**APÊNDICE 1. CONCEITOS DE JAVA E SUA API**

Alguns conceitos da linguagem Java e de sua API[[1]](#footnote-1) são essências para a compreensão do código-fonte do projeto. Abaixo estarão listados alguns destes conceitos que podem incorrer em dificuldades ao leitor quando este for analisar o código do Simulador.

* **Estrutura de dados com Generics:** Analise a classe abstrata *Conjunto* (*afd.gramatica.Conjunto*). Esta classe define as operações básicas sobre um conjunto qualquer de elementos e funciona com generics (veja o <E> na sua declaração):

*public abstract class Conjunto* ***<E>*** *...*

Aonde aparece o <E>, a classe espera que suas subclasses ou instâncias substituam por um tipo de classe específico (por exemplo, <String>).

Duas classes herdam de *Conjunto*: *Alfabeto* (*afd.gramatica.Alfabeto*) e *Estados* (*afd.gramatica.Estados*). *Alfabeto* define que o <E> será de objetos do tipo *Simbolo* (*afd.gramatica.Simbolo*), ou seja, *Alfabeto* será um conjunto de símbolos. Sua declaração é a seguinte:

*public class Alfabeto extends Conjunto****<Simbolo>*** *...*

*Estados* define que <E> será objetos do tipo *Estado* (*afd.gramatica.Estado*). Sua declaração é a seguinte:

*public class Estados extends Conjunto****<Estado>*** *...*

Ao executar um método *get(int idx)* de *Alfabeto* o objeto retornado será explicitamente do tipo *Simbolo*, não havendo a necessidade de um *cast[[2]](#footnote-2)*. Ao executar o mesmo método de *Estados*, o objeto retornado será um *Estado*, também sem a necessidade de cast. Ambas as classes não sobrescrevem o método get;

* **Interface Iterable**: O Autômato Finito depende extensivamente de conjuntos de elementos (alfabeto, estados, histórico do processamento). Para fazer a pesquisa sequencial (para frente) por estes conjuntos, todos implementam a interface *Iterable* (*java.lang.Iterable*). Para ver um exemplo, veja a declaração da classe *Conjunto*:

*public abstract class Conjunto <E> implements* ***Iterable<E>*** *...*

*Iterable* define um único método: *Iterator<T> iterator() -* que é responsável por devolver um objeto da classe *Iterator* que permitirá a pesquisa sequencial pelo conjunto.

Veja abaixo a implementação do métodona classe *Conjunto*:

*@Override*

*public Iterator<E> iterator() {*

*return conjunto.iterator();*

*}*

Note que a classe retorna o *Iterator* de seu objeto *ArrayList* **conjunto**.

Vamos ver um exemplo de utilização de *Iterable* na instrução abaixo com a subclasse de *Conjunto*, *Alfabeto*:

*Alfabeto alfabeto = new Alfabeto();*

*for (Simbolo s: alfabeto) System.out.println(s.getNome());*

* **Interface Comparable**: Para a ordenação de listas de elementos ou qualquer outro processo que necessite da comparação entre elementos para verificar se ambos são iguais ou se um é maior ou menor que outro, é preciso estabelecer os critérios para esta comparação.

Por exemplo, qual o critério para definir como um objeto *Pessoa* é maior, igual, ou menor que outro objeto do mesmo tipo? Talvez se use como critério os nomes destas pessoas, ou suas idades, ou ambos.

A interface *Comparable* (*java.lang.Comparable*) permite que se estabeleça estes critérios. Vamos analisar a classe *Simbolo* (*afd.gramatica.Simbolo*) que implementa esta interface.

Vejamos a declaração da classe:

*public class Simbolo implements* ***Comparable<Simbolo>****...*

A interface *Comparable* define um único método: *public int compareTo(T o)* - que deve retornar -1 caso o objeto passado seja menor que o objeto, 0, caso ambos sejam iguais, 1, caso o objeto passado seja maior que o objeto.

A implementação deste método na classe *Simbolo* é a seguinte:

*@Override*

*public int compareTo(Simbolo simbolo) {*

*return caractere.compareTo(simbolo.caractere);*

*}*

A comparação é feita entre os caracteres de ambos os símbolos.

* **Recursividade**: Um método fundamental do Autômato é recursivo. É o método *private Estado lerFita(Estado origem, UnidadeLeitura uLeitura)*, da classe Interna *UnidadeControle* que é declarada como private dentro da classe AutomatoFinito (*afd.AutomatoFinito*).

Esta classe, juntamente com *UnidadeLeitura* e *FitaEntrada*, forma um subsistema responsável pelo processamento da *Fita de Entrada*.

Veja a implementação deste método e cada detalhe envolvido no processamento de uma Palavra de Entrada *w*:

*private Estado lerFita(Estado origem, UnidadeLeitura uLeitura) {*

*Simbolo simbolo = uLeitura.getSimbolo();*

*Estado destino = null;*

*try {*

*destino = fTransicao.get(origem.getNome(), simbolo.get());*

*gerarHistorico(origem, simbolo, destino, uLeitura.parcelaNaoLidaFita());*

*if (uLeitura.existeProximo()) {*

*return lerFita(destino, uLeitura);*

*} else {*

*return destino;*

*}*

*} catch (TransicaoNaoDefinidaException ex) {*

*gerarHistorico(origem, simbolo, null, uLeitura.parcelaNaoLidaFita());*

*return null;*

*}*

*}*

A primeira instrução:

*Simbolo simbolo = uLeitura.getSimbolo();*

Faz com que a Unidade de Leitura retorne o símbolo na posição corrente da Fita e posicione a cabeça de leitura para ler o próximo símbolo da fita.

Na instrução:

*destino = fTransicao.get(origem.getNome(), simbolo.get());*

o método obtém o estado que o Autômato passa com a leitura do símbolo obtido da Fita de Entrada e estando no estado atual (variável “origem” na declaração do método).

A recursividade está no seguinte trecho de código:

*if (uLeitura.existeProximo()) {*

*return lerFita(destino, uLeitura);*

*} else {*

*return destino;*

*}*

Nele, verifica se ainda há algum símbolo não lido na fita. Se houver, o método chama a si mesmo, recursivamente, passando como estado atual (parâmetro “origem”) o estado obtido na instrução anterior, e a referência para a Unidade de Leitura que fará a leitura do próximo símbolo da Fita de Entrada. A instrução é a seguinte:

*return lerFita(destino, uLeitura);*

Quando não puder mover mais pela fita (chegou à última célula), o método retorna o estado que ele passou com a leitura do último símbolo para a instância que o chamou, que passa para instância que a chamou e assim, sucessivamente, num retorno “em cascata”.

* **Exceções Personalizadas**: Para tratar as situações de erro em que se quer obter uma transição quando esta não foi definida (pela leitura de um símbolo que não pertence ao alfabeto do Autômato), ou quando se quer processar uma Palavra de Entrada quando a função de transição ainda não está com todos os seus pares **Estado/Símbolo🡪Estado** definidos, foram criadas duas exceções personalizadas: *TransicaoNaoDefinidaException* e *FuncaoNaoDefinidaException*. Elas são utilizadas para tratar, respectivamente, cada uma destas condições de erro. Ambas herdam de *Exception* (*java.lang.Exception*).

Para lançar uma exceção faz-se:

*throw new TransicaoNaoDefinidaException();*

e para tratá-la, faz-se:

*try {*

*//Código que pode lançar a exceção.*

*} catch (TransicaoNaoDefinidaException ex) {*

*//Código para o tratamento, caso ela ocorra.*

*}*

* **Componentes de edição e exibição de uma JTable personalizados**: Um detalhe bastante interessante no projeto é a *Tabela Função de Transição (Figura 3.14)*. Suas células têm um componente de edição personalizado, que é uma lista contendo todos os estados do Autômato *(Figura 3.15)*. Além disso, outro detalhe incomum é a formatação da fonte dos componentes de exibição destas mesmas células (azul e negrito).

Isso foi conseguido, substituindo os componentes de edição e exibição, respectivamente, do componente *JTable* (*javax.swing.JTable*), que forma esta tabela.

Todo o código que faz essas alterações está no método *private void gerarTabelaFuncaoTransicao()*, da tela principal do programa (*afd.swing.TelaPrincipal*). É um método bastante complexo, por isso vamos entrar apenas nos detalhes que fazem com que essas alterações sejam possíveis.

Primeiramente, dentro deste método são declaradas três classes:

*class JTextFieldCelulasFixas extends JTextField {*

*public JTextFieldCelulasFixas(String text) {*

*super(text);*

*setBorder(*

*javax.swing.BorderFactory.*

*createBevelBorder(javax.swing.border.BevelBorder.RAISED)*

*);*

*setBackground(new Color(249, 250, 253));*

*setFont(new java.awt.Font("Tahoma", 1, 12));*

*setHorizontalAlignment(SwingConstants.CENTER);*

*}*

*}*

*class JTextFieldCelulasEditaveis extends JTextField {*

*public JTextFieldCelulasEditaveis(Object text) {*

*super(text != null ? (String) text : null);*

*setBorder(null);*

*setForeground(Color.BLUE);*

*setFont(new java.awt.Font("Tahoma", 1, 12));*

*setHorizontalAlignment(SwingConstants.CENTER);*

*setFocusable(true);*

*}*

*}*

class JComboboxEstados extends JComboBox<String> {

public JComboboxEstados() {

setModel(new javax.swing.DefaultComboBoxModel(toStringArray(estados)));

}

}

a

1. Conjunto de classes que fazem parte da distribuição padrão do Java. [↑](#footnote-ref-1)
2. Conversão explícita do tipo: Simbolo s = **(Simbolo)** alfabeto.get(0); [↑](#footnote-ref-2)