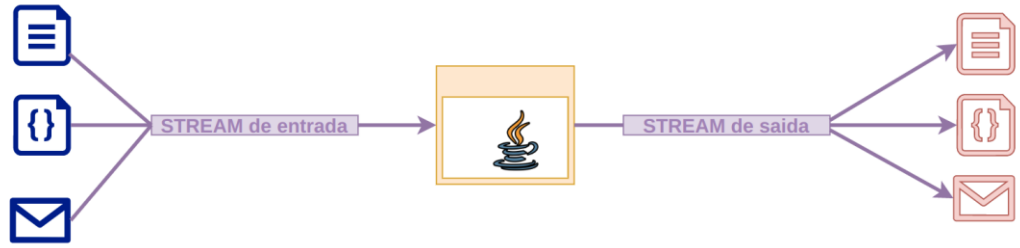
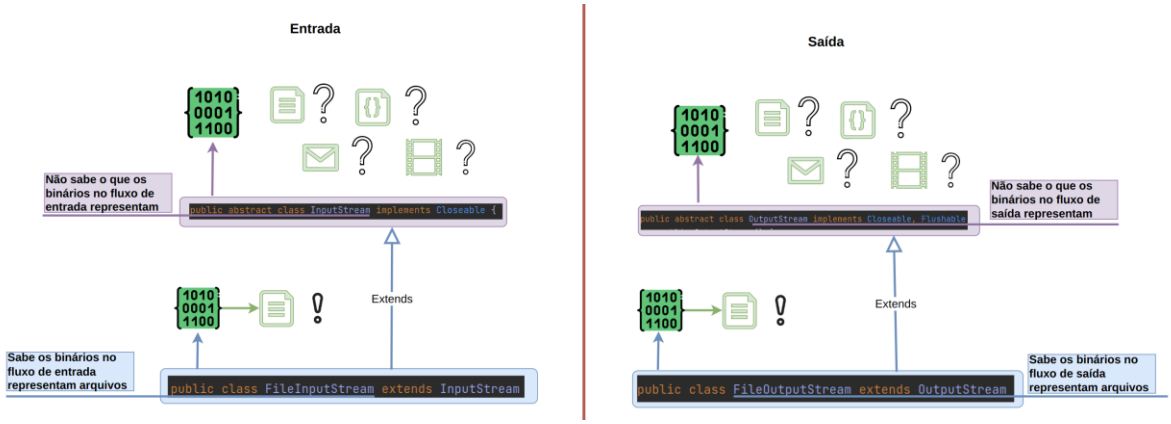
Stream de dados – Java

Dados podem vir de qualquer fonte diferente, de um ***arquivo, de um Json, de uma requisição web e etc...*** Essas **entradas** e **saídas** de **dados** são tratadas como **Stream** em uma **aplicação Java:**



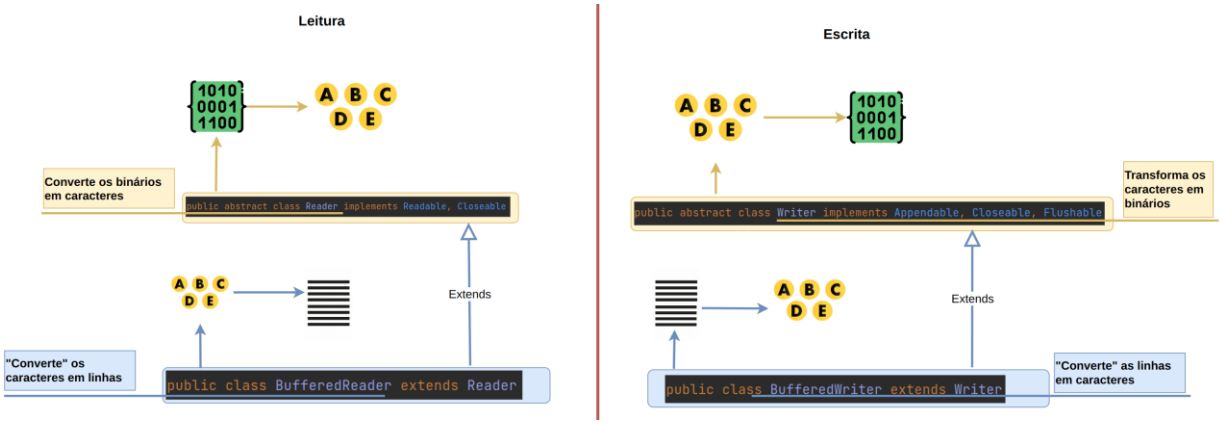
Existem duas famílias principais do Java para se tratar com entrada e saída de dados. São elas a ***InputStream*** e a ***OutputStream,*** respectivamente **entrada** e **saída** de dados.

Esses **Streams** são abstratos, o que exige uma **especialização** **da leitura e saída** **deles**, e pra isso o **Java** também oferece diversas implementações:



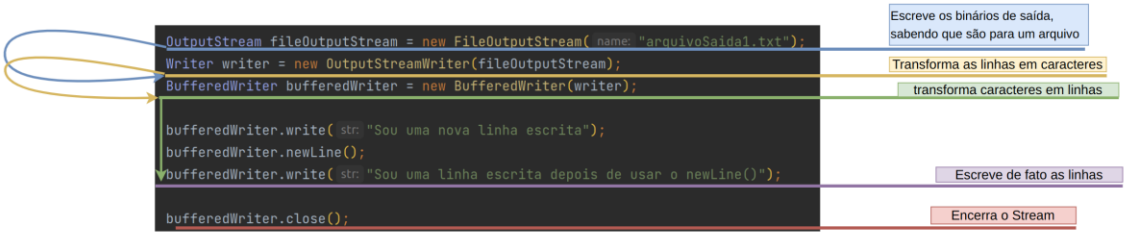
Leitura e escrita de arquivos – Java

Ainda que existam ***Stream de entrada e saída especializados,*** no fim das contas a **stream é só bit**. Eu não sei ler bits, nem quero saber também. E é por isso que o Java traz o conceito de **Readers** e **Writers,** basicamente falando eles humanizam os **bits de stream** em ***caracteres e linhas entendidas por humanos:***



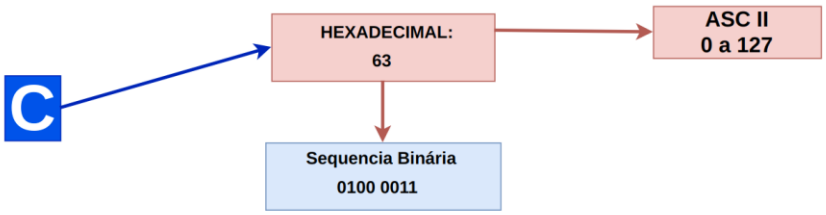
É assim que lemos um arquivo:

É assim que escrevemos um arquivo:



Encoding – Java

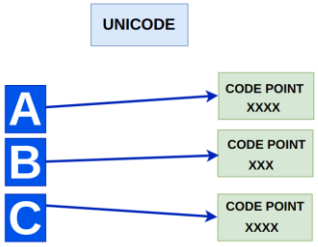
Alguém teve que **definir que cada letra seria representada por uma sequência determinada de bits**. Na verdade, existem várias dessas definições, uma das mais famosas é o **ASC II** que definia todas as



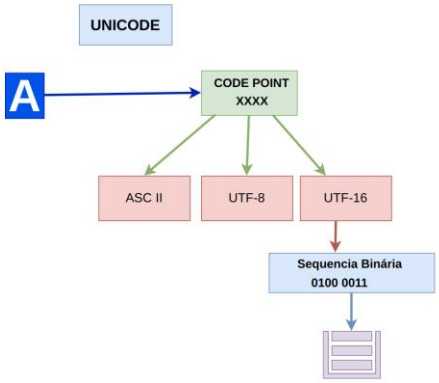
letras do alfabeto inglês:

Acontece que só não existem caracteres do alfabeto americano no mundo, então foram sendo criadas outras tabelas como extensões da ASC II, dezenas e dezenas de tabelas. Obviamente isso gerou dor de cabeça quando em desenvolvimento era preciso trocar informações globalizadas.

Como alternativa surgiu a tabela **UNICODE**, uma única tabela imensa com todos os **caracteres** do mundo sendo representados por **CODEPOINTS:**



A **Unicode** não grava fisicamente os **codepoints** no HD, isso é trabalho dos **Encodings**, que são outras tabelas que usam esse **CODEPOINT** para traduzir para uma **sequência binária** que vai ser de fato escrita no HD. O **java** funciona em cima desse conceito também:




```
String caractere = "A";
caractere.codePointAt( index: 0); Code point de: A = 65

Charset charsetDefault = Charset.defaultCharset();
charsetDefault.displayName();          Encoding padrão do sistema: UTF-8

byte[] charBytes = caractere.getBytes();
int length = charBytes.length;   Bytes do caractere A: 1   UTF-8

byte[] convertedBytes = caractere.getBytes(StandardCharsets.UTF_16);
int length1 = convertedBytes.length;Bytes do caractere A convertidos Para UTF-16: 4

String bytesToStringEncoding = new String(charBytes, StandardCharsets.US_ASCII);
```

De maneira simples, para ler ou escrever um arquivo usando um **Encoding** específico, basta sobrecarregar o construtor:

```
InputStream inputStream = new FileInputStream( name: "arquivoComAcentos.txt");
Reader inputStreamReader = new InputStreamReader(inputStream, StandardCharsets.UTF_8);
BufferedReader bufferedReader = new BufferedReader(inputStreamReader);
```

Sobrecarregando o leitor com o **encoding UTF-8**

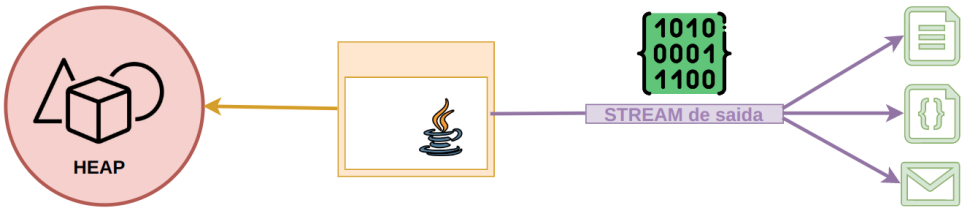
```
Writer writer = new FileWriter( fileName: "arquivoEscritoComAcentos.txt", StandardCharsets.UTF_8);
BufferedWriter bufferedWriter = new BufferedWriter(writer);

bufferedWriter.write( str: "Olá, vamos usar acentos, exclamações e interrogações!!");
bufferedWriter.newLine();
bufferedWriter.write( str: "Saída? entrada? sei lá, só quero usar caracteres especiais");
bufferedWriter.newLine();
bufferedWriter.write( str: "Aclimação? intrigante você!");
```

Sobrecarregando o escritor com o **encoding UTF-8**

Serialização – Java

A ***serialização é você transformar um objeto que está na memória da aplicação em um fluxo de bits e bytes***, geralmente transformados em **arquivos que representam o objeto em determinado estado**.



Os objetos utilizados no ***input e output do stream de objetos específicos*** são o **ObjectInputStream** e **ObjectOutputStream:**

```java
public void serializarCliente(){
    try(var outputStream = new ObjectOutputStream(new FileOutputStream( name: "Cliente-" + this.id + ".txt"))){

        outputStream.writeObject(this);

    }catch (Exception e){

        System.out.println("Nao foi possivel desserializar");
        e.printStackTrace();
    }
}
```

Objeto de outputStream com try with resources

Serializando o objeto

```java
public static Cliente desserializarCliente(int id){
    try(var objectInputStream = new ObjectInputStream(new FileInputStream( name: "Cliente-" + id + ".txt"))){

        return (Cliente) objectInputStream.readObject();

    }catch (Exception e){

        System.out.println("Nao foi possivel desserializar");
        e.printStackTrace();
        return null;
    }
}
```

Objeto de inputStream com try with resources

Desserializando o objeto

Objetos complexos que tem a intenção de serem serializados precisam explicitar isso **através da Interface Serializable.** Se você quiser que **objetos que sejam composições** um do outro sejam serializados, você precisa que **eles também implementem a Interface de serialização**:

```java
public class Pedido implements Serializable {

    private final int id;
    private final ArrayList<ItemPedido> itens;
    private final Cliente cliente;
```

ItemPedido compõe o objeto de Pedido, mas se eu quiser **serializar um Pedido junto com seus itens eu preciso Explicitar isso**. Já que o comportamento de composição não torna esse comportamento padrão

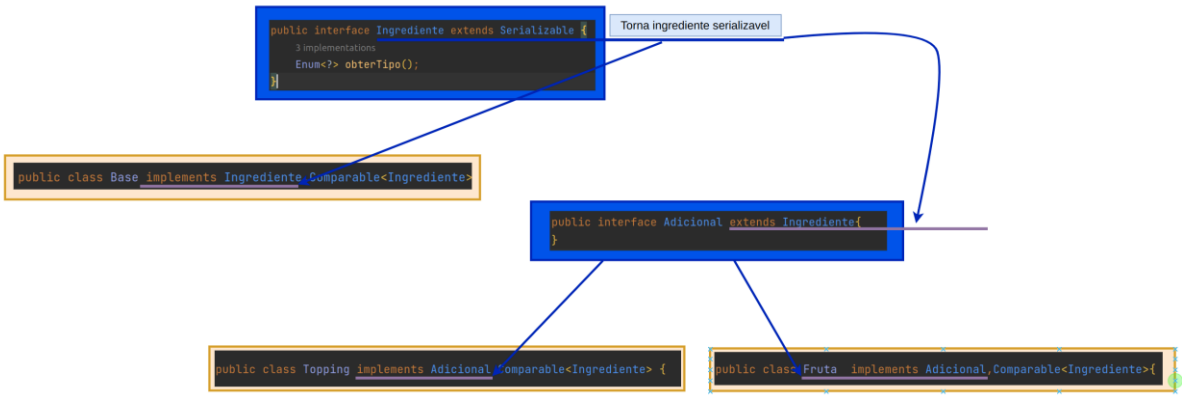
```java
public class ItemPedido implements Serializable {

    private final Shake shake;
    private int quantidade;
```

A interface de marcação diz que a classe pode ser serializada

A **Herança** por meio de **interfaces também pode ser aproveitada para facilitar o processo de serialização**, eu tenho **objetos concretos que representam Ingredientes**, basta eu fazer a **Interface ingrediente extender de Serializable** que **eles** se tornam serializáveis por ***implementar a interface***

Ingrediente:



O ***serialVersionUID*** é um atributo para **versionamento da classe de serialização**. Esse atributo quando não especificado é **gerado automaticamente pela JVM**, o que pode acontecer é serializar **uma classe com o serialversion randômico e ter dificuldades na desserialização dele caso algum atributo mude.**

Então o serialVersion serve como um id **que específica a versão da classe**, a cada mudança você pode ir gerenciando a versão do serial, incrementando-o por exemplo, **assim garantindo uma maior segurança na serialização e desserialização dos objetos:**