Volatile variables

- JVMには揮発性変数（Volatile variables）というものがある。
- Syncronizedに比べると軽い同期化の仕組み
- ステータスフラグに利用される：計算の完了・キャンセルを表すフラグ等
- 二つの利点
  - 変数への書き込み、読み出し操作の順序替え（reorder)がおこらない
  - 書き込みが全てのスレッドで即時に可視化される

- volatile example

```
class Page(val txt: String, var position: Int)
object Volatile extends App {
val pages = for (i <- 1 to 5) yield
  new Page("Na" * (100 - 20 * i) + " Batman!", -1)
@volatile var found = false
for (p <- pages) yield thread {
  var i = 0
  while (i < p.txt.length() && !found)
    if (p.txt(i) == '!') {
      p.position = i
      found = true
    } else i += 1
}
while(!found){}
log(s"results: ${pages.map(_.position)}")
}
```

  - あるスレッドtがfoundにtrueをセット、その後にメインスレッドがfoundを読むと、スレッドtがfoundにセットする前に可視だった値はメインスレッドにとっても可視となる　→　positionも可視

- ThreadSharedStateAccessReordering Exampleは全ての変数をvolatileにすることで修正できる
- Javaと違い、Scalaではローカル変数にvolatileを宣言できる
  - volatile変数も保持したオブジェクトがvolatile変数毎に作成される
  - 「オブジェクトへ”lift”された」

- volatileだけではアトミック性を保障できない
  - vogetUniqueIdの例はvolatileだけでは正しく実装できない

The Java Memory Model

- 明示してこなかったがこの章を通して大部分JMMについて示してきた
- 言語メモリモデルは変数への書き込みが他のスレッドでどのように可視となるかを記述した仕様
- メモリモデルを考える際に、逐次的一貫性(sequential consistency)と呼ばれるメモリモデルを想定するかもしれない
    - プロセッサーが変数vに書き込むと関連するメモリ領域が即時に変更され、他のプロセッサーがvの値を瞬時に見ることができる

- ThreadSharedStateAccessReordering Example で既にみたように、実際のモデルは逐次的一貫性モデルとほとんど関連しない
    - プロセッサはキャッシュ階層を持ち、メインメモリにいずれ書き込まれることしか保障しない
    - コンパイラはレジスタを利用してメモリ書き込みを避けることが許容され、（シングルスレッド内で）直列的なセマンティクスを維持する限り、最適化のため順序替え(reorder)を行う

- メモリモデルは並行プログラムの予測可能な振る舞いとコンパイラの最適化能力のトレードオフである
- 変数の書き換えが許されていない純粋な関数型言語ではメモリモデルは必要ない
- ScalaはJVMのメモリモデル(JMM)を継承している
    - JMMはプログラムの異なるアクション間での事前発生(happens-before)の関係を定義している

- JMMのアクション
    - 変数のリード/ライト
    - モニタのロック/アンロック
    - スレッドのスタート/ジョイン

- 事前発生のルール
    - Program order：スレッド内の各アクションは、そのスレッド内で後からくるアクションよりも事前発生する
    - Monitor locking：モニタのアンロックは、その後の同じモニタのロックよりも事前発生する
    - Volatile fields：揮発性フィールドへの書き込みは、同じフィールドのその後の読み出しよりも事前発生する
    - Thread start：スレッドのstart()呼出しは、開始したスレッドのすべてのアクションよりも事前発生する
    - Thread termination：スレッド内のどのアクションも、他のスレッドによるそ

のスレッドの終了の検出よりも事前発生する
    - Transitivity：AがBよりも事前発生し、BがCよりも事前発生するなら、AはCよりも事前発生する

- 事前発生のルールは異なるスレッドによる書き込みの可視性を記述する *
    - プログラム順序で揮発性リードの前の非揮発性リードはreorderされない
    - プログラム順序で揮発性ライトの前の非揮発性ライトはreorderされない

Immutable objects and final fields

- finalフィールドのみのオブジェクトで、コンストラクタの完了前に別のスレッドから可視でない場合、そのオブジェクトはimmutableとみなされ同期化の必要がない
- Scalaではフィールドのfinalは、サブクラスでgetterメソッドをoverrideできないことを表す
    - フィールド自体はvalで宣言すると常にfinalになる
    - Example

```
public class Foo {
  //class Foo(final val a: Int, val b: Int)の変換イメージ
  final private int a$;
  final private int b$;
  final public int a() {return a$;}
  public int b() {return b$;}
  public Foo(int a, int b) {
    a$ = a;
    b$ = b;
  }
}
```

- Scalaは関数型とオブジェクト指向のハイブリッドであるので、言語の特徴の多くはimmutable objectsにマップする
- ローカル変数numberはラムダによって更新されるためliftする必要がある

    - Example

```
object LambdaExample extends App {
  var inc: () => Unit = null
  val t = thread {
    if (inc != null) inc()
  }
  private var number = 1
  inc = () => {number += 1}
}
```

- 無名クラスfunction()で書き直した場合

```
number = new IntRef(1)
inc = new Function() {
  val $number = number
  def apply() = $number.elem += 1
}
```

- incのアサインと、スレッドtによるreadに事前発生の関係はない
- しかしながら、スレッドtでincがnullではない場合、incの実行は正しく動作する
  - $numberがimmutableのラムダオブジェクト項目として適切に初期化されるため
  - Scalaコンパイラはラムダの値が、finalで適切に初期化されていることを保障する
- 現行バージョンのScalaにおいては、ListやVectorなどのCollectionは同期化無しで共有してはいけない
  - Listの中身はfinalではないため
- たとえオブジェクトがimmutableに見えても、スレッド間で共有する場合は常に適切に同期化する