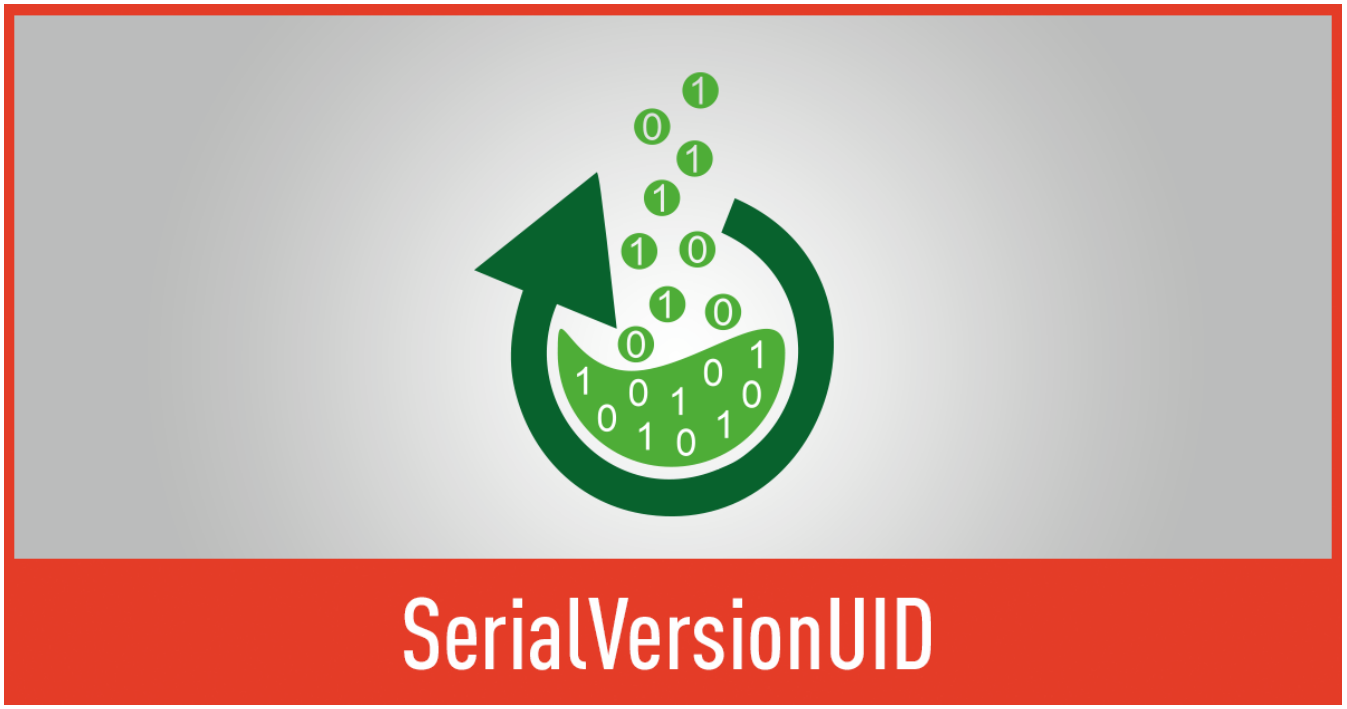
JAVA E OO

Entenda a variável serialVersionUID e sua importância na arquitetura Java

Escrito por **Alexandre Afonso**
em 02/03/2017



Apesar de tão presente no dia a dia dos desenvolvedores Java, muitos ainda tem dúvidas sobre o atributo **serialVersionUID**:

- Quando usar?
- De onde o Eclipse tira o valor dessa propriedade?
- Por que é um número tão grande? Não poderia ser 1?
- Se alterar, vai dar problemas?
- Ele pode ser removido?

Nesse artigo vou responder a essas dúvidas e, por tabela, algumas

outras também.

Acho que essas respostas vão até diminuir um pouco da estranheza que essa propriedade passa. Eu, pelo menos, achava esquisito ela ali, meio que solta no código.

Esse estranhamento é porque mesmo a linguagem Java tendo tantos recursos de organização (interfaces, enums, anotações, etc.), ainda assim, ela parece perdida no meio da classe.

Aqui você vai conhecer o propósito de vida da propriedade **serialVersionUID**.

Você sabe o que significa serializar um objeto?

Antes de entender o que é o **serialVersionUID**, é importante conversarmos sobre o que é serialização.

Serializar um objeto, dentro da plataforma Java, significa converter o estado atual dele em um formato padrão e depois disponibilizá-lo em um stream de bytes que poderá ser escrito em disco ou transmitido.

Repare que a palavra **serializar**, dentro do Java, é um pouco mais do que "entregar em partes" – que seria a definição da mesma. É preciso que essas partes tenham uma estrutura padronizada para que seja possível a **desserialização**.

O legal de notar é que, entre o momento em que ele foi colocado no *stream de bytes* até o momento de ser **desserializado**, você pode

encarar o “objeto” como se ele fosse um arquivo qualquer como, por exemplo, uma imagem ou um PDF, pois, ele nada mais será que um amontoado de *bytes*. Portanto, como mencionei, você pode persistir em disco ou transmiti-lo pela rede.

Cenários comuns para o uso do mecanismo de serialização, dentro do ecossistema Java, são as invocações de métodos remotos (RPC) e também na replicação de sessões dos servidores web ou de aplicação.

Provavelmente, você já sabe isso, mas quero lembrar que: para um objeto tornar candidato a ser serializado, ele e toda a sua hierarquia de propriedades devem implementar a interface **java.io.Serializable**. Lembrando que muitas classes no Java já fazem isso, inclusive, as classes **wrappers** juntamente com os seus primitivos – esses não implementam, pois, não são classes, mas podem ser serializados.

Entendido o porquê de se serializar um objeto, agora podemos passar para uma outra pergunta:

O que é o serialVersionUID?

Esse é um atributo utilizado para controlar explicitamente a compatibilidade entre o **.class** usado para serializar e o **.class** que será utilizado na desserialização.

O controle é necessário porque um **.class** pode ter sofrido alterações e ainda assim se manter compatível com sua versão anterior.

O ideal, claro, é utilizar as mesmas versões de **.class**, mas nem

sempre isso será possível.

Aí que entra o serialVersionUID.

Ele é o recurso que usamos para dizer ao Java que um objeto serializado é compatível ou não com o **.class** utilizado para desserializar.

Inclusive, sempre que temos uma classe estendendo a interface **java.io.Serializable** esse atributo é criado, mesmo que a implementação venha por alguma superclasse.

Caso ele não esteja declarado por você (explicitamente) o Java irá declará-lo no momento da compilação.

Adiantando... Deixar isso para o Java fazer implicitamente não é uma boa ideia. Mais para frente no artigo veremos melhor sobre.

Antes, vamos observar quais as regras que o Java utiliza para gerar o valor do atributo automaticamente.

Como a geração implícita é feita

Como disse, caso você não informe o atributo **serialVersionUID**, o Java o fará por você na hora em que for compilar e para gerar o valor de **serialVersionUID** são levados em consideração alguns aspectos de estrutura da classe:

- o nome da classe e seus modificadores;
- o nomes das interfaces que a classe implementa;
- os atributos e seus modificadores;

- e algumas outras coisas como você pode conferir na documentação do processo de serialização.

Esses elementos são organizados, então é aplicado um algoritmo de hash que depois vira o valor do atributo **serialVersionUID**.

Qualquer alteração nesses elementos acarretará em um valor diferente.

Uma coisa legal é que dá para saber qual o valor que o Java atribuiu implicitamente.

Para isso podemos usar a ferramenta **serialver** que vem com o **JDK**. Ainda nesse artigo vamos aprender como utilizá-la. Tem até um exemplo real de como ela pode ser útil para quem desenvolve em Java.

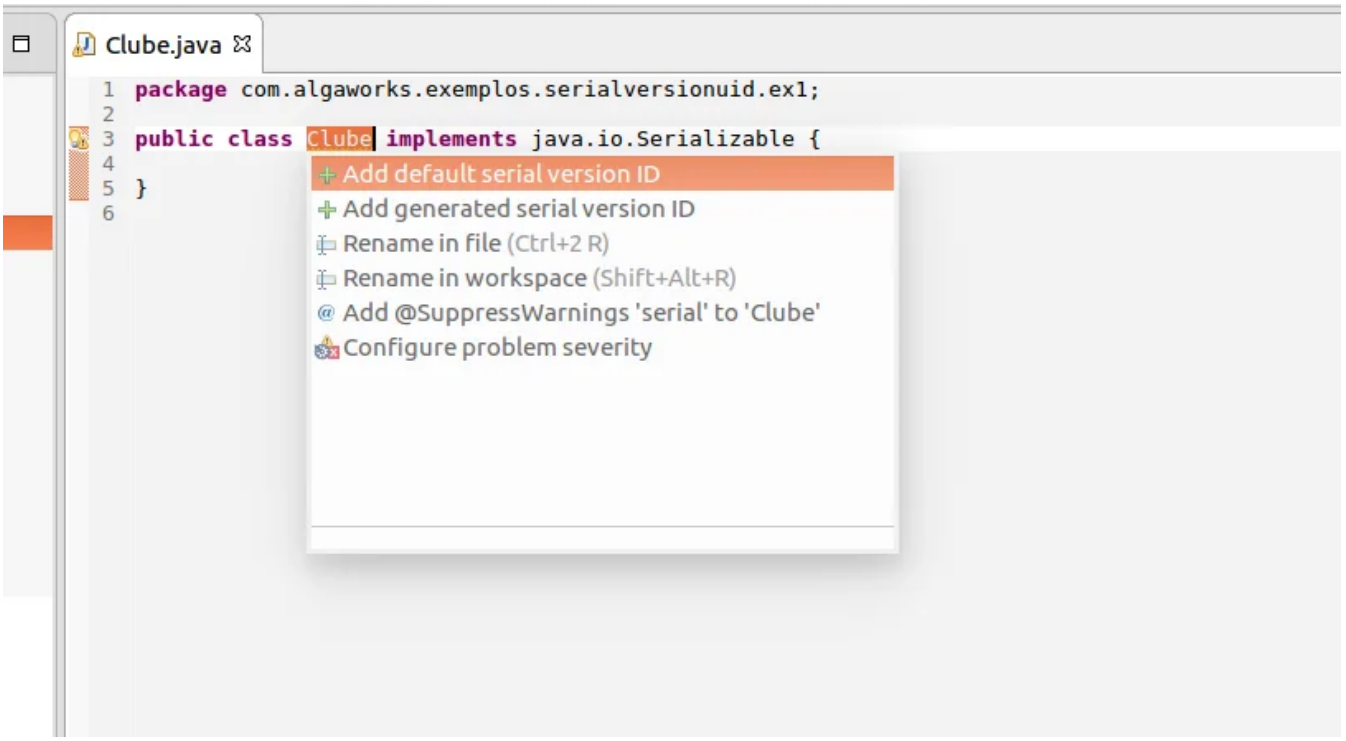
O número gerado pelo Eclipse

Você já deve ter reparado que o Eclipse gera um aviso quando temos uma classe que implementa **java.io.Serializable**, não é mesmo?

Clicando em cima desse aviso, ele nos dá 3 opções:

1. Adicionar um **serial version ID** padrão;
2. Gerar um **serial version ID**;Adicionar a anotação
3. **@SuppressWarnings("serial")**;

Na verdade, são mais opções. Veja:



... mas, as que nos interessam são as 3 que comentei.

Na primeira opção, o Eclipse simplesmente atribui o valor **1L**.

A segunda vou falar por último... Na terceira você diz ao Eclipse que assume o risco de não ter um **serialVersionUID.**

Quanto a segunda...

Muitos já devem ter utilizado ela e se perguntado:

De onde o Eclipse tira esse número doido?

Eu já. :)

Acontece que o número gerado pelo Eclipse **não é aleatório**.

Como existe um padrão sobre quais elementos da classe relevar e sobre o algoritmo utilizado para gerar o hash, então o Eclipse tira proveito disso para gerar esse número pra você, caso queira.

Ou seja, o número doido que o Eclipse gera tem uma lógica.

Utilizando a ferramenta serialver do Java

A ferramenta **serialver** que vem no JDK nos ajuda a descobrir o valor de **serialVersionUID**, independente dele ter sido gerado implicitamente ou explicitamente.

Veremos adiante que isso pode evitar algumas dores de cabeça.

Primeiro, vamos ver como utilizá-la.

Observe a classe **Clube.java** (clube de futebol), como abaixo:

```
1  package com.algaworks.serialversionuid.exemplo1;
2
3  public class Clube implements java.io.Serializab
```

Para descobrir o **serialVersionUID** dessa classe podemos utilizar:

```
$ serialver -classpath :target/artigo-java-serialVersionUID-1.0-SNAPS

com.algaworks.serialversionuid.exemplo1.Clube: private static final l
```

Talvez você esteja se perguntando:

"Por que eu precisaria saber qual o serialVersionUID que o Java atribuiu para minha classe?"

Isso é útil sim. Tem até um caso que aconteceu dentro do próprio Java que ainda quero mostrar pra você.

Contextualizando melhor a necessidade dessa ferramenta...

Imagine o Pedro. No sistema que ele está desenvolvendo para controlar o campeonato de futebol da cidade dele, existe a classe:

```
1  package com.algaworks.serialversionuid.exemplo2;
2
3  public class Clube implements java.io.Serializab
4
5      String nome;
6
7  }
```

Repare que ele NÃO declarou o **serialVersionUID**.

Tempos depois, ele precisou adicionar um atributo na classe **Clube** que já possuía objetos serializados e persistidos em disco.

Veja como a classe ficou:

```
1  package com.algaworks.serialversionuid.exemplo2;
2
3  public class Clube implements java.io.Serializab
4
5      String nome;
6      int titulos; // atributo novo
7
8  }
```

Ao tentar desserializar os objetos, que foram serializados com a versão antiga, ele recebe a exceção **java.io.InvalidClassException**.

Na verdade, esse é até o comportamento correto. Afinal de contas, o Pedro alterou a classe.

Mas, pela regra de negócio, pode acontecer da alteração feita na classe não ter impacto negativo no sistema.

Por exemplo, se você simplesmente coloca um atributo novo na classe, então não precisa ignorar os objetos serializados com a versão antiga. É possível reaproveitá-los.

Como o Pedro é um garoto esperto, ele utilizou a ferramenta **serialver** para descobrir qual o **serialVersionUID** da primeira versão da classe que foi computado pela JVM implicitamente.

Foi simplesmente rodar o comando:

```
$ serialver -classpath :target/artigo-java-serialVersionUID-1.0-SNAPS

com.algaworks.serialversionuid.exemplo2.Clube: private static final l
```

...apontando para o **.class** da versão antiga.

Descoberto o valor de **serialVersionUID** na versão antiga da classe, então, basta declará-lo explicitamente na nova versão para conseguir desserializar os objetos mais antigos:

```
1  package com.algaworks.serialversionuid.exemplo2
2
3  public class Clube implements java.io.Serializa
4
5      private static final long serialVersionUID
6
7      String nome;
8      int titulos;
9
10 }
```

Sobre o caso que aconteceu dentro do Java... Foi na evolução do

Java 1.3 para o 1.4.

Dentre as alterações, existiu uma pequena na classe **java.text.AttributedCharacterIterator.Attribute** que acabou impactando na geração automática do atributo **serialVersionUID** e isso gerou um bug.

Ou seja, um objeto dessa classe serializado com a versão Java 1.3 não poderia ser **desserializado** na versão 1.4 e vice-versa.

A solução para a classe **java.text.AttributedCharacterIterator.Attribute** foi a mesma adotada pelo Pedro.

Eles descobriram o valor de **serialVersionUID** na versão do Java 1.3 e declararam explicitamente na versão Java 1.4.

Omitir ou informar

Dentro da própria especificação existe uma nota recomendando que os desenvolvedores declarem a propriedade explicitamente.

Essa recomendação pode evitar problemas inesperados de **desserialização**.

Como cheguei a mencionar, quando incluímos um atributo em uma classe, não precisamos descartar os objetos **desserializados** com a versão antiga.

Por essas e outras que informar o **serialVersionUID** explicitamente pode evitar problemas em tempo de execução.

Quando alterá-lo?

Você pode e até deve alterá-lo quando a versão antiga não faz mais sentido ou quando não dá para manter a compatibilidade.

É possível colocar um novo número escolhido arbitrariamente ou deixar o Eclipse gerar ele de novo pra você.

Serializando e desserializando

Para percebermos isso na prática, veremos um pequeno exemplo de como serializar e **desserializar** um objeto.

Vamos trabalhar com a seguinte classe:

```
1  public class Clube implements Serializable {
2
3      private static final long serialVersionUID
4
5      String nome;
6      int titulos;
7      LocalDate nascimento;
8
9      //getters e setters omitidos
10
11 }
```

Para serializar utilizaremos uma classe como abaixo:

```
1  public class Serializador {
2
3      public static void main(String... args) thr
4          Clube oMelhorClube = new Clube();
5          oMelhorClube.setNome("São Paulo Futebol
```

```
6           oMelhorClube.setTitulos(2147483647);//p
7           oMelhorClube.setNascimento(LocalDate.of
8
9           FileOutputStream fos = new FileOutputSt
10          ObjectOutputStream oos = new ObjectOutp
11          oos.writeObject(oMelhorClube);
12          oos.close();
13
14          System.out.println("Pronto! Objeto seri
15      }
16  }
```

...e outra desserializadora:

```
1   public class Desserializador {
2
3       public static void main(String... args) thr
4           FileInputStream fis = new FileInputStre
5           ObjectInputStream ois = new ObjectInput
6           Clube clube = (Clube) ois.readObject();
7           ois.close();
8
9           System.out.println("Pronto! Objeto dess
10          System.out.println("Nome: " + clube.get
11      }
12  }
```

Testando com o mesmo *serialVersionUID*

Com o mesmo número o resultado seria:

```
$ java -classpath :target/artigo-java-serialVersionUID-1.0-SNAPSHOT.j
Pronto! Objeto serializado.

$ java -classpath :target/artigo-java-serialVersionUID-1.0-SNAPSHOT.j
Pronto! Objeto desserializado.
Nome: São Paulo Futebol Clube
```

Testando com números diferentes

Já com **serialVersionUID** diferentes o resultado seria a exceção:

```
$ java -classpath :target/artigo-java-serialVersionUID-1.0-SNAPSHOT.j
Exception in thread "main" java.io.InvalidClassException: com.algawor
  at java.io.ObjectStreamClass.initNonProxy(ObjectStreamClass.java:61
  at java.io.ObjectInputStream.readNonProxyDesc(ObjectInputStream.jav
  at java.io.ObjectInputStream.readClassDesc(ObjectInputStream.java:1
  at java.io.ObjectInputStream.readOrdinaryObject(ObjectInputStream.j
  at java.io.ObjectInputStream.readObject0(ObjectInputStream.java:135
  at java.io.ObjectInputStream.readObject(ObjectInputStream.java:373)
  at com.algaworks.serialversionuid.exemplo3.Desserializador.main(Des
```

Testando com o mesmo número e com tipo diferente

Nesse último teste vou alterar voltar o **serialVersionUID** para **1L** e alterar o tipo do atributo **nome** para **int**.

Veja o resultado:

```
$ java -classpath :target/artigo-java-serialVersionUID-1.0-SNAPSHOT.j
Exception in thread "main" java.io.InvalidClassException: com.algawor
  at java.io.ObjectStreamClass.matchFields(ObjectStreamClass.java:229
  at java.io.ObjectStreamClass.getReflector(ObjectStreamClass.java:21
  at java.io.ObjectStreamClass.initNonProxy(ObjectStreamClass.java:66
  at java.io.ObjectInputStream.readNonProxyDesc(ObjectInputStream.jav
  at java.io.ObjectInputStream.readClassDesc(ObjectInputStream.java:1
  at java.io.ObjectInputStream.readOrdinaryObject(ObjectInputStream.j
  at java.io.ObjectInputStream.readObject0(ObjectInputStream.java:135
  at java.io.ObjectInputStream.readObject(ObjectInputStream.java:373)
  at com.algaworks.serialversionuid.exemplo3.Desserializador.main(Des
```

Atributos transientes

Quando você não desejar que um atributo seja serializado juntamente com o resto do objeto, você adiciona o modificador *transient*:

```
1  public class Clube implements Serializable {
2
3      private static final long serialVersionUID
4
5      String nome;
6      int titulos;
7      LocalDate nascimento;
8
9      transient String hino;//nao sera serializac
10
11     //getters e setters omitidos
12
13 }
```

Repare o atributo **hino** na classe acima... No momento da serialização do objeto, será ignorado.

Quem usa **JPA** já deve conhecer esse conceito através da anotação **javax.persistence.Transient.** Ela tem um objetivo parecido com o modificador **transient.**

Customizando a serialização

Algumas vezes podemos querer algum tipo de customização no momento de serializar e/ou desserializar nossos objetos.

Podemos incluir esse comportamento extra quando implementamos esses dois métodos em nossa classe:

```
private void writeObject(java.io.ObjectOutputStr
    throws IOException;

private void readObject(java.io.ObjectInputStrea
    throws IOException, ClassNotFoundException;
```

Veja como eles ficariam dentro da nossa classe `Clube`:

```
public class Clube implements Serializable {

    private static final long serialVersionUID

    String nome;
    int titulos;
    LocalDate nascimento;

    transient String hino;

    private void writeObject(ObjectOutputStream
        out.defaultWriteObject();

        out.writeObject(hino);//hino eh um atri

        System.out.println("Serializacao custom
    }

    private void readObject(ObjectInputStream i
        in.defaultReadObject();

        hino = (String) in.readObject();

        System.out.println("Desserializacao cus
    }

    //getters e setters omitidos

}
```

Nesse caso o único comportamento adicionado foi a serialização adicional do atributo *hino*.

Mas, **writeObject e readObject** são métodos comuns e você pode adicionar o comportamento que desejar dentro deles.

Você pode testar essa versão com customização da mesma forma que fizemos da primeira vez. Inclusive com as mesmas classes serializadora e desserializadora.

Conclusão: Serialização de objetos Java e a variável serialVersionUID

Apesar de não ser um recurso que precisamos programar com frequência, é importante sabermos como funciona.

Vimos **para que serve o serialVersionUID, quando atribuir e alterar os valores dele** e espero que tenha sido útil pra você.

Vai ser bacana ler um comentário seu aí embaixo com sua opinião, crítica ou elogio.

Tenho certeza que as informações sobre serialização de objetos são importantes para quem deseja carreira como desenvolvedor Java.

Caso esteja comprometido a seguir em frente, então quero recomendar o nosso curso de Java e Orientação A Objetos.