OPERAÇÕES DE ENTRADA E SAÍDA

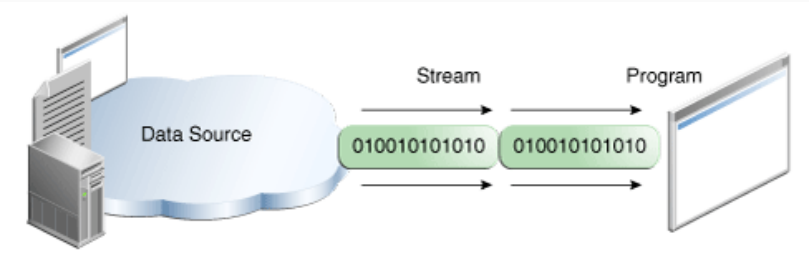
ACH 2003 — COMPUTAÇÃO ORIENTADA A OBJETOS

Daniel Cordeiro

Escola de Artes, Ciências e Humanidades | EACH | USP

FLUXOS DE E/S EM JAVA

- Um fluxo de E/S representa uma fonte de dados ou um destino de saída
- Pode representar dispositivos bem diferentes, como arquivos em disco, dispositivos, outros programas e vetores na memória
- Abstração simples, um fluxo é uma sequência de dados



Fluxos de bytes: `InputStream` e `OutputStream`

Fluxos de caracteres: `Reader` e `Writer`

Fluxos com buffers: (i) `BufferedInputStream` e `BufferedOutputStream`; (ii) `BufferedReader` e `BufferedWriter`

e detalhes de Strings sobre conversão em valores e formatação

DATA STREAMS

- *Data streams* auxiliam a E/S de tipos de dados primitivos de Java (`boolean`, `char`, `byte`, `short`, `int`, `long`, `float` e `double`), além de `String`
- implementam as interfaces `DataInput` ou `DataOutput`

```
static final String dataFile = "dados_da_nota_fiscal";

static final double[] prices = { 19.99, 9.99, 15.99, 3.99, 4.99 };
static final int[] units = { 12, 8, 13, 29, 50 };
static final String[] descs = {
    "Java T-shirt",
    "Java Mug",
    "Duke Juggling Dolls",
    "Java Pin",
    "Java Key Chain"
};

DataOutputStream out = null;
try {
    out = new DataOutputStream(new
            BufferedOutputStream(new FileOutputStream(dataFile)));

    for (int i = 0; i < prices.length; i ++) {
        out.writeDouble(prices[i]);
        out.writeInt(units[i]);
        out.writeUTF(descs[i]);
    }
} finally {
    out.close();
}
}
```

```
in = new DataInputStream(new
            BufferedInputStream(new FileInputStream(dataFile)));

double price;
int unit;
String desc;
double total = 0.0;

try {
    while (true) {
        price = in.readDouble();
        unit = in.readInt();
        desc = in.readUTF();
        System.out.format("Você comprou %d" + " unidades de %s por $%.2f%n",
            unit, desc, price);
        total += unit * price;
    }
} catch (EOFException e) {
    // fim do arquivo
}
```

[1] Obs: usar `double` para guardar valores monetários é má ideia. Use `java.math.BigDecimal` no lugar.

FLUXOS DE OBJETOS

- assim como *data streams* auxiliam a E/S de tipos primitivos, fluxos de objetos (*object streams*) permitem E/S de objetos inteiros
- objetos que permitem que seja feito E/S de suas instâncias implementam a interface `Serializable`
- use `ObjectInputStream` e `ObjectOutputStream` para ler/gravar um objeto `Serializable`
- `ObjectInputStream` e `ObjectOutputStream` implementam as interfaces `ObjectInput` e `ObjectOutput`, que por sua vez são subinterfaces de `DataInput` e `DataOutput`

SERIALIZABLE

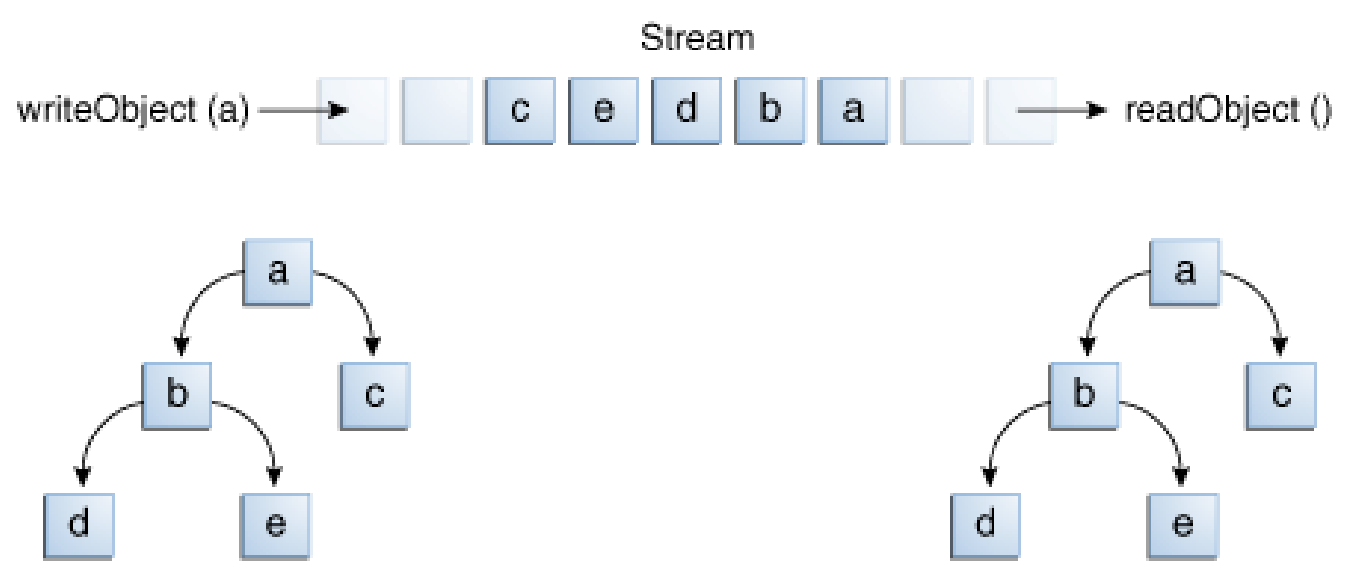
- a interface não impõe método nenhum. Por padrão todos os campos da classe serão gravados, exceto os estáticos e os `transient`
- permite que a implementação da classe assuma o controle do processo de seriação com os métodos:
  - `private void writeObject(java.io.ObjectOutputStream out)`

    — usa `out` para gravar os dados no fluxo
  - `private void readObject(java.io.ObjectInputStream in)`

    — instancia a classe usando o construtor padrão (sem argumentos) e então chama esse método para que ele leia os parâmetros e os inicialize corretamente
  - `private void readObjectNoData()`

    — chamado caso não seja possível restaurar o objeto usando os dados do fluxo

FLUXOS DE OBJETOS



- uma chamada a `writeObject(a)` grava não só o objeto `a`, mas todos os objetos necessários para recompor `a`
- um objeto só é escrito uma vez; `readObject` reconstitui as referências aos objetos

FLUXOS DE OBJETOS — ESCRITA

```
ObjectOutputStream out = null;
        try {
            out = new ObjectOutputStream(new
                    BufferedOutputStream(new FileOutputStream(dataFile)));

            out.writeObject(Calendar.getInstance());
            for (int i = 0; i < prices.length; i ++) {
                out.writeObject(prices[i]);
                out.writeInt(units[i]);
                out.writeUTF(descs[i]);
            }
        } finally {
            out.close();
        }
```

FLUXOS DE OBJETOS — LEITURA

```
ObjectInputStream in = null;
try {
    in = new ObjectInputStream(new
            BufferedInputStream(new FileInputStream(dataFile)));

    date = (Calendar) in.readObject();

    System.out.format("Em %tA, %<tB %<te, %<tY:%n", date);

    try {
        while (true) {
            price = (BigDecimal) in.readObject();
            unit = in.readInt();
            desc = in.readUTF();
            System.out.format("Você comprou %d unidades de %s por $%.2f%n",
                    unit, desc, price);
            total = total.add(price.multiply(new BigDecimal(unit)));
        }
    } catch (EOFException e) {}
    System.out.format("Por um TOTAL de: $%.2f%n", total);
} finally {
    in.close();
}
```

EXEMPLO 1 – USO DE SERIALIZABLE COM SERIAÇÃO PADRÃO

```java
import java.io.Serializable;
public class User implements Serializable {

    /**
     * Serial version ID (denota a versão da classe)
     */
    private static final long serialVersionUID = 42L;

    private String name;
    private String username;
    transient private String password;

    @Override
    public String toString() {
        String value = "name : " + name + "\nUserName : " + username
                + "\nPassword : " + password;
        return value;
    }
    // ...
}
```

Exemplo 2 – Uso de Serializable com seriação personalizada – java.util.ArrayList.writeObject()

```java
/**
 * Saves the state of the {@code ArrayList} instance to a stream
 * (that is, serializes it).
 *
 * @param s the stream
 * @throws java.io.IOException if an I/O error occurs
 * @serialData The length of the array backing the {@code ArrayList}
 *             instance is emitted (int), followed by all of its elements
 *             (each an {@code Object}) in the proper order.
 */
@java.io.Serial
private void writeObject(java.io.ObjectOutputStream s)
    throws java.io.IOException {
    // Write out element count, and any hidden stuff
    int expectedModCount = modCount;
    s.defaultWriteObject();

    // Write out size as capacity for behavioral compatibility with clone()
    s.writeInt(size);

    // Write out all elements in the proper order.
    for (int i=0; i<size; i++) {
        s.writeObject(elementData[i]);
    }

    if (modCount != expectedModCount) {
        throw new ConcurrentModificationException();
    }
}
```

Exemplo 2 – uso de Serializable com seriação personalizada – java.util.ArrayList.readObject()

```java
/**
 * Reconstitutes the {@code ArrayList} instance from a stream (that is,
 * deserializes it).
 * @param s the stream
 * @throws ClassNotFoundException if the class of a serialized object
 *         could not be found
 * @throws java.io.IOException if an I/O error occurs
 */
@java.io.Serial
private void readObject(java.io.ObjectInputStream s)
    throws java.io.IOException, ClassNotFoundException {

    // Read in size, and any hidden stuff
    s.defaultReadObject();
    // Read in capacity
    s.readInt(); // ignored

    if (size > 0) {
        // like clone(), allocate array based upon size not capacity
        SharedSecrets.getJavaObjectInputStreamAccess().checkArray(s, Object[].class, size);
        Object[] elements = new Object[size];

        // Read in all elements in the proper order.
        for (int i = 0; i < size; i++) {
            elements[i] = s.readObject();
        }
        elementData = elements;
    } else if (size == 0) {
        elementData = EMPTY_ELEMENTDATA;
    } else {
        throw new java.io.InvalidObjectException("Invalid size: " + size);
    }
}
```

BIBLIOGRAFIA

- The Java™ Tutorials – Basic I/O: https://docs.oracle.com/javase/tutorial/essential/io/
- Java Object Serialization: https://docs.oracle.com/javase/8/docs/technotes/guides/serialization/index.html
- Todd Greanier. Discover the secrets of the Java Serialization API: https://www.oracle.com/technical-resources/articles/java/serializationapi.html