# プログラミングA 第9回・宿題　回答



## 1　宿題 1

- 問題 A: 両スレッドは Scanner オブジェクトのフィールド data を介し存しているが、その参照・書き込み処理は排他制御されていない。従って、スレッド (特に Scanner) における処理において期待通りの結を得られない可能性がある。具体的には Scanner.run() 内で、data に値を代入した上で標準出力しようとしているが、代入の直後のタイミングで実行スレッドが Fax に切り替わると、Fax.run() 内で data に n が代入されてしまい、結果として Scanner.run() 内では標準出力に null が出力されてしまう。
  ⇒ data の読み書き処理について排他制御されるようにすることで解決できる。例えば、読み書き用の synchronized メソッドを作り用いる
- 問題 B:両スレッドは run() 内で、条件が満たされることを while(true) の無限ループでひたすら待ち続ける。これは単純なループ処理であるため、実行権が当該スレッドにある限りは、条件が満たされていなくとも繰り返され、時間効率的に無駄である。
  ⇒スレッドの wait()〜notify() を用いて、条件が満たされない場合に一明示的に待機させて、条件が満たされるようになったときに起こして実行させることにより解決できる。

参考資料：Scanner.java

---

```
public class Scanner extends Thread {
  String data = null;
  public synchronized void fax() throws InterruptedException {
    while(data == null) {
      wait();
    }
    System.out.println("fax: " + data);
    data = null;
    notifyAll();
  }
  public synchronized void scan() throws InterruptedException {
    while(data != null) {
      wait();
    }
    data = " 値が設定されています ";
    System.out.println("scanner: " + data);
    notifyAll();
  }
  public void run() {
    try {
      while(true) {
        scan();
      }
    } catch (InterruptedException e) {
    }
```

```
  }
  public static void main(String[] args) {
    Scanner scanner = new Scanner();
    Fax fax = new Fax(scanner);
    scanner.start();
    fax.start();
  }
}

class Fax extends Thread {
  Scanner scanner = null;

  public Fax(Scanner s) {
    super();
    scanner = s;
  }

  public void run() {
    while(true) {
      if(scanner.data != null) {
        System.out.println("fax : " + scanner.data);
        scanner.data = null;
      }
    }
  }
}
```

参考資料：Fax.java

```
class Fax extends Thread {
  Scanner scanner = null;
  public Fax(Scanner s) {
    super();
    scanner = s;
  }
  public void run() {
    try {
      while(true) {
        scanner.fax();
      }
    } catch(InterruptedException ie) {
    }
  }
}
```

## 2 宿題 2-1

参考資料：PersonClient.java

```
public class PersonClient {
   public static void main(String[] args) {
      Person p = new Person("Tokyo");
      System.out.println(p);
      StringBuffer text = p.getAddress().getText();
      text.replace(0, text.length(), "Paris");
      System.out.println(p);
   }
}
```

## 3 宿題 2-2

無論、text フィールドを String 型に変えても構わない。

参考資料：Address.java

```
public class Address {
   private final StringBuffer text;
   public Address(String s) {
      text = new StringBuffer(s);
   }
   public String getText() {
      return text.toString();
   }
   /*
   public StringBuffer getText() {
      return text;
   }
   */
}
```

## 4 宿題 3

(1)
- ReaderThread は、無限ループの中で DataBuffer からデータを読み込み (read)、" Thread-0 reads 00" のように読み込んだデータを標準出力に出力することを繰り返す。
  - DataBuffer の read メソッドが呼ばれると、
    - DataBuffer はデータを読む前に必ず ReadWriteLock の readLock メソッドを呼び出し、その呼び出しが完了することで読むためのロックを獲得したことになる。
      - readLock メソッドは、スレッドが実際のデータ読込みを開始する前に呼ばれる。ここで、書き込み中のスレッドがいるか (writingWriters $>$ 0)、もしくは、書き込みが優先

される状況で (preferWriter) 書き込みを待っているスレッドがいる場合は、同メソッドを呼び出したスレッドはいったん待機する。ガード条件が満たされるようになったら、読込み中スレッドの数を一つ増やして ReadWriteLock インスタンスのロックを解放してメソッドを抜ける。

* DataBuffer は読むためのロック獲得後に実際にデータを読み込み (value.toString)、読んだデータを返す。
* DataBuffer は ReadWriteLock の readUnLock メソッドを呼び出して読むためのロックを解放する。
  - readUnLock メソッドは、スレッドがデータ読込みを終えた後で呼ばれる。ここで、読込み中スレッドの数を一つ減らし、以降において書き込みが優先されるようにして (preferWriter = true)、ReadWriteLock インスタンス上で待機している他のスレッド群を起こしてから、同インスタンスのロックを解放してメソッドを抜ける。

- WriterThread は、0 から 9 までの数字を一つずつ、DataBuffer に追加し (append)、”Thread-4 appends 0” のように書き込んだ数字を標準出力に出力する。
  - DataBuffer の append メソッドが呼ばれると、
    * データを書く前に必ず ReadWriteLock の writeLock メソッドを呼び出し、その呼び出しが完了することで書くためのロックを獲得したことになる。
      - writeLock メソッドは、スレッドが実際のデータ書き込みを開始する前に呼ばれる。最初に書き込み待ち中のスレッド数を一つ増やす。続いて、読込み中のスレッドがいるか (readingReaders $> 0$)、もしくは、書き込み中のスレッドがいる (writingWriters $> 0$) 場合は、同メソッドを呼び出したスレッドはいったん待機する。ガード条件が満たされるようになったら、書き込み待ち中のスレッド数を一つ減らし、書込み中スレッドの数を一つ増やして ReadWriteLock インスタンスのロックを解放してメソッドを抜ける。
    * 獲得後に実際にデータを書き込む (value.append)。
    * ReadWriteLock の writeUnLock メソッドを呼び出して書くためのロックを解放してメソッドを抜ける。
      - writeUnLock メソッドは、スレッドがデータ書込みを終えた後で呼ばれる。ここで、書込み中スレッドの数を一つ減らし、以降において書き込みが優先されないようにして (preferWriter = false)、ReadWriteLock インスタンス上で待機している他のスレッド群を起こしてから、同インスタンスのロックを解放してメソッドを抜ける。

(2)
- 実行結果: ReadWriteLock を用いると ReaderThread が多数読みこみながらもしばしば WriterThread が書きこむ。しかし ReadWriteLock2 を用いると、ほぼ ReaderThread の読込みのみとなり、二つの WriterThread による書き込みは最初の 1 回ずつの書き込みを除いてなされない。
- 理由:
  - ある ReaderThread インスタンスが ReadWriteLock の読むためのロックを獲得して読込み処理を行っている間、他の ReaderThread インスタンスも排他制御されずに読むためのロックを獲得して読込み処理を行える。一方、その間、WriterThread インスタンスは ReadWriteLock の書くためのロックを獲得できず待たされる。
  - WriterThread インスタンスが二つに対して、ReaderThread インスタンスは四つと多く生成

され実行される。
  - さらに悪いことに、WriterThread は ReaderThread とは異なり、一度データを書き込むと自身で一時停止する (Thread.sleep)。
  - 従って、四つの ReaderThread は次々に並行して読込み処理を進めていき、途切れることがない。この間、二つの WriterThread は最初に書き込みを行って以降、一時停止を抜けると既に多くの ReaderThread が並行して読込み処理を進めており、いつまでたっても次の書き込みを行えない。

## 5 宿題 4

参考資料：MessageHost.java

```
public class MessageHost {

  private final MessageHelper helper = new MessageHelper();

  public void request(final int number) {
    new Thread() {
      public void run() {
        for(int i = 1; i <= 10; i++) {
          helper.handle(number + i);
        }
      }
    }.start();
  }

  public static void main(String[] args) {
    MessageHost host = new MessageHost();
    host.request(10);
    host.request(100);
    host.request(1000);
  }
}
```

## 6 宿題 5

- 構造
  - Data: データ取得のインタフェースである。
  - RealData: 実際のデータを表す。
    - コンストラクタによるインスタンス生成には時間を要する。
    - Data インタフェースを実装し、外部からのデータ取得を可能とする。
  - FutureData: 実際のデータのいわば「引換券」である。
    - Data インタフェースを実装し、外部からのデータ取得を可能とする。

    - データ取得メソッド getContent が呼ばれると、RealData が未設定であれば、Guarded Suspension パターンに従って呼び出したスレッドは当該 FutureData インスタンスのウェイトセットで待機し、起こされてから RealData が設定されていれば RealData から実際のデータを取得してそれを呼び出し元に返す。
    - RealData の設定メソッド setRealData が呼ばれると、設定済みであれば直ちに終了し (Balking パターン)、未設定であれば新たに RealData を設定し、当該 FutureData インスタンスのウェイトセットで待機していた全てのスレッドを起こして終わる。
  - Host: データのリクエストを受け付けてデータを返す。
    - request メソッドが呼ばれると、無名インナークラスとしてスレッドのサブクラスを定義してインスタンスを生成し (Thread-Per-Message パターン)、そのスレッドに RealData の生成、および、生成次第、FutureData に設定する。この処理には時間を要するため、呼び出し元には先行して FutureData インスタンスを返しておく。
- 振る舞い
  - Main の main メソッドを実行すると、main スレッドは Host のインスタンスを生成し、同インスタンスに 3 つのデータ取得リクエストを発行して (request メソッドの呼び出し)、Data 型で戻り値を直ちに受け取る。ここで、受け取った Data 型の実体は FutureData インスタンスであるが、Main からはそれを意識する必要がない。その後、3 秒待機してから、Data 型の何らかのインスタンス群 (実際には FutureData) からデータを取得し、以下のように標準出力に出力する。ただし取得は、実際のデータが生成されるまで待たされる。
    - aaaaaaaaaa
    - aaaaaaaaaaaaaaaaaaaa
    - aaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
  - Host は Main からリクエストを受けて、RealData のインスタンス生成を開始するがこれには時間を要する。そこで、新スレッドを生成して RealData インスタンスの生成を委ね、呼び出し元には先行して FutureData のインスタンスを返す。
  - RealData のインスタンスは生成されると、以下のようにその旨をデータと共に標準出力に出力する。生成にはそれぞれ 1 秒、2 秒、3 秒かかる。
    - RealData has been made: aaaaaaaaaa
    - RealData has been made: aaaaaaaaaaaaaaaaaaaa
    - RealData has been made: aaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
  - 結果として標準出力には以下を得る。

```
RealData has been made: aaaaaaaaaa
RealData has been made: aaaaaaaaaaaaaaaaaaaa
aaaaaaaaaa
aaaaaaaaaaaaaaaaaaaa
RealData has been made: aaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
```