Java

Ficheiros

adaptado de Donald W. Smith (TechNeTrain.com)

Objetivos

- Conhecer os formatos de ficheiros de texto e binário
- Usar acessos sequenciais e aleatórios
- Ler e escrever objetos usando serialização

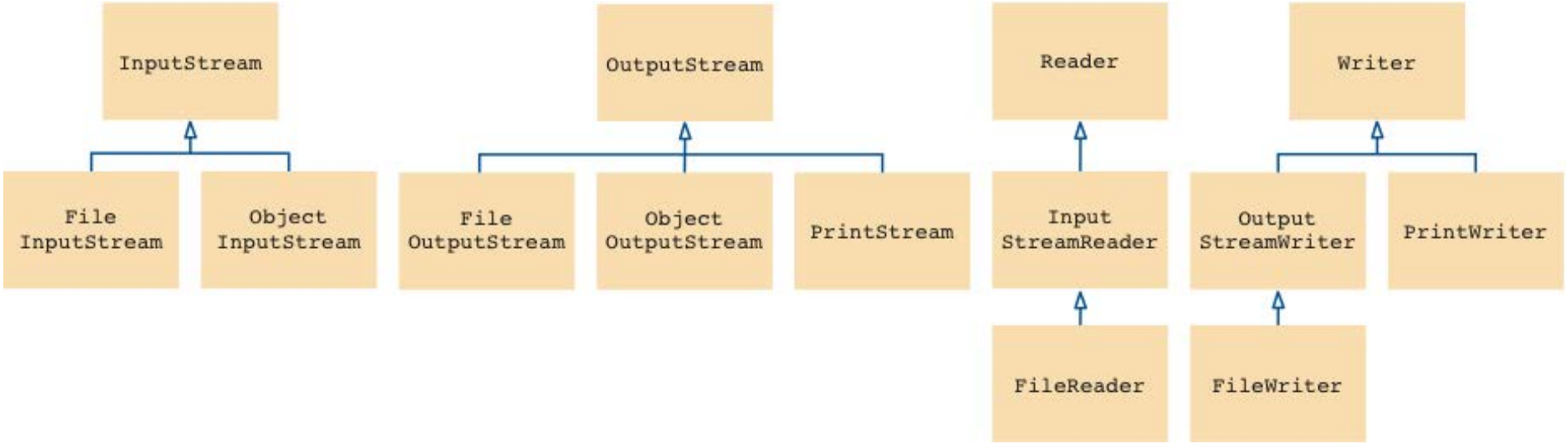
Conteúdos

- *Readers, Writers* e *Streams*
- Entrada e Saída Texto/Binárias
- Acesso Aleatório
- *Object Streams*

Readers, *Writers* e *Streams*

- Duas formas de armazenamento:
  - *Formato texto:* forma legível, sob a forma de uma sequência de caracteres
    - Ex. integer 12345 armazenado como '1' '2' '3' '4' '5'
    - Mais conveniente para humanos: facilita as operações de entrada e saída
    - *Readers* e *Writers* lidam com dados sob a forma de texto
  - *Formato binário:* dados são representados em *bytes*
    - Ex. integer 12345 armazenado através de uma sequência de quatro bytes: 0 0 48 57
    - Mais compacto e mais eficiente
    - *Streams* lidam com dados binários

Readers, Writers e Streams (cont.)



Formato Texto

- `Reader`, `Writer` e suas subclasses destinam-se ao processamento de texto (entrada e saída)
- A classe `Scanner` pode ser mais indicada que a classe `Reader`
- Estas classes têm a responsabilidade de realizar a conversão entre bytes e caracteres

```
// input pode ser um File ou InputStream
Scanner in = new Scanner(input, "UTF-8");

// output pode ser um File ou OutputStream
PrintWriter out = new PrintWriter(output, "UTF-8");
```

Formato Binário

- Usar `InputStream`, `OutputStream` e respetivas subclasses para processar entradas e saídas binárias
- Para entradas (leitura)

```
FileInputStream inputStream =
   new FileInputStream("input.bin");
```

- Para saídas (escrita)

```
FileOutputStream outputStream =
      new FileOutputStream("output.bin");
```

- `System.out` é um objeto `PrintStream`

Formato Binário – Entrada

- Usar o método `read` da classe `InputStream` para ler um único byte
  - retorna o próximo byte como um `int` entre 0 e 255
  - ou, o inteiro `-1` no caso de *end of file*

```
InputStream in = . . .;
int next = in.read();
if (next != -1)
{
   Processar next // um valor entre 0 e 255
}
```

Formato Binário – Saída

- Usar o método `write` da classe `OutputStream` para escrever um único byte:

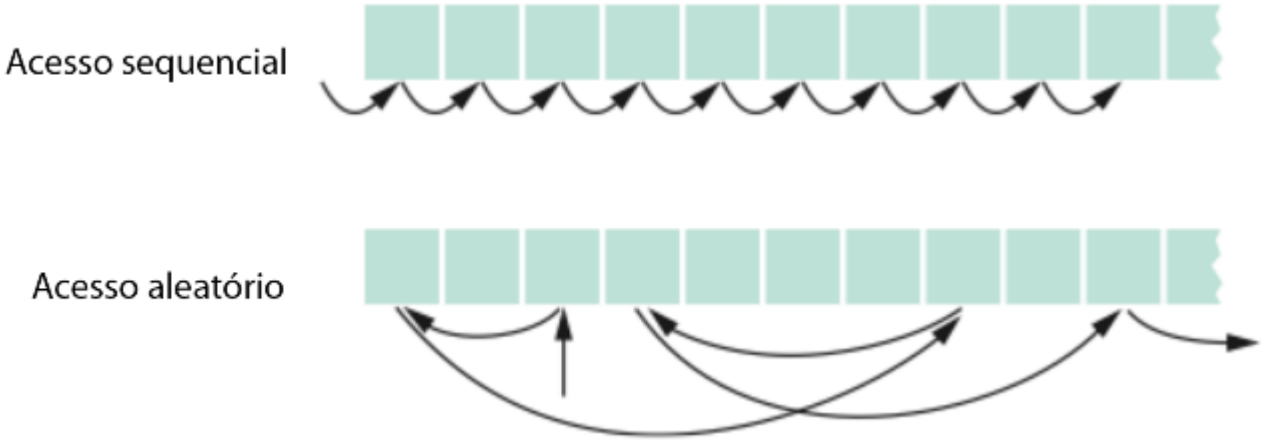
```
OutputStream out = . . .;
int value= . . .; // deve ser entre 0 e 255
out.write(value);
```

- Terminar a escrita encerrando o ficheiro:

```
out.close();
```

Acesso Aleatório

- **Acesso sequencial:** processa o ficheiro um byte de cada vez, de forma sequencial
- **Acesso aleatório (*random*):** Acesso ao ficheiro em posições arbitrárias
  - Apenas *disk files* suportam acesso aleatório
    - `System.in` e `System.out` não permitem o acesso aleatório
  - Cada *disk file* possui um apontador de posição (***file pointer***)
    - Permite ler ou escrever em qualquer posição



Acesso Aleatório – RandomAccessFile

- Abrir um ficheiro em modo de:
  - Leitura apenas ("r") – `FileNotFoundException` se não existir
  - Leitura e escrita ("rw") – `FileNotFoundException` se não puder ser criado

```
RandomAccessFile f =
    new RandomAcessFile("bank.dat","rw");
```

- Para mover o *file pointer* para um byte específico:

```
f.seek(position);
```

- Para obter a posição atual do *file pointer*:

```
//do tipo "long" porque a dimensão do ficheiro pode ser grande
long position = f.getFilePointer();
```

- Para obter o número de bytes num ficheiro:

```
long fileLength = f.length();
```

Serialização

- Processo no qual a instância de um objeto é transformada numa sequência de bytes
- Permite implementar a persistência dos objetos
- Pode ser usado para enviar objetos através de uma rede ou gravá-los em ficheiro
- Para que possa ser aplicado aos objetos de uma classe, a classe deve implementar a interface Serializable
  - Trata-se de uma interface de marcação, pois não define qualquer método, servindo apenas para que a JVM saiba que a classe pode ser serializada
- Uma vez que as variáveis estáticas estão associadas a uma classe e não às instâncias da classe, não é possível serializar variáveis estáticas

Object Streams

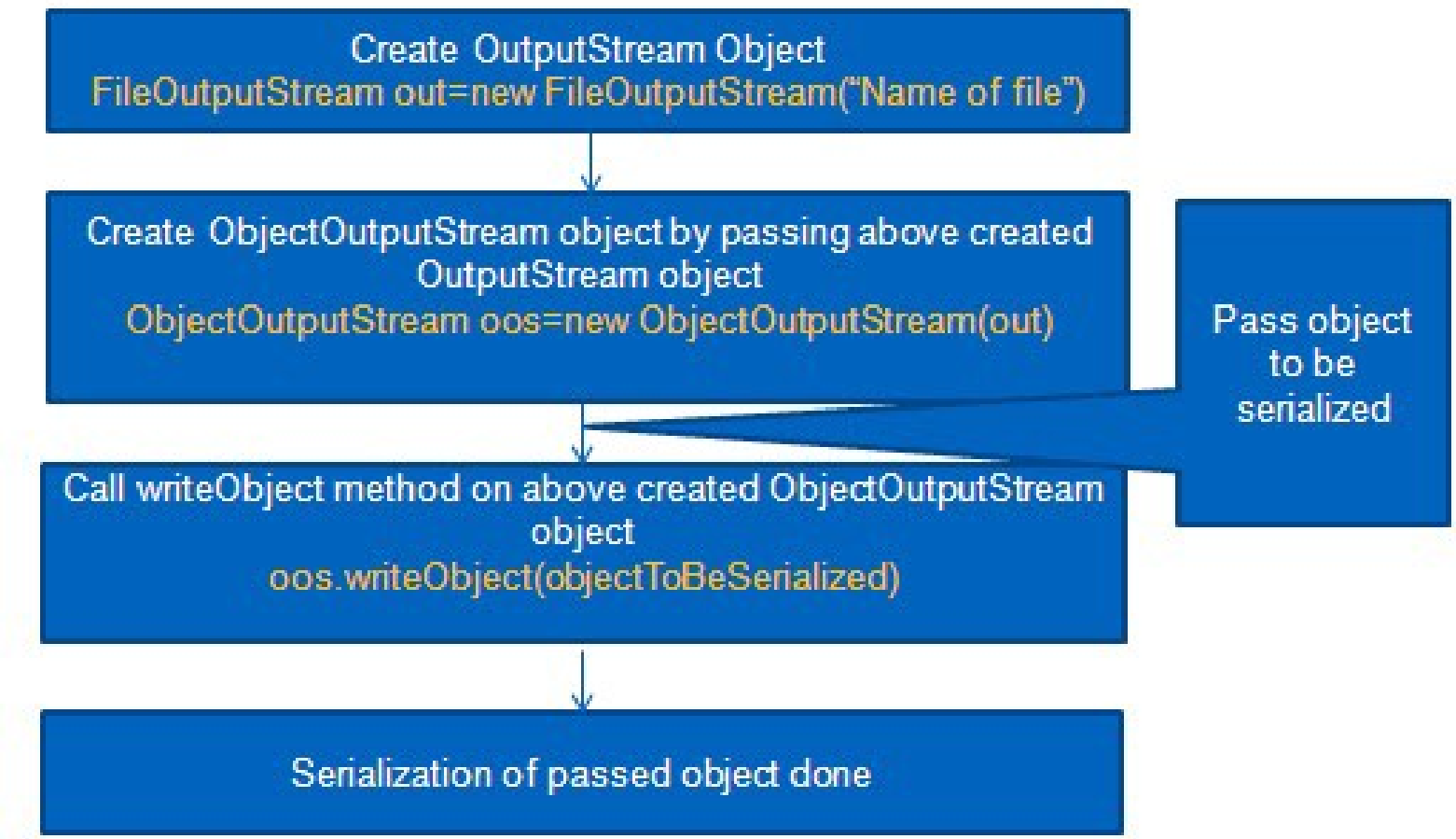
- A classe `ObjectOutputStream` pode gravar objetos para um ficheiro
- A classe `ObjectInputStream` pode lê-los
- Usar *streams* e não *writers* uma vez que os objetos são gravados em formato binário

Object Streams – Serialização

- Todas as variáveis de instância são gravadas:

```
//exception handling missing

//class BankAccount or superclass implements Serializable
BankAccount b = ...;

ObjectOutputStream out = new ObjectOutputStream(
   new FileOutputStream("bank.dat"));

out.writeObject(b);

out.close();
```

Serialização – Passos



https://medium.com/javarevisited/complete-guide-to-serialization-in-java-44b36032157

Serialização – Exemplo

```
import java.io.Serializable;

public class Employee implements Serializable{
   private int employeeId;
   private String employeeName;
   private String department;

   public int getEmployeeId() { return employeeId;}
   public void setEmployeeId(int employeeId) {
      this.employeeId = employeeId;
   }

   public String getEmployeeName() { return employeeName;}
   public void setEmployeeName(String employeeName) {
      this.employeeName = employeeName;
   }

   public String getDepartment() { return department;}
   public void setDepartment(String department) {
      this.department = department;
   }
}
```

Serialização – Exemplo (cont.)


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

public class SerializeMain {
   public static void main(String[] args) {

     Employee emp = new Employee();
     emp.setEmployeeId(101);
     emp.setEmployeeName("Arpit");
     emp.setDepartment("CS");

     try {
        FileOutputStream fileOut = new FileOutputStream("employee.ser");

        ObjectOutputStream outStream = new ObjectOutputStream(fileOut);
        outStream.writeObject(emp);

        //use finally ?
        outStream.close();
        fileOut.close();
     } catch(IOException i) {
        i.printStackTrace();
     }
   }
}
```

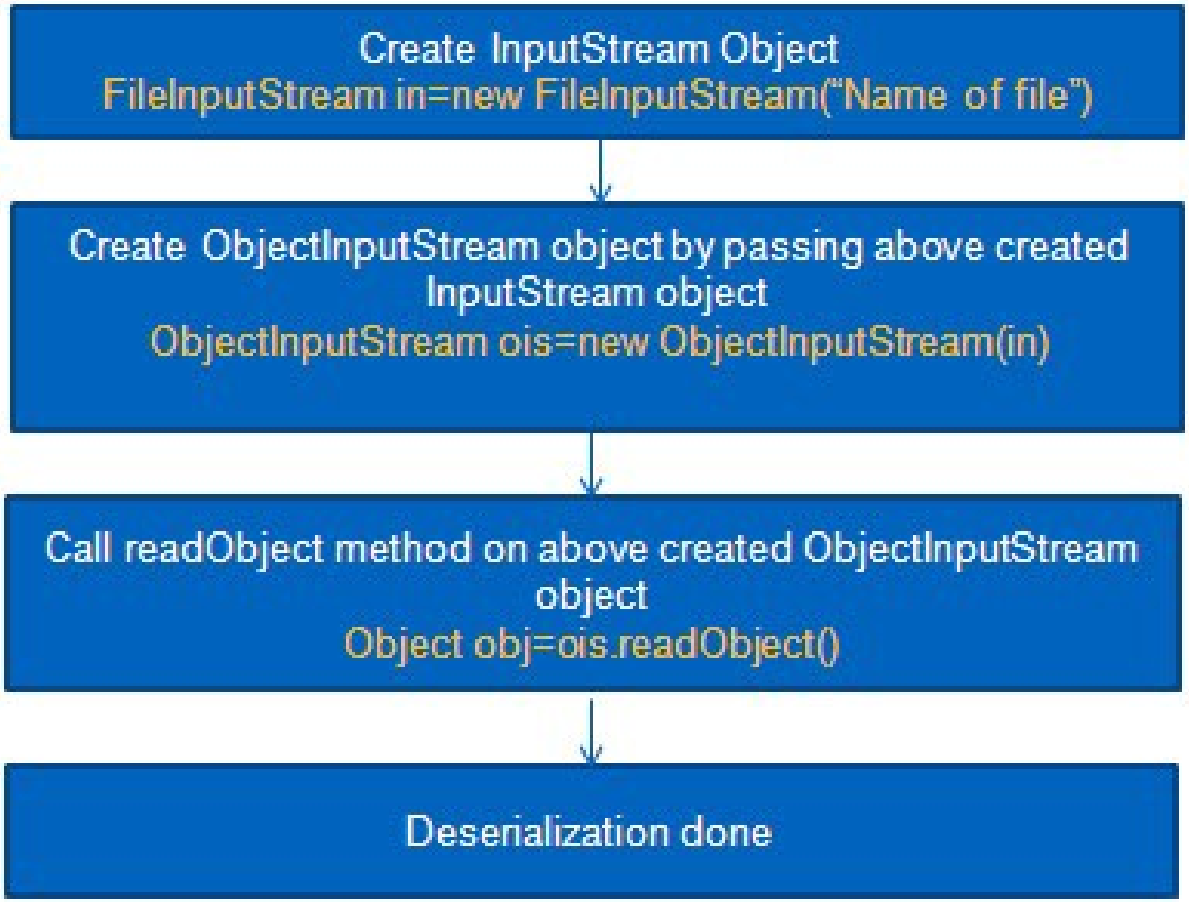
Object Streams – Desserialização

- O método readObject devolve uma referência para Object
- É necessário conhecer os tipos dos objetos gravados e fazer a respetiva conversão (*cast*)

```
ObjectInputStream in = new ObjectInputStream(
   new FileInputStream("bank.dat"));
BankAccount b =(BankAccount) in.readObject();
```

- O método readObject pode lançar uma exceção do tipo ClassNotFoundException caso alguma classe não esteja marcada com a interface Serializable
  - *É uma checked exception* ⇒ é necessário tratar

Desserialização – Passos



https://medium.com/javarevisited/complete-guide-to-serialization-in-java-44b36032157

Desserialização – Exemplo (cont.)

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

public class DeserializeMain {
   public static void main(String[] args) {
      Employee emp = null;

      try {
         FileInputStream fileIn =new FileInputStream("employee.ser");

         ObjectInputStream in = new ObjectInputStream(fileIn);
         emp = (Employee) in.readObject();

         //use finally ?
         in.close();
         fileIn.close();
      } catch(IOException i) {
         i.printStackTrace();
         return;
      } catch(ClassNotFoundException c) {
         System.out.println("Employee class not found");
         c.printStackTrace();
         return;
      }

      System.out.println("Deserialized Employee...");
      System.out.println("Emp id: " + emp.getEmployeeId());
      System.out.println("Name: '"+ emp.getEmployeeName());
      System.out.println("Department: " + emp.getDepartment());
   }
}
```

Serialização ArrayList

- Serialização

```
ArrayList<BankAccount> a =
   new ArrayList<BankAccount>();

// Adicionar várias instâncias de BankAccount em a
ObjectOutputStream out = new ObjectOutputStream(
   new FileOutputStream(”accounts.dat"));
out.writeObject(a);
```

- Desserialização

```
ObjectInputStream in = new ObjectInputStream(
   new FileInputStream("bank.dat"));

ArrayList<BankAccount> a =
   (ArrayList<BankAccount>) in.readObject();
```

Interface Serializable

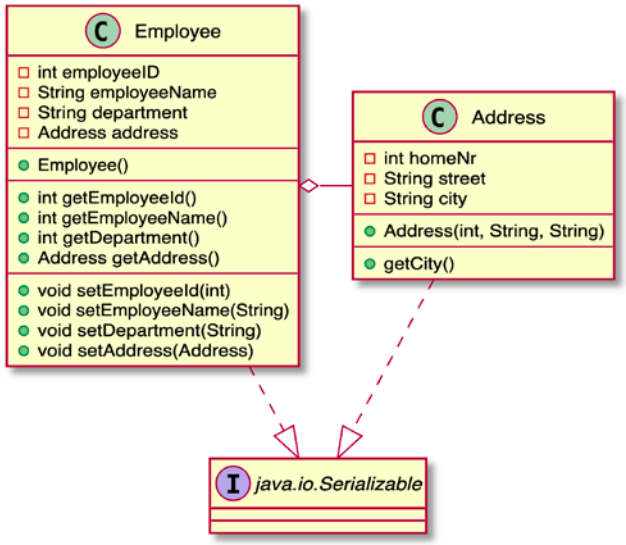
- Os objetos que são escritos num *object stream* devem pertencer a uma classe que implementa a interface `Serializable`:

```
class BankAccount implements Serializable
{
   …
}
```

- A interface `Serializable` não tem métodos
- **Serialização**
  - Cada objeto é identificado no *stream* por um número de série
  - Se o mesmo objeto é gravado duas vezes, na segunda vez apenas é gravado o seu número de série
  - Na leitura, números de série repetidos são restaurados como referências ao mesmo objeto

Serialização com referências

- Quando um objeto que contém referências para outros objetos é serializado, a JVM serializa todos os objetos relacionados
- *E.g.*, se um objeto `Employee` contém uma referência para um objeto do tipo `Address`, quando se serializa o objeto `Employee` o objeto `Address` também será serializado



Serialização com referências (cont.)

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

public class SerializeDeserializeMain {
   public static void main(String[] args) {
      Employee emp = new Employee();

      emp.setEmployeeId(101);
      emp.setEmployeeName(“Arpit”);
      emp.setDepartment(“CS”);

      Address address=new Address(88,”MG road”,”Pune”);
      emp.setAddress(address);

      try {
         FileOutputStream fileOut = new FileOutputStream(“employee.ser”);
         ObjectOutputStream outStream = new ObjectOutputStream(fileOut);

         outStream.writeObject(emp);

         outStream.close();
         fileOut.close();
      } catch(IOException i) {
         i.printStackTrace();
      }
```

Desserialização com referências


```
    emp = null;
    try {
      FileInputStream fileIn =new FileInputStream("employee.ser");
      ObjectInputStream in = new ObjectInputStream(fileIn);

      emp = (Employee) in.readObject();

      in.close();
      fileIn.close();
    } catch(IOException i) {
      i.printStackTrace();
      return;
    } catch(ClassNotFoundException c) {
      System.out.println("Employee class not found");
      c.printStackTrace();
      return;
    }

    System.out.println("Deserialized Employee...");
    System.out.println("Emp id: " + emp.getEmployeeId());
    System.out.println("Name: " + emp.getEmployeeName());
    System.out.println("Department: " + emp.getDepartment());

    address=emp.getAddress();
    System.out.println("City : "+address.getCity());
  }
}
```

Serialização – supressão de atributo

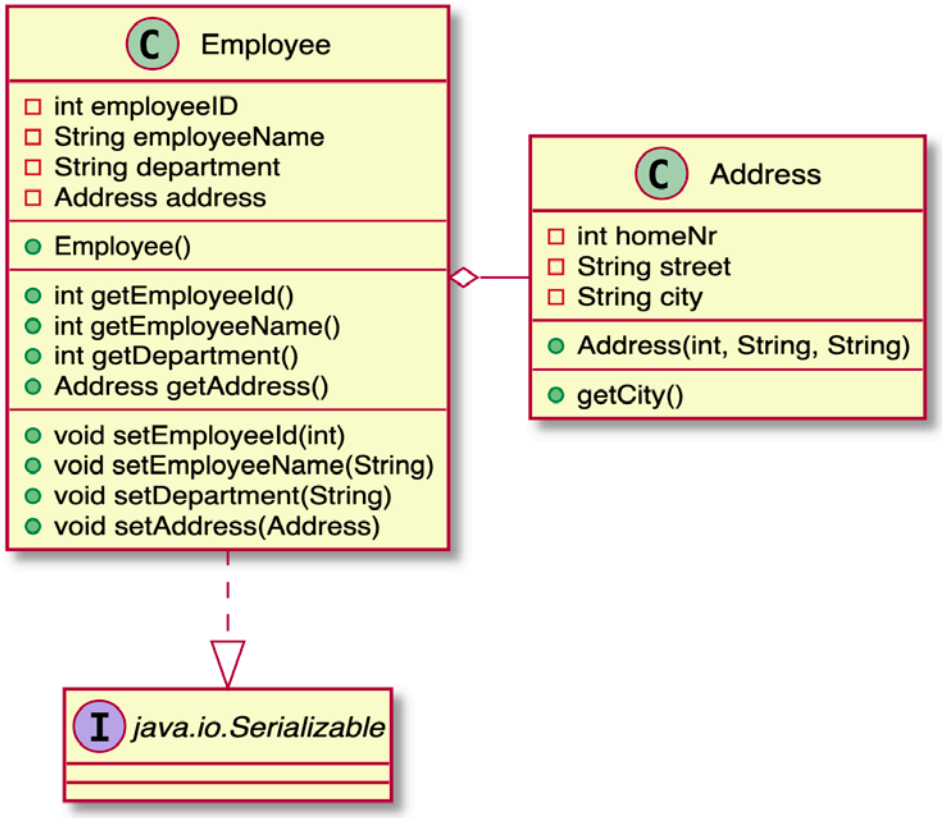
- Caso não se pretenda serializar um atributo específico de um determinado objeto, basta marcá-lo como `transient`
  - o objeto serializado não conterá a informação referente ao atributo `transient`
- *E.g.*, se pretendermos excluir o atributo `Address` da serialização dos objetos de `Employee`

```
private transient Address address;
```

  - Após a desserialização, se tentarmos aceder ao atributo `address` será lançada uma exceção `NullPointerException`

Serialização – reescrita

- Permite serializar `Employee` neste cenário



Serialização – reescrita (cont.)

- Uma classe que implemente a interface `Serializable` mas necessite de um procedimento especial durante o processo de serialização e desserialização, deve implementar os métodos `writeObject` e `readObject`
- Estes métodos devem ser definidos como `private`

Serialização – reescrita (cont.)

- Reescrita do método `writeObject` na classe que vamos serializar – `Employee`.

```
private void writeObject(ObjectOutputStream os) throws
        IOException, ClassNotFoundException {
   try {
      os.defaultWriteObject();

      os.writeInt(address.getHomeNo());
      os.writeObject(address.getStreet());
      os.writeObject(address.getCity());
   } catch (Exception e) {
      e.printStackTrace();
   }
}
```

Serialização por defeito

Serialização dos atributos do objeto Address

Serialização – reescrita (cont.)

- Reescrita do método `readObject` na classe que vamos desserializar – `Employee`.

```
private void readObject(ObjectInputStream is) throws
        IOException,  ClassNotFoundException {
   try {
      is.defaultReadObject();

      int homeNo=is.readInt();
      String street=(String) is.readObject();
      String city=(String) is.readObject();
      address=
         new Address(homeNo,street,city);
   } catch (Exception e) {
      e.printStackTrace();
   }
 }
```

Desserialização por defeito

`ObjectInputStream` deve ler os dados pela mesma ordem da escrita feita pelo `ObjectOutputStream`

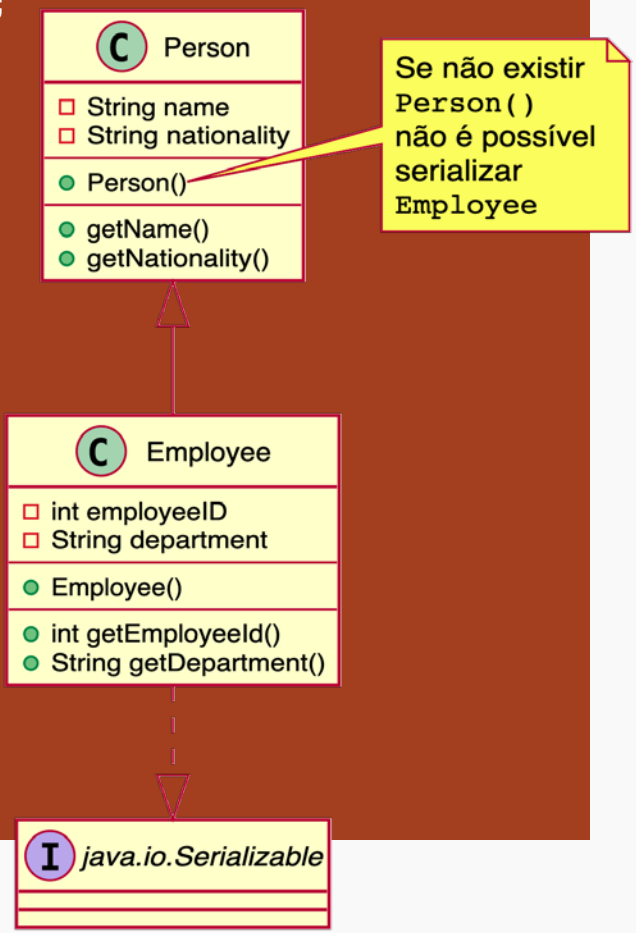
Serialização e Herança

- Se uma superclasse é serializável, então todas as suas subclasses são automaticamente serializáveis
- Se uma superclasse não é serializável, então esta superclasse deve conter a declaração do construtor sem argumentos
  - Todos os valores dos atributos de instância herdadas da superclasse serão iniciados através da invocação do construtor sem parâmetros da superclasse durante a desserialização

Serialização e Herança (cont.)


```
    emp = null;
    try {
       FileInputStream fileIn =new FileInputStream("employee.ser");
       ObjectInputStream in = new ObjectInputStream(fileIn);

       emp = (Employee) in.readObject();

       in.close();
       fileIn.close();
    } catch(IOException i) {
       i.printStackTrace();
       return;
    } catch(ClassNotFoundException c) {
       System.out.println("Employee class not found");
       c.printStackTrace();
       return;
    }

    System.out.println("After serializing");

    System.out.println("Emp id: " + emp.getEmployeeId());
    System.out.println("Name: " + emp.getName());
    System.out.println("Department: " + emp.getDepartment());
    System.out.println("Nationality: " + emp.getNationality());   }
}
```

Invoca automaticamente `Person()` ou os construtores sem parâmetros necessários



Serialização e Herança (cont.)

- Se pretendemos que uma subclasse não seja serializável, a subclasse terá que implementar os métodos `writeObject` e `readObject`, devendo, cada um dos métodos, lançar a exceção `NotSerializableException`

Conclusão

- *Streams* permitem o acesso a sequências de bytes
- *Readers* e *writers* permitem o acesso a sequências de caracteres
- Usar as classes `FileInputStream` e `FileOutputStream` para ler e escrever dados binários

Conclusão (cont.)

- No caso do acesso sequencial, um ficheiro é processado *byte* a *byte*, a partir do início
- O acesso aleatório permite o acesso a posições arbitrárias no ficheiro, sem necessidade de ler os bytes que precedem o local de acesso
- Um *file pointer* é uma posição num ficheiro acedido no modo de acesso aleatório, sendo do tipo `long`
- A classe `RandomAccessFile` lê e escreve dados sob a forma binária

Conclusão (cont.)

- Usar *object streams* para guardar e restaurar automaticamente todos os atributos de instância de um objeto
- Os objetos guardados num *object stream* devem pertencer a classes que implementem a interface `Serializable`
- Os atributos estáticos não podem ser armazenados desta forma
- É possível definir regras para guardar e restaurar atributos de instância e/ou super/subclasses