクタが例外をスローする場合にも起こりうる。ファイナライザへのオーバーライドを誤用するのは、信頼できないコードだけとは限らない。信頼できるコードでも、不注意にファイナライザを追加して this を逸出させてしまうことがありうる。
- 内部オブジェクトの状態を、よそ者メソッドへ渡す。結果的に、よそ者メソッドが内部メンバであるオブジェクトの参照を得ることができる。

10 このガイドラインは、https://www.securecoding.cert.org/confluence/display/java/に記述されている。

このガイドラインでは、オブジェクトの構築時に this 参照が逸出することを許してしまうことでもたらされる潜在的な影響(競合状態や不適切な初期化を含む)について記述している。たとえば、オブジェクトの構築時に this 参照が逸出しない場合に限り、final 宣言されたフィールドは、どのスレッドからも完全に初期化された状態で読み取られる。ガイドライン「TSM03-J. 初期化が不完全なオブジェクトを公開しない」では安全な公開のための様々な仕組みにより提供される保証について解説しているが、これらは本ガイドラインを遵守することが前提となっている。概して言えば、this 参照が実行時のコンテキストの範囲を越えて逸出してしまうような場合を見つけ出すことが重要である。とりわけ、public 宣言された変数およびメソッドは、慎重に精査されるべきである。

6.2.1. 違反コード (初期化前の this 参照の公開)

以下の違反コードでは、this 参照を、public static volatile 宣言されたフィールドに格納することにより、初期化が完了する前に公開している。

```
final class Publisher {
  public static volatile Publisher published;
  int num;

  Publisher(int number) {
    published = this;
    // 初期化処理
    this.num = number;
    // ...
  }
}
```

他のスレッドは、初期化が不完全な Publisher インスタンスを獲得しうる。また、オブジェクトの初期化処理が、コンストラクタ内で行われるセキュリティーチェック処理に依存する場合、信頼できない呼出し元により初期化が不完全なインスタンスを獲得されて、セキュリティチェックが回避されるかもしれない。

6.2.2. 違反コード (volatile 宣言していない public static フィールド)

以下の違反コードでは、コンストラクタの最後のステートメントで this 参照を公開しているが、published フィールドを volatile 宣言しておらず、また、パブリックアクセスが可能であり脆弱である。

```
final class Publisher {
  public static Publisher published;
  int num;

  Publisher(int number) {
    // 初期化処理
    this.num = number;
    // ...
    published = this;
  }
}
```

published フィールドは volatile 宣言も final 宣言もされていないので、コンストラクタ内のステートメントはコンパイラにより順序替えされて初期化ステートメントの実行前に this 参照が公開されてしまう可能性がある。

6.2.3. 適合コード (volatile フィールドと初期化後の公開)

以下の適合コードでは、published フィールドを volatile 宣言し、アクセスを package-private に限定しているので、外部パッケージスコープからの呼出し元は this 参照を獲得することができない。

```
final class Publisher {
  static volatile Publisher published;
  int num;

  Publisher(int number) {
    // 初期化処理
    this.num = number;
    // ...
    published = this;
  }
}
```

上記コードでは、コンストラクタは初期化の完了後に this 参照を公開している。しかし、Publisher クラスをインスタンス化する呼出し元は、初期化される前の num フィールドのデフォルト値を読み取らないように注意しなければならない（ガイドライン「TSM03-J. 初期化が不完全なオブジェクトを公開しない」への違反である)。呼出し元は、Publisher への参照を保持するフィールドを、volatile 宣言するなどの対応を検討すべきである。

published フィールドが volatile 宣言されていない場合、初期化処理ステートメントの実行順序は変更されうる。なお、Java コンパイラは、フィールドを volatile と final の修飾子を両方同時に宣言することは許していない。

Publisher クラスも final 宣言する必要がある。さもないと、サブクラスが、自身の初期化が完了する前に Publisher クラスのコンストラクタを呼び出し、this 参照を公開することもありうる。

6.2.4. 適合コード (public static ファクトリメソッド)

以下の適合コードでは、内部メンバーフィールドを削除し、Publisher インスタンスを生成して返す newInstance()ファクトリメソッドを提供している。

```
final class Publisher {
  final int num;
  private Publisher(int number) {
    // 初期化処理
    this.num = number;
  }
  public static Publisher newInstance(int number) {
    Publisher published = new Publisher(number);
    return published;
  }
}
```

上記のアプローチにより、構築途中の不完全な Publisher インスタンスがスレッドから読み取られない。num フィールドを final 宣言し、クラスを不変にすることで、呼出し元が不完全な初期化状態のオブジェクトを獲得する可能性を排除している。

6.2.5. 違反コード (コンストラクタでの this 参照公開)

ここで紹介する違反コードでは、以下に示す ExceptionReporter インターフェースを定義している。

```
public interface ExceptionReporter {
  public void setExceptionReporter(ExceptionReporter er);
  public void report(Throwable exception);
}
```

このインターフェースを以下の DefaultExceptionReporter クラスで実装する。DefaultExceptionReporter クラスは、機密情報をすべて除去してから、例外を報告する。

DefaultExceptionReporter クラスのコンストラクタは、オブジェクトの構築が完了する前に this 参照を公開してしまう。この問題は、コンストラクタの最終ステートメント

(er.setExceptionReporter(this))において、例外レポーターをセットしている箇所が該当するが、コンストラクタの最終ステートメントなので見落としやすい。

```
// DefaultExceptionReporter クラス
public class DefaultExceptionReporter implements ExceptionReporter {
   public DefaultExceptionReporter(ExceptionReporter er) {
      // 初期化処理の実行
      // 誤って「this」参照を公開している
      er.setExceptionReporter(this);
   }
   // setExceptionReporter()メソッドと report()メソッドの実装
}
```

以下に示す MyExceptionReporter クラスは、DefaultExceptionReporter クラスを継承し、例外を報告する前に重要なメッセージを記録するログ出力の仕組みを追加している。

```
// MyExceptionReporter クラスは、DefaultExceptionReporter を継承している
public class MyExceptionReporter extends DefaultExceptionReporter {
   private final Logger logger;
   public MyExceptionReporter(ExceptionReporter er) {
      super(er); // 上位クラスのコンストラクタを呼んでいる
      logger = Logger.getLogger("com.organization.Log"); // デフォルトの logger を獲得している
   }
   public void report(Throwable t) {
      logger.log(Level.FINEST,"Loggable exception occurred", t);
   }
}
```

このクラスのコンストラクタは DefaultExceptionReporter スーパークラスのコンストラクタを呼び出すが、サブクラスである MyExceptionReporter の初期化が完了する前にそのインスタンスが例外レポーターとして公開されてしまう。また、サブクラスの初期化処理は、デフォルトロガーのインスタンスを取得していることに注意。

例外レポーターが公開された時点から例外の受信と処理が始まるため、MyExceptionReporter サブクラスにおける Logger.getLogger()の呼出しの前に例外が発生した場合、その内容は記録されない。代わりに NullPointerException が生成され、その例外自体はログに出力されることなく処理される。

この問題は、例外の発生と MyExceptionReporter の初期化の間の競合状態に起因している。例外の発生が早過ぎると、初期化が完了する前の MyExceptionReporter オブジェクトを参照することになる。logger は final 宣言され、初期化されていない値を含むことは想定されていないので、この振舞いは非常に分かりにくい。

同様の問題がイベントリスナーの公開が早過ぎる場合にも発生しうる。この場合、サブクラスが初期化を完了する前に、イベント通知の受信を開始してしまう。

6.2.6. 適合コード

以下の適合コードでは、DefaultExceptionReporter クラスのコンストラクタで this 参照を公開する代わりに、DefaultExceptionReporter クラスに publishExceptionReporter()メソッドを追加して、例外レポーターをセットしている。サブクラスの初期化が完了後、このメソッドをサブクラスのインスタンスで呼び出すことができる。

```
public class DefaultExceptionReporter implements ExceptionReporter {
  public DefaultExceptionReporter(ExceptionReporter er) {
    // ...
  }
  // サブクラスの初期化処理の完了後に呼び出す必要がある
  public void publishExceptionReporter() {
    setExceptionReporter(this); // this 参照を例外レポーターに登録する
  }
  // setExceptionReporter()メソッドと report()メソッドの実装
}
```

MyExceptionReporter サブクラスは publishExceptionReporter()メソッドを継承しているので、MyExceptionReporter をインスタンス化した呼出し元は、初期化の完了後、そのインスタンスを使用して、例外レポーターをセットすることができる。

```
// MyExceptionReporter クラスは、 DefaultExceptionReporter クラスを継承している
public class MyExceptionReporter extends DefaultExceptionReporter {
  private final Logger logger;
  public MyExceptionReporter(ExceptionReporter er) {
    super(er); //上位クラスのコンストラクタを呼んでいる
    logger = Logger.getLogger("com.organization.Log");
  }
  // publishExceptionReporter()メソッドと, setExceptionReporter()メソッドおよび
  // report()メソッドの実装が継承されている
}
```

このアプローチにより、コンストラクタがサブクラスのインスタンスを初期化して、ログ出力が可能になるまでは、例外レポーターがセットされることはない。

6.2.7. 違反コード (内部クラス)

内部クラスは、外部クラスオブジェクトの this 参照のコピーを保持するが、this 参照がスコープ外に逸出する原因となりうる[Goetz 2002]。以下の違反コードでは、DefaultExceptionReporter クラスの異なる実装例を示しており、コンストラクタに含まれた内部クラスを使用して、filter()メソッドを呼び出す例外レポーターを公開している。

```
public class DefaultExceptionReporter implements ExceptionReporter {
  public DefaultExceptionReporter(ExceptionReporter er) {
    er.setExceptionReporter(new DefaultExceptionReporter(er) {
      public void report(Throwable t) {
        filter(t);
      }
    });
  }
  // setExceptionReporter()メソッドと report()メソッドのデフォルトの実装
}
```

外部クラスの this 参照は、内部クラスにより公開されるので、他のスレッドから読み取ることができる。また、クラスがサブクラス化される場合、前述の「6.2.5. 違反コード（コンストラクタでの this 参照の公開）」で説明されている問題も発生する。

6.2.8. 適合コード

public static 宣言されたファクトリメソッドと private 宣言されたコンストラクタを用い、filter()メソッドを呼び出す例外レポーターをコンストラクタ内部から安全に公開することができる [Goetz 2006]。

```
public class DefaultExceptionReporter implements ExceptionReporter {
  private final DefaultExceptionReporter defaultER;
  private DefaultExceptionReporter(ExceptionReporter excr) {
    defaultER = new DefaultExceptionReporter(excr) {
      public void report(Throwable t) {
        filter(t);
      }
    };
  }
  public static DefaultExceptionReporter newInstance(ExceptionReporter excr) {
    DefaultExceptionReporter der = new DefaultExceptionReporter(excr);
    excr.setExceptionReporter(der.defaultER);
    return der;
  }
  // setExceptionReporter()メソッドと report()メソッドのデフォルトの実装
}
```

コンストラクタを private 宣言し、信用できないコードがクラスのインスタンスを作成できないようにすることで、this 参照の逸出を防止している。また、新たなインスタンスの作成に public static 宣言されたファクトリメソッドを使用して、初期化が不完全なオブジェクトの公開を防ぐとともに(ガイドライン「TSM03-J. 初期化が不完全なオブジェクトを公開しない」を参照)、信頼できないコードにより内部オブジェクトの状態が操作されないようにしている。

6.2.9. 違反コード (スレッド)

以下の違反コードでは、コンストラクタ内部からスレッドを開始している。

```
final class ThreadStarter implements Runnable {
  public ThreadStarter() {
    Thread thread = new Thread(this);
    thread.start();
  }
  @Override public void run() {
    // ...
  }
}
```

この例では、新たなスレッドから、実行中のオブジェクトの this 参照にアクセス可能である [Goetz 2002、Goetz 2006]。Thread()コンストラクタは、ThreadStarter クラスから見てよそ者メソッドである。

6.2.10. 適合コード (スレッド)

以下の適合コードでは、コンストラクタではなく、他のメソッド内でスレッドを作成し、開始している。

```
final class ThreadStarter implements Runnable {
  public ThreadStarter() {
    // ...
  }

  public void startThread() {
    Thread thread = new Thread(this);
    thread.start();
  }

  @Override public void run() {
    // ...
  }
}
```

6.2.11. 例外

TSM01-EX1: オブジェクトの構築が完了するまでスレッドを開始しないのであれば、コンストラクタ内でスレッドを作成しても安全である。スレッドの start()メソッドの呼出しは、スレッド内で定義されている如何なる動作よりも事前発生する[Gosling 2005]。

以下のコード例において、this を参照するスレッドがコンストラクタで作成されているが、その start()メソッドが startThread()メソッドから呼ばれるまで、スレッドは開始しない [Goetz 2002、Goetz 2006]。

```
final class ThreadStarter implements Runnable {
  Thread thread;

  public ThreadStarter() {
    thread = new Thread(this);
  }

  public void startThread() {
    thread.start();
  }

  @Override public void run() {
    // ...
  }
}
```

TSM01-EX2: ガイドライン「TSM02-J. クラスの初期化中にバックグラウンドスレッドを使用しない」で説明している ObjectPreserver パターン[Grand 2002]も、このガイドラインの安全な例外である。

6.2.12. リスク評価

this 参照の逸出を許してしまうと、不適切な初期化およびランタイム例外という結果につながるかもしれない。

ガイドライン	深刻度	可能性	修正コスト	優先度	レベル
TSM01-J	中	中	高	P4	L3

6.2.13. 参考文献

[Goetz 2　002]	
[Goetz 2006]	Section 3.2, “Publication and Escape”
[Gosling 2005]	Keyword “this”
[Grand 2002]	Chapter 5, “Creational Patterns, Singleton”

6.3. TSM02-J. クラスの初期化中にバックグラウンドスレッドを使用しない

クラス初期化中にバックグラウンドスレッドを開始して使用する場合、クラスの初期化が循環し、デッドロック状態になるおそれがある。 たとえば、クラスの初期化を実行するメインスレッドがバックグラウンドスレッドの終了を待ち、バックグラウンドスレッドの方は、メインスレッドがクラスの初期化を終了するのを待つ状況である。このような問題は、クラスの初期化中にバックグラウンドスレッドにおいてデータベース接続を確立するような場合に発生しうる [Bloch 2005b]。

6.3.1. 違反コード (バックグラウンドスレッド)

以下の違反コードでは、static イニシャライザ(クラスのロード時に一度だけ実行される static 宣言されたコードブロック)が、クラス初期化の一環としてバックグラウンドスレッドを開始している。バックグラウンドスレッドは、データベース接続を初期化しようと試みるが、dbConnection を含む ConnectionFactory クラスの全メンバーが初期化されるまで待たなければならない。

```
public final class ConnectionFactory {
  private static Connection dbConnection;
  // 他のフィールド定義など
  static {
    Thread dbInitializerThread = new Thread(new Runnable() {
      @Override public void run() {
        // データベース接続を初期化する
        try {
          dbConnection = DriverManager.getConnection("connection string");
        } catch (SQLException e) {
          dbConnection = null;
        }
      }
    });

    // 他の初期化処理、たとえば、他のスレッドを開始する
    dbInitializerThread.start();
    try {
      dbInitializerThread.join();
    } catch (InterruptedException ie) {
      throw new AssertionError(ie);
    }
  }
  public static Connection getConnection() {
    if (dbConnection == null) {
      throw new IllegalStateException("Error initializing connection");
    }
    return dbConnection;
  }
  public static void main(String[] args) {
    // ...
    Connection connection = getConnection();
  }
}
```

静的に初期化されるフィールドは、それらが他のスレッドに可視となる前に、構築が完了することが保証される。(詳細は、ガイドライン「TSM03-J. 初期化が不完全なオブジェクトを公開しない」を参照。) したがって、バックグラウンドスレッドは、処理を進める前に、メインの(あるいはフォアグラウンドの)スレッドの初期化が終了するのを待つ必要がある。しかし、ConnectionFactory クラスのメインスレッドが join()メソッドを呼び出しており、その join()メソッドはバックグラウンドスレッドが終了するのを待っている。この相互依存関係によって、クラスの初期化処理に循環が発生し、デッドロックにつながる [Bloch 2005b]。

同様に、コンストラクタからスレッドを開始することは不適切である。(詳細は、ガイドライン「TSM01-J. オブジェクトの構築時に this 参照を逸出させない」を参照。)タスクを周

期的に実行するタイマーを作成し、その開始を初期化のためのコード内で行うと、生存性の問題を誘発しうる。

6.3.2. 適合コード (static イニシャライザでバックグラウンドスレッドを使用しない)

以下の適合コードでは、static イニシャライザからバックグラウンドスレッドを一切生成していない。代わりに、すべてのフィールドはメインスレッド内で初期化される。

```
public final class ConnectionFactory {
  private static Connection dbConnection;
  // 他のフィールドを定義する ...

  static {
    // データベース接続を初期化する
    try {
      dbConnection = DriverManager.getConnection("connection string");
    } catch (SQLException e) {
      dbConnection = null;
    }
    // 他の初期化処理 (いかなるスレッドも開始してはならない)
  }

  // ...
}
```

6.3.3. 適合コード (ThreadLocal オブジェクト)

以下の適合コードでは、ThreadLocal オブジェクトからデータベース接続を初期化しているので、全スレッドそれぞれがデータベース接続インスタンスを獲得できる。

```
public final class ConnectionFactory {
  private static final ThreadLocal<Connection> connectionHolder
    = new ThreadLocal<Connection>() {
    @Override public Connection initialValue() {
      try {
        Connection dbConnection = DriverManager.getConnection("connection string");
        return dbConnection;
      } catch (SQLException e) {
        return null;
      }
    }
  };

  // 他のフィールドを定義する...

  static {
    // 他の初期化処理 (いかなるスレッドも開始してはならない)
  }
  public static Connection getConnection() {
    Connection connection = connectionHolder.get();
    if (connection == null) {
      throw new IllegalStateException("Error initializing connection");
    }
    return connection;
  }

  public static void main(String[] args) {
    // ...
    Connection connection = getConnection();
  }
}
```

static イニシャライザは、クラスで共有される他のフィールドを初期化するために使うことができる。あるいは、initialValue()メソッドからフィールドを初期化することも可能である。

6.3.4. 例外

TSM02-EX1: フィールドへのアクセスを行わないスレッドであれば、クラス初期化中にバックグラウンドスレッドを開始してもかまわない。たとえば、以下に示す ObjectPreserver クラス(*Patterns in Java* [Grand 2002]に基づく)は、オブジェクト参照を格納する仕組みを提供しており、格納されたオブジェクトが参照されなくなっても、ガベージコレクションの対象にはならない。

```
public final class ObjectPreserver implements Runnable {
  private static final ObjectPreserver lifeLine = new ObjectPreserver();

  private ObjectPreserver() {
    Thread thread = new Thread(this);
    thread.setDaemon(true);
    thread.start(); // このオブジェクトを、生かし続ける
  }
  // このクラスも、HashMap クラスもガベージコレクションされることはない。
  // 更に、HashMap から他のオブジェクトへの参照は、このプロパティを示すことになる。
  private static final ConcurrentHashMap<Integer, Object> protectedMap
    = new ConcurrentHashMap<Integer, Object>();
  public synchronized void run() {
    try {
      wait();
    } catch (InterruptedException e) {
      Thread.currentThread().interrupt(); // 割込みステータスのリセットを行う
    }
  }

  // このメソッドに渡されるオブジェクトは、unpreserveObject()メソッドが
  // 呼ばれるまで、保存されることになる
  public static void preserveObject(Object obj) {
    protectedMap.put(0, obj);
  }
  // 毎回同じインスタンスを返している
  public static Object getObject() {
    return protectedMap.get(0);
  }
  // オブジェクトへの保護を解除し、ガベージコレクションを可能にする
  public static void unpreserveObject() {
    protectedMap.remove(0);
  }
}
```

これはシングルトンクラスである。初期化処理には、このクラスのインスタンスを使用するバックグラウンドスレッドの作成が含まれている。スレッドは、Object.wait()を呼び出すことにより無期限に待機状態となり、JVM が生存する間は存続する。このオブジェクトはデーモンスレッドにより管理されているので、スレッドが JVM の通常の終了を妨害することはない。

初期化処理はバックグラウンドスレッドを含んでいるが、そのスレッドはフィールドへのアクセスを行わないので、生存性や安全性の問題を引き起こすことはない。したがって、上記のコードは、安全かつ有用な例外である。

6.3.5. リスク評価

クラス初期化中にバックグラウンドスレッドを開始すると、デッドロック状態を引き起こす場合がある。

ガイドライン	深刻度	可能性	修正コスト	優先度	レベル
TSM02-J	低	中	高	P2	L3

6.3.6. 参考文献

[Bloch 2005b]	
[Grand 2002]	Chapter 5, “Creational Patterns, Singleton”

6.4. TSM03-J. 初期化が不完全なオブジェクトを公開しない

共有オブジェクトはその初期化中、オブジェクトを構築するスレッドのみによりアクセス可能でなければならない。ただし、一旦初期化が完了すれば、そのオブジェクトは安全に公開する(他のスレッドに対して可視とする)ことができる。Java メモリモデルは、オブジェクトの初期化を開始した直後から初期化が完了するまでの間、複数のスレッドからそのオブジェクトが参照されることを許している。したがって、初期化が不完全な状態のオブジェクトが公開されないようにすることは重要である。

このガイドラインでは、初期化途中で完全に初期化されていないメンバーオブジェクトのインスタンスへの参照を公開することを禁止する。ガイドライン「TSM01-J. オブジェクトの構築時に this 参照を逸出させない」では、処理中のオブジェクトの this 参照が逸出することを禁止している。

6.4.1. 違反コード

以下の違反コードでは、Foo クラスの initialize()メソッドで Helper オブジェクトを構築している。Helper オブジェクトのフィールドは、コンストラクタにより初期化されている。

```
class Foo {
  private Helper helper;

  public Helper getHelper() {
    return helper;
  }

  public void initialize() {
    helper = new Helper(42);
  }
}

public class Helper {
  private int n;

  public Helper(int n) {
    this.n = n;
  }
  // ...
}
```

あるスレッドが、initialize()メソッドの呼出し前に getHelper()メソッドを使用して helper にアクセスすると、そのスレッドは初期化されていない helper フィールドを参照することになるだろう。その後、あるスレッドが initialize()メソッドを、他のスレッドが getHelper()

メソッドを呼び出した場合、後者のスレッドは下記の内のいずれか一つの状態を認識することになる。

- 値が NULL の helper 参照
- フィールド n に 42 がセットされ、完全に初期化された Helper オブジェクト
- n は初期化されておらずデフォルト値 0 を持つ、初期化が不完全な Helper オブジェクト

Java メモリモデルでは、コンパイラが新しい Helper オブジェクトにメモリを割り当て、オブジェクトの初期化前に helper フィールドに代入することを許している。つまり、コンパイラは、helper インスタンスフィールドへの書込みと、Helper オブジェクトを初期化する書込み(即ち、this.n=n)の間で、前者が最初に発生するような順序変更をしてもよいのである。それゆえ、不完全な初期化状態の Helper オブジェクトのインスタンスが他のスレッドから参照されてしまうような競合ウィンドウが作られることになる。

他にも問題がある。二つのスレッドが initialize()メソッドを呼び出すと、Helper オブジェクトは二つ生成されるが、フィールド n は適切に初期化され、未使用の Helper オブジェクトはガベージコレクションされる。これは処理結果の正確性についての問題ではなく、性能面での問題である。

6.4.2. 適合コード (同期化)

以下の適合コードに示すように、不完全な状態のオブジェクト参照の公開は、メソッド同期の使用によって防ぐことができる。

```
class Foo {
  private Helper helper;

  public synchronized Helper getHelper() {
    return helper;
  }

  public synchronized void initialize() {
    helper = new Helper(42);
  }
}
```

両方のメソッドを同期することで、両者が同時に実行されないことを保証している。一方のスレッドが getHelper()メソッドを呼ぶ直前に他のスレッドが initialize()メソッドを呼ぶ場合、synchronized 宣言された initialize()メソッドは必ず先に終了する。synchronized キーワードは、二つのスレッド間の事前発生関係を確立する。これにより、getHelper()メソッドを呼び出すスレッドは、初期化が完了した Helper オブジェクトを読み取るか、あるいは何

も読み取らない (すなわち helper が null 参照を保持する)かのいずれかであることが保証される。このアプローチは、メンバーフィールドが可変であっても不変であってもオブジェクトを適切に公開することができる。

6.4.3. 適合コード (final 宣言されたフィールド)

helper フィールドが final 宣言されている場合、オブジェクトの参照が可視となる前にオブジェクトが完全に構築されていることが保証される。

```
class Foo {
  private final Helper helper;

  public Helper getHelper() {
    return helper;
  }

  public Foo() {
    helper = new Helper(42);
  }
}
```

上記の解決方法では、Foo のコンストラクタで helper フィールドに新規の Helper インスタンス割当てを要求している。*Java 言語仕様*の 17.5.2 節「構築中における final フィールドの読取り」[Gosling 2005]によれば、

> *あるスレッドにより構築されるオブジェクトの final フィールドの読取りは、コンストラクタ内の該当フィールドに対する初期化処理における事前発生の規則に基づいて順序付けられる。コンストラクタ内でフィールドがセットされた後に読取りが発生する場合、final フィールドに代入された値を読み取り、そうでない場合はデフォルト値を読み取る。*

したがって、Foo クラスのコンストラクタがその初期化を終了する前に、Helper インスタンスへの参照を公開してはならない (ガイドライン「TSM01-J. オブジェクトの構築時に this 参照を逸出させない」を参照)。

6.4.4. 適合コード (final フィールドとスレッドセーフコンポジション)

コレクションクラスの中には、自身が含む要素へのスレッドセーフなアクセスを提供するものもある。Helper オブジェクトがそのようなコレクションに格納される場合、そのオブジェクトの参照が可視となる前に初期化が完了することが保証される。以下の適合コードでは、helper フィールドを Vector<Helper>内にカプセル化している。

```
class Foo {
  private final Vector<Helper> helper;

  public Foo() {
    helper = new Vector<Helper>();
  }

  public Helper getHelper() {
    if (helper.isEmpty()) {
      initialize();
    }
    return helper.elementAt(0);
  }

  public synchronized void initialize() {
    if (helper.isEmpty()) {
      helper.add(new Helper(42));
    }
  }
}
```

helper フィールドを final 宣言し、アクセスが発生する前に Vector オブジェクトが生成されることを保証している。helper フィールドは、同期化された initialize ()メソッドを呼び出すことで安全に初期化することが可能であり、常に一つの Helper オブジェクトだけが Vector オブジェクトに追加される。getHelper()メソッドが initialize()メソッドより前に呼び出される場合、initialize()メソッドを呼び、クライアントによるヌルポインタ参照の可能性を回避する。getHelper()メソッドは単に Helper オブジェクトを返すだけなので同期化の必要はない。なぜなら、同期化された initialize()メソッドでは helper に新たな Helper オブジェクトを追加する前に helper が空かどうかを確認しており、Vector オブジェクトに二つ目のオブジェクトを追加するような競合状態を引き起こすことはありえない。

6.4.5. 適合コード (静的初期化)

以下の適合コードでは、helper フィールドは静的に初期化されており、そのフィールドが参照するオブジェクトが完全に初期化されてから可視となることを確実にしている。

```
// 不変の Foo クラス
final class Foo {
  private static final Helper helper = new Helper(42);

  public static Helper getHelper() {
    return helper;
  }
}
```

必須ではないが、helper フィールドを final 宣言し。クラスが不変であることを明示すべきである。

The Java Memory Model and Thread Specification の 9.2.3 節「static final 宣言されたフィールド」は以下のように記述している。[JSR-133 2004]

> *クラスの初期化ルールにより、すべてのスレッドによる static フィールドの読込みは、そのクラスの static イニシャライザ(すなわち static final 宣言されたフィールドの値をセットすることのできる唯一の場所)との間で同期がとられる。したがって、JMM において static final 宣言されたフィールドに関する特別な規則は必要ではない。*

6.4.6. 適合コード (不変オブジェクト - final フィールド、volatile 参照)

JMM では、オブジェクトが可視になる前に、オブジェクトの final 宣言されたフィールドが完全に初期化されることを保証している [Goetz 2006]。Helper クラスは、フィールド n を final 宣言することにより不変となる。また、ガイドライン「VNA01-J. 不変オブジェクトへの共有参照の可視性を確保する」に従い helper フィールドが volatile 宣言される場合、getHelper()メソッドを呼び出す任意のスレッドに対して、Helper オブジェクトへの参照は完全に初期化された後に可視となることが保証されている。

```
class Foo {
  private volatile Helper helper;

  public Helper getHelper() {
    return helper;
  }

  public void initialize() {
    helper = new Helper(42);
  }
}

// 不変の Helper クラス
public final class Helper {
  private final int n;

  public Helper(int n) {
    this.n = n;
  }
  // ...
}
```

上記の適合コードでは、helper フィールドが volatile 宣言され、かつ、Helper クラスが不変である必要がある。もし、Helper クラスが不変でなければ、コードはガイドライン「VNA06-J. オブジェクトへの参照を volatile 宣言することでメンバーの可視性が保証されると想定しない」に違反し、更に同期処理が必要となる(次の適合コードを参照)。また、helper フィールドが volatile 宣言されていないのであれば、ガイドライン「VNA01-J. 不変オブジェクトへの共有参照の可視性を確保する」に違反することになる。

Helper クラスに新しいインスタンスを返す public static 宣言したファクトリメソッドを定義してもよい。このアプローチでは、Helper インスタンスを private 宣言されたコンストラクタから作成することができる。

6.4.7. 適合コード (スレッドセーフな可変オブジェクト、volatile 参照)

Helper クラスが可変ではあるがスレッドセーフな場合、Foo クラスの helper フィールドを volatile 宣言して安全に公開することができる。

```
class Foo {
  private volatile Helper helper;

  public Helper getHelper() {
    return helper;
  }

  public void initialize() {
    helper = new Helper(42);
  }
}

// 可変ではあるが、スレッドセーフな Helper クラス
public class Helper {
  private volatile int n;
  private final Object lock = new Object();

  public Helper(int n) {
    this.n = n;
  }
  public void setN(int value) {
    synchronized (lock) {
      n = value;
    }
  }
}
```

Helper オブジェクトはその構築後に状態を変更することが可能なため、公開後に可変メンバーの可視性を確保するための同期化が必要である。上記適合コードでは、フィールド n を可視化するために、setN()メソッドを同期化している(ガイドライン「VNA06-J. オブジェクトへの参照を volatile 宣言することでメンバーの可視性が保証されると想定しない」を参照)。

Helper クラスが適切に同期されない場合、Foo クラスの helper フィールドを volatile 宣言することは、単に Helper オブジェクトの最初の公開の可視性を保証するだけで、その後の状態変更が可視となる保証はない。したがって、スレッドセーフではないオブジェクトを公開する場合、その参照を volatile 宣言するだけでは不十分である。

また、Foo クラスで helper フィールドが volatile 宣言されていない場合は、フィールド n の初期化と helper フィールドへの Helper オブジェクトの書込みとの間で事前発生関係が成立するように、フィールド n を volatile 宣言する。これは、ガイドライン「VNA06-J. オブジェクトへの参照を volatile 宣言することでメンバーの可視性が保証されると想定しない」に準じる。呼出し元(クラス Foo)による helper オブジェクトの volatile 宣言が期待できない場合に限り、フィールド n の volatile 宣言は必要である。

Helper クラスは public 宣言されているので、ガイドライン「LCK00-J. 信頼できないコードから使用されるクラスの同期化には private final ロックオブジェクトを使用する」に適合する同期を行うために private 宣言されたロックを使用している。

6.4.8. 例外

TSM03-EX1: 初期化が不完全なオブジェクトの使用を許さないクラスであれば、部分的にのみ初期化されたオブジェクトを公開してもかまわない。たとえば、初期化コードの最後で volatile 宣言された boolean 型のフラグを true にセットし、このフラグがセットされていなければクラスに定義されたメソッドが実行できないようにする場合など。

以下にこの手法を示す。

```
public class Helper {
  private int n;
  private volatile boolean initialized; // デフォルト値は false である

  public Helper(int n) {
    this.n = n;
    this.initialized = true;
  }
  public void doSomething() {
    if (!initialized) {
      throw new SecurityException("Cannot use partially initialized instance");
    }
    // ...
  }
  // ...
}
```

この手法では、初期化の完了前に Helper クラスのインスタンスへの参照が公開されたとしても、そのインスタンスを使用できないようにしている。Helper クラスのすべてのメソッドで初期化が終了したかどうかを判定するためのフラグをチェックしており、初期化が完了していないインスタンスは使用できない。

6.4.9. リスク評価

共有される可変データへのアクセスを同期化していないと、異なるスレッドが状態の異なるオブジェクト、あるいは、不完全に初期化された状態のオブジェクトを参照しうる。

ガイドライン	深刻度	可能性	修正コスト	優先度	レベル
TSM03-J	中	中	中	P8	L2

6.4.10. 参考文献

[Arnold 2006]	Section 14.10.2, “Final Fields and Security”
[Bloch 2001]	Item 48: “Synchronize access to shared mutable data”
[Goetz 2006]	Section 3.5.3, “Safe Publication Idioms”
[Goetz 2006c]	Pattern #2: “one-time safe publication”
[Pugh 2004]	
[Sun 2009b]	

6.5. TSM04-J. スレッド安全性の明文化にアノテーションを使用する [11]

Java 言語のアノテーション機能は、プログラムの設計意図を明文化するのに役立つ。ソースコードのアノテーションは、プログラムの構成要素にメタデータを関連付け、コンパイラやアナライザ、デバッガ、あるいは JVM がその内容を利用できるようにする仕組みである。アノテーションの中には、スレッドの安全性を記述したり、あるいは安全性が考慮されていないことを記述できるものもある。

6.5.1. 並行処理に関するアノテーション

並行処理に関する二組のアノテーションが公開されており、ライセンス上どんなコードでも利用できる。一つ目のアノテーションは、Java Concurrency in Practice(JCIP)に記述されている四つのアノテーションから構成されており [Goetz 2006]、jcip.net から JAR ファイルなどをダウンロードすることができる。JCIP に関するアノテーションは、Creative Commons Attribution License でリリースされている。

二つ目の、より大規模な並行処理に関するアノテーションは、SureLogic により提供されている。Apache License でリリースされており、surelogic.com からダウンロードすることができる。これらのアノテーションは SureLogic の JSure ツールで検証することができるが、ツールが利用不可能な場合でも、コードの文書化に有用である。JSure ツールは JCIP のアノテーションもサポートしており、SureLogic のアノテーションのセットには JCIP のアノテーションも含まれる。(JSure は JCIP の JAR ファイルの使用もサポートしている。)

これらのアノテーションを使用するには、前述の JAR ファイルの一方あるいは両方をダウンロードしてコードのビルドパスに追加する。これらのアノテーションを使ってスレッドの安全性について文書化する方法は以下のセクションで述べる。

6.5.2. スレッドの安全性を明文化

JCIP は、スレッドの安全性に関するプログラマの設計意図を記述するために三つのクラスレベルで使用するアノテーションを提供している。

@ThreadSafe アノテーションは、クラスがスレッドセーフであることを示すために用いる。このアノテーションは、いかなるアクセス(public 宣言されたフィールドの読取りや書込み、public 宣言されたメソッドの呼出し)もオブジェクトを矛盾した状態にはしないことを意味

[11] 2011 年 7 月現在、このガイドラインは削除されている。

する。たとえば、下記の Aircraft クラスでは、明文化されたロックポリシーの中で、スレッドセーフであることが明記されている。このクラスは、再入可能なロックを使用してフィールド x と y を保護している。

```
@ThreadSafe
@Region("private AircraftState")
@RegionLock("StateLock is stateLock protects AircraftState")
public final class Aircraft {
   private final Lock stateLock = new ReentrantLock();
   // ...
   @InRegion("AircraftState")
   private long x, y;
   // ...
   public void setPosition(long x, long y) {
      stateLock.lock();
      try {
         this.x = x;
         this.y = y;
      } finally {
         stateLock.unlock();
      }
   }
   // ...

}
```

@Region および@RegionLock の各アノテーションは、スレッドの安全性を約束するロックポリシーを明示している。

クラスのロックポリシーを明文化するために、たとえ@RegionLock あるいは@GuardedBy アノテーションが一回以上使用されたとしても、@ThreadSafe アノテーションが定義されていれば、レビュー担当者はそのクラスがスレッドセーフであることを直感的に知ることができる。

@Immutable アノテーションは、不変なクラスに適用される。不変なオブジェクトは、本質的にスレッドセーフである。不変なオブジェクトは構築が完了後、volatile 宣言された参照により公開され、複数のスレッドによって安全に共有されることになる。

以下の例では、不変な Point クラスを示している。

```
@Immutable
public final class Point {
   private final int f_x;
   private final int f_y;
   public Point(int x, int y) {
      f_x = x;
      f_y = y;
   }
   public int getX() {
      return f_x;
   }
   public int getY() {
      return f_y;
   }
}
```

Bloch は以下のように述べている[Bloch 2008] 。

> *enum 型変数については、不変性を明文化する必要はない。static なファクトリメソッドについては、返り値の型から明らかである場合を除いて、その返り値のスレッドの安全性について分かるようにしておく必要がある(たとえばCollections.synchronizedMap というメソッド名のように)。*

@NotThreadSafe アノテーションは、スレッドセーフではないクラスに適用される。ほとんどのクラスは、マルチスレッド環境で使用しても安全であるかどうかが明文化されていない。したがって、プログラマはクラスがスレッドセーフかどうかは容易には判定できない。このアノテーションは、クラスにおいてスレッドの安全性が欠如していることを明確に示している。

たとえば、java.util で提供されるほとんどのコレクションに関する実装は、スレッドセーフではない。java.util.ArrayList クラスであれば、このことを以下のように明文化できる。

```
package java.util.ArrayList;
@NotThreadSafe
public class ArrayList<E> extends ... {
   // ...
}
```

6.5.3. ロックポリシーの明文化

共有される状態変数を保護するために使用されているロックを、すべて明文化することは重要である。Goetz らは以下のように述べている[Goetz 2006] 。

複数のスレッドがアクセスする可変な状態変数は、その変数へのすべてのアクセスが同じロックのもとに行われる必要がある。このとき、その変数はロックにより保護されていると言われる。

この目的のために JCIP は @GuardedBy アノテーションを提供しており、一方で、SureLogic は @RegionLock アノテーションを提供している。@GuardedBy アノテーションが適用されるフィールドあるいはメソッドには、特定のロックを保持している場合にのみアクセスすることが可能である。これは固有ロックあるいは java.util.concurrent.locks のような動的なロックということになるであろう。

たとえば、以下の MovablePoint クラスでは、ArrayList クラスの memo を用いて過去の位置を記憶することのできる、移動可能なポイントを実装している。

```
@ThreadSafe
public final class MovablePoint {
  @GuardedBy("this")
    double xPos = 1.0;
  @GuardedBy("this")
    double yPos = 1.0;
  @GuardedBy("itself")
    static final List<MovablePoint> memo = new ArrayList<MovablePoint>();
  public void move(double slope, double distance) {
    synchronized (this) {
      rememberPoint(this);
      xPos += (1 / slope) * distance;
      yPos += slope * distance;
    }
  }

  public static void rememberPoint(MovablePoint value) {
    synchronized (memo) {
      memo.add(value);
    }
  }
}
```

xPos と yPos の各フィールドへの @GuardedBy アノテーションは、(これらのフィールドを更新する move()メソッドで行われているように)this 参照へのロックを保持することにより、各フィールドへのアクセスが保護されることを示している。memo への @GuardedBy アノテーションは、(rememberPoint()メソッドで行われているように)ArrayList オブジェクトへのロックがそのリストに含まれる各要素を保護していることを示している。

@GuardedBy アノテーションに関する問題の一つは、クラス内のフィールド間の関係を明確にすることができないことである。この制限は、SureLogic の @RegionLock アノテーションの使用により克服することができる。このアノテーションは、適用対象のクラスへの

新しい領域ロック(region lock)を宣言している。この宣言により、特有のロックオブジェクトをクラスの領域に関連付ける新しい名前付きのロックを作成する。ロックが保持される場合に限り、領域がアクセスされることになる。

たとえば、以下の SimpleLock のロックポリシーは、インスタンスへの同期化がその状態をすべて保護することを示している。

```
@RegionLock("SimpleLock is this protects Instance")
class Simple { ... }
```

@GuardedBy アノテーションとは異なり、@RegionLock アノテーションを使うことで、ロックポリシーに対して明示的、かつ、願わくば意味のある名前を付けることが可能になる。

ロックポリシーの命名に加えて、@Region アノテーションは、保護されている状態の領域の命名も許している。以下の例で示すように、その名前によって、状態とロックポリシーの対応を明らかにすることができる。

```
@Region("private AircraftPosition")
@RegionLock("StateLock is stateLock protects AircraftPosition")
public final class Aircraft {
   private final Lock stateLock = new ReentrantLock();

   @InRegion("AircraftPosition")
   private long x, y;

   @InRegion("AircraftPosition")
   private long altitude;
   // ...
   public void setPosition(long x, long y) {
      stateLock.lock();
      try {
         this.x = x;
         this.y = y;
      } finally {
         stateLock.unlock();
      }
   }
   // ...
}
```

上記の例において、StateLock という名前のロックポリシーは、stateLock へのロックにより、AircraftPosition と命名された領域(航空機の位置を表現するために使用される可変の状態を含む)が保護されることを示すために使用されている。

6.5.4. 可変オブジェクトの構築

オブジェクトは、オブジェクトを構築するスレッドに拘束されるので、オブジェクトの構築はロックポリシー上の例外として考えられる。オブジェクトは、そのインスタンスを作成する new 演算子を使用するスレッドに拘束される。オブジェクトは生成後に、他のスレッドに安全に公開することが可能となる。しかし、オブジェクトの共有は、そのインスタンスを生成したスレッドが容認するまでは行われない。オブジェクトを安全に公開するアプローチについては、ガイドライン「TSM01-J. オブジェクトの構築時に this 参照を逸出させない」で議論されており、@Unique("return")アノテーションにより簡潔に表現することが可能である。

たとえば、以下に示すコードでは、@Unique("return")アノテーションによって、コンストラクタから返されるオブジェクトの参照が一意である旨が明文化されている。

```
@RegionLock("Lock is this protects Instance")
public final class Example {
  private int x = 1;
  private int y;
  @Unique("return")
  public Example(int y) {
    this.y = y;
  }
  // ...
}
```

6.5.5. スレッド拘束ポリシーの明文化

Sutherland と Scherlis は、スレッド拘束（Thread Confinement）ポリシーを明文化することが可能なアノテーションを提案している。彼らのアプローチでは、コードをあるがままの形でアノテーションを検証することができる[Sutherland 2010]。

たとえば、以下のアノテーションでは、あるプログラムが最大でも一つの AWT イベント処理スレッドと複数の Compute スレッドを持ち、Compute スレッドが AWT のデータ構造やイベントを扱うことを禁じていることを、設計の意図として示している。

```
@ColorDeclare AWT, Compute
@IncompatibleColors AWT, Compute
@MaxColorCount AWT 1
```

6.5.6. 待機・通知プロトコルの文書化

Goetz は次のように述べている [Goetz 2006] 。

> *状態依存するクラスは、待機・通知プロトコルをサブクラスに完全に開示するか、サブクラスがそれらにまったく関与できないようにするべきである。 (これは「継承はそのための設計と文書化がなされなければ、禁止されるべきである」[EJ Item 15]の延長である。) 継承されることを想定された状態依存するクラスは、少なくとも条件キューとロックを開示し、条件述語と同期化ポリシーを文書化することが必要である。また、他の関連する条件変数の開示も必要になるかもしれない (状態依存するクラスが引き起こす最悪の事態は、その状態をサブクラスに開示するが、待機・通知プロトコルを文書化しないことである。これは、クラスがその状態変数をアクセス可能にするが、その不変項を文書化しないのと同じことである)。*

待機・通知プロトコルは、十分に明文化される必要があるが、現時点では、この目的に合うアノテーションについては確認できていない。

6.5.7. リスク評価

並行処理用のコードのアノテーションは、設計意図を明文化し、競合状態およびデータ競合の検出と防止を自動化するために使用することができる。

ガイドライン	深刻度	可能性	修正コスト	優先度	レベル
TSM04-J	低	中	中	P4	L3

6.5.8. 参考文献

[Bloch 2008]	Item 70: “Document thread safety”
[Goetz 2006]	
[Sutherland 2010]	

付録 用語定義

アトミック性

プリミティブデータへの操作に当てはめると、アトミック性が確保されているとは、そのデータへアクセス可能な他のスレッドは、必ず操作の開始前か完了後のデータを参照することになり、操作途中のデータの値を参照することはない。

安全性(safety)

安全性の主な目的は、すべてのオブジェクトがマルチスレッド環境で一貫した状態を維持できるように保証することである[Lea 2000a]。

インスタンス変数(instance variable)

インスタンス変数とは、static 宣言をしていないフィールドのことであり、そのフィールドが定義されたクラスのすべてのインスタンスに含まれている

オブジェクトの公開(publishing objects)

「オブジェクトを公開するとは、スコープ外からオブジェクトにアクセスできるようにすることである。たとえば、オブジェクトの参照を他のコードがアクセスできるところに格納する、private でないメソッドから参照を返す、参照を他のクラスのメソッドに引き渡すなどである」[Goetz 2006]。

飢餓状態(starvation)

一つ以上のスレッドが、他のスレッドが共有リソースにアクセスするのを長時間にわたって防げている状態のこと。たとえば、処理完了までに長時間を要する synchronized メソッドを呼び出すスレッドは、他のスレッドを飢餓状態にすることになる。

揮発性(volatile)

「volatile フィールド (§8.3.1.4) への書込みは、すべての後続の読取り操作の前に発生(happens-before)する」[Gosling 2005]。「volatile 変数のマスターコピーへの操作は、メインメモリがスレッドに代わり、スレッドが要求した順序通りに実行する」[Sun 1999b]。volatile 変数へのアクセスは逐次一貫性を有するが、これは、各操作へのコンパイラによる最適化が行われていないことを意味する。変数を volatile 宣言することは、あるスレッドが

その変数を修正した場合、すべてのスレッドがその変数の最新の値を見ることを保証する。揮発性は、プリミティブ値のアトミックな読取りおよび書込みを保証する。しかし、変数への加算(read-modify-write シーケンス)のような複合操作のアトミック性は保証しない。

競合状態(race condition)

「一般的な競合は、予期せぬ(nondeterministic)処理の実行につながり、意図した通りに処理が実行されるべきプログラムの欠陥である」[Netzer 1992]。「競合状態は、処理結果の正確性が、複数スレッドの処理タイミングや実行順序などのランタイム動作に、依存する場合に発生する」[Goetz 2006]。

競合するアクセス(conflicting accesses)

同一変数に対して、二つのアクセス(読込みまたは書込み)が行われ、アクセスの少なくとも一方が書込みであるもの[Gosling 2005]。

サニタイゼーション(sanitization)

サニタイゼーションは、入力を検証して、複雑なサブシステムの入力要件に一致する表現に変換することを意味する用語である。たとえば、データベースでは、データを保存する前に、無効な文字をすべてエスケープあるいは削除することを要求するかもしれない。入力のサニタイゼーションとは、文字を削除、置換、エンコード、あるいはエスケープすることにより、入力内容から不要な文字を消去することを意味する。

事前発生順序(happens-before order)

「二つの動作は、事前発生関係(happens-before relationship)によって順序付けることができる。ある動作が他の動作よりも事前発生する場合、最初の動作は二番目の動作から可視となり、それよりも前に順序付けられる。[…] 二つの動作間に事前発生関係が存在しても、必ずその順序通りに実装する必要があるわけではない。順序の並び替えが行われたとしても、正しい実行結果と整合性を保持していれば良い。[…] より具体的に表現すると、二つの動作が事前発生の関係を共有していたとしても、事前発生の関係を共有しないコードから、それらが必ずしも該当順序で実行されているように見える必要はない。たとえば、あるスレッドの複数の書込みが、他のスレッドの読取りと競合する時、読取りからは書込みが異なる順序で行われることが確認できることもある」[Gosling 2005]。

初期化の安全性

「オブジェクトは、そのコンストラクタが完了した時点で完全に初期化されたと言える。完全に初期化が済んだ後のオブジェクトを参照するスレッドは、該当オブジェクトの final フィールドの初期値を正しく参照できることが保証されている」[Gosling 2005]。

逐次一貫性(sequential consistency)

「逐次一貫性は、プログラムの実行の可視性と順序付けに関するとても強い保証である。逐次一貫性を有する実行においては、読取りや書込みなどの個々のすべての動作を含む全体的な実行順序が存在し、この実行順序はプログラム順序と整合性があり、かつ個々の動作がアトミックで、すべてのスレッドに即座に可視となる。[…]プログラムが正しく同期化されている場合、プログラムのすべての実行は逐次一貫性を有していることになる(§ 17.4.3)」[Gosling 2005]。逐次一貫性を有することは、動作ステートメントに関してコンパイラによる最適化が行われないことを示唆している。逐次一貫性をメモリモデルとして採用し、他のプリミティブの使用を却下することは、コンパイラによるコードの最適化や順序替えができなくなるため過度な制限となるだろう [Gosling 2005]。

条件述語(condition predicate)

条件述語はクラスの状態変数から構成された条件式で、スレッドが実行を継続するにはその結果は真(true)でなければならない。スレッドは、Object.wait()、Thread.sleep()、あるいはその他の仕組みにより実行を休止するが、その後、通知を受信しかつ条件述語が真となった時に処理を再開する [Goetz 2006]。

信頼できないコード(untrusted code)

その実行が一定の危害を引き起こすおそれのある出所不明のコードのこと。信頼できないコードに必ずしも悪意があるとは限らないが、このことを自動的に判断するのは通常困難である。したがって、信頼できないコードはサンドボックス環境で実行した方がよい。

信頼できるコード(trusted code)

Java API を構成しているかどうかに関係なく、組み込みクラスローダ（primordial class loader）によりロードされるコードのこと。本ドキュメントでは、この意味を拡張しており、既知のコードであり信頼できないコードには欠けている各種の許可が付与されているコードも、信頼できるコードとしている。この定義により、信頼できないコードと信頼できるコードが一つのクラスローダ(必ずしも組み込みクラスローダである必要はない)のネーム

スペースに共存することが可能となる。その場合、セキュリティポリシーは、信頼できるコードには適切な特権を割り当て、信頼できないコードには特権を与えないことで、両者を明確に区別しなくてはならない。

スレッドセーフ(thread-safe)

オブジェクトがスレッドセーフであるとは、データ競合を生じることなく複数のスレッドによって共有可能であることを意味する。「スレッドセーフなクラスは同期処理を内部的に実装しているので、複数のスレッドが、追加の同期処理を必要とせず、public 宣言されたインターフェースを通してアクセスすることができる」[Goetz 2006]。不変なクラスは、定義上スレッドセーフである。可変なクラスも、適切に同期が行われていれば、スレッドセーフとなる。

生存性(liveness)

すべての操作あるいはメソッド呼出しが、中断せずに最後まで実行される。

脆弱性

「攻撃者が明示的あるいは暗黙のセキュリティポリシーを破ることを可能にする、一連の条件のこと」[Seacord 2005]。

全順序(total order)

ある集合において、任意の二つの要素間に順序が付けられること。 たとえば、<= (以下である)は、整数における全順序である、すなわち、任意の二つの整数に関して、それらの内の一方が他方より小さいか等しいことを意味する [Black 2004b]。

データ競合

「同一変数への操作が競合するアクセスであり、事前発生関係により順序付けられていない」[Gosling 2005]。

デッドロック

複数のスレッドが互いのロックが解放されるのを待って自身の処理をブロックしている状況のこと。どちらのスレッドも先に進むことができない。

同期化(synchronization)

「プログラミング言語 Java は、スレッド間でコミュニケーションを取るためのメカニズムを複数提供している。これらの中で最も基本的な方法は、モニタ(monitor)を用いて実装された同期化（synchronization）である。各オブジェクトは、それぞれ一つのモニタに関連付けられる。スレッドは、このモニタをロックまたはアンロックすることができる。一度に一つのスレッドだけが、特定のモニタをロックすることを許されているため、該当モニタをロックしようとする他のスレッドは、そのロックを獲得できるまでブロックすることになる」[Gosling 2005]。

半順序(partial order)

必ずしもすべてではない、いくつかの要素が組み合わされた順序。たとえば、集合{a、b}および{a、c、d}は、集合{a, b, c, d}の部分集合ではあるが、いずれも他方の部分集合ではない。「～の部分集合」とは、集合における半順序を意味している。 [Black 2004a]

ヒープメモリ

「スレッド間で共有可能なメモリを共有メモリ(shared memory)、あるいはヒープメモリ(heap memory)と呼ぶ。すべてのインスタンスフィールド、静的フィールド、配列要素はヒープメモリに格納される。[...] ローカル変数(§14.4)、メソッドの仮引数 (§8.4.1)、例外ハンドラの引数はスレッド間で共有されず、メモリモデルによる影響は受けない」[Gosling 2005]。

不変(immutable)

不変オブジェクトとは、初期化後にその状態を変更することができないオブジェクトである。
「次の条件を満たすオブジェクトは不変オブジェクトである。

- 構築後、その状態を変えることができない。
- すべてのフィールドが final である[12]。
- 正しく構築されている (構築途中に this 参照が逸出しない)。

[12] 厳密には、すべてのフィールドが final でない不変オブジェクトも存在する。String はそのようなクラスの一例であるが、このようなクラスを作成するには、Java メモリモデルを深く理解した上で、データ競合が無害であることを慎重に確認する必要がある。(補足: String は最初に hashCode が呼ばれた時にハッシュ値の遅延計算を行い、その結果を final

でないフィールドに保存する。これが許されるのは、そのフィールドが取る値がデフォルト以外には、不変な状態から得られる常に計算結果が同じハッシュ値のただ一つしかないためである。…)」[Goetz 2006]。不変オブジェクトは本質的にスレッドセーフである。不変オブジェクトは、複数のスレッドにより共有されるか、同期化なしで公開されるかもしれないが、可視性を確保するためには、それらのオブジェクトへの参照を含むフィールドを volatile 宣言することが、通常必要とされている。不変オブジェクトは可変のサブオブジェクトを含んでもかまわないが、その場合不変オブジェクトの構築後にサブオブジェクトの状態が変更されてはならない。

プログラム順序(program order)

各スレッドのセマンティクスに従い形成され、実行されるスレッド間動作の実行順序。「プログラム順序とは、「.class ファイル」に含まれるバイトコードが、制御フローの値に従い実行される順序である。」(David Holmes、JMM メーリングリスト)。

メモリモデル

「Java プログラミング言語のメモリモデルは、メモリへのアクセスをどのように順序付けるか、およびそれらがいつ可視となることが保証されるかを規定している」[Arnold 2006]。「あるプログラムとそのプログラムの実行トレースが与えられた場合、メモリモデルによって、その実行トレースがプログラムの正しい動作であるかどうかを説明付けることができる」[Gosling 2005]。

よそ者メソッド(alien method)

「クラス C にとってよそ者のメソッドとは、クラス C がその振舞いを完全に規定できないメソッドであり、他のクラスのメソッドや、クラス C が持つ private でも final でもないオーバーライド可能なメソッドも含まれる」[Goetz 2006]。

割込みポリシー(interruption policy)

「割込みポリシーは、スレッドが割込み要求をどのように処理するかを決定する。割込み検出時に実行する処理、割込み発生時にアトミックに処理すべき処理単位、割込みへの応答タイミングなど」[Goetz 2006]。

参考文献

ここで示されている URL は、この文書(原書)の出版時点で有効なものである。

[Abadi 1996]

Abadi, Martin & Needham, Roger. “Prudent Engineering Practice for Cryptographic Protocols.” *IEEE* Transactions *on Software Engineering 22*, 1 (January1996): 6 - 15.

[Apache 2008]

Apache. *Class FunctionTable*, *Field detail, public static FuncLoader m_functions*. http://www.stylusstudio.com/api/xalan-j_2_6_0/org/apache/xpath/compiler/FunctionTable.htm (2008).

[Apache 2009a]

Apache Software Foundation. *Apache Tomcat 6.0 – Changelog.* http://tomcat.apache.org/tomcat-6.0-doc/changelog.html (2009).

[Apache 2009b]

Apache Software Foundation. *Apache Tomcat 6.x Vulnerabilities.* http://tomcat.apache.org/security-6.html (2009).

[Arnold 2006]

Arnold, Ken, Gosling, James, & Holmes, David. *The Java Programming Language, Fourth Edition*. Addison Wesley Professional, 2006.

[Austin 2000]

Austin, Calvin & Pawlan, Monica. *Advanced Programming for the Java 2 Platform*. Addison Wesley Longman, 2000.

[Black 2004a]

Black, Paul E. & Tananbaum, Paul J. *Partial Order.* http://www.itl.nist.gov/div897/sqg/dads/HTML/partialorder.html (2004).

[Black 2004b]

Black, Paul E. & Tananbaum, Paul J. *Total Order.* http://www.itl.nist.gov/div897/sqg/dads/HTML/totalorder.html (2004).

[Bloch 2001]

Bloch, Joshua. *Effective Java, Programming Language Guide*. Addison Wesley, 2001.

[Bloch 2005a]

Bloch, Joshua & Gafter, Neal. *Java Puzzlers: Traps, Pitfalls, and Corner Cases*. Pearson Education, Inc., 2005.

[Bloch 2005b]

Bloch, Joshua & Gafter, Neal. "Yet More Programming Puzzlers," *Proceedings of the JavaOne Conference.* San Francisco, CA, June 2005.

[Bloch 2007a]

Bloch, Joshua. "Effective Java Reloaded: This Time It's (not) for Real," *Proceedings of the JavaOne Conference.* San Francisco, CA, May 2007.

[Bloch 2008]

Bloch, Joshua. *Effective Java, 2nd edition*. Addison Wesley, 2008.

[Bloch 2009]

Bloch, Joshua & Gafter, Neal. "Return of the Puzzlers: Schlock and Awe," *Proceedings of the JavaOne Conference.* San Francisco, CA, June 2009.

[Boehm 2005]

Boem, Hans J. "Finalization, Threads, and the Java Technology-Based Memory Model," *Proceedings of the JavaOne Conference.* San Francisco, CA, June 2005.

[Bray 2008]

Bray, Tim, Paoli, Jean, Sperberg-McQueen, C.M., Maler, Eve, & Yergeau, Francois (eds). *Extensible Markup Language (XML) 1.0 (Fifth Edition).* http://www.w3.org/TR/xml/ (2008).

[Campione 1996]

Campione, Mary & Walrath, Kathy. *The Java Tutorial*. http://www.telecom.ntua.gr/HTML.Tutorials/index.html (1996).

[CCITT 1988]

CCITT. *CCITT Blue Book, Recommendation X.509 and IS0 9594-8: The Directory-Authentication Framework*. Technical Report, Geneva, 1988.

[Chan 1999]

Chan, Patrick, Lee, Rosanna, & Kramer, Douglas. *The Java Class Libraries: Supplement for the Java 2 Platform, v1.2, second edition, Volume 1*. Prentice Hall, 1999.

[Chess 2007]

Chess, Brian & West, Jacob. *Secure Programming with Static Analysis*. Addison-Wesley Professional, 2007.

[Christudas 2005]

Christudas, Binudlas. *Internals of Java Class Loading.* http://onjava.com/pub/a/onjava/2005/01/26/classloading.html (2005).

[Coomes 2007]

Coomes, Johm, Peter, Kessler, & Printezis, Tony. “Garbage Collection-Friendly Programming,” *Proceedings of the JavaOne Conference.* San Francisco, CA, May 2007.

[Cunningham 1995]

Cunningham, Ward. “The CHECKS Pattern Language of Information Integrity,” *Pattern Languages of Program Design*. Edited by James O. Coplien and Douglas C Schmidt. Addison-Wesley, 1995.

[Daconta 2000]

Daconta, Michael C. *When Runtime.exec() Won't.* http://www.javaworld.com/javaworld/jw-12-2000/jw-1229-traps.html (2000).

[Daconta 2003]

Daconta, Michael C., Smith, Kevin T., Avondolio, Donald, & Richardson, W. Clay. *More Java Pitfalls*. Wiley Publishing Inc., 2003.

[Davis 2008]

Davis, Mark & Suignard, Michel. *Unicode Technical Report #36, Unicode Security Considerations.* http://www.unicode.org/reports/tr36/ (2008).

[Davis 2009]

Davis, Mark, Whistler, Ken, & Martin Dürst. *Unicode Standard Annex #15, Unicode Normalization Forms*. http://unicode.org/reports/tr15/ (2009).

[Dormann 2008]

Dormann, Will. *Signed Java Applet Security: Worse than ActiveX?.* http://www.cert.org/blogs/vuls/2008/06/signed_java_security_worse_tha.html (2008).

[Darwin 2004]

Darwin, Ian F. *Java Cookbook*. O'Reilly Media, 2004.

[Doshi 2003]

Doshi, Gunjan. *Best Practices for Exception Handling*. http://onjava.com/pub/a/onjava/2003/11/19/exceptions.html (2003).

[ESA 2005]

European Space Agency (EAS) Board for Software Standardisation and Control (BSSC). *Java Coding Standards.* ftp://ftp.estec.esa.nl/pub/wm/wme/bssc/Java-Coding-Standards-20050303-releaseA.pdf (2005).

[FindBugs 2008]

FindBugs. *FindBugs Bug Descriptions.* http://findbugs.sourceforge.net/bugDescriptions.html (2008).

[Fisher 2003]

Fisher, Maydene, Ellis, Jon, & Bruce, Jonathan. *DBC API Tutorial and Reference, 3rd edition.* Prentice Hall, The Java Series, 2003.

[Flanagan 2005]

Flanagan, David. *Java in a Nutshell, 5th edition.* O'Reilly Media, Inc., 2005.

[Fortify 2008]

Fortify Softwaree. *Fortify Taxonomy: Software Security Errors.* http://www.fortify.com/vulncat/en/vulncat/index.html (2008).

[Fox 2001]

Fox, Joshua. *When is a Singleton not a Singleton?* Sun Developer Network (SDN), 2001.

[Gafter 2006]

Gafter, Neal. *Thoughts about the Future of Java Programming.* http://gafter.blogspot.com/ (2006-2010).

[Gamma 1995]

Gamma, Erich, Helm, Richard, Johnson, Ralph, & Vlissides, John M. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series, 1995.

[Garms 2001]

Garms, Jess & Somerfield, Daniel. *Professional Java Security*. Wrox Press Ltd., 2001.

[Goetz 2002]

Goetz, Brian. *Java theory and practice: Don't let the "this" reference escape during construction*. http://www.ibm.com/developerworks/java/library/j-jtp0618.html (2002).

[Goetz 2004a]

Goetz, Brian. *Java theory and practice: Garbage collection and performance*. http://www.ibm.com/developerworks/java/library/j-jtp01274.html (2004).

[Goetz 2004b]

Goetz, Brian. *Java theory and practice: The exceptions debate: To check, or not to check?* http://www.ibm.com/developerworks/java/library/j-jtp05254.html (2004).

[Goetz 2004c]

Goetz, Brian. *Java Theory and Practice: Going Atomic.* http://www.ibm.com/developerworks/java/library/j-jtp11234/ (2004).

[Goetz 2005a]

Goetz, Brian. *Java theory and practice: Be a good (event) listener, Guidelines for writing and supporting event listeners*. http://www.ibm.com/developerworks/java/library/j-jtp07265/index.html (2005).

[Goetz 2005b]

Goetz, Brian. *Java Theory and Practice: Plugging Memory Leaks with Weak References.* http://www.ibm.com/developerworks/java/library/j-jtp11225/ (2005).

[Goetz 2006]

Goetz, Brian, Pierels, Tim, Bloch, Joshua, Bowbeer, Joseph, Holmes, David, & Lea, Doug. *Java Concurrency in Practice*. Addison Wesley Professional, 2006.

[Goetz 2006b]

Goetz, Brian. *Java Theory and Practice: Good Housekeeping Practices.* http://www.ibm.com/developerworks/java/library/j-jtp03216.html (2006).

[Goetz 2006c]

Goetz, Brian. *Java theory and practice: Managing volatility, Guidelines for using volatile variables*. http://www.ibm.com/developerworks/java/library/j-jtp06197.html (2006).

[Goldberg 1991]

Goldberg, David. *What Every Computer Scientist Should Know About Floating-Point Arithmetic.* http://docs.sun.com/source/806-3568/ncg_goldberg.html (1991).

[Gong 2003]

Gong, LI, Ellison, Gary, & Dageford, Mary. *Inside Java 2 Platform Security: Architecture, API Design, and Implementation, 2nd edition*. Prentice Hall, The Java Series, 2003.

[Gosling 2005]

Gosling, James, Joy, Bill, Steel, Guy, & Bracha, Gilad. *Java Language Specification, 3rd edition.* Addison Wesley, 2005.

[Grand 2002]

Grand, Mark. *Patterns in Java, Volume 1, Second Edition*. Wiley, 2002.

[Greanier 2000]

Greanier, Todd. *Discover the Secrets of the Java Serialization API.* http://java.sun.com/developer/technicalArticles/Programming/serialization/ (2000).

[Green 2008]

Green, Roedy. *Canadian Mind Products Java & Internet Glossary.* http://mindprod.com/jgloss/jgloss.html (2008).

[Grosso 2001]

Grosso, William. *Java RMI*. O'Reilly Media, 2001.

[Gupta 2005]

Gupta, Satish Chandra & Palanki, Rajeev. *Java Memory Leaks - Catch Me if You Can*. http://www.ibm.com/developerworks/rational/library/05/0816_GuptaPalanki/ (2005).

[Haack 2007]

Haack, Christian, Poll, Erik, Schafer, Jan, & Schubert, Alesky. *Immutable Objects for a Java-Like Language,"* 347-362*. Proceedings of the 16th European Conference on Programming.* Braga, Portugal. Springer-Verlag, 2007.

[Haggar 2000]

Haggar, Peter. *Practical Java Programming Language Guide.* Addison-Wesley Professional, 2000.

[Halloway 2000]

Halloway, Stuart. *Java Developer Connection Tech Tips.* http://javaservice.net/~java/bbs/read.cgi?m=devtip&b=jdc&c=r_p_p&n=954297433 **(2000).**

[Harold 1997]

Harold, Elliotte Rusty. *Java Secrets.* Wiley, 1997.

[Harold 1999]

Harold, Elliotte Rusty. *Java I/O.* O'Reilly Media, 1999.

[Harold 2006]

Harold, Elliotte Rusty. *Java I/O, 2nd Edition.* O'Reilly Media, 2006.

[Hawtin 2008]

Hawtin, Thomas. *Secure Coding Antipatterns: Preventing Attacks and Avoiding Vulnerabilities.* http://www.makeitfly.co.uk/Presentations/london-securecoding.pdf (2008).

[Henney 2003]

Henney, Kevlin. *Null Object, Something for Nothing*. http://www.two-sdg.demon.co.uk/curbralan/papers/europlop/NullObject.pdf (2003).

[Hitchens 2002]

Hitchens, Ron. *Java NIO*. O'Reilly Media, 2002.

[Hornig 2007]

Hornig, Charles. *Advanced Java Globalization.* http://developers.sun.com/learning/javaoneonline/2007/pdf/TS-2873.pdf (2007).

[Horstmann 2004]

Horstmann, Cay & Cornell, Gary. *Core Java 2 Volume I - Fundamentals, Seventh Edition*. Prentice Hall PTR, 2004.

[Hovemeyer 2007]

Hovemeyer, David & Pugh, William. "Finding more null pointer bugs, but not too many," *Proceedings of the 7th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*. San Diego, CA. June 2007.

[Hunt 1998]

Hunt, J. & Long, F. *Java's reliability: an analysis of software defects in Java*. http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=00722326 (1998).

[IEC 2006]

International Electrotechnical Commission. *Analysis techniques for system reliability - Procedure for failure mode and effects analysis (FMEA)*, 2nd ed. (IEC 60812). IEC, January 2006.

[JSR-133 2004]

JSR-133. *JSR-133: Java Memory Model and Thread Specification.* http://www.cs.umd.edu/~pugh/java/memoryModel/jsr133.pdf (2004).

[Kabanov 2009]

Kabanov, Jevgengi. *The Ultimate Java Puzzler.* http://dow.ngra.de/2009/02/16/the-ultimate-java-puzzler/ (2009).

[Kabutz 2001]

Kabuts, Heinz M. *The Java Specialists' Newsletter*. http://www.javaspecialists.eu/archive/archive.jsp (2001).

[Kalinovsky 2004]

Kalinovsky, Ales. *Covert Java: Techniques for Decompiling, Patching, and Reverse Engineering*. SAMS Publishing, 2004.

[Knoernschild 2001]

Knoernschild, Kirk. *Java Design: Objects, UML, and Process*. Addison-Wesley Professional, 2001.

[Lai 2008]

Lai, C. "Java Insecurity: Accounting for Subtleties That Can Compromise Code,"*Software, IEEE 25,* 1 (Jan-Feb 2008): 13-19.

[Langer 2008]

Langer, Angelica. *Java Generics FAQ.* http://www.angelikalanger.com/GenericsFAQ/FAQSections/ProgrammingIdioms.html (2008).

[Lea 2000a]

Lea, Doug. *Concurrent Programming in Java, 2nd edition*. Addison Wesley, 2000.

[Lea 2000b]

Lea, Doug & Pugh, William. *Correct and Efficient Synchronization of Java Technology based Threads.* http://www.cs.umd.edu/~pugh/java/memoryModel/TS-754.pdf (2000).

[Lea 2008]

Lea, Doug. *The JSR-133 Cookbook for Compiler Writers.* http://g.oswego.edu/dl/jmm/cookbook.html (2008).

[Lee 2009]

Lee, Sangin, Somani, Mahesh, & Saha, Debashis. *Robust and Scalable Concurrent Programming: Lessons from the Trenches*. Oracle, 2009.

[Liang 1997]

Liang, Sheng. *The Java Native Interface, Programmer's Guide and Specification*. Addison-Wesley, 1997.

[Liang 1998]

Liang, Sheng & Bracha, Gilad. "Dynamic Class Loading in the Java Virtual Machine," *Proceedings of the 13th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, Vancouver, BC, 1998, ACM, 1998.

[Lieberman 1986]

Lieberman, Henry. "Using prototypical objects to implement shared behavior in object-oriented systems," *Proceedings of the 1986 conference on Object-oriented programming systems, languages and applications,* Portland, ME, ACM, 1986.

[Lo 2005]

Lo, Chia-Tien Dan, Srisa-an, Witawas, & Chang, J. Morris. "Security Issues in Garbage Collection," *STSC Crosstalk* (October 2005).

[Long 2005]

Long, Fred. *Software Vulnerabilities in Java* (CMU/SEI -2004-TN-044). Software Engineering Institute, Carnegie Mellon University, 2005. http://www.sei.cmu.edu/library/abstracts/reports/05tn044.cfm.

[Low 1997]

Low, Douglas. *Protecting Java Code via Obfuscation.* http://www.cs.arizona.edu/~collberg/Research/Students/DouglasLow/obfuscation.html (1997).

[Macgregor 1998]

Macgregor, Robert, Durbin, Dave, Owlett, John, &Yeomans, Andrew. *Java Network Security*. Prentice Hall, 1998.

[Mak 2002]

Mak, Ronald. *Java Number Cruncher, The Java Programmer's Guide to Numerical Computing*. Prentice Hall, 2002.

[Manson 2004]

Manson, Jeremy & Goetz, Brian. *JSR 133 (Java Memory Model) FAQ.* http://www.cs.umd.edu/~pugh/java/memoryModel/jsr-133-faq.html (2004).

[Marco 1999]

Pistoia, Marco, Reller, Duane F., Gupta, Deepak, Nagnur, Milind, & Ramani, Ashok K. *Java 2 Network Security*. IBM Corporation (1999).

[Mcgraw 1998]

Mcgraw, Gary & Felten, Edward. *Twelve rules for developing more secure Java code*. http://www.javaworld.com/javaworld/jw-12-1998/jw-12-securityrules.html (1998).

[Mcgraw 1999]

Mcgraw, Gary & Felten, Edward W. *Securing Java, Getting Down to Business with Mobile Code*. Wiley, 1999.

[Microsoft 2009]

Microsoft. *Using SQL Escape Sequences*. http://msdn.microsoft.com/en-us/library/ms378045%28SQL.90%29.aspx (2009).

[Miller 2009]

Miller, Alex. *Java Platform Concurrency Gotchas.* http://www.slideshare.net/alexmiller/java-concurrency-gotchas (2009).

[MITRE 2010]

MITRE. *Common Weakness Enumeration.* http://cwe.mitre.org/ (2010).

[Mocha 2007]

Mocha. *Mocha, the Java Decompiler.* http://www.brouhaha.com/~eric/software/mocha/ (2007).

[Muchow 2001]

Muchow, John W. *MIDlet Packaging with J2ME.* http://www.onjava.com/pub/a/onjava/2001/04/26/midlet.html (2001).

[Naftalin 2006]

Naftalin, Maurice & Wadler, Philip. *Java Generics and Collections.* O'Reilly Media, 2006.

[Netzer 1992]

Netzer, Robert H. & Miller, Barton, P. "What Are Race Conditions? Some Issues and Formalization." *ACM Letters on Programming Languages and Systems (LOPLAS) 1,1* (March 1992): 74-88.

[Neward 2004]

Neward, Ted. *Effective Enterprise Java.* Addison Wesley Professional, 2004.

[Nolan 2004]

Nolan, Godfrey. *Decompiling Java.* Apress, 2004.

[Oaks 1999]

Oaks, Scott & Wong, Henry. *Java Threads (2nd Edition.)* O'Reilly Media, 1999.

[Oaks 2001]

Oaks, Scott. *Java Security.* O'Reilly Media, 2001.

[Oaks 2004]

Oaks, Scott & Wong, Henry. *Java Threads (3rd Edition)*. O'Reilly Media, 2004.

[O'Reilly 2003]

O'Reilly Media. Java Enterprise Best Practices. 2003.

[OWASP 2007]

OWASP. *OWASP TOP 10 FOR JAVA EE.*
https://www.owasp.org/images/8/89/OWASP_Top_10_2007_for_JEE.pdf (2007).

[OWASP 2008]

OWASP. *OWASP.* http://www.owasp.org/index.php/Main_Page (2008).

[Philion 2003]

Philion, Paul. *Beware the Dangers of Generic Exceptions.*
http://www.javaworld.com/javaworld/jw-10-2003/jw-1003-generics.html (2003).

[Pistoia 2004]

Pistoia, Marco, Nagaratnam, Nataraj, Koved, Larry, & Nadalin, Anthony. *Enterprise Java Security: Building Secure J2EE Applications.* Addison Wesley, 2004.

[Pugh 2004]

Pugh, William. *The Java Memory Model (discussions reference).*
http://www.cs.umd.edu/~pugh/java/memoryModel/ (2004).

[Pugh 2008]

Pugh, William. *Defective Java Code: Turning WTF Code into a Learning Experience.* http://72.5.124.65/learning/javaoneonline/j1sessn.jsp?sessn=TS-6589&yr=2008&track=javase (2008).

[Reasoning 2003]

Reasoning Inspection Service. *Defect Data Tomcat v 1.4.24*. http://www.reasoning.com/pdf/Tomcat_Defect_Report.pdf (2003).

[Rotem-Gal-Oz 2008]

Rotem-Gal-Oz, Arnon. *Fallacies of Distributed Computing Explained.* http://www.rgoarchitects.com/Files/fallacies.pdf (2008).

[Roubtsov 2003a]

Roubtsov, Vladimir. *Breaking Java Exception-Handling Rules is Easy.* http://www.javaworld.com/javaworld/javaqa/2003-02/02-qa-0228-evilthrow.html (2003).

[Roubtsov 2003b]

Roubtsov, Vladimir. *Into the Mist of Serialization Myths.* http://www.javaworld.com/javaworld/javaqa/2003-06/02-qa-0627-mythser.html?page=1 (2003).

[Schneier 2000]

Schneier, Bruce. *Secrets and Lies—Digital Security in a Networked World*. John Wiley and Sons, 2000.

[Schildt 2007]

Schildt, Herb. *Herb Schildt's Java Programming Cookbook*. McGraw-Hill, 2007.

[Schoenefeld 2004]

Schoenefeld. (Nov. 2004). *Java Vulnerabilities in Opera 7.54 BUGTRAQ Mailing List* [online]. Available email: bugtraq@securityfocus.com.

[Schwarz 2004]

Schwarz, Don. *Avoiding Checked Exceptions*.
http://www.oreillynet.com/onjava/blog/2004/09/avoiding_checked_exceptions.html (2004).

[Schweisguth 2003]

Schweisguth, Dave. *Java Tip 134: When catching exceptions, don't cast your net too wide.*
http://www.javaworld.com/javaworld/javatips/jw-javatip134.html?page=2 (2003).

[Seacord 2005]

Seacord, Robert C. *Secure Coding in C and C++*. Addison-Wesley, 2005.

[Sen 2007]

Sen, Robi. *Avoid the dangers of XPath injection.*
http://www.ibm.com/developerworks/xml/library/x-xpathinjection.html (2007).

[Sun 1999a]

Sun Microsystems, Inc. *Why Are Thread.stop, Thread.suspend, Thread.resume and Runtime.runFinalizersOnExit Deprecated?*
http://java.sun.com/j2se/1.4.2/docs/guide/misc/threadPrimitiveDeprecation.html (1999).

[Sun 1999b]

Sun Microsystems, Inc. *The Java Virtual Machine Specification.*
http://java.sun.com/docs/books/jvms/ (1999).

[Sun 1999c]

Sun Microsystems, Inc. *Code Conventions for the Java Programming Language.*
http://java.sun.com/docs/codeconv/ (1999).

[Sun 2000]

Sun Microsystems, Inc. *Java 2 SDK, Standard Edition Documentation.*
http://java.sun.com/j2se/1.3/docs/guide/ (1995-2000).

[Sun 2002]

Sun Microsystems, Inc. *Default Policy Implementation and Policy File Syntax, Document revision 1.6*. http://java.sun.com/javase/6/docs/technotes/guides/security/PolicyFiles.html (2002).

[Sun 2003a]

Sun Microsystems, Inc. *Jar File Specification.* http://java.sun.com/j2se/1.5.0/docs/guide/jar/jar.html (2003).

[Sun 2003b]

Sun Microsystems, Inc. *Sun ONE Application Server 7 Performance Tuning Guide.* http://docs.sun.com/source/817-2180-10/ (2003).

[Sun 2004]

Sun Microsystems, Inc. *Java Platform Debugger Architecture (JPDA).* http://java.sun.com/javase/6/docs/technotes/guides/jpda/index.html (2004).

[Sun 2004b]

Sun Microsystems, Inc. *Generics.* http://java.sun.com/j2se/1.5.0/docs/guide/language/generics.html (2004).

[Sun 2006a]

Sun Microsystems, Inc. *Java Platform, Standard Edition 6 Documentation.* http://java.sun.com/javase/6/docs/index.html (2006).

[Sun 2006b]

Sun Microsystems, Inc. *Java Virtual Machine Tool Interface (JVM TI).* http://java.sun.com/javase/6/docs/technotes/guides/jvmti/index.html (2006).

[Sun 2006c]

Sun Microsystems, Inc. *Java 2 Platform Security Architecture.* http://java.sun.com/javase/6/docs/technotes/guides/security/spec/security-spec.doc.html (2006).

[Sun 2006d]

Sun Microsystems, Inc. *Java Security Guides.* http://java.sun.com/javase/6/docs/technotes/guides/security/ (2006).

[Sun 2006e]

Sun Microsystems. *Supported Encodings.* http://java.sun.com/javase/6/docs/technotes/guides/intl/encoding.doc.html (2006).

[Sun 2006f]

Sun Microsystems, Inc. *Monitoring and Management for the Java Platform.* http://java.sun.com/javase/6/docs/technotes/guides/management/index.html (2006).

[Sun 2006g]

Sun Microsystems, Inc. *Java Platform, Standard Edition 6.* http://java.sun.com/javase/6/docs/technotes/guides/management/toc.html (2006).

[Sun 2008a]

Sun Microsystems, Inc. *The Java Tutorials.* http://java.sun.com/docs/books/tutorial/index.html (2008).

[Sun 2008b]

Sun Microsystems, Inc. *SUN Developer Network*. http://developers.sun.com/ (1994-2008).

[Sun 2008c]

Sun Microsystems, Inc. *JDK 7 Documentation.* http://download.java.net/jdk7/docs/ (2008).

[Sun 2008d]

Sun Microsystems, Inc. *Java Security Architecture.* http://java.sun.com/j2se/1.5.0/docs/guide/security/spec/security-specTOC.fm.html (2008).

[Sun 2008e]

Sun Microsystems, Inc. *Java Plug-in and Applet Architecture.* http://java.sun.com/javase/6/docs/technotes/guides/jweb/applet/applet_execution.html (2008).

[Sun 2009a]

Sun Microsystems, Inc. *Secure Coding Guidelines for the Java Programming Language, Version 3.0.* http://java.sun.com/security/seccodeguide.html (2009).

[Sun 2009b]

Sun Microsystems. *Java Platform, Standard Edition 6 API Specification*. http://java.sun.com/javase/6/docs/api/ Sun Microsystems, Inc. (2009).

[Sun 2010a]

Sun Microsystems, Inc. *Java SE 6 HotSpot Virtual Machine Garbage Collection Tuning,* http://java.sun.com/javase/technologies/hotspot/gc/gc_tuning_6.html (2010).

[Sun 2010b]

Sun Microsystems, Inc. *java - the Java application launcher.* http://java.sun.com/javase/6/docs/technotes/tools/windows/java.html (2006).

[Steel 2005]

Steel, Christopher, Nagappan, Ramesh, & Lai, Ray. *Core Security Patterns: Best Practices and Strategies for J2EE, Web Services, and Identity Management*. Prentice Hall, 2005.

[Steuck 2002]

Steuck, Gregory. *XXE (Xml eXternal Entity) attack.* http://www.securityfocus.com/archive/1/297714 (2002).

[Sutherland 2010]

Sutherland, Dean F. & Scherlis, William L. "Composable thread coloring," *Proceedings of the 15th ACM SIGPLAN symposium on Principles and practice of parallel programming.* Bangalore, India, 2010, ACM, 2010.

[Tanenbaum 2002]

Tanenbaum, Andrew S. & Van Steen, Maarten. *Distributed Systems: Principles and Paradigms, 2/E*. Prentice Hall, 2002.

[Venners 1997]

Venners, Bill. *Security and the Class Loader Architecture.* http://www.javaworld.com/javaworld/jw-09-1997/jw-09-hood.html?page=1 (1997).

[Venners 2003]

Venners, Bill. *Failure and Exceptions, A Conversation with James Gosling, Part II*. http://www.artima.com/intv/solid.html (2003).

[Wheeler 2003]

Wheeler, David A. *Secure Programming for Linux and Unix HOWTO*. http://www.dwheeler.com/secure-programs/Secure-Programs-HOWTO/index.html (2003).

翻訳参考図書

ゲーツ他『Java 並行処理プログラミング』岩谷宏訳，ソフトバンククリエイティブ, 2006

ゴスリン他『Java 言語仕様第 3 版』村上雅章訳，ピアソンエデュケーション, 2006

リンドホルム他『Java 仮想マシン仕様』村上雅章訳，ピアソンエデュケーション, 2001