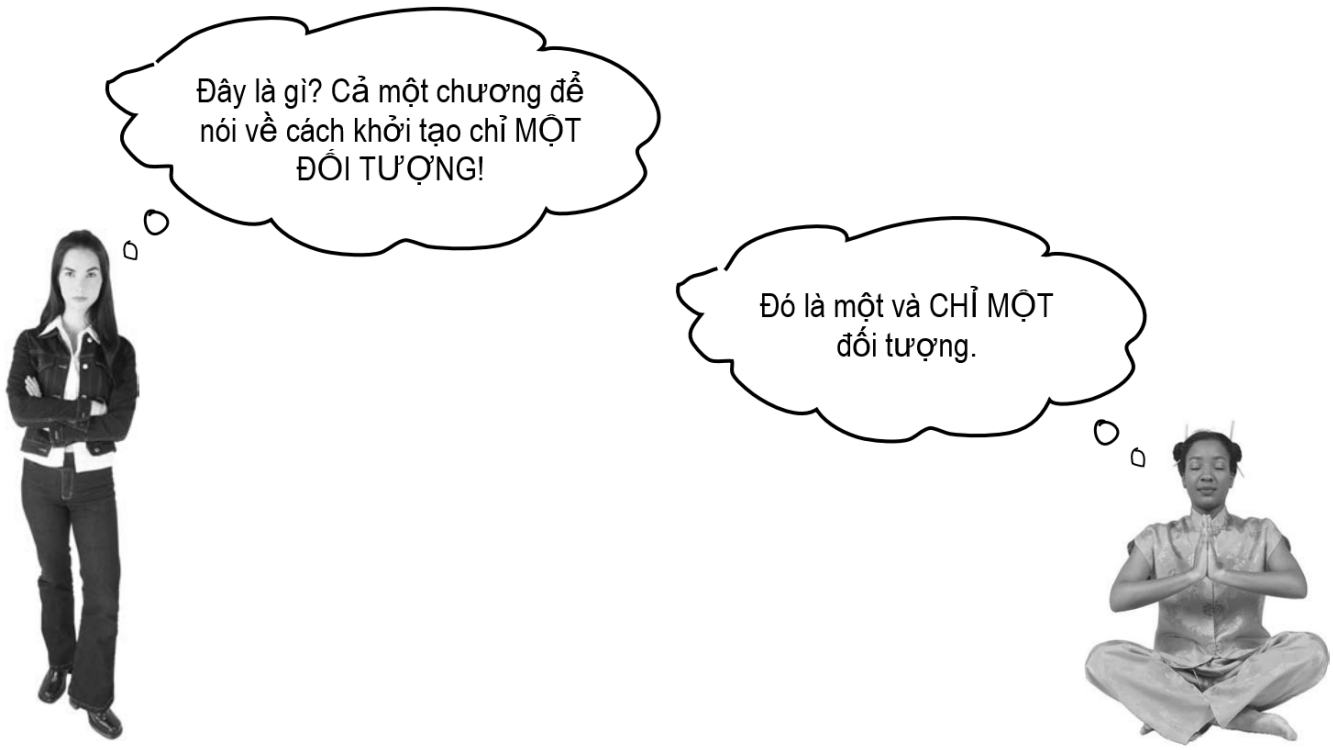
Chương 5: Singleton Pattern – Chỉ 1 cho mỗi loại

Singleton Pattern – Chỉ 1 cho mỗi loại object

Điểm dừng chân tiếp theo của chúng tôi chính là Singleton Pattern, để tạo ra một đối tượng độc nhất (Single) cho những thứ mà chỉ cần duy nhất một instance. Bạn có thể vui mừng khi biết rằng trong tất cả các mẫu, Singleton Pattern có sơ đồ lớp đơn giản nhất; trong thực tế, Singleton chỉ nằm trong một lớp duy nhất! Nhưng đừng quá thoải mái; mặc dù Singleton đơn giản trong thiết kế lớp, chúng ta vẫn sẽ gặp một vài vấn đề khi áp dụng nó.



Developer: Công dụng của nó là gì vậy ông chuyên gia?

Chuyên gia: Có nhiều đối tượng chúng ta chỉ cần một trong số chúng: ví dụ như thread pools này, caches này, dialog boxes, đối tượng xử lý preferences và cài đặt registry, đối tượng được sử dụng để ghi log và các đối tượng đóng vai trò là trình điều khiển thiết bị cho các thiết bị như máy in và card đồ họa…nhiều lắm. Trong thực tế, đối với nhiều loại đối tượng này, nếu chúng ta khởi tạo nhiều hơn một đối tượng chúng ta sẽ gặp phải tất cả các vấn đề như hành vi chương trình không chính xác, lạm dụng tài nguyên hoặc kết quả không nhất quán.

Developer: Được rồi, vì vậy có thể có các lớp chỉ nên được khởi tạo một lần, nhưng tôi có cần cả một chương cho việc này không? Tôi không thể làm điều này bởi convention hoặc biến toàn cục (global variable) sao? Ông biết đấy, như trong Java, tôi có thể làm điều đó chỉ với một biến static.

Chuyên gia: Theo nhiều cách, Singleton Pattern là một quy ước để đảm bảo một và chỉ một đối tượng được khởi tạo cho một lớp nhất định. Nếu chú em đã có một cái tốt hơn, mọi người sẽ muốn nghe về nó; nhưng hãy nhớ rằng, giống như tất cả các mẫu khác, Singleton Pattern là phương pháp được thử nghiệm nhiều lần bởi nhiều developer khác để đảm bảo chỉ có một đối tượng được tạo. Mẫu Singleton cũng cung cấp cho chúng ta một nơi truy cập toàn cục, giống như một biến toàn cục, nhưng không có nhược điểm.

Developer: Nhược điểm gì?

Chuyên gia: Vâng, đây là một ví dụ: nếu chú em gán một đối tượng cho một biến toàn cục, thì đối tượng đó có thể được tạo khi ứng dụng của chú em vừa bắt đầu. Đúng chứ? Điều gì xảy ra nếu đối tượng này sử dụng nhiều tài nguyên và ứng dụng của bạn không bao giờ kết thúc việc sử dụng nó? Như chú em thấy, với Singleton Pattern, chúng ta chỉ có thể tạo các đối tượng của mình khi cần.

Developer: Điều này dường như vẫn chưa thực sự quá hữu ích.

Chuyên gia: Nếu chú em đã xử lý tốt các biến và phương thức static cũng như các access modifiers (public, protected, private), thì nó không thành vấn đề. Nhưng, trong cả hai trường hợp, thật thú vị khi xem cách Singleton Pattern hoạt động, và code Singleton không khó để áp dụng. Chỉ cần tự hỏi: làm thế nào để tôi ngăn chặn nhiều hơn một đối tượng được khởi tạo?

The Little Singleton Pattern

Làm thế nào để tạo ra một đối tượng?

```
1 new MyObject();
```

Và, nếu một đối tượng khác muốn tạo **MyObject** thì sao? Nó có thể gọi new trên **MyObject** một lần nữa không?

- Vâng, đương nhiên.

Vì vậy, miễn là chúng ta có một lớp, chúng ta có thể khởi tạo nó một hoặc nhiều lần đúng không?

- Vâng. Nhưng chỉ khi nó là một public class.

Và nếu không?

- Chà, nếu nó không phải là một public class, chỉ các lớp trong cùng một package có thể khởi tạo nó. Nhưng chúng vẫn có thể khởi tạo nó nhiều hơn một lần.

Hừm, thú vị. Bạn có biết bạn có thể làm điều này?

```
1 public MyClass {
2     private MyClass() {}
3 }
```

- Không, tôi không bao giờ nghĩ về nó, nhưng tôi đoán chúng vẫn hợp lý bởi vì nó là một cú pháp đúng.

Nó có nghĩa là gì?

- Tôi cho rằng đó là một lớp không thể được khởi tạo bởi vì nó có một private constructor.

Chà, có **BẤT KỲ** object nào có thể sử dụng private constructor không?

- Hmm, tôi nghĩ bên trong lớp MyClass là nơi duy nhất có thể gọi nó. Nhưng điều đó không có nhiều ý nghĩa.

Tại sao không ?

- Bên trong lớp MyClass sẽ làm được điều này:
  public MyClass getMyClass() { return new MyClass(); }
  Nhưng cho dù có phương thức getMyClass() đi nữa cũng không thể đem ra khỏi lớp này được, bởi vì muốn sử dụng thì phải tạo instance mới và gọi instance.getMyClass(), nhưng không thể tạo được biến instance). Đây là một

vấn đề về con gà và quả trứng: Tôi có thể sử dụng constructor từ một đối tượng thuộc loại MyClass, nhưng tôi không bao giờ có thể khởi tạo đối tượng đó vì không đối tượng nào khác có thể sử dụng "new MyClass()".

Đúng vậy. Nhưng đó chỉ là một suy nghĩ của bạn. Vậy đoạn code này thế nào?

```
public MyClass {
      public static MyClass getInstance() {
      }
}
```

- MyClass là một lớp với một static method. Chúng ta có thể gọi static method như thế này: MyClass.getInstance();

Tại sao bạn sử dụng **MyClass.getInstance()**, thay vì gọi thông qua đối tượng **myObject.getInstance()**?

- Chà, getInstance() là một static method; nói cách khác, nó là một CLASS method (ý nói phương thức này thuộc về class, không phải thuộc về instance). Bạn cần sử dụng tên lớp để tham chiếu một static method.

Rất thú vị. Điều gì sẽ xảy ra nếu chúng ta đặt mọi thứ lại với nhau. Bây giờ tôi có thể khởi tạo MyClass không?

```
public MyClass {
      private MyClass() {}
      public static MyClass getInstance() {
            return new MyClass();
      }
}
```

- Wow, chắc chắn có thể.

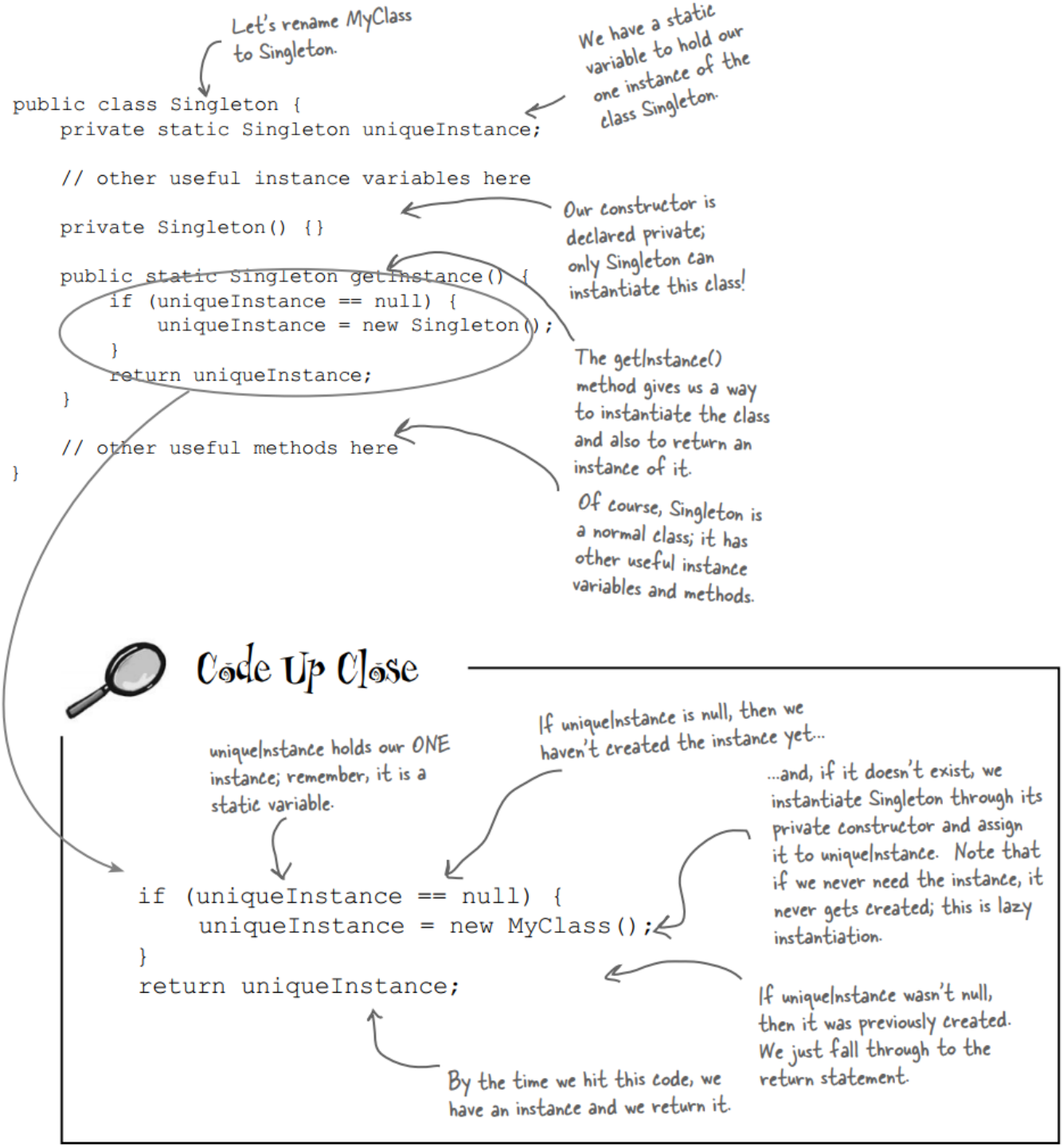
Vì vậy, bây giờ bạn có thể nghĩ ra một cách thứ hai để khởi tạo một đối tượng không?

```
MyClass.getInstance();
```

Bạn có thể viết đoạn code để chỉ **MỘT** phiên bản **MyClass** được tạo không?

- Vâng tôi cũng nghĩ có thể...

Triển khai Singleton Pattern cổ điển



Nếu bạn chỉ lướt qua [cuốn sách (https://toihocdesignpattern.com/dich-sach/head-first-design-patterns)](https://toihocdesignpattern.com/dich-sach/head-first-design-patterns), đừng copy code này một cách mù quáng, bạn sẽ thấy nó có một vài vấn đề phía sau trong chương này.

Cuộc phỏng vấn tuần này: Lời thú nhận của một Singleton Pattern

HeadFirst: Hôm nay chúng tôi rất vui khi được phỏng vấn bạn, Singleton. Tại sao bạn không bắt đầu bằng cách cho chúng tôi biết một chút về bản thân bạn.

Singleton Pattern: Chà, tôi hoàn toàn độc nhất (hơi cô đơn); nhưng thực sự tôi chỉ có một.

HeadFirst: Một?

Singleton Pattern: Vâng, chỉ một. Tôi đã dựa trên Singleton Pattern, đảm bảo rằng bất cứ lúc nào cũng chỉ có một instance của tôi.

HeadFirst: Có phải đó là một sự lãng phí? Ai đó đã dành thời gian để phát triển một lớp toàn diện và bây giờ tất cả những gì chúng ta có thể nhận được là chỉ một đối tượng trong số đó?

Singleton Pattern: Không hề! Có sức mạnh trong "MỘT". Hãy nói rằng bạn có một đối tượng có chứa các cài đặt trong registry. Bạn không muốn có nhiều bản sao của đối tượng đó và các giá trị của nó chạy nhiều nơi – điều đó sẽ dẫn đến sự hỗn loạn. Bằng cách sử dụng một đối tượng như tôi, bạn có thể đảm bảo rằng mọi đối tượng trong ứng dụng của bạn đang sử dụng cùng một tài nguyên global.

HeadFirst: Hãy cho chúng tôi biết thêm đi.

Singleton Pattern: Ồ, tôi rất tốt cho mọi thứ. Độc thân đôi khi có những lợi thế mà bạn chưa biết. Tôi thường được sử dụng để quản lý nhóm tài nguyên, nhóm kết nối (connection) hoặc luồng (thread).

HeadFirst: Tuy nhiên, chỉ có một thể hiện (instance) của bạn? Điều đó nghe thật cô đơn.

Singleton Pattern: Dù chỉ có một trong số tôi, tôi vẫn bận rộn, nhưng thật tuyệt nếu nhiều nhà phát triển biết đến tôi – nhiều nhà phát triển gặp phải lỗi vì họ có nhiều bản sao "trôi nổi" trong ứng dụng mà họ thậm chí không biết.

HeadFirst: Nhưng làm sao bạn biết chỉ có một mình bạn? Không ai có thể dùng **new** để tạo ra một "***new* you**" hay sao?

Singleton Pattern: Không! Tôi thực sự độc nhất.

HeadFirst: Chà, có chắc các nhà phát triển có thể sẽ không khởi tạo bạn nhiều lần?

Singleton Pattern: Chắc chắc. Sự thật tôi không có public constructor.

HeadFirst: KHÔNG PUBLIC CONSTRUCTOR! Ồ, xin lỗi, không có public constructor ư?

Singleton Pattern: Đúng vậy. Constructor của tôi được khai báo là **private**.

HeadFirst: Nó hoạt động như thế nào? Làm thế nào để bạn có được instance?

Singleton Pattern: Bạn thấy đấy, để có được một đối tượng Singleton, bạn không cần khởi tạo đối tượng, bạn chỉ cần yêu cầu một instance. Vì vậy, lớp của tôi có một phương thức tĩnh gọi là getInstance(). Gọi nó và tôi sẽ xuất hiện ngay lập tức, sẵn sàng làm việc. Trong thực tế, tôi có thể đang được các đối tượng khác gọi đến trong khi bạn yêu cầu tôi.

HeadFirst: Chà, Mr.Singleton, dường như có rất nhiều thứ dưới vỏ bọc của bạn để làm tất cả công việc này. Cảm ơn vì đã tiết lộ bản thân và chúng tôi hy vọng sẽ sớm gặp lại bạn!

Nhà máy Chocolate

Mọi người đều biết rằng tất cả các nhà máy sô cô la hiện đại đều có nồi hơi sô cô la điều khiển bằng máy tính. Công việc của lò hơi là lấy sô cô la và sữa, đun sôi chúng, sau đó chuyển chúng sang giai đoạn tiếp theo để làm thanh sô cô la.

Ở đây, lớp điều khiển của công ty Choc-O-Holic, Inc. Nồi hơi sô cô la cường độ công nghiệp. Kiểm tra code; bạn sẽ nhận thấy họ đã cố gắng hết sức cẩn thận để đảm bảo rằng những điều tồi tệ không xảy ra, như rút hết 500 gallon hỗn hợp chưa đun sôi, hoặc đổ vào nồi hơi đến khi nó đầy, hoặc kiểm tra việc đun một nồi hơi khi đang trống không!

```
public class ChocolateBoiler {
    private boolean empty;
    private boolean boiled;

    public  ChocolateBoiler() {
        empty = true;
        boiled = false;
    }

    public void fill() {
        if (isEmpty()) {
            empty = false;
            boiled = false;
            // fill the boiler with a milk/chocolate mixture
        }
    }

    public void drain() {
        if (!isEmpty() && isBoiled()) {
            // drain the boiled milk and chocolate
            empty = true;
        }
    }

    public void boil() {
        if (!isEmpty() && !isBoiled()) {
            // bring the contents to a boil
            boiled = true;
        }
    }

    public boolean isEmpty() {
        return empty;
    }

    public boolean isBoiled() {
        return boiled;
    }
}
```

This code is only started when the boiler is empty!

To fill the boiler it must be empty, and, once it's full, we set the empty and boiled flags.

To drain the boiler, it must be full (non empty) and also boiled. Once it is drained we set empty back to true.

To boil the mixture, the boiler has to be full and not already boiled. Once it's boiled we set the boiled flag to true.

Sử dụng sức mạnh bộ não

Choc-O-Holic đã thực hiện một việc tốt là đảm bảo những sai lầm không xảy ra, bạn có nghĩ thế không? Nhưng sau đó, bạn có thể nghi ngờ rằng nếu 2 instance của **ChocolateBoiler** được tạo, một số điều rất xấu có thể xảy ra.

Làm thế nào mọi thứ có thể đi sai nếu nhiều hơn một instance của ChocolateBoiler được tạo ra trong một ứng dụng? (Xem đáp án ở phần dưới).

Bạn có thể giúp Choc-O-Holic cải thiện lớp ChocolateBoiler của họ bằng cách biến nó thành một singleton không?

```
public class ChocolateBoiler {
    private boolean empty;
    private boolean boiled;

    [          ]

    [          ] ChocolateBoiler() {
        empty = true;
        boiled = false;
    }

    [          ]

    public void fill() {
        if (isEmpty()) {
            empty = false;
            boiled = false;
            // fill the boiler with a milk/chocolate mixture
        }
    }
    // rest of ChocolateBoiler code...
}
```

Đáp án:

Sharpen your pencil Can you help Choc-O-Holic improve their ChocolateBoiler class by turning it into a singleton?

```
public class ChocolateBoiler {
    private boolean empty;
    private boolean boiled;

    private static ChocolateBoiler uniqueInstance;

    private ChocolateBoiler() {
        empty = true;
        boiled = false;
    }

    public static ChocolateBoiler getInstance() {
        if (uniqueInstance == null) {
            uniqueInstance = new ChocolateBoiler();
        }
        return uniqueInstance;
    }

    public void fill() {
        if (isEmpty()) {
            empty = false;
            boiled = false;
            // fill the boiler with a milk/chocolate mixture
        }
    }
    // rest of ChocolateBoiler code...
}
```

Định nghĩa Singleton Pattern

Bây giờ bạn đã có bản triển khai **Singleton** cổ điển trong đầu, đã đến lúc ngồi lại, thưởng thức một thanh sô cô la và kiểm tra các điểm khác của Mẫu Singleton.

Hãy bắt đầu với định nghĩa ngắn gọn của mẫu:

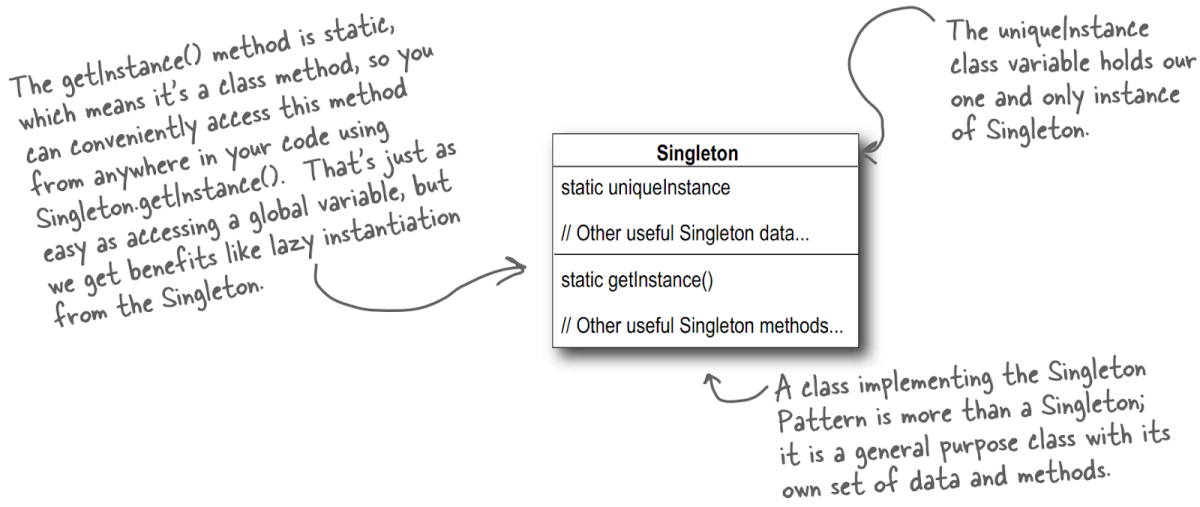
> **The Singleton Pattern** ensures a class has only one instance, and provides a global point of access to it.

(Mẫu Singleton đảm bảo rằng một lớp chỉ có một thể hiện (instance) duy nhất và cung cấp một điểm truy cập toàn cục đến nó)

Không có bất ngờ lớn ở đây. Nhưng, hãy tìm hiểu nó thêm một chút nữa:

- Điều gì đang thực sự xảy ra ở đây? Chúng tôi lấy một lớp và để nó tự quản lý một thể hiện (instance) của nó. Chúng tôi cũng không cho bất kỳ lớp nào khác tự tạo một instance mới. Để có được một instance, bạn đã phải gọi thông qua chính lớp đó.
- Chúng tôi cũng cung cấp một điểm truy cập toàn cục cho instance: bất cứ khi nào bạn cần một instance, chỉ cần truy vấn lớp và nó sẽ return lại cho bạn instance duy nhất. Như bạn đã thấy, chúng ta có thể thực hiện điều này để Singleton được tạo ra một cách lười biếng (https://en.wikipedia.org/wiki/Lazy_loading) (lazy – khi cần thì mới khởi tạo), điều này đặc biệt quan trọng đối với các đối tượng sử dụng nhiều tài nguyên.

Được rồi, hãy để kiểm tra sơ đồ lớp:



Chúng tôi có một vấn đề...

Có vẻ như Chocolate Boiler đã làm chúng ta thất vọng; mặc dù thực tế chúng tôi đã cải thiện code bằng Singleton cổ điển, nhưng bằng cách nào đó, phương thức **fill()** của **ChocolateBoiler** đã có thể đổ thêm sô cô la và sữa vào nồi hơi mặc dù đã có sữa và sô cô la đang sôi trong nồi! Và 500 gallon sữa (và sô cô la) bị tràn! Chuyện gì đã xảy ra!?

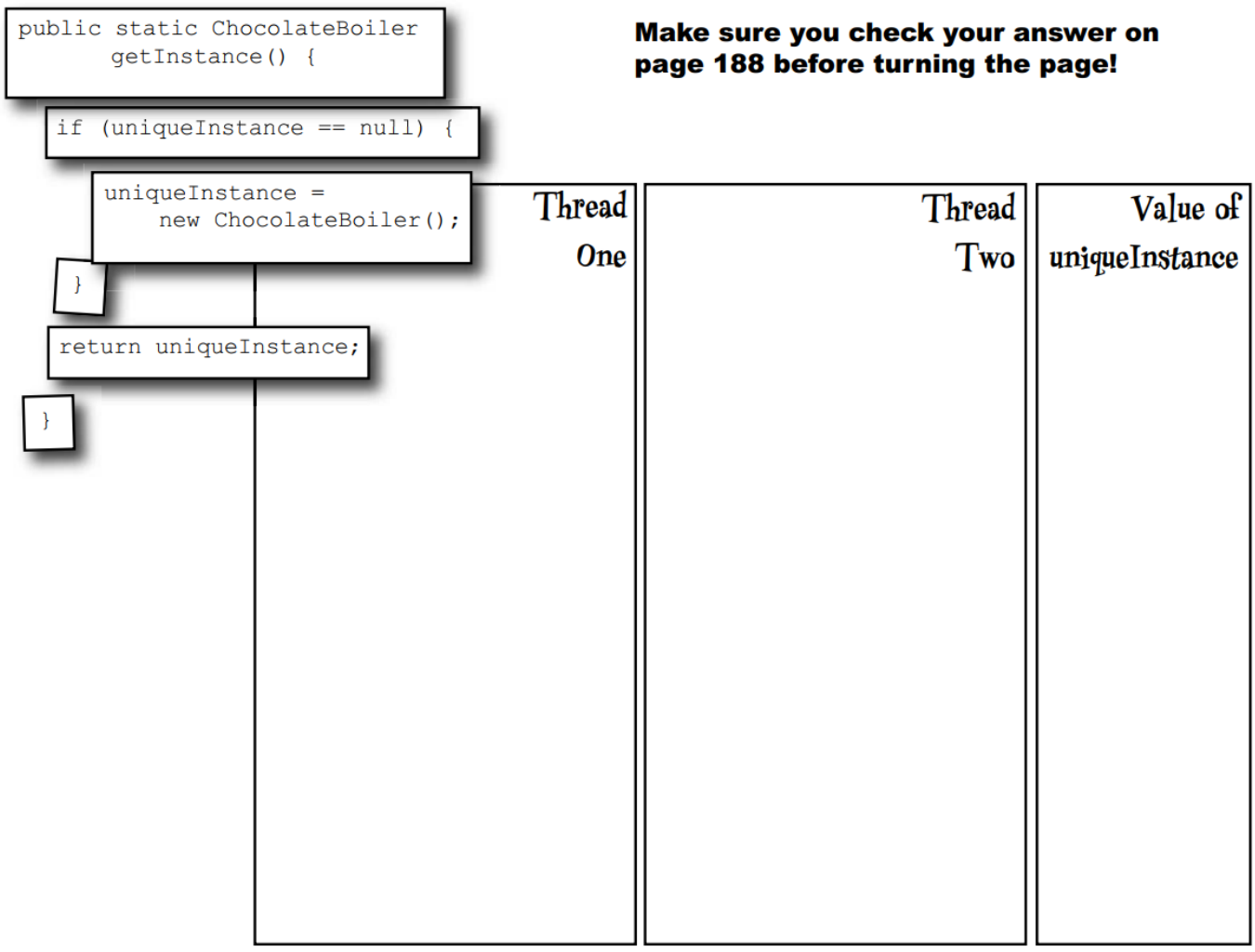
Có thể việc bổ sung các thread đã gây ra điều này? Có phải đó là trường hợp một khi chúng ta đã đặt biến ***uniqueInstance*** thành phiên bản duy nhất của **ChocolateBoiler**, tất cả các lệnh gọi **getInstance()** sẽ trả về cùng một instance? Đúng không?

Trở thành JVM (Java Virtual Machine)

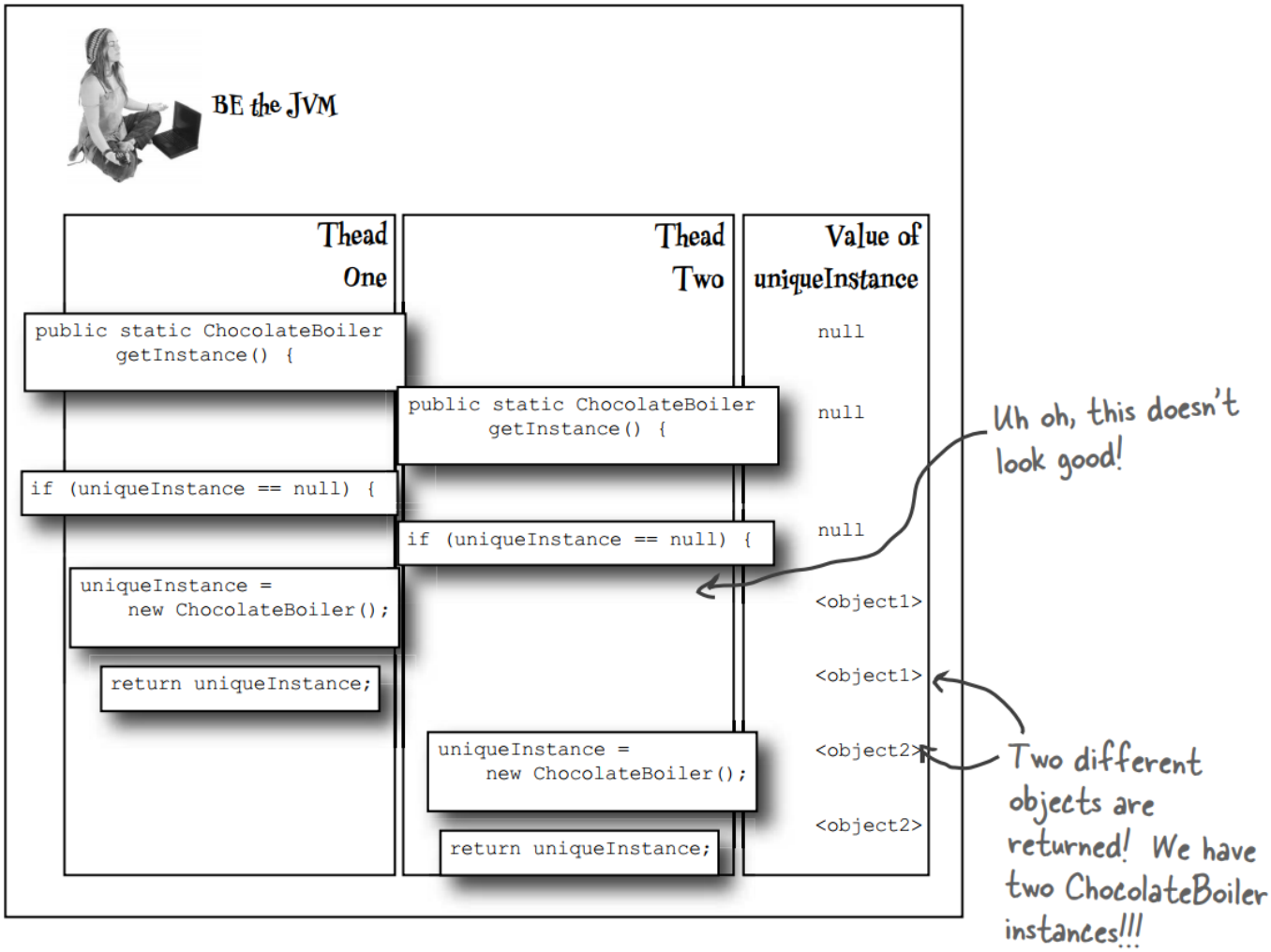
Chúng tôi có hai luồng (thread), mỗi luồng đều thực thi đoạn code bên dưới.

```
ChocolateBoiler boiler =
        ChocolateBoiler.getInstance();
fill();
boil();
drain();
```

Công việc của bạn là chạy JVM và xác định xem có trường hợp nào hai luồng có thể nhận được các đối tượng nồi hơi khác nhau hay không. Gợi ý: bạn thực sự chỉ cần xem chuỗi hoạt động trong phương thức **getInstance()** và giá trị của ***uniqueInstance*** để xem chúng có thể trùng nhau như thế nào. Sử dụng công cụ code **Magnets** để giúp bạn nghiên cứu cách code có thể xen kẽ để tạo hai đối tượng nồi hơi.



Đáp án: Đây là trường hợp vẫn tạo ra được 2 instance mặc dù dùng Singleton pattern.



Xử lý đa luồng cho Singleton Pattern

Các vấn đề trong đa luồng của chúng ta hầu như được sửa chữa một cách thông thường bằng cách biến getInstance() thành một phương thức được đồng bộ hóa (synchronized):

```
public class Singleton {
    private static Singleton uniqueInstance;

    // other useful instance variables here

    private Singleton() {}

    public static synchronized Singleton getInstance() {
        if (uniqueInstance == null) {
            uniqueInstance = new Singleton();
        }
        return uniqueInstance;
    }

    // other useful methods here
}
```

By adding the synchronized keyword to getInstance(), we force every thread to wait its turn before it can enter the method. That is, no two threads may enter the method at the same time.

Tôi đồng ý điều này đã khắc phục vấn đề.

Nhưng đồng bộ hóa là "quá tốn kém"; Đây có phải cũng là một vấn đề?

Suy nghĩ rất tốt, và nó thực sự tệ hơn một chút so với bạn nghĩ: thời gian duy nhất cần đến việc đồng bộ là lần đầu tiên gọi phương thức này. Nói cách khác, một khi chúng ta đã khởi tạo được biến uniqueInstance, chúng ta không cần phải đồng bộ hóa phương thức này nữa. Sau lần đầu tiên, việc đồng bộ hóa hoàn toàn không cần thiết!

Chúng ta có thể cải thiện đa luồng không?

Đối với hầu hết các ứng dụng Java, rõ ràng chúng ta cần đảm bảo rằng Singleton hoạt động với sự có mặt của nhiều luồng. Nhưng, có vẻ khá tốn kém để đồng bộ hóa phương thức **getInstance()**, vậy chúng ta phải làm gì?

Chà, chúng tôi có một vài lựa chọn...

1. Không làm gì nếu hiệu suất của getInstance() không quan trọng đối với ứng dụng của bạn

Đúng rồi; nếu gọi phương thức **getInstance()** là không gây ra chi phí đáng kể cho ứng dụng của bạn, hãy quên nó đi. Đồng bộ hóa **getInstance()** rất đơn giản và hiệu quả. Chỉ cần lưu ý rằng đồng bộ hóa một phương thức có thể làm giảm hiệu suất xuống 100 lần, vì vậy nếu nó chiếm một phần lưu lượng truy cập cao trong code của bạn, bạn có thể phải xem xét lại.

2. Sử dụng kĩ thuật eagerly created instance (load tất cả lên 1 lần duy nhất ngay từ đầu) thay vì một lazily created (dùng tới đâu load tới đó)

Nếu ứng dụng của bạn luôn tạo và sử dụng một phiên bản của Singleton hoặc chi phí việc tạo và thời gian tạo Singleton không phải là vấn đề, bạn có thể muốn tạo Singleton eagerly (khởi tạo ngay từ đầu), như thế này:

```
public class Singleton {
    private static Singleton uniqueInstance = new Singleton();

    private Singleton() {}

    public static Singleton getInstance() {
        return uniqueInstance;
    }
}
```

Go ahead and create an instance of Singleton in a static initializer. This code is guaranteed to be thread safe!

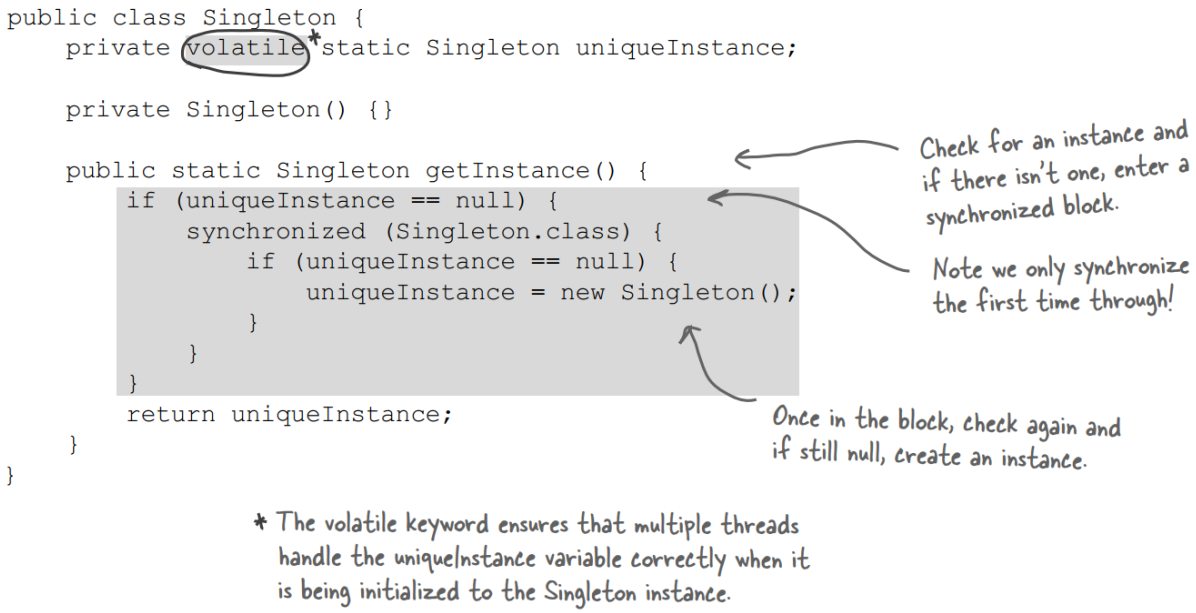
We've already got an instance, so just return it.

Sử dụng cách tiếp cận này, chúng tôi dựa vào JVM để tạo instance duy nhất của Singleton khi lớp được load. JVM đảm bảo rằng instance sẽ được tạo trước khi bất kỳ luồng nào truy cập vào biến static ***uniqueInstance***.

3. Sử dụng "double-checked locking", để giảm việc sử dụng synchronized trong getInstance()

Với double-check locking, trước tiên chúng tôi kiểm tra xem liệu một phiên bản có được tạo không và nếu không, thì chúng tôi sẽ đồng bộ hóa. Bằng cách này, chúng tôi chỉ đồng bộ hóa lần đầu tiên. Đây là những gì chúng ta muốn.

Hãy kiểm tra code:



Nếu hiệu suất là một vấn đề trong việc bạn sử dụng phương thức **getInstance()** thì phương thức triển khai Singleton này có thể làm giảm đáng kể chi phí.

Double-checked locking không hoạt động trong Java 1.4 trở về trước!

Thật không may, trong phiên bản Java 1.4 trở về trước, nhiều JVM chứa các triển khai từ khóa ***volatile*** cho phép đồng bộ hóa không đúng cách để double-check locking. Nếu bạn phải sử dụng một JVM cũ hơn Java 5, hãy xem xét các cách khác để triển khai Singleton.

Trong khi đó, trở lại tại Nhà máy Sô cô la...

Trong khi chúng tôi đã tắt chẩn đoán các vấn đề đa luồng, nồi hơi sô cô la đã được làm sạch và sẵn sàng để làm việc. Nhưng trước hết, chúng ta phải sửa các vấn đề đa luồng. Chúng ta có sẵn một vài giải pháp, mỗi giải pháp có sự đánh đổi khác nhau, vậy chúng ta sẽ sử dụng giải pháp nào?

For each solution, describe its applicability to the problem of fixing the Chocolate Boiler code:

Synchronize the getInstance() method:

_____

_____

Use eager instantiation:

_____

_____

Double-checked locking:

_____

_____

Đáp án:

Sharpen your pencil

For each solution, describe its applicability to the problem of fixing the Chocolate Boiler code:

Synchronize the getInstance() method:

A straightforward technique that is guaranteed to work. We don't seem to have any performance concerns with the chocolate boiler, so this would be a good choice.

Use eager instantiation:

We are always going to instantiate the chocolate boiler in our code, so statically inializing the instance would cause no concerns. This solution would work as well as the synchronized method, although perhaps be less obvious to a developer familar with the standard pattern.

Double checked locking:

Given we have no performance concerns, double-checked locking seems like overkill. In addition, we'd have to ensure that we are running at least Java 5.

Xin chúc mừng!

Tại thời điểm này, Choc-O-Holic rất vui khi có một số chuyên môn áp dụng cho code nồi hơi của họ. Bất kể bạn áp dụng giải pháp đa luồng nào, nồi hơi phải ở trong tình trạng tốt, không có nhiều rủi ro. Xin chúc mừng. Bạn không chỉ xoay sở để thoát khỏi 500 lbs (Pound – đơn vị tính của Anh) sô cô la nóng trong chương này, mà bạn còn phải trải qua tất cả các vấn đề tiềm ẩn của Singleton.

Không có câu hỏi ngớ ngẩn

Hỏi: Đối với một mẫu đơn giản như vậy chỉ bao gồm một lớp, Singletons dường như có một số vấn đề.

Trả lời: Vâng, chúng tôi đã cảnh báo bạn từ trước! Nhưng đừng để những vấn đề làm bạn nản lòng; Trong khi triển khai Singletons một cách chính xác có thể khó, nhưng sau khi đọc chương này, bạn đã được biết đầy đủ về các kỹ thuật tạo Singletons và nên sử dụng chúng bất cứ nơi nào bạn cần để kiểm

soát số lượng phiên bản bạn đang tạo.

Hỏi: Tôi có thể tạo một lớp trong đó tất cả các phương thức và biến được định nghĩa là static không? Điều đó có giống với Singleton không?

Trả lời: Có, nếu lớp của bạn được tự kiểm soát và không phụ thuộc vào việc khởi tạo phức tạp. Tuy nhiên, do cách xử lý khởi tạo static trong Java, điều này có thể trở nên rất lộn xộn, đặc biệt là nếu có nhiều lớp tham gia. Thông thường kịch bản này có thể dẫn đến các lỗi tinh vi, khó tìm thấy liên quan đến thứ tự khởi tạo. Trừ khi có một nhu cầu hấp dẫn để cài đặt chương trình singleton của bạn theo cách này, tốt hơn hết là ở lại trong thế giới đối tượng.

Hỏi: Thế còn class loaders? Tôi nghe nói có khả năng hai class loaders có thể kết thúc bằng instance Singleton của riêng chúng.
*(**Java ClassLoader** là công cụ của java, tiến hành nạp một tệp class của java vào máy ảo (JVM))*

Trả lời: Vâng, điều đó đúng vì mỗi class loader định nghĩa một namespace. Nếu bạn có hai hoặc nhiều class loader, bạn có thể tải cùng một lớp nhiều lần (một lần trong mỗi class loader). Bây giờ, nếu lớp đó là Singleton, thì vì chúng ta có nhiều hơn một phiên bản của lớp, chúng ta cũng có nhiều hơn một phiên bản của Singleton. Vì vậy, nếu bạn đang sử dụng nhiều class loader và Singletons, hãy cẩn thận. Một cách xung quanh vấn đề này là tự xác định class loader.

Thư giãn: Tin đồn về việc Singletons bị dọn dẹp bởi những trình thu gom rác garbage collectors được thổi phồng lên rất nhiều

Trước Java 1.2, một lỗi trong trình thu gom rác đã cho phép Singletons bị thu dọn sớm nếu không có tham chiếu toàn cục đến chúng. Nói cách khác, bạn có thể tạo một Singleton và nếu tham chiếu duy nhất đến Singleton là ở chính Singleton, nó sẽ được thu gom và phá hủy bởi garbage collector. Điều này dẫn đến những lỗi khó hiểu bởi vì sau khi Singleton được thu thập, thì lần gọi tiếp theo đến **getInstance()** đã tạo ra một Singleton mới hoàn toàn. Trong nhiều ứng dụng, điều này có thể gây ra hành vi khó hiểu vì trạng thái được đặt lại một cách bí ẩn thành các giá trị mặc định ban đầu hoặc những thứ như kết nối mạng được đặt lại.

Vì Java 1.2, lỗi này đã được sửa và không cần phải tham chiếu toàn cục nữa. Nếu bạn, vì một số lý do, vẫn đang sử dụng JVM 1.2 về trước, thì hãy lưu ý vấn đề này, còn nếu không, bạn có thể ngủ ngon khi biết rằng Singletons của bạn sẽ không được thu gom sớm.

Không có câu hỏi ngớ ngẩn

Hỏi: Tôi đã luôn được dạy rằng một lớp nên làm một việc và chỉ một việc duy nhất. Đối với một lớp để làm hai việc được coi là thiết kế OO xấu. Không phải Singleton vi phạm điều này sao?

Trả lời: Bạn sẽ đề cập đến nguyên tắc của "*Một lớp, một trách nhiệm*" (Single Responsibility Principle), và vâng, bạn đúng, Singleton không chỉ chịu trách nhiệm quản lí việc tạo 1 instance của nó (và cung cấp quyền truy cập toàn cục), nó còn chịu trách nhiệm cho bất kỳ vấn đề nào trong ứng dụng của bạn (kết nối DB, đăng kí registry, ghi log…). Vì vậy, chắc chắn có thể nói nó đang đảm nhận hai trách nhiệm. Tuy nhiên, không khó để thấy rằng có một lợi ích khi một lớp quản lý instance của chính nó; nó chắc chắn làm cho thiết kế tổng thể đơn giản hơn. Ngoài ra, nhiều nhà phát triển đã quen thuộc với mẫu Singleton vì nó được sử dụng rộng rãi. Điều đó nói rằng, một số nhà phát triển không cảm thấy cần phải trừu tượng hóa chức năng Singleton.

Hỏi: Tôi muốn phân lớp (subclass) code Singleton của mình, nhưng tôi gặp vấn đề. Có thể phân lớp Singleton không?

Trả lời: Một vấn đề với phân lớp Singleton là constructor là private. Bạn không thể mở rộng một lớp với một constructor private. Vì vậy, điều đầu tiên bạn sẽ phải làm là thay đổi công cụ xây dựng của mình để nó public hoặc protected. Nhưng sau đó, nó không thực sự là một Singleton nữa, bởi vì các lớp khác có thể khởi tạo nó. Và nếu bạn thay đổi constructor của bạn, vẫn có một vấn đề khác. Việc triển khai Singleton dựa trên một biến static, vì vậy nếu bạn implement một lớp con, tất cả các lớp dẫn xuất của bạn sẽ dùng chung một biến đối tượng. Đây có lẽ không phải là những gì bạn nghĩ trong đầu. Vì vậy, để việc phân lớp có thể hoạt động, việc thực hiện đăng ký các loại là bắt buộc trong lớp cơ sở. Trước khi thực hiện một kế hoạch như vậy, bạn nên tự hỏi những gì bạn thực sự đạt được từ việc phân lớp Singleton. Giống như hầu hết các mẫu, Singleton không nhất thiết phải là một giải pháp có thể phù hợp như một thư viện. Ngoài ra, Singleton là không đáng kể để thêm vào bất kỳ lớp hiện có nào. Cuối cùng, nếu bạn đang sử dụng một số lượng lớn Singletons trong ứng dụng của mình, bạn nên xem kỹ thiết kế của mình. Singletons có ý nghĩa khi được sử dụng một cách tiết kiệm.

Hỏi: Tôi vẫn chưa hoàn toàn hiểu tại sao các biến toàn cục (global variable) lại không bằng Singleton.

Trả lời: Trong Java, các biến toàn cục về cơ bản là các tham chiếu tĩnh đến các đối tượng. Có một vài nhược điểm khi sử dụng các biến toàn cục theo cách này. Chúng tôi đã đề cập đến một vấn đề: vấn đề khởi tạo lazy (https://en.wikipedia.org/wiki/Lazy_loading) (cần dùng mới load) và eager (load khi khởi tạo). Nhưng chúng ta cần ghi nhớ mục đích của mẫu: để đảm bảo chỉ có một thể hiện của một lớp tồn tại và cung cấp quyền truy cập global. Một biến toàn cục có thể cung cấp quyền truy cập global, nhưng không thể đảm bảo chỉ có một thể hiện của một lớp tồn tại. Các biến toàn cục cũng có xu hướng làm cho các nhà phát triển gây ô nhiễm không gian tên với nhiều tham chiếu toàn cầu đến các đối tượng nhỏ. Mẫu Singleton cũng không khuyến khích điều này, nhưng dù sao Singleton cũng có thể bị lạm dụng.

Tóm tắt

- Mẫu Singleton đảm bảo bạn có nhiều nhất một instance trong ứng dụng của mình.
- Mẫu Singleton cũng cung cấp một điểm truy cập toàn cục cho instance đó.
- Việc triển khai Java của mẫu Singleton sử dụng một private constructor, một static method kết hợp với một biến static.
- Kiểm tra các hạn chế về hiệu suất và tài nguyên của bạn và cẩn thận chọn một triển khai Singleton thích hợp cho các ứng dụng đa luồng (và chúng ta nên xem xét tất cả các ứng dụng đa luồng!).
- Cảnh giác với việc thực hiện double-checked locking; nó không an toàn trong các phiên bản trước Java 2, version 5.
- Hãy cẩn thận nếu bạn đang sử dụng nhiều class loaders; điều này có thể đánh bại việc thực hiện Singleton và dẫn đến nhiều instances.
- Nếu bạn đang sử dụng JVM trước 1.2, bạn sẽ cần phải tạo một sổ đăng ký Singletons để đánh bại trình thu gom rác (garbage collector).