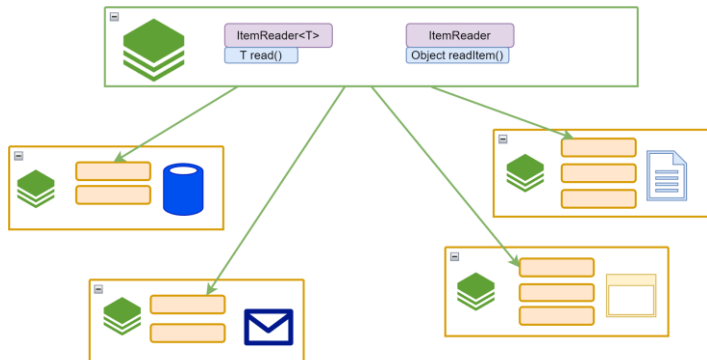


Leitores do Spring Batch

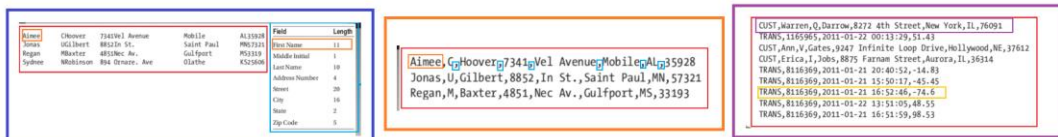
Todos os **leitores** do **Spring Batch** herdam da **Interface `ItemReader<T>`** (que pode ou não ser parametrizada), essa interface possui as assinaturas de método necessárias para as **implementações concretas de leitores**:



Arquivos flat

Um **arquivo flat** é um arquivo que possui **dados não estruturados**. E “**dados não estruturados**” simplesmente quer dizer que quando olhamos o conteúdo não dá pra saber o formato ou significado do que está representado **ali**.

Arquivos baseados em **largura fixa** (cada propriedade é representada por um determinado número de **colunas/caracteres**), **delimitadores** (cada propriedade é separada por algum delimitador), e de **múltiplos formatos** (contém mais de um objeto de negócio) são considerados **arquivos flat**:



The image shows three examples of flat file formats:

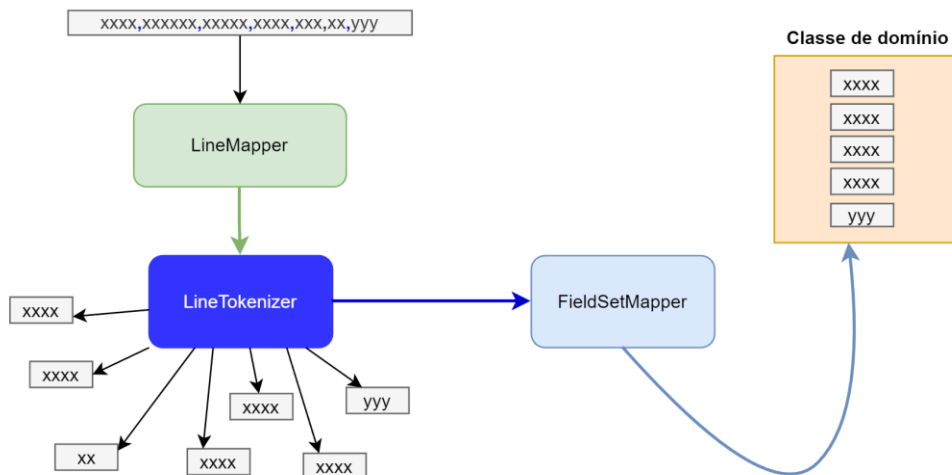
- Fixed-width format:** A table with columns for Name, Address, City, State, Zip Code, and a list of fields with their lengths.
- Delimited format:** A single line of data where fields are separated by commas.
- Multiple formats:** A single line of data where different parts of the line represent different objects, separated by a delimiter.

Lendo um arquivo flat de múltiplos formatos

Já que o **Spring Batch** tem uma implementação para leituras de arquivo flat, vamos usa-la (**`FlatFileItemReader<T>`**) para ler um arquivo de **múltiplos formatos** (cada linha pode ser um tipo diferente de domínio).

Aqui é preciso se aprofundar um pouco mais na maneira que o **`FlatFileItemReader<T>`** do **Spring Batch** lê os arquivos. Ele possui um objeto de **`LineMapper`** que tem dois atributos principais, sendo eles o **`LineTokenizer`** e o **`FieldSetMapper`**.

O **`LineMapper`** é o cara que **mapeia a linha e dá significado aos dados**, ele faz isso através do **`LineTokenizer`** que é o cara responsável por ler a linha e dividi-la em palavras (tokens da linha), esses tokens são convertidos em propriedades do objeto de domínio com ajuda do **`FieldSetMapper`**:



Saber disso é importante pois quando lemos um arquivo com **múltiplos formatos** o mapeamento tende a mudar para cada tipo de linha, ou seja, quando uma linha for um **objeto Y** eu vou usar um mapeador **Y**, quando **X** então mapeador **X** e etc... Fazendo com que seja necessário definir manualmente o comportamento do **LineMapper**.

Na **construção do Reader**, invés de informar o tipo do arquivo, eu vou **informar uma implementação customizada de LineMapper**:

```

@StepScope
@Bean
public FlatFileItemReader arquivoMultiplosFormatosItemReader(
    @Value("#{jobParameters['arquivoClientes']}") Resource arquivoClientes, LineMapper lineMapper) {
    return new FlatFileItemReaderBuilder()
        .name("arquivoMultiplosFormatosItemReader")
        .resource(arquivoClientes)
        .lineMapper(lineMapper)
        .build();
}

```

Aqui tem um ponto

bem interessante, estamos usando os **leitores/mappers que são parametrizados de forma cru**, ou seja, sem um tipo específico, nem mesmo um genérico, isso é porque “não sabemos” quais objetos estamos lendo.]

Na **implementação do LineMapper** usamos o objeto **PatternMatchingCompositeLineMapping**, basicamente falando, esse objeto é **capaz de identificar qual padrão de mapeamento usar dependendo do padrão da linha**, por exemplo, nesse **arquivo** os registros de **cliente** começam com 0 e os registros de **transação** com 1:

```

@Bean
public PatternMatchingCompositeLineMapper lineMapper(){
    PatternMatchingCompositeLineMapper lineMapper = new PatternMatchingCompositeLineMapper();
    lineMapper.setTokenizers(tokenizers());
    lineMapper.setFieldSetMappers(fieldSetMappers());
    return lineMapper;
}

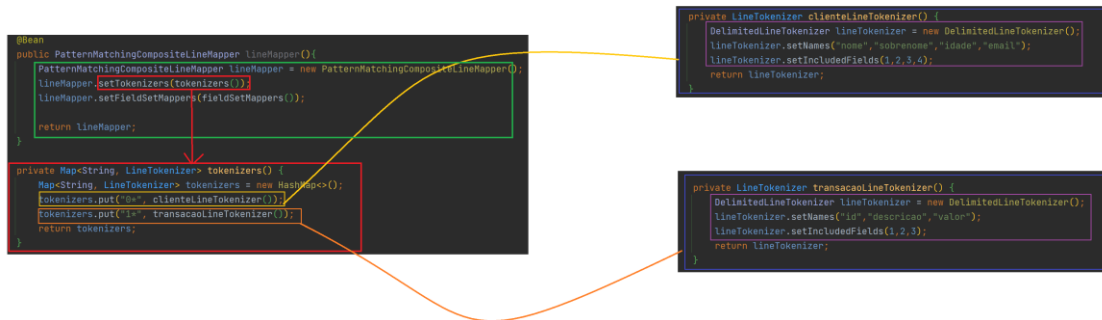
```

```

0,João,Silva,32,joao@test.com
1,t1c1,Cadeado,50.80
1,t2c1,TV,1500
0,Maria,Silva,30,maria@test.com
1,t1c2,Getadeira duplex,2000
1,t2c2,Mesa,4500
1,t3c2,Sofá,1099,99
0,José,Silva,20,jose@test.com
1,t1c3,Comida no Ifood,500.45

```

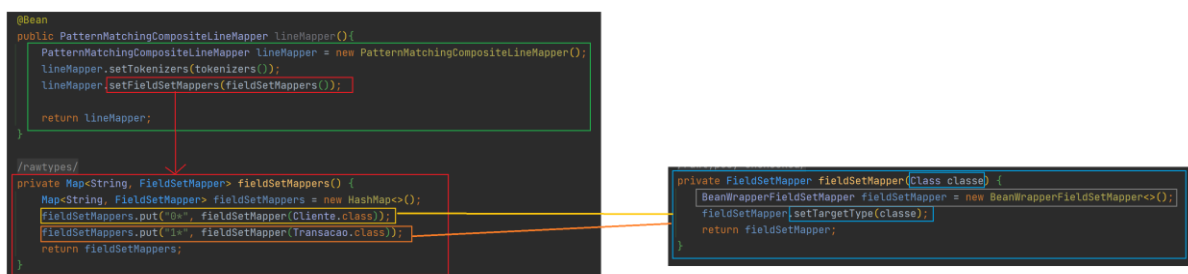
Esse **objeto** é capaz de alterar o mapeamento dependendo do padrão, mas isso significa que nós **ainda temos** que definir como os **Tokenizers** e **FieldMappers** dele vão se comportar. Nos **Tokenizers** eu defino um **padrão de comportamento para a variação da tokenização**, ou seja, quando a **String** começar com **“0”** usa o **tokenizer do cliente**, quando **começar em “1”** usa o **tokenizer de transação**, a partir disso eu consigo **transformar minha linha em tokens** (aqui eu digo o nome dos tokens e seu índice, pulando o 0 pois ele é o índice que contém a identificação da entidade e não um valor de fato):



Outro

ponto interessante aqui é que os tokenizers são do tipo delimitados, mas se eu estivesse usando um arquivo fixo bastaria mudar a implementação dos tokenizers para o tipo fixo.

O mesmo tipo de **sobrescrita de comportamento** precisa ser feito para o **FieldMappers** do **LineMapper**, a ideia é a mesma, se a **String começar com "0"** eu **defino o meu alvo de mapeamento como sendo a classe Cliente**, se **começar com "1"** então **minha classe alvo de mapeamento vai ser a Transacao**. O mapeamento é feito pelo objeto **BeanWrapperFieldSetMapper**:



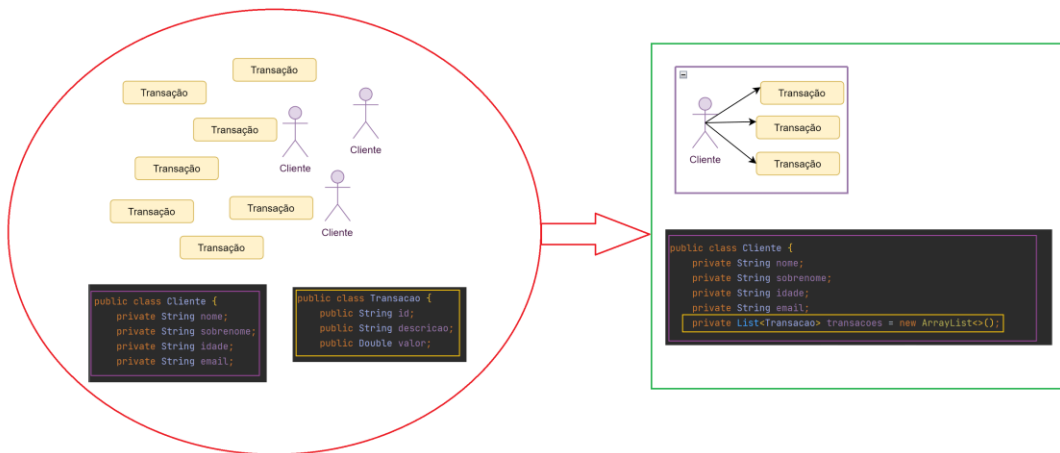
Se eu **rodar o batch** com esse comportamento de leitura, eu vou obter o resultado onde transações e clientes foram mapeados corretamente em seus objetos de domínio:

```
Cliente{nome='João', sobrenome='Silva', idade='32', email='joao@test.com'}
Transacao{id='t1c1', descricao='Cadeado', valor='50.8'}
Transacao{id='t2c1', descricao='TV', valor='1500.0'}
Cliente{nome='Maria', sobrenome='Silva', idade='30', email='maria@test.com'}
Transacao{id='t1c2', descricao='Geladeira duplex', valor='2000.0'}
Transacao{id='t2c2', descricao='Mesa', valor='4500.0'}
Transacao{id='t3c2', descricao='Sofá', valor='1099.0'}
Cliente{nome='José', sobrenome='Silva', idade='20', email='jose@test.com'}
Transacao{id='t1c3', descricao='Comida no Ifood', valor='500.45'}
```



Melhorando o resultado da leitura aos olhos do negócio

É comum que os **objetos trabalhados na leitura se relacionem de alguma forma**, por exemplo, as **Transações** são de um **Cliente**, portanto faz sentido que **elas estejam relacionadas ao cliente**. Então deixamos de tratar objetos isolados e passamos a tratar objetos relacionados de fato, entregando uma visão mais voltada ao negócio:



O papel de fazer esse relacionamento vai ficar por conta ainda do **processo de leitura**, mas com uma ajudinha. Essa ajudinha vai vir do **ItemStreamReader<T>** (uau, parece um reader normal com o acréscimo da palavra *Stream*), essa interface combina o **ItemStream** e o **ItemReader<T>**:

```

public interface ItemStreamReader<T> extends ItemStream, ItemReader<T> {
}

```

Aqui nesse ponto é importante saber também que um “**stream**” significa “**fluxo**”, então geralmente no Java quando a palavra **stream** está associada estamos querendo falar de um **fluxo de alguma coisa**, **ItemStream** é um **fluxo de itens** e o leitor é apenas um leitor. Então quando usamos a **ItemStreamReader<T>**, estamos falando que **vamos ler um fluxo de itens**, outro ponto importante é que o **Stream** é um **fluxo contínuo de dados** e serve pra dizer aqui nesse contexto que esse processo pode ser reiniciado, mantendo o estado que estava no contexto.

Mas **não é só sobre o fluxo**, é sobre operações nesse **fluxo**. A Stream do java, na verdade é uma **API para facilitar a iteração e operação em coleções** (mais do que isso até) de todos os tipos, **criando um novo fluxo** de dados sem alterar a fonte original da stream. Então o que vamos fazer de fato, é **alterar o fluxo de entrada de itens** criando um novo fluxo que vai adicionar as transações aos clientes.

Então o **Stream de leitura** vai funcionar da seguinte maneira: **Leia a linha atual, transforma ela em um modelo de domínio do tipo cliente** e **depois limpe a linhaAtual**, enquanto o cliente for diferente de nulo (ou seja, existir) **leia as próximas linhas e enquanto forem do tipo Transação, atribua ao cliente**. quando acabarem as transações do cliente, **retorne-o** e repita para a próxima linha:

```

public class RelacionalClienteTransacaoReader implements ItemStreamReader<Cliente> {
    private Object currentLine;
    private final ItemStreamReader<Object> delegateReader;

    @Override
    public Cliente read() throws Exception {
        if(currentLine == null){
            currentLine = delegateReader.read();
        }

        Cliente cliente = (Cliente) currentLine;
        currentLine = null;

        if(cliente != null){
            while(peek() instanceof Transacao){
                cliente.getTransacoes().add((Transacao) currentLine);
            }

            return cliente;
        }

        private Object peek() throws Exception {
            currentLine = delegateReader.read();
            return currentLine;
        }
    }
}

```

```

0,João,Silva,32,joao@test.com
1,t1c1,Casado,50.80
1,t2c1,TV,1500
0,Maria,Silva,30,maria@test.com
1,t1c2,Geladeira duplex,2000
1,t2c2,Mesa,4500
1,t3c2,Sofá,1099,99

```

Agora **nosso Stream de leitura** está fazendo as alterações que queríamos, atribuindo as **transações ao Cliente**, formando uma unidade mais coesa de negócio. Mas temos outros métodos que essa **Interface** precisa implementar.

Para isso podemos usar o **padrão Delegate**. Basicamente falando, quando assinamos a interface do **ItemStreamReader<T>** tínhamos a intenção de **mudar o comportamento de leitura**, mas uma interface não funciona assim, temos que implementar **TODAS AS ASSINATURAS**.

Mas nós já temos um objeto de leitura, esse cara é o **FlatFileItemReader<T>**, ele inclusive já está configurado com o **LineMapper** específico que **vai fazer o trabalho de identificar qual linha representa qual recurso na aplicação**, então podemos simplesmente deixar com que o **FlatFileItemReader<T>** faça a **implementação dos métodos que não precisamos mudar**:



Antigamente o **batch** tinha esse resultado:

```

Cliente{nome='João', sobrenome='Silva', idade='32', email='joao@test.com'}
Transacao{id='t1c1', descricao='Cadeado', valor='50.8'}
Transacao{id='t2c1', descricao='TV', valor='1500.0'}
Cliente{nome='Maria', sobrenome='Silva', idade='30', email='maria@test.com'}
Transacao{id='t1c2', descricao='Geladeira duplex', valor='2000.0'}
Transacao{id='t2c2', descricao='Mesa', valor='4500.0'}
Transacao{id='t3c2', descricao='Sofá', valor='1099.0'}
Cliente{nome='José', sobrenome='Silva', idade='20', email='jose@test.com'}
Transacao{id='t1c3', descricao='Comida no Ifood', valor='500.45'}

```

invés de eu ter 9 processos (um pra cada linha do arquivo) no meu job, eu vou ter apenas 3:

```

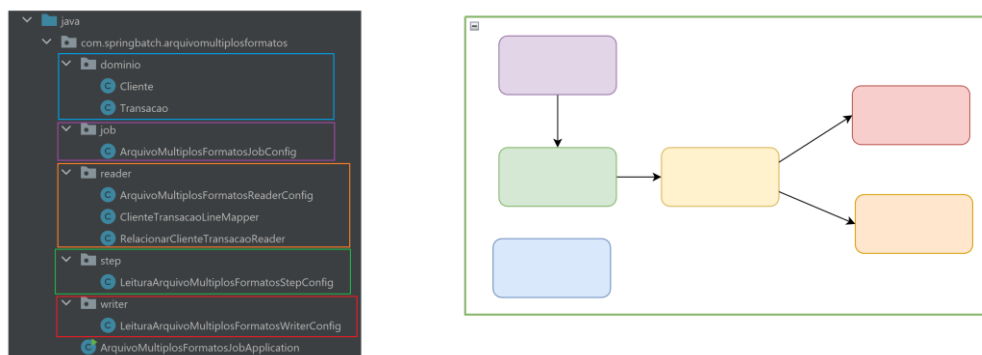
2022-07-20 19:56:20.662 INFO 20948 --- [main] o.s.batch.core.job.SimpleStepHandler : Executing step: [leituraArquivoMultiplasFormatosStep]
Cliente{nome='João', sobrenome='Silva', idade='32', email='joao@test.com'} Transacao{id='t1c1', descricao='Cadeado', valor='50.8'} Transacao{id='t2c1', descricao='TV', valor='1500.0'}
Cliente{nome='Maria', sobrenome='Silva', idade='30', email='maria@test.com'} Transacao{id='t1c2', descricao='Geladeira duplex', valor='2000.0'} Transacao{id='t2c2', descricao='Mesa', valor='4500.0'}
Cliente{nome='José', sobrenome='Silva', idade='20', email='jose@test.com'} Transacao{id='t1c3', descricao='Comida no Ifood', valor='500.45'}
2022-07-20 19:56:20.594 INFO 20948 --- [main] o.s.batch.core.step.AbstractStep : Step: [leituraArquivoMultiplasFormatosStep] executed in 152ms

```

Isso porque porque o leitor só conhece os clientes, então os metadados se mantêm consistentes ignorando as Transações.

Como a estrutura do Batch ficou

De forma geral, temos o **pacote para as classes de domínio** e os pacotes do batch. Nada fora do padrão, o **batch** é composto de um **Job**, que possui um **Step** que é um **chunk**, o **chunk** é composto por **leitura**, **processamento(opcional)** e **escrita**:



Todas as alterações foram feitas na etapa de leitura do arquivo.



Melhorando o resultado da leitura aos olhos do negócio

O tamanho do chunk influencia em como o job vai ser reexecutado em caso de falha, isso porque o chunk é uma unidade de contagem do job. Se eu tenho **100 registros** divididos em **10 chunks** de tamanho 10 (**10 registros são commitados por chunk**), então eu vou ter uma **contagem de 0 a 9** para esses chunks.

O que representa a **quantidade de chunks executados no banco de dados** é o atributo **SHORT_CONTEXT** da tabela **BATCH_STEP_EXECUTION_CONTEXT**, nele podemos ver o **count** que representa qual o próximo chunk daquele step a ser executado:

DN_ID	SHORT_CONTEXT
	{ "batch.taskletType": "org.springframework.batch.core.step.item.ChunkOrientedTasklet", "clienteReader.read.count": 0, "batch.stepType": "org.springframework..." }
	{ "batch.taskletType": "org.springframework.batch.core.step.item.ChunkOrientedTasklet", "clienteReader.read.count": 1, "batch.stepType": "org.springframework..." }
	{ "batch.taskletType": "org.springframework.batch.core.step.item.ChunkOrientedTasklet", "clienteReader.read.count": 2, "batch.stepType": "org.springframework..." }

nesse **exemplo aqui** eu li um arquivo com 3 registros onde cada **chunk tinha tamanho igual a 1**, depois de executar os **3 chunks com sucesso**, o próximo **chunk é o 4**:

```
{ "batch.taskletType": "org.springframework.batch.core.step.item.ChunkOrientedTasklet", "clienteReader.read.count": 4, "batch.stepType": "org.springframework..." }
```

Aqui **nenhum chunk foi feito com sucesso**, o que indica que o **próximo chunk desse step é o 0**:

```
{ "batch.taskletType": "org.springframework.batch.core.step.item.ChunkOrientedTasklet", "clienteReader.read.count": 0, "batch.stepType": "org.springframework..." }
```