1. Arquivos Binários

A principal diferença entre arquivos de texto e arquivos binários é a questão de armazenamento. Enquanto arquivos de texto armazenam caracteres, arquivos binários armazenam objetos serializados.

Um arquivo binário é formado por uma sequência de registros (ou objetos), sendo todos da mesma estrutura. Denominamos registro como a menor quantidade de informação a ser lida ou escrita em um arquivo binário.

Os objetos de um arquivo binário são formatados automaticamente pelo mecanismo de serialização de objetos, tendo como vantagem a rápida gravação de dados e a facilidade em gravar e recuperar objetos, porém, uma vez gravado, o objeto só poderá ser lido pelo programa que o gravou.

A. Serializando um Objeto

O primeiro passo para criar um arquivo binário é definir o objeto a ser escrito como *serializável.* Suponhamos uma classe denominada Trabalhador:

```
import java.io.Serializable;

public class Trabalhador implements Serializable {

    public String nome,CPF;

    public Trabalhador(String nome,String CPF){
           this.nome = nome;
           this.CPF = CPF;
    }
    public String getDados(){
           return "Dados de "+nome+": CPF:"+CPF;
    }
}
```

Ao declarar a classe como *Serializable* estamos “marcando” a mesma como uma classe cujos objetos podem ser serializados. A interface *Serializable* não contém nenhum método ou atributo, sendo apenas uma interface “sinalizadora”.

Após isso, devemos criar os objetos a serem serializados. Suponhamos que a classe *Trabalhador* seja superclasse das classes *Mecanico* e *Motorista*.

```
public class Mecanico extends Trabalhador {

    private double salario;
    private int nasc,ID;

    public Mecanico(String nome,String CPF,double salario,int nasc,int ID){
           super(nome,CPF);
           this.salario = salario;
           this.nasc = nasc;
           this.ID = ID;
    }
    public String getDados(){
                  return "Dados de "+nome+": CPF:"+CPF+", Salario: "+
                         salario+", Ano de Nascimento:"+nasc+", ID: "+ID;
    }
}
```

```
public class Motorista extends Trabalhador {

    private double salario;
    private int nasc,ID;

    public Motorista(String nome,String CPF,double salario,int nasc,int ID){
           super(nome,CPF);
           this.salario = salario;
           this.nasc = nasc;
           this.ID = ID;
    }
    public String getDados(){
                  return "Dados de "+nome+": CPF:"+CPF+", Salario: "+
                         salario+", Ano de Nascimento:"+nasc+", ID: "+ID;
    }
}
```

Para criarmos objetos desta classe e testar a serialização, iremos implementar uma classe *Principal* conforme a seguir:

```
import java.io.FileOutputStream;
import java.io.ObjectOutputStream;
import java.io.IOException;

public class Principal {

       public static void main(String [] args) throws IOException{

            // Definindo objetos a guardar
          Mecanico carlos= new Mecanico("Carlos", “123.425”,500, 1989, 10);
          Motorista jose = new Motorista ("José", “547.784”, 800, 1987, 12);

             // Definir path e arquivo a ser escrito
          FileOutputStream  arquivo = new FileOutputStream("c:\\rh.dat");

             //Abrindo arquivo no modo escrita
          ObjectOutputStream out = new ObjectOutputStream(arquivo);

             // Escrevendo os objetos no arquivo (Serializando)
          out.writeObject( carlos );
          out.writeObject( jose );

             // Fechando arquivo
          out.close();

       }
}
```

Note que o arquivo a ser escrito foi definido com a extensão **.dat**. Na prática podemos definir qualquer extensão para o arquivo, seja ela pré-existente ou não.

B. Desserialização de Objetos

Uma vez que um objeto foi serializado, ele pode ser recuperado. A recuperação do objeto - ou conversão de um objeto serializado em um objeto - é chamada de desserialização. A seguir a classe *Principal*, agora com a função de desserializar o objeto criado em **A**.

```
import java.io.FileInputStream;
import java.io.ObjectInputStream;
import java.io.IOException;

public class Principal {

       public static void main (String [] args)
                                throws IOException,ClassNotFoundException{

          FileInputStream  arquivo = new FileInputStream("c:\\rh.dat");

              // Abrir arquivo para ler
          ObjectInputStream in = new ObjectInputStream( arquivo );
              /* Ler objetos do ficheiro
               * método readObject() retorna um Object
               * downcasting para o tipo exato */
          Mecanico carlos = (Mecanico) in.readObject();
          Motorista jose = (Motorista) in.readObject();

          System.out.println(carlos.getDados());
          System.out.println(jose.getDados());
              // Fechar arquivo
          in.close();

       }
}
```

C. Possível Erro na Serialização/Desserialização de Objetos

Quando serializamos um objeto através da implementação da interface *Serializable,* há a possibilidade de enfrentar problemas de versão, e por causa destes problemas não seremos capazes de desserializar um ou mais objetos.

Para compreendermos este erro precisamos primeiro entender o que é o problema de versão.

Bem, digamos que uma classe foi criada, instanciada, e a mesma serializou um objeto. Esse objeto fica salvo no sistema de arquivos. Porém, após serializarmos o objeto, resolvemos atualizar o arquivo da classe serializadora, talvez adicionando um novo campo. Se fizermos isso e tentarmos desserializar o objeto criado anteriormente, uma exceção "*java.io.InvalidClassException*" será lançada. Por quê? Pelo que chamamos de *serialVersionUID*.

C.1 serialVersionUID

Durante a serialização de objetos, o mecanismo padrão de serialização Java grava os metadados sobre o objeto, que inclui o nome da classe, nomes de campos e tipos, e superclasse. Toda esta informação é armazenada como parte do objeto serializado. Quando desserializamos o objeto, esta informação auxilia na reconstrução do mesmo, identificado-o pelo *serialVersionUID*.

Então, toda vez que um objeto é serializado, o mecanismo de serialização java calcula automaticamente um valor de *hash* usando o método *computeSerialVersionUID()* do *ObjectStreamClass,* passando o nome da classe, os nomes ordenados dos membros, modificadores e interfaces para o algoritmo de *hash* seguro (SHA), que retorna um valor de *hash* denominado *serialVersionUID*.

Agora, quando o objeto serializado é recuperado, primeiro a JVM avalia o *serialVersionUID* da classe serializada e compara o valor com o *serialVersionUID* do objeto. Se os valores *serialVersionUID* corresponderem,

então, o objeto é dito ser *compatível com a classe* e, portanto, é desserializado. Se não, a exceção *InvalidClassException* é lançada.

C.2 Solução para o Problema

A solução é muito simples. Em vez de depender da JVM para gerar o *serialVersionUID*, basta mencioná-lo explicitamente na própria classe. A sintaxe para isso é:

private final static long serialVersionUID = <valor inteiro>L

Uma vez definido o *serialVersionUID* explicitamente na classe, não é necessário atualizá-lo e quaisquer alterações posteriores na classe não a tornarão incompatível. Veja o exemplo abaixo que exemplifica o problema e a importância de manter um *serialVersionUID* fixo.

```
import java.io.*;

public class TestaUID implements java.io.Serializable {

    private static final long serialVersionUID = 1L;
    private int ID;

    public TestaUID(int ID) {
        this.ID = ID;
    }

    public int getID() {
        return this.ID;
    }
}
```

```
public class Testes {

    public static void main(String args[]) throws Exception {

        File file = new File("temp.ser");

        //serialização
        FileOutputStream fos = new FileOutputStream(file);
        ObjectOutputStream oos = new ObjectOutputStream(fos);

        TestaUID writeSUID = new TestaUID(1);
        oos.writeObject(writeSUID);
        oos.close();

        //desserialização
        FileInputStream fis = new FileInputStream(file);
        ObjectInputStream ois = new ObjectInputStream(fis);

        TestaUID readSUID = (TestaUID) ois.readObject();
        System.out.println("ID: " + readSUID.getID());
        ois.close();

    }
}
```

Execute o programa acima. Agora comente a parte da serialização da classe *Testes,* de modo que a classe tente executar somente a desserialização do objeto, e modifique o *serialVersionUID* da classe *TestaUID.*Uma exceção similar a abaixo será lançada:

TestaUID; local class incompatible: stream classdesc serialVersionUID = 2, local class serialVersionUID = 1

D. Fluxos (Streams)

A linguagem java possui pacotes e classes predefinidos para a leitura e escrita de arquivos. Dependendo do tipo de dado a ser escrito/lido e do tipo de arquivo a ser criado, as classes podem variar (vide Figura 1 e Figura 2).

Categoria	Funcionalidade	Classes de Streams	Package	Formato Ficheiro
Streams de Objetos	Leitura/Escrita de objetos	ObjectInputStream ObjectOutputStream	java.io	Binário
Streams de Dados	Leitura/Escrita de tipos de dados primitivos (int, long, etc) e valores String	DataInputStream DataOutputStream	java.io	Binário
Streams de Bytes	Leitura/Escrita de bytes	FileInputStream FileOutputStream	java.io	Binário
Streams de Carateres	Leitura/Escrita de carateres Traduzem automaticamente os dados baseados na tabela de carateres local	BufferedReader BufferedWriter PrintWriter	java.io	Texto
Streams de *Scanning* e Formatação	Leitura/Escrita de texto formatado	Scanner Formatter	java.util	Texto
Streams com Buffers	Otimizar o I/O Conseguido através da redução do número de chamadas para a API do sistema operativo	BufferedInputStream BufferedOutputStream	java.io	

Figura 1.Categoria de Streams

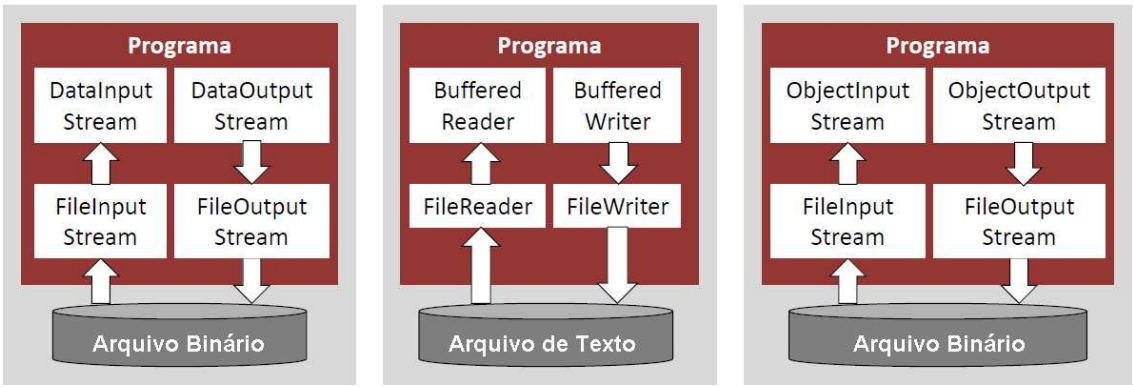

Figura 2. Camadas de Streams

2. Exercícios de Fixação

1. Implemente a classe abaixo denominada PRODUTOS e, pelo menos, 5 outras classes que herdem a classe PRODUTOS. Cada uma destas 5 classes deve ter no mínimo 5 atributos e 2 métodos.

```
public class Produtos{
   public String nome,serial;
   public double ID;
   public Produtos(String nome, String serial, double ID){
      this.nome = nome;
      this.serial = serial;
      this.ID = ID;
   }
}
```

2. Crie uma interface com o usuário (console ou JOptionPane) que instancie os atributos das classes criadas.

3. Crie duas classes distintas: uma para serializar os objetos das 5 classes e outra para desserializa-los. Ao desserializar, o programa deve imprimir os dados de cada classe. Exemplo: supondo a classe DVD e o método getDados() que retorna todos os atributos da classe DVD. A classe de desserialização deve invocar o método getDados do objeto serializado de DVD.

4. Crie uma agenda simples em que o programa solicite ao usuário nome e telefone, estes sejam lidos do teclado, instanciados numa classe denominada *contatos* e o(s) objeto(s) desta classe seja(m) salvo(s) em um arquivo denominado "*contatos.dados*". Devem ser inseridos pelo menos 5 contatos. Dica: utilize ArrayList ou LinkedList e serialize o objeto desta estrutura de dados no arquivo, ao invés de inserir objeto por objeto.

5. Crie um programa para um departamento de RH que utiliza a lista de contatos criada no exercício 4. O programa deve recuperar nome e telefone do contato, mostrá-lo em tela e solicitar outras informações como nível de escolaridade, cargo e salário pretendido. Estas informações deverão ser inseridas pelo usuário, via teclado. Após isso, o programa deve criar **um novo arquivo binário** e salvar nome, telefone, nível de escolaridade, cargo e salário do contato.

Os exercícios podem ser feitos em dupla e entregues à professora via moodle no formato .java ou .zip. Estes exercícios complementarão o conceito final de A1.