# OPERAÇÕES DE ENTRADA E SAÍDA

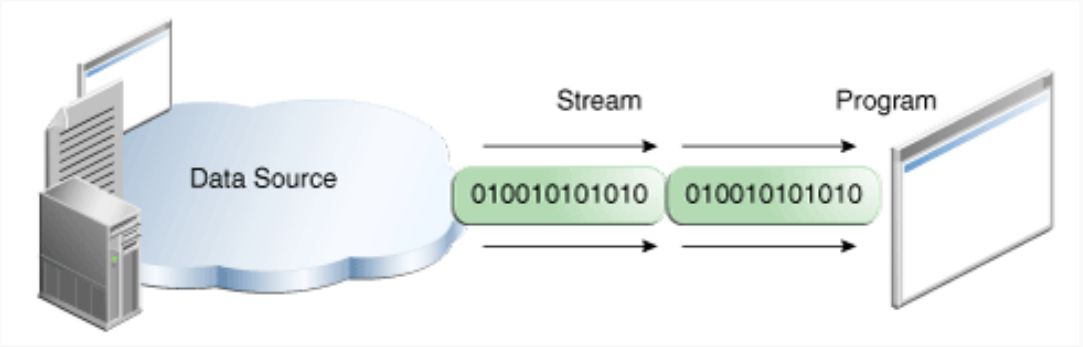
## ACH 2003 — COMPUTAÇÃO ORIENTADA A OBJETOS

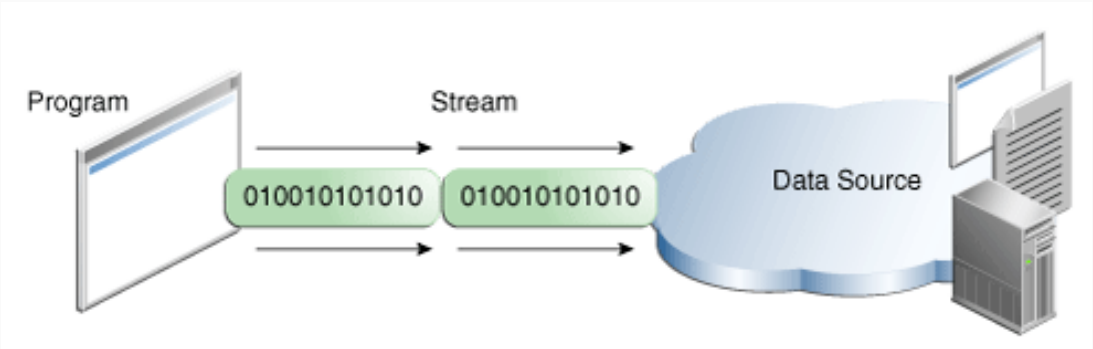
Daniel Cordeiro

13 de abril de 2016

Escola de Artes, Ciências e Humanidades | EACH | USP

# FLUXOS DE E/S




Vimos:

- fluxos de bytes (*byte streams*)
- fluxos de caracteres (*character streams*)

# BYTES

```
00000000: cafe babe 0000 0033 0061 0700 0201 0015  .......3.a......
00000010: 5374 7269 6e67 4361 6c63 756c 6174 6f72  StringCalculator
00000020: 3854 6573 7407 0004 0100 106a 6176 612f  8Test......java/
00000030: 6c61 6e67 2f4f 626a 6563 7401 0006 3c69  lang/Object...<i
00000040: 6e69 743e 0100 0328 2956 0100 0443 6f64  nit>...()V...Cod
00000050: 650a 0003 0009 0c00 0500 0601 000f 4c69  e.............Li
00000060: 6e65 4e75 6d62 6572 5461 626c 6501 0012  neNumberTable...
00000070: 4c6f 6361 6c56 6172 6961 626c 6554 6162  LocalVariableTab
00000080: 6c65 0100 0474 6869 7301 0017 4c53 7472  le...this...LStr
00000090: 696e 6743 616c 6375 6c61 746f 7238 5465  ingCalculator8Te
000000a0: 7374 3b01 002a 7768 656e 324e 756d 6265  st;..*when2Numbe
000000b0: 7273 4172 6555 7365 6454 6865 6e4e 6f45  rsAreUsedThenNoE
000000c0: 7863 6570 7469 6f6e 4973 5468 726f 776e  xceptionIsThrown
000000d0: 0100 1952 756e 7469 6d65 5669 7369 626c  ...RuntimeVisibl
000000e0: 6541 6e6e 6f74 6174 696f 6e73 0100 104c  eAnnotations...L
000000f0: 6f72 672f 6a75 6e69 742f 5465 7374 3b08  org/junit/Test;.
00000100: 0012 0100 0331 2c32 0a00 1400 1607 0015  .....1,2........
00000110: 0100 1153 7472 696e 6743 616c 6375 6c61  ...StringCalcula
00000120: 746f 7238 0c00 1700 1801 0003 6164 6401  tor8........add.
00000130: 0015 284c 6a61 7661 2f6c 616e 672f 5374  ..(Ljava/lang/St
00000140: 7269 6e67 3b29 490a 001a 001c 0700 1b01  ring;)I.........
```

Arquivo `.class` visto no hexl-mode do Emacs.

# E/S POR LINHAS

- é incomum manipularmos textos caractere a caractere
- normalmente o fazemos por *linhas*
- o término de uma linha é indicado por: uma sequência de *carriage-return/line-feed* ("\r\n"), um único *carriage-return* ("\r"), ou um único *line-feed* ("\n").

# CÓPIA LINHA A LINHA

```
import java.io.FileReader;      import java.io.FileWriter;
import java.io.BufferedReader; import java.io.PrintWriter;
import java.io.IOException;

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

        BufferedReader inputStream = null;
        PrintWriter outputStream = null;

        try {
            inputStream = new BufferedReader(new FileReader("xanadu.txt"));
            outputStream = new PrintWriter(new FileWriter("characteroutput.txt")

            String l;
            while ((l = inputStream.readLine()) != null) {
                outputStream.println(l);
            }
        } finally {
            if (inputStream != null) {
                inputStream.close();
            }
            if (outputStream != null) {
                outputStream.close();
            }
        }}}
```

# BUFFERED STREAMS

- operações de E/S com e sem uso de *buffer*
  - **sem uso** cada operação de leitura e escrita é realizada diretamente pelo sistema operacional
  - **com uso** dados são lidos de/escritos para uma área de memória (o *buffer* e só quando o *buffer* estiver vazio/cheio é que uma chamada ao sistema operacional é realizada
- um programa pode transformar um fluxo sem *buffer* em um com *buffer* usando uma técnica (*idiom*) de orientação a objetos chamada *wrapping* (embrulho)

**Para usar buffers no CopyCharacter:**

```
inputStream = new BufferedReader(
                      new FileReader("xanadu.txt"));
outputStream = new BufferedWriter(
                      new FileWriter("charoutput.txt"));
```

# BUFFERED STREAMS

Há quatro classes para embrulhar fluxos sem *buffers*:

- `BufferedInputStream` e `BufferedOutputStream` para criar fluxos de byte com *buffers*
- `BufferedReader` e `BufferedWriter` para criar fluxos de caracteres com *buffers*

## Flushing

- às vezes faz sentido querer gravar os dados do *buffer* antes que ele encha; chamamos isso de *flush*
- alguns fluxos com *buffer* possuem uma opção de `autoflush`, especificada no construtor, que realiza o *flush* após certas ações (ex: após o uso de `println` em `PrintWriter`)
- para realizar manualmente o *flush* de um fluxo, execute seu método `flush`

# SCANNING

- objetos do tipo `Scanner` são usados para quebrar uma entrada bem definida em "pedaços"[1] (*tokens*)
- por padrão, cada pedaço é separado por espaços em branco (o que inclui os caracteres de espaço, tabs e de quebras de linha)

[1] Ao pé da letra, a tradução de *token* seria símbolo, ficha, código, etc.

```
import java.io.*;
import java.util.Scanner;

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

      Scanner s = null;
      try {
          s = new Scanner(new BufferedReader(
                          new FileReader("xanadu.txt")));

          while (s.hasNext()) {
              System.out.println(s.next());
          }
      } finally {
          if (s != null) {
              s.close();
          }
      }}}
```

Note que:

- você deve chamar o método `close` do `Scanner`; mesmo que ele não seja um fluxo, você deve indicar que o `Scanner` pode fechar o fluxo que utilizou.
- para usar um delimitador diferente, use o método `useDelimiter()`.
  Ex: `s.useDelimiter(",\\s*");` (vírgula seguida de um ou mais espaços)

# CONVERSÃO DE TOKENS EM VALORES

```
import java.io.FileReader; import java.io.BufferedReader;
import java.util.Scanner; import java.io.IOException;

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

        Scanner s = null;
        double sum = 0;
        try {
            s = new Scanner(new BufferedReader(new FileReader("usnumbers.txt")))
            s.useLocale(Locale.US); // lá 32,767 é um inteiro, aqui é decimal

            while (s.hasNext()) {
                if (s.hasNextDouble()) {
                    sum += s.nextDouble();
                } else {
                    s.next();
                }
            }
        } finally {
            s.close();
        }

        System.out.println(sum); }}
```

- fluxos que implementam formatação são instâncias de `PrintWriter`, um fluxo de caracteres, ou `PrintStream`, fluxo de bytes
- `System.out` e `System.err` são provavelmente os únicos `PrintStream` que vocês irão utilizar
- use sempre `PrintWriter` para formatar sua saída

## Métodos para formatação

- `print` and `println` formatam valores individuais
- `format` formata vários valores usando uma string com o formato preciso

# PRINT E PRINTLN

```java
public class Root {
    public static void main(String[] args) {
        int i = 2;
        double r = Math.sqrt(i);

        System.out.print("The square root of ");
        System.out.print(i);
        System.out.print(" is ");
        System.out.print(r);
        System.out.println(".");

        i = 5;
        r = Math.sqrt(i);
        System.out.println("The square root of " + i + " is " + r + ".");
    }
}
```

## Saída:

```
The square root of 2 is 1.4142135623730951.
The square root of 5 is 2.23606797749979.
```

```java
public class Root2 {
    public static void main(String[] args) {
        int i = 2;
        double r = Math.sqrt(i);

        System.out.format("The square root of %d is %f.%n", i, r);
    }
}
```

## Saída:

The square root of 2 is 1.414214.

- **d** formata um valor inteiro como decimal
- **f** formata um valor de ponto flutuante como decimal
- **n** imprime a quebra de linha da plataforma
- **x** formata um inteiro como hexadecimal
- **s** formata qualquer valor como string
- **tB** formata um inteiro como o nome de um mês na língua definida

`https://docs.oracle.com/javase/8/docs/api/java/util/Formatter.html#syntax`

Fluxos padrão:

entrada padrão `System.in`

saída padrão `System.out`

saída de erro padrão `System.err`

- por razões históricas, `System.out` e `System.err` são fluxos de bytes e não de caracteres, apesar de se comportarem como fluxos de caracteres internamente
- `System.in` não se comporta como fluxo de caracteres, para usá-lo é preciso embrulhá-lo:

```
InputStreamReader cin =
        new InputStreamReader(System.in);
```

# CONSOLE

- em Java $\geq$ 1.6 a classe `Console` provê métodos para iteração com usuários
- possui métodos para leitura segura de senhas na linha de comando
- fornece fluxos **de caracteres** de entrada e saída que pelos métodos `reader` e `writer`
- uma instância do console deve ser obtida pelo método `System.console()`, que pode devolver `null` caso as operações no console não sejam permitidas

# CONSOLE – USO

```
import java.io.Console; import java.util.Arrays; import java.io.IOException;

public class Password {   // Troca a senha do usuário
    public static void main (String args[]) throws IOException {
        Console c = System.console();
        if (c == null) { System.err.println("No console."); System.exit(1); }

        String login = c.readLine("Enter your login: ");
        char [] oldPassword = c.readPassword("Enter your old password: ");

        if (verify(login, oldPassword)) {
            boolean noMatch;
            do {
                char [] newPassword1 = c.readPassword("Enter your new password:
                char [] newPassword2 = c.readPassword("Enter new password again:
                noMatch = ! Arrays.equals(newPassword1, newPassword2);
                if (noMatch) {
                    c.format("Passwords don't match. Try again.%n");
                } else {
                    change(login, newPassword1);
                    c.format("Password for %s changed.%n", login);
                }
                Arrays.fill(newPassword1, ' ');
                Arrays.fill(newPassword2, ' ');  } while (noMatch);
        }
        Arrays.fill(oldPassword, ' ');   }}
```

# BIBLIOGRAFIA


- The Java™ Tutorials – Basic I/O: `https://docs.oracle.com/javase/tutorial/essential/io/`