Lendo e Escrevendo Arquivos

Agenda

- Java I/O
- Java Files
- Streams
- Reader/Writer
- Serialization
- Socket
- Java NIO2

Console I/O

```
*************** CUNIT CONSOLE - MAIN MENU ***********************
(R)un all, (S)elect suite, (L)ist suites, Show (F)ailures, (Q)uit
Enter Command : r

Running Suite : Suite_success
        Running test : successful_test_1
        Running test : successful_test_2
        Running test : successful_test_3
WARNING - Suite initialization failed for Suite_init_failure.
Running Suite : Suite_clean_failure
        Running test : successful_test_4
        Running test : failed_test_2
        Running test : successful_test_1
WARNING - Suite cleanup failed for Suite_clean_failure.
Running Suite : Suite_mixed
        Running test : successful_test_2
        Running test : failed_test_4
        Running test : failed_test_2
        Running test : successful_test_4

--Run Summary: Type      Total     Ran  Passed  Failed
               suites        4       3     n/a       2
               tests        13      10       7       3
               asserts      10      10       7       3

*************** CUNIT CONSOLE - MAIN MENU ***********************
(R)un all, (S)elect suite, (L)ist suites, Show (F)ailures, (Q)uit
Enter Command :
```

```java
import java.io.Console;

public class Teste {

    Console con = System.console();
}
```

System.in

System.out

Usando a classe Scanner

Scanner engloba diversos métodos para facilitar a entrada de dados

```
public static void main(String[] args) {

    Scanner scan = new Scanner(System.in);
    String teste = scan.nextLine();
    System.out.println("palavra digitada: " + teste);
}
```

Streams

Os dados que entram e saem de um programa (I / O) são chamados de fluxo(Stream)

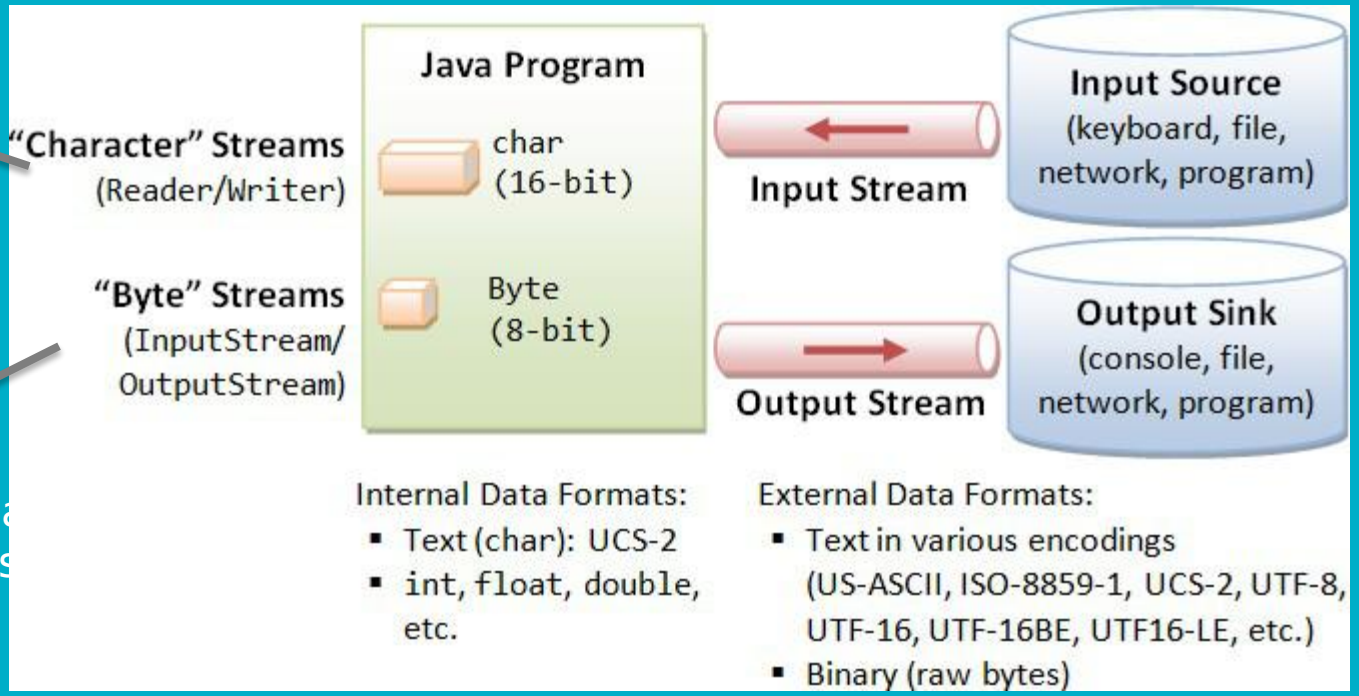
Os Stream são binário: baseado em byte

texto: baseado em caracteres (unicode)

A biblioteca java.io fornece classes para lidar com uma grande variedade de situações de I / O

I/O Stream

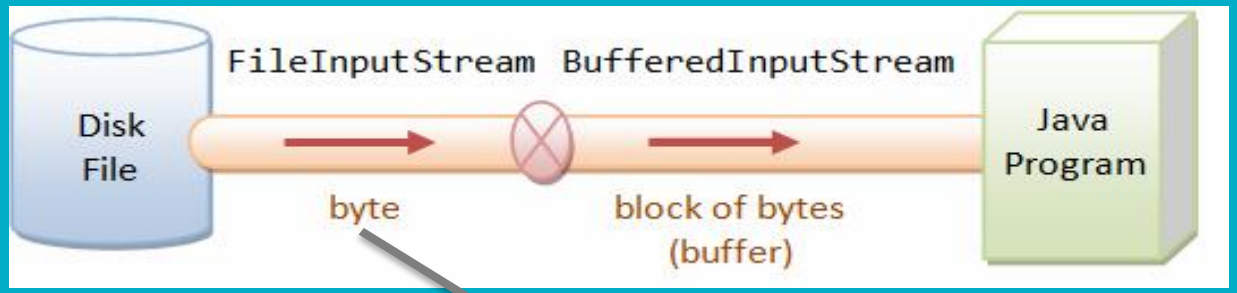
Usado para I/O arquivos de Texto Classes
InputStream
OutputStream



Usado para I/O a
de binários Class
InputStream
OutputStream

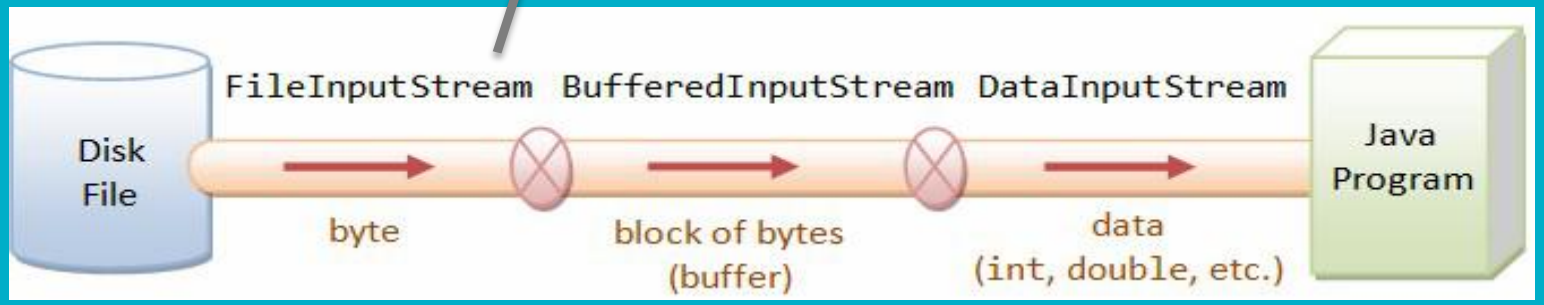
Stream, é uma cadeia de bits, como uma tubo conduzindo 'bits'! Lê escreve um caractere por vez.

Encadeando I/O Stream




```
InputStream bis = new BufferedInputStream(new FileInputStream(new File("file.zip")));

InputStream dis = new DataInputStream(new BufferedInputStream(new FileInputStream(new File("file.dat"))));
```



Java I/O Classes

- Text I/O
  - Stream of characters (Unicode format)
  - Support provided by ***Reader*** and ***Writer*** classes
- Binary I/O
  - Stream of bytes (raw format)
  - Support provided by ***InputStream*** and ***OutputStream*** classes

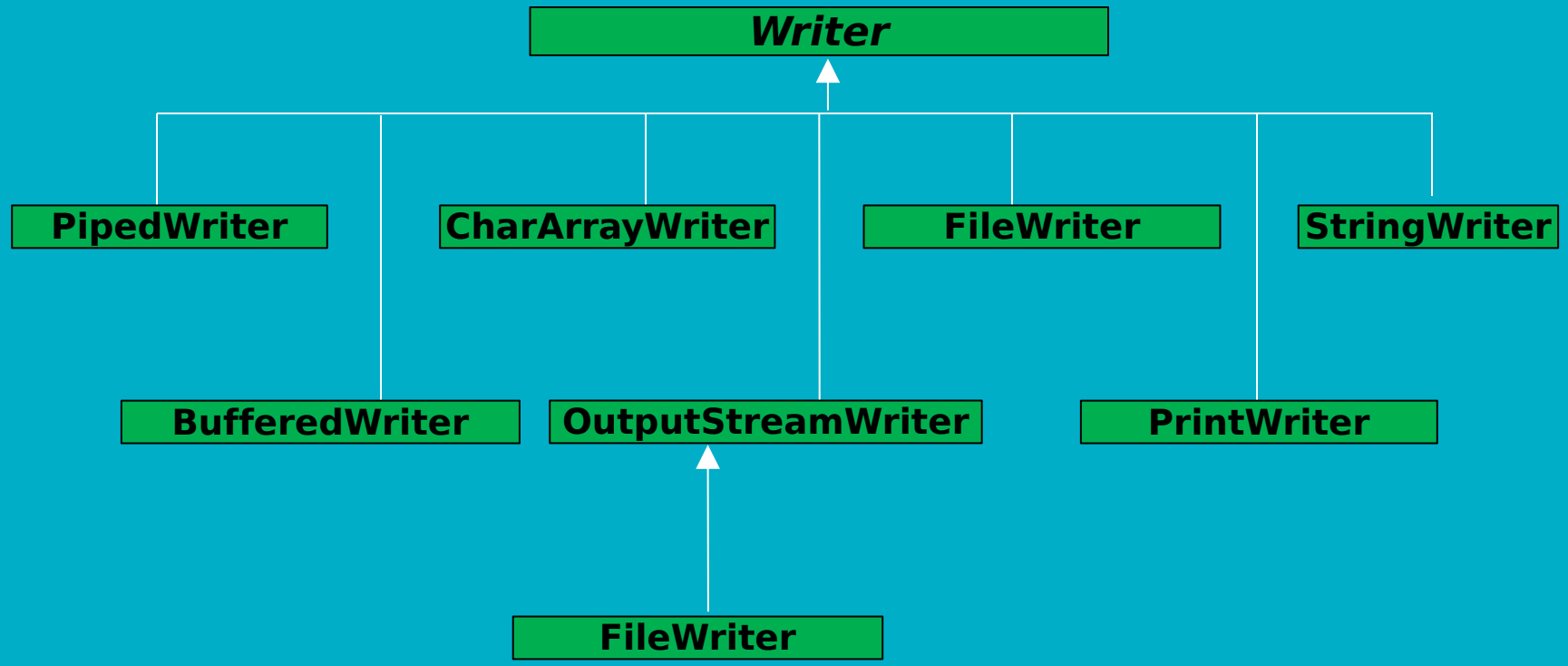
Text Files

- A ***text file*** is a common way to organize a file as a sequence of lines.
  - Each line is a sequence of characters
  - Each OS's file system has its own way to mark the ends of lines
  - java.io abstracts this in a consistent way

- Information from text files must be parsed to identify meaningful components
  - The Scanner class helps with parsing

Character-Oriented Writer Classes

A seguir está a hierarquia de classes do fluxo de entrada orientada a bytes:



FileWriter e BufferedWriter

```java
public static void main(String[] args) {

    try {

        File arquivo = new File("C:\\teste.txt");

        FileWriter fw = new FileWriter(arquivo);

        BufferedWriter bw = new BufferedWriter(fw);

        bw.write("Texto a ser escrito no txt");
        bw.newLine();
        bw.write("Quebra de linha");

        bw.close();

        fw.close();

    } catch (IOException e) {
        System.out.println("Arquivo não existe!");
    }

}
```

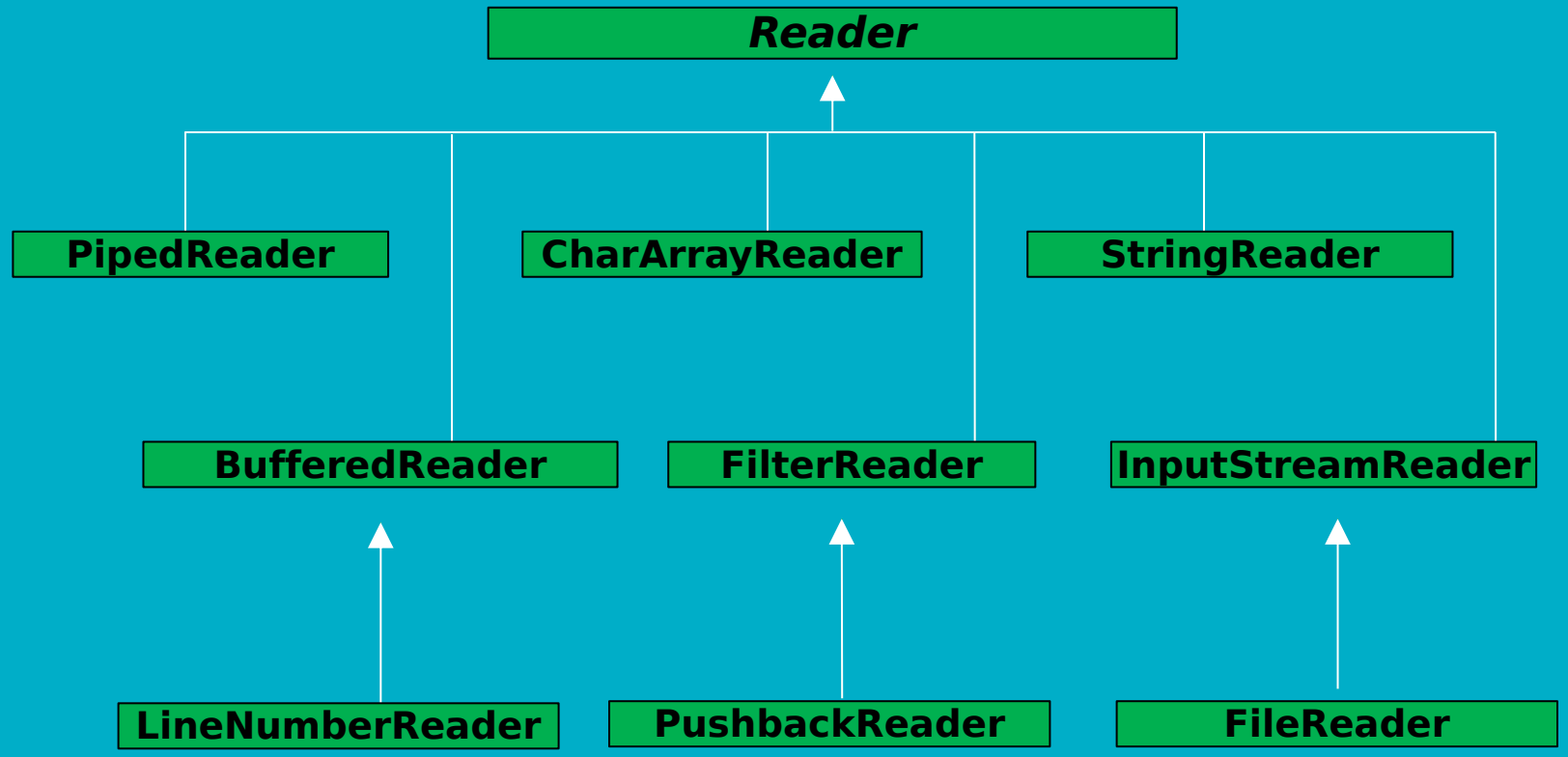
Um arquivo, caminho absoluto

Encadeando para FileWriter

Encadeando para BufferedWriter

Character-Oriented Reader Classes

A seguir está a hierarquia de classes do fluxo de entrada orientada a bytes:



FileReader e BufferedReader

```
public static void main(String[] args) {

    try {

        File arquivo = new File("C:\\teste.txt");

        FileReader fr = new FileReader(arquivo);

        BufferedReader br = new BufferedReader(fr);

        while (br.ready()) {
            String linha = br.readLine();
            System.out.println(linha);
        }

        br.close();
        fr.close();

    } catch (FileNotFoundException e) {
        System.out.println("Arquivo não existe!");
    } catch (IOException e) {
        System.out.println("Erro ao ler arquivo!");
    }

}
```

Um arquivo, caminho absoluto

Encadeando para FileReader

Encadeando para BufferedReader

Binary Files

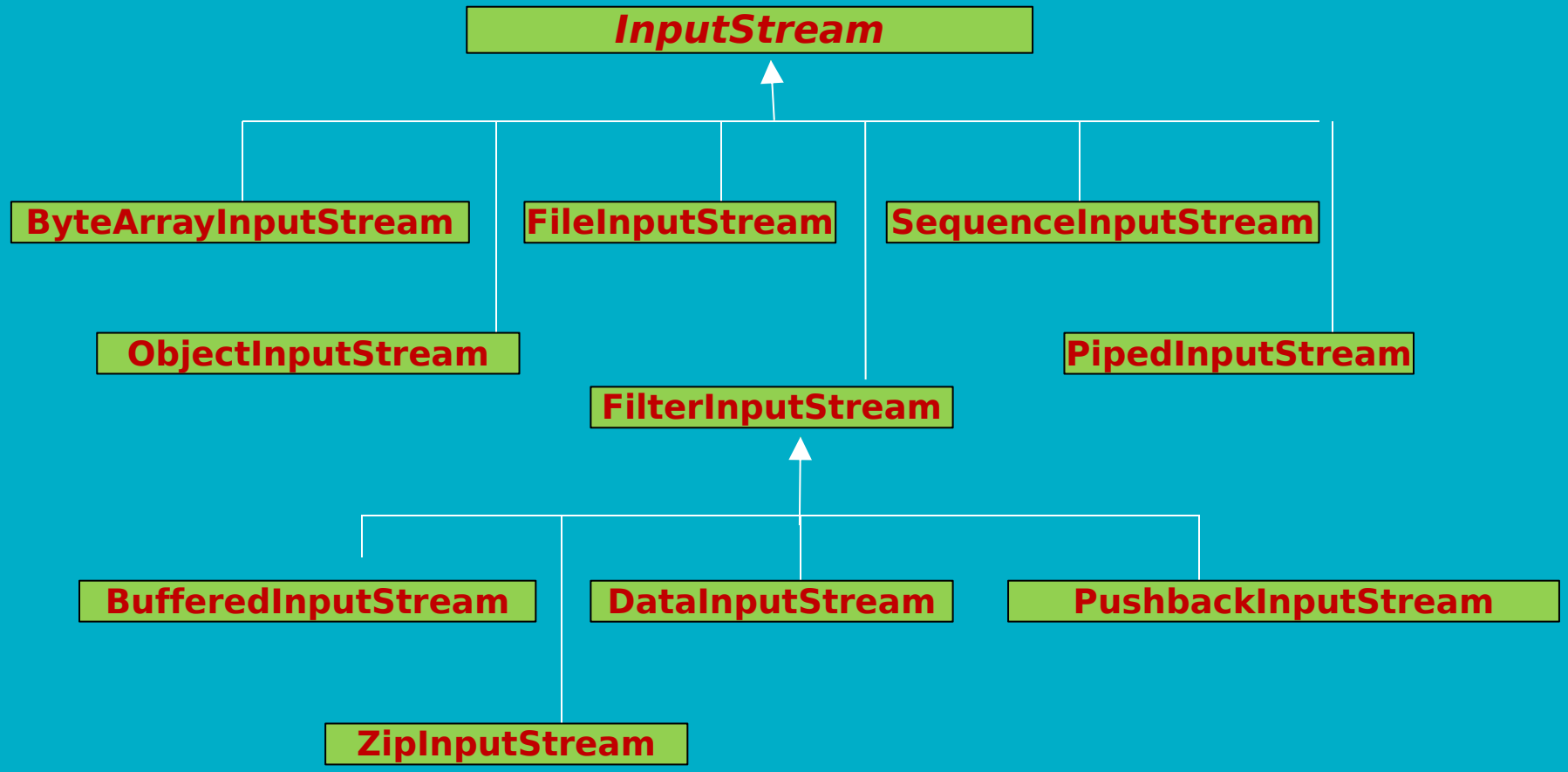
- The term binary file is used for every other type of file organization
  - Interpreting binary files requires knowledge of how the bytes are to be grouped and interpreted
    - Text files are also binary files;
    - but the bytes have predefined meanings (character and line data)
- Binary files provide highly efficient storage
  - Java allows entire objects to be serialized as byte sequences for this purpose

FileReader / FileWriter

- **FileReader** extends
  - **InputStreamReader** extends
    - **Reader** extends **Object**
- fr = new FileReader(*location of a file*);
  - Connects to and ***opens*** the file for character input
- **FileWriter** extends
  - **OutputStreamWriter** extends
    - **Writer** extends **Object**
- fw = new FileWriter(*location of a file*);
  - Creates and ***opens*** the file for character output
  - If the file exists, it is erased

Byte-Oriented Input Stream Classes

The following is the byte-oriented input stream class hierarchy:



ZipInputStream is defined in: java.util.zip

InputStream Methods

Reading

read() methods will block until data is available to be read

two of the three read() methods return the number of bytes read

-1 is returned if the Stream has ended

throws IOException if an I/O error occurs. This is a checked exception

There are 3 main read methods:

int read()

Reads a single character. Returns it as integer

int read(byte[] buffer)

Reads bytes and places them into buffer (max = size of buffer)

returns the number of bytes read

int read(byte[] buffer, int offset, int length)

Reads up to length bytes and places them into buffer

First byte read is stored in buffer[offset]

returns the number of bytes read

InputStream Methods

- **available()** method returns the number of bytes which can be read without blocking
- **skip()** method skips over a number of bytes in the input stream
- **close()** method will close the input stream and release any system resources
- input streams optionally support repositioning the stream

  can mark the stream at a certain point and 'rewind' the stream to that point later.
- methods that support repositioning are:

  markSupported() returns true if repositioning is **supported** **mark()** places a mark in the stream

  reset() 'rewinds' the stream to a previously set mark

Creating an InputStream

InputStream is an abstract class

- Programmers can only instantiate subclasses.

ByteArrayInputStream:

- Constructor is provided with a byte array.
- This byte array contains all the bytes provided by this stream
- Useful if the programmer wishes to provide access to a byte array using the stream interface.

FileInputStream:

- Constructor takes a filename, File object or FileDescriptor Object.
- Opens a stream to a file.

FilterInputStream:

- Provides a basis for filtered input streams

Creating an InputStream

ObjectInputStream

Created from another input stream (such as FileInputStream)
Reads bytes from the stream (which represent serialized Objects) and converts them back into Objects
More on Serialization later in the Chapter.

PipedInputStream:

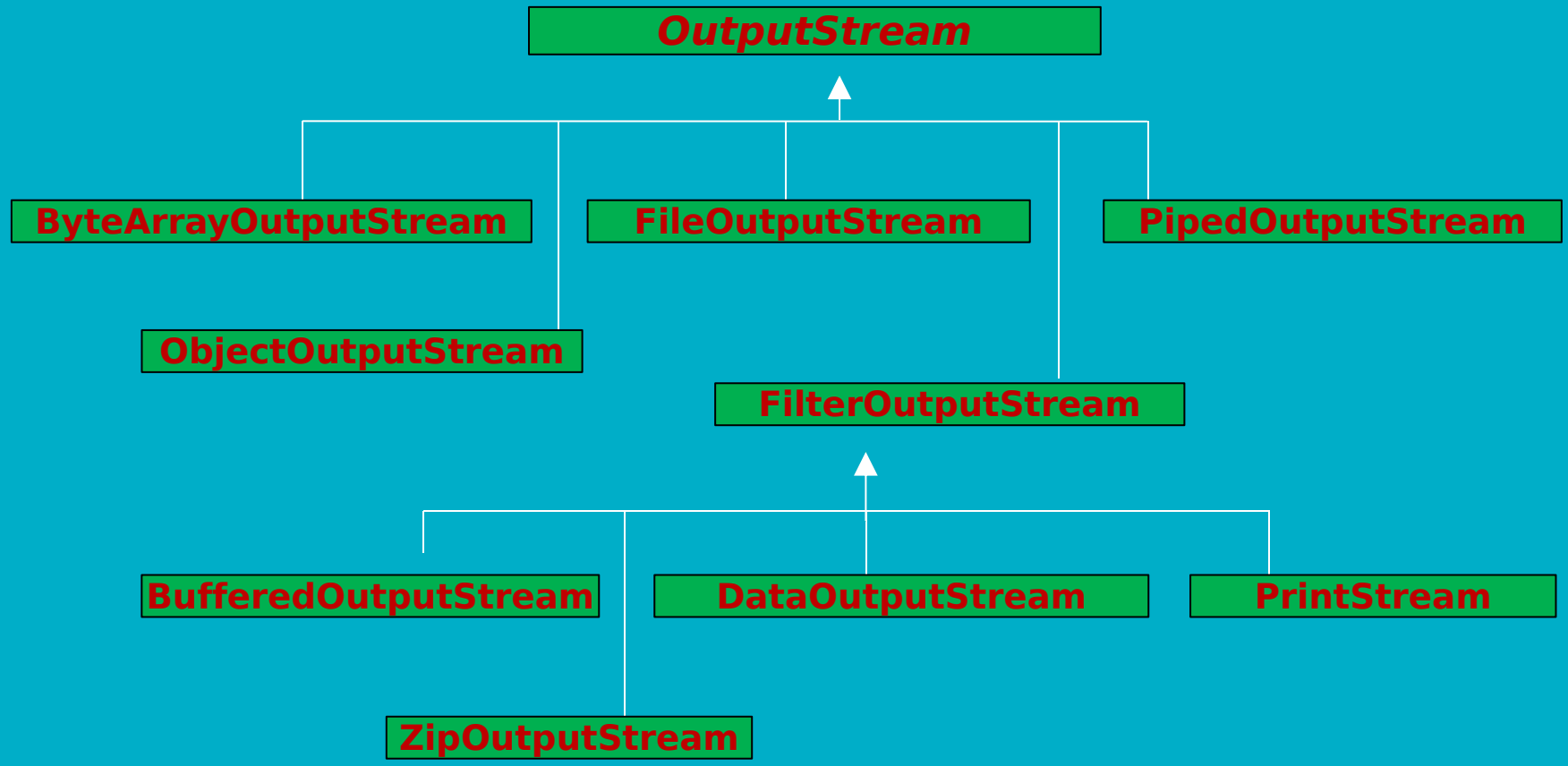
Connects to an Instance of PipedOutputStream
A pipe represents a one-way stream through which 2 threads may communicate
- Thread1 writes to a PipedOutputStream
- Thread2 reads from the PipedInputStream

SequenceInputStream:

Constructor takes multiple InputStreams
Allows reading. When one stream ends, it continues reading from next stream in the list

Byte-Oriented Output Stream Classes

The following is the byte-oriented input stream class hierarchy:



ZipOutputStream is defined in: java.util.zip

OutputStream Methods

Writing:

write() methods write data to the stream. Written data is buffered.
Use **flush()** to flush any buffered data from the stream.
throws **IOException** if an I/O error occurs. This is a checked exception

There are 3 main write methods:

void write(int data)

Writes a single character
Note: even though data is an integer, data must be set such that:
0 <= data <= 255

void write(byte[] buffer)

Writes all the bytes contained in buffer to the stream

void write(byte[] buffer, int offset, int length)

Writes length bytes to stream starting from buffer[offset]

OutputStream Methods

flush()

To improve performance, almost all output protocols buffer output.
Data written to a stream is not actually sent until buffering thresholds are met.
Invoking **flush()** causes the OutputStream to clear its internal buffers.

close()

Closes stream and releases any system resources.

Creating an OutputStream

OutputStream is an abstract class.

Programmers instantiate one of its subclasses

ByteArrayOutputStream:

Any bytes written to this stream will be stored in a byte array

The resulting byte array can be retrieved using toByteArray() method.

FileOutputStream:

Constructor takes a filename, File object, or FileDescriptor object.

Any bytes written to this stream will be written to the underlying file.

Has one constructor which allows for appending to file:

```
FileOutputStream(String filename, boolean append)
```

FilterOutputStream:

Provides a basis for Output Filter Streams.

Will be covered later in chapter.

Creating an OutputStream

ObjectOutputStream

Created from another output stream (such as FileOutputStream)

Programmers serialize objects to the stream using the writeObject() method

More on Serialization later in the Chapter.

- **PipedOutputStream:**

Connects to an Instance of PipedInputStream

A pipe represents a one-way stream through which 2 threads may communicate

Thread1 writes to a PipedOutputStream

Thread2 reads from the PipedInputStream

FileReader Example

```
FileReader inf = new FileReader("filename");
int chCode;
while(-1 != (chCode=inf.read()) )
  System.out.println(
    "Next char: "+(char)chCode);

inf.close();
```

Returned int

- Why does Reader.read() return int, not char ?
- Because you may read an eof
- which is -1
- and you'd have no way to distinguish between eof and a valid char value otherwise

Other Reader Methods

- Reader.read() is not commonly used
- Some other methods are (usually) better
  - int read(char[] cbuf, int off, int len)
  - int read(char[] cbuf)
  - int read(CharBuffer target)

FileWriter

```
FileWriter outf = new FileWriter("filename");

outf.write('A');
outf.write('\n');
outf.write("Strings too!\n");

outf.close();
```

Reader & Writers

FileReaders and **FileWriters** provide only very basic IO capabilities

The **read** and **write** methods are also overloaded to read and write an array of characters

FileWriter has a constructor with a boolean parameter

It can be used for appending the file

- FileWriter(String fileName, boolean append)

FileInputStream/FileOutputStream

FileInputStream extends
- **InputStream** extends **Object**

fr = new FileInputStream(*location of a file*);
- Connects to and ***opens*** the file for byte-oriented inpu

FileOutputStream extends
- **OutputStream** extends **Object**

fw = new FileOutputStream(*location of a file*);
- Creates and ***opens*** the file for byte-oriented output
- If the file exists, it is erased

FileInputStream

```
import java.io.File;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.IOException;

public class HandlerFileInputStream {

	public static void main(String[] args) {
		try {
		File myObj = new File("/home/weder/arquivo2.txt");
		FileInputStream inf = new FileInputStream(myObj);
		int bCode;

		while(( bCode= inf.read()) != -1 )
		  System.out.println( "Next byte: "+(byte)bCode);

		inf.close();
		}catch (FileNotFoundException e) {
			System.out.println("Error arquivo nao encontrado");
		}
		catch (IOException e) {
			System.out.println("Error de Leitura");
		}

	}

}
```

- Some other InputStream methods:
- int read(byte b[])
- int read(byte b[], int off, int len)

FileOutputStream

```
import java.io.File;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;

public class HandlerFileOutputStream {

	public static void main(String[] args) {
		try {
		File myObj = new File("/home/weder/arquivo7.txt");
		FileOutputStream  outf = new FileOutputStream(myObj);
		byte[] out = {87, 69, 68, 69, 82};
		outf.write(out);
		outf.close();

		}catch (FileNotFoundException e) {
			System.out.println("Error arquivo nao encontrado");
		}
		catch (IOException e) {
			System.out.println("Error de Leitura");
		}

	}

}
```

InputStream/OutputStream

- **FileInputStream** and **FileOutputStream** provides the same basic IO capabilities
- Transfer is in bytes rather than characters.
- There are no "lines" in these files.

- How to append to a file
  - FileOutputStream(String name, boolean append)

Paths and Filenames

- Microsoft chose to use the *backslash* character in path names
  - new FileReader("c:\textfiles\newfile.txt");
- What is wrong with this file name?
- In Java the *backslash* character in a String literal is an *escape character*
  - "c:**{tab}**textfiles**{newline}**newfile.txt"
- Either type *double* backslashes in String literals, or use the *forward slash*
  - "c:\\textfiles\\newfile.txt"
  - "c:/textfiles/newfile.txt"

NIO.2 - Path

Path pode ser um arquivo no diretório atual, caminho relativo ao programa

Path pode ser um arquivo, caminho absoluto, no windows use caractere de escape '\'

Path pode ser um diretório

```
Path p1 = Paths.get("in.txt");

Path p2 = Paths.get("c:\\projetos\\java\\Hello.java");

Path p3 = Paths.get("/use/local");
```

JAVA 7 NIO.2, mais simples, Buffered, sem bloqueio de IO. A classe **java.nio.Path**, especifica a localização de um arquivo, ou diretório, ou link simbólico.
Substitui **java.io.File**

NIO.2 – I/O Streams

```
String fileStr = "small_file.txt";
Path path = Paths.get(fileStr);
List<String> lines = new ArrayList<String>();
lines.add("Oi,您好! Olá,吃饱了没有?");

try {
   Files.write(path, lines, Charset.forName("UTF-8"));
} catch (IOException ex) {
   ex.printStackTrace();
}

byte[] bytes;
try {
   bytes = Files.readAllBytes(path);
   for (byte aByte: bytes) {
      System.out.printf("%02X ", aByte);
   }
   System.out.printf("%n%n");
} catch (IOException ex) { }

List<String> inLines;
try {
   inLines = Files.readAllLines(path, Charset.forName("UTF-8"));
   for (String aLine: inLines) {
      for (int i = 0; i < aLine.length(); ++i) {
         char charOut = aLine.charAt(i);
         System.out.printf("[%d]'%c'(%04X) ", (i+1), charOut, (int)charOut);
      }
      System.out.println();
   }
} catch (IOException ex) {}
```

Um arquivo no diretório atual, caminho relativo ao programa

Escreve dados para arquivo texto

Lê dados do arquivo como bytes

Lê dados do arquivo como caracteres UTF-8

Use com arquivos pequenos

NIO.2 – I/O Streams

```
InputStream in = Files.newInputStream(path);

OutputStream out = Files.newOutputStream(path);

Reader reader = Files.newBufferedReader(path);

Writer writer = Files.newBufferedWriter(path);
```

Leitura ou escrita orientada por streams, um caractere por vez.

```
try (OutputStream out = new BufferedOutputStream(
  Files.newOutputStream(p,StandardOpenOption.CREATE,
          StandardOpenOption.APPEND))) {
  out.write(data, 0, data.length);
} catch (IOException x) {
  System.err.println(x);
}
```

Compatibilidade com Java I/O Básico

Use com arquivos Grandes, para usar streams compativeis com java IO

NIO.2 – I/O Channel

```
private void leia(Path path) {

    try (SeekableByteChannel sbc = Files.newByteChannel(path)) {

        ByteBuffer buf = ByteBuffer.allocate(64);

        while (sbc.read(buf) > 0) {
            buf.rewind();
            System.out.print(Charset.forName("UTF-8").decode(buf));
            buf.flip();
        }

    } catch (IOException e) {
        log.warning(e.toString());
    }
}
```

I/O de arquivos conectados a um channel, dados são lidos para o buffer

O método rewind() muda ponteiro para inicio do buffer e o deixa pronto para leitura

O método flip() muda
ler dados a partir do escrever para arquivo

Use com arquivos Grandes. Channel lê um buffer por vez

RandomAccessFile

- This class is not a reader/writer
- nor a inputstream/outputstream
- You can use file as binary or text file
- Used to access desired location of file
- For read or write
- It has a **file pointer**
  - The place where you *read from/write into* the file
- You can move file pointer using ***seek(long)*** method
- It has different methods for reading and writing

```
import java.io.FileNotFoundException;
import java.io.IOException;
import java.io.RandomAccessFile;

public class HandleRandomAcessFile {

	public static void main(String[] args) {
		try {
		RandomAccessFile raf = new RandomAccessFile("/home/weder/arquivo7.txt", "rw");

		byte ch = raf.readByte();
		System.out.println("first character : " + (char)ch);
		ch = raf.readByte();
		System.out.println("second character : " + (char)ch);
		String line = raf.readLine();
		System.out.println("Read a line: " + line);
		raf.seek(5);
		float fl = raf.readFloat();
		System.out.println("Read a float from index 5: " + fl);

		raf.seek(26);

		raf.write('\r');
		raf.write('\n');
		raf.writeDouble(1.2);
		raf.writeBytes("This will complete the Demo");

		raf.close();
	}catch (FileNotFoundException e) {
		System.out.println("Error arquivo nao encontrado");
	}
	catch (IOException e) {
		System.out.println("Error de Leitura");
	}

	}
}
```

File Class

- The *java.io.File* class abstracts the connection to and properties of a file or folder (directory)
- It does not offer read/write operations

- File f = new File("c:/data/sample.txt");
  - Sample methods: f.delete(); f.length(); f.isFile(); …
- File d = new File("c:/");
  - This object represents a folder, not a file

File Methods

- **boolean canRead();**
- **boolean canWrite();**
- **boolean canExecute();**
- **boolean exists();**
- **boolean isFile() ;**
- **boolean isDirectory() ;**
- **boolean isAbsolute() ; //constructed by ”1” or “c:/test/1”**
- **String getName();**
- **String getPath(); // “1”**
- **String getAbsolutePath() ; // “c:/test/1”**
- **String getParent();**
- **long length() ;//zero for folders**
- **long lastModified() ;**
- **String[] list() ;**

Scanner

- The Scanner class is not technically an I/O class
- It is found in **java.util**
- You can use a Scanner wrapped around any InputStream object to provide sophisticated token-oriented input methods
  - new Scanner(System.in);
  - new Scanner(new FileInputStream("t.txt"));

  - scanner = new Scanner(new File("sample.txt));
  - scanner.nextDouble()
  - scanner.next()

Formatter

- Also found in **java.util**
- Used to format output to text files
  - Formatter f = new Formatter("afile.txt");
  - Formatter g = new Formatter(aFileObject);
- The format method is the most important
  - f.format("x=%d; s=%s\n", 23, "skidoo");
  - similar to printf in C++
- The stream can be closed using…
  - g.close();

Serialization

- Most Objects in Java are ***serializable***
  - Can turn themselves into a stream of bytes
  - Can reconstruct themselves from a stream of bytes
- A serialized object includes all instance variables
  - Unless marked as ***transient***
  - Members that are Object references are also serialized
- Serializable is an interface
- The serialized file is a binary file
  - Not a text file

```
public class Student implements Serializable{
    private String name;
    private String studentID;
    private double[] grades ;
    private transient double average = 17.27;


    public Student(String name, String studentID, double[] grades) {
        this.name = name;
        this.studentID = studentID;
        this.grades = grades;
    }
    public double getAverage() {
        double sum = 0;
        if(grades==null)
            return -1;
        for (double grade : grades) {
            sum+=grade;
        }
        return sum/grades.length;
    }
    //setters and getters for name, studentID and grades
}
```

Object Serialization

```
ObjectOutputStream output =
   new ObjectOutputStream(
   new FileOutputStream("c:/1.txt"));
Student student =
   new Student("Ali Alavi", "88305489", new
   double[]{17.2, 18.9, 20, 13});


output.writeObject(student);
output.close();
```

Object Deserialization

```
ObjectInputStream stream =
   new ObjectInputStream(
   new FileInputStream("c:/1.txt"));
Student student =
   (Student) stream.readObject();

System.out.println(student.getName());
System.out.println(student.getAverage());
stream.close();
```

java.net.Socket

- This class implements client sockets
  - also called just "sockets"
- A socket is an endpoint for communication between two machines.
- A stream of data is communicated between two nodes
- Very similar to local I/O operations

Writing into Socket

```
Socket socket = new Socket(“192.168.10.21", 8888);
OutputStream outputStream =
   socket.getOutputStream();
Formatter formatter = new Formatter(outputStream);
formatter.format("Salam!\n");
formatter.flush();
formatter.format("Chetori?\n");
formatter.flush();
formatter.format("exit");
formatter.flush();
socket.close();
System.out.println("finished");
```

Reading from a Socket

```
InputStream inputStream = socket.getInputStream();
Scanner scanner = new Scanner(inputStream);

while(true){
    String next = scanner.next();
    if(next.contains("exit"))
        break;
    System.out.println("Server : " + next);
    System.out.flush();
}
socket.close();
```

ServerSocket

- How to listen to other sockets?
- What do yahoo and google do?

```
ServerSocket serverSocket = new
  ServerSocket(8888);
Socket socket = serverSocket.accept();
```

Binary or Text?

- You can use a socket as a binary or text stream

The First Version of Java I/O APIs

- java.io package
- The **File** class limitations:
  - more significant functionality required (e.g. copy method)
  - defines many methods that return a **Boolean** value
    - In case of an error, an exception is better than a simple **false.**
  - Poor support for handling symbolic links
  - inefficient way of handling directories and paths
  - very limited set of file attributes

Java New IO (NIO)

- Introduced in Java 1.4 (2002)
- The key features of NIO were:
- **Channels and Selectors**
- **Buffers**
- **Charset**
  - java.nio.charset
  - encoders, and decoders to map bytes and Unicode symbols

NIO.2

- Introduced in Java 1.7 (2011)
- Java 7 introduces the **java.nio.file** package
- New interfaces and classes
  - **Path**, **Paths**, and **Files**

Path and Paths

- **Path** is an interface while **Paths** is a class

```
Path testFilePath = Paths.get("D:\\test\\testfile.txt");

// retrieve basic information about path
System.out.println("Printing file information: ");
System.out.println("\t file name: " + testFilePath.getFileName());
System.out.println("\t root of the path: " + testFilePath.getRoot());
System.out.println("\t parent of the target: " + testFilePath.getParent());

// print path elements
System.out.println("Printing elements of the path: ");
for(Path element : testFilePath) {
        System.out.println("\t path element: " + element);
}
```

Path interface

```
Path dirName = Paths.get("D:\\OCPJP7\\programs\\NIO2\\");
Path resolvedPath = dirName.resolve("Test");
System.out.println(resolvedPath);
```

```
D:\OCPJP7\programs\NIO2\Test
```

- The **toPath()** method in the **java.io.File** class
  - returns the Path object; this method was added in Java 7
- The **toFile()** method in the **Path** interface to get a File object

The **Files** Class

- the **java.nio.file** package
- Provides **static** methods for **copy**, **move**, **delete, …**
- New methods for
  - Symbolic linked files
  - Attributes
  - …

copy

```
Path pathSource = Paths.get(str1);
Path pathDestination = Paths.get(str2);
Files.copy(pathSource, pathDestination);
```

- it will not copy the files/directories contained in the source directory
- you need to explicitly copy them to the destination folder

Listening for Changes

```
Path path = Paths.get("..\\src");
WatchService watchService = null;


watchService =
path.getFileSystem().newWatchService();
path.register(watchService,
StandardWatchEventKinds.ENTRY_MODIFY);
```

Summary

- Streams access sequences of bytes
- Readers and Writers access sequences of characters
- FileReader, FileWriter, FileInputStream, FileOutputStream are the 4 major file access classes
- Scanner provides sophisticated input parsing
- Formatter provides sophisticated output formatting

Summary

- Most objects can be serialized for storage in a file
- The File class encapsulates files and paths of a file system

Further Reading

- Other java I/O classes
  - Buffered input and output
- Decorator pattern in java.io classes
- java.nio
- Socket Programming
- Object serialization applications
  - RMI

Perguntas?