# enum Singleton: O Melhor Singleton em Java

## Ameaças à Instância Única de um Singleton

A intenção de um Singleton é dupla [GoF]:

- Assegurar a existência de apenas uma instância
- Oferecer um ponto de acesso global a sua instância única.

Quanto ao primeiro objetivo, a literatura nos apresenta a categoria dos Singletons gulosos (Eager Singletons) com exatamente uma instância desde o tempo de compilação; e a categoria dos Singletons preguiçosos (Lazy Singletons) com zero instância, antes do primeiro uso, e com uma instância, somente após o primeiro uso.

Na versão inicial do Java, era possível o Garbage Collector coletar objetos Singleton, mesmo com referência estática. Caso a instância existente fosse removida indevidamente pelo Garbage Collector, poderia causar eventualmente algum transtorno no processamento esperado da aplicação por essa ocorrência!

Felizmente no Java 2 corrigiu-se esse problema e o Garbage Collector a partir de então, por causa da instância estática do objeto Singleton, não poderá mais coletar objetos Singletons durante o tempo de vida do ClassLoader de uma dada instância JVM (Java Virtual Machine).

O que de pior pode acontecer a alguma forma de implementação de Singletons durante o tempo de vida do Singleton numa aplicação Java?

Resp.: Não garantir a instância única do padrão, ao permitir que outras instâncias possam ser criadas!

As formas usuais de implementação do Padrão Singleton têm os seguintes 4 tipos de ameaças à unicidade de instância:

- Ambientes de múltiplos threads inseguros – Lazy Singletons são suscetíveis à geração de outras instâncias do Singleton quando há problemas de sincronização em ambientes de múltiplos threads!
- Clonagem de objetos Singleton – Considerando que a classe Singleton implementa a interface Cloneable, ao usar o método clone( ) num objeto Singleton, pode-se criar uma nova instância desse objeto. O modo mais simples de se proteger dessa ameaça é não implementar a interface Cloneable. Se a classe Singleton já implementa ou é subclasse de classe que implementa a interface Cloneable, uma saída seria sobrepor o método clone( ) no Singleton, fazendo-o lançar uma Exception do tipo "Não é possível criar o clone da classe Singleton".

- Instanciação reflexiva de objetos Singleton – Um usuário avançado pode, por meio dos mecanismos de reflexão do Java, alterar o modificador de acesso privado do construtor de um Singleton em tempo de execução para público. Se isso acontecer, fica fácil criar uma outra instância do Singleton.
- Serialização/desserialização de objetos Singleton – Para serializar uma classe Singleton dos tipos usuais, devemos implementar essas classes com uma interface Serializable. Mas fazer isso não é suficiente para impedir o ataque. Ao desserializar uma classe, novas instâncias são criadas, não importando se o construtor é privado ou não. Como consequência, poderá haver potencialmente mais de uma instância da mesma classe Singleton dentro da JVM, violando a propriedade da unicidade de instâncias do Singleton.

## O enum Singleton

Uma maneira interessante e pouco usada para definir uma classe Singleton em Java, sugerida por Joshua Bloch, no livro Effective Java, é usar o enum type de um elemento apenas:

```
public enum Singleton {
	INSTANCE;
	// Outras variáveis de instância
	// e outros métodos públicos seguem aqui!
	public void aSingletonMethod( ){
		// . . .
	}
	// . . .
}
```

Você usa o enum Singleton da seguinte maneira:

```
	Singleton.INSTANCE.aSingletonMethod( );

	// ou assim

	Singleton singleton = Singleton.INSTANCE;

	singleton.aSingletonMethod( );
```

Diferentemente das formas usuais de implementar o Padrão Singleton, enum Singleton não tem problemas em garantir a unicidade de instância em todas as situações possíveis.

Aproveita-se a garantia de Java de que os valores de enum são instanciados apenas uma vez em um programa Java, são acessíveis globalmente e são instanciados preguiçosamente sem problemas potenciais com respeito a multi-threading. Além disso, em Java não é possível fazer instanciação por reflexão ou por clonagem de tipos enum.

## Exemplo de Implementação do enum Singleton

A numeração, variando de (1) na classe FileLogger e de (2) a (6) na classe Cliente, indica os tempos de criação e uso do objeto Singleton de um enum type:

1. Cria apenas o campo estático INSTANCE em Tempo de Compilação. O objeto FileLogger só será criado efetivamente na JVM no seu primeiro acesso!

2. Obtém instância do FileLogger. Ao acessar pela primeira vez o campo estático INSTANCE de FileLogger, o objeto da classe FileLogger será carregado e inicializado pela JVM. Este processo inicializa o campo estático INSTANCE uma única vez, de forma preguiçosa! **Tempo de Carga de Objeto** (**Tempo de Execução**)

3. Usa instância (única?) do FileLogger: Tempo de Execução

4. Obtém instância do FileLogger. Como o enum Data Type implementa internamente o Padrão Singleton do GoF, ao acessar pela segunda vez o campo estático INSTANCE, será devolvida a instância única inicializada preguiçosamente na primeira vez! Tempo de Execução

5. Usa instância única do FileLogger: **Sim, a instância é verdadeiramente única!!!** Tempo de Execução

```
public enum FileLogger {
	// (1) Cria apenas o campo estático INSTANCE em "Tempo de Compilação"
	// O objeto FileLogger só será criado na JVM no seu primeiro acesso!
	INSTANCE;

	public void log(String msg) {
		// Simula código para fazer o log de mensagens
		System.out.println("Registrado no Log: " + msg);
	}

	// Outros métodos e váriáveis de instância próprios do FileLogger
}

public class Cliente {

	public static void main(String[] args) {
		// (2) Obtém instância do FileLogger
		// Ao acessar pela primeira vez o campo estático INSTANCE de Filelloger,
		// a classe Filelloger será carregada e inicializada pela JVM.
		// Este processo inicializa o campo estático INSTANCE uma vez,
		// de forma preguiçosa!
		FileLogger logger = FileLogger.INSTANCE;
		// (3) Usa instância do FileLogger
		logger.log("Uma mensagem para registrar");

		System.out.println();

		// (4) Obtém instância do FileLogger
		// Como o enum Data Type implementa internamente o Padrão Singleton do
```

```
                // GoF,ao acessar pela segunda vez o campo estático INSTANCE,
                // será devolvida a instância única (?) inicializada preguiçosamente na
                // primeira vez!
                FileLogger logger02 = FileLogger.INSTANCE;
                // (5) Usa instância do FileLogger e testa se o Singleton funciona mesmo!
                if (logger == logger02) {
                        System.out.println("Os objetos logger e logger02 são a mesma
instância");
                } else // Trecho de código a seguir inacessível, por princípio!
                        System.out.println("O padrão Singleton falhou!!!");

                System.out.println();
                // (6) Usa instância única do FileLogger
                logger02.log("Outra mensagem para registrar");
        }
}
```

## O enum Singleton Resiste às Ameaças à Unicidade de Instância?

É interessante notar que as subclasses da classe Enum são a maneira recomendada e segura de tipo em Java para representar um conjunto fixo consistindo em valores enumeráveis, serializáveis e comparáveis. Mais interessante ainda é constatar que o padrão Enum emprega uma variante do padrão de projeto Singleton, o que significa que cada valor é garantido ter apenas uma instância Enum por JVM.

Até aqui estamos nos referindo à classe Enum. O enum type é uma subclasse da classe Enum embutida na linguagem Java.

Os tipos enums são implicitamente declarados public, static e final, o que significa que o Garbage Collector não poderá coletá-los (por causa do static) durante o tempo de vida do ClassLoader de uma dada instância JVM. Além disso, você não poderá estendê-los (por causa do final).

Uma necessidade real de objeto Singleton é uma situação rara. Mais rara ainda é a necessidade de se ter subclasses de Singleton. Mas se esse for o seu caso e o seu Singleton tiver que ter subclasses, esqueça o uso do enum Singleton, uma vez que com ele não se pode derivar subclasses.

Quanto aos 4 tipos de ameaças à unicidade de instância, o enum Singleton não dá chance a nenhuma delas:

- Apesar de ser um tipo de Lazy Singleton, o enum Singleton é seguro em ambientes de múltiplos threads, funcionando como se fosse um Eager Singleton, não apresentando problemas de sincronização que poderiam dar margem à criação de outras instâncias!
- As constantes de enumeração não podem ser clonadas. No Eclipse, nem se consegue escrever código para clonar, pois aparece o seguinte erro: “O método clone( ) do tipo Enum<...> is not visible”. Isso deve ao fato do método

do clone ser final em Enum, da qual o tipo enum é subclasse, o que garante que as constantes de enum nunca possam ser clonadas.

Ou seja, não dá para clonar enum data type, nem mesmo compilar!

- Instanciação por reflexão de tipos enum é terminantemente proibida em Java. A menos que você seja o desenvolvedor Jan Ouwens, que mostra no artigo "Hacking Java enums", postado no seu blog em http://jqno.nl/post/2015/02/28/hacking-java-enums/, que uma biblioteca usada no Mockito e na maioria dos frameworks de mock testing, bem como no Spring Framework, possibilitou, de forma não planejada por ele, um ataque de instanciação por reflexão em um enum type da sua aplicação. Como resultado, permitiu criar uma segunda diferente instância para o seu objeto enum. Isso tudo sem precisar mudar bytecodes da aplicação em tempo de execução: apenas usando reflexão pura!
- O tratamento especial pelo mecanismo de serialização de tipos enum garante que instâncias duplicadas nunca possam ser criadas como resultado da desserialização.

Uma quinta ameaça no caso do tipo enum é a possibilidade de instanciar com new um tipo enum, descartada nos outros tipos de implementação, onde, por criação, os seus construtores são privados. Mas como o tipo enum tem implícito um construtor privado e não permite explicitar construtor público, apenas privado, também isso é impossível. Conforme declarado em Java Language Specification, Java SE 8 Edition section 8.9. Enum Types:

> "É um erro de tempo de compilação tentar instanciar explicitamente um tipo de enum (§15.9.1)"

Juntas, essas cinco características do tipo enum oferecem uma boa garantia de que nenhuma instância de um enum Singleton existirá além daquela definida pela constante enum.

Nós dissemos anteriormente que o enum Singleton é um tipo de Lazy Singleton. No exemplo apresentado acima, a numeração usada explicita essa criação preguiçosa do campo *INSTANCE* do Singleton FileLogger.

O campo enum, no caso *INSTANCE*, é constante definida em tempo de compilação (1); mas esse campo é instância de seu tipo de enum, no caso FileLogger. Quando seu código acessar pela primeira vez o *INSTANCE* (2), a classe FileLogger será carregada e inicializada pela JVM. Esse processo inicializará o campo estático acima uma vez e preguiçosamente, apenas e exatamente quando o tipo de enum é referenciado pela primeira vez.

Ao referenciar o campo INSTANCE pela segunda vez `(4)`, fica comprovado que é a instância única do Singleton FileLogger é que é devolvida!

As instâncias são criadas durante a inicialização estática, que é definida na Especificação da Linguagem Java, Seção 12.4:

> "Os campos Enum são constantes de tempo de compilação, mas instâncias do tipo enum. A semântica Enum garante que haverá apenas uma instância. A instância é construída quando o tipo de enum é referenciado pela primeira vez."