Padrões de Projeto:
uma Introdução ao Padrão Singleton

Clovis Torres Fernandes
2015

Gabriel Adriano de Melo
Dylan Nakandakari Sugimoto

Padrão Singleton [GoF: Padrão de Criação]

Objetivo: Assegurar que uma classe tenha somente uma instância e fornecer um ponto de acesso global a ela!

Por exemplo, suponha a existência da classe Application, que corresponda à interface com o usuário de uma dada aplicação. Assume-se que essa classe deva ter apenas uma instância e por convenção ela é instanciada apenas uma vez. Mas se alguém quisesse obter duas ou mais instâncias dela não seria impedido de fazer isso, como se pode ver abaixo:

```
public class Main {

    // ---------- Operations --------------------
    public static void    main(String[] args) {
        Application app1 = new Application();
        Application app2 = new Application();
        // …
    }  // method: main

} // class: Main
```

Por quê? Porque a classe Application possui um construtor padrão público e com ele pode-se criar qualquer número de instâncias.

Atividade: Como fazer então para garantir que a classe Application tenha uma e apenas uma instância?

A resposta é tornar o construtor padrão privado ou protegido, de modo que sem construtor padrão público clientes não possam criar quaisquer instâncias. Adicionalmente, se uma instância simples deve ser garantida, então a classe deve ter alguma operação para criar o singleton (instância única): operação de fábrica (Factory Method).

Em Java nós realizamos o padrão Singleton na classe Application com um método estático representando a operação de fábrica que garante que apenas uma instância esteja disponível. Esse método tem acesso ao atributo que se refere à instância única inicializada, como se pode ver abaixo:

```
public class Application {

        // ---------- Constructor --------------------
    private     Application() {
        // No initialization required
     }    // constructor: Application

        // ---------- Operations --------------------
    public static Application    getApplication() {
        return theApplication;
    }    // method: getApplication

        // Others methods protected by singleton-ness would be here
    // …

        // ---------- Attributes --------------------
    private static final Application    theApplication = new Application();

}    // class: Application
```

A classe Main revisada ficaria assim:

```
public class Main {
    // ---------- Operations --------------------
```

```
    public static void        main(String[] args) {
        Application app = Application.getApplication();
        app.run();
    }  // method: main

  } // class: Main
```

Como se poderia testar se realmente apenas uma instância será criada?

```
public class ApplicationTest {
        public static void main(String[] args) {
                Application app1 = Application.getApplication();
                app1.run( );
                Application app2 = Application.getApplication();
                app2.run( );
                if (app1 == app2)
                        System.out.println (“Same instance”);
                else
                        System.out.println (“Different instances”);
        }
}
```

Criação de Instância de Forma Preguiçosa

Nessa forma que foi estruturado o padrão singleton, uma instância da classe é criada em tempo de compilação. Supôs-se que tudo que era necessário para ela ser criada já existisse. Quando o único construtor é padrão isso é possível. Mas quando, para a criação da classe, seja necessária alguma informação de tempo de execução, essa abordagem não é apropriada. Como fazer então?

A saída é instanciar a instância única de forma preguiçosa, ou seja, apenas quando necessário em tempo de execução, como se pode ver abaixo, lembrando que se usa a classe Application apenas para efeito de demonstração:

```
public class Application {
```

```
        // ---------- Constructor --------------------
    private     Application() {
        //  construct object
        // …
    }    // constructor: Application

        // ---------- Operations --------------------
        // For lazy initialization
    public static Application    getApplication() {
        if (_theApplication == null) {
            _theApplication = new Application();
        }
        return _theApplication;
     }    // method: getApplication

        / Others methods protected by singleton-ness would be here
    // …

        // ---------- Attributes --------------------
    private static final Application    _theApplication = null;

}    // class: Application
```

Atividade: A propósito, como se poderia testar a classe singleton agora?

O teste anterior funciona para essa versão do singleton.

Singleton Usado de Forma Concorrente

Vale a pena apontar que a implementação anterior não é segura quando se usa um singleton no meio de *threads* do Java (linhas de execução concorrentes). A classe Application com instanciação preguiçosa e segura quanto a *threads* fica assim:

```
public class Application {
```

```
        // ---------- Constructor --------------------
    private     Application() {
        //  Construct object
        // …
    }    // constructor: Application

        // ---------- Operations --------------------
        // For lazy initialization
    public static synchronized Application getApplication() {
        if (! instanceFlag) {
            return create()
        } else {
            return _theApplication;
            }
    }    // method: getApplication

    private static Application create() {
        _theApplication = new Application();
        instanceFlag = true;
        return _theApplication;
    }    // method: create

protected void finalize() throws throwable {
        try {
            instanceFlag = false;        // free new singleton instance!
        } finally {
            super.finalize();
        }
    }    // method: finalize

        // Others methods protected by singleton-ness would be here
    // …
```

```
        // ---------- Attributes --------------------
    private static final Application    _theApplication = null;
    private static Boolean instanceFlag = false;

}   // class: Application
```

O método create é sincronizado, ou seja, implementa um **monitor**, de modo que quando estiver sendo executado, ele bloqueia a execução concorrente de qualquer método do objeto da classe Application, até que ele termine sua execução. Com isso, não há como alguma *thread* possa criar uma segunda instância de objeto Application.

Atividade: verificar se no exemplo acima não há mesmo como alguma *thread* possa criar uma segunda instância de objeto Application.

Como no método finalize() a varíavel instanceFlag é mudada para false, outra thread concorrente pode criar outra instância do singleton. Portanto, momentâneamente é possível existir duas instâncias do singleton, mas não há problema nisso, pois a segunda instância não possui referência e está sendo destruída pelo garbageCollector.

Destruição de Instâncias de Singleton

Em Java, a maneira de destruir um objeto é implícita. Deixa-se o objeto sem referência alguma para que o garbage collector passe e o colete, ou seja, o descarregue ou destrua. O método finalize, método de Object redefinido na classe Application, é chamado pelo garbage collector quando ele determina que não há nenhuma referência para o objeto. No caso acima, ao destruir a instância de Application, através de comando try-catch-finally ele seta o instanceFlag para false, indicando que não há instância criada, de modo que uma nova solicitação de getApplication irá resultar em nova construção de objeto Application.

Singleton Enum (específico de JAVA)

As implementações acima são usuais em várias linguagens com algumas adaptações.
Mas em Java, uma simplificação é implementar o singleton como um enum. Mas não vale para qualquer linguagem!

é tão simples que está até na wikipedia [4]

```
public enum Singleton {
    INSTANCE; //nome do objeto eh INSTANCE
    public void execute (String arg) {
        // Perform operation here
    }
}
```

Positive:
very easy to implement and has no drawbacks regarding serializable objects
Java's guarantee that any enum value is instantiated only once in a Java program.
Since Java enum values are globally accessible, so is the singleton, initialized lazily by the class loader.
Negative:
The drawback is that the enum type is somewhat inflexible.

Generalização: “Singleton até n”

Atividade – Pode-se então generalizar o problema: garantir que uma classe tenha no máximo n instâncias, sendo que instâncias podem ser destruídas, mesmo que de forma implícita. (Procure também pelo padrão Object Pool [1] [2])

Com base no exemplo dado, pode-se generalizar da seguinte maneira:

Máximo 1 instância	:var boolean	: False	: instância inexistente ou destruída
		: True	: instância criada

Máximo n instâncias	contador	: 0	: instância inexistente
		: 1	: 1 instância criada
		: 2	: 2 instâncias criadas
		:...	:...
		: n	: n instâncias criadas

Onde criar instância significa incrementar o contador e destruir instância significa decrementar contador. O “Singleton até 1” corresponde ao Singleton criado de forma preguiçosa!

Uma boa aplicação deste padrão é reusar objetos com criação custosa. Por exemplo, objetos que usam muita memória, e se fossem desalocados e novamente alocados, gastaria tempo e criaria espaços no heap. Ou objetos que são demorados para criar como uma conexão de rede ou de BD – pode ser melhor manter uma conexão aberta do que fecha-la e reabri-la continuamente.

Atividade: Estabelecer uma classe Application onde se pode criar até 5 instâncias de um objeto demorado para criar (insira um sleep de 3 segundos no construtor).
Então realize o seguinte teste (cada elemento deve ter um identificador seqüencial que deve aparecer nas impressões, para diferenciar):

1. crie um Singleton de n elementos (também chamado de Object Pool), pré-inicializado com 2 elementos e limitado a 3 elementos.
2. requisite 2 elementos, verifique que é instantâneo.
3. requisite um 3º elemento, verifique que demora os 3 segundos.
4. requisite um 4º elemento, verifique que é lançada uma exceção.
5. libere um dos elementos, devolvendo-o ao Pool
6. Agora realize uma última requisição, que deverá ser instantânea.

Como Evitar a Destruição de Objeto Singleton

Problema: Em Java, dado que a única referência à única instância do Singleton está na própria classe Singleton, o garbage collector poderá coletar a instância e descarregar a classe.

Solução: Registrar a instância em tabela do sistema. Por exemplo, usando um objeto da classe Hashtable, pode-se atribuir um nome à instância e então acessar a instância pelo nome, usando o método get(nome) de Hashtable. Já get(Object key) é um método de Hashtable que devolve o valor para o qual a chave especificada está mapeada no hashtable. Com isso, a instância única passa a ter uma referência implícita e não será destruída pelo garbage collector.

No exemplo de se ter exatamente uma instância única, deve-se criar uma hashtable com apenas um objeto Application, onde usa-se o string “Application”, por exemplo, como chave:

```
Hashtable application = new Hashtable();
application.put("Application", new Application.getApplication(1));
```

Para recuperar um objeto Application, usa-se o seguinte código:

```
Application app = (Application) application.get("Application");
```

Atividade: Como resolver o problema do garbage collector no caso da classe Application onde se podem criar até 5 instâncias?

Nesse caso, há um vetor estático que aponta para todas as Applications criadas. Dessa forma, enquanto a classe estiver carregada esses objetos ainda são apontados pelo vetor estático. Portanto, não existe esse problema com o garbage collector.

Problemas e críticas ao Singleton

[3] traz várias críticas ao Singleton, com variações e soluções alternativas.

Atividade: leia a referência crítica, e contraste com o conteúdo seguinte.

Herança

Um problema não tratado aqui diz respeito à questão de subclasses de Singleton! Recomendo que esse assunto seja pesquisado na Internet.
Se o Singleton não é lazy, a uníca instância é já criada com tipo definido em compile time.
E a versão preguiçosa???
Um bom exemplo positivo é mostrado em [3] (exemplo do FileSystem).

Singleton == variável global.

Algo acessível no programa inteiro não é uma variável global?
Não há a tentação de acessar o Singleton de vários lugares (“é fácil!”), criando acoplamentos?
Se decidirmos que o Singleton não deve ser mais Singleton (mais do que uma instância), então temos que mudar todas as classes que usam o Singleton.

Opinião a favor

Podemos realmente precisar de uma única instância reusada e global. Por exemplo, quando o singleton corresponde a um recurso único ou caro de construir. Portas, conexões, hardware, grandes blackboards na memória.
Também é comum Factories (ou Builders) que são singletons. Estas classes são usadas para construir objetos da classe algo. Como quem usa a factory/builder é quem usa a classe-alvo, não muda muito o acoplamento, e garante que existe apenas uma factory (por exemplo podemos controlar e/ou diferenciar quantas instancias são criadas)

Veja Também:

Objet Pool é muitas vezes um Singleton que gerencia vários objetos. Mas Object Pool pode não ser um Singleton, nesse caso cada Object Pool gerenciaria os seus objetos.
Thread Pool é uma especialização de Objet Pool. Aparece com freqüência em games e em frameworks de componentes como o ROS, onde se deseja limitar o numero de threads disponíveis: https://en.wikipedia.org/wiki/Thread_pool_pattern Thread Pool é comumente um singleton: não faz sentido podermos criar 2n threads se o propósito é limitar a aplicação à n threads.

[1] https://sourcemaking.com/design_patterns/object_pool

[2] http://gameprogrammingpatterns.com/object-pool.html

[3] http://gameprogrammingpatterns.com/singleton.html

[4] https://en.wikipedia.org/wiki/Singleton_pattern#The_enum_way

[5] Pankaj Kumar “Java Design Patterns A Programmer’s Approach” JournalDev 2014 (disponível como ebook na web (sem pirataria))