|  |  |
| --- | --- |
| http://www.linhadecodigo.com.br/artigos/img_artigos/EvertonCoimbraAraujo/Serializazao_Objetos/imagem001.jpg | Neste artigo apresento o Gustavo Rafael Valiati, que pode ser contatado pelo e-mail gustavovaliati@gmail.com. O Gustavo foi meu aluno em algumas disciplinas de sua gradução e está concluíndo o último período do curso de Tecnologia em Análise e Desenvolvimento de Sistemas da UTFPR campus Medianeira, neste primeiro semestre de 2012. |

**1. INTRODUÇÃO**

Nem sempre se deseja ter objetos ou estrutura de dados de um aplicativo, com um ciclo de vida que dura no máximo o tempo em que for armazenado em memória RAM. Ou seja, algumas vezes deseja-se guardá-los por um tempo indeterminado, sendo possível recuperá-los posteriormente. E neste caso, o termo “persistência” tem seu conceito definido.

Persistir dados, requer que estes possam ser serializáveis. A serialização possibilitará capturar o estado do objeto ou a estrutura de dados, e transformar em uma cadeia de bytes quando for necessário. Torna-se possível também, recuperar os bytes persistidos para fazer o processo inverso, e ter os dados de volta para a aplicação em execução.

Desta maneira, qualquer forma de dados que esteja sendo trabalhada na aplicação, e esta puder ser serializada, poderá ser passada para o meio físico, e ter o processo revertido.

A serialização não se limita apenas em possibilitar a gravação do dado em disco, mas permite que o mesmo seja transmitido por rede. Isto porquê a serialização cria um stream de bytes, que é requisito para transmissão de dados por rede.

Diversas linguagens possuem suporte à serialização, como: Java, C, C++, Python, PHP, .NET, entre outras. Neste artigo, será tratado apenas da serialização com Java, em sua implemetação básica, alguns problemas e soluções, com códigos para exemplificação.

**2. SERIALIZAÇÃO COM JAVA**

Serializar um objeto, em Java, só é possível caso sua classe esteja marcada como serializável. Essa marcação é feita com a simples implementação da interface “java.io.Serializable”. Essa interface não possui métodos para se implementar e nem atributos. Ainda não existiam as anotações no Java 1.1, versão que trouxe pela primeira vez a serialização ao Java, e desta maneira a implementação da interface era a melhor maneira de marcar uma classe.

A herança entre classes é naturalmente afetada pela serialização. Já que quando uma classe implementa a interface “Serializable”, toda classe que dela estender também estará implicitamente marcada como serializável

.

Para exemplo, será criada uma classe “pojo.Fruta” que contém dois atributos: nome e cor. No código exemplo (Listagem 1), pode-se notar a classe “Fruta” marcada para a serialização:

Listagem 1 - Marcando uma classe como serializável.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | import java.io.Serializable;    public class Fruta implements Serializable{      private String nome;      private String cor;        public Fruta(String nome, String cor) {          this.nome = nome;          this.cor = cor;      }        public String getCor() { return cor; }      public String getNome() { return nome; }      public void setCor(String cor) { this.cor = cor; }      public void setNome(String nome) { this.nome = nome; }  } |

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | import operacoes.Deserializador;  import operacoes.Serializador;  import pojo.Fruta;    public class Principal {      public static void main(String args[]){         //serializa         Serializador s = new Serializador();         Fruta fruta = new Fruta("Maça", "Vermelha");          try {              s.serializar("/home/gustavo/fruta", fruta);          } catch (Exception ex) {              System.err.println("Falha ao serializar! - " + ex.toString());          }            //deserializa          Deserializador d = new Deserializador();          fruta = null;          try {              fruta = (Fruta) d.deserializar("/home/gustavo/fruta");          } catch (Exception ex) {              System.err.println("Falha ao deserializar! - " + ex.toString());          }          System.out.println(fruta.getNome() + " - " + fruta.getCor());      }  } |

Na Listagem 5, encontra-se um exemplo de serialização e deserialização de um ArrayList da classe Fruta. É apenas mais um exemplo que demonstra a capacidade de se persistir qualquer objeto que seja serializável. Essa serialização só foi possível porque a classe “ArrayList” implementa a interface “Serializable” por padrão.

Listagem 5 - Serialização de ArrayList.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | //serialização e deserialização com ArrayList          ArrayList<Fruta> frutas = new ArrayList<Fruta>();          Fruta f1 = new Fruta("Laranja", "Amarela");          Fruta f2 = new Fruta("Abacate", "Verde");          Fruta f3 = new Fruta("Morango", "Vermelho");          frutas.add(f1);          frutas.add(f2);          frutas.add(f3);            try {              s.serializar("/home/gustavo/frutas", frutas);              frutas = null;              frutas = (ArrayList<Fruta>) d.deserializar("/home/gustavo/frutas");              for(Fruta f : frutas){                  System.out.println("ArrayList: " + f.getNome() + " - " + f.getCor());              }          } catch (Exception ex) {              System.err.println("Falha ao serializar ou deserializar! - " +                                 ex.toString());          } |

**4. PARTICULARIDADES DA SERIALIZAÇÃO**

Como dito, a serialização de um objeto em Java pode ser algo fácil de se fazer. Mas existem certas informações que são importantes e sabê-las pode evitar vários problemas.

Um dos problemas é que nem tudo pode ser serializado, como objetos das classes “Thread” e “Socket” e atributos estáticos. Os atributos estáticos estão vinculados à classe e não ao objeto. Assim quando se trata do estado do objeto, os campos estáticos não são considerados. Em relação às Threads, suas instâncias só são úteis enquanto estiverem em execução na JVM, e podem ser paradas/destruídas e iniciadas com instruções de sua classe. Não existe algo no estado destas instâncias que seja útil para se persistir, e por isso não há sentido em persistir tais instâncias para restaurar posteriormente.

Mas e se existir alguma classe em que seja necessário persisti-la, e esta possui uma instância da classe “Thread”? Neste caso, utilizando um recurso pode-se persistir a classe inteira, menos a instância thread. Este recurso, refere-se em indicar que tal instância thread é um objeto “Transient”. Na Listagem 6 encontra-se um exemplo da implementação com uso de “Transient”.

Listagem 6 - Uso do Transient

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | import java.io.Serializable;    public class ImplementacaoTransient implements Serializable, Runnable{        transient private Thread thread;      //marcado com "transient" para que sua instância não seja serializada        //Demais valores de estado serão serializados      private int valorParaPersistir;        public ImplementacaoTransient(int valorParaPersistir){          this.valorParaPersistir = valorParaPersistir;          thread = new Thread(this);          thread.start();      }        public void run()      {          while(true){              //código a ser executado pela Thread          }      }  } |

Vale ressaltar que, o que é serializado é apenas o estado do objeto, e não os métodos e a classe. No exemplo acima, apenas a instância do objeto “thread” e do objeto “valorParaPersistir” compunham o estado do objeto. Assim, evita-se que o sistema de serialização tente serializar um objeto não-serializável e gere erros. Após a deserialização de um objeto da classe “ImplementacaoTransient” este estará totalmente funcional.

Um outro problema que pode vir a ocorrer, se refere ao versionamento das classes. Toda classe possui um atributo oculto chamado “serialVersionUID” com um valor gerado automaticamente. Objetos criados levam consigo o “serialVersionUID” de sua classe, que indica sua versão. Toda vez que uma classe sofre alterações em seu código, um novo “serialVersionUID” é gerado. O problema com a serialização surge quando serializa-se o estado de um objeto e posteriormente tenta-se restaurar o estado para a aplicação, mas houveram mudanças no código da classe. Neste caso, uma exceção “java.io.InvalidClassException” é disparada, pois de fato, o estado do objeto corresponde a uma versão de classe do qual foi criado que já não existe mais. Portanto, isso só ocorre para evitar erros de programação, em que tenta-se restaurar o estado de um objeto para uma classe que possui versão diferente da qual o objeto foi serializado inicialmente. E isso pode significar diferentes versões de uma mesma classe “Fruta” por exemplo, ou diferentes versões por classes serem totalmente difere

|  |  |
| --- | --- |
| 1 | serialver pojo.Fruta |

A serialização acaba inflingindo alguns modelos de segurança em seu procedimento padrão, e criando então um novo problema. Quando um estado de objeto é persistido no sistema de arquivos ou enviado por rede, o fluxo binário pode ser detectado, analisado e obtidas informações sobre a classe e o estado do objeto. Por exemplo, se utilizado um editor de texto qualquer para abrir o arquivo persistido do estado do objeto da classe “Fruta” dos exemplos anteriores, é comum os atributos e outras informações aparecerem como um simples texto, total ou parcilamente legíveis, assim como demonstrado na Listagem 9.

Listagem 9 - Exemplo de leitura de arquivo binário de um estado de objeto persistido.

|  |  |
| --- | --- |
| 1  2 | ��##sr#  pojo.Fruta#######{###L##cort##Ljava/lang/String;L##nomeq#~##xpt##Vermelhat##Maça |

Com uma ferramenta menos simples seria possível aumentar a quantidade de informações legíveis, coletando uma maior quantia de dados.

No entando, o processo de serialização permite que haja uma certa personalização da serialização e deserialização de um objeto. Isso deve-se a possibilidade da sobrescrição dos métodos “writeObject” e “readObject” dentro da classe a ser serializada, como a classe exemplar “pojo.Fruta”. Esta sobrescrita permite um controle total sobre a serialização, pois as classes “java.io.ObjectOutputStream” e “java.io.ObjectInputStream”, ao serializar o estado de um objeto, primeiramente verificam se esse método foi sobrescrito na classe alvo e assim o utilizam, senão, será utilizado o protocolo padrão de serialização.

Realizando a sobrescrição dos métodos é possível criptografar os dados na serialização e descriptografar os dados na deserialização, por meio de algum algoritmo que o programador venha implementar. Assim os dados podem ser persistidos ou transmitidos por rede, sem que seja possível ler as partes mais importantes do estado de um objeto. Na listagem 10, a classe “pojo.Fruta” foi modificada dando origem a classe “pojo.FrutaSobrescrita” que possui adicionamente a sobrescrição dos métodos “writeObject” e “readObject”. Como este é apenas um exemplo, não foi escrito um verdadeiro algoritmo de criptografia, mas sim apenas uma troca no valor da String do campo “cor”, identificando que houveram alterações tanto no processo de serialização quanto no de deserialização. Na listagem 11 encontra-se a execução da serialização.

Listagem 10 - Sobrescrita dos métodos “writeObject” e “readObject”.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36 | import java.io.Serializable;    public class FrutaSobrescrita implements Serializable{      private String nome;      private String cor;      static final long serialVersionUID = 123L; //indica a versao da classe        public FrutaSobrescrita(String nome, String cor) {          this.nome = nome;          this.cor = cor;      }      public String getCor() { return cor; }      public String getNome() { return nome; }      public void setCor(String cor) { this.cor = cor; }      public void setNome(String nome) { this.nome = nome; }        private void writeObject(java.io.ObjectOutputStream stream) throws java.io.IOException      {          // criptografar os dados antes de liberar o stream          // EXEMPLO: setCor(criptografar(getCor()));          setCor(getCor()+ " > CRIPTOGRAFADO"); // apenas concatenação de string para indicar alteração            // liberar o stream          stream.defaultWriteObject();      }        private void readObject(java.io.ObjectInputStream stream) throws java.io.IOException, ClassNotFoundException      {          // liberar stream          stream.defaultReadObject();            // descriptografar os dados          // EXEMPLO: setCor(descriptografar(getCor()));          setCor(getCor()+ " > DESCRIPTOGRAFADO"); // apenas contatenação de string para indicar alteração      }  } |

Listagem 11 - Execução da serialização da classe “pojo.FrutaSobrescrita”.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | //serializa        Serializador s = new Serializador();        FrutaSobrescrita frutaSobrescrita = new FrutaSobrescrita("Maça", "Vermelha");         try {             s.serializar("/home/gustavo/frutaSobrescrita", frutaSobrescrita);         } catch (Exception ex) {             System.err.println("Falha ao serializar! - " + ex.toString());         }           //deserializa        Deserializador d = new Deserializador();         frutaSobrescrita = null;         try {             frutaSobrescrita = (FrutaSobrescrita) d.deserializar("/home/gustavo/frutaSobrescrita");         } catch (Exception ex) {             System.err.println("Falha ao deserializar! - " + ex.toString());         }         System.out.println("Sobrescrita: " + frutaSobrescrita.getNome() + " - " + frutaSobrescrita.getCor()); |

**5. CONCLUSÃO**

A serialização é um procedimento de simples execução por parte do programador. Com pouco tempo de codificação é possível implementar a persistencia de um estado de objeto para o sistema de arquivos ou de transmissão por rede. É possível reproduzir os exemplos criados neste artigo, em situações até mesmo mais elaboradas. A apresentação de alguns problemas e soluções trazidas por particularidades, abre o entendimento de que a serialização pode deixar de ser simples conforme for aplicada em diferentes contextos, e que ao mesmo tempo é muito flexível ao permitir ao programador implementar diferentes soluções.

[Assista ao vídeo Serialização/Deserialização de objetos com YAML](http://www.devmedia.com.br/serializacao-deserializacao-de-objetos-com-yaml-parte-1/8399)

**6. BIBLIOGRAFIA**

ORACLE - **java.io Interface Serializable** - Disponível em: <http://docs.oracle.com/javase/1.4.2/docs/api/java/io/Serializable.html>Acesso em: 11/03/2012

ORACLE - **Discover de Secrets of Java Serialization API** - Disponível em: <http://java.sun.com/developer/technicalArticles/Programming/serialization/> Acesso em: 11/03/2012

NEWARD, Ted - **5 coisas que você não sabia sobre... Serialização de Objetos Java** - Disponível em: <http://www.ibm.com/developerworks/br/library/j-5things1/index.html> Acesso em: 11/03/2012

DOWNS, Andrew - **Java Serialization** - Disponível em: <http://www.mactech.com/articles/mactech/Vol.14/14.04/JavaSerialization/index.html> Acesso em: 11/03/2012

PREISSLER, Luciano E.; GEYER, Cláudio F. R. - **Serialização em Java** - Disponível em: <http://www.inf.ufrgs.br/gppd/disc/cmp167/trabalhos/sem99-1/T1/luciano/final.htm> Acesso em: 11/03/2012

[[](http://www.linhadecodigo.com.br/colaborador.aspx?id=354)](http://www.linhadecodigo.com.br/colaborador.aspx?id=354)

[Everton Coimbra de Araújo - Desde 1987 atua na área de treinamento e desenvolvimento. Como Mestre em Ciência da Computação, é professor da UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ, Campus Medianeira, onde leciona disciplinas relacionadas ao desenvolvimento de aplicações web, com Java e .NET.   
  
É autor dos livros Desenvolvimento para WEB com Java, Orientação a Objetos com Java - Simples, Fácil e Eficiente, Algoritmo - Fundamento e Prática em sua terceira edição e dos livros Delphi - Implementação e Técnicas para Ambientes Virtuais e C++ Builder - Implementação e Técnicas para Ambientes Virtuais. Todos pela VisualBooks. Pode ser contactado através do e-mail everton@utfpr.edu.br ou evertoncoimbra@gmail.com.](http://www.linhadecodigo.com.br/colaborador.aspx?id=354)

 85 3 0 4

**Leia também**

* [Conhecendo a interface Map do Java](http://www.linhadecodigo.com.br/artigo/3670/conhecendo-a-interface-map-do-java.aspx)*Java*
* [Trabalhando com a Interface Set no Java](http://www.linhadecodigo.com.br/artigo/3669/trabalhando-com-a-interface-set-no-java.aspx)*Java*
* [Classes Wrappers em Java](http://www.linhadecodigo.com.br/artigo/3667/classes-wrappers-em-java.aspx)*Java*
* [ComboBoxModel em Java](http://www.linhadecodigo.com.br/artigo/3664/comboboxmodel-em-java.aspx)*Java*
* [Reduzindo a Mutabilidade em Java](http://www.linhadecodigo.com.br/artigo/3659/reduzindo-a-mutabilidade-em-java.aspx)*Java*

Publicidade

[](http://qconsp.com/?utm_source=devmedia&utm_medium=banner&utm_campaign=devmedia)

Linha de Código[Anuncie](http://www.web-03.net/linhadecodigo.asp)|[Fale Conosco](mailto:colabore@linhadecodigo.com.br)|[Publique](http://www.linhadecodigo.com.br/publique.aspx)

Copyright 2016 - todos os direitos reservados para [**Web03**](http://www.web-03.net/)

|  |  |
| --- | --- |
| http://www.linhadecodigo.com.br/artigos/img_artigos/EvertonCoimbraAraujo/Serializazao_Objetos/imagem001.jpg | Neste artigo apresento o Gustavo Rafael Valiati, que pode ser contatado pelo e-mail gustavovaliati@gmail.com. O Gustavo foi meu aluno em algumas disciplinas de sua gradução e está concluíndo o último período do curso de Tecnologia em Análise e Desenvolvimento de Sistemas da UTFPR campus Medianeira, neste primeiro semestre de 2012. |

**1. INTRODUÇÃO**

Nem sempre se deseja ter objetos ou estrutura de dados de um aplicativo, com um ciclo de vida que dura no máximo o tempo em que for armazenado em memória RAM. Ou seja, algumas vezes deseja-se guardá-los por um tempo indeterminado, sendo possível recuperá-los posteriormente. E neste caso, o termo “persistência” tem seu conceito definido.

Persistir dados, requer que estes possam ser serializáveis. A serialização possibilitará capturar o estado do objeto ou a estrutura de dados, e transformar em uma cadeia de bytes quando for necessário. Torna-se possível também, recuperar os bytes persistidos para fazer o processo inverso, e ter os dados de volta para a aplicação em execução.

Desta maneira, qualquer forma de dados que esteja sendo trabalhada na aplicação, e esta puder ser serializada, poderá ser passada para o meio físico, e ter o processo revertido.

A serialização não se limita apenas em possibilitar a gravação do dado em disco, mas permite que o mesmo seja transmitido por rede. Isto porquê a serialização cria um stream de bytes, que é requisito para transmissão de dados por rede.

Diversas linguagens possuem suporte à serialização, como: Java, C, C++, Python, PHP, .NET, entre outras. Neste artigo, será tratado apenas da serialização com Java, em sua implemetação básica, alguns problemas e soluções, com códigos para exemplificação.

**2. SERIALIZAÇÃO COM JAVA**

Serializar um objeto, em Java, só é possível caso sua classe esteja marcada como serializável. Essa marcação é feita com a simples implementação da interface “java.io.Serializable”. Essa interface não possui métodos para se implementar e nem atributos. Ainda não existiam as anotações no Java 1.1, versão que trouxe pela primeira vez a serialização ao Java, e desta maneira a implementação da interface era a melhor maneira de marcar uma classe.

A herança entre classes é naturalmente afetada pela serialização. Já que quando uma classe implementa a interface “Serializable”, toda classe que dela estender também estará implicitamente marcada como serializável

.

Para exemplo, será criada uma classe “pojo.Fruta” que contém dois atributos: nome e cor. No código exemplo (Listagem 1), pode-se notar a classe “Fruta” marcada para a serialização:

Listagem 1 - Marcando uma classe como serializável.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | import java.io.Serializable;    public class Fruta implements Serializable{      private String nome;      private String cor;        public Fruta(String nome, String cor) {          this.nome = nome;          this.cor = cor;      }        public String getCor() { return cor; }      public String getNome() { return nome; }      public void setCor(String cor) { this.cor = cor; }      public void setNome(String nome) { this.nome = nome; }  } |

Pode-se dizer que a implementação do processo de serialização e deserialização simples de um objeto, não possui um código difícil de se escrever. Isto porquê duas classes fazem a maior parte do trabalho, de maneira que com poucas linhas de código pode-se transformar um objeto em bytes e vice-versa. Essas duas classes são “java.io.ObjectInputStream” que recebe bytes para criar um objeto e “java.io.ObjectOutputStream” que cria uma cadeia de bytes a partir de um objeto.

**3. SERIALIZANDO E DESERIALIZANDO O ESTADO DE UM OBJETO**

Como exemplo, será persistido um objeto em disco. Para isto, além das classes “java.io.ObjectInputStream” e “java.io.ObjectOutputStream”, serão utilizadas duas outras que trabalham com arquivos, que são “java.io.FileInputStream” e “java.io.FileOutputStream”, que seguem a mesma lógica de InputStream e OutputStream, só que agora relacionado ao sistema de arquivos e não aos objetos.

Para serialização, deve-se criar um novo “java.io.FileOutputStream”, com o caminho (“path”) do arquivo onde se deseja armazenar o estado do objeto. Também deve ser criado um novo “java.io.ObjectOutputStream” que está ligado ao arquivo. E por último deve-se executar a escrita do objeto alvo, para dentro do objeto de stream que por sua vez está ligado ao arquivo destino. Após executados esses passos, é possível encontrar no sistema de arquivos um novo arquivo que contém os bytes correspondentes à descrição do estado do objeto. Na Listagem 2, encontra-se o código exemplo.

Listagem 2 - Serialização de um objeto.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | import java.io.FileOutputStream;  import java.io.ObjectOutputStream;    public class Serializador {        public Serializador() {    }        public void serializar(String path, Object obj) throws Exception {              FileOutputStream outFile = new FileOutputStream(path);      ObjectOutputStream s = new ObjectOutputStream(outFile);      s.writeObject(obj);      s.close();      }  } |

E para a deserialização, o processo é extremamente parecido com a serialização. Cria-se um novo “java.io.FileInputStream” relacionado ao arquivo onde o estado do objeto encontra-se persistido. Deve-se criar também um novo “java.io.ObjectInputStream” que estará vinculado ao arquivo de origem. Por último, executa-se a leitura do arquivo para a restauração do estado. Na Listagem 3 encontra-se o exemplo do código.

Listagem 3 - Deserializaçãode um objeto.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | import java.io.FileInputStream;  import java.io.ObjectInputStream;    public class Deserializador {        public Deserializador() {    }        public Object deserializar(String path) throws Exception {      FileInputStream inFile = new FileInputStream(path);      ObjectInputStream d = new ObjectInputStream(inFile);      Object o = d.readObject();      d.close();      return o;      }  } |

A Listagem 4, contém o exemplo da execução da serialização e em seguida da deserialização.

Listagem 4 - Serializando e Deserializando

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | import operacoes.Deserializador;  import operacoes.Serializador;  import pojo.Fruta;    public class Principal {      public static void main(String args[]){         //serializa         Serializador s = new Serializador();         Fruta fruta = new Fruta("Maça", "Vermelha");          try {              s.serializar("/home/gustavo/fruta", fruta);          } catch (Exception ex) {              System.err.println("Falha ao serializar! - " + ex.toString());          }            //deserializa          Deserializador d = new Deserializador();          fruta = null;          try {              fruta = (Fruta) d.deserializar("/home/gustavo/fruta");          } catch (Exception ex) {              System.err.println("Falha ao deserializar! - " + ex.toString());          }          System.out.println(fruta.getNome() + " - " + fruta.getCor());      }  } |

Na Listagem 5, encontra-se um exemplo de serialização e deserialização de um ArrayList da classe Fruta. É apenas mais um exemplo que demonstra a capacidade de se persistir qualquer objeto que seja serializável. Essa serialização só foi possível porque a classe “ArrayList” implementa a interface “Serializable” por padrão.

Listagem 5 - Serialização de ArrayList.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | //serialização e deserialização com ArrayList          ArrayList<Fruta> frutas = new ArrayList<Fruta>();          Fruta f1 = new Fruta("Laranja", "Amarela");          Fruta f2 = new Fruta("Abacate", "Verde");          Fruta f3 = new Fruta("Morango", "Vermelho");          frutas.add(f1);          frutas.add(f2);          frutas.add(f3);            try {              s.serializar("/home/gustavo/frutas", frutas);              frutas = null;              frutas = (ArrayList<Fruta>) d.deserializar("/home/gustavo/frutas");              for(Fruta f : frutas){                  System.out.println("ArrayList: " + f.getNome() + " - " + f.getCor());              }          } catch (Exception ex) {              System.err.println("Falha ao serializar ou deserializar! - " +                                 ex.toString());          } |

**4. PARTICULARIDADES DA SERIALIZAÇÃO**

Como dito, a serialização de um objeto em Java pode ser algo fácil de se fazer. Mas existem certas informações que são importantes e sabê-las pode evitar vários problemas.

Um dos problemas é que nem tudo pode ser serializado, como objetos das classes “Thread” e “Socket” e atributos estáticos. Os atributos estáticos estão vinculados à classe e não ao objeto. Assim quando se trata do estado do objeto, os campos estáticos não são considerados. Em relação às Threads, suas instâncias só são úteis enquanto estiverem em execução na JVM, e podem ser paradas/destruídas e iniciadas com instruções de sua classe. Não existe algo no estado destas instâncias que seja útil para se persistir, e por isso não há sentido em persistir tais instâncias para restaurar posteriormente.

Mas e se existir alguma classe em que seja necessário persisti-la, e esta possui uma instância da classe “Thread”? Neste caso, utilizando um recurso pode-se persistir a classe inteira, menos a instância thread. Este recurso, refere-se em indicar que tal instância thread é um objeto “Transient”. Na Listagem 6 encontra-se um exemplo da implementação com uso de “Transient”.

Listagem 6 - Uso do Transient

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | import java.io.Serializable;    public class ImplementacaoTransient implements Serializable, Runnable{        transient private Thread thread;      //marcado com "transient" para que sua instância não seja serializada        //Demais valores de estado serão serializados      private int valorParaPersistir;        public ImplementacaoTransient(int valorParaPersistir){          this.valorParaPersistir = valorParaPersistir;          thread = new Thread(this);          thread.start();      }        public void run()      {          while(true){              //código a ser executado pela Thread          }      }  } |

Vale ressaltar que, o que é serializado é apenas o estado do objeto, e não os métodos e a classe. No exemplo acima, apenas a instância do objeto “thread” e do objeto “valorParaPersistir” compunham o estado do objeto. Assim, evita-se que o sistema de serialização tente serializar um objeto não-serializável e gere erros. Após a deserialização de um objeto da classe “ImplementacaoTransient” este estará totalmente funcional.

Um outro problema que pode vir a ocorrer, se refere ao versionamento das classes. Toda classe possui um atributo oculto chamado “serialVersionUID” com um valor gerado automaticamente. Objetos criados levam consigo o “serialVersionUID” de sua classe, que indica sua versão. Toda vez que uma classe sofre alterações em seu código, um novo “serialVersionUID” é gerado. O problema com a serialização surge quando serializa-se o estado de um objeto e posteriormente tenta-se restaurar o estado para a aplicação, mas houveram mudanças no código da classe. Neste caso, uma exceção “java.io.InvalidClassException” é disparada, pois de fato, o estado do objeto corresponde a uma versão de classe do qual foi criado que já não existe mais. Portanto, isso só ocorre para evitar erros de programação, em que tenta-se restaurar o estado de um objeto para uma classe que possui versão diferente da qual o objeto foi serializado inicialmente. E isso pode significar diferentes versões de uma mesma classe “Fruta” por exemplo, ou diferentes versões por classes serem totalmente diferentes como: o estado de um objeto da classe “Fruta” ser restaurado para classe “Carro”.

Mas quando é necessário, conscientemente, recuperar o estado de um objeto, mesmo que houveram alterações na classe do qual foi gerado, é possível controlar manualmente o “serialVersionUID”. Para isto, basta declarar explicitamente na forma de atributo estático e final, um valor Long qualquer para corresponder à versão da classe. E assim, mesmo que houverem modificações na classe, a versão permenecerá a mesma, evitando as exceções da incompatiblidade de classes. No entanto, vale ressaltar que o versionamento não é um impecílio para o programador, mas sim muito importante em casos que deseja-se manter um controle sobre a compatibilidade de código na aplicação, definindo versões para classes No código da Listagem 7 encontra-se o exemplo do controle manual do “serialVersionUID”.

Listagem 7 - Uso manual do serialVersionUID

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | import java.io.Serializable;    public class Fruta implements Serializable{      private String nome;      private String cor;      static final long serialVersionUID = 123L; //indica a versao da classe        public Fruta(String nome, String cor) {          this.nome = nome;          this.cor = cor;      }      public String getCor() { return cor; }      public String getNome() { return nome; }      public void setCor(String cor) { this.cor = cor; }      public void setNome(String nome) { this.nome = nome; }  } |

O JDK fornece uma ferramenta que permite visualizar a versão de qualquer classe Java. Basta no terminal do sistema operacional utilizar o comando “serialver” seguido da classe qualificada, conforme exemplo da Listagem 8.

Listagem 8 - Verificação da versão de classe

|  |  |
| --- | --- |
| 1 | serialver pojo.Fruta |

A serialização acaba inflingindo alguns modelos de segurança em seu procedimento padrão, e criando então um novo problema. Quando um estado de objeto é persistido no sistema de arquivos ou enviado por rede, o fluxo binário pode ser detectado, analisado e obtidas informações sobre a classe e o estado do objeto. Por exemplo, se utilizado um editor de texto qualquer para abrir o arquivo persistido do estado do objeto da classe “Fruta” dos exemplos anteriores, é comum os atributos e outras informações aparecerem como um simples texto, total ou parcilamente legíveis, assim como demonstrado na Listagem 9.

Listagem 9 - Exemplo de leitura de arquivo binário de um estado de objeto persistido.

|  |  |
| --- | --- |
| 1  2 | ��##sr#  pojo.Fruta#######{###L##cort##Ljava/lang/String;L##nomeq#~##xpt##Vermelhat##Maça |

Com uma ferramenta menos simples seria possível aumentar a quantidade de informações legíveis, coletando uma maior quantia de dados.

No entando, o processo de serialização permite que haja uma certa personalização da serialização e deserialização de um objeto. Isso deve-se a possibilidade da sobrescrição dos métodos “writeObject” e “readObject” dentro da classe a ser serializada, como a classe exemplar “pojo.Fruta”. Esta sobrescrita permite um controle total sobre a serialização, pois as classes “java.io.ObjectOutputStream” e “java.io.ObjectInputStream”, ao serializar o estado de um objeto, primeiramente verificam se esse método foi sobrescrito na classe alvo e assim o utilizam, senão, será utilizado o protocolo padrão de serialização.

Realizando a sobrescrição dos métodos é possível criptografar os dados na serialização e descriptografar os dados na deserialização, por meio de algum algoritmo que o programador venha implementar. Assim os dados podem ser persistidos ou transmitidos por rede, sem que seja possível ler as partes mais importantes do estado de um objeto. Na listagem 10, a classe “pojo.Fruta” foi modificada dando origem a classe “pojo.FrutaSobrescrita” que possui adicionamente a sobrescrição dos métodos “writeObject” e “readObject”. Como este é apenas um exemplo, não foi escrito um verdadeiro algoritmo de criptografia, mas sim apenas uma troca no valor da String do campo “cor”, identificando que houveram alterações tanto no processo de serialização quanto no de deserialização. Na listagem 11 encontra-se a execução da serialização.

Listagem 10 - Sobrescrita dos métodos “writeObject” e “readObject”.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36 | import java.io.Serializable;    public class FrutaSobrescrita implements Serializable{      private String nome;      private String cor;      static final long serialVersionUID = 123L; //indica a versao da classe        public FrutaSobrescrita(String nome, String cor) {          this.nome = nome;          this.cor = cor;      }      public String getCor() { return cor; }      public String getNome() { return nome; }      public void setCor(String cor) { this.cor = cor; }      public void setNome(String nome) { this.nome = nome; }        private void writeObject(java.io.ObjectOutputStream stream) throws java.io.IOException      {          // criptografar os dados antes de liberar o stream          // EXEMPLO: setCor(criptografar(getCor()));          setCor(getCor()+ " > CRIPTOGRAFADO"); // apenas concatenação de string para indicar alteração            // liberar o stream          stream.defaultWriteObject();      }        private void readObject(java.io.ObjectInputStream stream) throws java.io.IOException, ClassNotFoundException      {          // liberar stream          stream.defaultReadObject();            // descriptografar os dados          // EXEMPLO: setCor(descriptografar(getCor()));          setCor(getCor()+ " > DESCRIPTOGRAFADO"); // apenas contatenação de string para indicar alteração      }  } |

Listagem 11 - Execução da serialização da classe “pojo.FrutaSobrescrita”.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | //serializa        Serializador s = new Serializador();        FrutaSobrescrita frutaSobrescrita = new FrutaSobrescrita("Maça", "Vermelha");         try {             s.serializar("/home/gustavo/frutaSobrescrita", frutaSobrescrita);         } catch (Exception ex) {             System.err.println("Falha ao serializar! - " + ex.toString());         }           //deserializa        Deserializador d = new Deserializador();         frutaSobrescrita = null;         try {             frutaSobrescrita = (FrutaSobrescrita) d.deserializar("/home/gustavo/frutaSobrescrita");         } catch (Exception ex) {             System.err.println("Falha ao deserializar! - " + ex.toString());         }         System.out.println("Sobrescrita: " + frutaSobrescrita.getNome() + " - " + frutaSobrescrita.getCor()); |

**5. CONCLUSÃO**

A serialização é um procedimento de simples execução por parte do programador. Com pouco tempo de codificação é possível implementar a persistencia de um estado de objeto para o sistema de arquivos ou de transmissão por rede. É possível reproduzir os exemplos criados neste artigo, em situações até mesmo mais elaboradas. A apresentação de alguns problemas e soluções trazidas por particularidades, abre o entendimento de que a serialização pode deixar de ser simples conforme for aplicada em diferentes contextos, e que ao mesmo tempo é muito flexível ao permitir ao programador implementar diferentes soluções.

[Assista ao vídeo Serialização/Deserialização de objetos com YAML](http://www.devmedia.com.br/serializacao-deserializacao-de-objetos-com-yaml-parte-1/8399)

**6. BIBLIOGRAFIA**

ORACLE - **java.io Interface Serializable** - Disponível em: <http://docs.oracle.com/javase/1.4.2/docs/api/java/io/Serializable.html>Acesso em: 11/03/2012

ORACLE - **Discover de Secrets of Java Serialization API** - Disponível em: <http://java.sun.com/developer/technicalArticles/Programming/serialization/> Acesso em: 11/03/2012

NEWARD, Ted - **5 coisas que você não sabia sobre... Serialização de Objetos Java** - Disponível em: <http://www.ibm.com/developerworks/br/library/j-5things1/index.html> Acesso em: 11/03/2012

DOWNS, Andrew - **Java Serialization** - Disponível em: <http://www.mactech.com/articles/mactech/Vol.14/14.04/JavaSerialization/index.html> Acesso em: 11/03/2012

PREISSLER, Luciano E.; GEYER, Cláudio F. R. - **Serialização em Java** - Disponível em: <http://www.inf.ufrgs.br/gppd/disc/cmp167/trabalhos/sem99-1/T1/luciano/final.htm> Acesso em: 11/03/2012

[[](http://www.linhadecodigo.com.br/colaborador.aspx?id=354)](http://www.linhadecodigo.com.br/colaborador.aspx?id=354)

[Everton Coimbra de Araújo - Desde 1987 atua na área de treinamento e desenvolvimento. Como Mestre em Ciência da Computação, é professor da UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ, Campus Medianeira, onde leciona disciplinas relacionadas ao desenvolvimento de aplicações web, com Java e .NET.   
  
É autor dos livros Desenvolvimento para WEB com Java, Orientação a Objetos com Java - Simples, Fácil e Eficiente, Algoritmo - Fundamento e Prática em sua terceira edição e dos livros Delphi - Implementação e Técnicas para Ambientes Virtuais e C++ Builder - Implementação e Técnicas para Ambientes Virtuais. Todos pela VisualBooks. Pode ser contactado através do e-mail everton@utfpr.edu.br ou evertoncoimbra@gmail.com.](http://www.linhadecodigo.com.br/colaborador.aspx?id=354)

 85 3 0 4

**Leia também**

* [Conhecendo a interface Map do Java](http://www.linhadecodigo.com.br/artigo/3670/conhecendo-a-interface-map-do-java.aspx)*Java*
* [Trabalhando com a Interface Set no Java](http://www.linhadecodigo.com.br/artigo/3669/trabalhando-com-a-interface-set-no-java.aspx)*Java*
* [Classes Wrappers em Java](http://www.linhadecodigo.com.br/artigo/3667/classes-wrappers-em-java.aspx)*Java*
* [ComboBoxModel em Java](http://www.linhadecodigo.com.br/artigo/3664/comboboxmodel-em-java.aspx)*Java*
* [Reduzindo a Mutabilidade em Java](http://www.linhadecodigo.com.br/artigo/3659/reduzindo-a-mutabilidade-em-java.aspx)*Java*

Publicidade

[](http://qconsp.com/?utm_source=devmedia&utm_medium=banner&utm_campaign=devmedia)

Linha de Código[Anuncie](http://www.web-03.net/linhadecodigo.asp)|[Fale Conosco](mailto:colabore@linhadecodigo.com.br)|[Publique](http://www.linhadecodigo.com.br/publique.aspx)

|  |  |
| --- | --- |
| http://www.linhadecodigo.com.br/artigos/img_artigos/EvertonCoimbraAraujo/Serializazao_Objetos/imagem001.jpg | Neste artigo apresento o Gustavo Rafael Valiati, que pode ser contatado pelo e-mail gustavovaliati@gmail.com. O Gustavo foi meu aluno em algumas disciplinas de sua gradução e está concluíndo o último período do curso de Tecnologia em Análise e Desenvolvimento de Sistemas da UTFPR campus Medianeira, neste primeiro semestre de 2012. |

**1. INTRODUÇÃO**

Nem sempre se deseja ter objetos ou estrutura de dados de um aplicativo, com um ciclo de vida que dura no máximo o tempo em que for armazenado em memória RAM. Ou seja, algumas vezes deseja-se guardá-los por um tempo indeterminado, sendo possível recuperá-los posteriormente. E neste caso, o termo “persistência” tem seu conceito definido.

Persistir dados, requer que estes possam ser serializáveis. A serialização possibilitará capturar o estado do objeto ou a estrutura de dados, e transformar em uma cadeia de bytes quando for necessário. Torna-se possível também, recuperar os bytes persistidos para fazer o processo inverso, e ter os dados de volta para a aplicação em execução.

Desta maneira, qualquer forma de dados que esteja sendo trabalhada na aplicação, e esta puder ser serializada, poderá ser passada para o meio físico, e ter o processo revertido.

A serialização não se limita apenas em possibilitar a gravação do dado em disco, mas permite que o mesmo seja transmitido por rede. Isto porquê a serialização cria um stream de bytes, que é requisito para transmissão de dados por rede.

Diversas linguagens possuem suporte à serialização, como: Java, C, C++, Python, PHP, .NET, entre outras. Neste artigo, será tratado apenas da serialização com Java, em sua implemetação básica, alguns problemas e soluções, com códigos para exemplificação.

**2. SERIALIZAÇÃO COM JAVA**

Serializar um objeto, em Java, só é possível caso sua classe esteja marcada como serializável. Essa marcação é feita com a simples implementação da interface “java.io.Serializable”. Essa interface não possui métodos para se implementar e nem atributos. Ainda não existiam as anotações no Java 1.1, versão que trouxe pela primeira vez a serialização ao Java, e desta maneira a implementação da interface era a melhor maneira de marcar uma classe.

A herança entre classes é naturalmente afetada pela serialização. Já que quando uma classe implementa a interface “Serializable”, toda classe que dela estender também estará implicitamente marcada como serializável

.

Para exemplo, será criada uma classe “pojo.Fruta” que contém dois atributos: nome e cor. No código exemplo (Listagem 1), pode-se notar a classe “Fruta” marcada para a serialização:

Listagem 1 - Marcando uma classe como serializável.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | import java.io.Serializable;    public class Fruta implements Serializable{      private String nome;      private String cor;        public Fruta(String nome, String cor) {          this.nome = nome;          this.cor = cor;      }        public String getCor() { return cor; }      public String getNome() { return nome; }      public void setCor(String cor) { this.cor = cor; }      public void setNome(String nome) { this.nome = nome; }  } |

Pode-se dizer que a implementação do processo de serialização e deserialização simples de um objeto, não possui um código difícil de se escrever. Isto porquê duas classes fazem a maior parte do trabalho, de maneira que com poucas linhas de código pode-se transformar um objeto em bytes e vice-versa. Essas duas classes são “java.io.ObjectInputStream” que recebe bytes para criar um objeto e “java.io.ObjectOutputStream” que cria uma cadeia de bytes a partir de um objeto.

**3. SERIALIZANDO E DESERIALIZANDO O ESTADO DE UM OBJETO**

Como exemplo, será persistido um objeto em disco. Para isto, além das classes “java.io.ObjectInputStream” e “java.io.ObjectOutputStream”, serão utilizadas duas outras que trabalham com arquivos, que são “java.io.FileInputStream” e “java.io.FileOutputStream”, que seguem a mesma lógica de InputStream e OutputStream, só que agora relacionado ao sistema de arquivos e não aos objetos.

Para serialização, deve-se criar um novo “java.io.FileOutputStream”, com o caminho (“path”) do arquivo onde se deseja armazenar o estado do objeto. Também deve ser criado um novo “java.io.ObjectOutputStream” que está ligado ao arquivo. E por último deve-se executar a escrita do objeto alvo, para dentro do objeto de stream que por sua vez está ligado ao arquivo destino. Após executados esses passos, é possível encontrar no sistema de arquivos um novo arquivo que contém os bytes correspondentes à descrição do estado do objeto. Na Listagem 2, encontra-se o código exemplo.

Listagem 2 - Serialização de um objeto.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | import java.io.FileOutputStream;  import java.io.ObjectOutputStream;    public class Serializador {        public Serializador() {    }        public void serializar(String path, Object obj) throws Exception {              FileOutputStream outFile = new FileOutputStream(path);      ObjectOutputStream s = new ObjectOutputStream(outFile);      s.writeObject(obj);      s.close();      }  } |

E para a deserialização, o processo é extremamente parecido com a serialização. Cria-se um novo “java.io.FileInputStream” relacionado ao arquivo onde o estado do objeto encontra-se persistido. Deve-se criar também um novo “java.io.ObjectInputStream” que estará vinculado ao arquivo de origem. Por último, executa-se a leitura do arquivo para a restauração do estado. Na Listagem 3 encontra-se o exemplo do código.

Listagem 3 - Deserializaçãode um objeto.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | import java.io.FileInputStream;  import java.io.ObjectInputStream;    public class Deserializador {        public Deserializador() {    }        public Object deserializar(String path) throws Exception {      FileInputStream inFile = new FileInputStream(path);      ObjectInputStream d = new ObjectInputStream(inFile);      Object o = d.readObject();      d.close();      return o;      }  } |

A Listagem 4, contém o exemplo da execução da serialização e em seguida da deserialização.

Listagem 4 - Serializando e Deserializando

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | import operacoes.Deserializador;  import operacoes.Serializador;  import pojo.Fruta;    public class Principal {      public static void main(String args[]){         //serializa         Serializador s = new Serializador();         Fruta fruta = new Fruta("Maça", "Vermelha");          try {              s.serializar("/home/gustavo/fruta", fruta);          } catch (Exception ex) {              System.err.println("Falha ao serializar! - " + ex.toString());          }            //deserializa          Deserializador d = new Deserializador();          fruta = null;          try {              fruta = (Fruta) d.deserializar("/home/gustavo/fruta");          } catch (Exception ex) {              System.err.println("Falha ao deserializar! - " + ex.toString());          }          System.out.println(fruta.getNome() + " - " + fruta.getCor());      }  } |

Na Listagem 5, encontra-se um exemplo de serialização e deserialização de um ArrayList da classe Fruta. É apenas mais um exemplo que demonstra a capacidade de se persistir qualquer objeto que seja serializável. Essa serialização só foi possível porque a classe “ArrayList” implementa a interface “Serializable” por padrão.

Listagem 5 - Serialização de ArrayList.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | //serialização e deserialização com ArrayList          ArrayList<Fruta> frutas = new ArrayList<Fruta>();          Fruta f1 = new Fruta("Laranja", "Amarela");          Fruta f2 = new Fruta("Abacate", "Verde");          Fruta f3 = new Fruta("Morango", "Vermelho");          frutas.add(f1);          frutas.add(f2);          frutas.add(f3);            try {              s.serializar("/home/gustavo/frutas", frutas);              frutas = null;              frutas = (ArrayList<Fruta>) d.deserializar("/home/gustavo/frutas");              for(Fruta f : frutas){                  System.out.println("ArrayList: " + f.getNome() + " - " + f.getCor());              }          } catch (Exception ex) {              System.err.println("Falha ao serializar ou deserializar! - " +                                 ex.toString());          } |

**4. PARTICULARIDADES DA SERIALIZAÇÃO**

Como dito, a serialização de um objeto em Java pode ser algo fácil de se fazer. Mas existem certas informações que são importantes e sabê-las pode evitar vários problemas.

Um dos problemas é que nem tudo pode ser serializado, como objetos das classes “Thread” e “Socket” e atributos estáticos. Os atributos estáticos estão vinculados à classe e não ao objeto. Assim quando se trata do estado do objeto, os campos estáticos não são considerados. Em relação às Threads, suas instâncias só são úteis enquanto estiverem em execução na JVM, e podem ser paradas/destruídas e iniciadas com instruções de sua classe. Não existe algo no estado destas instâncias que seja útil para se persistir, e por isso não há sentido em persistir tais instâncias para restaurar posteriormente.

Mas e se existir alguma classe em que seja necessário persisti-la, e esta possui uma instância da classe “Thread”? Neste caso, utilizando um recurso pode-se persistir a classe inteira, menos a instância thread. Este recurso, refere-se em indicar que tal instância thread é um objeto “Transient”. Na Listagem 6 encontra-se um exemplo da implementação com uso de “Transient”.

Listagem 6 - Uso do Transient

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | import java.io.Serializable;    public class ImplementacaoTransient implements Serializable, Runnable{        transient private Thread thread;      //marcado com "transient" para que sua instância não seja serializada        //Demais valores de estado serão serializados      private int valorParaPersistir;        public ImplementacaoTransient(int valorParaPersistir){          this.valorParaPersistir = valorParaPersistir;          thread = new Thread(this);          thread.start();      }        public void run()      {          while(true){              //código a ser executado pela Thread          }      }  } |

Vale ressaltar que, o que é serializado é apenas o estado do objeto, e não os métodos e a classe. No exemplo acima, apenas a instância do objeto “thread” e do objeto “valorParaPersistir” compunham o estado do objeto. Assim, evita-se que o sistema de serialização tente serializar um objeto não-serializável e gere erros. Após a deserialização de um objeto da classe “ImplementacaoTransient” este estará totalmente funcional.

Um outro problema que pode vir a ocorrer, se refere ao versionamento das classes. Toda classe possui um atributo oculto chamado “serialVersionUID” com um valor gerado automaticamente. Objetos criados levam consigo o “serialVersionUID” de sua classe, que indica sua versão. Toda vez que uma classe sofre alterações em seu código, um novo “serialVersionUID” é gerado. O problema com a serialização surge quando serializa-se o estado de um objeto e posteriormente tenta-se restaurar o estado para a aplicação, mas houveram mudanças no código da classe. Neste caso, uma exceção “java.io.InvalidClassException” é disparada, pois de fato, o estado do objeto corresponde a uma versão de classe do qual foi criado que já não existe mais. Portanto, isso só ocorre para evitar erros de programação, em que tenta-se restaurar o estado de um objeto para uma classe que possui versão diferente da qual o objeto foi serializado inicialmente. E isso pode significar diferentes versões de uma mesma classe “Fruta” por exemplo, ou diferentes versões por classes serem totalmente diferentes como: o estado de um objeto da classe “Fruta” ser restaurado para classe “Carro”.

Mas quando é necessário, conscientemente, recuperar o estado de um objeto, mesmo que houveram alterações na classe do qual foi gerado, é possível controlar manualmente o “serialVersionUID”. Para isto, basta declarar explicitamente na forma de atributo estático e final, um valor Long qualquer para corresponder à versão da classe. E assim, mesmo que houverem modificações na classe, a versão permenecerá a mesma, evitando as exceções da incompatiblidade de classes. No entanto, vale ressaltar que o versionamento não é um impecílio para o programador, mas sim muito importante em casos que deseja-se manter um controle sobre a compatibilidade de código na aplicação, definindo versões para classes No código da Listagem 7 encontra-se o exemplo do controle manual do “serialVersionUID”.

Listagem 7 - Uso manual do serialVersionUID

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | import java.io.Serializable;    public class Fruta implements Serializable{      private String nome;      private String cor;      static final long serialVersionUID = 123L; //indica a versao da classe        public Fruta(String nome, String cor) {          this.nome = nome;          this.cor = cor;      }      public String getCor() { return cor; }      public String getNome() { return nome; }      public void setCor(String cor) { this.cor = cor; }      public void setNome(String nome) { this.nome = nome; }  } |

O JDK fornece uma ferramenta que permite visualizar a versão de qualquer classe Java. Basta no terminal do sistema operacional utilizar o comando “serialver” seguido da classe qualificada, conforme exemplo da Listagem 8.

Listagem 8 - Verificação da versão de classe

|  |  |
| --- | --- |
| 1 | serialver pojo.Fruta |

A serialização acaba inflingindo alguns modelos de segurança em seu procedimento padrão, e criando então um novo problema. Quando um estado de objeto é persistido no sistema de arquivos ou enviado por rede, o fluxo binário pode ser detectado, analisado e obtidas informações sobre a classe e o estado do objeto. Por exemplo, se utilizado um editor de texto qualquer para abrir o arquivo persistido do estado do objeto da classe “Fruta” dos exemplos anteriores, é comum os atributos e outras informações aparecerem como um simples texto, total ou parcilamente legíveis, assim como demonstrado na Listagem 9.

Listagem 9 - Exemplo de leitura de arquivo binário de um estado de objeto persistido.

|  |  |
| --- | --- |
| 1  2 | ��##sr#  pojo.Fruta#######{###L##cort##Ljava/lang/String;L##nomeq#~##xpt##Vermelhat##Maça |

Com uma ferramenta menos simples seria possível aumentar a quantidade de informações legíveis, coletando uma maior quantia de dados.

No entando, o processo de serialização permite que haja uma certa personalização da serialização e deserialização de um objeto. Isso deve-se a possibilidade da sobrescrição dos métodos “writeObject” e “readObject” dentro da classe a ser serializada, como a classe exemplar “pojo.Fruta”. Esta sobrescrita permite um controle total sobre a serialização, pois as classes “java.io.ObjectOutputStream” e “java.io.ObjectInputStream”, ao serializar o estado de um objeto, primeiramente verificam se esse método foi sobrescrito na classe alvo e assim o utilizam, senão, será utilizado o protocolo padrão de serialização.

Realizando a sobrescrição dos métodos é possível criptografar os dados na serialização e descriptografar os dados na deserialização, por meio de algum algoritmo que o programador venha implementar. Assim os dados podem ser persistidos ou transmitidos por rede, sem que seja possível ler as partes mais importantes do estado de um objeto. Na listagem 10, a classe “pojo.Fruta” foi modificada dando origem a classe “pojo.FrutaSobrescrita” que possui adicionamente a sobrescrição dos métodos “writeObject” e “readObject”. Como este é apenas um exemplo, não foi escrito um verdadeiro algoritmo de criptografia, mas sim apenas uma troca no valor da String do campo “cor”, identificando que houveram alterações tanto no processo de serialização quanto no de deserialização. Na listagem 11 encontra-se a execução da serialização.

Listagem 10 - Sobrescrita dos métodos “writeObject” e “readObject”.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36 | import java.io.Serializable;    public class FrutaSobrescrita implements Serializable{      private String nome;      private String cor;      static final long serialVersionUID = 123L; //indica a versao da classe        public FrutaSobrescrita(String nome, String cor) {          this.nome = nome;          this.cor = cor;      }      public String getCor() { return cor; }      public String getNome() { return nome; }      public void setCor(String cor) { this.cor = cor; }      public void setNome(String nome) { this.nome = nome; }        private void writeObject(java.io.ObjectOutputStream stream) throws java.io.IOException      {          // criptografar os dados antes de liberar o stream          // EXEMPLO: setCor(criptografar(getCor()));          setCor(getCor()+ " > CRIPTOGRAFADO"); // apenas concatenação de string para indicar alteração            // liberar o stream          stream.defaultWriteObject();      }        private void readObject(java.io.ObjectInputStream stream) throws java.io.IOException, ClassNotFoundException      {          // liberar stream          stream.defaultReadObject();            // descriptografar os dados          // EXEMPLO: setCor(descriptografar(getCor()));          setCor(getCor()+ " > DESCRIPTOGRAFADO"); // apenas contatenação de string para indicar alteração      }  } |

Listagem 11 - Execução da serialização da classe “pojo.FrutaSobrescrita”.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | //serializa        Serializador s = new Serializador();        FrutaSobrescrita frutaSobrescrita = new FrutaSobrescrita("Maça", "Vermelha");         try {             s.serializar("/home/gustavo/frutaSobrescrita", frutaSobrescrita);         } catch (Exception ex) {             System.err.println("Falha ao serializar! - " + ex.toString());         }           //deserializa        Deserializador d = new Deserializador();         frutaSobrescrita = null;         try {             frutaSobrescrita = (FrutaSobrescrita) d.deserializar("/home/gustavo/frutaSobrescrita");         } catch (Exception ex) {             System.err.println("Falha ao deserializar! - " + ex.toString());         }         System.out.println("Sobrescrita: " + frutaSobrescrita.getNome() + " - " + frutaSobrescrita.getCor()); |

**5. CONCLUSÃO**

A serialização é um procedimento de simples execução por parte do programador. Com pouco tempo de codificação é possível implementar a persistencia de um estado de objeto para o sistema de arquivos ou de transmissão por rede. É possível reproduzir os exemplos criados neste artigo, em situações até mesmo mais elaboradas. A apresentação de alguns problemas e soluções trazidas por particularidades, abre o entendimento de que a serialização pode deixar de ser simples conforme for aplicada em diferentes contextos, e que ao mesmo tempo é muito flexível ao permitir ao programador implementar diferentes soluções.

[Assista ao vídeo Serialização/Deserialização de objetos com YAML](http://www.devmedia.com.br/serializacao-deserializacao-de-objetos-com-yaml-parte-1/8399)

**6. BIBLIOGRAFIA**

ORACLE - **java.io Interface Serializable** - Disponível em: <http://docs.oracle.com/javase/1.4.2/docs/api/java/io/Serializable.html>Acesso em: 11/03/2012

ORACLE - **Discover de Secrets of Java Serialization API** - Disponível em: <http://java.sun.com/developer/technicalArticles/Programming/serialization/> Acesso em: 11/03/2012

NEWARD, Ted - **5 coisas que você não sabia sobre... Serialização de Objetos Java** - Disponível em: <http://www.ibm.com/developerworks/br/library/j-5things1/index.html> Acesso em: 11/03/2012

DOWNS, Andrew - **Java Serialization** - Disponível em: <http://www.mactech.com/articles/mactech/Vol.14/14.04/JavaSerialization/index.html> Acesso em: 11/03/2012

PREISSLER, Luciano E.; GEYER, Cláudio F. R. - **Serialização em Java** - Disponível em: <http://www.inf.ufrgs.br/gppd/disc/cmp167/trabalhos/sem99-1/T1/luciano/final.htm> Acesso em: 11/03/2012

[[](http://www.linhadecodigo.com.br/colaborador.aspx?id=354)](http://www.linhadecodigo.com.br/colaborador.aspx?id=354)

[Everton Coimbra de Araújo - Desde 1987 atua na área de treinamento e desenvolvimento. Como Mestre em Ciência da Computação, é professor da UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ, Campus Medianeira, onde leciona disciplinas relacionadas ao desenvolvimento de aplicações web, com Java e .NET.   
  
É autor dos livros Desenvolvimento para WEB com Java, Orientação a Objetos com Java - Simples, Fácil e Eficiente, Algoritmo - Fundamento e Prática em sua terceira edição e dos livros Delphi - Implementação e Técnicas para Ambientes Virtuais e C++ Builder - Implementação e Técnicas para Ambientes Virtuais. Todos pela VisualBooks. Pode ser contactado através do e-mail everton@utfpr.edu.br ou evertoncoimbra@gmail.com.](http://www.linhadecodigo.com.br/colaborador.aspx?id=354)

 85 3 0 4

**Leia também**

* [Conhecendo a interface Map do Java](http://www.linhadecodigo.com.br/artigo/3670/conhecendo-a-interface-map-do-java.aspx)*Java*
* [Trabalhando com a Interface Set no Java](http://www.linhadecodigo.com.br/artigo/3669/trabalhando-com-a-interface-set-no-java.aspx)*Java*
* [Classes Wrappers em Java](http://www.linhadecodigo.com.br/artigo/3667/classes-wrappers-em-java.aspx)*Java*
* [ComboBoxModel em Java](http://www.linhadecodigo.com.br/artigo/3664/comboboxmodel-em-java.aspx)*Java*
* [Reduzindo a Mutabilidade em Java](http://www.linhadecodigo.com.br/artigo/3659/reduzindo-a-mutabilidade-em-java.aspx)*Java*

Publicidade

[](http://qconsp.com/?utm_source=devmedia&utm_medium=banner&utm_campaign=devmedia)

Linha de Código[Anuncie](http://www.web-03.net/linhadecodigo.asp)|[Fale Conosco](mailto:colabore@linhadecodigo.com.br)|[Publique](http://www.linhadecodigo.com.br/publique.aspx)

|  |  |
| --- | --- |
| http://www.linhadecodigo.com.br/artigos/img_artigos/EvertonCoimbraAraujo/Serializazao_Objetos/imagem001.jpg | Neste artigo apresento o Gustavo Rafael Valiati, que pode ser contatado pelo e-mail gustavovaliati@gmail.com. O Gustavo foi meu aluno em algumas disciplinas de sua gradução e está concluíndo o último período do curso de Tecnologia em Análise e Desenvolvimento de Sistemas da UTFPR campus Medianeira, neste primeiro semestre de 2012. |

**1. INTRODUÇÃO**

Nem sempre se deseja ter objetos ou estrutura de dados de um aplicativo, com um ciclo de vida que dura no máximo o tempo em que for armazenado em memória RAM. Ou seja, algumas vezes deseja-se guardá-los por um tempo indeterminado, sendo possível recuperá-los posteriormente. E neste caso, o termo “persistência” tem seu conceito definido.

Persistir dados, requer que estes possam ser serializáveis. A serialização possibilitará capturar o estado do objeto ou a estrutura de dados, e transformar em uma cadeia de bytes quando for necessário. Torna-se possível também, recuperar os bytes persistidos para fazer o processo inverso, e ter os dados de volta para a aplicação em execução.

Desta maneira, qualquer forma de dados que esteja sendo trabalhada na aplicação, e esta puder ser serializada, poderá ser passada para o meio físico, e ter o processo revertido.

A serialização não se limita apenas em possibilitar a gravação do dado em disco, mas permite que o mesmo seja transmitido por rede. Isto porquê a serialização cria um stream de bytes, que é requisito para transmissão de dados por rede.

Diversas linguagens possuem suporte à serialização, como: Java, C, C++, Python, PHP, .NET, entre outras. Neste artigo, será tratado apenas da serialização com Java, em sua implemetação básica, alguns problemas e soluções, com códigos para exemplificação.

**2. SERIALIZAÇÃO COM JAVA**

Serializar um objeto, em Java, só é possível caso sua classe esteja marcada como serializável. Essa marcação é feita com a simples implementação da interface “java.io.Serializable”. Essa interface não possui métodos para se implementar e nem atributos. Ainda não existiam as anotações no Java 1.1, versão que trouxe pela primeira vez a serialização ao Java, e desta maneira a implementação da interface era a melhor maneira de marcar uma classe.

A herança entre classes é naturalmente afetada pela serialização. Já que quando uma classe implementa a interface “Serializable”, toda classe que dela estender também estará implicitamente marcada como serializável

.

Para exemplo, será criada uma classe “pojo.Fruta” que contém dois atributos: nome e cor. No código exemplo (Listagem 1), pode-se notar a classe “Fruta” marcada para a serialização:

Listagem 1 - Marcando uma classe como serializável.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | import java.io.Serializable;    public class Fruta implements Serializable{      private String nome;      private String cor;        public Fruta(String nome, String cor) {          this.nome = nome;          this.cor = cor;      }        public String getCor() { return cor; }      public String getNome() { return nome; }      public void setCor(String cor) { this.cor = cor; }      public void setNome(String nome) { this.nome = nome; }  } |

Pode-se dizer que a implementação do processo de serialização e deserialização simples de um objeto, não possui um código difícil de se escrever. Isto porquê duas classes fazem a maior parte do trabalho, de maneira que com poucas linhas de código pode-se transformar um objeto em bytes e vice-versa. Essas duas classes são “java.io.ObjectInputStream” que recebe bytes para criar um objeto e “java.io.ObjectOutputStream” que cria uma cadeia de bytes a partir de um objeto.

**3. SERIALIZANDO E DESERIALIZANDO O ESTADO DE UM OBJETO**

Como exemplo, será persistido um objeto em disco. Para isto, além das classes “java.io.ObjectInputStream” e “java.io.ObjectOutputStream”, serão utilizadas duas outras que trabalham com arquivos, que são “java.io.FileInputStream” e “java.io.FileOutputStream”, que seguem a mesma lógica de InputStream e OutputStream, só que agora relacionado ao sistema de arquivos e não aos objetos.

Para serialização, deve-se criar um novo “java.io.FileOutputStream”, com o caminho (“path”) do arquivo onde se deseja armazenar o estado do objeto. Também deve ser criado um novo “java.io.ObjectOutputStream” que está ligado ao arquivo. E por último deve-se executar a escrita do objeto alvo, para dentro do objeto de stream que por sua vez está ligado ao arquivo destino. Após executados esses passos, é possível encontrar no sistema de arquivos um novo arquivo que contém os bytes correspondentes à descrição do estado do objeto. Na Listagem 2, encontra-se o código exemplo.

Listagem 2 - Serialização de um objeto.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | import java.io.FileOutputStream;  import java.io.ObjectOutputStream;    public class Serializador {        public Serializador() {    }        public void serializar(String path, Object obj) throws Exception {              FileOutputStream outFile = new FileOutputStream(path);      ObjectOutputStream s = new ObjectOutputStream(outFile);      s.writeObject(obj);      s.close();      }  } |

E para a deserialização, o processo é extremamente parecido com a serialização. Cria-se um novo “java.io.FileInputStream” relacionado ao arquivo onde o estado do objeto encontra-se persistido. Deve-se criar também um novo “java.io.ObjectInputStream” que estará vinculado ao arquivo de origem. Por último, executa-se a leitura do arquivo para a restauração do estado. Na Listagem 3 encontra-se o exemplo do código.

Listagem 3 - Deserializaçãode um objeto.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | import java.io.FileInputStream;  import java.io.ObjectInputStream;    public class Deserializador {        public Deserializador() {    }        public Object deserializar(String path) throws Exception {      FileInputStream inFile = new FileInputStream(path);      ObjectInputStream d = new ObjectInputStream(inFile);      Object o = d.readObject();      d.close();      return o;      }  } |

A Listagem 4, contém o exemplo da execução da serialização e em seguida da deserialização.

Listagem 4 - Serializando e Deserializando

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | import operacoes.Deserializador;  import operacoes.Serializador;  import pojo.Fruta;    public class Principal {      public static void main(String args[]){         //serializa         Serializador s = new Serializador();         Fruta fruta = new Fruta("Maça", "Vermelha");          try {              s.serializar("/home/gustavo/fruta", fruta);          } catch (Exception ex) {              System.err.println("Falha ao serializar! - " + ex.toString());          }            //deserializa          Deserializador d = new Deserializador();          fruta = null;          try {              fruta = (Fruta) d.deserializar("/home/gustavo/fruta");          } catch (Exception ex) {              System.err.println("Falha ao deserializar! - " + ex.toString());          }          System.out.println(fruta.getNome() + " - " + fruta.getCor());      }  } |

Na Listagem 5, encontra-se um exemplo de serialização e deserialização de um ArrayList da classe Fruta. É apenas mais um exemplo que demonstra a capacidade de se persistir qualquer objeto que seja serializável. Essa serialização só foi possível porque a classe “ArrayList” implementa a interface “Serializable” por padrão.

Listagem 5 - Serialização de ArrayList.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | //serialização e deserialização com ArrayList          ArrayList<Fruta> frutas = new ArrayList<Fruta>();          Fruta f1 = new Fruta("Laranja", "Amarela");          Fruta f2 = new Fruta("Abacate", "Verde");          Fruta f3 = new Fruta("Morango", "Vermelho");          frutas.add(f1);          frutas.add(f2);          frutas.add(f3);            try {              s.serializar("/home/gustavo/frutas", frutas);              frutas = null;              frutas = (ArrayList<Fruta>) d.deserializar("/home/gustavo/frutas");              for(Fruta f : frutas){                  System.out.println("ArrayList: " + f.getNome() + " - " + f.getCor());              }          } catch (Exception ex) {              System.err.println("Falha ao serializar ou deserializar! - " +                                 ex.toString());          } |

**4. PARTICULARIDADES DA SERIALIZAÇÃO**

Como dito, a serialização de um objeto em Java pode ser algo fácil de se fazer. Mas existem certas informações que são importantes e sabê-las pode evitar vários problemas.

Um dos problemas é que nem tudo pode ser serializado, como objetos das classes “Thread” e “Socket” e atributos estáticos. Os atributos estáticos estão vinculados à classe e não ao objeto. Assim quando se trata do estado do objeto, os campos estáticos não são considerados. Em relação às Threads, suas instâncias só são úteis enquanto estiverem em execução na JVM, e podem ser paradas/destruídas e iniciadas com instruções de sua classe. Não existe algo no estado destas instâncias que seja útil para se persistir, e por isso não há sentido em persistir tais instâncias para restaurar posteriormente.

Mas e se existir alguma classe em que seja necessário persisti-la, e esta possui uma instância da classe “Thread”? Neste caso, utilizando um recurso pode-se persistir a classe inteira, menos a instância thread. Este recurso, refere-se em indicar que tal instância thread é um objeto “Transient”. Na Listagem 6 encontra-se um exemplo da implementação com uso de “Transient”.

Listagem 6 - Uso do Transient

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | import java.io.Serializable;    public class ImplementacaoTransient implements Serializable, Runnable{        transient private Thread thread;      //marcado com "transient" para que sua instância não seja serializada        //Demais valores de estado serão serializados      private int valorParaPersistir;        public ImplementacaoTransient(int valorParaPersistir){          this.valorParaPersistir = valorParaPersistir;          thread = new Thread(this);          thread.start();      }        public void run()      {          while(true){              //código a ser executado pela Thread          }      }  } |

Vale ressaltar que, o que é serializado é apenas o estado do objeto, e não os métodos e a classe. No exemplo acima, apenas a instância do objeto “thread” e do objeto “valorParaPersistir” compunham o estado do objeto. Assim, evita-se que o sistema de serialização tente serializar um objeto não-serializável e gere erros. Após a deserialização de um objeto da classe “ImplementacaoTransient” este estará totalmente funcional.

Um outro problema que pode vir a ocorrer, se refere ao versionamento das classes. Toda classe possui um atributo oculto chamado “serialVersionUID” com um valor gerado automaticamente. Objetos criados levam consigo o “serialVersionUID” de sua classe, que indica sua versão. Toda vez que uma classe sofre alterações em seu código, um novo “serialVersionUID” é gerado. O problema com a serialização surge quando serializa-se o estado de um objeto e posteriormente tenta-se restaurar o estado para a aplicação, mas houveram mudanças no código da classe. Neste caso, uma exceção “java.io.InvalidClassException” é disparada, pois de fato, o estado do objeto corresponde a uma versão de classe do qual foi criado que já não existe mais. Portanto, isso só ocorre para evitar erros de programação, em que tenta-se restaurar o estado de um objeto para uma classe que possui versão diferente da qual o objeto foi serializado inicialmente. E isso pode significar diferentes versões de uma mesma classe “Fruta” por exemplo, ou diferentes versões por classes serem totalmente diferentes como: o estado de um objeto da classe “Fruta” ser restaurado para classe “Carro”.

Mas quando é necessário, conscientemente, recuperar o estado de um objeto, mesmo que houveram alterações na classe do qual foi gerado, é possível controlar manualmente o “serialVersionUID”. Para isto, basta declarar explicitamente na forma de atributo estático e final, um valor Long qualquer para corresponder à versão da classe. E assim, mesmo que houverem modificações na classe, a versão permenecerá a mesma, evitando as exceções da incompatiblidade de classes. No entanto, vale ressaltar que o versionamento não é um impecílio para o programador, mas sim muito importante em casos que deseja-se manter um controle sobre a compatibilidade de código na aplicação, definindo versões para classes No código da Listagem 7 encontra-se o exemplo do controle manual do “serialVersionUID”.

Listagem 7 - Uso manual do serialVersionUID

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | import java.io.Serializable;    public class Fruta implements Serializable{      private String nome;      private String cor;      static final long serialVersionUID = 123L; //indica a versao da classe        public Fruta(String nome, String cor) {          this.nome = nome;          this.cor = cor;      }      public String getCor() { return cor; }      public String getNome() { return nome; }      public void setCor(String cor) { this.cor = cor; }      public void setNome(String nome) { this.nome = nome; }  } |

O JDK fornece uma ferramenta que permite visualizar a versão de qualquer classe Java. Basta no terminal do sistema operacional utilizar o comando “serialver” seguido da classe qualificada, conforme exemplo da Listagem 8.

Listagem 8 - Verificação da versão de classe

|  |  |
| --- | --- |
| 1 | serialver pojo.Fruta |

A serialização acaba inflingindo alguns modelos de segurança em seu procedimento padrão, e criando então um novo problema. Quando um estado de objeto é persistido no sistema de arquivos ou enviado por rede, o fluxo binário pode ser detectado, analisado e obtidas informações sobre a classe e o estado do objeto. Por exemplo, se utilizado um editor de texto qualquer para abrir o arquivo persistido do estado do objeto da classe “Fruta” dos exemplos anteriores, é comum os atributos e outras informações aparecerem como um simples texto, total ou parcilamente legíveis, assim como demonstrado na Listagem 9.

Listagem 9 - Exemplo de leitura de arquivo binário de um estado de objeto persistido.

|  |  |
| --- | --- |
| 1  2 | ��##sr#  pojo.Fruta#######{###L##cort##Ljava/lang/String;L##nomeq#~##xpt##Vermelhat##Maça |

Com uma ferramenta menos simples seria possível aumentar a quantidade de informações legíveis, coletando uma maior quantia de dados.

No entando, o processo de serialização permite que haja uma certa personalização da serialização e deserialização de um objeto. Isso deve-se a possibilidade da sobrescrição dos métodos “writeObject” e “readObject” dentro da classe a ser serializada, como a classe exemplar “pojo.Fruta”. Esta sobrescrita permite um controle total sobre a serialização, pois as classes “java.io.ObjectOutputStream” e “java.io.ObjectInputStream”, ao serializar o estado de um objeto, primeiramente verificam se esse método foi sobrescrito na classe alvo e assim o utilizam, senão, será utilizado o protocolo padrão de serialização.

Realizando a sobrescrição dos métodos é possível criptografar os dados na serialização e descriptografar os dados na deserialização, por meio de algum algoritmo que o programador venha implementar. Assim os dados podem ser persistidos ou transmitidos por rede, sem que seja possível ler as partes mais importantes do estado de um objeto. Na listagem 10, a classe “pojo.Fruta” foi modificada dando origem a classe “pojo.FrutaSobrescrita” que possui adicionamente a sobrescrição dos métodos “writeObject” e “readObject”. Como este é apenas um exemplo, não foi escrito um verdadeiro algoritmo de criptografia, mas sim apenas uma troca no valor da String do campo “cor”, identificando que houveram alterações tanto no processo de serialização quanto no de deserialização. Na listagem 11 encontra-se a execução da serialização.

Listagem 10 - Sobrescrita dos métodos “writeObject” e “readObject”.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36 | import java.io.Serializable;    public class FrutaSobrescrita implements Serializable{      private String nome;      private String cor;      static final long serialVersionUID = 123L; //indica a versao da classe        public FrutaSobrescrita(String nome, String cor) {          this.nome = nome;          this.cor = cor;      }      public String getCor() { return cor; }      public String getNome() { return nome; }      public void setCor(String cor) { this.cor = cor; }      public void setNome(String nome) { this.nome = nome; }        private void writeObject(java.io.ObjectOutputStream stream) throws java.io.IOException      {          // criptografar os dados antes de liberar o stream          // EXEMPLO: setCor(criptografar(getCor()));          setCor(getCor()+ " > CRIPTOGRAFADO"); // apenas concatenação de string para indicar alteração            // liberar o stream          stream.defaultWriteObject();      }        private void readObject(java.io.ObjectInputStream stream) throws java.io.IOException, ClassNotFoundException      {          // liberar stream          stream.defaultReadObject();            // descriptografar os dados          // EXEMPLO: setCor(descriptografar(getCor()));          setCor(getCor()+ " > DESCRIPTOGRAFADO"); // apenas contatenação de string para indicar alteração      }  } |

Listagem 11 - Execução da serialização da classe “pojo.FrutaSobrescrita”.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | //serializa        Serializador s = new Serializador();        FrutaSobrescrita frutaSobrescrita = new FrutaSobrescrita("Maça", "Vermelha");         try {             s.serializar("/home/gustavo/frutaSobrescrita", frutaSobrescrita);         } catch (Exception ex) {             System.err.println("Falha ao serializar! - " + ex.toString());         }           //deserializa        Deserializador d = new Deserializador();         frutaSobrescrita = null;         try {             frutaSobrescrita = (FrutaSobrescrita) d.deserializar("/home/gustavo/frutaSobrescrita");         } catch (Exception ex) {             System.err.println("Falha ao deserializar! - " + ex.toString());         }         System.out.println("Sobrescrita: " + frutaSobrescrita.getNome() + " - " + frutaSobrescrita.getCor()); |

**5. CONCLUSÃO**

A serialização é um procedimento de simples execução por parte do programador. Com pouco tempo de codificação é possível implementar a persistencia de um estado de objeto para o sistema de arquivos ou de transmissão por rede. É possível reproduzir os exemplos criados neste artigo, em situações até mesmo mais elaboradas. A apresentação de alguns problemas e soluções trazidas por particularidades, abre o entendimento de que a serialização pode deixar de ser simples conforme for aplicada em diferentes contextos, e que ao mesmo tempo é muito flexível ao permitir ao programador implementar diferentes soluções.

[Assista ao vídeo Serialização/Deserialização de objetos com YAML](http://www.devmedia.com.br/serializacao-deserializacao-de-objetos-com-yaml-parte-1/8399)

**6. BIBLIOGRAFIA**

ORACLE - **java.io Interface Serializable** - Disponível em: <http://docs.oracle.com/javase/1.4.2/docs/api/java/io/Serializable.html>Acesso em: 11/03/2012

ORACLE - **Discover de Secrets of Java Serialization API** - Disponível em: <http://java.sun.com/developer/technicalArticles/Programming/serialization/> Acesso em: 11/03/2012

NEWARD, Ted - **5 coisas que você não sabia sobre... Serialização de Objetos Java** - Disponível em: <http://www.ibm.com/developerworks/br/library/j-5things1/index.html> Acesso em: 11/03/2012

DOWNS, Andrew - **Java Serialization** - Disponível em: <http://www.mactech.com/articles/mactech/Vol.14/14.04/JavaSerialization/index.html> Acesso em: 11/03/2012

PREISSLER, Luciano E.; GEYER, Cláudio F. R. - **Serialização em Java** - Disponível em: <http://www.inf.ufrgs.br/gppd/disc/cmp167/trabalhos/sem99-1/T1/luciano/final.htm> Acesso em: 11/03/2012

[[](http://www.linhadecodigo.com.br/colaborador.aspx?id=354)](http://www.linhadecodigo.com.br/colaborador.aspx?id=354)

[Everton Coimbra de Araújo - Desde 1987 atua na área de treinamento e desenvolvimento. Como Mestre em Ciência da Computação, é professor da UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ, Campus Medianeira, onde leciona disciplinas relacionadas ao desenvolvimento de aplicações web, com Java e .NET.   
  
É autor dos livros Desenvolvimento para WEB com Java, Orientação a Objetos com Java - Simples, Fácil e Eficiente, Algoritmo - Fundamento e Prática em sua terceira edição e dos livros Delphi - Implementação e Técnicas para Ambientes Virtuais e C++ Builder - Implementação e Técnicas para Ambientes Virtuais. Todos pela VisualBooks. Pode ser contactado através do e-mail everton@utfpr.edu.br ou evertoncoimbra@gmail.com.](http://www.linhadecodigo.com.br/colaborador.aspx?id=354)

 85 3 0 4

**Leia também**

* [Conhecendo a interface Map do Java](http://www.linhadecodigo.com.br/artigo/3670/conhecendo-a-interface-map-do-java.aspx)*Java*
* [Trabalhando com a Interface Set no Java](http://www.linhadecodigo.com.br/artigo/3669/trabalhando-com-a-interface-set-no-java.aspx)*Java*
* [Classes Wrappers em Java](http://www.linhadecodigo.com.br/artigo/3667/classes-wrappers-em-java.aspx)*Java*
* [ComboBoxModel em Java](http://www.linhadecodigo.com.br/artigo/3664/comboboxmodel-em-java.aspx)*Java*
* [Reduzindo a Mutabilidade em Java](http://www.linhadecodigo.com.br/artigo/3659/reduzindo-a-mutabilidade-em-java.aspx)*Java*

Publicidade

[](http://qconsp.com/?utm_source=devmedia&utm_medium=banner&utm_campaign=devmedia)

Linha de Código[Anuncie](http://www.web-03.net/linhadecodigo.asp)|[Fale Conosco](mailto:colabore@linhadecodigo.com.br)|[Publique](http://www.linhadecodigo.com.br/publique.aspx)

|  |  |
| --- | --- |
| http://www.linhadecodigo.com.br/artigos/img_artigos/EvertonCoimbraAraujo/Serializazao_Objetos/imagem001.jpg | Neste artigo apresento o Gustavo Rafael Valiati, que pode ser contatado pelo e-mail gustavovaliati@gmail.com. O Gustavo foi meu aluno em algumas disciplinas de sua gradução e está concluíndo o último período do curso de Tecnologia em Análise e Desenvolvimento de Sistemas da UTFPR campus Medianeira, neste primeiro semestre de 2012. |

**1. INTRODUÇÃO**

Nem sempre se deseja ter objetos ou estrutura de dados de um aplicativo, com um ciclo de vida que dura no máximo o tempo em que for armazenado em memória RAM. Ou seja, algumas vezes deseja-se guardá-los por um tempo indeterminado, sendo possível recuperá-los posteriormente. E neste caso, o termo “persistência” tem seu conceito definido.

Persistir dados, requer que estes possam ser serializáveis. A serialização possibilitará capturar o estado do objeto ou a estrutura de dados, e transformar em uma cadeia de bytes quando for necessário. Torna-se possível também, recuperar os bytes persistidos para fazer o processo inverso, e ter os dados de volta para a aplicação em execução.

Desta maneira, qualquer forma de dados que esteja sendo trabalhada na aplicação, e esta puder ser serializada, poderá ser passada para o meio físico, e ter o processo revertido.

A serialização não se limita apenas em possibilitar a gravação do dado em disco, mas permite que o mesmo seja transmitido por rede. Isto porquê a serialização cria um stream de bytes, que é requisito para transmissão de dados por rede.

Diversas linguagens possuem suporte à serialização, como: Java, C, C++, Python, PHP, .NET, entre outras. Neste artigo, será tratado apenas da serialização com Java, em sua implemetação básica, alguns problemas e soluções, com códigos para exemplificação.

**2. SERIALIZAÇÃO COM JAVA**

Serializar um objeto, em Java, só é possível caso sua classe esteja marcada como serializável. Essa marcação é feita com a simples implementação da interface “java.io.Serializable”. Essa interface não possui métodos para se implementar e nem atributos. Ainda não existiam as anotações no Java 1.1, versão que trouxe pela primeira vez a serialização ao Java, e desta maneira a implementação da interface era a melhor maneira de marcar uma classe.

A herança entre classes é naturalmente afetada pela serialização. Já que quando uma classe implementa a interface “Serializable”, toda classe que dela estender também estará implicitamente marcada como serializável

.

Para exemplo, será criada uma classe “pojo.Fruta” que contém dois atributos: nome e cor. No código exemplo (Listagem 1), pode-se notar a classe “Fruta” marcada para a serialização:

Listagem 1 - Marcando uma classe como serializável.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | import java.io.Serializable;    public class Fruta implements Serializable{      private String nome;      private String cor;        public Fruta(String nome, String cor) {          this.nome = nome;          this.cor = cor;      }        public String getCor() { return cor; }      public String getNome() { return nome; }      public void setCor(String cor) { this.cor = cor; }      public void setNome(String nome) { this.nome = nome; }  } |

Pode-se dizer que a implementação do processo de serialização e deserialização simples de um objeto, não possui um código difícil de se escrever. Isto porquê duas classes fazem a maior parte do trabalho, de maneira que com poucas linhas de código pode-se transformar um objeto em bytes e vice-versa. Essas duas classes são “java.io.ObjectInputStream” que recebe bytes para criar um objeto e “java.io.ObjectOutputStream” que cria uma cadeia de bytes a partir de um objeto.

**3. SERIALIZANDO E DESERIALIZANDO O ESTADO DE UM OBJETO**

Como exemplo, será persistido um objeto em disco. Para isto, além das classes “java.io.ObjectInputStream” e “java.io.ObjectOutputStream”, serão utilizadas duas outras que trabalham com arquivos, que são “java.io.FileInputStream” e “java.io.FileOutputStream”, que seguem a mesma lógica de InputStream e OutputStream, só que agora relacionado ao sistema de arquivos e não aos objetos.

Para serialização, deve-se criar um novo “java.io.FileOutputStream”, com o caminho (“path”) do arquivo onde se deseja armazenar o estado do objeto. Também deve ser criado um novo “java.io.ObjectOutputStream” que está ligado ao arquivo. E por último deve-se executar a escrita do objeto alvo, para dentro do objeto de stream que por sua vez está ligado ao arquivo destino. Após executados esses passos, é possível encontrar no sistema de arquivos um novo arquivo que contém os bytes correspondentes à descrição do estado do objeto. Na Listagem 2, encontra-se o código exemplo.

Listagem 2 - Serialização de um objeto.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | import java.io.FileOutputStream;  import java.io.ObjectOutputStream;    public class Serializador {        public Serializador() {    }        public void serializar(String path, Object obj) throws Exception {              FileOutputStream outFile = new FileOutputStream(path);      ObjectOutputStream s = new ObjectOutputStream(outFile);      s.writeObject(obj);      s.close();      }  } |

E para a deserialização, o processo é extremamente parecido com a serialização. Cria-se um novo “java.io.FileInputStream” relacionado ao arquivo onde o estado do objeto encontra-se persistido. Deve-se criar também um novo “java.io.ObjectInputStream” que estará vinculado ao arquivo de origem. Por último, executa-se a leitura do arquivo para a restauração do estado. Na Listagem 3 encontra-se o exemplo do código.

Listagem 3 - Deserializaçãode um objeto.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | import java.io.FileInputStream;  import java.io.ObjectInputStream;    public class Deserializador {        public Deserializador() {    }        public Object deserializar(String path) throws Exception {      FileInputStream inFile = new FileInputStream(path);      ObjectInputStream d = new ObjectInputStream(inFile);      Object o = d.readObject();      d.close();      return o;      }  } |

A Listagem 4, contém o exemplo da execução da serialização e em seguida da deserialização.

Listagem 4 - Serializando e Deserializando

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | import operacoes.Deserializador;  import operacoes.Serializador;  import pojo.Fruta;    public class Principal {      public static void main(String args[]){         //serializa         Serializador s = new Serializador();         Fruta fruta = new Fruta("Maça", "Vermelha");          try {              s.serializar("/home/gustavo/fruta", fruta);          } catch (Exception ex) {              System.err.println("Falha ao serializar! - " + ex.toString());          }            //deserializa          Deserializador d = new Deserializador();          fruta = null;          try {              fruta = (Fruta) d.deserializar("/home/gustavo/fruta");          } catch (Exception ex) {              System.err.println("Falha ao deserializar! - " + ex.toString());          }          System.out.println(fruta.getNome() + " - " + fruta.getCor());      }  } |

Na Listagem 5, encontra-se um exemplo de serialização e deserialização de um ArrayList da classe Fruta. É apenas mais um exemplo que demonstra a capacidade de se persistir qualquer objeto que seja serializável. Essa serialização só foi possível porque a classe “ArrayList” implementa a interface “Serializable” por padrão.

Listagem 5 - Serialização de ArrayList.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | //serialização e deserialização com ArrayList          ArrayList<Fruta> frutas = new ArrayList<Fruta>();          Fruta f1 = new Fruta("Laranja", "Amarela");          Fruta f2 = new Fruta("Abacate", "Verde");          Fruta f3 = new Fruta("Morango", "Vermelho");          frutas.add(f1);          frutas.add(f2);          frutas.add(f3);            try {              s.serializar("/home/gustavo/frutas", frutas);              frutas = null;              frutas = (ArrayList<Fruta>) d.deserializar("/home/gustavo/frutas");              for(Fruta f : frutas){                  System.out.println("ArrayList: " + f.getNome() + " - " + f.getCor());              }          } catch (Exception ex) {              System.err.println("Falha ao serializar ou deserializar! - " +                                 ex.toString());          } |

**4. PARTICULARIDADES DA SERIALIZAÇÃO**

Como dito, a serialização de um objeto em Java pode ser algo fácil de se fazer. Mas existem certas informações que são importantes e sabê-las pode evitar vários problemas.

Um dos problemas é que nem tudo pode ser serializado, como objetos das classes “Thread” e “Socket” e atributos estáticos. Os atributos estáticos estão vinculados à classe e não ao objeto. Assim quando se trata do estado do objeto, os campos estáticos não são considerados. Em relação às Threads, suas instâncias só são úteis enquanto estiverem em execução na JVM, e podem ser paradas/destruídas e iniciadas com instruções de sua classe. Não existe algo no estado destas instâncias que seja útil para se persistir, e por isso não há sentido em persistir tais instâncias para restaurar posteriormente.

Mas e se existir alguma classe em que seja necessário persisti-la, e esta possui uma instância da classe “Thread”? Neste caso, utilizando um recurso pode-se persistir a classe inteira, menos a instância thread. Este recurso, refere-se em indicar que tal instância thread é um objeto “Transient”. Na Listagem 6 encontra-se um exemplo da implementação com uso de “Transient”.

Listagem 6 - Uso do Transient

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | import java.io.Serializable;    public class ImplementacaoTransient implements Serializable, Runnable{        transient private Thread thread;      //marcado com "transient" para que sua instância não seja serializada        //Demais valores de estado serão serializados      private int valorParaPersistir;        public ImplementacaoTransient(int valorParaPersistir){          this.valorParaPersistir = valorParaPersistir;          thread = new Thread(this);          thread.start();      }        public void run()      {          while(true){              //código a ser executado pela Thread          }      }  } |

Vale ressaltar que, o que é serializado é apenas o estado do objeto, e não os métodos e a classe. No exemplo acima, apenas a instância do objeto “thread” e do objeto “valorParaPersistir” compunham o estado do objeto. Assim, evita-se que o sistema de serialização tente serializar um objeto não-serializável e gere erros. Após a deserialização de um objeto da classe “ImplementacaoTransient” este estará totalmente funcional.

Um outro problema que pode vir a ocorrer, se refere ao versionamento das classes. Toda classe possui um atributo oculto chamado “serialVersionUID” com um valor gerado automaticamente. Objetos criados levam consigo o “serialVersionUID” de sua classe, que indica sua versão. Toda vez que uma classe sofre alterações em seu código, um novo “serialVersionUID” é gerado. O problema com a serialização surge quando serializa-se o estado de um objeto e posteriormente tenta-se restaurar o estado para a aplicação, mas houveram mudanças no código da classe. Neste caso, uma exceção “java.io.InvalidClassException” é disparada, pois de fato, o estado do objeto corresponde a uma versão de classe do qual foi criado que já não existe mais. Portanto, isso só ocorre para evitar erros de programação, em que tenta-se restaurar o estado de um objeto para uma classe que possui versão diferente da qual o objeto foi serializado inicialmente. E isso pode significar diferentes versões de uma mesma classe “Fruta” por exemplo, ou diferentes versões por classes serem totalmente diferentes como: o estado de um objeto da classe “Fruta” ser restaurado para classe “Carro”.

Mas quando é necessário, conscientemente, recuperar o estado de um objeto, mesmo que houveram alterações na classe do qual foi gerado, é possível controlar manualmente o “serialVersionUID”. Para isto, basta declarar explicitamente na forma de atributo estático e final, um valor Long qualquer para corresponder à versão da classe. E assim, mesmo que houverem modificações na classe, a versão permenecerá a mesma, evitando as exceções da incompatiblidade de classes. No entanto, vale ressaltar que o versionamento não é um impecílio para o programador, mas sim muito importante em casos que deseja-se manter um controle sobre a compatibilidade de código na aplicação, definindo versões para classes No código da Listagem 7 encontra-se o exemplo do controle manual do “serialVersionUID”.

Listagem 7 - Uso manual do serialVersionUID

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | import java.io.Serializable;    public class Fruta implements Serializable{      private String nome;      private String cor;      static final long serialVersionUID = 123L; //indica a versao da classe        public Fruta(String nome, String cor) {          this.nome = nome;          this.cor = cor;      }      public String getCor() { return cor; }      public String getNome() { return nome; }      public void setCor(String cor) { this.cor = cor; }      public void setNome(String nome) { this.nome = nome; }  } |

O JDK fornece uma ferramenta que permite visualizar a versão de qualquer classe Java. Basta no terminal do sistema operacional utilizar o comando “serialver” seguido da classe qualificada, conforme exemplo da Listagem 8.

Listagem 8 - Verificação da versão de classe

|  |  |
| --- | --- |
| 1 | serialver pojo.Fruta |

A serialização acaba inflingindo alguns modelos de segurança em seu procedimento padrão, e criando então um novo problema. Quando um estado de objeto é persistido no sistema de arquivos ou enviado por rede, o fluxo binário pode ser detectado, analisado e obtidas informações sobre a classe e o estado do objeto. Por exemplo, se utilizado um editor de texto qualquer para abrir o arquivo persistido do estado do objeto da classe “Fruta” dos exemplos anteriores, é comum os atributos e outras informações aparecerem como um simples texto, total ou parcilamente legíveis, assim como demonstrado na Listagem 9.

Listagem 9 - Exemplo de leitura de arquivo binário de um estado de objeto persistido.

|  |  |
| --- | --- |
| 1  2 | ��##sr#  pojo.Fruta#######{###L##cort##Ljava/lang/String;L##nomeq#~##xpt##Vermelhat##Maça |

Com uma ferramenta menos simples seria possível aumentar a quantidade de informações legíveis, coletando uma maior quantia de dados.

No entando, o processo de serialização permite que haja uma certa personalização da serialização e deserialização de um objeto. Isso deve-se a possibilidade da sobrescrição dos métodos “writeObject” e “readObject” dentro da classe a ser serializada, como a classe exemplar “pojo.Fruta”. Esta sobrescrita permite um controle total sobre a serialização, pois as classes “java.io.ObjectOutputStream” e “java.io.ObjectInputStream”, ao serializar o estado de um objeto, primeiramente verificam se esse método foi sobrescrito na classe alvo e assim o utilizam, senão, será utilizado o protocolo padrão de serialização.

Realizando a sobrescrição dos métodos é possível criptografar os dados na serialização e descriptografar os dados na deserialização, por meio de algum algoritmo que o programador venha implementar. Assim os dados podem ser persistidos ou transmitidos por rede, sem que seja possível ler as partes mais importantes do estado de um objeto. Na listagem 10, a classe “pojo.Fruta” foi modificada dando origem a classe “pojo.FrutaSobrescrita” que possui adicionamente a sobrescrição dos métodos “writeObject” e “readObject”. Como este é apenas um exemplo, não foi escrito um verdadeiro algoritmo de criptografia, mas sim apenas uma troca no valor da String do campo “cor”, identificando que houveram alterações tanto no processo de serialização quanto no de deserialização. Na listagem 11 encontra-se a execução da serialização.

Listagem 10 - Sobrescrita dos métodos “writeObject” e “readObject”.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36 | import java.io.Serializable;    public class FrutaSobrescrita implements Serializable{      private String nome;      private String cor;      static final long serialVersionUID = 123L; //indica a versao da classe        public FrutaSobrescrita(String nome, String cor) {          this.nome = nome;          this.cor = cor;      }      public String getCor() { return cor; }      public String getNome() { return nome; }      public void setCor(String cor) { this.cor = cor; }      public void setNome(String nome) { this.nome = nome; }        private void writeObject(java.io.ObjectOutputStream stream) throws java.io.IOException      {          // criptografar os dados antes de liberar o stream          // EXEMPLO: setCor(criptografar(getCor()));          setCor(getCor()+ " > CRIPTOGRAFADO"); // apenas concatenação de string para indicar alteração            // liberar o stream          stream.defaultWriteObject();      }        private void readObject(java.io.ObjectInputStream stream) throws java.io.IOException, ClassNotFoundException      {          // liberar stream          stream.defaultReadObject();            // descriptografar os dados          // EXEMPLO: setCor(descriptografar(getCor()));          setCor(getCor()+ " > DESCRIPTOGRAFADO"); // apenas contatenação de string para indicar alteração      }  } |

Listagem 11 - Execução da serialização da classe “pojo.FrutaSobrescrita”.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | //serializa        Serializador s = new Serializador();        FrutaSobrescrita frutaSobrescrita = new FrutaSobrescrita("Maça", "Vermelha");         try {             s.serializar("/home/gustavo/frutaSobrescrita", frutaSobrescrita);         } catch (Exception ex) {             System.err.println("Falha ao serializar! - " + ex.toString());         }           //deserializa        Deserializador d = new Deserializador();         frutaSobrescrita = null;         try {             frutaSobrescrita = (FrutaSobrescrita) d.deserializar("/home/gustavo/frutaSobrescrita");         } catch (Exception ex) {             System.err.println("Falha ao deserializar! - " + ex.toString());         }         System.out.println("Sobrescrita: " + frutaSobrescrita.getNome() + " - " + frutaSobrescrita.getCor()); |

**5. CONCLUSÃO**

A serialização é um procedimento de simples execução por parte do programador. Com pouco tempo de codificação é possível implementar a persistencia de um estado de objeto para o sistema de arquivos ou de transmissão por rede. É possível reproduzir os exemplos criados neste artigo, em situações até mesmo mais elaboradas. A apresentação de alguns problemas e soluções trazidas por particularidades, abre o entendimento de que a serialização pode deixar de ser simples conforme for aplicada em diferentes contextos, e que ao mesmo tempo é muito flexível ao permitir ao programador implementar diferentes soluções.

[Assista ao vídeo Serialização/Deserialização de objetos com YAML](http://www.devmedia.com.br/serializacao-deserializacao-de-objetos-com-yaml-parte-1/8399)

**6. BIBLIOGRAFIA**

ORACLE - **java.io Interface Serializable** - Disponível em: <http://docs.oracle.com/javase/1.4.2/docs/api/java/io/Serializable.html>Acesso em: 11/03/2012

ORACLE - **Discover de Secrets of Java Serialization API** - Disponível em: <http://java.sun.com/developer/technicalArticles/Programming/serialization/> Acesso em: 11/03/2012

NEWARD, Ted - **5 coisas que você não sabia sobre... Serialização de Objetos Java** - Disponível em: <http://www.ibm.com/developerworks/br/library/j-5things1/index.html> Acesso em: 11/03/2012

DOWNS, Andrew - **Java Serialization** - Disponível em: <http://www.mactech.com/articles/mactech/Vol.14/14.04/JavaSerialization/index.html> Acesso em: 11/03/2012

PREISSLER, Luciano E.; GEYER, Cláudio F. R. - **Serialização em Java** - Disponível em: <http://www.inf.ufrgs.br/gppd/disc/cmp167/trabalhos/sem99-1/T1/luciano/final.htm> Acesso em: 11/03/2012

[[](http://www.linhadecodigo.com.br/colaborador.aspx?id=354)](http://www.linhadecodigo.com.br/colaborador.aspx?id=354)

[Everton Coimbra de Araújo - Desde 1987 atua na área de treinamento e desenvolvimento. Como Mestre em Ciência da Computação, é professor da UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ, Campus Medianeira, onde leciona disciplinas relacionadas ao desenvolvimento de aplicações web, com Java e .NET.   
  
É autor dos livros Desenvolvimento para WEB com Java, Orientação a Objetos com Java - Simples, Fácil e Eficiente, Algoritmo - Fundamento e Prática em sua terceira edição e dos livros Delphi - Implementação e Técnicas para Ambientes Virtuais e C++ Builder - Implementação e Técnicas para Ambientes Virtuais. Todos pela VisualBooks. Pode ser contactado através do e-mail everton@utfpr.edu.br ou evertoncoimbra@gmail.com.](http://www.linhadecodigo.com.br/colaborador.aspx?id=354)

 85 3 0 4

**Leia também**

* [Conhecendo a interface Map do Java](http://www.linhadecodigo.com.br/artigo/3670/conhecendo-a-interface-map-do-java.aspx)*Java*
* [Trabalhando com a Interface Set no Java](http://www.linhadecodigo.com.br/artigo/3669/trabalhando-com-a-interface-set-no-java.aspx)*Java*
* [Classes Wrappers em Java](http://www.linhadecodigo.com.br/artigo/3667/classes-wrappers-em-java.aspx)*Java*
* [ComboBoxModel em Java](http://www.linhadecodigo.com.br/artigo/3664/comboboxmodel-em-java.aspx)*Java*
* [Reduzindo a Mutabilidade em Java](http://www.linhadecodigo.com.br/artigo/3659/reduzindo-a-mutabilidade-em-java.aspx)*Java*

Publicidade

[](http://qconsp.com/?utm_source=devmedia&utm_medium=banner&utm_campaign=devmedia)

Linha de Código[Anuncie](http://www.web-03.net/linhadecodigo.asp)|[Fale Conosco](mailto:colabore@linhadecodigo.com.br)|[Publique](http://www.linhadecodigo.com.br/publique.aspx)

|  |  |
| --- | --- |
| http://www.linhadecodigo.com.br/artigos/img_artigos/EvertonCoimbraAraujo/Serializazao_Objetos/imagem001.jpg | Neste artigo apresento o Gustavo Rafael Valiati, que pode ser contatado pelo e-mail gustavovaliati@gmail.com. O Gustavo foi meu aluno em algumas disciplinas de sua gradução e está concluíndo o último período do curso de Tecnologia em Análise e Desenvolvimento de Sistemas da UTFPR campus Medianeira, neste primeiro semestre de 2012. |

**1. INTRODUÇÃO**

Nem sempre se deseja ter objetos ou estrutura de dados de um aplicativo, com um ciclo de vida que dura no máximo o tempo em que for armazenado em memória RAM. Ou seja, algumas vezes deseja-se guardá-los por um tempo indeterminado, sendo possível recuperá-los posteriormente. E neste caso, o termo “persistência” tem seu conceito definido.

Persistir dados, requer que estes possam ser serializáveis. A serialização possibilitará capturar o estado do objeto ou a estrutura de dados, e transformar em uma cadeia de bytes quando for necessário. Torna-se possível também, recuperar os bytes persistidos para fazer o processo inverso, e ter os dados de volta para a aplicação em execução.

Desta maneira, qualquer forma de dados que esteja sendo trabalhada na aplicação, e esta puder ser serializada, poderá ser passada para o meio físico, e ter o processo revertido.

A serialização não se limita apenas em possibilitar a gravação do dado em disco, mas permite que o mesmo seja transmitido por rede. Isto porquê a serialização cria um stream de bytes, que é requisito para transmissão de dados por rede.

Diversas linguagens possuem suporte à serialização, como: Java, C, C++, Python, PHP, .NET, entre outras. Neste artigo, será tratado apenas da serialização com Java, em sua implemetação básica, alguns problemas e soluções, com códigos para exemplificação.

**2. SERIALIZAÇÃO COM JAVA**

Serializar um objeto, em Java, só é possível caso sua classe esteja marcada como serializável. Essa marcação é feita com a simples implementação da interface “java.io.Serializable”. Essa interface não possui métodos para se implementar e nem atributos. Ainda não existiam as anotações no Java 1.1, versão que trouxe pela primeira vez a serialização ao Java, e desta maneira a implementação da interface era a melhor maneira de marcar uma classe.

A herança entre classes é naturalmente afetada pela serialização. Já que quando uma classe implementa a interface “Serializable”, toda classe que dela estender também estará implicitamente marcada como serializável

.

Para exemplo, será criada uma classe “pojo.Fruta” que contém dois atributos: nome e cor. No código exemplo (Listagem 1), pode-se notar a classe “Fruta” marcada para a serialização:

Listagem 1 - Marcando uma classe como serializável.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | import java.io.Serializable;    public class Fruta implements Serializable{      private String nome;      private String cor;        public Fruta(String nome, String cor) {          this.nome = nome;          this.cor = cor;      }        public String getCor() { return cor; }      public String getNome() { return nome; }      public void setCor(String cor) { this.cor = cor; }      public void setNome(String nome) { this.nome = nome; }  } |

Pode-se dizer que a implementação do processo de serialização e deserialização simples de um objeto, não possui um código difícil de se escrever. Isto porquê duas classes fazem a maior parte do trabalho, de maneira que com poucas linhas de código pode-se transformar um objeto em bytes e vice-versa. Essas duas classes são “java.io.ObjectInputStream” que recebe bytes para criar um objeto e “java.io.ObjectOutputStream” que cria uma cadeia de bytes a partir de um objeto.

**3. SERIALIZANDO E DESERIALIZANDO O ESTADO DE UM OBJETO**

Como exemplo, será persistido um objeto em disco. Para isto, além das classes “java.io.ObjectInputStream” e “java.io.ObjectOutputStream”, serão utilizadas duas outras que trabalham com arquivos, que são “java.io.FileInputStream” e “java.io.FileOutputStream”, que seguem a mesma lógica de InputStream e OutputStream, só que agora relacionado ao sistema de arquivos e não aos objetos.

Para serialização, deve-se criar um novo “java.io.FileOutputStream”, com o caminho (“path”) do arquivo onde se deseja armazenar o estado do objeto. Também deve ser criado um novo “java.io.ObjectOutputStream” que está ligado ao arquivo. E por último deve-se executar a escrita do objeto alvo, para dentro do objeto de stream que por sua vez está ligado ao arquivo destino. Após executados esses passos, é possível encontrar no sistema de arquivos um novo arquivo que contém os bytes correspondentes à descrição do estado do objeto. Na Listagem 2, encontra-se o código exemplo.

Listagem 2 - Serialização de um objeto.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | import java.io.FileOutputStream;  import java.io.ObjectOutputStream;    public class Serializador {        public Serializador() {    }        public void serializar(String path, Object obj) throws Exception {              FileOutputStream outFile = new FileOutputStream(path);      ObjectOutputStream s = new ObjectOutputStream(outFile);      s.writeObject(obj);      s.close();      }  } |

E para a deserialização, o processo é extremamente parecido com a serialização. Cria-se um novo “java.io.FileInputStream” relacionado ao arquivo onde o estado do objeto encontra-se persistido. Deve-se criar também um novo “java.io.ObjectInputStream” que estará vinculado ao arquivo de origem. Por último, executa-se a leitura do arquivo para a restauração do estado. Na Listagem 3 encontra-se o exemplo do código.

Listagem 3 - Deserializaçãode um objeto.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | import java.io.FileInputStream;  import java.io.ObjectInputStream;    public class Deserializador {        public Deserializador() {    }        public Object deserializar(String path) throws Exception {      FileInputStream inFile = new FileInputStream(path);      ObjectInputStream d = new ObjectInputStream(inFile);      Object o = d.readObject();      d.close();      return o;      }  } |

A Listagem 4, contém o exemplo da execução da serialização e em seguida da deserialização.

Listagem 4 - Serializando e Deserializando

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | import operacoes.Deserializador;  import operacoes.Serializador;  import pojo.Fruta;    public class Principal {      public static void main(String args[]){         //serializa         Serializador s = new Serializador();         Fruta fruta = new Fruta("Maça", "Vermelha");          try {              s.serializar("/home/gustavo/fruta", fruta);          } catch (Exception ex) {              System.err.println("Falha ao serializar! - " + ex.toString());          }            //deserializa          Deserializador d = new Deserializador();          fruta = null;          try {              fruta = (Fruta) d.deserializar("/home/gustavo/fruta");          } catch (Exception ex) {              System.err.println("Falha ao deserializar! - " + ex.toString());          }          System.out.println(fruta.getNome() + " - " + fruta.getCor());      }  } |

Na Listagem 5, encontra-se um exemplo de serialização e deserialização de um ArrayList da classe Fruta. É apenas mais um exemplo que demonstra a capacidade de se persistir qualquer objeto que seja serializável. Essa serialização só foi possível porque a classe “ArrayList” implementa a interface “Serializable” por padrão.

Listagem 5 - Serialização de ArrayList.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | //serialização e deserialização com ArrayList          ArrayList<Fruta> frutas = new ArrayList<Fruta>();          Fruta f1 = new Fruta("Laranja", "Amarela");          Fruta f2 = new Fruta("Abacate", "Verde");          Fruta f3 = new Fruta("Morango", "Vermelho");          frutas.add(f1);          frutas.add(f2);          frutas.add(f3);            try {              s.serializar("/home/gustavo/frutas", frutas);              frutas = null;              frutas = (ArrayList<Fruta>) d.deserializar("/home/gustavo/frutas");              for(Fruta f : frutas){                  System.out.println("ArrayList: " + f.getNome() + " - " + f.getCor());              }          } catch (Exception ex) {              System.err.println("Falha ao serializar ou deserializar! - " +                                 ex.toString());          } |

**4. PARTICULARIDADES DA SERIALIZAÇÃO**

Como dito, a serialização de um objeto em Java pode ser algo fácil de se fazer. Mas existem certas informações que são importantes e sabê-las pode evitar vários problemas.

Um dos problemas é que nem tudo pode ser serializado, como objetos das classes “Thread” e “Socket” e atributos estáticos. Os atributos estáticos estão vinculados à classe e não ao objeto. Assim quando se trata do estado do objeto, os campos estáticos não são considerados. Em relação às Threads, suas instâncias só são úteis enquanto estiverem em execução na JVM, e podem ser paradas/destruídas e iniciadas com instruções de sua classe. Não existe algo no estado destas instâncias que seja útil para se persistir, e por isso não há sentido em persistir tais instâncias para restaurar posteriormente.

Mas e se existir alguma classe em que seja necessário persisti-la, e esta possui uma instância da classe “Thread”? Neste caso, utilizando um recurso pode-se persistir a classe inteira, menos a instância thread. Este recurso, refere-se em indicar que tal instância thread é um objeto “Transient”. Na Listagem 6 encontra-se um exemplo da implementação com uso de “Transient”.

Listagem 6 - Uso do Transient

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | import java.io.Serializable;    public class ImplementacaoTransient implements Serializable, Runnable{        transient private Thread thread;      //marcado com "transient" para que sua instância não seja serializada        //Demais valores de estado serão serializados      private int valorParaPersistir;        public ImplementacaoTransient(int valorParaPersistir){          this.valorParaPersistir = valorParaPersistir;          thread = new Thread(this);          thread.start();      }        public void run()      {          while(true){              //código a ser executado pela Thread          }      }  } |

Vale ressaltar que, o que é serializado é apenas o estado do objeto, e não os métodos e a classe. No exemplo acima, apenas a instância do objeto “thread” e do objeto “valorParaPersistir” compunham o estado do objeto. Assim, evita-se que o sistema de serialização tente serializar um objeto não-serializável e gere erros. Após a deserialização de um objeto da classe “ImplementacaoTransient” este estará totalmente funcional.

Um outro problema que pode vir a ocorrer, se refere ao versionamento das classes. Toda classe possui um atributo oculto chamado “serialVersionUID” com um valor gerado automaticamente. Objetos criados levam consigo o “serialVersionUID” de sua classe, que indica sua versão. Toda vez que uma classe sofre alterações em seu código, um novo “serialVersionUID” é gerado. O problema com a serialização surge quando serializa-se o estado de um objeto e posteriormente tenta-se restaurar o estado para a aplicação, mas houveram mudanças no código da classe. Neste caso, uma exceção “java.io.InvalidClassException” é disparada, pois de fato, o estado do objeto corresponde a uma versão de classe do qual foi criado que já não existe mais. Portanto, isso só ocorre para evitar erros de programação, em que tenta-se restaurar o estado de um objeto para uma classe que possui versão diferente da qual o objeto foi serializado inicialmente. E isso pode significar diferentes versões de uma mesma classe “Fruta” por exemplo, ou diferentes versões por classes serem totalmente diferentes como: o estado de um objeto da classe “Fruta” ser restaurado para classe “Carro”.

Mas quando é necessário, conscientemente, recuperar o estado de um objeto, mesmo que houveram alterações na classe do qual foi gerado, é possível controlar manualmente o “serialVersionUID”. Para isto, basta declarar explicitamente na forma de atributo estático e final, um valor Long qualquer para corresponder à versão da classe. E assim, mesmo que houverem modificações na classe, a versão permenecerá a mesma, evitando as exceções da incompatiblidade de classes. No entanto, vale ressaltar que o versionamento não é um impecílio para o programador, mas sim muito importante em casos que deseja-se manter um controle sobre a compatibilidade de código na aplicação, definindo versões para classes No código da Listagem 7 encontra-se o exemplo do controle manual do “serialVersionUID”.

Listagem 7 - Uso manual do serialVersionUID

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | import java.io.Serializable;    public class Fruta implements Serializable{      private String nome;      private String cor;      static final long serialVersionUID = 123L; //indica a versao da classe        public Fruta(String nome, String cor) {          this.nome = nome;          this.cor = cor;      }      public String getCor() { return cor; }      public String getNome() { return nome; }      public void setCor(String cor) { this.cor = cor; }      public void setNome(String nome) { this.nome = nome; }  } |

O JDK fornece uma ferramenta que permite visualizar a versão de qualquer classe Java. Basta no terminal do sistema operacional utilizar o comando “serialver” seguido da classe qualificada, conforme exemplo da Listagem 8.

Listagem 8 - Verificação da versão de classe

|  |  |
| --- | --- |
| 1 | serialver pojo.Fruta |

A serialização acaba inflingindo alguns modelos de segurança em seu procedimento padrão, e criando então um novo problema. Quando um estado de objeto é persistido no sistema de arquivos ou enviado por rede, o fluxo binário pode ser detectado, analisado e obtidas informações sobre a classe e o estado do objeto. Por exemplo, se utilizado um editor de texto qualquer para abrir o arquivo persistido do estado do objeto da classe “Fruta” dos exemplos anteriores, é comum os atributos e outras informações aparecerem como um simples texto, total ou parcilamente legíveis, assim como demonstrado na Listagem 9.

Listagem 9 - Exemplo de leitura de arquivo binário de um estado de objeto persistido.

|  |  |
| --- | --- |
| 1  2 | ��##sr#  pojo.Fruta#######{###L##cort##Ljava/lang/String;L##nomeq#~##xpt##Vermelhat##Maça |

Com uma ferramenta menos simples seria possível aumentar a quantidade de informações legíveis, coletando uma maior quantia de dados.

No entando, o processo de serialização permite que haja uma certa personalização da serialização e deserialização de um objeto. Isso deve-se a possibilidade da sobrescrição dos métodos “writeObject” e “readObject” dentro da classe a ser serializada, como a classe exemplar “pojo.Fruta”. Esta sobrescrita permite um controle total sobre a serialização, pois as classes “java.io.ObjectOutputStream” e “java.io.ObjectInputStream”, ao serializar o estado de um objeto, primeiramente verificam se esse método foi sobrescrito na classe alvo e assim o utilizam, senão, será utilizado o protocolo padrão de serialização.

Realizando a sobrescrição dos métodos é possível criptografar os dados na serialização e descriptografar os dados na deserialização, por meio de algum algoritmo que o programador venha implementar. Assim os dados podem ser persistidos ou transmitidos por rede, sem que seja possível ler as partes mais importantes do estado de um objeto. Na listagem 10, a classe “pojo.Fruta” foi modificada dando origem a classe “pojo.FrutaSobrescrita” que possui adicionamente a sobrescrição dos métodos “writeObject” e “readObject”. Como este é apenas um exemplo, não foi escrito um verdadeiro algoritmo de criptografia, mas sim apenas uma troca no valor da String do campo “cor”, identificando que houveram alterações tanto no processo de serialização quanto no de deserialização. Na listagem 11 encontra-se a execução da serialização.

Listagem 10 - Sobrescrita dos métodos “writeObject” e “readObject”.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36 | import java.io.Serializable;    public class FrutaSobrescrita implements Serializable{      private String nome;      private String cor;      static final long serialVersionUID = 123L; //indica a versao da classe        public FrutaSobrescrita(String nome, String cor) {          this.nome = nome;          this.cor = cor;      }      public String getCor() { return cor; }      public String getNome() { return nome; }      public void setCor(String cor) { this.cor = cor; }      public void setNome(String nome) { this.nome = nome; }        private void writeObject(java.io.ObjectOutputStream stream) throws java.io.IOException      {          // criptografar os dados antes de liberar o stream          // EXEMPLO: setCor(criptografar(getCor()));          setCor(getCor()+ " > CRIPTOGRAFADO"); // apenas concatenação de string para indicar alteração            // liberar o stream          stream.defaultWriteObject();      }        private void readObject(java.io.ObjectInputStream stream) throws java.io.IOException, ClassNotFoundException      {          // liberar stream          stream.defaultReadObject();            // descriptografar os dados          // EXEMPLO: setCor(descriptografar(getCor()));          setCor(getCor()+ " > DESCRIPTOGRAFADO"); // apenas contatenação de string para indicar alteração      }  } |

Listagem 11 - Execução da serialização da classe “pojo.FrutaSobrescrita”.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | //serializa        Serializador s = new Serializador();        FrutaSobrescrita frutaSobrescrita = new FrutaSobrescrita("Maça", "Vermelha");         try {             s.serializar("/home/gustavo/frutaSobrescrita", frutaSobrescrita);         } catch (Exception ex) {             System.err.println("Falha ao serializar! - " + ex.toString());         }           //deserializa        Deserializador d = new Deserializador();         frutaSobrescrita = null;         try {             frutaSobrescrita = (FrutaSobrescrita) d.deserializar("/home/gustavo/frutaSobrescrita");         } catch (Exception ex) {             System.err.println("Falha ao deserializar! - " + ex.toString());         }         System.out.println("Sobrescrita: " + frutaSobrescrita.getNome() + " - " + frutaSobrescrita.getCor()); |

**5. CONCLUSÃO**

A serialização é um procedimento de simples execução por parte do programador. Com pouco tempo de codificação é possível implementar a persistencia de um estado de objeto para o sistema de arquivos ou de transmissão por rede. É possível reproduzir os exemplos criados neste artigo, em situações até mesmo mais elaboradas. A apresentação de alguns problemas e soluções trazidas por particularidades, abre o entendimento de que a serialização pode deixar de ser simples conforme for aplicada em diferentes contextos, e que ao mesmo tempo é muito flexível ao permitir ao programador implementar diferentes soluções.

[Assista ao vídeo Serialização/Deserialização de objetos com YAML](http://www.devmedia.com.br/serializacao-deserializacao-de-objetos-com-yaml-parte-1/8399)

**6. BIBLIOGRAFIA**

ORACLE - **java.io Interface Serializable** - Disponível em: <http://docs.oracle.com/javase/1.4.2/docs/api/java/io/Serializable.html>Acesso em: 11/03/2012

ORACLE - **Discover de Secrets of Java Serialization API** - Disponível em: <http://java.sun.com/developer/technicalArticles/Programming/serialization/> Acesso em: 11/03/2012

NEWARD, Ted - **5 coisas que você não sabia sobre... Serialização de Objetos Java** - Disponível em: <http://www.ibm.com/developerworks/br/library/j-5things1/index.html> Acesso em: 11/03/2012

DOWNS, Andrew - **Java Serialization** - Disponível em: <http://www.mactech.com/articles/mactech/Vol.14/14.04/JavaSerialization/index.html> Acesso em: 11/03/2012

PREISSLER, Luciano E.; GEYER, Cláudio F. R. - **Serialização em Java** - Disponível em: <http://www.inf.ufrgs.br/gppd/disc/cmp167/trabalhos/sem99-1/T1/luciano/final.htm> Acesso em: 11/03/2012

[[](http://www.linhadecodigo.com.br/colaborador.aspx?id=354)](http://www.linhadecodigo.com.br/colaborador.aspx?id=354)

[Everton Coimbra de Araújo - Desde 1987 atua na área de treinamento e desenvolvimento. Como Mestre em Ciência da Computação, é professor da UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ, Campus Medianeira, onde leciona disciplinas relacionadas ao desenvolvimento de aplicações web, com Java e .NET.   
  
É autor dos livros Desenvolvimento para WEB com Java, Orientação a Objetos com Java - Simples, Fácil e Eficiente, Algoritmo - Fundamento e Prática em sua terceira edição e dos livros Delphi - Implementação e Técnicas para Ambientes Virtuais e C++ Builder - Implementação e Técnicas para Ambientes Virtuais. Todos pela VisualBooks. Pode ser contactado através do e-mail everton@utfpr.edu.br ou evertoncoimbra@gmail.com.](http://www.linhadecodigo.com.br/colaborador.aspx?id=354)

 85 3 0 4

**Leia também**

* [Conhecendo a interface Map do Java](http://www.linhadecodigo.com.br/artigo/3670/conhecendo-a-interface-map-do-java.aspx)*Java*
* [Trabalhando com a Interface Set no Java](http://www.linhadecodigo.com.br/artigo/3669/trabalhando-com-a-interface-set-no-java.aspx)*Java*
* [Classes Wrappers em Java](http://www.linhadecodigo.com.br/artigo/3667/classes-wrappers-em-java.aspx)*Java*
* [ComboBoxModel em Java](http://www.linhadecodigo.com.br/artigo/3664/comboboxmodel-em-java.aspx)*Java*
* [Reduzindo a Mutabilidade em Java](http://www.linhadecodigo.com.br/artigo/3659/reduzindo-a-mutabilidade-em-java.aspx)*Java*

Publicidade

[](http://qconsp.com/?utm_source=devmedia&utm_medium=banner&utm_campaign=devmedia)

Linha de Código[Anuncie](http://www.web-03.net/linhadecodigo.asp)|[Fale Conosco](mailto:colabore@linhadecodigo.com.br)|[Publique](http://www.linhadecodigo.com.br/publique.aspx)

|  |  |
| --- | --- |
| http://www.linhadecodigo.com.br/artigos/img_artigos/EvertonCoimbraAraujo/Serializazao_Objetos/imagem001.jpg | Neste artigo apresento o Gustavo Rafael Valiati, que pode ser contatado pelo e-mail gustavovaliati@gmail.com. O Gustavo foi meu aluno em algumas disciplinas de sua gradução e está concluíndo o último período do curso de Tecnologia em Análise e Desenvolvimento de Sistemas da UTFPR campus Medianeira, neste primeiro semestre de 2012. |

**1. INTRODUÇÃO**

Nem sempre se deseja ter objetos ou estrutura de dados de um aplicativo, com um ciclo de vida que dura no máximo o tempo em que for armazenado em memória RAM. Ou seja, algumas vezes deseja-se guardá-los por um tempo indeterminado, sendo possível recuperá-los posteriormente. E neste caso, o termo “persistência” tem seu conceito definido.

Persistir dados, requer que estes possam ser serializáveis. A serialização possibilitará capturar o estado do objeto ou a estrutura de dados, e transformar em uma cadeia de bytes quando for necessário. Torna-se possível também, recuperar os bytes persistidos para fazer o processo inverso, e ter os dados de volta para a aplicação em execução.

Desta maneira, qualquer forma de dados que esteja sendo trabalhada na aplicação, e esta puder ser serializada, poderá ser passada para o meio físico, e ter o processo revertido.

A serialização não se limita apenas em possibilitar a gravação do dado em disco, mas permite que o mesmo seja transmitido por rede. Isto porquê a serialização cria um stream de bytes, que é requisito para transmissão de dados por rede.

Diversas linguagens possuem suporte à serialização, como: Java, C, C++, Python, PHP, .NET, entre outras. Neste artigo, será tratado apenas da serialização com Java, em sua implemetação básica, alguns problemas e soluções, com códigos para exemplificação.

**2. SERIALIZAÇÃO COM JAVA**

Serializar um objeto, em Java, só é possível caso sua classe esteja marcada como serializável. Essa marcação é feita com a simples implementação da interface “java.io.Serializable”. Essa interface não possui métodos para se implementar e nem atributos. Ainda não existiam as anotações no Java 1.1, versão que trouxe pela primeira vez a serialização ao Java, e desta maneira a implementação da interface era a melhor maneira de marcar uma classe.

A herança entre classes é naturalmente afetada pela serialização. Já que quando uma classe implementa a interface “Serializable”, toda classe que dela estender também estará implicitamente marcada como serializável

.

Para exemplo, será criada uma classe “pojo.Fruta” que contém dois atributos: nome e cor. No código exemplo (Listagem 1), pode-se notar a classe “Fruta” marcada para a serialização:

Listagem 1 - Marcando uma classe como serializável.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | import java.io.Serializable;    public class Fruta implements Serializable{      private String nome;      private String cor;        public Fruta(String nome, String cor) {          this.nome = nome;          this.cor = cor;      }        public String getCor() { return cor; }      public String getNome() { return nome; }      public void setCor(String cor) { this.cor = cor; }      public void setNome(String nome) { this.nome = nome; }  } |

Pode-se dizer que a implementação do processo de serialização e deserialização simples de um objeto, não possui um código difícil de se escrever. Isto porquê duas classes fazem a maior parte do trabalho, de maneira que com poucas linhas de código pode-se transformar um objeto em bytes e vice-versa. Essas duas classes são “java.io.ObjectInputStream” que recebe bytes para criar um objeto e “java.io.ObjectOutputStream” que cria uma cadeia de bytes a partir de um objeto.

**3. SERIALIZANDO E DESERIALIZANDO O ESTADO DE UM OBJETO**

Como exemplo, será persistido um objeto em disco. Para isto, além das classes “java.io.ObjectInputStream” e “java.io.ObjectOutputStream”, serão utilizadas duas outras que trabalham com arquivos, que são “java.io.FileInputStream” e “java.io.FileOutputStream”, que seguem a mesma lógica de InputStream e OutputStream, só que agora relacionado ao sistema de arquivos e não aos objetos.

Para serialização, deve-se criar um novo “java.io.FileOutputStream”, com o caminho (“path”) do arquivo onde se deseja armazenar o estado do objeto. Também deve ser criado um novo “java.io.ObjectOutputStream” que está ligado ao arquivo. E por último deve-se executar a escrita do objeto alvo, para dentro do objeto de stream que por sua vez está ligado ao arquivo destino. Após executados esses passos, é possível encontrar no sistema de arquivos um novo arquivo que contém os bytes correspondentes à descrição do estado do objeto. Na Listagem 2, encontra-se o código exemplo.

Listagem 2 - Serialização de um objeto.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | import java.io.FileOutputStream;  import java.io.ObjectOutputStream;    public class Serializador {        public Serializador() {    }        public void serializar(String path, Object obj) throws Exception {              FileOutputStream outFile = new FileOutputStream(path);      ObjectOutputStream s = new ObjectOutputStream(outFile);      s.writeObject(obj);      s.close();      }  } |

E para a deserialização, o processo é extremamente parecido com a serialização. Cria-se um novo “java.io.FileInputStream” relacionado ao arquivo onde o estado do objeto encontra-se persistido. Deve-se criar também um novo “java.io.ObjectInputStream” que estará vinculado ao arquivo de origem. Por último, executa-se a leitura do arquivo para a restauração do estado. Na Listagem 3 encontra-se o exemplo do código.

Listagem 3 - Deserializaçãode um objeto.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | import java.io.FileInputStream;  import java.io.ObjectInputStream;    public class Deserializador {        public Deserializador() {    }        public Object deserializar(String path) throws Exception {      FileInputStream inFile = new FileInputStream(path);      ObjectInputStream d = new ObjectInputStream(inFile);      Object o = d.readObject();      d.close();      return o;      }  } |

A Listagem 4, contém o exemplo da execução da serialização e em seguida da deserialização.

Listagem 4 - Serializando e Deserializando

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | import operacoes.Deserializador;  import operacoes.Serializador;  import pojo.Fruta;    public class Principal {      public static void main(String args[]){         //serializa         Serializador s = new Serializador();         Fruta fruta = new Fruta("Maça", "Vermelha");          try {              s.serializar("/home/gustavo/fruta", fruta);          } catch (Exception ex) {              System.err.println("Falha ao serializar! - " + ex.toString());          }            //deserializa          Deserializador d = new Deserializador();          fruta = null;          try {              fruta = (Fruta) d.deserializar("/home/gustavo/fruta");          } catch (Exception ex) {              System.err.println("Falha ao deserializar! - " + ex.toString());          }          System.out.println(fruta.getNome() + " - " + fruta.getCor());      }  } |

Na Listagem 5, encontra-se um exemplo de serialização e deserialização de um ArrayList da classe Fruta. É apenas mais um exemplo que demonstra a capacidade de se persistir qualquer objeto que seja serializável. Essa serialização só foi possível porque a classe “ArrayList” implementa a interface “Serializable” por padrão.

Listagem 5 - Serialização de ArrayList.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | //serialização e deserialização com ArrayList          ArrayList<Fruta> frutas = new ArrayList<Fruta>();          Fruta f1 = new Fruta("Laranja", "Amarela");          Fruta f2 = new Fruta("Abacate", "Verde");          Fruta f3 = new Fruta("Morango", "Vermelho");          frutas.add(f1);          frutas.add(f2);          frutas.add(f3);            try {              s.serializar("/home/gustavo/frutas", frutas);              frutas = null;              frutas = (ArrayList<Fruta>) d.deserializar("/home/gustavo/frutas");              for(Fruta f : frutas){                  System.out.println("ArrayList: " + f.getNome() + " - " + f.getCor());              }          } catch (Exception ex) {              System.err.println("Falha ao serializar ou deserializar! - " +                                 ex.toString());          } |

**4. PARTICULARIDADES DA SERIALIZAÇÃO**

Como dito, a serialização de um objeto em Java pode ser algo fácil de se fazer. Mas existem certas informações que são importantes e sabê-las pode evitar vários problemas.

Um dos problemas é que nem tudo pode ser serializado, como objetos das classes “Thread” e “Socket” e atributos estáticos. Os atributos estáticos estão vinculados à classe e não ao objeto. Assim quando se trata do estado do objeto, os campos estáticos não são considerados. Em relação às Threads, suas instâncias só são úteis enquanto estiverem em execução na JVM, e podem ser paradas/destruídas e iniciadas com instruções de sua classe. Não existe algo no estado destas instâncias que seja útil para se persistir, e por isso não há sentido em persistir tais instâncias para restaurar posteriormente.

Mas e se existir alguma classe em que seja necessário persisti-la, e esta possui uma instância da classe “Thread”? Neste caso, utilizando um recurso pode-se persistir a classe inteira, menos a instância thread. Este recurso, refere-se em indicar que tal instância thread é um objeto “Transient”. Na Listagem 6 encontra-se um exemplo da implementação com uso de “Transient”.

Listagem 6 - Uso do Transient

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | import java.io.Serializable;    public class ImplementacaoTransient implements Serializable, Runnable{        transient private Thread thread;      //marcado com "transient" para que sua instância não seja serializada        //Demais valores de estado serão serializados      private int valorParaPersistir;        public ImplementacaoTransient(int valorParaPersistir){          this.valorParaPersistir = valorParaPersistir;          thread = new Thread(this);          thread.start();      }        public void run()      {          while(true){              //código a ser executado pela Thread          }      }  } |

Vale ressaltar que, o que é serializado é apenas o estado do objeto, e não os métodos e a classe. No exemplo acima, apenas a instância do objeto “thread” e do objeto “valorParaPersistir” compunham o estado do objeto. Assim, evita-se que o sistema de serialização tente serializar um objeto não-serializável e gere erros. Após a deserialização de um objeto da classe “ImplementacaoTransient” este estará totalmente funcional.

Um outro problema que pode vir a ocorrer, se refere ao versionamento das classes. Toda classe possui um atributo oculto chamado “serialVersionUID” com um valor gerado automaticamente. Objetos criados levam consigo o “serialVersionUID” de sua classe, que indica sua versão. Toda vez que uma classe sofre alterações em seu código, um novo “serialVersionUID” é gerado. O problema com a serialização surge quando serializa-se o estado de um objeto e posteriormente tenta-se restaurar o estado para a aplicação, mas houveram mudanças no código da classe. Neste caso, uma exceção “java.io.InvalidClassException” é disparada, pois de fato, o estado do objeto corresponde a uma versão de classe do qual foi criado que já não existe mais. Portanto, isso só ocorre para evitar erros de programação, em que tenta-se restaurar o estado de um objeto para uma classe que possui versão diferente da qual o objeto foi serializado inicialmente. E isso pode significar diferentes versões de uma mesma classe “Fruta” por exemplo, ou diferentes versões por classes serem totalmente diferentes como: o estado de um objeto da classe “Fruta” ser restaurado para classe “Carro”.

Mas quando é necessário, conscientemente, recuperar o estado de um objeto, mesmo que houveram alterações na classe do qual foi gerado, é possível controlar manualmente o “serialVersionUID”. Para isto, basta declarar explicitamente na forma de atributo estático e final, um valor Long qualquer para corresponder à versão da classe. E assim, mesmo que houverem modificações na classe, a versão permenecerá a mesma, evitando as exceções da incompatiblidade de classes. No entanto, vale ressaltar que o versionamento não é um impecílio para o programador, mas sim muito importante em casos que deseja-se manter um controle sobre a compatibilidade de código na aplicação, definindo versões para classes No código da Listagem 7 encontra-se o exemplo do controle manual do “serialVersionUID”.

Listagem 7 - Uso manual do serialVersionUID

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | import java.io.Serializable;    public class Fruta implements Serializable{      private String nome;      private String cor;      static final long serialVersionUID = 123L; //indica a versao da classe        public Fruta(String nome, String cor) {          this.nome = nome;          this.cor = cor;      }      public String getCor() { return cor; }      public String getNome() { return nome; }      public void setCor(String cor) { this.cor = cor; }      public void setNome(String nome) { this.nome = nome; }  } |

O JDK fornece uma ferramenta que permite visualizar a versão de qualquer classe Java. Basta no terminal do sistema operacional utilizar o comando “serialver” seguido da classe qualificada, conforme exemplo da Listagem 8.

Listagem 8 - Verificação da versão de classe

|  |  |
| --- | --- |
| 1 | serialver pojo.Fruta |

A serialização acaba inflingindo alguns modelos de segurança em seu procedimento padrão, e criando então um novo problema. Quando um estado de objeto é persistido no sistema de arquivos ou enviado por rede, o fluxo binário pode ser detectado, analisado e obtidas informações sobre a classe e o estado do objeto. Por exemplo, se utilizado um editor de texto qualquer para abrir o arquivo persistido do estado do objeto da classe “Fruta” dos exemplos anteriores, é comum os atributos e outras informações aparecerem como um simples texto, total ou parcilamente legíveis, assim como demonstrado na Listagem 9.

Listagem 9 - Exemplo de leitura de arquivo binário de um estado de objeto persistido.

|  |  |
| --- | --- |
| 1  2 | ��##sr#  pojo.Fruta#######{###L##cort##Ljava/lang/String;L##nomeq#~##xpt##Vermelhat##Maça |

Com uma ferramenta menos simples seria possível aumentar a quantidade de informações legíveis, coletando uma maior quantia de dados.

No entando, o processo de serialização permite que haja uma certa personalização da serialização e deserialização de um objeto. Isso deve-se a possibilidade da sobrescrição dos métodos “writeObject” e “readObject” dentro da classe a ser serializada, como a classe exemplar “pojo.Fruta”. Esta sobrescrita permite um controle total sobre a serialização, pois as classes “java.io.ObjectOutputStream” e “java.io.ObjectInputStream”, ao serializar o estado de um objeto, primeiramente verificam se esse método foi sobrescrito na classe alvo e assim o utilizam, senão, será utilizado o protocolo padrão de serialização.

Realizando a sobrescrição dos métodos é possível criptografar os dados na serialização e descriptografar os dados na deserialização, por meio de algum algoritmo que o programador venha implementar. Assim os dados podem ser persistidos ou transmitidos por rede, sem que seja possível ler as partes mais importantes do estado de um objeto. Na listagem 10, a classe “pojo.Fruta” foi modificada dando origem a classe “pojo.FrutaSobrescrita” que possui adicionamente a sobrescrição dos métodos “writeObject” e “readObject”. Como este é apenas um exemplo, não foi escrito um verdadeiro algoritmo de criptografia, mas sim apenas uma troca no valor da String do campo “cor”, identificando que houveram alterações tanto no processo de serialização quanto no de deserialização. Na listagem 11 encontra-se a execução da serialização.

Listagem 10 - Sobrescrita dos métodos “writeObject” e “readObject”.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36 | import java.io.Serializable;    public class FrutaSobrescrita implements Serializable{      private String nome;      private String cor;      static final long serialVersionUID = 123L; //indica a versao da classe        public FrutaSobrescrita(String nome, String cor) {          this.nome = nome;          this.cor = cor;      }      public String getCor() { return cor; }      public String getNome() { return nome; }      public void setCor(String cor) { this.cor = cor; }      public void setNome(String nome) { this.nome = nome; }        private void writeObject(java.io.ObjectOutputStream stream) throws java.io.IOException      {          // criptografar os dados antes de liberar o stream          // EXEMPLO: setCor(criptografar(getCor()));          setCor(getCor()+ " > CRIPTOGRAFADO"); // apenas concatenação de string para indicar alteração            // liberar o stream          stream.defaultWriteObject();      }        private void readObject(java.io.ObjectInputStream stream) throws java.io.IOException, ClassNotFoundException      {          // liberar stream          stream.defaultReadObject();            // descriptografar os dados          // EXEMPLO: setCor(descriptografar(getCor()));          setCor(getCor()+ " > DESCRIPTOGRAFADO"); // apenas contatenação de string para indicar alteração      }  } |

Listagem 11 - Execução da serialização da classe “pojo.FrutaSobrescrita”.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | //serializa        Serializador s = new Serializador();        FrutaSobrescrita frutaSobrescrita = new FrutaSobrescrita("Maça", "Vermelha");         try {             s.serializar("/home/gustavo/frutaSobrescrita", frutaSobrescrita);         } catch (Exception ex) {             System.err.println("Falha ao serializar! - " + ex.toString());         }           //deserializa        Deserializador d = new Deserializador();         frutaSobrescrita = null;         try {             frutaSobrescrita = (FrutaSobrescrita) d.deserializar("/home/gustavo/frutaSobrescrita");         } catch (Exception ex) {             System.err.println("Falha ao deserializar! - " + ex.toString());         }         System.out.println("Sobrescrita: " + frutaSobrescrita.getNome() + " - " + frutaSobrescrita.getCor()); |

**5. CONCLUSÃO**

A serialização é um procedimento de simples execução por parte do programador. Com pouco tempo de codificação é possível implementar a persistencia de um estado de objeto para o sistema de arquivos ou de transmissão por rede. É possível reproduzir os exemplos criados neste artigo, em situações até mesmo mais elaboradas. A apresentação de alguns problemas e soluções trazidas por particularidades, abre o entendimento de que a serialização pode deixar de ser simples conforme for aplicada em diferentes contextos, e que ao mesmo tempo é muito flexível ao permitir ao programador implementar diferentes soluções.

[Assista ao vídeo Serialização/Deserialização de objetos com YAML](http://www.devmedia.com.br/serializacao-deserializacao-de-objetos-com-yaml-parte-1/8399)

**6. BIBLIOGRAFIA**

ORACLE - **java.io Interface Serializable** - Disponível em: <http://docs.oracle.com/javase/1.4.2/docs/api/java/io/Serializable.html>Acesso em: 11/03/2012

ORACLE - **Discover de Secrets of Java Serialization API** - Disponível em: <http://java.sun.com/developer/technicalArticles/Programming/serialization/> Acesso em: 11/03/2012

NEWARD, Ted - **5 coisas que você não sabia sobre... Serialização de Objetos Java** - Disponível em: <http://www.ibm.com/developerworks/br/library/j-5things1/index.html> Acesso em: 11/03/2012

DOWNS, Andrew - **Java Serialization** - Disponível em: <http://www.mactech.com/articles/mactech/Vol.14/14.04/JavaSerialization/index.html> Acesso em: 11/03/2012

PREISSLER, Luciano E.; GEYER, Cláudio F. R. - **Serialização em Java** - Disponível em: <http://www.inf.ufrgs.br/gppd/disc/cmp167/trabalhos/sem99-1/T1/luciano/final.htm> Acesso em: 11/03/2012

[[](http://www.linhadecodigo.com.br/colaborador.aspx?id=354)](http://www.linhadecodigo.com.br/colaborador.aspx?id=354)

[Everton Coimbra de Araújo - Desde 1987 atua na área de treinamento e desenvolvimento. Como Mestre em Ciência da Computação, é professor da UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ, Campus Medianeira, onde leciona disciplinas relacionadas ao desenvolvimento de aplicações web, com Java e .NET.   
  
É autor dos livros Desenvolvimento para WEB com Java, Orientação a Objetos com Java - Simples, Fácil e Eficiente, Algoritmo - Fundamento e Prática em sua terceira edição e dos livros Delphi - Implementação e Técnicas para Ambientes Virtuais e C++ Builder - Implementação e Técnicas para Ambientes Virtuais. Todos pela VisualBooks. Pode ser contactado através do e-mail everton@utfpr.edu.br ou evertoncoimbra@gmail.com.](http://www.linhadecodigo.com.br/colaborador.aspx?id=354)

 85 3 0 4

**Leia também**

* [Conhecendo a interface Map do Java](http://www.linhadecodigo.com.br/artigo/3670/conhecendo-a-interface-map-do-java.aspx)*Java*
* [Trabalhando com a Interface Set no Java](http://www.linhadecodigo.com.br/artigo/3669/trabalhando-com-a-interface-set-no-java.aspx)*Java*
* [Classes Wrappers em Java](http://www.linhadecodigo.com.br/artigo/3667/classes-wrappers-em-java.aspx)*Java*
* [ComboBoxModel em Java](http://www.linhadecodigo.com.br/artigo/3664/comboboxmodel-em-java.aspx)*Java*
* [Reduzindo a Mutabilidade em Java](http://www.linhadecodigo.com.br/artigo/3659/reduzindo-a-mutabilidade-em-java.aspx)*Java*

Publicidade

[](http://qconsp.com/?utm_source=devmedia&utm_medium=banner&utm_campaign=devmedia)

Linha de Código[Anuncie](http://www.web-03.net/linhadecodigo.asp)|[Fale Conosco](mailto:colabore@linhadecodigo.com.br)|[Publique](http://www.linhadecodigo.com.br/publique.aspx)

|  |  |
| --- | --- |
| http://www.linhadecodigo.com.br/artigos/img_artigos/EvertonCoimbraAraujo/Serializazao_Objetos/imagem001.jpg | Neste artigo apresento o Gustavo Rafael Valiati, que pode ser contatado pelo e-mail gustavovaliati@gmail.com. O Gustavo foi meu aluno em algumas disciplinas de sua gradução e está concluíndo o último período do curso de Tecnologia em Análise e Desenvolvimento de Sistemas da UTFPR campus Medianeira, neste primeiro semestre de 2012. |

**1. INTRODUÇÃO**

Nem sempre se deseja ter objetos ou estrutura de dados de um aplicativo, com um ciclo de vida que dura no máximo o tempo em que for armazenado em memória RAM. Ou seja, algumas vezes deseja-se guardá-los por um tempo indeterminado, sendo possível recuperá-los posteriormente. E neste caso, o termo “persistência” tem seu conceito definido.

Persistir dados, requer que estes possam ser serializáveis. A serialização possibilitará capturar o estado do objeto ou a estrutura de dados, e transformar em uma cadeia de bytes quando for necessário. Torna-se possível também, recuperar os bytes persistidos para fazer o processo inverso, e ter os dados de volta para a aplicação em execução.

Desta maneira, qualquer forma de dados que esteja sendo trabalhada na aplicação, e esta puder ser serializada, poderá ser passada para o meio físico, e ter o processo revertido.

A serialização não se limita apenas em possibilitar a gravação do dado em disco, mas permite que o mesmo seja transmitido por rede. Isto porquê a serialização cria um stream de bytes, que é requisito para transmissão de dados por rede.

Diversas linguagens possuem suporte à serialização, como: Java, C, C++, Python, PHP, .NET, entre outras. Neste artigo, será tratado apenas da serialização com Java, em sua implemetação básica, alguns problemas e soluções, com códigos para exemplificação.

**2. SERIALIZAÇÃO COM JAVA**

Serializar um objeto, em Java, só é possível caso sua classe esteja marcada como serializável. Essa marcação é feita com a simples implementação da interface “java.io.Serializable”. Essa interface não possui métodos para se implementar e nem atributos. Ainda não existiam as anotações no Java 1.1, versão que trouxe pela primeira vez a serialização ao Java, e desta maneira a implementação da interface era a melhor maneira de marcar uma classe.

A herança entre classes é naturalmente afetada pela serialização. Já que quando uma classe implementa a interface “Serializable”, toda classe que dela estender também estará implicitamente marcada como serializável

.

Para exemplo, será criada uma classe “pojo.Fruta” que contém dois atributos: nome e cor. No código exemplo (Listagem 1), pode-se notar a classe “Fruta” marcada para a serialização:

Listagem 1 - Marcando uma classe como serializável.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | import java.io.Serializable;    public class Fruta implements Serializable{      private String nome;      private String cor;        public Fruta(String nome, String cor) {          this.nome = nome;          this.cor = cor;      }        public String getCor() { return cor; }      public String getNome() { return nome; }      public void setCor(String cor) { this.cor = cor; }      public void setNome(String nome) { this.nome = nome; }  } |

Pode-se dizer que a implementação do processo de serialização e deserialização simples de um objeto, não possui um código difícil de se escrever. Isto porquê duas classes fazem a maior parte do trabalho, de maneira que com poucas linhas de código pode-se transformar um objeto em bytes e vice-versa. Essas duas classes são “java.io.ObjectInputStream” que recebe bytes para criar um objeto e “java.io.ObjectOutputStream” que cria uma cadeia de bytes a partir de um objeto.

**3. SERIALIZANDO E DESERIALIZANDO O ESTADO DE UM OBJETO**

Como exemplo, será persistido um objeto em disco. Para isto, além das classes “java.io.ObjectInputStream” e “java.io.ObjectOutputStream”, serão utilizadas duas outras que trabalham com arquivos, que são “java.io.FileInputStream” e “java.io.FileOutputStream”, que seguem a mesma lógica de InputStream e OutputStream, só que agora relacionado ao sistema de arquivos e não aos objetos.

Para serialização, deve-se criar um novo “java.io.FileOutputStream”, com o caminho (“path”) do arquivo onde se deseja armazenar o estado do objeto. Também deve ser criado um novo “java.io.ObjectOutputStream” que está ligado ao arquivo. E por último deve-se executar a escrita do objeto alvo, para dentro do objeto de stream que por sua vez está ligado ao arquivo destino. Após executados esses passos, é possível encontrar no sistema de arquivos um novo arquivo que contém os bytes correspondentes à descrição do estado do objeto. Na Listagem 2, encontra-se o código exemplo.

Listagem 2 - Serialização de um objeto.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | import java.io.FileOutputStream;  import java.io.ObjectOutputStream;    public class Serializador {        public Serializador() {    }        public void serializar(String path, Object obj) throws Exception {              FileOutputStream outFile = new FileOutputStream(path);      ObjectOutputStream s = new ObjectOutputStream(outFile);      s.writeObject(obj);      s.close();      }  } |

E para a deserialização, o processo é extremamente parecido com a serialização. Cria-se um novo “java.io.FileInputStream” relacionado ao arquivo onde o estado do objeto encontra-se persistido. Deve-se criar também um novo “java.io.ObjectInputStream” que estará vinculado ao arquivo de origem. Por último, executa-se a leitura do arquivo para a restauração do estado. Na Listagem 3 encontra-se o exemplo do código.

Listagem 3 - Deserializaçãode um objeto.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | import java.io.FileInputStream;  import java.io.ObjectInputStream;    public class Deserializador {        public Deserializador() {    }        public Object deserializar(String path) throws Exception {      FileInputStream inFile = new FileInputStream(path);      ObjectInputStream d = new ObjectInputStream(inFile);      Object o = d.readObject();      d.close();      return o;      }  } |

A Listagem 4, contém o exemplo da execução da serialização e em seguida da deserialização.

Listagem 4 - Serializando e Deserializando

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | import operacoes.Deserializador;  import operacoes.Serializador;  import pojo.Fruta;    public class Principal {      public static void main(String args[]){         //serializa         Serializador s = new Serializador();         Fruta fruta = new Fruta("Maça", "Vermelha");          try {              s.serializar("/home/gustavo/fruta", fruta);          } catch (Exception ex) {              System.err.println("Falha ao serializar! - " + ex.toString());          }            //deserializa          Deserializador d = new Deserializador();          fruta = null;          try {              fruta = (Fruta) d.deserializar("/home/gustavo/fruta");          } catch (Exception ex) {              System.err.println("Falha ao deserializar! - " + ex.toString());          }          System.out.println(fruta.getNome() + " - " + fruta.getCor());      }  } |

Na Listagem 5, encontra-se um exemplo de serialização e deserialização de um ArrayList da classe Fruta. É apenas mais um exemplo que demonstra a capacidade de se persistir qualquer objeto que seja serializável. Essa serialização só foi possível porque a classe “ArrayList” implementa a interface “Serializable” por padrão.

Listagem 5 - Serialização de ArrayList.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | //serialização e deserialização com ArrayList          ArrayList<Fruta> frutas = new ArrayList<Fruta>();          Fruta f1 = new Fruta("Laranja", "Amarela");          Fruta f2 = new Fruta("Abacate", "Verde");          Fruta f3 = new Fruta("Morango", "Vermelho");          frutas.add(f1);          frutas.add(f2);          frutas.add(f3);            try {              s.serializar("/home/gustavo/frutas", frutas);              frutas = null;              frutas = (ArrayList<Fruta>) d.deserializar("/home/gustavo/frutas");              for(Fruta f : frutas){                  System.out.println("ArrayList: " + f.getNome() + " - " + f.getCor());              }          } catch (Exception ex) {              System.err.println("Falha ao serializar ou deserializar! - " +                                 ex.toString());          } |

**4. PARTICULARIDADES DA SERIALIZAÇÃO**

Como dito, a serialização de um objeto em Java pode ser algo fácil de se fazer. Mas existem certas informações que são importantes e sabê-las pode evitar vários problemas.

Um dos problemas é que nem tudo pode ser serializado, como objetos das classes “Thread” e “Socket” e atributos estáticos. Os atributos estáticos estão vinculados à classe e não ao objeto. Assim quando se trata do estado do objeto, os campos estáticos não são considerados. Em relação às Threads, suas instâncias só são úteis enquanto estiverem em execução na JVM, e podem ser paradas/destruídas e iniciadas com instruções de sua classe. Não existe algo no estado destas instâncias que seja útil para se persistir, e por isso não há sentido em persistir tais instâncias para restaurar posteriormente.

Mas e se existir alguma classe em que seja necessário persisti-la, e esta possui uma instância da classe “Thread”? Neste caso, utilizando um recurso pode-se persistir a classe inteira, menos a instância thread. Este recurso, refere-se em indicar que tal instância thread é um objeto “Transient”. Na Listagem 6 encontra-se um exemplo da implementação com uso de “Transient”.

Listagem 6 - Uso do Transient

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | import java.io.Serializable;    public class ImplementacaoTransient implements Serializable, Runnable{        transient private Thread thread;      //marcado com "transient" para que sua instância não seja serializada        //Demais valores de estado serão serializados      private int valorParaPersistir;        public ImplementacaoTransient(int valorParaPersistir){          this.valorParaPersistir = valorParaPersistir;          thread = new Thread(this);          thread.start();      }        public void run()      {          while(true){              //código a ser executado pela Thread          }      }  } |

Vale ressaltar que, o que é serializado é apenas o estado do objeto, e não os métodos e a classe. No exemplo acima, apenas a instância do objeto “thread” e do objeto “valorParaPersistir” compunham o estado do objeto. Assim, evita-se que o sistema de serialização tente serializar um objeto não-serializável e gere erros. Após a deserialização de um objeto da classe “ImplementacaoTransient” este estará totalmente funcional.

Um outro problema que pode vir a ocorrer, se refere ao versionamento das classes. Toda classe possui um atributo oculto chamado “serialVersionUID” com um valor gerado automaticamente. Objetos criados levam consigo o “serialVersionUID” de sua classe, que indica sua versão. Toda vez que uma classe sofre alterações em seu código, um novo “serialVersionUID” é gerado. O problema com a serialização surge quando serializa-se o estado de um objeto e posteriormente tenta-se restaurar o estado para a aplicação, mas houveram mudanças no código da classe. Neste caso, uma exceção “java.io.InvalidClassException” é disparada, pois de fato, o estado do objeto corresponde a uma versão de classe do qual foi criado que já não existe mais. Portanto, isso só ocorre para evitar erros de programação, em que tenta-se restaurar o estado de um objeto para uma classe que possui versão diferente da qual o objeto foi serializado inicialmente. E isso pode significar diferentes versões de uma mesma classe “Fruta” por exemplo, ou diferentes versões por classes serem totalmente diferentes como: o estado de um objeto da classe “Fruta” ser restaurado para classe “Carro”.

Mas quando é necessário, conscientemente, recuperar o estado de um objeto, mesmo que houveram alterações na classe do qual foi gerado, é possível controlar manualmente o “serialVersionUID”. Para isto, basta declarar explicitamente na forma de atributo estático e final, um valor Long qualquer para corresponder à versão da classe. E assim, mesmo que houverem modificações na classe, a versão permenecerá a mesma, evitando as exceções da incompatiblidade de classes. No entanto, vale ressaltar que o versionamento não é um impecílio para o programador, mas sim muito importante em casos que deseja-se manter um controle sobre a compatibilidade de código na aplicação, definindo versões para classes No código da Listagem 7 encontra-se o exemplo do controle manual do “serialVersionUID”.

Listagem 7 - Uso manual do serialVersionUID

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | import java.io.Serializable;    public class Fruta implements Serializable{      private String nome;      private String cor;      static final long serialVersionUID = 123L; //indica a versao da classe        public Fruta(String nome, String cor) {          this.nome = nome;          this.cor = cor;      }      public String getCor() { return cor; }      public String getNome() { return nome; }      public void setCor(String cor) { this.cor = cor; }      public void setNome(String nome) { this.nome = nome; }  } |

O JDK fornece uma ferramenta que permite visualizar a versão de qualquer classe Java. Basta no terminal do sistema operacional utilizar o comando “serialver” seguido da classe qualificada, conforme exemplo da Listagem 8.

Listagem 8 - Verificação da versão de classe

|  |  |
| --- | --- |
| 1 | serialver pojo.Fruta |

A serialização acaba inflingindo alguns modelos de segurança em seu procedimento padrão, e criando então um novo problema. Quando um estado de objeto é persistido no sistema de arquivos ou enviado por rede, o fluxo binário pode ser detectado, analisado e obtidas informações sobre a classe e o estado do objeto. Por exemplo, se utilizado um editor de texto qualquer para abrir o arquivo persistido do estado do objeto da classe “Fruta” dos exemplos anteriores, é comum os atributos e outras informações aparecerem como um simples texto, total ou parcilamente legíveis, assim como demonstrado na Listagem 9.

Listagem 9 - Exemplo de leitura de arquivo binário de um estado de objeto persistido.

|  |  |
| --- | --- |
| 1  2 | ��##sr#  pojo.Fruta#######{###L##cort##Ljava/lang/String;L##nomeq#~##xpt##Vermelhat##Maça |

Com uma ferramenta menos simples seria possível aumentar a quantidade de informações legíveis, coletando uma maior quantia de dados.

No entando, o processo de serialização permite que haja uma certa personalização da serialização e deserialização de um objeto. Isso deve-se a possibilidade da sobrescrição dos métodos “writeObject” e “readObject” dentro da classe a ser serializada, como a classe exemplar “pojo.Fruta”. Esta sobrescrita permite um controle total sobre a serialização, pois as classes “java.io.ObjectOutputStream” e “java.io.ObjectInputStream”, ao serializar o estado de um objeto, primeiramente verificam se esse método foi sobrescrito na classe alvo e assim o utilizam, senão, será utilizado o protocolo padrão de serialização.

Realizando a sobrescrição dos métodos é possível criptografar os dados na serialização e descriptografar os dados na deserialização, por meio de algum algoritmo que o programador venha implementar. Assim os dados podem ser persistidos ou transmitidos por rede, sem que seja possível ler as partes mais importantes do estado de um objeto. Na listagem 10, a classe “pojo.Fruta” foi modificada dando origem a classe “pojo.FrutaSobrescrita” que possui adicionamente a sobrescrição dos métodos “writeObject” e “readObject”. Como este é apenas um exemplo, não foi escrito um verdadeiro algoritmo de criptografia, mas sim apenas uma troca no valor da String do campo “cor”, identificando que houveram alterações tanto no processo de serialização quanto no de deserialização. Na listagem 11 encontra-se a execução da serialização.

Listagem 10 - Sobrescrita dos métodos “writeObject” e “readObject”.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36 | import java.io.Serializable;    public class FrutaSobrescrita implements Serializable{      private String nome;      private String cor;      static final long serialVersionUID = 123L; //indica a versao da classe        public FrutaSobrescrita(String nome, String cor) {          this.nome = nome;          this.cor = cor;      }      public String getCor() { return cor; }      public String getNome() { return nome; }      public void setCor(String cor) { this.cor = cor; }      public void setNome(String nome) { this.nome = nome; }        private void writeObject(java.io.ObjectOutputStream stream) throws java.io.IOException      {          // criptografar os dados antes de liberar o stream          // EXEMPLO: setCor(criptografar(getCor()));          setCor(getCor()+ " > CRIPTOGRAFADO"); // apenas concatenação de string para indicar alteração            // liberar o stream          stream.defaultWriteObject();      }        private void readObject(java.io.ObjectInputStream stream) throws java.io.IOException, ClassNotFoundException      {          // liberar stream          stream.defaultReadObject();            // descriptografar os dados          // EXEMPLO: setCor(descriptografar(getCor()));          setCor(getCor()+ " > DESCRIPTOGRAFADO"); // apenas contatenação de string para indicar alteração      }  } |

Listagem 11 - Execução da serialização da classe “pojo.FrutaSobrescrita”.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | //serializa        Serializador s = new Serializador();        FrutaSobrescrita frutaSobrescrita = new FrutaSobrescrita("Maça", "Vermelha");         try {             s.serializar("/home/gustavo/frutaSobrescrita", frutaSobrescrita);         } catch (Exception ex) {             System.err.println("Falha ao serializar! - " + ex.toString());         }           //deserializa        Deserializador d = new Deserializador();         frutaSobrescrita = null;         try {             frutaSobrescrita = (FrutaSobrescrita) d.deserializar("/home/gustavo/frutaSobrescrita");         } catch (Exception ex) {             System.err.println("Falha ao deserializar! - " + ex.toString());         }         System.out.println("Sobrescrita: " + frutaSobrescrita.getNome() + " - " + frutaSobrescrita.getCor()); |

**5. CONCLUSÃO**

A serialização é um procedimento de simples execução por parte do programador. Com pouco tempo de codificação é possível implementar a persistencia de um estado de objeto para o sistema de arquivos ou de transmissão por rede. É possível reproduzir os exemplos criados neste artigo, em situações até mesmo mais elaboradas. A apresentação de alguns problemas e soluções trazidas por particularidades, abre o entendimento de que a serialização pode deixar de ser simples conforme for aplicada em diferentes contextos, e que ao mesmo tempo é muito flexível ao permitir ao programador implementar diferentes soluções.

[Assista ao vídeo Serialização/Deserialização de objetos com YAML](http://www.devmedia.com.br/serializacao-deserializacao-de-objetos-com-yaml-parte-1/8399)

**6. BIBLIOGRAFIA**

ORACLE - **java.io Interface Serializable** - Disponível em: <http://docs.oracle.com/javase/1.4.2/docs/api/java/io/Serializable.html>Acesso em: 11/03/2012

ORACLE - **Discover de Secrets of Java Serialization API** - Disponível em: <http://java.sun.com/developer/technicalArticles/Programming/serialization/> Acesso em: 11/03/2012

NEWARD, Ted - **5 coisas que você não sabia sobre... Serialização de Objetos Java** - Disponível em: <http://www.ibm.com/developerworks/br/library/j-5things1/index.html> Acesso em: 11/03/2012

DOWNS, Andrew - **Java Serialization** - Disponível em: <http://www.mactech.com/articles/mactech/Vol.14/14.04/JavaSerialization/index.html> Acesso em: 11/03/2012

PREISSLER, Luciano E.; GEYER, Cláudio F. R. - **Serialização em Java** - Disponível em: <http://www.inf.ufrgs.br/gppd/disc/cmp167/trabalhos/sem99-1/T1/luciano/final.htm> Acesso em: 11/03/2012

[[](http://www.linhadecodigo.com.br/colaborador.aspx?id=354)](http://www.linhadecodigo.com.br/colaborador.aspx?id=354)

[Everton Coimbra de Araújo - Desde 1987 atua na área de treinamento e desenvolvimento. Como Mestre em Ciência da Computação, é professor da UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ, Campus Medianeira, onde leciona disciplinas relacionadas ao desenvolvimento de aplicações web, com Java e .NET.   
  
É autor dos livros Desenvolvimento para WEB com Java, Orientação a Objetos com Java - Simples, Fácil e Eficiente, Algoritmo - Fundamento e Prática em sua terceira edição e dos livros Delphi - Implementação e Técnicas para Ambientes Virtuais e C++ Builder - Implementação e Técnicas para Ambientes Virtuais. Todos pela VisualBooks. Pode ser contactado através do e-mail everton@utfpr.edu.br ou evertoncoimbra@gmail.com.](http://www.linhadecodigo.com.br/colaborador.aspx?id=354)

 85 3 0 4

**Leia também**

* [Conhecendo a interface Map do Java](http://www.linhadecodigo.com.br/artigo/3670/conhecendo-a-interface-map-do-java.aspx)*Java*
* [Trabalhando com a Interface Set no Java](http://www.linhadecodigo.com.br/artigo/3669/trabalhando-com-a-interface-set-no-java.aspx)*Java*
* [Classes Wrappers em Java](http://www.linhadecodigo.com.br/artigo/3667/classes-wrappers-em-java.aspx)*Java*
* [ComboBoxModel em Java](http://www.linhadecodigo.com.br/artigo/3664/comboboxmodel-em-java.aspx)*Java*
* [Reduzindo a Mutabilidade em Java](http://www.linhadecodigo.com.br/artigo/3659/reduzindo-a-mutabilidade-em-java.aspx)*Java*

Publicidade

[](http://qconsp.com/?utm_source=devmedia&utm_medium=banner&utm_campaign=devmedia)

Linha de Código[Anuncie](http://www.web-03.net/linhadecodigo.asp)|[Fale Conosco](mailto:colabore@linhadecodigo.com.br)|[Publique](http://www.linhadecodigo.com.br/publique.aspx)

|  |  |
| --- | --- |
| http://www.linhadecodigo.com.br/artigos/img_artigos/EvertonCoimbraAraujo/Serializazao_Objetos/imagem001.jpg | Neste artigo apresento o Gustavo Rafael Valiati, que pode ser contatado pelo e-mail gustavovaliati@gmail.com. O Gustavo foi meu aluno em algumas disciplinas de sua gradução e está concluíndo o último período do curso de Tecnologia em Análise e Desenvolvimento de Sistemas da UTFPR campus Medianeira, neste primeiro semestre de 2012. |

**1. INTRODUÇÃO**

Nem sempre se deseja ter objetos ou estrutura de dados de um aplicativo, com um ciclo de vida que dura no máximo o tempo em que for armazenado em memória RAM. Ou seja, algumas vezes deseja-se guardá-los por um tempo indeterminado, sendo possível recuperá-los posteriormente. E neste caso, o termo “persistência” tem seu conceito definido.

Persistir dados, requer que estes possam ser serializáveis. A serialização possibilitará capturar o estado do objeto ou a estrutura de dados, e transformar em uma cadeia de bytes quando for necessário. Torna-se possível também, recuperar os bytes persistidos para fazer o processo inverso, e ter os dados de volta para a aplicação em execução.

Desta maneira, qualquer forma de dados que esteja sendo trabalhada na aplicação, e esta puder ser serializada, poderá ser passada para o meio físico, e ter o processo revertido.

A serialização não se limita apenas em possibilitar a gravação do dado em disco, mas permite que o mesmo seja transmitido por rede. Isto porquê a serialização cria um stream de bytes, que é requisito para transmissão de dados por rede.

Diversas linguagens possuem suporte à serialização, como: Java, C, C++, Python, PHP, .NET, entre outras. Neste artigo, será tratado apenas da serialização com Java, em sua implemetação básica, alguns problemas e soluções, com códigos para exemplificação.

**2. SERIALIZAÇÃO COM JAVA**

Serializar um objeto, em Java, só é possível caso sua classe esteja marcada como serializável. Essa marcação é feita com a simples implementação da interface “java.io.Serializable”. Essa interface não possui métodos para se implementar e nem atributos. Ainda não existiam as anotações no Java 1.1, versão que trouxe pela primeira vez a serialização ao Java, e desta maneira a implementação da interface era a melhor maneira de marcar uma classe.

A herança entre classes é naturalmente afetada pela serialização. Já que quando uma classe implementa a interface “Serializable”, toda classe que dela estender também estará implicitamente marcada como serializável

.

Para exemplo, será criada uma classe “pojo.Fruta” que contém dois atributos: nome e cor. No código exemplo (Listagem 1), pode-se notar a classe “Fruta” marcada para a serialização:

Listagem 1 - Marcando uma classe como serializável.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | import java.io.Serializable;    public class Fruta implements Serializable{      private String nome;      private String cor;        public Fruta(String nome, String cor) {          this.nome = nome;          this.cor = cor;      }        public String getCor() { return cor; }      public String getNome() { return nome; }      public void setCor(String cor) { this.cor = cor; }      public void setNome(String nome) { this.nome = nome; }  } |

Pode-se dizer que a implementação do processo de serialização e deserialização simples de um objeto, não possui um código difícil de se escrever. Isto porquê duas classes fazem a maior parte do trabalho, de maneira que com poucas linhas de código pode-se transformar um objeto em bytes e vice-versa. Essas duas classes são “java.io.ObjectInputStream” que recebe bytes para criar um objeto e “java.io.ObjectOutputStream” que cria uma cadeia de bytes a partir de um objeto.

**3. SERIALIZANDO E DESERIALIZANDO O ESTADO DE UM OBJETO**

Como exemplo, será persistido um objeto em disco. Para isto, além das classes “java.io.ObjectInputStream” e “java.io.ObjectOutputStream”, serão utilizadas duas outras que trabalham com arquivos, que são “java.io.FileInputStream” e “java.io.FileOutputStream”, que seguem a mesma lógica de InputStream e OutputStream, só que agora relacionado ao sistema de arquivos e não aos objetos.

Para serialização, deve-se criar um novo “java.io.FileOutputStream”, com o caminho (“path”) do arquivo onde se deseja armazenar o estado do objeto. Também deve ser criado um novo “java.io.ObjectOutputStream” que está ligado ao arquivo. E por último deve-se executar a escrita do objeto alvo, para dentro do objeto de stream que por sua vez está ligado ao arquivo destino. Após executados esses passos, é possível encontrar no sistema de arquivos um novo arquivo que contém os bytes correspondentes à descrição do estado do objeto. Na Listagem 2, encontra-se o código exemplo.

Listagem 2 - Serialização de um objeto.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | import java.io.FileOutputStream;  import java.io.ObjectOutputStream;    public class Serializador {        public Serializador() {    }        public void serializar(String path, Object obj) throws Exception {              FileOutputStream outFile = new FileOutputStream(path);      ObjectOutputStream s = new ObjectOutputStream(outFile);      s.writeObject(obj);      s.close();      }  } |

E para a deserialização, o processo é extremamente parecido com a serialização. Cria-se um novo “java.io.FileInputStream” relacionado ao arquivo onde o estado do objeto encontra-se persistido. Deve-se criar também um novo “java.io.ObjectInputStream” que estará vinculado ao arquivo de origem. Por último, executa-se a leitura do arquivo para a restauração do estado. Na Listagem 3 encontra-se o exemplo do código.

Listagem 3 - Deserializaçãode um objeto.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | import java.io.FileInputStream;  import java.io.ObjectInputStream;    public class Deserializador {        public Deserializador() {    }        public Object deserializar(String path) throws Exception {      FileInputStream inFile = new FileInputStream(path);      ObjectInputStream d = new ObjectInputStream(inFile);      Object o = d.readObject();      d.close();      return o;      }  } |

A Listagem 4, contém o exemplo da execução da serialização e em seguida da deserialização.

Listagem 4 - Serializando e Deserializando

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | import operacoes.Deserializador;  import operacoes.Serializador;  import pojo.Fruta;    public class Principal {      public static void main(String args[]){         //serializa         Serializador s = new Serializador();         Fruta fruta = new Fruta("Maça", "Vermelha");          try {              s.serializar("/home/gustavo/fruta", fruta);          } catch (Exception ex) {              System.err.println("Falha ao serializar! - " + ex.toString());          }            //deserializa          Deserializador d = new Deserializador();          fruta = null;          try {              fruta = (Fruta) d.deserializar("/home/gustavo/fruta");          } catch (Exception ex) {              System.err.println("Falha ao deserializar! - " + ex.toString());          }          System.out.println(fruta.getNome() + " - " + fruta.getCor());      }  } |

Na Listagem 5, encontra-se um exemplo de serialização e deserialização de um ArrayList da classe Fruta. É apenas mais um exemplo que demonstra a capacidade de se persistir qualquer objeto que seja serializável. Essa serialização só foi possível porque a classe “ArrayList” implementa a interface “Serializable” por padrão.

Listagem 5 - Serialização de ArrayList.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | //serialização e deserialização com ArrayList          ArrayList<Fruta> frutas = new ArrayList<Fruta>();          Fruta f1 = new Fruta("Laranja", "Amarela");          Fruta f2 = new Fruta("Abacate", "Verde");          Fruta f3 = new Fruta("Morango", "Vermelho");          frutas.add(f1);          frutas.add(f2);          frutas.add(f3);            try {              s.serializar("/home/gustavo/frutas", frutas);              frutas = null;              frutas = (ArrayList<Fruta>) d.deserializar("/home/gustavo/frutas");              for(Fruta f : frutas){                  System.out.println("ArrayList: " + f.getNome() + " - " + f.getCor());              }          } catch (Exception ex) {              System.err.println("Falha ao serializar ou deserializar! - " +                                 ex.toString());          } |

**4. PARTICULARIDADES DA SERIALIZAÇÃO**

Como dito, a serialização de um objeto em Java pode ser algo fácil de se fazer. Mas existem certas informações que são importantes e sabê-las pode evitar vários problemas.

Um dos problemas é que nem tudo pode ser serializado, como objetos das classes “Thread” e “Socket” e atributos estáticos. Os atributos estáticos estão vinculados à classe e não ao objeto. Assim quando se trata do estado do objeto, os campos estáticos não são considerados. Em relação às Threads, suas instâncias só são úteis enquanto estiverem em execução na JVM, e podem ser paradas/destruídas e iniciadas com instruções de sua classe. Não existe algo no estado destas instâncias que seja útil para se persistir, e por isso não há sentido em persistir tais instâncias para restaurar posteriormente.

Mas e se existir alguma classe em que seja necessário persisti-la, e esta possui uma instância da classe “Thread”? Neste caso, utilizando um recurso pode-se persistir a classe inteira, menos a instância thread. Este recurso, refere-se em indicar que tal instância thread é um objeto “Transient”. Na Listagem 6 encontra-se um exemplo da implementação com uso de “Transient”.

Listagem 6 - Uso do Transient

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | import java.io.Serializable;    public class ImplementacaoTransient implements Serializable, Runnable{        transient private Thread thread;      //marcado com "transient" para que sua instância não seja serializada        //Demais valores de estado serão serializados      private int valorParaPersistir;        public ImplementacaoTransient(int valorParaPersistir){          this.valorParaPersistir = valorParaPersistir;          thread = new Thread(this);          thread.start();      }        public void run()      {          while(true){              //código a ser executado pela Thread          }      }  } |

Vale ressaltar que, o que é serializado é apenas o estado do objeto, e não os métodos e a classe. No exemplo acima, apenas a instância do objeto “thread” e do objeto “valorParaPersistir” compunham o estado do objeto. Assim, evita-se que o sistema de serialização tente serializar um objeto não-serializável e gere erros. Após a deserialização de um objeto da classe “ImplementacaoTransient” este estará totalmente funcional.

**2. SERIALIZAÇÃO COM JAVA**

Serializar um objeto, em Java, só é possível caso sua classe esteja marcada como serializável. Essa marcação é feita com a simples implementação da interface “java.io.Serializable”. Essa interface não possui métodos para se implementar e nem atributos. Ainda não existiam as anotações no Java 1.1, versão que trouxe pela primeira vez a serialização ao Java, e desta maneira a implementação da interface era a melhor maneira de marcar uma classe.

A herança entre classes é naturalmente afetada pela serialização. Já que quando uma classe implementa a interface “Serializable”, toda classe que dela estender também estará implicitamente marcada como serializável

.

Para exemplo, será criada uma classe “pojo.Fruta” que contém dois atributos: nome e cor. No código exemplo (Listagem 1), pode-se notar a classe “Fruta” marcada para a serialização:

Listagem 1 - Marcando uma classe como serializável.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | import java.io.Serializable;    public class Fruta implements Serializable{      private String nome;      private String cor;        public Fruta(String nome, String cor) {          this.nome = nome;          this.cor = cor;      }        public String getCor() { return cor; }      public String getNome() { return nome; }      public void setCor(String cor) { this.cor = cor; }      public void setNome(String nome) { this.nome = nome; }  } |

Pode-se dizer que a implementação do processo de serialização e deserialização simples de um objeto, não possui um código difícil de se escrever. Isto porquê duas classes fazem a maior parte do trabalho, de maneira que com poucas linhas de código pode-se transformar um objeto em bytes e vice-versa. Essas duas classes são “java.io.ObjectInputStream” que recebe bytes para criar um objeto e “java.io.ObjectOutputStream” que cria uma cadeia de bytes a partir de um objeto.

**3. SERIALIZANDO E DESERIALIZANDO O ESTADO DE UM OBJETO**

Como exemplo, será persistido um objeto em disco. Para isto, além das classes “java.io.ObjectInputStream” e “java.io.ObjectOutputStream”, serão utilizadas duas outras que trabalham com arquivos, que são “java.io.FileInputStream” e “java.io.FileOutputStream”, que seguem a mesma lógica de InputStream e OutputStream, só que agora relacionado ao sistema de arquivos e não aos objetos.

Para serialização, deve-se criar um novo “java.io.FileOutputStream”, com o caminho (“path”) do arquivo onde se deseja armazenar o estado do objeto. Também deve ser criado um novo “java.io.ObjectOutputStream” que está ligado ao arquivo. E por último deve-se executar a escrita do objeto alvo, para dentro do objeto de stream que por sua vez está ligado ao arquivo destino. Após executados esses passos, é possível encontrar no sistema de arquivos um novo arquivo que contém os bytes correspondentes à descrição do estado do objeto. Na Listagem 2, encontra-se o código exemplo.

Listagem 2 - Serialização de um objeto.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | import java.io.FileOutputStream;  import java.io.ObjectOutputStream;    public class Serializador {        public Serializador() {    }        public void serializar(String path, Object obj) throws Exception {              FileOutputStream outFile = new FileOutputStream(path);      ObjectOutputStream s = new ObjectOutputStream(outFile);      s.writeObject(obj);      s.close();      }  } |

E para a deserialização, o processo é extremamente parecido com a serialização. Cria-se um novo “java.io.FileInputStream” relacionado ao arquivo onde o estado do objeto encontra-se persistido. Deve-se criar também um novo “java.io.ObjectInputStream” que estará vinculado ao arquivo de origem. Por último, executa-se a leitura do arquivo para a restauração do estado. Na Listagem 3 encontra-se o exemplo do código.

Listagem 3 - Deserializaçãode um objeto.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | import java.io.FileInputStream;  import java.io.ObjectInputStream;    public class Deserializador {        public Deserializador() {    }        public Object deserializar(String path) throws Exception {      FileInputStream inFile = new FileInputStream(path);      ObjectInputStream d = new ObjectInputStream(inFile);      Object o = d.readObject();      d.close();      return o;      }  } |

A Listagem 4, contém o exemplo da execução da serialização e em seguida da deserialização.

Listagem 4 - Serializando e Deserializando

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | import operacoes.Deserializador;  import operacoes.Serializador;  import pojo.Fruta;    public class Principal {      public static void main(String args[]){         //serializa         Serializador s = new Serializador();         Fruta fruta = new Fruta("Maça", "Vermelha");          try {              s.serializar("/home/gustavo/fruta", fruta);          } catch (Exception ex) {              System.err.println("Falha ao serializar! - " + ex.toString());          }            //deserializa          Deserializador d = new Deserializador();          fruta = null;          try {              fruta = (Fruta) d.deserializar("/home/gustavo/fruta");          } catch (Exception ex) {              System.err.println("Falha ao deserializar! - " + ex.toString());          }          System.out.println(fruta.getNome() + " - " + fruta.getCor());      }  } |

Na Listagem 5, encontra-se um exemplo de serialização e deserialização de um ArrayList da classe Fruta. É apenas mais um exemplo que demonstra a capacidade de se persistir qualquer objeto que seja serializável. Essa serialização só foi possível porque a classe “ArrayList” implementa a interface “Serializable” por padrão.

Listagem 5 - Serialização de ArrayList.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | //serialização e deserialização com ArrayList          ArrayList<Fruta> frutas = new ArrayList<Fruta>();          Fruta f1 = new Fruta("Laranja", "Amarela");          Fruta f2 = new Fruta("Abacate", "Verde");          Fruta f3 = new Fruta("Morango", "Vermelho");          frutas.add(f1);          frutas.add(f2);          frutas.add(f3);            try {              s.serializar("/home/gustavo/frutas", frutas);              frutas = null;              frutas = (ArrayList<Fruta>) d.deserializar("/home/gustavo/frutas");              for(Fruta f : frutas){                  System.out.println("ArrayList: " + f.getNome() + " - " + f.getCor());              }          } catch (Exception ex) {              System.err.println("Falha ao serializar ou deserializar! - " +                                 ex.toString());          } |

**4. PARTICULARIDADES DA SERIALIZAÇÃO**

Como dito, a serialização de um objeto em Java pode ser algo fácil de se fazer. Mas existem certas informações que são importantes e sabê-las pode evitar vários problemas.

Um dos problemas é que nem tudo pode ser serializado, como objetos das classes “Thread” e “Socket” e atributos estáticos. Os atributos estáticos estão vinculados à classe e não ao objeto. Assim quando se trata do estado do objeto, os campos estáticos não são considerados. Em relação às Threads, suas instâncias só são úteis enquanto estiverem em execução na JVM, e podem ser paradas/destruídas e iniciadas com instruções de sua classe. Não existe algo no estado destas instâncias que seja útil para se persistir, e por isso não há sentido em persistir tais instâncias para restaurar posteriormente.

Mas e se existir alguma classe em que seja necessário persisti-la, e esta possui uma instância da classe “Thread”? Neste caso, utilizando um recurso pode-se persistir a classe inteira, menos a instância thread. Este recurso, refere-se em indicar que tal instância thread é um objeto “Transient”. Na Listagem 6 encontra-se um exemplo da implementação com uso de “Transient”.

Listagem 6 - Uso do Transient

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | import java.io.Serializable;    public class ImplementacaoTransient implements Serializable, Runnable{        transient private Thread thread;      //marcado com "transient" para que sua instância não seja serializada        //Demais valores de estado serão serializados      private int valorParaPersistir;        public ImplementacaoTransient(int valorParaPersistir){          this.valorParaPersistir = valorParaPersistir;          thread = new Thread(this);          thread.start();      }        public void run()      {          while(true){              //código a ser executado pela Thread          }      }  } |

Vale ressaltar que, o que é serializado é apenas o estado do objeto, e não os métodos e a classe. No exemplo acima, apenas a instância do objeto “thread” e do objeto “valorParaPersistir” compunham o estado do objeto. Assim, evita-se que o sistema de serialização tente serializar um objeto não-serializável e gere erros. Após a deserialização de um objeto da classe “ImplementacaoTransient” este estará totalmente funcional.

Um outro problema que pode vir a ocorrer, se refere ao versionamento das classes. Toda classe possui um atributo oculto chamado “serialVersionUID” com um valor gerado automaticamente. Objetos criados levam consigo o “serialVersionUID” de sua classe, que indica sua versão. Toda vez que uma classe sofre alterações em seu código, um novo “serialVersionUID” é gerado. O problema com a serialização surge quando serializa-se o estado de um objeto e posteriormente tenta-se restaurar o estado para a aplicação, mas houveram mudanças no código da classe. Neste caso, uma exceção “java.io.InvalidClassException” é disparada, pois de fato, o estado do objeto corresponde a uma versão de classe do qual foi criado que já não existe mais. Portanto, isso só ocorre para evitar erros de programação, em que tenta-se restaurar o estado de um objeto para uma classe que possui versão diferente da qual o objeto foi serializado inicialmente. E isso pode significar diferentes versões de uma mesma classe “Fruta” por exemplo, ou diferentes versões por classes serem totalmente diferentes como: o estado de um objeto da classe “Fruta” ser restaurado para classe “Carro”.

Mas quando é necessário, conscientemente, recuperar o estado de um objeto, mesmo que houveram alterações na classe do qual foi gerado, é possível controlar manualmente o “serialVersionUID”. Para isto, basta declarar explicitamente na forma de atributo estático e final, um valor Long qualquer para corresponder à versão da classe. E assim, mesmo que houverem modificações na classe, a versão permenecerá a mesma, evitando as exceções da incompatiblidade de classes. No entanto, vale ressaltar que o versionamento não é um impecílio para o programador, mas sim muito importante em casos que deseja-se manter um controle sobre a compatibilidade de código na aplicação, definindo versões para classes No código da Listagem 7 encontra-se o exemplo do controle manual do “serialVersionUID”.

Listagem 7 - Uso manual do serialVersionUID

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | import java.io.Serializable;    public class Fruta implements Serializable{      private String nome;      private String cor;      static final long serialVersionUID = 123L; //indica a versao da classe        public Fruta(String nome, String cor) {          this.nome = nome;          this.cor = cor;      }      public String getCor() { return cor; }      public String getNome() { return nome; }      public void setCor(String cor) { this.cor = cor; }      public void setNome(String nome) { this.nome = nome; }  } |

O JDK fornece uma ferramenta que permite visualizar a versão de qualquer classe Java. Basta no terminal do sistema operacional utilizar o comando “serialver” seguido da classe qualificada, conforme exemplo da Listagem 8.

Listagem 8 - Verificação da versão de classe

|  |  |
| --- | --- |
| http://www.linhadecodigo.com.br/artigos/img_artigos/EvertonCoimbraAraujo/Serializazao_Objetos/imagem001.jpg | Neste artigo apresento o Gustavo Rafael Valiati, que pode ser contatado pelo e-mail gustavovaliati@gmail.com. O Gustavo foi meu aluno em algumas disciplinas de sua gradução e está concluíndo o último período do curso de Tecnologia em Análise e Desenvolvimento de Sistemas da UTFPR campus Medianeira, neste primeiro semestre de 2012. |

**1. INTRODUÇÃO**

Nem sempre se deseja ter objetos ou estrutura de dados de um aplicativo, com um ciclo de vida que dura no máximo o tempo em que for armazenado em memória RAM. Ou seja, algumas vezes deseja-se guardá-los por um tempo indeterminado, sendo possível recuperá-los posteriormente. E neste caso, o termo “persistência” tem seu conceito definido.

Persistir dados, requer que estes possam ser serializáveis. A serialização possibilitará capturar o estado do objeto ou a estrutura de dados, e transformar em uma cadeia de bytes quando for necessário. Torna-se possível também, recuperar os bytes persistidos para fazer o processo inverso, e ter os dados de volta para a aplicação em execução.

Desta maneira, qualquer forma de dados que esteja sendo trabalhada na aplicação, e esta puder ser serializada, poderá ser passada para o meio físico, e ter o processo revertido.

A serialização não se limita apenas em possibilitar a gravação do dado em disco, mas permite que o mesmo seja transmitido por rede. Isto porquê a serialização cria um stream de bytes, que é requisito para transmissão de dados por rede.

Diversas linguagens possuem suporte à serialização, como: Java, C, C++, Python, PHP, .NET, entre outras. Neste artigo, será tratado apenas da serialização com Java, em sua implemetação básica, alguns problemas e soluções, com códigos para exemplificação.

**2. SERIALIZAÇÃO COM JAVA**

Serializar um objeto, em Java, só é possível caso sua classe esteja marcada como serializável. Essa marcação é feita com a simples implementação da interface “java.io.Serializable”. Essa interface não possui métodos para se implementar e nem atributos. Ainda não existiam as anotações no Java 1.1, versão que trouxe pela primeira vez a serialização ao Java, e desta maneira a implementação da interface era a melhor maneira de marcar uma classe.

A herança entre classes é naturalmente afetada pela serialização. Já que quando uma classe implementa a interface “Serializable”, toda classe que dela estender também estará implicitamente marcada como serializável

.

Para exemplo, será criada uma classe “pojo.Fruta” que contém dois atributos: nome e cor. No código exemplo (Listagem 1), pode-se notar a classe “Fruta” marcada para a serialização:

Listagem 1 - Marcando uma classe como serializável.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | import java.io.Serializable;    public class Fruta implements Serializable{      private String nome;      private String cor;        public Fruta(String nome, String cor) {          this.nome = nome;          this.cor = cor;      }        public String getCor() { return cor; }      public String getNome() { return nome; }      public void setCor(String cor) { this.cor = cor; }      public void setNome(String nome) { this.nome = nome; }  } |

Pode-se dizer que a implementação do processo de serialização e deserialização simples de um objeto, não possui um código difícil de se escrever. Isto porquê duas classes fazem a maior parte do trabalho, de maneira que com poucas linhas de código pode-se transformar um objeto em bytes e vice-versa. Essas duas classes são “java.io.ObjectInputStream” que recebe bytes para criar um objeto e “java.io.ObjectOutputStream” que cria uma cadeia de bytes a partir de um objeto.

**3. SERIALIZANDO E DESERIALIZANDO O ESTADO DE UM OBJETO**

Como exemplo, será persistido um objeto em disco. Para isto, além das classes “java.io.ObjectInputStream” e “java.io.ObjectOutputStream”, serão utilizadas duas outras que trabalham com arquivos, que são “java.io.FileInputStream” e “java.io.FileOutputStream”, que seguem a mesma lógica de InputStream e OutputStream, só que agora relacionado ao sistema de arquivos e não aos objetos.

Para serialização, deve-se criar um novo “java.io.FileOutputStream”, com o caminho (“path”) do arquivo onde se deseja armazenar o estado do objeto. Também deve ser criado um novo “java.io.ObjectOutputStream” que está ligado ao arquivo. E por último deve-se executar a escrita do objeto alvo, para dentro do objeto de stream que por sua vez está ligado ao arquivo destino. Após executados esses passos, é possível encontrar no sistema de arquivos um novo arquivo que contém os bytes correspondentes à descrição do estado do objeto. Na Listagem 2, encontra-se o código exemplo.

Listagem 2 - Serialização de um objeto.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | import java.io.FileOutputStream;  import java.io.ObjectOutputStream;    public class Serializador {        public Serializador() {    }        public void serializar(String path, Object obj) throws Exception {              FileOutputStream outFile = new FileOutputStream(path);      ObjectOutputStream s = new ObjectOutputStream(outFile);      s.writeObject(obj);      s.close();      }  } |

E para a deserialização, o processo é extremamente parecido com a serialização. Cria-se um novo “java.io.FileInputStream” relacionado ao arquivo onde o estado do objeto encontra-se persistido. Deve-se criar também um novo “java.io.ObjectInputStream” que estará vinculado ao arquivo de origem. Por último, executa-se a leitura do arquivo para a restauração do estado. Na Listagem 3 encontra-se o exemplo do código.

Listagem 3 - Deserializaçãode um objeto.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | import java.io.FileInputStream;  import java.io.ObjectInputStream;    public class Deserializador {        public Deserializador() {    }        public Object deserializar(String path) throws Exception {      FileInputStream inFile = new FileInputStream(path);      ObjectInputStream d = new ObjectInputStream(inFile);      Object o = d.readObject();      d.close();      return o;      }  } |

A Listagem 4, contém o exemplo da execução da serialização e em seguida da deserialização.

Listagem 4 - Serializando e Deserializando

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | import operacoes.Deserializador;  import operacoes.Serializador;  import pojo.Fruta;    public class Principal {      public static void main(String args[]){         //serializa         Serializador s = new Serializador();         Fruta fruta = new Fruta("Maça", "Vermelha");          try {              s.serializar("/home/gustavo/fruta", fruta);          } catch (Exception ex) {              System.err.println("Falha ao serializar! - " + ex.toString());          }            //deserializa          Deserializador d = new Deserializador();          fruta = null;          try {              fruta = (Fruta) d.deserializar("/home/gustavo/fruta");          } catch (Exception ex) {              System.err.println("Falha ao deserializar! - " + ex.toString());          }          System.out.println(fruta.getNome() + " - " + fruta.getCor());      }  } |

Na Listagem 5, encontra-se um exemplo de serialização e deserialização de um ArrayList da classe Fruta. É apenas mais um exemplo que demonstra a capacidade de se persistir qualquer objeto que seja serializável. Essa serialização só foi possível porque a classe “ArrayList” implementa a interface “Serializable” por padrão.

Listagem 5 - Serialização de ArrayList.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | //serialização e deserialização com ArrayList          ArrayList<Fruta> frutas = new ArrayList<Fruta>();          Fruta f1 = new Fruta("Laranja", "Amarela");          Fruta f2 = new Fruta("Abacate", "Verde");          Fruta f3 = new Fruta("Morango", "Vermelho");          frutas.add(f1);          frutas.add(f2);          frutas.add(f3);            try {              s.serializar("/home/gustavo/frutas", frutas);              frutas = null;              frutas = (ArrayList<Fruta>) d.deserializar("/home/gustavo/frutas");              for(Fruta f : frutas){                  System.out.println("ArrayList: " + f.getNome() + " - " + f.getCor());              }          } catch (Exception ex) {              System.err.println("Falha ao serializar ou deserializar! - " +                                 ex.toString());          } |

**4. PARTICULARIDADES DA SERIALIZAÇÃO**

Como dito, a serialização de um objeto em Java pode ser algo fácil de se fazer. Mas existem certas informações que são importantes e sabê-las pode evitar vários problemas.

Um dos problemas é que nem tudo pode ser serializado, como objetos das classes “Thread” e “Socket” e atributos estáticos. Os atributos estáticos estão vinculados à classe e não ao objeto. Assim quando se trata do estado do objeto, os campos estáticos não são considerados. Em relação às Threads, suas instâncias só são úteis enquanto estiverem em execução na JVM, e podem ser paradas/destruídas e iniciadas com instruções de sua classe. Não existe algo no estado destas instâncias que seja útil para se persistir, e por isso não há sentido em persistir tais instâncias para restaurar posteriormente.

Mas e se existir alguma classe em que seja necessário persisti-la, e esta possui uma instância da classe “Thread”? Neste caso, utilizando um recurso pode-se persistir a classe inteira, menos a instância thread. Este recurso, refere-se em indicar que tal instância thread é um objeto “Transient”. Na Listagem 6 encontra-se um exemplo da implementação com uso de “Transient”.

Listagem 6 - Uso do Transient

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | import java.io.Serializable;    public class ImplementacaoTransient implements Serializable, Runnable{        transient private Thread thread;      //marcado com "transient" para que sua instância não seja serializada        //Demais valores de estado serão serializados      private int valorParaPersistir;        public ImplementacaoTransient(int valorParaPersistir){          this.valorParaPersistir = valorParaPersistir;          thread = new Thread(this);          thread.start();      }        public void run()      {          while(true){              //código a ser executado pela Thread          }      }  } |

Vale ressaltar que, o que é serializado é apenas o estado do objeto, e não os métodos e a classe. No exemplo acima, apenas a instância do objeto “thread” e do objeto “valorParaPersistir” compunham o estado do objeto. Assim, evita-se que o sistema de serialização tente serializar um objeto não-serializável e gere erros. Após a deserialização de um objeto da classe “ImplementacaoTransient” este estará totalmente funcional.

Um outro problema que pode vir a ocorrer, se refere ao versionamento das classes. Toda classe possui um atributo oculto chamado “serialVersionUID” com um valor gerado automaticamente. Objetos criados levam consigo o “serialVersionUID” de sua classe, que indica sua versão. Toda vez que uma classe sofre alterações em seu código, um novo “serialVersionUID” é gerado. O problema com a serialização surge quando serializa-se o estado de um objeto e posteriormente tenta-se restaurar o estado para a aplicação, mas houveram mudanças no código da classe. Neste caso, uma exceção “java.io.InvalidClassException” é disparada, pois de fato, o estado do objeto corresponde a uma versão de classe do qual foi criado que já não existe mais. Portanto, isso só ocorre para evitar erros de programação, em que tenta-se restaurar o estado de um objeto para uma classe que possui versão diferente da qual o objeto foi serializado inicialmente. E isso pode significar diferentes versões de uma mesma classe “Fruta” por exemplo, ou diferentes versões por classes serem totalmente difere

|  |  |
| --- | --- |
| http://www.linhadecodigo.com.br/artigos/img_artigos/EvertonCoimbraAraujo/Serializazao_Objetos/imagem001.jpg | Neste artigo apresento o Gustavo Rafael Valiati, que pode ser contatado pelo e-mail gustavovaliati@gmail.com. O Gustavo foi meu aluno em algumas disciplinas de sua gradução e está concluíndo o último período do curso de Tecnologia em Análise e Desenvolvimento de Sistemas da UTFPR campus Medianeira, neste primeiro semestre de 2012. |

**1. INTRODUÇÃO**

Nem sempre se deseja ter objetos ou estrutura de dados de um aplicativo, com um ciclo de vida que dura no máximo o tempo em que for armazenado em memória RAM. Ou seja, algumas vezes deseja-se guardá-los por um tempo indeterminado, sendo possível recuperá-los posteriormente. E neste caso, o termo “persistência” tem seu conceito definido.

Persistir dados, requer que estes possam ser serializáveis. A serialização possibilitará capturar o estado do objeto ou a estrutura de dados, e transformar em uma cadeia de bytes quando for necessário. Torna-se possível também, recuperar os bytes persistidos para fazer o processo inverso, e ter os dados de volta para a aplicação em execução.

Desta maneira, qualquer forma de dados que esteja sendo trabalhada na aplicação, e esta puder ser serializada, poderá ser passada para o meio físico, e ter o processo revertido.

A serialização não se limita apenas em possibilitar a gravação do dado em disco, mas permite que o mesmo seja transmitido por rede. Isto porquê a serialização cria um stream de bytes, que é requisito para transmissão de dados por rede.

Diversas linguagens possuem suporte à serialização, como: Java, C, C++, Python, PHP, .NET, entre outras. Neste artigo, será tratado apenas da serialização com Java, em sua implemetação básica, alguns problemas e soluções, com códigos para exemplificação.

**2. SERIALIZAÇÃO COM JAVA**

Serializar um objeto, em Java, só é possível caso sua classe esteja marcada como serializável. Essa marcação é feita com a simples implementação da interface “java.io.Serializable”. Essa interface não possui métodos para se implementar e nem atributos. Ainda não existiam as anotações no Java 1.1, versão que trouxe pela primeira vez a serialização ao Java, e desta maneira a implementação da interface era a melhor maneira de marcar uma classe.

A herança entre classes é naturalmente afetada pela serialização. Já que quando uma classe implementa a interface “Serializable”, toda classe que dela estender também estará implicitamente marcada como serializável

.

Para exemplo, será criada uma classe “pojo.Fruta” que contém dois atributos: nome e cor. No código exemplo (Listagem 1), pode-se notar a classe “Fruta” marcada para a serialização:

Listagem 1 - Marcando uma classe como serializável.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | import java.io.Serializable;    public class Fruta implements Serializable{      private String nome;      private String cor;        public Fruta(String nome, String cor) {          this.nome = nome;          this.cor = cor;      }        public String getCor() { return cor; }      public String getNome() { return nome; }      public void setCor(String cor) { this.cor = cor; }      public void setNome(String nome) { this.nome = nome; }  } |

Pode-se dizer que a implementação do processo de serialização e deserialização simples de um objeto, não possui um código difícil de se escrever. Isto porquê duas classes fazem a maior parte do trabalho, de maneira que com poucas linhas de código pode-se transformar um objeto em bytes e vice-versa. Essas duas classes são “java.io.ObjectInputStream” que recebe bytes para criar um objeto e “java.io.ObjectOutputStream” que cria uma cadeia de bytes a partir de um objeto.

**3. SERIALIZANDO E DESERIALIZANDO O ESTADO DE UM OBJETO**

Como exemplo, será persistido um objeto em disco. Para isto, além das classes “java.io.ObjectInputStream” e “java.io.ObjectOutputStream”, serão utilizadas duas outras que trabalham com arquivos, que são “java.io.FileInputStream” e “java.io.FileOutputStream”, que seguem a mesma lógica de InputStream e OutputStream, só que agora relacionado ao sistema de arquivos e não aos objetos.

Para serialização, deve-se criar um novo “java.io.FileOutputStream”, com o caminho (“path”) do arquivo onde se deseja armazenar o estado do objeto. Também deve ser criado um novo “java.io.ObjectOutputStream” que está ligado ao arquivo. E por último deve-se executar a escrita do objeto alvo, para dentro do objeto de stream que por sua vez está ligado ao arquivo destino. Após executados esses passos, é possível encontrar no sistema de arquivos um novo arquivo que contém os bytes correspondentes à descrição do estado do objeto. Na Listagem 2, encontra-se o código exemplo.

Listagem 2 - Serialização de um objeto.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | import java.io.FileOutputStream;  import java.io.ObjectOutputStream;    public class Serializador {        public Serializador() {    }        public void serializar(String path, Object obj) throws Exception {              FileOutputStream outFile = new FileOutputStream(path);      ObjectOutputStream s = new ObjectOutputStream(outFile);      s.writeObject(obj);      s.close();      }  } |

E para a deserialização, o processo é extremamente parecido com a serialização. Cria-se um novo “java.io.FileInputStream” relacionado ao arquivo onde o estado do objeto encontra-se persistido. Deve-se criar também um novo “java.io.ObjectInputStream” que estará vinculado ao arquivo de origem. Por último, executa-se a leitura do arquivo para a restauração do estado. Na Listagem 3 encontra-se o exemplo do código.

Listagem 3 - Deserializaçãode um objeto.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | import java.io.FileInputStream;  import java.io.ObjectInputStream;    public class Deserializador {        public Deserializador() {    }        public Object deserializar(String path) throws Exception {      FileInputStream inFile = new FileInputStream(path);      ObjectInputStream d = new ObjectInputStream(inFile);      Object o = d.readObject();      d.close();      return o;      }  } |

A Listagem 4, contém o exemplo da execução da serialização e em seguida da deserialização.

Listagem 4 - Serializando e Deserializando

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | import operacoes.Deserializador;  import operacoes.Serializador;  import pojo.Fruta;    public class Principal {      public static void main(String args[]){         //serializa         Serializador s = new Serializador();         Fruta fruta = new Fruta("Maça", "Vermelha");          try {              s.serializar("/home/gustavo/fruta", fruta);          } catch (Exception ex) {              System.err.println("Falha ao serializar! - " + ex.toString());          }            //deserializa          Deserializador d = new Deserializador();          fruta = null;          try {              fruta = (Fruta) d.deserializar("/home/gustavo/fruta");          } catch (Exception ex) {              System.err.println("Falha ao deserializar! - " + ex.toString());          }          System.out.println(fruta.getNome() + " - " + fruta.getCor());      }  } |

Na Listagem 5, encontra-se um exemplo de serialização e deserialização de um ArrayList da classe Fruta. É apenas mais um exemplo que demonstra a capacidade de se persistir qualquer objeto que seja serializável. Essa serialização só foi possível porque a classe “ArrayList” implementa a interface “Serializable” por padrão.

Listagem 5 - Serialização de ArrayList.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | //serialização e deserialização com ArrayList          ArrayList<Fruta> frutas = new ArrayList<Fruta>();          Fruta f1 = new Fruta("Laranja", "Amarela");          Fruta f2 = new Fruta("Abacate", "Verde");          Fruta f3 = new Fruta("Morango", "Vermelho");          frutas.add(f1);          frutas.add(f2);          frutas.add(f3);            try {              s.serializar("/home/gustavo/frutas", frutas);              frutas = null;              frutas = (ArrayList<Fruta>) d.deserializar("/home/gustavo/frutas");              for(Fruta f : frutas){                  System.out.println("ArrayList: " + f.getNome() + " - " + f.getCor());              }          } catch (Exception ex) {              System.err.println("Falha ao serializar ou deserializar! - " +                                 ex.toString());          } |

**4. PARTICULARIDADES DA SERIALIZAÇÃO**

Como dito, a serialização de um objeto em Java pode ser algo fácil de se fazer. Mas existem certas informações que são importantes e sabê-las pode evitar vários problemas.

Um dos problemas é que nem tudo pode ser serializado, como objetos das classes “Thread” e “Socket” e atributos estáticos. Os atributos estáticos estão vinculados à classe e não ao objeto. Assim quando se trata do estado do objeto, os campos estáticos não são considerados. Em relação às Threads, suas instâncias só são úteis enquanto estiverem em execução na JVM, e podem ser paradas/destruídas e iniciadas com instruções de sua classe. Não existe algo no estado destas instâncias que seja útil para se persistir, e por isso não há sentido em persistir tais instâncias para restaurar posteriormente.

Mas e se existir alguma classe em que seja necessário persisti-la, e esta possui uma instância da classe “Thread”? Neste caso, utilizando um recurso pode-se persistir a classe inteira, menos a instância thread. Este recurso, refere-se em indicar que tal instância thread é um objeto “Transient”. Na Listagem 6 encontra-se um exemplo da implementação com uso de “Transient”.

Listagem 6 - Uso do Transient

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | import java.io.Serializable;    public class ImplementacaoTransient implements Serializable, Runnable{        transient private Thread thread;      //marcado com "transient" para que sua instância não seja serializada        //Demais valores de estado serão serializados      private int valorParaPersistir;        public ImplementacaoTransient(int valorParaPersistir){          this.valorParaPersistir = valorParaPersistir;          thread = new Thread(this);          thread.start();      }        public void run()      {          while(true){              //código a ser executado pela Thread          }      }  } |

Vale ressaltar que, o que é serializado é apenas o estado do objeto, e não os métodos e a classe. No exemplo acima, apenas a instância do objeto “thread” e do objeto “valorParaPersistir” compunham o estado do objeto. Assim, evita-se que o sistema de serialização tente serializar um objeto não-serializável e gere erros. Após a deserialização de um objeto da classe “ImplementacaoTransient” este estará totalmente funcional.

Um outro problema que pode vir a ocorrer, se refere ao versionamento das classes. Toda classe possui um atributo oculto chamado “serialVersionUID” com um valor gerado automaticamente. Objetos criados levam consigo o “serialVersionUID” de sua classe, que indica sua versão. Toda vez que uma classe sofre alterações em seu código, um novo “serialVersionUID” é gerado. O problema com a serialização surge quando serializa-se o estado de um objeto e posteriormente tenta-se restaurar o estado para a aplicação, mas houveram mudanças no código da classe. Neste caso, uma exceção “java.io.InvalidClassException” é disparada, pois de fato, o estado do objeto corresponde a uma versão de classe do qual foi criado que já não existe mais. Portanto, isso só ocorre para evitar erros de programação, em que tenta-se restaurar o estado de um objeto para uma classe que possui versão diferente da qual o objeto foi serializado inicialmente. E isso pode significar diferentes versões de uma mesma classe “Fruta” por exemplo, ou diferentes versões por classes serem totalmente difere

|  |  |
| --- | --- |
| http://www.linhadecodigo.com.br/artigos/img_artigos/EvertonCoimbraAraujo/Serializazao_Objetos/imagem001.jpg | Neste artigo apresento o Gustavo Rafael Valiati, que pode ser contatado pelo e-mail gustavovaliati@gmail.com. O Gustavo foi meu aluno em algumas disciplinas de sua gradução e está concluíndo o último período do curso de Tecnologia em Análise e Desenvolvimento de Sistemas da UTFPR campus Medianeira, neste primeiro semestre de 2012. |

**1. INTRODUÇÃO**

Nem sempre se deseja ter objetos ou estrutura de dados de um aplicativo, com um ciclo de vida que dura no máximo o tempo em que for armazenado em memória RAM. Ou seja, algumas vezes deseja-se guardá-los por um tempo indeterminado, sendo possível recuperá-los posteriormente. E neste caso, o termo “persistência” tem seu conceito definido.

Persistir dados, requer que estes possam ser serializáveis. A serialização possibilitará capturar o estado do objeto ou a estrutura de dados, e transformar em uma cadeia de bytes quando for necessário. Torna-se possível também, recuperar os bytes persistidos para fazer o processo inverso, e ter os dados de volta para a aplicação em execução.

Desta maneira, qualquer forma de dados que esteja sendo trabalhada na aplicação, e esta puder ser serializada, poderá ser passada para o meio físico, e ter o processo revertido.

A serialização não se limita apenas em possibilitar a gravação do dado em disco, mas permite que o mesmo seja transmitido por rede. Isto porquê a serialização cria um stream de bytes, que é requisito para transmissão de dados por rede.

Diversas linguagens possuem suporte à serialização, como: Java, C, C++, Python, PHP, .NET, entre outras. Neste artigo, será tratado apenas da serialização com Java, em sua implemetação básica, alguns problemas e soluções, com códigos para exemplificação.

**2. SERIALIZAÇÃO COM JAVA**

Serializar um objeto, em Java, só é possível caso sua classe esteja marcada como serializável. Essa marcação é feita com a simples implementação da interface “java.io.Serializable”. Essa interface não possui métodos para se implementar e nem atributos. Ainda não existiam as anotações no Java 1.1, versão que trouxe pela primeira vez a serialização ao Java, e desta maneira a implementação da interface era a melhor maneira de marcar uma classe.

A herança entre classes é naturalmente afetada pela serialização. Já que quando uma classe implementa a interface “Serializable”, toda classe que dela estender também estará implicitamente marcada como serializável

.

Para exemplo, será criada uma classe “pojo.Fruta” que contém dois atributos: nome e cor. No código exemplo (Listagem 1), pode-se notar a classe “Fruta” marcada para a serialização:

Listagem 1 - Marcando uma classe como serializável.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | import java.io.Serializable;    public class Fruta implements Serializable{      private String nome;      private String cor;        public Fruta(String nome, String cor) {          this.nome = nome;          this.cor = cor;      }        public String getCor() { return cor; }      public String getNome() { return nome; }      public void setCor(String cor) { this.cor = cor; }      public void setNome(String nome) { this.nome = nome; }  } |

Pode-se dizer que a implementação do processo de serialização e deserialização simples de um objeto, não possui um código difícil de se escrever. Isto porquê duas classes fazem a maior parte do trabalho, de maneira que com poucas linhas de código pode-se transformar um objeto em bytes e vice-versa. Essas duas classes são “java.io.ObjectInputStream” que recebe bytes para criar um objeto e “java.io.ObjectOutputStream” que cria uma cadeia de bytes a partir de um objeto.

**3. SERIALIZANDO E DESERIALIZANDO O ESTADO DE UM OBJETO**

Como exemplo, será persistido um objeto em disco. Para isto, além das classes “java.io.ObjectInputStream” e “java.io.ObjectOutputStream”, serão utilizadas duas outras que trabalham com arquivos, que são “java.io.FileInputStream” e “java.io.FileOutputStream”, que seguem a mesma lógica de InputStream e OutputStream, só que agora relacionado ao sistema de arquivos e não aos objetos.

Para serialização, deve-se criar um novo “java.io.FileOutputStream”, com o caminho (“path”) do arquivo onde se deseja armazenar o estado do objeto. Também deve ser criado um novo “java.io.ObjectOutputStream” que está ligado ao arquivo. E por último deve-se executar a escrita do objeto alvo, para dentro do objeto de stream que por sua vez está ligado ao arquivo destino. Após executados esses passos, é possível encontrar no sistema de arquivos um novo arquivo que contém os bytes correspondentes à descrição do estado do objeto. Na Listagem 2, encontra-se o código exemplo.

Listagem 2 - Serialização de um objeto.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | import java.io.FileOutputStream;  import java.io.ObjectOutputStream;    public class Serializador {        public Serializador() {    }        public void serializar(String path, Object obj) throws Exception {              FileOutputStream outFile = new FileOutputStream(path);      ObjectOutputStream s = new ObjectOutputStream(outFile);      s.writeObject(obj);      s.close();      }  } |

E para a deserialização, o processo é extremamente parecido com a serialização. Cria-se um novo “java.io.FileInputStream” relacionado ao arquivo onde o estado do objeto encontra-se persistido. Deve-se criar também um novo “java.io.ObjectInputStream” que estará vinculado ao arquivo de origem. Por último, executa-se a leitura do arquivo para a restauração do estado. Na Listagem 3 encontra-se o exemplo do código.

Listagem 3 - Deserializaçãode um objeto.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | import java.io.FileInputStream;  import java.io.ObjectInputStream;    public class Deserializador {        public Deserializador() {    }        public Object deserializar(String path) throws Exception {      FileInputStream inFile = new FileInputStream(path);      ObjectInputStream d = new ObjectInputStream(inFile);      Object o = d.readObject();      d.close();      return o;      }  } |

A Listagem 4, contém o exemplo da execução da serialização e em seguida da deserialização.

Listagem 4 - Serializando e Deserializando

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | import operacoes.Deserializador;  import operacoes.Serializador;  import pojo.Fruta;    public class Principal {      public static void main(String args[]){         //serializa         Serializador s = new Serializador();         Fruta fruta = new Fruta("Maça", "Vermelha");          try {              s.serializar("/home/gustavo/fruta", fruta);          } catch (Exception ex) {              System.err.println("Falha ao serializar! - " + ex.toString());          }            //deserializa          Deserializador d = new Deserializador();          fruta = null;          try {              fruta = (Fruta) d.deserializar("/home/gustavo/fruta");          } catch (Exception ex) {              System.err.println("Falha ao deserializar! - " + ex.toString());          }          System.out.println(fruta.getNome() + " - " + fruta.getCor());      }  } |

Na Listagem 5, encontra-se um exemplo de serialização e deserialização de um ArrayList da classe Fruta. É apenas mais um exemplo que demonstra a capacidade de se persistir qualquer objeto que seja serializável. Essa serialização só foi possível porque a classe “ArrayList” implementa a interface “Serializable” por padrão.

Listagem 5 - Serialização de ArrayList.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | //serialização e deserialização com ArrayList          ArrayList<Fruta> frutas = new ArrayList<Fruta>();          Fruta f1 = new Fruta("Laranja", "Amarela");          Fruta f2 = new Fruta("Abacate", "Verde");          Fruta f3 = new Fruta("Morango", "Vermelho");          frutas.add(f1);          frutas.add(f2);          frutas.add(f3);            try {              s.serializar("/home/gustavo/frutas", frutas);              frutas = null;              frutas = (ArrayList<Fruta>) d.deserializar("/home/gustavo/frutas");              for(Fruta f : frutas){                  System.out.println("ArrayList: " + f.getNome() + " - " + f.getCor());              }          } catch (Exception ex) {              System.err.println("Falha ao serializar ou deserializar! - " +                                 ex.toString());          } |

**4. PARTICULARIDADES DA SERIALIZAÇÃO**

Como dito, a serialização de um objeto em Java pode ser algo fácil de se fazer. Mas existem certas informações que são importantes e sabê-las pode evitar vários problemas.

Um dos problemas é que nem tudo pode ser serializado, como objetos das classes “Thread” e “Socket” e atributos estáticos. Os atributos estáticos estão vinculados à classe e não ao objeto. Assim quando se trata do estado do objeto, os campos estáticos não são considerados. Em relação às Threads, suas instâncias só são úteis enquanto estiverem em execução na JVM, e podem ser paradas/destruídas e iniciadas com instruções de sua classe. Não existe algo no estado destas instâncias que seja útil para se persistir, e por isso não há sentido em persistir tais instâncias para restaurar posteriormente.

Mas e se existir alguma classe em que seja necessário persisti-la, e esta possui uma instância da classe “Thread”? Neste caso, utilizando um recurso pode-se persistir a classe inteira, menos a instância thread. Este recurso, refere-se em indicar que tal instância thread é um objeto “Transient”. Na Listagem 6 encontra-se um exemplo da implementação com uso de “Transient”.

Listagem 6 - Uso do Transient

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | import java.io.Serializable;    public class ImplementacaoTransient implements Serializable, Runnable{        transient private Thread thread;      //marcado com "transient" para que sua instância não seja serializada        //Demais valores de estado serão serializados      private int valorParaPersistir;        public ImplementacaoTransient(int valorParaPersistir){          this.valorParaPersistir = valorParaPersistir;          thread = new Thread(this);          thread.start();      }        public void run()      {          while(true){              //código a ser executado pela Thread          }      }  } |

Vale ressaltar que, o que é serializado é apenas o estado do objeto, e não os métodos e a classe. No exemplo acima, apenas a instância do objeto “thread” e do objeto “valorParaPersistir” compunham o estado do objeto. Assim, evita-se que o sistema de serialização tente serializar um objeto não-serializável e gere erros. Após a deserialização de um objeto da classe “ImplementacaoTransient” este estará totalmente funcional.

Um outro problema que pode vir a ocorrer, se refere ao versionamento das classes. Toda classe possui um atributo oculto chamado “serialVersionUID” com um valor gerado automaticamente. Objetos criados levam consigo o “serialVersionUID” de sua classe, que indica sua versão. Toda vez que uma classe sofre alterações em seu código, um novo “serialVersionUID” é gerado. O problema com a serialização surge quando serializa-se o estado de um objeto e posteriormente tenta-se restaurar o estado para a aplicação, mas houveram mudanças no código da classe. Neste caso, uma exceção “java.io.InvalidClassException” é disparada, pois de fato, o estado do objeto corresponde a uma versão de classe do qual foi criado que já não existe mais. Portanto, isso só ocorre para evitar erros de programação, em que tenta-se restaurar o estado de um objeto para uma classe que possui versão diferente da qual o objeto foi serializado inicialmente. E isso pode significar diferentes versões de uma mesma classe “Fruta” por exemplo, ou diferentes versões por classes serem totalmente difere

|  |  |
| --- | --- |
| http://www.linhadecodigo.com.br/artigos/img_artigos/EvertonCoimbraAraujo/Serializazao_Objetos/imagem001.jpg | Neste artigo apresento o Gustavo Rafael Valiati, que pode ser contatado pelo e-mail gustavovaliati@gmail.com. O Gustavo foi meu aluno em algumas disciplinas de sua gradução e está concluíndo o último período do curso de Tecnologia em Análise e Desenvolvimento de Sistemas da UTFPR campus Medianeira, neste primeiro semestre de 2012. |

**1. INTRODUÇÃO**

Nem sempre se deseja ter objetos ou estrutura de dados de um aplicativo, com um ciclo de vida que dura no máximo o tempo em que for armazenado em memória RAM. Ou seja, algumas vezes deseja-se guardá-los por um tempo indeterminado, sendo possível recuperá-los posteriormente. E neste caso, o termo “persistência” tem seu conceito definido.

Persistir dados, requer que estes possam ser serializáveis. A serialização possibilitará capturar o estado do objeto ou a estrutura de dados, e transformar em uma cadeia de bytes quando for necessário. Torna-se possível também, recuperar os bytes persistidos para fazer o processo inverso, e ter os dados de volta para a aplicação em execução.

Desta maneira, qualquer forma de dados que esteja sendo trabalhada na aplicação, e esta puder ser serializada, poderá ser passada para o meio físico, e ter o processo revertido.

A serialização não se limita apenas em possibilitar a gravação do dado em disco, mas permite que o mesmo seja transmitido por rede. Isto porquê a serialização cria um stream de bytes, que é requisito para transmissão de dados por rede.

Diversas linguagens possuem suporte à serialização, como: Java, C, C++, Python, PHP, .NET, entre outras. Neste artigo, será tratado apenas da serialização com Java, em sua implemetação básica, alguns problemas e soluções, com códigos para exemplificação.

**2. SERIALIZAÇÃO COM JAVA**

Serializar um objeto, em Java, só é possível caso sua classe esteja marcada como serializável. Essa marcação é feita com a simples implementação da interface “java.io.Serializable”. Essa interface não possui métodos para se implementar e nem atributos. Ainda não existiam as anotações no Java 1.1, versão que trouxe pela primeira vez a serialização ao Java, e desta maneira a implementação da interface era a melhor maneira de marcar uma classe.

A herança entre classes é naturalmente afetada pela serialização. Já que quando uma classe implementa a interface “Serializable”, toda classe que dela estender também estará implicitamente marcada como serializável

.

Para exemplo, será criada uma classe “pojo.Fruta” que contém dois atributos: nome e cor. No código exemplo (Listagem 1), pode-se notar a classe “Fruta” marcada para a serialização:

Listagem 1 - Marcando uma classe como serializável.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | import java.io.Serializable;    public class Fruta implements Serializable{      private String nome;      private String cor;        public Fruta(String nome, String cor) {          this.nome = nome;          this.cor = cor;      }        public String getCor() { return cor; }      public String getNome() { return nome; }      public void setCor(String cor) { this.cor = cor; }      public void setNome(String nome) { this.nome = nome; }  } |

Pode-se dizer que a implementação do processo de serialização e deserialização simples de um objeto, não possui um código difícil de se escrever. Isto porquê duas classes fazem a maior parte do trabalho, de maneira que com poucas linhas de código pode-se transformar um objeto em bytes e vice-versa. Essas duas classes são “java.io.ObjectInputStream” que recebe bytes para criar um objeto e “java.io.ObjectOutputStream” que cria uma cadeia de bytes a partir de um objeto.

**3. SERIALIZANDO E DESERIALIZANDO O ESTADO DE UM OBJETO**

Como exemplo, será persistido um objeto em disco. Para isto, além das classes “java.io.ObjectInputStream” e “java.io.ObjectOutputStream”, serão utilizadas duas outras que trabalham com arquivos, que são “java.io.FileInputStream” e “java.io.FileOutputStream”, que seguem a mesma lógica de InputStream e OutputStream, só que agora relacionado ao sistema de arquivos e não aos objetos.

Para serialização, deve-se criar um novo “java.io.FileOutputStream”, com o caminho (“path”) do arquivo onde se deseja armazenar o estado do objeto. Também deve ser criado um novo “java.io.ObjectOutputStream” que está ligado ao arquivo. E por último deve-se executar a escrita do objeto alvo, para dentro do objeto de stream que por sua vez está ligado ao arquivo destino. Após executados esses passos, é possível encontrar no sistema de arquivos um novo arquivo que contém os bytes correspondentes à descrição do estado do objeto. Na Listagem 2, encontra-se o código exemplo.

Listagem 2 - Serialização de um objeto.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | import java.io.FileOutputStream;  import java.io.ObjectOutputStream;    public class Serializador {        public Serializador() {    }        public void serializar(String path, Object obj) throws Exception {              FileOutputStream outFile = new FileOutputStream(path);      ObjectOutputStream s = new ObjectOutputStream(outFile);      s.writeObject(obj);      s.close();      }  } |

E para a deserialização, o processo é extremamente parecido com a serialização. Cria-se um novo “java.io.FileInputStream” relacionado ao arquivo onde o estado do objeto encontra-se persistido. Deve-se criar também um novo “java.io.ObjectInputStream” que estará vinculado ao arquivo de origem. Por último, executa-se a leitura do arquivo para a restauração do estado. Na Listagem 3 encontra-se o exemplo do código.

Listagem 3 - Deserializaçãode um objeto.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | import java.io.FileInputStream;  import java.io.ObjectInputStream;    public class Deserializador {        public Deserializador() {    }        public Object deserializar(String path) throws Exception {      FileInputStream inFile = new FileInputStream(path);      ObjectInputStream d = new ObjectInputStream(inFile);      Object o = d.readObject();      d.close();      return o;      }  } |

A Listagem 4, contém o exemplo da execução da serialização e em seguida da deserialização.

Listagem 4 - Serializando e Deserializando

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | import operacoes.Deserializador;  import operacoes.Serializador;  import pojo.Fruta;    public class Principal {      public static void main(String args[]){         //serializa         Serializador s = new Serializador();         Fruta fruta = new Fruta("Maça", "Vermelha");          try {              s.serializar("/home/gustavo/fruta", fruta);          } catch (Exception ex) {              System.err.println("Falha ao serializar! - " + ex.toString());          }            //deserializa          Deserializador d = new Deserializador();          fruta = null;          try {              fruta = (Fruta) d.deserializar("/home/gustavo/fruta");          } catch (Exception ex) {              System.err.println("Falha ao deserializar! - " + ex.toString());          }          System.out.println(fruta.getNome() + " - " + fruta.getCor());      }  } |

Na Listagem 5, encontra-se um exemplo de serialização e deserialização de um ArrayList da classe Fruta. É apenas mais um exemplo que demonstra a capacidade de se persistir qualquer objeto que seja serializável. Essa serialização só foi possível porque a classe “ArrayList” implementa a interface “Serializable” por padrão.

Listagem 5 - Serialização de ArrayList.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | //serialização e deserialização com ArrayList          ArrayList<Fruta> frutas = new ArrayList<Fruta>();          Fruta f1 = new Fruta("Laranja", "Amarela");          Fruta f2 = new Fruta("Abacate", "Verde");          Fruta f3 = new Fruta("Morango", "Vermelho");          frutas.add(f1);          frutas.add(f2);          frutas.add(f3);            try {              s.serializar("/home/gustavo/frutas", frutas);              frutas = null;              frutas = (ArrayList<Fruta>) d.deserializar("/home/gustavo/frutas");              for(Fruta f : frutas){                  System.out.println("ArrayList: " + f.getNome() + " - " + f.getCor());              }          } catch (Exception ex) {              System.err.println("Falha ao serializar ou deserializar! - " +                                 ex.toString());          } |

**4. PARTICULARIDADES DA SERIALIZAÇÃO**

Como dito, a serialização de um objeto em Java pode ser algo fácil de se fazer. Mas existem certas informações que são importantes e sabê-las pode evitar vários problemas.

Um dos problemas é que nem tudo pode ser serializado, como objetos das classes “Thread” e “Socket” e atributos estáticos. Os atributos estáticos estão vinculados à classe e não ao objeto. Assim quando se trata do estado do objeto, os campos estáticos não são considerados. Em relação às Threads, suas instâncias só são úteis enquanto estiverem em execução na JVM, e podem ser paradas/destruídas e iniciadas com instruções de sua classe. Não existe algo no estado destas instâncias que seja útil para se persistir, e por isso não há sentido em persistir tais instâncias para restaurar posteriormente.

Mas e se existir alguma classe em que seja necessário persisti-la, e esta possui uma instância da classe “Thread”? Neste caso, utilizando um recurso pode-se persistir a classe inteira, menos a instância thread. Este recurso, refere-se em indicar que tal instância thread é um objeto “Transient”. Na Listagem 6 encontra-se um exemplo da implementação com uso de “Transient”.

Listagem 6 - Uso do Transient

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | import java.io.Serializable;    public class ImplementacaoTransient implements Serializable, Runnable{        transient private Thread thread;      //marcado com "transient" para que sua instância não seja serializada        //Demais valores de estado serão serializados      private int valorParaPersistir;        public ImplementacaoTransient(int valorParaPersistir){          this.valorParaPersistir = valorParaPersistir;          thread = new Thread(this);          thread.start();      }        public void run()      {          while(true){              //código a ser executado pela Thread          }      }  } |

Vale ressaltar que, o que é serializado é apenas o estado do objeto, e não os métodos e a classe. No exemplo acima, apenas a instância do objeto “thread” e do objeto “valorParaPersistir” compunham o estado do objeto. Assim, evita-se que o sistema de serialização tente serializar um objeto não-serializável e gere erros. Após a deserialização de um objeto da classe “ImplementacaoTransient” este estará totalmente funcional.

Um outro problema que pode vir a ocorrer, se refere ao versionamento das classes. Toda classe possui um atributo oculto chamado “serialVersionUID” com um valor gerado automaticamente. Objetos criados levam consigo o “serialVersionUID” de sua classe, que indica sua versão. Toda vez que uma classe sofre alterações em seu código, um novo “serialVersionUID” é gerado. O problema com a serialização surge quando serializa-se o estado de um objeto e posteriormente tenta-se restaurar o estado para a aplicação, mas houveram mudanças no código da classe. Neste caso, uma exceção “java.io.InvalidClassException” é disparada, pois de fato, o estado do objeto corresponde a uma versão de classe do qual foi criado que já não existe mais. Portanto, isso só ocorre para evitar erros de programação, em que tenta-se restaurar o estado de um objeto para uma classe que possui versão diferente da qual o objeto foi serializado inicialmente. E isso pode significar diferentes versões de uma mesma classe “Fruta” por exemplo, ou diferentes versões por classes serem totalmente difere

|  |  |
| --- | --- |
| http://www.linhadecodigo.com.br/artigos/img_artigos/EvertonCoimbraAraujo/Serializazao_Objetos/imagem001.jpg | Neste artigo apresento o Gustavo Rafael Valiati, que pode ser contatado pelo e-mail gustavovaliati@gmail.com. O Gustavo foi meu aluno em algumas disciplinas de sua gradução e está concluíndo o último período do curso de Tecnologia em Análise e Desenvolvimento de Sistemas da UTFPR campus Medianeira, neste primeiro semestre de 2012. |

**1. INTRODUÇÃO**

Nem sempre se deseja ter objetos ou estrutura de dados de um aplicativo, com um ciclo de vida que dura no máximo o tempo em que for armazenado em memória RAM. Ou seja, algumas vezes deseja-se guardá-los por um tempo indeterminado, sendo possível recuperá-los posteriormente. E neste caso, o termo “persistência” tem seu conceito definido.

Persistir dados, requer que estes possam ser serializáveis. A serialização possibilitará capturar o estado do objeto ou a estrutura de dados, e transformar em uma cadeia de bytes quando for necessário. Torna-se possível também, recuperar os bytes persistidos para fazer o processo inverso, e ter os dados de volta para a aplicação em execução.

Desta maneira, qualquer forma de dados que esteja sendo trabalhada na aplicação, e esta puder ser serializada, poderá ser passada para o meio físico, e ter o processo revertido.

A serialização não se limita apenas em possibilitar a gravação do dado em disco, mas permite que o mesmo seja transmitido por rede. Isto porquê a serialização cria um stream de bytes, que é requisito para transmissão de dados por rede.

Diversas linguagens possuem suporte à serialização, como: Java, C, C++, Python, PHP, .NET, entre outras. Neste artigo, será tratado apenas da serialização com Java, em sua implemetação básica, alguns problemas e soluções, com códigos para exemplificação.

**2. SERIALIZAÇÃO COM JAVA**

Serializar um objeto, em Java, só é possível caso sua classe esteja marcada como serializável. Essa marcação é feita com a simples implementação da interface “java.io.Serializable”. Essa interface não possui métodos para se implementar e nem atributos. Ainda não existiam as anotações no Java 1.1, versão que trouxe pela primeira vez a serialização ao Java, e desta maneira a implementação da interface era a melhor maneira de marcar uma classe.

A herança entre classes é naturalmente afetada pela serialização. Já que quando uma classe implementa a interface “Serializable”, toda classe que dela estender também estará implicitamente marcada como serializável

.

Para exemplo, será criada uma classe “pojo.Fruta” que contém dois atributos: nome e cor. No código exemplo (Listagem 1), pode-se notar a classe “Fruta” marcada para a serialização:

Listagem 1 - Marcando uma classe como serializável.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | import java.io.Serializable;    public class Fruta implements Serializable{      private String nome;      private String cor;        public Fruta(String nome, String cor) {          this.nome = nome;          this.cor = cor;      }        public String getCor() { return cor; }      public String getNome() { return nome; }      public void setCor(String cor) { this.cor = cor; }      public void setNome(String nome) { this.nome = nome; }  } |

Pode-se dizer que a implementação do processo de serialização e deserialização simples de um objeto, não possui um código difícil de se escrever. Isto porquê duas classes fazem a maior parte do trabalho, de maneira que com poucas linhas de código pode-se transformar um objeto em bytes e vice-versa. Essas duas classes são “java.io.ObjectInputStream” que recebe bytes para criar um objeto e “java.io.ObjectOutputStream” que cria uma cadeia de bytes a partir de um objeto.

**3. SERIALIZANDO E DESERIALIZANDO O ESTADO DE UM OBJETO**

Como exemplo, será persistido um objeto em disco. Para isto, além das classes “java.io.ObjectInputStream” e “java.io.ObjectOutputStream”, serão utilizadas duas outras que trabalham com arquivos, que são “java.io.FileInputStream” e “java.io.FileOutputStream”, que seguem a mesma lógica de InputStream e OutputStream, só que agora relacionado ao sistema de arquivos e não aos objetos.

Para serialização, deve-se criar um novo “java.io.FileOutputStream”, com o caminho (“path”) do arquivo onde se deseja armazenar o estado do objeto. Também deve ser criado um novo “java.io.ObjectOutputStream” que está ligado ao arquivo. E por último deve-se executar a escrita do objeto alvo, para dentro do objeto de stream que por sua vez está ligado ao arquivo destino. Após executados esses passos, é possível encontrar no sistema de arquivos um novo arquivo que contém os bytes correspondentes à descrição do estado do objeto. Na Listagem 2, encontra-se o código exemplo.

Listagem 2 - Serialização de um objeto.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | import java.io.FileOutputStream;  import java.io.ObjectOutputStream;    public class Serializador {        public Serializador() {    }        public void serializar(String path, Object obj) throws Exception {              FileOutputStream outFile = new FileOutputStream(path);      ObjectOutputStream s = new ObjectOutputStream(outFile);      s.writeObject(obj);      s.close();      }  } |

E para a deserialização, o processo é extremamente parecido com a serialização. Cria-se um novo “java.io.FileInputStream” relacionado ao arquivo onde o estado do objeto encontra-se persistido. Deve-se criar também um novo “java.io.ObjectInputStream” que estará vinculado ao arquivo de origem. Por último, executa-se a leitura do arquivo para a restauração do estado. Na Listagem 3 encontra-se o exemplo do código.

Listagem 3 - Deserializaçãode um objeto.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | import java.io.FileInputStream;  import java.io.ObjectInputStream;    public class Deserializador {        public Deserializador() {    }        public Object deserializar(String path) throws Exception {      FileInputStream inFile = new FileInputStream(path);      ObjectInputStream d = new ObjectInputStream(inFile);      Object o = d.readObject();      d.close();      return o;      }  } |

A Listagem 4, contém o exemplo da execução da serialização e em seguida da deserialização.

Listagem 4 - Serializando e Deserializando

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | import operacoes.Deserializador;  import operacoes.Serializador;  import pojo.Fruta;    public class Principal {      public static void main(String args[]){         //serializa         Serializador s = new Serializador();         Fruta fruta = new Fruta("Maça", "Vermelha");          try {              s.serializar("/home/gustavo/fruta", fruta);          } catch (Exception ex) {              System.err.println("Falha ao serializar! - " + ex.toString());          }            //deserializa          Deserializador d = new Deserializador();          fruta = null;          try {              fruta = (Fruta) d.deserializar("/home/gustavo/fruta");          } catch (Exception ex) {              System.err.println("Falha ao deserializar! - " + ex.toString());          }          System.out.println(fruta.getNome() + " - " + fruta.getCor());      }  } |

Na Listagem 5, encontra-se um exemplo de serialização e deserialização de um ArrayList da classe Fruta. É apenas mais um exemplo que demonstra a capacidade de se persistir qualquer objeto que seja serializável. Essa serialização só foi possível porque a classe “ArrayList” implementa a interface “Serializable” por padrão.

Listagem 5 - Serialização de ArrayList.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | //serialização e deserialização com ArrayList          ArrayList<Fruta> frutas = new ArrayList<Fruta>();          Fruta f1 = new Fruta("Laranja", "Amarela");          Fruta f2 = new Fruta("Abacate", "Verde");          Fruta f3 = new Fruta("Morango", "Vermelho");          frutas.add(f1);          frutas.add(f2);          frutas.add(f3);            try {              s.serializar("/home/gustavo/frutas", frutas);              frutas = null;              frutas = (ArrayList<Fruta>) d.deserializar("/home/gustavo/frutas");              for(Fruta f : frutas){                  System.out.println("ArrayList: " + f.getNome() + " - " + f.getCor());              }          } catch (Exception ex) {              System.err.println("Falha ao serializar ou deserializar! - " +                                 ex.toString());          } |

**4. PARTICULARIDADES DA SERIALIZAÇÃO**

Como dito, a serialização de um objeto em Java pode ser algo fácil de se fazer. Mas existem certas informações que são importantes e sabê-las pode evitar vários problemas.

Um dos problemas é que nem tudo pode ser serializado, como objetos das classes “Thread” e “Socket” e atributos estáticos. Os atributos estáticos estão vinculados à classe e não ao objeto. Assim quando se trata do estado do objeto, os campos estáticos não são considerados. Em relação às Threads, suas instâncias só são úteis enquanto estiverem em execução na JVM, e podem ser paradas/destruídas e iniciadas com instruções de sua classe. Não existe algo no estado destas instâncias que seja útil para se persistir, e por isso não há sentido em persistir tais instâncias para restaurar posteriormente.

Mas e se existir alguma classe em que seja necessário persisti-la, e esta possui uma instância da classe “Thread”? Neste caso, utilizando um recurso pode-se persistir a classe inteira, menos a instância thread. Este recurso, refere-se em indicar que tal instância thread é um objeto “Transient”. Na Listagem 6 encontra-se um exemplo da implementação com uso de “Transient”.

Listagem 6 - Uso do Transient

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | import java.io.Serializable;    public class ImplementacaoTransient implements Serializable, Runnable{        transient private Thread thread;      //marcado com "transient" para que sua instância não seja serializada        //Demais valores de estado serão serializados      private int valorParaPersistir;        public ImplementacaoTransient(int valorParaPersistir){          this.valorParaPersistir = valorParaPersistir;          thread = new Thread(this);          thread.start();      }        public void run()      {          while(true){              //código a ser executado pela Thread          }      }  } |

Vale ressaltar que, o que é serializado é apenas o estado do objeto, e não os métodos e a classe. No exemplo acima, apenas a instância do objeto “thread” e do objeto “valorParaPersistir” compunham o estado do objeto. Assim, evita-se que o sistema de serialização tente serializar um objeto não-serializável e gere erros. Após a deserialização de um objeto da classe “ImplementacaoTransient” este estará totalmente funcional.

Um outro problema que pode vir a ocorrer, se refere ao versionamento das classes. Toda classe possui um atributo oculto chamado “serialVersionUID” com um valor gerado automaticamente. Objetos criados levam consigo o “serialVersionUID” de sua classe, que indica sua versão. Toda vez que uma classe sofre alterações em seu código, um novo “serialVersionUID” é gerado. O problema com a serialização surge quando serializa-se o estado de um objeto e posteriormente tenta-se restaurar o estado para a aplicação, mas houveram mudanças no código da classe. Neste caso, uma exceção “java.io.InvalidClassException” é disparada, pois de fato, o estado do objeto corresponde a uma versão de classe do qual foi criado que já não existe mais. Portanto, isso só ocorre para evitar erros de programação, em que tenta-se restaurar o estado de um objeto para uma classe que possui versão diferente da qual o objeto foi serializado inicialmente. E isso pode significar diferentes versões de uma mesma classe “Fruta” por exemplo, ou diferentes versões por classes serem totalmente difere

|  |  |
| --- | --- |
| http://www.linhadecodigo.com.br/artigos/img_artigos/EvertonCoimbraAraujo/Serializazao_Objetos/imagem001.jpg | Neste artigo apresento o Gustavo Rafael Valiati, que pode ser contatado pelo e-mail gustavovaliati@gmail.com. O Gustavo foi meu aluno em algumas disciplinas de sua gradução e está concluíndo o último período do curso de Tecnologia em Análise e Desenvolvimento de Sistemas da UTFPR campus Medianeira, neste primeiro semestre de 2012. |

**1. INTRODUÇÃO**

Nem sempre se deseja ter objetos ou estrutura de dados de um aplicativo, com um ciclo de vida que dura no máximo o tempo em que for armazenado em memória RAM. Ou seja, algumas vezes deseja-se guardá-los por um tempo indeterminado, sendo possível recuperá-los posteriormente. E neste caso, o termo “persistência” tem seu conceito definido.

Persistir dados, requer que estes possam ser serializáveis. A serialização possibilitará capturar o estado do objeto ou a estrutura de dados, e transformar em uma cadeia de bytes quando for necessário. Torna-se possível também, recuperar os bytes persistidos para fazer o processo inverso, e ter os dados de volta para a aplicação em execução.

Desta maneira, qualquer forma de dados que esteja sendo trabalhada na aplicação, e esta puder ser serializada, poderá ser passada para o meio físico, e ter o processo revertido.

A serialização não se limita apenas em possibilitar a gravação do dado em disco, mas permite que o mesmo seja transmitido por rede. Isto porquê a serialização cria um stream de bytes, que é requisito para transmissão de dados por rede.

Diversas linguagens possuem suporte à serialização, como: Java, C, C++, Python, PHP, .NET, entre outras. Neste artigo, será tratado apenas da serialização com Java, em sua implemetação básica, alguns problemas e soluções, com códigos para exemplificação.

**2. SERIALIZAÇÃO COM JAVA**

Serializar um objeto, em Java, só é possível caso sua classe esteja marcada como serializável. Essa marcação é feita com a simples implementação da interface “java.io.Serializable”. Essa interface não possui métodos para se implementar e nem atributos. Ainda não existiam as anotações no Java 1.1, versão que trouxe pela primeira vez a serialização ao Java, e desta maneira a implementação da interface era a melhor maneira de marcar uma classe.

A herança entre classes é naturalmente afetada pela serialização. Já que quando uma classe implementa a interface “Serializable”, toda classe que dela estender também estará implicitamente marcada como serializável

.

Para exemplo, será criada uma classe “pojo.Fruta” que contém dois atributos: nome e cor. No código exemplo (Listagem 1), pode-se notar a classe “Fruta” marcada para a serialização:

Listagem 1 - Marcando uma classe como serializável.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | import java.io.Serializable;    public class Fruta implements Serializable{      private String nome;      private String cor;        public Fruta(String nome, String cor) {          this.nome = nome;          this.cor = cor;      }        public String getCor() { return cor; }      public String getNome() { return nome; }      public void setCor(String cor) { this.cor = cor; }      public void setNome(String nome) { this.nome = nome; }  } |

Pode-se dizer que a implementação do processo de serialização e deserialização simples de um objeto, não possui um código difícil de se escrever. Isto porquê duas classes fazem a maior parte do trabalho, de maneira que com poucas linhas de código pode-se transformar um objeto em bytes e vice-versa. Essas duas classes são “java.io.ObjectInputStream” que recebe bytes para criar um objeto e “java.io.ObjectOutputStream” que cria uma cadeia de bytes a partir de um objeto.

**3. SERIALIZANDO E DESERIALIZANDO O ESTADO DE UM OBJETO**

Como exemplo, será persistido um objeto em disco. Para isto, além das classes “java.io.ObjectInputStream” e “java.io.ObjectOutputStream”, serão utilizadas duas outras que trabalham com arquivos, que são “java.io.FileInputStream” e “java.io.FileOutputStream”, que seguem a mesma lógica de InputStream e OutputStream, só que agora relacionado ao sistema de arquivos e não aos objetos.

Para serialização, deve-se criar um novo “java.io.FileOutputStream”, com o caminho (“path”) do arquivo onde se deseja armazenar o estado do objeto. Também deve ser criado um novo “java.io.ObjectOutputStream” que está ligado ao arquivo. E por último deve-se executar a escrita do objeto alvo, para dentro do objeto de stream que por sua vez está ligado ao arquivo destino. Após executados esses passos, é possível encontrar no sistema de arquivos um novo arquivo que contém os bytes correspondentes à descrição do estado do objeto. Na Listagem 2, encontra-se o código exemplo.

Listagem 2 - Serialização de um objeto.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | import java.io.FileOutputStream;  import java.io.ObjectOutputStream;    public class Serializador {        public Serializador() {    }        public void serializar(String path, Object obj) throws Exception {              FileOutputStream outFile = new FileOutputStream(path);      ObjectOutputStream s = new ObjectOutputStream(outFile);      s.writeObject(obj);      s.close();      }  } |

E para a deserialização, o processo é extremamente parecido com a serialização. Cria-se um novo “java.io.FileInputStream” relacionado ao arquivo onde o estado do objeto encontra-se persistido. Deve-se criar também um novo “java.io.ObjectInputStream” que estará vinculado ao arquivo de origem. Por último, executa-se a leitura do arquivo para a restauração do estado. Na Listagem 3 encontra-se o exemplo do código.

Listagem 3 - Deserializaçãode um objeto.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | import java.io.FileInputStream;  import java.io.ObjectInputStream;    public class Deserializador {        public Deserializador() {    }        public Object deserializar(String path) throws Exception {      FileInputStream inFile = new FileInputStream(path);      ObjectInputStream d = new ObjectInputStream(inFile);      Object o = d.readObject();      d.close();      return o;      }  } |

A Listagem 4, contém o exemplo da execução da serialização e em seguida da deserialização.

Listagem 4 - Serializando e Deserializando

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | import operacoes.Deserializador;  import operacoes.Serializador;  import pojo.Fruta;    public class Principal {      public static void main(String args[]){         //serializa         Serializador s = new Serializador();         Fruta fruta = new Fruta("Maça", "Vermelha");          try {              s.serializar("/home/gustavo/fruta", fruta);          } catch (Exception ex) {              System.err.println("Falha ao serializar! - " + ex.toString());          }            //deserializa          Deserializador d = new Deserializador();          fruta = null;          try {              fruta = (Fruta) d.deserializar("/home/gustavo/fruta");          } catch (Exception ex) {              System.err.println("Falha ao deserializar! - " + ex.toString());          }          System.out.println(fruta.getNome() + " - " + fruta.getCor());      }  } |

Na Listagem 5, encontra-se um exemplo de serialização e deserialização de um ArrayList da classe Fruta. É apenas mais um exemplo que demonstra a capacidade de se persistir qualquer objeto que seja serializável. Essa serialização só foi possível porque a classe “ArrayList” implementa a interface “Serializable” por padrão.

Listagem 5 - Serialização de ArrayList.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | //serialização e deserialização com ArrayList          ArrayList<Fruta> frutas = new ArrayList<Fruta>();          Fruta f1 = new Fruta("Laranja", "Amarela");          Fruta f2 = new Fruta("Abacate", "Verde");          Fruta f3 = new Fruta("Morango", "Vermelho");          frutas.add(f1);          frutas.add(f2);          frutas.add(f3);            try {              s.serializar("/home/gustavo/frutas", frutas);              frutas = null;              frutas = (ArrayList<Fruta>) d.deserializar("/home/gustavo/frutas");              for(Fruta f : frutas){                  System.out.println("ArrayList: " + f.getNome() + " - " + f.getCor());              }          } catch (Exception ex) {              System.err.println("Falha ao serializar ou deserializar! - " +                                 ex.toString());          } |

**4. PARTICULARIDADES DA SERIALIZAÇÃO**

Como dito, a serialização de um objeto em Java pode ser algo fácil de se fazer. Mas existem certas informações que são importantes e sabê-las pode evitar vários problemas.

Um dos problemas é que nem tudo pode ser serializado, como objetos das classes “Thread” e “Socket” e atributos estáticos. Os atributos estáticos estão vinculados à classe e não ao objeto. Assim quando se trata do estado do objeto, os campos estáticos não são considerados. Em relação às Threads, suas instâncias só são úteis enquanto estiverem em execução na JVM, e podem ser paradas/destruídas e iniciadas com instruções de sua classe. Não existe algo no estado destas instâncias que seja útil para se persistir, e por isso não há sentido em persistir tais instâncias para restaurar posteriormente.

Mas e se existir alguma classe em que seja necessário persisti-la, e esta possui uma instância da classe “Thread”? Neste caso, utilizando um recurso pode-se persistir a classe inteira, menos a instância thread. Este recurso, refere-se em indicar que tal instância thread é um objeto “Transient”. Na Listagem 6 encontra-se um exemplo da implementação com uso de “Transient”.

Listagem 6 - Uso do Transient

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | import java.io.Serializable;    public class ImplementacaoTransient implements Serializable, Runnable{        transient private Thread thread;      //marcado com "transient" para que sua instância não seja serializada        //Demais valores de estado serão serializados      private int valorParaPersistir;        public ImplementacaoTransient(int valorParaPersistir){          this.valorParaPersistir = valorParaPersistir;          thread = new Thread(this);          thread.start();      }        public void run()      {          while(true){              //código a ser executado pela Thread          }      }  } |

Vale ressaltar que, o que é serializado é apenas o estado do objeto, e não os métodos e a classe. No exemplo acima, apenas a instância do objeto “thread” e do objeto “valorParaPersistir” compunham o estado do objeto. Assim, evita-se que o sistema de serialização tente serializar um objeto não-serializável e gere erros. Após a deserialização de um objeto da classe “ImplementacaoTransient” este estará totalmente funcional.

Um outro problema que pode vir a ocorrer, se refere ao versionamento das classes. Toda classe possui um atributo oculto chamado “serialVersionUID” com um valor gerado automaticamente. Objetos criados levam consigo o “serialVersionUID” de sua classe, que indica sua versão. Toda vez que uma classe sofre alterações em seu código, um novo “serialVersionUID” é gerado. O problema com a serialização surge quando serializa-se o estado de um objeto e posteriormente tenta-se restaurar o estado para a aplicação, mas houveram mudanças no código da classe. Neste caso, uma exceção “java.io.InvalidClassException” é disparada, pois de fato, o estado do objeto corresponde a uma versão de classe do qual foi criado que já não existe mais. Portanto, isso só ocorre para evitar erros de programação, em que tenta-se restaurar o estado de um objeto para uma classe que possui versão diferente da qual o objeto foi serializado inicialmente. E isso pode significar diferentes versões de uma mesma classe “Fruta” por exemplo, ou diferentes versões por classes serem totalmente difere

|  |  |
| --- | --- |
| http://www.linhadecodigo.com.br/artigos/img_artigos/EvertonCoimbraAraujo/Serializazao_Objetos/imagem001.jpg | Neste artigo apresento o Gustavo Rafael Valiati, que pode ser contatado pelo e-mail gustavovaliati@gmail.com. O Gustavo foi meu aluno em algumas disciplinas de sua gradução e está concluíndo o último período do curso de Tecnologia em Análise e Desenvolvimento de Sistemas da UTFPR campus Medianeira, neste primeiro semestre de 2012. |

**1. INTRODUÇÃO**

Nem sempre se deseja ter objetos ou estrutura de dados de um aplicativo, com um ciclo de vida que dura no máximo o tempo em que for armazenado em memória RAM. Ou seja, algumas vezes deseja-se guardá-los por um tempo indeterminado, sendo possível recuperá-los posteriormente. E neste caso, o termo “persistência” tem seu conceito definido.

Persistir dados, requer que estes possam ser serializáveis. A serialização possibilitará capturar o estado do objeto ou a estrutura de dados, e transformar em uma cadeia de bytes quando for necessário. Torna-se possível também, recuperar os bytes persistidos para fazer o processo inverso, e ter os dados de volta para a aplicação em execução.

Desta maneira, qualquer forma de dados que esteja sendo trabalhada na aplicação, e esta puder ser serializada, poderá ser passada para o meio físico, e ter o processo revertido.

A serialização não se limita apenas em possibilitar a gravação do dado em disco, mas permite que o mesmo seja transmitido por rede. Isto porquê a serialização cria um stream de bytes, que é requisito para transmissão de dados por rede.

Diversas linguagens possuem suporte à serialização, como: Java, C, C++, Python, PHP, .NET, entre outras. Neste artigo, será tratado apenas da serialização com Java, em sua implemetação básica, alguns problemas e soluções, com códigos para exemplificação.

**2. SERIALIZAÇÃO COM JAVA**

Serializar um objeto, em Java, só é possível caso sua classe esteja marcada como serializável. Essa marcação é feita com a simples implementação da interface “java.io.Serializable”. Essa interface não possui métodos para se implementar e nem atributos. Ainda não existiam as anotações no Java 1.1, versão que trouxe pela primeira vez a serialização ao Java, e desta maneira a implementação da interface era a melhor maneira de marcar uma classe.

A herança entre classes é naturalmente afetada pela serialização. Já que quando uma classe implementa a interface “Serializable”, toda classe que dela estender também estará implicitamente marcada como serializável

.

Para exemplo, será criada uma classe “pojo.Fruta” que contém dois atributos: nome e cor. No código exemplo (Listagem 1), pode-se notar a classe “Fruta” marcada para a serialização:

Listagem 1 - Marcando uma classe como serializável.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | import java.io.Serializable;    public class Fruta implements Serializable{      private String nome;      private String cor;        public Fruta(String nome, String cor) {          this.nome = nome;          this.cor = cor;      }        public String getCor() { return cor; }      public String getNome() { return nome; }      public void setCor(String cor) { this.cor = cor; }      public void setNome(String nome) { this.nome = nome; }  } |

Pode-se dizer que a implementação do processo de serialização e deserialização simples de um objeto, não possui um código difícil de se escrever. Isto porquê duas classes fazem a maior parte do trabalho, de maneira que com poucas linhas de código pode-se transformar um objeto em bytes e vice-versa. Essas duas classes são “java.io.ObjectInputStream” que recebe bytes para criar um objeto e “java.io.ObjectOutputStream” que cria uma cadeia de bytes a partir de um objeto.

**3. SERIALIZANDO E DESERIALIZANDO O ESTADO DE UM OBJETO**

Como exemplo, será persistido um objeto em disco. Para isto, além das classes “java.io.ObjectInputStream” e “java.io.ObjectOutputStream”, serão utilizadas duas outras que trabalham com arquivos, que são “java.io.FileInputStream” e “java.io.FileOutputStream”, que seguem a mesma lógica de InputStream e OutputStream, só que agora relacionado ao sistema de arquivos e não aos objetos.

Para serialização, deve-se criar um novo “java.io.FileOutputStream”, com o caminho (“path”) do arquivo onde se deseja armazenar o estado do objeto. Também deve ser criado um novo “java.io.ObjectOutputStream” que está ligado ao arquivo. E por último deve-se executar a escrita do objeto alvo, para dentro do objeto de stream que por sua vez está ligado ao arquivo destino. Após executados esses passos, é possível encontrar no sistema de arquivos um novo arquivo que contém os bytes correspondentes à descrição do estado do objeto. Na Listagem 2, encontra-se o código exemplo.

Listagem 2 - Serialização de um objeto.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | import java.io.FileOutputStream;  import java.io.ObjectOutputStream;    public class Serializador {        public Serializador() {    }        public void serializar(String path, Object obj) throws Exception {              FileOutputStream outFile = new FileOutputStream(path);      ObjectOutputStream s = new ObjectOutputStream(outFile);      s.writeObject(obj);      s.close();      }  } |

E para a deserialização, o processo é extremamente parecido com a serialização. Cria-se um novo “java.io.FileInputStream” relacionado ao arquivo onde o estado do objeto encontra-se persistido. Deve-se criar também um novo “java.io.ObjectInputStream” que estará vinculado ao arquivo de origem. Por último, executa-se a leitura do arquivo para a restauração do estado. Na Listagem 3 encontra-se o exemplo do código.

Listagem 3 - Deserializaçãode um objeto.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | import java.io.FileInputStream;  import java.io.ObjectInputStream;    public class Deserializador {        public Deserializador() {    }        public Object deserializar(String path) throws Exception {      FileInputStream inFile = new FileInputStream(path);      ObjectInputStream d = new ObjectInputStream(inFile);      Object o = d.readObject();      d.close();      return o;      }  } |

A Listagem 4, contém o exemplo da execução da serialização e em seguida da deserialização.

Listagem 4 - Serializando e Deserializando

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | import operacoes.Deserializador;  import operacoes.Serializador;  import pojo.Fruta;    public class Principal {      public static void main(String args[]){         //serializa         Serializador s = new Serializador();         Fruta fruta = new Fruta("Maça", "Vermelha");          try {              s.serializar("/home/gustavo/fruta", fruta);          } catch (Exception ex) {              System.err.println("Falha ao serializar! - " + ex.toString());          }            //deserializa          Deserializador d = new Deserializador();          fruta = null;          try {              fruta = (Fruta) d.deserializar("/home/gustavo/fruta");          } catch (Exception ex) {              System.err.println("Falha ao deserializar! - " + ex.toString());          }          System.out.println(fruta.getNome() + " - " + fruta.getCor());      }  } |

Na Listagem 5, encontra-se um exemplo de serialização e deserialização de um ArrayList da classe Fruta. É apenas mais um exemplo que demonstra a capacidade de se persistir qualquer objeto que seja serializável. Essa serialização só foi possível porque a classe “ArrayList” implementa a interface “Serializable” por padrão.

Listagem 5 - Serialização de ArrayList.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | //serialização e deserialização com ArrayList          ArrayList<Fruta> frutas = new ArrayList<Fruta>();          Fruta f1 = new Fruta("Laranja", "Amarela");          Fruta f2 = new Fruta("Abacate", "Verde");          Fruta f3 = new Fruta("Morango", "Vermelho");          frutas.add(f1);          frutas.add(f2);          frutas.add(f3);            try {              s.serializar("/home/gustavo/frutas", frutas);              frutas = null;              frutas = (ArrayList<Fruta>) d.deserializar("/home/gustavo/frutas");              for(Fruta f : frutas){                  System.out.println("ArrayList: " + f.getNome() + " - " + f.getCor());              }          } catch (Exception ex) {              System.err.println("Falha ao serializar ou deserializar! - " +                                 ex.toString());          } |

**4. PARTICULARIDADES DA SERIALIZAÇÃO**

Como dito, a serialização de um objeto em Java pode ser algo fácil de se fazer. Mas existem certas informações que são importantes e sabê-las pode evitar vários problemas.

Um dos problemas é que nem tudo pode ser serializado, como objetos das classes “Thread” e “Socket” e atributos estáticos. Os atributos estáticos estão vinculados à classe e não ao objeto. Assim quando se trata do estado do objeto, os campos estáticos não são considerados. Em relação às Threads, suas instâncias só são úteis enquanto estiverem em execução na JVM, e podem ser paradas/destruídas e iniciadas com instruções de sua classe. Não existe algo no estado destas instâncias que seja útil para se persistir, e por isso não há sentido em persistir tais instâncias para restaurar posteriormente.

Mas e se existir alguma classe em que seja necessário persisti-la, e esta possui uma instância da classe “Thread”? Neste caso, utilizando um recurso pode-se persistir a classe inteira, menos a instância thread. Este recurso, refere-se em indicar que tal instância thread é um objeto “Transient”. Na Listagem 6 encontra-se um exemplo da implementação com uso de “Transient”.

Listagem 6 - Uso do Transient

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | import java.io.Serializable;    public class ImplementacaoTransient implements Serializable, Runnable{        transient private Thread thread;      //marcado com "transient" para que sua instância não seja serializada        //Demais valores de estado serão serializados      private int valorParaPersistir;        public ImplementacaoTransient(int valorParaPersistir){          this.valorParaPersistir = valorParaPersistir;          thread = new Thread(this);          thread.start();      }        public void run()      {          while(true){              //código a ser executado pela Thread          }      }  } |

Vale ressaltar que, o que é serializado é apenas o estado do objeto, e não os métodos e a classe. No exemplo acima, apenas a instância do objeto “thread” e do objeto “valorParaPersistir” compunham o estado do objeto. Assim, evita-se que o sistema de serialização tente serializar um objeto não-serializável e gere erros. Após a deserialização de um objeto da classe “ImplementacaoTransient” este estará totalmente funcional.

Um outro problema que pode vir a ocorrer, se refere ao versionamento das classes. Toda classe possui um atributo oculto chamado “serialVersionUID” com um valor gerado automaticamente. Objetos criados levam consigo o “serialVersionUID” de sua classe, que indica sua versão. Toda vez que uma classe sofre alterações em seu código, um novo “serialVersionUID” é gerado. O problema com a serialização surge quando serializa-se o estado de um objeto e posteriormente tenta-se restaurar o estado para a aplicação, mas houveram mudanças no código da classe. Neste caso, uma exceção “java.io.InvalidClassException” é disparada, pois de fato, o estado do objeto corresponde a uma versão de classe do qual foi criado que já não existe mais. Portanto, isso só ocorre para evitar erros de programação, em que tenta-se restaurar o estado de um objeto para uma classe que possui versão diferente da qual o objeto foi serializado inicialmente. E isso pode significar diferentes versões de uma mesma classe “Fruta” por exemplo, ou diferentes versões por classes serem totalmente difere

|  |  |
| --- | --- |
| http://www.linhadecodigo.com.br/artigos/img_artigos/EvertonCoimbraAraujo/Serializazao_Objetos/imagem001.jpg | Neste artigo apresento o Gustavo Rafael Valiati, que pode ser contatado pelo e-mail gustavovaliati@gmail.com. O Gustavo foi meu aluno em algumas disciplinas de sua gradução e está concluíndo o último período do curso de Tecnologia em Análise e Desenvolvimento de Sistemas da UTFPR campus Medianeira, neste primeiro semestre de 2012. |

**1. INTRODUÇÃO**

Nem sempre se deseja ter objetos ou estrutura de dados de um aplicativo, com um ciclo de vida que dura no máximo o tempo em que for armazenado em memória RAM. Ou seja, algumas vezes deseja-se guardá-los por um tempo indeterminado, sendo possível recuperá-los posteriormente. E neste caso, o termo “persistência” tem seu conceito definido.

Persistir dados, requer que estes possam ser serializáveis. A serialização possibilitará capturar o estado do objeto ou a estrutura de dados, e transformar em uma cadeia de bytes quando for necessário. Torna-se possível também, recuperar os bytes persistidos para fazer o processo inverso, e ter os dados de volta para a aplicação em execução.

Desta maneira, qualquer forma de dados que esteja sendo trabalhada na aplicação, e esta puder ser serializada, poderá ser passada para o meio físico, e ter o processo revertido.

A serialização não se limita apenas em possibilitar a gravação do dado em disco, mas permite que o mesmo seja transmitido por rede. Isto porquê a serialização cria um stream de bytes, que é requisito para transmissão de dados por rede.

Diversas linguagens possuem suporte à serialização, como: Java, C, C++, Python, PHP, .NET, entre outras. Neste artigo, será tratado apenas da serialização com Java, em sua implemetação básica, alguns problemas e soluções, com códigos para exemplificação.

**2. SERIALIZAÇÃO COM JAVA**

Serializar um objeto, em Java, só é possível caso sua classe esteja marcada como serializável. Essa marcação é feita com a simples implementação da interface “java.io.Serializable”. Essa interface não possui métodos para se implementar e nem atributos. Ainda não existiam as anotações no Java 1.1, versão que trouxe pela primeira vez a serialização ao Java, e desta maneira a implementação da interface era a melhor maneira de marcar uma classe.

A herança entre classes é naturalmente afetada pela serialização. Já que quando uma classe implementa a interface “Serializable”, toda classe que dela estender também estará implicitamente marcada como serializável

.

Para exemplo, será criada uma classe “pojo.Fruta” que contém dois atributos: nome e cor. No código exemplo (Listagem 1), pode-se notar a classe “Fruta” marcada para a serialização:

Listagem 1 - Marcando uma classe como serializável.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | import java.io.Serializable;    public class Fruta implements Serializable{      private String nome;      private String cor;        public Fruta(String nome, String cor) {          this.nome = nome;          this.cor = cor;      }        public String getCor() { return cor; }      public String getNome() { return nome; }      public void setCor(String cor) { this.cor = cor; }      public void setNome(String nome) { this.nome = nome; }  } |

Pode-se dizer que a implementação do processo de serialização e deserialização simples de um objeto, não possui um código difícil de se escrever. Isto porquê duas classes fazem a maior parte do trabalho, de maneira que com poucas linhas de código pode-se transformar um objeto em bytes e vice-versa. Essas duas classes são “java.io.ObjectInputStream” que recebe bytes para criar um objeto e “java.io.ObjectOutputStream” que cria uma cadeia de bytes a partir de um objeto.

**3. SERIALIZANDO E DESERIALIZANDO O ESTADO DE UM OBJETO**

Como exemplo, será persistido um objeto em disco. Para isto, além das classes “java.io.ObjectInputStream” e “java.io.ObjectOutputStream”, serão utilizadas duas outras que trabalham com arquivos, que são “java.io.FileInputStream” e “java.io.FileOutputStream”, que seguem a mesma lógica de InputStream e OutputStream, só que agora relacionado ao sistema de arquivos e não aos objetos.

Para serialização, deve-se criar um novo “java.io.FileOutputStream”, com o caminho (“path”) do arquivo onde se deseja armazenar o estado do objeto. Também deve ser criado um novo “java.io.ObjectOutputStream” que está ligado ao arquivo. E por último deve-se executar a escrita do objeto alvo, para dentro do objeto de stream que por sua vez está ligado ao arquivo destino. Após executados esses passos, é possível encontrar no sistema de arquivos um novo arquivo que contém os bytes correspondentes à descrição do estado do objeto. Na Listagem 2, encontra-se o código exemplo.

Listagem 2 - Serialização de um objeto.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | import java.io.FileOutputStream;  import java.io.ObjectOutputStream;    public class Serializador {        public Serializador() {    }        public void serializar(String path, Object obj) throws Exception {              FileOutputStream outFile = new FileOutputStream(path);      ObjectOutputStream s = new ObjectOutputStream(outFile);      s.writeObject(obj);      s.close();      }  } |

E para a deserialização, o processo é extremamente parecido com a serialização. Cria-se um novo “java.io.FileInputStream” relacionado ao arquivo onde o estado do objeto encontra-se persistido. Deve-se criar também um novo “java.io.ObjectInputStream” que estará vinculado ao arquivo de origem. Por último, executa-se a leitura do arquivo para a restauração do estado. Na Listagem 3 encontra-se o exemplo do código.

Listagem 3 - Deserializaçãode um objeto.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | import java.io.FileInputStream;  import java.io.ObjectInputStream;    public class Deserializador {        public Deserializador() {    }        public Object deserializar(String path) throws Exception {      FileInputStream inFile = new FileInputStream(path);      ObjectInputStream d = new ObjectInputStream(inFile);      Object o = d.readObject();      d.close();      return o;      }  } |

A Listagem 4, contém o exemplo da execução da serialização e em seguida da deserialização.

Listagem 4 - Serializando e Deserializando

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | import operacoes.Deserializador;  import operacoes.Serializador;  import pojo.Fruta;    public class Principal {      public static void main(String args[]){         //serializa         Serializador s = new Serializador();         Fruta fruta = new Fruta("Maça", "Vermelha");          try {              s.serializar("/home/gustavo/fruta", fruta);          } catch (Exception ex) {              System.err.println("Falha ao serializar! - " + ex.toString());          }            //deserializa          Deserializador d = new Deserializador();          fruta = null;          try {              fruta = (Fruta) d.deserializar("/home/gustavo/fruta");          } catch (Exception ex) {              System.err.println("Falha ao deserializar! - " + ex.toString());          }          System.out.println(fruta.getNome() + " - " + fruta.getCor());      }  } |

Na Listagem 5, encontra-se um exemplo de serialização e deserialização de um ArrayList da classe Fruta. É apenas mais um exemplo que demonstra a capacidade de se persistir qualquer objeto que seja serializável. Essa serialização só foi possível porque a classe “ArrayList” implementa a interface “Serializable” por padrão.

Listagem 5 - Serialização de ArrayList.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | //serialização e deserialização com ArrayList          ArrayList<Fruta> frutas = new ArrayList<Fruta>();          Fruta f1 = new Fruta("Laranja", "Amarela");          Fruta f2 = new Fruta("Abacate", "Verde");          Fruta f3 = new Fruta("Morango", "Vermelho");          frutas.add(f1);          frutas.add(f2);          frutas.add(f3);            try {              s.serializar("/home/gustavo/frutas", frutas);              frutas = null;              frutas = (ArrayList<Fruta>) d.deserializar("/home/gustavo/frutas");              for(Fruta f : frutas){                  System.out.println("ArrayList: " + f.getNome() + " - " + f.getCor());              }          } catch (Exception ex) {              System.err.println("Falha ao serializar ou deserializar! - " +                                 ex.toString());          } |

**4. PARTICULARIDADES DA SERIALIZAÇÃO**

Como dito, a serialização de um objeto em Java pode ser algo fácil de se fazer. Mas existem certas informações que são importantes e sabê-las pode evitar vários problemas.

Um dos problemas é que nem tudo pode ser serializado, como objetos das classes “Thread” e “Socket” e atributos estáticos. Os atributos estáticos estão vinculados à classe e não ao objeto. Assim quando se trata do estado do objeto, os campos estáticos não são considerados. Em relação às Threads, suas instâncias só são úteis enquanto estiverem em execução na JVM, e podem ser paradas/destruídas e iniciadas com instruções de sua classe. Não existe algo no estado destas instâncias que seja útil para se persistir, e por isso não há sentido em persistir tais instâncias para restaurar posteriormente.

Mas e se existir alguma classe em que seja necessário persisti-la, e esta possui uma instância da classe “Thread”? Neste caso, utilizando um recurso pode-se persistir a classe inteira, menos a instância thread. Este recurso, refere-se em indicar que tal instância thread é um objeto “Transient”. Na Listagem 6 encontra-se um exemplo da implementação com uso de “Transient”.

Listagem 6 - Uso do Transient

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | import java.io.Serializable;    public class ImplementacaoTransient implements Serializable, Runnable{        transient private Thread thread;      //marcado com "transient" para que sua instância não seja serializada        //Demais valores de estado serão serializados      private int valorParaPersistir;        public ImplementacaoTransient(int valorParaPersistir){          this.valorParaPersistir = valorParaPersistir;          thread = new Thread(this);          thread.start();      }        public void run()      {          while(true){              //código a ser executado pela Thread          }      }  } |

Vale ressaltar que, o que é serializado é apenas o estado do objeto, e não os métodos e a classe. No exemplo acima, apenas a instância do objeto “thread” e do objeto “valorParaPersistir” compunham o estado do objeto. Assim, evita-se que o sistema de serialização tente serializar um objeto não-serializável e gere erros. Após a deserialização de um objeto da classe “ImplementacaoTransient” este estará totalmente funcional.

Um outro problema que pode vir a ocorrer, se refere ao versionamento das classes. Toda classe possui um atributo oculto chamado “serialVersionUID” com um valor gerado automaticamente. Objetos criados levam consigo o “serialVersionUID” de sua classe, que indica sua versão. Toda vez que uma classe sofre alterações em seu código, um novo “serialVersionUID” é gerado. O problema com a serialização surge quando serializa-se o estado de um objeto e posteriormente tenta-se restaurar o estado para a aplicação, mas houveram mudanças no código da classe. Neste caso, uma exceção “java.io.InvalidClassException” é disparada, pois de fato, o estado do objeto corresponde a uma versão de classe do qual foi criado que já não existe mais. Portanto, isso só ocorre para evitar erros de programação, em que tenta-se restaurar o estado de um objeto para uma classe que possui versão diferente da qual o objeto foi serializado inicialmente. E isso pode significar diferentes versões de uma mesma classe “Fruta” por exemplo, ou diferentes versões por classes serem totalmente difere

|  |  |
| --- | --- |
| http://www.linhadecodigo.com.br/artigos/img_artigos/EvertonCoimbraAraujo/Serializazao_Objetos/imagem001.jpg | Neste artigo apresento o Gustavo Rafael Valiati, que pode ser contatado pelo e-mail gustavovaliati@gmail.com. O Gustavo foi meu aluno em algumas disciplinas de sua gradução e está concluíndo o último período do curso de Tecnologia em Análise e Desenvolvimento de Sistemas da UTFPR campus Medianeira, neste primeiro semestre de 2012. |

**1. INTRODUÇÃO**

Nem sempre se deseja ter objetos ou estrutura de dados de um aplicativo, com um ciclo de vida que dura no máximo o tempo em que for armazenado em memória RAM. Ou seja, algumas vezes deseja-se guardá-los por um tempo indeterminado, sendo possível recuperá-los posteriormente. E neste caso, o termo “persistência” tem seu conceito definido.

Persistir dados, requer que estes possam ser serializáveis. A serialização possibilitará capturar o estado do objeto ou a estrutura de dados, e transformar em uma cadeia de bytes quando for necessário. Torna-se possível também, recuperar os bytes persistidos para fazer o processo inverso, e ter os dados de volta para a aplicação em execução.

Desta maneira, qualquer forma de dados que esteja sendo trabalhada na aplicação, e esta puder ser serializada, poderá ser passada para o meio físico, e ter o processo revertido.

A serialização não se limita apenas em possibilitar a gravação do dado em disco, mas permite que o mesmo seja transmitido por rede. Isto porquê a serialização cria um stream de bytes, que é requisito para transmissão de dados por rede.

Diversas linguagens possuem suporte à serialização, como: Java, C, C++, Python, PHP, .NET, entre outras. Neste artigo, será tratado apenas da serialização com Java, em sua implemetação básica, alguns problemas e soluções, com códigos para exemplificação.

**2. SERIALIZAÇÃO COM JAVA**

Serializar um objeto, em Java, só é possível caso sua classe esteja marcada como serializável. Essa marcação é feita com a simples implementação da interface “java.io.Serializable”. Essa interface não possui métodos para se implementar e nem atributos. Ainda não existiam as anotações no Java 1.1, versão que trouxe pela primeira vez a serialização ao Java, e desta maneira a implementação da interface era a melhor maneira de marcar uma classe.

A herança entre classes é naturalmente afetada pela serialização. Já que quando uma classe implementa a interface “Serializable”, toda classe que dela estender também estará implicitamente marcada como serializável

.

Para exemplo, será criada uma classe “pojo.Fruta” que contém dois atributos: nome e cor. No código exemplo (Listagem 1), pode-se notar a classe “Fruta” marcada para a serialização:

Listagem 1 - Marcando uma classe como serializável.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | import java.io.Serializable;    public class Fruta implements Serializable{      private String nome;      private String cor;        public Fruta(String nome, String cor) {          this.nome = nome;          this.cor = cor;      }        public String getCor() { return cor; }      public String getNome() { return nome; }      public void setCor(String cor) { this.cor = cor; }      public void setNome(String nome) { this.nome = nome; }  } |

Pode-se dizer que a implementação do processo de serialização e deserialização simples de um objeto, não possui um código difícil de se escrever. Isto porquê duas classes fazem a maior parte do trabalho, de maneira que com poucas linhas de código pode-se transformar um objeto em bytes e vice-versa. Essas duas classes são “java.io.ObjectInputStream” que recebe bytes para criar um objeto e “java.io.ObjectOutputStream” que cria uma cadeia de bytes a partir de um objeto.

**3. SERIALIZANDO E DESERIALIZANDO O ESTADO DE UM OBJETO**

Como exemplo, será persistido um objeto em disco. Para isto, além das classes “java.io.ObjectInputStream” e “java.io.ObjectOutputStream”, serão utilizadas duas outras que trabalham com arquivos, que são “java.io.FileInputStream” e “java.io.FileOutputStream”, que seguem a mesma lógica de InputStream e OutputStream, só que agora relacionado ao sistema de arquivos e não aos objetos.

Para serialização, deve-se criar um novo “java.io.FileOutputStream”, com o caminho (“path”) do arquivo onde se deseja armazenar o estado do objeto. Também deve ser criado um novo “java.io.ObjectOutputStream” que está ligado ao arquivo. E por último deve-se executar a escrita do objeto alvo, para dentro do objeto de stream que por sua vez está ligado ao arquivo destino. Após executados esses passos, é possível encontrar no sistema de arquivos um novo arquivo que contém os bytes correspondentes à descrição do estado do objeto. Na Listagem 2, encontra-se o código exemplo.

Listagem 2 - Serialização de um objeto.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | import java.io.FileOutputStream;  import java.io.ObjectOutputStream;    public class Serializador {        public Serializador() {    }        public void serializar(String path, Object obj) throws Exception {              FileOutputStream outFile = new FileOutputStream(path);      ObjectOutputStream s = new ObjectOutputStream(outFile);      s.writeObject(obj);      s.close();      }  } |

E para a deserialização, o processo é extremamente parecido com a serialização. Cria-se um novo “java.io.FileInputStream” relacionado ao arquivo onde o estado do objeto encontra-se persistido. Deve-se criar também um novo “java.io.ObjectInputStream” que estará vinculado ao arquivo de origem. Por último, executa-se a leitura do arquivo para a restauração do estado. Na Listagem 3 encontra-se o exemplo do código.

Listagem 3 - Deserializaçãode um objeto.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | import java.io.FileInputStream;  import java.io.ObjectInputStream;    public class Deserializador {        public Deserializador() {    }        public Object deserializar(String path) throws Exception {      FileInputStream inFile = new FileInputStream(path);      ObjectInputStream d = new ObjectInputStream(inFile);      Object o = d.readObject();      d.close();      return o;      }  } |

A Listagem 4, contém o exemplo da execução da serialização e em seguida da deserialização.

Listagem 4 - Serializando e Deserializando

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | import operacoes.Deserializador;  import operacoes.Serializador;  import pojo.Fruta;    public class Principal {      public static void main(String args[]){         //serializa         Serializador s = new Serializador();         Fruta fruta = new Fruta("Maça", "Vermelha");          try {              s.serializar("/home/gustavo/fruta", fruta);          } catch (Exception ex) {              System.err.println("Falha ao serializar! - " + ex.toString());          }            //deserializa          Deserializador d = new Deserializador();          fruta = null;          try {              fruta = (Fruta) d.deserializar("/home/gustavo/fruta");          } catch (Exception ex) {              System.err.println("Falha ao deserializar! - " + ex.toString());          }          System.out.println(fruta.getNome() + " - " + fruta.getCor());      }  } |

Na Listagem 5, encontra-se um exemplo de serialização e deserialização de um ArrayList da classe Fruta. É apenas mais um exemplo que demonstra a capacidade de se persistir qualquer objeto que seja serializável. Essa serialização só foi possível porque a classe “ArrayList” implementa a interface “Serializable” por padrão.

Listagem 5 - Serialização de ArrayList.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | //serialização e deserialização com ArrayList          ArrayList<Fruta> frutas = new ArrayList<Fruta>();          Fruta f1 = new Fruta("Laranja", "Amarela");          Fruta f2 = new Fruta("Abacate", "Verde");          Fruta f3 = new Fruta("Morango", "Vermelho");          frutas.add(f1);          frutas.add(f2);          frutas.add(f3);            try {              s.serializar("/home/gustavo/frutas", frutas);              frutas = null;              frutas = (ArrayList<Fruta>) d.deserializar("/home/gustavo/frutas");              for(Fruta f : frutas){                  System.out.println("ArrayList: " + f.getNome() + " - " + f.getCor());              }          } catch (Exception ex) {              System.err.println("Falha ao serializar ou deserializar! - " +                                 ex.toString());          } |

**4. PARTICULARIDADES DA SERIALIZAÇÃO**

Como dito, a serialização de um objeto em Java pode ser algo fácil de se fazer. Mas existem certas informações que são importantes e sabê-las pode evitar vários problemas.

Um dos problemas é que nem tudo pode ser serializado, como objetos das classes “Thread” e “Socket” e atributos estáticos. Os atributos estáticos estão vinculados à classe e não ao objeto. Assim quando se trata do estado do objeto, os campos estáticos não são considerados. Em relação às Threads, suas instâncias só são úteis enquanto estiverem em execução na JVM, e podem ser paradas/destruídas e iniciadas com instruções de sua classe. Não existe algo no estado destas instâncias que seja útil para se persistir, e por isso não há sentido em persistir tais instâncias para restaurar posteriormente.

Mas e se existir alguma classe em que seja necessário persisti-la, e esta possui uma instância da classe “Thread”? Neste caso, utilizando um recurso pode-se persistir a classe inteira, menos a instância thread. Este recurso, refere-se em indicar que tal instância thread é um objeto “Transient”. Na Listagem 6 encontra-se um exemplo da implementação com uso de “Transient”.

Listagem 6 - Uso do Transient

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | import java.io.Serializable;    public class ImplementacaoTransient implements Serializable, Runnable{        transient private Thread thread;      //marcado com "transient" para que sua instância não seja serializada        //Demais valores de estado serão serializados      private int valorParaPersistir;        public ImplementacaoTransient(int valorParaPersistir){          this.valorParaPersistir = valorParaPersistir;          thread = new Thread(this);          thread.start();      }        public void run()      {          while(true){              //código a ser executado pela Thread          }      }  } |

Vale ressaltar que, o que é serializado é apenas o estado do objeto, e não os métodos e a classe. No exemplo acima, apenas a instância do objeto “thread” e do objeto “valorParaPersistir” compunham o estado do objeto. Assim, evita-se que o sistema de serialização tente serializar um objeto não-serializável e gere erros. Após a deserialização de um objeto da classe “ImplementacaoTransient” este estará totalmente funcional.

Um outro problema que pode vir a ocorrer, se refere ao versionamento das classes. Toda classe possui um atributo oculto chamado “serialVersionUID” com um valor gerado automaticamente. Objetos criados levam consigo o “serialVersionUID” de sua classe, que indica sua versão. Toda vez que uma classe sofre alterações em seu código, um novo “serialVersionUID” é gerado. O problema com a serialização surge quando serializa-se o estado de um objeto e posteriormente tenta-se restaurar o estado para a aplicação, mas houveram mudanças no código da classe. Neste caso, uma exceção “java.io.InvalidClassException” é disparada, pois de fato, o estado do objeto corresponde a uma versão de classe do qual foi criado que já não existe mais. Portanto, isso só ocorre para evitar erros de programação, em que tenta-se restaurar o estado de um objeto para uma classe que possui versão diferente da qual o objeto foi serializado inicialmente. E isso pode significar diferentes versões de uma mesma classe “Fruta” por exemplo, ou diferentes versões por classes serem totalmente difere

|  |  |
| --- | --- |
| http://www.linhadecodigo.com.br/artigos/img_artigos/EvertonCoimbraAraujo/Serializazao_Objetos/imagem001.jpg | Neste artigo apresento o Gustavo Rafael Valiati, que pode ser contatado pelo e-mail gustavovaliati@gmail.com. O Gustavo foi meu aluno em algumas disciplinas de sua gradução e está concluíndo o último período do curso de Tecnologia em Análise e Desenvolvimento de Sistemas da UTFPR campus Medianeira, neste primeiro semestre de 2012. |

**1. INTRODUÇÃO**

Nem sempre se deseja ter objetos ou estrutura de dados de um aplicativo, com um ciclo de vida que dura no máximo o tempo em que for armazenado em memória RAM. Ou seja, algumas vezes deseja-se guardá-los por um tempo indeterminado, sendo possível recuperá-los posteriormente. E neste caso, o termo “persistência” tem seu conceito definido.

Persistir dados, requer que estes possam ser serializáveis. A serialização possibilitará capturar o estado do objeto ou a estrutura de dados, e transformar em uma cadeia de bytes quando for necessário. Torna-se possível também, recuperar os bytes persistidos para fazer o processo inverso, e ter os dados de volta para a aplicação em execução.

Desta maneira, qualquer forma de dados que esteja sendo trabalhada na aplicação, e esta puder ser serializada, poderá ser passada para o meio físico, e ter o processo revertido.

A serialização não se limita apenas em possibilitar a gravação do dado em disco, mas permite que o mesmo seja transmitido por rede. Isto porquê a serialização cria um stream de bytes, que é requisito para transmissão de dados por rede.

Diversas linguagens possuem suporte à serialização, como: Java, C, C++, Python, PHP, .NET, entre outras. Neste artigo, será tratado apenas da serialização com Java, em sua implemetação básica, alguns problemas e soluções, com códigos para exemplificação.

**2. SERIALIZAÇÃO COM JAVA**

Serializar um objeto, em Java, só é possível caso sua classe esteja marcada como serializável. Essa marcação é feita com a simples implementação da interface “java.io.Serializable”. Essa interface não possui métodos para se implementar e nem atributos. Ainda não existiam as anotações no Java 1.1, versão que trouxe pela primeira vez a serialização ao Java, e desta maneira a implementação da interface era a melhor maneira de marcar uma classe.

A herança entre classes é naturalmente afetada pela serialização. Já que quando uma classe implementa a interface “Serializable”, toda classe que dela estender também estará implicitamente marcada como serializável

.

Para exemplo, será criada uma classe “pojo.Fruta” que contém dois atributos: nome e cor. No código exemplo (Listagem 1), pode-se notar a classe “Fruta” marcada para a serialização:

Listagem 1 - Marcando uma classe como serializável.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | import java.io.Serializable;    public class Fruta implements Serializable{      private String nome;      private String cor;        public Fruta(String nome, String cor) {          this.nome = nome;          this.cor = cor;      }        public String getCor() { return cor; }      public String getNome() { return nome; }      public void setCor(String cor) { this.cor = cor; }      public void setNome(String nome) { this.nome = nome; }  } |

Pode-se dizer que a implementação do processo de serialização e deserialização simples de um objeto, não possui um código difícil de se escrever. Isto porquê duas classes fazem a maior parte do trabalho, de maneira que com poucas linhas de código pode-se transformar um objeto em bytes e vice-versa. Essas duas classes são “java.io.ObjectInputStream” que recebe bytes para criar um objeto e “java.io.ObjectOutputStream” que cria uma cadeia de bytes a partir de um objeto.

**3. SERIALIZANDO E DESERIALIZANDO O ESTADO DE UM OBJETO**

Como exemplo, será persistido um objeto em disco. Para isto, além das classes “java.io.ObjectInputStream” e “java.io.ObjectOutputStream”, serão utilizadas duas outras que trabalham com arquivos, que são “java.io.FileInputStream” e “java.io.FileOutputStream”, que seguem a mesma lógica de InputStream e OutputStream, só que agora relacionado ao sistema de arquivos e não aos objetos.

Para serialização, deve-se criar um novo “java.io.FileOutputStream”, com o caminho (“path”) do arquivo onde se deseja armazenar o estado do objeto. Também deve ser criado um novo “java.io.ObjectOutputStream” que está ligado ao arquivo. E por último deve-se executar a escrita do objeto alvo, para dentro do objeto de stream que por sua vez está ligado ao arquivo destino. Após executados esses passos, é possível encontrar no sistema de arquivos um novo arquivo que contém os bytes correspondentes à descrição do estado do objeto. Na Listagem 2, encontra-se o código exemplo.

Listagem 2 - Serialização de um objeto.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | import java.io.FileOutputStream;  import java.io.ObjectOutputStream;    public class Serializador {        public Serializador() {    }        public void serializar(String path, Object obj) throws Exception {              FileOutputStream outFile = new FileOutputStream(path);      ObjectOutputStream s = new ObjectOutputStream(outFile);      s.writeObject(obj);      s.close();      }  } |

E para a deserialização, o processo é extremamente parecido com a serialização. Cria-se um novo “java.io.FileInputStream” relacionado ao arquivo onde o estado do objeto encontra-se persistido. Deve-se criar também um novo “java.io.ObjectInputStream” que estará vinculado ao arquivo de origem. Por último, executa-se a leitura do arquivo para a restauração do estado. Na Listagem 3 encontra-se o exemplo do código.

Listagem 3 - Deserializaçãode um objeto.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | import java.io.FileInputStream;  import java.io.ObjectInputStream;    public class Deserializador {        public Deserializador() {    }        public Object deserializar(String path) throws Exception {      FileInputStream inFile = new FileInputStream(path);      ObjectInputStream d = new ObjectInputStream(inFile);      Object o = d.readObject();      d.close();      return o;      }  } |

A Listagem 4, contém o exemplo da execução da serialização e em seguida da deserialização.

Listagem 4 - Serializando e Deserializando

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | import operacoes.Deserializador;  import operacoes.Serializador;  import pojo.Fruta;    public class Principal {      public static void main(String args[]){         //serializa         Serializador s = new Serializador();         Fruta fruta = new Fruta("Maça", "Vermelha");          try {              s.serializar("/home/gustavo/fruta", fruta);          } catch (Exception ex) {              System.err.println("Falha ao serializar! - " + ex.toString());          }            //deserializa          Deserializador d = new Deserializador();          fruta = null;          try {              fruta = (Fruta) d.deserializar("/home/gustavo/fruta");          } catch (Exception ex) {              System.err.println("Falha ao deserializar! - " + ex.toString());          }          System.out.println(fruta.getNome() + " - " + fruta.getCor());      }  } |

Na Listagem 5, encontra-se um exemplo de serialização e deserialização de um ArrayList da classe Fruta. É apenas mais um exemplo que demonstra a capacidade de se persistir qualquer objeto que seja serializável. Essa serialização só foi possível porque a classe “ArrayList” implementa a interface “Serializable” por padrão.

Listagem 5 - Serialização de ArrayList.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | //serialização e deserialização com ArrayList          ArrayList<Fruta> frutas = new ArrayList<Fruta>();          Fruta f1 = new Fruta("Laranja", "Amarela");          Fruta f2 = new Fruta("Abacate", "Verde");          Fruta f3 = new Fruta("Morango", "Vermelho");          frutas.add(f1);          frutas.add(f2);          frutas.add(f3);            try {              s.serializar("/home/gustavo/frutas", frutas);              frutas = null;              frutas = (ArrayList<Fruta>) d.deserializar("/home/gustavo/frutas");              for(Fruta f : frutas){                  System.out.println("ArrayList: " + f.getNome() + " - " + f.getCor());              }          } catch (Exception ex) {              System.err.println("Falha ao serializar ou deserializar! - " +                                 ex.toString());          } |

**4. PARTICULARIDADES DA SERIALIZAÇÃO**

Como dito, a serialização de um objeto em Java pode ser algo fácil de se fazer. Mas existem certas informações que são importantes e sabê-las pode evitar vários problemas.

Um dos problemas é que nem tudo pode ser serializado, como objetos das classes “Thread” e “Socket” e atributos estáticos. Os atributos estáticos estão vinculados à classe e não ao objeto. Assim quando se trata do estado do objeto, os campos estáticos não são considerados. Em relação às Threads, suas instâncias só são úteis enquanto estiverem em execução na JVM, e podem ser paradas/destruídas e iniciadas com instruções de sua classe. Não existe algo no estado destas instâncias que seja útil para se persistir, e por isso não há sentido em persistir tais instâncias para restaurar posteriormente.

Mas e se existir alguma classe em que seja necessário persisti-la, e esta possui uma instância da classe “Thread”? Neste caso, utilizando um recurso pode-se persistir a classe inteira, menos a instância thread. Este recurso, refere-se em indicar que tal instância thread é um objeto “Transient”. Na Listagem 6 encontra-se um exemplo da implementação com uso de “Transient”.

Listagem 6 - Uso do Transient

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | import java.io.Serializable;    public class ImplementacaoTransient implements Serializable, Runnable{        transient private Thread thread;      //marcado com "transient" para que sua instância não seja serializada        //Demais valores de estado serão serializados      private int valorParaPersistir;        public ImplementacaoTransient(int valorParaPersistir){          this.valorParaPersistir = valorParaPersistir;          thread = new Thread(this);          thread.start();      }        public void run()      {          while(true){              //código a ser executado pela Thread          }      }  } |

Vale ressaltar que, o que é serializado é apenas o estado do objeto, e não os métodos e a classe. No exemplo acima, apenas a instância do objeto “thread” e do objeto “valorParaPersistir” compunham o estado do objeto. Assim, evita-se que o sistema de serialização tente serializar um objeto não-serializável e gere erros. Após a deserialização de um objeto da classe “ImplementacaoTransient” este estará totalmente funcional.

Um outro problema que pode vir a ocorrer, se refere ao versionamento das classes. Toda classe possui um atributo oculto chamado “serialVersionUID” com um valor gerado automaticamente. Objetos criados levam consigo o “serialVersionUID” de sua classe, que indica sua versão. Toda vez que uma classe sofre alterações em seu código, um novo “serialVersionUID” é gerado. O problema com a serialização surge quando serializa-se o estado de um objeto e posteriormente tenta-se restaurar o estado para a aplicação, mas houveram mudanças no código da classe. Neste caso, uma exceção “java.io.InvalidClassException” é disparada, pois de fato, o estado do objeto corresponde a uma versão de classe do qual foi criado que já não existe mais. Portanto, isso só ocorre para evitar erros de programação, em que tenta-se restaurar o estado de um objeto para uma classe que possui versão diferente da qual o objeto foi serializado inicialmente. E isso pode significar diferentes versões de uma mesma classe “Fruta” por exemplo, ou diferentes versões por classes serem totalmente difere

|  |  |
| --- | --- |
| http://www.linhadecodigo.com.br/artigos/img_artigos/EvertonCoimbraAraujo/Serializazao_Objetos/imagem001.jpg | Neste artigo apresento o Gustavo Rafael Valiati, que pode ser contatado pelo e-mail gustavovaliati@gmail.com. O Gustavo foi meu aluno em algumas disciplinas de sua gradução e está concluíndo o último período do curso de Tecnologia em Análise e Desenvolvimento de Sistemas da UTFPR campus Medianeira, neste primeiro semestre de 2012. |

**1. INTRODUÇÃO**

Nem sempre se deseja ter objetos ou estrutura de dados de um aplicativo, com um ciclo de vida que dura no máximo o tempo em que for armazenado em memória RAM. Ou seja, algumas vezes deseja-se guardá-los por um tempo indeterminado, sendo possível recuperá-los posteriormente. E neste caso, o termo “persistência” tem seu conceito definido.

Persistir dados, requer que estes possam ser serializáveis. A serialização possibilitará capturar o estado do objeto ou a estrutura de dados, e transformar em uma cadeia de bytes quando for necessário. Torna-se possível também, recuperar os bytes persistidos para fazer o processo inverso, e ter os dados de volta para a aplicação em execução.

Desta maneira, qualquer forma de dados que esteja sendo trabalhada na aplicação, e esta puder ser serializada, poderá ser passada para o meio físico, e ter o processo revertido.

A serialização não se limita apenas em possibilitar a gravação do dado em disco, mas permite que o mesmo seja transmitido por rede. Isto porquê a serialização cria um stream de bytes, que é requisito para transmissão de dados por rede.

Diversas linguagens possuem suporte à serialização, como: Java, C, C++, Python, PHP, .NET, entre outras. Neste artigo, será tratado apenas da serialização com Java, em sua implemetação básica, alguns problemas e soluções, com códigos para exemplificação.

**2. SERIALIZAÇÃO COM JAVA**

Serializar um objeto, em Java, só é possível caso sua classe esteja marcada como serializável. Essa marcação é feita com a simples implementação da interface “java.io.Serializable”. Essa interface não possui métodos para se implementar e nem atributos. Ainda não existiam as anotações no Java 1.1, versão que trouxe pela primeira vez a serialização ao Java, e desta maneira a implementação da interface era a melhor maneira de marcar uma classe.

A herança entre classes é naturalmente afetada pela serialização. Já que quando uma classe implementa a interface “Serializable”, toda classe que dela estender também estará implicitamente marcada como serializável

.

Para exemplo, será criada uma classe “pojo.Fruta” que contém dois atributos: nome e cor. No código exemplo (Listagem 1), pode-se notar a classe “Fruta” marcada para a serialização:

Listagem 1 - Marcando uma classe como serializável.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | import java.io.Serializable;    public class Fruta implements Serializable{      private String nome;      private String cor;        public Fruta(String nome, String cor) {          this.nome = nome;          this.cor = cor;      }        public String getCor() { return cor; }      public String getNome() { return nome; }      public void setCor(String cor) { this.cor = cor; }      public void setNome(String nome) { this.nome = nome; }  } |

Pode-se dizer que a implementação do processo de serialização e deserialização simples de um objeto, não possui um código difícil de se escrever. Isto porquê duas classes fazem a maior parte do trabalho, de maneira que com poucas linhas de código pode-se transformar um objeto em bytes e vice-versa. Essas duas classes são “java.io.ObjectInputStream” que recebe bytes para criar um objeto e “java.io.ObjectOutputStream” que cria uma cadeia de bytes a partir de um objeto.

**3. SERIALIZANDO E DESERIALIZANDO O ESTADO DE UM OBJETO**

Como exemplo, será persistido um objeto em disco. Para isto, além das classes “java.io.ObjectInputStream” e “java.io.ObjectOutputStream”, serão utilizadas duas outras que trabalham com arquivos, que são “java.io.FileInputStream” e “java.io.FileOutputStream”, que seguem a mesma lógica de InputStream e OutputStream, só que agora relacionado ao sistema de arquivos e não aos objetos.

Para serialização, deve-se criar um novo “java.io.FileOutputStream”, com o caminho (“path”) do arquivo onde se deseja armazenar o estado do objeto. Também deve ser criado um novo “java.io.ObjectOutputStream” que está ligado ao arquivo. E por último deve-se executar a escrita do objeto alvo, para dentro do objeto de stream que por sua vez está ligado ao arquivo destino. Após executados esses passos, é possível encontrar no sistema de arquivos um novo arquivo que contém os bytes correspondentes à descrição do estado do objeto. Na Listagem 2, encontra-se o código exemplo.

Listagem 2 - Serialização de um objeto.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | import java.io.FileOutputStream;  import java.io.ObjectOutputStream;    public class Serializador {        public Serializador() {    }        public void serializar(String path, Object obj) throws Exception {              FileOutputStream outFile = new FileOutputStream(path);      ObjectOutputStream s = new ObjectOutputStream(outFile);      s.writeObject(obj);      s.close();      }  } |

E para a deserialização, o processo é extremamente parecido com a serialização. Cria-se um novo “java.io.FileInputStream” relacionado ao arquivo onde o estado do objeto encontra-se persistido. Deve-se criar também um novo “java.io.ObjectInputStream” que estará vinculado ao arquivo de origem. Por último, executa-se a leitura do arquivo para a restauração do estado. Na Listagem 3 encontra-se o exemplo do código.

Listagem 3 - Deserializaçãode um objeto.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | import java.io.FileInputStream;  import java.io.ObjectInputStream;    public class Deserializador {        public Deserializador() {    }        public Object deserializar(String path) throws Exception {      FileInputStream inFile = new FileInputStream(path);      ObjectInputStream d = new ObjectInputStream(inFile);      Object o = d.readObject();      d.close();      return o;      }  } |

A Listagem 4, contém o exemplo da execução da serialização e em seguida da deserialização.

Listagem 4 - Serializando e Deserializando

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | import operacoes.Deserializador;  import operacoes.Serializador;  import pojo.Fruta;    public class Principal {      public static void main(String args[]){         //serializa         Serializador s = new Serializador();         Fruta fruta = new Fruta("Maça", "Vermelha");          try {              s.serializar("/home/gustavo/fruta", fruta);          } catch (Exception ex) {              System.err.println("Falha ao serializar! - " + ex.toString());          }            //deserializa          Deserializador d = new Deserializador();          fruta = null;          try {              fruta = (Fruta) d.deserializar("/home/gustavo/fruta");          } catch (Exception ex) {              System.err.println("Falha ao deserializar! - " + ex.toString());          }          System.out.println(fruta.getNome() + " - " + fruta.getCor());      }  } |

Na Listagem 5, encontra-se um exemplo de serialização e deserialização de um ArrayList da classe Fruta. É apenas mais um exemplo que demonstra a capacidade de se persistir qualquer objeto que seja serializável. Essa serialização só foi possível porque a classe “ArrayList” implementa a interface “Serializable” por padrão.

Listagem 5 - Serialização de ArrayList.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | //serialização e deserialização com ArrayList          ArrayList<Fruta> frutas = new ArrayList<Fruta>();          Fruta f1 = new Fruta("Laranja", "Amarela");          Fruta f2 = new Fruta("Abacate", "Verde");          Fruta f3 = new Fruta("Morango", "Vermelho");          frutas.add(f1);          frutas.add(f2);          frutas.add(f3);            try {              s.serializar("/home/gustavo/frutas", frutas);              frutas = null;              frutas = (ArrayList<Fruta>) d.deserializar("/home/gustavo/frutas");              for(Fruta f : frutas){                  System.out.println("ArrayList: " + f.getNome() + " - " + f.getCor());              }          } catch (Exception ex) {              System.err.println("Falha ao serializar ou deserializar! - " +                                 ex.toString());          } |

**4. PARTICULARIDADES DA SERIALIZAÇÃO**

Como dito, a serialização de um objeto em Java pode ser algo fácil de se fazer. Mas existem certas informações que são importantes e sabê-las pode evitar vários problemas.

Um dos problemas é que nem tudo pode ser serializado, como objetos das classes “Thread” e “Socket” e atributos estáticos. Os atributos estáticos estão vinculados à classe e não ao objeto. Assim quando se trata do estado do objeto, os campos estáticos não são considerados. Em relação às Threads, suas instâncias só são úteis enquanto estiverem em execução na JVM, e podem ser paradas/destruídas e iniciadas com instruções de sua classe. Não existe algo no estado destas instâncias que seja útil para se persistir, e por isso não há sentido em persistir tais instâncias para restaurar posteriormente.

Mas e se existir alguma classe em que seja necessário persisti-la, e esta possui uma instância da classe “Thread”? Neste caso, utilizando um recurso pode-se persistir a classe inteira, menos a instância thread. Este recurso, refere-se em indicar que tal instância thread é um objeto “Transient”. Na Listagem 6 encontra-se um exemplo da implementação com uso de “Transient”.

Listagem 6 - Uso do Transient

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | import java.io.Serializable;    public class ImplementacaoTransient implements Serializable, Runnable{        transient private Thread thread;      //marcado com "transient" para que sua instância não seja serializada        //Demais valores de estado serão serializados      private int valorParaPersistir;        public ImplementacaoTransient(int valorParaPersistir){          this.valorParaPersistir = valorParaPersistir;          thread = new Thread(this);          thread.start();      }        public void run()      {          while(true){              //código a ser executado pela Thread          }      }  } |

Vale ressaltar que, o que é serializado é apenas o estado do objeto, e não os métodos e a classe. No exemplo acima, apenas a instância do objeto “thread” e do objeto “valorParaPersistir” compunham o estado do objeto. Assim, evita-se que o sistema de serialização tente serializar um objeto não-serializável e gere erros. Após a deserialização de um objeto da classe “ImplementacaoTransient” este estará totalmente funcional.

Um outro problema que pode vir a ocorrer, se refere ao versionamento das classes. Toda classe possui um atributo oculto chamado “serialVersionUID” com um valor gerado automaticamente. Objetos criados levam consigo o “serialVersionUID” de sua classe, que indica sua versão. Toda vez que uma classe sofre alterações em seu código, um novo “serialVersionUID” é gerado. O problema com a serialização surge quando serializa-se o estado de um objeto e posteriormente tenta-se restaurar o estado para a aplicação, mas houveram mudanças no código da classe. Neste caso, uma exceção “java.io.InvalidClassException” é disparada, pois de fato, o estado do objeto corresponde a uma versão de classe do qual foi criado que já não existe mais. Portanto, isso só ocorre para evitar erros de programação, em que tenta-se restaurar o estado de um objeto para uma classe que possui versão diferente da qual o objeto foi serializado inicialmente. E isso pode significar diferentes versões de uma mesma classe “Fruta” por exemplo, ou diferentes versões por classes serem totalmente difere

|  |  |
| --- | --- |
| http://www.linhadecodigo.com.br/artigos/img_artigos/EvertonCoimbraAraujo/Serializazao_Objetos/imagem001.jpg | Neste artigo apresento o Gustavo Rafael Valiati, que pode ser contatado pelo e-mail gustavovaliati@gmail.com. O Gustavo foi meu aluno em algumas disciplinas de sua gradução e está concluíndo o último período do curso de Tecnologia em Análise e Desenvolvimento de Sistemas da UTFPR campus Medianeira, neste primeiro semestre de 2012. |

**1. INTRODUÇÃO**

Nem sempre se deseja ter objetos ou estrutura de dados de um aplicativo, com um ciclo de vida que dura no máximo o tempo em que for armazenado em memória RAM. Ou seja, algumas vezes deseja-se guardá-los por um tempo indeterminado, sendo possível recuperá-los posteriormente. E neste caso, o termo “persistência” tem seu conceito definido.

Persistir dados, requer que estes possam ser serializáveis. A serialização possibilitará capturar o estado do objeto ou a estrutura de dados, e transformar em uma cadeia de bytes quando for necessário. Torna-se possível também, recuperar os bytes persistidos para fazer o processo inverso, e ter os dados de volta para a aplicação em execução.

Desta maneira, qualquer forma de dados que esteja sendo trabalhada na aplicação, e esta puder ser serializada, poderá ser passada para o meio físico, e ter o processo revertido.

A serialização não se limita apenas em possibilitar a gravação do dado em disco, mas permite que o mesmo seja transmitido por rede. Isto porquê a serialização cria um stream de bytes, que é requisito para transmissão de dados por rede.

Diversas linguagens possuem suporte à serialização, como: Java, C, C++, Python, PHP, .NET, entre outras. Neste artigo, será tratado apenas da serialização com Java, em sua implemetação básica, alguns problemas e soluções, com códigos para exemplificação.

**2. SERIALIZAÇÃO COM JAVA**

Serializar um objeto, em Java, só é possível caso sua classe esteja marcada como serializável. Essa marcação é feita com a simples implementação da interface “java.io.Serializable”. Essa interface não possui métodos para se implementar e nem atributos. Ainda não existiam as anotações no Java 1.1, versão que trouxe pela primeira vez a serialização ao Java, e desta maneira a implementação da interface era a melhor maneira de marcar uma classe.

A herança entre classes é naturalmente afetada pela serialização. Já que quando uma classe implementa a interface “Serializable”, toda classe que dela estender também estará implicitamente marcada como serializável

.

Para exemplo, será criada uma classe “pojo.Fruta” que contém dois atributos: nome e cor. No código exemplo (Listagem 1), pode-se notar a classe “Fruta” marcada para a serialização:

Listagem 1 - Marcando uma classe como serializável.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | import java.io.Serializable;    public class Fruta implements Serializable{      private String nome;      private String cor;        public Fruta(String nome, String cor) {          this.nome = nome;          this.cor = cor;      }        public String getCor() { return cor; }      public String getNome() { return nome; }      public void setCor(String cor) { this.cor = cor; }      public void setNome(String nome) { this.nome = nome; }  } |

Pode-se dizer que a implementação do processo de serialização e deserialização simples de um objeto, não possui um código difícil de se escrever. Isto porquê duas classes fazem a maior parte do trabalho, de maneira que com poucas linhas de código pode-se transformar um objeto em bytes e vice-versa. Essas duas classes são “java.io.ObjectInputStream” que recebe bytes para criar um objeto e “java.io.ObjectOutputStream” que cria uma cadeia de bytes a partir de um objeto.

**3. SERIALIZANDO E DESERIALIZANDO O ESTADO DE UM OBJETO**

Como exemplo, será persistido um objeto em disco. Para isto, além das classes “java.io.ObjectInputStream” e “java.io.ObjectOutputStream”, serão utilizadas duas outras que trabalham com arquivos, que são “java.io.FileInputStream” e “java.io.FileOutputStream”, que seguem a mesma lógica de InputStream e OutputStream, só que agora relacionado ao sistema de arquivos e não aos objetos.

Para serialização, deve-se criar um novo “java.io.FileOutputStream”, com o caminho (“path”) do arquivo onde se deseja armazenar o estado do objeto. Também deve ser criado um novo “java.io.ObjectOutputStream” que está ligado ao arquivo. E por último deve-se executar a escrita do objeto alvo, para dentro do objeto de stream que por sua vez está ligado ao arquivo destino. Após executados esses passos, é possível encontrar no sistema de arquivos um novo arquivo que contém os bytes correspondentes à descrição do estado do objeto. Na Listagem 2, encontra-se o código exemplo.

Listagem 2 - Serialização de um objeto.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | import java.io.FileOutputStream;  import java.io.ObjectOutputStream;    public class Serializador {        public Serializador() {    }        public void serializar(String path, Object obj) throws Exception {              FileOutputStream outFile = new FileOutputStream(path);      ObjectOutputStream s = new ObjectOutputStream(outFile);      s.writeObject(obj);      s.close();      }  } |

E para a deserialização, o processo é extremamente parecido com a serialização. Cria-se um novo “java.io.FileInputStream” relacionado ao arquivo onde o estado do objeto encontra-se persistido. Deve-se criar também um novo “java.io.ObjectInputStream” que estará vinculado ao arquivo de origem. Por último, executa-se a leitura do arquivo para a restauração do estado. Na Listagem 3 encontra-se o exemplo do código.

Listagem 3 - Deserializaçãode um objeto.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | import java.io.FileInputStream;  import java.io.ObjectInputStream;    public class Deserializador {        public Deserializador() {    }        public Object deserializar(String path) throws Exception {      FileInputStream inFile = new FileInputStream(path);      ObjectInputStream d = new ObjectInputStream(inFile);      Object o = d.readObject();      d.close();      return o;      }  } |

A Listagem 4, contém o exemplo da execução da serialização e em seguida da deserialização.

Listagem 4 - Serializando e Deserializando

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | import operacoes.Deserializador;  import operacoes.Serializador;  import pojo.Fruta;    public class Principal {      public static void main(String args[]){         //serializa         Serializador s = new Serializador();         Fruta fruta = new Fruta("Maça", "Vermelha");          try {              s.serializar("/home/gustavo/fruta", fruta);          } catch (Exception ex) {              System.err.println("Falha ao serializar! - " + ex.toString());          }            //deserializa          Deserializador d = new Deserializador();          fruta = null;          try {              fruta = (Fruta) d.deserializar("/home/gustavo/fruta");          } catch (Exception ex) {              System.err.println("Falha ao deserializar! - " + ex.toString());          }          System.out.println(fruta.getNome() + " - " + fruta.getCor());      }  } |

Na Listagem 5, encontra-se um exemplo de serialização e deserialização de um ArrayList da classe Fruta. É apenas mais um exemplo que demonstra a capacidade de se persistir qualquer objeto que seja serializável. Essa serialização só foi possível porque a classe “ArrayList” implementa a interface “Serializable” por padrão.

Listagem 5 - Serialização de ArrayList.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | //serialização e deserialização com ArrayList          ArrayList<Fruta> frutas = new ArrayList<Fruta>();          Fruta f1 = new Fruta("Laranja", "Amarela");          Fruta f2 = new Fruta("Abacate", "Verde");          Fruta f3 = new Fruta("Morango", "Vermelho");          frutas.add(f1);          frutas.add(f2);          frutas.add(f3);            try {              s.serializar("/home/gustavo/frutas", frutas);              frutas = null;              frutas = (ArrayList<Fruta>) d.deserializar("/home/gustavo/frutas");              for(Fruta f : frutas){                  System.out.println("ArrayList: " + f.getNome() + " - " + f.getCor());              }          } catch (Exception ex) {              System.err.println("Falha ao serializar ou deserializar! - " +                                 ex.toString());          } |

**4. PARTICULARIDADES DA SERIALIZAÇÃO**

Como dito, a serialização de um objeto em Java pode ser algo fácil de se fazer. Mas existem certas informações que são importantes e sabê-las pode evitar vários problemas.

Um dos problemas é que nem tudo pode ser serializado, como objetos das classes “Thread” e “Socket” e atributos estáticos. Os atributos estáticos estão vinculados à classe e não ao objeto. Assim quando se trata do estado do objeto, os campos estáticos não são considerados. Em relação às Threads, suas instâncias só são úteis enquanto estiverem em execução na JVM, e podem ser paradas/destruídas e iniciadas com instruções de sua classe. Não existe algo no estado destas instâncias que seja útil para se persistir, e por isso não há sentido em persistir tais instâncias para restaurar posteriormente.

Mas e se existir alguma classe em que seja necessário persisti-la, e esta possui uma instância da classe “Thread”? Neste caso, utilizando um recurso pode-se persistir a classe inteira, menos a instância thread. Este recurso, refere-se em indicar que tal instância thread é um objeto “Transient”. Na Listagem 6 encontra-se um exemplo da implementação com uso de “Transient”.

Listagem 6 - Uso do Transient

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | import java.io.Serializable;    public class ImplementacaoTransient implements Serializable, Runnable{        transient private Thread thread;      //marcado com "transient" para que sua instância não seja serializada        //Demais valores de estado serão serializados      private int valorParaPersistir;        public ImplementacaoTransient(int valorParaPersistir){          this.valorParaPersistir = valorParaPersistir;          thread = new Thread(this);          thread.start();      }        public void run()      {          while(true){              //código a ser executado pela Thread          }      }  } |

Vale ressaltar que, o que é serializado é apenas o estado do objeto, e não os métodos e a classe. No exemplo acima, apenas a instância do objeto “thread” e do objeto “valorParaPersistir” compunham o estado do objeto. Assim, evita-se que o sistema de serialização tente serializar um objeto não-serializável e gere erros. Após a deserialização de um objeto da classe “ImplementacaoTransient” este estará totalmente funcional.

Um outro problema que pode vir a ocorrer, se refere ao versionamento das classes. Toda classe possui um atributo oculto chamado “serialVersionUID” com um valor gerado automaticamente. Objetos criados levam consigo o “serialVersionUID” de sua classe, que indica sua versão. Toda vez que uma classe sofre alterações em seu código, um novo “serialVersionUID” é gerado. O problema com a serialização surge quando serializa-se o estado de um objeto e posteriormente tenta-se restaurar o estado para a aplicação, mas houveram mudanças no código da classe. Neste caso, uma exceção “java.io.InvalidClassException” é disparada, pois de fato, o estado do objeto corresponde a uma versão de classe do qual foi criado que já não existe mais. Portanto, isso só ocorre para evitar erros de programação, em que tenta-se restaurar o estado de um objeto para uma classe que possui versão diferente da qual o objeto foi serializado inicialmente. E isso pode significar diferentes versões de uma mesma classe “Fruta” por exemplo, ou diferentes versões por classes serem totalmente difere

|  |  |
| --- | --- |
| http://www.linhadecodigo.com.br/artigos/img_artigos/EvertonCoimbraAraujo/Serializazao_Objetos/imagem001.jpg | Neste artigo apresento o Gustavo Rafael Valiati, que pode ser contatado pelo e-mail gustavovaliati@gmail.com. O Gustavo foi meu aluno em algumas disciplinas de sua gradução e está concluíndo o último período do curso de Tecnologia em Análise e Desenvolvimento de Sistemas da UTFPR campus Medianeira, neste primeiro semestre de 2012. |

**1. INTRODUÇÃO**

Nem sempre se deseja ter objetos ou estrutura de dados de um aplicativo, com um ciclo de vida que dura no máximo o tempo em que for armazenado em memória RAM. Ou seja, algumas vezes deseja-se guardá-los por um tempo indeterminado, sendo possível recuperá-los posteriormente. E neste caso, o termo “persistência” tem seu conceito definido.

Persistir dados, requer que estes possam ser serializáveis. A serialização possibilitará capturar o estado do objeto ou a estrutura de dados, e transformar em uma cadeia de bytes quando for necessário. Torna-se possível também, recuperar os bytes persistidos para fazer o processo inverso, e ter os dados de volta para a aplicação em execução.

Desta maneira, qualquer forma de dados que esteja sendo trabalhada na aplicação, e esta puder ser serializada, poderá ser passada para o meio físico, e ter o processo revertido.

A serialização não se limita apenas em possibilitar a gravação do dado em disco, mas permite que o mesmo seja transmitido por rede. Isto porquê a serialização cria um stream de bytes, que é requisito para transmissão de dados por rede.

Diversas linguagens possuem suporte à serialização, como: Java, C, C++, Python, PHP, .NET, entre outras. Neste artigo, será tratado apenas da serialização com Java, em sua implemetação básica, alguns problemas e soluções, com códigos para exemplificação.

**2. SERIALIZAÇÃO COM JAVA**

Serializar um objeto, em Java, só é possível caso sua classe esteja marcada como serializável. Essa marcação é feita com a simples implementação da interface “java.io.Serializable”. Essa interface não possui métodos para se implementar e nem atributos. Ainda não existiam as anotações no Java 1.1, versão que trouxe pela primeira vez a serialização ao Java, e desta maneira a implementação da interface era a melhor maneira de marcar uma classe.

A herança entre classes é naturalmente afetada pela serialização. Já que quando uma classe implementa a interface “Serializable”, toda classe que dela estender também estará implicitamente marcada como serializável

.

Para exemplo, será criada uma classe “pojo.Fruta” que contém dois atributos: nome e cor. No código exemplo (Listagem 1), pode-se notar a classe “Fruta” marcada para a serialização:

Listagem 1 - Marcando uma classe como serializável.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | import java.io.Serializable;    public class Fruta implements Serializable{      private String nome;      private String cor;        public Fruta(String nome, String cor) {          this.nome = nome;          this.cor = cor;      }        public String getCor() { return cor; }      public String getNome() { return nome; }      public void setCor(String cor) { this.cor = cor; }      public void setNome(String nome) { this.nome = nome; }  } |

Pode-se dizer que a implementação do processo de serialização e deserialização simples de um objeto, não possui um código difícil de se escrever. Isto porquê duas classes fazem a maior parte do trabalho, de maneira que com poucas linhas de código pode-se transformar um objeto em bytes e vice-versa. Essas duas classes são “java.io.ObjectInputStream” que recebe bytes para criar um objeto e “java.io.ObjectOutputStream” que cria uma cadeia de bytes a partir de um objeto.

**3. SERIALIZANDO E DESERIALIZANDO O ESTADO DE UM OBJETO**

Como exemplo, será persistido um objeto em disco. Para isto, além das classes “java.io.ObjectInputStream” e “java.io.ObjectOutputStream”, serão utilizadas duas outras que trabalham com arquivos, que são “java.io.FileInputStream” e “java.io.FileOutputStream”, que seguem a mesma lógica de InputStream e OutputStream, só que agora relacionado ao sistema de arquivos e não aos objetos.

Para serialização, deve-se criar um novo “java.io.FileOutputStream”, com o caminho (“path”) do arquivo onde se deseja armazenar o estado do objeto. Também deve ser criado um novo “java.io.ObjectOutputStream” que está ligado ao arquivo. E por último deve-se executar a escrita do objeto alvo, para dentro do objeto de stream que por sua vez está ligado ao arquivo destino. Após executados esses passos, é possível encontrar no sistema de arquivos um novo arquivo que contém os bytes correspondentes à descrição do estado do objeto. Na Listagem 2, encontra-se o código exemplo.

Listagem 2 - Serialização de um objeto.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | import java.io.FileOutputStream;  import java.io.ObjectOutputStream;    public class Serializador {        public Serializador() {    }        public void serializar(String path, Object obj) throws Exception {              FileOutputStream outFile = new FileOutputStream(path);      ObjectOutputStream s = new ObjectOutputStream(outFile);      s.writeObject(obj);      s.close();      }  } |

E para a deserialização, o processo é extremamente parecido com a serialização. Cria-se um novo “java.io.FileInputStream” relacionado ao arquivo onde o estado do objeto encontra-se persistido. Deve-se criar também um novo “java.io.ObjectInputStream” que estará vinculado ao arquivo de origem. Por último, executa-se a leitura do arquivo para a restauração do estado. Na Listagem 3 encontra-se o exemplo do código.

Listagem 3 - Deserializaçãode um objeto.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | import java.io.FileInputStream;  import java.io.ObjectInputStream;    public class Deserializador {        public Deserializador() {    }        public Object deserializar(String path) throws Exception {      FileInputStream inFile = new FileInputStream(path);      ObjectInputStream d = new ObjectInputStream(inFile);      Object o = d.readObject();      d.close();      return o;      }  } |

A Listagem 4, contém o exemplo da execução da serialização e em seguida da deserialização.

Listagem 4 - Serializando e Deserializando

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | import operacoes.Deserializador;  import operacoes.Serializador;  import pojo.Fruta;    public class Principal {      public static void main(String args[]){         //serializa         Serializador s = new Serializador();         Fruta fruta = new Fruta("Maça", "Vermelha");          try {              s.serializar("/home/gustavo/fruta", fruta);          } catch (Exception ex) {              System.err.println("Falha ao serializar! - " + ex.toString());          }            //deserializa          Deserializador d = new Deserializador();          fruta = null;          try {              fruta = (Fruta) d.deserializar("/home/gustavo/fruta");          } catch (Exception ex) {              System.err.println("Falha ao deserializar! - " + ex.toString());          }          System.out.println(fruta.getNome() + " - " + fruta.getCor());      }  } |

Na Listagem 5, encontra-se um exemplo de serialização e deserialização de um ArrayList da classe Fruta. É apenas mais um exemplo que demonstra a capacidade de se persistir qualquer objeto que seja serializável. Essa serialização só foi possível porque a classe “ArrayList” implementa a interface “Serializable” por padrão.

Listagem 5 - Serialização de ArrayList.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | //serialização e deserialização com ArrayList          ArrayList<Fruta> frutas = new ArrayList<Fruta>();          Fruta f1 = new Fruta("Laranja", "Amarela");          Fruta f2 = new Fruta("Abacate", "Verde");          Fruta f3 = new Fruta("Morango", "Vermelho");          frutas.add(f1);          frutas.add(f2);          frutas.add(f3);            try {              s.serializar("/home/gustavo/frutas", frutas);              frutas = null;              frutas = (ArrayList<Fruta>) d.deserializar("/home/gustavo/frutas");              for(Fruta f : frutas){                  System.out.println("ArrayList: " + f.getNome() + " - " + f.getCor());              }          } catch (Exception ex) {              System.err.println("Falha ao serializar ou deserializar! - " +                                 ex.toString());          } |

**4. PARTICULARIDADES DA SERIALIZAÇÃO**

Como dito, a serialização de um objeto em Java pode ser algo fácil de se fazer. Mas existem certas informações que são importantes e sabê-las pode evitar vários problemas.

Um dos problemas é que nem tudo pode ser serializado, como objetos das classes “Thread” e “Socket” e atributos estáticos. Os atributos estáticos estão vinculados à classe e não ao objeto. Assim quando se trata do estado do objeto, os campos estáticos não são considerados. Em relação às Threads, suas instâncias só são úteis enquanto estiverem em execução na JVM, e podem ser paradas/destruídas e iniciadas com instruções de sua classe. Não existe algo no estado destas instâncias que seja útil para se persistir, e por isso não há sentido em persistir tais instâncias para restaurar posteriormente.

Mas e se existir alguma classe em que seja necessário persisti-la, e esta possui uma instância da classe “Thread”? Neste caso, utilizando um recurso pode-se persistir a classe inteira, menos a instância thread. Este recurso, refere-se em indicar que tal instância thread é um objeto “Transient”. Na Listagem 6 encontra-se um exemplo da implementação com uso de “Transient”.

Listagem 6 - Uso do Transient

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | import java.io.Serializable;    public class ImplementacaoTransient implements Serializable, Runnable{        transient private Thread thread;      //marcado com "transient" para que sua instância não seja serializada        //Demais valores de estado serão serializados      private int valorParaPersistir;        public ImplementacaoTransient(int valorParaPersistir){          this.valorParaPersistir = valorParaPersistir;          thread = new Thread(this);          thread.start();      }        public void run()      {          while(true){              //código a ser executado pela Thread          }      }  } |

Vale ressaltar que, o que é serializado é apenas o estado do objeto, e não os métodos e a classe. No exemplo acima, apenas a instância do objeto “thread” e do objeto “valorParaPersistir” compunham o estado do objeto. Assim, evita-se que o sistema de serialização tente serializar um objeto não-serializável e gere erros. Após a deserialização de um objeto da classe “ImplementacaoTransient” este estará totalmente funcional.

Um outro problema que pode vir a ocorrer, se refere ao versionamento das classes. Toda classe possui um atributo oculto chamado “serialVersionUID” com um valor gerado automaticamente. Objetos criados levam consigo o “serialVersionUID” de sua classe, que indica sua versão. Toda vez que uma classe sofre alterações em seu código, um novo “serialVersionUID” é gerado. O problema com a serialização surge quando serializa-se o estado de um objeto e posteriormente tenta-se restaurar o estado para a aplicação, mas houveram mudanças no código da classe. Neste caso, uma exceção “java.io.InvalidClassException” é disparada, pois de fato, o estado do objeto corresponde a uma versão de classe do qual foi criado que já não existe mais. Portanto, isso só ocorre para evitar erros de programação, em que tenta-se restaurar o estado de um objeto para uma classe que possui versão diferente da qual o objeto foi serializado inicialmente. E isso pode significar diferentes versões de uma mesma classe “Fruta” por exemplo, ou diferentes versões por classes serem totalmente difere

|  |  |
| --- | --- |
| http://www.linhadecodigo.com.br/artigos/img_artigos/EvertonCoimbraAraujo/Serializazao_Objetos/imagem001.jpg | Neste artigo apresento o Gustavo Rafael Valiati, que pode ser contatado pelo e-mail gustavovaliati@gmail.com. O Gustavo foi meu aluno em algumas disciplinas de sua gradução e está concluíndo o último período do curso de Tecnologia em Análise e Desenvolvimento de Sistemas da UTFPR campus Medianeira, neste primeiro semestre de 2012. |

**1. INTRODUÇÃO**

Nem sempre se deseja ter objetos ou estrutura de dados de um aplicativo, com um ciclo de vida que dura no máximo o tempo em que for armazenado em memória RAM. Ou seja, algumas vezes deseja-se guardá-los por um tempo indeterminado, sendo possível recuperá-los posteriormente. E neste caso, o termo “persistência” tem seu conceito definido.

Persistir dados, requer que estes possam ser serializáveis. A serialização possibilitará capturar o estado do objeto ou a estrutura de dados, e transformar em uma cadeia de bytes quando for necessário. Torna-se possível também, recuperar os bytes persistidos para fazer o processo inverso, e ter os dados de volta para a aplicação em execução.

Desta maneira, qualquer forma de dados que esteja sendo trabalhada na aplicação, e esta puder ser serializada, poderá ser passada para o meio físico, e ter o processo revertido.

A serialização não se limita apenas em possibilitar a gravação do dado em disco, mas permite que o mesmo seja transmitido por rede. Isto porquê a serialização cria um stream de bytes, que é requisito para transmissão de dados por rede.

Diversas linguagens possuem suporte à serialização, como: Java, C, C++, Python, PHP, .NET, entre outras. Neste artigo, será tratado apenas da serialização com Java, em sua implemetação básica, alguns problemas e soluções, com códigos para exemplificação.

**2. SERIALIZAÇÃO COM JAVA**

Serializar um objeto, em Java, só é possível caso sua classe esteja marcada como serializável. Essa marcação é feita com a simples implementação da interface “java.io.Serializable”. Essa interface não possui métodos para se implementar e nem atributos. Ainda não existiam as anotações no Java 1.1, versão que trouxe pela primeira vez a serialização ao Java, e desta maneira a implementação da interface era a melhor maneira de marcar uma classe.

A herança entre classes é naturalmente afetada pela serialização. Já que quando uma classe implementa a interface “Serializable”, toda classe que dela estender também estará implicitamente marcada como serializável

.

Para exemplo, será criada uma classe “pojo.Fruta” que contém dois atributos: nome e cor. No código exemplo (Listagem 1), pode-se notar a classe “Fruta” marcada para a serialização:

Listagem 1 - Marcando uma classe como serializável.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | import java.io.Serializable;    public class Fruta implements Serializable{      private String nome;      private String cor;        public Fruta(String nome, String cor) {          this.nome = nome;          this.cor = cor;      }        public String getCor() { return cor; }      public String getNome() { return nome; }      public void setCor(String cor) { this.cor = cor; }      public void setNome(String nome) { this.nome = nome; }  } |

Pode-se dizer que a implementação do processo de serialização e deserialização simples de um objeto, não possui um código difícil de se escrever. Isto porquê duas classes fazem a maior parte do trabalho, de maneira que com poucas linhas de código pode-se transformar um objeto em bytes e vice-versa. Essas duas classes são “java.io.ObjectInputStream” que recebe bytes para criar um objeto e “java.io.ObjectOutputStream” que cria uma cadeia de bytes a partir de um objeto.

**3. SERIALIZANDO E DESERIALIZANDO O ESTADO DE UM OBJETO**

Como exemplo, será persistido um objeto em disco. Para isto, além das classes “java.io.ObjectInputStream” e “java.io.ObjectOutputStream”, serão utilizadas duas outras que trabalham com arquivos, que são “java.io.FileInputStream” e “java.io.FileOutputStream”, que seguem a mesma lógica de InputStream e OutputStream, só que agora relacionado ao sistema de arquivos e não aos objetos.

Para serialização, deve-se criar um novo “java.io.FileOutputStream”, com o caminho (“path”) do arquivo onde se deseja armazenar o estado do objeto. Também deve ser criado um novo “java.io.ObjectOutputStream” que está ligado ao arquivo. E por último deve-se executar a escrita do objeto alvo, para dentro do objeto de stream que por sua vez está ligado ao arquivo destino. Após executados esses passos, é possível encontrar no sistema de arquivos um novo arquivo que contém os bytes correspondentes à descrição do estado do objeto. Na Listagem 2, encontra-se o código exemplo.

Listagem 2 - Serialização de um objeto.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | import java.io.FileOutputStream;  import java.io.ObjectOutputStream;    public class Serializador {        public Serializador() {    }        public void serializar(String path, Object obj) throws Exception {              FileOutputStream outFile = new FileOutputStream(path);      ObjectOutputStream s = new ObjectOutputStream(outFile);      s.writeObject(obj);      s.close();      }  } |

E para a deserialização, o processo é extremamente parecido com a serialização. Cria-se um novo “java.io.FileInputStream” relacionado ao arquivo onde o estado do objeto encontra-se persistido. Deve-se criar também um novo “java.io.ObjectInputStream” que estará vinculado ao arquivo de origem. Por último, executa-se a leitura do arquivo para a restauração do estado. Na Listagem 3 encontra-se o exemplo do código.

Listagem 3 - Deserializaçãode um objeto.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | import java.io.FileInputStream;  import java.io.ObjectInputStream;    public class Deserializador {        public Deserializador() {    }        public Object deserializar(String path) throws Exception {      FileInputStream inFile = new FileInputStream(path);      ObjectInputStream d = new ObjectInputStream(inFile);      Object o = d.readObject();      d.close();      return o;      }  } |

A Listagem 4, contém o exemplo da execução da serialização e em seguida da deserialização.

Listagem 4 - Serializando e Deserializando

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | import operacoes.Deserializador;  import operacoes.Serializador;  import pojo.Fruta;    public class Principal {      public static void main(String args[]){         //serializa         Serializador s = new Serializador();         Fruta fruta = new Fruta("Maça", "Vermelha");          try {              s.serializar("/home/gustavo/fruta", fruta);          } catch (Exception ex) {              System.err.println("Falha ao serializar! - " + ex.toString());          }            //deserializa          Deserializador d = new Deserializador();          fruta = null;          try {              fruta = (Fruta) d.deserializar("/home/gustavo/fruta");          } catch (Exception ex) {              System.err.println("Falha ao deserializar! - " + ex.toString());          }          System.out.println(fruta.getNome() + " - " + fruta.getCor());      }  } |

Na Listagem 5, encontra-se um exemplo de serialização e deserialização de um ArrayList da classe Fruta. É apenas mais um exemplo que demonstra a capacidade de se persistir qualquer objeto que seja serializável. Essa serialização só foi possível porque a classe “ArrayList” implementa a interface “Serializable” por padrão.

Listagem 5 - Serialização de ArrayList.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | //serialização e deserialização com ArrayList          ArrayList<Fruta> frutas = new ArrayList<Fruta>();          Fruta f1 = new Fruta("Laranja", "Amarela");          Fruta f2 = new Fruta("Abacate", "Verde");          Fruta f3 = new Fruta("Morango", "Vermelho");          frutas.add(f1);          frutas.add(f2);          frutas.add(f3);            try {              s.serializar("/home/gustavo/frutas", frutas);              frutas = null;              frutas = (ArrayList<Fruta>) d.deserializar("/home/gustavo/frutas");              for(Fruta f : frutas){                  System.out.println("ArrayList: " + f.getNome() + " - " + f.getCor());              }          } catch (Exception ex) {              System.err.println("Falha ao serializar ou deserializar! - " +                                 ex.toString());          } |

**4. PARTICULARIDADES DA SERIALIZAÇÃO**

Como dito, a serialização de um objeto em Java pode ser algo fácil de se fazer. Mas existem certas informações que são importantes e sabê-las pode evitar vários problemas.

Um dos problemas é que nem tudo pode ser serializado, como objetos das classes “Thread” e “Socket” e atributos estáticos. Os atributos estáticos estão vinculados à classe e não ao objeto. Assim quando se trata do estado do objeto, os campos estáticos não são considerados. Em relação às Threads, suas instâncias só são úteis enquanto estiverem em execução na JVM, e podem ser paradas/destruídas e iniciadas com instruções de sua classe. Não existe algo no estado destas instâncias que seja útil para se persistir, e por isso não há sentido em persistir tais instâncias para restaurar posteriormente.

Mas e se existir alguma classe em que seja necessário persisti-la, e esta possui uma instância da classe “Thread”? Neste caso, utilizando um recurso pode-se persistir a classe inteira, menos a instância thread. Este recurso, refere-se em indicar que tal instância thread é um objeto “Transient”. Na Listagem 6 encontra-se um exemplo da implementação com uso de “Transient”.

Listagem 6 - Uso do Transient

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | import java.io.Serializable;    public class ImplementacaoTransient implements Serializable, Runnable{        transient private Thread thread;      //marcado com "transient" para que sua instância não seja serializada        //Demais valores de estado serão serializados      private int valorParaPersistir;        public ImplementacaoTransient(int valorParaPersistir){          this.valorParaPersistir = valorParaPersistir;          thread = new Thread(this);          thread.start();      }        public void run()      {          while(true){              //código a ser executado pela Thread          }      }  } |

Vale ressaltar que, o que é serializado é apenas o estado do objeto, e não os métodos e a classe. No exemplo acima, apenas a instância do objeto “thread” e do objeto “valorParaPersistir” compunham o estado do objeto. Assim, evita-se que o sistema de serialização tente serializar um objeto não-serializável e gere erros. Após a deserialização de um objeto da classe “ImplementacaoTransient” este estará totalmente funcional.

Um outro problema que pode vir a ocorrer, se refere ao versionamento das classes. Toda classe possui um atributo oculto chamado “serialVersionUID” com um valor gerado automaticamente. Objetos criados levam consigo o “serialVersionUID” de sua classe, que indica sua versão. Toda vez que uma classe sofre alterações em seu código, um novo “serialVersionUID” é gerado. O problema com a serialização surge quando serializa-se o estado de um objeto e posteriormente tenta-se restaurar o estado para a aplicação, mas houveram mudanças no código da classe. Neste caso, uma exceção “java.io.InvalidClassException” é disparada, pois de fato, o estado do objeto corresponde a uma versão de classe do qual foi criado que já não existe mais. Portanto, isso só ocorre para evitar erros de programação, em que tenta-se restaurar o estado de um objeto para uma classe que possui versão diferente da qual o objeto foi serializado inicialmente. E isso pode significar diferentes versões de uma mesma classe “Fruta” por exemplo, ou diferentes versões por classes serem totalmente difere

|  |  |
| --- | --- |
| http://www.linhadecodigo.com.br/artigos/img_artigos/EvertonCoimbraAraujo/Serializazao_Objetos/imagem001.jpg | Neste artigo apresento o Gustavo Rafael Valiati, que pode ser contatado pelo e-mail gustavovaliati@gmail.com. O Gustavo foi meu aluno em algumas disciplinas de sua gradução e está concluíndo o último período do curso de Tecnologia em Análise e Desenvolvimento de Sistemas da UTFPR campus Medianeira, neste primeiro semestre de 2012. |

**1. INTRODUÇÃO**

Nem sempre se deseja ter objetos ou estrutura de dados de um aplicativo, com um ciclo de vida que dura no máximo o tempo em que for armazenado em memória RAM. Ou seja, algumas vezes deseja-se guardá-los por um tempo indeterminado, sendo possível recuperá-los posteriormente. E neste caso, o termo “persistência” tem seu conceito definido.

Persistir dados, requer que estes possam ser serializáveis. A serialização possibilitará capturar o estado do objeto ou a estrutura de dados, e transformar em uma cadeia de bytes quando for necessário. Torna-se possível também, recuperar os bytes persistidos para fazer o processo inverso, e ter os dados de volta para a aplicação em execução.

Desta maneira, qualquer forma de dados que esteja sendo trabalhada na aplicação, e esta puder ser serializada, poderá ser passada para o meio físico, e ter o processo revertido.

A serialização não se limita apenas em possibilitar a gravação do dado em disco, mas permite que o mesmo seja transmitido por rede. Isto porquê a serialização cria um stream de bytes, que é requisito para transmissão de dados por rede.

Diversas linguagens possuem suporte à serialização, como: Java, C, C++, Python, PHP, .NET, entre outras. Neste artigo, será tratado apenas da serialização com Java, em sua implemetação básica, alguns problemas e soluções, com códigos para exemplificação.

**2. SERIALIZAÇÃO COM JAVA**

Serializar um objeto, em Java, só é possível caso sua classe esteja marcada como serializável. Essa marcação é feita com a simples implementação da interface “java.io.Serializable”. Essa interface não possui métodos para se implementar e nem atributos. Ainda não existiam as anotações no Java 1.1, versão que trouxe pela primeira vez a serialização ao Java, e desta maneira a implementação da interface era a melhor maneira de marcar uma classe.

A herança entre classes é naturalmente afetada pela serialização. Já que quando uma classe implementa a interface “Serializable”, toda classe que dela estender também estará implicitamente marcada como serializável

.

Para exemplo, será criada uma classe “pojo.Fruta” que contém dois atributos: nome e cor. No código exemplo (Listagem 1), pode-se notar a classe “Fruta” marcada para a serialização:

Listagem 1 - Marcando uma classe como serializável.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | import java.io.Serializable;    public class Fruta implements Serializable{      private String nome;      private String cor;        public Fruta(String nome, String cor) {          this.nome = nome;          this.cor = cor;      }        public String getCor() { return cor; }      public String getNome() { return nome; }      public void setCor(String cor) { this.cor = cor; }      public void setNome(String nome) { this.nome = nome; }  } |

Pode-se dizer que a implementação do processo de serialização e deserialização simples de um objeto, não possui um código difícil de se escrever. Isto porquê duas classes fazem a maior parte do trabalho, de maneira que com poucas linhas de código pode-se transformar um objeto em bytes e vice-versa. Essas duas classes são “java.io.ObjectInputStream” que recebe bytes para criar um objeto e “java.io.ObjectOutputStream” que cria uma cadeia de bytes a partir de um objeto.

**3. SERIALIZANDO E DESERIALIZANDO O ESTADO DE UM OBJETO**

Como exemplo, será persistido um objeto em disco. Para isto, além das classes “java.io.ObjectInputStream” e “java.io.ObjectOutputStream”, serão utilizadas duas outras que trabalham com arquivos, que são “java.io.FileInputStream” e “java.io.FileOutputStream”, que seguem a mesma lógica de InputStream e OutputStream, só que agora relacionado ao sistema de arquivos e não aos objetos.

Para serialização, deve-se criar um novo “java.io.FileOutputStream”, com o caminho (“path”) do arquivo onde se deseja armazenar o estado do objeto. Também deve ser criado um novo “java.io.ObjectOutputStream” que está ligado ao arquivo. E por último deve-se executar a escrita do objeto alvo, para dentro do objeto de stream que por sua vez está ligado ao arquivo destino. Após executados esses passos, é possível encontrar no sistema de arquivos um novo arquivo que contém os bytes correspondentes à descrição do estado do objeto. Na Listagem 2, encontra-se o código exemplo.

Listagem 2 - Serialização de um objeto.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | import java.io.FileOutputStream;  import java.io.ObjectOutputStream;    public class Serializador {        public Serializador() {    }        public void serializar(String path, Object obj) throws Exception {              FileOutputStream outFile = new FileOutputStream(path);      ObjectOutputStream s = new ObjectOutputStream(outFile);      s.writeObject(obj);      s.close();      }  } |

E para a deserialização, o processo é extremamente parecido com a serialização. Cria-se um novo “java.io.FileInputStream” relacionado ao arquivo onde o estado do objeto encontra-se persistido. Deve-se criar também um novo “java.io.ObjectInputStream” que estará vinculado ao arquivo de origem. Por último, executa-se a leitura do arquivo para a restauração do estado. Na Listagem 3 encontra-se o exemplo do código.

Listagem 3 - Deserializaçãode um objeto.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | import java.io.FileInputStream;  import java.io.ObjectInputStream;    public class Deserializador {        public Deserializador() {    }        public Object deserializar(String path) throws Exception {      FileInputStream inFile = new FileInputStream(path);      ObjectInputStream d = new ObjectInputStream(inFile);      Object o = d.readObject();      d.close();      return o;      }  } |

A Listagem 4, contém o exemplo da execução da serialização e em seguida da deserialização.

Listagem 4 - Serializando e Deserializando

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | import operacoes.Deserializador;  import operacoes.Serializador;  import pojo.Fruta;    public class Principal {      public static void main(String args[]){         //serializa         Serializador s = new Serializador();         Fruta fruta = new Fruta("Maça", "Vermelha");          try {              s.serializar("/home/gustavo/fruta", fruta);          } catch (Exception ex) {              System.err.println("Falha ao serializar! - " + ex.toString());          }            //deserializa          Deserializador d = new Deserializador();          fruta = null;          try {              fruta = (Fruta) d.deserializar("/home/gustavo/fruta");          } catch (Exception ex) {              System.err.println("Falha ao deserializar! - " + ex.toString());          }          System.out.println(fruta.getNome() + " - " + fruta.getCor());      }  } |

Na Listagem 5, encontra-se um exemplo de serialização e deserialização de um ArrayList da classe Fruta. É apenas mais um exemplo que demonstra a capacidade de se persistir qualquer objeto que seja serializável. Essa serialização só foi possível porque a classe “ArrayList” implementa a interface “Serializable” por padrão.

Listagem 5 - Serialização de ArrayList.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | //serialização e deserialização com ArrayList          ArrayList<Fruta> frutas = new ArrayList<Fruta>();          Fruta f1 = new Fruta("Laranja", "Amarela");          Fruta f2 = new Fruta("Abacate", "Verde");          Fruta f3 = new Fruta("Morango", "Vermelho");          frutas.add(f1);          frutas.add(f2);          frutas.add(f3);            try {              s.serializar("/home/gustavo/frutas", frutas);              frutas = null;              frutas = (ArrayList<Fruta>) d.deserializar("/home/gustavo/frutas");              for(Fruta f : frutas){                  System.out.println("ArrayList: " + f.getNome() + " - " + f.getCor());              }          } catch (Exception ex) {              System.err.println("Falha ao serializar ou deserializar! - " +                                 ex.toString());          } |

**4. PARTICULARIDADES DA SERIALIZAÇÃO**

Como dito, a serialização de um objeto em Java pode ser algo fácil de se fazer. Mas existem certas informações que são importantes e sabê-las pode evitar vários problemas.

Um dos problemas é que nem tudo pode ser serializado, como objetos das classes “Thread” e “Socket” e atributos estáticos. Os atributos estáticos estão vinculados à classe e não ao objeto. Assim quando se trata do estado do objeto, os campos estáticos não são considerados. Em relação às Threads, suas instâncias só são úteis enquanto estiverem em execução na JVM, e podem ser paradas/destruídas e iniciadas com instruções de sua classe. Não existe algo no estado destas instâncias que seja útil para se persistir, e por isso não há sentido em persistir tais instâncias para restaurar posteriormente.

Mas e se existir alguma classe em que seja necessário persisti-la, e esta possui uma instância da classe “Thread”? Neste caso, utilizando um recurso pode-se persistir a classe inteira, menos a instância thread. Este recurso, refere-se em indicar que tal instância thread é um objeto “Transient”. Na Listagem 6 encontra-se um exemplo da implementação com uso de “Transient”.

Listagem 6 - Uso do Transient

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | import java.io.Serializable;    public class ImplementacaoTransient implements Serializable, Runnable{        transient private Thread thread;      //marcado com "transient" para que sua instância não seja serializada        //Demais valores de estado serão serializados      private int valorParaPersistir;        public ImplementacaoTransient(int valorParaPersistir){          this.valorParaPersistir = valorParaPersistir;          thread = new Thread(this);          thread.start();      }        public void run()      {          while(true){              //código a ser executado pela Thread          }      }  } |

Vale ressaltar que, o que é serializado é apenas o estado do objeto, e não os métodos e a classe. No exemplo acima, apenas a instância do objeto “thread” e do objeto “valorParaPersistir” compunham o estado do objeto. Assim, evita-se que o sistema de serialização tente serializar um objeto não-serializável e gere erros. Após a deserialização de um objeto da classe “ImplementacaoTransient” este estará totalmente funcional.

Um outro problema que pode vir a ocorrer, se refere ao versionamento das classes. Toda classe possui um atributo oculto chamado “serialVersionUID” com um valor gerado automaticamente. Objetos criados levam consigo o “serialVersionUID” de sua classe, que indica sua versão. Toda vez que uma classe sofre alterações em seu código, um novo “serialVersionUID” é gerado. O problema com a serialização surge quando serializa-se o estado de um objeto e posteriormente tenta-se restaurar o estado para a aplicação, mas houveram mudanças no código da classe. Neste caso, uma exceção “java.io.InvalidClassException” é disparada, pois de fato, o estado do objeto corresponde a uma versão de classe do qual foi criado que já não existe mais. Portanto, isso só ocorre para evitar erros de programação, em que tenta-se restaurar o estado de um objeto para uma classe que possui versão diferente da qual o objeto foi serializado inicialmente. E isso pode significar diferentes versões de uma mesma classe “Fruta” por exemplo, ou diferentes versões por classes serem totalmente difere

|  |  |
| --- | --- |
| http://www.linhadecodigo.com.br/artigos/img_artigos/EvertonCoimbraAraujo/Serializazao_Objetos/imagem001.jpg | Neste artigo apresento o Gustavo Rafael Valiati, que pode ser contatado pelo e-mail gustavovaliati@gmail.com. O Gustavo foi meu aluno em algumas disciplinas de sua gradução e está concluíndo o último período do curso de Tecnologia em Análise e Desenvolvimento de Sistemas da UTFPR campus Medianeira, neste primeiro semestre de 2012. |

**1. INTRODUÇÃO**

Nem sempre se deseja ter objetos ou estrutura de dados de um aplicativo, com um ciclo de vida que dura no máximo o tempo em que for armazenado em memória RAM. Ou seja, algumas vezes deseja-se guardá-los por um tempo indeterminado, sendo possível recuperá-los posteriormente. E neste caso, o termo “persistência” tem seu conceito definido.

Persistir dados, requer que estes possam ser serializáveis. A serialização possibilitará capturar o estado do objeto ou a estrutura de dados, e transformar em uma cadeia de bytes quando for necessário. Torna-se possível também, recuperar os bytes persistidos para fazer o processo inverso, e ter os dados de volta para a aplicação em execução.

Desta maneira, qualquer forma de dados que esteja sendo trabalhada na aplicação, e esta puder ser serializada, poderá ser passada para o meio físico, e ter o processo revertido.

A serialização não se limita apenas em possibilitar a gravação do dado em disco, mas permite que o mesmo seja transmitido por rede. Isto porquê a serialização cria um stream de bytes, que é requisito para transmissão de dados por rede.

Diversas linguagens possuem suporte à serialização, como: Java, C, C++, Python, PHP, .NET, entre outras. Neste artigo, será tratado apenas da serialização com Java, em sua implemetação básica, alguns problemas e soluções, com códigos para exemplificação.

**2. SERIALIZAÇÃO COM JAVA**

Serializar um objeto, em Java, só é possível caso sua classe esteja marcada como serializável. Essa marcação é feita com a simples implementação da interface “java.io.Serializable”. Essa interface não possui métodos para se implementar e nem atributos. Ainda não existiam as anotações no Java 1.1, versão que trouxe pela primeira vez a serialização ao Java, e desta maneira a implementação da interface era a melhor maneira de marcar uma classe.

A herança entre classes é naturalmente afetada pela serialização. Já que quando uma classe implementa a interface “Serializable”, toda classe que dela estender também estará implicitamente marcada como serializável

.

Para exemplo, será criada uma classe “pojo.Fruta” que contém dois atributos: nome e cor. No código exemplo (Listagem 1), pode-se notar a classe “Fruta” marcada para a serialização:

Listagem 1 - Marcando uma classe como serializável.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | import java.io.Serializable;    public class Fruta implements Serializable{      private String nome;      private String cor;        public Fruta(String nome, String cor) {          this.nome = nome;          this.cor = cor;      }        public String getCor() { return cor; }      public String getNome() { return nome; }      public void setCor(String cor) { this.cor = cor; }      public void setNome(String nome) { this.nome = nome; }  } |

Pode-se dizer que a implementação do processo de serialização e deserialização simples de um objeto, não possui um código difícil de se escrever. Isto porquê duas classes fazem a maior parte do trabalho, de maneira que com poucas linhas de código pode-se transformar um objeto em bytes e vice-versa. Essas duas classes são “java.io.ObjectInputStream” que recebe bytes para criar um objeto e “java.io.ObjectOutputStream” que cria uma cadeia de bytes a partir de um objeto.

**3. SERIALIZANDO E DESERIALIZANDO O ESTADO DE UM OBJETO**

Como exemplo, será persistido um objeto em disco. Para isto, além das classes “java.io.ObjectInputStream” e “java.io.ObjectOutputStream”, serão utilizadas duas outras que trabalham com arquivos, que são “java.io.FileInputStream” e “java.io.FileOutputStream”, que seguem a mesma lógica de InputStream e OutputStream, só que agora relacionado ao sistema de arquivos e não aos objetos.

Para serialização, deve-se criar um novo “java.io.FileOutputStream”, com o caminho (“path”) do arquivo onde se deseja armazenar o estado do objeto. Também deve ser criado um novo “java.io.ObjectOutputStream” que está ligado ao arquivo. E por último deve-se executar a escrita do objeto alvo, para dentro do objeto de stream que por sua vez está ligado ao arquivo destino. Após executados esses passos, é possível encontrar no sistema de arquivos um novo arquivo que contém os bytes correspondentes à descrição do estado do objeto. Na Listagem 2, encontra-se o código exemplo.

Listagem 2 - Serialização de um objeto.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | import java.io.FileOutputStream;  import java.io.ObjectOutputStream;    public class Serializador {        public Serializador() {    }        public void serializar(String path, Object obj) throws Exception {              FileOutputStream outFile = new FileOutputStream(path);      ObjectOutputStream s = new ObjectOutputStream(outFile);      s.writeObject(obj);      s.close();      }  } |

E para a deserialização, o processo é extremamente parecido com a serialização. Cria-se um novo “java.io.FileInputStream” relacionado ao arquivo onde o estado do objeto encontra-se persistido. Deve-se criar também um novo “java.io.ObjectInputStream” que estará vinculado ao arquivo de origem. Por último, executa-se a leitura do arquivo para a restauração do estado. Na Listagem 3 encontra-se o exemplo do código.

Listagem 3 - Deserializaçãode um objeto.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | import java.io.FileInputStream;  import java.io.ObjectInputStream;    public class Deserializador {        public Deserializador() {    }        public Object deserializar(String path) throws Exception {      FileInputStream inFile = new FileInputStream(path);      ObjectInputStream d = new ObjectInputStream(inFile);      Object o = d.readObject();      d.close();      return o;      }  } |

A Listagem 4, contém o exemplo da execução da serialização e em seguida da deserialização.

Listagem 4 - Serializando e Deserializando

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | import operacoes.Deserializador;  import operacoes.Serializador;  import pojo.Fruta;    public class Principal {      public static void main(String args[]){         //serializa         Serializador s = new Serializador();         Fruta fruta = new Fruta("Maça", "Vermelha");          try {              s.serializar("/home/gustavo/fruta", fruta);          } catch (Exception ex) {              System.err.println("Falha ao serializar! - " + ex.toString());          }            //deserializa          Deserializador d = new Deserializador();          fruta = null;          try {              fruta = (Fruta) d.deserializar("/home/gustavo/fruta");          } catch (Exception ex) {              System.err.println("Falha ao deserializar! - " + ex.toString());          }          System.out.println(fruta.getNome() + " - " + fruta.getCor());      }  } |

Na Listagem 5, encontra-se um exemplo de serialização e deserialização de um ArrayList da classe Fruta. É apenas mais um exemplo que demonstra a capacidade de se persistir qualquer objeto que seja serializável. Essa serialização só foi possível porque a classe “ArrayList” implementa a interface “Serializable” por padrão.

Listagem 5 - Serialização de ArrayList.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | //serialização e deserialização com ArrayList          ArrayList<Fruta> frutas = new ArrayList<Fruta>();          Fruta f1 = new Fruta("Laranja", "Amarela");          Fruta f2 = new Fruta("Abacate", "Verde");          Fruta f3 = new Fruta("Morango", "Vermelho");          frutas.add(f1);          frutas.add(f2);          frutas.add(f3);            try {              s.serializar("/home/gustavo/frutas", frutas);              frutas = null;              frutas = (ArrayList<Fruta>) d.deserializar("/home/gustavo/frutas");              for(Fruta f : frutas){                  System.out.println("ArrayList: " + f.getNome() + " - " + f.getCor());              }          } catch (Exception ex) {              System.err.println("Falha ao serializar ou deserializar! - " +                                 ex.toString());          } |

**4. PARTICULARIDADES DA SERIALIZAÇÃO**

Como dito, a serialização de um objeto em Java pode ser algo fácil de se fazer. Mas existem certas informações que são importantes e sabê-las pode evitar vários problemas.

Um dos problemas é que nem tudo pode ser serializado, como objetos das classes “Thread” e “Socket” e atributos estáticos. Os atributos estáticos estão vinculados à classe e não ao objeto. Assim quando se trata do estado do objeto, os campos estáticos não são considerados. Em relação às Threads, suas instâncias só são úteis enquanto estiverem em execução na JVM, e podem ser paradas/destruídas e iniciadas com instruções de sua classe. Não existe algo no estado destas instâncias que seja útil para se persistir, e por isso não há sentido em persistir tais instâncias para restaurar posteriormente.

Mas e se existir alguma classe em que seja necessário persisti-la, e esta possui uma instância da classe “Thread”? Neste caso, utilizando um recurso pode-se persistir a classe inteira, menos a instância thread. Este recurso, refere-se em indicar que tal instância thread é um objeto “Transient”. Na Listagem 6 encontra-se um exemplo da implementação com uso de “Transient”.

Listagem 6 - Uso do Transient

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | import java.io.Serializable;    public class ImplementacaoTransient implements Serializable, Runnable{        transient private Thread thread;      //marcado com "transient" para que sua instância não seja serializada        //Demais valores de estado serão serializados      private int valorParaPersistir;        public ImplementacaoTransient(int valorParaPersistir){          this.valorParaPersistir = valorParaPersistir;          thread = new Thread(this);          thread.start();      }        public void run()      {          while(true){              //código a ser executado pela Thread          }      }  } |

Vale ressaltar que, o que é serializado é apenas o estado do objeto, e não os métodos e a classe. No exemplo acima, apenas a instância do objeto “thread” e do objeto “valorParaPersistir” compunham o estado do objeto. Assim, evita-se que o sistema de serialização tente serializar um objeto não-serializável e gere erros. Após a deserialização de um objeto da classe “ImplementacaoTransient” este estará totalmente funcional.

Um outro problema que pode vir a ocorrer, se refere ao versionamento das classes. Toda classe possui um atributo oculto chamado “serialVersionUID” com um valor gerado automaticamente. Objetos criados levam consigo o “serialVersionUID” de sua classe, que indica sua versão. Toda vez que uma classe sofre alterações em seu código, um novo “serialVersionUID” é gerado. O problema com a serialização surge quando serializa-se o estado de um objeto e posteriormente tenta-se restaurar o estado para a aplicação, mas houveram mudanças no código da classe. Neste caso, uma exceção “java.io.InvalidClassException” é disparada, pois de fato, o estado do objeto corresponde a uma versão de classe do qual foi criado que já não existe mais. Portanto, isso só ocorre para evitar erros de programação, em que tenta-se restaurar o estado de um objeto para uma classe que possui versão diferente da qual o objeto foi serializado inicialmente. E isso pode significar diferentes versões de uma mesma classe “Fruta” por exemplo, ou diferentes versões por classes serem totalmente difere

|  |  |
| --- | --- |
| http://www.linhadecodigo.com.br/artigos/img_artigos/EvertonCoimbraAraujo/Serializazao_Objetos/imagem001.jpg | Neste artigo apresento o Gustavo Rafael Valiati, que pode ser contatado pelo e-mail gustavovaliati@gmail.com. O Gustavo foi meu aluno em algumas disciplinas de sua gradução e está concluíndo o último período do curso de Tecnologia em Análise e Desenvolvimento de Sistemas da UTFPR campus Medianeira, neste primeiro semestre de 2012. |

**1. INTRODUÇÃO**

Nem sempre se deseja ter objetos ou estrutura de dados de um aplicativo, com um ciclo de vida que dura no máximo o tempo em que for armazenado em memória RAM. Ou seja, algumas vezes deseja-se guardá-los por um tempo indeterminado, sendo possível recuperá-los posteriormente. E neste caso, o termo “persistência” tem seu conceito definido.

Persistir dados, requer que estes possam ser serializáveis. A serialização possibilitará capturar o estado do objeto ou a estrutura de dados, e transformar em uma cadeia de bytes quando for necessário. Torna-se possível também, recuperar os bytes persistidos para fazer o processo inverso, e ter os dados de volta para a aplicação em execução.

Desta maneira, qualquer forma de dados que esteja sendo trabalhada na aplicação, e esta puder ser serializada, poderá ser passada para o meio físico, e ter o processo revertido.

A serialização não se limita apenas em possibilitar a gravação do dado em disco, mas permite que o mesmo seja transmitido por rede. Isto porquê a serialização cria um stream de bytes, que é requisito para transmissão de dados por rede.

Diversas linguagens possuem suporte à serialização, como: Java, C, C++, Python, PHP, .NET, entre outras. Neste artigo, será tratado apenas da serialização com Java, em sua implemetação básica, alguns problemas e soluções, com códigos para exemplificação.

**2. SERIALIZAÇÃO COM JAVA**

Serializar um objeto, em Java, só é possível caso sua classe esteja marcada como serializável. Essa marcação é feita com a simples implementação da interface “java.io.Serializable”. Essa interface não possui métodos para se implementar e nem atributos. Ainda não existiam as anotações no Java 1.1, versão que trouxe pela primeira vez a serialização ao Java, e desta maneira a implementação da interface era a melhor maneira de marcar uma classe.

A herança entre classes é naturalmente afetada pela serialização. Já que quando uma classe implementa a interface “Serializable”, toda classe que dela estender também estará implicitamente marcada como serializável

.

Para exemplo, será criada uma classe “pojo.Fruta” que contém dois atributos: nome e cor. No código exemplo (Listagem 1), pode-se notar a classe “Fruta” marcada para a serialização:

Listagem 1 - Marcando uma classe como serializável.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | import java.io.Serializable;    public class Fruta implements Serializable{      private String nome;      private String cor;        public Fruta(String nome, String cor) {          this.nome = nome;          this.cor = cor;      }        public String getCor() { return cor; }      public String getNome() { return nome; }      public void setCor(String cor) { this.cor = cor; }      public void setNome(String nome) { this.nome = nome; }  } |

Pode-se dizer que a implementação do processo de serialização e deserialização simples de um objeto, não possui um código difícil de se escrever. Isto porquê duas classes fazem a maior parte do trabalho, de maneira que com poucas linhas de código pode-se transformar um objeto em bytes e vice-versa. Essas duas classes são “java.io.ObjectInputStream” que recebe bytes para criar um objeto e “java.io.ObjectOutputStream” que cria uma cadeia de bytes a partir de um objeto.

**3. SERIALIZANDO E DESERIALIZANDO O ESTADO DE UM OBJETO**

Como exemplo, será persistido um objeto em disco. Para isto, além das classes “java.io.ObjectInputStream” e “java.io.ObjectOutputStream”, serão utilizadas duas outras que trabalham com arquivos, que são “java.io.FileInputStream” e “java.io.FileOutputStream”, que seguem a mesma lógica de InputStream e OutputStream, só que agora relacionado ao sistema de arquivos e não aos objetos.

Para serialização, deve-se criar um novo “java.io.FileOutputStream”, com o caminho (“path”) do arquivo onde se deseja armazenar o estado do objeto. Também deve ser criado um novo “java.io.ObjectOutputStream” que está ligado ao arquivo. E por último deve-se executar a escrita do objeto alvo, para dentro do objeto de stream que por sua vez está ligado ao arquivo destino. Após executados esses passos, é possível encontrar no sistema de arquivos um novo arquivo que contém os bytes correspondentes à descrição do estado do objeto. Na Listagem 2, encontra-se o código exemplo.

Listagem 2 - Serialização de um objeto.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | import java.io.FileOutputStream;  import java.io.ObjectOutputStream;    public class Serializador {        public Serializador() {    }        public void serializar(String path, Object obj) throws Exception {              FileOutputStream outFile = new FileOutputStream(path);      ObjectOutputStream s = new ObjectOutputStream(outFile);      s.writeObject(obj);      s.close();      }  } |

E para a deserialização, o processo é extremamente parecido com a serialização. Cria-se um novo “java.io.FileInputStream” relacionado ao arquivo onde o estado do objeto encontra-se persistido. Deve-se criar também um novo “java.io.ObjectInputStream” que estará vinculado ao arquivo de origem. Por último, executa-se a leitura do arquivo para a restauração do estado. Na Listagem 3 encontra-se o exemplo do código.

Listagem 3 - Deserializaçãode um objeto.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | import java.io.FileInputStream;  import java.io.ObjectInputStream;    public class Deserializador {        public Deserializador() {    }        public Object deserializar(String path) throws Exception {      FileInputStream inFile = new FileInputStream(path);      ObjectInputStream d = new ObjectInputStream(inFile);      Object o = d.readObject();      d.close();      return o;      }  } |

A Listagem 4, contém o exemplo da execução da serialização e em seguida da deserialização.

Listagem 4 - Serializando e Deserializando

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | import operacoes.Deserializador;  import operacoes.Serializador;  import pojo.Fruta;    public class Principal {      public static void main(String args[]){         //serializa         Serializador s = new Serializador();         Fruta fruta = new Fruta("Maça", "Vermelha");          try {              s.serializar("/home/gustavo/fruta", fruta);          } catch (Exception ex) {              System.err.println("Falha ao serializar! - " + ex.toString());          }            //deserializa          Deserializador d = new Deserializador();          fruta = null;          try {              fruta = (Fruta) d.deserializar("/home/gustavo/fruta");          } catch (Exception ex) {              System.err.println("Falha ao deserializar! - " + ex.toString());          }          System.out.println(fruta.getNome() + " - " + fruta.getCor());      }  } |

Na Listagem 5, encontra-se um exemplo de serialização e deserialização de um ArrayList da classe Fruta. É apenas mais um exemplo que demonstra a capacidade de se persistir qualquer objeto que seja serializável. Essa serialização só foi possível porque a classe “ArrayList” implementa a interface “Serializable” por padrão.

Listagem 5 - Serialização de ArrayList.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | //serialização e deserialização com ArrayList          ArrayList<Fruta> frutas = new ArrayList<Fruta>();          Fruta f1 = new Fruta("Laranja", "Amarela");          Fruta f2 = new Fruta("Abacate", "Verde");          Fruta f3 = new Fruta("Morango", "Vermelho");          frutas.add(f1);          frutas.add(f2);          frutas.add(f3);            try {              s.serializar("/home/gustavo/frutas", frutas);              frutas = null;              frutas = (ArrayList<Fruta>) d.deserializar("/home/gustavo/frutas");              for(Fruta f : frutas){                  System.out.println("ArrayList: " + f.getNome() + " - " + f.getCor());              }          } catch (Exception ex) {              System.err.println("Falha ao serializar ou deserializar! - " +                                 ex.toString());          } |

**4. PARTICULARIDADES DA SERIALIZAÇÃO**

Como dito, a serialização de um objeto em Java pode ser algo fácil de se fazer. Mas existem certas informações que são importantes e sabê-las pode evitar vários problemas.

Um dos problemas é que nem tudo pode ser serializado, como objetos das classes “Thread” e “Socket” e atributos estáticos. Os atributos estáticos estão vinculados à classe e não ao objeto. Assim quando se trata do estado do objeto, os campos estáticos não são considerados. Em relação às Threads, suas instâncias só são úteis enquanto estiverem em execução na JVM, e podem ser paradas/destruídas e iniciadas com instruções de sua classe. Não existe algo no estado destas instâncias que seja útil para se persistir, e por isso não há sentido em persistir tais instâncias para restaurar posteriormente.

Mas e se existir alguma classe em que seja necessário persisti-la, e esta possui uma instância da classe “Thread”? Neste caso, utilizando um recurso pode-se persistir a classe inteira, menos a instância thread. Este recurso, refere-se em indicar que tal instância thread é um objeto “Transient”. Na Listagem 6 encontra-se um exemplo da implementação com uso de “Transient”.

Listagem 6 - Uso do Transient

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | import java.io.Serializable;    public class ImplementacaoTransient implements Serializable, Runnable{        transient private Thread thread;      //marcado com "transient" para que sua instância não seja serializada        //Demais valores de estado serão serializados      private int valorParaPersistir;        public ImplementacaoTransient(int valorParaPersistir){          this.valorParaPersistir = valorParaPersistir;          thread = new Thread(this);          thread.start();      }        public void run()      {          while(true){              //código a ser executado pela Thread          }      }  } |

Vale ressaltar que, o que é serializado é apenas o estado do objeto, e não os métodos e a classe. No exemplo acima, apenas a instância do objeto “thread” e do objeto “valorParaPersistir” compunham o estado do objeto. Assim, evita-se que o sistema de serialização tente serializar um objeto não-serializável e gere erros. Após a deserialização de um objeto da classe “ImplementacaoTransient” este estará totalmente funcional.

Um outro problema que pode vir a ocorrer, se refere ao versionamento das classes. Toda classe possui um atributo oculto chamado “serialVersionUID” com um valor gerado automaticamente. Objetos criados levam consigo o “serialVersionUID” de sua classe, que indica sua versão. Toda vez que uma classe sofre alterações em seu código, um novo “serialVersionUID” é gerado. O problema com a serialização surge quando serializa-se o estado de um objeto e posteriormente tenta-se restaurar o estado para a aplicação, mas houveram mudanças no código da classe. Neste caso, uma exceção “java.io.InvalidClassException” é disparada, pois de fato, o estado do objeto corresponde a uma versão de classe do qual foi criado que já não existe mais. Portanto, isso só ocorre para evitar erros de programação, em que tenta-se restaurar o estado de um objeto para uma classe que possui versão diferente da qual o objeto foi serializado inicialmente. E isso pode significar diferentes versões de uma mesma classe “Fruta” por exemplo, ou diferentes versões por classes serem totalmente difere

|  |  |
| --- | --- |
| 1 | serialver pojo.Fruta |

A serialização acaba inflingindo alguns modelos de segurança em seu procedimento padrão, e criando então um novo problema. Quando um estado de objeto é persistido no sistema de arquivos ou enviado por rede, o fluxo binário pode ser detectado, analisado e obtidas informações sobre a classe e o estado do objeto. Por exemplo, se utilizado um editor de texto qualquer para abrir o arquivo persistido do estado do objeto da classe “Fruta” dos exemplos anteriores, é comum os atributos e outras informações aparecerem como um simples texto, total ou parcilamente legíveis, assim como demonstrado na Listagem 9.

Listagem 9 - Exemplo de leitura de arquivo binário de um estado de objeto persistido.

|  |  |
| --- | --- |
| 1  2 | ��##sr#  pojo.Fruta#######{###L##cort##Ljava/lang/String;L##nomeq#~##xpt##Vermelhat##Maça |

Com uma ferramenta menos simples seria possível aumentar a quantidade de informações legíveis, coletando uma maior quantia de dados.

No entando, o processo de serialização permite que haja uma certa personalização da serialização e deserialização de um objeto. Isso deve-se a possibilidade da sobrescrição dos métodos “writeObject” e “readObject” dentro da classe a ser serializada, como a classe exemplar “pojo.Fruta”. Esta sobrescrita permite um controle total sobre a serialização, pois as classes “java.io.ObjectOutputStream” e “java.io.ObjectInputStream”, ao serializar o estado de um objeto, primeiramente verificam se esse método foi sobrescrito na classe alvo e assim o utilizam, senão, será utilizado o protocolo padrão de serialização.

Realizando a sobrescrição dos métodos é possível criptografar os dados na serialização e descriptografar os dados na deserialização, por meio de algum algoritmo que o programador venha implementar. Assim os dados podem ser persistidos ou transmitidos por rede, sem que seja possível ler as partes mais importantes do estado de um objeto. Na listagem 10, a classe “pojo.Fruta” foi modificada dando origem a classe “pojo.FrutaSobrescrita” que possui adicionamente a sobrescrição dos métodos “writeObject” e “readObject”. Como este é apenas um exemplo, não foi escrito um verdadeiro algoritmo de criptografia, mas sim apenas uma troca no valor da String do campo “cor”, identificando que houveram alterações tanto no processo de serialização quanto no de deserialização. Na listagem 11 encontra-se a execução da serialização.

Listagem 10 - Sobrescrita dos métodos “writeObject” e “readObject”.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36 | import java.io.Serializable;    public class FrutaSobrescrita implements Serializable{      private String nome;      private String cor;      static final long serialVersionUID = 123L; //indica a versao da classe        public FrutaSobrescrita(String nome, String cor) {          this.nome = nome;          this.cor = cor;      }      public String getCor() { return cor; }      public String getNome() { return nome; }      public void setCor(String cor) { this.cor = cor; }      public void setNome(String nome) { this.nome = nome; }        private void writeObject(java.io.ObjectOutputStream stream) throws java.io.IOException      {          // criptografar os dados antes de liberar o stream          // EXEMPLO: setCor(criptografar(getCor()));          setCor(getCor()+ " > CRIPTOGRAFADO"); // apenas concatenação de string para indicar alteração            // liberar o stream          stream.defaultWriteObject();      }        private void readObject(java.io.ObjectInputStream stream) throws java.io.IOException, ClassNotFoundException      {          // liberar stream          stream.defaultReadObject();            // descriptografar os dados          // EXEMPLO: setCor(descriptografar(getCor()));          setCor(getCor()+ " > DESCRIPTOGRAFADO"); // apenas contatenação de string para indicar alteração      }  } |

Listagem 11 - Execução da serialização da classe “pojo.FrutaSobrescrita”.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | //serializa        Serializador s = new Serializador();        FrutaSobrescrita frutaSobrescrita = new FrutaSobrescrita("Maça", "Vermelha");         try {             s.serializar("/home/gustavo/frutaSobrescrita", frutaSobrescrita);         } catch (Exception ex) {             System.err.println("Falha ao serializar! - " + ex.toString());         }           //deserializa        Deserializador d = new Deserializador();         frutaSobrescrita = null;         try {             frutaSobrescrita = (FrutaSobrescrita) d.deserializar("/home/gustavo/frutaSobrescrita");         } catch (Exception ex) {             System.err.println("Falha ao deserializar! - " + ex.toString());         }         System.out.println("Sobrescrita: " + frutaSobrescrita.getNome() + " - " + frutaSobrescrita.getCor()); |

**5. CONCLUSÃO**

A serialização é um procedimento de simples execução por parte do programador. Com pouco tempo de codificação é possível implementar a persistencia de um estado de objeto para o sistema de arquivos ou de transmissão por rede. É possível reproduzir os exemplos criados neste artigo, em situações até mesmo mais elaboradas. A apresentação de alguns problemas e soluções trazidas por particularidades, abre o entendimento de que a serialização pode deixar de ser simples conforme for aplicada em diferentes contextos, e que ao mesmo tempo é muito flexível ao permitir ao programador implementar diferentes soluções.

[Assista ao vídeo Serialização/Deserialização de objetos com YAML](http://www.devmedia.com.br/serializacao-deserializacao-de-objetos-com-yaml-parte-1/8399)

**6. BIBLIOGRAFIA**

ORACLE - **java.io Interface Serializable** - Disponível em: <http://docs.oracle.com/javase/1.4.2/docs/api/java/io/Serializable.html>Acesso em: 11/03/2012

ORACLE - **Discover de Secrets of Java Serialization API** - Disponível em: <http://java.sun.com/developer/technicalArticles/Programming/serialization/> Acesso em: 11/03/2012

NEWARD, Ted - **5 coisas que você não sabia sobre... Serialização de Objetos Java** - Disponível em: <http://www.ibm.com/developerworks/br/library/j-5things1/index.html> Acesso em: 11/03/2012

DOWNS, Andrew - **Java Serialization** - Disponível em: <http://www.mactech.com/articles/mactech/Vol.14/14.04/JavaSerialization/index.html> Acesso em: 11/03/2012

PREISSLER, Luciano E.; GEYER, Cláudio F. R. - **Serialização em Java** - Disponível em: <http://www.inf.ufrgs.br/gppd/disc/cmp167/trabalhos/sem99-1/T1/luciano/final.htm> Acesso em: 11/03/2012

[[](http://www.linhadecodigo.com.br/colaborador.aspx?id=354)](http://www.linhadecodigo.com.br/colaborador.aspx?id=354)

[Everton Coimbra de Araújo - Desde 1987 atua na área de treinamento e desenvolvimento. Como Mestre em Ciência da Computação, é professor da UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ, Campus Medianeira, onde leciona disciplinas relacionadas ao desenvolvimento de aplicações web, com Java e .NET.   
  
É autor dos livros Desenvolvimento para WEB com Java, Orientação a Objetos com Java - Simples, Fácil e Eficiente, Algoritmo - Fundamento e Prática em sua terceira edição e dos livros Delphi - Implementação e Técnicas para Ambientes Virtuais e C++ Builder - Implementação e Técnicas para Ambientes Virtuais. Todos pela VisualBooks. Pode ser contactado através do e-mail everton@utfpr.edu.br ou evertoncoimbra@gmail.com.](http://www.linhadecodigo.com.br/colaborador.aspx?id=354)

 85 3 0 4

**Leia também**

* [Conhecendo a interface Map do Java](http://www.linhadecodigo.com.br/artigo/3670/conhecendo-a-interface-map-do-java.aspx)*Java*
* [Trabalhando com a Interface Set no Java](http://www.linhadecodigo.com.br/artigo/3669/trabalhando-com-a-interface-set-no-java.aspx)*Java*
* [Classes Wrappers em Java](http://www.linhadecodigo.com.br/artigo/3667/classes-wrappers-em-java.aspx)*Java*
* [ComboBoxModel em Java](http://www.linhadecodigo.com.br/artigo/3664/comboboxmodel-em-java.aspx)*Java*
* [Reduzindo a Mutabilidade em Java](http://www.linhadecodigo.com.br/artigo/3659/reduzindo-a-mutabilidade-em-java.aspx)*Java*

Publicidade

[](http://qconsp.com/?utm_source=devmedia&utm_medium=banner&utm_campaign=devmedia)

Linha de Código[Anuncie](http://www.web-03.net/linhadecodigo.asp)|[Fale Conosco](mailto:colabore@linhadecodigo.com.br)|[Publique](http://www.linhadecodigo.com.br/publique.aspx)

Copyright 2016 - todos os direitos reservados para [**Web03**](http://www.web-03.net/)

|  |  |
| --- | --- |
| http://www.linhadecodigo.com.br/artigos/img_artigos/EvertonCoimbraAraujo/Serializazao_Objetos/imagem001.jpg | Neste artigo apresento o Gustavo Rafael Valiati, que pode ser contatado pelo e-mail gustavovaliati@gmail.com. O Gustavo foi meu aluno em algumas disciplinas de sua gradução e está concluíndo o último período do curso de Tecnologia em Análise e Desenvolvimento de Sistemas da UTFPR campus Medianeira, neste primeiro semestre de 2012. |

**1. INTRODUÇÃO**

Nem sempre se deseja ter objetos ou estrutura de dados de um aplicativo, com um ciclo de vida que dura no máximo o tempo em que for armazenado em memória RAM. Ou seja, algumas vezes deseja-se guardá-los por um tempo indeterminado, sendo possível recuperá-los posteriormente. E neste caso, o termo “persistência” tem seu conceito definido.

Persistir dados, requer que estes possam ser serializáveis. A serialização possibilitará capturar o estado do objeto ou a estrutura de dados, e transformar em uma cadeia de bytes quando for necessário. Torna-se possível também, recuperar os bytes persistidos para fazer o processo inverso, e ter os dados de volta para a aplicação em execução.

Desta maneira, qualquer forma de dados que esteja sendo trabalhada na aplicação, e esta puder ser serializada, poderá ser passada para o meio físico, e ter o processo revertido.

A serialização não se limita apenas em possibilitar a gravação do dado em disco, mas permite que o mesmo seja transmitido por rede. Isto porquê a serialização cria um stream de bytes, que é requisito para transmissão de dados por rede.

Diversas linguagens possuem suporte à serialização, como: Java, C, C++, Python, PHP, .NET, entre outras. Neste artigo, será tratado apenas da serialização com Java, em sua implemetação básica, alguns problemas e soluções, com códigos para exemplificação.

**2. SERIALIZAÇÃO COM JAVA**

Serializar um objeto, em Java, só é possível caso sua classe esteja marcada como serializável. Essa marcação é feita com a simples implementação da interface “java.io.Serializable”. Essa interface não possui métodos para se implementar e nem atributos. Ainda não existiam as anotações no Java 1.1, versão que trouxe pela primeira vez a serialização ao Java, e desta maneira a implementação da interface era a melhor maneira de marcar uma classe.

A herança entre classes é naturalmente afetada pela serialização. Já que quando uma classe implementa a interface “Serializable”, toda classe que dela estender também estará implicitamente marcada como serializável

.

Para exemplo, será criada uma classe “pojo.Fruta” que contém dois atributos: nome e cor. No código exemplo (Listagem 1), pode-se notar a classe “Fruta” marcada para a serialização:

Listagem 1 - Marcando uma classe como serializável.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | import java.io.Serializable;    public class Fruta implements Serializable{      private String nome;      private String cor;        public Fruta(String nome, String cor) {          this.nome = nome;          this.cor = cor;      }        public String getCor() { return cor; }      public String getNome() { return nome; }      public void setCor(String cor) { this.cor = cor; }      public void setNome(String nome) { this.nome = nome; }  } |

Pode-se dizer que a implementação do processo de serialização e deserialização simples de um objeto, não possui um código difícil de se escrever. Isto porquê duas classes fazem a maior parte do trabalho, de maneira que com poucas linhas de código pode-se transformar um objeto em bytes e vice-versa. Essas duas classes são “java.io.ObjectInputStream” que recebe bytes para criar um objeto e “java.io.ObjectOutputStream” que cria uma cadeia de bytes a partir de um objeto.

**3. SERIALIZANDO E DESERIALIZANDO O ESTADO DE UM OBJETO**

Como exemplo, será persistido um objeto em disco. Para isto, além das classes “java.io.ObjectInputStream” e “java.io.ObjectOutputStream”, serão utilizadas duas outras que trabalham com arquivos, que são “java.io.FileInputStream” e “java.io.FileOutputStream”, que seguem a mesma lógica de InputStream e OutputStream, só que agora relacionado ao sistema de arquivos e não aos objetos.

Para serialização, deve-se criar um novo “java.io.FileOutputStream”, com o caminho (“path”) do arquivo onde se deseja armazenar o estado do objeto. Também deve ser criado um novo “java.io.ObjectOutputStream” que está ligado ao arquivo. E por último deve-se executar a escrita do objeto alvo, para dentro do objeto de stream que por sua vez está ligado ao arquivo destino. Após executados esses passos, é possível encontrar no sistema de arquivos um novo arquivo que contém os bytes correspondentes à descrição do estado do objeto. Na Listagem 2, encontra-se o código exemplo.

Listagem 2 - Serialização de um objeto.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | import java.io.FileOutputStream;  import java.io.ObjectOutputStream;    public class Serializador {        public Serializador() {    }        public void serializar(String path, Object obj) throws Exception {              FileOutputStream outFile = new FileOutputStream(path);      ObjectOutputStream s = new ObjectOutputStream(outFile);      s.writeObject(obj);      s.close();      }  } |

E para a deserialização, o processo é extremamente parecido com a serialização. Cria-se um novo “java.io.FileInputStream” relacionado ao arquivo onde o estado do objeto encontra-se persistido. Deve-se criar também um novo “java.io.ObjectInputStream” que estará vinculado ao arquivo de origem. Por último, executa-se a leitura do arquivo para a restauração do estado. Na Listagem 3 encontra-se o exemplo do código.

Listagem 3 - Deserializaçãode um objeto.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | import java.io.FileInputStream;  import java.io.ObjectInputStream;    public class Deserializador {        public Deserializador() {    }        public Object deserializar(String path) throws Exception {      FileInputStream inFile = new FileInputStream(path);      ObjectInputStream d = new ObjectInputStream(inFile);      Object o = d.readObject();      d.close();      return o;      }  } |

A Listagem 4, contém o exemplo da execução da serialização e em seguida da deserialização.

Listagem 4 - Serializando e Deserializando

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | import operacoes.Deserializador;  import operacoes.Serializador;  import pojo.Fruta;    public class Principal {      public static void main(String args[]){         //serializa         Serializador s = new Serializador();         Fruta fruta = new Fruta("Maça", "Vermelha");          try {              s.serializar("/home/gustavo/fruta", fruta);          } catch (Exception ex) {              System.err.println("Falha ao serializar! - " + ex.toString());          }            //deserializa          Deserializador d = new Deserializador();          fruta = null;          try {              fruta = (Fruta) d.deserializar("/home/gustavo/fruta");          } catch (Exception ex) {              System.err.println("Falha ao deserializar! - " + ex.toString());          }          System.out.println(fruta.getNome() + " - " + fruta.getCor());      }  } |

Na Listagem 5, encontra-se um exemplo de serialização e deserialização de um ArrayList da classe Fruta. É apenas mais um exemplo que demonstra a capacidade de se persistir qualquer objeto que seja serializável. Essa serialização só foi possível porque a classe “ArrayList” implementa a interface “Serializable” por padrão.

Listagem 5 - Serialização de ArrayList.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | //serialização e deserialização com ArrayList          ArrayList<Fruta> frutas = new ArrayList<Fruta>();          Fruta f1 = new Fruta("Laranja", "Amarela");          Fruta f2 = new Fruta("Abacate", "Verde");          Fruta f3 = new Fruta("Morango", "Vermelho");          frutas.add(f1);          frutas.add(f2);          frutas.add(f3);            try {              s.serializar("/home/gustavo/frutas", frutas);              frutas = null;              frutas = (ArrayList<Fruta>) d.deserializar("/home/gustavo/frutas");              for(Fruta f : frutas){                  System.out.println("ArrayList: " + f.getNome() + " - " + f.getCor());              }          } catch (Exception ex) {              System.err.println("Falha ao serializar ou deserializar! - " +                                 ex.toString());          } |

**4. PARTICULARIDADES DA SERIALIZAÇÃO**

Como dito, a serialização de um objeto em Java pode ser algo fácil de se fazer. Mas existem certas informações que são importantes e sabê-las pode evitar vários problemas.

Um dos problemas é que nem tudo pode ser serializado, como objetos das classes “Thread” e “Socket” e atributos estáticos. Os atributos estáticos estão vinculados à classe e não ao objeto. Assim quando se trata do estado do objeto, os campos estáticos não são considerados. Em relação às Threads, suas instâncias só são úteis enquanto estiverem em execução na JVM, e podem ser paradas/destruídas e iniciadas com instruções de sua classe. Não existe algo no estado destas instâncias que seja útil para se persistir, e por isso não há sentido em persistir tais instâncias para restaurar posteriormente.

Mas e se existir alguma classe em que seja necessário persisti-la, e esta possui uma instância da classe “Thread”? Neste caso, utilizando um recurso pode-se persistir a classe inteira, menos a instância thread. Este recurso, refere-se em indicar que tal instância thread é um objeto “Transient”. Na Listagem 6 encontra-se um exemplo da implementação com uso de “Transient”.

Listagem 6 - Uso do Transient

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | import java.io.Serializable;    public class ImplementacaoTransient implements Serializable, Runnable{        transient private Thread thread;      //marcado com "transient" para que sua instância não seja serializada        //Demais valores de estado serão serializados      private int valorParaPersistir;        public ImplementacaoTransient(int valorParaPersistir){          this.valorParaPersistir = valorParaPersistir;          thread = new Thread(this);          thread.start();      }        public void run()      {          while(true){              //código a ser executado pela Thread          }      }  } |

Vale ressaltar que, o que é serializado é apenas o estado do objeto, e não os métodos e a classe. No exemplo acima, apenas a instância do objeto “thread” e do objeto “valorParaPersistir” compunham o estado do objeto. Assim, evita-se que o sistema de serialização tente serializar um objeto não-serializável e gere erros. Após a deserialização de um objeto da classe “ImplementacaoTransient” este estará totalmente funcional.

Um outro problema que pode vir a ocorrer, se refere ao versionamento das classes. Toda classe possui um atributo oculto chamado “serialVersionUID” com um valor gerado automaticamente. Objetos criados levam consigo o “serialVersionUID” de sua classe, que indica sua versão. Toda vez que uma classe sofre alterações em seu código, um novo “serialVersionUID” é gerado. O problema com a serialização surge quando serializa-se o estado de um objeto e posteriormente tenta-se restaurar o estado para a aplicação, mas houveram mudanças no código da classe. Neste caso, uma exceção “java.io.InvalidClassException” é disparada, pois de fato, o estado do objeto corresponde a uma versão de classe do qual foi criado que já não existe mais. Portanto, isso só ocorre para evitar erros de programação, em que tenta-se restaurar o estado de um objeto para uma classe que possui versão diferente da qual o objeto foi serializado inicialmente. E isso pode significar diferentes versões de uma mesma classe “Fruta” por exemplo, ou diferentes versões por classes serem totalmente diferentes como: o estado de um objeto da classe “Fruta” ser restaurado para classe “Carro”.

Mas quando é necessário, conscientemente, recuperar o estado de um objeto, mesmo que houveram alterações na classe do qual foi gerado, é possível controlar manualmente o “serialVersionUID”. Para isto, basta declarar explicitamente na forma de atributo estático e final, um valor Long qualquer para corresponder à versão da classe. E assim, mesmo que houverem modificações na classe, a versão permenecerá a mesma, evitando as exceções da incompatiblidade de classes. No entanto, vale ressaltar que o versionamento não é um impecílio para o programador, mas sim muito importante em casos que deseja-se manter um controle sobre a compatibilidade de código na aplicação, definindo versões para classes No código da Listagem 7 encontra-se o exemplo do controle manual do “serialVersionUID”.

Listagem 7 - Uso manual do serialVersionUID

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | import java.io.Serializable;    public class Fruta implements Serializable{      private String nome;      private String cor;      static final long serialVersionUID = 123L; //indica a versao da classe        public Fruta(String nome, String cor) {          this.nome = nome;          this.cor = cor;      }      public String getCor() { return cor; }      public String getNome() { return nome; }      public void setCor(String cor) { this.cor = cor; }      public void setNome(String nome) { this.nome = nome; }  } |

O JDK fornece uma ferramenta que permite visualizar a versão de qualquer classe Java. Basta no terminal do sistema operacional utilizar o comando “serialver” seguido da classe qualificada, conforme exemplo da Listagem 8.

Listagem 8 - Verificação da versão de classe

|  |  |
| --- | --- |
| 1 | serialver pojo.Fruta |

A serialização acaba inflingindo alguns modelos de segurança em seu procedimento padrão, e criando então um novo problema. Quando um estado de objeto é persistido no sistema de arquivos ou enviado por rede, o fluxo binário pode ser detectado, analisado e obtidas informações sobre a classe e o estado do objeto. Por exemplo, se utilizado um editor de texto qualquer para abrir o arquivo persistido do estado do objeto da classe “Fruta” dos exemplos anteriores, é comum os atributos e outras informações aparecerem como um simples texto, total ou parcilamente legíveis, assim como demonstrado na Listagem 9.

Listagem 9 - Exemplo de leitura de arquivo binário de um estado de objeto persistido.

|  |  |
| --- | --- |
| 1  2 | ��##sr#  pojo.Fruta#######{###L##cort##Ljava/lang/String;L##nomeq#~##xpt##Vermelhat##Maça |

Com uma ferramenta menos simples seria possível aumentar a quantidade de informações legíveis, coletando uma maior quantia de dados.

No entando, o processo de serialização permite que haja uma certa personalização da serialização e deserialização de um objeto. Isso deve-se a possibilidade da sobrescrição dos métodos “writeObject” e “readObject” dentro da classe a ser serializada, como a classe exemplar “pojo.Fruta”. Esta sobrescrita permite um controle total sobre a serialização, pois as classes “java.io.ObjectOutputStream” e “java.io.ObjectInputStream”, ao serializar o estado de um objeto, primeiramente verificam se esse método foi sobrescrito na classe alvo e assim o utilizam, senão, será utilizado o protocolo padrão de serialização.

Realizando a sobrescrição dos métodos é possível criptografar os dados na serialização e descriptografar os dados na deserialização, por meio de algum algoritmo que o programador venha implementar. Assim os dados podem ser persistidos ou transmitidos por rede, sem que seja possível ler as partes mais importantes do estado de um objeto. Na listagem 10, a classe “pojo.Fruta” foi modificada dando origem a classe “pojo.FrutaSobrescrita” que possui adicionamente a sobrescrição dos métodos “writeObject” e “readObject”. Como este é apenas um exemplo, não foi escrito um verdadeiro algoritmo de criptografia, mas sim apenas uma troca no valor da String do campo “cor”, identificando que houveram alterações tanto no processo de serialização quanto no de deserialização. Na listagem 11 encontra-se a execução da serialização.

Listagem 10 - Sobrescrita dos métodos “writeObject” e “readObject”.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36 | import java.io.Serializable;    public class FrutaSobrescrita implements Serializable{      private String nome;      private String cor;      static final long serialVersionUID = 123L; //indica a versao da classe        public FrutaSobrescrita(String nome, String cor) {          this.nome = nome;          this.cor = cor;      }      public String getCor() { return cor; }      public String getNome() { return nome; }      public void setCor(String cor) { this.cor = cor; }      public void setNome(String nome) { this.nome = nome; }        private void writeObject(java.io.ObjectOutputStream stream) throws java.io.IOException      {          // criptografar os dados antes de liberar o stream          // EXEMPLO: setCor(criptografar(getCor()));          setCor(getCor()+ " > CRIPTOGRAFADO"); // apenas concatenação de string para indicar alteração            // liberar o stream          stream.defaultWriteObject();      }        private void readObject(java.io.ObjectInputStream stream) throws java.io.IOException, ClassNotFoundException      {          // liberar stream          stream.defaultReadObject();            // descriptografar os dados          // EXEMPLO: setCor(descriptografar(getCor()));          setCor(getCor()+ " > DESCRIPTOGRAFADO"); // apenas contatenação de string para indicar alteração      }  } |

Listagem 11 - Execução da serialização da classe “pojo.FrutaSobrescrita”.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | //serializa        Serializador s = new Serializador();        FrutaSobrescrita frutaSobrescrita = new FrutaSobrescrita("Maça", "Vermelha");         try {             s.serializar("/home/gustavo/frutaSobrescrita", frutaSobrescrita);         } catch (Exception ex) {             System.err.println("Falha ao serializar! - " + ex.toString());         }           //deserializa        Deserializador d = new Deserializador();         frutaSobrescrita = null;         try {             frutaSobrescrita = (FrutaSobrescrita) d.deserializar("/home/gustavo/frutaSobrescrita");         } catch (Exception ex) {             System.err.println("Falha ao deserializar! - " + ex.toString());         }         System.out.println("Sobrescrita: " + frutaSobrescrita.getNome() + " - " + frutaSobrescrita.getCor()); |

**5. CONCLUSÃO**

A serialização é um procedimento de simples execução por parte do programador. Com pouco tempo de codificação é possível implementar a persistencia de um estado de objeto para o sistema de arquivos ou de transmissão por rede. É possível reproduzir os exemplos criados neste artigo, em situações até mesmo mais elaboradas. A apresentação de alguns problemas e soluções trazidas por particularidades, abre o entendimento de que a serialização pode deixar de ser simples conforme for aplicada em diferentes contextos, e que ao mesmo tempo é muito flexível ao permitir ao programador implementar diferentes soluções.

[Assista ao vídeo Serialização/Deserialização de objetos com YAML](http://www.devmedia.com.br/serializacao-deserializacao-de-objetos-com-yaml-parte-1/8399)

**6. BIBLIOGRAFIA**

ORACLE - **java.io Interface Serializable** - Disponível em: <http://docs.oracle.com/javase/1.4.2/docs/api/java/io/Serializable.html>Acesso em: 11/03/2012

ORACLE - **Discover de Secrets of Java Serialization API** - Disponível em: <http://java.sun.com/developer/technicalArticles/Programming/serialization/> Acesso em: 11/03/2012

NEWARD, Ted - **5 coisas que você não sabia sobre... Serialização de Objetos Java** - Disponível em: <http://www.ibm.com/developerworks/br/library/j-5things1/index.html> Acesso em: 11/03/2012

DOWNS, Andrew - **Java Serialization** - Disponível em: <http://www.mactech.com/articles/mactech/Vol.14/14.04/JavaSerialization/index.html> Acesso em: 11/03/2012

PREISSLER, Luciano E.; GEYER, Cláudio F. R. - **Serialização em Java** - Disponível em: <http://www.inf.ufrgs.br/gppd/disc/cmp167/trabalhos/sem99-1/T1/luciano/final.htm> Acesso em: 11/03/2012

[[](http://www.linhadecodigo.com.br/colaborador.aspx?id=354)](http://www.linhadecodigo.com.br/colaborador.aspx?id=354)

[Everton Coimbra de Araújo - Desde 1987 atua na área de treinamento e desenvolvimento. Como Mestre em Ciência da Computação, é professor da UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ, Campus Medianeira, onde leciona disciplinas relacionadas ao desenvolvimento de aplicações web, com Java e .NET.   
  
É autor dos livros Desenvolvimento para WEB com Java, Orientação a Objetos com Java - Simples, Fácil e Eficiente, Algoritmo - Fundamento e Prática em sua terceira edição e dos livros Delphi - Implementação e Técnicas para Ambientes Virtuais e C++ Builder - Implementação e Técnicas para Ambientes Virtuais. Todos pela VisualBooks. Pode ser contactado através do e-mail everton@utfpr.edu.br ou evertoncoimbra@gmail.com.](http://www.linhadecodigo.com.br/colaborador.aspx?id=354)

 85 3 0 4

**Leia também**

* [Conhecendo a interface Map do Java](http://www.linhadecodigo.com.br/artigo/3670/conhecendo-a-interface-map-do-java.aspx)*Java*
* [Trabalhando com a Interface Set no Java](http://www.linhadecodigo.com.br/artigo/3669/trabalhando-com-a-interface-set-no-java.aspx)*Java*
* [Classes Wrappers em Java](http://www.linhadecodigo.com.br/artigo/3667/classes-wrappers-em-java.aspx)*Java*
* [ComboBoxModel em Java](http://www.linhadecodigo.com.br/artigo/3664/comboboxmodel-em-java.aspx)*Java*
* [Reduzindo a Mutabilidade em Java](http://www.linhadecodigo.com.br/artigo/3659/reduzindo-a-mutabilidade-em-java.aspx)*Java*

Publicidade

[](http://qconsp.com/?utm_source=devmedia&utm_medium=banner&utm_campaign=devmedia)

Linha de Código[Anuncie](http://www.web-03.net/linhadecodigo.asp)|[Fale Conosco](mailto:colabore@linhadecodigo.com.br)|[Publique](http://www.linhadecodigo.com.br/publique.aspx)

|  |  |
| --- | --- |
| http://www.linhadecodigo.com.br/artigos/img_artigos/EvertonCoimbraAraujo/Serializazao_Objetos/imagem001.jpg | Neste artigo apresento o Gustavo Rafael Valiati, que pode ser contatado pelo e-mail gustavovaliati@gmail.com. O Gustavo foi meu aluno em algumas disciplinas de sua gradução e está concluíndo o último período do curso de Tecnologia em Análise e Desenvolvimento de Sistemas da UTFPR campus Medianeira, neste primeiro semestre de 2012. |

**1. INTRODUÇÃO**

Nem sempre se deseja ter objetos ou estrutura de dados de um aplicativo, com um ciclo de vida que dura no máximo o tempo em que for armazenado em memória RAM. Ou seja, algumas vezes deseja-se guardá-los por um tempo indeterminado, sendo possível recuperá-los posteriormente. E neste caso, o termo “persistência” tem seu conceito definido.

Persistir dados, requer que estes possam ser serializáveis. A serialização possibilitará capturar o estado do objeto ou a estrutura de dados, e transformar em uma cadeia de bytes quando for necessário. Torna-se possível também, recuperar os bytes persistidos para fazer o processo inverso, e ter os dados de volta para a aplicação em execução.

Desta maneira, qualquer forma de dados que esteja sendo trabalhada na aplicação, e esta puder ser serializada, poderá ser passada para o meio físico, e ter o processo revertido.

A serialização não se limita apenas em possibilitar a gravação do dado em disco, mas permite que o mesmo seja transmitido por rede. Isto porquê a serialização cria um stream de bytes, que é requisito para transmissão de dados por rede.

Diversas linguagens possuem suporte à serialização, como: Java, C, C++, Python, PHP, .NET, entre outras. Neste artigo, será tratado apenas da serialização com Java, em sua implemetação básica, alguns problemas e soluções, com códigos para exemplificação.

**2. SERIALIZAÇÃO COM JAVA**

Serializar um objeto, em Java, só é possível caso sua classe esteja marcada como serializável. Essa marcação é feita com a simples implementação da interface “java.io.Serializable”. Essa interface não possui métodos para se implementar e nem atributos. Ainda não existiam as anotações no Java 1.1, versão que trouxe pela primeira vez a serialização ao Java, e desta maneira a implementação da interface era a melhor maneira de marcar uma classe.

A herança entre classes é naturalmente afetada pela serialização. Já que quando uma classe implementa a interface “Serializable”, toda classe que dela estender também estará implicitamente marcada como serializável

.

Para exemplo, será criada uma classe “pojo.Fruta” que contém dois atributos: nome e cor. No código exemplo (Listagem 1), pode-se notar a classe “Fruta” marcada para a serialização:

Listagem 1 - Marcando uma classe como serializável.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | import java.io.Serializable;    public class Fruta implements Serializable{      private String nome;      private String cor;        public Fruta(String nome, String cor) {          this.nome = nome;          this.cor = cor;      }        public String getCor() { return cor; }      public String getNome() { return nome; }      public void setCor(String cor) { this.cor = cor; }      public void setNome(String nome) { this.nome = nome; }  } |

Pode-se dizer que a implementação do processo de serialização e deserialização simples de um objeto, não possui um código difícil de se escrever. Isto porquê duas classes fazem a maior parte do trabalho, de maneira que com poucas linhas de código pode-se transformar um objeto em bytes e vice-versa. Essas duas classes são “java.io.ObjectInputStream” que recebe bytes para criar um objeto e “java.io.ObjectOutputStream” que cria uma cadeia de bytes a partir de um objeto.

**3. SERIALIZANDO E DESERIALIZANDO O ESTADO DE UM OBJETO**

Como exemplo, será persistido um objeto em disco. Para isto, além das classes “java.io.ObjectInputStream” e “java.io.ObjectOutputStream”, serão utilizadas duas outras que trabalham com arquivos, que são “java.io.FileInputStream” e “java.io.FileOutputStream”, que seguem a mesma lógica de InputStream e OutputStream, só que agora relacionado ao sistema de arquivos e não aos objetos.

Para serialização, deve-se criar um novo “java.io.FileOutputStream”, com o caminho (“path”) do arquivo onde se deseja armazenar o estado do objeto. Também deve ser criado um novo “java.io.ObjectOutputStream” que está ligado ao arquivo. E por último deve-se executar a escrita do objeto alvo, para dentro do objeto de stream que por sua vez está ligado ao arquivo destino. Após executados esses passos, é possível encontrar no sistema de arquivos um novo arquivo que contém os bytes correspondentes à descrição do estado do objeto. Na Listagem 2, encontra-se o código exemplo.

Listagem 2 - Serialização de um objeto.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | import java.io.FileOutputStream;  import java.io.ObjectOutputStream;    public class Serializador {        public Serializador() {    }        public void serializar(String path, Object obj) throws Exception {              FileOutputStream outFile = new FileOutputStream(path);      ObjectOutputStream s = new ObjectOutputStream(outFile);      s.writeObject(obj);      s.close();      }  } |

E para a deserialização, o processo é extremamente parecido com a serialização. Cria-se um novo “java.io.FileInputStream” relacionado ao arquivo onde o estado do objeto encontra-se persistido. Deve-se criar também um novo “java.io.ObjectInputStream” que estará vinculado ao arquivo de origem. Por último, executa-se a leitura do arquivo para a restauração do estado. Na Listagem 3 encontra-se o exemplo do código.

Listagem 3 - Deserializaçãode um objeto.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | import java.io.FileInputStream;  import java.io.ObjectInputStream;    public class Deserializador {        public Deserializador() {    }        public Object deserializar(String path) throws Exception {      FileInputStream inFile = new FileInputStream(path);      ObjectInputStream d = new ObjectInputStream(inFile);      Object o = d.readObject();      d.close();      return o;      }  } |

A Listagem 4, contém o exemplo da execução da serialização e em seguida da deserialização.

Listagem 4 - Serializando e Deserializando

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | import operacoes.Deserializador;  import operacoes.Serializador;  import pojo.Fruta;    public class Principal {      public static void main(String args[]){         //serializa         Serializador s = new Serializador();         Fruta fruta = new Fruta("Maça", "Vermelha");          try {              s.serializar("/home/gustavo/fruta", fruta);          } catch (Exception ex) {              System.err.println("Falha ao serializar! - " + ex.toString());          }            //deserializa          Deserializador d = new Deserializador();          fruta = null;          try {              fruta = (Fruta) d.deserializar("/home/gustavo/fruta");          } catch (Exception ex) {              System.err.println("Falha ao deserializar! - " + ex.toString());          }          System.out.println(fruta.getNome() + " - " + fruta.getCor());      }  } |

Na Listagem 5, encontra-se um exemplo de serialização e deserialização de um ArrayList da classe Fruta. É apenas mais um exemplo que demonstra a capacidade de se persistir qualquer objeto que seja serializável. Essa serialização só foi possível porque a classe “ArrayList” implementa a interface “Serializable” por padrão.

Listagem 5 - Serialização de ArrayList.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | //serialização e deserialização com ArrayList          ArrayList<Fruta> frutas = new ArrayList<Fruta>();          Fruta f1 = new Fruta("Laranja", "Amarela");          Fruta f2 = new Fruta("Abacate", "Verde");          Fruta f3 = new Fruta("Morango", "Vermelho");          frutas.add(f1);          frutas.add(f2);          frutas.add(f3);            try {              s.serializar("/home/gustavo/frutas", frutas);              frutas = null;              frutas = (ArrayList<Fruta>) d.deserializar("/home/gustavo/frutas");              for(Fruta f : frutas){                  System.out.println("ArrayList: " + f.getNome() + " - " + f.getCor());              }          } catch (Exception ex) {              System.err.println("Falha ao serializar ou deserializar! - " +                                 ex.toString());          } |

**4. PARTICULARIDADES DA SERIALIZAÇÃO**

Como dito, a serialização de um objeto em Java pode ser algo fácil de se fazer. Mas existem certas informações que são importantes e sabê-las pode evitar vários problemas.

Um dos problemas é que nem tudo pode ser serializado, como objetos das classes “Thread” e “Socket” e atributos estáticos. Os atributos estáticos estão vinculados à classe e não ao objeto. Assim quando se trata do estado do objeto, os campos estáticos não são considerados. Em relação às Threads, suas instâncias só são úteis enquanto estiverem em execução na JVM, e podem ser paradas/destruídas e iniciadas com instruções de sua classe. Não existe algo no estado destas instâncias que seja útil para se persistir, e por isso não há sentido em persistir tais instâncias para restaurar posteriormente.

Mas e se existir alguma classe em que seja necessário persisti-la, e esta possui uma instância da classe “Thread”? Neste caso, utilizando um recurso pode-se persistir a classe inteira, menos a instância thread. Este recurso, refere-se em indicar que tal instância thread é um objeto “Transient”. Na Listagem 6 encontra-se um exemplo da implementação com uso de “Transient”.

Listagem 6 - Uso do Transient

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | import java.io.Serializable;    public class ImplementacaoTransient implements Serializable, Runnable{        transient private Thread thread;      //marcado com "transient" para que sua instância não seja serializada        //Demais valores de estado serão serializados      private int valorParaPersistir;        public ImplementacaoTransient(int valorParaPersistir){          this.valorParaPersistir = valorParaPersistir;          thread = new Thread(this);          thread.start();      }        public void run()      {          while(true){              //código a ser executado pela Thread          }      }  } |

Vale ressaltar que, o que é serializado é apenas o estado do objeto, e não os métodos e a classe. No exemplo acima, apenas a instância do objeto “thread” e do objeto “valorParaPersistir” compunham o estado do objeto. Assim, evita-se que o sistema de serialização tente serializar um objeto não-serializável e gere erros. Após a deserialização de um objeto da classe “ImplementacaoTransient” este estará totalmente funcional.

Um outro problema que pode vir a ocorrer, se refere ao versionamento das classes. Toda classe possui um atributo oculto chamado “serialVersionUID” com um valor gerado automaticamente. Objetos criados levam consigo o “serialVersionUID” de sua classe, que indica sua versão. Toda vez que uma classe sofre alterações em seu código, um novo “serialVersionUID” é gerado. O problema com a serialização surge quando serializa-se o estado de um objeto e posteriormente tenta-se restaurar o estado para a aplicação, mas houveram mudanças no código da classe. Neste caso, uma exceção “java.io.InvalidClassException” é disparada, pois de fato, o estado do objeto corresponde a uma versão de classe do qual foi criado que já não existe mais. Portanto, isso só ocorre para evitar erros de programação, em que tenta-se restaurar o estado de um objeto para uma classe que possui versão diferente da qual o objeto foi serializado inicialmente. E isso pode significar diferentes versões de uma mesma classe “Fruta” por exemplo, ou diferentes versões por classes serem totalmente diferentes como: o estado de um objeto da classe “Fruta” ser restaurado para classe “Carro”.

Mas quando é necessário, conscientemente, recuperar o estado de um objeto, mesmo que houveram alterações na classe do qual foi gerado, é possível controlar manualmente o “serialVersionUID”. Para isto, basta declarar explicitamente na forma de atributo estático e final, um valor Long qualquer para corresponder à versão da classe. E assim, mesmo que houverem modificações na classe, a versão permenecerá a mesma, evitando as exceções da incompatiblidade de classes. No entanto, vale ressaltar que o versionamento não é um impecílio para o programador, mas sim muito importante em casos que deseja-se manter um controle sobre a compatibilidade de código na aplicação, definindo versões para classes No código da Listagem 7 encontra-se o exemplo do controle manual do “serialVersionUID”.

Listagem 7 - Uso manual do serialVersionUID

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | import java.io.Serializable;    public class Fruta implements Serializable{      private String nome;      private String cor;      static final long serialVersionUID = 123L; //indica a versao da classe        public Fruta(String nome, String cor) {          this.nome = nome;          this.cor = cor;      }      public String getCor() { return cor; }      public String getNome() { return nome; }      public void setCor(String cor) { this.cor = cor; }      public void setNome(String nome) { this.nome = nome; }  } |

O JDK fornece uma ferramenta que permite visualizar a versão de qualquer classe Java. Basta no terminal do sistema operacional utilizar o comando “serialver” seguido da classe qualificada, conforme exemplo da Listagem 8.

Listagem 8 - Verificação da versão de classe

|  |  |
| --- | --- |
| 1 | serialver pojo.Fruta |

A serialização acaba inflingindo alguns modelos de segurança em seu procedimento padrão, e criando então um novo problema. Quando um estado de objeto é persistido no sistema de arquivos ou enviado por rede, o fluxo binário pode ser detectado, analisado e obtidas informações sobre a classe e o estado do objeto. Por exemplo, se utilizado um editor de texto qualquer para abrir o arquivo persistido do estado do objeto da classe “Fruta” dos exemplos anteriores, é comum os atributos e outras informações aparecerem como um simples texto, total ou parcilamente legíveis, assim como demonstrado na Listagem 9.

Listagem 9 - Exemplo de leitura de arquivo binário de um estado de objeto persistido.

|  |  |
| --- | --- |
| 1  2 | ��##sr#  pojo.Fruta#######{###L##cort##Ljava/lang/String;L##nomeq#~##xpt##Vermelhat##Maça |

Com uma ferramenta menos simples seria possível aumentar a quantidade de informações legíveis, coletando uma maior quantia de dados.

No entando, o processo de serialização permite que haja uma certa personalização da serialização e deserialização de um objeto. Isso deve-se a possibilidade da sobrescrição dos métodos “writeObject” e “readObject” dentro da classe a ser serializada, como a classe exemplar “pojo.Fruta”. Esta sobrescrita permite um controle total sobre a serialização, pois as classes “java.io.ObjectOutputStream” e “java.io.ObjectInputStream”, ao serializar o estado de um objeto, primeiramente verificam se esse método foi sobrescrito na classe alvo e assim o utilizam, senão, será utilizado o protocolo padrão de serialização.

Realizando a sobrescrição dos métodos é possível criptografar os dados na serialização e descriptografar os dados na deserialização, por meio de algum algoritmo que o programador venha implementar. Assim os dados podem ser persistidos ou transmitidos por rede, sem que seja possível ler as partes mais importantes do estado de um objeto. Na listagem 10, a classe “pojo.Fruta” foi modificada dando origem a classe “pojo.FrutaSobrescrita” que possui adicionamente a sobrescrição dos métodos “writeObject” e “readObject”. Como este é apenas um exemplo, não foi escrito um verdadeiro algoritmo de criptografia, mas sim apenas uma troca no valor da String do campo “cor”, identificando que houveram alterações tanto no processo de serialização quanto no de deserialização. Na listagem 11 encontra-se a execução da serialização.

Listagem 10 - Sobrescrita dos métodos “writeObject” e “readObject”.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36 | import java.io.Serializable;    public class FrutaSobrescrita implements Serializable{      private String nome;      private String cor;      static final long serialVersionUID = 123L; //indica a versao da classe        public FrutaSobrescrita(String nome, String cor) {          this.nome = nome;          this.cor = cor;      }      public String getCor() { return cor; }      public String getNome() { return nome; }      public void setCor(String cor) { this.cor = cor; }      public void setNome(String nome) { this.nome = nome; }        private void writeObject(java.io.ObjectOutputStream stream) throws java.io.IOException      {          // criptografar os dados antes de liberar o stream          // EXEMPLO: setCor(criptografar(getCor()));          setCor(getCor()+ " > CRIPTOGRAFADO"); // apenas concatenação de string para indicar alteração            // liberar o stream          stream.defaultWriteObject();      }        private void readObject(java.io.ObjectInputStream stream) throws java.io.IOException, ClassNotFoundException      {          // liberar stream          stream.defaultReadObject();            // descriptografar os dados          // EXEMPLO: setCor(descriptografar(getCor()));          setCor(getCor()+ " > DESCRIPTOGRAFADO"); // apenas contatenação de string para indicar alteração      }  } |

Listagem 11 - Execução da serialização da classe “pojo.FrutaSobrescrita”.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | //serializa        Serializador s = new Serializador();        FrutaSobrescrita frutaSobrescrita = new FrutaSobrescrita("Maça", "Vermelha");         try {             s.serializar("/home/gustavo/frutaSobrescrita", frutaSobrescrita);         } catch (Exception ex) {             System.err.println("Falha ao serializar! - " + ex.toString());         }           //deserializa        Deserializador d = new Deserializador();         frutaSobrescrita = null;         try {             frutaSobrescrita = (FrutaSobrescrita) d.deserializar("/home/gustavo/frutaSobrescrita");         } catch (Exception ex) {             System.err.println("Falha ao deserializar! - " + ex.toString());         }         System.out.println("Sobrescrita: " + frutaSobrescrita.getNome() + " - " + frutaSobrescrita.getCor()); |

**5. CONCLUSÃO**

A serialização é um procedimento de simples execução por parte do programador. Com pouco tempo de codificação é possível implementar a persistencia de um estado de objeto para o sistema de arquivos ou de transmissão por rede. É possível reproduzir os exemplos criados neste artigo, em situações até mesmo mais elaboradas. A apresentação de alguns problemas e soluções trazidas por particularidades, abre o entendimento de que a serialização pode deixar de ser simples conforme for aplicada em diferentes contextos, e que ao mesmo tempo é muito flexível ao permitir ao programador implementar diferentes soluções.

[Assista ao vídeo Serialização/Deserialização de objetos com YAML](http://www.devmedia.com.br/serializacao-deserializacao-de-objetos-com-yaml-parte-1/8399)

**6. BIBLIOGRAFIA**

ORACLE - **java.io Interface Serializable** - Disponível em: <http://docs.oracle.com/javase/1.4.2/docs/api/java/io/Serializable.html>Acesso em: 11/03/2012

ORACLE - **Discover de Secrets of Java Serialization API** - Disponível em: <http://java.sun.com/developer/technicalArticles/Programming/serialization/> Acesso em: 11/03/2012

NEWARD, Ted - **5 coisas que você não sabia sobre... Serialização de Objetos Java** - Disponível em: <http://www.ibm.com/developerworks/br/library/j-5things1/index.html> Acesso em: 11/03/2012

DOWNS, Andrew - **Java Serialization** - Disponível em: <http://www.mactech.com/articles/mactech/Vol.14/14.04/JavaSerialization/index.html> Acesso em: 11/03/2012

PREISSLER, Luciano E.; GEYER, Cláudio F. R. - **Serialização em Java** - Disponível em: <http://www.inf.ufrgs.br/gppd/disc/cmp167/trabalhos/sem99-1/T1/luciano/final.htm> Acesso em: 11/03/2012

[[](http://www.linhadecodigo.com.br/colaborador.aspx?id=354)](http://www.linhadecodigo.com.br/colaborador.aspx?id=354)

[Everton Coimbra de Araújo - Desde 1987 atua na área de treinamento e desenvolvimento. Como Mestre em Ciência da Computação, é professor da UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ, Campus Medianeira, onde leciona disciplinas relacionadas ao desenvolvimento de aplicações web, com Java e .NET.   
  
É autor dos livros Desenvolvimento para WEB com Java, Orientação a Objetos com Java - Simples, Fácil e Eficiente, Algoritmo - Fundamento e Prática em sua terceira edição e dos livros Delphi - Implementação e Técnicas para Ambientes Virtuais e C++ Builder - Implementação e Técnicas para Ambientes Virtuais. Todos pela VisualBooks. Pode ser contactado através do e-mail everton@utfpr.edu.br ou evertoncoimbra@gmail.com.](http://www.linhadecodigo.com.br/colaborador.aspx?id=354)

 85 3 0 4

**Leia também**

* [Conhecendo a interface Map do Java](http://www.linhadecodigo.com.br/artigo/3670/conhecendo-a-interface-map-do-java.aspx)*Java*
* [Trabalhando com a Interface Set no Java](http://www.linhadecodigo.com.br/artigo/3669/trabalhando-com-a-interface-set-no-java.aspx)*Java*
* [Classes Wrappers em Java](http://www.linhadecodigo.com.br/artigo/3667/classes-wrappers-em-java.aspx)*Java*
* [ComboBoxModel em Java](http://www.linhadecodigo.com.br/artigo/3664/comboboxmodel-em-java.aspx)*Java*
* [Reduzindo a Mutabilidade em Java](http://www.linhadecodigo.com.br/artigo/3659/reduzindo-a-mutabilidade-em-java.aspx)*Java*

Publicidade

[](http://qconsp.com/?utm_source=devmedia&utm_medium=banner&utm_campaign=devmedia)

Linha de Código[Anuncie](http://www.web-03.net/linhadecodigo.asp)|[Fale Conosco](mailto:colabore@linhadecodigo.com.br)|[Publique](http://www.linhadecodigo.com.br/publique.aspx)

|  |  |
| --- | --- |
| http://www.linhadecodigo.com.br/artigos/img_artigos/EvertonCoimbraAraujo/Serializazao_Objetos/imagem001.jpg | Neste artigo apresento o Gustavo Rafael Valiati, que pode ser contatado pelo e-mail gustavovaliati@gmail.com. O Gustavo foi meu aluno em algumas disciplinas de sua gradução e está concluíndo o último período do curso de Tecnologia em Análise e Desenvolvimento de Sistemas da UTFPR campus Medianeira, neste primeiro semestre de 2012. |

**1. INTRODUÇÃO**

Nem sempre se deseja ter objetos ou estrutura de dados de um aplicativo, com um ciclo de vida que dura no máximo o tempo em que for armazenado em memória RAM. Ou seja, algumas vezes deseja-se guardá-los por um tempo indeterminado, sendo possível recuperá-los posteriormente. E neste caso, o termo “persistência” tem seu conceito definido.

Persistir dados, requer que estes possam ser serializáveis. A serialização possibilitará capturar o estado do objeto ou a estrutura de dados, e transformar em uma cadeia de bytes quando for necessário. Torna-se possível também, recuperar os bytes persistidos para fazer o processo inverso, e ter os dados de volta para a aplicação em execução.

Desta maneira, qualquer forma de dados que esteja sendo trabalhada na aplicação, e esta puder ser serializada, poderá ser passada para o meio físico, e ter o processo revertido.

A serialização não se limita apenas em possibilitar a gravação do dado em disco, mas permite que o mesmo seja transmitido por rede. Isto porquê a serialização cria um stream de bytes, que é requisito para transmissão de dados por rede.

Diversas linguagens possuem suporte à serialização, como: Java, C, C++, Python, PHP, .NET, entre outras. Neste artigo, será tratado apenas da serialização com Java, em sua implemetação básica, alguns problemas e soluções, com códigos para exemplificação.

**2. SERIALIZAÇÃO COM JAVA**

Serializar um objeto, em Java, só é possível caso sua classe esteja marcada como serializável. Essa marcação é feita com a simples implementação da interface “java.io.Serializable”. Essa interface não possui métodos para se implementar e nem atributos. Ainda não existiam as anotações no Java 1.1, versão que trouxe pela primeira vez a serialização ao Java, e desta maneira a implementação da interface era a melhor maneira de marcar uma classe.

A herança entre classes é naturalmente afetada pela serialização. Já que quando uma classe implementa a interface “Serializable”, toda classe que dela estender também estará implicitamente marcada como serializável

.

Para exemplo, será criada uma classe “pojo.Fruta” que contém dois atributos: nome e cor. No código exemplo (Listagem 1), pode-se notar a classe “Fruta” marcada para a serialização:

Listagem 1 - Marcando uma classe como serializável.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | import java.io.Serializable;    public class Fruta implements Serializable{      private String nome;      private String cor;        public Fruta(String nome, String cor) {          this.nome = nome;          this.cor = cor;      }        public String getCor() { return cor; }      public String getNome() { return nome; }      public void setCor(String cor) { this.cor = cor; }      public void setNome(String nome) { this.nome = nome; }  } |

Pode-se dizer que a implementação do processo de serialização e deserialização simples de um objeto, não possui um código difícil de se escrever. Isto porquê duas classes fazem a maior parte do trabalho, de maneira que com poucas linhas de código pode-se transformar um objeto em bytes e vice-versa. Essas duas classes são “java.io.ObjectInputStream” que recebe bytes para criar um objeto e “java.io.ObjectOutputStream” que cria uma cadeia de bytes a partir de um objeto.

**3. SERIALIZANDO E DESERIALIZANDO O ESTADO DE UM OBJETO**

Como exemplo, será persistido um objeto em disco. Para isto, além das classes “java.io.ObjectInputStream” e “java.io.ObjectOutputStream”, serão utilizadas duas outras que trabalham com arquivos, que são “java.io.FileInputStream” e “java.io.FileOutputStream”, que seguem a mesma lógica de InputStream e OutputStream, só que agora relacionado ao sistema de arquivos e não aos objetos.

Para serialização, deve-se criar um novo “java.io.FileOutputStream”, com o caminho (“path”) do arquivo onde se deseja armazenar o estado do objeto. Também deve ser criado um novo “java.io.ObjectOutputStream” que está ligado ao arquivo. E por último deve-se executar a escrita do objeto alvo, para dentro do objeto de stream que por sua vez está ligado ao arquivo destino. Após executados esses passos, é possível encontrar no sistema de arquivos um novo arquivo que contém os bytes correspondentes à descrição do estado do objeto. Na Listagem 2, encontra-se o código exemplo.

Listagem 2 - Serialização de um objeto.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | import java.io.FileOutputStream;  import java.io.ObjectOutputStream;    public class Serializador {        public Serializador() {    }        public void serializar(String path, Object obj) throws Exception {              FileOutputStream outFile = new FileOutputStream(path);      ObjectOutputStream s = new ObjectOutputStream(outFile);      s.writeObject(obj);      s.close();      }  } |

E para a deserialização, o processo é extremamente parecido com a serialização. Cria-se um novo “java.io.FileInputStream” relacionado ao arquivo onde o estado do objeto encontra-se persistido. Deve-se criar também um novo “java.io.ObjectInputStream” que estará vinculado ao arquivo de origem. Por último, executa-se a leitura do arquivo para a restauração do estado. Na Listagem 3 encontra-se o exemplo do código.

Listagem 3 - Deserializaçãode um objeto.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | import java.io.FileInputStream;  import java.io.ObjectInputStream;    public class Deserializador {        public Deserializador() {    }        public Object deserializar(String path) throws Exception {      FileInputStream inFile = new FileInputStream(path);      ObjectInputStream d = new ObjectInputStream(inFile);      Object o = d.readObject();      d.close();      return o;      }  } |

A Listagem 4, contém o exemplo da execução da serialização e em seguida da deserialização.

Listagem 4 - Serializando e Deserializando

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | import operacoes.Deserializador;  import operacoes.Serializador;  import pojo.Fruta;    public class Principal {      public static void main(String args[]){         //serializa         Serializador s = new Serializador();         Fruta fruta = new Fruta("Maça", "Vermelha");          try {              s.serializar("/home/gustavo/fruta", fruta);          } catch (Exception ex) {              System.err.println("Falha ao serializar! - " + ex.toString());          }            //deserializa          Deserializador d = new Deserializador();          fruta = null;          try {              fruta = (Fruta) d.deserializar("/home/gustavo/fruta");          } catch (Exception ex) {              System.err.println("Falha ao deserializar! - " + ex.toString());          }          System.out.println(fruta.getNome() + " - " + fruta.getCor());      }  } |

Na Listagem 5, encontra-se um exemplo de serialização e deserialização de um ArrayList da classe Fruta. É apenas mais um exemplo que demonstra a capacidade de se persistir qualquer objeto que seja serializável. Essa serialização só foi possível porque a classe “ArrayList” implementa a interface “Serializable” por padrão.

Listagem 5 - Serialização de ArrayList.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | //serialização e deserialização com ArrayList          ArrayList<Fruta> frutas = new ArrayList<Fruta>();          Fruta f1 = new Fruta("Laranja", "Amarela");          Fruta f2 = new Fruta("Abacate", "Verde");          Fruta f3 = new Fruta("Morango", "Vermelho");          frutas.add(f1);          frutas.add(f2);          frutas.add(f3);            try {              s.serializar("/home/gustavo/frutas", frutas);              frutas = null;              frutas = (ArrayList<Fruta>) d.deserializar("/home/gustavo/frutas");              for(Fruta f : frutas){                  System.out.println("ArrayList: " + f.getNome() + " - " + f.getCor());              }          } catch (Exception ex) {              System.err.println("Falha ao serializar ou deserializar! - " +                                 ex.toString());          } |

**4. PARTICULARIDADES DA SERIALIZAÇÃO**

Como dito, a serialização de um objeto em Java pode ser algo fácil de se fazer. Mas existem certas informações que são importantes e sabê-las pode evitar vários problemas.

Um dos problemas é que nem tudo pode ser serializado, como objetos das classes “Thread” e “Socket” e atributos estáticos. Os atributos estáticos estão vinculados à classe e não ao objeto. Assim quando se trata do estado do objeto, os campos estáticos não são considerados. Em relação às Threads, suas instâncias só são úteis enquanto estiverem em execução na JVM, e podem ser paradas/destruídas e iniciadas com instruções de sua classe. Não existe algo no estado destas instâncias que seja útil para se persistir, e por isso não há sentido em persistir tais instâncias para restaurar posteriormente.

Mas e se existir alguma classe em que seja necessário persisti-la, e esta possui uma instância da classe “Thread”? Neste caso, utilizando um recurso pode-se persistir a classe inteira, menos a instância thread. Este recurso, refere-se em indicar que tal instância thread é um objeto “Transient”. Na Listagem 6 encontra-se um exemplo da implementação com uso de “Transient”.

Listagem 6 - Uso do Transient

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | import java.io.Serializable;    public class ImplementacaoTransient implements Serializable, Runnable{        transient private Thread thread;      //marcado com "transient" para que sua instância não seja serializada        //Demais valores de estado serão serializados      private int valorParaPersistir;        public ImplementacaoTransient(int valorParaPersistir){          this.valorParaPersistir = valorParaPersistir;          thread = new Thread(this);          thread.start();      }        public void run()      {          while(true){              //código a ser executado pela Thread          }      }  } |

Vale ressaltar que, o que é serializado é apenas o estado do objeto, e não os métodos e a classe. No exemplo acima, apenas a instância do objeto “thread” e do objeto “valorParaPersistir” compunham o estado do objeto. Assim, evita-se que o sistema de serialização tente serializar um objeto não-serializável e gere erros. Após a deserialização de um objeto da classe “ImplementacaoTransient” este estará totalmente funcional.

Um outro problema que pode vir a ocorrer, se refere ao versionamento das classes. Toda classe possui um atributo oculto chamado “serialVersionUID” com um valor gerado automaticamente. Objetos criados levam consigo o “serialVersionUID” de sua classe, que indica sua versão. Toda vez que uma classe sofre alterações em seu código, um novo “serialVersionUID” é gerado. O problema com a serialização surge quando serializa-se o estado de um objeto e posteriormente tenta-se restaurar o estado para a aplicação, mas houveram mudanças no código da classe. Neste caso, uma exceção “java.io.InvalidClassException” é disparada, pois de fato, o estado do objeto corresponde a uma versão de classe do qual foi criado que já não existe mais. Portanto, isso só ocorre para evitar erros de programação, em que tenta-se restaurar o estado de um objeto para uma classe que possui versão diferente da qual o objeto foi serializado inicialmente. E isso pode significar diferentes versões de uma mesma classe “Fruta” por exemplo, ou diferentes versões por classes serem totalmente diferentes como: o estado de um objeto da classe “Fruta” ser restaurado para classe “Carro”.

Mas quando é necessário, conscientemente, recuperar o estado de um objeto, mesmo que houveram alterações na classe do qual foi gerado, é possível controlar manualmente o “serialVersionUID”. Para isto, basta declarar explicitamente na forma de atributo estático e final, um valor Long qualquer para corresponder à versão da classe. E assim, mesmo que houverem modificações na classe, a versão permenecerá a mesma, evitando as exceções da incompatiblidade de classes. No entanto, vale ressaltar que o versionamento não é um impecílio para o programador, mas sim muito importante em casos que deseja-se manter um controle sobre a compatibilidade de código na aplicação, definindo versões para classes No código da Listagem 7 encontra-se o exemplo do controle manual do “serialVersionUID”.

Listagem 7 - Uso manual do serialVersionUID

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | import java.io.Serializable;    public class Fruta implements Serializable{      private String nome;      private String cor;      static final long serialVersionUID = 123L; //indica a versao da classe        public Fruta(String nome, String cor) {          this.nome = nome;          this.cor = cor;      }      public String getCor() { return cor; }      public String getNome() { return nome; }      public void setCor(String cor) { this.cor = cor; }      public void setNome(String nome) { this.nome = nome; }  } |

O JDK fornece uma ferramenta que permite visualizar a versão de qualquer classe Java. Basta no terminal do sistema operacional utilizar o comando “serialver” seguido da classe qualificada, conforme exemplo da Listagem 8.

Listagem 8 - Verificação da versão de classe

|  |  |
| --- | --- |
| 1 | serialver pojo.Fruta |

A serialização acaba inflingindo alguns modelos de segurança em seu procedimento padrão, e criando então um novo problema. Quando um estado de objeto é persistido no sistema de arquivos ou enviado por rede, o fluxo binário pode ser detectado, analisado e obtidas informações sobre a classe e o estado do objeto. Por exemplo, se utilizado um editor de texto qualquer para abrir o arquivo persistido do estado do objeto da classe “Fruta” dos exemplos anteriores, é comum os atributos e outras informações aparecerem como um simples texto, total ou parcilamente legíveis, assim como demonstrado na Listagem 9.

Listagem 9 - Exemplo de leitura de arquivo binário de um estado de objeto persistido.

|  |  |
| --- | --- |
| 1  2 | ��##sr#  pojo.Fruta#######{###L##cort##Ljava/lang/String;L##nomeq#~##xpt##Vermelhat##Maça |

Com uma ferramenta menos simples seria possível aumentar a quantidade de informações legíveis, coletando uma maior quantia de dados.

No entando, o processo de serialização permite que haja uma certa personalização da serialização e deserialização de um objeto. Isso deve-se a possibilidade da sobrescrição dos métodos “writeObject” e “readObject” dentro da classe a ser serializada, como a classe exemplar “pojo.Fruta”. Esta sobrescrita permite um controle total sobre a serialização, pois as classes “java.io.ObjectOutputStream” e “java.io.ObjectInputStream”, ao serializar o estado de um objeto, primeiramente verificam se esse método foi sobrescrito na classe alvo e assim o utilizam, senão, será utilizado o protocolo padrão de serialização.

Realizando a sobrescrição dos métodos é possível criptografar os dados na serialização e descriptografar os dados na deserialização, por meio de algum algoritmo que o programador venha implementar. Assim os dados podem ser persistidos ou transmitidos por rede, sem que seja possível ler as partes mais importantes do estado de um objeto. Na listagem 10, a classe “pojo.Fruta” foi modificada dando origem a classe “pojo.FrutaSobrescrita” que possui adicionamente a sobrescrição dos métodos “writeObject” e “readObject”. Como este é apenas um exemplo, não foi escrito um verdadeiro algoritmo de criptografia, mas sim apenas uma troca no valor da String do campo “cor”, identificando que houveram alterações tanto no processo de serialização quanto no de deserialização. Na listagem 11 encontra-se a execução da serialização.

Listagem 10 - Sobrescrita dos métodos “writeObject” e “readObject”.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36 | import java.io.Serializable;    public class FrutaSobrescrita implements Serializable{      private String nome;      private String cor;      static final long serialVersionUID = 123L; //indica a versao da classe        public FrutaSobrescrita(String nome, String cor) {          this.nome = nome;          this.cor = cor;      }      public String getCor() { return cor; }      public String getNome() { return nome; }      public void setCor(String cor) { this.cor = cor; }      public void setNome(String nome) { this.nome = nome; }        private void writeObject(java.io.ObjectOutputStream stream) throws java.io.IOException      {          // criptografar os dados antes de liberar o stream          // EXEMPLO: setCor(criptografar(getCor()));          setCor(getCor()+ " > CRIPTOGRAFADO"); // apenas concatenação de string para indicar alteração            // liberar o stream          stream.defaultWriteObject();      }        private void readObject(java.io.ObjectInputStream stream) throws java.io.IOException, ClassNotFoundException      {          // liberar stream          stream.defaultReadObject();            // descriptografar os dados          // EXEMPLO: setCor(descriptografar(getCor()));          setCor(getCor()+ " > DESCRIPTOGRAFADO"); // apenas contatenação de string para indicar alteração      }  } |

Listagem 11 - Execução da serialização da classe “pojo.FrutaSobrescrita”.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | //serializa        Serializador s = new Serializador();        FrutaSobrescrita frutaSobrescrita = new FrutaSobrescrita("Maça", "Vermelha");         try {             s.serializar("/home/gustavo/frutaSobrescrita", frutaSobrescrita);         } catch (Exception ex) {             System.err.println("Falha ao serializar! - " + ex.toString());         }           //deserializa        Deserializador d = new Deserializador();         frutaSobrescrita = null;         try {             frutaSobrescrita = (FrutaSobrescrita) d.deserializar("/home/gustavo/frutaSobrescrita");         } catch (Exception ex) {             System.err.println("Falha ao deserializar! - " + ex.toString());         }         System.out.println("Sobrescrita: " + frutaSobrescrita.getNome() + " - " + frutaSobrescrita.getCor()); |

**5. CONCLUSÃO**

A serialização é um procedimento de simples execução por parte do programador. Com pouco tempo de codificação é possível implementar a persistencia de um estado de objeto para o sistema de arquivos ou de transmissão por rede. É possível reproduzir os exemplos criados neste artigo, em situações até mesmo mais elaboradas. A apresentação de alguns problemas e soluções trazidas por particularidades, abre o entendimento de que a serialização pode deixar de ser simples conforme for aplicada em diferentes contextos, e que ao mesmo tempo é muito flexível ao permitir ao programador implementar diferentes soluções.

[Assista ao vídeo Serialização/Deserialização de objetos com YAML](http://www.devmedia.com.br/serializacao-deserializacao-de-objetos-com-yaml-parte-1/8399)

**6. BIBLIOGRAFIA**

ORACLE - **java.io Interface Serializable** - Disponível em: <http://docs.oracle.com/javase/1.4.2/docs/api/java/io/Serializable.html>Acesso em: 11/03/2012

ORACLE - **Discover de Secrets of Java Serialization API** - Disponível em: <http://java.sun.com/developer/technicalArticles/Programming/serialization/> Acesso em: 11/03/2012

NEWARD, Ted - **5 coisas que você não sabia sobre... Serialização de Objetos Java** - Disponível em: <http://www.ibm.com/developerworks/br/library/j-5things1/index.html> Acesso em: 11/03/2012

DOWNS, Andrew - **Java Serialization** - Disponível em: <http://www.mactech.com/articles/mactech/Vol.14/14.04/JavaSerialization/index.html> Acesso em: 11/03/2012

PREISSLER, Luciano E.; GEYER, Cláudio F. R. - **Serialização em Java** - Disponível em: <http://www.inf.ufrgs.br/gppd/disc/cmp167/trabalhos/sem99-1/T1/luciano/final.htm> Acesso em: 11/03/2012

[[](http://www.linhadecodigo.com.br/colaborador.aspx?id=354)](http://www.linhadecodigo.com.br/colaborador.aspx?id=354)

[Everton Coimbra de Araújo - Desde 1987 atua na área de treinamento e desenvolvimento. Como Mestre em Ciência da Computação, é professor da UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ, Campus Medianeira, onde leciona disciplinas relacionadas ao desenvolvimento de aplicações web, com Java e .NET.   
  
É autor dos livros Desenvolvimento para WEB com Java, Orientação a Objetos com Java - Simples, Fácil e Eficiente, Algoritmo - Fundamento e Prática em sua terceira edição e dos livros Delphi - Implementação e Técnicas para Ambientes Virtuais e C++ Builder - Implementação e Técnicas para Ambientes Virtuais. Todos pela VisualBooks. Pode ser contactado através do e-mail everton@utfpr.edu.br ou evertoncoimbra@gmail.com.](http://www.linhadecodigo.com.br/colaborador.aspx?id=354)

 85 3 0 4

**Leia também**

* [Conhecendo a interface Map do Java](http://www.linhadecodigo.com.br/artigo/3670/conhecendo-a-interface-map-do-java.aspx)*Java*
* [Trabalhando com a Interface Set no Java](http://www.linhadecodigo.com.br/artigo/3669/trabalhando-com-a-interface-set-no-java.aspx)*Java*
* [Classes Wrappers em Java](http://www.linhadecodigo.com.br/artigo/3667/classes-wrappers-em-java.aspx)*Java*
* [ComboBoxModel em Java](http://www.linhadecodigo.com.br/artigo/3664/comboboxmodel-em-java.aspx)*Java*
* [Reduzindo a Mutabilidade em Java](http://www.linhadecodigo.com.br/artigo/3659/reduzindo-a-mutabilidade-em-java.aspx)*Java*

Publicidade

[](http://qconsp.com/?utm_source=devmedia&utm_medium=banner&utm_campaign=devmedia)

Linha de Código[Anuncie](http://www.web-03.net/linhadecodigo.asp)|[Fale Conosco](mailto:colabore@linhadecodigo.com.br)|[Publique](http://www.linhadecodigo.com.br/publique.aspx)

|  |  |
| --- | --- |
| http://www.linhadecodigo.com.br/artigos/img_artigos/EvertonCoimbraAraujo/Serializazao_Objetos/imagem001.jpg | Neste artigo apresento o Gustavo Rafael Valiati, que pode ser contatado pelo e-mail gustavovaliati@gmail.com. O Gustavo foi meu aluno em algumas disciplinas de sua gradução e está concluíndo o último período do curso de Tecnologia em Análise e Desenvolvimento de Sistemas da UTFPR campus Medianeira, neste primeiro semestre de 2012. |

**1. INTRODUÇÃO**

Nem sempre se deseja ter objetos ou estrutura de dados de um aplicativo, com um ciclo de vida que dura no máximo o tempo em que for armazenado em memória RAM. Ou seja, algumas vezes deseja-se guardá-los por um tempo indeterminado, sendo possível recuperá-los posteriormente. E neste caso, o termo “persistência” tem seu conceito definido.

Persistir dados, requer que estes possam ser serializáveis. A serialização possibilitará capturar o estado do objeto ou a estrutura de dados, e transformar em uma cadeia de bytes quando for necessário. Torna-se possível também, recuperar os bytes persistidos para fazer o processo inverso, e ter os dados de volta para a aplicação em execução.

Desta maneira, qualquer forma de dados que esteja sendo trabalhada na aplicação, e esta puder ser serializada, poderá ser passada para o meio físico, e ter o processo revertido.

A serialização não se limita apenas em possibilitar a gravação do dado em disco, mas permite que o mesmo seja transmitido por rede. Isto porquê a serialização cria um stream de bytes, que é requisito para transmissão de dados por rede.

Diversas linguagens possuem suporte à serialização, como: Java, C, C++, Python, PHP, .NET, entre outras. Neste artigo, será tratado apenas da serialização com Java, em sua implemetação básica, alguns problemas e soluções, com códigos para exemplificação.

**2. SERIALIZAÇÃO COM JAVA**

Serializar um objeto, em Java, só é possível caso sua classe esteja marcada como serializável. Essa marcação é feita com a simples implementação da interface “java.io.Serializable”. Essa interface não possui métodos para se implementar e nem atributos. Ainda não existiam as anotações no Java 1.1, versão que trouxe pela primeira vez a serialização ao Java, e desta maneira a implementação da interface era a melhor maneira de marcar uma classe.

A herança entre classes é naturalmente afetada pela serialização. Já que quando uma classe implementa a interface “Serializable”, toda classe que dela estender também estará implicitamente marcada como serializável

.

Para exemplo, será criada uma classe “pojo.Fruta” que contém dois atributos: nome e cor. No código exemplo (Listagem 1), pode-se notar a classe “Fruta” marcada para a serialização:

Listagem 1 - Marcando uma classe como serializável.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | import java.io.Serializable;    public class Fruta implements Serializable{      private String nome;      private String cor;        public Fruta(String nome, String cor) {          this.nome = nome;          this.cor = cor;      }        public String getCor() { return cor; }      public String getNome() { return nome; }      public void setCor(String cor) { this.cor = cor; }      public void setNome(String nome) { this.nome = nome; }  } |

Pode-se dizer que a implementação do processo de serialização e deserialização simples de um objeto, não possui um código difícil de se escrever. Isto porquê duas classes fazem a maior parte do trabalho, de maneira que com poucas linhas de código pode-se transformar um objeto em bytes e vice-versa. Essas duas classes são “java.io.ObjectInputStream” que recebe bytes para criar um objeto e “java.io.ObjectOutputStream” que cria uma cadeia de bytes a partir de um objeto.

**3. SERIALIZANDO E DESERIALIZANDO O ESTADO DE UM OBJETO**

Como exemplo, será persistido um objeto em disco. Para isto, além das classes “java.io.ObjectInputStream” e “java.io.ObjectOutputStream”, serão utilizadas duas outras que trabalham com arquivos, que são “java.io.FileInputStream” e “java.io.FileOutputStream”, que seguem a mesma lógica de InputStream e OutputStream, só que agora relacionado ao sistema de arquivos e não aos objetos.

Para serialização, deve-se criar um novo “java.io.FileOutputStream”, com o caminho (“path”) do arquivo onde se deseja armazenar o estado do objeto. Também deve ser criado um novo “java.io.ObjectOutputStream” que está ligado ao arquivo. E por último deve-se executar a escrita do objeto alvo, para dentro do objeto de stream que por sua vez está ligado ao arquivo destino. Após executados esses passos, é possível encontrar no sistema de arquivos um novo arquivo que contém os bytes correspondentes à descrição do estado do objeto. Na Listagem 2, encontra-se o código exemplo.

Listagem 2 - Serialização de um objeto.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | import java.io.FileOutputStream;  import java.io.ObjectOutputStream;    public class Serializador {        public Serializador() {    }        public void serializar(String path, Object obj) throws Exception {              FileOutputStream outFile = new FileOutputStream(path);      ObjectOutputStream s = new ObjectOutputStream(outFile);      s.writeObject(obj);      s.close();      }  } |

E para a deserialização, o processo é extremamente parecido com a serialização. Cria-se um novo “java.io.FileInputStream” relacionado ao arquivo onde o estado do objeto encontra-se persistido. Deve-se criar também um novo “java.io.ObjectInputStream” que estará vinculado ao arquivo de origem. Por último, executa-se a leitura do arquivo para a restauração do estado. Na Listagem 3 encontra-se o exemplo do código.

Listagem 3 - Deserializaçãode um objeto.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | import java.io.FileInputStream;  import java.io.ObjectInputStream;    public class Deserializador {        public Deserializador() {    }        public Object deserializar(String path) throws Exception {      FileInputStream inFile = new FileInputStream(path);      ObjectInputStream d = new ObjectInputStream(inFile);      Object o = d.readObject();      d.close();      return o;      }  } |

A Listagem 4, contém o exemplo da execução da serialização e em seguida da deserialização.

Listagem 4 - Serializando e Deserializando

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | import operacoes.Deserializador;  import operacoes.Serializador;  import pojo.Fruta;    public class Principal {      public static void main(String args[]){         //serializa         Serializador s = new Serializador();         Fruta fruta = new Fruta("Maça", "Vermelha");          try {              s.serializar("/home/gustavo/fruta", fruta);          } catch (Exception ex) {              System.err.println("Falha ao serializar! - " + ex.toString());          }            //deserializa          Deserializador d = new Deserializador();          fruta = null;          try {              fruta = (Fruta) d.deserializar("/home/gustavo/fruta");          } catch (Exception ex) {              System.err.println("Falha ao deserializar! - " + ex.toString());          }          System.out.println(fruta.getNome() + " - " + fruta.getCor());      }  } |

Na Listagem 5, encontra-se um exemplo de serialização e deserialização de um ArrayList da classe Fruta. É apenas mais um exemplo que demonstra a capacidade de se persistir qualquer objeto que seja serializável. Essa serialização só foi possível porque a classe “ArrayList” implementa a interface “Serializable” por padrão.

Listagem 5 - Serialização de ArrayList.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | //serialização e deserialização com ArrayList          ArrayList<Fruta> frutas = new ArrayList<Fruta>();          Fruta f1 = new Fruta("Laranja", "Amarela");          Fruta f2 = new Fruta("Abacate", "Verde");          Fruta f3 = new Fruta("Morango", "Vermelho");          frutas.add(f1);          frutas.add(f2);          frutas.add(f3);            try {              s.serializar("/home/gustavo/frutas", frutas);              frutas = null;              frutas = (ArrayList<Fruta>) d.deserializar("/home/gustavo/frutas");              for(Fruta f : frutas){                  System.out.println("ArrayList: " + f.getNome() + " - " + f.getCor());              }          } catch (Exception ex) {              System.err.println("Falha ao serializar ou deserializar! - " +                                 ex.toString());          } |

**4. PARTICULARIDADES DA SERIALIZAÇÃO**

Como dito, a serialização de um objeto em Java pode ser algo fácil de se fazer. Mas existem certas informações que são importantes e sabê-las pode evitar vários problemas.

Um dos problemas é que nem tudo pode ser serializado, como objetos das classes “Thread” e “Socket” e atributos estáticos. Os atributos estáticos estão vinculados à classe e não ao objeto. Assim quando se trata do estado do objeto, os campos estáticos não são considerados. Em relação às Threads, suas instâncias só são úteis enquanto estiverem em execução na JVM, e podem ser paradas/destruídas e iniciadas com instruções de sua classe. Não existe algo no estado destas instâncias que seja útil para se persistir, e por isso não há sentido em persistir tais instâncias para restaurar posteriormente.

Mas e se existir alguma classe em que seja necessário persisti-la, e esta possui uma instância da classe “Thread”? Neste caso, utilizando um recurso pode-se persistir a classe inteira, menos a instância thread. Este recurso, refere-se em indicar que tal instância thread é um objeto “Transient”. Na Listagem 6 encontra-se um exemplo da implementação com uso de “Transient”.

Listagem 6 - Uso do Transient

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | import java.io.Serializable;    public class ImplementacaoTransient implements Serializable, Runnable{        transient private Thread thread;      //marcado com "transient" para que sua instância não seja serializada        //Demais valores de estado serão serializados      private int valorParaPersistir;        public ImplementacaoTransient(int valorParaPersistir){          this.valorParaPersistir = valorParaPersistir;          thread = new Thread(this);          thread.start();      }        public void run()      {          while(true){              //código a ser executado pela Thread          }      }  } |

Vale ressaltar que, o que é serializado é apenas o estado do objeto, e não os métodos e a classe. No exemplo acima, apenas a instância do objeto “thread” e do objeto “valorParaPersistir” compunham o estado do objeto. Assim, evita-se que o sistema de serialização tente serializar um objeto não-serializável e gere erros. Após a deserialização de um objeto da classe “ImplementacaoTransient” este estará totalmente funcional.

Um outro problema que pode vir a ocorrer, se refere ao versionamento das classes. Toda classe possui um atributo oculto chamado “serialVersionUID” com um valor gerado automaticamente. Objetos criados levam consigo o “serialVersionUID” de sua classe, que indica sua versão. Toda vez que uma classe sofre alterações em seu código, um novo “serialVersionUID” é gerado. O problema com a serialização surge quando serializa-se o estado de um objeto e posteriormente tenta-se restaurar o estado para a aplicação, mas houveram mudanças no código da classe. Neste caso, uma exceção “java.io.InvalidClassException” é disparada, pois de fato, o estado do objeto corresponde a uma versão de classe do qual foi criado que já não existe mais. Portanto, isso só ocorre para evitar erros de programação, em que tenta-se restaurar o estado de um objeto para uma classe que possui versão diferente da qual o objeto foi serializado inicialmente. E isso pode significar diferentes versões de uma mesma classe “Fruta” por exemplo, ou diferentes versões por classes serem totalmente diferentes como: o estado de um objeto da classe “Fruta” ser restaurado para classe “Carro”.

Mas quando é necessário, conscientemente, recuperar o estado de um objeto, mesmo que houveram alterações na classe do qual foi gerado, é possível controlar manualmente o “serialVersionUID”. Para isto, basta declarar explicitamente na forma de atributo estático e final, um valor Long qualquer para corresponder à versão da classe. E assim, mesmo que houverem modificações na classe, a versão permenecerá a mesma, evitando as exceções da incompatiblidade de classes. No entanto, vale ressaltar que o versionamento não é um impecílio para o programador, mas sim muito importante em casos que deseja-se manter um controle sobre a compatibilidade de código na aplicação, definindo versões para classes No código da Listagem 7 encontra-se o exemplo do controle manual do “serialVersionUID”.

Listagem 7 - Uso manual do serialVersionUID

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | import java.io.Serializable;    public class Fruta implements Serializable{      private String nome;      private String cor;      static final long serialVersionUID = 123L; //indica a versao da classe        public Fruta(String nome, String cor) {          this.nome = nome;          this.cor = cor;      }      public String getCor() { return cor; }      public String getNome() { return nome; }      public void setCor(String cor) { this.cor = cor; }      public void setNome(String nome) { this.nome = nome; }  } |

O JDK fornece uma ferramenta que permite visualizar a versão de qualquer classe Java. Basta no terminal do sistema operacional utilizar o comando “serialver” seguido da classe qualificada, conforme exemplo da Listagem 8.

Listagem 8 - Verificação da versão de classe

|  |  |
| --- | --- |
| 1 | serialver pojo.Fruta |

A serialização acaba inflingindo alguns modelos de segurança em seu procedimento padrão, e criando então um novo problema. Quando um estado de objeto é persistido no sistema de arquivos ou enviado por rede, o fluxo binário pode ser detectado, analisado e obtidas informações sobre a classe e o estado do objeto. Por exemplo, se utilizado um editor de texto qualquer para abrir o arquivo persistido do estado do objeto da classe “Fruta” dos exemplos anteriores, é comum os atributos e outras informações aparecerem como um simples texto, total ou parcilamente legíveis, assim como demonstrado na Listagem 9.

Listagem 9 - Exemplo de leitura de arquivo binário de um estado de objeto persistido.

|  |  |
| --- | --- |
| 1  2 | ��##sr#  pojo.Fruta#######{###L##cort##Ljava/lang/String;L##nomeq#~##xpt##Vermelhat##Maça |

Com uma ferramenta menos simples seria possível aumentar a quantidade de informações legíveis, coletando uma maior quantia de dados.

No entando, o processo de serialização permite que haja uma certa personalização da serialização e deserialização de um objeto. Isso deve-se a possibilidade da sobrescrição dos métodos “writeObject” e “readObject” dentro da classe a ser serializada, como a classe exemplar “pojo.Fruta”. Esta sobrescrita permite um controle total sobre a serialização, pois as classes “java.io.ObjectOutputStream” e “java.io.ObjectInputStream”, ao serializar o estado de um objeto, primeiramente verificam se esse método foi sobrescrito na classe alvo e assim o utilizam, senão, será utilizado o protocolo padrão de serialização.

Realizando a sobrescrição dos métodos é possível criptografar os dados na serialização e descriptografar os dados na deserialização, por meio de algum algoritmo que o programador venha implementar. Assim os dados podem ser persistidos ou transmitidos por rede, sem que seja possível ler as partes mais importantes do estado de um objeto. Na listagem 10, a classe “pojo.Fruta” foi modificada dando origem a classe “pojo.FrutaSobrescrita” que possui adicionamente a sobrescrição dos métodos “writeObject” e “readObject”. Como este é apenas um exemplo, não foi escrito um verdadeiro algoritmo de criptografia, mas sim apenas uma troca no valor da String do campo “cor”, identificando que houveram alterações tanto no processo de serialização quanto no de deserialização. Na listagem 11 encontra-se a execução da serialização.

Listagem 10 - Sobrescrita dos métodos “writeObject” e “readObject”.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36 | import java.io.Serializable;    public class FrutaSobrescrita implements Serializable{      private String nome;      private String cor;      static final long serialVersionUID = 123L; //indica a versao da classe        public FrutaSobrescrita(String nome, String cor) {          this.nome = nome;          this.cor = cor;      }      public String getCor() { return cor; }      public String getNome() { return nome; }      public void setCor(String cor) { this.cor = cor; }      public void setNome(String nome) { this.nome = nome; }        private void writeObject(java.io.ObjectOutputStream stream) throws java.io.IOException      {          // criptografar os dados antes de liberar o stream          // EXEMPLO: setCor(criptografar(getCor()));          setCor(getCor()+ " > CRIPTOGRAFADO"); // apenas concatenação de string para indicar alteração            // liberar o stream          stream.defaultWriteObject();      }        private void readObject(java.io.ObjectInputStream stream) throws java.io.IOException, ClassNotFoundException      {          // liberar stream          stream.defaultReadObject();            // descriptografar os dados          // EXEMPLO: setCor(descriptografar(getCor()));          setCor(getCor()+ " > DESCRIPTOGRAFADO"); // apenas contatenação de string para indicar alteração      }  } |

Listagem 11 - Execução da serialização da classe “pojo.FrutaSobrescrita”.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | //serializa        Serializador s = new Serializador();        FrutaSobrescrita frutaSobrescrita = new FrutaSobrescrita("Maça", "Vermelha");         try {             s.serializar("/home/gustavo/frutaSobrescrita", frutaSobrescrita);         } catch (Exception ex) {             System.err.println("Falha ao serializar! - " + ex.toString());         }           //deserializa        Deserializador d = new Deserializador();         frutaSobrescrita = null;         try {             frutaSobrescrita = (FrutaSobrescrita) d.deserializar("/home/gustavo/frutaSobrescrita");         } catch (Exception ex) {             System.err.println("Falha ao deserializar! - " + ex.toString());         }         System.out.println("Sobrescrita: " + frutaSobrescrita.getNome() + " - " + frutaSobrescrita.getCor()); |

**5. CONCLUSÃO**

A serialização é um procedimento de simples execução por parte do programador. Com pouco tempo de codificação é possível implementar a persistencia de um estado de objeto para o sistema de arquivos ou de transmissão por rede. É possível reproduzir os exemplos criados neste artigo, em situações até mesmo mais elaboradas. A apresentação de alguns problemas e soluções trazidas por particularidades, abre o entendimento de que a serialização pode deixar de ser simples conforme for aplicada em diferentes contextos, e que ao mesmo tempo é muito flexível ao permitir ao programador implementar diferentes soluções.

[Assista ao vídeo Serialização/Deserialização de objetos com YAML](http://www.devmedia.com.br/serializacao-deserializacao-de-objetos-com-yaml-parte-1/8399)

**6. BIBLIOGRAFIA**

ORACLE - **java.io Interface Serializable** - Disponível em: <http://docs.oracle.com/javase/1.4.2/docs/api/java/io/Serializable.html>Acesso em: 11/03/2012

ORACLE - **Discover de Secrets of Java Serialization API** - Disponível em: <http://java.sun.com/developer/technicalArticles/Programming/serialization/> Acesso em: 11/03/2012

NEWARD, Ted - **5 coisas que você não sabia sobre... Serialização de Objetos Java** - Disponível em: <http://www.ibm.com/developerworks/br/library/j-5things1/index.html> Acesso em: 11/03/2012

DOWNS, Andrew - **Java Serialization** - Disponível em: <http://www.mactech.com/articles/mactech/Vol.14/14.04/JavaSerialization/index.html> Acesso em: 11/03/2012

PREISSLER, Luciano E.; GEYER, Cláudio F. R. - **Serialização em Java** - Disponível em: <http://www.inf.ufrgs.br/gppd/disc/cmp167/trabalhos/sem99-1/T1/luciano/final.htm> Acesso em: 11/03/2012

[[](http://www.linhadecodigo.com.br/colaborador.aspx?id=354)](http://www.linhadecodigo.com.br/colaborador.aspx?id=354)

[Everton Coimbra de Araújo - Desde 1987 atua na área de treinamento e desenvolvimento. Como Mestre em Ciência da Computação, é professor da UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ, Campus Medianeira, onde leciona disciplinas relacionadas ao desenvolvimento de aplicações web, com Java e .NET.   
  
É autor dos livros Desenvolvimento para WEB com Java, Orientação a Objetos com Java - Simples, Fácil e Eficiente, Algoritmo - Fundamento e Prática em sua terceira edição e dos livros Delphi - Implementação e Técnicas para Ambientes Virtuais e C++ Builder - Implementação e Técnicas para Ambientes Virtuais. Todos pela VisualBooks. Pode ser contactado através do e-mail everton@utfpr.edu.br ou evertoncoimbra@gmail.com.](http://www.linhadecodigo.com.br/colaborador.aspx?id=354)

 85 3 0 4

**Leia também**

* [Conhecendo a interface Map do Java](http://www.linhadecodigo.com.br/artigo/3670/conhecendo-a-interface-map-do-java.aspx)*Java*
* [Trabalhando com a Interface Set no Java](http://www.linhadecodigo.com.br/artigo/3669/trabalhando-com-a-interface-set-no-java.aspx)*Java*
* [Classes Wrappers em Java](http://www.linhadecodigo.com.br/artigo/3667/classes-wrappers-em-java.aspx)*Java*
* [ComboBoxModel em Java](http://www.linhadecodigo.com.br/artigo/3664/comboboxmodel-em-java.aspx)*Java*
* [Reduzindo a Mutabilidade em Java](http://www.linhadecodigo.com.br/artigo/3659/reduzindo-a-mutabilidade-em-java.aspx)*Java*

Publicidade

[](http://qconsp.com/?utm_source=devmedia&utm_medium=banner&utm_campaign=devmedia)

Linha de Código[Anuncie](http://www.web-03.net/linhadecodigo.asp)|[Fale Conosco](mailto:colabore@linhadecodigo.com.br)|[Publique](http://www.linhadecodigo.com.br/publique.aspx)

|  |  |
| --- | --- |
| http://www.linhadecodigo.com.br/artigos/img_artigos/EvertonCoimbraAraujo/Serializazao_Objetos/imagem001.jpg | Neste artigo apresento o Gustavo Rafael Valiati, que pode ser contatado pelo e-mail gustavovaliati@gmail.com. O Gustavo foi meu aluno em algumas disciplinas de sua gradução e está concluíndo o último período do curso de Tecnologia em Análise e Desenvolvimento de Sistemas da UTFPR campus Medianeira, neste primeiro semestre de 2012. |

**1. INTRODUÇÃO**

Nem sempre se deseja ter objetos ou estrutura de dados de um aplicativo, com um ciclo de vida que dura no máximo o tempo em que for armazenado em memória RAM. Ou seja, algumas vezes deseja-se guardá-los por um tempo indeterminado, sendo possível recuperá-los posteriormente. E neste caso, o termo “persistência” tem seu conceito definido.

Persistir dados, requer que estes possam ser serializáveis. A serialização possibilitará capturar o estado do objeto ou a estrutura de dados, e transformar em uma cadeia de bytes quando for necessário. Torna-se possível também, recuperar os bytes persistidos para fazer o processo inverso, e ter os dados de volta para a aplicação em execução.

Desta maneira, qualquer forma de dados que esteja sendo trabalhada na aplicação, e esta puder ser serializada, poderá ser passada para o meio físico, e ter o processo revertido.

A serialização não se limita apenas em possibilitar a gravação do dado em disco, mas permite que o mesmo seja transmitido por rede. Isto porquê a serialização cria um stream de bytes, que é requisito para transmissão de dados por rede.

Diversas linguagens possuem suporte à serialização, como: Java, C, C++, Python, PHP, .NET, entre outras. Neste artigo, será tratado apenas da serialização com Java, em sua implemetação básica, alguns problemas e soluções, com códigos para exemplificação.

**2. SERIALIZAÇÃO COM JAVA**

Serializar um objeto, em Java, só é possível caso sua classe esteja marcada como serializável. Essa marcação é feita com a simples implementação da interface “java.io.Serializable”. Essa interface não possui métodos para se implementar e nem atributos. Ainda não existiam as anotações no Java 1.1, versão que trouxe pela primeira vez a serialização ao Java, e desta maneira a implementação da interface era a melhor maneira de marcar uma classe.

A herança entre classes é naturalmente afetada pela serialização. Já que quando uma classe implementa a interface “Serializable”, toda classe que dela estender também estará implicitamente marcada como serializável

.

Para exemplo, será criada uma classe “pojo.Fruta” que contém dois atributos: nome e cor. No código exemplo (Listagem 1), pode-se notar a classe “Fruta” marcada para a serialização:

Listagem 1 - Marcando uma classe como serializável.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | import java.io.Serializable;    public class Fruta implements Serializable{      private String nome;      private String cor;        public Fruta(String nome, String cor) {          this.nome = nome;          this.cor = cor;      }        public String getCor() { return cor; }      public String getNome() { return nome; }      public void setCor(String cor) { this.cor = cor; }      public void setNome(String nome) { this.nome = nome; }  } |

Pode-se dizer que a implementação do processo de serialização e deserialização simples de um objeto, não possui um código difícil de se escrever. Isto porquê duas classes fazem a maior parte do trabalho, de maneira que com poucas linhas de código pode-se transformar um objeto em bytes e vice-versa. Essas duas classes são “java.io.ObjectInputStream” que recebe bytes para criar um objeto e “java.io.ObjectOutputStream” que cria uma cadeia de bytes a partir de um objeto.

**3. SERIALIZANDO E DESERIALIZANDO O ESTADO DE UM OBJETO**

Como exemplo, será persistido um objeto em disco. Para isto, além das classes “java.io.ObjectInputStream” e “java.io.ObjectOutputStream”, serão utilizadas duas outras que trabalham com arquivos, que são “java.io.FileInputStream” e “java.io.FileOutputStream”, que seguem a mesma lógica de InputStream e OutputStream, só que agora relacionado ao sistema de arquivos e não aos objetos.

Para serialização, deve-se criar um novo “java.io.FileOutputStream”, com o caminho (“path”) do arquivo onde se deseja armazenar o estado do objeto. Também deve ser criado um novo “java.io.ObjectOutputStream” que está ligado ao arquivo. E por último deve-se executar a escrita do objeto alvo, para dentro do objeto de stream que por sua vez está ligado ao arquivo destino. Após executados esses passos, é possível encontrar no sistema de arquivos um novo arquivo que contém os bytes correspondentes à descrição do estado do objeto. Na Listagem 2, encontra-se o código exemplo.

Listagem 2 - Serialização de um objeto.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | import java.io.FileOutputStream;  import java.io.ObjectOutputStream;    public class Serializador {        public Serializador() {    }        public void serializar(String path, Object obj) throws Exception {              FileOutputStream outFile = new FileOutputStream(path);      ObjectOutputStream s = new ObjectOutputStream(outFile);      s.writeObject(obj);      s.close();      }  } |

E para a deserialização, o processo é extremamente parecido com a serialização. Cria-se um novo “java.io.FileInputStream” relacionado ao arquivo onde o estado do objeto encontra-se persistido. Deve-se criar também um novo “java.io.ObjectInputStream” que estará vinculado ao arquivo de origem. Por último, executa-se a leitura do arquivo para a restauração do estado. Na Listagem 3 encontra-se o exemplo do código.

Listagem 3 - Deserializaçãode um objeto.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | import java.io.FileInputStream;  import java.io.ObjectInputStream;    public class Deserializador {        public Deserializador() {    }        public Object deserializar(String path) throws Exception {      FileInputStream inFile = new FileInputStream(path);      ObjectInputStream d = new ObjectInputStream(inFile);      Object o = d.readObject();      d.close();      return o;      }  } |

A Listagem 4, contém o exemplo da execução da serialização e em seguida da deserialização.

Listagem 4 - Serializando e Deserializando

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | import operacoes.Deserializador;  import operacoes.Serializador;  import pojo.Fruta;    public class Principal {      public static void main(String args[]){         //serializa         Serializador s = new Serializador();         Fruta fruta = new Fruta("Maça", "Vermelha");          try {              s.serializar("/home/gustavo/fruta", fruta);          } catch (Exception ex) {              System.err.println("Falha ao serializar! - " + ex.toString());          }            //deserializa          Deserializador d = new Deserializador();          fruta = null;          try {              fruta = (Fruta) d.deserializar("/home/gustavo/fruta");          } catch (Exception ex) {              System.err.println("Falha ao deserializar! - " + ex.toString());          }          System.out.println(fruta.getNome() + " - " + fruta.getCor());      }  } |

Na Listagem 5, encontra-se um exemplo de serialização e deserialização de um ArrayList da classe Fruta. É apenas mais um exemplo que demonstra a capacidade de se persistir qualquer objeto que seja serializável. Essa serialização só foi possível porque a classe “ArrayList” implementa a interface “Serializable” por padrão.

Listagem 5 - Serialização de ArrayList.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | //serialização e deserialização com ArrayList          ArrayList<Fruta> frutas = new ArrayList<Fruta>();          Fruta f1 = new Fruta("Laranja", "Amarela");          Fruta f2 = new Fruta("Abacate", "Verde");          Fruta f3 = new Fruta("Morango", "Vermelho");          frutas.add(f1);          frutas.add(f2);          frutas.add(f3);            try {              s.serializar("/home/gustavo/frutas", frutas);              frutas = null;              frutas = (ArrayList<Fruta>) d.deserializar("/home/gustavo/frutas");              for(Fruta f : frutas){                  System.out.println("ArrayList: " + f.getNome() + " - " + f.getCor());              }          } catch (Exception ex) {              System.err.println("Falha ao serializar ou deserializar! - " +                                 ex.toString());          } |

**4. PARTICULARIDADES DA SERIALIZAÇÃO**

Como dito, a serialização de um objeto em Java pode ser algo fácil de se fazer. Mas existem certas informações que são importantes e sabê-las pode evitar vários problemas.

Um dos problemas é que nem tudo pode ser serializado, como objetos das classes “Thread” e “Socket” e atributos estáticos. Os atributos estáticos estão vinculados à classe e não ao objeto. Assim quando se trata do estado do objeto, os campos estáticos não são considerados. Em relação às Threads, suas instâncias só são úteis enquanto estiverem em execução na JVM, e podem ser paradas/destruídas e iniciadas com instruções de sua classe. Não existe algo no estado destas instâncias que seja útil para se persistir, e por isso não há sentido em persistir tais instâncias para restaurar posteriormente.

Mas e se existir alguma classe em que seja necessário persisti-la, e esta possui uma instância da classe “Thread”? Neste caso, utilizando um recurso pode-se persistir a classe inteira, menos a instância thread. Este recurso, refere-se em indicar que tal instância thread é um objeto “Transient”. Na Listagem 6 encontra-se um exemplo da implementação com uso de “Transient”.

Listagem 6 - Uso do Transient

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | import java.io.Serializable;    public class ImplementacaoTransient implements Serializable, Runnable{        transient private Thread thread;      //marcado com "transient" para que sua instância não seja serializada        //Demais valores de estado serão serializados      private int valorParaPersistir;        public ImplementacaoTransient(int valorParaPersistir){          this.valorParaPersistir = valorParaPersistir;          thread = new Thread(this);          thread.start();      }        public void run()      {          while(true){              //código a ser executado pela Thread          }      }  } |

Vale ressaltar que, o que é serializado é apenas o estado do objeto, e não os métodos e a classe. No exemplo acima, apenas a instância do objeto “thread” e do objeto “valorParaPersistir” compunham o estado do objeto. Assim, evita-se que o sistema de serialização tente serializar um objeto não-serializável e gere erros. Após a deserialização de um objeto da classe “ImplementacaoTransient” este estará totalmente funcional.

Um outro problema que pode vir a ocorrer, se refere ao versionamento das classes. Toda classe possui um atributo oculto chamado “serialVersionUID” com um valor gerado automaticamente. Objetos criados levam consigo o “serialVersionUID” de sua classe, que indica sua versão. Toda vez que uma classe sofre alterações em seu código, um novo “serialVersionUID” é gerado. O problema com a serialização surge quando serializa-se o estado de um objeto e posteriormente tenta-se restaurar o estado para a aplicação, mas houveram mudanças no código da classe. Neste caso, uma exceção “java.io.InvalidClassException” é disparada, pois de fato, o estado do objeto corresponde a uma versão de classe do qual foi criado que já não existe mais. Portanto, isso só ocorre para evitar erros de programação, em que tenta-se restaurar o estado de um objeto para uma classe que possui versão diferente da qual o objeto foi serializado inicialmente. E isso pode significar diferentes versões de uma mesma classe “Fruta” por exemplo, ou diferentes versões por classes serem totalmente diferentes como: o estado de um objeto da classe “Fruta” ser restaurado para classe “Carro”.

Mas quando é necessário, conscientemente, recuperar o estado de um objeto, mesmo que houveram alterações na classe do qual foi gerado, é possível controlar manualmente o “serialVersionUID”. Para isto, basta declarar explicitamente na forma de atributo estático e final, um valor Long qualquer para corresponder à versão da classe. E assim, mesmo que houverem modificações na classe, a versão permenecerá a mesma, evitando as exceções da incompatiblidade de classes. No entanto, vale ressaltar que o versionamento não é um impecílio para o programador, mas sim muito importante em casos que deseja-se manter um controle sobre a compatibilidade de código na aplicação, definindo versões para classes No código da Listagem 7 encontra-se o exemplo do controle manual do “serialVersionUID”.

Listagem 7 - Uso manual do serialVersionUID

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | import java.io.Serializable;    public class Fruta implements Serializable{      private String nome;      private String cor;      static final long serialVersionUID = 123L; //indica a versao da classe        public Fruta(String nome, String cor) {          this.nome = nome;          this.cor = cor;      }      public String getCor() { return cor; }      public String getNome() { return nome; }      public void setCor(String cor) { this.cor = cor; }      public void setNome(String nome) { this.nome = nome; }  } |

O JDK fornece uma ferramenta que permite visualizar a versão de qualquer classe Java. Basta no terminal do sistema operacional utilizar o comando “serialver” seguido da classe qualificada, conforme exemplo da Listagem 8.

Listagem 8 - Verificação da versão de classe

|  |  |
| --- | --- |
| 1 | serialver pojo.Fruta |

A serialização acaba inflingindo alguns modelos de segurança em seu procedimento padrão, e criando então um novo problema. Quando um estado de objeto é persistido no sistema de arquivos ou enviado por rede, o fluxo binário pode ser detectado, analisado e obtidas informações sobre a classe e o estado do objeto. Por exemplo, se utilizado um editor de texto qualquer para abrir o arquivo persistido do estado do objeto da classe “Fruta” dos exemplos anteriores, é comum os atributos e outras informações aparecerem como um simples texto, total ou parcilamente legíveis, assim como demonstrado na Listagem 9.

Listagem 9 - Exemplo de leitura de arquivo binário de um estado de objeto persistido.

|  |  |
| --- | --- |
| 1  2 | ��##sr#  pojo.Fruta#######{###L##cort##Ljava/lang/String;L##nomeq#~##xpt##Vermelhat##Maça |

Com uma ferramenta menos simples seria possível aumentar a quantidade de informações legíveis, coletando uma maior quantia de dados.

No entando, o processo de serialização permite que haja uma certa personalização da serialização e deserialização de um objeto. Isso deve-se a possibilidade da sobrescrição dos métodos “writeObject” e “readObject” dentro da classe a ser serializada, como a classe exemplar “pojo.Fruta”. Esta sobrescrita permite um controle total sobre a serialização, pois as classes “java.io.ObjectOutputStream” e “java.io.ObjectInputStream”, ao serializar o estado de um objeto, primeiramente verificam se esse método foi sobrescrito na classe alvo e assim o utilizam, senão, será utilizado o protocolo padrão de serialização.

Realizando a sobrescrição dos métodos é possível criptografar os dados na serialização e descriptografar os dados na deserialização, por meio de algum algoritmo que o programador venha implementar. Assim os dados podem ser persistidos ou transmitidos por rede, sem que seja possível ler as partes mais importantes do estado de um objeto. Na listagem 10, a classe “pojo.Fruta” foi modificada dando origem a classe “pojo.FrutaSobrescrita” que possui adicionamente a sobrescrição dos métodos “writeObject” e “readObject”. Como este é apenas um exemplo, não foi escrito um verdadeiro algoritmo de criptografia, mas sim apenas uma troca no valor da String do campo “cor”, identificando que houveram alterações tanto no processo de serialização quanto no de deserialização. Na listagem 11 encontra-se a execução da serialização.

Listagem 10 - Sobrescrita dos métodos “writeObject” e “readObject”.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36 | import java.io.Serializable;    public class FrutaSobrescrita implements Serializable{      private String nome;      private String cor;      static final long serialVersionUID = 123L; //indica a versao da classe        public FrutaSobrescrita(String nome, String cor) {          this.nome = nome;          this.cor = cor;      }      public String getCor() { return cor; }      public String getNome() { return nome; }      public void setCor(String cor) { this.cor = cor; }      public void setNome(String nome) { this.nome = nome; }        private void writeObject(java.io.ObjectOutputStream stream) throws java.io.IOException      {          // criptografar os dados antes de liberar o stream          // EXEMPLO: setCor(criptografar(getCor()));          setCor(getCor()+ " > CRIPTOGRAFADO"); // apenas concatenação de string para indicar alteração            // liberar o stream          stream.defaultWriteObject();      }        private void readObject(java.io.ObjectInputStream stream) throws java.io.IOException, ClassNotFoundException      {          // liberar stream          stream.defaultReadObject();            // descriptografar os dados          // EXEMPLO: setCor(descriptografar(getCor()));          setCor(getCor()+ " > DESCRIPTOGRAFADO"); // apenas contatenação de string para indicar alteração      }  } |

Listagem 11 - Execução da serialização da classe “pojo.FrutaSobrescrita”.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | //serializa        Serializador s = new Serializador();        FrutaSobrescrita frutaSobrescrita = new FrutaSobrescrita("Maça", "Vermelha");         try {             s.serializar("/home/gustavo/frutaSobrescrita", frutaSobrescrita);         } catch (Exception ex) {             System.err.println("Falha ao serializar! - " + ex.toString());         }           //deserializa        Deserializador d = new Deserializador();         frutaSobrescrita = null;         try {             frutaSobrescrita = (FrutaSobrescrita) d.deserializar("/home/gustavo/frutaSobrescrita");         } catch (Exception ex) {             System.err.println("Falha ao deserializar! - " + ex.toString());         }         System.out.println("Sobrescrita: " + frutaSobrescrita.getNome() + " - " + frutaSobrescrita.getCor()); |

**5. CONCLUSÃO**

A serialização é um procedimento de simples execução por parte do programador. Com pouco tempo de codificação é possível implementar a persistencia de um estado de objeto para o sistema de arquivos ou de transmissão por rede. É possível reproduzir os exemplos criados neste artigo, em situações até mesmo mais elaboradas. A apresentação de alguns problemas e soluções trazidas por particularidades, abre o entendimento de que a serialização pode deixar de ser simples conforme for aplicada em diferentes contextos, e que ao mesmo tempo é muito flexível ao permitir ao programador implementar diferentes soluções.

[Assista ao vídeo Serialização/Deserialização de objetos com YAML](http://www.devmedia.com.br/serializacao-deserializacao-de-objetos-com-yaml-parte-1/8399)

**6. BIBLIOGRAFIA**

ORACLE - **java.io Interface Serializable** - Disponível em: <http://docs.oracle.com/javase/1.4.2/docs/api/java/io/Serializable.html>Acesso em: 11/03/2012

ORACLE - **Discover de Secrets of Java Serialization API** - Disponível em: <http://java.sun.com/developer/technicalArticles/Programming/serialization/> Acesso em: 11/03/2012

NEWARD, Ted - **5 coisas que você não sabia sobre... Serialização de Objetos Java** - Disponível em: <http://www.ibm.com/developerworks/br/library/j-5things1/index.html> Acesso em: 11/03/2012

DOWNS, Andrew - **Java Serialization** - Disponível em: <http://www.mactech.com/articles/mactech/Vol.14/14.04/JavaSerialization/index.html> Acesso em: 11/03/2012

PREISSLER, Luciano E.; GEYER, Cláudio F. R. - **Serialização em Java** - Disponível em: <http://www.inf.ufrgs.br/gppd/disc/cmp167/trabalhos/sem99-1/T1/luciano/final.htm> Acesso em: 11/03/2012

[[](http://www.linhadecodigo.com.br/colaborador.aspx?id=354)](http://www.linhadecodigo.com.br/colaborador.aspx?id=354)

[Everton Coimbra de Araújo - Desde 1987 atua na área de treinamento e desenvolvimento. Como Mestre em Ciência da Computação, é professor da UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ, Campus Medianeira, onde leciona disciplinas relacionadas ao desenvolvimento de aplicações web, com Java e .NET.   
  
É autor dos livros Desenvolvimento para WEB com Java, Orientação a Objetos com Java - Simples, Fácil e Eficiente, Algoritmo - Fundamento e Prática em sua terceira edição e dos livros Delphi - Implementação e Técnicas para Ambientes Virtuais e C++ Builder - Implementação e Técnicas para Ambientes Virtuais. Todos pela VisualBooks. Pode ser contactado através do e-mail everton@utfpr.edu.br ou evertoncoimbra@gmail.com.](http://www.linhadecodigo.com.br/colaborador.aspx?id=354)

 85 3 0 4

**Leia também**

* [Conhecendo a interface Map do Java](http://www.linhadecodigo.com.br/artigo/3670/conhecendo-a-interface-map-do-java.aspx)*Java*
* [Trabalhando com a Interface Set no Java](http://www.linhadecodigo.com.br/artigo/3669/trabalhando-com-a-interface-set-no-java.aspx)*Java*
* [Classes Wrappers em Java](http://www.linhadecodigo.com.br/artigo/3667/classes-wrappers-em-java.aspx)*Java*
* [ComboBoxModel em Java](http://www.linhadecodigo.com.br/artigo/3664/comboboxmodel-em-java.aspx)*Java*
* [Reduzindo a Mutabilidade em Java](http://www.linhadecodigo.com.br/artigo/3659/reduzindo-a-mutabilidade-em-java.aspx)*Java*

Publicidade

[](http://qconsp.com/?utm_source=devmedia&utm_medium=banner&utm_campaign=devmedia)

Linha de Código[Anuncie](http://www.web-03.net/linhadecodigo.asp)|[Fale Conosco](mailto:colabore@linhadecodigo.com.br)|[Publique](http://www.linhadecodigo.com.br/publique.aspx)

|  |  |
| --- | --- |
| http://www.linhadecodigo.com.br/artigos/img_artigos/EvertonCoimbraAraujo/Serializazao_Objetos/imagem001.jpg | Neste artigo apresento o Gustavo Rafael Valiati, que pode ser contatado pelo e-mail gustavovaliati@gmail.com. O Gustavo foi meu aluno em algumas disciplinas de sua gradução e está concluíndo o último período do curso de Tecnologia em Análise e Desenvolvimento de Sistemas da UTFPR campus Medianeira, neste primeiro semestre de 2012. |

**1. INTRODUÇÃO**

Nem sempre se deseja ter objetos ou estrutura de dados de um aplicativo, com um ciclo de vida que dura no máximo o tempo em que for armazenado em memória RAM. Ou seja, algumas vezes deseja-se guardá-los por um tempo indeterminado, sendo possível recuperá-los posteriormente. E neste caso, o termo “persistência” tem seu conceito definido.

Persistir dados, requer que estes possam ser serializáveis. A serialização possibilitará capturar o estado do objeto ou a estrutura de dados, e transformar em uma cadeia de bytes quando for necessário. Torna-se possível também, recuperar os bytes persistidos para fazer o processo inverso, e ter os dados de volta para a aplicação em execução.

Desta maneira, qualquer forma de dados que esteja sendo trabalhada na aplicação, e esta puder ser serializada, poderá ser passada para o meio físico, e ter o processo revertido.

A serialização não se limita apenas em possibilitar a gravação do dado em disco, mas permite que o mesmo seja transmitido por rede. Isto porquê a serialização cria um stream de bytes, que é requisito para transmissão de dados por rede.

Diversas linguagens possuem suporte à serialização, como: Java, C, C++, Python, PHP, .NET, entre outras. Neste artigo, será tratado apenas da serialização com Java, em sua implemetação básica, alguns problemas e soluções, com códigos para exemplificação.

**2. SERIALIZAÇÃO COM JAVA**

Serializar um objeto, em Java, só é possível caso sua classe esteja marcada como serializável. Essa marcação é feita com a simples implementação da interface “java.io.Serializable”. Essa interface não possui métodos para se implementar e nem atributos. Ainda não existiam as anotações no Java 1.1, versão que trouxe pela primeira vez a serialização ao Java, e desta maneira a implementação da interface era a melhor maneira de marcar uma classe.

A herança entre classes é naturalmente afetada pela serialização. Já que quando uma classe implementa a interface “Serializable”, toda classe que dela estender também estará implicitamente marcada como serializável

.

Para exemplo, será criada uma classe “pojo.Fruta” que contém dois atributos: nome e cor. No código exemplo (Listagem 1), pode-se notar a classe “Fruta” marcada para a serialização:

Listagem 1 - Marcando uma classe como serializável.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | import java.io.Serializable;    public class Fruta implements Serializable{      private String nome;      private String cor;        public Fruta(String nome, String cor) {          this.nome = nome;          this.cor = cor;      }        public String getCor() { return cor; }      public String getNome() { return nome; }      public void setCor(String cor) { this.cor = cor; }      public void setNome(String nome) { this.nome = nome; }  } |

Pode-se dizer que a implementação do processo de serialização e deserialização simples de um objeto, não possui um código difícil de se escrever. Isto porquê duas classes fazem a maior parte do trabalho, de maneira que com poucas linhas de código pode-se transformar um objeto em bytes e vice-versa. Essas duas classes são “java.io.ObjectInputStream” que recebe bytes para criar um objeto e “java.io.ObjectOutputStream” que cria uma cadeia de bytes a partir de um objeto.

**3. SERIALIZANDO E DESERIALIZANDO O ESTADO DE UM OBJETO**

Como exemplo, será persistido um objeto em disco. Para isto, além das classes “java.io.ObjectInputStream” e “java.io.ObjectOutputStream”, serão utilizadas duas outras que trabalham com arquivos, que são “java.io.FileInputStream” e “java.io.FileOutputStream”, que seguem a mesma lógica de InputStream e OutputStream, só que agora relacionado ao sistema de arquivos e não aos objetos.

Para serialização, deve-se criar um novo “java.io.FileOutputStream”, com o caminho (“path”) do arquivo onde se deseja armazenar o estado do objeto. Também deve ser criado um novo “java.io.ObjectOutputStream” que está ligado ao arquivo. E por último deve-se executar a escrita do objeto alvo, para dentro do objeto de stream que por sua vez está ligado ao arquivo destino. Após executados esses passos, é possível encontrar no sistema de arquivos um novo arquivo que contém os bytes correspondentes à descrição do estado do objeto. Na Listagem 2, encontra-se o código exemplo.

Listagem 2 - Serialização de um objeto.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | import java.io.FileOutputStream;  import java.io.ObjectOutputStream;    public class Serializador {        public Serializador() {    }        public void serializar(String path, Object obj) throws Exception {              FileOutputStream outFile = new FileOutputStream(path);      ObjectOutputStream s = new ObjectOutputStream(outFile);      s.writeObject(obj);      s.close();      }  } |

E para a deserialização, o processo é extremamente parecido com a serialização. Cria-se um novo “java.io.FileInputStream” relacionado ao arquivo onde o estado do objeto encontra-se persistido. Deve-se criar também um novo “java.io.ObjectInputStream” que estará vinculado ao arquivo de origem. Por último, executa-se a leitura do arquivo para a restauração do estado. Na Listagem 3 encontra-se o exemplo do código.

Listagem 3 - Deserializaçãode um objeto.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | import java.io.FileInputStream;  import java.io.ObjectInputStream;    public class Deserializador {        public Deserializador() {    }        public Object deserializar(String path) throws Exception {      FileInputStream inFile = new FileInputStream(path);      ObjectInputStream d = new ObjectInputStream(inFile);      Object o = d.readObject();      d.close();      return o;      }  } |

A Listagem 4, contém o exemplo da execução da serialização e em seguida da deserialização.

Listagem 4 - Serializando e Deserializando

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | import operacoes.Deserializador;  import operacoes.Serializador;  import pojo.Fruta;    public class Principal {      public static void main(String args[]){         //serializa         Serializador s = new Serializador();         Fruta fruta = new Fruta("Maça", "Vermelha");          try {              s.serializar("/home/gustavo/fruta", fruta);          } catch (Exception ex) {              System.err.println("Falha ao serializar! - " + ex.toString());          }            //deserializa          Deserializador d = new Deserializador();          fruta = null;          try {              fruta = (Fruta) d.deserializar("/home/gustavo/fruta");          } catch (Exception ex) {              System.err.println("Falha ao deserializar! - " + ex.toString());          }          System.out.println(fruta.getNome() + " - " + fruta.getCor());      }  } |

Na Listagem 5, encontra-se um exemplo de serialização e deserialização de um ArrayList da classe Fruta. É apenas mais um exemplo que demonstra a capacidade de se persistir qualquer objeto que seja serializável. Essa serialização só foi possível porque a classe “ArrayList” implementa a interface “Serializable” por padrão.

Listagem 5 - Serialização de ArrayList.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | //serialização e deserialização com ArrayList          ArrayList<Fruta> frutas = new ArrayList<Fruta>();          Fruta f1 = new Fruta("Laranja", "Amarela");          Fruta f2 = new Fruta("Abacate", "Verde");          Fruta f3 = new Fruta("Morango", "Vermelho");          frutas.add(f1);          frutas.add(f2);          frutas.add(f3);            try {              s.serializar("/home/gustavo/frutas", frutas);              frutas = null;              frutas = (ArrayList<Fruta>) d.deserializar("/home/gustavo/frutas");              for(Fruta f : frutas){                  System.out.println("ArrayList: " + f.getNome() + " - " + f.getCor());              }          } catch (Exception ex) {              System.err.println("Falha ao serializar ou deserializar! - " +                                 ex.toString());          } |

**4. PARTICULARIDADES DA SERIALIZAÇÃO**

Como dito, a serialização de um objeto em Java pode ser algo fácil de se fazer. Mas existem certas informações que são importantes e sabê-las pode evitar vários problemas.

Um dos problemas é que nem tudo pode ser serializado, como objetos das classes “Thread” e “Socket” e atributos estáticos. Os atributos estáticos estão vinculados à classe e não ao objeto. Assim quando se trata do estado do objeto, os campos estáticos não são considerados. Em relação às Threads, suas instâncias só são úteis enquanto estiverem em execução na JVM, e podem ser paradas/destruídas e iniciadas com instruções de sua classe. Não existe algo no estado destas instâncias que seja útil para se persistir, e por isso não há sentido em persistir tais instâncias para restaurar posteriormente.

Mas e se existir alguma classe em que seja necessário persisti-la, e esta possui uma instância da classe “Thread”? Neste caso, utilizando um recurso pode-se persistir a classe inteira, menos a instância thread. Este recurso, refere-se em indicar que tal instância thread é um objeto “Transient”. Na Listagem 6 encontra-se um exemplo da implementação com uso de “Transient”.

Listagem 6 - Uso do Transient

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | import java.io.Serializable;    public class ImplementacaoTransient implements Serializable, Runnable{        transient private Thread thread;      //marcado com "transient" para que sua instância não seja serializada        //Demais valores de estado serão serializados      private int valorParaPersistir;        public ImplementacaoTransient(int valorParaPersistir){          this.valorParaPersistir = valorParaPersistir;          thread = new Thread(this);          thread.start();      }        public void run()      {          while(true){              //código a ser executado pela Thread          }      }  } |

Vale ressaltar que, o que é serializado é apenas o estado do objeto, e não os métodos e a classe. No exemplo acima, apenas a instância do objeto “thread” e do objeto “valorParaPersistir” compunham o estado do objeto. Assim, evita-se que o sistema de serialização tente serializar um objeto não-serializável e gere erros. Após a deserialização de um objeto da classe “ImplementacaoTransient” este estará totalmente funcional.

Um outro problema que pode vir a ocorrer, se refere ao versionamento das classes. Toda classe possui um atributo oculto chamado “serialVersionUID” com um valor gerado automaticamente. Objetos criados levam consigo o “serialVersionUID” de sua classe, que indica sua versão. Toda vez que uma classe sofre alterações em seu código, um novo “serialVersionUID” é gerado. O problema com a serialização surge quando serializa-se o estado de um objeto e posteriormente tenta-se restaurar o estado para a aplicação, mas houveram mudanças no código da classe. Neste caso, uma exceção “java.io.InvalidClassException” é disparada, pois de fato, o estado do objeto corresponde a uma versão de classe do qual foi criado que já não existe mais. Portanto, isso só ocorre para evitar erros de programação, em que tenta-se restaurar o estado de um objeto para uma classe que possui versão diferente da qual o objeto foi serializado inicialmente. E isso pode significar diferentes versões de uma mesma classe “Fruta” por exemplo, ou diferentes versões por classes serem totalmente diferentes como: o estado de um objeto da classe “Fruta” ser restaurado para classe “Carro”.

Mas quando é necessário, conscientemente, recuperar o estado de um objeto, mesmo que houveram alterações na classe do qual foi gerado, é possível controlar manualmente o “serialVersionUID”. Para isto, basta declarar explicitamente na forma de atributo estático e final, um valor Long qualquer para corresponder à versão da classe. E assim, mesmo que houverem modificações na classe, a versão permenecerá a mesma, evitando as exceções da incompatiblidade de classes. No entanto, vale ressaltar que o versionamento não é um impecílio para o programador, mas sim muito importante em casos que deseja-se manter um controle sobre a compatibilidade de código na aplicação, definindo versões para classes No código da Listagem 7 encontra-se o exemplo do controle manual do “serialVersionUID”.

Listagem 7 - Uso manual do serialVersionUID

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | import java.io.Serializable;    public class Fruta implements Serializable{      private String nome;      private String cor;      static final long serialVersionUID = 123L; //indica a versao da classe        public Fruta(String nome, String cor) {          this.nome = nome;          this.cor = cor;      }      public String getCor() { return cor; }      public String getNome() { return nome; }      public void setCor(String cor) { this.cor = cor; }      public void setNome(String nome) { this.nome = nome; }  } |

O JDK fornece uma ferramenta que permite visualizar a versão de qualquer classe Java. Basta no terminal do sistema operacional utilizar o comando “serialver” seguido da classe qualificada, conforme exemplo da Listagem 8.

Listagem 8 - Verificação da versão de classe

|  |  |
| --- | --- |
| 1 | serialver pojo.Fruta |

A serialização acaba inflingindo alguns modelos de segurança em seu procedimento padrão, e criando então um novo problema. Quando um estado de objeto é persistido no sistema de arquivos ou enviado por rede, o fluxo binário pode ser detectado, analisado e obtidas informações sobre a classe e o estado do objeto. Por exemplo, se utilizado um editor de texto qualquer para abrir o arquivo persistido do estado do objeto da classe “Fruta” dos exemplos anteriores, é comum os atributos e outras informações aparecerem como um simples texto, total ou parcilamente legíveis, assim como demonstrado na Listagem 9.

Listagem 9 - Exemplo de leitura de arquivo binário de um estado de objeto persistido.

|  |  |
| --- | --- |
| 1  2 | ��##sr#  pojo.Fruta#######{###L##cort##Ljava/lang/String;L##nomeq#~##xpt##Vermelhat##Maça |

Com uma ferramenta menos simples seria possível aumentar a quantidade de informações legíveis, coletando uma maior quantia de dados.

No entando, o processo de serialização permite que haja uma certa personalização da serialização e deserialização de um objeto. Isso deve-se a possibilidade da sobrescrição dos métodos “writeObject” e “readObject” dentro da classe a ser serializada, como a classe exemplar “pojo.Fruta”. Esta sobrescrita permite um controle total sobre a serialização, pois as classes “java.io.ObjectOutputStream” e “java.io.ObjectInputStream”, ao serializar o estado de um objeto, primeiramente verificam se esse método foi sobrescrito na classe alvo e assim o utilizam, senão, será utilizado o protocolo padrão de serialização.

Realizando a sobrescrição dos métodos é possível criptografar os dados na serialização e descriptografar os dados na deserialização, por meio de algum algoritmo que o programador venha implementar. Assim os dados podem ser persistidos ou transmitidos por rede, sem que seja possível ler as partes mais importantes do estado de um objeto. Na listagem 10, a classe “pojo.Fruta” foi modificada dando origem a classe “pojo.FrutaSobrescrita” que possui adicionamente a sobrescrição dos métodos “writeObject” e “readObject”. Como este é apenas um exemplo, não foi escrito um verdadeiro algoritmo de criptografia, mas sim apenas uma troca no valor da String do campo “cor”, identificando que houveram alterações tanto no processo de serialização quanto no de deserialização. Na listagem 11 encontra-se a execução da serialização.

Listagem 10 - Sobrescrita dos métodos “writeObject” e “readObject”.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36 | import java.io.Serializable;    public class FrutaSobrescrita implements Serializable{      private String nome;      private String cor;      static final long serialVersionUID = 123L; //indica a versao da classe        public FrutaSobrescrita(String nome, String cor) {          this.nome = nome;          this.cor = cor;      }      public String getCor() { return cor; }      public String getNome() { return nome; }      public void setCor(String cor) { this.cor = cor; }      public void setNome(String nome) { this.nome = nome; }        private void writeObject(java.io.ObjectOutputStream stream) throws java.io.IOException      {          // criptografar os dados antes de liberar o stream          // EXEMPLO: setCor(criptografar(getCor()));          setCor(getCor()+ " > CRIPTOGRAFADO"); // apenas concatenação de string para indicar alteração            // liberar o stream          stream.defaultWriteObject();      }        private void readObject(java.io.ObjectInputStream stream) throws java.io.IOException, ClassNotFoundException      {          // liberar stream          stream.defaultReadObject();            // descriptografar os dados          // EXEMPLO: setCor(descriptografar(getCor()));          setCor(getCor()+ " > DESCRIPTOGRAFADO"); // apenas contatenação de string para indicar alteração      }  } |

Listagem 11 - Execução da serialização da classe “pojo.FrutaSobrescrita”.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | //serializa        Serializador s = new Serializador();        FrutaSobrescrita frutaSobrescrita = new FrutaSobrescrita("Maça", "Vermelha");         try {             s.serializar("/home/gustavo/frutaSobrescrita", frutaSobrescrita);         } catch (Exception ex) {             System.err.println("Falha ao serializar! - " + ex.toString());         }           //deserializa        Deserializador d = new Deserializador();         frutaSobrescrita = null;         try {             frutaSobrescrita = (FrutaSobrescrita) d.deserializar("/home/gustavo/frutaSobrescrita");         } catch (Exception ex) {             System.err.println("Falha ao deserializar! - " + ex.toString());         }         System.out.println("Sobrescrita: " + frutaSobrescrita.getNome() + " - " + frutaSobrescrita.getCor()); |

**5. CONCLUSÃO**

A serialização é um procedimento de simples execução por parte do programador. Com pouco tempo de codificação é possível implementar a persistencia de um estado de objeto para o sistema de arquivos ou de transmissão por rede. É possível reproduzir os exemplos criados neste artigo, em situações até mesmo mais elaboradas. A apresentação de alguns problemas e soluções trazidas por particularidades, abre o entendimento de que a serialização pode deixar de ser simples conforme for aplicada em diferentes contextos, e que ao mesmo tempo é muito flexível ao permitir ao programador implementar diferentes soluções.

[Assista ao vídeo Serialização/Deserialização de objetos com YAML](http://www.devmedia.com.br/serializacao-deserializacao-de-objetos-com-yaml-parte-1/8399)

**6. BIBLIOGRAFIA**

ORACLE - **java.io Interface Serializable** - Disponível em: <http://docs.oracle.com/javase/1.4.2/docs/api/java/io/Serializable.html>Acesso em: 11/03/2012

ORACLE - **Discover de Secrets of Java Serialization API** - Disponível em: <http://java.sun.com/developer/technicalArticles/Programming/serialization/> Acesso em: 11/03/2012

NEWARD, Ted - **5 coisas que você não sabia sobre... Serialização de Objetos Java** - Disponível em: <http://www.ibm.com/developerworks/br/library/j-5things1/index.html> Acesso em: 11/03/2012

DOWNS, Andrew - **Java Serialization** - Disponível em: <http://www.mactech.com/articles/mactech/Vol.14/14.04/JavaSerialization/index.html> Acesso em: 11/03/2012

PREISSLER, Luciano E.; GEYER, Cláudio F. R. - **Serialização em Java** - Disponível em: <http://www.inf.ufrgs.br/gppd/disc/cmp167/trabalhos/sem99-1/T1/luciano/final.htm> Acesso em: 11/03/2012

[[](http://www.linhadecodigo.com.br/colaborador.aspx?id=354)](http://www.linhadecodigo.com.br/colaborador.aspx?id=354)

[Everton Coimbra de Araújo - Desde 1987 atua na área de treinamento e desenvolvimento. Como Mestre em Ciência da Computação, é professor da UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ, Campus Medianeira, onde leciona disciplinas relacionadas ao desenvolvimento de aplicações web, com Java e .NET.   
  
É autor dos livros Desenvolvimento para WEB com Java, Orientação a Objetos com Java - Simples, Fácil e Eficiente, Algoritmo - Fundamento e Prática em sua terceira edição e dos livros Delphi - Implementação e Técnicas para Ambientes Virtuais e C++ Builder - Implementação e Técnicas para Ambientes Virtuais. Todos pela VisualBooks. Pode ser contactado através do e-mail everton@utfpr.edu.br ou evertoncoimbra@gmail.com.](http://www.linhadecodigo.com.br/colaborador.aspx?id=354)

 85 3 0 4

**Leia também**

* [Conhecendo a interface Map do Java](http://www.linhadecodigo.com.br/artigo/3670/conhecendo-a-interface-map-do-java.aspx)*Java*
* [Trabalhando com a Interface Set no Java](http://www.linhadecodigo.com.br/artigo/3669/trabalhando-com-a-interface-set-no-java.aspx)*Java*
* [Classes Wrappers em Java](http://www.linhadecodigo.com.br/artigo/3667/classes-wrappers-em-java.aspx)*Java*
* [ComboBoxModel em Java](http://www.linhadecodigo.com.br/artigo/3664/comboboxmodel-em-java.aspx)*Java*
* [Reduzindo a Mutabilidade em Java](http://www.linhadecodigo.com.br/artigo/3659/reduzindo-a-mutabilidade-em-java.aspx)*Java*

Publicidade

[](http://qconsp.com/?utm_source=devmedia&utm_medium=banner&utm_campaign=devmedia)

Linha de Código[Anuncie](http://www.web-03.net/linhadecodigo.asp)|[Fale Conosco](mailto:colabore@linhadecodigo.com.br)|[Publique](http://www.linhadecodigo.com.br/publique.aspx)

|  |  |
| --- | --- |
| http://www.linhadecodigo.com.br/artigos/img_artigos/EvertonCoimbraAraujo/Serializazao_Objetos/imagem001.jpg | Neste artigo apresento o Gustavo Rafael Valiati, que pode ser contatado pelo e-mail gustavovaliati@gmail.com. O Gustavo foi meu aluno em algumas disciplinas de sua gradução e está concluíndo o último período do curso de Tecnologia em Análise e Desenvolvimento de Sistemas da UTFPR campus Medianeira, neste primeiro semestre de 2012. |

**1. INTRODUÇÃO**

Nem sempre se deseja ter objetos ou estrutura de dados de um aplicativo, com um ciclo de vida que dura no máximo o tempo em que for armazenado em memória RAM. Ou seja, algumas vezes deseja-se guardá-los por um tempo indeterminado, sendo possível recuperá-los posteriormente. E neste caso, o termo “persistência” tem seu conceito definido.

Persistir dados, requer que estes possam ser serializáveis. A serialização possibilitará capturar o estado do objeto ou a estrutura de dados, e transformar em uma cadeia de bytes quando for necessário. Torna-se possível também, recuperar os bytes persistidos para fazer o processo inverso, e ter os dados de volta para a aplicação em execução.

Desta maneira, qualquer forma de dados que esteja sendo trabalhada na aplicação, e esta puder ser serializada, poderá ser passada para o meio físico, e ter o processo revertido.

A serialização não se limita apenas em possibilitar a gravação do dado em disco, mas permite que o mesmo seja transmitido por rede. Isto porquê a serialização cria um stream de bytes, que é requisito para transmissão de dados por rede.

Diversas linguagens possuem suporte à serialização, como: Java, C, C++, Python, PHP, .NET, entre outras. Neste artigo, será tratado apenas da serialização com Java, em sua implemetação básica, alguns problemas e soluções, com códigos para exemplificação.

**2. SERIALIZAÇÃO COM JAVA**

Serializar um objeto, em Java, só é possível caso sua classe esteja marcada como serializável. Essa marcação é feita com a simples implementação da interface “java.io.Serializable”. Essa interface não possui métodos para se implementar e nem atributos. Ainda não existiam as anotações no Java 1.1, versão que trouxe pela primeira vez a serialização ao Java, e desta maneira a implementação da interface era a melhor maneira de marcar uma classe.

A herança entre classes é naturalmente afetada pela serialização. Já que quando uma classe implementa a interface “Serializable”, toda classe que dela estender também estará implicitamente marcada como serializável

.

Para exemplo, será criada uma classe “pojo.Fruta” que contém dois atributos: nome e cor. No código exemplo (Listagem 1), pode-se notar a classe “Fruta” marcada para a serialização:

Listagem 1 - Marcando uma classe como serializável.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | import java.io.Serializable;    public class Fruta implements Serializable{      private String nome;      private String cor;        public Fruta(String nome, String cor) {          this.nome = nome;          this.cor = cor;      }        public String getCor() { return cor; }      public String getNome() { return nome; }      public void setCor(String cor) { this.cor = cor; }      public void setNome(String nome) { this.nome = nome; }  } |

Pode-se dizer que a implementação do processo de serialização e deserialização simples de um objeto, não possui um código difícil de se escrever. Isto porquê duas classes fazem a maior parte do trabalho, de maneira que com poucas linhas de código pode-se transformar um objeto em bytes e vice-versa. Essas duas classes são “java.io.ObjectInputStream” que recebe bytes para criar um objeto e “java.io.ObjectOutputStream” que cria uma cadeia de bytes a partir de um objeto.

**3. SERIALIZANDO E DESERIALIZANDO O ESTADO DE UM OBJETO**

Como exemplo, será persistido um objeto em disco. Para isto, além das classes “java.io.ObjectInputStream” e “java.io.ObjectOutputStream”, serão utilizadas duas outras que trabalham com arquivos, que são “java.io.FileInputStream” e “java.io.FileOutputStream”, que seguem a mesma lógica de InputStream e OutputStream, só que agora relacionado ao sistema de arquivos e não aos objetos.

Para serialização, deve-se criar um novo “java.io.FileOutputStream”, com o caminho (“path”) do arquivo onde se deseja armazenar o estado do objeto. Também deve ser criado um novo “java.io.ObjectOutputStream” que está ligado ao arquivo. E por último deve-se executar a escrita do objeto alvo, para dentro do objeto de stream que por sua vez está ligado ao arquivo destino. Após executados esses passos, é possível encontrar no sistema de arquivos um novo arquivo que contém os bytes correspondentes à descrição do estado do objeto. Na Listagem 2, encontra-se o código exemplo.

Listagem 2 - Serialização de um objeto.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | import java.io.FileOutputStream;  import java.io.ObjectOutputStream;    public class Serializador {        public Serializador() {    }        public void serializar(String path, Object obj) throws Exception {              FileOutputStream outFile = new FileOutputStream(path);      ObjectOutputStream s = new ObjectOutputStream(outFile);      s.writeObject(obj);      s.close();      }  } |

E para a deserialização, o processo é extremamente parecido com a serialização. Cria-se um novo “java.io.FileInputStream” relacionado ao arquivo onde o estado do objeto encontra-se persistido. Deve-se criar também um novo “java.io.ObjectInputStream” que estará vinculado ao arquivo de origem. Por último, executa-se a leitura do arquivo para a restauração do estado. Na Listagem 3 encontra-se o exemplo do código.

Listagem 3 - Deserializaçãode um objeto.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | import java.io.FileInputStream;  import java.io.ObjectInputStream;    public class Deserializador {        public Deserializador() {    }        public Object deserializar(String path) throws Exception {      FileInputStream inFile = new FileInputStream(path);      ObjectInputStream d = new ObjectInputStream(inFile);      Object o = d.readObject();      d.close();      return o;      }  } |

A Listagem 4, contém o exemplo da execução da serialização e em seguida da deserialização.

Listagem 4 - Serializando e Deserializando

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | import operacoes.Deserializador;  import operacoes.Serializador;  import pojo.Fruta;    public class Principal {      public static void main(String args[]){         //serializa         Serializador s = new Serializador();         Fruta fruta = new Fruta("Maça", "Vermelha");          try {              s.serializar("/home/gustavo/fruta", fruta);          } catch (Exception ex) {              System.err.println("Falha ao serializar! - " + ex.toString());          }            //deserializa          Deserializador d = new Deserializador();          fruta = null;          try {              fruta = (Fruta) d.deserializar("/home/gustavo/fruta");          } catch (Exception ex) {              System.err.println("Falha ao deserializar! - " + ex.toString());          }          System.out.println(fruta.getNome() + " - " + fruta.getCor());      }  } |

Na Listagem 5, encontra-se um exemplo de serialização e deserialização de um ArrayList da classe Fruta. É apenas mais um exemplo que demonstra a capacidade de se persistir qualquer objeto que seja serializável. Essa serialização só foi possível porque a classe “ArrayList” implementa a interface “Serializable” por padrão.

Listagem 5 - Serialização de ArrayList.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | //serialização e deserialização com ArrayList          ArrayList<Fruta> frutas = new ArrayList<Fruta>();          Fruta f1 = new Fruta("Laranja", "Amarela");          Fruta f2 = new Fruta("Abacate", "Verde");          Fruta f3 = new Fruta("Morango", "Vermelho");          frutas.add(f1);          frutas.add(f2);          frutas.add(f3);            try {              s.serializar("/home/gustavo/frutas", frutas);              frutas = null;              frutas = (ArrayList<Fruta>) d.deserializar("/home/gustavo/frutas");              for(Fruta f : frutas){                  System.out.println("ArrayList: " + f.getNome() + " - " + f.getCor());              }          } catch (Exception ex) {              System.err.println("Falha ao serializar ou deserializar! - " +                                 ex.toString());          } |

**4. PARTICULARIDADES DA SERIALIZAÇÃO**

Como dito, a serialização de um objeto em Java pode ser algo fácil de se fazer. Mas existem certas informações que são importantes e sabê-las pode evitar vários problemas.

Um dos problemas é que nem tudo pode ser serializado, como objetos das classes “Thread” e “Socket” e atributos estáticos. Os atributos estáticos estão vinculados à classe e não ao objeto. Assim quando se trata do estado do objeto, os campos estáticos não são considerados. Em relação às Threads, suas instâncias só são úteis enquanto estiverem em execução na JVM, e podem ser paradas/destruídas e iniciadas com instruções de sua classe. Não existe algo no estado destas instâncias que seja útil para se persistir, e por isso não há sentido em persistir tais instâncias para restaurar posteriormente.

Mas e se existir alguma classe em que seja necessário persisti-la, e esta possui uma instância da classe “Thread”? Neste caso, utilizando um recurso pode-se persistir a classe inteira, menos a instância thread. Este recurso, refere-se em indicar que tal instância thread é um objeto “Transient”. Na Listagem 6 encontra-se um exemplo da implementação com uso de “Transient”.

Listagem 6 - Uso do Transient

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | import java.io.Serializable;    public class ImplementacaoTransient implements Serializable, Runnable{        transient private Thread thread;      //marcado com "transient" para que sua instância não seja serializada        //Demais valores de estado serão serializados      private int valorParaPersistir;        public ImplementacaoTransient(int valorParaPersistir){          this.valorParaPersistir = valorParaPersistir;          thread = new Thread(this);          thread.start();      }        public void run()      {          while(true){              //código a ser executado pela Thread          }      }  } |

Vale ressaltar que, o que é serializado é apenas o estado do objeto, e não os métodos e a classe. No exemplo acima, apenas a instância do objeto “thread” e do objeto “valorParaPersistir” compunham o estado do objeto. Assim, evita-se que o sistema de serialização tente serializar um objeto não-serializável e gere erros. Após a deserialização de um objeto da classe “ImplementacaoTransient” este estará totalmente funcional.

Um outro problema que pode vir a ocorrer, se refere ao versionamento das classes. Toda classe possui um atributo oculto chamado “serialVersionUID” com um valor gerado automaticamente. Objetos criados levam consigo o “serialVersionUID” de sua classe, que indica sua versão. Toda vez que uma classe sofre alterações em seu código, um novo “serialVersionUID” é gerado. O problema com a serialização surge quando serializa-se o estado de um objeto e posteriormente tenta-se restaurar o estado para a aplicação, mas houveram mudanças no código da classe. Neste caso, uma exceção “java.io.InvalidClassException” é disparada, pois de fato, o estado do objeto corresponde a uma versão de classe do qual foi criado que já não existe mais. Portanto, isso só ocorre para evitar erros de programação, em que tenta-se restaurar o estado de um objeto para uma classe que possui versão diferente da qual o objeto foi serializado inicialmente. E isso pode significar diferentes versões de uma mesma classe “Fruta” por exemplo, ou diferentes versões por classes serem totalmente diferentes como: o estado de um objeto da classe “Fruta” ser restaurado para classe “Carro”.

Mas quando é necessário, conscientemente, recuperar o estado de um objeto, mesmo que houveram alterações na classe do qual foi gerado, é possível controlar manualmente o “serialVersionUID”. Para isto, basta declarar explicitamente na forma de atributo estático e final, um valor Long qualquer para corresponder à versão da classe. E assim, mesmo que houverem modificações na classe, a versão permenecerá a mesma, evitando as exceções da incompatiblidade de classes. No entanto, vale ressaltar que o versionamento não é um impecílio para o programador, mas sim muito importante em casos que deseja-se manter um controle sobre a compatibilidade de código na aplicação, definindo versões para classes No código da Listagem 7 encontra-se o exemplo do controle manual do “serialVersionUID”.

Listagem 7 - Uso manual do serialVersionUID

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | import java.io.Serializable;    public class Fruta implements Serializable{      private String nome;      private String cor;      static final long serialVersionUID = 123L; //indica a versao da classe        public Fruta(String nome, String cor) {          this.nome = nome;          this.cor = cor;      }      public String getCor() { return cor; }      public String getNome() { return nome; }      public void setCor(String cor) { this.cor = cor; }      public void setNome(String nome) { this.nome = nome; }  } |

O JDK fornece uma ferramenta que permite visualizar a versão de qualquer classe Java. Basta no terminal do sistema operacional utilizar o comando “serialver” seguido da classe qualificada, conforme exemplo da Listagem 8.

Listagem 8 - Verificação da versão de classe

|  |  |
| --- | --- |
| 1 | serialver pojo.Fruta |

A serialização acaba inflingindo alguns modelos de segurança em seu procedimento padrão, e criando então um novo problema. Quando um estado de objeto é persistido no sistema de arquivos ou enviado por rede, o fluxo binário pode ser detectado, analisado e obtidas informações sobre a classe e o estado do objeto. Por exemplo, se utilizado um editor de texto qualquer para abrir o arquivo persistido do estado do objeto da classe “Fruta” dos exemplos anteriores, é comum os atributos e outras informações aparecerem como um simples texto, total ou parcilamente legíveis, assim como demonstrado na Listagem 9.

Listagem 9 - Exemplo de leitura de arquivo binário de um estado de objeto persistido.

|  |  |
| --- | --- |
| 1  2 | ��##sr#  pojo.Fruta#######{###L##cort##Ljava/lang/String;L##nomeq#~##xpt##Vermelhat##Maça |

Com uma ferramenta menos simples seria possível aumentar a quantidade de informações legíveis, coletando uma maior quantia de dados.

No entando, o processo de serialização permite que haja uma certa personalização da serialização e deserialização de um objeto. Isso deve-se a possibilidade da sobrescrição dos métodos “writeObject” e “readObject” dentro da classe a ser serializada, como a classe exemplar “pojo.Fruta”. Esta sobrescrita permite um controle total sobre a serialização, pois as classes “java.io.ObjectOutputStream” e “java.io.ObjectInputStream”, ao serializar o estado de um objeto, primeiramente verificam se esse método foi sobrescrito na classe alvo e assim o utilizam, senão, será utilizado o protocolo padrão de serialização.

Realizando a sobrescrição dos métodos é possível criptografar os dados na serialização e descriptografar os dados na deserialização, por meio de algum algoritmo que o programador venha implementar. Assim os dados podem ser persistidos ou transmitidos por rede, sem que seja possível ler as partes mais importantes do estado de um objeto. Na listagem 10, a classe “pojo.Fruta” foi modificada dando origem a classe “pojo.FrutaSobrescrita” que possui adicionamente a sobrescrição dos métodos “writeObject” e “readObject”. Como este é apenas um exemplo, não foi escrito um verdadeiro algoritmo de criptografia, mas sim apenas uma troca no valor da String do campo “cor”, identificando que houveram alterações tanto no processo de serialização quanto no de deserialização. Na listagem 11 encontra-se a execução da serialização.

Listagem 10 - Sobrescrita dos métodos “writeObject” e “readObject”.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36 | import java.io.Serializable;    public class FrutaSobrescrita implements Serializable{      private String nome;      private String cor;      static final long serialVersionUID = 123L; //indica a versao da classe        public FrutaSobrescrita(String nome, String cor) {          this.nome = nome;          this.cor = cor;      }      public String getCor() { return cor; }      public String getNome() { return nome; }      public void setCor(String cor) { this.cor = cor; }      public void setNome(String nome) { this.nome = nome; }        private void writeObject(java.io.ObjectOutputStream stream) throws java.io.IOException      {          // criptografar os dados antes de liberar o stream          // EXEMPLO: setCor(criptografar(getCor()));          setCor(getCor()+ " > CRIPTOGRAFADO"); // apenas concatenação de string para indicar alteração            // liberar o stream          stream.defaultWriteObject();      }        private void readObject(java.io.ObjectInputStream stream) throws java.io.IOException, ClassNotFoundException      {          // liberar stream          stream.defaultReadObject();            // descriptografar os dados          // EXEMPLO: setCor(descriptografar(getCor()));          setCor(getCor()+ " > DESCRIPTOGRAFADO"); // apenas contatenação de string para indicar alteração      }  } |

Listagem 11 - Execução da serialização da classe “pojo.FrutaSobrescrita”.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | //serializa        Serializador s = new Serializador();        FrutaSobrescrita frutaSobrescrita = new FrutaSobrescrita("Maça", "Vermelha");         try {             s.serializar("/home/gustavo/frutaSobrescrita", frutaSobrescrita);         } catch (Exception ex) {             System.err.println("Falha ao serializar! - " + ex.toString());         }           //deserializa        Deserializador d = new Deserializador();         frutaSobrescrita = null;         try {             frutaSobrescrita = (FrutaSobrescrita) d.deserializar("/home/gustavo/frutaSobrescrita");         } catch (Exception ex) {             System.err.println("Falha ao deserializar! - " + ex.toString());         }         System.out.println("Sobrescrita: " + frutaSobrescrita.getNome() + " - " + frutaSobrescrita.getCor()); |

**5. CONCLUSÃO**

A serialização é um procedimento de simples execução por parte do programador. Com pouco tempo de codificação é possível implementar a persistencia de um estado de objeto para o sistema de arquivos ou de transmissão por rede. É possível reproduzir os exemplos criados neste artigo, em situações até mesmo mais elaboradas. A apresentação de alguns problemas e soluções trazidas por particularidades, abre o entendimento de que a serialização pode deixar de ser simples conforme for aplicada em diferentes contextos, e que ao mesmo tempo é muito flexível ao permitir ao programador implementar diferentes soluções.

[Assista ao vídeo Serialização/Deserialização de objetos com YAML](http://www.devmedia.com.br/serializacao-deserializacao-de-objetos-com-yaml-parte-1/8399)

**6. BIBLIOGRAFIA**

ORACLE - **java.io Interface Serializable** - Disponível em: <http://docs.oracle.com/javase/1.4.2/docs/api/java/io/Serializable.html>Acesso em: 11/03/2012

ORACLE - **Discover de Secrets of Java Serialization API** - Disponível em: <http://java.sun.com/developer/technicalArticles/Programming/serialization/> Acesso em: 11/03/2012

NEWARD, Ted - **5 coisas que você não sabia sobre... Serialização de Objetos Java** - Disponível em: <http://www.ibm.com/developerworks/br/library/j-5things1/index.html> Acesso em: 11/03/2012

DOWNS, Andrew - **Java Serialization** - Disponível em: <http://www.mactech.com/articles/mactech/Vol.14/14.04/JavaSerialization/index.html> Acesso em: 11/03/2012

PREISSLER, Luciano E.; GEYER, Cláudio F. R. - **Serialização em Java** - Disponível em: <http://www.inf.ufrgs.br/gppd/disc/cmp167/trabalhos/sem99-1/T1/luciano/final.htm> Acesso em: 11/03/2012

[[](http://www.linhadecodigo.com.br/colaborador.aspx?id=354)](http://www.linhadecodigo.com.br/colaborador.aspx?id=354)

[Everton Coimbra de Araújo - Desde 1987 atua na área de treinamento e desenvolvimento. Como Mestre em Ciência da Computação, é professor da UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ, Campus Medianeira, onde leciona disciplinas relacionadas ao desenvolvimento de aplicações web, com Java e .NET.   
  
É autor dos livros Desenvolvimento para WEB com Java, Orientação a Objetos com Java - Simples, Fácil e Eficiente, Algoritmo - Fundamento e Prática em sua terceira edição e dos livros Delphi - Implementação e Técnicas para Ambientes Virtuais e C++ Builder - Implementação e Técnicas para Ambientes Virtuais. Todos pela VisualBooks. Pode ser contactado através do e-mail everton@utfpr.edu.br ou evertoncoimbra@gmail.com.](http://www.linhadecodigo.com.br/colaborador.aspx?id=354)

 85 3 0 4

**Leia também**

* [Conhecendo a interface Map do Java](http://www.linhadecodigo.com.br/artigo/3670/conhecendo-a-interface-map-do-java.aspx)*Java*
* [Trabalhando com a Interface Set no Java](http://www.linhadecodigo.com.br/artigo/3669/trabalhando-com-a-interface-set-no-java.aspx)*Java*
* [Classes Wrappers em Java](http://www.linhadecodigo.com.br/artigo/3667/classes-wrappers-em-java.aspx)*Java*
* [ComboBoxModel em Java](http://www.linhadecodigo.com.br/artigo/3664/comboboxmodel-em-java.aspx)*Java*
* [Reduzindo a Mutabilidade em Java](http://www.linhadecodigo.com.br/artigo/3659/reduzindo-a-mutabilidade-em-java.aspx)*Java*

Publicidade

[](http://qconsp.com/?utm_source=devmedia&utm_medium=banner&utm_campaign=devmedia)

Linha de Código[Anuncie](http://www.web-03.net/linhadecodigo.asp)|[Fale Conosco](mailto:colabore@linhadecodigo.com.br)|[Publique](http://www.linhadecodigo.com.br/publique.aspx)

|  |  |
| --- | --- |
| http://www.linhadecodigo.com.br/artigos/img_artigos/EvertonCoimbraAraujo/Serializazao_Objetos/imagem001.jpg | Neste artigo apresento o Gustavo Rafael Valiati, que pode ser contatado pelo e-mail gustavovaliati@gmail.com. O Gustavo foi meu aluno em algumas disciplinas de sua gradução e está concluíndo o último período do curso de Tecnologia em Análise e Desenvolvimento de Sistemas da UTFPR campus Medianeira, neste primeiro semestre de 2012. |

**1. INTRODUÇÃO**

Nem sempre se deseja ter objetos ou estrutura de dados de um aplicativo, com um ciclo de vida que dura no máximo o tempo em que for armazenado em memória RAM. Ou seja, algumas vezes deseja-se guardá-los por um tempo indeterminado, sendo possível recuperá-los posteriormente. E neste caso, o termo “persistência” tem seu conceito definido.

Persistir dados, requer que estes possam ser serializáveis. A serialização possibilitará capturar o estado do objeto ou a estrutura de dados, e transformar em uma cadeia de bytes quando for necessário. Torna-se possível também, recuperar os bytes persistidos para fazer o processo inverso, e ter os dados de volta para a aplicação em execução.

Desta maneira, qualquer forma de dados que esteja sendo trabalhada na aplicação, e esta puder ser serializada, poderá ser passada para o meio físico, e ter o processo revertido.

A serialização não se limita apenas em possibilitar a gravação do dado em disco, mas permite que o mesmo seja transmitido por rede. Isto porquê a serialização cria um stream de bytes, que é requisito para transmissão de dados por rede.

Diversas linguagens possuem suporte à serialização, como: Java, C, C++, Python, PHP, .NET, entre outras. Neste artigo, será tratado apenas da serialização com Java, em sua implemetação básica, alguns problemas e soluções, com códigos para exemplificação.

**2. SERIALIZAÇÃO COM JAVA**

Serializar um objeto, em Java, só é possível caso sua classe esteja marcada como serializável. Essa marcação é feita com a simples implementação da interface “java.io.Serializable”. Essa interface não possui métodos para se implementar e nem atributos. Ainda não existiam as anotações no Java 1.1, versão que trouxe pela primeira vez a serialização ao Java, e desta maneira a implementação da interface era a melhor maneira de marcar uma classe.

A herança entre classes é naturalmente afetada pela serialização. Já que quando uma classe implementa a interface “Serializable”, toda classe que dela estender também estará implicitamente marcada como serializável

.

Para exemplo, será criada uma classe “pojo.Fruta” que contém dois atributos: nome e cor. No código exemplo (Listagem 1), pode-se notar a classe “Fruta” marcada para a serialização:

Listagem 1 - Marcando uma classe como serializável.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | import java.io.Serializable;    public class Fruta implements Serializable{      private String nome;      private String cor;        public Fruta(String nome, String cor) {          this.nome = nome;          this.cor = cor;      }        public String getCor() { return cor; }      public String getNome() { return nome; }      public void setCor(String cor) { this.cor = cor; }      public void setNome(String nome) { this.nome = nome; }  } |

Pode-se dizer que a implementação do processo de serialização e deserialização simples de um objeto, não possui um código difícil de se escrever. Isto porquê duas classes fazem a maior parte do trabalho, de maneira que com poucas linhas de código pode-se transformar um objeto em bytes e vice-versa. Essas duas classes são “java.io.ObjectInputStream” que recebe bytes para criar um objeto e “java.io.ObjectOutputStream” que cria uma cadeia de bytes a partir de um objeto.

**3. SERIALIZANDO E DESERIALIZANDO O ESTADO DE UM OBJETO**

Como exemplo, será persistido um objeto em disco. Para isto, além das classes “java.io.ObjectInputStream” e “java.io.ObjectOutputStream”, serão utilizadas duas outras que trabalham com arquivos, que são “java.io.FileInputStream” e “java.io.FileOutputStream”, que seguem a mesma lógica de InputStream e OutputStream, só que agora relacionado ao sistema de arquivos e não aos objetos.

Para serialização, deve-se criar um novo “java.io.FileOutputStream”, com o caminho (“path”) do arquivo onde se deseja armazenar o estado do objeto. Também deve ser criado um novo “java.io.ObjectOutputStream” que está ligado ao arquivo. E por último deve-se executar a escrita do objeto alvo, para dentro do objeto de stream que por sua vez está ligado ao arquivo destino. Após executados esses passos, é possível encontrar no sistema de arquivos um novo arquivo que contém os bytes correspondentes à descrição do estado do objeto. Na Listagem 2, encontra-se o código exemplo.

Listagem 2 - Serialização de um objeto.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | import java.io.FileOutputStream;  import java.io.ObjectOutputStream;    public class Serializador {        public Serializador() {    }        public void serializar(String path, Object obj) throws Exception {              FileOutputStream outFile = new FileOutputStream(path);      ObjectOutputStream s = new ObjectOutputStream(outFile);      s.writeObject(obj);      s.close();      }  } |

E para a deserialização, o processo é extremamente parecido com a serialização. Cria-se um novo “java.io.FileInputStream” relacionado ao arquivo onde o estado do objeto encontra-se persistido. Deve-se criar também um novo “java.io.ObjectInputStream” que estará vinculado ao arquivo de origem. Por último, executa-se a leitura do arquivo para a restauração do estado. Na Listagem 3 encontra-se o exemplo do código.

Listagem 3 - Deserializaçãode um objeto.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | import java.io.FileInputStream;  import java.io.ObjectInputStream;    public class Deserializador {        public Deserializador() {    }        public Object deserializar(String path) throws Exception {      FileInputStream inFile = new FileInputStream(path);      ObjectInputStream d = new ObjectInputStream(inFile);      Object o = d.readObject();      d.close();      return o;      }  } |

A Listagem 4, contém o exemplo da execução da serialização e em seguida da deserialização.

Listagem 4 - Serializando e Deserializando

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | import operacoes.Deserializador;  import operacoes.Serializador;  import pojo.Fruta;    public class Principal {      public static void main(String args[]){         //serializa         Serializador s = new Serializador();         Fruta fruta = new Fruta("Maça", "Vermelha");          try {              s.serializar("/home/gustavo/fruta", fruta);          } catch (Exception ex) {              System.err.println("Falha ao serializar! - " + ex.toString());          }            //deserializa          Deserializador d = new Deserializador();          fruta = null;          try {              fruta = (Fruta) d.deserializar("/home/gustavo/fruta");          } catch (Exception ex) {              System.err.println("Falha ao deserializar! - " + ex.toString());          }          System.out.println(fruta.getNome() + " - " + fruta.getCor());      }  } |

Na Listagem 5, encontra-se um exemplo de serialização e deserialização de um ArrayList da classe Fruta. É apenas mais um exemplo que demonstra a capacidade de se persistir qualquer objeto que seja serializável. Essa serialização só foi possível porque a classe “ArrayList” implementa a interface “Serializable” por padrão.

Listagem 5 - Serialização de ArrayList.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | //serialização e deserialização com ArrayList          ArrayList<Fruta> frutas = new ArrayList<Fruta>();          Fruta f1 = new Fruta("Laranja", "Amarela");          Fruta f2 = new Fruta("Abacate", "Verde");          Fruta f3 = new Fruta("Morango", "Vermelho");          frutas.add(f1);          frutas.add(f2);          frutas.add(f3);            try {              s.serializar("/home/gustavo/frutas", frutas);              frutas = null;              frutas = (ArrayList<Fruta>) d.deserializar("/home/gustavo/frutas");              for(Fruta f : frutas){                  System.out.println("ArrayList: " + f.getNome() + " - " + f.getCor());              }          } catch (Exception ex) {              System.err.println("Falha ao serializar ou deserializar! - " +                                 ex.toString());          } |

**4. PARTICULARIDADES DA SERIALIZAÇÃO**

Como dito, a serialização de um objeto em Java pode ser algo fácil de se fazer. Mas existem certas informações que são importantes e sabê-las pode evitar vários problemas.

Um dos problemas é que nem tudo pode ser serializado, como objetos das classes “Thread” e “Socket” e atributos estáticos. Os atributos estáticos estão vinculados à classe e não ao objeto. Assim quando se trata do estado do objeto, os campos estáticos não são considerados. Em relação às Threads, suas instâncias só são úteis enquanto estiverem em execução na JVM, e podem ser paradas/destruídas e iniciadas com instruções de sua classe. Não existe algo no estado destas instâncias que seja útil para se persistir, e por isso não há sentido em persistir tais instâncias para restaurar posteriormente.

Mas e se existir alguma classe em que seja necessário persisti-la, e esta possui uma instância da classe “Thread”? Neste caso, utilizando um recurso pode-se persistir a classe inteira, menos a instância thread. Este recurso, refere-se em indicar que tal instância thread é um objeto “Transient”. Na Listagem 6 encontra-se um exemplo da implementação com uso de “Transient”.

Listagem 6 - Uso do Transient

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | import java.io.Serializable;    public class ImplementacaoTransient implements Serializable, Runnable{        transient private Thread thread;      //marcado com "transient" para que sua instância não seja serializada        //Demais valores de estado serão serializados      private int valorParaPersistir;        public ImplementacaoTransient(int valorParaPersistir){          this.valorParaPersistir = valorParaPersistir;          thread = new Thread(this);          thread.start();      }        public void run()      {          while(true){              //código a ser executado pela Thread          }      }  } |

Vale ressaltar que, o que é serializado é apenas o estado do objeto, e não os métodos e a classe. No exemplo acima, apenas a instância do objeto “thread” e do objeto “valorParaPersistir” compunham o estado do objeto. Assim, evita-se que o sistema de serialização tente serializar um objeto não-serializável e gere erros. Após a deserialização de um objeto da classe “ImplementacaoTransient” este estará totalmente funcional.

**2. SERIALIZAÇÃO COM JAVA**

Serializar um objeto, em Java, só é possível caso sua classe esteja marcada como serializável. Essa marcação é feita com a simples implementação da interface “java.io.Serializable”. Essa interface não possui métodos para se implementar e nem atributos. Ainda não existiam as anotações no Java 1.1, versão que trouxe pela primeira vez a serialização ao Java, e desta maneira a implementação da interface era a melhor maneira de marcar uma classe.

A herança entre classes é naturalmente afetada pela serialização. Já que quando uma classe implementa a interface “Serializable”, toda classe que dela estender também estará implicitamente marcada como serializável

.

Para exemplo, será criada uma classe “pojo.Fruta” que contém dois atributos: nome e cor. No código exemplo (Listagem 1), pode-se notar a classe “Fruta” marcada para a serialização:

Listagem 1 - Marcando uma classe como serializável.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | import java.io.Serializable;    public class Fruta implements Serializable{      private String nome;      private String cor;        public Fruta(String nome, String cor) {          this.nome = nome;          this.cor = cor;      }        public String getCor() { return cor; }      public String getNome() { return nome; }      public void setCor(String cor) { this.cor = cor; }      public void setNome(String nome) { this.nome = nome; }  } |

Pode-se dizer que a implementação do processo de serialização e deserialização simples de um objeto, não possui um código difícil de se escrever. Isto porquê duas classes fazem a maior parte do trabalho, de maneira que com poucas linhas de código pode-se transformar um objeto em bytes e vice-versa. Essas duas classes são “java.io.ObjectInputStream” que recebe bytes para criar um objeto e “java.io.ObjectOutputStream” que cria uma cadeia de bytes a partir de um objeto.

**3. SERIALIZANDO E DESERIALIZANDO O ESTADO DE UM OBJETO**

Como exemplo, será persistido um objeto em disco. Para isto, além das classes “java.io.ObjectInputStream” e “java.io.ObjectOutputStream”, serão utilizadas duas outras que trabalham com arquivos, que são “java.io.FileInputStream” e “java.io.FileOutputStream”, que seguem a mesma lógica de InputStream e OutputStream, só que agora relacionado ao sistema de arquivos e não aos objetos.

Para serialização, deve-se criar um novo “java.io.FileOutputStream”, com o caminho (“path”) do arquivo onde se deseja armazenar o estado do objeto. Também deve ser criado um novo “java.io.ObjectOutputStream” que está ligado ao arquivo. E por último deve-se executar a escrita do objeto alvo, para dentro do objeto de stream que por sua vez está ligado ao arquivo destino. Após executados esses passos, é possível encontrar no sistema de arquivos um novo arquivo que contém os bytes correspondentes à descrição do estado do objeto. Na Listagem 2, encontra-se o código exemplo.

Listagem 2 - Serialização de um objeto.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | import java.io.FileOutputStream;  import java.io.ObjectOutputStream;    public class Serializador {        public Serializador() {    }        public void serializar(String path, Object obj) throws Exception {              FileOutputStream outFile = new FileOutputStream(path);      ObjectOutputStream s = new ObjectOutputStream(outFile);      s.writeObject(obj);      s.close();      }  } |

E para a deserialização, o processo é extremamente parecido com a serialização. Cria-se um novo “java.io.FileInputStream” relacionado ao arquivo onde o estado do objeto encontra-se persistido. Deve-se criar também um novo “java.io.ObjectInputStream” que estará vinculado ao arquivo de origem. Por último, executa-se a leitura do arquivo para a restauração do estado. Na Listagem 3 encontra-se o exemplo do código.

Listagem 3 - Deserializaçãode um objeto.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | import java.io.FileInputStream;  import java.io.ObjectInputStream;    public class Deserializador {        public Deserializador() {    }        public Object deserializar(String path) throws Exception {      FileInputStream inFile = new FileInputStream(path);      ObjectInputStream d = new ObjectInputStream(inFile);      Object o = d.readObject();      d.close();      return o;      }  } |

A Listagem 4, contém o exemplo da execução da serialização e em seguida da deserialização.

Listagem 4 - Serializando e Deserializando

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | import operacoes.Deserializador;  import operacoes.Serializador;  import pojo.Fruta;    public class Principal {      public static void main(String args[]){         //serializa         Serializador s = new Serializador();         Fruta fruta = new Fruta("Maça", "Vermelha");          try {              s.serializar("/home/gustavo/fruta", fruta);          } catch (Exception ex) {              System.err.println("Falha ao serializar! - " + ex.toString());          }            //deserializa          Deserializador d = new Deserializador();          fruta = null;          try {              fruta = (Fruta) d.deserializar("/home/gustavo/fruta");          } catch (Exception ex) {              System.err.println("Falha ao deserializar! - " + ex.toString());          }          System.out.println(fruta.getNome() + " - " + fruta.getCor());      }  } |

Na Listagem 5, encontra-se um exemplo de serialização e deserialização de um ArrayList da classe Fruta. É apenas mais um exemplo que demonstra a capacidade de se persistir qualquer objeto que seja serializável. Essa serialização só foi possível porque a classe “ArrayList” implementa a interface “Serializable” por padrão.

Listagem 5 - Serialização de ArrayList.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | //serialização e deserialização com ArrayList          ArrayList<Fruta> frutas = new ArrayList<Fruta>();          Fruta f1 = new Fruta("Laranja", "Amarela");          Fruta f2 = new Fruta("Abacate", "Verde");          Fruta f3 = new Fruta("Morango", "Vermelho");          frutas.add(f1);          frutas.add(f2);          frutas.add(f3);            try {              s.serializar("/home/gustavo/frutas", frutas);              frutas = null;              frutas = (ArrayList<Fruta>) d.deserializar("/home/gustavo/frutas");              for(Fruta f : frutas){                  System.out.println("ArrayList: " + f.getNome() + " - " + f.getCor());              }          } catch (Exception ex) {              System.err.println("Falha ao serializar ou deserializar! - " +                                 ex.toString());          } |

**4. PARTICULARIDADES DA SERIALIZAÇÃO**

Como dito, a serialização de um objeto em Java pode ser algo fácil de se fazer. Mas existem certas informações que são importantes e sabê-las pode evitar vários problemas.

Um dos problemas é que nem tudo pode ser serializado, como objetos das classes “Thread” e “Socket” e atributos estáticos. Os atributos estáticos estão vinculados à classe e não ao objeto. Assim quando se trata do estado do objeto, os campos estáticos não são considerados. Em relação às Threads, suas instâncias só são úteis enquanto estiverem em execução na JVM, e podem ser paradas/destruídas e iniciadas com instruções de sua classe. Não existe algo no estado destas instâncias que seja útil para se persistir, e por isso não há sentido em persistir tais instâncias para restaurar posteriormente.

Mas e se existir alguma classe em que seja necessário persisti-la, e esta possui uma instância da classe “Thread”? Neste caso, utilizando um recurso pode-se persistir a classe inteira, menos a instância thread. Este recurso, refere-se em indicar que tal instância thread é um objeto “Transient”. Na Listagem 6 encontra-se um exemplo da implementação com uso de “Transient”.

Listagem 6 - Uso do Transient

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | import java.io.Serializable;    public class ImplementacaoTransient implements Serializable, Runnable{        transient private Thread thread;      //marcado com "transient" para que sua instância não seja serializada        //Demais valores de estado serão serializados      private int valorParaPersistir;        public ImplementacaoTransient(int valorParaPersistir){          this.valorParaPersistir = valorParaPersistir;          thread = new Thread(this);          thread.start();      }        public void run()      {          while(true){              //código a ser executado pela Thread          }      }  } |

Vale ressaltar que, o que é serializado é apenas o estado do objeto, e não os métodos e a classe. No exemplo acima, apenas a instância do objeto “thread” e do objeto “valorParaPersistir” compunham o estado do objeto. Assim, evita-se que o sistema de serialização tente serializar um objeto não-serializável e gere erros. Após a deserialização de um objeto da classe “ImplementacaoTransient” este estará totalmente funcional.

Um outro problema que pode vir a ocorrer, se refere ao versionamento das classes. Toda classe possui um atributo oculto chamado “serialVersionUID” com um valor gerado automaticamente. Objetos criados levam consigo o “serialVersionUID” de sua classe, que indica sua versão. Toda vez que uma classe sofre alterações em seu código, um novo “serialVersionUID” é gerado. O problema com a serialização surge quando serializa-se o estado de um objeto e posteriormente tenta-se restaurar o estado para a aplicação, mas houveram mudanças no código da classe. Neste caso, uma exceção “java.io.InvalidClassException” é disparada, pois de fato, o estado do objeto corresponde a uma versão de classe do qual foi criado que já não existe mais. Portanto, isso só ocorre para evitar erros de programação, em que tenta-se restaurar o estado de um objeto para uma classe que possui versão diferente da qual o objeto foi serializado inicialmente. E isso pode significar diferentes versões de uma mesma classe “Fruta” por exemplo, ou diferentes versões por classes serem totalmente diferentes como: o estado de um objeto da classe “Fruta” ser restaurado para classe “Carro”.

Mas quando é necessário, conscientemente, recuperar o estado de um objeto, mesmo que houveram alterações na classe do qual foi gerado, é possível controlar manualmente o “serialVersionUID”. Para isto, basta declarar explicitamente na forma de atributo estático e final, um valor Long qualquer para corresponder à versão da classe. E assim, mesmo que houverem modificações na classe, a versão permenecerá a mesma, evitando as exceções da incompatiblidade de classes. No entanto, vale ressaltar que o versionamento não é um impecílio para o programador, mas sim muito importante em casos que deseja-se manter um controle sobre a compatibilidade de código na aplicação, definindo versões para classes No código da Listagem 7 encontra-se o exemplo do controle manual do “serialVersionUID”.

Listagem 7 - Uso manual do serialVersionUID

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | import java.io.Serializable;    public class Fruta implements Serializable{      private String nome;      private String cor;      static final long serialVersionUID = 123L; //indica a versao da classe        public Fruta(String nome, String cor) {          this.nome = nome;          this.cor = cor;      }      public String getCor() { return cor; }      public String getNome() { return nome; }      public void setCor(String cor) { this.cor = cor; }      public void setNome(String nome) { this.nome = nome; }  } |

O JDK fornece uma ferramenta que permite visualizar a versão de qualquer classe Java. Basta no terminal do sistema operacional utilizar o comando “serialver” seguido da classe qualificada, conforme exemplo da Listagem 8.

Listagem 8 - Verificação da versão de classe

|  |  |
| --- | --- |
| http://www.linhadecodigo.com.br/artigos/img_artigos/EvertonCoimbraAraujo/Serializazao_Objetos/imagem001.jpg | Neste artigo apresento o Gustavo Rafael Valiati, que pode ser contatado pelo e-mail gustavovaliati@gmail.com. O Gustavo foi meu aluno em algumas disciplinas de sua gradução e está concluíndo o último período do curso de Tecnologia em Análise e Desenvolvimento de Sistemas da UTFPR campus Medianeira, neste primeiro semestre de 2012. |

**1. INTRODUÇÃO**

Nem sempre se deseja ter objetos ou estrutura de dados de um aplicativo, com um ciclo de vida que dura no máximo o tempo em que for armazenado em memória RAM. Ou seja, algumas vezes deseja-se guardá-los por um tempo indeterminado, sendo possível recuperá-los posteriormente. E neste caso, o termo “persistência” tem seu conceito definido.

Persistir dados, requer que estes possam ser serializáveis. A serialização possibilitará capturar o estado do objeto ou a estrutura de dados, e transformar em uma cadeia de bytes quando for necessário. Torna-se possível também, recuperar os bytes persistidos para fazer o processo inverso, e ter os dados de volta para a aplicação em execução.

Desta maneira, qualquer forma de dados que esteja sendo trabalhada na aplicação, e esta puder ser serializada, poderá ser passada para o meio físico, e ter o processo revertido.

A serialização não se limita apenas em possibilitar a gravação do dado em disco, mas permite que o mesmo seja transmitido por rede. Isto porquê a serialização cria um stream de bytes, que é requisito para transmissão de dados por rede.

Diversas linguagens possuem suporte à serialização, como: Java, C, C++, Python, PHP, .NET, entre outras. Neste artigo, será tratado apenas da serialização com Java, em sua implemetação básica, alguns problemas e soluções, com códigos para exemplificação.

**2. SERIALIZAÇÃO COM JAVA**

Serializar um objeto, em Java, só é possível caso sua classe esteja marcada como serializável. Essa marcação é feita com a simples implementação da interface “java.io.Serializable”. Essa interface não possui métodos para se implementar e nem atributos. Ainda não existiam as anotações no Java 1.1, versão que trouxe pela primeira vez a serialização ao Java, e desta maneira a implementação da interface era a melhor maneira de marcar uma classe.

A herança entre classes é naturalmente afetada pela serialização. Já que quando uma classe implementa a interface “Serializable”, toda classe que dela estender também estará implicitamente marcada como serializável

.

Para exemplo, será criada uma classe “pojo.Fruta” que contém dois atributos: nome e cor. No código exemplo (Listagem 1), pode-se notar a classe “Fruta” marcada para a serialização:

Listagem 1 - Marcando uma classe como serializável.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | import java.io.Serializable;    public class Fruta implements Serializable{      private String nome;      private String cor;        public Fruta(String nome, String cor) {          this.nome = nome;          this.cor = cor;      }        public String getCor() { return cor; }      public String getNome() { return nome; }      public void setCor(String cor) { this.cor = cor; }      public void setNome(String nome) { this.nome = nome; }  } |

Pode-se dizer que a implementação do processo de serialização e deserialização simples de um objeto, não possui um código difícil de se escrever. Isto porquê duas classes fazem a maior parte do trabalho, de maneira que com poucas linhas de código pode-se transformar um objeto em bytes e vice-versa. Essas duas classes são “java.io.ObjectInputStream” que recebe bytes para criar um objeto e “java.io.ObjectOutputStream” que cria uma cadeia de bytes a partir de um objeto.

**3. SERIALIZANDO E DESERIALIZANDO O ESTADO DE UM OBJETO**

Como exemplo, será persistido um objeto em disco. Para isto, além das classes “java.io.ObjectInputStream” e “java.io.ObjectOutputStream”, serão utilizadas duas outras que trabalham com arquivos, que são “java.io.FileInputStream” e “java.io.FileOutputStream”, que seguem a mesma lógica de InputStream e OutputStream, só que agora relacionado ao sistema de arquivos e não aos objetos.

Para serialização, deve-se criar um novo “java.io.FileOutputStream”, com o caminho (“path”) do arquivo onde se deseja armazenar o estado do objeto. Também deve ser criado um novo “java.io.ObjectOutputStream” que está ligado ao arquivo. E por último deve-se executar a escrita do objeto alvo, para dentro do objeto de stream que por sua vez está ligado ao arquivo destino. Após executados esses passos, é possível encontrar no sistema de arquivos um novo arquivo que contém os bytes correspondentes à descrição do estado do objeto. Na Listagem 2, encontra-se o código exemplo.

Listagem 2 - Serialização de um objeto.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | import java.io.FileOutputStream;  import java.io.ObjectOutputStream;    public class Serializador {        public Serializador() {    }        public void serializar(String path, Object obj) throws Exception {              FileOutputStream outFile = new FileOutputStream(path);      ObjectOutputStream s = new ObjectOutputStream(outFile);      s.writeObject(obj);      s.close();      }  } |

E para a deserialização, o processo é extremamente parecido com a serialização. Cria-se um novo “java.io.FileInputStream” relacionado ao arquivo onde o estado do objeto encontra-se persistido. Deve-se criar também um novo “java.io.ObjectInputStream” que estará vinculado ao arquivo de origem. Por último, executa-se a leitura do arquivo para a restauração do estado. Na Listagem 3 encontra-se o exemplo do código.

Listagem 3 - Deserializaçãode um objeto.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | import java.io.FileInputStream;  import java.io.ObjectInputStream;    public class Deserializador {        public Deserializador() {    }        public Object deserializar(String path) throws Exception {      FileInputStream inFile = new FileInputStream(path);      ObjectInputStream d = new ObjectInputStream(inFile);      Object o = d.readObject();      d.close();      return o;      }  } |

A Listagem 4, contém o exemplo da execução da serialização e em seguida da deserialização.

Listagem 4 - Serializando e Deserializando

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | import operacoes.Deserializador;  import operacoes.Serializador;  import pojo.Fruta;    public class Principal {      public static void main(String args[]){         //serializa         Serializador s = new Serializador();         Fruta fruta = new Fruta("Maça", "Vermelha");          try {              s.serializar("/home/gustavo/fruta", fruta);          } catch (Exception ex) {              System.err.println("Falha ao serializar! - " + ex.toString());          }            //deserializa          Deserializador d = new Deserializador();          fruta = null;          try {              fruta = (Fruta) d.deserializar("/home/gustavo/fruta");          } catch (Exception ex) {              System.err.println("Falha ao deserializar! - " + ex.toString());          }          System.out.println(fruta.getNome() + " - " + fruta.getCor());      }  } |

Na Listagem 5, encontra-se um exemplo de serialização e deserialização de um ArrayList da classe Fruta. É apenas mais um exemplo que demonstra a capacidade de se persistir qualquer objeto que seja serializável. Essa serialização só foi possível porque a classe “ArrayList” implementa a interface “Serializable” por padrão.

Listagem 5 - Serialização de ArrayList.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | //serialização e deserialização com ArrayList          ArrayList<Fruta> frutas = new ArrayList<Fruta>();          Fruta f1 = new Fruta("Laranja", "Amarela");          Fruta f2 = new Fruta("Abacate", "Verde");          Fruta f3 = new Fruta("Morango", "Vermelho");          frutas.add(f1);          frutas.add(f2);          frutas.add(f3);            try {              s.serializar("/home/gustavo/frutas", frutas);              frutas = null;              frutas = (ArrayList<Fruta>) d.deserializar("/home/gustavo/frutas");              for(Fruta f : frutas){                  System.out.println("ArrayList: " + f.getNome() + " - " + f.getCor());              }          } catch (Exception ex) {              System.err.println("Falha ao serializar ou deserializar! - " +                                 ex.toString());          } |

**4. PARTICULARIDADES DA SERIALIZAÇÃO**

Como dito, a serialização de um objeto em Java pode ser algo fácil de se fazer. Mas existem certas informações que são importantes e sabê-las pode evitar vários problemas.

Um dos problemas é que nem tudo pode ser serializado, como objetos das classes “Thread” e “Socket” e atributos estáticos. Os atributos estáticos estão vinculados à classe e não ao objeto. Assim quando se trata do estado do objeto, os campos estáticos não são considerados. Em relação às Threads, suas instâncias só são úteis enquanto estiverem em execução na JVM, e podem ser paradas/destruídas e iniciadas com instruções de sua classe. Não existe algo no estado destas instâncias que seja útil para se persistir, e por isso não há sentido em persistir tais instâncias para restaurar posteriormente.

Mas e se existir alguma classe em que seja necessário persisti-la, e esta possui uma instância da classe “Thread”? Neste caso, utilizando um recurso pode-se persistir a classe inteira, menos a instância thread. Este recurso, refere-se em indicar que tal instância thread é um objeto “Transient”. Na Listagem 6 encontra-se um exemplo da implementação com uso de “Transient”.

Listagem 6 - Uso do Transient

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | import java.io.Serializable;    public class ImplementacaoTransient implements Serializable, Runnable{        transient private Thread thread;      //marcado com "transient" para que sua instância não seja serializada        //Demais valores de estado serão serializados      private int valorParaPersistir;        public ImplementacaoTransient(int valorParaPersistir){          this.valorParaPersistir = valorParaPersistir;          thread = new Thread(this);          thread.start();      }        public void run()      {          while(true){              //código a ser executado pela Thread          }      }  } |

Vale ressaltar que, o que é serializado é apenas o estado do objeto, e não os métodos e a classe. No exemplo acima, apenas a instância do objeto “thread” e do objeto “valorParaPersistir” compunham o estado do objeto. Assim, evita-se que o sistema de serialização tente serializar um objeto não-serializável e gere erros. Após a deserialização de um objeto da classe “ImplementacaoTransient” este estará totalmente funcional.

Um outro problema que pode vir a ocorrer, se refere ao versionamento das classes. Toda classe possui um atributo oculto chamado “serialVersionUID” com um valor gerado automaticamente. Objetos criados levam consigo o “serialVersionUID” de sua classe, que indica sua versão. Toda vez que uma classe sofre alterações em seu código, um novo “serialVersionUID” é gerado. O problema com a serialização surge quando serializa-se o estado de um objeto e posteriormente tenta-se restaurar o estado para a aplicação, mas houveram mudanças no código da classe. Neste caso, uma exceção “java.io.InvalidClassException” é disparada, pois de fato, o estado do objeto corresponde a uma versão de classe do qual foi criado que já não existe mais. Portanto, isso só ocorre para evitar erros de programação, em que tenta-se restaurar o estado de um objeto para uma classe que possui versão diferente da qual o objeto foi serializado inicialmente. E isso pode significar diferentes versões de uma mesma classe “Fruta” por exemplo, ou diferentes versões por classes serem totalmente difere

|  |  |
| --- | --- |
| http://www.linhadecodigo.com.br/artigos/img_artigos/EvertonCoimbraAraujo/Serializazao_Objetos/imagem001.jpg | Neste artigo apresento o Gustavo Rafael Valiati, que pode ser contatado pelo e-mail gustavovaliati@gmail.com. O Gustavo foi meu aluno em algumas disciplinas de sua gradução e está concluíndo o último período do curso de Tecnologia em Análise e Desenvolvimento de Sistemas da UTFPR campus Medianeira, neste primeiro semestre de 2012. |

**1. INTRODUÇÃO**

Nem sempre se deseja ter objetos ou estrutura de dados de um aplicativo, com um ciclo de vida que dura no máximo o tempo em que for armazenado em memória RAM. Ou seja, algumas vezes deseja-se guardá-los por um tempo indeterminado, sendo possível recuperá-los posteriormente. E neste caso, o termo “persistência” tem seu conceito definido.

Persistir dados, requer que estes possam ser serializáveis. A serialização possibilitará capturar o estado do objeto ou a estrutura de dados, e transformar em uma cadeia de bytes quando for necessário. Torna-se possível também, recuperar os bytes persistidos para fazer o processo inverso, e ter os dados de volta para a aplicação em execução.

Desta maneira, qualquer forma de dados que esteja sendo trabalhada na aplicação, e esta puder ser serializada, poderá ser passada para o meio físico, e ter o processo revertido.

A serialização não se limita apenas em possibilitar a gravação do dado em disco, mas permite que o mesmo seja transmitido por rede. Isto porquê a serialização cria um stream de bytes, que é requisito para transmissão de dados por rede.

Diversas linguagens possuem suporte à serialização, como: Java, C, C++, Python, PHP, .NET, entre outras. Neste artigo, será tratado apenas da serialização com Java, em sua implemetação básica, alguns problemas e soluções, com códigos para exemplificação.

**2. SERIALIZAÇÃO COM JAVA**

Serializar um objeto, em Java, só é possível caso sua classe esteja marcada como serializável. Essa marcação é feita com a simples implementação da interface “java.io.Serializable”. Essa interface não possui métodos para se implementar e nem atributos. Ainda não existiam as anotações no Java 1.1, versão que trouxe pela primeira vez a serialização ao Java, e desta maneira a implementação da interface era a melhor maneira de marcar uma classe.

A herança entre classes é naturalmente afetada pela serialização. Já que quando uma classe implementa a interface “Serializable”, toda classe que dela estender também estará implicitamente marcada como serializável

.

Para exemplo, será criada uma classe “pojo.Fruta” que contém dois atributos: nome e cor. No código exemplo (Listagem 1), pode-se notar a classe “Fruta” marcada para a serialização:

Listagem 1 - Marcando uma classe como serializável.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | import java.io.Serializable;    public class Fruta implements Serializable{      private String nome;      private String cor;        public Fruta(String nome, String cor) {          this.nome = nome;          this.cor = cor;      }        public String getCor() { return cor; }      public String getNome() { return nome; }      public void setCor(String cor) { this.cor = cor; }      public void setNome(String nome) { this.nome = nome; }  } |

Pode-se dizer que a implementação do processo de serialização e deserialização simples de um objeto, não possui um código difícil de se escrever. Isto porquê duas classes fazem a maior parte do trabalho, de maneira que com poucas linhas de código pode-se transformar um objeto em bytes e vice-versa. Essas duas classes são “java.io.ObjectInputStream” que recebe bytes para criar um objeto e “java.io.ObjectOutputStream” que cria uma cadeia de bytes a partir de um objeto.

**3. SERIALIZANDO E DESERIALIZANDO O ESTADO DE UM OBJETO**

Como exemplo, será persistido um objeto em disco. Para isto, além das classes “java.io.ObjectInputStream” e “java.io.ObjectOutputStream”, serão utilizadas duas outras que trabalham com arquivos, que são “java.io.FileInputStream” e “java.io.FileOutputStream”, que seguem a mesma lógica de InputStream e OutputStream, só que agora relacionado ao sistema de arquivos e não aos objetos.

Para serialização, deve-se criar um novo “java.io.FileOutputStream”, com o caminho (“path”) do arquivo onde se deseja armazenar o estado do objeto. Também deve ser criado um novo “java.io.ObjectOutputStream” que está ligado ao arquivo. E por último deve-se executar a escrita do objeto alvo, para dentro do objeto de stream que por sua vez está ligado ao arquivo destino. Após executados esses passos, é possível encontrar no sistema de arquivos um novo arquivo que contém os bytes correspondentes à descrição do estado do objeto. Na Listagem 2, encontra-se o código exemplo.

Listagem 2 - Serialização de um objeto.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | import java.io.FileOutputStream;  import java.io.ObjectOutputStream;    public class Serializador {        public Serializador() {    }        public void serializar(String path, Object obj) throws Exception {              FileOutputStream outFile = new FileOutputStream(path);      ObjectOutputStream s = new ObjectOutputStream(outFile);      s.writeObject(obj);      s.close();      }  } |

E para a deserialização, o processo é extremamente parecido com a serialização. Cria-se um novo “java.io.FileInputStream” relacionado ao arquivo onde o estado do objeto encontra-se persistido. Deve-se criar também um novo “java.io.ObjectInputStream” que estará vinculado ao arquivo de origem. Por último, executa-se a leitura do arquivo para a restauração do estado. Na Listagem 3 encontra-se o exemplo do código.

Listagem 3 - Deserializaçãode um objeto.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | import java.io.FileInputStream;  import java.io.ObjectInputStream;    public class Deserializador {        public Deserializador() {    }        public Object deserializar(String path) throws Exception {      FileInputStream inFile = new FileInputStream(path);      ObjectInputStream d = new ObjectInputStream(inFile);      Object o = d.readObject();      d.close();      return o;      }  } |

A Listagem 4, contém o exemplo da execução da serialização e em seguida da deserialização.

Listagem 4 - Serializando e Deserializando

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | import operacoes.Deserializador;  import operacoes.Serializador;  import pojo.Fruta;    public class Principal {      public static void main(String args[]){         //serializa         Serializador s = new Serializador();         Fruta fruta = new Fruta("Maça", "Vermelha");          try {              s.serializar("/home/gustavo/fruta", fruta);          } catch (Exception ex) {              System.err.println("Falha ao serializar! - " + ex.toString());          }            //deserializa          Deserializador d = new Deserializador();          fruta = null;          try {              fruta = (Fruta) d.deserializar("/home/gustavo/fruta");          } catch (Exception ex) {              System.err.println("Falha ao deserializar! - " + ex.toString());          }          System.out.println(fruta.getNome() + " - " + fruta.getCor());      }  } |

Na Listagem 5, encontra-se um exemplo de serialização e deserialização de um ArrayList da classe Fruta. É apenas mais um exemplo que demonstra a capacidade de se persistir qualquer objeto que seja serializável. Essa serialização só foi possível porque a classe “ArrayList” implementa a interface “Serializable” por padrão.

Listagem 5 - Serialização de ArrayList.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | //serialização e deserialização com ArrayList          ArrayList<Fruta> frutas = new ArrayList<Fruta>();          Fruta f1 = new Fruta("Laranja", "Amarela");          Fruta f2 = new Fruta("Abacate", "Verde");          Fruta f3 = new Fruta("Morango", "Vermelho");          frutas.add(f1);          frutas.add(f2);          frutas.add(f3);            try {              s.serializar("/home/gustavo/frutas", frutas);              frutas = null;              frutas = (ArrayList<Fruta>) d.deserializar("/home/gustavo/frutas");              for(Fruta f : frutas){                  System.out.println("ArrayList: " + f.getNome() + " - " + f.getCor());              }          } catch (Exception ex) {              System.err.println("Falha ao serializar ou deserializar! - " +                                 ex.toString());          } |

**4. PARTICULARIDADES DA SERIALIZAÇÃO**

Como dito, a serialização de um objeto em Java pode ser algo fácil de se fazer. Mas existem certas informações que são importantes e sabê-las pode evitar vários problemas.

Um dos problemas é que nem tudo pode ser serializado, como objetos das classes “Thread” e “Socket” e atributos estáticos. Os atributos estáticos estão vinculados à classe e não ao objeto. Assim quando se trata do estado do objeto, os campos estáticos não são considerados. Em relação às Threads, suas instâncias só são úteis enquanto estiverem em execução na JVM, e podem ser paradas/destruídas e iniciadas com instruções de sua classe. Não existe algo no estado destas instâncias que seja útil para se persistir, e por isso não há sentido em persistir tais instâncias para restaurar posteriormente.

Mas e se existir alguma classe em que seja necessário persisti-la, e esta possui uma instância da classe “Thread”? Neste caso, utilizando um recurso pode-se persistir a classe inteira, menos a instância thread. Este recurso, refere-se em indicar que tal instância thread é um objeto “Transient”. Na Listagem 6 encontra-se um exemplo da implementação com uso de “Transient”.

Listagem 6 - Uso do Transient

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | import java.io.Serializable;    public class ImplementacaoTransient implements Serializable, Runnable{        transient private Thread thread;      //marcado com "transient" para que sua instância não seja serializada        //Demais valores de estado serão serializados      private int valorParaPersistir;        public ImplementacaoTransient(int valorParaPersistir){          this.valorParaPersistir = valorParaPersistir;          thread = new Thread(this);          thread.start();      }        public void run()      {          while(true){              //código a ser executado pela Thread          }      }  } |

Vale ressaltar que, o que é serializado é apenas o estado do objeto, e não os métodos e a classe. No exemplo acima, apenas a instância do objeto “thread” e do objeto “valorParaPersistir” compunham o estado do objeto. Assim, evita-se que o sistema de serialização tente serializar um objeto não-serializável e gere erros. Após a deserialização de um objeto da classe “ImplementacaoTransient” este estará totalmente funcional.

Um outro problema que pode vir a ocorrer, se refere ao versionamento das classes. Toda classe possui um atributo oculto chamado “serialVersionUID” com um valor gerado automaticamente. Objetos criados levam consigo o “serialVersionUID” de sua classe, que indica sua versão. Toda vez que uma classe sofre alterações em seu código, um novo “serialVersionUID” é gerado. O problema com a serialização surge quando serializa-se o estado de um objeto e posteriormente tenta-se restaurar o estado para a aplicação, mas houveram mudanças no código da classe. Neste caso, uma exceção “java.io.InvalidClassException” é disparada, pois de fato, o estado do objeto corresponde a uma versão de classe do qual foi criado que já não existe mais. Portanto, isso só ocorre para evitar erros de programação, em que tenta-se restaurar o estado de um objeto para uma classe que possui versão diferente da qual o objeto foi serializado inicialmente. E isso pode significar diferentes versões de uma mesma classe “Fruta” por exemplo, ou diferentes versões por classes serem totalmente difere

|  |  |
| --- | --- |
| http://www.linhadecodigo.com.br/artigos/img_artigos/EvertonCoimbraAraujo/Serializazao_Objetos/imagem001.jpg | Neste artigo apresento o Gustavo Rafael Valiati, que pode ser contatado pelo e-mail gustavovaliati@gmail.com. O Gustavo foi meu aluno em algumas disciplinas de sua gradução e está concluíndo o último período do curso de Tecnologia em Análise e Desenvolvimento de Sistemas da UTFPR campus Medianeira, neste primeiro semestre de 2012. |

**1. INTRODUÇÃO**

Nem sempre se deseja ter objetos ou estrutura de dados de um aplicativo, com um ciclo de vida que dura no máximo o tempo em que for armazenado em memória RAM. Ou seja, algumas vezes deseja-se guardá-los por um tempo indeterminado, sendo possível recuperá-los posteriormente. E neste caso, o termo “persistência” tem seu conceito definido.

Persistir dados, requer que estes possam ser serializáveis. A serialização possibilitará capturar o estado do objeto ou a estrutura de dados, e transformar em uma cadeia de bytes quando for necessário. Torna-se possível também, recuperar os bytes persistidos para fazer o processo inverso, e ter os dados de volta para a aplicação em execução.

Desta maneira, qualquer forma de dados que esteja sendo trabalhada na aplicação, e esta puder ser serializada, poderá ser passada para o meio físico, e ter o processo revertido.

A serialização não se limita apenas em possibilitar a gravação do dado em disco, mas permite que o mesmo seja transmitido por rede. Isto porquê a serialização cria um stream de bytes, que é requisito para transmissão de dados por rede.

Diversas linguagens possuem suporte à serialização, como: Java, C, C++, Python, PHP, .NET, entre outras. Neste artigo, será tratado apenas da serialização com Java, em sua implemetação básica, alguns problemas e soluções, com códigos para exemplificação.

**2. SERIALIZAÇÃO COM JAVA**

Serializar um objeto, em Java, só é possível caso sua classe esteja marcada como serializável. Essa marcação é feita com a simples implementação da interface “java.io.Serializable”. Essa interface não possui métodos para se implementar e nem atributos. Ainda não existiam as anotações no Java 1.1, versão que trouxe pela primeira vez a serialização ao Java, e desta maneira a implementação da interface era a melhor maneira de marcar uma classe.

A herança entre classes é naturalmente afetada pela serialização. Já que quando uma classe implementa a interface “Serializable”, toda classe que dela estender também estará implicitamente marcada como serializável

.

Para exemplo, será criada uma classe “pojo.Fruta” que contém dois atributos: nome e cor. No código exemplo (Listagem 1), pode-se notar a classe “Fruta” marcada para a serialização:

Listagem 1 - Marcando uma classe como serializável.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | import java.io.Serializable;    public class Fruta implements Serializable{      private String nome;      private String cor;        public Fruta(String nome, String cor) {          this.nome = nome;          this.cor = cor;      }        public String getCor() { return cor; }      public String getNome() { return nome; }      public void setCor(String cor) { this.cor = cor; }      public void setNome(String nome) { this.nome = nome; }  } |

Pode-se dizer que a implementação do processo de serialização e deserialização simples de um objeto, não possui um código difícil de se escrever. Isto porquê duas classes fazem a maior parte do trabalho, de maneira que com poucas linhas de código pode-se transformar um objeto em bytes e vice-versa. Essas duas classes são “java.io.ObjectInputStream” que recebe bytes para criar um objeto e “java.io.ObjectOutputStream” que cria uma cadeia de bytes a partir de um objeto.

**3. SERIALIZANDO E DESERIALIZANDO O ESTADO DE UM OBJETO**

Como exemplo, será persistido um objeto em disco. Para isto, além das classes “java.io.ObjectInputStream” e “java.io.ObjectOutputStream”, serão utilizadas duas outras que trabalham com arquivos, que são “java.io.FileInputStream” e “java.io.FileOutputStream”, que seguem a mesma lógica de InputStream e OutputStream, só que agora relacionado ao sistema de arquivos e não aos objetos.

Para serialização, deve-se criar um novo “java.io.FileOutputStream”, com o caminho (“path”) do arquivo onde se deseja armazenar o estado do objeto. Também deve ser criado um novo “java.io.ObjectOutputStream” que está ligado ao arquivo. E por último deve-se executar a escrita do objeto alvo, para dentro do objeto de stream que por sua vez está ligado ao arquivo destino. Após executados esses passos, é possível encontrar no sistema de arquivos um novo arquivo que contém os bytes correspondentes à descrição do estado do objeto. Na Listagem 2, encontra-se o código exemplo.

Listagem 2 - Serialização de um objeto.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | import java.io.FileOutputStream;  import java.io.ObjectOutputStream;    public class Serializador {        public Serializador() {    }        public void serializar(String path, Object obj) throws Exception {              FileOutputStream outFile = new FileOutputStream(path);      ObjectOutputStream s = new ObjectOutputStream(outFile);      s.writeObject(obj);      s.close();      }  } |

E para a deserialização, o processo é extremamente parecido com a serialização. Cria-se um novo “java.io.FileInputStream” relacionado ao arquivo onde o estado do objeto encontra-se persistido. Deve-se criar também um novo “java.io.ObjectInputStream” que estará vinculado ao arquivo de origem. Por último, executa-se a leitura do arquivo para a restauração do estado. Na Listagem 3 encontra-se o exemplo do código.

Listagem 3 - Deserializaçãode um objeto.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | import java.io.FileInputStream;  import java.io.ObjectInputStream;    public class Deserializador {        public Deserializador() {    }        public Object deserializar(String path) throws Exception {      FileInputStream inFile = new FileInputStream(path);      ObjectInputStream d = new ObjectInputStream(inFile);      Object o = d.readObject();      d.close();      return o;      }  } |

A Listagem 4, contém o exemplo da execução da serialização e em seguida da deserialização.

Listagem 4 - Serializando e Deserializando

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | import operacoes.Deserializador;  import operacoes.Serializador;  import pojo.Fruta;    public class Principal {      public static void main(String args[]){         //serializa         Serializador s = new Serializador();         Fruta fruta = new Fruta("Maça", "Vermelha");          try {              s.serializar("/home/gustavo/fruta", fruta);          } catch (Exception ex) {              System.err.println("Falha ao serializar! - " + ex.toString());          }            //deserializa          Deserializador d = new Deserializador();          fruta = null;          try {              fruta = (Fruta) d.deserializar("/home/gustavo/fruta");          } catch (Exception ex) {              System.err.println("Falha ao deserializar! - " + ex.toString());          }          System.out.println(fruta.getNome() + " - " + fruta.getCor());      }  } |

Na Listagem 5, encontra-se um exemplo de serialização e deserialização de um ArrayList da classe Fruta. É apenas mais um exemplo que demonstra a capacidade de se persistir qualquer objeto que seja serializável. Essa serialização só foi possível porque a classe “ArrayList” implementa a interface “Serializable” por padrão.

Listagem 5 - Serialização de ArrayList.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | //serialização e deserialização com ArrayList          ArrayList<Fruta> frutas = new ArrayList<Fruta>();          Fruta f1 = new Fruta("Laranja", "Amarela");          Fruta f2 = new Fruta("Abacate", "Verde");          Fruta f3 = new Fruta("Morango", "Vermelho");          frutas.add(f1);          frutas.add(f2);          frutas.add(f3);            try {              s.serializar("/home/gustavo/frutas", frutas);              frutas = null;              frutas = (ArrayList<Fruta>) d.deserializar("/home/gustavo/frutas");              for(Fruta f : frutas){                  System.out.println("ArrayList: " + f.getNome() + " - " + f.getCor());              }          } catch (Exception ex) {              System.err.println("Falha ao serializar ou deserializar! - " +                                 ex.toString());          } |

**4. PARTICULARIDADES DA SERIALIZAÇÃO**

Como dito, a serialização de um objeto em Java pode ser algo fácil de se fazer. Mas existem certas informações que são importantes e sabê-las pode evitar vários problemas.

Um dos problemas é que nem tudo pode ser serializado, como objetos das classes “Thread” e “Socket” e atributos estáticos. Os atributos estáticos estão vinculados à classe e não ao objeto. Assim quando se trata do estado do objeto, os campos estáticos não são considerados. Em relação às Threads, suas instâncias só são úteis enquanto estiverem em execução na JVM, e podem ser paradas/destruídas e iniciadas com instruções de sua classe. Não existe algo no estado destas instâncias que seja útil para se persistir, e por isso não há sentido em persistir tais instâncias para restaurar posteriormente.

Mas e se existir alguma classe em que seja necessário persisti-la, e esta possui uma instância da classe “Thread”? Neste caso, utilizando um recurso pode-se persistir a classe inteira, menos a instância thread. Este recurso, refere-se em indicar que tal instância thread é um objeto “Transient”. Na Listagem 6 encontra-se um exemplo da implementação com uso de “Transient”.

Listagem 6 - Uso do Transient

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | import java.io.Serializable;    public class ImplementacaoTransient implements Serializable, Runnable{        transient private Thread thread;      //marcado com "transient" para que sua instância não seja serializada        //Demais valores de estado serão serializados      private int valorParaPersistir;        public ImplementacaoTransient(int valorParaPersistir){          this.valorParaPersistir = valorParaPersistir;          thread = new Thread(this);          thread.start();      }        public void run()      {          while(true){              //código a ser executado pela Thread          }      }  } |

Vale ressaltar que, o que é serializado é apenas o estado do objeto, e não os métodos e a classe. No exemplo acima, apenas a instância do objeto “thread” e do objeto “valorParaPersistir” compunham o estado do objeto. Assim, evita-se que o sistema de serialização tente serializar um objeto não-serializável e gere erros. Após a deserialização de um objeto da classe “ImplementacaoTransient” este estará totalmente funcional.

Um outro problema que pode vir a ocorrer, se refere ao versionamento das classes. Toda classe possui um atributo oculto chamado “serialVersionUID” com um valor gerado automaticamente. Objetos criados levam consigo o “serialVersionUID” de sua classe, que indica sua versão. Toda vez que uma classe sofre alterações em seu código, um novo “serialVersionUID” é gerado. O problema com a serialização surge quando serializa-se o estado de um objeto e posteriormente tenta-se restaurar o estado para a aplicação, mas houveram mudanças no código da classe. Neste caso, uma exceção “java.io.InvalidClassException” é disparada, pois de fato, o estado do objeto corresponde a uma versão de classe do qual foi criado que já não existe mais. Portanto, isso só ocorre para evitar erros de programação, em que tenta-se restaurar o estado de um objeto para uma classe que possui versão diferente da qual o objeto foi serializado inicialmente. E isso pode significar diferentes versões de uma mesma classe “Fruta” por exemplo, ou diferentes versões por classes serem totalmente difere

|  |  |
| --- | --- |
| http://www.linhadecodigo.com.br/artigos/img_artigos/EvertonCoimbraAraujo/Serializazao_Objetos/imagem001.jpg | Neste artigo apresento o Gustavo Rafael Valiati, que pode ser contatado pelo e-mail gustavovaliati@gmail.com. O Gustavo foi meu aluno em algumas disciplinas de sua gradução e está concluíndo o último período do curso de Tecnologia em Análise e Desenvolvimento de Sistemas da UTFPR campus Medianeira, neste primeiro semestre de 2012. |

**1. INTRODUÇÃO**

Nem sempre se deseja ter objetos ou estrutura de dados de um aplicativo, com um ciclo de vida que dura no máximo o tempo em que for armazenado em memória RAM. Ou seja, algumas vezes deseja-se guardá-los por um tempo indeterminado, sendo possível recuperá-los posteriormente. E neste caso, o termo “persistência” tem seu conceito definido.

Persistir dados, requer que estes possam ser serializáveis. A serialização possibilitará capturar o estado do objeto ou a estrutura de dados, e transformar em uma cadeia de bytes quando for necessário. Torna-se possível também, recuperar os bytes persistidos para fazer o processo inverso, e ter os dados de volta para a aplicação em execução.

Desta maneira, qualquer forma de dados que esteja sendo trabalhada na aplicação, e esta puder ser serializada, poderá ser passada para o meio físico, e ter o processo revertido.

A serialização não se limita apenas em possibilitar a gravação do dado em disco, mas permite que o mesmo seja transmitido por rede. Isto porquê a serialização cria um stream de bytes, que é requisito para transmissão de dados por rede.

Diversas linguagens possuem suporte à serialização, como: Java, C, C++, Python, PHP, .NET, entre outras. Neste artigo, será tratado apenas da serialização com Java, em sua implemetação básica, alguns problemas e soluções, com códigos para exemplificação.

**2. SERIALIZAÇÃO COM JAVA**

Serializar um objeto, em Java, só é possível caso sua classe esteja marcada como serializável. Essa marcação é feita com a simples implementação da interface “java.io.Serializable”. Essa interface não possui métodos para se implementar e nem atributos. Ainda não existiam as anotações no Java 1.1, versão que trouxe pela primeira vez a serialização ao Java, e desta maneira a implementação da interface era a melhor maneira de marcar uma classe.

A herança entre classes é naturalmente afetada pela serialização. Já que quando uma classe implementa a interface “Serializable”, toda classe que dela estender também estará implicitamente marcada como serializável

.

Para exemplo, será criada uma classe “pojo.Fruta” que contém dois atributos: nome e cor. No código exemplo (Listagem 1), pode-se notar a classe “Fruta” marcada para a serialização:

Listagem 1 - Marcando uma classe como serializável.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | import java.io.Serializable;    public class Fruta implements Serializable{      private String nome;      private String cor;        public Fruta(String nome, String cor) {          this.nome = nome;          this.cor = cor;      }        public String getCor() { return cor; }      public String getNome() { return nome; }      public void setCor(String cor) { this.cor = cor; }      public void setNome(String nome) { this.nome = nome; }  } |

Pode-se dizer que a implementação do processo de serialização e deserialização simples de um objeto, não possui um código difícil de se escrever. Isto porquê duas classes fazem a maior parte do trabalho, de maneira que com poucas linhas de código pode-se transformar um objeto em bytes e vice-versa. Essas duas classes são “java.io.ObjectInputStream” que recebe bytes para criar um objeto e “java.io.ObjectOutputStream” que cria uma cadeia de bytes a partir de um objeto.

**3. SERIALIZANDO E DESERIALIZANDO O ESTADO DE UM OBJETO**

Como exemplo, será persistido um objeto em disco. Para isto, além das classes “java.io.ObjectInputStream” e “java.io.ObjectOutputStream”, serão utilizadas duas outras que trabalham com arquivos, que são “java.io.FileInputStream” e “java.io.FileOutputStream”, que seguem a mesma lógica de InputStream e OutputStream, só que agora relacionado ao sistema de arquivos e não aos objetos.

Para serialização, deve-se criar um novo “java.io.FileOutputStream”, com o caminho (“path”) do arquivo onde se deseja armazenar o estado do objeto. Também deve ser criado um novo “java.io.ObjectOutputStream” que está ligado ao arquivo. E por último deve-se executar a escrita do objeto alvo, para dentro do objeto de stream que por sua vez está ligado ao arquivo destino. Após executados esses passos, é possível encontrar no sistema de arquivos um novo arquivo que contém os bytes correspondentes à descrição do estado do objeto. Na Listagem 2, encontra-se o código exemplo.

Listagem 2 - Serialização de um objeto.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | import java.io.FileOutputStream;  import java.io.ObjectOutputStream;    public class Serializador {        public Serializador() {    }        public void serializar(String path, Object obj) throws Exception {              FileOutputStream outFile = new FileOutputStream(path);      ObjectOutputStream s = new ObjectOutputStream(outFile);      s.writeObject(obj);      s.close();      }  } |

E para a deserialização, o processo é extremamente parecido com a serialização. Cria-se um novo “java.io.FileInputStream” relacionado ao arquivo onde o estado do objeto encontra-se persistido. Deve-se criar também um novo “java.io.ObjectInputStream” que estará vinculado ao arquivo de origem. Por último, executa-se a leitura do arquivo para a restauração do estado. Na Listagem 3 encontra-se o exemplo do código.

Listagem 3 - Deserializaçãode um objeto.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | import java.io.FileInputStream;  import java.io.ObjectInputStream;    public class Deserializador {        public Deserializador() {    }        public Object deserializar(String path) throws Exception {      FileInputStream inFile = new FileInputStream(path);      ObjectInputStream d = new ObjectInputStream(inFile);      Object o = d.readObject();      d.close();      return o;      }  } |

A Listagem 4, contém o exemplo da execução da serialização e em seguida da deserialização.

Listagem 4 - Serializando e Deserializando

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | import operacoes.Deserializador;  import operacoes.Serializador;  import pojo.Fruta;    public class Principal {      public static void main(String args[]){         //serializa         Serializador s = new Serializador();         Fruta fruta = new Fruta("Maça", "Vermelha");          try {              s.serializar("/home/gustavo/fruta", fruta);          } catch (Exception ex) {              System.err.println("Falha ao serializar! - " + ex.toString());          }            //deserializa          Deserializador d = new Deserializador();          fruta = null;          try {              fruta = (Fruta) d.deserializar("/home/gustavo/fruta");          } catch (Exception ex) {              System.err.println("Falha ao deserializar! - " + ex.toString());          }          System.out.println(fruta.getNome() + " - " + fruta.getCor());      }  } |

Na Listagem 5, encontra-se um exemplo de serialização e deserialização de um ArrayList da classe Fruta. É apenas mais um exemplo que demonstra a capacidade de se persistir qualquer objeto que seja serializável. Essa serialização só foi possível porque a classe “ArrayList” implementa a interface “Serializable” por padrão.

Listagem 5 - Serialização de ArrayList.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | //serialização e deserialização com ArrayList          ArrayList<Fruta> frutas = new ArrayList<Fruta>();          Fruta f1 = new Fruta("Laranja", "Amarela");          Fruta f2 = new Fruta("Abacate", "Verde");          Fruta f3 = new Fruta("Morango", "Vermelho");          frutas.add(f1);          frutas.add(f2);          frutas.add(f3);            try {              s.serializar("/home/gustavo/frutas", frutas);              frutas = null;              frutas = (ArrayList<Fruta>) d.deserializar("/home/gustavo/frutas");              for(Fruta f : frutas){                  System.out.println("ArrayList: " + f.getNome() + " - " + f.getCor());              }          } catch (Exception ex) {              System.err.println("Falha ao serializar ou deserializar! - " +                                 ex.toString());          } |

**4. PARTICULARIDADES DA SERIALIZAÇÃO**

Como dito, a serialização de um objeto em Java pode ser algo fácil de se fazer. Mas existem certas informações que são importantes e sabê-las pode evitar vários problemas.

Um dos problemas é que nem tudo pode ser serializado, como objetos das classes “Thread” e “Socket” e atributos estáticos. Os atributos estáticos estão vinculados à classe e não ao objeto. Assim quando se trata do estado do objeto, os campos estáticos não são considerados. Em relação às Threads, suas instâncias só são úteis enquanto estiverem em execução na JVM, e podem ser paradas/destruídas e iniciadas com instruções de sua classe. Não existe algo no estado destas instâncias que seja útil para se persistir, e por isso não há sentido em persistir tais instâncias para restaurar posteriormente.

Mas e se existir alguma classe em que seja necessário persisti-la, e esta possui uma instância da classe “Thread”? Neste caso, utilizando um recurso pode-se persistir a classe inteira, menos a instância thread. Este recurso, refere-se em indicar que tal instância thread é um objeto “Transient”. Na Listagem 6 encontra-se um exemplo da implementação com uso de “Transient”.

Listagem 6 - Uso do Transient

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | import java.io.Serializable;    public class ImplementacaoTransient implements Serializable, Runnable{        transient private Thread thread;      //marcado com "transient" para que sua instância não seja serializada        //Demais valores de estado serão serializados      private int valorParaPersistir;        public ImplementacaoTransient(int valorParaPersistir){          this.valorParaPersistir = valorParaPersistir;          thread = new Thread(this);          thread.start();      }        public void run()      {          while(true){              //código a ser executado pela Thread          }      }  } |

Vale ressaltar que, o que é serializado é apenas o estado do objeto, e não os métodos e a classe. No exemplo acima, apenas a instância do objeto “thread” e do objeto “valorParaPersistir” compunham o estado do objeto. Assim, evita-se que o sistema de serialização tente serializar um objeto não-serializável e gere erros. Após a deserialização de um objeto da classe “ImplementacaoTransient” este estará totalmente funcional.

Um outro problema que pode vir a ocorrer, se refere ao versionamento das classes. Toda classe possui um atributo oculto chamado “serialVersionUID” com um valor gerado automaticamente. Objetos criados levam consigo o “serialVersionUID” de sua classe, que indica sua versão. Toda vez que uma classe sofre alterações em seu código, um novo “serialVersionUID” é gerado. O problema com a serialização surge quando serializa-se o estado de um objeto e posteriormente tenta-se restaurar o estado para a aplicação, mas houveram mudanças no código da classe. Neste caso, uma exceção “java.io.InvalidClassException” é disparada, pois de fato, o estado do objeto corresponde a uma versão de classe do qual foi criado que já não existe mais. Portanto, isso só ocorre para evitar erros de programação, em que tenta-se restaurar o estado de um objeto para uma classe que possui versão diferente da qual o objeto foi serializado inicialmente. E isso pode significar diferentes versões de uma mesma classe “Fruta” por exemplo, ou diferentes versões por classes serem totalmente difere

|  |  |
| --- | --- |
| http://www.linhadecodigo.com.br/artigos/img_artigos/EvertonCoimbraAraujo/Serializazao_Objetos/imagem001.jpg | Neste artigo apresento o Gustavo Rafael Valiati, que pode ser contatado pelo e-mail gustavovaliati@gmail.com. O Gustavo foi meu aluno em algumas disciplinas de sua gradução e está concluíndo o último período do curso de Tecnologia em Análise e Desenvolvimento de Sistemas da UTFPR campus Medianeira, neste primeiro semestre de 2012. |

**1. INTRODUÇÃO**

Nem sempre se deseja ter objetos ou estrutura de dados de um aplicativo, com um ciclo de vida que dura no máximo o tempo em que for armazenado em memória RAM. Ou seja, algumas vezes deseja-se guardá-los por um tempo indeterminado, sendo possível recuperá-los posteriormente. E neste caso, o termo “persistência” tem seu conceito definido.

Persistir dados, requer que estes possam ser serializáveis. A serialização possibilitará capturar o estado do objeto ou a estrutura de dados, e transformar em uma cadeia de bytes quando for necessário. Torna-se possível também, recuperar os bytes persistidos para fazer o processo inverso, e ter os dados de volta para a aplicação em execução.

Desta maneira, qualquer forma de dados que esteja sendo trabalhada na aplicação, e esta puder ser serializada, poderá ser passada para o meio físico, e ter o processo revertido.

A serialização não se limita apenas em possibilitar a gravação do dado em disco, mas permite que o mesmo seja transmitido por rede. Isto porquê a serialização cria um stream de bytes, que é requisito para transmissão de dados por rede.

Diversas linguagens possuem suporte à serialização, como: Java, C, C++, Python, PHP, .NET, entre outras. Neste artigo, será tratado apenas da serialização com Java, em sua implemetação básica, alguns problemas e soluções, com códigos para exemplificação.

**2. SERIALIZAÇÃO COM JAVA**

Serializar um objeto, em Java, só é possível caso sua classe esteja marcada como serializável. Essa marcação é feita com a simples implementação da interface “java.io.Serializable”. Essa interface não possui métodos para se implementar e nem atributos. Ainda não existiam as anotações no Java 1.1, versão que trouxe pela primeira vez a serialização ao Java, e desta maneira a implementação da interface era a melhor maneira de marcar uma classe.

A herança entre classes é naturalmente afetada pela serialização. Já que quando uma classe implementa a interface “Serializable”, toda classe que dela estender também estará implicitamente marcada como serializável

.

Para exemplo, será criada uma classe “pojo.Fruta” que contém dois atributos: nome e cor. No código exemplo (Listagem 1), pode-se notar a classe “Fruta” marcada para a serialização:

Listagem 1 - Marcando uma classe como serializável.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | import java.io.Serializable;    public class Fruta implements Serializable{      private String nome;      private String cor;        public Fruta(String nome, String cor) {          this.nome = nome;          this.cor = cor;      }        public String getCor() { return cor; }      public String getNome() { return nome; }      public void setCor(String cor) { this.cor = cor; }      public void setNome(String nome) { this.nome = nome; }  } |

Pode-se dizer que a implementação do processo de serialização e deserialização simples de um objeto, não possui um código difícil de se escrever. Isto porquê duas classes fazem a maior parte do trabalho, de maneira que com poucas linhas de código pode-se transformar um objeto em bytes e vice-versa. Essas duas classes são “java.io.ObjectInputStream” que recebe bytes para criar um objeto e “java.io.ObjectOutputStream” que cria uma cadeia de bytes a partir de um objeto.

**3. SERIALIZANDO E DESERIALIZANDO O ESTADO DE UM OBJETO**

Como exemplo, será persistido um objeto em disco. Para isto, além das classes “java.io.ObjectInputStream” e “java.io.ObjectOutputStream”, serão utilizadas duas outras que trabalham com arquivos, que são “java.io.FileInputStream” e “java.io.FileOutputStream”, que seguem a mesma lógica de InputStream e OutputStream, só que agora relacionado ao sistema de arquivos e não aos objetos.

Para serialização, deve-se criar um novo “java.io.FileOutputStream”, com o caminho (“path”) do arquivo onde se deseja armazenar o estado do objeto. Também deve ser criado um novo “java.io.ObjectOutputStream” que está ligado ao arquivo. E por último deve-se executar a escrita do objeto alvo, para dentro do objeto de stream que por sua vez está ligado ao arquivo destino. Após executados esses passos, é possível encontrar no sistema de arquivos um novo arquivo que contém os bytes correspondentes à descrição do estado do objeto. Na Listagem 2, encontra-se o código exemplo.

Listagem 2 - Serialização de um objeto.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | import java.io.FileOutputStream;  import java.io.ObjectOutputStream;    public class Serializador {        public Serializador() {    }        public void serializar(String path, Object obj) throws Exception {              FileOutputStream outFile = new FileOutputStream(path);      ObjectOutputStream s = new ObjectOutputStream(outFile);      s.writeObject(obj);      s.close();      }  } |

E para a deserialização, o processo é extremamente parecido com a serialização. Cria-se um novo “java.io.FileInputStream” relacionado ao arquivo onde o estado do objeto encontra-se persistido. Deve-se criar também um novo “java.io.ObjectInputStream” que estará vinculado ao arquivo de origem. Por último, executa-se a leitura do arquivo para a restauração do estado. Na Listagem 3 encontra-se o exemplo do código.

Listagem 3 - Deserializaçãode um objeto.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | import java.io.FileInputStream;  import java.io.ObjectInputStream;    public class Deserializador {        public Deserializador() {    }        public Object deserializar(String path) throws Exception {      FileInputStream inFile = new FileInputStream(path);      ObjectInputStream d = new ObjectInputStream(inFile);      Object o = d.readObject();      d.close();      return o;      }  } |

A Listagem 4, contém o exemplo da execução da serialização e em seguida da deserialização.

Listagem 4 - Serializando e Deserializando

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | import operacoes.Deserializador;  import operacoes.Serializador;  import pojo.Fruta;    public class Principal {      public static void main(String args[]){         //serializa         Serializador s = new Serializador();         Fruta fruta = new Fruta("Maça", "Vermelha");          try {              s.serializar("/home/gustavo/fruta", fruta);          } catch (Exception ex) {              System.err.println("Falha ao serializar! - " + ex.toString());          }            //deserializa          Deserializador d = new Deserializador();          fruta = null;          try {              fruta = (Fruta) d.deserializar("/home/gustavo/fruta");          } catch (Exception ex) {              System.err.println("Falha ao deserializar! - " + ex.toString());          }          System.out.println(fruta.getNome() + " - " + fruta.getCor());      }  } |

Na Listagem 5, encontra-se um exemplo de serialização e deserialização de um ArrayList da classe Fruta. É apenas mais um exemplo que demonstra a capacidade de se persistir qualquer objeto que seja serializável. Essa serialização só foi possível porque a classe “ArrayList” implementa a interface “Serializable” por padrão.

Listagem 5 - Serialização de ArrayList.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | //serialização e deserialização com ArrayList          ArrayList<Fruta> frutas = new ArrayList<Fruta>();          Fruta f1 = new Fruta("Laranja", "Amarela");          Fruta f2 = new Fruta("Abacate", "Verde");          Fruta f3 = new Fruta("Morango", "Vermelho");          frutas.add(f1);          frutas.add(f2);          frutas.add(f3);            try {              s.serializar("/home/gustavo/frutas", frutas);              frutas = null;              frutas = (ArrayList<Fruta>) d.deserializar("/home/gustavo/frutas");              for(Fruta f : frutas){                  System.out.println("ArrayList: " + f.getNome() + " - " + f.getCor());              }          } catch (Exception ex) {              System.err.println("Falha ao serializar ou deserializar! - " +                                 ex.toString());          } |

**4. PARTICULARIDADES DA SERIALIZAÇÃO**

Como dito, a serialização de um objeto em Java pode ser algo fácil de se fazer. Mas existem certas informações que são importantes e sabê-las pode evitar vários problemas.

Um dos problemas é que nem tudo pode ser serializado, como objetos das classes “Thread” e “Socket” e atributos estáticos. Os atributos estáticos estão vinculados à classe e não ao objeto. Assim quando se trata do estado do objeto, os campos estáticos não são considerados. Em relação às Threads, suas instâncias só são úteis enquanto estiverem em execução na JVM, e podem ser paradas/destruídas e iniciadas com instruções de sua classe. Não existe algo no estado destas instâncias que seja útil para se persistir, e por isso não há sentido em persistir tais instâncias para restaurar posteriormente.

Mas e se existir alguma classe em que seja necessário persisti-la, e esta possui uma instância da classe “Thread”? Neste caso, utilizando um recurso pode-se persistir a classe inteira, menos a instância thread. Este recurso, refere-se em indicar que tal instância thread é um objeto “Transient”. Na Listagem 6 encontra-se um exemplo da implementação com uso de “Transient”.

Listagem 6 - Uso do Transient

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | import java.io.Serializable;    public class ImplementacaoTransient implements Serializable, Runnable{        transient private Thread thread;      //marcado com "transient" para que sua instância não seja serializada        //Demais valores de estado serão serializados      private int valorParaPersistir;        public ImplementacaoTransient(int valorParaPersistir){          this.valorParaPersistir = valorParaPersistir;          thread = new Thread(this);          thread.start();      }        public void run()      {          while(true){              //código a ser executado pela Thread          }      }  } |

Vale ressaltar que, o que é serializado é apenas o estado do objeto, e não os métodos e a classe. No exemplo acima, apenas a instância do objeto “thread” e do objeto “valorParaPersistir” compunham o estado do objeto. Assim, evita-se que o sistema de serialização tente serializar um objeto não-serializável e gere erros. Após a deserialização de um objeto da classe “ImplementacaoTransient” este estará totalmente funcional.

Um outro problema que pode vir a ocorrer, se refere ao versionamento das classes. Toda classe possui um atributo oculto chamado “serialVersionUID” com um valor gerado automaticamente. Objetos criados levam consigo o “serialVersionUID” de sua classe, que indica sua versão. Toda vez que uma classe sofre alterações em seu código, um novo “serialVersionUID” é gerado. O problema com a serialização surge quando serializa-se o estado de um objeto e posteriormente tenta-se restaurar o estado para a aplicação, mas houveram mudanças no código da classe. Neste caso, uma exceção “java.io.InvalidClassException” é disparada, pois de fato, o estado do objeto corresponde a uma versão de classe do qual foi criado que já não existe mais. Portanto, isso só ocorre para evitar erros de programação, em que tenta-se restaurar o estado de um objeto para uma classe que possui versão diferente da qual o objeto foi serializado inicialmente. E isso pode significar diferentes versões de uma mesma classe “Fruta” por exemplo, ou diferentes versões por classes serem totalmente difere

|  |  |
| --- | --- |
| http://www.linhadecodigo.com.br/artigos/img_artigos/EvertonCoimbraAraujo/Serializazao_Objetos/imagem001.jpg | Neste artigo apresento o Gustavo Rafael Valiati, que pode ser contatado pelo e-mail gustavovaliati@gmail.com. O Gustavo foi meu aluno em algumas disciplinas de sua gradução e está concluíndo o último período do curso de Tecnologia em Análise e Desenvolvimento de Sistemas da UTFPR campus Medianeira, neste primeiro semestre de 2012. |

**1. INTRODUÇÃO**

Nem sempre se deseja ter objetos ou estrutura de dados de um aplicativo, com um ciclo de vida que dura no máximo o tempo em que for armazenado em memória RAM. Ou seja, algumas vezes deseja-se guardá-los por um tempo indeterminado, sendo possível recuperá-los posteriormente. E neste caso, o termo “persistência” tem seu conceito definido.

Persistir dados, requer que estes possam ser serializáveis. A serialização possibilitará capturar o estado do objeto ou a estrutura de dados, e transformar em uma cadeia de bytes quando for necessário. Torna-se possível também, recuperar os bytes persistidos para fazer o processo inverso, e ter os dados de volta para a aplicação em execução.

Desta maneira, qualquer forma de dados que esteja sendo trabalhada na aplicação, e esta puder ser serializada, poderá ser passada para o meio físico, e ter o processo revertido.

A serialização não se limita apenas em possibilitar a gravação do dado em disco, mas permite que o mesmo seja transmitido por rede. Isto porquê a serialização cria um stream de bytes, que é requisito para transmissão de dados por rede.

Diversas linguagens possuem suporte à serialização, como: Java, C, C++, Python, PHP, .NET, entre outras. Neste artigo, será tratado apenas da serialização com Java, em sua implemetação básica, alguns problemas e soluções, com códigos para exemplificação.

**2. SERIALIZAÇÃO COM JAVA**

Serializar um objeto, em Java, só é possível caso sua classe esteja marcada como serializável. Essa marcação é feita com a simples implementação da interface “java.io.Serializable”. Essa interface não possui métodos para se implementar e nem atributos. Ainda não existiam as anotações no Java 1.1, versão que trouxe pela primeira vez a serialização ao Java, e desta maneira a implementação da interface era a melhor maneira de marcar uma classe.

A herança entre classes é naturalmente afetada pela serialização. Já que quando uma classe implementa a interface “Serializable”, toda classe que dela estender também estará implicitamente marcada como serializável

.

Para exemplo, será criada uma classe “pojo.Fruta” que contém dois atributos: nome e cor. No código exemplo (Listagem 1), pode-se notar a classe “Fruta” marcada para a serialização:

Listagem 1 - Marcando uma classe como serializável.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | import java.io.Serializable;    public class Fruta implements Serializable{      private String nome;      private String cor;        public Fruta(String nome, String cor) {          this.nome = nome;          this.cor = cor;      }        public String getCor() { return cor; }      public String getNome() { return nome; }      public void setCor(String cor) { this.cor = cor; }      public void setNome(String nome) { this.nome = nome; }  } |

Pode-se dizer que a implementação do processo de serialização e deserialização simples de um objeto, não possui um código difícil de se escrever. Isto porquê duas classes fazem a maior parte do trabalho, de maneira que com poucas linhas de código pode-se transformar um objeto em bytes e vice-versa. Essas duas classes são “java.io.ObjectInputStream” que recebe bytes para criar um objeto e “java.io.ObjectOutputStream” que cria uma cadeia de bytes a partir de um objeto.

**3. SERIALIZANDO E DESERIALIZANDO O ESTADO DE UM OBJETO**

Como exemplo, será persistido um objeto em disco. Para isto, além das classes “java.io.ObjectInputStream” e “java.io.ObjectOutputStream”, serão utilizadas duas outras que trabalham com arquivos, que são “java.io.FileInputStream” e “java.io.FileOutputStream”, que seguem a mesma lógica de InputStream e OutputStream, só que agora relacionado ao sistema de arquivos e não aos objetos.

Para serialização, deve-se criar um novo “java.io.FileOutputStream”, com o caminho (“path”) do arquivo onde se deseja armazenar o estado do objeto. Também deve ser criado um novo “java.io.ObjectOutputStream” que está ligado ao arquivo. E por último deve-se executar a escrita do objeto alvo, para dentro do objeto de stream que por sua vez está ligado ao arquivo destino. Após executados esses passos, é possível encontrar no sistema de arquivos um novo arquivo que contém os bytes correspondentes à descrição do estado do objeto. Na Listagem 2, encontra-se o código exemplo.

Listagem 2 - Serialização de um objeto.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | import java.io.FileOutputStream;  import java.io.ObjectOutputStream;    public class Serializador {        public Serializador() {    }        public void serializar(String path, Object obj) throws Exception {              FileOutputStream outFile = new FileOutputStream(path);      ObjectOutputStream s = new ObjectOutputStream(outFile);      s.writeObject(obj);      s.close();      }  } |

E para a deserialização, o processo é extremamente parecido com a serialização. Cria-se um novo “java.io.FileInputStream” relacionado ao arquivo onde o estado do objeto encontra-se persistido. Deve-se criar também um novo “java.io.ObjectInputStream” que estará vinculado ao arquivo de origem. Por último, executa-se a leitura do arquivo para a restauração do estado. Na Listagem 3 encontra-se o exemplo do código.

Listagem 3 - Deserializaçãode um objeto.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | import java.io.FileInputStream;  import java.io.ObjectInputStream;    public class Deserializador {        public Deserializador() {    }        public Object deserializar(String path) throws Exception {      FileInputStream inFile = new FileInputStream(path);      ObjectInputStream d = new ObjectInputStream(inFile);      Object o = d.readObject();      d.close();      return o;      }  } |

A Listagem 4, contém o exemplo da execução da serialização e em seguida da deserialização.

Listagem 4 - Serializando e Deserializando

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | import operacoes.Deserializador;  import operacoes.Serializador;  import pojo.Fruta;    public class Principal {      public static void main(String args[]){         //serializa         Serializador s = new Serializador();         Fruta fruta = new Fruta("Maça", "Vermelha");          try {              s.serializar("/home/gustavo/fruta", fruta);          } catch (Exception ex) {              System.err.println("Falha ao serializar! - " + ex.toString());          }            //deserializa          Deserializador d = new Deserializador();          fruta = null;          try {              fruta = (Fruta) d.deserializar("/home/gustavo/fruta");          } catch (Exception ex) {              System.err.println("Falha ao deserializar! - " + ex.toString());          }          System.out.println(fruta.getNome() + " - " + fruta.getCor());      }  } |

Na Listagem 5, encontra-se um exemplo de serialização e deserialização de um ArrayList da classe Fruta. É apenas mais um exemplo que demonstra a capacidade de se persistir qualquer objeto que seja serializável. Essa serialização só foi possível porque a classe “ArrayList” implementa a interface “Serializable” por padrão.

Listagem 5 - Serialização de ArrayList.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | //serialização e deserialização com ArrayList          ArrayList<Fruta> frutas = new ArrayList<Fruta>();          Fruta f1 = new Fruta("Laranja", "Amarela");          Fruta f2 = new Fruta("Abacate", "Verde");          Fruta f3 = new Fruta("Morango", "Vermelho");          frutas.add(f1);          frutas.add(f2);          frutas.add(f3);            try {              s.serializar("/home/gustavo/frutas", frutas);              frutas = null;              frutas = (ArrayList<Fruta>) d.deserializar("/home/gustavo/frutas");              for(Fruta f : frutas){                  System.out.println("ArrayList: " + f.getNome() + " - " + f.getCor());              }          } catch (Exception ex) {              System.err.println("Falha ao serializar ou deserializar! - " +                                 ex.toString());          } |

**4. PARTICULARIDADES DA SERIALIZAÇÃO**

Como dito, a serialização de um objeto em Java pode ser algo fácil de se fazer. Mas existem certas informações que são importantes e sabê-las pode evitar vários problemas.

Um dos problemas é que nem tudo pode ser serializado, como objetos das classes “Thread” e “Socket” e atributos estáticos. Os atributos estáticos estão vinculados à classe e não ao objeto. Assim quando se trata do estado do objeto, os campos estáticos não são considerados. Em relação às Threads, suas instâncias só são úteis enquanto estiverem em execução na JVM, e podem ser paradas/destruídas e iniciadas com instruções de sua classe. Não existe algo no estado destas instâncias que seja útil para se persistir, e por isso não há sentido em persistir tais instâncias para restaurar posteriormente.

Mas e se existir alguma classe em que seja necessário persisti-la, e esta possui uma instância da classe “Thread”? Neste caso, utilizando um recurso pode-se persistir a classe inteira, menos a instância thread. Este recurso, refere-se em indicar que tal instância thread é um objeto “Transient”. Na Listagem 6 encontra-se um exemplo da implementação com uso de “Transient”.

Listagem 6 - Uso do Transient

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | import java.io.Serializable;    public class ImplementacaoTransient implements Serializable, Runnable{        transient private Thread thread;      //marcado com "transient" para que sua instância não seja serializada        //Demais valores de estado serão serializados      private int valorParaPersistir;        public ImplementacaoTransient(int valorParaPersistir){          this.valorParaPersistir = valorParaPersistir;          thread = new Thread(this);          thread.start();      }        public void run()      {          while(true){              //código a ser executado pela Thread          }      }  } |

Vale ressaltar que, o que é serializado é apenas o estado do objeto, e não os métodos e a classe. No exemplo acima, apenas a instância do objeto “thread” e do objeto “valorParaPersistir” compunham o estado do objeto. Assim, evita-se que o sistema de serialização tente serializar um objeto não-serializável e gere erros. Após a deserialização de um objeto da classe “ImplementacaoTransient” este estará totalmente funcional.

Um outro problema que pode vir a ocorrer, se refere ao versionamento das classes. Toda classe possui um atributo oculto chamado “serialVersionUID” com um valor gerado automaticamente. Objetos criados levam consigo o “serialVersionUID” de sua classe, que indica sua versão. Toda vez que uma classe sofre alterações em seu código, um novo “serialVersionUID” é gerado. O problema com a serialização surge quando serializa-se o estado de um objeto e posteriormente tenta-se restaurar o estado para a aplicação, mas houveram mudanças no código da classe. Neste caso, uma exceção “java.io.InvalidClassException” é disparada, pois de fato, o estado do objeto corresponde a uma versão de classe do qual foi criado que já não existe mais. Portanto, isso só ocorre para evitar erros de programação, em que tenta-se restaurar o estado de um objeto para uma classe que possui versão diferente da qual o objeto foi serializado inicialmente. E isso pode significar diferentes versões de uma mesma classe “Fruta” por exemplo, ou diferentes versões por classes serem totalmente difere

|  |  |
| --- | --- |
| http://www.linhadecodigo.com.br/artigos/img_artigos/EvertonCoimbraAraujo/Serializazao_Objetos/imagem001.jpg | Neste artigo apresento o Gustavo Rafael Valiati, que pode ser contatado pelo e-mail gustavovaliati@gmail.com. O Gustavo foi meu aluno em algumas disciplinas de sua gradução e está concluíndo o último período do curso de Tecnologia em Análise e Desenvolvimento de Sistemas da UTFPR campus Medianeira, neste primeiro semestre de 2012. |

**1. INTRODUÇÃO**

Nem sempre se deseja ter objetos ou estrutura de dados de um aplicativo, com um ciclo de vida que dura no máximo o tempo em que for armazenado em memória RAM. Ou seja, algumas vezes deseja-se guardá-los por um tempo indeterminado, sendo possível recuperá-los posteriormente. E neste caso, o termo “persistência” tem seu conceito definido.

Persistir dados, requer que estes possam ser serializáveis. A serialização possibilitará capturar o estado do objeto ou a estrutura de dados, e transformar em uma cadeia de bytes quando for necessário. Torna-se possível também, recuperar os bytes persistidos para fazer o processo inverso, e ter os dados de volta para a aplicação em execução.

Desta maneira, qualquer forma de dados que esteja sendo trabalhada na aplicação, e esta puder ser serializada, poderá ser passada para o meio físico, e ter o processo revertido.

A serialização não se limita apenas em possibilitar a gravação do dado em disco, mas permite que o mesmo seja transmitido por rede. Isto porquê a serialização cria um stream de bytes, que é requisito para transmissão de dados por rede.

Diversas linguagens possuem suporte à serialização, como: Java, C, C++, Python, PHP, .NET, entre outras. Neste artigo, será tratado apenas da serialização com Java, em sua implemetação básica, alguns problemas e soluções, com códigos para exemplificação.

**2. SERIALIZAÇÃO COM JAVA**

Serializar um objeto, em Java, só é possível caso sua classe esteja marcada como serializável. Essa marcação é feita com a simples implementação da interface “java.io.Serializable”. Essa interface não possui métodos para se implementar e nem atributos. Ainda não existiam as anotações no Java 1.1, versão que trouxe pela primeira vez a serialização ao Java, e desta maneira a implementação da interface era a melhor maneira de marcar uma classe.

A herança entre classes é naturalmente afetada pela serialização. Já que quando uma classe implementa a interface “Serializable”, toda classe que dela estender também estará implicitamente marcada como serializável

.

Para exemplo, será criada uma classe “pojo.Fruta” que contém dois atributos: nome e cor. No código exemplo (Listagem 1), pode-se notar a classe “Fruta” marcada para a serialização:

Listagem 1 - Marcando uma classe como serializável.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | import java.io.Serializable;    public class Fruta implements Serializable{      private String nome;      private String cor;        public Fruta(String nome, String cor) {          this.nome = nome;          this.cor = cor;      }        public String getCor() { return cor; }      public String getNome() { return nome; }      public void setCor(String cor) { this.cor = cor; }      public void setNome(String nome) { this.nome = nome; }  } |

Pode-se dizer que a implementação do processo de serialização e deserialização simples de um objeto, não possui um código difícil de se escrever. Isto porquê duas classes fazem a maior parte do trabalho, de maneira que com poucas linhas de código pode-se transformar um objeto em bytes e vice-versa. Essas duas classes são “java.io.ObjectInputStream” que recebe bytes para criar um objeto e “java.io.ObjectOutputStream” que cria uma cadeia de bytes a partir de um objeto.

**3. SERIALIZANDO E DESERIALIZANDO O ESTADO DE UM OBJETO**

Como exemplo, será persistido um objeto em disco. Para isto, além das classes “java.io.ObjectInputStream” e “java.io.ObjectOutputStream”, serão utilizadas duas outras que trabalham com arquivos, que são “java.io.FileInputStream” e “java.io.FileOutputStream”, que seguem a mesma lógica de InputStream e OutputStream, só que agora relacionado ao sistema de arquivos e não aos objetos.

Para serialização, deve-se criar um novo “java.io.FileOutputStream”, com o caminho (“path”) do arquivo onde se deseja armazenar o estado do objeto. Também deve ser criado um novo “java.io.ObjectOutputStream” que está ligado ao arquivo. E por último deve-se executar a escrita do objeto alvo, para dentro do objeto de stream que por sua vez está ligado ao arquivo destino. Após executados esses passos, é possível encontrar no sistema de arquivos um novo arquivo que contém os bytes correspondentes à descrição do estado do objeto. Na Listagem 2, encontra-se o código exemplo.

Listagem 2 - Serialização de um objeto.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | import java.io.FileOutputStream;  import java.io.ObjectOutputStream;    public class Serializador {        public Serializador() {    }        public void serializar(String path, Object obj) throws Exception {              FileOutputStream outFile = new FileOutputStream(path);      ObjectOutputStream s = new ObjectOutputStream(outFile);      s.writeObject(obj);      s.close();      }  } |

E para a deserialização, o processo é extremamente parecido com a serialização. Cria-se um novo “java.io.FileInputStream” relacionado ao arquivo onde o estado do objeto encontra-se persistido. Deve-se criar também um novo “java.io.ObjectInputStream” que estará vinculado ao arquivo de origem. Por último, executa-se a leitura do arquivo para a restauração do estado. Na Listagem 3 encontra-se o exemplo do código.

Listagem 3 - Deserializaçãode um objeto.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | import java.io.FileInputStream;  import java.io.ObjectInputStream;    public class Deserializador {        public Deserializador() {    }        public Object deserializar(String path) throws Exception {      FileInputStream inFile = new FileInputStream(path);      ObjectInputStream d = new ObjectInputStream(inFile);      Object o = d.readObject();      d.close();      return o;      }  } |

A Listagem 4, contém o exemplo da execução da serialização e em seguida da deserialização.

Listagem 4 - Serializando e Deserializando

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | import operacoes.Deserializador;  import operacoes.Serializador;  import pojo.Fruta;    public class Principal {      public static void main(String args[]){         //serializa         Serializador s = new Serializador();         Fruta fruta = new Fruta("Maça", "Vermelha");          try {              s.serializar("/home/gustavo/fruta", fruta);          } catch (Exception ex) {              System.err.println("Falha ao serializar! - " + ex.toString());          }            //deserializa          Deserializador d = new Deserializador();          fruta = null;          try {              fruta = (Fruta) d.deserializar("/home/gustavo/fruta");          } catch (Exception ex) {              System.err.println("Falha ao deserializar! - " + ex.toString());          }          System.out.println(fruta.getNome() + " - " + fruta.getCor());      }  } |

Na Listagem 5, encontra-se um exemplo de serialização e deserialização de um ArrayList da classe Fruta. É apenas mais um exemplo que demonstra a capacidade de se persistir qualquer objeto que seja serializável. Essa serialização só foi possível porque a classe “ArrayList” implementa a interface “Serializable” por padrão.

Listagem 5 - Serialização de ArrayList.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | //serialização e deserialização com ArrayList          ArrayList<Fruta> frutas = new ArrayList<Fruta>();          Fruta f1 = new Fruta("Laranja", "Amarela");          Fruta f2 = new Fruta("Abacate", "Verde");          Fruta f3 = new Fruta("Morango", "Vermelho");          frutas.add(f1);          frutas.add(f2);          frutas.add(f3);            try {              s.serializar("/home/gustavo/frutas", frutas);              frutas = null;              frutas = (ArrayList<Fruta>) d.deserializar("/home/gustavo/frutas");              for(Fruta f : frutas){                  System.out.println("ArrayList: " + f.getNome() + " - " + f.getCor());              }          } catch (Exception ex) {              System.err.println("Falha ao serializar ou deserializar! - " +                                 ex.toString());          } |

**4. PARTICULARIDADES DA SERIALIZAÇÃO**

Como dito, a serialização de um objeto em Java pode ser algo fácil de se fazer. Mas existem certas informações que são importantes e sabê-las pode evitar vários problemas.

Um dos problemas é que nem tudo pode ser serializado, como objetos das classes “Thread” e “Socket” e atributos estáticos. Os atributos estáticos estão vinculados à classe e não ao objeto. Assim quando se trata do estado do objeto, os campos estáticos não são considerados. Em relação às Threads, suas instâncias só são úteis enquanto estiverem em execução na JVM, e podem ser paradas/destruídas e iniciadas com instruções de sua classe. Não existe algo no estado destas instâncias que seja útil para se persistir, e por isso não há sentido em persistir tais instâncias para restaurar posteriormente.

Mas e se existir alguma classe em que seja necessário persisti-la, e esta possui uma instância da classe “Thread”? Neste caso, utilizando um recurso pode-se persistir a classe inteira, menos a instância thread. Este recurso, refere-se em indicar que tal instância thread é um objeto “Transient”. Na Listagem 6 encontra-se um exemplo da implementação com uso de “Transient”.

Listagem 6 - Uso do Transient

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | import java.io.Serializable;    public class ImplementacaoTransient implements Serializable, Runnable{        transient private Thread thread;      //marcado com "transient" para que sua instância não seja serializada        //Demais valores de estado serão serializados      private int valorParaPersistir;        public ImplementacaoTransient(int valorParaPersistir){          this.valorParaPersistir = valorParaPersistir;          thread = new Thread(this);          thread.start();      }        public void run()      {          while(true){              //código a ser executado pela Thread          }      }  } |

Vale ressaltar que, o que é serializado é apenas o estado do objeto, e não os métodos e a classe. No exemplo acima, apenas a instância do objeto “thread” e do objeto “valorParaPersistir” compunham o estado do objeto. Assim, evita-se que o sistema de serialização tente serializar um objeto não-serializável e gere erros. Após a deserialização de um objeto da classe “ImplementacaoTransient” este estará totalmente funcional.

Um outro problema que pode vir a ocorrer, se refere ao versionamento das classes. Toda classe possui um atributo oculto chamado “serialVersionUID” com um valor gerado automaticamente. Objetos criados levam consigo o “serialVersionUID” de sua classe, que indica sua versão. Toda vez que uma classe sofre alterações em seu código, um novo “serialVersionUID” é gerado. O problema com a serialização surge quando serializa-se o estado de um objeto e posteriormente tenta-se restaurar o estado para a aplicação, mas houveram mudanças no código da classe. Neste caso, uma exceção “java.io.InvalidClassException” é disparada, pois de fato, o estado do objeto corresponde a uma versão de classe do qual foi criado que já não existe mais. Portanto, isso só ocorre para evitar erros de programação, em que tenta-se restaurar o estado de um objeto para uma classe que possui versão diferente da qual o objeto foi serializado inicialmente. E isso pode significar diferentes versões de uma mesma classe “Fruta” por exemplo, ou diferentes versões por classes serem totalmente difere

|  |  |
| --- | --- |
| http://www.linhadecodigo.com.br/artigos/img_artigos/EvertonCoimbraAraujo/Serializazao_Objetos/imagem001.jpg | Neste artigo apresento o Gustavo Rafael Valiati, que pode ser contatado pelo e-mail gustavovaliati@gmail.com. O Gustavo foi meu aluno em algumas disciplinas de sua gradução e está concluíndo o último período do curso de Tecnologia em Análise e Desenvolvimento de Sistemas da UTFPR campus Medianeira, neste primeiro semestre de 2012. |

**1. INTRODUÇÃO**

Nem sempre se deseja ter objetos ou estrutura de dados de um aplicativo, com um ciclo de vida que dura no máximo o tempo em que for armazenado em memória RAM. Ou seja, algumas vezes deseja-se guardá-los por um tempo indeterminado, sendo possível recuperá-los posteriormente. E neste caso, o termo “persistência” tem seu conceito definido.

Persistir dados, requer que estes possam ser serializáveis. A serialização possibilitará capturar o estado do objeto ou a estrutura de dados, e transformar em uma cadeia de bytes quando for necessário. Torna-se possível também, recuperar os bytes persistidos para fazer o processo inverso, e ter os dados de volta para a aplicação em execução.

Desta maneira, qualquer forma de dados que esteja sendo trabalhada na aplicação, e esta puder ser serializada, poderá ser passada para o meio físico, e ter o processo revertido.

A serialização não se limita apenas em possibilitar a gravação do dado em disco, mas permite que o mesmo seja transmitido por rede. Isto porquê a serialização cria um stream de bytes, que é requisito para transmissão de dados por rede.

Diversas linguagens possuem suporte à serialização, como: Java, C, C++, Python, PHP, .NET, entre outras. Neste artigo, será tratado apenas da serialização com Java, em sua implemetação básica, alguns problemas e soluções, com códigos para exemplificação.

**2. SERIALIZAÇÃO COM JAVA**

Serializar um objeto, em Java, só é possível caso sua classe esteja marcada como serializável. Essa marcação é feita com a simples implementação da interface “java.io.Serializable”. Essa interface não possui métodos para se implementar e nem atributos. Ainda não existiam as anotações no Java 1.1, versão que trouxe pela primeira vez a serialização ao Java, e desta maneira a implementação da interface era a melhor maneira de marcar uma classe.

A herança entre classes é naturalmente afetada pela serialização. Já que quando uma classe implementa a interface “Serializable”, toda classe que dela estender também estará implicitamente marcada como serializável

.

Para exemplo, será criada uma classe “pojo.Fruta” que contém dois atributos: nome e cor. No código exemplo (Listagem 1), pode-se notar a classe “Fruta” marcada para a serialização:

Listagem 1 - Marcando uma classe como serializável.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | import java.io.Serializable;    public class Fruta implements Serializable{      private String nome;      private String cor;        public Fruta(String nome, String cor) {          this.nome = nome;          this.cor = cor;      }        public String getCor() { return cor; }      public String getNome() { return nome; }      public void setCor(String cor) { this.cor = cor; }      public void setNome(String nome) { this.nome = nome; }  } |

Pode-se dizer que a implementação do processo de serialização e deserialização simples de um objeto, não possui um código difícil de se escrever. Isto porquê duas classes fazem a maior parte do trabalho, de maneira que com poucas linhas de código pode-se transformar um objeto em bytes e vice-versa. Essas duas classes são “java.io.ObjectInputStream” que recebe bytes para criar um objeto e “java.io.ObjectOutputStream” que cria uma cadeia de bytes a partir de um objeto.

**3. SERIALIZANDO E DESERIALIZANDO O ESTADO DE UM OBJETO**

Como exemplo, será persistido um objeto em disco. Para isto, além das classes “java.io.ObjectInputStream” e “java.io.ObjectOutputStream”, serão utilizadas duas outras que trabalham com arquivos, que são “java.io.FileInputStream” e “java.io.FileOutputStream”, que seguem a mesma lógica de InputStream e OutputStream, só que agora relacionado ao sistema de arquivos e não aos objetos.

Para serialização, deve-se criar um novo “java.io.FileOutputStream”, com o caminho (“path”) do arquivo onde se deseja armazenar o estado do objeto. Também deve ser criado um novo “java.io.ObjectOutputStream” que está ligado ao arquivo. E por último deve-se executar a escrita do objeto alvo, para dentro do objeto de stream que por sua vez está ligado ao arquivo destino. Após executados esses passos, é possível encontrar no sistema de arquivos um novo arquivo que contém os bytes correspondentes à descrição do estado do objeto. Na Listagem 2, encontra-se o código exemplo.

Listagem 2 - Serialização de um objeto.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | import java.io.FileOutputStream;  import java.io.ObjectOutputStream;    public class Serializador {        public Serializador() {    }        public void serializar(String path, Object obj) throws Exception {              FileOutputStream outFile = new FileOutputStream(path);      ObjectOutputStream s = new ObjectOutputStream(outFile);      s.writeObject(obj);      s.close();      }  } |

E para a deserialização, o processo é extremamente parecido com a serialização. Cria-se um novo “java.io.FileInputStream” relacionado ao arquivo onde o estado do objeto encontra-se persistido. Deve-se criar também um novo “java.io.ObjectInputStream” que estará vinculado ao arquivo de origem. Por último, executa-se a leitura do arquivo para a restauração do estado. Na Listagem 3 encontra-se o exemplo do código.

Listagem 3 - Deserializaçãode um objeto.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | import java.io.FileInputStream;  import java.io.ObjectInputStream;    public class Deserializador {        public Deserializador() {    }        public Object deserializar(String path) throws Exception {      FileInputStream inFile = new FileInputStream(path);      ObjectInputStream d = new ObjectInputStream(inFile);      Object o = d.readObject();      d.close();      return o;      }  } |

A Listagem 4, contém o exemplo da execução da serialização e em seguida da deserialização.

Listagem 4 - Serializando e Deserializando

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | import operacoes.Deserializador;  import operacoes.Serializador;  import pojo.Fruta;    public class Principal {      public static void main(String args[]){         //serializa         Serializador s = new Serializador();         Fruta fruta = new Fruta("Maça", "Vermelha");          try {              s.serializar("/home/gustavo/fruta", fruta);          } catch (Exception ex) {              System.err.println("Falha ao serializar! - " + ex.toString());          }            //deserializa          Deserializador d = new Deserializador();          fruta = null;          try {              fruta = (Fruta) d.deserializar("/home/gustavo/fruta");          } catch (Exception ex) {              System.err.println("Falha ao deserializar! - " + ex.toString());          }          System.out.println(fruta.getNome() + " - " + fruta.getCor());      }  } |

Na Listagem 5, encontra-se um exemplo de serialização e deserialização de um ArrayList da classe Fruta. É apenas mais um exemplo que demonstra a capacidade de se persistir qualquer objeto que seja serializável. Essa serialização só foi possível porque a classe “ArrayList” implementa a interface “Serializable” por padrão.

Listagem 5 - Serialização de ArrayList.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | //serialização e deserialização com ArrayList          ArrayList<Fruta> frutas = new ArrayList<Fruta>();          Fruta f1 = new Fruta("Laranja", "Amarela");          Fruta f2 = new Fruta("Abacate", "Verde");          Fruta f3 = new Fruta("Morango", "Vermelho");          frutas.add(f1);          frutas.add(f2);          frutas.add(f3);            try {              s.serializar("/home/gustavo/frutas", frutas);              frutas = null;              frutas = (ArrayList<Fruta>) d.deserializar("/home/gustavo/frutas");              for(Fruta f : frutas){                  System.out.println("ArrayList: " + f.getNome() + " - " + f.getCor());              }          } catch (Exception ex) {              System.err.println("Falha ao serializar ou deserializar! - " +                                 ex.toString());          } |

**4. PARTICULARIDADES DA SERIALIZAÇÃO**

Como dito, a serialização de um objeto em Java pode ser algo fácil de se fazer. Mas existem certas informações que são importantes e sabê-las pode evitar vários problemas.

Um dos problemas é que nem tudo pode ser serializado, como objetos das classes “Thread” e “Socket” e atributos estáticos. Os atributos estáticos estão vinculados à classe e não ao objeto. Assim quando se trata do estado do objeto, os campos estáticos não são considerados. Em relação às Threads, suas instâncias só são úteis enquanto estiverem em execução na JVM, e podem ser paradas/destruídas e iniciadas com instruções de sua classe. Não existe algo no estado destas instâncias que seja útil para se persistir, e por isso não há sentido em persistir tais instâncias para restaurar posteriormente.

Mas e se existir alguma classe em que seja necessário persisti-la, e esta possui uma instância da classe “Thread”? Neste caso, utilizando um recurso pode-se persistir a classe inteira, menos a instância thread. Este recurso, refere-se em indicar que tal instância thread é um objeto “Transient”. Na Listagem 6 encontra-se um exemplo da implementação com uso de “Transient”.

Listagem 6 - Uso do Transient

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | import java.io.Serializable;    public class ImplementacaoTransient implements Serializable, Runnable{        transient private Thread thread;      //marcado com "transient" para que sua instância não seja serializada        //Demais valores de estado serão serializados      private int valorParaPersistir;        public ImplementacaoTransient(int valorParaPersistir){          this.valorParaPersistir = valorParaPersistir;          thread = new Thread(this);          thread.start();      }        public void run()      {          while(true){              //código a ser executado pela Thread          }      }  } |

Vale ressaltar que, o que é serializado é apenas o estado do objeto, e não os métodos e a classe. No exemplo acima, apenas a instância do objeto “thread” e do objeto “valorParaPersistir” compunham o estado do objeto. Assim, evita-se que o sistema de serialização tente serializar um objeto não-serializável e gere erros. Após a deserialização de um objeto da classe “ImplementacaoTransient” este estará totalmente funcional.

Um outro problema que pode vir a ocorrer, se refere ao versionamento das classes. Toda classe possui um atributo oculto chamado “serialVersionUID” com um valor gerado automaticamente. Objetos criados levam consigo o “serialVersionUID” de sua classe, que indica sua versão. Toda vez que uma classe sofre alterações em seu código, um novo “serialVersionUID” é gerado. O problema com a serialização surge quando serializa-se o estado de um objeto e posteriormente tenta-se restaurar o estado para a aplicação, mas houveram mudanças no código da classe. Neste caso, uma exceção “java.io.InvalidClassException” é disparada, pois de fato, o estado do objeto corresponde a uma versão de classe do qual foi criado que já não existe mais. Portanto, isso só ocorre para evitar erros de programação, em que tenta-se restaurar o estado de um objeto para uma classe que possui versão diferente da qual o objeto foi serializado inicialmente. E isso pode significar diferentes versões de uma mesma classe “Fruta” por exemplo, ou diferentes versões por classes serem totalmente difere

|  |  |
| --- | --- |
| http://www.linhadecodigo.com.br/artigos/img_artigos/EvertonCoimbraAraujo/Serializazao_Objetos/imagem001.jpg | Neste artigo apresento o Gustavo Rafael Valiati, que pode ser contatado pelo e-mail gustavovaliati@gmail.com. O Gustavo foi meu aluno em algumas disciplinas de sua gradução e está concluíndo o último período do curso de Tecnologia em Análise e Desenvolvimento de Sistemas da UTFPR campus Medianeira, neste primeiro semestre de 2012. |

**1. INTRODUÇÃO**

Nem sempre se deseja ter objetos ou estrutura de dados de um aplicativo, com um ciclo de vida que dura no máximo o tempo em que for armazenado em memória RAM. Ou seja, algumas vezes deseja-se guardá-los por um tempo indeterminado, sendo possível recuperá-los posteriormente. E neste caso, o termo “persistência” tem seu conceito definido.

Persistir dados, requer que estes possam ser serializáveis. A serialização possibilitará capturar o estado do objeto ou a estrutura de dados, e transformar em uma cadeia de bytes quando for necessário. Torna-se possível também, recuperar os bytes persistidos para fazer o processo inverso, e ter os dados de volta para a aplicação em execução.

Desta maneira, qualquer forma de dados que esteja sendo trabalhada na aplicação, e esta puder ser serializada, poderá ser passada para o meio físico, e ter o processo revertido.

A serialização não se limita apenas em possibilitar a gravação do dado em disco, mas permite que o mesmo seja transmitido por rede. Isto porquê a serialização cria um stream de bytes, que é requisito para transmissão de dados por rede.

Diversas linguagens possuem suporte à serialização, como: Java, C, C++, Python, PHP, .NET, entre outras. Neste artigo, será tratado apenas da serialização com Java, em sua implemetação básica, alguns problemas e soluções, com códigos para exemplificação.

**2. SERIALIZAÇÃO COM JAVA**

Serializar um objeto, em Java, só é possível caso sua classe esteja marcada como serializável. Essa marcação é feita com a simples implementação da interface “java.io.Serializable”. Essa interface não possui métodos para se implementar e nem atributos. Ainda não existiam as anotações no Java 1.1, versão que trouxe pela primeira vez a serialização ao Java, e desta maneira a implementação da interface era a melhor maneira de marcar uma classe.

A herança entre classes é naturalmente afetada pela serialização. Já que quando uma classe implementa a interface “Serializable”, toda classe que dela estender também estará implicitamente marcada como serializável

.

Para exemplo, será criada uma classe “pojo.Fruta” que contém dois atributos: nome e cor. No código exemplo (Listagem 1), pode-se notar a classe “Fruta” marcada para a serialização:

Listagem 1 - Marcando uma classe como serializável.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | import java.io.Serializable;    public class Fruta implements Serializable{      private String nome;      private String cor;        public Fruta(String nome, String cor) {          this.nome = nome;          this.cor = cor;      }        public String getCor() { return cor; }      public String getNome() { return nome; }      public void setCor(String cor) { this.cor = cor; }      public void setNome(String nome) { this.nome = nome; }  } |

Pode-se dizer que a implementação do processo de serialização e deserialização simples de um objeto, não possui um código difícil de se escrever. Isto porquê duas classes fazem a maior parte do trabalho, de maneira que com poucas linhas de código pode-se transformar um objeto em bytes e vice-versa. Essas duas classes são “java.io.ObjectInputStream” que recebe bytes para criar um objeto e “java.io.ObjectOutputStream” que cria uma cadeia de bytes a partir de um objeto.

**3. SERIALIZANDO E DESERIALIZANDO O ESTADO DE UM OBJETO**

Como exemplo, será persistido um objeto em disco. Para isto, além das classes “java.io.ObjectInputStream” e “java.io.ObjectOutputStream”, serão utilizadas duas outras que trabalham com arquivos, que são “java.io.FileInputStream” e “java.io.FileOutputStream”, que seguem a mesma lógica de InputStream e OutputStream, só que agora relacionado ao sistema de arquivos e não aos objetos.

Para serialização, deve-se criar um novo “java.io.FileOutputStream”, com o caminho (“path”) do arquivo onde se deseja armazenar o estado do objeto. Também deve ser criado um novo “java.io.ObjectOutputStream” que está ligado ao arquivo. E por último deve-se executar a escrita do objeto alvo, para dentro do objeto de stream que por sua vez está ligado ao arquivo destino. Após executados esses passos, é possível encontrar no sistema de arquivos um novo arquivo que contém os bytes correspondentes à descrição do estado do objeto. Na Listagem 2, encontra-se o código exemplo.

Listagem 2 - Serialização de um objeto.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | import java.io.FileOutputStream;  import java.io.ObjectOutputStream;    public class Serializador {        public Serializador() {    }        public void serializar(String path, Object obj) throws Exception {              FileOutputStream outFile = new FileOutputStream(path);      ObjectOutputStream s = new ObjectOutputStream(outFile);      s.writeObject(obj);      s.close();      }  } |

E para a deserialização, o processo é extremamente parecido com a serialização. Cria-se um novo “java.io.FileInputStream” relacionado ao arquivo onde o estado do objeto encontra-se persistido. Deve-se criar também um novo “java.io.ObjectInputStream” que estará vinculado ao arquivo de origem. Por último, executa-se a leitura do arquivo para a restauração do estado. Na Listagem 3 encontra-se o exemplo do código.

Listagem 3 - Deserializaçãode um objeto.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | import java.io.FileInputStream;  import java.io.ObjectInputStream;    public class Deserializador {        public Deserializador() {    }        public Object deserializar(String path) throws Exception {      FileInputStream inFile = new FileInputStream(path);      ObjectInputStream d = new ObjectInputStream(inFile);      Object o = d.readObject();      d.close();      return o;      }  } |

A Listagem 4, contém o exemplo da execução da serialização e em seguida da deserialização.

Listagem 4 - Serializando e Deserializando

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | import operacoes.Deserializador;  import operacoes.Serializador;  import pojo.Fruta;    public class Principal {      public static void main(String args[]){         //serializa         Serializador s = new Serializador();         Fruta fruta = new Fruta("Maça", "Vermelha");          try {              s.serializar("/home/gustavo/fruta", fruta);          } catch (Exception ex) {              System.err.println("Falha ao serializar! - " + ex.toString());          }            //deserializa          Deserializador d = new Deserializador();          fruta = null;          try {              fruta = (Fruta) d.deserializar("/home/gustavo/fruta");          } catch (Exception ex) {              System.err.println("Falha ao deserializar! - " + ex.toString());          }          System.out.println(fruta.getNome() + " - " + fruta.getCor());      }  } |

Na Listagem 5, encontra-se um exemplo de serialização e deserialização de um ArrayList da classe Fruta. É apenas mais um exemplo que demonstra a capacidade de se persistir qualquer objeto que seja serializável. Essa serialização só foi possível porque a classe “ArrayList” implementa a interface “Serializable” por padrão.

Listagem 5 - Serialização de ArrayList.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | //serialização e deserialização com ArrayList          ArrayList<Fruta> frutas = new ArrayList<Fruta>();          Fruta f1 = new Fruta("Laranja", "Amarela");          Fruta f2 = new Fruta("Abacate", "Verde");          Fruta f3 = new Fruta("Morango", "Vermelho");          frutas.add(f1);          frutas.add(f2);          frutas.add(f3);            try {              s.serializar("/home/gustavo/frutas", frutas);              frutas = null;              frutas = (ArrayList<Fruta>) d.deserializar("/home/gustavo/frutas");              for(Fruta f : frutas){                  System.out.println("ArrayList: " + f.getNome() + " - " + f.getCor());              }          } catch (Exception ex) {              System.err.println("Falha ao serializar ou deserializar! - " +                                 ex.toString());          } |

**4. PARTICULARIDADES DA SERIALIZAÇÃO**

Como dito, a serialização de um objeto em Java pode ser algo fácil de se fazer. Mas existem certas informações que são importantes e sabê-las pode evitar vários problemas.

Um dos problemas é que nem tudo pode ser serializado, como objetos das classes “Thread” e “Socket” e atributos estáticos. Os atributos estáticos estão vinculados à classe e não ao objeto. Assim quando se trata do estado do objeto, os campos estáticos não são considerados. Em relação às Threads, suas instâncias só são úteis enquanto estiverem em execução na JVM, e podem ser paradas/destruídas e iniciadas com instruções de sua classe. Não existe algo no estado destas instâncias que seja útil para se persistir, e por isso não há sentido em persistir tais instâncias para restaurar posteriormente.

Mas e se existir alguma classe em que seja necessário persisti-la, e esta possui uma instância da classe “Thread”? Neste caso, utilizando um recurso pode-se persistir a classe inteira, menos a instância thread. Este recurso, refere-se em indicar que tal instância thread é um objeto “Transient”. Na Listagem 6 encontra-se um exemplo da implementação com uso de “Transient”.

Listagem 6 - Uso do Transient

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | import java.io.Serializable;    public class ImplementacaoTransient implements Serializable, Runnable{        transient private Thread thread;      //marcado com "transient" para que sua instância não seja serializada        //Demais valores de estado serão serializados      private int valorParaPersistir;        public ImplementacaoTransient(int valorParaPersistir){          this.valorParaPersistir = valorParaPersistir;          thread = new Thread(this);          thread.start();      }        public void run()      {          while(true){              //código a ser executado pela Thread          }      }  } |

Vale ressaltar que, o que é serializado é apenas o estado do objeto, e não os métodos e a classe. No exemplo acima, apenas a instância do objeto “thread” e do objeto “valorParaPersistir” compunham o estado do objeto. Assim, evita-se que o sistema de serialização tente serializar um objeto não-serializável e gere erros. Após a deserialização de um objeto da classe “ImplementacaoTransient” este estará totalmente funcional.

Um outro problema que pode vir a ocorrer, se refere ao versionamento das classes. Toda classe possui um atributo oculto chamado “serialVersionUID” com um valor gerado automaticamente. Objetos criados levam consigo o “serialVersionUID” de sua classe, que indica sua versão. Toda vez que uma classe sofre alterações em seu código, um novo “serialVersionUID” é gerado. O problema com a serialização surge quando serializa-se o estado de um objeto e posteriormente tenta-se restaurar o estado para a aplicação, mas houveram mudanças no código da classe. Neste caso, uma exceção “java.io.InvalidClassException” é disparada, pois de fato, o estado do objeto corresponde a uma versão de classe do qual foi criado que já não existe mais. Portanto, isso só ocorre para evitar erros de programação, em que tenta-se restaurar o estado de um objeto para uma classe que possui versão diferente da qual o objeto foi serializado inicialmente. E isso pode significar diferentes versões de uma mesma classe “Fruta” por exemplo, ou diferentes versões por classes serem totalmente difere

|  |  |
| --- | --- |
| http://www.linhadecodigo.com.br/artigos/img_artigos/EvertonCoimbraAraujo/Serializazao_Objetos/imagem001.jpg | Neste artigo apresento o Gustavo Rafael Valiati, que pode ser contatado pelo e-mail gustavovaliati@gmail.com. O Gustavo foi meu aluno em algumas disciplinas de sua gradução e está concluíndo o último período do curso de Tecnologia em Análise e Desenvolvimento de Sistemas da UTFPR campus Medianeira, neste primeiro semestre de 2012. |

**1. INTRODUÇÃO**

Nem sempre se deseja ter objetos ou estrutura de dados de um aplicativo, com um ciclo de vida que dura no máximo o tempo em que for armazenado em memória RAM. Ou seja, algumas vezes deseja-se guardá-los por um tempo indeterminado, sendo possível recuperá-los posteriormente. E neste caso, o termo “persistência” tem seu conceito definido.

Persistir dados, requer que estes possam ser serializáveis. A serialização possibilitará capturar o estado do objeto ou a estrutura de dados, e transformar em uma cadeia de bytes quando for necessário. Torna-se possível também, recuperar os bytes persistidos para fazer o processo inverso, e ter os dados de volta para a aplicação em execução.

Desta maneira, qualquer forma de dados que esteja sendo trabalhada na aplicação, e esta puder ser serializada, poderá ser passada para o meio físico, e ter o processo revertido.

A serialização não se limita apenas em possibilitar a gravação do dado em disco, mas permite que o mesmo seja transmitido por rede. Isto porquê a serialização cria um stream de bytes, que é requisito para transmissão de dados por rede.

Diversas linguagens possuem suporte à serialização, como: Java, C, C++, Python, PHP, .NET, entre outras. Neste artigo, será tratado apenas da serialização com Java, em sua implemetação básica, alguns problemas e soluções, com códigos para exemplificação.

**2. SERIALIZAÇÃO COM JAVA**

Serializar um objeto, em Java, só é possível caso sua classe esteja marcada como serializável. Essa marcação é feita com a simples implementação da interface “java.io.Serializable”. Essa interface não possui métodos para se implementar e nem atributos. Ainda não existiam as anotações no Java 1.1, versão que trouxe pela primeira vez a serialização ao Java, e desta maneira a implementação da interface era a melhor maneira de marcar uma classe.

A herança entre classes é naturalmente afetada pela serialização. Já que quando uma classe implementa a interface “Serializable”, toda classe que dela estender também estará implicitamente marcada como serializável

.

Para exemplo, será criada uma classe “pojo.Fruta” que contém dois atributos: nome e cor. No código exemplo (Listagem 1), pode-se notar a classe “Fruta” marcada para a serialização:

Listagem 1 - Marcando uma classe como serializável.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | import java.io.Serializable;    public class Fruta implements Serializable{      private String nome;      private String cor;        public Fruta(String nome, String cor) {          this.nome = nome;          this.cor = cor;      }        public String getCor() { return cor; }      public String getNome() { return nome; }      public void setCor(String cor) { this.cor = cor; }      public void setNome(String nome) { this.nome = nome; }  } |

Pode-se dizer que a implementação do processo de serialização e deserialização simples de um objeto, não possui um código difícil de se escrever. Isto porquê duas classes fazem a maior parte do trabalho, de maneira que com poucas linhas de código pode-se transformar um objeto em bytes e vice-versa. Essas duas classes são “java.io.ObjectInputStream” que recebe bytes para criar um objeto e “java.io.ObjectOutputStream” que cria uma cadeia de bytes a partir de um objeto.

**3. SERIALIZANDO E DESERIALIZANDO O ESTADO DE UM OBJETO**

Como exemplo, será persistido um objeto em disco. Para isto, além das classes “java.io.ObjectInputStream” e “java.io.ObjectOutputStream”, serão utilizadas duas outras que trabalham com arquivos, que são “java.io.FileInputStream” e “java.io.FileOutputStream”, que seguem a mesma lógica de InputStream e OutputStream, só que agora relacionado ao sistema de arquivos e não aos objetos.

Para serialização, deve-se criar um novo “java.io.FileOutputStream”, com o caminho (“path”) do arquivo onde se deseja armazenar o estado do objeto. Também deve ser criado um novo “java.io.ObjectOutputStream” que está ligado ao arquivo. E por último deve-se executar a escrita do objeto alvo, para dentro do objeto de stream que por sua vez está ligado ao arquivo destino. Após executados esses passos, é possível encontrar no sistema de arquivos um novo arquivo que contém os bytes correspondentes à descrição do estado do objeto. Na Listagem 2, encontra-se o código exemplo.

Listagem 2 - Serialização de um objeto.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | import java.io.FileOutputStream;  import java.io.ObjectOutputStream;    public class Serializador {        public Serializador() {    }        public void serializar(String path, Object obj) throws Exception {              FileOutputStream outFile = new FileOutputStream(path);      ObjectOutputStream s = new ObjectOutputStream(outFile);      s.writeObject(obj);      s.close();      }  } |

E para a deserialização, o processo é extremamente parecido com a serialização. Cria-se um novo “java.io.FileInputStream” relacionado ao arquivo onde o estado do objeto encontra-se persistido. Deve-se criar também um novo “java.io.ObjectInputStream” que estará vinculado ao arquivo de origem. Por último, executa-se a leitura do arquivo para a restauração do estado. Na Listagem 3 encontra-se o exemplo do código.

Listagem 3 - Deserializaçãode um objeto.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | import java.io.FileInputStream;  import java.io.ObjectInputStream;    public class Deserializador {        public Deserializador() {    }        public Object deserializar(String path) throws Exception {      FileInputStream inFile = new FileInputStream(path);      ObjectInputStream d = new ObjectInputStream(inFile);      Object o = d.readObject();      d.close();      return o;      }  } |

A Listagem 4, contém o exemplo da execução da serialização e em seguida da deserialização.

Listagem 4 - Serializando e Deserializando

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | import operacoes.Deserializador;  import operacoes.Serializador;  import pojo.Fruta;    public class Principal {      public static void main(String args[]){         //serializa         Serializador s = new Serializador();         Fruta fruta = new Fruta("Maça", "Vermelha");          try {              s.serializar("/home/gustavo/fruta", fruta);          } catch (Exception ex) {              System.err.println("Falha ao serializar! - " + ex.toString());          }            //deserializa          Deserializador d = new Deserializador();          fruta = null;          try {              fruta = (Fruta) d.deserializar("/home/gustavo/fruta");          } catch (Exception ex) {              System.err.println("Falha ao deserializar! - " + ex.toString());          }          System.out.println(fruta.getNome() + " - " + fruta.getCor());      }  } |

Na Listagem 5, encontra-se um exemplo de serialização e deserialização de um ArrayList da classe Fruta. É apenas mais um exemplo que demonstra a capacidade de se persistir qualquer objeto que seja serializável. Essa serialização só foi possível porque a classe “ArrayList” implementa a interface “Serializable” por padrão.

Listagem 5 - Serialização de ArrayList.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | //serialização e deserialização com ArrayList          ArrayList<Fruta> frutas = new ArrayList<Fruta>();          Fruta f1 = new Fruta("Laranja", "Amarela");          Fruta f2 = new Fruta("Abacate", "Verde");          Fruta f3 = new Fruta("Morango", "Vermelho");          frutas.add(f1);          frutas.add(f2);          frutas.add(f3);            try {              s.serializar("/home/gustavo/frutas", frutas);              frutas = null;              frutas = (ArrayList<Fruta>) d.deserializar("/home/gustavo/frutas");              for(Fruta f : frutas){                  System.out.println("ArrayList: " + f.getNome() + " - " + f.getCor());              }          } catch (Exception ex) {              System.err.println("Falha ao serializar ou deserializar! - " +                                 ex.toString());          } |

**4. PARTICULARIDADES DA SERIALIZAÇÃO**

Como dito, a serialização de um objeto em Java pode ser algo fácil de se fazer. Mas existem certas informações que são importantes e sabê-las pode evitar vários problemas.

Um dos problemas é que nem tudo pode ser serializado, como objetos das classes “Thread” e “Socket” e atributos estáticos. Os atributos estáticos estão vinculados à classe e não ao objeto. Assim quando se trata do estado do objeto, os campos estáticos não são considerados. Em relação às Threads, suas instâncias só são úteis enquanto estiverem em execução na JVM, e podem ser paradas/destruídas e iniciadas com instruções de sua classe. Não existe algo no estado destas instâncias que seja útil para se persistir, e por isso não há sentido em persistir tais instâncias para restaurar posteriormente.

Mas e se existir alguma classe em que seja necessário persisti-la, e esta possui uma instância da classe “Thread”? Neste caso, utilizando um recurso pode-se persistir a classe inteira, menos a instância thread. Este recurso, refere-se em indicar que tal instância thread é um objeto “Transient”. Na Listagem 6 encontra-se um exemplo da implementação com uso de “Transient”.

Listagem 6 - Uso do Transient

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | import java.io.Serializable;    public class ImplementacaoTransient implements Serializable, Runnable{        transient private Thread thread;      //marcado com "transient" para que sua instância não seja serializada        //Demais valores de estado serão serializados      private int valorParaPersistir;        public ImplementacaoTransient(int valorParaPersistir){          this.valorParaPersistir = valorParaPersistir;          thread = new Thread(this);          thread.start();      }        public void run()      {          while(true){              //código a ser executado pela Thread          }      }  } |

Vale ressaltar que, o que é serializado é apenas o estado do objeto, e não os métodos e a classe. No exemplo acima, apenas a instância do objeto “thread” e do objeto “valorParaPersistir” compunham o estado do objeto. Assim, evita-se que o sistema de serialização tente serializar um objeto não-serializável e gere erros. Após a deserialização de um objeto da classe “ImplementacaoTransient” este estará totalmente funcional.

Um outro problema que pode vir a ocorrer, se refere ao versionamento das classes. Toda classe possui um atributo oculto chamado “serialVersionUID” com um valor gerado automaticamente. Objetos criados levam consigo o “serialVersionUID” de sua classe, que indica sua versão. Toda vez que uma classe sofre alterações em seu código, um novo “serialVersionUID” é gerado. O problema com a serialização surge quando serializa-se o estado de um objeto e posteriormente tenta-se restaurar o estado para a aplicação, mas houveram mudanças no código da classe. Neste caso, uma exceção “java.io.InvalidClassException” é disparada, pois de fato, o estado do objeto corresponde a uma versão de classe do qual foi criado que já não existe mais. Portanto, isso só ocorre para evitar erros de programação, em que tenta-se restaurar o estado de um objeto para uma classe que possui versão diferente da qual o objeto foi serializado inicialmente. E isso pode significar diferentes versões de uma mesma classe “Fruta” por exemplo, ou diferentes versões por classes serem totalmente difere

|  |  |
| --- | --- |
| http://www.linhadecodigo.com.br/artigos/img_artigos/EvertonCoimbraAraujo/Serializazao_Objetos/imagem001.jpg | Neste artigo apresento o Gustavo Rafael Valiati, que pode ser contatado pelo e-mail gustavovaliati@gmail.com. O Gustavo foi meu aluno em algumas disciplinas de sua gradução e está concluíndo o último período do curso de Tecnologia em Análise e Desenvolvimento de Sistemas da UTFPR campus Medianeira, neste primeiro semestre de 2012. |

**1. INTRODUÇÃO**

Nem sempre se deseja ter objetos ou estrutura de dados de um aplicativo, com um ciclo de vida que dura no máximo o tempo em que for armazenado em memória RAM. Ou seja, algumas vezes deseja-se guardá-los por um tempo indeterminado, sendo possível recuperá-los posteriormente. E neste caso, o termo “persistência” tem seu conceito definido.

Persistir dados, requer que estes possam ser serializáveis. A serialização possibilitará capturar o estado do objeto ou a estrutura de dados, e transformar em uma cadeia de bytes quando for necessário. Torna-se possível também, recuperar os bytes persistidos para fazer o processo inverso, e ter os dados de volta para a aplicação em execução.

Desta maneira, qualquer forma de dados que esteja sendo trabalhada na aplicação, e esta puder ser serializada, poderá ser passada para o meio físico, e ter o processo revertido.

A serialização não se limita apenas em possibilitar a gravação do dado em disco, mas permite que o mesmo seja transmitido por rede. Isto porquê a serialização cria um stream de bytes, que é requisito para transmissão de dados por rede.

Diversas linguagens possuem suporte à serialização, como: Java, C, C++, Python, PHP, .NET, entre outras. Neste artigo, será tratado apenas da serialização com Java, em sua implemetação básica, alguns problemas e soluções, com códigos para exemplificação.

**2. SERIALIZAÇÃO COM JAVA**

Serializar um objeto, em Java, só é possível caso sua classe esteja marcada como serializável. Essa marcação é feita com a simples implementação da interface “java.io.Serializable”. Essa interface não possui métodos para se implementar e nem atributos. Ainda não existiam as anotações no Java 1.1, versão que trouxe pela primeira vez a serialização ao Java, e desta maneira a implementação da interface era a melhor maneira de marcar uma classe.

A herança entre classes é naturalmente afetada pela serialização. Já que quando uma classe implementa a interface “Serializable”, toda classe que dela estender também estará implicitamente marcada como serializável

.

Para exemplo, será criada uma classe “pojo.Fruta” que contém dois atributos: nome e cor. No código exemplo (Listagem 1), pode-se notar a classe “Fruta” marcada para a serialização:

Listagem 1 - Marcando uma classe como serializável.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | import java.io.Serializable;    public class Fruta implements Serializable{      private String nome;      private String cor;        public Fruta(String nome, String cor) {          this.nome = nome;          this.cor = cor;      }        public String getCor() { return cor; }      public String getNome() { return nome; }      public void setCor(String cor) { this.cor = cor; }      public void setNome(String nome) { this.nome = nome; }  } |

Pode-se dizer que a implementação do processo de serialização e deserialização simples de um objeto, não possui um código difícil de se escrever. Isto porquê duas classes fazem a maior parte do trabalho, de maneira que com poucas linhas de código pode-se transformar um objeto em bytes e vice-versa. Essas duas classes são “java.io.ObjectInputStream” que recebe bytes para criar um objeto e “java.io.ObjectOutputStream” que cria uma cadeia de bytes a partir de um objeto.

**3. SERIALIZANDO E DESERIALIZANDO O ESTADO DE UM OBJETO**

Como exemplo, será persistido um objeto em disco. Para isto, além das classes “java.io.ObjectInputStream” e “java.io.ObjectOutputStream”, serão utilizadas duas outras que trabalham com arquivos, que são “java.io.FileInputStream” e “java.io.FileOutputStream”, que seguem a mesma lógica de InputStream e OutputStream, só que agora relacionado ao sistema de arquivos e não aos objetos.

Para serialização, deve-se criar um novo “java.io.FileOutputStream”, com o caminho (“path”) do arquivo onde se deseja armazenar o estado do objeto. Também deve ser criado um novo “java.io.ObjectOutputStream” que está ligado ao arquivo. E por último deve-se executar a escrita do objeto alvo, para dentro do objeto de stream que por sua vez está ligado ao arquivo destino. Após executados esses passos, é possível encontrar no sistema de arquivos um novo arquivo que contém os bytes correspondentes à descrição do estado do objeto. Na Listagem 2, encontra-se o código exemplo.

Listagem 2 - Serialização de um objeto.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | import java.io.FileOutputStream;  import java.io.ObjectOutputStream;    public class Serializador {        public Serializador() {    }        public void serializar(String path, Object obj) throws Exception {              FileOutputStream outFile = new FileOutputStream(path);      ObjectOutputStream s = new ObjectOutputStream(outFile);      s.writeObject(obj);      s.close();      }  } |

E para a deserialização, o processo é extremamente parecido com a serialização. Cria-se um novo “java.io.FileInputStream” relacionado ao arquivo onde o estado do objeto encontra-se persistido. Deve-se criar também um novo “java.io.ObjectInputStream” que estará vinculado ao arquivo de origem. Por último, executa-se a leitura do arquivo para a restauração do estado. Na Listagem 3 encontra-se o exemplo do código.

Listagem 3 - Deserializaçãode um objeto.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | import java.io.FileInputStream;  import java.io.ObjectInputStream;    public class Deserializador {        public Deserializador() {    }        public Object deserializar(String path) throws Exception {      FileInputStream inFile = new FileInputStream(path);      ObjectInputStream d = new ObjectInputStream(inFile);      Object o = d.readObject();      d.close();      return o;      }  } |

A Listagem 4, contém o exemplo da execução da serialização e em seguida da deserialização.

Listagem 4 - Serializando e Deserializando

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | import operacoes.Deserializador;  import operacoes.Serializador;  import pojo.Fruta;    public class Principal {      public static void main(String args[]){         //serializa         Serializador s = new Serializador();         Fruta fruta = new Fruta("Maça", "Vermelha");          try {              s.serializar("/home/gustavo/fruta", fruta);          } catch (Exception ex) {              System.err.println("Falha ao serializar! - " + ex.toString());          }            //deserializa          Deserializador d = new Deserializador();          fruta = null;          try {              fruta = (Fruta) d.deserializar("/home/gustavo/fruta");          } catch (Exception ex) {              System.err.println("Falha ao deserializar! - " + ex.toString());          }          System.out.println(fruta.getNome() + " - " + fruta.getCor());      }  } |

Na Listagem 5, encontra-se um exemplo de serialização e deserialização de um ArrayList da classe Fruta. É apenas mais um exemplo que demonstra a capacidade de se persistir qualquer objeto que seja serializável. Essa serialização só foi possível porque a classe “ArrayList” implementa a interface “Serializable” por padrão.

Listagem 5 - Serialização de ArrayList.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | //serialização e deserialização com ArrayList          ArrayList<Fruta> frutas = new ArrayList<Fruta>();          Fruta f1 = new Fruta("Laranja", "Amarela");          Fruta f2 = new Fruta("Abacate", "Verde");          Fruta f3 = new Fruta("Morango", "Vermelho");          frutas.add(f1);          frutas.add(f2);          frutas.add(f3);            try {              s.serializar("/home/gustavo/frutas", frutas);              frutas = null;              frutas = (ArrayList<Fruta>) d.deserializar("/home/gustavo/frutas");              for(Fruta f : frutas){                  System.out.println("ArrayList: " + f.getNome() + " - " + f.getCor());              }          } catch (Exception ex) {              System.err.println("Falha ao serializar ou deserializar! - " +                                 ex.toString());          } |

**4. PARTICULARIDADES DA SERIALIZAÇÃO**

Como dito, a serialização de um objeto em Java pode ser algo fácil de se fazer. Mas existem certas informações que são importantes e sabê-las pode evitar vários problemas.

Um dos problemas é que nem tudo pode ser serializado, como objetos das classes “Thread” e “Socket” e atributos estáticos. Os atributos estáticos estão vinculados à classe e não ao objeto. Assim quando se trata do estado do objeto, os campos estáticos não são considerados. Em relação às Threads, suas instâncias só são úteis enquanto estiverem em execução na JVM, e podem ser paradas/destruídas e iniciadas com instruções de sua classe. Não existe algo no estado destas instâncias que seja útil para se persistir, e por isso não há sentido em persistir tais instâncias para restaurar posteriormente.

Mas e se existir alguma classe em que seja necessário persisti-la, e esta possui uma instância da classe “Thread”? Neste caso, utilizando um recurso pode-se persistir a classe inteira, menos a instância thread. Este recurso, refere-se em indicar que tal instância thread é um objeto “Transient”. Na Listagem 6 encontra-se um exemplo da implementação com uso de “Transient”.

Listagem 6 - Uso do Transient

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | import java.io.Serializable;    public class ImplementacaoTransient implements Serializable, Runnable{        transient private Thread thread;      //marcado com "transient" para que sua instância não seja serializada        //Demais valores de estado serão serializados      private int valorParaPersistir;        public ImplementacaoTransient(int valorParaPersistir){          this.valorParaPersistir = valorParaPersistir;          thread = new Thread(this);          thread.start();      }        public void run()      {          while(true){              //código a ser executado pela Thread          }      }  } |

Vale ressaltar que, o que é serializado é apenas o estado do objeto, e não os métodos e a classe. No exemplo acima, apenas a instância do objeto “thread” e do objeto “valorParaPersistir” compunham o estado do objeto. Assim, evita-se que o sistema de serialização tente serializar um objeto não-serializável e gere erros. Após a deserialização de um objeto da classe “ImplementacaoTransient” este estará totalmente funcional.

Um outro problema que pode vir a ocorrer, se refere ao versionamento das classes. Toda classe possui um atributo oculto chamado “serialVersionUID” com um valor gerado automaticamente. Objetos criados levam consigo o “serialVersionUID” de sua classe, que indica sua versão. Toda vez que uma classe sofre alterações em seu código, um novo “serialVersionUID” é gerado. O problema com a serialização surge quando serializa-se o estado de um objeto e posteriormente tenta-se restaurar o estado para a aplicação, mas houveram mudanças no código da classe. Neste caso, uma exceção “java.io.InvalidClassException” é disparada, pois de fato, o estado do objeto corresponde a uma versão de classe do qual foi criado que já não existe mais. Portanto, isso só ocorre para evitar erros de programação, em que tenta-se restaurar o estado de um objeto para uma classe que possui versão diferente da qual o objeto foi serializado inicialmente. E isso pode significar diferentes versões de uma mesma classe “Fruta” por exemplo, ou diferentes versões por classes serem totalmente difere

|  |  |
| --- | --- |
| http://www.linhadecodigo.com.br/artigos/img_artigos/EvertonCoimbraAraujo/Serializazao_Objetos/imagem001.jpg | Neste artigo apresento o Gustavo Rafael Valiati, que pode ser contatado pelo e-mail gustavovaliati@gmail.com. O Gustavo foi meu aluno em algumas disciplinas de sua gradução e está concluíndo o último período do curso de Tecnologia em Análise e Desenvolvimento de Sistemas da UTFPR campus Medianeira, neste primeiro semestre de 2012. |

**1. INTRODUÇÃO**

Nem sempre se deseja ter objetos ou estrutura de dados de um aplicativo, com um ciclo de vida que dura no máximo o tempo em que for armazenado em memória RAM. Ou seja, algumas vezes deseja-se guardá-los por um tempo indeterminado, sendo possível recuperá-los posteriormente. E neste caso, o termo “persistência” tem seu conceito definido.

Persistir dados, requer que estes possam ser serializáveis. A serialização possibilitará capturar o estado do objeto ou a estrutura de dados, e transformar em uma cadeia de bytes quando for necessário. Torna-se possível também, recuperar os bytes persistidos para fazer o processo inverso, e ter os dados de volta para a aplicação em execução.

Desta maneira, qualquer forma de dados que esteja sendo trabalhada na aplicação, e esta puder ser serializada, poderá ser passada para o meio físico, e ter o processo revertido.

A serialização não se limita apenas em possibilitar a gravação do dado em disco, mas permite que o mesmo seja transmitido por rede. Isto porquê a serialização cria um stream de bytes, que é requisito para transmissão de dados por rede.

Diversas linguagens possuem suporte à serialização, como: Java, C, C++, Python, PHP, .NET, entre outras. Neste artigo, será tratado apenas da serialização com Java, em sua implemetação básica, alguns problemas e soluções, com códigos para exemplificação.

**2. SERIALIZAÇÃO COM JAVA**

Serializar um objeto, em Java, só é possível caso sua classe esteja marcada como serializável. Essa marcação é feita com a simples implementação da interface “java.io.Serializable”. Essa interface não possui métodos para se implementar e nem atributos. Ainda não existiam as anotações no Java 1.1, versão que trouxe pela primeira vez a serialização ao Java, e desta maneira a implementação da interface era a melhor maneira de marcar uma classe.

A herança entre classes é naturalmente afetada pela serialização. Já que quando uma classe implementa a interface “Serializable”, toda classe que dela estender também estará implicitamente marcada como serializável

.

Para exemplo, será criada uma classe “pojo.Fruta” que contém dois atributos: nome e cor. No código exemplo (Listagem 1), pode-se notar a classe “Fruta” marcada para a serialização:

Listagem 1 - Marcando uma classe como serializável.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | import java.io.Serializable;    public class Fruta implements Serializable{      private String nome;      private String cor;        public Fruta(String nome, String cor) {          this.nome = nome;          this.cor = cor;      }        public String getCor() { return cor; }      public String getNome() { return nome; }      public void setCor(String cor) { this.cor = cor; }      public void setNome(String nome) { this.nome = nome; }  } |

Pode-se dizer que a implementação do processo de serialização e deserialização simples de um objeto, não possui um código difícil de se escrever. Isto porquê duas classes fazem a maior parte do trabalho, de maneira que com poucas linhas de código pode-se transformar um objeto em bytes e vice-versa. Essas duas classes são “java.io.ObjectInputStream” que recebe bytes para criar um objeto e “java.io.ObjectOutputStream” que cria uma cadeia de bytes a partir de um objeto.

**3. SERIALIZANDO E DESERIALIZANDO O ESTADO DE UM OBJETO**

Como exemplo, será persistido um objeto em disco. Para isto, além das classes “java.io.ObjectInputStream” e “java.io.ObjectOutputStream”, serão utilizadas duas outras que trabalham com arquivos, que são “java.io.FileInputStream” e “java.io.FileOutputStream”, que seguem a mesma lógica de InputStream e OutputStream, só que agora relacionado ao sistema de arquivos e não aos objetos.

Para serialização, deve-se criar um novo “java.io.FileOutputStream”, com o caminho (“path”) do arquivo onde se deseja armazenar o estado do objeto. Também deve ser criado um novo “java.io.ObjectOutputStream” que está ligado ao arquivo. E por último deve-se executar a escrita do objeto alvo, para dentro do objeto de stream que por sua vez está ligado ao arquivo destino. Após executados esses passos, é possível encontrar no sistema de arquivos um novo arquivo que contém os bytes correspondentes à descrição do estado do objeto. Na Listagem 2, encontra-se o código exemplo.

Listagem 2 - Serialização de um objeto.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | import java.io.FileOutputStream;  import java.io.ObjectOutputStream;    public class Serializador {        public Serializador() {    }        public void serializar(String path, Object obj) throws Exception {              FileOutputStream outFile = new FileOutputStream(path);      ObjectOutputStream s = new ObjectOutputStream(outFile);      s.writeObject(obj);      s.close();      }  } |

E para a deserialização, o processo é extremamente parecido com a serialização. Cria-se um novo “java.io.FileInputStream” relacionado ao arquivo onde o estado do objeto encontra-se persistido. Deve-se criar também um novo “java.io.ObjectInputStream” que estará vinculado ao arquivo de origem. Por último, executa-se a leitura do arquivo para a restauração do estado. Na Listagem 3 encontra-se o exemplo do código.

Listagem 3 - Deserializaçãode um objeto.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | import java.io.FileInputStream;  import java.io.ObjectInputStream;    public class Deserializador {        public Deserializador() {    }        public Object deserializar(String path) throws Exception {      FileInputStream inFile = new FileInputStream(path);      ObjectInputStream d = new ObjectInputStream(inFile);      Object o = d.readObject();      d.close();      return o;      }  } |

A Listagem 4, contém o exemplo da execução da serialização e em seguida da deserialização.

Listagem 4 - Serializando e Deserializando

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | import operacoes.Deserializador;  import operacoes.Serializador;  import pojo.Fruta;    public class Principal {      public static void main(String args[]){         //serializa         Serializador s = new Serializador();         Fruta fruta = new Fruta("Maça", "Vermelha");          try {              s.serializar("/home/gustavo/fruta", fruta);          } catch (Exception ex) {              System.err.println("Falha ao serializar! - " + ex.toString());          }            //deserializa          Deserializador d = new Deserializador();          fruta = null;          try {              fruta = (Fruta) d.deserializar("/home/gustavo/fruta");          } catch (Exception ex) {              System.err.println("Falha ao deserializar! - " + ex.toString());          }          System.out.println(fruta.getNome() + " - " + fruta.getCor());      }  } |

Na Listagem 5, encontra-se um exemplo de serialização e deserialização de um ArrayList da classe Fruta. É apenas mais um exemplo que demonstra a capacidade de se persistir qualquer objeto que seja serializável. Essa serialização só foi possível porque a classe “ArrayList” implementa a interface “Serializable” por padrão.

Listagem 5 - Serialização de ArrayList.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | //serialização e deserialização com ArrayList          ArrayList<Fruta> frutas = new ArrayList<Fruta>();          Fruta f1 = new Fruta("Laranja", "Amarela");          Fruta f2 = new Fruta("Abacate", "Verde");          Fruta f3 = new Fruta("Morango", "Vermelho");          frutas.add(f1);          frutas.add(f2);          frutas.add(f3);            try {              s.serializar("/home/gustavo/frutas", frutas);              frutas = null;              frutas = (ArrayList<Fruta>) d.deserializar("/home/gustavo/frutas");              for(Fruta f : frutas){                  System.out.println("ArrayList: " + f.getNome() + " - " + f.getCor());              }          } catch (Exception ex) {              System.err.println("Falha ao serializar ou deserializar! - " +                                 ex.toString());          } |

**4. PARTICULARIDADES DA SERIALIZAÇÃO**

Como dito, a serialização de um objeto em Java pode ser algo fácil de se fazer. Mas existem certas informações que são importantes e sabê-las pode evitar vários problemas.

Um dos problemas é que nem tudo pode ser serializado, como objetos das classes “Thread” e “Socket” e atributos estáticos. Os atributos estáticos estão vinculados à classe e não ao objeto. Assim quando se trata do estado do objeto, os campos estáticos não são considerados. Em relação às Threads, suas instâncias só são úteis enquanto estiverem em execução na JVM, e podem ser paradas/destruídas e iniciadas com instruções de sua classe. Não existe algo no estado destas instâncias que seja útil para se persistir, e por isso não há sentido em persistir tais instâncias para restaurar posteriormente.

Mas e se existir alguma classe em que seja necessário persisti-la, e esta possui uma instância da classe “Thread”? Neste caso, utilizando um recurso pode-se persistir a classe inteira, menos a instância thread. Este recurso, refere-se em indicar que tal instância thread é um objeto “Transient”. Na Listagem 6 encontra-se um exemplo da implementação com uso de “Transient”.

Listagem 6 - Uso do Transient

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | import java.io.Serializable;    public class ImplementacaoTransient implements Serializable, Runnable{        transient private Thread thread;      //marcado com "transient" para que sua instância não seja serializada        //Demais valores de estado serão serializados      private int valorParaPersistir;        public ImplementacaoTransient(int valorParaPersistir){          this.valorParaPersistir = valorParaPersistir;          thread = new Thread(this);          thread.start();      }        public void run()      {          while(true){              //código a ser executado pela Thread          }      }  } |

Vale ressaltar que, o que é serializado é apenas o estado do objeto, e não os métodos e a classe. No exemplo acima, apenas a instância do objeto “thread” e do objeto “valorParaPersistir” compunham o estado do objeto. Assim, evita-se que o sistema de serialização tente serializar um objeto não-serializável e gere erros. Após a deserialização de um objeto da classe “ImplementacaoTransient” este estará totalmente funcional.

Um outro problema que pode vir a ocorrer, se refere ao versionamento das classes. Toda classe possui um atributo oculto chamado “serialVersionUID” com um valor gerado automaticamente. Objetos criados levam consigo o “serialVersionUID” de sua classe, que indica sua versão. Toda vez que uma classe sofre alterações em seu código, um novo “serialVersionUID” é gerado. O problema com a serialização surge quando serializa-se o estado de um objeto e posteriormente tenta-se restaurar o estado para a aplicação, mas houveram mudanças no código da classe. Neste caso, uma exceção “java.io.InvalidClassException” é disparada, pois de fato, o estado do objeto corresponde a uma versão de classe do qual foi criado que já não existe mais. Portanto, isso só ocorre para evitar erros de programação, em que tenta-se restaurar o estado de um objeto para uma classe que possui versão diferente da qual o objeto foi serializado inicialmente. E isso pode significar diferentes versões de uma mesma classe “Fruta” por exemplo, ou diferentes versões por classes serem totalmente difere

|  |  |
| --- | --- |
| http://www.linhadecodigo.com.br/artigos/img_artigos/EvertonCoimbraAraujo/Serializazao_Objetos/imagem001.jpg | Neste artigo apresento o Gustavo Rafael Valiati, que pode ser contatado pelo e-mail gustavovaliati@gmail.com. O Gustavo foi meu aluno em algumas disciplinas de sua gradução e está concluíndo o último período do curso de Tecnologia em Análise e Desenvolvimento de Sistemas da UTFPR campus Medianeira, neste primeiro semestre de 2012. |

**1. INTRODUÇÃO**

Nem sempre se deseja ter objetos ou estrutura de dados de um aplicativo, com um ciclo de vida que dura no máximo o tempo em que for armazenado em memória RAM. Ou seja, algumas vezes deseja-se guardá-los por um tempo indeterminado, sendo possível recuperá-los posteriormente. E neste caso, o termo “persistência” tem seu conceito definido.

Persistir dados, requer que estes possam ser serializáveis. A serialização possibilitará capturar o estado do objeto ou a estrutura de dados, e transformar em uma cadeia de bytes quando for necessário. Torna-se possível também, recuperar os bytes persistidos para fazer o processo inverso, e ter os dados de volta para a aplicação em execução.

Desta maneira, qualquer forma de dados que esteja sendo trabalhada na aplicação, e esta puder ser serializada, poderá ser passada para o meio físico, e ter o processo revertido.

A serialização não se limita apenas em possibilitar a gravação do dado em disco, mas permite que o mesmo seja transmitido por rede. Isto porquê a serialização cria um stream de bytes, que é requisito para transmissão de dados por rede.

Diversas linguagens possuem suporte à serialização, como: Java, C, C++, Python, PHP, .NET, entre outras. Neste artigo, será tratado apenas da serialização com Java, em sua implemetação básica, alguns problemas e soluções, com códigos para exemplificação.

**2. SERIALIZAÇÃO COM JAVA**

Serializar um objeto, em Java, só é possível caso sua classe esteja marcada como serializável. Essa marcação é feita com a simples implementação da interface “java.io.Serializable”. Essa interface não possui métodos para se implementar e nem atributos. Ainda não existiam as anotações no Java 1.1, versão que trouxe pela primeira vez a serialização ao Java, e desta maneira a implementação da interface era a melhor maneira de marcar uma classe.

A herança entre classes é naturalmente afetada pela serialização. Já que quando uma classe implementa a interface “Serializable”, toda classe que dela estender também estará implicitamente marcada como serializável

.

Para exemplo, será criada uma classe “pojo.Fruta” que contém dois atributos: nome e cor. No código exemplo (Listagem 1), pode-se notar a classe “Fruta” marcada para a serialização:

Listagem 1 - Marcando uma classe como serializável.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | import java.io.Serializable;    public class Fruta implements Serializable{      private String nome;      private String cor;        public Fruta(String nome, String cor) {          this.nome = nome;          this.cor = cor;      }        public String getCor() { return cor; }      public String getNome() { return nome; }      public void setCor(String cor) { this.cor = cor; }      public void setNome(String nome) { this.nome = nome; }  } |

Pode-se dizer que a implementação do processo de serialização e deserialização simples de um objeto, não possui um código difícil de se escrever. Isto porquê duas classes fazem a maior parte do trabalho, de maneira que com poucas linhas de código pode-se transformar um objeto em bytes e vice-versa. Essas duas classes são “java.io.ObjectInputStream” que recebe bytes para criar um objeto e “java.io.ObjectOutputStream” que cria uma cadeia de bytes a partir de um objeto.

**3. SERIALIZANDO E DESERIALIZANDO O ESTADO DE UM OBJETO**

Como exemplo, será persistido um objeto em disco. Para isto, além das classes “java.io.ObjectInputStream” e “java.io.ObjectOutputStream”, serão utilizadas duas outras que trabalham com arquivos, que são “java.io.FileInputStream” e “java.io.FileOutputStream”, que seguem a mesma lógica de InputStream e OutputStream, só que agora relacionado ao sistema de arquivos e não aos objetos.

Para serialização, deve-se criar um novo “java.io.FileOutputStream”, com o caminho (“path”) do arquivo onde se deseja armazenar o estado do objeto. Também deve ser criado um novo “java.io.ObjectOutputStream” que está ligado ao arquivo. E por último deve-se executar a escrita do objeto alvo, para dentro do objeto de stream que por sua vez está ligado ao arquivo destino. Após executados esses passos, é possível encontrar no sistema de arquivos um novo arquivo que contém os bytes correspondentes à descrição do estado do objeto. Na Listagem 2, encontra-se o código exemplo.

Listagem 2 - Serialização de um objeto.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | import java.io.FileOutputStream;  import java.io.ObjectOutputStream;    public class Serializador {        public Serializador() {    }        public void serializar(String path, Object obj) throws Exception {              FileOutputStream outFile = new FileOutputStream(path);      ObjectOutputStream s = new ObjectOutputStream(outFile);      s.writeObject(obj);      s.close();      }  } |

E para a deserialização, o processo é extremamente parecido com a serialização. Cria-se um novo “java.io.FileInputStream” relacionado ao arquivo onde o estado do objeto encontra-se persistido. Deve-se criar também um novo “java.io.ObjectInputStream” que estará vinculado ao arquivo de origem. Por último, executa-se a leitura do arquivo para a restauração do estado. Na Listagem 3 encontra-se o exemplo do código.

Listagem 3 - Deserializaçãode um objeto.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | import java.io.FileInputStream;  import java.io.ObjectInputStream;    public class Deserializador {        public Deserializador() {    }        public Object deserializar(String path) throws Exception {      FileInputStream inFile = new FileInputStream(path);      ObjectInputStream d = new ObjectInputStream(inFile);      Object o = d.readObject();      d.close();      return o;      }  } |

A Listagem 4, contém o exemplo da execução da serialização e em seguida da deserialização.

Listagem 4 - Serializando e Deserializando

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | import operacoes.Deserializador;  import operacoes.Serializador;  import pojo.Fruta;    public class Principal {      public static void main(String args[]){         //serializa         Serializador s = new Serializador();         Fruta fruta = new Fruta("Maça", "Vermelha");          try {              s.serializar("/home/gustavo/fruta", fruta);          } catch (Exception ex) {              System.err.println("Falha ao serializar! - " + ex.toString());          }            //deserializa          Deserializador d = new Deserializador();          fruta = null;          try {              fruta = (Fruta) d.deserializar("/home/gustavo/fruta");          } catch (Exception ex) {              System.err.println("Falha ao deserializar! - " + ex.toString());          }          System.out.println(fruta.getNome() + " - " + fruta.getCor());      }  } |

Na Listagem 5, encontra-se um exemplo de serialização e deserialização de um ArrayList da classe Fruta. É apenas mais um exemplo que demonstra a capacidade de se persistir qualquer objeto que seja serializável. Essa serialização só foi possível porque a classe “ArrayList” implementa a interface “Serializable” por padrão.

Listagem 5 - Serialização de ArrayList.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | //serialização e deserialização com ArrayList          ArrayList<Fruta> frutas = new ArrayList<Fruta>();          Fruta f1 = new Fruta("Laranja", "Amarela");          Fruta f2 = new Fruta("Abacate", "Verde");          Fruta f3 = new Fruta("Morango", "Vermelho");          frutas.add(f1);          frutas.add(f2);          frutas.add(f3);            try {              s.serializar("/home/gustavo/frutas", frutas);              frutas = null;              frutas = (ArrayList<Fruta>) d.deserializar("/home/gustavo/frutas");              for(Fruta f : frutas){                  System.out.println("ArrayList: " + f.getNome() + " - " + f.getCor());              }          } catch (Exception ex) {              System.err.println("Falha ao serializar ou deserializar! - " +                                 ex.toString());          } |

**4. PARTICULARIDADES DA SERIALIZAÇÃO**

Como dito, a serialização de um objeto em Java pode ser algo fácil de se fazer. Mas existem certas informações que são importantes e sabê-las pode evitar vários problemas.

Um dos problemas é que nem tudo pode ser serializado, como objetos das classes “Thread” e “Socket” e atributos estáticos. Os atributos estáticos estão vinculados à classe e não ao objeto. Assim quando se trata do estado do objeto, os campos estáticos não são considerados. Em relação às Threads, suas instâncias só são úteis enquanto estiverem em execução na JVM, e podem ser paradas/destruídas e iniciadas com instruções de sua classe. Não existe algo no estado destas instâncias que seja útil para se persistir, e por isso não há sentido em persistir tais instâncias para restaurar posteriormente.

Mas e se existir alguma classe em que seja necessário persisti-la, e esta possui uma instância da classe “Thread”? Neste caso, utilizando um recurso pode-se persistir a classe inteira, menos a instância thread. Este recurso, refere-se em indicar que tal instância thread é um objeto “Transient”. Na Listagem 6 encontra-se um exemplo da implementação com uso de “Transient”.

Listagem 6 - Uso do Transient

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | import java.io.Serializable;    public class ImplementacaoTransient implements Serializable, Runnable{        transient private Thread thread;      //marcado com "transient" para que sua instância não seja serializada        //Demais valores de estado serão serializados      private int valorParaPersistir;        public ImplementacaoTransient(int valorParaPersistir){          this.valorParaPersistir = valorParaPersistir;          thread = new Thread(this);          thread.start();      }        public void run()      {          while(true){              //código a ser executado pela Thread          }      }  } |

Vale ressaltar que, o que é serializado é apenas o estado do objeto, e não os métodos e a classe. No exemplo acima, apenas a instância do objeto “thread” e do objeto “valorParaPersistir” compunham o estado do objeto. Assim, evita-se que o sistema de serialização tente serializar um objeto não-serializável e gere erros. Após a deserialização de um objeto da classe “ImplementacaoTransient” este estará totalmente funcional.

Um outro problema que pode vir a ocorrer, se refere ao versionamento das classes. Toda classe possui um atributo oculto chamado “serialVersionUID” com um valor gerado automaticamente. Objetos criados levam consigo o “serialVersionUID” de sua classe, que indica sua versão. Toda vez que uma classe sofre alterações em seu código, um novo “serialVersionUID” é gerado. O problema com a serialização surge quando serializa-se o estado de um objeto e posteriormente tenta-se restaurar o estado para a aplicação, mas houveram mudanças no código da classe. Neste caso, uma exceção “java.io.InvalidClassException” é disparada, pois de fato, o estado do objeto corresponde a uma versão de classe do qual foi criado que já não existe mais. Portanto, isso só ocorre para evitar erros de programação, em que tenta-se restaurar o estado de um objeto para uma classe que possui versão diferente da qual o objeto foi serializado inicialmente. E isso pode significar diferentes versões de uma mesma classe “Fruta” por exemplo, ou diferentes versões por classes serem totalmente difere

|  |  |
| --- | --- |
| http://www.linhadecodigo.com.br/artigos/img_artigos/EvertonCoimbraAraujo/Serializazao_Objetos/imagem001.jpg | Neste artigo apresento o Gustavo Rafael Valiati, que pode ser contatado pelo e-mail gustavovaliati@gmail.com. O Gustavo foi meu aluno em algumas disciplinas de sua gradução e está concluíndo o último período do curso de Tecnologia em Análise e Desenvolvimento de Sistemas da UTFPR campus Medianeira, neste primeiro semestre de 2012. |

**1. INTRODUÇÃO**

Nem sempre se deseja ter objetos ou estrutura de dados de um aplicativo, com um ciclo de vida que dura no máximo o tempo em que for armazenado em memória RAM. Ou seja, algumas vezes deseja-se guardá-los por um tempo indeterminado, sendo possível recuperá-los posteriormente. E neste caso, o termo “persistência” tem seu conceito definido.

Persistir dados, requer que estes possam ser serializáveis. A serialização possibilitará capturar o estado do objeto ou a estrutura de dados, e transformar em uma cadeia de bytes quando for necessário. Torna-se possível também, recuperar os bytes persistidos para fazer o processo inverso, e ter os dados de volta para a aplicação em execução.

Desta maneira, qualquer forma de dados que esteja sendo trabalhada na aplicação, e esta puder ser serializada, poderá ser passada para o meio físico, e ter o processo revertido.

A serialização não se limita apenas em possibilitar a gravação do dado em disco, mas permite que o mesmo seja transmitido por rede. Isto porquê a serialização cria um stream de bytes, que é requisito para transmissão de dados por rede.

Diversas linguagens possuem suporte à serialização, como: Java, C, C++, Python, PHP, .NET, entre outras. Neste artigo, será tratado apenas da serialização com Java, em sua implemetação básica, alguns problemas e soluções, com códigos para exemplificação.

**2. SERIALIZAÇÃO COM JAVA**

Serializar um objeto, em Java, só é possível caso sua classe esteja marcada como serializável. Essa marcação é feita com a simples implementação da interface “java.io.Serializable”. Essa interface não possui métodos para se implementar e nem atributos. Ainda não existiam as anotações no Java 1.1, versão que trouxe pela primeira vez a serialização ao Java, e desta maneira a implementação da interface era a melhor maneira de marcar uma classe.

A herança entre classes é naturalmente afetada pela serialização. Já que quando uma classe implementa a interface “Serializable”, toda classe que dela estender também estará implicitamente marcada como serializável

.

Para exemplo, será criada uma classe “pojo.Fruta” que contém dois atributos: nome e cor. No código exemplo (Listagem 1), pode-se notar a classe “Fruta” marcada para a serialização:

Listagem 1 - Marcando uma classe como serializável.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | import java.io.Serializable;    public class Fruta implements Serializable{      private String nome;      private String cor;        public Fruta(String nome, String cor) {          this.nome = nome;          this.cor = cor;      }        public String getCor() { return cor; }      public String getNome() { return nome; }      public void setCor(String cor) { this.cor = cor; }      public void setNome(String nome) { this.nome = nome; }  } |

Pode-se dizer que a implementação do processo de serialização e deserialização simples de um objeto, não possui um código difícil de se escrever. Isto porquê duas classes fazem a maior parte do trabalho, de maneira que com poucas linhas de código pode-se transformar um objeto em bytes e vice-versa. Essas duas classes são “java.io.ObjectInputStream” que recebe bytes para criar um objeto e “java.io.ObjectOutputStream” que cria uma cadeia de bytes a partir de um objeto.

**3. SERIALIZANDO E DESERIALIZANDO O ESTADO DE UM OBJETO**

Como exemplo, será persistido um objeto em disco. Para isto, além das classes “java.io.ObjectInputStream” e “java.io.ObjectOutputStream”, serão utilizadas duas outras que trabalham com arquivos, que são “java.io.FileInputStream” e “java.io.FileOutputStream”, que seguem a mesma lógica de InputStream e OutputStream, só que agora relacionado ao sistema de arquivos e não aos objetos.

Para serialização, deve-se criar um novo “java.io.FileOutputStream”, com o caminho (“path”) do arquivo onde se deseja armazenar o estado do objeto. Também deve ser criado um novo “java.io.ObjectOutputStream” que está ligado ao arquivo. E por último deve-se executar a escrita do objeto alvo, para dentro do objeto de stream que por sua vez está ligado ao arquivo destino. Após executados esses passos, é possível encontrar no sistema de arquivos um novo arquivo que contém os bytes correspondentes à descrição do estado do objeto. Na Listagem 2, encontra-se o código exemplo.

Listagem 2 - Serialização de um objeto.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | import java.io.FileOutputStream;  import java.io.ObjectOutputStream;    public class Serializador {        public Serializador() {    }        public void serializar(String path, Object obj) throws Exception {              FileOutputStream outFile = new FileOutputStream(path);      ObjectOutputStream s = new ObjectOutputStream(outFile);      s.writeObject(obj);      s.close();      }  } |

E para a deserialização, o processo é extremamente parecido com a serialização. Cria-se um novo “java.io.FileInputStream” relacionado ao arquivo onde o estado do objeto encontra-se persistido. Deve-se criar também um novo “java.io.ObjectInputStream” que estará vinculado ao arquivo de origem. Por último, executa-se a leitura do arquivo para a restauração do estado. Na Listagem 3 encontra-se o exemplo do código.

Listagem 3 - Deserializaçãode um objeto.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | import java.io.FileInputStream;  import java.io.ObjectInputStream;    public class Deserializador {        public Deserializador() {    }        public Object deserializar(String path) throws Exception {      FileInputStream inFile = new FileInputStream(path);      ObjectInputStream d = new ObjectInputStream(inFile);      Object o = d.readObject();      d.close();      return o;      }  } |

A Listagem 4, contém o exemplo da execução da serialização e em seguida da deserialização.

Listagem 4 - Serializando e Deserializando

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | import operacoes.Deserializador;  import operacoes.Serializador;  import pojo.Fruta;    public class Principal {      public static void main(String args[]){         //serializa         Serializador s = new Serializador();         Fruta fruta = new Fruta("Maça", "Vermelha");          try {              s.serializar("/home/gustavo/fruta", fruta);          } catch (Exception ex) {              System.err.println("Falha ao serializar! - " + ex.toString());          }            //deserializa          Deserializador d = new Deserializador();          fruta = null;          try {              fruta = (Fruta) d.deserializar("/home/gustavo/fruta");          } catch (Exception ex) {              System.err.println("Falha ao deserializar! - " + ex.toString());          }          System.out.println(fruta.getNome() + " - " + fruta.getCor());      }  } |

Na Listagem 5, encontra-se um exemplo de serialização e deserialização de um ArrayList da classe Fruta. É apenas mais um exemplo que demonstra a capacidade de se persistir qualquer objeto que seja serializável. Essa serialização só foi possível porque a classe “ArrayList” implementa a interface “Serializable” por padrão.

Listagem 5 - Serialização de ArrayList.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | //serialização e deserialização com ArrayList          ArrayList<Fruta> frutas = new ArrayList<Fruta>();          Fruta f1 = new Fruta("Laranja", "Amarela");          Fruta f2 = new Fruta("Abacate", "Verde");          Fruta f3 = new Fruta("Morango", "Vermelho");          frutas.add(f1);          frutas.add(f2);          frutas.add(f3);            try {              s.serializar("/home/gustavo/frutas", frutas);              frutas = null;              frutas = (ArrayList<Fruta>) d.deserializar("/home/gustavo/frutas");              for(Fruta f : frutas){                  System.out.println("ArrayList: " + f.getNome() + " - " + f.getCor());              }          } catch (Exception ex) {              System.err.println("Falha ao serializar ou deserializar! - " +                                 ex.toString());          } |

**4. PARTICULARIDADES DA SERIALIZAÇÃO**

Como dito, a serialização de um objeto em Java pode ser algo fácil de se fazer. Mas existem certas informações que são importantes e sabê-las pode evitar vários problemas.

Um dos problemas é que nem tudo pode ser serializado, como objetos das classes “Thread” e “Socket” e atributos estáticos. Os atributos estáticos estão vinculados à classe e não ao objeto. Assim quando se trata do estado do objeto, os campos estáticos não são considerados. Em relação às Threads, suas instâncias só são úteis enquanto estiverem em execução na JVM, e podem ser paradas/destruídas e iniciadas com instruções de sua classe. Não existe algo no estado destas instâncias que seja útil para se persistir, e por isso não há sentido em persistir tais instâncias para restaurar posteriormente.

Mas e se existir alguma classe em que seja necessário persisti-la, e esta possui uma instância da classe “Thread”? Neste caso, utilizando um recurso pode-se persistir a classe inteira, menos a instância thread. Este recurso, refere-se em indicar que tal instância thread é um objeto “Transient”. Na Listagem 6 encontra-se um exemplo da implementação com uso de “Transient”.

Listagem 6 - Uso do Transient

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | import java.io.Serializable;    public class ImplementacaoTransient implements Serializable, Runnable{        transient private Thread thread;      //marcado com "transient" para que sua instância não seja serializada        //Demais valores de estado serão serializados      private int valorParaPersistir;        public ImplementacaoTransient(int valorParaPersistir){          this.valorParaPersistir = valorParaPersistir;          thread = new Thread(this);          thread.start();      }        public void run()      {          while(true){              //código a ser executado pela Thread          }      }  } |

Vale ressaltar que, o que é serializado é apenas o estado do objeto, e não os métodos e a classe. No exemplo acima, apenas a instância do objeto “thread” e do objeto “valorParaPersistir” compunham o estado do objeto. Assim, evita-se que o sistema de serialização tente serializar um objeto não-serializável e gere erros. Após a deserialização de um objeto da classe “ImplementacaoTransient” este estará totalmente funcional.

Um outro problema que pode vir a ocorrer, se refere ao versionamento das classes. Toda classe possui um atributo oculto chamado “serialVersionUID” com um valor gerado automaticamente. Objetos criados levam consigo o “serialVersionUID” de sua classe, que indica sua versão. Toda vez que uma classe sofre alterações em seu código, um novo “serialVersionUID” é gerado. O problema com a serialização surge quando serializa-se o estado de um objeto e posteriormente tenta-se restaurar o estado para a aplicação, mas houveram mudanças no código da classe. Neste caso, uma exceção “java.io.InvalidClassException” é disparada, pois de fato, o estado do objeto corresponde a uma versão de classe do qual foi criado que já não existe mais. Portanto, isso só ocorre para evitar erros de programação, em que tenta-se restaurar o estado de um objeto para uma classe que possui versão diferente da qual o objeto foi serializado inicialmente. E isso pode significar diferentes versões de uma mesma classe “Fruta” por exemplo, ou diferentes versões por classes serem totalmente difere

|  |  |
| --- | --- |
| http://www.linhadecodigo.com.br/artigos/img_artigos/EvertonCoimbraAraujo/Serializazao_Objetos/imagem001.jpg | Neste artigo apresento o Gustavo Rafael Valiati, que pode ser contatado pelo e-mail gustavovaliati@gmail.com. O Gustavo foi meu aluno em algumas disciplinas de sua gradução e está concluíndo o último período do curso de Tecnologia em Análise e Desenvolvimento de Sistemas da UTFPR campus Medianeira, neste primeiro semestre de 2012. |

**1. INTRODUÇÃO**

Nem sempre se deseja ter objetos ou estrutura de dados de um aplicativo, com um ciclo de vida que dura no máximo o tempo em que for armazenado em memória RAM. Ou seja, algumas vezes deseja-se guardá-los por um tempo indeterminado, sendo possível recuperá-los posteriormente. E neste caso, o termo “persistência” tem seu conceito definido.

Persistir dados, requer que estes possam ser serializáveis. A serialização possibilitará capturar o estado do objeto ou a estrutura de dados, e transformar em uma cadeia de bytes quando for necessário. Torna-se possível também, recuperar os bytes persistidos para fazer o processo inverso, e ter os dados de volta para a aplicação em execução.

Desta maneira, qualquer forma de dados que esteja sendo trabalhada na aplicação, e esta puder ser serializada, poderá ser passada para o meio físico, e ter o processo revertido.

A serialização não se limita apenas em possibilitar a gravação do dado em disco, mas permite que o mesmo seja transmitido por rede. Isto porquê a serialização cria um stream de bytes, que é requisito para transmissão de dados por rede.

Diversas linguagens possuem suporte à serialização, como: Java, C, C++, Python, PHP, .NET, entre outras. Neste artigo, será tratado apenas da serialização com Java, em sua implemetação básica, alguns problemas e soluções, com códigos para exemplificação.

**2. SERIALIZAÇÃO COM JAVA**

Serializar um objeto, em Java, só é possível caso sua classe esteja marcada como serializável. Essa marcação é feita com a simples implementação da interface “java.io.Serializable”. Essa interface não possui métodos para se implementar e nem atributos. Ainda não existiam as anotações no Java 1.1, versão que trouxe pela primeira vez a serialização ao Java, e desta maneira a implementação da interface era a melhor maneira de marcar uma classe.

A herança entre classes é naturalmente afetada pela serialização. Já que quando uma classe implementa a interface “Serializable”, toda classe que dela estender também estará implicitamente marcada como serializável

.

Para exemplo, será criada uma classe “pojo.Fruta” que contém dois atributos: nome e cor. No código exemplo (Listagem 1), pode-se notar a classe “Fruta” marcada para a serialização:

Listagem 1 - Marcando uma classe como serializável.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | import java.io.Serializable;    public class Fruta implements Serializable{      private String nome;      private String cor;        public Fruta(String nome, String cor) {          this.nome = nome;          this.cor = cor;      }        public String getCor() { return cor; }      public String getNome() { return nome; }      public void setCor(String cor) { this.cor = cor; }      public void setNome(String nome) { this.nome = nome; }  } |

Pode-se dizer que a implementação do processo de serialização e deserialização simples de um objeto, não possui um código difícil de se escrever. Isto porquê duas classes fazem a maior parte do trabalho, de maneira que com poucas linhas de código pode-se transformar um objeto em bytes e vice-versa. Essas duas classes são “java.io.ObjectInputStream” que recebe bytes para criar um objeto e “java.io.ObjectOutputStream” que cria uma cadeia de bytes a partir de um objeto.

**3. SERIALIZANDO E DESERIALIZANDO O ESTADO DE UM OBJETO**

Como exemplo, será persistido um objeto em disco. Para isto, além das classes “java.io.ObjectInputStream” e “java.io.ObjectOutputStream”, serão utilizadas duas outras que trabalham com arquivos, que são “java.io.FileInputStream” e “java.io.FileOutputStream”, que seguem a mesma lógica de InputStream e OutputStream, só que agora relacionado ao sistema de arquivos e não aos objetos.

Para serialização, deve-se criar um novo “java.io.FileOutputStream”, com o caminho (“path”) do arquivo onde se deseja armazenar o estado do objeto. Também deve ser criado um novo “java.io.ObjectOutputStream” que está ligado ao arquivo. E por último deve-se executar a escrita do objeto alvo, para dentro do objeto de stream que por sua vez está ligado ao arquivo destino. Após executados esses passos, é possível encontrar no sistema de arquivos um novo arquivo que contém os bytes correspondentes à descrição do estado do objeto. Na Listagem 2, encontra-se o código exemplo.

Listagem 2 - Serialização de um objeto.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | import java.io.FileOutputStream;  import java.io.ObjectOutputStream;    public class Serializador {        public Serializador() {    }        public void serializar(String path, Object obj) throws Exception {              FileOutputStream outFile = new FileOutputStream(path);      ObjectOutputStream s = new ObjectOutputStream(outFile);      s.writeObject(obj);      s.close();      }  } |

E para a deserialização, o processo é extremamente parecido com a serialização. Cria-se um novo “java.io.FileInputStream” relacionado ao arquivo onde o estado do objeto encontra-se persistido. Deve-se criar também um novo “java.io.ObjectInputStream” que estará vinculado ao arquivo de origem. Por último, executa-se a leitura do arquivo para a restauração do estado. Na Listagem 3 encontra-se o exemplo do código.

Listagem 3 - Deserializaçãode um objeto.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | import java.io.FileInputStream;  import java.io.ObjectInputStream;    public class Deserializador {        public Deserializador() {    }        public Object deserializar(String path) throws Exception {      FileInputStream inFile = new FileInputStream(path);      ObjectInputStream d = new ObjectInputStream(inFile);      Object o = d.readObject();      d.close();      return o;      }  } |

A Listagem 4, contém o exemplo da execução da serialização e em seguida da deserialização.

Listagem 4 - Serializando e Deserializando

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | import operacoes.Deserializador;  import operacoes.Serializador;  import pojo.Fruta;    public class Principal {      public static void main(String args[]){         //serializa         Serializador s = new Serializador();         Fruta fruta = new Fruta("Maça", "Vermelha");          try {              s.serializar("/home/gustavo/fruta", fruta);          } catch (Exception ex) {              System.err.println("Falha ao serializar! - " + ex.toString());          }            //deserializa          Deserializador d = new Deserializador();          fruta = null;          try {              fruta = (Fruta) d.deserializar("/home/gustavo/fruta");          } catch (Exception ex) {              System.err.println("Falha ao deserializar! - " + ex.toString());          }          System.out.println(fruta.getNome() + " - " + fruta.getCor());      }  } |

Na Listagem 5, encontra-se um exemplo de serialização e deserialização de um ArrayList da classe Fruta. É apenas mais um exemplo que demonstra a capacidade de se persistir qualquer objeto que seja serializável. Essa serialização só foi possível porque a classe “ArrayList” implementa a interface “Serializable” por padrão.

Listagem 5 - Serialização de ArrayList.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | //serialização e deserialização com ArrayList          ArrayList<Fruta> frutas = new ArrayList<Fruta>();          Fruta f1 = new Fruta("Laranja", "Amarela");          Fruta f2 = new Fruta("Abacate", "Verde");          Fruta f3 = new Fruta("Morango", "Vermelho");          frutas.add(f1);          frutas.add(f2);          frutas.add(f3);            try {              s.serializar("/home/gustavo/frutas", frutas);              frutas = null;              frutas = (ArrayList<Fruta>) d.deserializar("/home/gustavo/frutas");              for(Fruta f : frutas){                  System.out.println("ArrayList: " + f.getNome() + " - " + f.getCor());              }          } catch (Exception ex) {              System.err.println("Falha ao serializar ou deserializar! - " +                                 ex.toString());          } |

**4. PARTICULARIDADES DA SERIALIZAÇÃO**

Como dito, a serialização de um objeto em Java pode ser algo fácil de se fazer. Mas existem certas informações que são importantes e sabê-las pode evitar vários problemas.

Um dos problemas é que nem tudo pode ser serializado, como objetos das classes “Thread” e “Socket” e atributos estáticos. Os atributos estáticos estão vinculados à classe e não ao objeto. Assim quando se trata do estado do objeto, os campos estáticos não são considerados. Em relação às Threads, suas instâncias só são úteis enquanto estiverem em execução na JVM, e podem ser paradas/destruídas e iniciadas com instruções de sua classe. Não existe algo no estado destas instâncias que seja útil para se persistir, e por isso não há sentido em persistir tais instâncias para restaurar posteriormente.

Mas e se existir alguma classe em que seja necessário persisti-la, e esta possui uma instância da classe “Thread”? Neste caso, utilizando um recurso pode-se persistir a classe inteira, menos a instância thread. Este recurso, refere-se em indicar que tal instância thread é um objeto “Transient”. Na Listagem 6 encontra-se um exemplo da implementação com uso de “Transient”.

Listagem 6 - Uso do Transient

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | import java.io.Serializable;    public class ImplementacaoTransient implements Serializable, Runnable{        transient private Thread thread;      //marcado com "transient" para que sua instância não seja serializada        //Demais valores de estado serão serializados      private int valorParaPersistir;        public ImplementacaoTransient(int valorParaPersistir){          this.valorParaPersistir = valorParaPersistir;          thread = new Thread(this);          thread.start();      }        public void run()      {          while(true){              //código a ser executado pela Thread          }      }  } |

Vale ressaltar que, o que é serializado é apenas o estado do objeto, e não os métodos e a classe. No exemplo acima, apenas a instância do objeto “thread” e do objeto “valorParaPersistir” compunham o estado do objeto. Assim, evita-se que o sistema de serialização tente serializar um objeto não-serializável e gere erros. Após a deserialização de um objeto da classe “ImplementacaoTransient” este estará totalmente funcional.

Um outro problema que pode vir a ocorrer, se refere ao versionamento das classes. Toda classe possui um atributo oculto chamado “serialVersionUID” com um valor gerado automaticamente. Objetos criados levam consigo o “serialVersionUID” de sua classe, que indica sua versão. Toda vez que uma classe sofre alterações em seu código, um novo “serialVersionUID” é gerado. O problema com a serialização surge quando serializa-se o estado de um objeto e posteriormente tenta-se restaurar o estado para a aplicação, mas houveram mudanças no código da classe. Neste caso, uma exceção “java.io.InvalidClassException” é disparada, pois de fato, o estado do objeto corresponde a uma versão de classe do qual foi criado que já não existe mais. Portanto, isso só ocorre para evitar erros de programação, em que tenta-se restaurar o estado de um objeto para uma classe que possui versão diferente da qual o objeto foi serializado inicialmente. E isso pode significar diferentes versões de uma mesma classe “Fruta” por exemplo, ou diferentes versões por classes serem totalmente difere

|  |  |
| --- | --- |
| http://www.linhadecodigo.com.br/artigos/img_artigos/EvertonCoimbraAraujo/Serializazao_Objetos/imagem001.jpg | Neste artigo apresento o Gustavo Rafael Valiati, que pode ser contatado pelo e-mail gustavovaliati@gmail.com. O Gustavo foi meu aluno em algumas disciplinas de sua gradução e está concluíndo o último período do curso de Tecnologia em Análise e Desenvolvimento de Sistemas da UTFPR campus Medianeira, neste primeiro semestre de 2012. |

**1. INTRODUÇÃO**

Nem sempre se deseja ter objetos ou estrutura de dados de um aplicativo, com um ciclo de vida que dura no máximo o tempo em que for armazenado em memória RAM. Ou seja, algumas vezes deseja-se guardá-los por um tempo indeterminado, sendo possível recuperá-los posteriormente. E neste caso, o termo “persistência” tem seu conceito definido.

Persistir dados, requer que estes possam ser serializáveis. A serialização possibilitará capturar o estado do objeto ou a estrutura de dados, e transformar em uma cadeia de bytes quando for necessário. Torna-se possível também, recuperar os bytes persistidos para fazer o processo inverso, e ter os dados de volta para a aplicação em execução.

Desta maneira, qualquer forma de dados que esteja sendo trabalhada na aplicação, e esta puder ser serializada, poderá ser passada para o meio físico, e ter o processo revertido.

A serialização não se limita apenas em possibilitar a gravação do dado em disco, mas permite que o mesmo seja transmitido por rede. Isto porquê a serialização cria um stream de bytes, que é requisito para transmissão de dados por rede.

Diversas linguagens possuem suporte à serialização, como: Java, C, C++, Python, PHP, .NET, entre outras. Neste artigo, será tratado apenas da serialização com Java, em sua implemetação básica, alguns problemas e soluções, com códigos para exemplificação.

**2. SERIALIZAÇÃO COM JAVA**

Serializar um objeto, em Java, só é possível caso sua classe esteja marcada como serializável. Essa marcação é feita com a simples implementação da interface “java.io.Serializable”. Essa interface não possui métodos para se implementar e nem atributos. Ainda não existiam as anotações no Java 1.1, versão que trouxe pela primeira vez a serialização ao Java, e desta maneira a implementação da interface era a melhor maneira de marcar uma classe.

A herança entre classes é naturalmente afetada pela serialização. Já que quando uma classe implementa a interface “Serializable”, toda classe que dela estender também estará implicitamente marcada como serializável

.

Para exemplo, será criada uma classe “pojo.Fruta” que contém dois atributos: nome e cor. No código exemplo (Listagem 1), pode-se notar a classe “Fruta” marcada para a serialização:

Listagem 1 - Marcando uma classe como serializável.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | import java.io.Serializable;    public class Fruta implements Serializable{      private String nome;      private String cor;        public Fruta(String nome, String cor) {          this.nome = nome;          this.cor = cor;      }        public String getCor() { return cor; }      public String getNome() { return nome; }      public void setCor(String cor) { this.cor = cor; }      public void setNome(String nome) { this.nome = nome; }  } |

Pode-se dizer que a implementação do processo de serialização e deserialização simples de um objeto, não possui um código difícil de se escrever. Isto porquê duas classes fazem a maior parte do trabalho, de maneira que com poucas linhas de código pode-se transformar um objeto em bytes e vice-versa. Essas duas classes são “java.io.ObjectInputStream” que recebe bytes para criar um objeto e “java.io.ObjectOutputStream” que cria uma cadeia de bytes a partir de um objeto.

**3. SERIALIZANDO E DESERIALIZANDO O ESTADO DE UM OBJETO**

Como exemplo, será persistido um objeto em disco. Para isto, além das classes “java.io.ObjectInputStream” e “java.io.ObjectOutputStream”, serão utilizadas duas outras que trabalham com arquivos, que são “java.io.FileInputStream” e “java.io.FileOutputStream”, que seguem a mesma lógica de InputStream e OutputStream, só que agora relacionado ao sistema de arquivos e não aos objetos.

Para serialização, deve-se criar um novo “java.io.FileOutputStream”, com o caminho (“path”) do arquivo onde se deseja armazenar o estado do objeto. Também deve ser criado um novo “java.io.ObjectOutputStream” que está ligado ao arquivo. E por último deve-se executar a escrita do objeto alvo, para dentro do objeto de stream que por sua vez está ligado ao arquivo destino. Após executados esses passos, é possível encontrar no sistema de arquivos um novo arquivo que contém os bytes correspondentes à descrição do estado do objeto. Na Listagem 2, encontra-se o código exemplo.

Listagem 2 - Serialização de um objeto.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | import java.io.FileOutputStream;  import java.io.ObjectOutputStream;    public class Serializador {        public Serializador() {    }        public void serializar(String path, Object obj) throws Exception {              FileOutputStream outFile = new FileOutputStream(path);      ObjectOutputStream s = new ObjectOutputStream(outFile);      s.writeObject(obj);      s.close();      }  } |

E para a deserialização, o processo é extremamente parecido com a serialização. Cria-se um novo “java.io.FileInputStream” relacionado ao arquivo onde o estado do objeto encontra-se persistido. Deve-se criar também um novo “java.io.ObjectInputStream” que estará vinculado ao arquivo de origem. Por último, executa-se a leitura do arquivo para a restauração do estado. Na Listagem 3 encontra-se o exemplo do código.

Listagem 3 - Deserializaçãode um objeto.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | import java.io.FileInputStream;  import java.io.ObjectInputStream;    public class Deserializador {        public Deserializador() {    }        public Object deserializar(String path) throws Exception {      FileInputStream inFile = new FileInputStream(path);      ObjectInputStream d = new ObjectInputStream(inFile);      Object o = d.readObject();      d.close();      return o;      }  } |

A Listagem 4, contém o exemplo da execução da serialização e em seguida da deserialização.

Listagem 4 - Serializando e Deserializando

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | import operacoes.Deserializador;  import operacoes.Serializador;  import pojo.Fruta;    public class Principal {      public static void main(String args[]){         //serializa         Serializador s = new Serializador();         Fruta fruta = new Fruta("Maça", "Vermelha");          try {              s.serializar("/home/gustavo/fruta", fruta);          } catch (Exception ex) {              System.err.println("Falha ao serializar! - " + ex.toString());          }            //deserializa          Deserializador d = new Deserializador();          fruta = null;          try {              fruta = (Fruta) d.deserializar("/home/gustavo/fruta");          } catch (Exception ex) {              System.err.println("Falha ao deserializar! - " + ex.toString());          }          System.out.println(fruta.getNome() + " - " + fruta.getCor());      }  } |

Na Listagem 5, encontra-se um exemplo de serialização e deserialização de um ArrayList da classe Fruta. É apenas mais um exemplo que demonstra a capacidade de se persistir qualquer objeto que seja serializável. Essa serialização só foi possível porque a classe “ArrayList” implementa a interface “Serializable” por padrão.

Listagem 5 - Serialização de ArrayList.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | //serialização e deserialização com ArrayList          ArrayList<Fruta> frutas = new ArrayList<Fruta>();          Fruta f1 = new Fruta("Laranja", "Amarela");          Fruta f2 = new Fruta("Abacate", "Verde");          Fruta f3 = new Fruta("Morango", "Vermelho");          frutas.add(f1);          frutas.add(f2);          frutas.add(f3);            try {              s.serializar("/home/gustavo/frutas", frutas);              frutas = null;              frutas = (ArrayList<Fruta>) d.deserializar("/home/gustavo/frutas");              for(Fruta f : frutas){                  System.out.println("ArrayList: " + f.getNome() + " - " + f.getCor());              }          } catch (Exception ex) {              System.err.println("Falha ao serializar ou deserializar! - " +                                 ex.toString());          } |

**4. PARTICULARIDADES DA SERIALIZAÇÃO**

Como dito, a serialização de um objeto em Java pode ser algo fácil de se fazer. Mas existem certas informações que são importantes e sabê-las pode evitar vários problemas.

Um dos problemas é que nem tudo pode ser serializado, como objetos das classes “Thread” e “Socket” e atributos estáticos. Os atributos estáticos estão vinculados à classe e não ao objeto. Assim quando se trata do estado do objeto, os campos estáticos não são considerados. Em relação às Threads, suas instâncias só são úteis enquanto estiverem em execução na JVM, e podem ser paradas/destruídas e iniciadas com instruções de sua classe. Não existe algo no estado destas instâncias que seja útil para se persistir, e por isso não há sentido em persistir tais instâncias para restaurar posteriormente.

Mas e se existir alguma classe em que seja necessário persisti-la, e esta possui uma instância da classe “Thread”? Neste caso, utilizando um recurso pode-se persistir a classe inteira, menos a instância thread. Este recurso, refere-se em indicar que tal instância thread é um objeto “Transient”. Na Listagem 6 encontra-se um exemplo da implementação com uso de “Transient”.

Listagem 6 - Uso do Transient

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | import java.io.Serializable;    public class ImplementacaoTransient implements Serializable, Runnable{        transient private Thread thread;      //marcado com "transient" para que sua instância não seja serializada        //Demais valores de estado serão serializados      private int valorParaPersistir;        public ImplementacaoTransient(int valorParaPersistir){          this.valorParaPersistir = valorParaPersistir;          thread = new Thread(this);          thread.start();      }        public void run()      {          while(true){              //código a ser executado pela Thread          }      }  } |

Vale ressaltar que, o que é serializado é apenas o estado do objeto, e não os métodos e a classe. No exemplo acima, apenas a instância do objeto “thread” e do objeto “valorParaPersistir” compunham o estado do objeto. Assim, evita-se que o sistema de serialização tente serializar um objeto não-serializável e gere erros. Após a deserialização de um objeto da classe “ImplementacaoTransient” este estará totalmente funcional.

Um outro problema que pode vir a ocorrer, se refere ao versionamento das classes. Toda classe possui um atributo oculto chamado “serialVersionUID” com um valor gerado automaticamente. Objetos criados levam consigo o “serialVersionUID” de sua classe, que indica sua versão. Toda vez que uma classe sofre alterações em seu código, um novo “serialVersionUID” é gerado. O problema com a serialização surge quando serializa-se o estado de um objeto e posteriormente tenta-se restaurar o estado para a aplicação, mas houveram mudanças no código da classe. Neste caso, uma exceção “java.io.InvalidClassException” é disparada, pois de fato, o estado do objeto corresponde a uma versão de classe do qual foi criado que já não existe mais. Portanto, isso só ocorre para evitar erros de programação, em que tenta-se restaurar o estado de um objeto para uma classe que possui versão diferente da qual o objeto foi serializado inicialmente. E isso pode significar diferentes versões de uma mesma classe “Fruta” por exemplo, ou diferentes versões por classes serem totalmente difere

|  |  |
| --- | --- |
| http://www.linhadecodigo.com.br/artigos/img_artigos/EvertonCoimbraAraujo/Serializazao_Objetos/imagem001.jpg | Neste artigo apresento o Gustavo Rafael Valiati, que pode ser contatado pelo e-mail gustavovaliati@gmail.com. O Gustavo foi meu aluno em algumas disciplinas de sua gradução e está concluíndo o último período do curso de Tecnologia em Análise e Desenvolvimento de Sistemas da UTFPR campus Medianeira, neste primeiro semestre de 2012. |

**1. INTRODUÇÃO**

Nem sempre se deseja ter objetos ou estrutura de dados de um aplicativo, com um ciclo de vida que dura no máximo o tempo em que for armazenado em memória RAM. Ou seja, algumas vezes deseja-se guardá-los por um tempo indeterminado, sendo possível recuperá-los posteriormente. E neste caso, o termo “persistência” tem seu conceito definido.

Persistir dados, requer que estes possam ser serializáveis. A serialização possibilitará capturar o estado do objeto ou a estrutura de dados, e transformar em uma cadeia de bytes quando for necessário. Torna-se possível também, recuperar os bytes persistidos para fazer o processo inverso, e ter os dados de volta para a aplicação em execução.

Desta maneira, qualquer forma de dados que esteja sendo trabalhada na aplicação, e esta puder ser serializada, poderá ser passada para o meio físico, e ter o processo revertido.

A serialização não se limita apenas em possibilitar a gravação do dado em disco, mas permite que o mesmo seja transmitido por rede. Isto porquê a serialização cria um stream de bytes, que é requisito para transmissão de dados por rede.

Diversas linguagens possuem suporte à serialização, como: Java, C, C++, Python, PHP, .NET, entre outras. Neste artigo, será tratado apenas da serialização com Java, em sua implemetação básica, alguns problemas e soluções, com códigos para exemplificação.

**2. SERIALIZAÇÃO COM JAVA**

Serializar um objeto, em Java, só é possível caso sua classe esteja marcada como serializável. Essa marcação é feita com a simples implementação da interface “java.io.Serializable”. Essa interface não possui métodos para se implementar e nem atributos. Ainda não existiam as anotações no Java 1.1, versão que trouxe pela primeira vez a serialização ao Java, e desta maneira a implementação da interface era a melhor maneira de marcar uma classe.

A herança entre classes é naturalmente afetada pela serialização. Já que quando uma classe implementa a interface “Serializable”, toda classe que dela estender também estará implicitamente marcada como serializável

.

Para exemplo, será criada uma classe “pojo.Fruta” que contém dois atributos: nome e cor. No código exemplo (Listagem 1), pode-se notar a classe “Fruta” marcada para a serialização:

Listagem 1 - Marcando uma classe como serializável.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | import java.io.Serializable;    public class Fruta implements Serializable{      private String nome;      private String cor;        public Fruta(String nome, String cor) {          this.nome = nome;          this.cor = cor;      }        public String getCor() { return cor; }      public String getNome() { return nome; }      public void setCor(String cor) { this.cor = cor; }      public void setNome(String nome) { this.nome = nome; }  } |

Pode-se dizer que a implementação do processo de serialização e deserialização simples de um objeto, não possui um código difícil de se escrever. Isto porquê duas classes fazem a maior parte do trabalho, de maneira que com poucas linhas de código pode-se transformar um objeto em bytes e vice-versa. Essas duas classes são “java.io.ObjectInputStream” que recebe bytes para criar um objeto e “java.io.ObjectOutputStream” que cria uma cadeia de bytes a partir de um objeto.

**3. SERIALIZANDO E DESERIALIZANDO O ESTADO DE UM OBJETO**

Como exemplo, será persistido um objeto em disco. Para isto, além das classes “java.io.ObjectInputStream” e “java.io.ObjectOutputStream”, serão utilizadas duas outras que trabalham com arquivos, que são “java.io.FileInputStream” e “java.io.FileOutputStream”, que seguem a mesma lógica de InputStream e OutputStream, só que agora relacionado ao sistema de arquivos e não aos objetos.

Para serialização, deve-se criar um novo “java.io.FileOutputStream”, com o caminho (“path”) do arquivo onde se deseja armazenar o estado do objeto. Também deve ser criado um novo “java.io.ObjectOutputStream” que está ligado ao arquivo. E por último deve-se executar a escrita do objeto alvo, para dentro do objeto de stream que por sua vez está ligado ao arquivo destino. Após executados esses passos, é possível encontrar no sistema de arquivos um novo arquivo que contém os bytes correspondentes à descrição do estado do objeto. Na Listagem 2, encontra-se o código exemplo.

Listagem 2 - Serialização de um objeto.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | import java.io.FileOutputStream;  import java.io.ObjectOutputStream;    public class Serializador {        public Serializador() {    }        public void serializar(String path, Object obj) throws Exception {              FileOutputStream outFile = new FileOutputStream(path);      ObjectOutputStream s = new ObjectOutputStream(outFile);      s.writeObject(obj);      s.close();      }  } |

E para a deserialização, o processo é extremamente parecido com a serialização. Cria-se um novo “java.io.FileInputStream” relacionado ao arquivo onde o estado do objeto encontra-se persistido. Deve-se criar também um novo “java.io.ObjectInputStream” que estará vinculado ao arquivo de origem. Por último, executa-se a leitura do arquivo para a restauração do estado. Na Listagem 3 encontra-se o exemplo do código.

Listagem 3 - Deserializaçãode um objeto.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | import java.io.FileInputStream;  import java.io.ObjectInputStream;    public class Deserializador {        public Deserializador() {    }        public Object deserializar(String path) throws Exception {      FileInputStream inFile = new FileInputStream(path);      ObjectInputStream d = new ObjectInputStream(inFile);      Object o = d.readObject();      d.close();      return o;      }  } |

A Listagem 4, contém o exemplo da execução da serialização e em seguida da deserialização.

Listagem 4 - Serializando e Deserializando

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | import operacoes.Deserializador;  import operacoes.Serializador;  import pojo.Fruta;    public class Principal {      public static void main(String args[]){         //serializa         Serializador s = new Serializador();         Fruta fruta = new Fruta("Maça", "Vermelha");          try {              s.serializar("/home/gustavo/fruta", fruta);          } catch (Exception ex) {              System.err.println("Falha ao serializar! - " + ex.toString());          }            //deserializa          Deserializador d = new Deserializador();          fruta = null;          try {              fruta = (Fruta) d.deserializar("/home/gustavo/fruta");          } catch (Exception ex) {              System.err.println("Falha ao deserializar! - " + ex.toString());          }          System.out.println(fruta.getNome() + " - " + fruta.getCor());      }  } |

Na Listagem 5, encontra-se um exemplo de serialização e deserialização de um ArrayList da classe Fruta. É apenas mais um exemplo que demonstra a capacidade de se persistir qualquer objeto que seja serializável. Essa serialização só foi possível porque a classe “ArrayList” implementa a interface “Serializable” por padrão.

Listagem 5 - Serialização de ArrayList.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | //serialização e deserialização com ArrayList          ArrayList<Fruta> frutas = new ArrayList<Fruta>();          Fruta f1 = new Fruta("Laranja", "Amarela");          Fruta f2 = new Fruta("Abacate", "Verde");          Fruta f3 = new Fruta("Morango", "Vermelho");          frutas.add(f1);          frutas.add(f2);          frutas.add(f3);            try {              s.serializar("/home/gustavo/frutas", frutas);              frutas = null;              frutas = (ArrayList<Fruta>) d.deserializar("/home/gustavo/frutas");              for(Fruta f : frutas){                  System.out.println("ArrayList: " + f.getNome() + " - " + f.getCor());              }          } catch (Exception ex) {              System.err.println("Falha ao serializar ou deserializar! - " +                                 ex.toString());          } |

**4. PARTICULARIDADES DA SERIALIZAÇÃO**

Como dito, a serialização de um objeto em Java pode ser algo fácil de se fazer. Mas existem certas informações que são importantes e sabê-las pode evitar vários problemas.

Um dos problemas é que nem tudo pode ser serializado, como objetos das classes “Thread” e “Socket” e atributos estáticos. Os atributos estáticos estão vinculados à classe e não ao objeto. Assim quando se trata do estado do objeto, os campos estáticos não são considerados. Em relação às Threads, suas instâncias só são úteis enquanto estiverem em execução na JVM, e podem ser paradas/destruídas e iniciadas com instruções de sua classe. Não existe algo no estado destas instâncias que seja útil para se persistir, e por isso não há sentido em persistir tais instâncias para restaurar posteriormente.

Mas e se existir alguma classe em que seja necessário persisti-la, e esta possui uma instância da classe “Thread”? Neste caso, utilizando um recurso pode-se persistir a classe inteira, menos a instância thread. Este recurso, refere-se em indicar que tal instância thread é um objeto “Transient”. Na Listagem 6 encontra-se um exemplo da implementação com uso de “Transient”.

Listagem 6 - Uso do Transient

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | import java.io.Serializable;    public class ImplementacaoTransient implements Serializable, Runnable{        transient private Thread thread;      //marcado com "transient" para que sua instância não seja serializada        //Demais valores de estado serão serializados      private int valorParaPersistir;        public ImplementacaoTransient(int valorParaPersistir){          this.valorParaPersistir = valorParaPersistir;          thread = new Thread(this);          thread.start();      }        public void run()      {          while(true){              //código a ser executado pela Thread          }      }  } |

Vale ressaltar que, o que é serializado é apenas o estado do objeto, e não os métodos e a classe. No exemplo acima, apenas a instância do objeto “thread” e do objeto “valorParaPersistir” compunham o estado do objeto. Assim, evita-se que o sistema de serialização tente serializar um objeto não-serializável e gere erros. Após a deserialização de um objeto da classe “ImplementacaoTransient” este estará totalmente funcional.

Um outro problema que pode vir a ocorrer, se refere ao versionamento das classes. Toda classe possui um atributo oculto chamado “serialVersionUID” com um valor gerado automaticamente. Objetos criados levam consigo o “serialVersionUID” de sua classe, que indica sua versão. Toda vez que uma classe sofre alterações em seu código, um novo “serialVersionUID” é gerado. O problema com a serialização surge quando serializa-se o estado de um objeto e posteriormente tenta-se restaurar o estado para a aplicação, mas houveram mudanças no código da classe. Neste caso, uma exceção “java.io.InvalidClassException” é disparada, pois de fato, o estado do objeto corresponde a uma versão de classe do qual foi criado que já não existe mais. Portanto, isso só ocorre para evitar erros de programação, em que tenta-se restaurar o estado de um objeto para uma classe que possui versão diferente da qual o objeto foi serializado inicialmente. E isso pode significar diferentes versões de uma mesma classe “Fruta” por exemplo, ou diferentes versões por classes serem totalmente difere

|  |  |
| --- | --- |
| 1 | serialver pojo.Fruta |

A serialização acaba inflingindo alguns modelos de segurança em seu procedimento padrão, e criando então um novo problema. Quando um estado de objeto é persistido no sistema de arquivos ou enviado por rede, o fluxo binário pode ser detectado, analisado e obtidas informações sobre a classe e o estado do objeto. Por exemplo, se utilizado um editor de texto qualquer para abrir o arquivo persistido do estado do objeto da classe “Fruta” dos exemplos anteriores, é comum os atributos e outras informações aparecerem como um simples texto, total ou parcilamente legíveis, assim como demonstrado na Listagem 9.

Listagem 9 - Exemplo de leitura de arquivo binário de um estado de objeto persistido.

|  |  |
| --- | --- |
| 1  2 | ��##sr#  pojo.Fruta#######{###L##cort##Ljava/lang/String;L##nomeq#~##xpt##Vermelhat##Maça |

Com uma ferramenta menos simples seria possível aumentar a quantidade de informações legíveis, coletando uma maior quantia de dados.

No entando, o processo de serialização permite que haja uma certa personalização da serialização e deserialização de um objeto. Isso deve-se a possibilidade da sobrescrição dos métodos “writeObject” e “readObject” dentro da classe a ser serializada, como a classe exemplar “pojo.Fruta”. Esta sobrescrita permite um controle total sobre a serialização, pois as classes “java.io.ObjectOutputStream” e “java.io.ObjectInputStream”, ao serializar o estado de um objeto, primeiramente verificam se esse método foi sobrescrito na classe alvo e assim o utilizam, senão, será utilizado o protocolo padrão de serialização.

Realizando a sobrescrição dos métodos é possível criptografar os dados na serialização e descriptografar os dados na deserialização, por meio de algum algoritmo que o programador venha implementar. Assim os dados podem ser persistidos ou transmitidos por rede, sem que seja possível ler as partes mais importantes do estado de um objeto. Na listagem 10, a classe “pojo.Fruta” foi modificada dando origem a classe “pojo.FrutaSobrescrita” que possui adicionamente a sobrescrição dos métodos “writeObject” e “readObject”. Como este é apenas um exemplo, não foi escrito um verdadeiro algoritmo de criptografia, mas sim apenas uma troca no valor da String do campo “cor”, identificando que houveram alterações tanto no processo de serialização quanto no de deserialização. Na listagem 11 encontra-se a execução da serialização.

Listagem 10 - Sobrescrita dos métodos “writeObject” e “readObject”.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36 | import java.io.Serializable;    public class FrutaSobrescrita implements Serializable{      private String nome;      private String cor;      static final long serialVersionUID = 123L; //indica a versao da classe        public FrutaSobrescrita(String nome, String cor) {          this.nome = nome;          this.cor = cor;      }      public String getCor() { return cor; }      public String getNome() { return nome; }      public void setCor(String cor) { this.cor = cor; }      public void setNome(String nome) { this.nome = nome; }        private void writeObject(java.io.ObjectOutputStream stream) throws java.io.IOException      {          // criptografar os dados antes de liberar o stream          // EXEMPLO: setCor(criptografar(getCor()));          setCor(getCor()+ " > CRIPTOGRAFADO"); // apenas concatenação de string para indicar alteração            // liberar o stream          stream.defaultWriteObject();      }        private void readObject(java.io.ObjectInputStream stream) throws java.io.IOException, ClassNotFoundException      {          // liberar stream          stream.defaultReadObject();            // descriptografar os dados          // EXEMPLO: setCor(descriptografar(getCor()));          setCor(getCor()+ " > DESCRIPTOGRAFADO"); // apenas contatenação de string para indicar alteração      }  } |

Listagem 11 - Execução da serialização da classe “pojo.FrutaSobrescrita”.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | //serializa        Serializador s = new Serializador();        FrutaSobrescrita frutaSobrescrita = new FrutaSobrescrita("Maça", "Vermelha");         try {             s.serializar("/home/gustavo/frutaSobrescrita", frutaSobrescrita);         } catch (Exception ex) {             System.err.println("Falha ao serializar! - " + ex.toString());         }           //deserializa        Deserializador d = new Deserializador();         frutaSobrescrita = null;         try {             frutaSobrescrita = (FrutaSobrescrita) d.deserializar("/home/gustavo/frutaSobrescrita");         } catch (Exception ex) {             System.err.println("Falha ao deserializar! - " + ex.toString());         }         System.out.println("Sobrescrita: " + frutaSobrescrita.getNome() + " - " + frutaSobrescrita.getCor()); |

**5. CONCLUSÃO**

A serialização é um procedimento de simples execução por parte do programador. Com pouco tempo de codificação é possível implementar a persistencia de um estado de objeto para o sistema de arquivos ou de transmissão por rede. É possível reproduzir os exemplos criados neste artigo, em situações até mesmo mais elaboradas. A apresentação de alguns problemas e soluções trazidas por particularidades, abre o entendimento de que a serialização pode deixar de ser simples conforme for aplicada em diferentes contextos, e que ao mesmo tempo é muito flexível ao permitir ao programador implementar diferentes soluções.

[Assista ao vídeo Serialização/Deserialização de objetos com YAML](http://www.devmedia.com.br/serializacao-deserializacao-de-objetos-com-yaml-parte-1/8399)

**6. BIBLIOGRAFIA**

ORACLE - **java.io Interface Serializable** - Disponível em: <http://docs.oracle.com/javase/1.4.2/docs/api/java/io/Serializable.html>Acesso em: 11/03/2012

ORACLE - **Discover de Secrets of Java Serialization API** - Disponível em: <http://java.sun.com/developer/technicalArticles/Programming/serialization/> Acesso em: 11/03/2012

NEWARD, Ted - **5 coisas que você não sabia sobre... Serialização de Objetos Java** - Disponível em: <http://www.ibm.com/developerworks/br/library/j-5things1/index.html> Acesso em: 11/03/2012

DOWNS, Andrew - **Java Serialization** - Disponível em: <http://www.mactech.com/articles/mactech/Vol.14/14.04/JavaSerialization/index.html> Acesso em: 11/03/2012

PREISSLER, Luciano E.; GEYER, Cláudio F. R. - **Serialização em Java** - Disponível em: <http://www.inf.ufrgs.br/gppd/disc/cmp167/trabalhos/sem99-1/T1/luciano/final.htm> Acesso em: 11/03/2012

[[](http://www.linhadecodigo.com.br/colaborador.aspx?id=354)](http://www.linhadecodigo.com.br/colaborador.aspx?id=354)

[Everton Coimbra de Araújo - Desde 1987 atua na área de treinamento e desenvolvimento. Como Mestre em Ciência da Computação, é professor da UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ, Campus Medianeira, onde leciona disciplinas relacionadas ao desenvolvimento de aplicações web, com Java e .NET.   
  
É autor dos livros Desenvolvimento para WEB com Java, Orientação a Objetos com Java - Simples, Fácil e Eficiente, Algoritmo - Fundamento e Prática em sua terceira edição e dos livros Delphi - Implementação e Técnicas para Ambientes Virtuais e C++ Builder - Implementação e Técnicas para Ambientes Virtuais. Todos pela VisualBooks. Pode ser contactado através do e-mail everton@utfpr.edu.br ou evertoncoimbra@gmail.com.](http://www.linhadecodigo.com.br/colaborador.aspx?id=354)

 85 3 0 4

**Leia também**

* [Conhecendo a interface Map do Java](http://www.linhadecodigo.com.br/artigo/3670/conhecendo-a-interface-map-do-java.aspx)*Java*
* [Trabalhando com a Interface Set no Java](http://www.linhadecodigo.com.br/artigo/3669/trabalhando-com-a-interface-set-no-java.aspx)*Java*
* [Classes Wrappers em Java](http://www.linhadecodigo.com.br/artigo/3667/classes-wrappers-em-java.aspx)*Java*
* [ComboBoxModel em Java](http://www.linhadecodigo.com.br/artigo/3664/comboboxmodel-em-java.aspx)*Java*
* [Reduzindo a Mutabilidade em Java](http://www.linhadecodigo.com.br/artigo/3659/reduzindo-a-mutabilidade-em-java.aspx)*Java*

Publicidade

[](http://qconsp.com/?utm_source=devmedia&utm_medium=banner&utm_campaign=devmedia)

Linha de Código[Anuncie](http://www.web-03.net/linhadecodigo.asp)|[Fale Conosco](mailto:colabore@linhadecodigo.com.br)|[Publique](http://www.linhadecodigo.com.br/publique.aspx)

Copyright 2016 - todos os direitos reservados para [**Web03**](http://www.web-03.net/)

|  |  |
| --- | --- |
| http://www.linhadecodigo.com.br/artigos/img_artigos/EvertonCoimbraAraujo/Serializazao_Objetos/imagem001.jpg | Neste artigo apresento o Gustavo Rafael Valiati, que pode ser contatado pelo e-mail gustavovaliati@gmail.com. O Gustavo foi meu aluno em algumas disciplinas de sua gradução e está concluíndo o último período do curso de Tecnologia em Análise e Desenvolvimento de Sistemas da UTFPR campus Medianeira, neste primeiro semestre de 2012. |

**1. INTRODUÇÃO**

Nem sempre se deseja ter objetos ou estrutura de dados de um aplicativo, com um ciclo de vida que dura no máximo o tempo em que for armazenado em memória RAM. Ou seja, algumas vezes deseja-se guardá-los por um tempo indeterminado, sendo possível recuperá-los posteriormente. E neste caso, o termo “persistência” tem seu conceito definido.

Persistir dados, requer que estes possam ser serializáveis. A serialização possibilitará capturar o estado do objeto ou a estrutura de dados, e transformar em uma cadeia de bytes quando for necessário. Torna-se possível também, recuperar os bytes persistidos para fazer o processo inverso, e ter os dados de volta para a aplicação em execução.

Desta maneira, qualquer forma de dados que esteja sendo trabalhada na aplicação, e esta puder ser serializada, poderá ser passada para o meio físico, e ter o processo revertido.

A serialização não se limita apenas em possibilitar a gravação do dado em disco, mas permite que o mesmo seja transmitido por rede. Isto porquê a serialização cria um stream de bytes, que é requisito para transmissão de dados por rede.

Diversas linguagens possuem suporte à serialização, como: Java, C, C++, Python, PHP, .NET, entre outras. Neste artigo, será tratado apenas da serialização com Java, em sua implemetação básica, alguns problemas e soluções, com códigos para exemplificação.

**2. SERIALIZAÇÃO COM JAVA**

Serializar um objeto, em Java, só é possível caso sua classe esteja marcada como serializável. Essa marcação é feita com a simples implementação da interface “java.io.Serializable”. Essa interface não possui métodos para se implementar e nem atributos. Ainda não existiam as anotações no Java 1.1, versão que trouxe pela primeira vez a serialização ao Java, e desta maneira a implementação da interface era a melhor maneira de marcar uma classe.

A herança entre classes é naturalmente afetada pela serialização. Já que quando uma classe implementa a interface “Serializable”, toda classe que dela estender também estará implicitamente marcada como serializável

.

Para exemplo, será criada uma classe “pojo.Fruta” que contém dois atributos: nome e cor. No código exemplo (Listagem 1), pode-se notar a classe “Fruta” marcada para a serialização:

Listagem 1 - Marcando uma classe como serializável.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | import java.io.Serializable;    public class Fruta implements Serializable{      private String nome;      private String cor;        public Fruta(String nome, String cor) {          this.nome = nome;          this.cor = cor;      }        public String getCor() { return cor; }      public String getNome() { return nome; }      public void setCor(String cor) { this.cor = cor; }      public void setNome(String nome) { this.nome = nome; }  } |

Pode-se dizer que a implementação do processo de serialização e deserialização simples de um objeto, não possui um código difícil de se escrever. Isto porquê duas classes fazem a maior parte do trabalho, de maneira que com poucas linhas de código pode-se transformar um objeto em bytes e vice-versa. Essas duas classes são “java.io.ObjectInputStream” que recebe bytes para criar um objeto e “java.io.ObjectOutputStream” que cria uma cadeia de bytes a partir de um objeto.

**3. SERIALIZANDO E DESERIALIZANDO O ESTADO DE UM OBJETO**

Como exemplo, será persistido um objeto em disco. Para isto, além das classes “java.io.ObjectInputStream” e “java.io.ObjectOutputStream”, serão utilizadas duas outras que trabalham com arquivos, que são “java.io.FileInputStream” e “java.io.FileOutputStream”, que seguem a mesma lógica de InputStream e OutputStream, só que agora relacionado ao sistema de arquivos e não aos objetos.

Para serialização, deve-se criar um novo “java.io.FileOutputStream”, com o caminho (“path”) do arquivo onde se deseja armazenar o estado do objeto. Também deve ser criado um novo “java.io.ObjectOutputStream” que está ligado ao arquivo. E por último deve-se executar a escrita do objeto alvo, para dentro do objeto de stream que por sua vez está ligado ao arquivo destino. Após executados esses passos, é possível encontrar no sistema de arquivos um novo arquivo que contém os bytes correspondentes à descrição do estado do objeto. Na Listagem 2, encontra-se o código exemplo.

Listagem 2 - Serialização de um objeto.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | import java.io.FileOutputStream;  import java.io.ObjectOutputStream;    public class Serializador {        public Serializador() {    }        public void serializar(String path, Object obj) throws Exception {              FileOutputStream outFile = new FileOutputStream(path);      ObjectOutputStream s = new ObjectOutputStream(outFile);      s.writeObject(obj);      s.close();      }  } |

E para a deserialização, o processo é extremamente parecido com a serialização. Cria-se um novo “java.io.FileInputStream” relacionado ao arquivo onde o estado do objeto encontra-se persistido. Deve-se criar também um novo “java.io.ObjectInputStream” que estará vinculado ao arquivo de origem. Por último, executa-se a leitura do arquivo para a restauração do estado. Na Listagem 3 encontra-se o exemplo do código.

Listagem 3 - Deserializaçãode um objeto.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | import java.io.FileInputStream;  import java.io.ObjectInputStream;    public class Deserializador {        public Deserializador() {    }        public Object deserializar(String path) throws Exception {      FileInputStream inFile = new FileInputStream(path);      ObjectInputStream d = new ObjectInputStream(inFile);      Object o = d.readObject();      d.close();      return o;      }  } |

A Listagem 4, contém o exemplo da execução da serialização e em seguida da deserialização.

Listagem 4 - Serializando e Deserializando

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | import operacoes.Deserializador;  import operacoes.Serializador;  import pojo.Fruta;    public class Principal {      public static void main(String args[]){         //serializa         Serializador s = new Serializador();         Fruta fruta = new Fruta("Maça", "Vermelha");          try {              s.serializar("/home/gustavo/fruta", fruta);          } catch (Exception ex) {              System.err.println("Falha ao serializar! - " + ex.toString());          }            //deserializa          Deserializador d = new Deserializador();          fruta = null;          try {              fruta = (Fruta) d.deserializar("/home/gustavo/fruta");          } catch (Exception ex) {              System.err.println("Falha ao deserializar! - " + ex.toString());          }          System.out.println(fruta.getNome() + " - " + fruta.getCor());      }  } |

Na Listagem 5, encontra-se um exemplo de serialização e deserialização de um ArrayList da classe Fruta. É apenas mais um exemplo que demonstra a capacidade de se persistir qualquer objeto que seja serializável. Essa serialização só foi possível porque a classe “ArrayList” implementa a interface “Serializable” por padrão.

Listagem 5 - Serialização de ArrayList.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | //serialização e deserialização com ArrayList          ArrayList<Fruta> frutas = new ArrayList<Fruta>();          Fruta f1 = new Fruta("Laranja", "Amarela");          Fruta f2 = new Fruta("Abacate", "Verde");          Fruta f3 = new Fruta("Morango", "Vermelho");          frutas.add(f1);          frutas.add(f2);          frutas.add(f3);            try {              s.serializar("/home/gustavo/frutas", frutas);              frutas = null;              frutas = (ArrayList<Fruta>) d.deserializar("/home/gustavo/frutas");              for(Fruta f : frutas){                  System.out.println("ArrayList: " + f.getNome() + " - " + f.getCor());              }          } catch (Exception ex) {              System.err.println("Falha ao serializar ou deserializar! - " +                                 ex.toString());          } |

**4. PARTICULARIDADES DA SERIALIZAÇÃO**

Como dito, a serialização de um objeto em Java pode ser algo fácil de se fazer. Mas existem certas informações que são importantes e sabê-las pode evitar vários problemas.

Um dos problemas é que nem tudo pode ser serializado, como objetos das classes “Thread” e “Socket” e atributos estáticos. Os atributos estáticos estão vinculados à classe e não ao objeto. Assim quando se trata do estado do objeto, os campos estáticos não são considerados. Em relação às Threads, suas instâncias só são úteis enquanto estiverem em execução na JVM, e podem ser paradas/destruídas e iniciadas com instruções de sua classe. Não existe algo no estado destas instâncias que seja útil para se persistir, e por isso não há sentido em persistir tais instâncias para restaurar posteriormente.

Mas e se existir alguma classe em que seja necessário persisti-la, e esta possui uma instância da classe “Thread”? Neste caso, utilizando um recurso pode-se persistir a classe inteira, menos a instância thread. Este recurso, refere-se em indicar que tal instância thread é um objeto “Transient”. Na Listagem 6 encontra-se um exemplo da implementação com uso de “Transient”.

Listagem 6 - Uso do Transient

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | import java.io.Serializable;    public class ImplementacaoTransient implements Serializable, Runnable{        transient private Thread thread;      //marcado com "transient" para que sua instância não seja serializada        //Demais valores de estado serão serializados      private int valorParaPersistir;        public ImplementacaoTransient(int valorParaPersistir){          this.valorParaPersistir = valorParaPersistir;          thread = new Thread(this);          thread.start();      }        public void run()      {          while(true){              //código a ser executado pela Thread          }      }  } |

Vale ressaltar que, o que é serializado é apenas o estado do objeto, e não os métodos e a classe. No exemplo acima, apenas a instância do objeto “thread” e do objeto “valorParaPersistir” compunham o estado do objeto. Assim, evita-se que o sistema de serialização tente serializar um objeto não-serializável e gere erros. Após a deserialização de um objeto da classe “ImplementacaoTransient” este estará totalmente funcional.

Um outro problema que pode vir a ocorrer, se refere ao versionamento das classes. Toda classe possui um atributo oculto chamado “serialVersionUID” com um valor gerado automaticamente. Objetos criados levam consigo o “serialVersionUID” de sua classe, que indica sua versão. Toda vez que uma classe sofre alterações em seu código, um novo “serialVersionUID” é gerado. O problema com a serialização surge quando serializa-se o estado de um objeto e posteriormente tenta-se restaurar o estado para a aplicação, mas houveram mudanças no código da classe. Neste caso, uma exceção “java.io.InvalidClassException” é disparada, pois de fato, o estado do objeto corresponde a uma versão de classe do qual foi criado que já não existe mais. Portanto, isso só ocorre para evitar erros de programação, em que tenta-se restaurar o estado de um objeto para uma classe que possui versão diferente da qual o objeto foi serializado inicialmente. E isso pode significar diferentes versões de uma mesma classe “Fruta” por exemplo, ou diferentes versões por classes serem totalmente diferentes como: o estado de um objeto da classe “Fruta” ser restaurado para classe “Carro”.

Mas quando é necessário, conscientemente, recuperar o estado de um objeto, mesmo que houveram alterações na classe do qual foi gerado, é possível controlar manualmente o “serialVersionUID”. Para isto, basta declarar explicitamente na forma de atributo estático e final, um valor Long qualquer para corresponder à versão da classe. E assim, mesmo que houverem modificações na classe, a versão permenecerá a mesma, evitando as exceções da incompatiblidade de classes. No entanto, vale ressaltar que o versionamento não é um impecílio para o programador, mas sim muito importante em casos que deseja-se manter um controle sobre a compatibilidade de código na aplicação, definindo versões para classes No código da Listagem 7 encontra-se o exemplo do controle manual do “serialVersionUID”.

Listagem 7 - Uso manual do serialVersionUID

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | import java.io.Serializable;    public class Fruta implements Serializable{      private String nome;      private String cor;      static final long serialVersionUID = 123L; //indica a versao da classe        public Fruta(String nome, String cor) {          this.nome = nome;          this.cor = cor;      }      public String getCor() { return cor; }      public String getNome() { return nome; }      public void setCor(String cor) { this.cor = cor; }      public void setNome(String nome) { this.nome = nome; }  } |

O JDK fornece uma ferramenta que permite visualizar a versão de qualquer classe Java. Basta no terminal do sistema operacional utilizar o comando “serialver” seguido da classe qualificada, conforme exemplo da Listagem 8.

Listagem 8 - Verificação da versão de classe

|  |  |
| --- | --- |
| 1 | serialver pojo.Fruta |

A serialização acaba inflingindo alguns modelos de segurança em seu procedimento padrão, e criando então um novo problema. Quando um estado de objeto é persistido no sistema de arquivos ou enviado por rede, o fluxo binário pode ser detectado, analisado e obtidas informações sobre a classe e o estado do objeto. Por exemplo, se utilizado um editor de texto qualquer para abrir o arquivo persistido do estado do objeto da classe “Fruta” dos exemplos anteriores, é comum os atributos e outras informações aparecerem como um simples texto, total ou parcilamente legíveis, assim como demonstrado na Listagem 9.

Listagem 9 - Exemplo de leitura de arquivo binário de um estado de objeto persistido.

|  |  |
| --- | --- |
| 1  2 | ��##sr#  pojo.Fruta#######{###L##cort##Ljava/lang/String;L##nomeq#~##xpt##Vermelhat##Maça |

Com uma ferramenta menos simples seria possível aumentar a quantidade de informações legíveis, coletando uma maior quantia de dados.

No entando, o processo de serialização permite que haja uma certa personalização da serialização e deserialização de um objeto. Isso deve-se a possibilidade da sobrescrição dos métodos “writeObject” e “readObject” dentro da classe a ser serializada, como a classe exemplar “pojo.Fruta”. Esta sobrescrita permite um controle total sobre a serialização, pois as classes “java.io.ObjectOutputStream” e “java.io.ObjectInputStream”, ao serializar o estado de um objeto, primeiramente verificam se esse método foi sobrescrito na classe alvo e assim o utilizam, senão, será utilizado o protocolo padrão de serialização.

Realizando a sobrescrição dos métodos é possível criptografar os dados na serialização e descriptografar os dados na deserialização, por meio de algum algoritmo que o programador venha implementar. Assim os dados podem ser persistidos ou transmitidos por rede, sem que seja possível ler as partes mais importantes do estado de um objeto. Na listagem 10, a classe “pojo.Fruta” foi modificada dando origem a classe “pojo.FrutaSobrescrita” que possui adicionamente a sobrescrição dos métodos “writeObject” e “readObject”. Como este é apenas um exemplo, não foi escrito um verdadeiro algoritmo de criptografia, mas sim apenas uma troca no valor da String do campo “cor”, identificando que houveram alterações tanto no processo de serialização quanto no de deserialização. Na listagem 11 encontra-se a execução da serialização.

Listagem 10 - Sobrescrita dos métodos “writeObject” e “readObject”.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36 | import java.io.Serializable;    public class FrutaSobrescrita implements Serializable{      private String nome;      private String cor;      static final long serialVersionUID = 123L; //indica a versao da classe        public FrutaSobrescrita(String nome, String cor) {          this.nome = nome;          this.cor = cor;      }      public String getCor() { return cor; }      public String getNome() { return nome; }      public void setCor(String cor) { this.cor = cor; }      public void setNome(String nome) { this.nome = nome; }        private void writeObject(java.io.ObjectOutputStream stream) throws java.io.IOException      {          // criptografar os dados antes de liberar o stream          // EXEMPLO: setCor(criptografar(getCor()));          setCor(getCor()+ " > CRIPTOGRAFADO"); // apenas concatenação de string para indicar alteração            // liberar o stream          stream.defaultWriteObject();      }        private void readObject(java.io.ObjectInputStream stream) throws java.io.IOException, ClassNotFoundException      {          // liberar stream          stream.defaultReadObject();            // descriptografar os dados          // EXEMPLO: setCor(descriptografar(getCor()));          setCor(getCor()+ " > DESCRIPTOGRAFADO"); // apenas contatenação de string para indicar alteração      }  } |

Listagem 11 - Execução da serialização da classe “pojo.FrutaSobrescrita”.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | //serializa        Serializador s = new Serializador();        FrutaSobrescrita frutaSobrescrita = new FrutaSobrescrita("Maça", "Vermelha");         try {             s.serializar("/home/gustavo/frutaSobrescrita", frutaSobrescrita);         } catch (Exception ex) {             System.err.println("Falha ao serializar! - " + ex.toString());         }           //deserializa        Deserializador d = new Deserializador();         frutaSobrescrita = null;         try {             frutaSobrescrita = (FrutaSobrescrita) d.deserializar("/home/gustavo/frutaSobrescrita");         } catch (Exception ex) {             System.err.println("Falha ao deserializar! - " + ex.toString());         }         System.out.println("Sobrescrita: " + frutaSobrescrita.getNome() + " - " + frutaSobrescrita.getCor()); |

**5. CONCLUSÃO**

A serialização é um procedimento de simples execução por parte do programador. Com pouco tempo de codificação é possível implementar a persistencia de um estado de objeto para o sistema de arquivos ou de transmissão por rede. É possível reproduzir os exemplos criados neste artigo, em situações até mesmo mais elaboradas. A apresentação de alguns problemas e soluções trazidas por particularidades, abre o entendimento de que a serialização pode deixar de ser simples conforme for aplicada em diferentes contextos, e que ao mesmo tempo é muito flexível ao permitir ao programador implementar diferentes soluções.

[Assista ao vídeo Serialização/Deserialização de objetos com YAML](http://www.devmedia.com.br/serializacao-deserializacao-de-objetos-com-yaml-parte-1/8399)

**6. BIBLIOGRAFIA**

ORACLE - **java.io Interface Serializable** - Disponível em: <http://docs.oracle.com/javase/1.4.2/docs/api/java/io/Serializable.html>Acesso em: 11/03/2012

ORACLE - **Discover de Secrets of Java Serialization API** - Disponível em: <http://java.sun.com/developer/technicalArticles/Programming/serialization/> Acesso em: 11/03/2012

NEWARD, Ted - **5 coisas que você não sabia sobre... Serialização de Objetos Java** - Disponível em: <http://www.ibm.com/developerworks/br/library/j-5things1/index.html> Acesso em: 11/03/2012

DOWNS, Andrew - **Java Serialization** - Disponível em: <http://www.mactech.com/articles/mactech/Vol.14/14.04/JavaSerialization/index.html> Acesso em: 11/03/2012

PREISSLER, Luciano E.; GEYER, Cláudio F. R. - **Serialização em Java** - Disponível em: <http://www.inf.ufrgs.br/gppd/disc/cmp167/trabalhos/sem99-1/T1/luciano/final.htm> Acesso em: 11/03/2012

[[](http://www.linhadecodigo.com.br/colaborador.aspx?id=354)](http://www.linhadecodigo.com.br/colaborador.aspx?id=354)

[Everton Coimbra de Araújo - Desde 1987 atua na área de treinamento e desenvolvimento. Como Mestre em Ciência da Computação, é professor da UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ, Campus Medianeira, onde leciona disciplinas relacionadas ao desenvolvimento de aplicações web, com Java e .NET.   
  
É autor dos livros Desenvolvimento para WEB com Java, Orientação a Objetos com Java - Simples, Fácil e Eficiente, Algoritmo - Fundamento e Prática em sua terceira edição e dos livros Delphi - Implementação e Técnicas para Ambientes Virtuais e C++ Builder - Implementação e Técnicas para Ambientes Virtuais. Todos pela VisualBooks. Pode ser contactado através do e-mail everton@utfpr.edu.br ou evertoncoimbra@gmail.com.](http://www.linhadecodigo.com.br/colaborador.aspx?id=354)

 85 3 0 4

**Leia também**

* [Conhecendo a interface Map do Java](http://www.linhadecodigo.com.br/artigo/3670/conhecendo-a-interface-map-do-java.aspx)*Java*
* [Trabalhando com a Interface Set no Java](http://www.linhadecodigo.com.br/artigo/3669/trabalhando-com-a-interface-set-no-java.aspx)*Java*
* [Classes Wrappers em Java](http://www.linhadecodigo.com.br/artigo/3667/classes-wrappers-em-java.aspx)*Java*
* [ComboBoxModel em Java](http://www.linhadecodigo.com.br/artigo/3664/comboboxmodel-em-java.aspx)*Java*
* [Reduzindo a Mutabilidade em Java](http://www.linhadecodigo.com.br/artigo/3659/reduzindo-a-mutabilidade-em-java.aspx)*Java*

Publicidade

[](http://qconsp.com/?utm_source=devmedia&utm_medium=banner&utm_campaign=devmedia)

Linha de Código[Anuncie](http://www.web-03.net/linhadecodigo.asp)|[Fale Conosco](mailto:colabore@linhadecodigo.com.br)|[Publique](http://www.linhadecodigo.com.br/publique.aspx)

|  |  |
| --- | --- |
| http://www.linhadecodigo.com.br/artigos/img_artigos/EvertonCoimbraAraujo/Serializazao_Objetos/imagem001.jpg | Neste artigo apresento o Gustavo Rafael Valiati, que pode ser contatado pelo e-mail gustavovaliati@gmail.com. O Gustavo foi meu aluno em algumas disciplinas de sua gradução e está concluíndo o último período do curso de Tecnologia em Análise e Desenvolvimento de Sistemas da UTFPR campus Medianeira, neste primeiro semestre de 2012. |

**1. INTRODUÇÃO**

Nem sempre se deseja ter objetos ou estrutura de dados de um aplicativo, com um ciclo de vida que dura no máximo o tempo em que for armazenado em memória RAM. Ou seja, algumas vezes deseja-se guardá-los por um tempo indeterminado, sendo possível recuperá-los posteriormente. E neste caso, o termo “persistência” tem seu conceito definido.

Persistir dados, requer que estes possam ser serializáveis. A serialização possibilitará capturar o estado do objeto ou a estrutura de dados, e transformar em uma cadeia de bytes quando for necessário. Torna-se possível também, recuperar os bytes persistidos para fazer o processo inverso, e ter os dados de volta para a aplicação em execução.

Desta maneira, qualquer forma de dados que esteja sendo trabalhada na aplicação, e esta puder ser serializada, poderá ser passada para o meio físico, e ter o processo revertido.

A serialização não se limita apenas em possibilitar a gravação do dado em disco, mas permite que o mesmo seja transmitido por rede. Isto porquê a serialização cria um stream de bytes, que é requisito para transmissão de dados por rede.

Diversas linguagens possuem suporte à serialização, como: Java, C, C++, Python, PHP, .NET, entre outras. Neste artigo, será tratado apenas da serialização com Java, em sua implemetação básica, alguns problemas e soluções, com códigos para exemplificação.

**2. SERIALIZAÇÃO COM JAVA**

Serializar um objeto, em Java, só é possível caso sua classe esteja marcada como serializável. Essa marcação é feita com a simples implementação da interface “java.io.Serializable”. Essa interface não possui métodos para se implementar e nem atributos. Ainda não existiam as anotações no Java 1.1, versão que trouxe pela primeira vez a serialização ao Java, e desta maneira a implementação da interface era a melhor maneira de marcar uma classe.

A herança entre classes é naturalmente afetada pela serialização. Já que quando uma classe implementa a interface “Serializable”, toda classe que dela estender também estará implicitamente marcada como serializável

.

Para exemplo, será criada uma classe “pojo.Fruta” que contém dois atributos: nome e cor. No código exemplo (Listagem 1), pode-se notar a classe “Fruta” marcada para a serialização:

Listagem 1 - Marcando uma classe como serializável.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | import java.io.Serializable;    public class Fruta implements Serializable{      private String nome;      private String cor;        public Fruta(String nome, String cor) {          this.nome = nome;          this.cor = cor;      }        public String getCor() { return cor; }      public String getNome() { return nome; }      public void setCor(String cor) { this.cor = cor; }      public void setNome(String nome) { this.nome = nome; }  } |

Pode-se dizer que a implementação do processo de serialização e deserialização simples de um objeto, não possui um código difícil de se escrever. Isto porquê duas classes fazem a maior parte do trabalho, de maneira que com poucas linhas de código pode-se transformar um objeto em bytes e vice-versa. Essas duas classes são “java.io.ObjectInputStream” que recebe bytes para criar um objeto e “java.io.ObjectOutputStream” que cria uma cadeia de bytes a partir de um objeto.

**3. SERIALIZANDO E DESERIALIZANDO O ESTADO DE UM OBJETO**

Como exemplo, será persistido um objeto em disco. Para isto, além das classes “java.io.ObjectInputStream” e “java.io.ObjectOutputStream”, serão utilizadas duas outras que trabalham com arquivos, que são “java.io.FileInputStream” e “java.io.FileOutputStream”, que seguem a mesma lógica de InputStream e OutputStream, só que agora relacionado ao sistema de arquivos e não aos objetos.

Para serialização, deve-se criar um novo “java.io.FileOutputStream”, com o caminho (“path”) do arquivo onde se deseja armazenar o estado do objeto. Também deve ser criado um novo “java.io.ObjectOutputStream” que está ligado ao arquivo. E por último deve-se executar a escrita do objeto alvo, para dentro do objeto de stream que por sua vez está ligado ao arquivo destino. Após executados esses passos, é possível encontrar no sistema de arquivos um novo arquivo que contém os bytes correspondentes à descrição do estado do objeto. Na Listagem 2, encontra-se o código exemplo.

Listagem 2 - Serialização de um objeto.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | import java.io.FileOutputStream;  import java.io.ObjectOutputStream;    public class Serializador {        public Serializador() {    }        public void serializar(String path, Object obj) throws Exception {              FileOutputStream outFile = new FileOutputStream(path);      ObjectOutputStream s = new ObjectOutputStream(outFile);      s.writeObject(obj);      s.close();      }  } |

E para a deserialização, o processo é extremamente parecido com a serialização. Cria-se um novo “java.io.FileInputStream” relacionado ao arquivo onde o estado do objeto encontra-se persistido. Deve-se criar também um novo “java.io.ObjectInputStream” que estará vinculado ao arquivo de origem. Por último, executa-se a leitura do arquivo para a restauração do estado. Na Listagem 3 encontra-se o exemplo do código.

Listagem 3 - Deserializaçãode um objeto.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | import java.io.FileInputStream;  import java.io.ObjectInputStream;    public class Deserializador {        public Deserializador() {    }        public Object deserializar(String path) throws Exception {      FileInputStream inFile = new FileInputStream(path);      ObjectInputStream d = new ObjectInputStream(inFile);      Object o = d.readObject();      d.close();      return o;      }  } |

A Listagem 4, contém o exemplo da execução da serialização e em seguida da deserialização.

Listagem 4 - Serializando e Deserializando

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | import operacoes.Deserializador;  import operacoes.Serializador;  import pojo.Fruta;    public class Principal {      public static void main(String args[]){         //serializa         Serializador s = new Serializador();         Fruta fruta = new Fruta("Maça", "Vermelha");          try {              s.serializar("/home/gustavo/fruta", fruta);          } catch (Exception ex) {              System.err.println("Falha ao serializar! - " + ex.toString());          }            //deserializa          Deserializador d = new Deserializador();          fruta = null;          try {              fruta = (Fruta) d.deserializar("/home/gustavo/fruta");          } catch (Exception ex) {              System.err.println("Falha ao deserializar! - " + ex.toString());          }          System.out.println(fruta.getNome() + " - " + fruta.getCor());      }  } |

Na Listagem 5, encontra-se um exemplo de serialização e deserialização de um ArrayList da classe Fruta. É apenas mais um exemplo que demonstra a capacidade de se persistir qualquer objeto que seja serializável. Essa serialização só foi possível porque a classe “ArrayList” implementa a interface “Serializable” por padrão.

Listagem 5 - Serialização de ArrayList.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | //serialização e deserialização com ArrayList          ArrayList<Fruta> frutas = new ArrayList<Fruta>();          Fruta f1 = new Fruta("Laranja", "Amarela");          Fruta f2 = new Fruta("Abacate", "Verde");          Fruta f3 = new Fruta("Morango", "Vermelho");          frutas.add(f1);          frutas.add(f2);          frutas.add(f3);            try {              s.serializar("/home/gustavo/frutas", frutas);              frutas = null;              frutas = (ArrayList<Fruta>) d.deserializar("/home/gustavo/frutas");              for(Fruta f : frutas){                  System.out.println("ArrayList: " + f.getNome() + " - " + f.getCor());              }          } catch (Exception ex) {              System.err.println("Falha ao serializar ou deserializar! - " +                                 ex.toString());          } |

**4. PARTICULARIDADES DA SERIALIZAÇÃO**

Como dito, a serialização de um objeto em Java pode ser algo fácil de se fazer. Mas existem certas informações que são importantes e sabê-las pode evitar vários problemas.

Um dos problemas é que nem tudo pode ser serializado, como objetos das classes “Thread” e “Socket” e atributos estáticos. Os atributos estáticos estão vinculados à classe e não ao objeto. Assim quando se trata do estado do objeto, os campos estáticos não são considerados. Em relação às Threads, suas instâncias só são úteis enquanto estiverem em execução na JVM, e podem ser paradas/destruídas e iniciadas com instruções de sua classe. Não existe algo no estado destas instâncias que seja útil para se persistir, e por isso não há sentido em persistir tais instâncias para restaurar posteriormente.

Mas e se existir alguma classe em que seja necessário persisti-la, e esta possui uma instância da classe “Thread”? Neste caso, utilizando um recurso pode-se persistir a classe inteira, menos a instância thread. Este recurso, refere-se em indicar que tal instância thread é um objeto “Transient”. Na Listagem 6 encontra-se um exemplo da implementação com uso de “Transient”.

Listagem 6 - Uso do Transient

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | import java.io.Serializable;    public class ImplementacaoTransient implements Serializable, Runnable{        transient private Thread thread;      //marcado com "transient" para que sua instância não seja serializada        //Demais valores de estado serão serializados      private int valorParaPersistir;        public ImplementacaoTransient(int valorParaPersistir){          this.valorParaPersistir = valorParaPersistir;          thread = new Thread(this);          thread.start();      }        public void run()      {          while(true){              //código a ser executado pela Thread          }      }  } |

Vale ressaltar que, o que é serializado é apenas o estado do objeto, e não os métodos e a classe. No exemplo acima, apenas a instância do objeto “thread” e do objeto “valorParaPersistir” compunham o estado do objeto. Assim, evita-se que o sistema de serialização tente serializar um objeto não-serializável e gere erros. Após a deserialização de um objeto da classe “ImplementacaoTransient” este estará totalmente funcional.

Um outro problema que pode vir a ocorrer, se refere ao versionamento das classes. Toda classe possui um atributo oculto chamado “serialVersionUID” com um valor gerado automaticamente. Objetos criados levam consigo o “serialVersionUID” de sua classe, que indica sua versão. Toda vez que uma classe sofre alterações em seu código, um novo “serialVersionUID” é gerado. O problema com a serialização surge quando serializa-se o estado de um objeto e posteriormente tenta-se restaurar o estado para a aplicação, mas houveram mudanças no código da classe. Neste caso, uma exceção “java.io.InvalidClassException” é disparada, pois de fato, o estado do objeto corresponde a uma versão de classe do qual foi criado que já não existe mais. Portanto, isso só ocorre para evitar erros de programação, em que tenta-se restaurar o estado de um objeto para uma classe que possui versão diferente da qual o objeto foi serializado inicialmente. E isso pode significar diferentes versões de uma mesma classe “Fruta” por exemplo, ou diferentes versões por classes serem totalmente diferentes como: o estado de um objeto da classe “Fruta” ser restaurado para classe “Carro”.

Mas quando é necessário, conscientemente, recuperar o estado de um objeto, mesmo que houveram alterações na classe do qual foi gerado, é possível controlar manualmente o “serialVersionUID”. Para isto, basta declarar explicitamente na forma de atributo estático e final, um valor Long qualquer para corresponder à versão da classe. E assim, mesmo que houverem modificações na classe, a versão permenecerá a mesma, evitando as exceções da incompatiblidade de classes. No entanto, vale ressaltar que o versionamento não é um impecílio para o programador, mas sim muito importante em casos que deseja-se manter um controle sobre a compatibilidade de código na aplicação, definindo versões para classes No código da Listagem 7 encontra-se o exemplo do controle manual do “serialVersionUID”.

Listagem 7 - Uso manual do serialVersionUID

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | import java.io.Serializable;    public class Fruta implements Serializable{      private String nome;      private String cor;      static final long serialVersionUID = 123L; //indica a versao da classe        public Fruta(String nome, String cor) {          this.nome = nome;          this.cor = cor;      }      public String getCor() { return cor; }      public String getNome() { return nome; }      public void setCor(String cor) { this.cor = cor; }      public void setNome(String nome) { this.nome = nome; }  } |

O JDK fornece uma ferramenta que permite visualizar a versão de qualquer classe Java. Basta no terminal do sistema operacional utilizar o comando “serialver” seguido da classe qualificada, conforme exemplo da Listagem 8.

Listagem 8 - Verificação da versão de classe

|  |  |
| --- | --- |
| 1 | serialver pojo.Fruta |

A serialização acaba inflingindo alguns modelos de segurança em seu procedimento padrão, e criando então um novo problema. Quando um estado de objeto é persistido no sistema de arquivos ou enviado por rede, o fluxo binário pode ser detectado, analisado e obtidas informações sobre a classe e o estado do objeto. Por exemplo, se utilizado um editor de texto qualquer para abrir o arquivo persistido do estado do objeto da classe “Fruta” dos exemplos anteriores, é comum os atributos e outras informações aparecerem como um simples texto, total ou parcilamente legíveis, assim como demonstrado na Listagem 9.

Listagem 9 - Exemplo de leitura de arquivo binário de um estado de objeto persistido.

|  |  |
| --- | --- |
| 1  2 | ��##sr#  pojo.Fruta#######{###L##cort##Ljava/lang/String;L##nomeq#~##xpt##Vermelhat##Maça |

Com uma ferramenta menos simples seria possível aumentar a quantidade de informações legíveis, coletando uma maior quantia de dados.

No entando, o processo de serialização permite que haja uma certa personalização da serialização e deserialização de um objeto. Isso deve-se a possibilidade da sobrescrição dos métodos “writeObject” e “readObject” dentro da classe a ser serializada, como a classe exemplar “pojo.Fruta”. Esta sobrescrita permite um controle total sobre a serialização, pois as classes “java.io.ObjectOutputStream” e “java.io.ObjectInputStream”, ao serializar o estado de um objeto, primeiramente verificam se esse método foi sobrescrito na classe alvo e assim o utilizam, senão, será utilizado o protocolo padrão de serialização.

Realizando a sobrescrição dos métodos é possível criptografar os dados na serialização e descriptografar os dados na deserialização, por meio de algum algoritmo que o programador venha implementar. Assim os dados podem ser persistidos ou transmitidos por rede, sem que seja possível ler as partes mais importantes do estado de um objeto. Na listagem 10, a classe “pojo.Fruta” foi modificada dando origem a classe “pojo.FrutaSobrescrita” que possui adicionamente a sobrescrição dos métodos “writeObject” e “readObject”. Como este é apenas um exemplo, não foi escrito um verdadeiro algoritmo de criptografia, mas sim apenas uma troca no valor da String do campo “cor”, identificando que houveram alterações tanto no processo de serialização quanto no de deserialização. Na listagem 11 encontra-se a execução da serialização.

Listagem 10 - Sobrescrita dos métodos “writeObject” e “readObject”.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36 | import java.io.Serializable;    public class FrutaSobrescrita implements Serializable{      private String nome;      private String cor;      static final long serialVersionUID = 123L; //indica a versao da classe        public FrutaSobrescrita(String nome, String cor) {          this.nome = nome;          this.cor = cor;      }      public String getCor() { return cor; }      public String getNome() { return nome; }      public void setCor(String cor) { this.cor = cor; }      public void setNome(String nome) { this.nome = nome; }        private void writeObject(java.io.ObjectOutputStream stream) throws java.io.IOException      {          // criptografar os dados antes de liberar o stream          // EXEMPLO: setCor(criptografar(getCor()));          setCor(getCor()+ " > CRIPTOGRAFADO"); // apenas concatenação de string para indicar alteração            // liberar o stream          stream.defaultWriteObject();      }        private void readObject(java.io.ObjectInputStream stream) throws java.io.IOException, ClassNotFoundException      {          // liberar stream          stream.defaultReadObject();            // descriptografar os dados          // EXEMPLO: setCor(descriptografar(getCor()));          setCor(getCor()+ " > DESCRIPTOGRAFADO"); // apenas contatenação de string para indicar alteração      }  } |

Listagem 11 - Execução da serialização da classe “pojo.FrutaSobrescrita”.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | //serializa        Serializador s = new Serializador();        FrutaSobrescrita frutaSobrescrita = new FrutaSobrescrita("Maça", "Vermelha");         try {             s.serializar("/home/gustavo/frutaSobrescrita", frutaSobrescrita);         } catch (Exception ex) {             System.err.println("Falha ao serializar! - " + ex.toString());         }           //deserializa        Deserializador d = new Deserializador();         frutaSobrescrita = null;         try {             frutaSobrescrita = (FrutaSobrescrita) d.deserializar("/home/gustavo/frutaSobrescrita");         } catch (Exception ex) {             System.err.println("Falha ao deserializar! - " + ex.toString());         }         System.out.println("Sobrescrita: " + frutaSobrescrita.getNome() + " - " + frutaSobrescrita.getCor()); |

**5. CONCLUSÃO**

A serialização é um procedimento de simples execução por parte do programador. Com pouco tempo de codificação é possível implementar a persistencia de um estado de objeto para o sistema de arquivos ou de transmissão por rede. É possível reproduzir os exemplos criados neste artigo, em situações até mesmo mais elaboradas. A apresentação de alguns problemas e soluções trazidas por particularidades, abre o entendimento de que a serialização pode deixar de ser simples conforme for aplicada em diferentes contextos, e que ao mesmo tempo é muito flexível ao permitir ao programador implementar diferentes soluções.

[Assista ao vídeo Serialização/Deserialização de objetos com YAML](http://www.devmedia.com.br/serializacao-deserializacao-de-objetos-com-yaml-parte-1/8399)

**6. BIBLIOGRAFIA**

ORACLE - **java.io Interface Serializable** - Disponível em: <http://docs.oracle.com/javase/1.4.2/docs/api/java/io/Serializable.html>Acesso em: 11/03/2012

ORACLE - **Discover de Secrets of Java Serialization API** - Disponível em: <http://java.sun.com/developer/technicalArticles/Programming/serialization/> Acesso em: 11/03/2012

NEWARD, Ted - **5 coisas que você não sabia sobre... Serialização de Objetos Java** - Disponível em: <http://www.ibm.com/developerworks/br/library/j-5things1/index.html> Acesso em: 11/03/2012

DOWNS, Andrew - **Java Serialization** - Disponível em: <http://www.mactech.com/articles/mactech/Vol.14/14.04/JavaSerialization/index.html> Acesso em: 11/03/2012

PREISSLER, Luciano E.; GEYER, Cláudio F. R. - **Serialização em Java** - Disponível em: <http://www.inf.ufrgs.br/gppd/disc/cmp167/trabalhos/sem99-1/T1/luciano/final.htm> Acesso em: 11/03/2012

[[](http://www.linhadecodigo.com.br/colaborador.aspx?id=354)](http://www.linhadecodigo.com.br/colaborador.aspx?id=354)

[Everton Coimbra de Araújo - Desde 1987 atua na área de treinamento e desenvolvimento. Como Mestre em Ciência da Computação, é professor da UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ, Campus Medianeira, onde leciona disciplinas relacionadas ao desenvolvimento de aplicações web, com Java e .NET.   
  
É autor dos livros Desenvolvimento para WEB com Java, Orientação a Objetos com Java - Simples, Fácil e Eficiente, Algoritmo - Fundamento e Prática em sua terceira edição e dos livros Delphi - Implementação e Técnicas para Ambientes Virtuais e C++ Builder - Implementação e Técnicas para Ambientes Virtuais. Todos pela VisualBooks. Pode ser contactado através do e-mail everton@utfpr.edu.br ou evertoncoimbra@gmail.com.](http://www.linhadecodigo.com.br/colaborador.aspx?id=354)

 85 3 0 4

**Leia também**

* [Conhecendo a interface Map do Java](http://www.linhadecodigo.com.br/artigo/3670/conhecendo-a-interface-map-do-java.aspx)*Java*
* [Trabalhando com a Interface Set no Java](http://www.linhadecodigo.com.br/artigo/3669/trabalhando-com-a-interface-set-no-java.aspx)*Java*
* [Classes Wrappers em Java](http://www.linhadecodigo.com.br/artigo/3667/classes-wrappers-em-java.aspx)*Java*
* [ComboBoxModel em Java](http://www.linhadecodigo.com.br/artigo/3664/comboboxmodel-em-java.aspx)*Java*
* [Reduzindo a Mutabilidade em Java](http://www.linhadecodigo.com.br/artigo/3659/reduzindo-a-mutabilidade-em-java.aspx)*Java*

Publicidade

[](http://qconsp.com/?utm_source=devmedia&utm_medium=banner&utm_campaign=devmedia)

Linha de Código[Anuncie](http://www.web-03.net/linhadecodigo.asp)|[Fale Conosco](mailto:colabore@linhadecodigo.com.br)|[Publique](http://www.linhadecodigo.com.br/publique.aspx)

|  |  |
| --- | --- |
| http://www.linhadecodigo.com.br/artigos/img_artigos/EvertonCoimbraAraujo/Serializazao_Objetos/imagem001.jpg | Neste artigo apresento o Gustavo Rafael Valiati, que pode ser contatado pelo e-mail gustavovaliati@gmail.com. O Gustavo foi meu aluno em algumas disciplinas de sua gradução e está concluíndo o último período do curso de Tecnologia em Análise e Desenvolvimento de Sistemas da UTFPR campus Medianeira, neste primeiro semestre de 2012. |

**1. INTRODUÇÃO**

Nem sempre se deseja ter objetos ou estrutura de dados de um aplicativo, com um ciclo de vida que dura no máximo o tempo em que for armazenado em memória RAM. Ou seja, algumas vezes deseja-se guardá-los por um tempo indeterminado, sendo possível recuperá-los posteriormente. E neste caso, o termo “persistência” tem seu conceito definido.

Persistir dados, requer que estes possam ser serializáveis. A serialização possibilitará capturar o estado do objeto ou a estrutura de dados, e transformar em uma cadeia de bytes quando for necessário. Torna-se possível também, recuperar os bytes persistidos para fazer o processo inverso, e ter os dados de volta para a aplicação em execução.

Desta maneira, qualquer forma de dados que esteja sendo trabalhada na aplicação, e esta puder ser serializada, poderá ser passada para o meio físico, e ter o processo revertido.

A serialização não se limita apenas em possibilitar a gravação do dado em disco, mas permite que o mesmo seja transmitido por rede. Isto porquê a serialização cria um stream de bytes, que é requisito para transmissão de dados por rede.

Diversas linguagens possuem suporte à serialização, como: Java, C, C++, Python, PHP, .NET, entre outras. Neste artigo, será tratado apenas da serialização com Java, em sua implemetação básica, alguns problemas e soluções, com códigos para exemplificação.

**2. SERIALIZAÇÃO COM JAVA**

Serializar um objeto, em Java, só é possível caso sua classe esteja marcada como serializável. Essa marcação é feita com a simples implementação da interface “java.io.Serializable”. Essa interface não possui métodos para se implementar e nem atributos. Ainda não existiam as anotações no Java 1.1, versão que trouxe pela primeira vez a serialização ao Java, e desta maneira a implementação da interface era a melhor maneira de marcar uma classe.

A herança entre classes é naturalmente afetada pela serialização. Já que quando uma classe implementa a interface “Serializable”, toda classe que dela estender também estará implicitamente marcada como serializável

.

Para exemplo, será criada uma classe “pojo.Fruta” que contém dois atributos: nome e cor. No código exemplo (Listagem 1), pode-se notar a classe “Fruta” marcada para a serialização:

Listagem 1 - Marcando uma classe como serializável.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | import java.io.Serializable;    public class Fruta implements Serializable{      private String nome;      private String cor;        public Fruta(String nome, String cor) {          this.nome = nome;          this.cor = cor;      }        public String getCor() { return cor; }      public String getNome() { return nome; }      public void setCor(String cor) { this.cor = cor; }      public void setNome(String nome) { this.nome = nome; }  } |

Pode-se dizer que a implementação do processo de serialização e deserialização simples de um objeto, não possui um código difícil de se escrever. Isto porquê duas classes fazem a maior parte do trabalho, de maneira que com poucas linhas de código pode-se transformar um objeto em bytes e vice-versa. Essas duas classes são “java.io.ObjectInputStream” que recebe bytes para criar um objeto e “java.io.ObjectOutputStream” que cria uma cadeia de bytes a partir de um objeto.

**3. SERIALIZANDO E DESERIALIZANDO O ESTADO DE UM OBJETO**

Como exemplo, será persistido um objeto em disco. Para isto, além das classes “java.io.ObjectInputStream” e “java.io.ObjectOutputStream”, serão utilizadas duas outras que trabalham com arquivos, que são “java.io.FileInputStream” e “java.io.FileOutputStream”, que seguem a mesma lógica de InputStream e OutputStream, só que agora relacionado ao sistema de arquivos e não aos objetos.

Para serialização, deve-se criar um novo “java.io.FileOutputStream”, com o caminho (“path”) do arquivo onde se deseja armazenar o estado do objeto. Também deve ser criado um novo “java.io.ObjectOutputStream” que está ligado ao arquivo. E por último deve-se executar a escrita do objeto alvo, para dentro do objeto de stream que por sua vez está ligado ao arquivo destino. Após executados esses passos, é possível encontrar no sistema de arquivos um novo arquivo que contém os bytes correspondentes à descrição do estado do objeto. Na Listagem 2, encontra-se o código exemplo.

Listagem 2 - Serialização de um objeto.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | import java.io.FileOutputStream;  import java.io.ObjectOutputStream;    public class Serializador {        public Serializador() {    }        public void serializar(String path, Object obj) throws Exception {              FileOutputStream outFile = new FileOutputStream(path);      ObjectOutputStream s = new ObjectOutputStream(outFile);      s.writeObject(obj);      s.close();      }  } |

E para a deserialização, o processo é extremamente parecido com a serialização. Cria-se um novo “java.io.FileInputStream” relacionado ao arquivo onde o estado do objeto encontra-se persistido. Deve-se criar também um novo “java.io.ObjectInputStream” que estará vinculado ao arquivo de origem. Por último, executa-se a leitura do arquivo para a restauração do estado. Na Listagem 3 encontra-se o exemplo do código.

Listagem 3 - Deserializaçãode um objeto.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | import java.io.FileInputStream;  import java.io.ObjectInputStream;    public class Deserializador {        public Deserializador() {    }        public Object deserializar(String path) throws Exception {      FileInputStream inFile = new FileInputStream(path);      ObjectInputStream d = new ObjectInputStream(inFile);      Object o = d.readObject();      d.close();      return o;      }  } |

A Listagem 4, contém o exemplo da execução da serialização e em seguida da deserialização.

Listagem 4 - Serializando e Deserializando

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | import operacoes.Deserializador;  import operacoes.Serializador;  import pojo.Fruta;    public class Principal {      public static void main(String args[]){         //serializa         Serializador s = new Serializador();         Fruta fruta = new Fruta("Maça", "Vermelha");          try {              s.serializar("/home/gustavo/fruta", fruta);          } catch (Exception ex) {              System.err.println("Falha ao serializar! - " + ex.toString());          }            //deserializa          Deserializador d = new Deserializador();          fruta = null;          try {              fruta = (Fruta) d.deserializar("/home/gustavo/fruta");          } catch (Exception ex) {              System.err.println("Falha ao deserializar! - " + ex.toString());          }          System.out.println(fruta.getNome() + " - " + fruta.getCor());      }  } |

Na Listagem 5, encontra-se um exemplo de serialização e deserialização de um ArrayList da classe Fruta. É apenas mais um exemplo que demonstra a capacidade de se persistir qualquer objeto que seja serializável. Essa serialização só foi possível porque a classe “ArrayList” implementa a interface “Serializable” por padrão.

Listagem 5 - Serialização de ArrayList.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | //serialização e deserialização com ArrayList          ArrayList<Fruta> frutas = new ArrayList<Fruta>();          Fruta f1 = new Fruta("Laranja", "Amarela");          Fruta f2 = new Fruta("Abacate", "Verde");          Fruta f3 = new Fruta("Morango", "Vermelho");          frutas.add(f1);          frutas.add(f2);          frutas.add(f3);            try {              s.serializar("/home/gustavo/frutas", frutas);              frutas = null;              frutas = (ArrayList<Fruta>) d.deserializar("/home/gustavo/frutas");              for(Fruta f : frutas){                  System.out.println("ArrayList: " + f.getNome() + " - " + f.getCor());              }          } catch (Exception ex) {              System.err.println("Falha ao serializar ou deserializar! - " +                                 ex.toString());          } |

**4. PARTICULARIDADES DA SERIALIZAÇÃO**

Como dito, a serialização de um objeto em Java pode ser algo fácil de se fazer. Mas existem certas informações que são importantes e sabê-las pode evitar vários problemas.

Um dos problemas é que nem tudo pode ser serializado, como objetos das classes “Thread” e “Socket” e atributos estáticos. Os atributos estáticos estão vinculados à classe e não ao objeto. Assim quando se trata do estado do objeto, os campos estáticos não são considerados. Em relação às Threads, suas instâncias só são úteis enquanto estiverem em execução na JVM, e podem ser paradas/destruídas e iniciadas com instruções de sua classe. Não existe algo no estado destas instâncias que seja útil para se persistir, e por isso não há sentido em persistir tais instâncias para restaurar posteriormente.

Mas e se existir alguma classe em que seja necessário persisti-la, e esta possui uma instância da classe “Thread”? Neste caso, utilizando um recurso pode-se persistir a classe inteira, menos a instância thread. Este recurso, refere-se em indicar que tal instância thread é um objeto “Transient”. Na Listagem 6 encontra-se um exemplo da implementação com uso de “Transient”.

Listagem 6 - Uso do Transient

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | import java.io.Serializable;    public class ImplementacaoTransient implements Serializable, Runnable{        transient private Thread thread;      //marcado com "transient" para que sua instância não seja serializada        //Demais valores de estado serão serializados      private int valorParaPersistir;        public ImplementacaoTransient(int valorParaPersistir){          this.valorParaPersistir = valorParaPersistir;          thread = new Thread(this);          thread.start();      }        public void run()      {          while(true){              //código a ser executado pela Thread          }      }  } |

Vale ressaltar que, o que é serializado é apenas o estado do objeto, e não os métodos e a classe. No exemplo acima, apenas a instância do objeto “thread” e do objeto “valorParaPersistir” compunham o estado do objeto. Assim, evita-se que o sistema de serialização tente serializar um objeto não-serializável e gere erros. Após a deserialização de um objeto da classe “ImplementacaoTransient” este estará totalmente funcional.

Um outro problema que pode vir a ocorrer, se refere ao versionamento das classes. Toda classe possui um atributo oculto chamado “serialVersionUID” com um valor gerado automaticamente. Objetos criados levam consigo o “serialVersionUID” de sua classe, que indica sua versão. Toda vez que uma classe sofre alterações em seu código, um novo “serialVersionUID” é gerado. O problema com a serialização surge quando serializa-se o estado de um objeto e posteriormente tenta-se restaurar o estado para a aplicação, mas houveram mudanças no código da classe. Neste caso, uma exceção “java.io.InvalidClassException” é disparada, pois de fato, o estado do objeto corresponde a uma versão de classe do qual foi criado que já não existe mais. Portanto, isso só ocorre para evitar erros de programação, em que tenta-se restaurar o estado de um objeto para uma classe que possui versão diferente da qual o objeto foi serializado inicialmente. E isso pode significar diferentes versões de uma mesma classe “Fruta” por exemplo, ou diferentes versões por classes serem totalmente diferentes como: o estado de um objeto da classe “Fruta” ser restaurado para classe “Carro”.

Mas quando é necessário, conscientemente, recuperar o estado de um objeto, mesmo que houveram alterações na classe do qual foi gerado, é possível controlar manualmente o “serialVersionUID”. Para isto, basta declarar explicitamente na forma de atributo estático e final, um valor Long qualquer para corresponder à versão da classe. E assim, mesmo que houverem modificações na classe, a versão permenecerá a mesma, evitando as exceções da incompatiblidade de classes. No entanto, vale ressaltar que o versionamento não é um impecílio para o programador, mas sim muito importante em casos que deseja-se manter um controle sobre a compatibilidade de código na aplicação, definindo versões para classes No código da Listagem 7 encontra-se o exemplo do controle manual do “serialVersionUID”.

Listagem 7 - Uso manual do serialVersionUID

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | import java.io.Serializable;    public class Fruta implements Serializable{      private String nome;      private String cor;      static final long serialVersionUID = 123L; //indica a versao da classe        public Fruta(String nome, String cor) {          this.nome = nome;          this.cor = cor;      }      public String getCor() { return cor; }      public String getNome() { return nome; }      public void setCor(String cor) { this.cor = cor; }      public void setNome(String nome) { this.nome = nome; }  } |

O JDK fornece uma ferramenta que permite visualizar a versão de qualquer classe Java. Basta no terminal do sistema operacional utilizar o comando “serialver” seguido da classe qualificada, conforme exemplo da Listagem 8.

Listagem 8 - Verificação da versão de classe

|  |  |
| --- | --- |
| 1 | serialver pojo.Fruta |

A serialização acaba inflingindo alguns modelos de segurança em seu procedimento padrão, e criando então um novo problema. Quando um estado de objeto é persistido no sistema de arquivos ou enviado por rede, o fluxo binário pode ser detectado, analisado e obtidas informações sobre a classe e o estado do objeto. Por exemplo, se utilizado um editor de texto qualquer para abrir o arquivo persistido do estado do objeto da classe “Fruta” dos exemplos anteriores, é comum os atributos e outras informações aparecerem como um simples texto, total ou parcilamente legíveis, assim como demonstrado na Listagem 9.

Listagem 9 - Exemplo de leitura de arquivo binário de um estado de objeto persistido.

|  |  |
| --- | --- |
| 1  2 | ��##sr#  pojo.Fruta#######{###L##cort##Ljava/lang/String;L##nomeq#~##xpt##Vermelhat##Maça |

Com uma ferramenta menos simples seria possível aumentar a quantidade de informações legíveis, coletando uma maior quantia de dados.

No entando, o processo de serialização permite que haja uma certa personalização da serialização e deserialização de um objeto. Isso deve-se a possibilidade da sobrescrição dos métodos “writeObject” e “readObject” dentro da classe a ser serializada, como a classe exemplar “pojo.Fruta”. Esta sobrescrita permite um controle total sobre a serialização, pois as classes “java.io.ObjectOutputStream” e “java.io.ObjectInputStream”, ao serializar o estado de um objeto, primeiramente verificam se esse método foi sobrescrito na classe alvo e assim o utilizam, senão, será utilizado o protocolo padrão de serialização.

Realizando a sobrescrição dos métodos é possível criptografar os dados na serialização e descriptografar os dados na deserialização, por meio de algum algoritmo que o programador venha implementar. Assim os dados podem ser persistidos ou transmitidos por rede, sem que seja possível ler as partes mais importantes do estado de um objeto. Na listagem 10, a classe “pojo.Fruta” foi modificada dando origem a classe “pojo.FrutaSobrescrita” que possui adicionamente a sobrescrição dos métodos “writeObject” e “readObject”. Como este é apenas um exemplo, não foi escrito um verdadeiro algoritmo de criptografia, mas sim apenas uma troca no valor da String do campo “cor”, identificando que houveram alterações tanto no processo de serialização quanto no de deserialização. Na listagem 11 encontra-se a execução da serialização.

Listagem 10 - Sobrescrita dos métodos “writeObject” e “readObject”.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36 | import java.io.Serializable;    public class FrutaSobrescrita implements Serializable{      private String nome;      private String cor;      static final long serialVersionUID = 123L; //indica a versao da classe        public FrutaSobrescrita(String nome, String cor) {          this.nome = nome;          this.cor = cor;      }      public String getCor() { return cor; }      public String getNome() { return nome; }      public void setCor(String cor) { this.cor = cor; }      public void setNome(String nome) { this.nome = nome; }        private void writeObject(java.io.ObjectOutputStream stream) throws java.io.IOException      {          // criptografar os dados antes de liberar o stream          // EXEMPLO: setCor(criptografar(getCor()));          setCor(getCor()+ " > CRIPTOGRAFADO"); // apenas concatenação de string para indicar alteração            // liberar o stream          stream.defaultWriteObject();      }        private void readObject(java.io.ObjectInputStream stream) throws java.io.IOException, ClassNotFoundException      {          // liberar stream          stream.defaultReadObject();            // descriptografar os dados          // EXEMPLO: setCor(descriptografar(getCor()));          setCor(getCor()+ " > DESCRIPTOGRAFADO"); // apenas contatenação de string para indicar alteração      }  } |

Listagem 11 - Execução da serialização da classe “pojo.FrutaSobrescrita”.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | //serializa        Serializador s = new Serializador();        FrutaSobrescrita frutaSobrescrita = new FrutaSobrescrita("Maça", "Vermelha");         try {             s.serializar("/home/gustavo/frutaSobrescrita", frutaSobrescrita);         } catch (Exception ex) {             System.err.println("Falha ao serializar! - " + ex.toString());         }           //deserializa        Deserializador d = new Deserializador();         frutaSobrescrita = null;         try {             frutaSobrescrita = (FrutaSobrescrita) d.deserializar("/home/gustavo/frutaSobrescrita");         } catch (Exception ex) {             System.err.println("Falha ao deserializar! - " + ex.toString());         }         System.out.println("Sobrescrita: " + frutaSobrescrita.getNome() + " - " + frutaSobrescrita.getCor()); |

**5. CONCLUSÃO**

A serialização é um procedimento de simples execução por parte do programador. Com pouco tempo de codificação é possível implementar a persistencia de um estado de objeto para o sistema de arquivos ou de transmissão por rede. É possível reproduzir os exemplos criados neste artigo, em situações até mesmo mais elaboradas. A apresentação de alguns problemas e soluções trazidas por particularidades, abre o entendimento de que a serialização pode deixar de ser simples conforme for aplicada em diferentes contextos, e que ao mesmo tempo é muito flexível ao permitir ao programador implementar diferentes soluções.

[Assista ao vídeo Serialização/Deserialização de objetos com YAML](http://www.devmedia.com.br/serializacao-deserializacao-de-objetos-com-yaml-parte-1/8399)

**6. BIBLIOGRAFIA**

ORACLE - **java.io Interface Serializable** - Disponível em: <http://docs.oracle.com/javase/1.4.2/docs/api/java/io/Serializable.html>Acesso em: 11/03/2012

ORACLE - **Discover de Secrets of Java Serialization API** - Disponível em: <http://java.sun.com/developer/technicalArticles/Programming/serialization/> Acesso em: 11/03/2012

NEWARD, Ted - **5 coisas que você não sabia sobre... Serialização de Objetos Java** - Disponível em: <http://www.ibm.com/developerworks/br/library/j-5things1/index.html> Acesso em: 11/03/2012

DOWNS, Andrew - **Java Serialization** - Disponível em: <http://www.mactech.com/articles/mactech/Vol.14/14.04/JavaSerialization/index.html> Acesso em: 11/03/2012

PREISSLER, Luciano E.; GEYER, Cláudio F. R. - **Serialização em Java** - Disponível em: <http://www.inf.ufrgs.br/gppd/disc/cmp167/trabalhos/sem99-1/T1/luciano/final.htm> Acesso em: 11/03/2012

[[](http://www.linhadecodigo.com.br/colaborador.aspx?id=354)](http://www.linhadecodigo.com.br/colaborador.aspx?id=354)

[Everton Coimbra de Araújo - Desde 1987 atua na área de treinamento e desenvolvimento. Como Mestre em Ciência da Computação, é professor da UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ, Campus Medianeira, onde leciona disciplinas relacionadas ao desenvolvimento de aplicações web, com Java e .NET.   
  
É autor dos livros Desenvolvimento para WEB com Java, Orientação a Objetos com Java - Simples, Fácil e Eficiente, Algoritmo - Fundamento e Prática em sua terceira edição e dos livros Delphi - Implementação e Técnicas para Ambientes Virtuais e C++ Builder - Implementação e Técnicas para Ambientes Virtuais. Todos pela VisualBooks. Pode ser contactado através do e-mail everton@utfpr.edu.br ou evertoncoimbra@gmail.com.](http://www.linhadecodigo.com.br/colaborador.aspx?id=354)

 85 3 0 4

**Leia também**

* [Conhecendo a interface Map do Java](http://www.linhadecodigo.com.br/artigo/3670/conhecendo-a-interface-map-do-java.aspx)*Java*
* [Trabalhando com a Interface Set no Java](http://www.linhadecodigo.com.br/artigo/3669/trabalhando-com-a-interface-set-no-java.aspx)*Java*
* [Classes Wrappers em Java](http://www.linhadecodigo.com.br/artigo/3667/classes-wrappers-em-java.aspx)*Java*
* [ComboBoxModel em Java](http://www.linhadecodigo.com.br/artigo/3664/comboboxmodel-em-java.aspx)*Java*
* [Reduzindo a Mutabilidade em Java](http://www.linhadecodigo.com.br/artigo/3659/reduzindo-a-mutabilidade-em-java.aspx)*Java*

Publicidade

[](http://qconsp.com/?utm_source=devmedia&utm_medium=banner&utm_campaign=devmedia)

Linha de Código[Anuncie](http://www.web-03.net/linhadecodigo.asp)|[Fale Conosco](mailto:colabore@linhadecodigo.com.br)|[Publique](http://www.linhadecodigo.com.br/publique.aspx)

|  |  |
| --- | --- |
| http://www.linhadecodigo.com.br/artigos/img_artigos/EvertonCoimbraAraujo/Serializazao_Objetos/imagem001.jpg | Neste artigo apresento o Gustavo Rafael Valiati, que pode ser contatado pelo e-mail gustavovaliati@gmail.com. O Gustavo foi meu aluno em algumas disciplinas de sua gradução e está concluíndo o último período do curso de Tecnologia em Análise e Desenvolvimento de Sistemas da UTFPR campus Medianeira, neste primeiro semestre de 2012. |

**1. INTRODUÇÃO**

Nem sempre se deseja ter objetos ou estrutura de dados de um aplicativo, com um ciclo de vida que dura no máximo o tempo em que for armazenado em memória RAM. Ou seja, algumas vezes deseja-se guardá-los por um tempo indeterminado, sendo possível recuperá-los posteriormente. E neste caso, o termo “persistência” tem seu conceito definido.

Persistir dados, requer que estes possam ser serializáveis. A serialização possibilitará capturar o estado do objeto ou a estrutura de dados, e transformar em uma cadeia de bytes quando for necessário. Torna-se possível também, recuperar os bytes persistidos para fazer o processo inverso, e ter os dados de volta para a aplicação em execução.

Desta maneira, qualquer forma de dados que esteja sendo trabalhada na aplicação, e esta puder ser serializada, poderá ser passada para o meio físico, e ter o processo revertido.

A serialização não se limita apenas em possibilitar a gravação do dado em disco, mas permite que o mesmo seja transmitido por rede. Isto porquê a serialização cria um stream de bytes, que é requisito para transmissão de dados por rede.

Diversas linguagens possuem suporte à serialização, como: Java, C, C++, Python, PHP, .NET, entre outras. Neste artigo, será tratado apenas da serialização com Java, em sua implemetação básica, alguns problemas e soluções, com códigos para exemplificação.

**2. SERIALIZAÇÃO COM JAVA**

Serializar um objeto, em Java, só é possível caso sua classe esteja marcada como serializável. Essa marcação é feita com a simples implementação da interface “java.io.Serializable”. Essa interface não possui métodos para se implementar e nem atributos. Ainda não existiam as anotações no Java 1.1, versão que trouxe pela primeira vez a serialização ao Java, e desta maneira a implementação da interface era a melhor maneira de marcar uma classe.

A herança entre classes é naturalmente afetada pela serialização. Já que quando uma classe implementa a interface “Serializable”, toda classe que dela estender também estará implicitamente marcada como serializável

.

Para exemplo, será criada uma classe “pojo.Fruta” que contém dois atributos: nome e cor. No código exemplo (Listagem 1), pode-se notar a classe “Fruta” marcada para a serialização:

Listagem 1 - Marcando uma classe como serializável.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | import java.io.Serializable;    public class Fruta implements Serializable{      private String nome;      private String cor;        public Fruta(String nome, String cor) {          this.nome = nome;          this.cor = cor;      }        public String getCor() { return cor; }      public String getNome() { return nome; }      public void setCor(String cor) { this.cor = cor; }      public void setNome(String nome) { this.nome = nome; }  } |

Pode-se dizer que a implementação do processo de serialização e deserialização simples de um objeto, não possui um código difícil de se escrever. Isto porquê duas classes fazem a maior parte do trabalho, de maneira que com poucas linhas de código pode-se transformar um objeto em bytes e vice-versa. Essas duas classes são “java.io.ObjectInputStream” que recebe bytes para criar um objeto e “java.io.ObjectOutputStream” que cria uma cadeia de bytes a partir de um objeto.

**3. SERIALIZANDO E DESERIALIZANDO O ESTADO DE UM OBJETO**

Como exemplo, será persistido um objeto em disco. Para isto, além das classes “java.io.ObjectInputStream” e “java.io.ObjectOutputStream”, serão utilizadas duas outras que trabalham com arquivos, que são “java.io.FileInputStream” e “java.io.FileOutputStream”, que seguem a mesma lógica de InputStream e OutputStream, só que agora relacionado ao sistema de arquivos e não aos objetos.

Para serialização, deve-se criar um novo “java.io.FileOutputStream”, com o caminho (“path”) do arquivo onde se deseja armazenar o estado do objeto. Também deve ser criado um novo “java.io.ObjectOutputStream” que está ligado ao arquivo. E por último deve-se executar a escrita do objeto alvo, para dentro do objeto de stream que por sua vez está ligado ao arquivo destino. Após executados esses passos, é possível encontrar no sistema de arquivos um novo arquivo que contém os bytes correspondentes à descrição do estado do objeto. Na Listagem 2, encontra-se o código exemplo.

Listagem 2 - Serialização de um objeto.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | import java.io.FileOutputStream;  import java.io.ObjectOutputStream;    public class Serializador {        public Serializador() {    }        public void serializar(String path, Object obj) throws Exception {              FileOutputStream outFile = new FileOutputStream(path);      ObjectOutputStream s = new ObjectOutputStream(outFile);      s.writeObject(obj);      s.close();      }  } |

E para a deserialização, o processo é extremamente parecido com a serialização. Cria-se um novo “java.io.FileInputStream” relacionado ao arquivo onde o estado do objeto encontra-se persistido. Deve-se criar também um novo “java.io.ObjectInputStream” que estará vinculado ao arquivo de origem. Por último, executa-se a leitura do arquivo para a restauração do estado. Na Listagem 3 encontra-se o exemplo do código.

Listagem 3 - Deserializaçãode um objeto.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | import java.io.FileInputStream;  import java.io.ObjectInputStream;    public class Deserializador {        public Deserializador() {    }        public Object deserializar(String path) throws Exception {      FileInputStream inFile = new FileInputStream(path);      ObjectInputStream d = new ObjectInputStream(inFile);      Object o = d.readObject();      d.close();      return o;      }  } |

A Listagem 4, contém o exemplo da execução da serialização e em seguida da deserialização.

Listagem 4 - Serializando e Deserializando

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | import operacoes.Deserializador;  import operacoes.Serializador;  import pojo.Fruta;    public class Principal {      public static void main(String args[]){         //serializa         Serializador s = new Serializador();         Fruta fruta = new Fruta("Maça", "Vermelha");          try {              s.serializar("/home/gustavo/fruta", fruta);          } catch (Exception ex) {              System.err.println("Falha ao serializar! - " + ex.toString());          }            //deserializa          Deserializador d = new Deserializador();          fruta = null;          try {              fruta = (Fruta) d.deserializar("/home/gustavo/fruta");          } catch (Exception ex) {              System.err.println("Falha ao deserializar! - " + ex.toString());          }          System.out.println(fruta.getNome() + " - " + fruta.getCor());      }  } |

Na Listagem 5, encontra-se um exemplo de serialização e deserialização de um ArrayList da classe Fruta. É apenas mais um exemplo que demonstra a capacidade de se persistir qualquer objeto que seja serializável. Essa serialização só foi possível porque a classe “ArrayList” implementa a interface “Serializable” por padrão.

Listagem 5 - Serialização de ArrayList.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | //serialização e deserialização com ArrayList          ArrayList<Fruta> frutas = new ArrayList<Fruta>();          Fruta f1 = new Fruta("Laranja", "Amarela");          Fruta f2 = new Fruta("Abacate", "Verde");          Fruta f3 = new Fruta("Morango", "Vermelho");          frutas.add(f1);          frutas.add(f2);          frutas.add(f3);            try {              s.serializar("/home/gustavo/frutas", frutas);              frutas = null;              frutas = (ArrayList<Fruta>) d.deserializar("/home/gustavo/frutas");              for(Fruta f : frutas){                  System.out.println("ArrayList: " + f.getNome() + " - " + f.getCor());              }          } catch (Exception ex) {              System.err.println("Falha ao serializar ou deserializar! - " +                                 ex.toString());          } |

**4. PARTICULARIDADES DA SERIALIZAÇÃO**

Como dito, a serialização de um objeto em Java pode ser algo fácil de se fazer. Mas existem certas informações que são importantes e sabê-las pode evitar vários problemas.

Um dos problemas é que nem tudo pode ser serializado, como objetos das classes “Thread” e “Socket” e atributos estáticos. Os atributos estáticos estão vinculados à classe e não ao objeto. Assim quando se trata do estado do objeto, os campos estáticos não são considerados. Em relação às Threads, suas instâncias só são úteis enquanto estiverem em execução na JVM, e podem ser paradas/destruídas e iniciadas com instruções de sua classe. Não existe algo no estado destas instâncias que seja útil para se persistir, e por isso não há sentido em persistir tais instâncias para restaurar posteriormente.

Mas e se existir alguma classe em que seja necessário persisti-la, e esta possui uma instância da classe “Thread”? Neste caso, utilizando um recurso pode-se persistir a classe inteira, menos a instância thread. Este recurso, refere-se em indicar que tal instância thread é um objeto “Transient”. Na Listagem 6 encontra-se um exemplo da implementação com uso de “Transient”.

Listagem 6 - Uso do Transient

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | import java.io.Serializable;    public class ImplementacaoTransient implements Serializable, Runnable{        transient private Thread thread;      //marcado com "transient" para que sua instância não seja serializada        //Demais valores de estado serão serializados      private int valorParaPersistir;        public ImplementacaoTransient(int valorParaPersistir){          this.valorParaPersistir = valorParaPersistir;          thread = new Thread(this);          thread.start();      }        public void run()      {          while(true){              //código a ser executado pela Thread          }      }  } |

Vale ressaltar que, o que é serializado é apenas o estado do objeto, e não os métodos e a classe. No exemplo acima, apenas a instância do objeto “thread” e do objeto “valorParaPersistir” compunham o estado do objeto. Assim, evita-se que o sistema de serialização tente serializar um objeto não-serializável e gere erros. Após a deserialização de um objeto da classe “ImplementacaoTransient” este estará totalmente funcional.

Um outro problema que pode vir a ocorrer, se refere ao versionamento das classes. Toda classe possui um atributo oculto chamado “serialVersionUID” com um valor gerado automaticamente. Objetos criados levam consigo o “serialVersionUID” de sua classe, que indica sua versão. Toda vez que uma classe sofre alterações em seu código, um novo “serialVersionUID” é gerado. O problema com a serialização surge quando serializa-se o estado de um objeto e posteriormente tenta-se restaurar o estado para a aplicação, mas houveram mudanças no código da classe. Neste caso, uma exceção “java.io.InvalidClassException” é disparada, pois de fato, o estado do objeto corresponde a uma versão de classe do qual foi criado que já não existe mais. Portanto, isso só ocorre para evitar erros de programação, em que tenta-se restaurar o estado de um objeto para uma classe que possui versão diferente da qual o objeto foi serializado inicialmente. E isso pode significar diferentes versões de uma mesma classe “Fruta” por exemplo, ou diferentes versões por classes serem totalmente diferentes como: o estado de um objeto da classe “Fruta” ser restaurado para classe “Carro”.

Mas quando é necessário, conscientemente, recuperar o estado de um objeto, mesmo que houveram alterações na classe do qual foi gerado, é possível controlar manualmente o “serialVersionUID”. Para isto, basta declarar explicitamente na forma de atributo estático e final, um valor Long qualquer para corresponder à versão da classe. E assim, mesmo que houverem modificações na classe, a versão permenecerá a mesma, evitando as exceções da incompatiblidade de classes. No entanto, vale ressaltar que o versionamento não é um impecílio para o programador, mas sim muito importante em casos que deseja-se manter um controle sobre a compatibilidade de código na aplicação, definindo versões para classes No código da Listagem 7 encontra-se o exemplo do controle manual do “serialVersionUID”.

Listagem 7 - Uso manual do serialVersionUID

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | import java.io.Serializable;    public class Fruta implements Serializable{      private String nome;      private String cor;      static final long serialVersionUID = 123L; //indica a versao da classe        public Fruta(String nome, String cor) {          this.nome = nome;          this.cor = cor;      }      public String getCor() { return cor; }      public String getNome() { return nome; }      public void setCor(String cor) { this.cor = cor; }      public void setNome(String nome) { this.nome = nome; }  } |

O JDK fornece uma ferramenta que permite visualizar a versão de qualquer classe Java. Basta no terminal do sistema operacional utilizar o comando “serialver” seguido da classe qualificada, conforme exemplo da Listagem 8.

Listagem 8 - Verificação da versão de classe

|  |  |
| --- | --- |
| 1 | serialver pojo.Fruta |

A serialização acaba inflingindo alguns modelos de segurança em seu procedimento padrão, e criando então um novo problema. Quando um estado de objeto é persistido no sistema de arquivos ou enviado por rede, o fluxo binário pode ser detectado, analisado e obtidas informações sobre a classe e o estado do objeto. Por exemplo, se utilizado um editor de texto qualquer para abrir o arquivo persistido do estado do objeto da classe “Fruta” dos exemplos anteriores, é comum os atributos e outras informações aparecerem como um simples texto, total ou parcilamente legíveis, assim como demonstrado na Listagem 9.

Listagem 9 - Exemplo de leitura de arquivo binário de um estado de objeto persistido.

|  |  |
| --- | --- |
| 1  2 | ��##sr#  pojo.Fruta#######{###L##cort##Ljava/lang/String;L##nomeq#~##xpt##Vermelhat##Maça |

Com uma ferramenta menos simples seria possível aumentar a quantidade de informações legíveis, coletando uma maior quantia de dados.

No entando, o processo de serialização permite que haja uma certa personalização da serialização e deserialização de um objeto. Isso deve-se a possibilidade da sobrescrição dos métodos “writeObject” e “readObject” dentro da classe a ser serializada, como a classe exemplar “pojo.Fruta”. Esta sobrescrita permite um controle total sobre a serialização, pois as classes “java.io.ObjectOutputStream” e “java.io.ObjectInputStream”, ao serializar o estado de um objeto, primeiramente verificam se esse método foi sobrescrito na classe alvo e assim o utilizam, senão, será utilizado o protocolo padrão de serialização.

Realizando a sobrescrição dos métodos é possível criptografar os dados na serialização e descriptografar os dados na deserialização, por meio de algum algoritmo que o programador venha implementar. Assim os dados podem ser persistidos ou transmitidos por rede, sem que seja possível ler as partes mais importantes do estado de um objeto. Na listagem 10, a classe “pojo.Fruta” foi modificada dando origem a classe “pojo.FrutaSobrescrita” que possui adicionamente a sobrescrição dos métodos “writeObject” e “readObject”. Como este é apenas um exemplo, não foi escrito um verdadeiro algoritmo de criptografia, mas sim apenas uma troca no valor da String do campo “cor”, identificando que houveram alterações tanto no processo de serialização quanto no de deserialização. Na listagem 11 encontra-se a execução da serialização.

Listagem 10 - Sobrescrita dos métodos “writeObject” e “readObject”.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36 | import java.io.Serializable;    public class FrutaSobrescrita implements Serializable{      private String nome;      private String cor;      static final long serialVersionUID = 123L; //indica a versao da classe        public FrutaSobrescrita(String nome, String cor) {          this.nome = nome;          this.cor = cor;      }      public String getCor() { return cor; }      public String getNome() { return nome; }      public void setCor(String cor) { this.cor = cor; }      public void setNome(String nome) { this.nome = nome; }        private void writeObject(java.io.ObjectOutputStream stream) throws java.io.IOException      {          // criptografar os dados antes de liberar o stream          // EXEMPLO: setCor(criptografar(getCor()));          setCor(getCor()+ " > CRIPTOGRAFADO"); // apenas concatenação de string para indicar alteração            // liberar o stream          stream.defaultWriteObject();      }        private void readObject(java.io.ObjectInputStream stream) throws java.io.IOException, ClassNotFoundException      {          // liberar stream          stream.defaultReadObject();            // descriptografar os dados          // EXEMPLO: setCor(descriptografar(getCor()));          setCor(getCor()+ " > DESCRIPTOGRAFADO"); // apenas contatenação de string para indicar alteração      }  } |

Listagem 11 - Execução da serialização da classe “pojo.FrutaSobrescrita”.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | //serializa        Serializador s = new Serializador();        FrutaSobrescrita frutaSobrescrita = new FrutaSobrescrita("Maça", "Vermelha");         try {             s.serializar("/home/gustavo/frutaSobrescrita", frutaSobrescrita);         } catch (Exception ex) {             System.err.println("Falha ao serializar! - " + ex.toString());         }           //deserializa        Deserializador d = new Deserializador();         frutaSobrescrita = null;         try {             frutaSobrescrita = (FrutaSobrescrita) d.deserializar("/home/gustavo/frutaSobrescrita");         } catch (Exception ex) {             System.err.println("Falha ao deserializar! - " + ex.toString());         }         System.out.println("Sobrescrita: " + frutaSobrescrita.getNome() + " - " + frutaSobrescrita.getCor()); |

**5. CONCLUSÃO**

A serialização é um procedimento de simples execução por parte do programador. Com pouco tempo de codificação é possível implementar a persistencia de um estado de objeto para o sistema de arquivos ou de transmissão por rede. É possível reproduzir os exemplos criados neste artigo, em situações até mesmo mais elaboradas. A apresentação de alguns problemas e soluções trazidas por particularidades, abre o entendimento de que a serialização pode deixar de ser simples conforme for aplicada em diferentes contextos, e que ao mesmo tempo é muito flexível ao permitir ao programador implementar diferentes soluções.

[Assista ao vídeo Serialização/Deserialização de objetos com YAML](http://www.devmedia.com.br/serializacao-deserializacao-de-objetos-com-yaml-parte-1/8399)

**6. BIBLIOGRAFIA**

ORACLE - **java.io Interface Serializable** - Disponível em: <http://docs.oracle.com/javase/1.4.2/docs/api/java/io/Serializable.html>Acesso em: 11/03/2012

ORACLE - **Discover de Secrets of Java Serialization API** - Disponível em: <http://java.sun.com/developer/technicalArticles/Programming/serialization/> Acesso em: 11/03/2012

NEWARD, Ted - **5 coisas que você não sabia sobre... Serialização de Objetos Java** - Disponível em: <http://www.ibm.com/developerworks/br/library/j-5things1/index.html> Acesso em: 11/03/2012

DOWNS, Andrew - **Java Serialization** - Disponível em: <http://www.mactech.com/articles/mactech/Vol.14/14.04/JavaSerialization/index.html> Acesso em: 11/03/2012

PREISSLER, Luciano E.; GEYER, Cláudio F. R. - **Serialização em Java** - Disponível em: <http://www.inf.ufrgs.br/gppd/disc/cmp167/trabalhos/sem99-1/T1/luciano/final.htm> Acesso em: 11/03/2012

[[](http://www.linhadecodigo.com.br/colaborador.aspx?id=354)](http://www.linhadecodigo.com.br/colaborador.aspx?id=354)

[Everton Coimbra de Araújo - Desde 1987 atua na área de treinamento e desenvolvimento. Como Mestre em Ciência da Computação, é professor da UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ, Campus Medianeira, onde leciona disciplinas relacionadas ao desenvolvimento de aplicações web, com Java e .NET.   
  
É autor dos livros Desenvolvimento para WEB com Java, Orientação a Objetos com Java - Simples, Fácil e Eficiente, Algoritmo - Fundamento e Prática em sua terceira edição e dos livros Delphi - Implementação e Técnicas para Ambientes Virtuais e C++ Builder - Implementação e Técnicas para Ambientes Virtuais. Todos pela VisualBooks. Pode ser contactado através do e-mail everton@utfpr.edu.br ou evertoncoimbra@gmail.com.](http://www.linhadecodigo.com.br/colaborador.aspx?id=354)

 85 3 0 4

**Leia também**

* [Conhecendo a interface Map do Java](http://www.linhadecodigo.com.br/artigo/3670/conhecendo-a-interface-map-do-java.aspx)*Java*
* [Trabalhando com a Interface Set no Java](http://www.linhadecodigo.com.br/artigo/3669/trabalhando-com-a-interface-set-no-java.aspx)*Java*
* [Classes Wrappers em Java](http://www.linhadecodigo.com.br/artigo/3667/classes-wrappers-em-java.aspx)*Java*
* [ComboBoxModel em Java](http://www.linhadecodigo.com.br/artigo/3664/comboboxmodel-em-java.aspx)*Java*
* [Reduzindo a Mutabilidade em Java](http://www.linhadecodigo.com.br/artigo/3659/reduzindo-a-mutabilidade-em-java.aspx)*Java*

Publicidade

[](http://qconsp.com/?utm_source=devmedia&utm_medium=banner&utm_campaign=devmedia)

Linha de Código[Anuncie](http://www.web-03.net/linhadecodigo.asp)|[Fale Conosco](mailto:colabore@linhadecodigo.com.br)|[Publique](http://www.linhadecodigo.com.br/publique.aspx)

|  |  |
| --- | --- |
| http://www.linhadecodigo.com.br/artigos/img_artigos/EvertonCoimbraAraujo/Serializazao_Objetos/imagem001.jpg | Neste artigo apresento o Gustavo Rafael Valiati, que pode ser contatado pelo e-mail gustavovaliati@gmail.com. O Gustavo foi meu aluno em algumas disciplinas de sua gradução e está concluíndo o último período do curso de Tecnologia em Análise e Desenvolvimento de Sistemas da UTFPR campus Medianeira, neste primeiro semestre de 2012. |

**1. INTRODUÇÃO**

Nem sempre se deseja ter objetos ou estrutura de dados de um aplicativo, com um ciclo de vida que dura no máximo o tempo em que for armazenado em memória RAM. Ou seja, algumas vezes deseja-se guardá-los por um tempo indeterminado, sendo possível recuperá-los posteriormente. E neste caso, o termo “persistência” tem seu conceito definido.

Persistir dados, requer que estes possam ser serializáveis. A serialização possibilitará capturar o estado do objeto ou a estrutura de dados, e transformar em uma cadeia de bytes quando for necessário. Torna-se possível também, recuperar os bytes persistidos para fazer o processo inverso, e ter os dados de volta para a aplicação em execução.

Desta maneira, qualquer forma de dados que esteja sendo trabalhada na aplicação, e esta puder ser serializada, poderá ser passada para o meio físico, e ter o processo revertido.

A serialização não se limita apenas em possibilitar a gravação do dado em disco, mas permite que o mesmo seja transmitido por rede. Isto porquê a serialização cria um stream de bytes, que é requisito para transmissão de dados por rede.

Diversas linguagens possuem suporte à serialização, como: Java, C, C++, Python, PHP, .NET, entre outras. Neste artigo, será tratado apenas da serialização com Java, em sua implemetação básica, alguns problemas e soluções, com códigos para exemplificação.

**2. SERIALIZAÇÃO COM JAVA**

Serializar um objeto, em Java, só é possível caso sua classe esteja marcada como serializável. Essa marcação é feita com a simples implementação da interface “java.io.Serializable”. Essa interface não possui métodos para se implementar e nem atributos. Ainda não existiam as anotações no Java 1.1, versão que trouxe pela primeira vez a serialização ao Java, e desta maneira a implementação da interface era a melhor maneira de marcar uma classe.

A herança entre classes é naturalmente afetada pela serialização. Já que quando uma classe implementa a interface “Serializable”, toda classe que dela estender também estará implicitamente marcada como serializável

.

Para exemplo, será criada uma classe “pojo.Fruta” que contém dois atributos: nome e cor. No código exemplo (Listagem 1), pode-se notar a classe “Fruta” marcada para a serialização:

Listagem 1 - Marcando uma classe como serializável.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | import java.io.Serializable;    public class Fruta implements Serializable{      private String nome;      private String cor;        public Fruta(String nome, String cor) {          this.nome = nome;          this.cor = cor;      }        public String getCor() { return cor; }      public String getNome() { return nome; }      public void setCor(String cor) { this.cor = cor; }      public void setNome(String nome) { this.nome = nome; }  } |

Pode-se dizer que a implementação do processo de serialização e deserialização simples de um objeto, não possui um código difícil de se escrever. Isto porquê duas classes fazem a maior parte do trabalho, de maneira que com poucas linhas de código pode-se transformar um objeto em bytes e vice-versa. Essas duas classes são “java.io.ObjectInputStream” que recebe bytes para criar um objeto e “java.io.ObjectOutputStream” que cria uma cadeia de bytes a partir de um objeto.

**3. SERIALIZANDO E DESERIALIZANDO O ESTADO DE UM OBJETO**

Como exemplo, será persistido um objeto em disco. Para isto, além das classes “java.io.ObjectInputStream” e “java.io.ObjectOutputStream”, serão utilizadas duas outras que trabalham com arquivos, que são “java.io.FileInputStream” e “java.io.FileOutputStream”, que seguem a mesma lógica de InputStream e OutputStream, só que agora relacionado ao sistema de arquivos e não aos objetos.

Para serialização, deve-se criar um novo “java.io.FileOutputStream”, com o caminho (“path”) do arquivo onde se deseja armazenar o estado do objeto. Também deve ser criado um novo “java.io.ObjectOutputStream” que está ligado ao arquivo. E por último deve-se executar a escrita do objeto alvo, para dentro do objeto de stream que por sua vez está ligado ao arquivo destino. Após executados esses passos, é possível encontrar no sistema de arquivos um novo arquivo que contém os bytes correspondentes à descrição do estado do objeto. Na Listagem 2, encontra-se o código exemplo.

Listagem 2 - Serialização de um objeto.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | import java.io.FileOutputStream;  import java.io.ObjectOutputStream;    public class Serializador {        public Serializador() {    }        public void serializar(String path, Object obj) throws Exception {              FileOutputStream outFile = new FileOutputStream(path);      ObjectOutputStream s = new ObjectOutputStream(outFile);      s.writeObject(obj);      s.close();      }  } |

E para a deserialização, o processo é extremamente parecido com a serialização. Cria-se um novo “java.io.FileInputStream” relacionado ao arquivo onde o estado do objeto encontra-se persistido. Deve-se criar também um novo “java.io.ObjectInputStream” que estará vinculado ao arquivo de origem. Por último, executa-se a leitura do arquivo para a restauração do estado. Na Listagem 3 encontra-se o exemplo do código.

Listagem 3 - Deserializaçãode um objeto.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | import java.io.FileInputStream;  import java.io.ObjectInputStream;    public class Deserializador {        public Deserializador() {    }        public Object deserializar(String path) throws Exception {      FileInputStream inFile = new FileInputStream(path);      ObjectInputStream d = new ObjectInputStream(inFile);      Object o = d.readObject();      d.close();      return o;      }  } |

A Listagem 4, contém o exemplo da execução da serialização e em seguida da deserialização.

Listagem 4 - Serializando e Deserializando

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | import operacoes.Deserializador;  import operacoes.Serializador;  import pojo.Fruta;    public class Principal {      public static void main(String args[]){         //serializa         Serializador s = new Serializador();         Fruta fruta = new Fruta("Maça", "Vermelha");          try {              s.serializar("/home/gustavo/fruta", fruta);          } catch (Exception ex) {              System.err.println("Falha ao serializar! - " + ex.toString());          }            //deserializa          Deserializador d = new Deserializador();          fruta = null;          try {              fruta = (Fruta) d.deserializar("/home/gustavo/fruta");          } catch (Exception ex) {              System.err.println("Falha ao deserializar! - " + ex.toString());          }          System.out.println(fruta.getNome() + " - " + fruta.getCor());      }  } |

Na Listagem 5, encontra-se um exemplo de serialização e deserialização de um ArrayList da classe Fruta. É apenas mais um exemplo que demonstra a capacidade de se persistir qualquer objeto que seja serializável. Essa serialização só foi possível porque a classe “ArrayList” implementa a interface “Serializable” por padrão.

Listagem 5 - Serialização de ArrayList.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | //serialização e deserialização com ArrayList          ArrayList<Fruta> frutas = new ArrayList<Fruta>();          Fruta f1 = new Fruta("Laranja", "Amarela");          Fruta f2 = new Fruta("Abacate", "Verde");          Fruta f3 = new Fruta("Morango", "Vermelho");          frutas.add(f1);          frutas.add(f2);          frutas.add(f3);            try {              s.serializar("/home/gustavo/frutas", frutas);              frutas = null;              frutas = (ArrayList<Fruta>) d.deserializar("/home/gustavo/frutas");              for(Fruta f : frutas){                  System.out.println("ArrayList: " + f.getNome() + " - " + f.getCor());              }          } catch (Exception ex) {              System.err.println("Falha ao serializar ou deserializar! - " +                                 ex.toString());          } |

**4. PARTICULARIDADES DA SERIALIZAÇÃO**

Como dito, a serialização de um objeto em Java pode ser algo fácil de se fazer. Mas existem certas informações que são importantes e sabê-las pode evitar vários problemas.

Um dos problemas é que nem tudo pode ser serializado, como objetos das classes “Thread” e “Socket” e atributos estáticos. Os atributos estáticos estão vinculados à classe e não ao objeto. Assim quando se trata do estado do objeto, os campos estáticos não são considerados. Em relação às Threads, suas instâncias só são úteis enquanto estiverem em execução na JVM, e podem ser paradas/destruídas e iniciadas com instruções de sua classe. Não existe algo no estado destas instâncias que seja útil para se persistir, e por isso não há sentido em persistir tais instâncias para restaurar posteriormente.

Mas e se existir alguma classe em que seja necessário persisti-la, e esta possui uma instância da classe “Thread”? Neste caso, utilizando um recurso pode-se persistir a classe inteira, menos a instância thread. Este recurso, refere-se em indicar que tal instância thread é um objeto “Transient”. Na Listagem 6 encontra-se um exemplo da implementação com uso de “Transient”.

Listagem 6 - Uso do Transient

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | import java.io.Serializable;    public class ImplementacaoTransient implements Serializable, Runnable{        transient private Thread thread;      //marcado com "transient" para que sua instância não seja serializada        //Demais valores de estado serão serializados      private int valorParaPersistir;        public ImplementacaoTransient(int valorParaPersistir){          this.valorParaPersistir = valorParaPersistir;          thread = new Thread(this);          thread.start();      }        public void run()      {          while(true){              //código a ser executado pela Thread          }      }  } |

Vale ressaltar que, o que é serializado é apenas o estado do objeto, e não os métodos e a classe. No exemplo acima, apenas a instância do objeto “thread” e do objeto “valorParaPersistir” compunham o estado do objeto. Assim, evita-se que o sistema de serialização tente serializar um objeto não-serializável e gere erros. Após a deserialização de um objeto da classe “ImplementacaoTransient” este estará totalmente funcional.

Um outro problema que pode vir a ocorrer, se refere ao versionamento das classes. Toda classe possui um atributo oculto chamado “serialVersionUID” com um valor gerado automaticamente. Objetos criados levam consigo o “serialVersionUID” de sua classe, que indica sua versão. Toda vez que uma classe sofre alterações em seu código, um novo “serialVersionUID” é gerado. O problema com a serialização surge quando serializa-se o estado de um objeto e posteriormente tenta-se restaurar o estado para a aplicação, mas houveram mudanças no código da classe. Neste caso, uma exceção “java.io.InvalidClassException” é disparada, pois de fato, o estado do objeto corresponde a uma versão de classe do qual foi criado que já não existe mais. Portanto, isso só ocorre para evitar erros de programação, em que tenta-se restaurar o estado de um objeto para uma classe que possui versão diferente da qual o objeto foi serializado inicialmente. E isso pode significar diferentes versões de uma mesma classe “Fruta” por exemplo, ou diferentes versões por classes serem totalmente diferentes como: o estado de um objeto da classe “Fruta” ser restaurado para classe “Carro”.

Mas quando é necessário, conscientemente, recuperar o estado de um objeto, mesmo que houveram alterações na classe do qual foi gerado, é possível controlar manualmente o “serialVersionUID”. Para isto, basta declarar explicitamente na forma de atributo estático e final, um valor Long qualquer para corresponder à versão da classe. E assim, mesmo que houverem modificações na classe, a versão permenecerá a mesma, evitando as exceções da incompatiblidade de classes. No entanto, vale ressaltar que o versionamento não é um impecílio para o programador, mas sim muito importante em casos que deseja-se manter um controle sobre a compatibilidade de código na aplicação, definindo versões para classes No código da Listagem 7 encontra-se o exemplo do controle manual do “serialVersionUID”.

Listagem 7 - Uso manual do serialVersionUID

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | import java.io.Serializable;    public class Fruta implements Serializable{      private String nome;      private String cor;      static final long serialVersionUID = 123L; //indica a versao da classe        public Fruta(String nome, String cor) {          this.nome = nome;          this.cor = cor;      }      public String getCor() { return cor; }      public String getNome() { return nome; }      public void setCor(String cor) { this.cor = cor; }      public void setNome(String nome) { this.nome = nome; }  } |

O JDK fornece uma ferramenta que permite visualizar a versão de qualquer classe Java. Basta no terminal do sistema operacional utilizar o comando “serialver” seguido da classe qualificada, conforme exemplo da Listagem 8.

Listagem 8 - Verificação da versão de classe

|  |  |
| --- | --- |
| 1 | serialver pojo.Fruta |

A serialização acaba inflingindo alguns modelos de segurança em seu procedimento padrão, e criando então um novo problema. Quando um estado de objeto é persistido no sistema de arquivos ou enviado por rede, o fluxo binário pode ser detectado, analisado e obtidas informações sobre a classe e o estado do objeto. Por exemplo, se utilizado um editor de texto qualquer para abrir o arquivo persistido do estado do objeto da classe “Fruta” dos exemplos anteriores, é comum os atributos e outras informações aparecerem como um simples texto, total ou parcilamente legíveis, assim como demonstrado na Listagem 9.

Listagem 9 - Exemplo de leitura de arquivo binário de um estado de objeto persistido.

|  |  |
| --- | --- |
| 1  2 | ��##sr#  pojo.Fruta#######{###L##cort##Ljava/lang/String;L##nomeq#~##xpt##Vermelhat##Maça |

Com uma ferramenta menos simples seria possível aumentar a quantidade de informações legíveis, coletando uma maior quantia de dados.

No entando, o processo de serialização permite que haja uma certa personalização da serialização e deserialização de um objeto. Isso deve-se a possibilidade da sobrescrição dos métodos “writeObject” e “readObject” dentro da classe a ser serializada, como a classe exemplar “pojo.Fruta”. Esta sobrescrita permite um controle total sobre a serialização, pois as classes “java.io.ObjectOutputStream” e “java.io.ObjectInputStream”, ao serializar o estado de um objeto, primeiramente verificam se esse método foi sobrescrito na classe alvo e assim o utilizam, senão, será utilizado o protocolo padrão de serialização.

Realizando a sobrescrição dos métodos é possível criptografar os dados na serialização e descriptografar os dados na deserialização, por meio de algum algoritmo que o programador venha implementar. Assim os dados podem ser persistidos ou transmitidos por rede, sem que seja possível ler as partes mais importantes do estado de um objeto. Na listagem 10, a classe “pojo.Fruta” foi modificada dando origem a classe “pojo.FrutaSobrescrita” que possui adicionamente a sobrescrição dos métodos “writeObject” e “readObject”. Como este é apenas um exemplo, não foi escrito um verdadeiro algoritmo de criptografia, mas sim apenas uma troca no valor da String do campo “cor”, identificando que houveram alterações tanto no processo de serialização quanto no de deserialização. Na listagem 11 encontra-se a execução da serialização.

Listagem 10 - Sobrescrita dos métodos “writeObject” e “readObject”.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36 | import java.io.Serializable;    public class FrutaSobrescrita implements Serializable{      private String nome;      private String cor;      static final long serialVersionUID = 123L; //indica a versao da classe        public FrutaSobrescrita(String nome, String cor) {          this.nome = nome;          this.cor = cor;      }      public String getCor() { return cor; }      public String getNome() { return nome; }      public void setCor(String cor) { this.cor = cor; }      public void setNome(String nome) { this.nome = nome; }        private void writeObject(java.io.ObjectOutputStream stream) throws java.io.IOException      {          // criptografar os dados antes de liberar o stream          // EXEMPLO: setCor(criptografar(getCor()));          setCor(getCor()+ " > CRIPTOGRAFADO"); // apenas concatenação de string para indicar alteração            // liberar o stream          stream.defaultWriteObject();      }        private void readObject(java.io.ObjectInputStream stream) throws java.io.IOException, ClassNotFoundException      {          // liberar stream          stream.defaultReadObject();            // descriptografar os dados          // EXEMPLO: setCor(descriptografar(getCor()));          setCor(getCor()+ " > DESCRIPTOGRAFADO"); // apenas contatenação de string para indicar alteração      }  } |

Listagem 11 - Execução da serialização da classe “pojo.FrutaSobrescrita”.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | //serializa        Serializador s = new Serializador();        FrutaSobrescrita frutaSobrescrita = new FrutaSobrescrita("Maça", "Vermelha");         try {             s.serializar("/home/gustavo/frutaSobrescrita", frutaSobrescrita);         } catch (Exception ex) {             System.err.println("Falha ao serializar! - " + ex.toString());         }           //deserializa        Deserializador d = new Deserializador();         frutaSobrescrita = null;         try {             frutaSobrescrita = (FrutaSobrescrita) d.deserializar("/home/gustavo/frutaSobrescrita");         } catch (Exception ex) {             System.err.println("Falha ao deserializar! - " + ex.toString());         }         System.out.println("Sobrescrita: " + frutaSobrescrita.getNome() + " - " + frutaSobrescrita.getCor()); |

**5. CONCLUSÃO**

A serialização é um procedimento de simples execução por parte do programador. Com pouco tempo de codificação é possível implementar a persistencia de um estado de objeto para o sistema de arquivos ou de transmissão por rede. É possível reproduzir os exemplos criados neste artigo, em situações até mesmo mais elaboradas. A apresentação de alguns problemas e soluções trazidas por particularidades, abre o entendimento de que a serialização pode deixar de ser simples conforme for aplicada em diferentes contextos, e que ao mesmo tempo é muito flexível ao permitir ao programador implementar diferentes soluções.

[Assista ao vídeo Serialização/Deserialização de objetos com YAML](http://www.devmedia.com.br/serializacao-deserializacao-de-objetos-com-yaml-parte-1/8399)

**6. BIBLIOGRAFIA**

ORACLE - **java.io Interface Serializable** - Disponível em: <http://docs.oracle.com/javase/1.4.2/docs/api/java/io/Serializable.html>Acesso em: 11/03/2012

ORACLE - **Discover de Secrets of Java Serialization API** - Disponível em: <http://java.sun.com/developer/technicalArticles/Programming/serialization/> Acesso em: 11/03/2012

NEWARD, Ted - **5 coisas que você não sabia sobre... Serialização de Objetos Java** - Disponível em: <http://www.ibm.com/developerworks/br/library/j-5things1/index.html> Acesso em: 11/03/2012

DOWNS, Andrew - **Java Serialization** - Disponível em: <http://www.mactech.com/articles/mactech/Vol.14/14.04/JavaSerialization/index.html> Acesso em: 11/03/2012

PREISSLER, Luciano E.; GEYER, Cláudio F. R. - **Serialização em Java** - Disponível em: <http://www.inf.ufrgs.br/gppd/disc/cmp167/trabalhos/sem99-1/T1/luciano/final.htm> Acesso em: 11/03/2012

[[](http://www.linhadecodigo.com.br/colaborador.aspx?id=354)](http://www.linhadecodigo.com.br/colaborador.aspx?id=354)

[Everton Coimbra de Araújo - Desde 1987 atua na área de treinamento e desenvolvimento. Como Mestre em Ciência da Computação, é professor da UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ, Campus Medianeira, onde leciona disciplinas relacionadas ao desenvolvimento de aplicações web, com Java e .NET.   
  
É autor dos livros Desenvolvimento para WEB com Java, Orientação a Objetos com Java - Simples, Fácil e Eficiente, Algoritmo - Fundamento e Prática em sua terceira edição e dos livros Delphi - Implementação e Técnicas para Ambientes Virtuais e C++ Builder - Implementação e Técnicas para Ambientes Virtuais. Todos pela VisualBooks. Pode ser contactado através do e-mail everton@utfpr.edu.br ou evertoncoimbra@gmail.com.](http://www.linhadecodigo.com.br/colaborador.aspx?id=354)

 85 3 0 4

**Leia também**

* [Conhecendo a interface Map do Java](http://www.linhadecodigo.com.br/artigo/3670/conhecendo-a-interface-map-do-java.aspx)*Java*
* [Trabalhando com a Interface Set no Java](http://www.linhadecodigo.com.br/artigo/3669/trabalhando-com-a-interface-set-no-java.aspx)*Java*
* [Classes Wrappers em Java](http://www.linhadecodigo.com.br/artigo/3667/classes-wrappers-em-java.aspx)*Java*
* [ComboBoxModel em Java](http://www.linhadecodigo.com.br/artigo/3664/comboboxmodel-em-java.aspx)*Java*
* [Reduzindo a Mutabilidade em Java](http://www.linhadecodigo.com.br/artigo/3659/reduzindo-a-mutabilidade-em-java.aspx)*Java*

Publicidade

[](http://qconsp.com/?utm_source=devmedia&utm_medium=banner&utm_campaign=devmedia)

Linha de Código[Anuncie](http://www.web-03.net/linhadecodigo.asp)|[Fale Conosco](mailto:colabore@linhadecodigo.com.br)|[Publique](http://www.linhadecodigo.com.br/publique.aspx)

|  |  |
| --- | --- |
| http://www.linhadecodigo.com.br/artigos/img_artigos/EvertonCoimbraAraujo/Serializazao_Objetos/imagem001.jpg | Neste artigo apresento o Gustavo Rafael Valiati, que pode ser contatado pelo e-mail gustavovaliati@gmail.com. O Gustavo foi meu aluno em algumas disciplinas de sua gradução e está concluíndo o último período do curso de Tecnologia em Análise e Desenvolvimento de Sistemas da UTFPR campus Medianeira, neste primeiro semestre de 2012. |

**1. INTRODUÇÃO**

Nem sempre se deseja ter objetos ou estrutura de dados de um aplicativo, com um ciclo de vida que dura no máximo o tempo em que for armazenado em memória RAM. Ou seja, algumas vezes deseja-se guardá-los por um tempo indeterminado, sendo possível recuperá-los posteriormente. E neste caso, o termo “persistência” tem seu conceito definido.

Persistir dados, requer que estes possam ser serializáveis. A serialização possibilitará capturar o estado do objeto ou a estrutura de dados, e transformar em uma cadeia de bytes quando for necessário. Torna-se possível também, recuperar os bytes persistidos para fazer o processo inverso, e ter os dados de volta para a aplicação em execução.

Desta maneira, qualquer forma de dados que esteja sendo trabalhada na aplicação, e esta puder ser serializada, poderá ser passada para o meio físico, e ter o processo revertido.

A serialização não se limita apenas em possibilitar a gravação do dado em disco, mas permite que o mesmo seja transmitido por rede. Isto porquê a serialização cria um stream de bytes, que é requisito para transmissão de dados por rede.

Diversas linguagens possuem suporte à serialização, como: Java, C, C++, Python, PHP, .NET, entre outras. Neste artigo, será tratado apenas da serialização com Java, em sua implemetação básica, alguns problemas e soluções, com códigos para exemplificação.

**2. SERIALIZAÇÃO COM JAVA**

Serializar um objeto, em Java, só é possível caso sua classe esteja marcada como serializável. Essa marcação é feita com a simples implementação da interface “java.io.Serializable”. Essa interface não possui métodos para se implementar e nem atributos. Ainda não existiam as anotações no Java 1.1, versão que trouxe pela primeira vez a serialização ao Java, e desta maneira a implementação da interface era a melhor maneira de marcar uma classe.

A herança entre classes é naturalmente afetada pela serialização. Já que quando uma classe implementa a interface “Serializable”, toda classe que dela estender também estará implicitamente marcada como serializável

.

Para exemplo, será criada uma classe “pojo.Fruta” que contém dois atributos: nome e cor. No código exemplo (Listagem 1), pode-se notar a classe “Fruta” marcada para a serialização:

Listagem 1 - Marcando uma classe como serializável.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | import java.io.Serializable;    public class Fruta implements Serializable{      private String nome;      private String cor;        public Fruta(String nome, String cor) {          this.nome = nome;          this.cor = cor;      }        public String getCor() { return cor; }      public String getNome() { return nome; }      public void setCor(String cor) { this.cor = cor; }      public void setNome(String nome) { this.nome = nome; }  } |

Pode-se dizer que a implementação do processo de serialização e deserialização simples de um objeto, não possui um código difícil de se escrever. Isto porquê duas classes fazem a maior parte do trabalho, de maneira que com poucas linhas de código pode-se transformar um objeto em bytes e vice-versa. Essas duas classes são “java.io.ObjectInputStream” que recebe bytes para criar um objeto e “java.io.ObjectOutputStream” que cria uma cadeia de bytes a partir de um objeto.

**3. SERIALIZANDO E DESERIALIZANDO O ESTADO DE UM OBJETO**

Como exemplo, será persistido um objeto em disco. Para isto, além das classes “java.io.ObjectInputStream” e “java.io.ObjectOutputStream”, serão utilizadas duas outras que trabalham com arquivos, que são “java.io.FileInputStream” e “java.io.FileOutputStream”, que seguem a mesma lógica de InputStream e OutputStream, só que agora relacionado ao sistema de arquivos e não aos objetos.

Para serialização, deve-se criar um novo “java.io.FileOutputStream”, com o caminho (“path”) do arquivo onde se deseja armazenar o estado do objeto. Também deve ser criado um novo “java.io.ObjectOutputStream” que está ligado ao arquivo. E por último deve-se executar a escrita do objeto alvo, para dentro do objeto de stream que por sua vez está ligado ao arquivo destino. Após executados esses passos, é possível encontrar no sistema de arquivos um novo arquivo que contém os bytes correspondentes à descrição do estado do objeto. Na Listagem 2, encontra-se o código exemplo.

Listagem 2 - Serialização de um objeto.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | import java.io.FileOutputStream;  import java.io.ObjectOutputStream;    public class Serializador {        public Serializador() {    }        public void serializar(String path, Object obj) throws Exception {              FileOutputStream outFile = new FileOutputStream(path);      ObjectOutputStream s = new ObjectOutputStream(outFile);      s.writeObject(obj);      s.close();      }  } |

E para a deserialização, o processo é extremamente parecido com a serialização. Cria-se um novo “java.io.FileInputStream” relacionado ao arquivo onde o estado do objeto encontra-se persistido. Deve-se criar também um novo “java.io.ObjectInputStream” que estará vinculado ao arquivo de origem. Por último, executa-se a leitura do arquivo para a restauração do estado. Na Listagem 3 encontra-se o exemplo do código.

Listagem 3 - Deserializaçãode um objeto.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | import java.io.FileInputStream;  import java.io.ObjectInputStream;    public class Deserializador {        public Deserializador() {    }        public Object deserializar(String path) throws Exception {      FileInputStream inFile = new FileInputStream(path);      ObjectInputStream d = new ObjectInputStream(inFile);      Object o = d.readObject();      d.close();      return o;      }  } |

A Listagem 4, contém o exemplo da execução da serialização e em seguida da deserialização.

Listagem 4 - Serializando e Deserializando

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | import operacoes.Deserializador;  import operacoes.Serializador;  import pojo.Fruta;    public class Principal {      public static void main(String args[]){         //serializa         Serializador s = new Serializador();         Fruta fruta = new Fruta("Maça", "Vermelha");          try {              s.serializar("/home/gustavo/fruta", fruta);          } catch (Exception ex) {              System.err.println("Falha ao serializar! - " + ex.toString());          }            //deserializa          Deserializador d = new Deserializador();          fruta = null;          try {              fruta = (Fruta) d.deserializar("/home/gustavo/fruta");          } catch (Exception ex) {              System.err.println("Falha ao deserializar! - " + ex.toString());          }          System.out.println(fruta.getNome() + " - " + fruta.getCor());      }  } |

Na Listagem 5, encontra-se um exemplo de serialização e deserialização de um ArrayList da classe Fruta. É apenas mais um exemplo que demonstra a capacidade de se persistir qualquer objeto que seja serializável. Essa serialização só foi possível porque a classe “ArrayList” implementa a interface “Serializable” por padrão.

Listagem 5 - Serialização de ArrayList.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | //serialização e deserialização com ArrayList          ArrayList<Fruta> frutas = new ArrayList<Fruta>();          Fruta f1 = new Fruta("Laranja", "Amarela");          Fruta f2 = new Fruta("Abacate", "Verde");          Fruta f3 = new Fruta("Morango", "Vermelho");          frutas.add(f1);          frutas.add(f2);          frutas.add(f3);            try {              s.serializar("/home/gustavo/frutas", frutas);              frutas = null;              frutas = (ArrayList<Fruta>) d.deserializar("/home/gustavo/frutas");              for(Fruta f : frutas){                  System.out.println("ArrayList: " + f.getNome() + " - " + f.getCor());              }          } catch (Exception ex) {              System.err.println("Falha ao serializar ou deserializar! - " +                                 ex.toString());          } |

**4. PARTICULARIDADES DA SERIALIZAÇÃO**

Como dito, a serialização de um objeto em Java pode ser algo fácil de se fazer. Mas existem certas informações que são importantes e sabê-las pode evitar vários problemas.

Um dos problemas é que nem tudo pode ser serializado, como objetos das classes “Thread” e “Socket” e atributos estáticos. Os atributos estáticos estão vinculados à classe e não ao objeto. Assim quando se trata do estado do objeto, os campos estáticos não são considerados. Em relação às Threads, suas instâncias só são úteis enquanto estiverem em execução na JVM, e podem ser paradas/destruídas e iniciadas com instruções de sua classe. Não existe algo no estado destas instâncias que seja útil para se persistir, e por isso não há sentido em persistir tais instâncias para restaurar posteriormente.

Mas e se existir alguma classe em que seja necessário persisti-la, e esta possui uma instância da classe “Thread”? Neste caso, utilizando um recurso pode-se persistir a classe inteira, menos a instância thread. Este recurso, refere-se em indicar que tal instância thread é um objeto “Transient”. Na Listagem 6 encontra-se um exemplo da implementação com uso de “Transient”.

Listagem 6 - Uso do Transient

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | import java.io.Serializable;    public class ImplementacaoTransient implements Serializable, Runnable{        transient private Thread thread;      //marcado com "transient" para que sua instância não seja serializada        //Demais valores de estado serão serializados      private int valorParaPersistir;        public ImplementacaoTransient(int valorParaPersistir){          this.valorParaPersistir = valorParaPersistir;          thread = new Thread(this);          thread.start();      }        public void run()      {          while(true){              //código a ser executado pela Thread          }      }  } |

Vale ressaltar que, o que é serializado é apenas o estado do objeto, e não os métodos e a classe. No exemplo acima, apenas a instância do objeto “thread” e do objeto “valorParaPersistir” compunham o estado do objeto. Assim, evita-se que o sistema de serialização tente serializar um objeto não-serializável e gere erros. Após a deserialização de um objeto da classe “ImplementacaoTransient” este estará totalmente funcional.

Um outro problema que pode vir a ocorrer, se refere ao versionamento das classes. Toda classe possui um atributo oculto chamado “serialVersionUID” com um valor gerado automaticamente. Objetos criados levam consigo o “serialVersionUID” de sua classe, que indica sua versão. Toda vez que uma classe sofre alterações em seu código, um novo “serialVersionUID” é gerado. O problema com a serialização surge quando serializa-se o estado de um objeto e posteriormente tenta-se restaurar o estado para a aplicação, mas houveram mudanças no código da classe. Neste caso, uma exceção “java.io.InvalidClassException” é disparada, pois de fato, o estado do objeto corresponde a uma versão de classe do qual foi criado que já não existe mais. Portanto, isso só ocorre para evitar erros de programação, em que tenta-se restaurar o estado de um objeto para uma classe que possui versão diferente da qual o objeto foi serializado inicialmente. E isso pode significar diferentes versões de uma mesma classe “Fruta” por exemplo, ou diferentes versões por classes serem totalmente diferentes como: o estado de um objeto da classe “Fruta” ser restaurado para classe “Carro”.

Mas quando é necessário, conscientemente, recuperar o estado de um objeto, mesmo que houveram alterações na classe do qual foi gerado, é possível controlar manualmente o “serialVersionUID”. Para isto, basta declarar explicitamente na forma de atributo estático e final, um valor Long qualquer para corresponder à versão da classe. E assim, mesmo que houverem modificações na classe, a versão permenecerá a mesma, evitando as exceções da incompatiblidade de classes. No entanto, vale ressaltar que o versionamento não é um impecílio para o programador, mas sim muito importante em casos que deseja-se manter um controle sobre a compatibilidade de código na aplicação, definindo versões para classes No código da Listagem 7 encontra-se o exemplo do controle manual do “serialVersionUID”.

Listagem 7 - Uso manual do serialVersionUID

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | import java.io.Serializable;    public class Fruta implements Serializable{      private String nome;      private String cor;      static final long serialVersionUID = 123L; //indica a versao da classe        public Fruta(String nome, String cor) {          this.nome = nome;          this.cor = cor;      }      public String getCor() { return cor; }      public String getNome() { return nome; }      public void setCor(String cor) { this.cor = cor; }      public void setNome(String nome) { this.nome = nome; }  } |

O JDK fornece uma ferramenta que permite visualizar a versão de qualquer classe Java. Basta no terminal do sistema operacional utilizar o comando “serialver” seguido da classe qualificada, conforme exemplo da Listagem 8.

Listagem 8 - Verificação da versão de classe

|  |  |
| --- | --- |
| 1 | serialver pojo.Fruta |

A serialização acaba inflingindo alguns modelos de segurança em seu procedimento padrão, e criando então um novo problema. Quando um estado de objeto é persistido no sistema de arquivos ou enviado por rede, o fluxo binário pode ser detectado, analisado e obtidas informações sobre a classe e o estado do objeto. Por exemplo, se utilizado um editor de texto qualquer para abrir o arquivo persistido do estado do objeto da classe “Fruta” dos exemplos anteriores, é comum os atributos e outras informações aparecerem como um simples texto, total ou parcilamente legíveis, assim como demonstrado na Listagem 9.

Listagem 9 - Exemplo de leitura de arquivo binário de um estado de objeto persistido.

|  |  |
| --- | --- |
| 1  2 | ��##sr#  pojo.Fruta#######{###L##cort##Ljava/lang/String;L##nomeq#~##xpt##Vermelhat##Maça |

Com uma ferramenta menos simples seria possível aumentar a quantidade de informações legíveis, coletando uma maior quantia de dados.

No entando, o processo de serialização permite que haja uma certa personalização da serialização e deserialização de um objeto. Isso deve-se a possibilidade da sobrescrição dos métodos “writeObject” e “readObject” dentro da classe a ser serializada, como a classe exemplar “pojo.Fruta”. Esta sobrescrita permite um controle total sobre a serialização, pois as classes “java.io.ObjectOutputStream” e “java.io.ObjectInputStream”, ao serializar o estado de um objeto, primeiramente verificam se esse método foi sobrescrito na classe alvo e assim o utilizam, senão, será utilizado o protocolo padrão de serialização.

Realizando a sobrescrição dos métodos é possível criptografar os dados na serialização e descriptografar os dados na deserialização, por meio de algum algoritmo que o programador venha implementar. Assim os dados podem ser persistidos ou transmitidos por rede, sem que seja possível ler as partes mais importantes do estado de um objeto. Na listagem 10, a classe “pojo.Fruta” foi modificada dando origem a classe “pojo.FrutaSobrescrita” que possui adicionamente a sobrescrição dos métodos “writeObject” e “readObject”. Como este é apenas um exemplo, não foi escrito um verdadeiro algoritmo de criptografia, mas sim apenas uma troca no valor da String do campo “cor”, identificando que houveram alterações tanto no processo de serialização quanto no de deserialização. Na listagem 11 encontra-se a execução da serialização.

Listagem 10 - Sobrescrita dos métodos “writeObject” e “readObject”.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36 | import java.io.Serializable;    public class FrutaSobrescrita implements Serializable{      private String nome;      private String cor;      static final long serialVersionUID = 123L; //indica a versao da classe        public FrutaSobrescrita(String nome, String cor) {          this.nome = nome;          this.cor = cor;      }      public String getCor() { return cor; }      public String getNome() { return nome; }      public void setCor(String cor) { this.cor = cor; }      public void setNome(String nome) { this.nome = nome; }        private void writeObject(java.io.ObjectOutputStream stream) throws java.io.IOException      {          // criptografar os dados antes de liberar o stream          // EXEMPLO: setCor(criptografar(getCor()));          setCor(getCor()+ " > CRIPTOGRAFADO"); // apenas concatenação de string para indicar alteração            // liberar o stream          stream.defaultWriteObject();      }        private void readObject(java.io.ObjectInputStream stream) throws java.io.IOException, ClassNotFoundException      {          // liberar stream          stream.defaultReadObject();            // descriptografar os dados          // EXEMPLO: setCor(descriptografar(getCor()));          setCor(getCor()+ " > DESCRIPTOGRAFADO"); // apenas contatenação de string para indicar alteração      }  } |

Listagem 11 - Execução da serialização da classe “pojo.FrutaSobrescrita”.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | //serializa        Serializador s = new Serializador();        FrutaSobrescrita frutaSobrescrita = new FrutaSobrescrita("Maça", "Vermelha");         try {             s.serializar("/home/gustavo/frutaSobrescrita", frutaSobrescrita);         } catch (Exception ex) {             System.err.println("Falha ao serializar! - " + ex.toString());         }           //deserializa        Deserializador d = new Deserializador();         frutaSobrescrita = null;         try {             frutaSobrescrita = (FrutaSobrescrita) d.deserializar("/home/gustavo/frutaSobrescrita");         } catch (Exception ex) {             System.err.println("Falha ao deserializar! - " + ex.toString());         }         System.out.println("Sobrescrita: " + frutaSobrescrita.getNome() + " - " + frutaSobrescrita.getCor()); |

**5. CONCLUSÃO**

A serialização é um procedimento de simples execução por parte do programador. Com pouco tempo de codificação é possível implementar a persistencia de um estado de objeto para o sistema de arquivos ou de transmissão por rede. É possível reproduzir os exemplos criados neste artigo, em situações até mesmo mais elaboradas. A apresentação de alguns problemas e soluções trazidas por particularidades, abre o entendimento de que a serialização pode deixar de ser simples conforme for aplicada em diferentes contextos, e que ao mesmo tempo é muito flexível ao permitir ao programador implementar diferentes soluções.

[Assista ao vídeo Serialização/Deserialização de objetos com YAML](http://www.devmedia.com.br/serializacao-deserializacao-de-objetos-com-yaml-parte-1/8399)

**6. BIBLIOGRAFIA**

ORACLE - **java.io Interface Serializable** - Disponível em: <http://docs.oracle.com/javase/1.4.2/docs/api/java/io/Serializable.html>Acesso em: 11/03/2012

ORACLE - **Discover de Secrets of Java Serialization API** - Disponível em: <http://java.sun.com/developer/technicalArticles/Programming/serialization/> Acesso em: 11/03/2012

NEWARD, Ted - **5 coisas que você não sabia sobre... Serialização de Objetos Java** - Disponível em: <http://www.ibm.com/developerworks/br/library/j-5things1/index.html> Acesso em: 11/03/2012

DOWNS, Andrew - **Java Serialization** - Disponível em: <http://www.mactech.com/articles/mactech/Vol.14/14.04/JavaSerialization/index.html> Acesso em: 11/03/2012

PREISSLER, Luciano E.; GEYER, Cláudio F. R. - **Serialização em Java** - Disponível em: <http://www.inf.ufrgs.br/gppd/disc/cmp167/trabalhos/sem99-1/T1/luciano/final.htm> Acesso em: 11/03/2012

[[](http://www.linhadecodigo.com.br/colaborador.aspx?id=354)](http://www.linhadecodigo.com.br/colaborador.aspx?id=354)

[Everton Coimbra de Araújo - Desde 1987 atua na área de treinamento e desenvolvimento. Como Mestre em Ciência da Computação, é professor da UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ, Campus Medianeira, onde leciona disciplinas relacionadas ao desenvolvimento de aplicações web, com Java e .NET.   
  
É autor dos livros Desenvolvimento para WEB com Java, Orientação a Objetos com Java - Simples, Fácil e Eficiente, Algoritmo - Fundamento e Prática em sua terceira edição e dos livros Delphi - Implementação e Técnicas para Ambientes Virtuais e C++ Builder - Implementação e Técnicas para Ambientes Virtuais. Todos pela VisualBooks. Pode ser contactado através do e-mail everton@utfpr.edu.br ou evertoncoimbra@gmail.com.](http://www.linhadecodigo.com.br/colaborador.aspx?id=354)

 85 3 0 4

**Leia também**

* [Conhecendo a interface Map do Java](http://www.linhadecodigo.com.br/artigo/3670/conhecendo-a-interface-map-do-java.aspx)*Java*
* [Trabalhando com a Interface Set no Java](http://www.linhadecodigo.com.br/artigo/3669/trabalhando-com-a-interface-set-no-java.aspx)*Java*
* [Classes Wrappers em Java](http://www.linhadecodigo.com.br/artigo/3667/classes-wrappers-em-java.aspx)*Java*
* [ComboBoxModel em Java](http://www.linhadecodigo.com.br/artigo/3664/comboboxmodel-em-java.aspx)*Java*
* [Reduzindo a Mutabilidade em Java](http://www.linhadecodigo.com.br/artigo/3659/reduzindo-a-mutabilidade-em-java.aspx)*Java*

Publicidade

[](http://qconsp.com/?utm_source=devmedia&utm_medium=banner&utm_campaign=devmedia)

Linha de Código[Anuncie](http://www.web-03.net/linhadecodigo.asp)|[Fale Conosco](mailto:colabore@linhadecodigo.com.br)|[Publique](http://www.linhadecodigo.com.br/publique.aspx)

|  |  |
| --- | --- |
| http://www.linhadecodigo.com.br/artigos/img_artigos/EvertonCoimbraAraujo/Serializazao_Objetos/imagem001.jpg | Neste artigo apresento o Gustavo Rafael Valiati, que pode ser contatado pelo e-mail gustavovaliati@gmail.com. O Gustavo foi meu aluno em algumas disciplinas de sua gradução e está concluíndo o último período do curso de Tecnologia em Análise e Desenvolvimento de Sistemas da UTFPR campus Medianeira, neste primeiro semestre de 2012. |

**1. INTRODUÇÃO**

Nem sempre se deseja ter objetos ou estrutura de dados de um aplicativo, com um ciclo de vida que dura no máximo o tempo em que for armazenado em memória RAM. Ou seja, algumas vezes deseja-se guardá-los por um tempo indeterminado, sendo possível recuperá-los posteriormente. E neste caso, o termo “persistência” tem seu conceito definido.

Persistir dados, requer que estes possam ser serializáveis. A serialização possibilitará capturar o estado do objeto ou a estrutura de dados, e transformar em uma cadeia de bytes quando for necessário. Torna-se possível também, recuperar os bytes persistidos para fazer o processo inverso, e ter os dados de volta para a aplicação em execução.

Desta maneira, qualquer forma de dados que esteja sendo trabalhada na aplicação, e esta puder ser serializada, poderá ser passada para o meio físico, e ter o processo revertido.

A serialização não se limita apenas em possibilitar a gravação do dado em disco, mas permite que o mesmo seja transmitido por rede. Isto porquê a serialização cria um stream de bytes, que é requisito para transmissão de dados por rede.

Diversas linguagens possuem suporte à serialização, como: Java, C, C++, Python, PHP, .NET, entre outras. Neste artigo, será tratado apenas da serialização com Java, em sua implemetação básica, alguns problemas e soluções, com códigos para exemplificação.

**2. SERIALIZAÇÃO COM JAVA**

Serializar um objeto, em Java, só é possível caso sua classe esteja marcada como serializável. Essa marcação é feita com a simples implementação da interface “java.io.Serializable”. Essa interface não possui métodos para se implementar e nem atributos. Ainda não existiam as anotações no Java 1.1, versão que trouxe pela primeira vez a serialização ao Java, e desta maneira a implementação da interface era a melhor maneira de marcar uma classe.

A herança entre classes é naturalmente afetada pela serialização. Já que quando uma classe implementa a interface “Serializable”, toda classe que dela estender também estará implicitamente marcada como serializável

.

Para exemplo, será criada uma classe “pojo.Fruta” que contém dois atributos: nome e cor. No código exemplo (Listagem 1), pode-se notar a classe “Fruta” marcada para a serialização:

Listagem 1 - Marcando uma classe como serializável.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | import java.io.Serializable;    public class Fruta implements Serializable{      private String nome;      private String cor;        public Fruta(String nome, String cor) {          this.nome = nome;          this.cor = cor;      }        public String getCor() { return cor; }      public String getNome() { return nome; }      public void setCor(String cor) { this.cor = cor; }      public void setNome(String nome) { this.nome = nome; }  } |

Pode-se dizer que a implementação do processo de serialização e deserialização simples de um objeto, não possui um código difícil de se escrever. Isto porquê duas classes fazem a maior parte do trabalho, de maneira que com poucas linhas de código pode-se transformar um objeto em bytes e vice-versa. Essas duas classes são “java.io.ObjectInputStream” que recebe bytes para criar um objeto e “java.io.ObjectOutputStream” que cria uma cadeia de bytes a partir de um objeto.

**3. SERIALIZANDO E DESERIALIZANDO O ESTADO DE UM OBJETO**

Como exemplo, será persistido um objeto em disco. Para isto, além das classes “java.io.ObjectInputStream” e “java.io.ObjectOutputStream”, serão utilizadas duas outras que trabalham com arquivos, que são “java.io.FileInputStream” e “java.io.FileOutputStream”, que seguem a mesma lógica de InputStream e OutputStream, só que agora relacionado ao sistema de arquivos e não aos objetos.

Para serialização, deve-se criar um novo “java.io.FileOutputStream”, com o caminho (“path”) do arquivo onde se deseja armazenar o estado do objeto. Também deve ser criado um novo “java.io.ObjectOutputStream” que está ligado ao arquivo. E por último deve-se executar a escrita do objeto alvo, para dentro do objeto de stream que por sua vez está ligado ao arquivo destino. Após executados esses passos, é possível encontrar no sistema de arquivos um novo arquivo que contém os bytes correspondentes à descrição do estado do objeto. Na Listagem 2, encontra-se o código exemplo.

Listagem 2 - Serialização de um objeto.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | import java.io.FileOutputStream;  import java.io.ObjectOutputStream;    public class Serializador {        public Serializador() {    }        public void serializar(String path, Object obj) throws Exception {              FileOutputStream outFile = new FileOutputStream(path);      ObjectOutputStream s = new ObjectOutputStream(outFile);      s.writeObject(obj);      s.close();      }  } |

E para a deserialização, o processo é extremamente parecido com a serialização. Cria-se um novo “java.io.FileInputStream” relacionado ao arquivo onde o estado do objeto encontra-se persistido. Deve-se criar também um novo “java.io.ObjectInputStream” que estará vinculado ao arquivo de origem. Por último, executa-se a leitura do arquivo para a restauração do estado. Na Listagem 3 encontra-se o exemplo do código.

Listagem 3 - Deserializaçãode um objeto.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | import java.io.FileInputStream;  import java.io.ObjectInputStream;    public class Deserializador {        public Deserializador() {    }        public Object deserializar(String path) throws Exception {      FileInputStream inFile = new FileInputStream(path);      ObjectInputStream d = new ObjectInputStream(inFile);      Object o = d.readObject();      d.close();      return o;      }  } |

A Listagem 4, contém o exemplo da execução da serialização e em seguida da deserialização.

Listagem 4 - Serializando e Deserializando

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | import operacoes.Deserializador;  import operacoes.Serializador;  import pojo.Fruta;    public class Principal {      public static void main(String args[]){         //serializa         Serializador s = new Serializador();         Fruta fruta = new Fruta("Maça", "Vermelha");          try {              s.serializar("/home/gustavo/fruta", fruta);          } catch (Exception ex) {              System.err.println("Falha ao serializar! - " + ex.toString());          }            //deserializa          Deserializador d = new Deserializador();          fruta = null;          try {              fruta = (Fruta) d.deserializar("/home/gustavo/fruta");          } catch (Exception ex) {              System.err.println("Falha ao deserializar! - " + ex.toString());          }          System.out.println(fruta.getNome() + " - " + fruta.getCor());      }  } |

Na Listagem 5, encontra-se um exemplo de serialização e deserialização de um ArrayList da classe Fruta. É apenas mais um exemplo que demonstra a capacidade de se persistir qualquer objeto que seja serializável. Essa serialização só foi possível porque a classe “ArrayList” implementa a interface “Serializable” por padrão.

Listagem 5 - Serialização de ArrayList.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | //serialização e deserialização com ArrayList          ArrayList<Fruta> frutas = new ArrayList<Fruta>();          Fruta f1 = new Fruta("Laranja", "Amarela");          Fruta f2 = new Fruta("Abacate", "Verde");          Fruta f3 = new Fruta("Morango", "Vermelho");          frutas.add(f1);          frutas.add(f2);          frutas.add(f3);            try {              s.serializar("/home/gustavo/frutas", frutas);              frutas = null;              frutas = (ArrayList<Fruta>) d.deserializar("/home/gustavo/frutas");              for(Fruta f : frutas){                  System.out.println("ArrayList: " + f.getNome() + " - " + f.getCor());              }          } catch (Exception ex) {              System.err.println("Falha ao serializar ou deserializar! - " +                                 ex.toString());          } |

**4. PARTICULARIDADES DA SERIALIZAÇÃO**

Como dito, a serialização de um objeto em Java pode ser algo fácil de se fazer. Mas existem certas informações que são importantes e sabê-las pode evitar vários problemas.

Um dos problemas é que nem tudo pode ser serializado, como objetos das classes “Thread” e “Socket” e atributos estáticos. Os atributos estáticos estão vinculados à classe e não ao objeto. Assim quando se trata do estado do objeto, os campos estáticos não são considerados. Em relação às Threads, suas instâncias só são úteis enquanto estiverem em execução na JVM, e podem ser paradas/destruídas e iniciadas com instruções de sua classe. Não existe algo no estado destas instâncias que seja útil para se persistir, e por isso não há sentido em persistir tais instâncias para restaurar posteriormente.

Mas e se existir alguma classe em que seja necessário persisti-la, e esta possui uma instância da classe “Thread”? Neste caso, utilizando um recurso pode-se persistir a classe inteira, menos a instância thread. Este recurso, refere-se em indicar que tal instância thread é um objeto “Transient”. Na Listagem 6 encontra-se um exemplo da implementação com uso de “Transient”.

Listagem 6 - Uso do Transient

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | import java.io.Serializable;    public class ImplementacaoTransient implements Serializable, Runnable{        transient private Thread thread;      //marcado com "transient" para que sua instância não seja serializada        //Demais valores de estado serão serializados      private int valorParaPersistir;        public ImplementacaoTransient(int valorParaPersistir){          this.valorParaPersistir = valorParaPersistir;          thread = new Thread(this);          thread.start();      }        public void run()      {          while(true){              //código a ser executado pela Thread          }      }  } |

Vale ressaltar que, o que é serializado é apenas o estado do objeto, e não os métodos e a classe. No exemplo acima, apenas a instância do objeto “thread” e do objeto “valorParaPersistir” compunham o estado do objeto. Assim, evita-se que o sistema de serialização tente serializar um objeto não-serializável e gere erros. Após a deserialização de um objeto da classe “ImplementacaoTransient” este estará totalmente funcional.

Um outro problema que pode vir a ocorrer, se refere ao versionamento das classes. Toda classe possui um atributo oculto chamado “serialVersionUID” com um valor gerado automaticamente. Objetos criados levam consigo o “serialVersionUID” de sua classe, que indica sua versão. Toda vez que uma classe sofre alterações em seu código, um novo “serialVersionUID” é gerado. O problema com a serialização surge quando serializa-se o estado de um objeto e posteriormente tenta-se restaurar o estado para a aplicação, mas houveram mudanças no código da classe. Neste caso, uma exceção “java.io.InvalidClassException” é disparada, pois de fato, o estado do objeto corresponde a uma versão de classe do qual foi criado que já não existe mais. Portanto, isso só ocorre para evitar erros de programação, em que tenta-se restaurar o estado de um objeto para uma classe que possui versão diferente da qual o objeto foi serializado inicialmente. E isso pode significar diferentes versões de uma mesma classe “Fruta” por exemplo, ou diferentes versões por classes serem totalmente diferentes como: o estado de um objeto da classe “Fruta” ser restaurado para classe “Carro”.

Mas quando é necessário, conscientemente, recuperar o estado de um objeto, mesmo que houveram alterações na classe do qual foi gerado, é possível controlar manualmente o “serialVersionUID”. Para isto, basta declarar explicitamente na forma de atributo estático e final, um valor Long qualquer para corresponder à versão da classe. E assim, mesmo que houverem modificações na classe, a versão permenecerá a mesma, evitando as exceções da incompatiblidade de classes. No entanto, vale ressaltar que o versionamento não é um impecílio para o programador, mas sim muito importante em casos que deseja-se manter um controle sobre a compatibilidade de código na aplicação, definindo versões para classes No código da Listagem 7 encontra-se o exemplo do controle manual do “serialVersionUID”.

Listagem 7 - Uso manual do serialVersionUID

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | import java.io.Serializable;    public class Fruta implements Serializable{      private String nome;      private String cor;      static final long serialVersionUID = 123L; //indica a versao da classe        public Fruta(String nome, String cor) {          this.nome = nome;          this.cor = cor;      }      public String getCor() { return cor; }      public String getNome() { return nome; }      public void setCor(String cor) { this.cor = cor; }      public void setNome(String nome) { this.nome = nome; }  } |

O JDK fornece uma ferramenta que permite visualizar a versão de qualquer classe Java. Basta no terminal do sistema operacional utilizar o comando “serialver” seguido da classe qualificada, conforme exemplo da Listagem 8.

Listagem 8 - Verificação da versão de classe

|  |  |
| --- | --- |
| 1 | serialver pojo.Fruta |

A serialização acaba inflingindo alguns modelos de segurança em seu procedimento padrão, e criando então um novo problema. Quando um estado de objeto é persistido no sistema de arquivos ou enviado por rede, o fluxo binário pode ser detectado, analisado e obtidas informações sobre a classe e o estado do objeto. Por exemplo, se utilizado um editor de texto qualquer para abrir o arquivo persistido do estado do objeto da classe “Fruta” dos exemplos anteriores, é comum os atributos e outras informações aparecerem como um simples texto, total ou parcilamente legíveis, assim como demonstrado na Listagem 9.

Listagem 9 - Exemplo de leitura de arquivo binário de um estado de objeto persistido.

|  |  |
| --- | --- |
| 1  2 | ��##sr#  pojo.Fruta#######{###L##cort##Ljava/lang/String;L##nomeq#~##xpt##Vermelhat##Maça |

Com uma ferramenta menos simples seria possível aumentar a quantidade de informações legíveis, coletando uma maior quantia de dados.

No entando, o processo de serialização permite que haja uma certa personalização da serialização e deserialização de um objeto. Isso deve-se a possibilidade da sobrescrição dos métodos “writeObject” e “readObject” dentro da classe a ser serializada, como a classe exemplar “pojo.Fruta”. Esta sobrescrita permite um controle total sobre a serialização, pois as classes “java.io.ObjectOutputStream” e “java.io.ObjectInputStream”, ao serializar o estado de um objeto, primeiramente verificam se esse método foi sobrescrito na classe alvo e assim o utilizam, senão, será utilizado o protocolo padrão de serialização.

Realizando a sobrescrição dos métodos é possível criptografar os dados na serialização e descriptografar os dados na deserialização, por meio de algum algoritmo que o programador venha implementar. Assim os dados podem ser persistidos ou transmitidos por rede, sem que seja possível ler as partes mais importantes do estado de um objeto. Na listagem 10, a classe “pojo.Fruta” foi modificada dando origem a classe “pojo.FrutaSobrescrita” que possui adicionamente a sobrescrição dos métodos “writeObject” e “readObject”. Como este é apenas um exemplo, não foi escrito um verdadeiro algoritmo de criptografia, mas sim apenas uma troca no valor da String do campo “cor”, identificando que houveram alterações tanto no processo de serialização quanto no de deserialização. Na listagem 11 encontra-se a execução da serialização.

Listagem 10 - Sobrescrita dos métodos “writeObject” e “readObject”.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36 | import java.io.Serializable;    public class FrutaSobrescrita implements Serializable{      private String nome;      private String cor;      static final long serialVersionUID = 123L; //indica a versao da classe        public FrutaSobrescrita(String nome, String cor) {          this.nome = nome;          this.cor = cor;      }      public String getCor() { return cor; }      public String getNome() { return nome; }      public void setCor(String cor) { this.cor = cor; }      public void setNome(String nome) { this.nome = nome; }        private void writeObject(java.io.ObjectOutputStream stream) throws java.io.IOException      {          // criptografar os dados antes de liberar o stream          // EXEMPLO: setCor(criptografar(getCor()));          setCor(getCor()+ " > CRIPTOGRAFADO"); // apenas concatenação de string para indicar alteração            // liberar o stream          stream.defaultWriteObject();      }        private void readObject(java.io.ObjectInputStream stream) throws java.io.IOException, ClassNotFoundException      {          // liberar stream          stream.defaultReadObject();            // descriptografar os dados          // EXEMPLO: setCor(descriptografar(getCor()));          setCor(getCor()+ " > DESCRIPTOGRAFADO"); // apenas contatenação de string para indicar alteração      }  } |

Listagem 11 - Execução da serialização da classe “pojo.FrutaSobrescrita”.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | //serializa        Serializador s = new Serializador();        FrutaSobrescrita frutaSobrescrita = new FrutaSobrescrita("Maça", "Vermelha");         try {             s.serializar("/home/gustavo/frutaSobrescrita", frutaSobrescrita);         } catch (Exception ex) {             System.err.println("Falha ao serializar! - " + ex.toString());         }           //deserializa        Deserializador d = new Deserializador();         frutaSobrescrita = null;         try {             frutaSobrescrita = (FrutaSobrescrita) d.deserializar("/home/gustavo/frutaSobrescrita");         } catch (Exception ex) {             System.err.println("Falha ao deserializar! - " + ex.toString());         }         System.out.println("Sobrescrita: " + frutaSobrescrita.getNome() + " - " + frutaSobrescrita.getCor()); |

**5. CONCLUSÃO**

A serialização é um procedimento de simples execução por parte do programador. Com pouco tempo de codificação é possível implementar a persistencia de um estado de objeto para o sistema de arquivos ou de transmissão por rede. É possível reproduzir os exemplos criados neste artigo, em situações até mesmo mais elaboradas. A apresentação de alguns problemas e soluções trazidas por particularidades, abre o entendimento de que a serialização pode deixar de ser simples conforme for aplicada em diferentes contextos, e que ao mesmo tempo é muito flexível ao permitir ao programador implementar diferentes soluções.

[Assista ao vídeo Serialização/Deserialização de objetos com YAML](http://www.devmedia.com.br/serializacao-deserializacao-de-objetos-com-yaml-parte-1/8399)

**6. BIBLIOGRAFIA**

ORACLE - **java.io Interface Serializable** - Disponível em: <http://docs.oracle.com/javase/1.4.2/docs/api/java/io/Serializable.html>Acesso em: 11/03/2012

ORACLE - **Discover de Secrets of Java Serialization API** - Disponível em: <http://java.sun.com/developer/technicalArticles/Programming/serialization/> Acesso em: 11/03/2012

NEWARD, Ted - **5 coisas que você não sabia sobre... Serialização de Objetos Java** - Disponível em: <http://www.ibm.com/developerworks/br/library/j-5things1/index.html> Acesso em: 11/03/2012

DOWNS, Andrew - **Java Serialization** - Disponível em: <http://www.mactech.com/articles/mactech/Vol.14/14.04/JavaSerialization/index.html> Acesso em: 11/03/2012

PREISSLER, Luciano E.; GEYER, Cláudio F. R. - **Serialização em Java** - Disponível em: <http://www.inf.ufrgs.br/gppd/disc/cmp167/trabalhos/sem99-1/T1/luciano/final.htm> Acesso em: 11/03/2012

[[](http://www.linhadecodigo.com.br/colaborador.aspx?id=354)](http://www.linhadecodigo.com.br/colaborador.aspx?id=354)

[Everton Coimbra de Araújo - Desde 1987 atua na área de treinamento e desenvolvimento. Como Mestre em Ciência da Computação, é professor da UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ, Campus Medianeira, onde leciona disciplinas relacionadas ao desenvolvimento de aplicações web, com Java e .NET.   
  
É autor dos livros Desenvolvimento para WEB com Java, Orientação a Objetos com Java - Simples, Fácil e Eficiente, Algoritmo - Fundamento e Prática em sua terceira edição e dos livros Delphi - Implementação e Técnicas para Ambientes Virtuais e C++ Builder - Implementação e Técnicas para Ambientes Virtuais. Todos pela VisualBooks. Pode ser contactado através do e-mail everton@utfpr.edu.br ou evertoncoimbra@gmail.com.](http://www.linhadecodigo.com.br/colaborador.aspx?id=354)

 85 3 0 4

**Leia também**

* [Conhecendo a interface Map do Java](http://www.linhadecodigo.com.br/artigo/3670/conhecendo-a-interface-map-do-java.aspx)*Java*
* [Trabalhando com a Interface Set no Java](http://www.linhadecodigo.com.br/artigo/3669/trabalhando-com-a-interface-set-no-java.aspx)*Java*
* [Classes Wrappers em Java](http://www.linhadecodigo.com.br/artigo/3667/classes-wrappers-em-java.aspx)*Java*
* [ComboBoxModel em Java](http://www.linhadecodigo.com.br/artigo/3664/comboboxmodel-em-java.aspx)*Java*
* [Reduzindo a Mutabilidade em Java](http://www.linhadecodigo.com.br/artigo/3659/reduzindo-a-mutabilidade-em-java.aspx)*Java*

Publicidade

[](http://qconsp.com/?utm_source=devmedia&utm_medium=banner&utm_campaign=devmedia)

Linha de Código[Anuncie](http://www.web-03.net/linhadecodigo.asp)|[Fale Conosco](mailto:colabore@linhadecodigo.com.br)|[Publique](http://www.linhadecodigo.com.br/publique.aspx)

|  |  |
| --- | --- |
| http://www.linhadecodigo.com.br/artigos/img_artigos/EvertonCoimbraAraujo/Serializazao_Objetos/imagem001.jpg | Neste artigo apresento o Gustavo Rafael Valiati, que pode ser contatado pelo e-mail gustavovaliati@gmail.com. O Gustavo foi meu aluno em algumas disciplinas de sua gradução e está concluíndo o último período do curso de Tecnologia em Análise e Desenvolvimento de Sistemas da UTFPR campus Medianeira, neste primeiro semestre de 2012. |

**1. INTRODUÇÃO**

Nem sempre se deseja ter objetos ou estrutura de dados de um aplicativo, com um ciclo de vida que dura no máximo o tempo em que for armazenado em memória RAM. Ou seja, algumas vezes deseja-se guardá-los por um tempo indeterminado, sendo possível recuperá-los posteriormente. E neste caso, o termo “persistência” tem seu conceito definido.

Persistir dados, requer que estes possam ser serializáveis. A serialização possibilitará capturar o estado do objeto ou a estrutura de dados, e transformar em uma cadeia de bytes quando for necessário. Torna-se possível também, recuperar os bytes persistidos para fazer o processo inverso, e ter os dados de volta para a aplicação em execução.

Desta maneira, qualquer forma de dados que esteja sendo trabalhada na aplicação, e esta puder ser serializada, poderá ser passada para o meio físico, e ter o processo revertido.

A serialização não se limita apenas em possibilitar a gravação do dado em disco, mas permite que o mesmo seja transmitido por rede. Isto porquê a serialização cria um stream de bytes, que é requisito para transmissão de dados por rede.

Diversas linguagens possuem suporte à serialização, como: Java, C, C++, Python, PHP, .NET, entre outras. Neste artigo, será tratado apenas da serialização com Java, em sua implemetação básica, alguns problemas e soluções, com códigos para exemplificação.

**2. SERIALIZAÇÃO COM JAVA**

Serializar um objeto, em Java, só é possível caso sua classe esteja marcada como serializável. Essa marcação é feita com a simples implementação da interface “java.io.Serializable”. Essa interface não possui métodos para se implementar e nem atributos. Ainda não existiam as anotações no Java 1.1, versão que trouxe pela primeira vez a serialização ao Java, e desta maneira a implementação da interface era a melhor maneira de marcar uma classe.

A herança entre classes é naturalmente afetada pela serialização. Já que quando uma classe implementa a interface “Serializable”, toda classe que dela estender também estará implicitamente marcada como serializável

.

Para exemplo, será criada uma classe “pojo.Fruta” que contém dois atributos: nome e cor. No código exemplo (Listagem 1), pode-se notar a classe “Fruta” marcada para a serialização:

Listagem 1 - Marcando uma classe como serializável.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | import java.io.Serializable;    public class Fruta implements Serializable{      private String nome;      private String cor;        public Fruta(String nome, String cor) {          this.nome = nome;          this.cor = cor;      }        public String getCor() { return cor; }      public String getNome() { return nome; }      public void setCor(String cor) { this.cor = cor; }      public void setNome(String nome) { this.nome = nome; }  } |

Pode-se dizer que a implementação do processo de serialização e deserialização simples de um objeto, não possui um código difícil de se escrever. Isto porquê duas classes fazem a maior parte do trabalho, de maneira que com poucas linhas de código pode-se transformar um objeto em bytes e vice-versa. Essas duas classes são “java.io.ObjectInputStream” que recebe bytes para criar um objeto e “java.io.ObjectOutputStream” que cria uma cadeia de bytes a partir de um objeto.

**3. SERIALIZANDO E DESERIALIZANDO O ESTADO DE UM OBJETO**

Como exemplo, será persistido um objeto em disco. Para isto, além das classes “java.io.ObjectInputStream” e “java.io.ObjectOutputStream”, serão utilizadas duas outras que trabalham com arquivos, que são “java.io.FileInputStream” e “java.io.FileOutputStream”, que seguem a mesma lógica de InputStream e OutputStream, só que agora relacionado ao sistema de arquivos e não aos objetos.

Para serialização, deve-se criar um novo “java.io.FileOutputStream”, com o caminho (“path”) do arquivo onde se deseja armazenar o estado do objeto. Também deve ser criado um novo “java.io.ObjectOutputStream” que está ligado ao arquivo. E por último deve-se executar a escrita do objeto alvo, para dentro do objeto de stream que por sua vez está ligado ao arquivo destino. Após executados esses passos, é possível encontrar no sistema de arquivos um novo arquivo que contém os bytes correspondentes à descrição do estado do objeto. Na Listagem 2, encontra-se o código exemplo.

Listagem 2 - Serialização de um objeto.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | import java.io.FileOutputStream;  import java.io.ObjectOutputStream;    public class Serializador {        public Serializador() {    }        public void serializar(String path, Object obj) throws Exception {              FileOutputStream outFile = new FileOutputStream(path);      ObjectOutputStream s = new ObjectOutputStream(outFile);      s.writeObject(obj);      s.close();      }  } |

E para a deserialização, o processo é extremamente parecido com a serialização. Cria-se um novo “java.io.FileInputStream” relacionado ao arquivo onde o estado do objeto encontra-se persistido. Deve-se criar também um novo “java.io.ObjectInputStream” que estará vinculado ao arquivo de origem. Por último, executa-se a leitura do arquivo para a restauração do estado. Na Listagem 3 encontra-se o exemplo do código.

Listagem 3 - Deserializaçãode um objeto.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | import java.io.FileInputStream;  import java.io.ObjectInputStream;    public class Deserializador {        public Deserializador() {    }        public Object deserializar(String path) throws Exception {      FileInputStream inFile = new FileInputStream(path);      ObjectInputStream d = new ObjectInputStream(inFile);      Object o = d.readObject();      d.close();      return o;      }  } |

A Listagem 4, contém o exemplo da execução da serialização e em seguida da deserialização.

Listagem 4 - Serializando e Deserializando

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | import operacoes.Deserializador;  import operacoes.Serializador;  import pojo.Fruta;    public class Principal {      public static void main(String args[]){         //serializa         Serializador s = new Serializador();         Fruta fruta = new Fruta("Maça", "Vermelha");          try {              s.serializar("/home/gustavo/fruta", fruta);          } catch (Exception ex) {              System.err.println("Falha ao serializar! - " + ex.toString());          }            //deserializa          Deserializador d = new Deserializador();          fruta = null;          try {              fruta = (Fruta) d.deserializar("/home/gustavo/fruta");          } catch (Exception ex) {              System.err.println("Falha ao deserializar! - " + ex.toString());          }          System.out.println(fruta.getNome() + " - " + fruta.getCor());      }  } |

Na Listagem 5, encontra-se um exemplo de serialização e deserialização de um ArrayList da classe Fruta. É apenas mais um exemplo que demonstra a capacidade de se persistir qualquer objeto que seja serializável. Essa serialização só foi possível porque a classe “ArrayList” implementa a interface “Serializable” por padrão.

Listagem 5 - Serialização de ArrayList.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | //serialização e deserialização com ArrayList          ArrayList<Fruta> frutas = new ArrayList<Fruta>();          Fruta f1 = new Fruta("Laranja", "Amarela");          Fruta f2 = new Fruta("Abacate", "Verde");          Fruta f3 = new Fruta("Morango", "Vermelho");          frutas.add(f1);          frutas.add(f2);          frutas.add(f3);            try {              s.serializar("/home/gustavo/frutas", frutas);              frutas = null;              frutas = (ArrayList<Fruta>) d.deserializar("/home/gustavo/frutas");              for(Fruta f : frutas){                  System.out.println("ArrayList: " + f.getNome() + " - " + f.getCor());              }          } catch (Exception ex) {              System.err.println("Falha ao serializar ou deserializar! - " +                                 ex.toString());          } |

**4. PARTICULARIDADES DA SERIALIZAÇÃO**

Como dito, a serialização de um objeto em Java pode ser algo fácil de se fazer. Mas existem certas informações que são importantes e sabê-las pode evitar vários problemas.

Um dos problemas é que nem tudo pode ser serializado, como objetos das classes “Thread” e “Socket” e atributos estáticos. Os atributos estáticos estão vinculados à classe e não ao objeto. Assim quando se trata do estado do objeto, os campos estáticos não são considerados. Em relação às Threads, suas instâncias só são úteis enquanto estiverem em execução na JVM, e podem ser paradas/destruídas e iniciadas com instruções de sua classe. Não existe algo no estado destas instâncias que seja útil para se persistir, e por isso não há sentido em persistir tais instâncias para restaurar posteriormente.

Mas e se existir alguma classe em que seja necessário persisti-la, e esta possui uma instância da classe “Thread”? Neste caso, utilizando um recurso pode-se persistir a classe inteira, menos a instância thread. Este recurso, refere-se em indicar que tal instância thread é um objeto “Transient”. Na Listagem 6 encontra-se um exemplo da implementação com uso de “Transient”.

Listagem 6 - Uso do Transient

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | import java.io.Serializable;    public class ImplementacaoTransient implements Serializable, Runnable{        transient private Thread thread;      //marcado com "transient" para que sua instância não seja serializada        //Demais valores de estado serão serializados      private int valorParaPersistir;        public ImplementacaoTransient(int valorParaPersistir){          this.valorParaPersistir = valorParaPersistir;          thread = new Thread(this);          thread.start();      }        public void run()      {          while(true){              //código a ser executado pela Thread          }      }  } |

Vale ressaltar que, o que é serializado é apenas o estado do objeto, e não os métodos e a classe. No exemplo acima, apenas a instância do objeto “thread” e do objeto “valorParaPersistir” compunham o estado do objeto. Assim, evita-se que o sistema de serialização tente serializar um objeto não-serializável e gere erros. Após a deserialização de um objeto da classe “ImplementacaoTransient” este estará totalmente funcional.

Fim