Linguagem Nice

R. M. Teles *C. L. de Carvalho*

Technical Report - RT-INF_003-11 - Relatório Técnico
May - 2011 - Maio

Instituto de Informática
Universidade Federal de Goiás
www.inf.ufg.br

Linguagem Nice

Ronneesley Moura Teles [*] Cedric L. de Carvalho [†]
ronneesley@gmail.com.br cedric@inf.ufg.br

Abstract. *With the emergence of easy learning languages, the Java language has become very laborious for the development of new software. However, the legacy of Java softwares is immense and can not be neglected, rewriting it entirely in another language. Thus, this work describes the characteristics of Nice language that is a modern language for Java platform that enables the use of legacy software written in Java language in new software and vice versa.*

Keywords: Nice language, Java language, Legacy software

Resumo. *Com o surgimento das linguagens de fácil aprendizado, a linguagem Java tem se tornado uma alternativa muito trabalhosa para o desenvolvimento de novos softwares. Entretanto, o legado de software escrito na linguagem Java é imenso e não pode ser desprezado, reescrevendo-o totalmente em outra linguagem. Este trabalho descreve as características da linguagem Nice que é uma linguagem moderna para a plataforma Java que permite utilizar o legado de software escrito na linguagem Java nos novos softwares e vice-versa.*

Palavras-Chave: Linguagem Nice, Linguagem Java, Legado de software

1 Introdução

A linguagem Java é muito conhecida no mercado e atualmente possui uma base sólida de ferramentas que auxiliam cada vez mais pessoas a aprenderem utilizá-la. Um dos fatores chave do sucesso da plataforma Java é a independência de sistema operacional e de dispositivo.

Mesmo com facilidade de saber apenas uma linguagem, normalmente é necessário uma especialização nos arcabouços. Aprender a trabalhar com arcabouços diferentes em geral é mais fácil do que aprender uma linguagem diferente; assim, a vantagem de desenvolver softwares com a linguagem Java é grande.

Com o passar dos anos novas linguagens surgiram para diminuir o tempo de aprendizado e o trabalho para manter os softwares. Dentre estas linguagens, as mais conhecidas no mercado são: PHP (Hypertext Preprocessor) [6], Python [7] e Ruby [5].

Uma das fontes mais polêmicas sobre comparação do uso de linguagens é o artigo "TIOBE Programming Community Index for March 2009" [13], neste trabalho não será discutida a forma pela qual esta comparação foi realizada, mas o resultado da comparação com

[*] Mestrando em Ciência da Computação – INF/UFG.
[†] Orientador – INF/UFG.

alguns cenários comuns das equipes de desenvolvimento. Abaixo segue a estatística de uso da linguagem Java no período de 2002 à 2009, publicada no artigo.

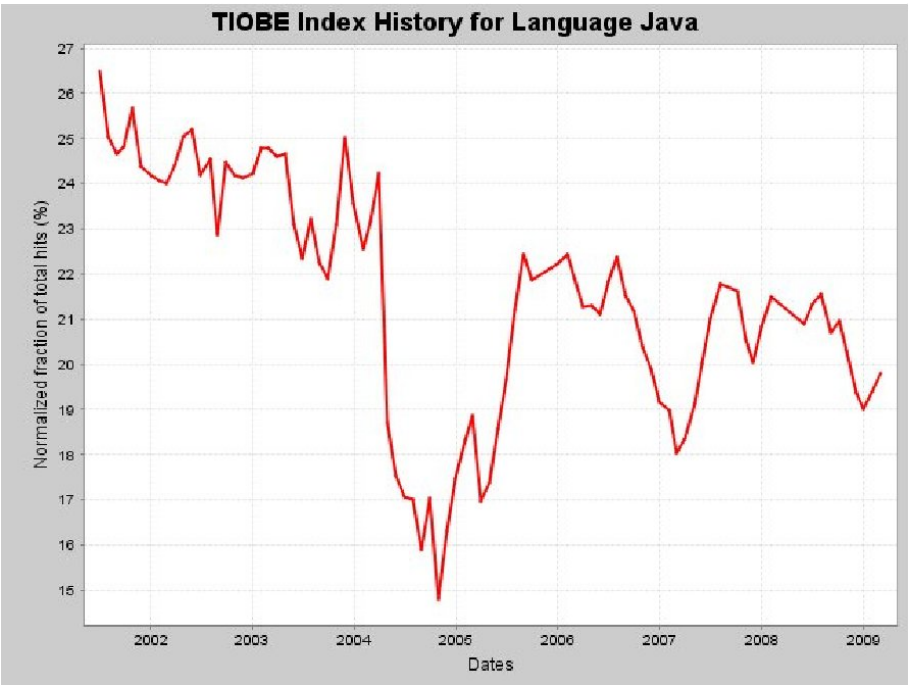

Figura 1: Estatística de uso da linguagem Java no período de 2002 à 2009

Outra informação contida no artigo é a tabela das taxas de crescimento de uso das linguagens no período de 2004 à 2009. A Tabela 1, apresenta um fragmento da tabela original.

Tabela 1: Taxa de crescimento do uso das linguagens no período de 2004 à 2009

Linguagem	Taxa
Python	+4.17%
Ruby	+2.44%
PHP	+1.85%

Ao visualizar a Figura 1 pode-se concluir que a linguagem Java teve queda no uso durante o ano de 2008 e a partir da Tabela 1 pode-se concluir que as linguagens Python, Ruby e PHP obtiveram um crescimento do uso no período de 2004 à 2009. Por mais que a fonte e o meio dos cálculos sejam questionáveis, não é difícil de imaginar este cenário, pois muitas equipes iniciam projetos em linguagens cuja o tempo de aprendizado é pequeno e que não seja necessário aprender muitos conceitos para escrever um programa.

Iniciantes, normalmente, começam a aprender uma linguagem, criando um programa que escreve uma mensagem no dispositivo de saída, normalmente o monitor. O Algoritmo 1, é utilizado para escrever uma mensagem.

Algoritmo 1: – Algoritmo para escrever uma mensagem no dispositivo de saída

```
escreva "Olá mundo"
```

Um iniciante que deseja implementar o Algoritmo 1 em uma das linguagens Java, Ruby, Python e PHP deverá criar códigos fontes assim como os exemplos a seguir. Exemplos em outras linguagens podem ser encontrados em [20].

Programa 1 – Implementação do Algoritmo 1 na linguagem Java

```
1 public class Java {
2         public static void main(String[] args){
3                 System.out.println("Olá mundo!");
4         }
5 }
```

Programa 2 – Implementação do Algoritmo 1 na linguagem Ruby

```
1 puts "Olá mundo!"
```

Programa 3 – Implementação do Algoritmo 1 na linguagem Python

```
1 print "Olá mundo!"
```

Programa 4 – Implementação do Algoritmo 1 na linguagem PHP

```
1 <?php
2 echo "Olá mundo!";
3 ?>
```

Os programas 1, 2, 3 e 4 demonstram a grande verbosidade e conceitos necessários para escrever o Algoritmo 1 na linguagem Java comparado com a verbosidade e conceitos das linguagens Ruby, Python e PHP, pois é necessário escrever 5 (cinco) linhas totalizando 102 caracteres, enquanto que para as outras linguagens a média é 2 (duas) linhas totalizando 22 caracteres. Entretanto, o problema não é apenas este, para escrever o algoritmo nas outras linguagens é necessário aprender 1 (um) comando[1], respectivamente *puts*, *print* e *echo*, ao passo que em Java é necessário o aprendizado de no mínimo 10 (dez) comandos, por exemplo *public*, *class*, *static*, etc.

Com base nestes dados, aprender a linguagem Java está em desvantagem, pois sua sintaxe é mais complexa do que muitas linguagens conhecidas. Assim, os iniciantes criam resistência para aprender a linguagem Java, pois existem linguagens mais fáceis de aprender e manter os programas.

Antes mesmo do surgimento da linguagem Java, estabeleceu-se a preocupação com a criação de linguagens de fácil aprendizado, assim como foi definido na criação da linguagem Haskell [15]. Esta linguagem possui várias características que possibilitam a construção de funções complexas de forma quase trivial, porém sua sintaxe é diferente das linguagens conhecidas no mercado.

A linguagem Nice foi projetada visando proporcionar as características de segurança encontrada em linguagens, tais como: Haskell e ML [17], porém com uma sintaxe mais parecida com a linguagem Java, assim aproveita-se o conhecimento dos programadores Java. Além disto, a linguagem Nice disponibiliza várias características que facilitam a escrita de programas, são elas:

1. Capacidade de comunicação com códigos escritos na linguagem Java e vice-versa;

[1]Em PHP as marcações <?php e ?> respectivamente iniciam e terminam blocos de código.

2. Não é necessário nenhuma instalação além do Java Runtime Environment (JRE), para rodar um programa escrito na linguagem Nice;
3. Possui inferência de tipos de dados (subseção 4.2);
4. Possui laços compactos para coleções (subseção 4.7);
5. Possibilita que funções sejam argumentos de outras funções (subseção 4.23);
6. Possibilita a passagem de parâmetros com o uso de rótulos (subseção 4.20);
7. Possibilita adicionar funções em classes existentes, sem a necessidade de alterar o código fonte da classe, esta propriedade é chamada de multi-métodos (do inglês *multi-methods*) (subseção 4.13);
8. Permite que uma função retorne mais de um valor (subseção 4.21);
9. Permite que funções sejam declaradas fora de uma classe (subseção 4.9);
10. Permite a criação de funções anônimas (subseção 4.24);
11. Permite declarar funções dentro de outras funções (subseção 4.19);
12. Não permite que erros do tipo *NullPointerException* e *ClassCastException*, sejam apresentados em seus programas;
13. Compila os diretórios e não somente arquivos (subseção 3.2);
14. Maior expressividade na escrita, de: expressões, funções, conversões, laços e sobrescrita de funções;
15. Implementa nativamente Padrão por Contrato (do inglês *Design by Contract*) (subseção 4.18);
16. Implementa nativamente classes paramétricas (subseção 4.12);
17. Implementa despacho por valor de forma parecida ao Prolog (subseção 4.16); e
18. Implementa um novo recurso chamado "interface abstrata"(subseção 4.30);

Neste documento será utilizado a palavra Nice para referenciar a linguagem Nice e não a cidade francesa [18]. Na Seção 2, será apresentada a linguagem Nice, na Seção 3, será mostrado como configurar o ambiente para o desenvolvimento de programas na linguagem Nice, na Seção 4, serão analisadas as características específicas da linguagem Nice. Neste caso, é necessário conhecer a sintaxe da linguagem Java, pois esta seção tem como foco a análise das diferenças entre estas linguagens, na Seção 5, serão apresentados exemplos de aplicações, na Seção 6, serão feitas sugestões de melhoria para a linguagem Nice e, finalmente, na Seção 7, será feita uma análise geral da linguagem Nice.

2 O que é a linguagem Nice?

O nome Nice é uma homenagem a deusa grega da vitória, Niké [19]. Uma das características da linguagem Nice é a capacidade de comunicação com os programas escritos na linguagem Java e vice-versa.

Desta maneira, todos os programas escritos na linguagem Java, podem ser usados juntamente com programas escritos na linguagem Nice, e os programas escritos na linguagem Nice, podem ser usados juntamente com programas escritos na linguagem Java, com total transparência, ou seja, não é necessário notificar que se trata de um programa escrito na linguagem Nice.

Como se não bastasse ser uma linguagem onde representar uma idéia, utiliza-se poucas linhas de código, comparada a linguagem Java. A linguagem Nice possui novos recursos e outros já conhecidos em algumas linguagens, tais como: a inferência de tipos de dados, que permite não especificar diretamente o tipo de dado de uma variável, aceita funções retornarem mais de um valor, por exemplo, uma função pode retornar um inteiro e um literal, sem a necessidade de criar outra classe para tal finalidade, aceita a criação de funções anônimas, que permitem criar pequenas funções, sem a necessidade da declaração completa do protótipo, e compila diretórios e não somente os arquivos, logo, não é mais necessário aprender a configurar scripts (roteiros) de compilação, para programas pequenos. Estes e outros recursos da linguagem Nice possibilitam que ela tenha uma maior expressividade, modularidade e segurança.

Deve-se destacar que, a linguagem Nice possibilita uma adaptação rápida para programadores em Java, pois em grande parte dos casos, possibilita utilizar a sintaxe da linguagem Java.

Estes recursos serão detalhados na Seção 4. Antes, a Seção 3, irá mostrar como configurar o ambiente de desenvolvimento, para possibilitar os testes destes recursos.

3 Configuração do ambiente

Para desenvolver códigos de maneira fácil é necessário um Ambiente de Desenvolvimento Integrado (do inglês *Integrated Development Environment* ou simplesmente IDE) [14] ou um editor de textos que possua a realce de sintaxe [22] e tenha suporte a operações de compilação e execução de programas.

Atualmente, está em desenvolvimento uma extensão para o Eclipse [1] e existe uma extensão para o editor jEdit [4], fornecido por Bryn Keller [9] que permite o realce de sintaxe.

3.1 Instalação do compilador

O compilador Nice está disponível para os sistemas operacionais: Linux e Windows. Esta subseção se dedica ao procedimento de instalação nestes dois sistemas operacionais.

3.1.1 Instalação do compilador no Linux

No Linux podem ser encontrados três procedimentos para a instalação, dois destes procedimentos se referem às distribuições com arquivos no formato **.deb** e no formato **.rpm** e o terceiro procedimento, refere-se a utilizar o compilador já preparado para o uso, este último funciona para todas as distribuições.

- Procedimento para instalação com o formato **.deb**:

O procedimento padrão utiliza o aplicativo **apt-get**, desta forma é necessário executar o comando: **apt-get install nice**, caso não esteja disponível, o arquivo pode ser obtido no endereço `http://nice.sourceforge.net/nice.deb`, em seguida deve ser executado o comando: **dpkg -i nice_*.deb**.

- Procedimento para instalação com o formato **.rpm**:

  O arquivo pode ser obtido no endereço `http://nice.sourceforge.net/nice.rpm` e deve ser executado o comando: **rpm -i Nice_*.rpm**.

- Utilizar o compilador preparado:

  O compilador pode ser obtido no endereço `http://nice.sourceforge.net/Nice.tar` em seguida deve ser descompactado, com o comando: **tar xvfz Nice*.tar.gz**, dentro da pasta *bin* estará o arquivo binário *nicec* que é o compilador.

3.1.2 Instalação do compilador no Windows

O compilador pode ser obtido no endereço `http://nice.sourceforge.net/Nice.zip`. Após a obtenção, descompacte-o. Em seguida, é necessário criar uma variável de ambiente chamada NICE e adicioná-la à variável de ambiente chamada PATH. Para isto, é necessário acessar o menu Iniciar -> Configurações -> Painel de controle, clicar no ícone "Sistema" e acessar a guia "Avançado". A Figura 2 representa a janela neste momento. Em seguida é necessário clicar no botão "Variáveis de ambiente".

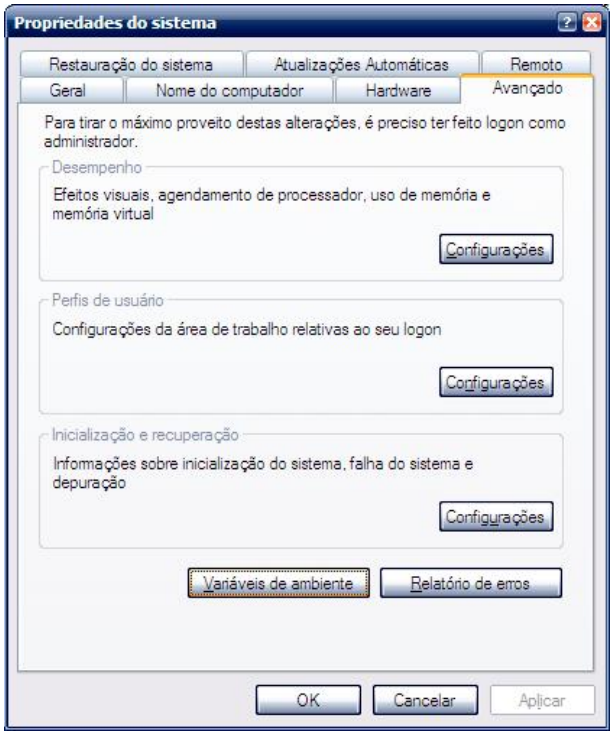

Figura 2: Guia "Avançado" da opção Sistema

Logo a seguir, é necessário clicar no botão "Nova". Uma janela como mostrada na Figura 3, é exibida.

O campo "Nome da variável" deve ser preenchido com o valor NICE. O campo "Valor da variável" deve ser preenchido com caminho da pasta onde se encontra o compilador. Para completar a configuração da variável NICE, deve-se clicar no botão "Ok".

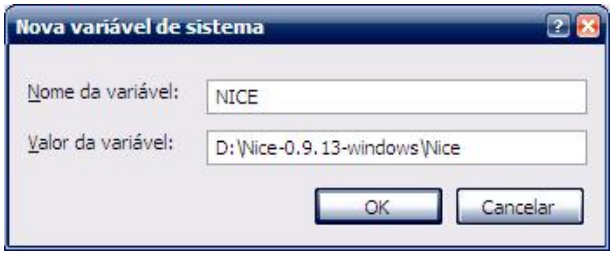

Figura 3: Janela de adicionar nova variável de ambiente

A variável de sistema chamada "Path" deve ser selecionada, em seguida se deve clicar no botão "Editar". Da mesma forma, irá aparecer uma janela como mostrada na Figura 3, porém com os campos preenchidos. O conteúdo ";%NICE%" deve ser adicionado no final do campo "Valor da variável" e todas as janelas abertas devem ser finalizadas ao se clicar no botão "Ok".

3.2 Compilação e execução de um programa

Após a instalação do compilador, é necessário saber como compilar um programa escrito na linguagem Nice; desta forma, esta subseção se dedica ao processo de compilação de códigos escritos na linguagem Nice.

Para compilar um código escrito na linguagem Nice é necessário que este esteja dentro de uma pasta. Por exemplo, dado um arquivo chamado *Arquivo.nice*, seu conteúdo é mostrado no Exemplo 1, dentro de uma pasta chamada *Pacote*, assim como mostrado na Figura 4.

Exemplo 1: – Conteúdo do arquivo chamado *Arquivo.nice*

```
1 void main(String[] args){
2         println("Olá mundo");
3 }
```

Resultado 1 – Resultado do Exemplo 1

```
Olá mundo
```

Figura 4: Estrutura de teste para compilação

Para compilar este código fonte no sistema operacional Linux, é necessário executar o seguinte comando: **nicec Pacote -a MeuPrograma.jar**, nota-se que no sistema operacional Windows, o comando é: **nicec.bat Pacote -a MeuPrograma.jar**. Após a compilação a estrutura ficará assim como mostrado na Figura 5.

A pasta chamada *nice* e os arquivos *dispach.class*, *fun.class* e *package.nicei* são arquivos resultantes do processo de compilação. Neste caso, o arquivo mais importante é o arquivo *MeuPrograma.jar*, que é a aplicação no formato Java Archive (JAR) [16]. Nota-se que o nome deste arquivo é especificado após a opção **-a** do comando de compilação. Para se testar a aplicação é necessário ter instalado o *Java Runtime Environment* (JRE) que pode ser obtido

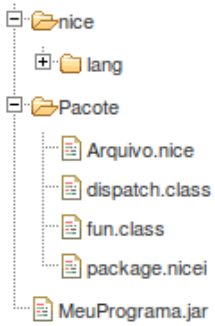

Figura 5: Estrutura da Figura 4 após compilação

no endereço `http://java.sun.com/javase/downloads`. Após a instalação do JRE, deve ser executado o comando: **java -jar MeuPrograma.jar**.

Para evitar misturar os códigos fontes com os códigos compilados é necessário especificar o destino do resultado da compilação com a opção **-d**, assim o comando ficaria da seguinte forma: **nicec Pacote -a MeuPrograma.jar -d bin**. Neste caso, *bin* é a pasta de destino dos códigos compilados. Após esta compilação a estrutura fica como ilustrado na Figura 6.

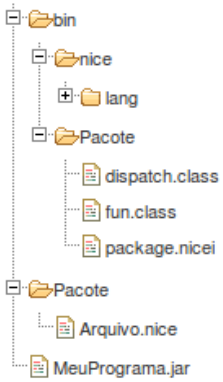

Figura 6: Estrutura da Figura 4 após compilação com destino especificado

Uma das características que facilitam a compilação de programas escritos na linguagem Nice é que **não** se especifica o código fonte, e sim o pacote que deseja-se compilar. Desta forma, o compilador resolve as dependências de pacotes compilando-os, se necessário.

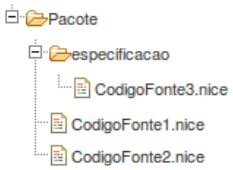

Figura 7: Estrutura de teste para compilação com sub-pacotes

No caso representado na Figura 7, existe apenas a subpasta "especificacao". Assim se não houver dependência dos arquivos contidos na pasta "Pacote" com à pasta "especificacao", a pasta "especificacao"não será compilada. As dependências serão especificadas na Subseção 4.1.

Para obter os resultados esperados das subseções 4.17 e 4.18 é necessário executar os arquivos com o comando: **java -ea -jar MeuPrograma.jar**. O parâmetro **-ea** habilita as asserções que por padrão são desabilitadas.

Uma característica dos programas compilados na linguagens Nice é que estes são executados com o mesmo JRE utilizado para aplicações compiladas na linguagem Java; assim, pode-se escrever aplicações sem a necessidade de reconfigurar o ambiente de produção.

3.3 Edição de códigos fontes

A edição de códigos fontes na linguagem Nice não exige nenhum editor especial. Assim, é possível utilizar qualquer editor de textos para esta finalidade. O editor Emacs [2] é automaticamente configurado para os usuários do sistema operacional Linux que instalarem o compilador pelo pacote no formato **.deb**.

As duas próximas subseções mostram como configurar os editores jEdit e o Eclipse, para facilitar a edição de códigos fontes.

3.4 Configuração do jEdit

Para instalar o pacote que fornece realce de sintaxe para o jEdit é necessário obter um arquivo chamado *nice.xml*. Este arquivo é distribuído no endereço `http://www.xoltar.org/old_site/2004/aug/05/jedit-nice-mode.html` e deve ser incluído na pasta *modes* do editor e em seguida alterar o arquivo *catalog* adicionando o conteúdo a seguir:

Exemplo 2: – Texto a ser adicionado no arquivo *catalog* do editor jEdit

```
1 <MODE NAME="nice" FILE="nice.xml" FILE_NAME_GLOB="*.nice" />
```

Este conteúdo informa ao editor que existe um novo módulo chamado *nice* descrito na propriedade *NAME*, na propriedade *FILE*, especifica que o arquivo *nice.xml* contém as definições do módulo e suas definições devem ser aplicadas nos arquivos com o padrão de nome **.nice* descrito na propriedade *FILE_NAME_GLOB* da Tag [24] *MODE*.

Após esta configuração, o editor estará preparado para realizar o realce de sintaxe nos códigos escritos na linguagem Nice.

3.5 Configuração do Eclipse

Para iniciar a configuração do Eclipse é necessário obter a extensão disponibilizada no site oficial da linguagem Nice, no seguinte endereço `http://nice.sourceforge.net/cgi-bin/twiki/view/Dev/EclipsePlugin#Download`.

O conteúdo do pacote deve ser movido para a pasta *plugins* no diretório da instalação do Eclipse. Caso este já esteja em execução é necessário reiniciá-lo. A partir deste momento o Eclipse estará configurado para criar projetos na linguagem Nice. É importante ressaltar que esta extensão está em desenvolvimento, mas já é muito útil.

Após a configuração é necessário acessar o menu *File -> New -> Project*, como mostrado na Figura 8.

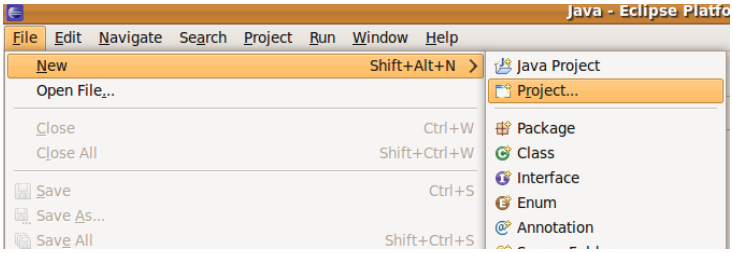

Figura 8: Primeiro passo para criar um projeto em Nice

Em seguida a opção *Nice Project* deve ser selecionada, como mostrado na Figura 9, em seguida clique no botão *Next*. Esta opção informa ao Eclipse o tipo de projeto que será criado, assim este irá realçar a sintaxe dos arquivos criados com a extensão *.nice*.

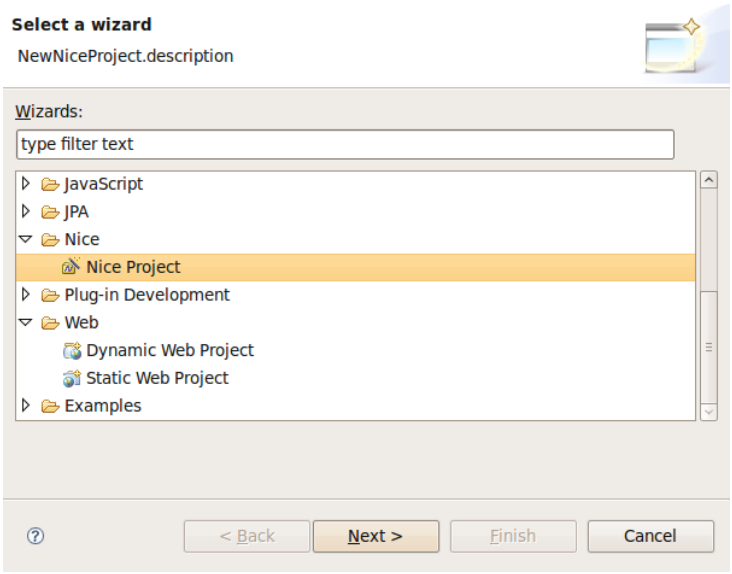

Figura 9: Segundo passo para criar um projeto em Nice

O passo seguinte é especificar o nome do projeto e clicar no botão *Finish*. Em seguida uma pasta raiz para o projeto deve ser criada, por exemplo uma pasta chamada *br*. O último passo é abrir as propriedades do projeto criado, para isso deve ser selecionado o projeto, em seguida acessar o menu *Project -> Properties*, selecionar a opção do painel esquerdo *Nice Project Properties*, assim como mostrado na Figura 10.

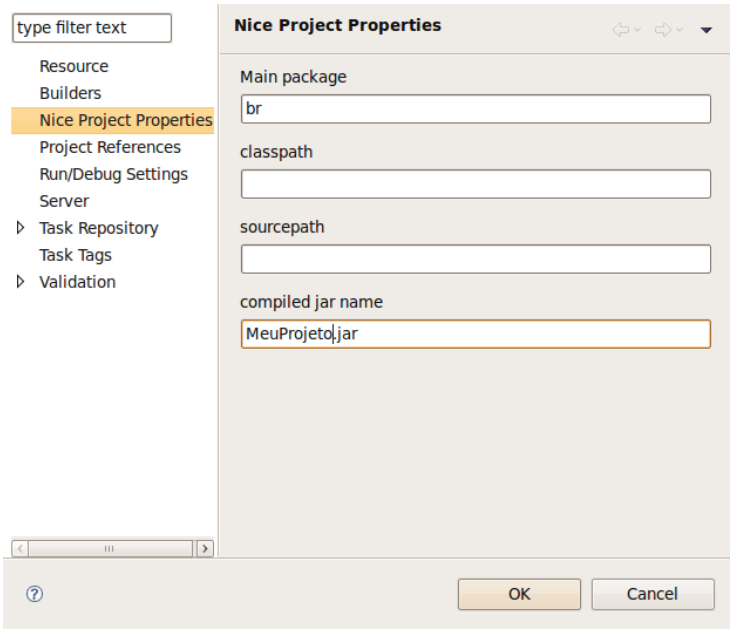

Figura 10: Terceiro passo para criar um projeto em Nice

Na janela da Figura 10, deve ser especificado o pacote principal que foi chamado de *br* e, caso seja necessário, especificar outro nome para o arquivo JAR que será criado.

Finalmente, o último passo é criar pelo menos um arquivo no pacote principal e salvá-lo.

O Eclipse irá compilar a aplicação a cada vez que um arquivo for alterado e salvo. O arquivo JAR resultante irá aparecer na pasta do projeto, caso erros sejam encontrados nos códigos, estes aparecerão na guia *Problems*.

4 Características da linguagem Nice

A linguagem Nice é orientada a objetos e baseada na linguagem Java, com características de programação funcional e possui soluções para muitos problemas encontrados na linguagem Java.

Ela foi criada com o uso de ferramentas disponibilizadas pela Sun Microsystems [23] como o JavaCC [3]. Desta forma, não há nenhum problema em definir outra linguagem com o objetivo de criar arquivos executáveis para a plataforma Java.

As características que serão apresentadas ao longo das subseções a seguir, proveem: expressividade, modularidade e segurança à linguagem Nice.

4.1 Importação de entidades

Para usar as entidades (classes, interfaces, enumerações, etc) de um pacote, assim como na linguagem Java é utilizado o comando *import*. Entretanto, existe uma diferença em relação a este comando com a linguagem Java. Em Java é possível a importação de classes específicas, como o comando: *import java.util.Random*. A linguagem Nice obriga a importação do pacote completo.

A importação do pacote completo pode ser expressa de duas formas diferentes, como tradicionalmente na linguagem Java: *import java.util.**, ou sem o .* (ponto asterisco) no final quando forem entidades escritas em Nice, por exemplo: *import br.rmt*, importa todas as entidades do pacote *br.rmt*.

Desta forma, para importar programas escritos na linguagem Nice é utilizado a importação do pacote, sem especificar a entidade desejada, e para importar entidades escritas na linguagem Java ou classes de um pacote JAR é utilizado a sintaxe com .* (ponto asterisco) no final. Assim, para importar a classe *Random* é necessário o seguinte comando: *import java.util.**, ou seja, importar o pacote completo.

4.2 Declaração de variáveis

Uma das grandes vantagens da linguagem Nice é a inferência de tipos de dados. Assim, para declarar uma variável, não é necessário especificar o tipo do dado, isto diminui o esforço de codificação. O Exemplo 3 mostra quatro maneiras distintas de declarar variáveis.

Exemplo 3: – Declaração de variáveis

```
1 //Forma genérica
2 //var [tipo] nome [= valor-inicial];
3 var nomeCompleto = "Ronnneesley Moura Teles";
4 var String nome;
5 var primeiroNome = "Ronneesley";
6 var String sobreNome = "Moura Teles";
7
8 //Compatibilidade
9 String meuNome = "Ronnneesley Moura Teles";
```

Para especificar uma variável é utilizada a palavra-chave **var**, assim como nas linhas 3 a 6. A compatibilidade com a declaração de variáveis com a linguagem Java é preservada para possibilitar uma adaptação mais rápida para os programadores da linguagem Java à linguagem Nice. Deste modo é possível declarar variáveis como na linha 9, sem utilizar a palavra-chave **var**.

Para que uma variável possa ser nula é preciso explicitar isso na declaração pelo uso do sinal de interrogação (?) antes do tipo de dado. Essa medida evita exceções de ponteiros nulos (*NullPointerException*). No Exemplo 4 é declarado uma variável chamada *nome* na linha 3 que pode ser nula.

Exemplo 4: – Declaração de variáveis que podem ser nulas

```
1 // Declaração de variável que pode ser nula, chamada nome
2 //de tipo literal
3 ?String nome = "Ronneesley";
4
5 // Para utilizar qualquer função de uma variável que pode
6 //ser nula, e obrigatório a verificação de seu conteúdo
7 if (nome != null) {
8         println(nome.length());
9 }
```

Resultado 2 – Resultado do Exemplo 4

```
10
```

Uma variável que não pode assumir o valor nulo é definida apenas pelo tipo de dado, mas também se pode utilizar a notação *!tipo_de_dado*. O Exemplo 5 mostra a declaração de uma variável que não pode assumir o valor nulo na linha 5.

Exemplo 5: – Declaração de variáveis não nulas utilizando a sintaxe *!tipo_de_dado*

```
1 //Variável não nula
2 String nome = "Ronneesley Moura Teles";
3
4 //Outra variável não nula
5 !String meuNome = "Ronneesley";
```

No exemplo anterior são definidas duas variáveis, uma chamada *nome*, na linha 2, e outra chamada *meuNome*, na linha 5; ambas não podem possuir o valor nulo. Desta forma, pode-se imaginar que a notação *!tipo_de_dado* não precisa ser utilizada em nenhum momento, pois apenas a notação *tipo_de_dado* atende a necessidade e o código fica simples de ser lido; entretanto, existe uma outra utilização para tal sintaxe que será explicada na subseção 4.27.

4.3 Declaração de constantes

Da mesma forma como a declaração de variáveis descrita na seção anterior, declaração de constantes muda apenas a palavra-chave **let** para indicar ao compilador que é uma constante.

Exemplo 6: – Declaração de constantes

```
1 //Forma genérica
2 //let [tipo] nome [= valor-inicial];
3 let pi = 3.14;
4 let CODIGO_SEXO_FEMININO = 2;
5 let CODIGO_SEXO_MASCULINO = 1;
6 let String nomeAutor = "Ronneesley";
```

Alterar o valor de uma constante após a primeira atribuição resulta em um erro de compilação pois como em todas as linguagens de programação, constantes não podem ser redefinidas. Assim, o código apresentado no Exemplo 7 parece uma alteração de constantes, mas não é, pois o valor só foi atribuído a constante *pi* na linha 3.

Exemplo 7: – Parece, mas não é alteração de constante

```
1 let double pi;
2
3 pi = 3.14;
```

Neste exemplo, é necessário declarar o tipo de dado, neste caso *double*, especificado na linha 1, pois como não há um valor inicial, assim o compilador não consegue fazer a inferência do tipo de dado.

4.4 Concatenação de literais

Em algumas linguagens de programação, literais são concatenadas com o operador de soma (+). Na linguagem Nice também é possível concatenar literais colocando-as lado a lado. O Exemplo 8 apresenta três maneiras diferentes de concatenar literais. A primeira utilizando o operador de soma na linha 2, a segunda por justaposição de literais na linha 5 e a terceira por justaposicação de literal e variável na linha 9.

Exemplo 8: – Concatenação de literais

```
1 //Concatenação convencional
2 var nome = "Ronneesley " + "Moura Teles";
3
4 //Concatenação por justaposição
5 var nomeCompleto = "Ronneesley " "Moura Teles";
6
7 //Concatenação por justaposição com variáveis
8 var sobreNome = "Moura Teles"
9 var nomeAutor = "Ronneesley " sobreNome;
```

As variáveis *nome*, *nomeCompleto* e *nomeAutor* possuirão o mesmo conteúdo, que será "Ronneesley Moura Teles", após o fim da execução do programa.

4.5 Exemplo de utilização de variáveis e constantes

O Exemplo 9, apresenta a utilização de variáveis e constantes.

Exemplo 9: – Utilização de variáveis e constantes

```
println("Novo comando equivalente ao System.out.println do Java");

//Compatibilidade
System.out.println("Conteúdo");

println("Utilização de variáveis e constantes");

var nome = "Ronneesley";
println("Nome: " nome);

nome += " Moura Teles";
println("Novo nome: " nome);

println("\nUtilização de constantes");

let pi = 3.14;
println("PI: " pi);

// Se a instrução da linha 22 não estivesse comentada
//resultaria em um erro, pois pi é uma constante
//e seu valor não pode ser alterado após a definição
//pi = 2.3;
```

Resultado 3 – Resultado do Exemplo 9

```
Novo comando equivalente ao System.out.println do Java
Conteúdo
Utilização de variáveis e constantes
Nome: Ronneesley
Novo nome: Ronneesley Moura Teles

Utilização de constantes
PI: 3.14
```

Neste exemplo foi apresentado uma nova função da linguagem Nice, chamada **println**, equivalente a função *System.out.println* da linguagem Java, nas linhas 1, 6, 9, 12, 14 e 17. Esta função facilita a escrita da função *System.out.println* muito utilizada nos programas sem interface gráfica. Por questões de compatibilidade e aprendizado é possível utilizar a função *System.out.println*, assim como demonstrado na linha 4 da mesma forma como é utilizada na linguagem Java.

4.6 Declaração de vetores

Além da declaração tradicional da linguagem Java, vetores podem ser declarados com a utilização de colchetes. O Exemplo 10, apresenta a declaração de um vetor de literais com quatro elementos.

Exemplo 10: – Declaração de vetor

```
1 //Declaração de um vetor chamado nomes com 4 elementos
2 String[] nomes = ["Ronneesley", "Ronneery", "Vilani",
3         "Sebastião"];
```

Caso seja um vetor comum é necessário informar o tipo de dado, neste exemplo *nomes* é um vetor de literais (String) com 4 elementos. A linguagem Nice também faz a inferência dos tipos de vetores, neste exemplo poderia ser retirado o tipo da variável *nomes*, neste caso (*String[]*) e colocar a palavra-chave **var**. Porém, ao fazer isto, a linguagem permite que o vetor possa conter mais de um tipo de dado, ou seja, poderá ter elementos inteiros, literais, flutuantes, etc.

A linguagem Nice possui uma função chamada *fill* para a atribuição de elementos de um vetor. O Exemplo 11 atribui o valor 5.000 a todas as 10 (dez) posições do vetor *salarios* utilizando a função *fill* na linha 3.

Exemplo 11: – Atribuição de valores simples as células de vetor

```
1 //Declaração de um vetor chamado salários
2 float[] salarios = new float[10];
3 fill(salarios, 5000.0);
```

Outra facilidade é atribuição de valores calculados com base na célula utilizando a função *fill*. O Exemplo 12, utiliza a função *fill* em conjunto com uma função de preenchimento.

Exemplo 12: – Atribuição de valores calculados as células de vetor

```
1 // Declaração de um vetor cuja os valores são o cubo
2 //do índice
3 int[] cubos = fill(new integer[4], int i => i ** 3);
```

Neste exemplo na linha 3 é utilizado o operador ** (asterisco asterisco) para elevar o índice *i* ao cubo (3), na linguagem Java seria o equivalente à: *(int) Math.pow(i, 3)*. Neste caso a função *fill* recebe dois parâmetros, o primeiro é o espaço de memória que irá trabalhar, neste caso *new integer[4]* e o segundo recebe uma função como parâmetro, esta característica será explicada na subseção 4.23, o valor passado como segundo parâmetro neste exemplo é uma função anônima, esta característica será explicada na subseção 4.24.

4.7 O laço *for* compacto

Apesar de não ser mais uma novidade, a linguagem Nice possui uma forma compacta do laço **for** para percorrer coleções. Especificamente, este laço compacto funciona para coleções dos tipos de dados: *Collections*, *Ranges*, *Strings* e *StringBuffers*. O Exemplo 13 imprime todos os nomes contidos no vetor *nomes* declarado na linha 1 utilizando um laço compacto na linha 4.

Exemplo 13: – Utilização de laços compactos

```
1 var String[] nomes = ["Ronneesley", "Ronneery",
2         "Maria", "Sebastião"];
3
4 for (String nome: nomes){
5         println("Nome: " nome);
6 }
```

Resultado 4 – Resultado do Exemplo 13

```
Nome: Ronneesley
Nome: Ronneery
Nome: Maria
Nome: Sebastião
```

Uma utilização do laço **for** compacto que é extremamente útil, usa coleções do tipo de dado *Ranges* que especifica um valor inicial e um valor final, com a seguinte notação: *numero-inicial..numero-final*. O Exemplo 14 apresenta um laço compacto que irá repetir 5 vezes e atribuir à variável *numero* os valores 10, 11, 12, 13, 14 e 15.

Exemplo 14: – Utilização de laços compactos com coleção do tipo de dado *Ranges*

```
1 for (int numero: 10..15){
2         println("Número " numero);
3 }
```

Resultado 5 – Resultado do Exemplo 14

```
Número 10
Número 11
Número 12
Número 13
Número 14
Número 15
```

O Exemplo 15 mostra como implementar laços utilizando a função *foreach* dos vetores.

Exemplo 15: – Utilização de laço com a função *foreach*

```
1 void calculo(int valor){
2         println("Valor: " valor " ** 2 = " + (valor ** 2));
3 }
4
5 void main(String[] args){
6         var colecao = [1, 1, 2, 3, 5, 8];
7
8         colecao.foreach(calculo);
9 }
```

Resultado 6 – Resultado do Exemplo 15

```
Valor: 1 ** 2 = 1
Valor: 1 ** 2 = 1
Valor: 2 ** 2 = 4
Valor: 3 ** 2 = 9
Valor: 5 ** 2 = 25
Valor: 8 ** 2 = 64
```

No Exemplo 15, cada item do vetor declarado na linha 6, chamado de *colecao*, será passado como parâmetro à função *calculo* declarada na linha 1, a declaração de função será detalhada na subseção 4.9.

4.8 Função *main*

Uma função com o nome *main* que recebe um vetor de literais (*String[]*) como parâmetro e não retorna nada (*void*). Ela é tratada de forma diferenciada para os programas escritos em Nice. Ao executar o programa, a função *main* é acionada e é passado a ela como parâmetros os argumentos da linha de comando do sistema operacional.

Outra característica muito importante desta função é que deve ser declarada fora da classe, ou seja, é uma função do pacote o que diferencia de forma significativa dos programas escritos na linguagem Java. O Exemplo 16, apresenta uma função *main*.

Exemplo 16: – Função *main*

```
1  void main(String[] args){
2   //Cria uma variável para contador
3   //O valor do parâmetro se inicia em 1
4   int contador = 1;
5
6   //Percorre todos os argumentos passados pelo
7   //sistema operacional
8   for (String argumento: args){
9    //Exibe na saída padrão o número do parâmetro e o valor
10   println("Parâmetro: " contador ", Valor: " argumento);
11   //Incrementa o contador
12   contador++;
13  }
14 }
```

Resultado 7 – Resultado do Exemplo 16 (**java -jar Testes.jar Ronneesley Moura Teles**)

```
Parâmetro: 1, Valor: Ronneesley
Parâmetro: 2, Valor: Moura
Parâmetro: 3, Valor: Teles
```

Por se tratar de uma função especial, deve-se possuir apenas uma função *main* em todo o projeto, caso contrário o compilador irá retornar uma mensagem de erro.

4.9 Declaração de funções

Ao contrário da linguagem Java, a linguagem Nice permite a declaração de função sem vínculo com uma classe. O Exemplo 17 apresenta a declaração da função *escreverNomeAutor*, na linha 2, e, *escreverPI*, na linha 7.

Exemplo 17: – Declaração de função

```
1  //Retorno do tipo void, nome da função: escreverNomeAutor
2  void escreverNomeAutor(){
3          println("Ronneesley Moura Teles");
4  }
5
6  //Retorno do tipo void, nome da função: escreverPI
7  void escreverPI(){
8          println(3.14);
9  }
10
11 //Função main
12 void main(String[] args){
13         //Chamada à função: escreverNomeAutor
14         escreverNomeAutor();
15
16         //Chamada à função: escreverPI
17         escreverPI();
18 }
```

Resultado 8 – Resultado do Exemplo 17

```
Ronneesley Moura Teles
3.14
```

Funções que possuem apenas uma expressão podem ser escritas em uma linha na linguagem Nice. No Exemplo 17 as funções *escreverNomeAutor* e *escreverPI*, possuem apenas uma expressão, logo podem ser escritas como no Exemplo 18.

Exemplo 18: – Declaração de função de apenas uma expressão

```
1  //Retorno do tipo void, nome da função: escreverNomeAutor
2  void escreverNomeAutor() = println("Ronneesley Moura Teles");
3
4  //Retorno do tipo void, nome da função: escreverPI
5  void escreverPI() = println(3.14);
```

Resultado 9 – Resultado do Exemplo 18

Ronneesley Moura Teles
3.14

Assim como as variáveis, os parâmetros de uma função podem ser nulos. O Exemplo 19 declara uma função chamada *boasVindas* na linha 3, que possui o segundo parâmetro podendo receber o valor nulo.

Exemplo 19: – Função com parâmetros nulos

```
1  // Função de boas vindas, com o parâmetro sobreNome podendo
2  //ser nulo
3  void boasVindas(String nome, ?String sobreNome) {
4      //Verifica se a variável sobreNome e nula
5      if (sobreNome != null) {
6          //Imprime "Seja bem vindo Fulano Sobrenome"
7          println("Seja bem vindo, " nome " " sobreNome);
8      } else {
9          //Imprime "Seja bem vindo Fulano"
10         println("Seja bem vindo, " nome);
11     }
12 }
13
14 //Função main
15 void main(String[] args){
16     //Chamada à função de boas vindas com o sobre nome nulo
17     boasVindas("Ronneesley", null);
18 }
```

Resultado 10 – Resultado do Exemplo 19

Seja bem vindo, Ronneesley

Uma grande vantagem da linguagem Nice é a existência de parâmetros opcionais de funções. Desta forma, no Exemplo 19 não seria necessário passar o segundo parâmetro como nulo (*null*) na linha 17, para isso se deve atribuir o valor desejado como padrão ao parâmetro *sobreNome*.

Exemplo 20: – Função com parâmetros opcionais

```
1  // Função de boas vindas, com o parâmetro sobreNome opcional
2  void boasVindas(String nome, ?String sobreNome = null) {
3      //Verifica se a variável sobreNome é nula
4      if (sobreNome != null) {
5          //Imprime "Seja bem vindo Fulano Sobrenome"
6          println("Seja bem vindo, " nome " " sobreNome);
7      } else {
8          //Imprime "Seja bem vindo Fulano"
9          println("Seja bem vindo, " nome);
10     }
11 }
12
```

```
13 //Função main
14 void main(String[] args){
15     //Chamada a função de boas vindas com o sobre nome nulo
16     boasVindas("Ronneesley");
17 }
```

Resultado 11 – Resultado do Exemplo 20

```
Seja bem vindo, Ronneesley
```

É importante ressaltar que o compilador Nice não proíbe a declaração de parâmetros opcionais antes dos parâmetros obrigatórios, pois esta linguagem permite a especificação explícita e não somente sequencial dos parâmetros. O Exemplo 21 declara o parâmetro *sobreNome* que pode receber o valor nulo, antes do parâmetro *nome*, na função *boasVindas* declarada na linha 2.

Exemplo 21: – Especificação explícita de parâmetros

```
1  // Função de boas vindas, com o parâmetro sobreNome opcional
2  void boasVindas(?String sobreNome = null, String nome) {
3      //Verifica se a variável sobreNome é nula
4      if (sobreNome != null) {
5          //Imprime "Seja bem vindo Fulano Sobrenome"
6          println("Seja bem vindo, " nome " " sobreNome);
7      } else {
8          //Imprime "Seja bem vindo Fulano"
9          println("Seja bem vindo, " nome);
10     }
11 }
12
13 //Função main
14 void main(String[] args){
15     // Chamada a função de boas vindas com o sobrenome nulo
16     //e o nome especificado explicitamente
17     boasVindas(nome:"Ronneesley");
18 }
```

Resultado 12 – Resultado do Exemplo 21

```
Seja bem vindo, Ronneesley
```

Neste exemplo, a função *boasVindas* declarada na linha 2, possui em sequência os parâmetros *sobreNome* e *nome*. Neste caso, para especificar somente o parâmetro *nome*, é necessário especificar explicitamente, assim como na chamada de função na linha 17, utilizando a sintaxe: *parâmetro:valor*.

4.10 Declaração de classes

Da mesma forma como na linguagem Java, classes são declaradas como mostrado no Exemplo 22.

Exemplo 22: – Declaração de Classe

```
1 //Declaração da classe chamada NomeClasse
2 class NomeClasse { //Início corpo da classe
3
4         //Código da classe
5
6 } //Fim do corpo da classe
```

4.10.1 Atributos

Um dos componentes de uma classe é o atributo. Este permite o armazenamento de informações em uma instância da classe. Um atributo possui um tipo, um nome e pode possuir um valor inicial. O Exemplo 23 declara um atributo na linha 6 e outro na linha 10.

Exemplo 23: – Atributos de uma classe

```
1  //Declaração da classe
2  class Exemplo {
3   //Outros códigos
4
5   //Atributo contador do tipo int
6   int contador;
7
8   //Atributo nome do tipo literal
9    //valor inicial "Ronneesley"
10  String nome = "Ronneesley";
11
12  //Outros códigos
13 } //Fim da declaração da classe
```

Assim como as variáveis e parâmetros, os atributos de uma classe também podem possuir valor nulo. O Exemplo 24 mostra na linha 8 um atributo chamado *remuneracao* que pode receber valor nulo.

Exemplo 24: – Atributos de uma classe

```
1 //Declaração da classe
2 class Colaborador {
3  //Atributo nome do tipo literal
4   //valor inicial "Ronneesley"
5  String nome = "Ronneesley";
6
7  //Variável que pode ter valor nulo
8  ?float remuneracao;
9 } //Fim da declaração da classe
```

No Exemplo 24, o atributo *remuneracao*, declarado na linha 8, poderá receber o valor nulo (*null*). Da mesma forma como as variáveis, é utilizado o caractere de interrogação (?) para indicar ao compilador que o atributo poderá ser nulo.

4.10.2 Construtores

Uma das características da linguagem Nice é a criação automática de construtores pelo compilador. Essa criação automática utiliza os atributos que não possuem valor inicial. Desta forma, é obrigatório especificar explicitamente, como parâmetro do construtor, os atributos da classe que não são inicializados. O Exemplo 25 declara uma classe com 3 (três) atributos, sendo um deles opcional declarado na linha 8.

Exemplo 25: – Instanciando uma classe

```
1  //Declaração da classe
2  class Colaborador {
3   //Atributo nome do tipo String
4    //valor inicial "Ronneesley"
5   String nome = "Ronneesley";
6
7   //Variável que pode ter valor nulo
8   ?float remuneracao;
9
10  int cargaHoraria;
11
12  //Descrições gerais sobre o colaborador
13  String toString(){
14    return "Nome: " nome ", remuneração: " remuneracao
15       ", carga horária: " cargaHoraria;
16  }
17 } //Fim da declaração da classe
18
19 void main(String[] args){
20    var colaborador = new Colaborador(cargaHoraria:8,
21       remuneracao:5000);
22    println(colaborador);
23 }
```

Resultado 13 – Resultado do Exemplo 25

```
Nome: Ronneesley, remuneração: 5000, carga horária: 8
```

Nota-se que na classe *Colaborador*, declarada na linha 2 a 23 do Exemplo 25, o atributo *nome*, declarado na linha 5, possui o valor inicial *"Ronneesley"*, o atributo *cargaHoraria*, declarado na linha 10, não possui valor inicial, e o atributo *remuneracao*, declarado na linha 8, pode receber o valor nulo, mas não possui valor inicial. Assim, o compilador forçará a especificação explícita do atributo *remuneracao* e *cargaHoraria*.

As chamadas aos construtores de classe assim como acontece na linha 20 com a classe *Colaborador* necessitam que os parâmetros sejam especificados explicitamente, pois um problema com relação a linguagem Java é que os parâmetros devem ser informados na mesma ordem da declaração do construtor, deste modo podem surgir dois tipos de problemas com relação a esta característica da linguagem Java.

Nota-se que esta característica somente funcionará para os códigos escritos na linguagem Nice. O Exemplo 26 demonstra a chamada de um construtor de uma classe não escrita na linguagem Nice, o mesmo se aplica para bibliotecas externas JAR.

Exemplo 26: – Construtor com classes externas

```
1 void main(String[] args){
2         var nome = new String("Ronneesley");
3
4         println(nome);
5 }
```

Resultado 14 – Resultado do Exemplo 26

```
Ronneesley
```

Neste exemplo é necessário especificar a sequência exata dos parâmetros. Desta forma, os problemas citados a seguir poderão ocorrer nestes casos.

O primeiro problema se manifesta quando é necessário realizar uma troca da sequência entre dois parâmetros do construtor e estes parâmetros possuem tipos de dados diferentes. Neste caso, é criado um problema de sintaxe em todas as chamadas ao construtor. O segundo problema se manifesta quando é necessário realizar uma troca da sequência entre dois parâmetros do construtor e estes parâmetros possuem o mesmo tipo de dado. Neste caso, nenhum erro de sintaxe é retornado, pois os dois parâmetros possuem o mesmo tipo de dado. Assim, todas as chamadas ao construtor estarão sintaticamente corretas, mas logicamente passarão valores para as variáveis erradas. Este conceito também se aplica às funções descritas na subseção 4.9.

No Exemplo 27 é mostrado o primeiro destes problemas quando o erro resultante é de sintaxe e os parâmetros possuem tipos de dados diferentes.

Exemplo 27: – Classe na linguagem Java antes da alteração do construtor

```
1  //Classe Colaborador em Java
2  public class Colaborador {
3      //Atributo nome do tipo String
4       //valor inicial "Ronneelsey"
5      String nome = "Ronneesley";
6
7      //Variável de remuneração
8      float remuneracao;
9
10     int cargaHoraria;
11
12     //Descrições gerais sobre o colaborador
13     public String toString(){
14         return "Nome: " + nome + ", remuneração: " +
15             remuneracao + ", carga horária: " +
16             cargaHoraria;
17     }
18
19     //Construtor
20     public Colaborador(int cargaHoraria, float remuneracao){
21         this.remuneracao = remuneracao;
22         this.cargaHoraria = cargaHoraria;
23     }
24
25     //Método principal
```

```
26      public static void main(String[] args){
27          Colaborador colaborador = new Colaborador(8, 5000.0F);
28          System.out.println(colaborador);
29      }
30  } //Fim da declaração da classe
```

Resultado 15 – Resultado do Exemplo 27

```
Nome: Ronneesley, remuneração: 5000.0, carga horária: 8
```

Caso a ordem dos parâmetros do construtor tenha que ser alterada para a forma mostrada no Exemplo 28. O compilador Java acusaria um erro de sintaxe na linha 27 do exemplo 27.

Exemplo 28: – Construtor na linguagem Java depois da alteração

```
1  //Construtor
2  public Colaborador(float remuneracao, int cargaHoraria){
3      this.remuneracao = remuneracao;
4      this.cargaHoraria = cargaHoraria;
5  }
```

Resultado 16 – Resultado do Exemplo 28 (**javac Colaborador.java**)

```
Colaborador.java:27: cannot find symbol
symbol : constructor Colaborador(int,float)
location: class Colaborador
Colaborador colaborador = new Colaborador(8, 5000.0F);
1 error
```

O segundo problema é quando o erro resultante é lógico e os parâmetros possuem o mesmo tipo de dado. O Exemplo 29 mostra a classe *Colaborador* com os atributos *remuneracao* na linha 8 e *cargaHoraria* na linha 10, ambos com o tipo de dado *float*.

Exemplo 29: – Construtor na linguagem Java antes da alteração do construtor

```
1  //Classe Colaborador na linguagem Java
2  public class Colaborador {
3      //Atributo nome do tipo String
4       //valor inicial "Ronneelsey"
5      String nome = "Ronneesley";
6
7      //Variável de remuneração
8      float remuneracao;
9
10     float cargaHoraria;
11
12     //Descrições gerais sobre o colaborador
13     public String toString(){
14         return "Nome: " + nome + ", remuneração: " +
15             remuneracao + ", carga horária: " +
16             cargaHoraria;
17     }
```

```
18
19      //Construtor
20      public Colaborador(float cargaHoraria, float remuneracao){
21          this.remuneracao = remuneracao;
22          this.cargaHoraria = cargaHoraria;
23      }
24
25      //Método principal
26      public static void main(String[] args){
27          Colaborador colaborador = new Colaborador(8.0F, 5000.0F);
28          System.out.println(colaborador);
29      }
30  } //Fim da declaração da classe
```

Resultado 17 – Resultado do Exemplo 29

```
Nome: Ronneesley, remuneração: 5000.0, carga horária: 8.0
```

A única diferença do Exemplo 27, para o Exemplo 29 é a modificação do tipo de dado do atributo *cargaHoraria* na linha 10 e do tipo de dado do parâmetro *cargaHoraria* na linha 20 do construtor, ambos foram modificados para o tipo de dado flutuante (*float*).

Da mesma forma, se a ordem dos parâmetros do construtor tiver que ser alterada para a forma mostrada no Exemplo 30, o compilador Java não irá acusar nenhum erro na linha 27 do Exemplo 29, pois ambos os parâmetros são do tipo flutuante.

Exemplo 30: – Construtor na linguagem Java depois da alteração

```
1  //Construtor
2  public Colaborador(float remuneracao, float cargaHoraria){
3      this.remuneracao = remuneracao;
4      this.cargaHoraria = cargaHoraria;
5  }
```

Resultado 18 – Resultado do Exemplo 30

```
Nome: Ronneesley, remuneração: 8.0, carga horária: 5000.0
```

Neste caso, o erro é mais grave do que o descrito anteriormente, pois é um erro lógico. Se este problema acontecesse em um programa em produção, seria interpretado da seguinte maneira:

"O colaborador Ronneesley terá uma remuneração de $8,00$ *R$ e trabalhará* 5.000 *horas por dia."*.

Assim, até que uma pessoa perceba o erro lógico no código, o problema pode se tornar grave. Neste exemplo, provavelmente no primeiro mês de trabalho o colaborador iria se manifestar, pois seu pagamento teria uma diferença negativa de R$ $4.992,00$ e ainda contaria 4.992 horas por dia em débito, ao estimar um mês de 30 dias, o colaborador deveria 149.760 horas para a empresa, o que significa 6.240 dias de trabalho. Neste caso, o colaborador não ficaria satisfeito com esta situação.

Outra questão relativa a construtores é a maneira de escrever construtores personalizados. Assim como a função *main*, construtores personalizados também são declarados fora da classe. O Exemplo 31 declara um construtor personalizado na linha 21.

Exemplo 31: – Construtor personalizado na linguagem Nice

```
1  //Classe Pessoa
2  class Pessoa {
3      //Atributos
4      String nome;
5      ?String sobreNome = "Moura Teles";
6      int idade;
7
8      //Dados da pessoa
9      String toString(){
10         var auxSobreNome = sobreNome;
11         if (auxSobreNome != null) {
12             return "Nome: " nome ", sobrenome: " auxSobreNome
13                 ", idade: " idade;
14         } else {
15             return "Nome: " nome ", idade: " idade;
16         }
17     }
18 }
19
20 //Construtor personalizado
21 new Pessoa(){
22     println("Outras instruções");
23
24     this(nome:"Ronneesley", idade:21);
25 }
26
27 //Função main
28 void main(String[] args){
29     var pessoa = new Pessoa();
30     println(pessoa);
31 }
```

Resultado 19 – Resultado do Exemplo 31

```
Outras instruções
Nome: Ronneesley, sobrenome: Moura Teles, idade: 21
```

Neste exemplo, é utilizada a palavra chave **new** na linha 21 e, em seguida, o nome da classe para indicar que se trata de um construtor personalizado. Outro detalhe importante é que a última instrução de um construtor personalizado é a chamada à função *this* na linha 24 que ativará outro construtor. Muitas vezes, este construtor a ser acionado é um construtor criado automaticamente, como descrito anteriormente.

4.11 Inicializadores

Na linguagem Nice, também é possível a definição de inicializadores para uma classe. Inicializadores (do inglês *initializers*) é um bloco de código que é executado depois do término de qualquer chamada a um construtor. O Exemplo 32, apresenta a declaração de um inicializador nas linhas 4 a 6.

Exemplo 32: – Classe com 1 (um) inicializador

```
1  //Classe Carro
2  class Carro {
3      //Inicializador
4      {
5          println("Brum");
6      }
7      //Fim do inicializador
8
9      void acelarar() = println("Vrummmmmm");
10 }
11
12 //Classe principal
13 void main(String[] args){
14     //Instância a classe carro
15     var carro = new Carro();
16     carro.acelarar();
17 }
```

Resultado 20 – Resultado do Exemplo 32

```
Brum
Vrummmmmm
```

No Exemplo 32 a chave de abertura ({) na linha 4, marca o início e a chave de fechamento (}), na linha 6, marca o fim do inicializador. As instruções contidas neste bloco de código serão executadas a cada vez que uma nova instância de classe for dimensionada, assim como acontece na linha 15.

Uma classe pode ter mais de um inicializador. O Exemplo 33, apresenta dois inicializadores para uma classe, um declarado na linha 4 e o outro declarado nas linhas 6 a 7.

Exemplo 33: – Classe com 2 (dois) inicializadores

```
1  class Pessoa {
2          String nome = "Roni";
3
4          { println("Inicializador 1"); }
5
6          { println("Inicializador 2");
7            nome = "Ronneesley"; }
8  }
9
10 void main(String[] args){
11         var pessoa = new Pessoa();
12         println(pessoa.nome);
13 }
```

Resultado 21 – Resultado do Exemplo 33

```
Inicializador 1
Inicializador 2
Ronneesley
```

Nestes casos, a sequência de execução é feita de cima para baixo, assim será executado primeiro o inicializador declarado na linha 4 de depois o declarado nas linhas 6 a 7.

4.12 Classes paramétricas

Programação com classes paramétricas também é comumente chamada de programação genérica (*generic programming*). Assim como a linguagem Java, a linguagem Nice também possui este recurso. O Exemplo 34, apresenta uma classe chamada *Carro* (linha 4), cujo atributo *rodas* (linha 6), é determinado por um genérico, chamado *RODA*.

Exemplo 34: – Programação genérica

```
1  import java.util.*;
2
3  //Classe de carro
4  class Carro<RODA> {
5      //Rodas
6      List<RODA> rodas = new ArrayList();
7
8      //Adicionar roda
9      void adicionar(RODA roda) =
10         if (rodas.size() < 4)
11             rodas.add(roda)
12         else
13             println("Excesso de rodas");
14
15     void quantidadeRodas() =
16         println("Quantidade de rodas: " rodas.size());
17 }
18
19 //Classes de roda
20 class Roda {}
21 class RodaAro15 extends Roda {}
22 class RodaAro19 extends Roda {}
23
24 //Função main
25 void main(String[] args){
26     //Instância com o tipo de parâmetro RodaAro15
27     Carro<RodaAro15> carro = new Carro();
28
29     carro.adicionar(new RodaAro15());
30     carro.adicionar(new RodaAro15());
31     carro.adicionar(new RodaAro15());
32
33     // A linha 35 resultaria em uma excessão se não estivesse
34     //comentada
35     //carro.adicionar(new RodaAro19());
36
37     carro.adicionar(new RodaAro15());
38
39     carro.quantidadeRodas();
40 }
```

Resultado 22 – Resultado do Exemplo 34
Quantidade de rodas: 4

A grande diferença na declaração da classe é o identificador auxiliar na linha 4, chamado de *RODA*, dentro da classe ele será considerado como um tipo de dado.

Assim quando uma classe é instanciada deve ser especificado o tipo de dado do parâmetro genérico, como na linha 27. Ao contrário da linguagem Java, não é permitido especificar duas vezes o tipo de dado do parâmetro genérico, esta característica diminui o esforço de codificação.

A grande importância deste recurso é a especificação de um tipo de dado para a instância, assim como especificado na linha 27, para isso a classe possui uma definição genérica, assim como especificado na linha 4. O Exemplo 35, apresenta uma segunda utilização dos genéricos.

Exemplo 35: – Programação genérica com uma classe de contrato

```
1  //Classe de contrato
2  class Contrato<PESSOA> {
3      ?PESSOA pessoa = null;
4  }
5
6  //Classes
7  class Cliente {}
8  class Fornecedor {}
9
10 //Função main
11 void main(String[] args){
12     Contrato<Cliente> contrato = new Contrato();
13     contrato.pessoa = new Cliente();
14
15     // A linha 17 resultaria em um erro, caso não estivesse
16     //comentada, pois é esperado um elemento da classe cliente
17     //contrato.pessoa = new Fornecedor();
18 }
```

4.13 Multi-métodos

A linguagem Nice permite a criação de métodos que não são declarados dentro de uma entidade (classe, interface, etc.). Esta característica é chamada de multi-métodos (do inglês *multi-methods*). O Exemplo 36, apresenta uma classe chamada *Carro* (linha 1), que será utilizada no Exemplo 37.

Exemplo 36: – Classe *Carro*

```
1  class Carro {
2          int rodas = 4;
3          int velocidade = 0;
4
5          public void acelerar(){
6                  this.velocidade++;
7          }
8
9          //Método comum
10         public void buzinar(){
```

```
11                println("Fonfom");
12            }
13    }
```

Se a classe *Carro* do Exemplo 36, estivesse em um arquivo ou biblioteca que não pudesse ser alterada e necessita-se adicionar novas funções para esta classe. Logo, pode-se utilizar esta característica para declaração destas novas funções. Estas novas funções são declaradas em um outro arquivo. O Exemplo 37 apresenta uma nova função declarada na linha 1, para a classe *Carro* do Exemplo 36.

Exemplo 37: – Utilizador da classe Carro

```
1  public void novaBuzina(Carro carro){
2          println("Honk");
3  }
4
5  void main(String[] args){
6          var carro = new Carro();
7          carro.buzinar();
8          carro.novaBuzina();
9  }
```

Resultado 23 – Resultado do Exemplo 37

```
Fonfom
Honk
```

No Exemplo 37, o método *novaBuzina* na linha 1, torna-se um método da classe *Carro* e poderá ser utilizado, assim como na linha 8, da mesma forma que uma função declarada na própria classe, tal como a função *buzinar*, declarada na linha 10, do Exemplo 36, e chamada na linha 7 deste exemplo.

4.14 Declaração de interfaces

A seguir um exemplo de declaração clássica de interface:

Exemplo 38: – Declaração clássica de interface

```
1  interface Celular {
2          public void ligar();
3          public void desligar();
4          public void realizarChamada(String numero);
5          public void terminarChamada();
6  }
```

No Exemplo 38, a interface *Celular* obriga a classe que a implementa, a codificar todos os protótipos dos métodos declarados. Assim, é necessário implementar as funções: *ligar*, *desligar*, *realizarChamada* e *terminarChamada*, respectivamente declaradas nas linhas 2, 3, 4 e 5.

O Exemplo 39, mostra uma classe chamada *Celular123* que implementa a interface *Celular*, assim esta classe deverá implementar todos os protótipos declarados nesta interface. Assim como na linguagem Java, quando uma classe implementa uma interface, não é obrigatório que

esta classe se restrinja à implementação dos protótipos declarados na interface, desta forma a classe *Celular123* implementa uma função chamada *outraImplementacao* na linha 13 que não foi definida pela interface *Celular*.

Exemplo 39: – Classe que implementa a interface *Celular*

```
1  class Celular123 implements Celular {
2    public void ligar() = println("Seja bem vindo...");
3    public void desligar() = println("Obrigado nosso celular");
4
5    public void realizarChamada(String numero) =
6      println("Ligando para " numero "...");
7
8    public void terminarChamada(){
9      println("Chamada terminada...");
10     println("Duração: 00:10:03");
11   }
12
13   public void outraImplementacao() =
14     println("Outra implementação qualquer, não obrigatória");
15 }
```

Entretanto, as interfaces podem definir implementações e não somente protótipos, ao contrário da linguagem Java. O Exemplo 40, apresenta uma interface que possui apenas funções.

Exemplo 40: – Interface com implementações

```
1  interface CelularPadrao {
2    public void ligar() = println("Seja bem vindo...");
3    public void desligar() = println("Obrigado nosso celular");
4
5    public void realizarChamada(String numero) =
6          println("Ligando para " numero "...");
7
8    public void terminarChamada() {
9      println("Chamada terminada...");
10     println("Duração: 00:10:03");
11   }
12 }
```

Desta forma, qualquer classe que implementar a interface *CelularPadrao*, precisará codificar apenas os protótipos. Neste caso, não seria necessário implementar nenhum protótipo, pois na interface há somente funções. Existe uma diferença entre protótipo e função, protótipo não define um bloco de código, já uma função, define. O Exemplo 41, apresenta a classe *Celular321*, que implementa a interface *CelularPadrao*.

Exemplo 41: – Classe que implementa a interface *CelularPadrao*

```
1  class Celular321 implements CelularPadrao {
2  }
```

Assim, as classes definidas anteriormente: *Celular123* no Exemplo 39 e *Celular321* neste exemplo, poderiam ser usadas da seguinte forma:

Exemplo 42: – Classe principal para uso das implementações das interfaces

```
1  void main(String[] args){
2          println("Celular 123");
3          var celular123 = new Celular123();
4          celular123.ligar();
5          celular123.realizarChamada("1234-4321");
6          celular123.terminarChamada();
7          celular123.desligar();
8
9          println("\nCelular 321");
10         var celular321 = new Celular321();
11         celular321.ligar();
12         celular321.realizarChamada("1234-4321");
13         celular321.terminarChamada();
14         celular321.desligar();
15 }
```

Resultado 24 – Resultado do Exemplo 42

```
Celular 123
Seja bem vindo...
Ligando para 1234-4321...
Chamada terminada...
Duração: 00:10:03
Obrigado nosso celular

Celular 321
Seja bem vindo...
Ligando para 1234-4321...
Chamada terminada...
Duração: 00:10:03
Obrigado nosso celular
```

Outra opção para declaração dos protótipos e das funções de uma interface é a declaração destes externamente ao bloco de código da interface. O Exemplo 43 apresenta a declaração dos protótipos fora do bloco de código da interface.

Exemplo 43: – Protótipos fora do bloco da interface

```
1  interface Celular {
2  }
3
4  public void ligar(Celular celular);
5  public void desligar(Celular celular);
6  public void realizarChamada(Celular celular, String numero);
7  public void terminarChamada(Celular celular);
```

Neste exemplo os métodos: *ligar*, *desligar*, *realizarChamada* e *terminarChamada*, são exemplos de multi-métodos, descritos anteriormente na subseção 4.13.

Uma sugestão feita em [10] é declarar as funções externamente ao bloco de código da interface e deixar os protótipos dentro do bloco de código da interface. Desta forma pode-se encontrar, de forma rápida, o que deve ser implementado nas outras classes.

Ao implementar as funções de uma interface, como no Exemplo 39, o compilador irá notificar que não é necessário especificar o tipo do retorno. Assim, o programa ficaria da forma representada no Exemplo 44.

Exemplo 44: – Classe que implementa a interface Celular sem especificar o retorno

```
1  class Celular123 implements Celular {
2    public ligar() = println("Seja bem vindo...");
3    public desligar() = println("Obrigado nosso celular");
4
5    public realizarChamada(String numero) =
6      println("Ligando para " numero "...");
7
8
9    public terminarChamada(){
10     println("Chamada terminada...");
11     println("Duração: 00:10:03");
12   }
13
14   public void outrasImplementacoes() =
15     println("Outra implementação qualquer, não obrigatória");
16 }
```

Outra forma de implementar os métodos de uma interface é a especificação explícita, através da palavra-chave **override**, das funções que estão sendo implementadas. O Exemplo 45, apresenta o uso da palavra-chave **override** nas linhas 2, 3, 6 e 10.

Exemplo 45: – Classe que implementa a interface Celular com o uso da palavra-chave override

```
1  class Celular123 implements Celular {
2    public override ligar() = println("Seja bem vindo...");
3    public override desligar() =
4          println("Obrigado nosso celular");
5
6    public override realizarChamada(String numero) =
7      println("Ligando para " numero "...");
8
9
10   public override terminarChamada(){
11     println("Chamada terminada...");
12     println("Duração: 00:10:03");
13   }
14
15   public void outrasImplementacoes() =
16     println("Outra implementação qualquer, não obrigatória");
17 }
```

Outra característica dos métodos implementados de uma interface, é que os tipos de dados dos parâmetros das funções que foram definidas na interface, podem ser omitidos, pois o compilador consegue fazer a inferência dos tipos de dados através da interface. Assim, o tipo de dado literal (*String*) do parâmetro *numero*, na linha 6 deste exemplo, poderia ser omitido.

As propriedades descritas nos exemplos 44 e 45 também se aplicam em casos de herença entre classes, assim como será explicado na subseção 4.29.

4.15 Enumerações

Assim como outras linguagens, a linguagem Nice suporta a definição de constantes para conter um valor pré-definido. O Exemplo 46, utiliza duas constantes declaradas nas linhas 1 e 2, para imprimir um texto na função *escreverTipo*, declarada nas linhas 11 a 18.

Exemplo 46: – Constantes

```
1  let int FORNECEDOR = 1;
2  let int CLIENTE = 2;
3
4  class Pessoa {
5          private int tipo;
6
7          public void modificarTipo(int tipo){
8                  this.tipo = tipo;
9          }
10
11         public void escreverTipo(){
12                 if (this.tipo == CLIENTE)
13                         println("Cliente");
14                 else if (this.tipo == FORNECEDOR)
15                         println("Fornecedor");
16                 else
17                         println("Tipo não definido");
18         }
19 }
20
21 void main(String[] args){
22         var pessoa = new Pessoa(tipo:FORNECEDOR);
23         pessoa.escreverTipo(); //Saída: Fornecedor
24
25         pessoa.modificarTipo(CLIENTE);
26         pessoa.escreverTipo(); //Saída: Cliente
27
28         //A linha 30 está logicamente incorreta,
29         //mas sintaticamente correta
30         pessoa.modificarTipo(3000);
31         pessoa.escreverTipo(); //Saída: Tipo não definido
32 }
```

Resultado 25 – Resultado do Exemplo 46

```
Fornecedor
Cliente
Tipo não definido
```

No Exemplo 46, as constantes podem servir para atribuir valores pré-definidos, assim como acontece nas linhas 1 e 2. Entretanto, não garante que parâmetros de função e atributos recebam somente estes valores, pois estas constantes não definem um novo tipo de dado. Este fato pode ser visualizado na linha 29, onde é atribuído o valor 3000 a um parâmetro que logicamente deveria aceitar apenas os valores 1 e 2. Porém, neste caso não há nenhum erro

sintático envolvido, pois a função *modificarTipo*, declarada na linha 7, especifica um parâmetro chamado *tipo* que possui o tipo de dado inteiro (*int*), assim o valor 3000 da linha 29 também deve ser aceito, pois é um número inteiro.

Devido a este problema, foram criados as enumerações (*enums*). Com as enumerações o compilador consegue detectar estes erros lógicos em tempo de compilação, pois quando se utiliza enumerações, estes erros se transformam em erros sintáticos. O Exemplo 47, representa a maneira correta de reescrever o código do Exemplo 46.

Exemplo 47: – Enumeração

```
1  enum Tipo { FORNECEDOR, CLIENTE }
2
3  class Pessoa {
4          private Tipo tipo;
5
6          public void modificarTipo(Tipo tipo){
7                  this.tipo = tipo;
8          }
9
10         public void escreverTipo(){
11                 if (this.tipo == CLIENTE)
12                         println("Cliente");
13                 else if (this.tipo == FORNECEDOR)
14                         println("Fornecedor");
15                 else
16                         println("Tipo não definido");
17         }
18 }
19
20 void main(String[] args){
21         var pessoa = new Pessoa(tipo:FORNECEDOR);
22         pessoa.escreverTipo();
23
24         pessoa.modificarTipo(CLIENTE);
25         pessoa.escreverTipo();
26
27         // Esta chamada abaixo resultaria em um erro
28         //de sintaxe, se estivesse descomentada
29         /*pessoa.modificarTipo(3000);
30         pessoa.escreverTipo();*/
31 }
```

Resultado 26 – Resultado do Exemplo 47

```
Fornecedor
Cliente
```

Desta forma, o compilador restringe o uso dos valores aceitos. Assim, seria possível detectar o erro na linha 29, pois não é um valor da enumeração definida na linha 1, com os valores *FORNECEDOR* e *CLIENTE*.

Uma grande diferença das enumerações da linguagem Nice com relação a linguagem Java, é que na linguagem Nice não é permitido especificar o nome da enumeração juntamente com

seu valor. Desta forma, as sintaxes: *Tipo.CLIENTE* e *Tipo.FORNECEDOR*, não funcionariam. Outra questão importante é que se houver duas enumerações com valores de mesmo nome, o compilador acusaria um erro de ambiguidade.

Outra forma de uso das enumerações é a determinação de um valor associado a um valor da enumeração. O Exemplo 48, mostra na linha 1, como atribuir um número inteiro a um valor da enumeração.

Exemplo 48: – Enumeração com valor associado

```
1  enum Tipo(int valor) { FORNECEDOR(1), CLIENTE(2) }
2
3  class Pessoa {
4          private Tipo tipo;
5
6          public void modificarTipo(Tipo tipo){
7                  this.tipo = tipo;
8          }
9
10         public void escreverTipo(){
11                 if (this.tipo == CLIENTE)
12                         println("Cliente");
13                 else if (this.tipo == FORNECEDOR)
14                         println("Fornecedor");
15                 else
16                         println("Tipo não definido");
17         }
18 }
19
20 void main(String[] args){
21         var pessoa = new Pessoa(tipo:FORNECEDOR);
22         pessoa.escreverTipo();
23
24         println(pessoa.tipo.valor); //Saída 1
25
26         pessoa.modificarTipo(CLIENTE);
27         pessoa.escreverTipo();
28
29         println(pessoa.tipo.valor); //Saída 2
30 }
```

Resultado 27 – Resultado do Exemplo 48

```
Fornecedor
1
Cliente
2
```

No Exemplo 48 é associado o valor 1 à *FORNECEDOR* e 2 à *CLIENTE*. Este valor pode ser lido pelo nome do atributo, neste caso *valor*, assim como nas linhas 24 e 29.

4.16 Despacho por valor

É a sobrescrita de funções com base nos valores dos parâmetros da função. Esta característica funciona para inteiros, booleanos, literais, caracteres, enumerações e instâncias de classes. O Exemplo 49, usa o despacho por valor, nas linhas 7 e 8.

Exemplo 49: – Despacho por valor

```
1  //Função padrão
2  boolean pai(String pai, String filho) {
3    return false;
4  }
5
6  //Sobrescrita de função definida pelos valores dos parâmetros
7  pai("Sebastião", "Ronneesley") = true;
8  pai("João", "Sebastião") = true;
9
10 void main(String[] args){
11   println(pai("João", "Sebastião")); //Saída: true
12   println(pai("João", "Maria")); //Saída: false
13
14   //Saída: true
15   println(pai("Sebastião", "Ronneesley"));
16 }
```

Resultado 28 – Resultado do Exemplo 49

```
true
false
true
```

O modo de tratar estas funções lembra a maneira como a linguagem Prolog [21] define fatos.

4.17 Asserções

Asserções servem para verificar uma determinada condição para o funcionamento de um algoritmo. Caso esta condição seja verdadeira o programa irá continuar a execução normalmente, mas se a condição for falsa é disparada uma exceção do tipo *AssertionFailed* parando a execução do algoritmo. O exemplo a seguir demonstra um uso de asserções.

Exemplo 50: – Asserções

```
1  void main(String[] args){
2    var variavel = 10;
3
4    assert variavel == 10;
5
6    println("Tudo bem");
7
8    assert variavel == 11 : "A variável deveria ter valor 11";
9
10   println("Não chegará aqui se não tiver valor 11");
```

```
11 }
```

Resultado 29 – Resultado do Exemplo 50

```
Tudo bem
Exception in thread "main"nice.lang.AssertionFailed: A variável deveria ter valor 11
at principal.fun.main(Teste.nice:8)
at principal.dispatch.main(Unknown Source)
```

Nas linhas 4 e 8 do Exemplo 50, foram feitas duas asserções cuja condição era a igualdade da variável chamada de *variavel*, com os respectivos valores 10 e 11, na asserção da linha 8, além da comparação, é definido a mensagem *"A variável deveria ter valor 11"* caso a condição falhe. É interessante ressaltar que a condição pode ser realizada com qualquer expressão que retorne um valor booleano e não somente condições de igualdade.

Para testar o Exemplo 50 é necessário especificar o parâmetro **-ea** para o JRE, assim como definido na subseção 3.2.

Asserções fornecem uma garantia da qualidade, desde que especificadas corretamente, o fato de ter que habilitá-las se explica por serem utilizadas somente no ambiente de testes. Assim, não se torna necessário verificar as asserções no ambiente de produção, pois haveria uma perda de desempenho. Este conceito também se aplica na subseção 4.18.

4.18 Projeto por contrato

Projeto por contrato (do inglês *Design by Contract* ou simplesmente DbC) foi criado pelo francês Bertrand Meyer e especificado em [11] e consiste em estabelecer asserções de pré-condições, pós-condições e invariantes de modo a possibilitar testes durante a fase de desenvolvimento de software que não serão levados para a fase de produção.

Exemplo 51: – DbC com folha de pagamento

```
1  /**
2   * Classe para cálculos de exemplo da folha
3   */
4  class Folha {
5  /**
6   * Calcular salário
7   */
8   public float calcularFerias(float salario)
9    requires //Requerimentos
10     salario > 0 : "Salário deve possuir um valor positivo"
11   ensures //Garantias
12     result >= salario :
13           "Resultado não é maior do que o salário"
14   {
15    //Retorna o valor unitário mais 1/3 do salário
16    return 4F/3 * salario;
17   }
18 }
19
20 /**
21  * Função principal
```

```
22  */
23 void main(String[] args){
24  //Cria uma variável chamada folha do tipo folha
25  var folha = new Folha();
26  //Cria uma variável com o retorno do salário
27  var salarioFerias = folha.calcularFerias(1800);
28  //Exibe o resultado do cálculo
29  println("Salário de férias " salarioFerias);
30 }
```

Resultado 30 – Resultado do Exemplo 51

```
Salário de férias 2400.0
```

No Exemplo 51, foi utilizada a palavra-chave **requires** na linha 9, para especificar as pré-condições (requerimentos) para o funcionamento correto da função *calcularFerias*. Na linha 11, foi utilizada a palavra-chave **ensures** para especificar as pós-condições (garantias) ao ponto da chamada da função, neste caso na linha 27.

Da mesma forma como nas asserções especificadas na subseção 4.17, para testar este exemplo é necessário especificar o parâmetro **-ea** para JRE, assim como definido na subseção 3.2.

4.19 Funções locais

Funções locais são declaradas como todas as outras funções. Entretanto, este tipo de função é declarada dentro de um bloco de código de outra função. Este tipo de função, utiliza as variáveis locais da função em que está contida. O Exemplo 52 mostra uma função chamada *metodoSimples* escrita dentro da função *main*.

Exemplo 52: – Função local

```
1  void main(String[] args){
2          var valor = 10;
3
4          println("Valor anterior: " valor); //Saída: 10
5
6          void metodoSimples(){
7                  valor++;
8
9                  println("Novo valor: " valor); //Saída: 11
10         }
11
12         metodoSimples();
13
14         println("Último valor: " valor); //Saída: 11
15 }
```

Resultado 31 – Resultado do Exemplo 52

```
Valor anterior: 10
Novo valor: 11
Último valor: 11
```

No Exemplo 52 é declarada uma função chamada *metodoSimples* (linha 6) que utiliza a variável local chamada *valor* (linha 7), seu valor é alterado dentro e fora do escopo da função local quando chamada na linha 12.

4.20 Chamada de função

A chamada de função é feita normalmente na forma: *nome_função(parâmetro1, ..., parâmetroN)*. Entretanto, a linguagem Nice permite a chamada da seguinte maneira: *parâmetro1.nome_função(parâmetro2, ..., parâmetroN)*. O Exemplo 53, mostra nas linhas 7 a 9, as distintas maneira de chamar uma função.

Exemplo 53: – Chamada de função

```
1  int somar(int a, int b){
2          return a + b;
3  }
4
5  void main(String[] args){
6          //Saída: 3 em todos as chamadas
7          println(somar(1, 2));
8          println(somar(a:1, b:2));
9          println(1.somar(2));
10 }
```

Resultado 32 – Resultado do Exemplo 53

```
3
3
3
```

No Exemplo 53, na linha 7, é feita a chamada convencional, na linha 8 é feita a chamada com os parâmetros especificados explicitamente e na linha 9 é utilizado o primeiro parâmetro para chamar a função *somar*.

4.21 Conjunto de valores

Conjunto de valores (do inglês *Tuples*) representa um conjunto que podem ser de tipos de dados diferentes. O Exemplo 54, apresenta um conjunto de valores no parâmetro da função *distancia*, declarada na linha 1, e, um conjunto de valores na variável *ponto* (linha 6).

Exemplo 54: – Conjunto de valores

```
1  int distancia( (int x1, int x2) ){
2          return x2 - x1;
3  }
4
5  void main(String[] args){
6          var ponto = (3, 5);
7
8          //Distância
9          println(distancia( ponto ));
10 }
```

Resultado 33 – Resultado do Exemplo 54

```
2
```

No Exemplo 54, é definida uma variável chamada *ponto* na linha 6 cuja o conteúdo é um conjunto de valores e este é utilizado como parâmetro na chamada da função *distancia* na linha 9. A seguir um outro exemplo de utilização de conjunto de valores:

Exemplo 55: – Conjunto de valores com dados pessoais

```
1 void main(String[] args){
2         (int idade, String nome, String sobreNome) =
3                 (21, "Ronneesley", "Moura Teles");
4
5         println("Nome: " nome " " sobreNome);
6         println("Idade: " idade);
7 }
```

Resultado 34 – Resultado do Exemplo 55

```
Nome: Ronneesley Moura Teles
Idade: 21
```

4.22 Literal em múltiplas linhas

A linguagem Nice utiliza 3 (três) aspas assim como a linguagem Python para definição de literais com múltiplas linhas.

Exemplo 56: – Literal em múltiplas linhas

```
1 void main(String[] args){
2   var nome = """Ronneesley
3     Moura
4       Teles""";
5
6   println(nome);
7 }
```

Resultado 35 – Resultado do Exemplo 56

```
Ronneesley
Moura
Teles
```

Esta sintaxe permite a definição de um literal sem a necessidade de concatenar literais a cada quebra de linha.

4.23 Função como parâmetro de função

A linguagem Nice possibilita passar uma função como parâmetro de outra função. O Exemplo 57, apresenta nas linhas 1 e 2, a forma de tratar os parâmetros que representam funções.

Exemplo 57: – Função como parâmetro de função

```
1  void acao(?String mensagem, void->void sucesso,
2    void->void erro){
3    if (mensagem != null && mensagem.length > 0) {
4      println(mensagem);
5
6      sucesso();
7    } else
8       erro();
9  }
10
11 void sucessoAcao(){
12   println("Sucesso na execução da operação");
13 }
14
15 void erroAcao(){
16   println("Erro na execução da operação");
17 }
18
19 void main(String[] args){
20   //Sucesso na execução
21   acao("Seja bem vindo", sucessoAcao, erroAcao);
22
23   //Erro na execução
24   acao("", sucessoAcao, erroAcao);
25
26   //Erro na execução
27   acao(null, sucessoAcao, erroAcao);
28 }
```

Resultado 36 – Resultado do Exemplo 57

```
Seja bem vindo
Sucesso na execução da operação
Erro na execução da operação
Erro na execução da operação
```

A única modificação na declaração da função é o tipo de dado do parâmetro. Assim como nas linhas 1 e 2, este é representado na seguinte forma: *(parâmetros)->retorno*. Os parênteses que identificam os parâmetros são opcionais quando há somente um parâmetro. Quando a função não possui parâmetros, é utilizado o tipo de dado *void* para representar a ausência de parâmetros.

No Exemplo 57, as funções *sucessoAcao* e *erroAcao* não possuem parâmetros, logo é necessário identificá-las com o tipo de dado *void* e retornam *void*. Desta forma, o tipo de dado dos parâmetros *sucesso* e *erro* da função *acao*, declarada na linha 1, é formado com a seguinte sintaxe *void->void*.

O Exemplo 58, mostra uma função que recebe como parâmetro outra função, que possui um parâmetro inteiro e retorna um inteiro, formando a sintaxe *int->int*.

Exemplo 58: – Função como parâmetro de função, formato *int->int*

```
1  void funcao(int numero, int->int calculo){
2          println( numero + calculo(numero) );
3  }
4
5  int quadrado(int numero){
6          return int(numero ** 2);
7  }
8
9  int negativo(int numero){
10         return -numero;
11 }
12
13 void main(String[] args){
14         funcao(2, quadrado);
15
16         funcao(2, negativo);
17 }
```

Resultado 37 – Resultado do Exemplo 58

```
6
0
```

No Exemplo 58 são retornados os valores 6 e 0, respectivamente, para as chamadas das linhas 14 e 16, tendo em vista que a chamada da linha 14 passa a função *quadrado* como argumento e a chamada da linha 16 passa a função *negativo* como argumento.

Às vezes, é necessário passar mais de um parâmetro para a função, que é parâmetro de outra função. O Exemplo 59, demonstra como passar dois parâmetros inteiros para uma função que retorna um inteiro, formando a sintaxe *(int, int)->int*.

Exemplo 59: – Função como parâmetro de função, formato *(int, int)->int*

```
1  void funcaoComplexa(int numero,
2          (int, int)->int calculo){
3
4          println( numero + calculo(10, numero) );
5  }
6
7  int soma(int numero, int numero2){
8          return numero + numero2;
9  }
10
11 int subtracao(int numero, int numero2){
12         return numero - numero2;
13 }
14
15 void main(String[] args){
16         funcaoComplexa(2, soma);
17         funcaoComplexa(2, subtracao);
18 }
```

Resultado 38 – Resultado do Exemplo 59

```
14
10
```

4.24 Funções anônimas

Funções que não possuem nome e retorno são chamadas de funções anônimas. O compilador deve fazer inferência para descobrir o tipo de dado do retorno destas funções. Por não possuir um nome, estas funções não podem ser sobrescritas. O Exemplo 60, apresenta duas funções anônimas, uma nas linhas 8 e 9, e outra na linha 11.

Exemplo 60: – Função anônima

```
1  void mensagem(void->void operacao){
2    print("Mensagem: ");
3
4    operacao();
5  }
6
7  void main(String[] args){
8    mensagem( () =>
9      println("Sucesso na execução da operação") );
10
11   mensagem( () => println("Erro ao persistir") );
12 }
```

Resultado 39 – Resultado do Exemplo 60

```
Mensagem: Sucesso na execução da operação
Mensagem: Erro ao persistir
```

Não é necessário declarar uma função da maneira clássica, tendo em vista que ela só será chamada uma vez e o entendimento do código é facilitado com sua declaração no local de sua chamada.

No Exemplo 60, são utilizadas duas funções anônimas, respectivamente nas linhas 8 e 11. A sintaxe de declaração é a seguinte *(parâmetros) => expressão* ou *(parâmetros) => {expressões}*, quando a função anônima possuir apenas um parâmetro, o uso dos parênteses é opcional. O Exemplo 61, mostra o uso facultativo dos parênteses do retorno de uma função anônima, na linha 7.

Exemplo 61: – Função anônima com um parâmetro

```
1  void calculo(int->int operacao){
2          println(operacao(10));
3  }
4
5  void main(String[] args){
6          calculo( (int num) => 10 * num );
7          calculo( int num => 11 * num );
8  }
```

Resultado 40 – Resultado do Exemplo 61

```
100
110
```

No Exemplo 61, as saídas serão respectivamente 100 e 110. Nota-se que a manipulação destas funções utilizam os conceitos detalhados anteriormente na subseção 4.23.

Às vezes, pode ser necessário chamar uma função anônima mais de uma vez. Desta forma, pode-se utilizar uma variável para referenciá-la. O exemplo a seguir utiliza uma variável chamada *funcaoAnonima* para referenciar a função anônima.

Exemplo 62: – Função anônima com uma variável de referência

```
1 void main(String[] args){
2         var int->int funcaoAnonima = (int num) => 10 * num;
3
4         println(funcaoAnonima(10));
5         println(funcaoAnonima(11));
6 }
```

Resultado 41 – Resultado do Exemplo 62

```
100
110
```

É importante ressaltar que, nestes casos, muitas vezes é melhor declarar uma função da forma clássica, pois facilita o entendimento do código.

Da mesma forma, uma variável de referência pode ser passada como parâmetro de uma função. O Exemplo 63, mostra na linha 8, como passar uma variável que referencia uma função anônima, como parâmetro de uma função.

Exemplo 63: – Função anônima com uma variável de referência como parâmetro de função

```
1 void imprimir(int->int operacao){
2         println(operacao(15));
3 }
4
5 void main(String[] args){
6         var int->int funcaoAnonima = (int num) => 10 * num;
7
8         imprimir(funcaoAnonima);
9 }
```

Resultado 42 – Resultado do Exemplo 63

```
150
```

4.25 Conversão entre tipos primitivos

Os tipos primitivos de dados numéricos da linguagem Nice em ordem decrescente de tamanho são: *double*, *float*, *long*, *int*, *short* e *byte*. A conversão de um tipo de dado de tamanho menor para um tipo de dado de tamanho maior é automática, assim como apresentado no Exemplo 64.

Exemplo 64: – Conversão automática

```
1  void main(String[] args){
2          double numeroMaior = 10000;
3          println("Número maior: " numeroMaior);
4
5          byte numeroMenor = 10;
6          println("Número menor: " numeroMenor);
7
8          //Conversão automática
9          numeroMaior = numeroMenor;
10         println("Número maior: " numeroMaior);
11 }
```

Resultado 43 – Resultado do Exemplo 64

```
Número maior: 10000.0
Número menor: 10
Número maior: 10.0
```

A conversão de um tipo de dado de tamanho maior para um tipo de dado de tamanho menor é feita explicitamente, pois existe uma perda da magnitude do valor. Para converter explicitamente, é necessário chamar uma função com o nome do tipo de dado de destino, cuja o parâmetro é o valor a ser convertido, assim como apresentado no Exemplo 65, linha 9.

Exemplo 65: – Conversão explícita

```
1  void main(String[] args){
2          double numeroMaior = 8;
3          println("Número maior: " numeroMaior);
4
5          byte numeroMenor = 3;
6          println("Número menor: " numeroMenor);
7
8          //Conversão explicita
9          numeroMenor = byte(numeroMaior);
10         println("Número menor: " numeroMenor);
11 }
```

Resultado 44 – Resultado do Exemplo 65

```
Número maior: 8.0
Número menor: 3
Número menor: 8
```

No Exemplo 65, é feita a conversão do tipo *double* para *byte* na linha 9, um tipo de dado de tamanho maior para um tipo de dado de tamanho menor, desta forma foi necessário a conversão utilizando a função *byte*, esta operação é equivalente a conversão na seguinte forma na linguagem Java: *(byte) numeroMaior*.

Na linguagem Nice, o tipo de dado *char* não é um tipo de dado numérico, as operações com o tipo de dado *char* contém as funções da classe *java.lang.Character*. Para converter um valor do tipo de dado *char* para o código do tipo de dado *int*, é utilizada a função *int(caractere)*

e para converter um código do tipo de dado *int* para o caractere do tipo de dado *char*, é utilizado a função *char(inteiro)*. O Exemplo 66, apresenta o uso destas funções de conversão.

Exemplo 66: – Conversão do tipo *char* para o tipo *int* e vice-versa

```
1  void main(String[] args){
2          //Conversão de caractere para inteiro
3          char cA = 'A';
4          println("Caracter: " cA);
5
6          int iA = int(cA); //Conversão
7          println("Inteiro: " iA); //Imprime 65
8
9          //Conversão de inteiro para caractere
10         iA = 97; //Código do caractere 'a'
11         println("Inteiro: " iA);
12
13         cA = char(iA); //Conversão
14         println("Caracter: " cA);
15 }
```

Resultado 45 – Resultado do Exemplo 66

```
Caracter: A
Inteiro: 65
Inteiro: 97
Caracter: a
```

No Exemplo 66, o caractere 'A' é convertido para o respectivo inteiro na linha 6. O código inteiro que representa o caractere 'a' é o 97. Ele é atribuído à variável *iA*, na linha 10, e é convertido para o respectivo caractere na linha 13.

4.26 Conversão entre classes

A conversão entre classes é feita mostrando que o objeto em questão é da instância desejada. Para especificar que um objeto é da instância desejada, é utilizada a palavra-chave **instanceof** em uma condição. O Exemplo 67, apresenta a conversão entre classes na linha 13.

Exemplo 67: – Conversão de classes

```
1  interface Carro {
2          void acelarar() = println("Vrummmmmm");
3  }
4
5  class CarroCorrida implements Carro {
6          void turbina() = println("Tchiuuu");
7  }
8
9  void main(String[] args){
10         Carro carro = new CarroCorrida();
11         carro.acelarar();
12
13         if (carro instanceof CarroCorrida)
```

```
14                  carro.turbina();
15  }
```

Resultado 46 – Resultado do Exemplo 67

```
Vrummmmmm
Tchiuuu
```

No Exemplo 67, há uma classe chamada *CarroCorrida* na linha 5 e uma interface chamada *Carro* na linha 1. A classe *CarroCorrida* implementa a interface *Carro*. Assim, utiliza-se uma variável chamada *carro* com o tipo de dado da interface *Carro* para manipular um objeto em memória da classe *CarroCorrida* assim como especificado na linha 10. Na linha 14, pretende-se chamar uma função que somente instâncias da classe *CarroCorrida* possuem. Para isso, é necessário utilizar a palavra-chave **instanceof**, assim como demonstrado na linha 13, para especificar que a variável *carro* trata de uma instância da classe *CarroCorrida*.

4.27 Tipos de dados variáveis

Como descrito na subseção 4.12, a linguagem Nice também permite a definição de parâmetros genéricos para funções. A única diferença é que os parâmetros são especificados antes da definição do protótipo da função. O Exemplo 68, apresenta o uso de genéricos em funções, na linha 1.

Exemplo 68: – Tipos de dados variáveis

```
1  <T> void imprimir(T objeto){
2          println(objeto);
3  }
4
5  void main(String[] args){
6          imprimir(10);
7          imprimir("Ronneesley");
8          imprimir(2.3);
9  }
```

Resultado 47 – Resultado do Exemplo 68

```
10
Ronneesley
2.3
```

No Exemplo 68 o tipo de dado variável *T* do parâmetro *objeto* é declarado no início da definição do protótipo da função *imprimir* na linha 1. O exemplo a seguir demonstra uma aplicação desta funcionalidade.

Exemplo 69: – Aplicação de tipos de dados variáveis

```
1  <T> void imprimir(T[] vetor){
2          int contador = 0;
3
4          for (T item: vetor){
5                  if (contador++ != 0) print(", ");
```

```
6
7                   print(item);
8           }
9
10          println("");
11  }
12
13  void main(String[] args){
14          imprimir([1, 2, 3, 4, 5]);
15          imprimir(["A", "B", "C", "D", "E", "F"]);
16          imprimir([1.2, 3.5]);
17          imprimir([true, false, true]);
18          imprimir([1, "A", true]);
19  }
```

Resultado 48 – Resultado do Exemplo 69

```
1, 2, 3, 4, 5
A, B, C, D, E, F
1.2, 3.5
true, false, true
1, A, true
```

No Exemplo 69, a função *imprimir* na linha 1, recebe um vetor do tipo de dado variável *T* e imprime os itens separando-os com uma vírgula. As linhas 14 à 18 utilizam a função *imprimir*, com diferentes tipos de dados atribuídos à *T*. O Exemplo 70, mostra que é possível estabelecer relacionamentos entre os tipos de dados variáveis.

Exemplo 70: – Relações com tipos de dados variáveis

```
1   <Pessoa A, B | B <: A> void imprimir(A a, B b){
2           println("" a.nomeFormatado()
3                           " => " b.nomeFormatado());
4   }
5
6   class Pessoa {
7           private String nome;
8           public String nomeFormatado() = nome;
9   }
10
11  class Bacharel extends Pessoa {
12          public override nomeFormatado() = "Bel. " nome;
13  }
14
15  void main(String[] args){
16          var nome = "Ronneesley Moura Teles";
17
18          var pessoa = new Pessoa(nome:nome);
19          var bacharel = new Bacharel(nome:nome);
20
21          imprimir(pessoa, bacharel);
22  }
```

Resultado 49 – Resultado do Exemplo 70

```
Ronneesley Moura Teles => Bel. Ronneesley Moura Teles
```

O Exemplo 70, mostra que também é possível estabelecer relações para os tipos de dados variáveis pois, às vezes, é necessário invocar métodos específicos. Na linha 1, são definidos dois tipos de dados variáveis, o primeiro se chama *A* e se restringe à classe *Pessoa* e o segundo chamado de *B*, onde *B* deve ser um subtipo de dado de *A*. Esta declaração permite a chamada da função *nomeFormatado* (linha 3) para o parâmetro *b*. Desta forma, é necessário que o segundo parâmetro passado na chamada de função da linha 21 seja um subtipo do primeiro parâmetro, assim como acontece com as classes *Pessoa*, declarada na linha 6, e *Bacharel*, declarada na linha 11, esta herda da classe *Pessoa*. Desta forma, é satisfeita a relação entre os dois parâmetros instanciados nas linhas 18 e 19.

Uma detalhe importante é que parâmetros com tipos de dados variáveis podem ser nulos por padrão. O Exemplo 71, mostra que um tipo de dado variável pode receber o valor nulo.

Exemplo 71: – Tipo de dado variável que pode receber o valor nulo

```
1 <A> void escrever(A parametro){
2         println(parametro);
3 }
4
5 void main(String[] args){
6         ?String valor = null;
7         escrever(valor);
8 }
```

Resultado 50 – Resultado do Exemplo 71

```
null
```

No Exemplo 71, é definida uma variável chamada *valor* na linha 6 que pode e recebe o valor nulo (*null*). Em seguida, esta variável é informada como parâmetro para a função *escrever*. Desta forma, a variável *parametro* (linha 2) terá em seu conteúdo o valor nulo.

Para tratar estes casos, utiliza-se da sintaxe explicita de declaração de variáveis que não podem receber o valor nulo (*null*), como descrito na subseção 4.2. O Exemplo 72, mostra na linha 1, como não possibilitar que o valor nulo, seja passado como argumento para um tipo de dado variável.

Exemplo 72: – Tipo de dado variável que **não** pode receber o valor nulo

```
1  <A> void escrever(!A parametro){
2          println(parametro);
3  }
4
5  void main(String[] args){
6          String valor = "Ronneesley";
7          escrever(valor);
8
9          //A linha 10 resultaria em um erro de compilação
10         //escrever(null);
11 }
```

Resultado 51 – Resultado do Exemplo 72

Ronneesley

No Exmeplo 72, se a linha 10 não estivesse comentada, o compilador notificaria que o parâmetro não pode receber um valor nulo.

4.28 Parâmetros exatos

A linguagem Nice admite a definição de funções com parâmetros exatos, ou seja, caso o parâmetro seja uma classe, suas subclasses não utilizarão a mesma função definida. Assim, torna-se necessário implementar as funções para cada classe. Normalmente, utiliza-se este recurso quando os comportamentos são diferentes para cada classe; mas, deve-se implementar a função em todas. O Exemplo 73, usa de parâmetros exatos nas funções declaradas nas linhas 11 e 15.

Exemplo 73: – Parâmetros exatos

```
1  class Carro {
2          int quantidadeRodas = 4;
3
4          void turbinar();
5  }
6
7  class CarroCorrida extends Carro {
8          int quantidadeTurbinas = 10;
9  }
10
11 turbinar(#Carro carro){
12         println("Instalando turbinas");
13 }
14
15 turbinar(#CarroCorrida carro){
16         println("O carro já possui turbinas");
17 }
18
19 void main(String[] args){
20         var carro = new Carro();
21         carro.turbinar();
22
23         var carroCorrida = new CarroCorrida();
24         carroCorrida.turbinar();
25 }
```

Resultado 52 – Resultado do Exemplo 73

Instalando turbinas
O carro já possui turbinas

No Exemplo 73, a classe *Carro* é definida com uma função chamada *turbinar* na linha 4 que não é implementada no corpo da classe. Assim, torna-se necessária a implementação desta função para que as linhas 21 e 24 funcionem corretamente os tratamentos diferentes para

as subclasses da classe *Carro*. Desta forma, para especificar a função *turbinar* para objetos da classe *Carro* e *CarroCorrida* o parâmetro *carro* possui a notação: *#tipo-de-dado nome-parâmetro*, assim como utilizado nas linhas 11 e 15. Esta notação força que somente objetos do mesmo tipo de dado sejam aceitos. Assim, é necessário definir funções para cada classe relacionada com a classe *Carro*, da maneira como foi realizada nas funções declaradas nas linhas 11 e 15.

Este tipo de tratamento é particularmente observado em funções de duplicação de objetos. O Exemplo 74, apresenta duas funções, nas linhas 16 e 17 que duplicam um objeto.

Exemplo 74: – Parâmetros exatos com a palavra-chave **alike**

```
1  class Pessoa {
2    String nome;
3
4    override alike clone();
5
6    override toString() = "Nome: " nome;
7  }
8
9  class Administrador extends Pessoa {
10   int permissao;
11
12   override toString() = "Nome: " nome
13     ", Permissão: " permissao;
14 }
15
16 clone(#Pessoa pessoa) = new Pessoa(nome:pessoa.nome);
17 clone(#Administrador admin) =
18   new Administrador(nome:admin.nome,
19     permissao:admin.permissao);
20
21 void main(String[] args){
22   var pessoa = new Pessoa(nome:"Ronneesley Moura Teles");
23   var nPessoa = pessoa.clone();
24   nPessoa.nome = "Ronneesley";
25   println("Pessoa: " pessoa);
26   println("Nova Pessoa: " nPessoa);
27
28   //Clone do administrador
29   var adm = new Administrador(nome:"Ronneesley",
30     permissao:7);
31   var nAdmin = adm.clone();
32
33   nAdmin.permissao = 10;
34
35   println("Administrador " adm);
36   println("Novo Administrador " nAdmin);
37 }
```

Resultado 53 – Resultado do Exemplo 74

```
Pessoa: Nome: Ronneesley Moura Teles
Nova Pessoa: Nome: Ronneesley
Administrador Nome: Ronneesley, Permissão: 7
Novo Administrador Nome: Ronneesley, Permissão: 10
```

Na linha 4 do Exemplo 74, é utilizada a palavra-chave **alike**. Neste caso específico, esta palavra-chave significa que o retorno da função *clone*, definido na mesma linha, é da mesma classe do objeto que à invocou. Desta forma, são definidas as funções *clone* para as classes *Pessoa* e *Administrador*, nas linhas 16 e 17 respectivamente. Neste caso, para duplicar objetos, é necessário que o parâmetro e o retorno das funções *clone*, sejam do mesmo tipo de dado. Desta forma, é utilizada a notação *#tipo-de-dado nome-parâmetro* para a especificação destas funções.

Assim, é possível utilizar a função *clone* (linha 23) a partir de um objeto da classe *Pessoa* e obter como retorno um objeto da classe *Pessoa*, idêntico ao original. Também é possível utilizar a função *clone* na linha 31, a partir de um objeto da classe *Administrador*, e obter como retorno um objeto da classe *Administrador* idêntico ao original, em ambos os casos sem a necessidade de conversão do tipo de dado.

Outra utilidade para a palavra-chave **alike** é não repetir o nome da classe nos parâmetros. Desta forma, caso o nome da classe mude, as funções não precisarão serem modificadas.

Exemplo 75: – Outra utilidade da palavra chave **alike**

```
1  class Livro {
2    String nome;
3    float preco;
4
5    float diferencaPreco(alike objeto){
6      return abs(objeto.preco - this.preco);
7    }
8  }
9
10 void main(String[] args){
11   var livroNice = new Livro(nome:"Linguagem Nice", preco:100);
12   var livroJava = new Livro(nome:"Linguagem Java", preco:200);
13
14   println(livroNice.diferencaPreco(livroJava));
15 }
```

Resultado 54 – Resultado do Exemplo 75

```
100.0
```

No Exemplo 75, se o nome da classe *Livro* fosse modificado para *Book*, o corpo da classe não seria modificado, pois na função *diferencaPreco* não é utilizado o nome da classe como tipo de dado do parâmetro *objeto*, e sim a palavra-chave **alike**. Se houvesse tal alteração, ainda seria necessário modificar as linhas 11 e 12 para o novo nome.

4.29 Sobrescrita de funções com inferência de tipos dos parâmetros

Quando há sobrescrita de função e estas possuem parâmetros, não existe a necessidade de especificar os tipos de dados destes parâmetros, pois estes são inferidos automaticamente pela linguagem Nice, assim como acontece com o tipo de dado do retorno da função como descrito anteriormente na subseção 4.14. Desta forma, deve-se repetir os nomes dos parâmetros da função a ser sobrescrita.

No Exemplo 76, a definição da função *acelerar* na linha 10 da classe *CarroCorrida* não especifica o retorno e também não especifica o tipo de dado do parâmetro *quantidade*. Neste caso, é obrigatório que os nomes dos parâmetros sejam iguais aos nomes dos parâmetros da superclasse. Desta forma, o compilador consegue inferir os tipos de dados dos parâmetros e o tipo de dado do retorno automaticamente.

Exemplo 76: – Inferência nos tipos de dados dos parâmetros e retorno de uma função

```
1  class Carro {
2          int velocidade = 0;
3
4          void acelerar(int quantidade) {
5                  velocidade += quantidade;
6          }
7  }
8
9  class CarroCorrida extends Carro {
10         acelerar(quantidade){
11                 velocidade += 2 * quantidade;
12         }
13 }
14
15 void main(String[] args){
16         var carroCorrida = new CarroCorrida();
17         carroCorrida.acelerar(2);
18
19         println(carroCorrida.velocidade);
20 }
```

Resultado 55 – Resultado do Exemplo 76

```
4
```

Este recurso facilita a troca dos tipos de dados em apenas um ponto, ou seja, na classe *Carro* e também facilita a codificação.

4.30 Interfaces abstratas

Um recurso muito poderoso da linguagem Nice é chamado de "interfaces abstratas". Estas interfaces se comportam como as normais, já mencionadas na subseção 4.14, exceto por duas características:

1. Pode-se especificar que uma classe implementa uma interface abstrata **depois** desta classe ter sido definida; e

2. Uma interface abstrata **não** é um tipo de dado, logo não pode ser utilizado como tal.

Às vezes, pode-se encontrar classes que possuem funções com o mesmo protótipo, mas não implementam uma interface em comum. Assim, as interfaces abstratas, tornam possível a definição depois da declaração destas classes. O Exemplo 77, demonstra uma utilização das interfaces abstratas.

Exemplo 77: – Interface abstrata em classes nativas

```
1  abstract interface Autor {
2      void escreverNomeAutor() =
3          println("Ronneesley Moura Teles");
4  }
5
6  class java.lang.String implements Autor;
7  class nice.lang.int implements Autor;
8
9  void main(String[] args){
10     var literal = "Interface abstrata";
11     literal.escreverNomeAutor();
12
13     var numero = 10;
14     numero.escreverNomeAutor();
15 }
```

Resultado 56 – Resultado do Exemplo 77

```
Ronneesley Moura Teles
Ronneesley Moura Teles
```

No Exemplo 77, é definida uma interface abstrata chamada *Autor* com uma função chamada *escreverNomeAutor* (linhas 1 à 4), nas linhas 6 e 7 é definido que as classes *java.lang.String* e *nice.lang.int* implementam a interface abstrata *Autor*. Desta forma, estas classes irão possuir o método *escreverNomeAutor*. Poderiam ser definidos tratamentos diferentes para as implementações de *java.lang.String* e *nice.lang.int*. Para isto deve-se utilizar os conceitos definidos na subseção 4.13.

Como comentado, uma interface abstrata não pode ser utilizada como um tipo de dado. Logo, os exemplos 78 e 79 que utilizam o Exemplo 77 retornam um erro de compilação.

Exemplo 78: – Primeiro erro de utilização de interfaces abstratas

```
1  void chamar(Autor implementacaoAutor){
2      implementacaoAutor.escreverNomeAutor();
3  }
```

No Exemplo 78, o erro está em utilizar a interface abstrata *Autor* como o tipo de dado do parâmetro *implementacaoAutor* na linha 1.

Exemplo 79: – Segundo erro de utilização de interfaces abstratas

```
1  var Autor autor = literal;
```

No Exemplo 79, o erro está em utilizar a interface abstrata *Autor* como o tipo de dado da variável *autor*.

Devido a esta restrição, a forma correta de utilizar as classes *java.lang.String* e *nice.lang.int* do Exemplo 77 como parâmetro, sem a necessidade de especificar diretamente uma das classes, é utilizando parâmetros com tipos de dados variáveis, definidos na subseção 4.27. O Exemplo 80, apresenta a utilização correta das interfaces abstratas.

Exemplo 80: – Utilização correta de interfaces abstratas

```
1 <Autor T> void chamar(T implementacaoAutor){
2     implementacaoAutor.escreverNomeAutor();
3 }
```

No Exemplo 80, a interface abstrata *Autor* é utilizada como tipo de dado variável da função *chamar* definida na linha 1. Deste modo, é possível chamar a função *escreverNomeAutor* para o parâmetro *implementacaoAutor* que utiliza este tipo de dado variável.

5 Desenvolvimento de aplicações

Além da linguagem Nice fornecer uma maior expressividade é necessário verificar o impacto destes recursos em aplicações reais, para isso será apresentada uma aplicação com janelas, uma aplicação Web e uma aplicação para dispositivos móveis.

5.1 Utilizando a linguagem Nice com banco de dados

Uma das ferramentas mais utilizadas atualmente são os Sistema Gerenciador de Banco de Dados (SGBD). Esta subseção apresenta em Nice uma aplicação que utiliza o MySQL.

Neste exemplo, são utilizados dois padrões de projetos, são eles: Data Transfer Object (DTO) e o Data Access Object (DAO). O Exemplo 81, apresenta a classe *Pessoa* que representa o DTO de uma pessoa.

Exemplo 81: – Classe DTO de uma pessoa

```
1 class Pessoa {
2         private ?int id = null;
3         private String nome;
4 }
```

No Exemplo 81, a classe *Pessoa* definida na linha 1, possui o atributo *id* (linha 2) que representa um número único para uma pessoa e o atributo *nome* (linha 3). Uma característica da linguagem Nice é que ela cria automaticamente o encapsulamento dos atributos de uma classe. Os dados de instâncias da classe *Pessoa* serão armazenados na tabela *pessoas* do banco de dados criado pela Structured Query Language (SQL) do Exemplo 82.

Exemplo 82: – Script SQL do banco de dados para o MySQL

```
1 create database banco;
2
3 use banco;
4
5 create table pessoas (
6         id integer auto_increment,
7         nome varchar(100) not null,
8         primary key (id)
9 ) type=InnoDB;
```

Para inserir os dados da entidade *Pessoa* é utilizada uma classe específica chamada de *pessoaDAO*, que representa o DAO de uma pessoa.

Exemplo 83: – Classe DAO de uma pessoa

```
1  import java.util.*;
2  import java.sql.*;
3
4  class PessoaDAO {
5    public void inserir(Pessoa dto){
6      //Solicita o driver do banco de dados
7      Class.forName("com.mysql.jdbc.Driver");
8
9      // Conecta com o banco de dados: banco, no servidor
10     //localhost na porta 3306, com o usuário e senha
11     var con = DriverManager.getConnection(
12       "jdbc:mysql://localhost:3306/banco", "usuário",
13       "senha");
14
15     //Prepara a instrução SQL
16     var stmt = con.prepareStatement(
17       "insert into pessoas (nome) values(?)");
18
19     stmt.setString(1, dto.nome);
20
21     //Executa o SQL
22     stmt.execute();
23   }
24 }
```

No Exemplo 83, são utilizadas as classes da biblioteca Java Database Connectivity (JDBC) para a conexão com o banco de dados. A classe principal da aplicação é apresentada no Exemplo 84.

Exemplo 84: – Classe principal

```
1  void main(String[] args){
2    println("JDBC Nice");
3
4    var pessoaDAO = new PessoaDAO();
5    pessoaDAO.inserir(
6      new Pessoa(nome:"Ronneesley Moura Teles"));
7  }
```

Com base nos exemplos 81 e 83, pôde-se notar que as principais características da linguagem Nice utilizadas para o desenvolvimento destes códigos são:

- A inferência de tipos das variáveis;
- Encapsulamento automático de atributos de classe;

5.2 Desenvolvimento de aplicações com janelas

Atualmente é muito utilizada a biblioteca Swing, contida no pacote *javax.swing* do Java, para desenvolvimento de aplicativos com janelas. Da mesma forma será utilizado a biblioteca

Swing para verificar o impacto na escrita de aplicativos desenvolvidos na linguagem Nice. O exemplo a seguir mostra como criar uma janela utilizando a linguagem Nice.

Exemplo 85: – Janela utilizando a biblioteca Swing

```
1  import javax.swing.*;
2
3  class Janela extends JFrame {
4    void mostrar(){
5      this.setTitle("Janela 1");
6      this.setSize(300, 200);
7      this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
8      this.centralizar();
9      this.setVisible(true);
10   }
11 }
```

Na linha 1, é feita a importação do pacote *javax.swing*, na linha 3 é declarada a classe *Janela* que herda da classe *JFrame*, responsável pelas características da janela e na função *mostrar*, são configuradas as características principais da janela, respectivamente: título, tamanho e operação padrão ao fechar. Na linha 8, é utilizada a função *centralizar*, definida no Exemplo 86, e, finalmente, na linha 9, a janela se torna visível.

Exemplo 86: – Utilitários para as janelas Swing

```
1  void centralizar(JFrame janela){
2    var ferramenta = Toolkit.getDefaultToolkit();
3    var tela = ferramenta.getScreenSize();
4    janela.setLocation(
5      int((tela.getWidth() - janela.getWidth()) / 2),
6      int((tela.getHeight() - janela.getHeight()) / 2)
7    );
8  }
```

A característica multi-métodos da linguagem Nice permite adicionar o método *centralizar* na classe *JFrame* do pacote *javax.swing*. O resultado desta do Exemplo 85 é apresentado na Figura 11.

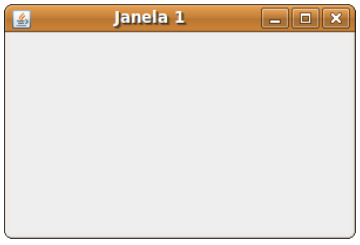

Figura 11: Janela do Exemplo 85

O Exemplo 87, demonstra como fazer uma interface utilizando a biblioteca Swing, com o intuito de fazer uma janela para calcular o quadrado de um número.

Exemplo 87: – Janela para cálculo do quadrado de um número

```
1  import javax.swing.*;
2  import java.awt.*;
3  import java.awt.event.*;
4
5  class JanelaCalculoQuadrado extends JFrame
6    implements ActionListener {
7
8    JButton btnCalcular = new JButton("Calcular");
9    JTextField txtNumero = new JTextField();
10   JLabel lblResposta = new JLabel("", JLabel.CENTER);
11
12   void mostrar(){
13     this.setTitle("Cálculo Quadrado");
14     this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
15     this.setSize(440, 180);
16     this.centralizar();
17
18     var lblNumero = new JLabel("Número");
19     var pnl = this.getContentPane();
20     pnl.setLayout(null);
21
22     btnCalcular.addActionListener(this);
23     btnCalcular.setActionCommand("calcular");
24
25     lblNumero.setBounds(10, 10, 100, 30);
26     txtNumero.setBounds(120, 10, 300, 30);
27     lblResposta.setBounds(10, 90, 420, 30);
28     btnCalcular.setBounds(320, 50, 100, 30);
29
30     pnl.add(lblNumero);
31     pnl.add(lblResposta);
32     pnl.add(btnCalcular);
33     pnl.add(txtNumero);
34
35     this.setVisible(true);
36   }
37
38   actionPerformed(evt){
39     if (evt != null){
40       var comando = evt.getActionCommand();
41
42       if (comando.equals("calcular")){
43         try {
44           var numero = Integer.parseInt(
45             this.txtNumero.getText());
46
47           lblResposta.setText("" + (numero ** 2));
48         } catch (NumberFormatException ex){
49           JOptionPane.showMessageDialog(null,
50             "Número mal formatado",
```

```
51                  "Validação", JOptionPane.WARNING_MESSAGE);
52              }
53          }
54      }
55  }
56  }
```

Nas linhas 8 à 10 são definidos os componentes: botão, caixa de texto e a marcação para a resposta do cálculo.

A função *mostrar* configura os componentes da interface, respectivamente o título na linha 13, a operação padrão de fechamento na linha 14, o tamanho da janela na linha 15, chamada da função centralizar, definida no Exemplo 86, na linha 16, definição da marcação explicativa do campo de texto na linha 18, painel da janela na linha 19, configuração do layout do painel na linha 20, nas linhas 22 e 23 são configuradas as propriedades do evento de clique do botão, nas linhas 25 à 28 são configuradas as posições dos componentes, nas linhas 30 à 33 os componentes são adicionados no painel e finalmente na linha 35 a janela se torna visível.

A função *actionPerformed*, definida nas linhas 38 à 53, é implementada por causa da interface *ActionListener*, definida na linha 6, na linha 42 é verificado se o nome do comando é *calcular*, que foi atribuído ao botão na linha 23, desta forma calcula o quadrado do conteúdo da caixa de texto na linha 44 e 46.

O resultado do Exemplo 87 é apresentado na Figura 12.

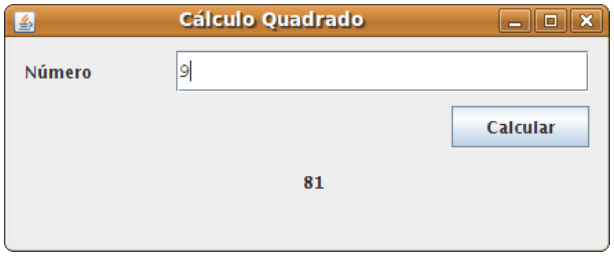

Figura 12: Janela do Exemplo 87

Com base nos exemplos 85 e 87, pôde-se notar que as principais características da linguagem Nice, utilizadas para o desenvolvimento de aplicativos com janelas são:

- A inferência de tipos das variáveis;
- A inferência de tipos nas funções herdadas ou implementadas devido as interfaces; e
- A possibilidade de utilizar multi-métodos para criação de novas funções para algumas classes.

5.3 Desenvolvimento de Servlets

Os Servlets permitem o desenvolvimento de aplicativos para a plataforma Java, visando a interface com servidores Web. Com o grande crescimento das aplicações Web, a utilização da plataforma Java se tornou uma alternativa muito boa, pois não é necessário reescrever todos os códigos para o desenvolvimento de um sistema Web, caso já exista a aplicação escrita em Java. Assim, esta subseção se dedica ao estudo do impacto na escrita destes aplicativos na linguagem Nice.

O Exemplo 88 apresenta a escrita de um Servlet na linguagem Nice para exibir uma mensagem de texto.

Exemplo 88: – Servlet que retorna uma mensagem

```
1  import javax.servlet.http.*;
2
3  class MeuServlet extends HttpServlet {
4    doPost(request, response) = this.doGet(request, response);
5
6    doGet(request, response){
7      if (response != null){
8        var escritor = response.getWriter();
9        escritor.write("Linguagem Nice com Servlet");
10     }
11   }
12 }
```

Na linha 1, é feita a importação dos pacotes necessários para implementação do Servlet, na linha 3 é definida a classe *MeuServlet*, na linha 4 é implementada a função *doPost* que realiza as mesmas operações da função *doGet*, na linha 6 é especificada a função *doGet*, nesta função é verificado se o parâmetro *response* é diferente de nulo na linha 7, caso seja, é escrito a mensagem *"Linguagem Nice com Servlet"* nas linhas 8 e 9. No Exemplo 88, as funções *doPost* e *doGet* são herdadas da classe *HttpServlet* definida na linha 3.

Como ainda não há um IDE para edição destes tipos de projetos na linguagem Nice, para testar, de forma mais simples, este exemplo, pode-se criar um projeto em Java para Web utilizando o Eclipse. Após a criação, deve-se adicionar o arquivo JAR do exemplo como uma dependência do projeto criado. Em seguida, é necessário adicionar o conteúdo do Exemplo 89 ao arquivo *web.xml* da pasta *WebContent/WEB-INF* do projeto, para registrar o Servlet escrito na linguagem Nice.

Exemplo 89: – Conteúdo a ser adicionado no arquivo *web.xml*

```
1  <servlet>
2    <description></description>
3    <display-name>MeuServlet</display-name>
4    <servlet-name>MeuServlet</servlet-name>
5    <servlet-class>meupacote.MeuServlet</servlet-class>
6  </servlet>
7  <servlet-mapping>
8    <servlet-name>MeuServlet</servlet-name>
9    <url-pattern>/MeuServlet</url-pattern>
10 </servlet-mapping>
```

O conteúdo deste exemplo é adicionado, da mesma forma que um Servlet escrito na linguagem Java. Pode-se notar que não há diferença, pois trata-se de *bytecode* e não dos códigos fontes.

O Exemplo 90, apresenta a escrita de um Servlet na linguagem Nice para calcular o quadrado de um número informado como parâmetro.

Exemplo 90: – Servlet para calcular o quadrado de um número

```
1  import javax.servlet.http.*;
2
3  class ServletCalcularQuadrado extends HttpServlet {
4    doPost(request, response) = this.doGet(request, response);
5
```

```
6    doGet(request, response){
7      if (response != null && request != null){
8        response.setContentType("text/html;charset=UTF-8");
9        var escritor = response.getWriter();
10
11       try {
12         var numero = Integer.parseInt(
13           request.getParameter("numero"));
14
15         escritor.write("O quadrado de " numero " é "
16           + (numero ** 2));
17       } catch (Exception ex){
18         escritor.write("Informe o parâmetro número");
19       }
20     }
21   }
22 }
```

Na linha 7, é verificado se os dois parâmetros *request* e *response* são diferentes de nulo. Na linha 8, é especificado a codificação de retorno. Nas linhas 12 e 13 é convertido o parâmetro chamado de *numero* para inteiro e, finalmente, nas linhas 15 e 16 é escrito quadrado do parâmetro *numero*. No Exemplo 90, é necessário adicionar uma bloco *try* definido nas linhas 11 a 17 para tratar o parâmetro *numero*, caso este parâmetro não esteja no formato correto ou não seja informado, será tratado no bloco *catch* especificado nas linhas 17 a 19.

Da mesma forma como no Exemplo 88, é necessário adicionar o conteúdo do Exemplo 91 ao arquivo *web.xml* da pasta *WebContent/WEB-INF* do projeto.

Exemplo 91: – Conteúdo a ser adicionado no arquivo *web.xml*

```
1  <servlet>
2    <description></description>
3    <display-name>ServletCalcularQuadrado</display-name>
4    <servlet-name>ServletCalcularQuadrado</servlet-name>
5    <servlet-class>meupacote.ServletCalcularQuadrado
6  </servlet-class>
7  </servlet>
8  <servlet-mapping>
9    <servlet-name>ServletCalcularQuadrado</servlet-name>
10   <url-pattern>/ServletCalcularQuadrado</url-pattern>
11 </servlet-mapping>
```

Com base nos exemplos 88 e 90, pôde-se notar que as principais características da linguagem Nice utilizadas para o desenvolvimento de Servlets são:

- A inferência de tipos das variáveis; e
- A inferência de tipos nas funções herdadas ou implementadas devido as interfaces;

5.4 Desenvolvimento de Applets

Applets permitem o desenvolvimento de aplicativos para Web que serão executados através do navegador. Um dos pontos bem explorados no desenvolvimento de aplicativos com

a linguagem Nice são os Applets. Esta subseção se dedica ao estudo do impacto da escrita de Applets na linguagem Nice.

O exemplo a seguir demonstra um Applet escrito na linguagem Nice para escrever uma mensagem.

Exemplo 92: – Applet para escrita de uma mensagem

```
1  import javax.swing.*;
2
3  class MeuApplet extends JApplet {
4    init(){
5      var painel = this.getContentPane();
6      painel.add(
7        new JLabel(
8          "Desenvolvimento de Applet com a linguagem Nice"));
9    }
10 }
```

Na linha 1, são importadas as classes necessárias para o funcionamento do Applet. Na linha 3, é declarada a classe *MeuApplet*. A função *init* é definida na linha 4, dentro dela é definida a variável *painel* (linha 5) e adicionada a mensagem *"Desenvolvimento de Applet com a linguagem Nice"*. Para mostrar o Applet é necessário criar um arquivo no formato HyperText Markup Language (HTML) com o conteúdo do Exemplo 93.

Exemplo 93: – HTML para mostrar o Applet do Exemplo 92

```
1  <!DOCTYPE HTML PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
2   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
3  <html>
4    <head>
5      <title>Applet em Nice</title>
6    </head>
7
8    <body>
9      <h2>Applet em Nice</h2>
10     <applet code="meupacote.MeuApplet.class"
11       archive="MeuApplet.jar"
12       height="50"
13       width="400">
14     </applet>
15   </body>
16 </html>
```

O exemplo a seguir demonstra a escrita de um Applet na linguagem Nice para calcular o quadrado de um número.

Exemplo 94: – Applet para calcular o quadrado de um número

```
1  import javax.swing.*;
2  import java.awt.event.*;
3
4  class MeuAppletCalcularQuadrado extends JApplet
5    implements ActionListener {
6
7    JButton btnCalcular = new JButton("Calcular");
```

```
8    JTextField txtNumero = new JTextField();
9    JLabel lblResposta = new JLabel("", JLabel.CENTER);
10
11   init(){
12     var lblNumero = new JLabel("Número");
13     var pnl = this.getContentPane();
14     pnl.setLayout(null);
15
16     btnCalcular.addActionListener(this);
17     btnCalcular.setActionCommand("calcular");
18
19     lblNumero.setBounds(10, 10, 100, 30);
20     txtNumero.setBounds(120, 10, 300, 30);
21     btnCalcular.setBounds(320, 50, 100, 30);
22     lblResposta.setBounds(10, 90, 420, 30);
23
24     pnl.add(lblNumero);
25     pnl.add(lblResposta);
26     pnl.add(btnCalcular);
27     pnl.add(txtNumero);
28
29     this.repaint();
30   }
31
32   actionPerformed(evt){
33     if (evt != null){
34       var comando = evt.getActionCommand();
35
36       if (comando.equals("calcular")){
37         try {
38           var numero = Integer.parseInt(
39             this.txtNumero.getText());
40
41           lblResposta.setText("" + (numero ** 2));
42         } catch (NumberFormatException ex){
43           JOptionPane.showMessageDialog(null,
44             "Número mal formatado",
45             "Validação", JOptionPane.WARNING_MESSAGE);
46         }
47       }
48     }
49   }
50 }
```

O Exemplo 94, é similar ao Exemplo 87, exceto na linha 11 onde define a função *init* herdada da classe *JApplet*, definida na linha 4 e, na linha 29, onde é solicitado o redesenho do Applet. Da mesma forma como no Exemplo 92, para visualizar o resultado é necessário criar um arquivo no formato HTML com o conteúdo do exemplo a seguir:

Exemplo 95: – HTML para mostrar o Applet do Exemplo 94

```
1 <!DOCTYPE HTML PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
2  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
```

```
3  <html>
4    <head>
5      <title>Applet em Nice para calcular o quadrado
6        de um n&uacute;mero</title>
7    </head>
8
9    <body>
10     <h2>Applet em Nice para calcular o quadrado
11       de um n&uacute;mero</h2>
12
13     <applet code="meupacote.MeuAppletCalcularQuadrado.class"
14       archive="MeuApplet.jar"
15       height="180"
16       width="440">
17     </applet>
18   </body>
19 </html>
```

5.5 Desenvolvimento de MIDlets

MIDlets são aplicativos desenvolvidos para dispositivos móveis. Atualmente, o aumento da performance e capacidade dos dispositivos móveis veem atraindo o desenvolvimento de muitas aplicações, desta forma esta subseção se dedica ao estudo do impacto da escrita de MIDlets na linguagem Nice.

O exemplo a seguir demonstra um MIDlet escrito na linguagem Nice para mostrar uma mensagem de alerta na tela de um dispositivo móvel.

Exemplo 96: – MIDlet escrito na linguagem Nice

```
1  import javax.microedition.midlet.*;
2  import javax.microedition.lcdui.*;
3
4  class MidletNice extends MIDlet {
5    startApp(){
6      var alerta = new Alert("Alerta",
7        "Meu primeiro midlet em Java",
8        null, AlertType.INFO);
9      Display.getDisplay(this).setCurrent(alerta);
10   }
11
12   pauseApp(){}
13   destroyApp(unconditional){}
14 }
```

Nas linhas 1 e 2, são importadas as classes necessárias para o funcionamento do MIDlet. Na linha 4, é definido a classe *MidletNice*. A função *startApp* é definida na linha 5. Na linha 6, é instanciada a mensagem de alerta e, na linha 9, é dado o foco na mensagem de alerta. A função *pauseApp* (linha 12) e *destroyApp* (linha 13) não foram implementadas neste exemplo. As funções *startApp*, *pauseApp* e *destroyApp* são herdadas da classe *MIDlet* definida na linha 4.

Da mesma forma como os Servlets, não há um IDE para criar projetos para dispositivos móveis. Assim, pode-se criar um projeto em um IDE como o Netbeans e adicionar a dependência ao arquivo JAR do exemplo.

O desenvolvimento de aplicativos para dispositivos móveis utiliza poucos recursos da plataforma Java tendo em vista que não possuem a mesma capacidade e performance de um computador pessoal. Devido a este motivo, os recursos da máquina virtual são diminuídos. Este fato faz com que os aplicativos desenvolvidos na linguagem Nice **não** funcionem para estes dispositivos, pois ainda há a necessidade de diminuir os recursos utilizados pela linguagem Nice.

Mesmo **não** sendo possível o desenvolvimento de MIDlets com a linguagem Nice, é possível estimar o impacto com base no Exemplo 96, desta forma as principais características utilizadas seriam:

- A inferência de tipos das variáveis; e
- A inferência de tipos nas funções herdadas ou implementadas devido as interfaces;

Neste ponto, é interessante ressaltar que atualmente os dispositivos móveis necessitam aplicações pequenas. Assim, o desenvolvimento de aplicativos na linguagem Nice para dispositivos móveis deve ser estudado em detalhe, pois se deve destacar os pontos positivos e negativos tendo em vista as restrições de tais dispositivos. Este estudo só poderá ser realizado depois da diminuição dos recursos utilizados pela linguagem Nice.

6 Sugestões de melhoria

Apesar de ser uma linguagem com muitas vantagens, esta seção se dedica a sugestões de melhorias futuras para a linguagem Nice.

6.1 Laço *for* compacto decrescente

Apesar do laço *for* possuir um forma compacta de escrita que utiliza instâncias da classe *Range*, o compilador não consegue entender corretamente expressões decrescentes, assim como apresentado no Exemplo 97.

Exemplo 97: – Problema laço *for* com *Range* decrescente

```
1 for (int i: 10..1){
2         println("Contador: " i);
3 }
```

6.2 Enumerações

A compatibilidade com enumerações não é totalmente garantida para a utilização em Java, pois uma enumeração é tratada de maneira diferente na linguagem Nice. O Exemplo 98, apresenta uma enumeração escrita na linguagem Nice.

Exemplo 98: – Enumeração definida na linguagem Nice

```
1 enum Sexo {
2         MASCULINO, FEMININO
3 }
```

No Exemplo 98, é definida uma enumeração chamada *Sexo*, que possui dois possíveis valores, são eles: *MASCULINO* e *FEMININO*. Desta forma, para ser totalmente compatível com a linguagem Java, deve ser possível utilizar esta enumeração sem perceber nenhuma diferença. Assim, o Exemplo 99, que é uma tentativa de utilizar a enumeração definida no Exemplo 98 em um código escrito na linguagem Java.

Exemplo 99: – Tentativa de utilizar a enumeração *Sexo* na linguagem Java

```
1 package br;
2
3 public class TesteSexo {
4         public static void main(String[] args) {
5                 //Sexo.MASCULINO;
6         }
7 }
```

Se a linha 5 não estivesse comentada seria retornado uma mensagem de erro, pois não há o valor *MASCULINO* na enumeração *Sexo* quando utilizada na linguagem Java.

6.3 Switch

A linguagem Nice não possui o comando escolhas **switch**, muito utilizado nas outras linguagens como C, C++, PHP e Java. Devido a este problema o exemplo a seguir não será compilado.

Exemplo 100: – Problema do comando *switch*

```
1  void main(String[] args){
2          println("Teste");
3
4          var valor = 1;
5
6          switch (valor){
7                  case 1:
8                          println("Valor vale 1");
9                          break;
10         }
11 }
```

A solução para o problema evidenciado no Exemplo 100 é feita no Exemplo 101. Apesar de solucionar o problema, não é uma solução simples, pois necessita declarar uma função fora do bloco de código.

Como acontece na linha 7, do Exemplo 101, e utilizar as características discutidas na subseção 4.16 para o tratamento de cada caso.

Exemplo 101: – Solução alternativa *switch*

```
1 void main(String[] args){
2   int valor = 2;
3
4   testar(valor);
5 }
6
7 void testar(int numero)
```

```
8  = println("O número não foi tratado");
9
10 testar(1) = println("Primeiro");
11 testar(2) = println("Segundo");
12 testar(3) = println("Terceiro");
```

Desta forma, pode-se considerar que a solução proposta não se adequa à uma linguagem moderna, esta solução cabe apenas como uma solução alternativa.

6.4 Funções de tratamento obrigatório

A linguagem Nice não permite a criação de funções que devem ser tratadas pelo código que à invoca. Tal recurso é muito utilizado na linguagem Java com a palavra reservada **throws**. Na linguagem Nice, por mais que uma função retorne um erro, não é obrigatório o tratamento pelo código invoca. O Exemplo 102, apresenta na linha 1, uma função que retorna uma excessão, porém não é obrigatório o tratamento, na linha 11.

Exemplo 102: – Problema do comando *throws*

```
1  void comando() //Não existe, throws Exception
2  {
3          throw new Exception("Problema");
4  }
5
6
7  void main(String[] args){
8          // Pode ocorrer um erro, e não há maneira
9          //de forçar o código que invoca o método
10         //"comando" de colocar um bloco try
11         comando();
12 }
```

6.5 Desenvolvimento de uma biblioteca reduzida

Tendo em vista os resultados obtidos na subseção 5.5, há a necessidade do desenvolvimento de uma biblioteca reduzida para a linguagem Nice que permita o desenvolvimento de aplicações para dispositivos móveis.

6.6 Manipulação dos objetos da classe *java.util.Map*

A grande maioria das linguagens permitem a manipulação de vetores com chaves de forma fácil. Assim, a linguagem Nice deve facilitar a manipulação deste tipo de dado. O Exemplo 103, apresenta como fazer a manipulação destes objetos na linguagem PHP.

Exemplo 103: – Manipulação de vetores com chaves na linguagem PHP

```
1  <?php
2  $vetor = array( "nome"=>"Ronneesley",
3                  "idade"=>21);
4
5  $vetor["sobreNome"] = "Moura Teles";
6
```

```
7  echo "Nome: "    . $vetor["nome"] . " "
8                   . $vetor["sobreNome"]. nl2br("\n");
9
10 echo "Idade: " . $vetor["idade"] . nl2br("\n");
11 ?>
```

O Exemplo 104, apresenta como fazer a manipulação destes objetos na linguagem Ruby.

Exemplo 104: – Manipulação de vetores com chaves na linguagem Ruby

```
1 vetor = {"nome"=>"Ronneesley", "idade"=>21}
2
3 vetor["sobreNome"] = "Moura Teles"
4
5 puts "Nome: " + vetor["nome"] + " " +
6                 vetor["sobreNome"] + "\n"
7
8 puts "Idade: " + vetor["idade"].to_s + "\n"
```

O Exemplo 105, apresenta como fazer a manipulação destes objetos na linguagem Nice.

Exemplo 105: – Manipulação de vetores com chaves na linguagem Nice

```
1  void main(String[] args){
2          var Map<String, Object> vetor = new HashMap();
3
4          vetor.put("nome", "Ronneesley");
5          vetor.put("idade", 21);
6          vetor.put("sobreNome", "Moura Teles");
7
8          println("Nome: " vetor.get("nome")
9                  " " vetor.get("sobreNome"));
10
11         println("Idade: " vetor.get("idade"));
12 }
```

Comparando os exemplos 103, 104 e 105, pode-se concluir que as linguagens PHP e Ruby possuem formas mais ágeis para a definição inicial, escrita e leitura de valores do que a linguagem Nice.

6.7 Inferência de tipo de dado para conjunto de valores

Da mesma forma como realizado com vetores, verificar a possibilidade de inferência dos tipos de dados para um conjunto de valores (Tuples), de forma parecida como especificado no Exemplo 106:[2]

Exemplo 106: – Sugestão para inferência de tipo de dado de conjunto de valores

```
1 void main(String[] args){
2         (idade, nome, sobreNome) =
3                 (21, "Ronneesley", "Moura Teles");
4
5         println("Nome: " nome " " sobreNome);
```

[2]Este exemplo não funciona atualmente na linguagem Nice

```
6           println("Idade: " idade);
7   }
```

6.8 Inferência de tipo de dado dos retornos de funções

Da mesma forma como realizado com funções anônimas, sugere-se verificar a possibilidade de inferência do tipo de dado de retorno de uma função convencional. O Exemplo 107, apresenta a inferência do tipo de dado do retorno de uma função anônima.

Exemplo 107: – Sugestão para inferência de tipo de dado de retorno de função

```
1   void main(String[] args){
2           var funcao = () => {
3                   println("Teste de função");
4
5                   return "Ronneesley " + (1 + 2);
6           };
7
8           var valor = funcao();
9
10          println(valor.getClass());
11  }
```

6.9 Anotações

Anotações são muito utilizadas na linguagem Java e permitem a descrição de entidades, atributos e funções. Atualmente, um arcabouço muito utilizado na linguagem Java é o JPA (Java Persistence API), um dos meios de utilizar este arcabouço é através do uso de anotações. Como a linguagem Nice não dá suporte a esta característica, impossibilita o uso de algumas bibliotecas. Este assunto pode ser aprofundado em [12].

6.10 Necessita de um IDE

Apesar de não ser uma questão especificamente da linguagem, a linguagem Nice ainda não possui um IDE (*Integrated Development Environment*) [14] consolidado para edição de programas comparado ao NetBeans [8] ou ao Eclipse para facilitar a manipulação dos projetos. Como foi comentado anteriormente, já está em fase de desenvolvimento a extensão para o funcionamento no Eclipse, logo a questão do IDE não será um problema tão grave futuramente.

7 Considerações finais

A linguagem Nice facilita a construção de programas gerais, janelas, Servlets e Applets para a plataforma Java. Ainda não há suporte para o desenvolvimento de MIDlets.

Suas principais vantagens estão na sintaxe moderna, na inferência de tipos de dados, na possibilidade de funções como argumentos de outras funções, nos multi-métodos, nos conjuntos de valores, nas funções anônimas, no tratamento de erros de conversões e ponteiros nulos, na facilidade de compilação, na implementação de Padrão por Contrato nativamente, no novo recurso chamado "interface abstrata" e na compatibilidade com a linguagem Java.

8 Agradecimento

Ao Prof. Dr. Vinícius Sebba Patto pela avaliação do presente texto e pelas sugestões feitas, as quais muito contribuíram para a melhoria do texto original.

Referências

[1] **Eclipse.org home**. `http://www.eclipse.org/`, último acesso em Maio de 2009, 2009.

[2] **GNU Emacs - GNU Project - Free Software Foundation (FSF)**. `http://www.gnu.org/software/emacs/`, último acesso em Junho de 2009, 2009.

[3] **javacc: JavaCC Home**. `https://javacc.dev.java.net/`, último acesso em Maio de 2009, 2009.

[4] **jEdit - Programmer's Text Editor - overview**. `http://jedit.org/`, último acesso em Maio de 2009, 2009.

[5] **Linguagem de Programação Ruby**. `http://www.ruby-lang.org/pt/`, último acesso em Maio de 2009, 2009.

[6] **PHP: Hypertext Preprocessor**. `http://www.php.net/`, último acesso em Maio de 2009, 2009.

[7] **Python Programming Language – Official Website**. `http://www.python.org/`, último acesso em Maio de 2009, 2009.

[8] **Welcome to NetBeans**. `http://www.netbeans.org/`, último acesso em Maio de 2009, 2009.

[9] **Xoltar**. `www.xoltar.org`, último acesso em Agosto de 2009, 2009.

[10] DANIEL BONNIOT, B. K; BARBER, F. **The Nice user's manual**, 2003.

[11] MEYER, B. **Object-Oriented Software Construction**. Prentice Hall PTR, 2nd edition, 2000.

[12] SUN MICROSYSTEMS, I. **Annotations**. `http://java.sun.com/j2se/1.5.0/docs/guide/language/annotations.html`, último acesso em Junho de 2009, 2004.

[13] TIOBE. **TIOBE Programming Community Index for March 2009**. `www.tiobe.com`, último acesso em Abril de 2009, 2009.

[14] WIKIPÉDIA. **Ambiente de desenvolvimento integrado**. `http://pt.wikipedia.org/wiki/Ambiente_de_Desenvolvimento_Integrado`, último acesso em Março de 2009, 2009.

[15] WIKIPÉDIA. **Haskell (linguagem de programação)**. `http://pt.wikipedia.org/wiki/Haskell_(linguagem_de_programação)`, último acesso em Agosto de 2009, 2009.

[16] WIKIPÉDIA. **Java Archive**. http://pt.wikipedia.org/wiki/Java_Archive, último acesso em Junho de 2009, 2009.

[17] WIKIPÉDIA. **ML (linguagem de programação)**. http://pt.wikipedia.org/wiki/ML_(linguagem_de_programação), último acesso em Agosto de 2009, 2009.

[18] WIKIPÉDIA. **Nice**. http://pt.wikipedia.org/wiki/Nice, último acesso em Maio de 2009, 2009.

[19] WIKIPÉDIA. **Niké**. http://pt.wikipedia.org/wiki/Niké_(mitologia_grega), último acesso em Agosto de 2009, 2009.

[20] WIKIPÉDIA. **Programa Olá Mundo**. http://pt.wikipedia.org/wiki/Programa_Olá_Mundo, último acesso em Abril de 2009, 2009.

[21] WIKIPÉDIA. **Prolog**. http://pt.wikipedia.org/wiki/Prolog, último acesso em Abril de 2009, 2009.

[22] WIKIPÉDIA. **Realce de sintaxe**. http://pt.wikipedia.org/wiki/Realce_de_sintaxe, último acesso em Março de 2009, 2009.

[23] WIKIPÉDIA. **Sun Microsystems**. http://pt.wikipedia.org/wiki/Sun_Microsystems, último acesso em Maio de 2009, 2009.

[24] WIKIPÉDIA. **Tag (programação)**. http://pt.wikipedia.org/wiki/Tag_(programação), último acesso em Março de 2009, 2009.