7

SERIALIZAÇÃO

Paradigmas de Programação

LEI - ISEP

Luiz Faria, 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

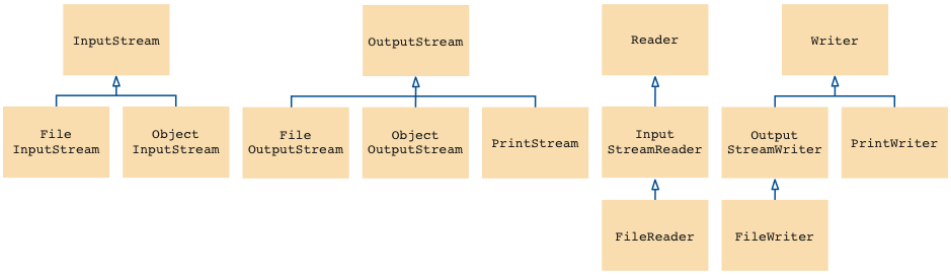
Conteúdos

- *Readers, Writers* e *Streams*
- Entrada e Saída 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 12,345 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 12,345 armazenado através de uma sequência de quatro bytes: `0 0 48 57`
    - Mais compacto e mais eficiente
    - *Streams* lidam com dados binários

Classes Java para *Input* e *Output*



Dados em Formato Texto

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

```
Scanner in = new Scanner(input, "UTF-8");
   // input pode ser uma File ou InputStream
PrintWriter out = new PrintWriter(output, "UTF-8");
   // output pode ser uma File ou OutputStream
```

Entradas e Saídas Binárias

- Usar `InputStream`, `OutputStream` e suas subclasses para processar entradas e saídas binárias
- Para ler:

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

- Para escrever:

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

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

Entrada Binária

- 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
}
```

Saída Binária

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

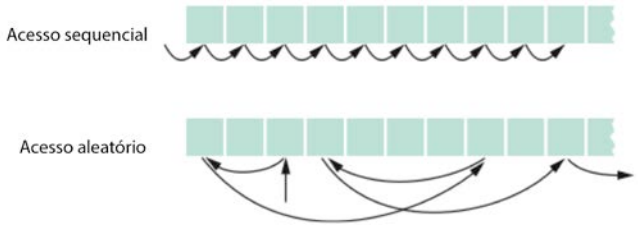
```
OutputStream out = . . .;
int value= . . .; // deve ser entre 0 e 255
out.write(value);
```

- 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



Classe RandomAccessFile

- Abrir um ficheiro em modo de:
  - Leitura apenas (`"r"`)
  - Leitura e escrita (`"rw"`)

```
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*:

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

- 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

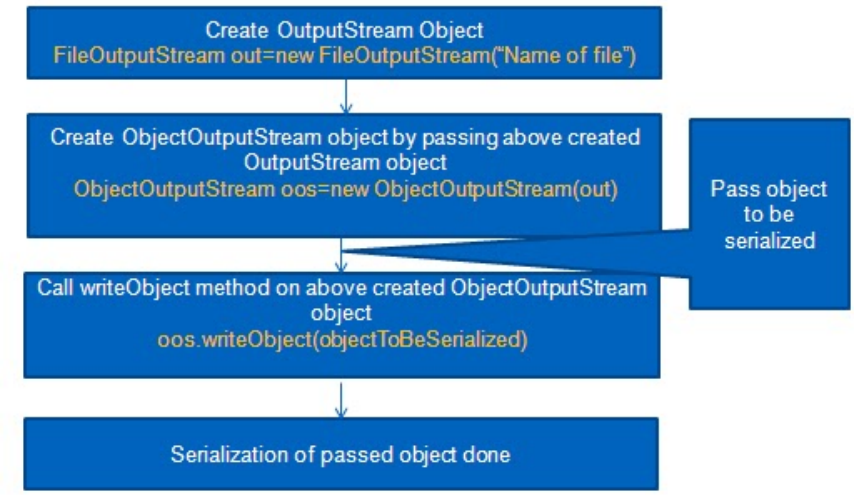
- A classe `ObjectOutputStream` pode gravar objetos para um ficheiro
- A classe `ObjectInputStream` pode lê-los
- Usar *streams* e não *writers* uma vez que os objetos são gravados em formato binário

Escrita de um Objeto para Ficheiro

- Todas as variáveis de instância são gravadas:

```
BankAccount b = ...;
ObjectOutputStream out = new ObjectOutputStream(
   new FileOutputStream("bank.dat"));
out.writeObject(b);
```

Serialização - Passos



Exemplo: Employee.java

```
import java.io.Serializable;
public class Employee implements Serializable{
    int employeeId;
    String employeeName;
    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;
    }
}
```

Exemplo: SerializeMain.java

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

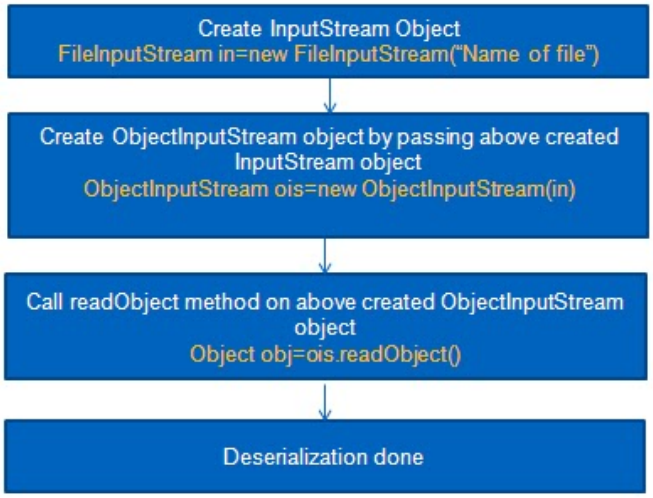
Leitura de um Objeto a partir de um Ficheiro

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

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

- O método `readObject` pode lançar `ClassNotFoundException`
  - *Checked exception* ⇒ é necessário tratá-la ou declará-la

Deserialização - Passos



Exemplo: DeserializeMain.java

```
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();
          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());
 }
}
```

```
run:
Deserialized Employee...
Emp id: 101
Name: Arpit
Department: CS
```

Escrita e Leitura de um *Array List*

- Escrita:

```
ArrayList<BankAccount> a =
   new ArrayList<BankAccount>();
// Adicionar várias instâncias de BankAccount em a
out.writeObject(a);
```

- Leitura:

```
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 objecto

Serialização: referências para outros objetos

- Quando um objeto que contém referências para outros objetos é serializado o Java serializa todos os objetos relacionados
- Por exemplo, se um objeto `Employee` contém uma referência para um objeto do tipo `Address`, quando se serializa o objeto `Employee` também o objeto `Address` será serializado

Serialização: referências para outros objetos (exemplo 1)

```
import java.io.Serializable;
public class Employee implements Serializable{
 int employeeId;
 String employeeName;
 String department;
 Address address;
 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;
 }
 public Address getAddress() { return address; }
 public void setAddress(Address address) {this.address = address; }
}
```

Serialização: referências para outros objetos (exemplo 2)

```
import java.io.Serializable;
public class Address implements Serializable{
 int homeNo;
 String street;
 String city;
 public Address(int homeNo, String street, String city) {
  this.homeNo = homeNo;
  this.street = street;
  this.city = city;
 }
 public int getHomeNo() { return homeNo; }
 public void setHomeNo(int homeNo) { this.homeNo = homeNo; }
 public String getStreet() { return street; }
 public void setStreet(String street) { this.street = street; }
 public String getCity() { return city; }
 public void setCity(String city) { this.city = city; }
}
```

A classe Address deve também implementar a interface Serializable

Serialização: referências para outros objetos (exemplo 3)

```
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);
  //Serialize
  try {
   FileOutputStream fileOut = new FileOutputStream('employee.ser');
   ObjectOutputStream outStream = new ObjectOutputStream(fileOut);
   outStream.writeObject(emp);
   outStream.close();
   fileOut.close();
  } catch(IOException i) {
   i.printStackTrace();
  }
```

Serialização: referências para outros objetos (exemplo 4)

```
  //Deserialize
  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());
 }
}
```

```
run:
Deserialized Employee...
Emp id: 101
Name: Arpit
Department: CS
City: Pune
```

Serialização: supressão de um atributo específico

- Caso não se pretenda serializar algum atributo específico de determinado objeto, basta marcá-lo como `transient`
  - o objeto serializado não conterá a informação referente ao atributo `transient`
- Ex.: Se pretendermos excluir o atributo `Address` da serialização dos objetos de `Employee`: `transient Address address`
  - Após a deserialização, se tentarmos aceder ao atributo `address` obteremos `nullPointerException`

Personalização do Processo de Serialização

- Consideremos que no exemplo anterior a classe `Address` não implementa a interface `Serializable`, mas pretendemos guardar a informação relativa aos objetos do tipo `Address`
- Uma classe que implementa a interface `Serializable` e necessita de um tratamento especial durante o processo de serialização e deserialização, deve implementar os métodos `writeObject` e `readObject`
- Estes métodos devem ser definidos como `private`

Personalização do Processo de Serialização

- Vamos adicionar as implementações dos métodos `writeObject` e `readObject` à classe que pretendemos 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(); }
 }
 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(); }
 }
…
```

Serialização por defeito

Serialização dos atributos do objeto Address

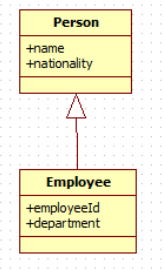
ObjectInputStream deve ler os dados pela mesma sequência que estes foram escritos em ObjectOutputStream

Serialização e Herança

- Se uma superclasse é serializável, então todas as suas subclasses são automaticamente serializáveis
- Se uma superclasse não é serializável, então a superclasse deve conter a declaração do construtor sem argumentos
  - Todos os valores das variáveis de instância herdadas da superclasse serão inicializados através da chamada do construtor da superclasse durante o processo de deserialização

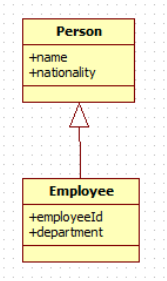
Serialização e Herança (exemplo 1)

```
public class Person {
    String name;
    String nationality;

    public Person() {
        this.name = "default";
        System.out.println("Person:Constructor");
    }
    public Person(String name, String nationality) {
        this.name = name;
        this.nationality = nationality;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public String getNationality() {
        return nationality;
    }
    public void setNationality(String nationality) {
        this.nationality = nationality;
    }
}
```



Serialização e Herança (exemplo 2)

```
public class Employee extends Person implements Serializable {
    int employeeId;
    String department;

    public Employee(int employeeId, String name, String department,
String nationality) {
        super(name, nationality);
        this.employeeId = employeeId;
        this.department = department;
        System.out.println("Employee:Constructor");
    }
    public int getEmployeeId() {
        return employeeId;
    }
    public void setEmployeeId(int employeeId) {
        this.employeeId = employeeId;
    }
    public String getDepartment() {
        return department;
    }
    public void setDepartment(String department) {
        this.department = department;
    }
}
```



Serialização e Herança (exemplo 3)

```
public class SerializeDeserializeMain2 {
   public static void main(String[] args) {
        //Serialize
        Employee emp = new Employee(101,"Arpit","CS","Indian");
        System.out.println("Before 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());
        System.out.println("************");
        System.out.println("Serializing");
        try {
            FileOutputStream fileOut = new FileOutputStream("employee.ser");
            ObjectOutputStream outStream = new ObjectOutputStream(fileOut);
            outStream.writeObject(emp);
            outStream.close();
            fileOut.close();
        }catch(IOException i) {
            i.printStackTrace();
        }
        …
```

Serialização e Herança (exemplo 4)

```
        …
        //Deserialize
        System.out.println("************");
        System.out.println("Deserializing");
        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());
    }
}
```

Durante a deserialização, são chamados os construtores sem argumentos de todas as superclasses da hierarquia que não implementam `Serializable`

```
run:
Employee:Constructor
Before serializing
Emp id: 101
Name: Arpit
Department: CS
Nationality: Indian
************
Serializing
************
Deserializing
Person:Constructor
After serializing
Emp id: 101
Name: default
Department: CS
Nationality: null
```

Serialização e Herança

- Como referido, se uma superclasse é serializável todas as suas subclasses também o serão
- Se pretendemos que uma subclasse não seja serializável, a subclasse terá que implementar os métodos `writeObject()` e `readObject()`, devendo estes lançar a exceção `NotSerializableException`

Resumo: Classes Java para tratamento de entrada e saída

- *Streams* permitem o acesso a sequências de bytes
- *Readers* e *writers* permitem o acesso a sequências de caracteres

Resumo: Entrada e saída de dados binários

- Usar as classes `FileInputStream` e `FileOutputStream` para ler e escrever dados binários
- O método `InputStream.read` devolve um inteiro, -1 para indicar final da entrada, ou um byte entre 0 e 255
- O método `OutputStream.write` escreve um único byte

Resumo: Acesso aleatório

- 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

Resumo: *Object Streams*

- Usar *object streams* para guardar e restaurar automaticamente todas as variáveis de instância de um objeto
- Os objetos guardados num *object stream* devem pertencer a classes que implementem a interface `Serializable`
- As variáveis estáticas não podem ser armazenadas desta forma