マルチスレッド
ハンズオン

第１回　スレッドセーフとロック

アジェンダ

- スレッドとは
- スレッドとプロセスの違い
- スレッドの利点
- スレッドのリスク

スレッドとは

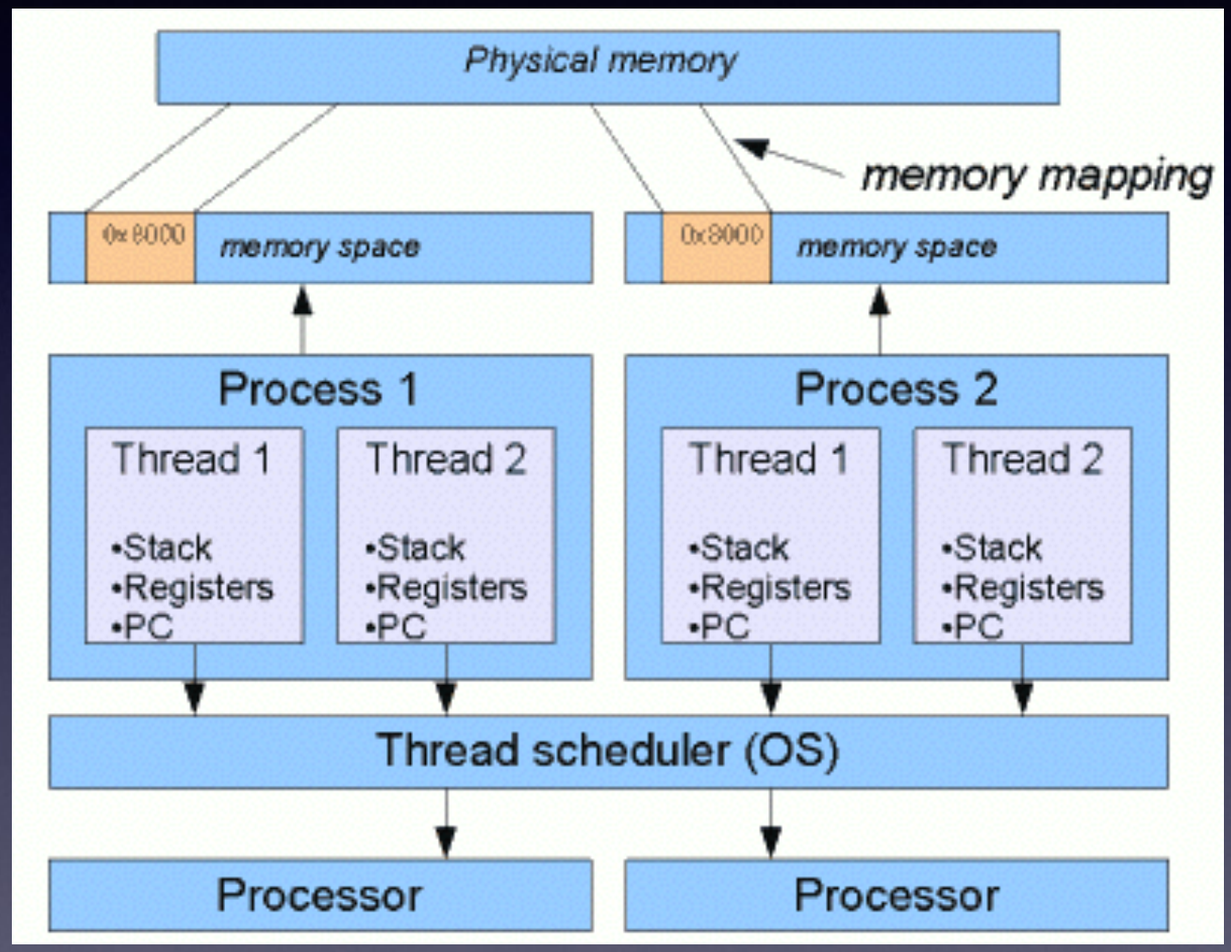
スレッドとは

- 並列処理に対応したOS上でのプログラムの最小の実行単位

スレッドとプロセスの違い

スレッドとプロセスの違い

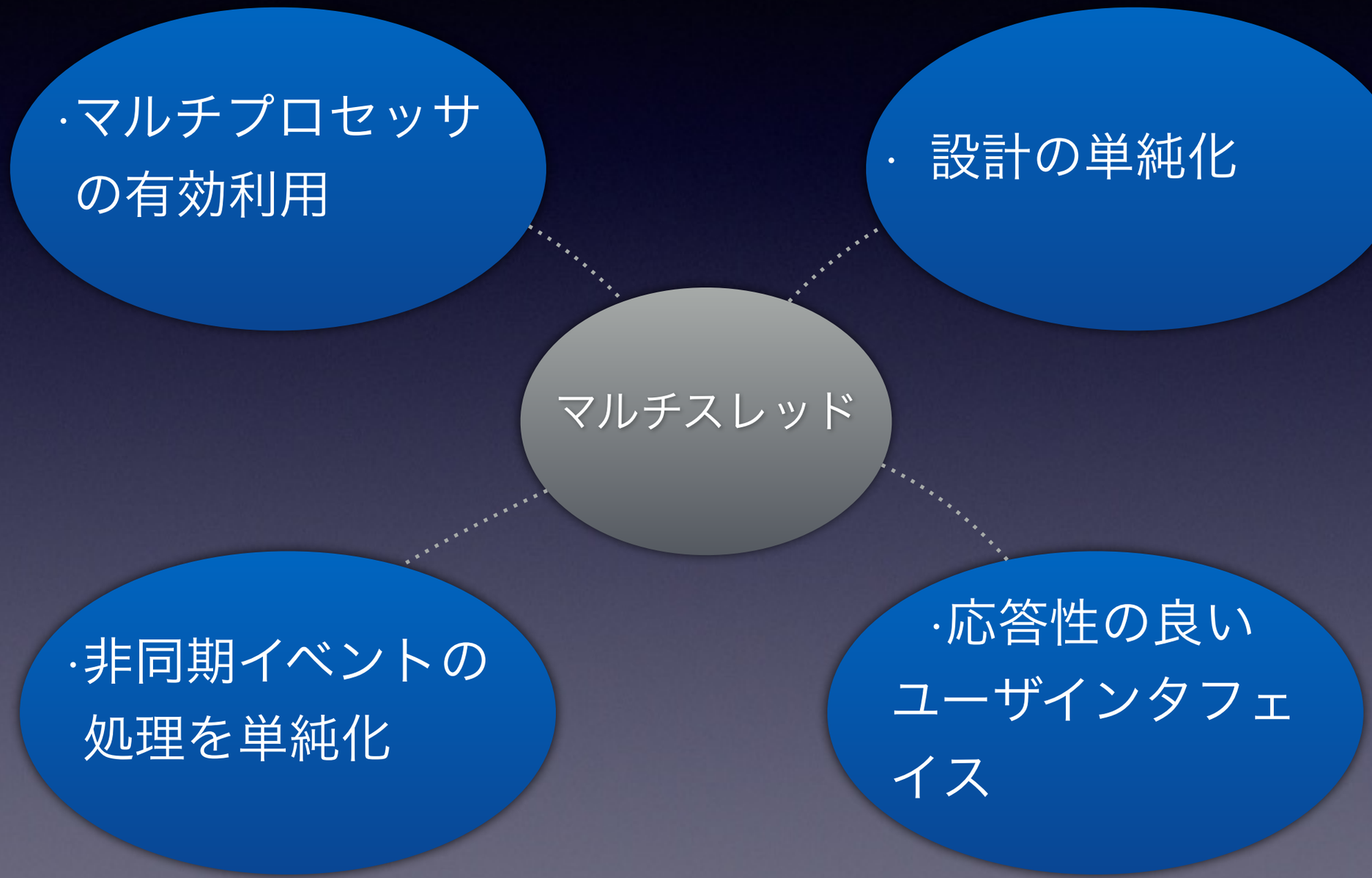
- プロセスは、それぞれCPUやメモリ空間などが割り当てる
- プロセス間でのやり取りは、ファイル、ソケット、メッセージングなどのプロセス間通信が必要
- スレッド間でのやり取りは、メモリの読み書きで可能



- 図：http://www.javamex.com/tutorials/threads/how_threads_work.shtml

スレッドの利点

スレッドの利点


·マルチプロセッサの有効利用
· 設計の単純化
マルチスレッド
·非同期イベントの処理を単純化
·応答性の良いユーザインタフェイス

スレッドのリスク

スレッドセーフでない順序生成クラス

- List01_1.java

```
    private static class UnsafeSequence {
        private int nextValue = 1;

        /* 重複のない数を返す */
        public int getNext() {
            return nextValue++;
        }
    }
```

i++

i=10

10+1

i=11

j=11

11+1

j=12

i=10

10+1

i=11

i=10

10+1

i=11

安全性の危機

・仕様通りに動作するかどうかが、実行時におけるスレッドの実行順やタイミングに依存してしまうこと。
→競り合い状態(race condition)

スレッドセーフな順序生成クラス

- List01_2.java

```
    private static class UnsafeSequence {
        private int nextValue = 1;

        /* 重複のない数を返す */
        public synchronized int getNext() {
            return nextValue++;
        }
    }
```

i++

i=10

10+1

i=11

j=11

11+1

j=12

i=10

10+1

i=11

i=10

10+1

i=11

synchronizedメソッド

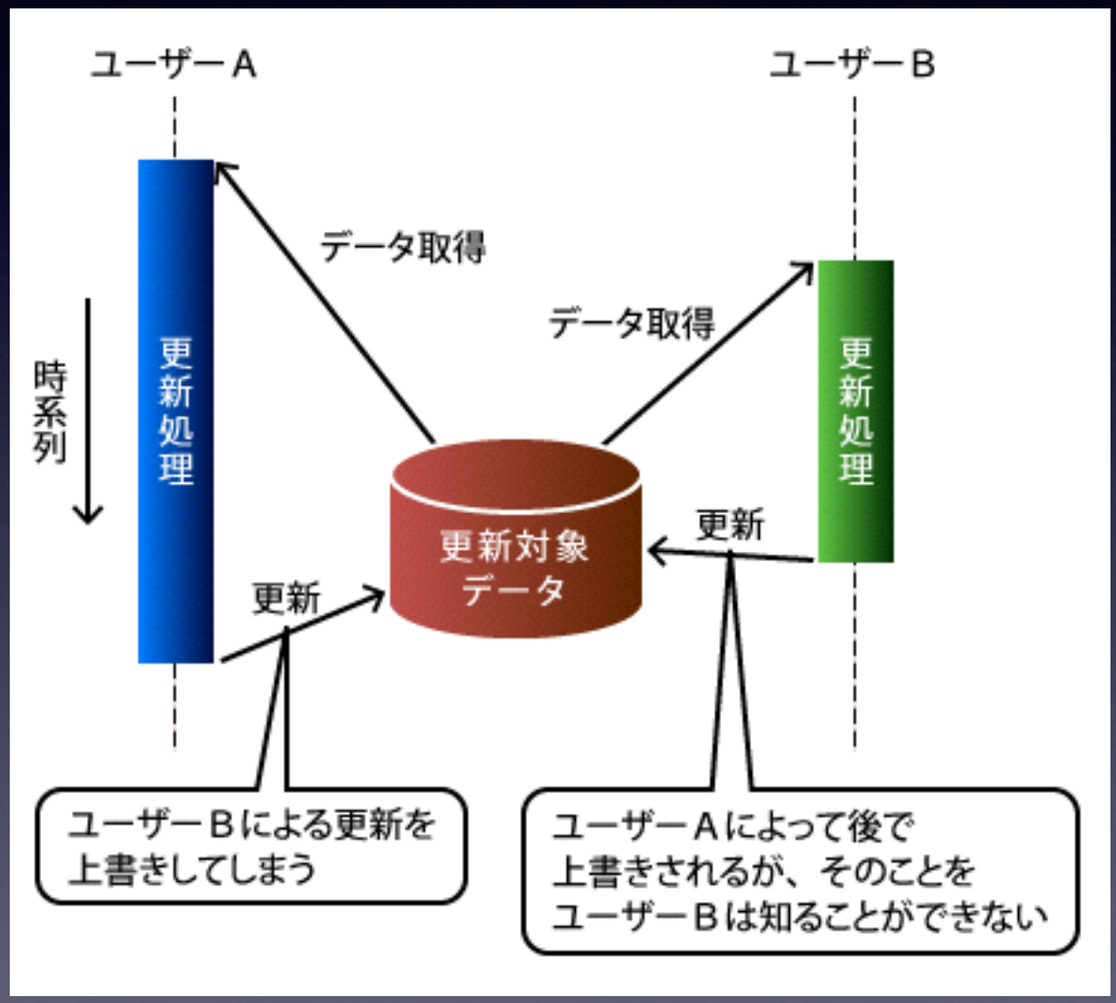
- synchronizedメソッドは，実行前にロックを取得する。クラス (static) メソッドに対しては，そのメソッドのクラスに対するClassオブジェクトと関連したロックを使用する。インスタンスメソッドに対しては，this(そのメソッドが呼び出されたオブジェクト)と関連したロックを使用する。

スレッドセーフとデータベース

排他制御

- すでに登録されている予約情報の時間を2人がほぼ同時に更新したとします。このとき、先に変更を行ったユーザーは、他のユーザーがほぼ同時に同じ予約を変更したとは思っていません。もちろんエラーも発生しないため、自分の行った変更が正しく反映されたと思うでしょう。

しかし実際は、後から変更したユーザーの入力内容で予約情報が上書きされてしまっています。



- 図：http://codezine.jp/article/detail/6764

スレッドセーフでない遅延初期化

- List02_1.java

```
private static class LazyInitRace {
    private ExpensiveObject instance = null;

    public ExpensiveObject getInstance() {
        if(instance == null)
            instance = new ExpensiveObject();
        return instance;
    }
}
```

遅延初期化

instance=
null

new
Object

instance=
obj

instance=
obj

instance=
null

new
Object

instance=
obj

instance=
null

new
Object

instance=
obj2

スレッドセーフな遅延初期化

- List02_2.java

```
private static class LazyInitRace {
    private ExpensiveObject instance = null;

    public ExpensiveObject getInstance() {
        synchronized(this) {
            if(instance == null)
                instance = new ExpensiveObject();
            return instance;
        }
    }
}
```

遅延初期化

instance=
null

new
Object

instance=
obj

instance=
obj

instance=
null

new
Object

instance=
obj

instance=
null

new
Object

instance=
obj2

synchronized文

- synchronized文は，実行中のスレッドに代わって相互排他ロックを獲得して，ロックを実行し，ロックを解除する。実行中のスレッドがロックを所有している間，他のいかなるスレッドもロックを獲得できない。

スレッドセーフでない可変な整数クラス

- List03_1.java

```
    private static class MutableInteger {
        private int value;

        public int get() {
            return value;
        }

        public void set(int value) {
            this.value = value;
        }
    }
```

同期化されていない getter/setter

set(100)

get

set(100)

get

スレッドセーフな可変な整数クラス

- List03_2.java

```
    private static class MutableInteger {
        private int value;

        public synchronized int get() {
            return value;
        }

        public synchronized void set(int value) {
            this.value = value;
        }
    }
```

同期化された getter/setter

set(100)

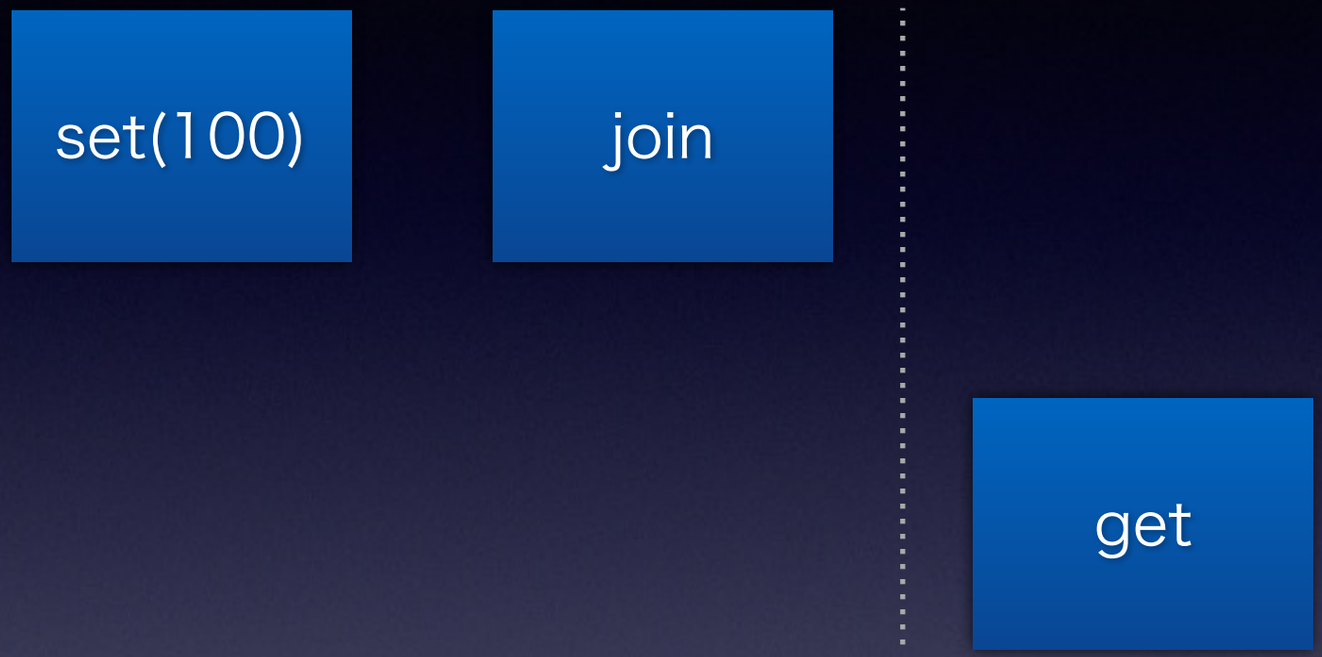
get

set(100)

get

- 同期化しても、スレッドの実行される順番を制御できるわけではない。

Thread#join



- スレッド1が完了するのを待って、スレッド2の実行を開始する
- 順序保証できる一方、直列化する（並列実行できない）ので、性能向上できない。

プット・イフ・アブセントを実装する（スレッドセーフでない）

- List04_1.java

```
private static class ListHelper<E> {
    public List<E> list = Collections.synchronizedList(new ArrayList<E>());

    public synchronized boolean putIfAbsent(E x) {
        boolean absent = !list.contains(x);
        if(absent) {
            list.add(x);
        }
        return absent;
    }
}
```

プット・イフ・アブセント

contains?

add

contains?

contains?

add

contains?

add

プット・イフ・アブセントを実装する（スレッドセーフ？）

- List04_2.java

```
private static class ListHelper<E> {
    public List<E> list = Collections.synchronizedList(new ArrayList<E>());

    public boolean putIfAbsent(E x) {
        synchronized(list) {
            boolean absent = !list.contains(x);
            if(absent) {
                list.add(x);
            }
            return absent;
        }
    }
}
```

プット・イフ・アブセント

contains?

add

contains?

contains?

add

contains?

add

プット・イフ・アブセントを実装する（スレッドセーフ）

- List04_3.java

```
    private static class ListHelper<E> {
        private List<E> list = Collections.synchronizedList(new
ArrayList<E>());

        public boolean putIfAbsent(E x) {
            synchronized(list) {
                boolean absent = !list.contains(x);
                if(absent) {
                    list.add(x);
                }
                return absent;
            }
        }
    }
```

- オブジェクトの公開範囲には要注意

チェック・ゼン・アクトを実装する（スレッドセーフでない）

- List05_1.java

```
    private static class VectorHelper {
        public static <E> E getLast(Vector<E> list) {
            int lastIndex = list.size() - 1;
            return list.get(lastIndex);
        }

        public static <E> void deleteLast(Vector<E> list) {
            int lastIndex = list.size() - 1;
            list.remove(lastIndex);
        }
    }
```

チェック・ゼン・アクト

list.size

lastIndex
=size - 1

list.remov
e(lastInde
x)

list.size

lastIndex
=size - 1

list.get(la
stIndex)

list.size

lastIndex
=size - 1

list.remov
e(lastInde
x)

list.size

lastIndex
=size - 1

list.get(la
stIndex)

チェック・ゼン・アクトを実装する（スレッドセーフ）

- List05_2.java

```
private static class VectorHelper {
    public static <E> E getLast(Vector<E> list) {
        synchronized(list) {
            int lastIndex = list.size() - 1;
            return list.get(lastIndex);
        }
    }

    public static <E> void deleteLast(Vector<E> list) {
        synchronized(list) {
            int lastIndex = list.size() - 1;
            list.remove(lastIndex);
        }
    }
}
```

チェック・ゼン・アクト

list.size

lastIndex
=size - 1

list.remov
e(lastInde
x)

list.size

lastIndex
=size - 1

list.get(la
stIndex)

list.size

lastIndex
=size - 1

list.remo
ve(lastIn

list.size

lastIndex
=size - 1

list.get(la
stIndex)

まとめ

- ある変数に複数のスレッドがアクセスし、どれかのスレッドが変数への書き込み（値を変える操作）をする可能性があるなら、すべてのスレッドに対して、その変数へのアクセスを必ず同期化しなければなりません
- 複数のスレッドが同じ変数に正しい同期化なしでアクセスしているなら、そのプログラムは欠陥プログラムです。なおす方法は三つあります。
  - その変数を複数のスレッドが共有しないようにする
  - その変数を不可変にする
  - その変数へのアクセスをつねに同期化する

リソース

- ハンズオンソースコード
  https://github.com/taka2/MultiThreadHandsOn

- Java並行処理プログラミング ―その「基盤」と「最新API」を究める―
  ISBN: 978-4797337204

宿題

- 複数スレッドからjava.lang.StringBuilderのインスタンスにアクセスし、データが壊れるサンプルを作成せよ。また、このサンプルをスレッドセーフになるように改良せよ。

- java.text.SimpleDateFormatがスレッドセーフでない理由を、JDKのソースコードを元に説明せよ。

Object#wait/notifyとThread#sleepの違い

- java - Thread.sleep and object.wait - Stack Overflow http://stackoverflow.com/questions/10693361/thread-sleep-and-object-wait

- One is used to synchronize Threads together while the other one is used to sleep for a given amount of time.
  If you want to synchronize Threads together, user wait/notify. If you want to sleep for a known amount of time, use Thread.sleep.

内部クラスから参照するローカル変数がfinalでなければならない理由

- [S016 A-14]
  http://homepage1.nifty.com/docs/java/faq/S016.html#S016-14

- ローカルな内部クラスのインスタンスのライフタイム(生成されてからGCされるまでの期間)は、それを宣言するブロックやメソッドのライフタイムより長いことがあり、その場合、内部クラスから参照しているローカル変数や引数が実際に参照を行なうタイミングまで存在し続けていないことがあります。そういったケースでも正しく値が参照できることを保証するため、参照可能なものを内部クラスが生成される時点以降に値が変更されないもの、すなわち、final宣言されているローカル変数や引数のみに制限し、その値をコピーしてインスタンス内部に保持するようになっています。