LINGUAGEM DE PROGRAMAÇÃO II

AUTORES
Fábio José Parreira
Teresinha Letícia da Silva
Cristiano Bertolini
Guilherme Bernardino da Cunha

LICENCIATURA EM COMPUTAÇÃO

LINGUAGEM DE PROGRAMAÇÃO II

AUTORES

Fábio José Parreira

Teresinha Letícia da Silva

Cristiano Bertolini

Guilherme Bernardino da Cunha

1ª Edição

UAB/NTE/UFSM

UNIVERSIDADE FEDERAL DE SANTA MARIA

Santa Maria | RS

2019

©Núcleo de Tecnologia Educacional – NTE.
Este caderno foi elaborado pelo Núcleo de Tecnologia Educacional da Universidade Federal de Santa Maria para os cursos da UAB.

PRESIDENTE DA REPÚBLICA FEDERATIVA DO BRASIL
Jair Messias Bolsonaro

MINISTRO DA EDUCAÇÃO
Abraham Weintraub

PRESIDENTE DA CAPES
Anderson Ribeiro Correia

UNIVERSIDADE FEDERAL DE SANTA MARIA

REITOR
Paulo Afonso Burmann

VICE-REITOR
Luciano Schuch

PRÓ-REITOR DE PLANEJAMENTO
Frank Leonardo Casado

PRÓ-REITOR DE GRADUAÇÃO
Martha Bohrer Adaime

COORDENADOR DE PLANEJAMENTO ACADÊMICO E DE EDUCAÇÃO A DISTÂNCIA
Jerônimo Siqueira Tybusch

COORDENADOR DO CURSO DE LICENCIATURA EM COMPUTAÇÃO
Sidnei Renato Silveira

NÚCLEO DE TECNOLOGIA EDUCACIONAL

DIRETOR DO NTE
Paulo Roberto Colusso

COORDENADOR UAB
Reisoli Bender Filho

COORDENADOR ADJUNTO UAB
Paulo Roberto Colusso

NÚCLEO DE TECNOLOGIA EDUCACIONAL

DIRETOR DO NTE
Paulo Roberto Colusso

ELABORAÇÃO DO CONTEÚDO
Fábio José Parreira, Teresinha Letícia da Silva, Cristiano Bertolini,
Guilherme Bernardino da Cunha

REVISÃO LINGUÍSTICA
Camila Marchesan Cargnelutti

APOIO PEDAGÓGICO
Carmen Eloísa Berlote Brenner
Keila de Oliveira Urrutia

EQUIPE DE DESIGN
Carlo Pozzobon de Moraes – Ilustrações
Juliana Facco Segalla – Diagramação
Matheus Tanuri Pascotini – Capa e Ilustrações
Raquel Bottino Pivetta – Diagramação

PROJETO GRÁFICO
Ana Letícia Oliveira do Amaral

L755 Linguagem de programação II [recurso eletrônico] / Fábio José Parreira ... [et al.]. – 1. ed. – Santa Maria, RS : UFSM, NTE, 2019.
1 e-book : il.

Este caderno foi elaborado pelo Núcleo de Tecnologia Educacional da Universidade Federal de Santa Maria para os cursos da UAB
Acima do título: Licenciatura em Computação
ISBN 978-85-8341-257-1

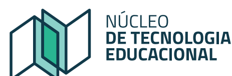
1. Linguagem de programação I. Parreira, Fábio José II. Universidade Federal de Santa Maria. Núcleo de Tecnologia Educacional

CDU 004.438

Ficha catalográfica elaborada por Lizandra Veleda Arabidian - CRB-10/1492
Biblioteca Central da UFSM

MINISTÉRIO DA EDUCAÇÃO

PROGRAD UFSM



APRESENTAÇÃO

As linguagens de programação, basicamente, estão subdivididas em um dos 4 principais paradigmas de programação, que são: Programação Orientada a Objetos (POO), imperativo, funcional e lógico. Em nosso livro, denominado *Linguagem de Programação II,* iremos nos dedicar ao estudo da POO. Destaca-se que, na POO, cada elemento do mundo real é representado por um objeto que, por sua vez, possui características e ações próprias, assim como vemos na realidade. Neste sentido, podemos dizer que um dos objetivos da POO é aproximar o mundo real do mundo computacional, além de promover, também, a unificação de dados e processos, o reaproveitamento e a manutenção de códigos.

O reaproveitamento de códigos é um dos principais requisitos no desenvolvimento de software. Destaca-se que essa prática diminui o tempo de desenvolvimento, bem como o número de linhas de código. O que torna essa prática possível nas linguagens de POO são as representações muito claras de cada um dos elementos em classes. Quanto à manutenção, podemos afirmar que um fator facilitador é a representação do sistema computacional muito próximo ao que vemos na vida real, o entendimento do sistema como um todo e de cada parte individualmente fica muito mais transparente aos olhos dos desenvolvedores.

Para que possamos trabalhar com desenvolvimento de software orientado a objetos, primeiramente, temos que escolher uma linguagem de programação. Atualmente existem várias, tais como *Java, C#, PHP, Python, C++*, entre outras. Em nosso livro, vamos trabalhar com a linguagem de programação Java, por ser uma das linguagens mais difundidas atualmente. Além disso, visando integrar a compreensão da teoria com a prática de programação, este livro foi dividido em 6 unidades:

- **Unidade 1**: Conceitos básicos da linguagem – apresenta uma breve explanação da tecnologia Java, apresentando exemplos tanto em linhas de código como na parte de interface gráfica;

- **Unidade 2**: Princípios da orientação a objetos – apresenta os conceitos básicos para o entendimento da programação orientada a objetos. Nesta unidade, iremos tratar de conceitos como a abstração, classes, visibilidade, encapsulamento, escopo de variáveis, objetos e instanciação;

- **Unidade 3**: Interações entre Objetos – apresenta a interação entre objetos que ocorre por meio de recebimento/envio de mensagens. Nesse sentido, aborda os seguintes conteúdos: declaração de métodos, construtores, sobrecarga de métodos, visibilidade das informações e métodos de acesso a atributos (*getters* e *setters*);

- **Unidade 4**: Agrupamento de Objetos – apresenta os conceitos de agrupamento de objetos utilizando *ArrayList*;

- **Unidade 5**: Herança e outras relações entre objetos – apresenta e implementa o conceito de herança e polimorfismo;

- **Unidade 6**: Manipulação de exceção – apresenta as principais técnicas de tratamentos de erros utilizando o bloco *try/catch/finally*.

Vale lembrar que até o presente momento, nas disciplinas de programação do Curso de Licenciatura e Computação, foi abordado o paradigma de programação imperativo, utilizando a linguagem C. Agora iremos navegar em um mundo novo, que é o contexto da orientação a objetos. Nesta direção, cabe ressaltar que os conteúdos supracitados são essenciais para a compreensão da POO, sem esses conceitos a programação utilizada atualmente fica inviável.

Assim como qualquer conteúdo na área de programação computacional, é preciso disponibilidade de tempo e dedicação, *POO* só se aprende programando. De posse do livro *Linguagem de Programação II*, disponibilize no mínimo 1 hora por dia e siga em frente, compartilhando a teoria com a prática computacional e vamos programar em Java.

Bons estudos.

ENTENDA OS ÍCONES

ATENÇÃO: faz uma chamada ao leitor sobre um assunto, abordado no texto, que merece destaque pela relevância.

INTERATIVIDADE: aponta recursos disponíveis na internet (sites, vídeos, jogos, artigos, objetos de aprendizagem) que auxiliam na compreensão do conteúdo da disciplina.

SAIBA MAIS: traz sugestões de conhecimentos relacionados ao tema abordado, facilitando a aprendizagem do aluno.

TERMO DO GLOSSÁRIO: indica definição mais detalhada de um termo, palavra ou expressão utilizada no texto.

SUMÁRIO

▷ **APRESENTAÇÃO ·5**

▷ **UNIDADE 1 – CONCEITOS BÁSICOS DA LINGUAGEM ·10**

Introdução ·12
1.1 Tecnologia Java ·13
1.2 Interface gráfica - NetBeans ·15
1.3 Depuração do código ·36
Atividades de reflexão ou fixação ·38

▷ **UNIDADE 2 - PRINCÍPIOS DA ORIENTAÇÃO A OBJETOS ·39**

Introdução ·41
2.1 Abstração no desenvolvimento de software ·42
2.2 Classe ·44
2.3 Objetos ·53
Atividades de reflexão ou fixação ·62

▷ **UNIDADE 3 - INTERAÇÕES ENTRE OBJETOS ·63**

Introdução ·65
3.1 Declarando métodos ·66
3.2 Construtores ·69
3.3 Sobrecarga de método/construtor ·72
3.4 Métodos get e set ·73
Atividades de reflexão ou fixação ·74

▷ **UNIDADE 4 - AGRUPAMENTO DE OBJETOS ·77**

Introdução ·79
4.1 Definições: *Collections Framework* ·80
4.2 ArrayList ·82
4.3 Coleção de CDs e DVDs ·85
Atividades de reflexão ou fixação ·102

▷ **UNIDADE 5 - HERANÇA E OUTRAS RELAÇÕES ENTRE OBJETOS ·103**

Introdução ·105
5.1 Herança ·84
5.2 Sobreposição de métodos ·117
5.3 Polimorfismo ·120
5.4 Classe abstrata ·124
5.5 Interface ·126
5.6 Herança múltipla ·130
Atividades de reflexão ou fixação ·134

▷ **UNIDADE 6 - MANIPULAÇÃO DE EXCEÇÕES ·138**

Introdução ·140
6.1 Exceções em JAVA ·141
6.2 Lançamento de exceções ·143
6.3 Princípios do Tratamento de Exceção ·147
Atividades de reflexão ou fixação ·151

▷ **CONSIDERAÇÕES FINAIS ·152**

▷ **REFERÊNCIAS ·153**

▷ **APRESENTAÇÃO DOS PROFESSORES ·154**

1

CONCEITOS BÁSICOS DA LINGUAGEM

INTRODUÇÃO

Conforme apresentado no livro *Linguagem de programação I*, as linguagens de programação são usadas para fazer a comunicação com o computador. Elas são constituídas de comandos específicos que, quando utilizados corretamente, executam uma ação. Historicamente falando não conseguimos definir, com exatidão, o surgimento da linguagem de programação, mas é possível afirmar que tudo começou na década de 30, com os primeiros computadores elétricos. Juntamente às linguagens, surgiram os paradigmas da programação, em sua maioria, na década de 1970. Dentre elas, são apresentadas algumas:

- Simula: inventada nos anos 1960 por Nygaard e Dahl, foi a primeira linguagem a suportar o conceito de classes, iniciando o *paradigma orientado a objetos*,
- C: uma das primeiras linguagens de programação de sistemas, criada por Dennis Ritchie e Ken Thompson, tem uma das maiores influências no mundo dentro do *paradigma estruturado*;
- Prolog: projetada em 1972, foi a primeira linguagem de programação com *paradigma lógico*;
- Java: com o surgimento da internet, nos anos 1990, surgiram também as linguagens *Java* e *JavaScript*, ambas possuem relação direta com a internet. Neste sentido, ressaltamos que a linguagem Java é orientada para objetos, além de ser multiplataforma.

Por definição, Java é mais que uma linguagem de programação orientada a objetos – é uma plataforma, pois ela é composta por um conjunto que engloba a linguagem de programação, máquina virtual Java (*JVM – Java Virtual Machine*) e a interface de programação de aplicativos (*API – Java Application Programming Interface*).

Sendo assim, toda vez que formos desenvolver um programa em Java temos de fazer *download* da *JDK* (*Java SE Development Kit*), pois não existe a opção de baixar somente a *JVM*. O Java *JDK* é composto pelo compilador, pelas bibliotecas (*APIs*) necessárias para criação de programas, por ferramentas utilizadas durante o desenvolvimento e por ferramentas de testes dos aplicativos Java. Em nossos exemplos, além do *JDK*, vamos utilizar o *NetBeans* como interface de desenvolvimento.

Para exemplificar a utilização do *JDK* e do *NetBenas*, esta unidade apresenta conceitos referentes à tecnologia Java, sobre programação utilizando interface gráfica no *NetBeans* (passando por pacotes, interface gráfica usando o *Form JFrame*, variáveis e operadores aritméticos, operadores relacionais, estruturas condicionais, vetores e estrutura de repetição) e, por fim, apresenta as ferramentas de depuração do código Java.

1.1 TECNOLOGIA JAVA

A tecnologia Java foi desenvolvida em meados dos anos 1990, na empresa *Sun Microsystems*, por uma equipe de programadores chefiada por James Gosling. Ao final, a equipe projetou uma linguagem multiplataforma – isso significa que podemos desenvolver uma aplicação independente do sistema operacional em que ela irá rodar, seja no *Windows, Linux* ou *MacOS*. Basicamente, no conceito multiplataforma temos um código, chamado de *bytecode*, que pode ser executado em qualquer plataforma, desde que tenha uma máquina virtual, o que difere das outras linguagens, como, por exemplo, a linguagem C, que cria um código binário para cada plataforma.

De forma simplista, podemos definir Java como uma linguagem orientada a objetos, mas ela é muito mais que uma linguagem, é uma plataforma, pois ela é um conjunto que engloba:

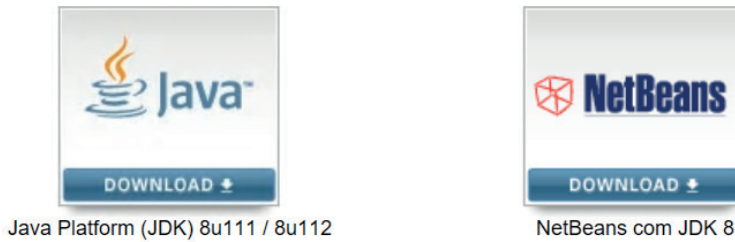
- Linguagem de programação – a linguagem de programação Java é orientada a objetos, ela é considerada como sendo simples (similar ao C, C++) e robusta, possui suporte a *threads* nativo com acoplamento em tempo de execução, possui um *garbage colletor* (coletor de lixo) para remover valores que não são mais utilizados da memória de forma automática;
- Máquina virtual Java (*JVM – Java Virtual Machine*) – tem a função de rodar instruções; logo, ela interpreta e executa os programas escritos na linguagem Java;
- Interface de programação de aplicativos – (*API – Java Application Programming Interface*) – um conjunto de bibliotecas de classes e interfaces, que desempenham funções específicas, conhecidas como *packages*, tais como *java.*, javax.**, etc.

Para iniciar a programação em Java, precisamos instalar uma *JVM* e um Ambiente de Desenvolvimento Integrado (*IDE – Integrated Development Environment*), conforme apresentado na figura 1. Atualmente, não existe a opção de baixar somente a *JVM*, uma das opções existentes é fazer o *download* da *JDK* (*Java SE Development Kit*).

INTERATIVIDADE: para fazer o *download* da *JDK + NetBeans*, abra o endereço: https://www.oracle.com/technetwork/pt/java/javase/downloads/index.html e escolha a opção *NetBeans* com *JDK8*, conforme apresentado na Figura 1.

O Java *JDK* é composto pelo compilador, pelas bibliotecas (*APIs*) necessárias para criação de programas, por ferramentas utilizadas durante o desenvolvimento e ferramentas de testes dos aplicativos Java. Neste livro, iremos utilizar a *IDE NetBeans*, totalmente gratuita e de código aberto para desenvolvedores de *softwares* nas linguagens Java, C, C++, PHP, Ruby, dentre outras.

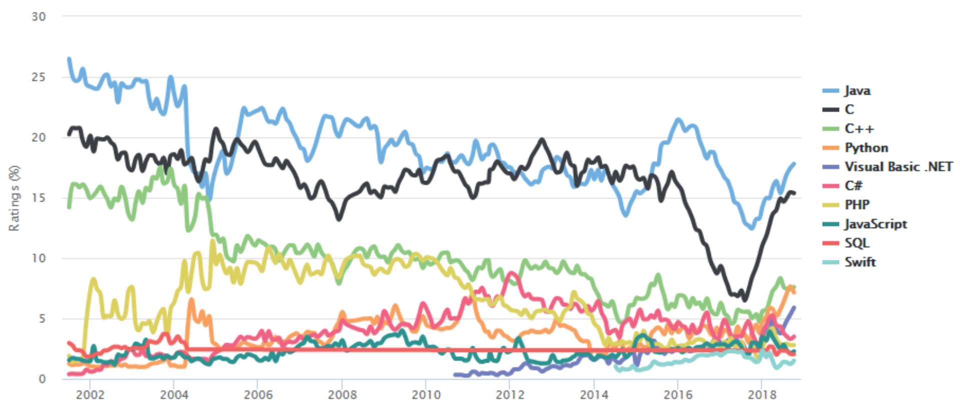
Figura 1 – *Download* da plataforma de desenvolvimento JAVA



Fonte: Oracle. Disponível em: https://www.oracle.com/technetwork/pt/java/javase/downloads/index.html

A escolha da linguagem Java se fundamenta no índice TIOBE (TIOBE, 2018). Ao analisar tal índice, na Figura 2, podemos constatar que Java é uma linguagem amplamente difundida entre os desenvolvedores, além de possuir uma grande comunidade que participa ativamente. Dessa forma, um desenvolvedor iniciante consegue achar facilmente materiais e recursos para começar a programar nessa linguagem, assim como obter ajuda da comunidade. A Figura 2 apresenta graficamente o resultado do índice TIOBE de 2002 a 2018.

Figura 2 – Índice de utilização das linguagens de programação



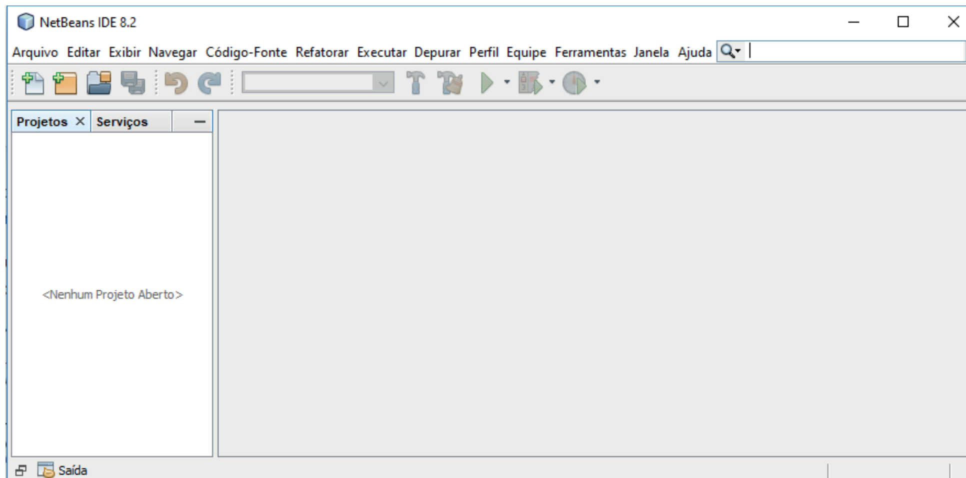
Fonte: Tiobe (2018). Disponível em: www.tiobe.com

É importante ressaltar que o índice de Programação TIOBE, apresentado na figura 2, é um indicador de popularidade das linguagens de programação. Sendo assim, ele não leva em conta a facilidade de programação ou a linguagem que possui mais linhas de códigos programada. O índice de popularidade é calculado levando em considerações o número de profissionais especializados nas linguagens de programação, cursos oferecidos e produtos disponíveis no mercado. Além disso, são analisadas as buscas realizadas nos mecanismos mais populares, como o Google, Bing, Yahoo!, Wikipedia, Amazon, YouTube e Baidu.

1.2
INTERFACE GRÁFICA – NETBEANS

Para criar os exemplos propostos neste livro, em Java, será utilizada a IDE (*Integrated Development Environment*) *NetBeans*. Após a instalação, a tela inicial do *NetBeans* é apresentada, graficamente, na Figura 3.

Figura 3 – Tela inicial do NetBeans



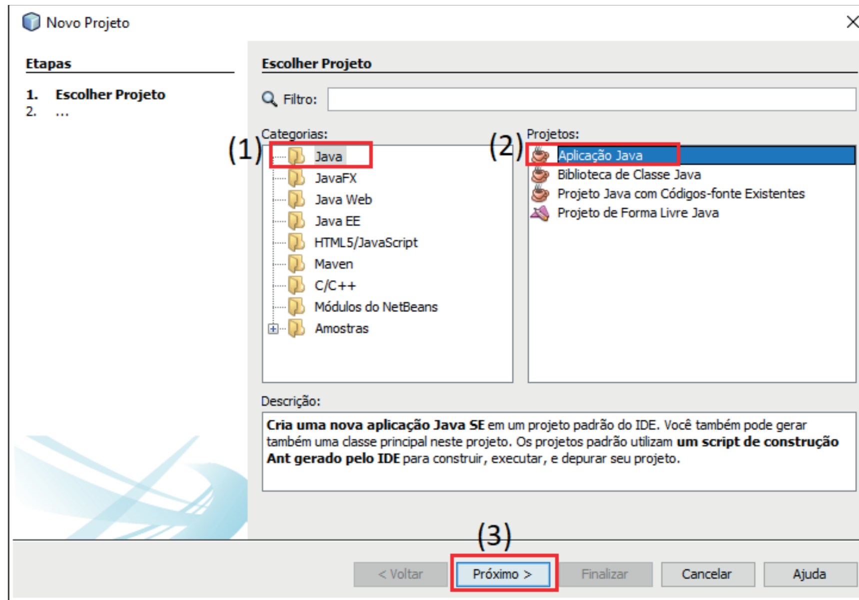
Fonte: Autores.

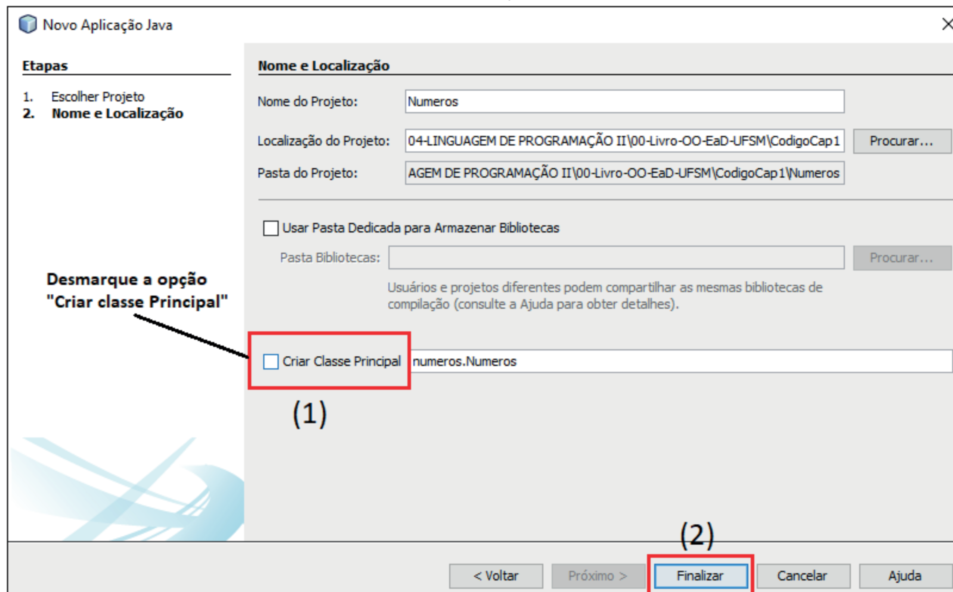
Inicialmente, vamos criar uma interface gráfica, usando o *NetBeans*, para resolver o seguinte problema:

Ler 4 números, calcular a soma de todos eles, encontrar o maior número e encontrar o menor número.

Vamos começar criando uma interface gráfica. Para isso, temos que criar um novo projeto, acessando o menu ***Arquivos* -> *Novo projeto***. Uma nova tela será apresentada. Nesta tela deve-se clicar em Java e escolher a opção “aplicação Java”. Logo após, ao clicar no botão próximo, uma nova tela abrirá. Nela, digite o nome do projeto (em nosso exemplo, foi inserido o nome de Numeros), a localização do projeto no computador, desmarque a opção criar classe principal e clique no botão finalizar, conforme apresentado na Figura 4.

Figura 4 – Novo projeto Java

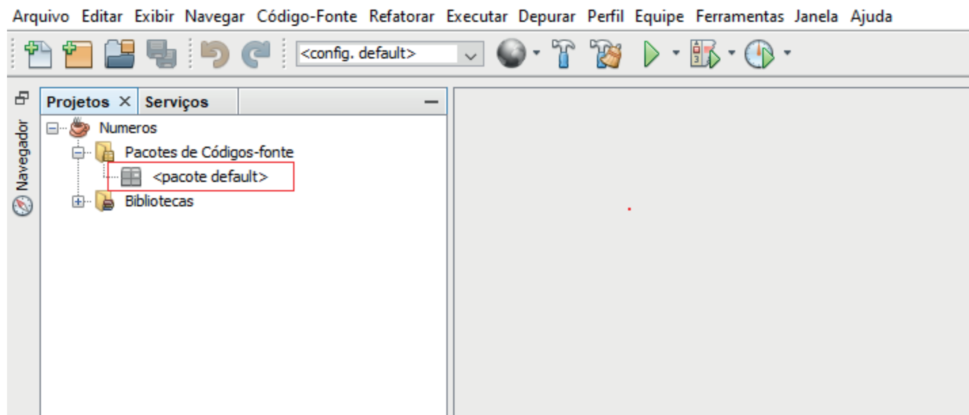




Fonte: Autores.

Ao finalizar esses passos, temos o nosso primeiro projeto criado, conforme apresentado na Figura 5. Podemos notar que não existe nenhum arquivo e ainda aparece “pacote default”. Significa que não criamos nenhuma classe e também não criamos nenhum pacote. A seguir, será explanado o conceito de pacote.

Figura 5 – Projeto Criado



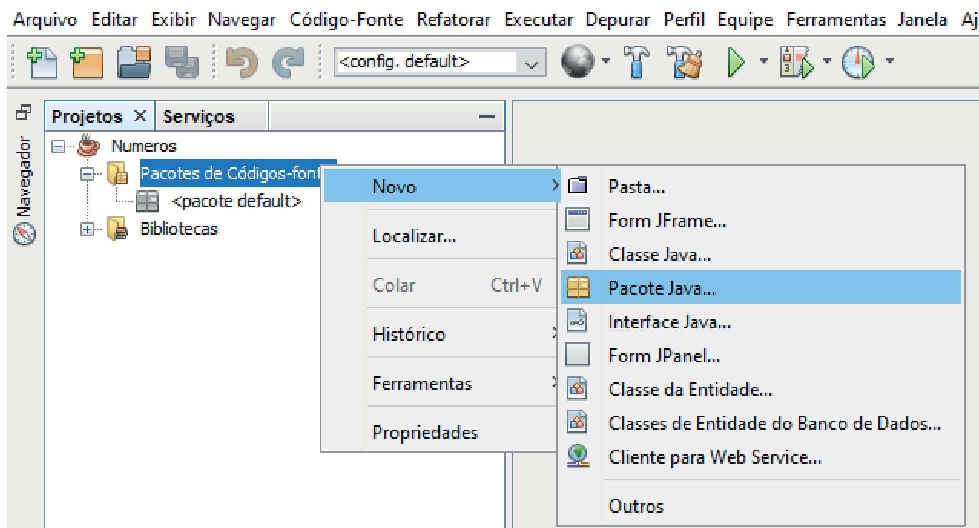
Fonte: Autores.

Pacotes

Em Java, um pacote ou *package*, a grosso modo, pode ser considerado como sendo uma pasta ou diretório em que ficam armazenados os arquivos fonte de Java que tenham semelhança. Todos os arquivos dentro do mesmo pacote são visíveis entre si, e podem acessar as informações uns dos outros.

Embora Java possua um pacote padrão quando criamos um projeto, recomenda-se criar um pacote para organizar as classes semelhantes. Para isso, clique com o botão direito do mouse sobre *"Pacotes de código-fonte"* -> *Novo* -> *Pacote Java*, conforme mostra, graficamente, a Figura 6.

Figura 6 – Criando um novo pacote



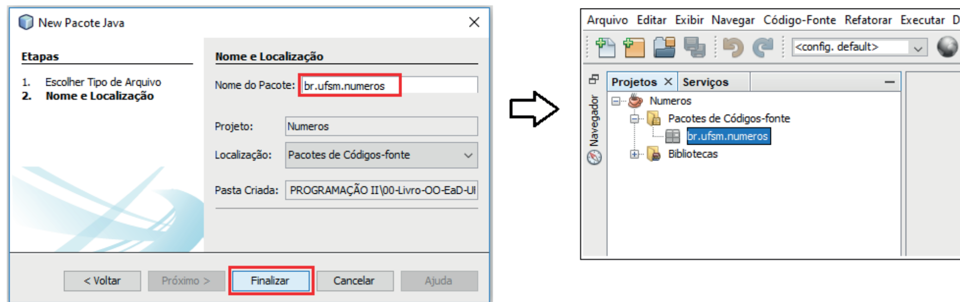
Fonte: Autores.

Após finalizar a sequência de ações apresentadas na Figura 6, temos que inserir o nome do pacote, conforme mostra a Figura 7. Para criar o nome do pacote, a SUM normatizou um padrão que relaciona o endereço eletrônico da empresa desen-

volvedora do projeto ou instituição de ensino, com o nome do pacote. Em nosso exemplo, vamos usar o endereço eletrônico da UFSM. Veja como fica o nome do nosso pacote: br.ufsm.numeros.

É importante observar que os pacotes só possuem letras minúsculas, independente de quantas palavras formam o nome.

Figura 7 – Inserindo o nome do pacote

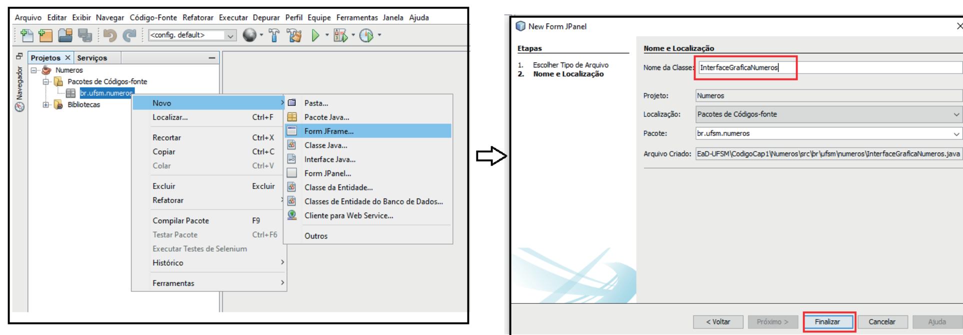


Fonte: Autores.

Interface Gráfica – Form JFrame

Agora que já temos o pacote devidamente criado, vamos prosseguir com a construção da interface. Para isso, temos que criar um contêiner Java. Esse contêiner receberá todos os outros componentes que iremos utilizar na nossa interface gráfica. Em nosso exemplo, vamos criar um contêiner usando o formulário *JFrame*. A Figura 8 representa a criação do *JFrame*.

Figura 8 – Criando JFrame

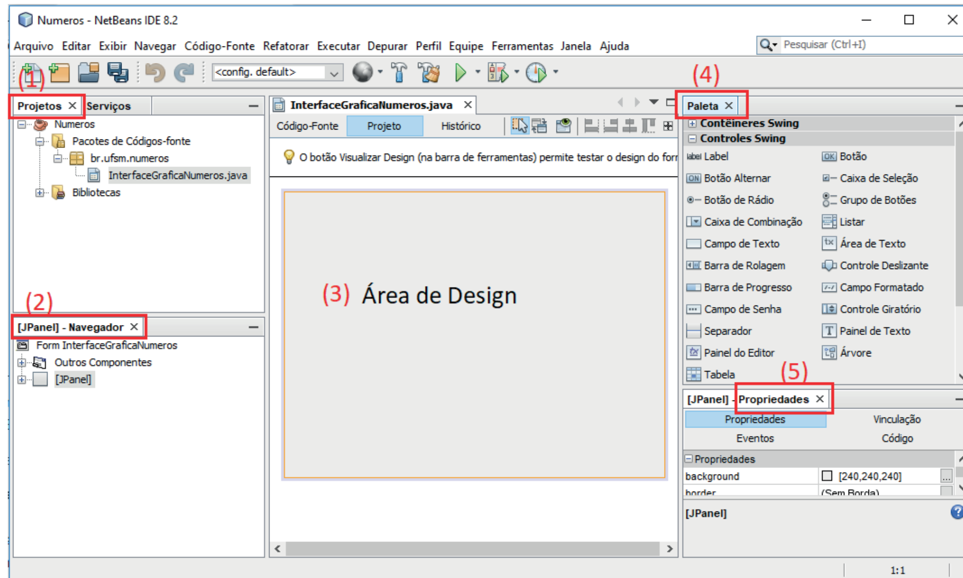


Fonte: Autores.

Após a criação do *JFrame*, conforme apresenta graficamente a Figura 8, vamos conhecer a interface do *GUI Builder*, no *NetBeans*, como mostra a Figura 9.

TERMO DO GLOSSÁRIO: GUI *Graphical User Interface* – Interface gráfica

Figura 9 – Interface gráfica no *NetBeans*



Fonte: Autores.

Ao criar um form no *NetBeans*, são abertas algumas janelas adicionais na IDE, permitindo navegar, organizar e editar componentes na área de design. A seguir, vamos descrever cada uma dessas janelas, mostradas na Figura 9:

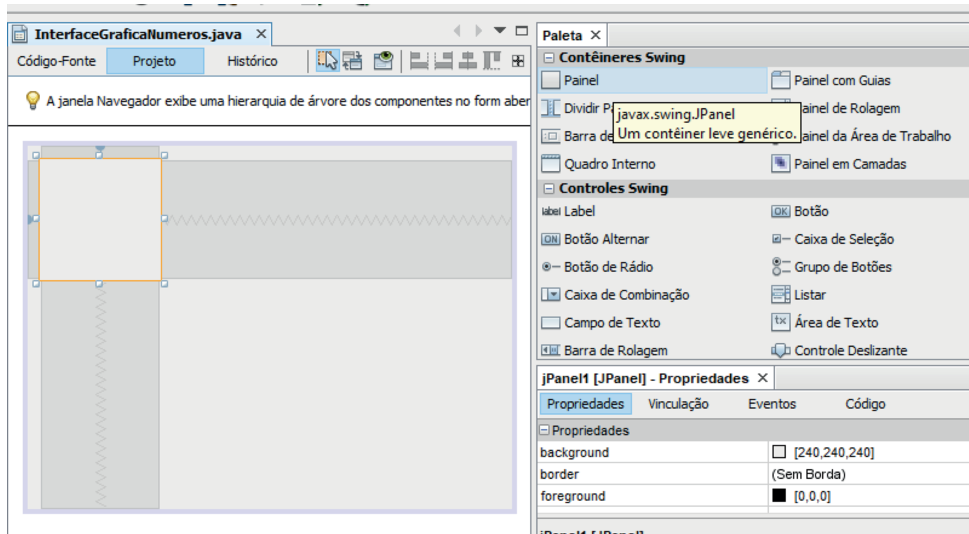
1. Projetos: Apresenta, de forma organizada, os arquivos do projeto em forma de árvore hierárquica;
2. Navegador: Representa em forma de árvore hierárquica todos os componentes do *form*, sendo eles invisíveis ou visíveis. Além disso, o navegador apresenta, de forma visual, o componente que está em edição;
3. Área de *design*: Nessa janela, podemos inserir, mover ou editar componentes em um *form*. Note que a área de *design* está com o botão *Projeto* ativo; por isso, exibe os componentes graficamente. Já o botão *Código-fonte* faz a exibição do código-fonte referente ao *form* que está ativo. O botão *Visualizar Design* exibe graficamente todos os componentes, apresentando uma previa da aplicação gráfica. O botão *Histórico* apresenta as alterações do arquivo;
4. Paleta: Apresenta todos os componentes disponíveis organizados em guias, que são denominadas de contêineres *swing*, controles *swing*, menus *swing*, janelas *swing*, preenchedor *swing*, AWT, *beans* e Java Persistência;
5. Propriedades: Ao selecionar um componente na área de *design*, as propriedades do mesmo serão exibidas.

SAIBA MAIS: traz sugestões de conhecimentos relacionados ao tema abordado, facilitando a aprendizagem do aluno.

Relembrando o problema a ser resolvido: temos que ler 4 números, calcular a soma, encontrar o maior e o menor. Para separar a leitura dos cálculos, vamos usar um painel (ou componente *JPanel*), que se localiza na guia de contêineres

swing. Para inseri-lo, clique sobre o componente painel, arraste até o canto esquerdo superior do *form* e depois solte, conforme apresentado na Figura 10.

Figura 10 – Interface gráfica no NetBeans



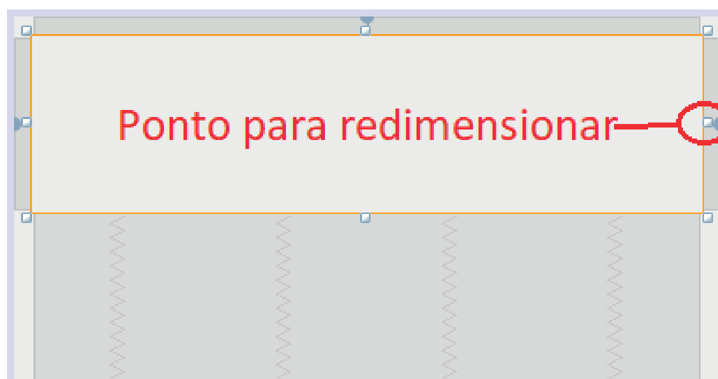
Fonte: Autores.

Uma vez inserido, temos que redimensionar o componente, de acordo com a sequência de ações:

1. Selecione o painel – ao selecionar ele fica com as bordas ressaltadas, como mostra a Figura 10;
2. Clique e segure sobre o ponto de redimensionamento na margem direita do Painel, arrastando até as proximidades da margem direita do *form*;
3. Solte o ponto de redimensionamento do componente.

Ao final, o Painel redimensionado é apresentado na Figura 11.

Figura 11– Redimensionando o Painel



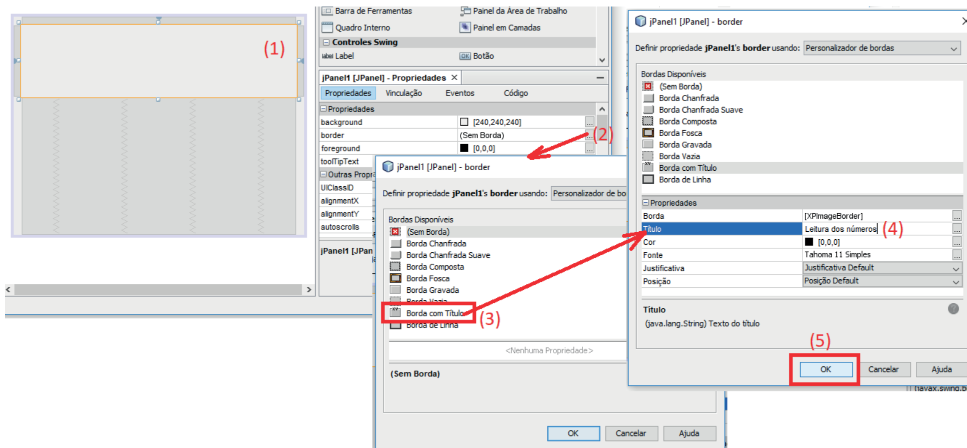
Fonte: Autores.

Iremos inserir dois componentes do tipo Painel. Para diferenciá-los, teremos que inserir bordas e título para cada um deles. Para inserir a borda e o título, siga os

passos abaixo, apresentados na Figura 12:

1. Selecione o Painel a ser alterado;
2. Localize a janela de propriedades, no canto direito, localize a propriedade "*Border*" e clique no botão de reticências (...)
3. Escolha a opção borda com título;
4. Na propriedade título insira "Leitura dos números";
5. Clique em OK para finalizar os procedimentos.

Figura 12 – Inserindo borda e título no Painel



Fonte: Autores.

O resultado final é apresentado na Figura 13. Além do primeiro painel, chamado "Leitura dos números", devemos inserir um segundo, chamado "Resultado".

Figura 13 – Inserção dos dois painéis

Fonte: Autores.

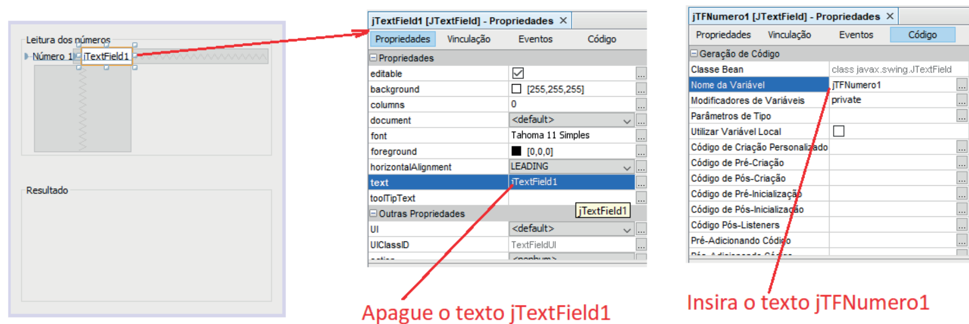
Precisamos iniciar a inserção dos componentes. Primeiramente, no Painel "Leitura dos números", vamos inserir os componentes *label (JLabel)* e campo de texto *(JTextField)*. Os passos para inserir um *label* serão detalhados a seguir:

1. Selecione o componente *Label*, na categoria Controles do *Swing*;
2. Arraste o componente *Label* para o Painel "Leitura dos números" até que apareçam as linhas guias, escolha a melhor posição no canto superior direito e solte o componente;
3. Renomeie o texto "jLabel1" para "Número 1"; para isso, clique sobre o *label* e aperte F2; em seguida, digite o texto "Número 1".

Dando continuidade, precisamos adicionar um campo de texto *(JTextField)* ao Painel "Leitura dos números":

1. Selecione o componente Campo de texto, na Paleta, na categoria Controles do *Swing*;
2. Arraste o componente campo de texto para o Painel "Leitura dos números" até que apareçam as linhas guias, escolha a melhor posição de tal forma que fique alinhado com o *label*, inserido anteriormente, e solte o componente;
3. Apague o texto "jTextField1"; para isso, vá em Propriedades, e delete o texto em "*Text*". Depois altere o nome do campo de texto para "jTFNumero1", conforme mostra a Figura 14.

Figura 14 – Alteração da propriedade Text e Nome do campo de texto



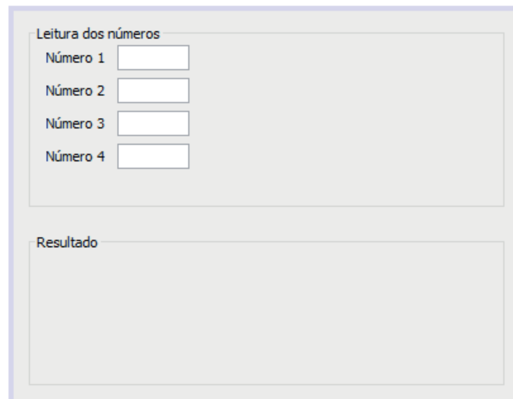
Fonte: Autores.

Ainda não terminamos – faltam 3 *labels* e 3 campos de texto, para armazenar os 3 números que ainda falta ler. Ao inserir cada componente, é importante observar as linhas de guias horizontal e vertical, elas servem de referência para posicionar os componentes, sugerindo os espaçamentos entre os componentes. Isso faz com que a nossa interface gráfica respeite a aparência do sistema operacional, ao ser executada. Insira os *labels* e os campos de texto obedecendo aos seguintes critérios:

1. Insira o *label* 2 e coloque o texto Número 2, o campo de texto 2 e renomeie o campo de texto para jTFNumero2;
2. Insira o *label* 3 e coloque o texto Número 3, o campo de texto 3 e renomeie o campo de texto para jTFNumero3;
3. Insira o label 4 e coloque o texto Número 4, o campo de texto 4 e renomeie o campo de texto para jTFNumero4.

O resultado final da interface do Painel “Leitura dos números” é apresentado na Figura 15.

Figura 15 – Componentes do Painel “Leitura dos números”



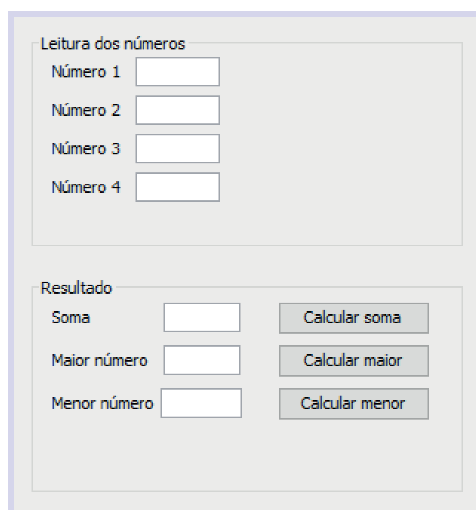
Fonte: Autores.

Passaremos agora ao Painel “Resultado”. Nesse Painel, iremos inserir componentes do tipo *label*, campo de texto e botão:

1. Insira um *label* e altere o texto para soma; insira um campo de texto, delete o texto e altere o nome para jTFSoma; insira um botão e altere o nome para jBSoma;
2. Insira um *label* e altere o texto para Maior número; insira um campo de texto, delete o texto e altere seu nome para jTFMaior; insira um botão e altere o nome para jBMaior;
3. Insira um *label* e altere o texto para jTFMenor número; insira um campo de texto, delete o texto e altere o nome para menor; insira um botão e altere o nome para jBMenor.

O resultado da inserção dos componentes é apresentado na Figura 16.

Figura 16 – Componentes do Painel “Resultado”



Fonte: Autores.

Vamos pôr a mão na massa? Vamos inserir o código para que cada botão cumpra as suas ações corretamente. Ao clicar no botão "Calcular soma", deve-se adquirir as informações referentes aos números 1, 2, 3 e 4, fazer a soma e apresentar o resultado. Para realizar essa tarefa, precisaremos declarar atributos (variáveis) da classe e dos métodos. Classe e método serão explicados mais à frente. Nesse momento, teremos que definir um padrão de nomenclatura de variáveis e aprender sobre operadores aritméticos.

Variáveis e operadores aritméticos

Podemos definir variável como sendo uma estrutura computacional que permite armazenar informações na memória do computador, quando o programa for executado. Obrigatoriamente, em Java, todas as variáveis devem ser declaradas antes de serem usadas. Entendemos por declaração de variáveis o ato de criá-las em algum ponto do programa.

O Java é *case sensitive*, o que significa que os nomes de variáveis diferenciam entre si, quando são escritos com letras maiúsculas ou minúsculas. Sendo assim, para criar os nomes das variáveis precisamos seguir algumas regras. Neste livro, vamos adotar a notação *CamelCase*. Cabe ressaltar que este padrão, além de ser adotado como boa prática na programação, é largamente utilizado em várias linguagens de programação, como Java, C#, Ruby, PHP e Python, sendo utilizado nas definições de classes, objetos e variáveis.

Podemos descrever *CamelCase* como sendo uma denominação do inglês que normatiza a prática de escrever palavras compostas ou até mesmo frases. Nesse contexto, cada palavra é iniciada com letras maiúsculas e unidas sem espaços.

Em um programa de computador, o *CamelCase* ajuda a manter o código legível, além de auxiliar na identificação dos nomes. Para que possamos ter noção da importância dessa nomenclatura, tente ler as frases: "LicenciaturaEmComputaçãoNaUFSM" e "licenciaturaemcomputaçãonaUFSM". Pergunto: qual das duas frases conseguimos interpretar com maior facilidade? Você concorda que foi a primeira frase?

Em se tratando da aplicação do *CamelCase* na programação Java, iremos aplicar constantemente nas classes, variáveis e métodos. Neste universo, temos duas variações:

1. *UpperCamelCase*: onde a primeira letra obrigatoriamente é Maiúscula para:
    - nome de classe
    - nome de enum
    - nome de interfaces
2. *lowerCamelCase*: Nessa variação a primeira letra obrigatoriamente é minúscula para:
    - Métodos
    - Variáveis (local e instância)

Agora vamos verificar o nosso código e analisar se ele foi construído conforme as regras supracitadas. A Figura 17 apresenta o código construído até o momento.

Para acessar o código fonte do projeto clique em Código-fonte, conforme mostra a Figura 17. Na linha 6, temos a declaração do pacote ao qual o código pertence. Neste caso, essa classe pertence ao pacote "br.ufsm.numeros". Na linha 11, temos a declaração da classe com o nome "InterfaceGraficaNumeros". Observe que foi obedecido rigorosamente o padrão *CamelCase*. Já na linha 220, temos a criação do método *main*, responsável por iniciar a execução do nosso projeto, conforme citado anteriormente, começa com letras minúsculas. Nas linhas de 254 a 272 temos as declarações das variáveis, todas começando com letra minúscula.

Figura 17 – Análise do código

Código-Fonte Projeto Histórico

Clique aqui para acessar o código-fonte

```
1    ...5 linhas
6    package br.ufsm.numeros;
7
8    /**
9     * @author prof. Fabio Parreira
10    */
11   public class InterfaceGraficaNumeros extends javax.swing.JFrame {
12
13       /**
14        * Creates new form NewJFrame
15        */
16       public InterfaceGraficaNumeros() {
17           initComponents();
18       }
19
20       /** This method is called from within the constructor to initi
25       @SuppressWarnings("unchecked")
26       Generated Code
⋮
217      /**...3 linhas */
220      public static void main(String args[]) {
221          /* Set the Nimbus look and feel */
222          Look and feel setting code (optional)
243          //</editor-fold>
244
245          /* Create and display the form */
             java.awt.EventQueue.invokeLater(new Runnable() {
                 public void run() {
248                  new InterfaceGraficaNumeros().setVisible(true);
249              }
250          });
251      }
252
253      // Variables declaration - do not modify
254      private javax.swing.JButton jBMaior;
255      private javax.swing.JButton jBMenor;
256      private javax.swing.JButton jBSoma;
⋮
268      private javax.swing.JTextField jTFNumero1;
269      private javax.swing.JTextField jTFNumero2;
270      private javax.swing.JTextField jTFNumero3;
271      private javax.swing.JTextField jTFNumero4;
272      private javax.swing.JTextField jTFSoma;
273      // End of variables declaration
274  }
```

Fonte: Autores.

Agora que você já sabe como criar os nomes das variáveis, precisamos entender como podemos declará-las. Neste momento, vamos nos ater somente a variáveis com tipos primitivos, que podem ser: *byte, short, int, long, float, double, char* ou

boolean. A seguir, serão detalhados cada um dos tipos. A Figura 18 detalha os tipos inteiros.

Figura 18 – Tipo inteiro

	Tipo	Memória consumida	Valor Mínimo	Valor Máximo
Tipos Inteiros	byte	1 byte	-128	127
	short	2 bytes	-32.768	32.767
	int	4 bytes	-2.147.483.648	2.147.483.647
	long	8 bytes	-9.223.372.036.854.770.000	9.223.372.036.854.770.000

Fonte: Autores.

Os valores numéricos podem conter partes fracionárias (ou ponto flutuante). Neste caso, eles são armazenados como sendo dos tipos *float* ou *double*, conforme mostra a Figura 19.

Figura 19 – Ponto flutuante

	Tipo	Memória consumida	Valor Mínimo	Valor Máximo	Precisão
Ponto flutuante	float	4 bytes	- 3,4028E + 38	3,4028E + 38	6 – 7 dígitos
	double	8 bytes	- 1,7976E + 308	1,7976E + 308	15 dígitos

Fonte: Autores.

Em Java, temos duas formas para armazenar elementos textuais, que são: caracteres e textos. A representação por caractere é feita pelo tipo *char* e armazena uma única letra, conforme mostra a Figura 20. Já a representação de textos é feita pela classe *String*, que será detalhada mais tarde.

Figura 20 – Caracteres

	Tipo	Memória consumida	Valor Mínimo	Valor Máximo
Caractere	char	16 bytes	0	65535

Fonte: Autores.

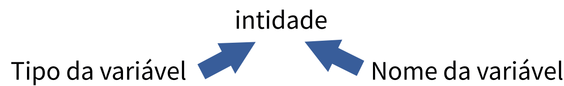
Já os dados lógicos são representados pelo tipo *booleano*, que armazena os valores true ou false, conforme apresentado na Figura 21.

Figura 21 – Tipo de Dados *Booleano*

	Tipo	Memória consumida	Valor Mínimo	Valor Máximo
Booleano	boolean	1 bit	false	true

Fonte: Autores.

Java é uma linguagem fortemente tipada. Isso significa que cada variável obrigatoriamente deve ter um tipo declarado antes que possa ser utilizada. A sintaxe de declaração de variáveis é apresentada abaixo:



Além de nos preocuparmos com o tipo da variável, temos que decidir onde iremos declarar a variável dentro da classe. Seguindo este contexto teremos variáveis locais e variáveis de instância.

As variáveis locais são criadas enquanto o método que as contém for executado, por isso a variável só terá vida enquanto o método estiver em execução. Logo, elas são declaradas dentro dos métodos para serem utilizadas somente dentro destes métodos, não sendo acessíveis em outras partes do programa. Já as variáveis de Instância são criadas quando uma nova instância da classe é criada, e tem vida até essa instância da classe ser removida. Elas são preferencialmente declaradas logo após o nome da classe, por isso ficam fora dos métodos e são acessíveis em todas as partes do programa.

Dizemos que escopo de uma variável corresponde ao seu tempo de vida. Por exemplo, as variáveis locais possuem escopo dentro do método que foram declaradas, já as variáveis de Instância possuem escopo dentro da classe em que foram criadas.

Aprendemos a declarar as variáveis. Mas, quais são as operações aritméticas possíveis entre elas? Dizemos que operadores aritméticos são aqueles que efetuam operações matemáticas entre uma ou mais variáveis dos tipos primitivos. A Figura 22 apresenta graficamente os operadores aritméticos.

Figura 22 – Operadores aritméticos

Operador	Exemplo	Descrição
+	a+b	operador de adição
-	a-b	operador de subtração
*	a*b	operador de multiplicação
/	a/b	operador de divisão
%	a%b	operador de módulo (ou resto da divisão)

Fonte: Autores.

De posse dos conhecimentos sobre variáveis, vamos voltar ao nosso código. Inicialmente, vamos declarar variáveis locais para armazenar os números. Para inserir o código referente ao botão “Calcular soma”, dê um duplo clique neste botão para construir o método onde iremos digitar o nosso código, conforme mostrado na Figura 23.

Figura 23 – Variáveis locais

```
193     private void jBSomaActionPerformed(java.awt.event.ActionEvent evt) {
194         // TODO add your handling code here:
195         //Declara as variáveis para receber os numeros 1, 2, 3 e 4
196         double numero1, numero2, numero3, numero4, soma;
197
198         //Lê o texto do campo de texto (jTFNumero1.getText(),
199         //como o campo de texto retorna uma string, temos que
200         //converter para double usando o Double.parseDouble()
201         numero1 =  Double.parseDouble(jTFNumero1.getText());
202         numero2 =  Double.parseDouble(jTFNumero2.getText());
203         numero3 =  Double.parseDouble(jTFNumero3.getText());
204         numero4 =  Double.parseDouble(jTFNumero4.getText());
205
206         //Faz a soma dos 4 numeros e armazena na variável soma
207         soma=numero1+numero2+numero3+numero4;
208
209         //como o campo de texto só trabalha com string temos
210         //que converter de double para string
211         //usando String.valueOf(soma)
212         //Agora atribuir o valor convertido de soma
213         //ao Campo de texto usando jTFSoma.setText()
214         jTFSoma.setText(String.valueOf(soma));
215     }
```

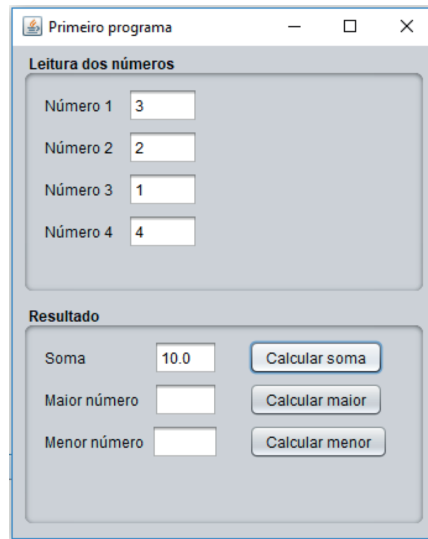
Fonte: Autores.

Ao analisar a Figura 23, na linha 196 foram declaradas todas as variáveis locais para fazer armazenar os números digitados na interface (chamados numero1, numero2, numero3 e numero4) e uma variável para armazenar a soma dos quatro números (chamada soma). Todas estas variáveis foram definidas com o tipo *double*.

Agora temos que ler o conteúdo digitado pelo usuário no campo de texto JTFNumero1 (jTFNumero1.getText( )), converter para *double* (*Double.parseDouble( )*) e armazenar na variável numero1, conforme apresenta a linha 201. Na linha 207, são realizadas as somas dos numero1, numero2, numero3 e numero4 e é atribuído o resultado à variável soma.

Por fim, na linha 214, temos que atribuir o valor da variável soma, que é do tipo *double*, ao campo de texto *jTFSoma*, que aceita somente *string*. Por isso, antes de mais nada, temos que fazer a conversão de double para *string* (*String.valueOf(soma)*) e depois atribuir ao *jTFSoma* (*jTFSoma.setText( )*). O resultado final da execução do código que faz a soma é apresentado na Figura 24.

Figura 24 – Execução do método que faz a soma



Fonte: Autores.

Ao finalizar o cálculo da soma dos quatro números, vamos encontrar o maior número digitado entre os quatro. Para isso, precisamos conhecer os operadores relacionais e as estruturas condicionais, que serão detalhadas a seguir.

Operadores relacionais

Os operadores relacionais servem para avaliar dois operandos, averiguando se o operando à esquerda é maior >, menor <, maior ou igual >=, menor ou igual <=, quando comparado com o operador da direita, retornando ao final um valor *booleano*. Tais operadores geralmente são utilizados em tomadas de decisões. Desta forma, as decisões baseiam-se em testes do estado das variáveis. A Figura 25 apresenta, graficamente, os operadores relacionais.

Figura 25 – Operadores relacionais

Operador	Exemplo	Descrição
>	x > y	x é maior que y
<	x < y	x é menor que y
>=	x >= y	x é maior que ou igual a y
<=	x <= y	x é menor que ou igual a y

Fonte: Autores.

Estruturas condicionais

Na linguagem Java, temos três tipos de instruções de seleção: *if, if..else* e *switch*. A primeira instrução *if* possui uma seleção, caso o teste da condição seja verdadeiro é realizada a ação, caso seja falsa pula para a próxima ação. Para exemplificar, veja o algoritmo abaixo:

```
if (condição) {
   caso a condição seja verdadeira esse bloco de código será executado
}
```

Já a instrução *if... else*, tem a possibilidade de selecionar uma das duas ações, realizando uma ação se a condição do *if* for verdadeira e realiza outra ação, diferente da primeira, se a condição do *if* for falsa, neste caso executa a ação contida no *else*, conforme descrito abaixo:

```
if ( condição ){
   caso a condição seja verdadeira esse bloco de código será executado
} else {
   caso a condição seja falsa esse bloco de código será executado
}
```

Ainda temos a estrutura *if... else...if*, ou seja, estrutura encadeada de *if... else*. Veja abaixo:

```
if (condição 1) {
   caso a condição 1 seja verdadeira esse bloco de código será executado
} else if (condição  2) {
   Caso a condição 1 seja falsa e a condição 2 seja verdadeira esse bloco de código
será executado
} else {
   Caso a condição 2 seja falsa esse bloco de código será executado
}
```

Notem que na estrutura encadeada de *if...else* podemos adicionar infinitos *else... if*, ou seja, quantos forem necessários. Mas o *else* deve ser adicionado apenas uma vez, dentro da estrutura, como alternativa na falha de todos os testes anteriores.

Por fim, a instrução de seleção *switch* tem a possibilidade de selecionar uma entre muitas ações diferentes, dependendo do valor de uma variável ou expressão. Cada opção, *switch (opção)* está associada com o valor que uma variável ou expressão pode assumir, por exemplo, *case opção 1*. Veja o exemplo descrito abaixo:

```
switch( opção ){
   case opção 1:
         comandos caso a opção 1 tenha sido escolhida
         break;
```

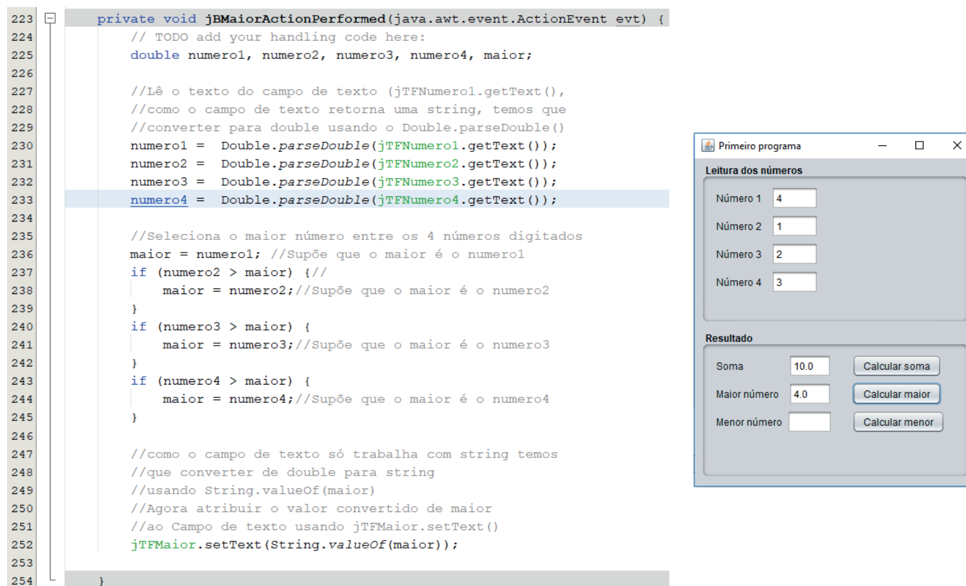
case opção 2:
comandos caso a opção 2 tenha sido escolhida
break;
case opção 3:
comandos caso a opção 3 tenha sido escolhida
break;
default:
comandos caso nenhuma das opções anteriores tenha sido escolhida
}

No exemplo de código acima, temos a instrução *break*, que tem a função de alterar o fluxo de controle. Logo, ela ocasiona a saída imediata da instrução e o fluxo de execução continua na primeira instrução abaixo do *break*.

Para implementar o cálculo do maior número, iremos usar o operador relacional > e a estrutura condicional *if*. A Figura 26 apresenta o código para calcular o maior número.

Inicialmente, na linha 236, é atribuído à variável maior o valor de numero1, na 237 é realizada a verificação (numero2 > maior), caso seja verdadeiro é atribuído à variável maior o valor de numero2. Na 240 é realizada a verificação (numero3 > maior), caso seja verdadeiro é atribuído à variável maior o valor de numero3. Por fim, na 243 é realizada a verificação (numero4 > maior), caso seja verdadeiro é atribuído à variável maior o valor de numero4.

Figura 26 – Cálculo do maior número



Fonte: Autores.

Para finalizar o nosso exemplo, temos que implementar o código do botão "Calcular menor". Neste código, serão utilizados vetores e estrutura de repetição. A seguir, serão explanados esses conceitos.

Vetores

Por definição, vetores são estruturas de dados, ou seja, são formas de organizar os dados em um programa, tornando as informações referenciáveis como um todo. Também são homogêneos porque, a princípio, armazenam o mesmo tipo de dado.

Quando declararmos um vetor, estamos reservando na memória principal do computador uma série de espaços para uso da variável daquele tipo. Para isso, é preciso definir o tamanho do vetor, isto é, a quantidade total de elementos que terá de armazenar. Em seguida, é necessário reservar espaço de memória para armazenar os elementos, por meio do operador *new*. A sintaxe da declaração e criação do vetor (*array*) é a seguinte:

tipo[] nomeDoArray;
nomeDoArray = new tipo[numeroDeElementos];

Em Java podemos declarar um vetor, basicamente, de duas formas. Na primeira, faz-se a declaração do vetor e, em seguida, a criação do vetor com tamanho determinado. Veja:

int vet[]; // declaração do vetor
vet = new int[5]; // *criação do vetor determinando seu tamanho, neste exemplo com 5 posições*

A outra possibilidade é fazer a declaração e a criação do vetor numa única linha, como segue:

int vet[] = new int[5]; // Declaração e criação em uma única linha:

Em nosso exemplo, estamos trabalhando com números fracionários, por isso temos que declarar um vetor com o tipo *double*, conforme abaixo:

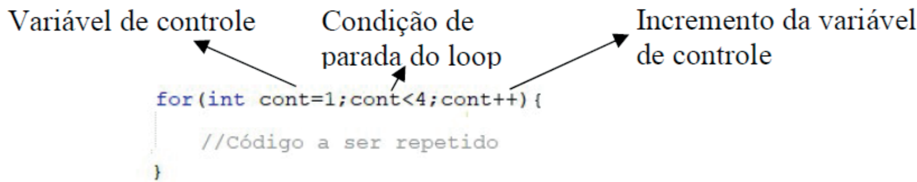
double numeros[] = new double[4];

Estrutura de repetição

Geralmente, em um programa de computador, vamos precisar repetir a execução de um bloco de códigos, até que determinada condição ou número de repetições seja satisfeito. Para criar essas estruturas de repetições, vamos usar as estruturas de repetições, que são os laços *for*, *while* e o *do-while*. A seguir, serão detalhadas cada uma destas estruturas de repetição.

O laço ***for*** (para) garante uma certa segurança para que a aplicação não entre em *looping*, pois esse laço traz no cabeçalho da instrução todos os valores necessários para a execução dos ciclos. A sintaxe do laço *for* é apresentada na Figura 27.

Figura 27 – Laço for

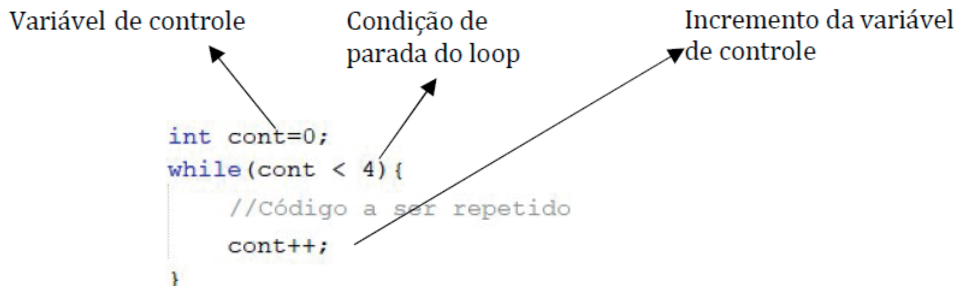


Fonte: Autores.

Este laço deve ser usado quando soubermos a quantidade de vezes que a estrutura deve ser repetida, ou quando o teste precisa ser feito antes da execução da estrutura a ser repetida.

Outro laço bastante usado é o *while* (enquanto). Este laço se destina a aplicações em que não sabemos a quantidade de repetição necessárias a um bloco de código. O *while* executa as instruções, de forma continua, até que a condição de parada seja falsa; portanto, a condição de parada retorna um valor *booleano* (verdadeiro ou falso). A Figura 28 apresenta a estrutura do laço *while*.

Figura 28 – Laço *while*

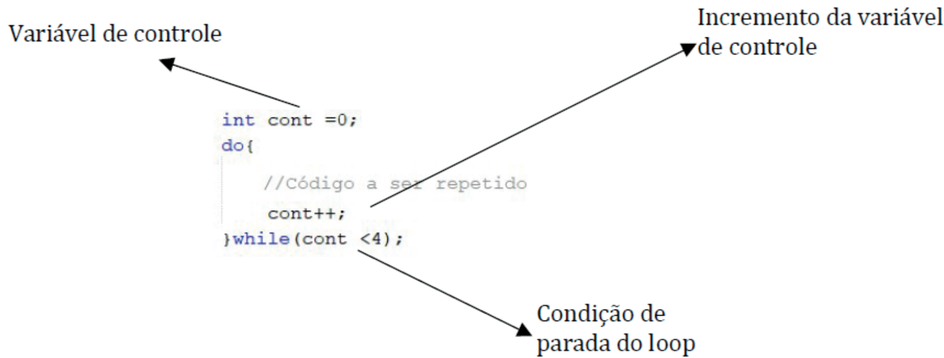


Fonte: Autores.

Sempre que formos utilizar o laço *while*, devemos lembrar que ele deve ser empregado quando não soubermos exatamente quantas vezes o laço deve ser repetido ou quando o teste condicional precisa ser feito antes de iniciar a execução do bloco de comandos a serem repetidos.

O laço *do...while* (faça...enquanto) é bem semelhante ao *while*. No laço *while*, o programa testa a condição de continuação do *loop* no começo do *loop*, antes de executar o corpo do *loop*. Vale lembrar que, se a condição for falsa, o corpo nunca será executado. Já no laço *do...while*, o teste é feito depois de executar o bloco de comandos a serem repetidos, no corpo do laço, o que garante que eles serão executados ao menos uma única vez. A Figura 29 apresenta a sintaxe do laço *do...while*.

Figura 29 – Laço *do...while*



Fonte: Autores.

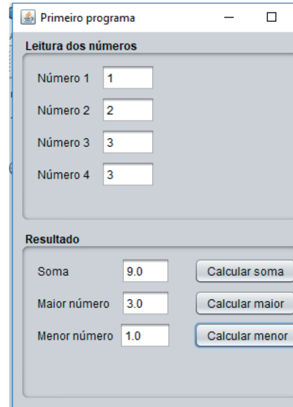
Vale ressaltar que o laço ***do...while*** deve ser usado sempre que não soubermos a quantidade de vezes que o laço deve repetir o bloco de comandos a ser executado ou quando o bloco de comandos precisa ser executado no mínimo uma única vez.

Dentre as estruturas de repetição apresentadas acima, para calcular o menor número, vamos utilizar o laço *for*. Além do laço, vamos também utilizar um vetor para armazenar os quatro números. A Figura 30 apresenta o código para calcular o menor número.

Inicialmente, na linha 280, é declarado um vetor chamado números, com tamanho 4, do tipo *double*. Na linha 281, é declarada a variável para armazenar o menor número armazenado no vetor. Nas linhas 285 a 288 são realizadas as leituras dos campos de texto (por exemplo, o *jTFNumero1.getText()*) e na sequência é realizada a conversão de texto para *double*, usando o método *Double.parseDouble()*. Na linha 290, é atribuído o primeiro elemento armazenado no vetor números à variável menor, na linha 291 é construído o laço de repetição for iniciando de 1, tem a condição de parada do *loop* como sendo *cont* < *4*. Isso permite repetir por três vezes o bloco de código. Na linha 292, é feita a verificação para saber se o elemento do vetor, no índice *cont*, é menor que o conteúdo armazenado na variável *menor*, caso seja, na linha 294 é realizada a atribuição do elemento contido no vetor à variável *menor*. Por fim, na linha 302, é feita a conversão de *double* para *String* e atribuído o resultado da conversão ao campo de texto usando o método *jTFMaior.setText()*.

Figura 30 – Cálculo do menor número

```java
278    private void jBMenorActionPerformed(java.awt.event.ActionEvent evt) {
279        // TODO add your handling code here:
280        double numeros[] = new double[4];
281        double menor;
282        //Lê o texto do campo de texto (jTFNumero1.getText(),
283        //como o campo de texto retorna uma string, temos que
284        //converter para double usando o Double.parseDouble()
285        numeros[0] =  Double.parseDouble(jTFNumero1.getText());
286        numeros[1] =  Double.parseDouble(jTFNumero2.getText());
287        numeros[2] =  Double.parseDouble(jTFNumero3.getText());
288        numeros[3] =  Double.parseDouble(jTFNumero4.getText());
289
290        menor= numeros[0];
291        for(int cont=1;cont<4;cont++){
292            if( numeros[cont] < menor){//Código a ser repetido
293                menor= numeros[cont];  //Código a ser repetido
294            }
295        }
296
297        //como o campo de texto só trabalha com string temos
298        //que converter de double para string
299        //usando String.valueOf(maior)
300        //Agora atribuir o valor convertido de maior
301        //ao Campo de texto usando jTFMaior.setText()
302        jTFMenor.setText(String.valueOf(menor));
303    }
```



Fonte: Autores.

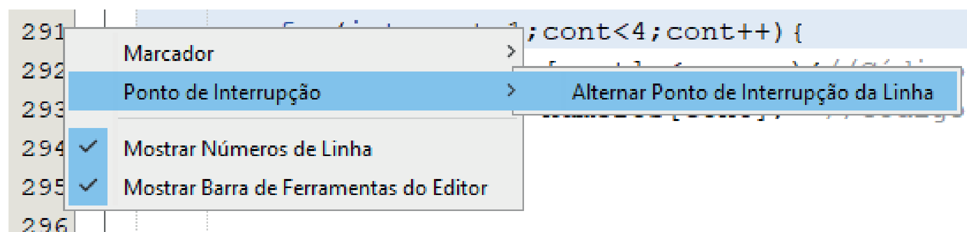
1.3
DEPURAÇÃO DO CÓDIGO

Embora não seja impossível trabalhar sem depuração, ela facilita muito a inspeção do código, principalmente quando nos deparamos com erros na programação. Sendo assim, podemos dizer que depuração em programação é o ato de inspecionar um programa em modo de execução e observar os valores contidos em suas variáveis e, também, observar qual e quando cada método é invocado. Por isso, ela é uma ferramenta poderosa que auxilia na análise do código, independentemente da linguagem utilizada. Um bom depurador oferece a análise linha a linha do código, e permite rastrear a execução e descobrir eventuais erros, por meio da exibição dos valores contidos nas variáveis e endereços de objetos.

Ressalto a você que o depurador é uma ferramenta essencial durante o desenvolvimento. Sendo assim, é importante aprender a usar o depurador do *NetBeans* para depurar nossas aplicações Java. Além de facilitar a localização de erros, ele pode ser utilizado quando há a necessidade de entender o funcionamento de um código.

Para iniciar a depuração, antes de tudo temos que criar um ponto de interrupção, na linha de código que queremos analisar. Nesse ponto, a execução do programa é interrompida e o depurador espera por um novo comando para continuar. Para criar um ponto de interrupção, clique com o botão direito no número da linha, no lado esquerdo do código-fonte, onde a execução deve ser interrompida. Um menu de contexto será exibido, nele selecione *"Alternar Linha de pontos de interrupção"*. A Figura 31 apresenta, graficamente, a inserção de um ponto de interrupção *(breakpoint)* na linha 291.

Figura 31 – Ponto de interrupção ou breakpoint



Fonte: Autores.

Após a inserção do ponto de interrupção, podemos iniciar a depuração. Pressione as teclas *"Ctrl + F5"* para iniciar a depuração do projeto no *NetBeans*. O depurador irá executar o programa até o primeiro ponto de interrupção. Agora você pode passar o *mouse* sobre as variáveis e janelas de informação que aparecem junto a elas. Estas janelas de informação exibirão o valor da variável e o tipo. Conforme apresentado na Figura 32, visualiza-se o conteúdo da variável números. Observe que a variável números é um vetor de 4 posições, na posição 0 está armazenado o valor 1, na posição 1 o valor 2, na posição 2 o valor 3 e na posição 3 o valor 4.

Figura 32 – Conteúdo da variável

Fonte: Autores.

Para prosseguir com a execução linha a linha, pressione a tecla *"F7"* ou *"F8"*. A tecla *F7* faz com que o depurador entre no código executado, nesse sentido, quando há muitas chamadas de métodos o depurador irá percorrer todas elas, ocasionando uma análise profunda do código. Já, ao usar a tecla *F8*, faz com que o depurador passe por cima do código, sem entrar nas chamadas dos métodos.

Figura 33 – Opções para fazer depuração

Fonte: Autores.

A figura 33 apresenta outras formas de fazer a depuração do código. Para descobrir mais opções, vá no menu *principal->janelas->depuração.*

ATIVIDADES - UNIDADE 1

Todas as atividades abaixo devem ser postadas no Moodle/UAB-UFSM, conforme direcionamento do professor da disciplina.

1) Construa uma interface gráfica para ler 3 números e informar se o primeiro é maior do que a soma dos dois últimos.

2) Faça um protótipo usando interface gráfica para calcular a multa de um veículo que trafega por uma rodovia com fiscalização eletrônica. Para isso:

1- Leia a velocidade do carro e a velocidade máxima permitida na rodovia
2 - Informe 50 reais se estiver até 5km/h acima da velocidade permitida
3 - Informe 100 reais se estiver entre 6km/h e 20km/h acima da velocidade permitida
4 - Informe 300 reais se estiver acima de 21km/h acima da velocidade permitida.

2

PRINCÍPIOS DA ORIENTAÇÃO A OBJETOS

INTRODUÇÃO

Anteriormente, estudamos os pilares da tecnologia *Java*, na Unidade 1. Agora passaremos a estudar os princípios da programação orientação a objetos. Por definição a programação orientada a objetos é um paradigma, no qual é implementado um conjunto de classes, que definem os objetos através de seus estados e comportamentos necessários ao software. Para projetar um sistema orientado a objetos, primeiramente, é necessário entender como funciona a abstração de objetos do mundo real para o mundo computacional, que contribui para a redução da complexidade do problema a ser resolvido, pois iremos capturar apenas as informações necessárias do objeto, para representá-lo no mundo computacional.

Outro conceito fundamental que iremos abordar é o conceito de classe. Dizemos que este conceito é um dos pilares na programação orientada a objetos, pois ele permite a reutilização efetiva de código. Neste contexto, uma classe representa, de maneira abstrata, um conjunto de objetos do mundo real que possuem características afins, com comportamentos definidos através dos métodos e o gerenciamento do estado definido por meio de atributos.

Uma vez criadas as classes, podemos falar dos objetos no mundo computacional. Após projetar uma classe, temos de criar instâncias para utilizá-la. O resultado desta ação, a grosso modo, é um objeto, e cada objeto possui um estado próprio. Dessa forma, com apenas uma classe podemos criar diversos objetos, cada um com seu estado e comportamentos próprio.

Para que possamos entender a diferença entre classes e objetos façamos a analogia da construção de uma casa ou apartamento. Neste sentido, a planta de uma casa é um modelo, um planejamento, tal como a classe. Essa casa tem diversas características que não estão expressas no modelo (classe), tais como cor, quando será construída e o valor da venda. Já o objeto, é uma classe materializada, ou seja, o objeto possui todos os atributos atualizados com valores próprios, por exemplo: casa azul, construída em 2018, com valor final de venda de R$180.000,00.

Diante do exposto, o assunto supracitado, que trata do conhecimento básico sobre o paradigma orientado a objetos, será abordado inicialmente com a abstração de objetos, depois explanaremos alguns conceitos sobre classe, tais como: a visibilidade, this, representação gráfica, encapsulamento e escopo de variáveis. E, por fim, abordaremos os conceitos sobre objetos, dentre eles a instanciação e relacionamentos.

2.1
ABSTRAÇÃO NO DESENVOLVIMENTO DE SOFTWARE

Para compreender o mundo, os seres humanos utilizam-se de três princípios de organização dos pensamentos, são eles:

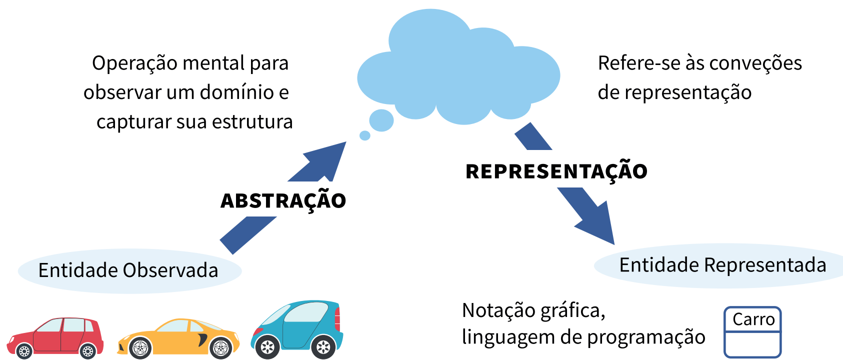
- diferenciação,
- distinção entre todo e parte,
- classificação.

A diferenciação corresponde ao ato ou efeito de estabelecer diferenças entre duas ou mais coisas. Na distinção entre todo e parte, iremos fazer a separação entre as características gerais de um grupo, que sejam semelhantes, com as caraterísticas individuais de cada coisa ou elemento. Por fim, a classificação pode ser definida como a ação e o efeito de classificar. Entendemos que classificar seja o ato de ordenar ou dispor por classes coisas ou elementos.

Ao final do dia, o cérebro humano acumula milhares de informações coletadas ao nosso redor. No entanto, tais informações normalmente são simplificadas por meio de um processo conhecido como abstração. Na abstração computacional, também utilizamos os conceitos supracitados para extrair as principais informações durante o planejamento de uma classe. Dentro do escopo de desenvolvimento de software, podemos definir abstração como "a capacidade de ignorar detalhes de partes para focar a atenção em um nível mais elevado de um problema" (BARNES; KOLLING, 2008, p. 50).

Basicamente, ao aplicarmos a abstração em um problema estamos reduzindo a complexidade do problema a ser resolvido, pois iremos capturar apenas as informações necessárias. Quando escrevemos um programa em Java, que é uma linguagem orientada a objetos, estaremos criando um modelo computacional baseado em classes, para um problema vivenciado no mundo. Para criarmos tal modelo teremos que eliminar muitos detalhes desnecessários, visando reunir informações genéricas que se apliquem a uma ampla gama de situações ou objetos. Essas abstrações genéricas podem frequentemente ser muito úteis, por exemplo, conforme apresentado na Figura 34. Nela, é apresentado, graficamente, o conceito de abstração para o objeto carro. Observe que existem vários tipos de carro; nesse caso, temos que extrair as informações para resolver o nosso problema, que sejam comuns a todos os carros. Além disso, teremos que extrair as informações gerais do grupo, por exemplo, marca, modelo, cor predominante, lugares e combustível.

Figura 34 – Abstração



Fonte: Autores.

Neste panorama, estamos reunindo as informações sobre o problema do mundo real, que consiste na representação computacional do objeto carro; logo, os detalhes são geralmente uma combinação entre:

- As informações oferecidas por pessoas interessadas no sistema e
- As informações observadas por nós

Cabe a nós, os desenvolvedores, fazer a abstração por meio da seleção das informações mais relevantes para o bom funcionamento do programa computacional. Isso é essencial, pois incluir todas as informações torna o programa computacional muito difícil de projetar, programar, testar, depurar, documentar e dar manutenção. Também é importante frisar que a seleção de tais informações deve ser realizada respeitando o contexto do problema.

Para exemplificar, vamos construir uma abstração que represente um aluno? Neste software, é importante saber a cor da pele, a cor dos olhos, comida favorita, salário, curso que frequenta ou até mesmo o período que está estudando? A resposta é: qualquer uma dessas características do aluno pode ser relevante ou irrelevante, dependendo do sistema a ser desenvolvido. As informações necessárias para desenvolver um sistema de gerenciamento de bibliotecas são diferentes de um sistema de vendas *on-line* para uma determinada loja. Nesta abstração, vamos delimitar o contexto do problema em um sistema de gerenciamento de bibliotecas. Neste sentido, dentro das informações supracitadas podemos selecionar: curso que frequenta e o período que está estudando.

2.2
CLASSE

Antes de iniciar o conceito de classe em programação orientada a objetos, em Java, precisamos relembrar o conceito de objeto no mundo real, de acordo com o dicionário Michaelis (2018) uma classe pode ser uma "Coisa material e perceptível pelos sentidos. Qualquer coisa (física ou mental) para a qual uma ação, um pensamento ou sentimento se dirige. Pessoa ou objeto real ou imaginário".

A primeira definição se refere a objetos do mundo real no sentido em que normalmente pensamos neles, ou seja, como uma determinada coisa que podemos ver e tocar, e que ocupa espaço, sendo considerada como substantivo concreto. Imagine agora que você esteja na sala de aula, no polo da UAB de sua cidade, pense em alguns objetos físicos que fazem parte deste cenário:

- Os alunos;
- Os professores;
- As salas de aula físicas no polo;
- Computadores;
- Calculadora.

Passaremos a analisar a segunda definição, que, de acordo com Barker (2005, p. 71), enfatiza a parte "mental para a qual uma ação, um pensamento ou sentimento se dirige". Sendo assim, teremos várias concepções de objetos do mundo real que são abstratos e podem desempenhar papéis importantes dentro do contexto do problema exposto, ao qual se refere a sala de aula, no polo da UAB de sua cidade. A seguir, são apresentados alguns objetos abstratos que fazem parte do nosso cotidiano:

- O curso que você está matriculado;
- O departamento ao qual o curso pertence;
- A sala de aula virtual no *Moodle*.

Uma classe na programação orientada a objetos é, na verdade, o projeto de um objeto do mundo real. Sendo assim, ela é uma abstração que descreve os recursos comuns a todos os objetos pertencentes a grupos semelhantes. Entendemos como objetos do mundo real, geralmente, como sendo os substantivos concretos, por exemplo, aluno, carro, calculadora ou substantivos abstratos como curso, departamento.

Uma classe obrigatoriamente deve possuir:

- um nome;
- visibilidade, por exemplo: *public, private, protected*;
- os atributos (ou variáveis de instância) que definem as informações, como nome e tipo do atributo;
- os métodos (ou as operações) a serem executados.

Por exemplo, a classe soma, pertencente a uma calculadora, pode ser projeta-

da para ter dois atributos:

- *double* numero1;
- *double* numero2;

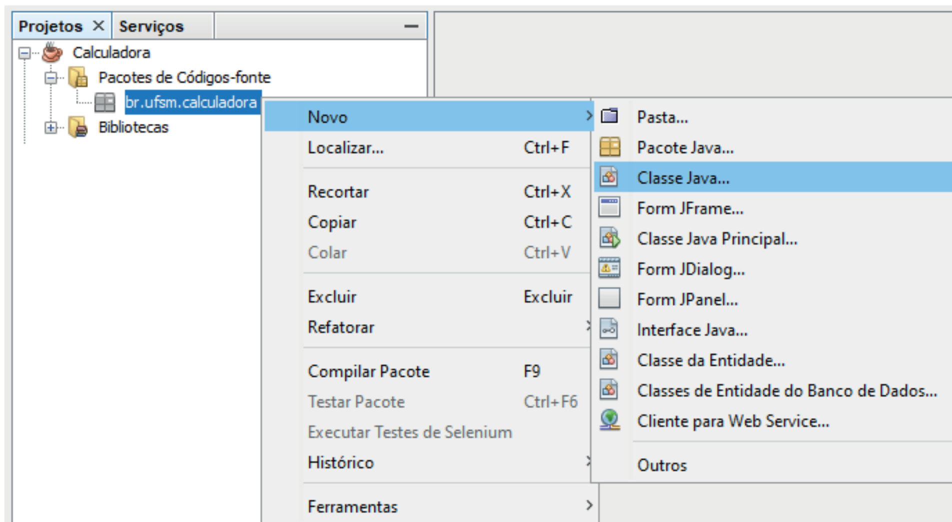
Já o método, definido para a classe soma é:

- *calcularSoma* – retorna a soma do numero1 com o numero2.

É importante ressaltar que uma classe só pode oferecer as ações (os métodos) para as quais ela foi projetada. Quando pensamos em uma calculadora simples, ela deve ser projetada para executar as ações de soma, subtração, multiplicação e divisão. Já uma calculadora científica, além dessas quatro operações, executa operações como logaritmo e exponenciação. Cabe ressaltar que um aspecto importante para ter sucesso no projeto de uma classe é garantir antecipadamente todos os comportamentos (ou métodos) desta classe. Após determinarmos quais as informações (atributos) e as ações (métodos) que são comuns ao conjunto de objetos do mundo real, devemos formalmente declará-los como atributos e métodos em uma classe Java.

Para exemplificar a criação de classes, vamos criar um projeto chamado *Calculadora*. Nesse projeto, crie um pacote chamado *br.ufsm.calculadora*. Após essas ações, podemos criar a classe chamada *Soma*; para isso, clique sobre o nome do pacote, e vá em *novo=>Classe Java...* conforme apresentado na Figura 35.

Figura 35 – Criação do projeto, pacote e classe

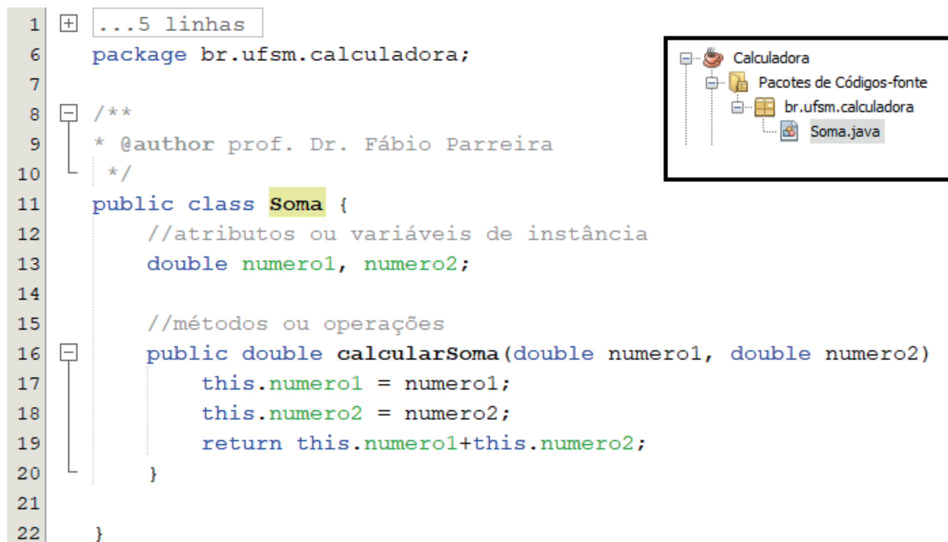


Fonte: Autores.

A seguir, a Figura 36 apresenta o resultado das ações solicitadas anteriormente. Observe que a classe foi criada em um arquivo chamado *Soma.java*. Esse arquivo possui todas as definições da classe, tais como atributos e métodos. Analisando a Figura 36, a começar na linha 11, temos descrita a visibilidade como sendo *public*, (posteriormente será detalhado este conceito em Java), a palavra reservada *class* que indica a criação de uma nova classe e Soma que representa o nome da classe.

Observe que o nome da classe obedece ao padrão *UpperCamelCase*, explicado anteriormente. Na linha 13, foram declarados os atributos do tipo *double* para as variáveis denominadas *numero1* e *numero2*. Na linha 16, temos a declaração do método *calcularSoma*. Ele possui visibilidade *public*, retorna um parâmetro *double* e recebe dois parâmetros do tipo *double*. Nas linhas 17 e 18, são atribuídos os parâmetros (*numero1* e *numero2*) aos atributos (*this.numero1* e *this.numero2*) (posteriormente, será explicado o significado da palavra reservada *this*). Por fim, na linha 19, é feito o retorno (*return*) da soma do *numero1* com o *numero2*.

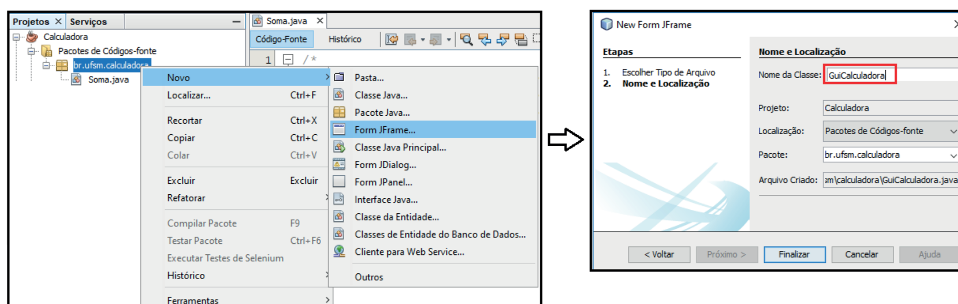
Figura 36 – Classe Soma em Java



Fonte: Autores.

Agora vamos criar a interface gráfica da calculadora. Para isso, inicialmente, vamos criar um contêiner usando o formulário *JFrame*. A Figura 37 representa a criação do *JFrame* da calculadora.

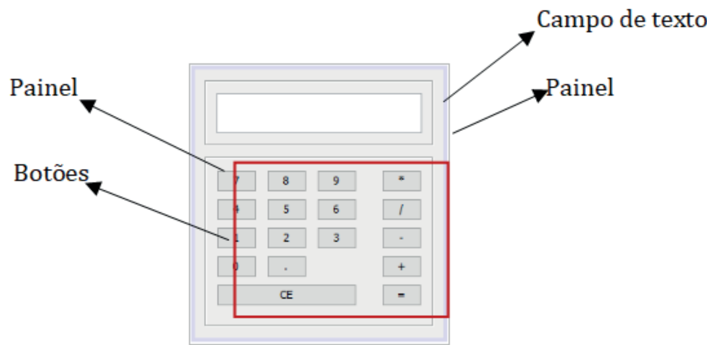
Figura 37 – Criando o *JFrame* da calculadora



Fonte: Autores.

Após a criação do *JFrame*, vamos inserir os objetos conforme apresentados, graficamente, na Figura 38.

Figura 38 – Interface gráfica da calculadora



Fonte: Autores.

Na sequência, depois da inserção dos objetos gráficos, temos que renomear todos eles, conforme apresentado na Figura 39. Caso ainda tenha dúvidas quanto à renomeação dos objetos gráficos consulte a unidade 1, no item "1.2 Interface gráfica – *NetBeans*".

Figura 39 – Renomeação dos objetos gráficos da calculadora

```
// Variables declaration - do not modify
private javax.swing.JButton jB_0;
private javax.swing.JButton jB_1;
private javax.swing.JButton jB_2;
private javax.swing.JButton jB_3;
private javax.swing.JButton jB_4;
private javax.swing.JButton jB_5;
private javax.swing.JButton jB_6;
private javax.swing.JButton jB_7;
private javax.swing.JButton jB_8;
private javax.swing.JButton jB_9;
private javax.swing.JButton jB_CE;
private javax.swing.JButton jB_Divisao;
private javax.swing.JButton jB_Igual;
private javax.swing.JButton jB_Multiplicacao;
private javax.swing.JButton jB_Ponto;
private javax.swing.JButton jB_Soma;
private javax.swing.JButton jB_Subracao;
private javax.swing.JPanel jPanel1;
private javax.swing.JPanel jPanel2;
private javax.swing.JTextField jTF_Visor;
// End of variables declaration
```

```
Calculadora
  Pacotes de Códigos-fonte
    br.ufsm.calculadora
      GuiCalculadora.java
      Soma.java
```

Fonte: Autores.

Visibilidade

Os modificadores de acesso definem os padrões de visibilidade de acessos para as classes, atributos e métodos. Existem somente 3 modificadores (*private, protected* e *public*), e com isso temos 4 níveis de visibilidade: *private, default, protected* e *public*. A seguir, serão detalhados cada um deles:

- *private* – Ao fazer uso deste modificador, a única classe que tem acesso ao elemento definido como privado é a própria classe que o define. Os Atributos só podem ser acessados por objetos da mesma classe e os métodos só

podem ser chamados por métodos da própria Classe. Vale ressaltar que este modificador não é utilizado para classes, somente em métodos ou atributos;

- *default* – Quando deixamos de definir a visibilidade, ou seja, não definimos nenhum tipo de modificador, o compilador define a visibilidade padrão (*default*). Isso significa que todos as classes que estiverem dentro do mesmo pacote terão acesso à classe, métodos ou atributos;
- *protected* – Este modificador é praticamente igual *default*, com a diferença de que se uma classe, podendo estar dentro ou fora do mesmo pacote, estende a classe com o atributo *protected*, para ter acesso a ele. Sendo assim, o acesso se concretiza por pacote e por herança. Esse atributo não se aplica a classes;
- *public* – Ao utilizar o modificador *public*, a classe pode ser acessada de qualquer lugar e por qualquer entidade que possa visualizar a classe, ou seja, todos tem acesso. Sendo assim, a classe pode ser instanciada por qualquer outra classe e os atributos e métodos são acessíveis (leitura, escrita) por objetos de qualquer classe.

A Figura 40 apresenta, de forma sucinta, onde podemos aplicar cada um dos modificadores de acesso e suas simbologias utilizadas na representação gráfica da classe.

Figura 40 – Aplicação dos modificadores e representação gráfica

Modificador	Símbolo	Podemos aplicar em:		
		Classe	Atributos	Métodos
public	+	sim	sim	sim
protected	#	não	sim	sim
default	~	sim	sim	sim
private	-	não	sim	sim

Fonte: Autores.

A Figura 41 apresenta, graficamente, quais elementos podem ter acesso em cada um dos modificadores.

Figura 41 – Modificadores de acesso

Visibilidade	public	protected	default	private
Dentro da própria classe	sim	sim	sim	sim
Classes no mesmo pacote	sim	sim	sim	não
Classes filha no mesmo pacote	sim	sim	sim	não
Classe filha em pacote diferente	sim	sim	não	não
Classe em pacote diferente	sim	não	não	não

Fonte: Autores.

This

Usamos a palavra reservada *this* para fazer uma autorreferência ao próprio contexto em que a classe se encontra. Sendo assim, *this* sempre faz alusão à própria classe. Em nosso exemplo da Figura 36, criamos o método que recebe dois argumentos chamados *numero1* e *numero2*. Na sequência, queremos atribuir para os atributos da classe, que também se chamam *numero1* e *numero2*. Para fazer tal atribuição, temos que diferenciar os argumentos dos atributos, fazendo uso do *this*, pois ele se refere ao contexto classe. Logo, podemos concluir que *numero1* e *numero2* com o *this* são os atributos da classe; e *numero1* e *numero2* e sem o *this* se referem aos parâmetros do método.

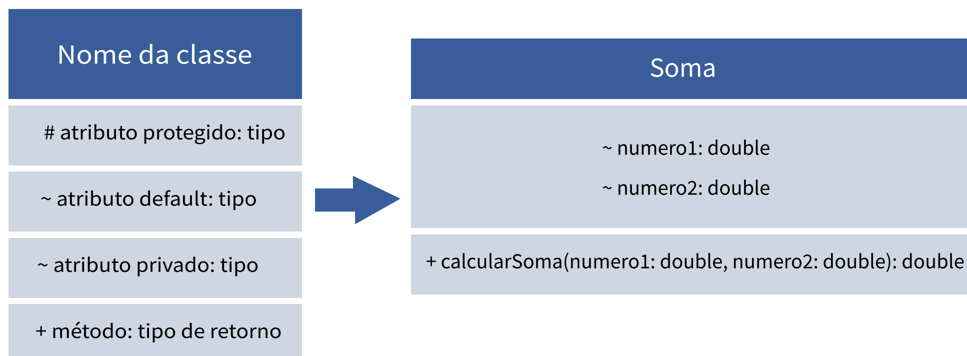
Embora a autorreferência this seja muito utilizada, temos que tomar bastante cuidado nos seguintes quesitos:

- Não pode ser usada dentro de métodos estáticos;
- É obrigatório o uso de *this* quando há ambiguidade entre variáveis locais (ou parâmetros) e atributos.

Representação gráfica

Uma classe é representada graficamente, em UML (*Unified Modeling Language*), na forma de um retângulo, contendo duas linhas que a separam em três partes. A primeira contém o nome, a segunda os atributos e a última os métodos da classe, como mostra a Figura 42.

Figura 42 – Representação gráfica da classe



Fonte: Autores.

Observe que na Figura 36 foram atribuídos os modificadores de acesso para os atributos e métodos, alterando a visibilidade destes, conforme apresentados na Figura 40.

Encapsulamento

O encapsulamento é um termo que existe apenas nas linguagens de programação orientada a objetos. Logo, encapsular significa reunir os atributos e métodos em

uma única classe, que foi construída especificamente para representar uma abstração de uma entidade do mundo real. Sendo assim, quando iremos encapsular informações estamos referindo a um processo que agrupa o estado de objetos (os atributos da classe) e seus comportamentos (os métodos da classe) em conjuntos, levando em consideração o grau de semelhança entre eles. Desta forma, tudo que precisamos saber sobre o comportamento de um objeto, por exemplo, a classe *soma*, da *calculadora*, está contido dentro dos limites desta classe, seja:

- diretamente, como um atributo, ou
- indiretamente, como um método que pode fornecer uma resposta ou determinar os valores iniciais dos atributos, determinando desta forma o estado do objeto.

Dizemos que o principal propósito do encapsulamento é a organização dos dados relacionados, agrupando-os (encapsulando-os) em classes e, consequentemente, reduzindo as colisões de nomes de variáveis, pois as variáveis com o mesmo nome estarão em espaços distintos. O mesmo ocorre com os métodos, pois utilizam os atributos pertencentes a cada um deles e estão dentro do espaço individual, que é reservado a eles. Este padrão ajuda a manter o programa computacional mais legível e fácil de dar manutenção.

A Figura 43 exemplifica uma aplicação de encapsulamento tendo como exemplo duas classes, a Subtração e Soma. Nestas classes, existem características, que são os atributos, comuns, como *numero1* e *numero2*. Ainda temos métodos que são específicos a cada uma das classes. Desta forma, podemos notar que estas classes encapsulam os dados relacionados a cada uma delas, de forma que possamos acessá-los de forma independente e sem conflito, por estarem cada um deles em seu domínio, inseridos em suas respectivas classes. Neste contexto, supomos que queremos descobrir o número 1, da soma, podemos perguntar por Soma.numero1 e, da mesma forma, para saber o número 1 da subtração, podemos perguntar por Subtracao.numero1. Embora os dois atributos possuam o mesmo nome, cada um tem o seu próprio domínio, ou seja, estão declarados em classes distintas. O mesmo pode ser dito sobre os métodos das classes. Concluímos que os dados estão encapsulados nas respectivas classes.

Figura 43 – Representação gráfica das classes Subtração e Soma

Subtracao	Soma
~ numero1: double ~ numero2: double	~ numero1: double ~ numero2: double
+ calcularSubtracao(numero1: double, numero2: double): double	+ calcularSoma(numero1: double, numero2: double): double

Fonte: Autores.

Além do conceito supracitado, temos a ocultação de informações que é considerada parte importante do encapsulamento. Ocultar determinada informação significa que não há necessidade de apresentar todo o conhecimento da classe; logo, podemos usar qualquer módulo como se fosse uma caixa preta, sem saber como

as ações serão executadas, nos atendo somente à entrada e saída. Assim, não precisamos nos preocupar com a lógica computacional de determinados métodos, devemos nos ater apenas a chamá-los e utilizar os seus resultados.

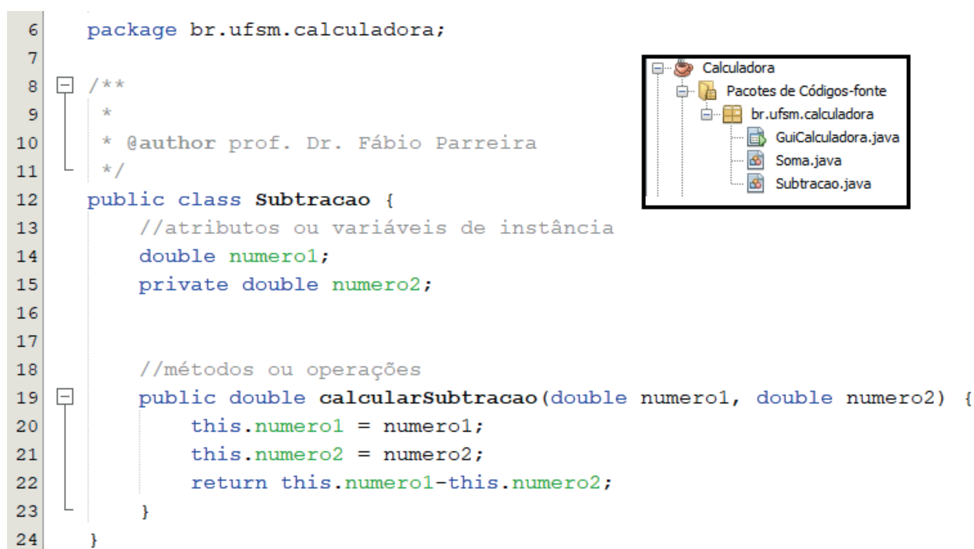
Retornando à análise da Figura 36, não é possível saber o valor da variável *numero2*, na classe subtração, pois o mesmo está declarado como privado; logo, não é visível. Já o método *CalcularSubtracao* possui visibilidade pública; portanto, é visível, ou seja, é acessível por meio de *Subtração.CalcularSubtracao(num1, num2)*.

Escopo de Variáveis

Definimos como escopo o tempo de vida de uma variável. Na prática, dizemos que são os locais em que a variável pode ser acessada. Em Java, o escopo de uma variável depende do local em que ela foi declarada, ou seja, em qual bloco ela foi declarada, sendo que um bloco é delimitado por chaves. Nesse sentido, a variável é criada durante o primeiro acesso do interpretador ao bloco de código, e quando finalizar a execução deste bloco a variável é destruída automaticamente. Logo, se uma variável for criada dentro de um determinado método, quando a execução do método iniciar a variável é criada e, ao finalizar a execução do mesmo, a variável é destruída. Por outro lado, quando uma variável é declarada no começo da classe, sendo declarada fora dos métodos, esta variável pode ser acessada por todos os métodos da classe e será destruída ao finalizar a execução da classe. A estas variáveis daremos o nome de atributos da classe.

Continuado o exemplo da calculadora vamos criar uma classe que seja responsável por fazer a subtração de dois números. Após a criação da classe, digite o código da classe Subtração, conforme apresentado graficamente na Figura 44.

Figura 44 – Classe Subtração em Java



Fonte: Autores.

Agora vamos analisar o escopo das variáveis na classe *Subtracao*, apresentadas na Figura 44. Observe que nas linhas 14 e 15 foram declarados os atributos da classe, denominados de *numero1* e *numero2*. Logo, estes atributos possuem escopo em todas as partes da classe; por isso, eles são acessados por todos os métodos pertencentes à classe. Neste caso, temos apenas o método *calcularSubtracao*. Relembrando, o uso da palavra reservada *this*, que é utilizado para fazer uma autorreferência ao próprio contexto em que a classe se encontra, serve para referenciar os atributos da própria classe, diferenciando os atributos da classe com as variáveis locais, que também foram declaradas como *numero1* e *numero2*. Vale ressaltar que usamos a palavra *this* quando referenciamos um atributo que tem o escopo delimitado em toda a classe. Por outro lado, não se usa *this* em variáveis locais, declaradas dentro de métodos. Nas linhas 20, 21 utilizamos o *this.numero1=numero1* e *this.numero2=numero2*, logo, é atribuído o valor passado por parâmetro para as variáveis locais *numero1* e *numero2* para *this.numero1* e *this.numero2*. Na linha 22, é feita a subtração destes valores, retornando o resultado.

2.3
OBJETOS

Agora que entendemos a definição de objetos no mundo real, vamos entender a definição computacional de um objeto. Do ponto de vista de um *software*, o objeto é um módulo ou construção de software que agrupa ao mesmo tempo o **estado** (que são os dados) e o **comportamento** (que são as ações) – juntos, representam uma abstração de um objeto do mundo real (físico ou conceitual) (BARKER, 2005). A seguir, serão detalhados o estado e o comportamento de um objeto.

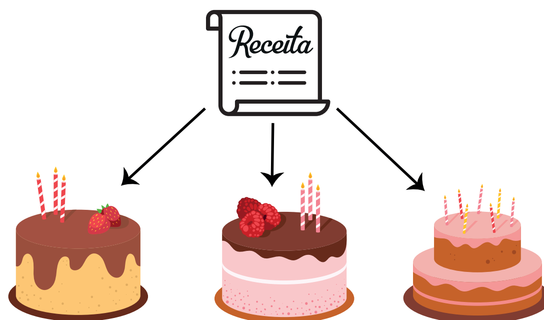
O estado de um objeto é definido pelos valores dos seus atributos, quando analisados coletivamente. Retornando ao exemplo da classe *Soma*, se quiséssemos somar 4 + 6, o estado desse objeto seria *numero1=4* e *numero2=6*. Porém, antes de definir o estado de um objeto é preciso instanciá-lo, explicado a seguir. Por outro lado, o comportamento (ou as **operações**) trata-se especificamente das ações que um objeto pode fazer para acessar, modificar ou manter os valores dos atributos.

Instanciação

O termo instanciação é usado para referir-se ao processo pelo qual um objeto é criado na memória do computador, em tempo de execução, com base em uma definição de classe. Vale ressaltar que, a partir de uma única definição de classe, podemos criar muitos objetos com a mesma estrutura dos atributos e comportamentos idênticos. Embora a estrutura dos atributos seja a mesma, os valores fornecidos aos atritos são diferentes. Sendo assim, uma classe, que tem a função de definir os recursos (atributos e métodos), deve ser instanciada para que possamos utilizá-la. Isso significa que devemos definir um objeto de referência para a classe e por meio desse objeto acessar os recursos necessários.

Para ilustrar, imagine que a classe seja uma receita (modelo) e o objeto é o próprio bolo, que foi criado a partir da receita. Ou seja, a instanciação da classe receita resulta em um ou mais objetos bolo, conforme apresentado na Figura 45.

Figura 45 – Instanciação



Fonte: Autores.

Agora, falando dos trâmites de instanciação em Java, temos dois passos:

- criação da variável de referência;
- atribuição do novo objeto à variável de referência.

Inicialmente, a criação da variável de referência corresponde à declaração da variável para um tipo de dados, definido pelo usuário, por exemplo, a variável soma declarada abaixo:

- *Soma soma*;

É importante notar que esta variável não possui nenhum valor, pois até o momento não foi atribuído nenhum valor de referência a ela; apenas declaramos uma variável de referência do tipo *Soma* chamada *soma*. Lembre-se que Java é *case sensitive*, difere maiúsculo de minúsculo, portanto *Soma* e *soma* são duas variáveis distintas.

A variável de referência, a *soma*, tem potencial para referenciar um objeto da classe *Soma*, mas ainda não foi atribuído nenhum objeto a ela. Nesse momento, o valor da variável soma é indefinido, até que seja explicitamente atribuído um valor. Para instanciar um novo objeto da classe *Soma*, para a variável *soma*, temos que usar a palavra reservada em Java, *new*, que aloca um novo objeto *Soma* dentro da JVM (Máquina Virtual Java). Na sequência, que corresponde ao segundo passo, temos que associar o novo objeto à variável de referência *soma* através de atribuição, como segue:

- *soma = new Soma( );*

Toda vez que quisermos acessar as informações do objeto recém-criado vamos utilizar a variável *soma*, pois ela faz esse controle. Outra maneira de criar e inicializar uma variável de referência é combinar as duas etapas apresentadas anteriormente, em uma única linha de código. Logo, podemos declarar a variável de referência e instanciar o objeto para essa variável, unindo as duas linhas, apresentadas anteriormente, veja:

- *Soma soma = new Soma( );*

Retomando o nosso exemplo da calculadora, vamos criar o código da classe *GuiCalculadora*, com a finalidade de resolver a equação: 3 + 3 = 6. Para resolver esta operação, vamos precisar declarar duas variáveis do tipo *double* para armazenar os números e um atributo do tipo *String* para armazenar a operação, conforme apresentado nas linhas 14 e 15, da Figura 46. Observe que esses dois atributos estão declarados logo abaixo do nome da classe, fora dos métodos; por isso, possuem visibilidade em toda a classe.

Figura 46 – Declaração das variáveis de classe – classe *GuiCalculadora*

```
1  ...5 linhas
6  package br.ufsm.calculadora;
7
8
9  /**
10  *
11  * @author prof. Dr. Fábio Parreira
12  */
13 public class GuiCalculadora extends javax.swing.JFrame {
14     private String operacao="";
15     private double numero1, numero2;
```

Fonte: Autores.

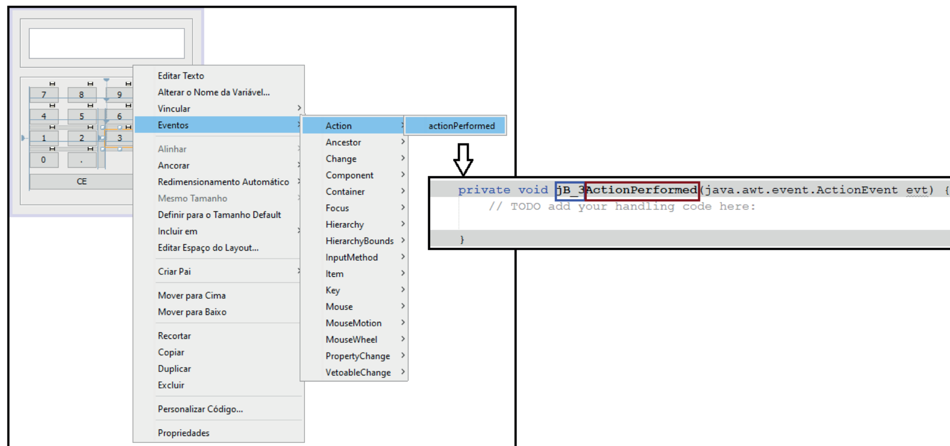
Já criamos os atributos para armazenar o primeiro número, a operação e o segundo número; agora vamos programar os eventos dos botões envolvidos. Por definição, um evento é uma mensagem que um objeto dispara em determinadas situações. Para que o evento funcione, temos que criar um método para ser executado toda vez que o evento for disparado. Existem vários eventos disponíveis em Java, vamos trabalhar com o evento *"actionPerformed"*, responsável por tomar uma ação, caso um evento ocorra. Sendo assim, a interface fica à espera de algum evento, e caso ocorra ele é imediatamente passado para *"actionPerformed"*. Por exemplo, a interface fica à espera de um clique do mouse no botão3; caso ocorra o clique, a ação é passada imediatamente para o evento *"jB_3ActionPerformed....."*. A Figura 47 apresenta as etapas para criar o evento supracitado:

1. Clique com o botão direito sobre o botão, com o texto "3";
2. Vá em Eventos;
3. *Action*;
4. *ActionPerformed*.

ATENÇÃO: pesquise mais sobre mensagens entre objetos.

Ao final, é criado um método, que será acionado toda vez que o usuário clicar no botão com texto "3". Veja que o nome do método é uma junção do nome atribuído à variável botão, que foi *"jB_3"* com o nome do evento *"actionPerformed"*, conforme representado na Figura 47.

Figura 47 – Evento *ActionPerformed*



Fonte: Autores.

Para esse evento recém-criado, vamos atribuir a ação de pegar o conteúdo que está armazenado no visor, *"jTF_Visor"*, e concatenar com o texto do botão, que neste caso é *"3"*. O código do botão *"jB_3"* está apresentado na Figura 48. Na linha 264, é feita a junção (concatenação) das informações contidas no visor, *"jTF_Visor"*, com a informação contida no texto no botão,"3".

Figura 48 – Código para o botão *jB_3*

```
262    private void jB_3ActionPerformed(java.awt.event.ActionEvent evt) {
263        // TODO add your handling code here:
264        jTF_Visor.setText(jTF_Visor.getText()+jB_3.getText());
265    }
```

Fonte: Autores.

Após programar o código para o botão *jB_3*, que é o primeiro número da equação proposta, vamos escolher a operação. Neste caso será a soma, que deve ser realizada toda vez que o usuário clicar no botão soma (*jB_Soma*), lembrando que para acionar as ações deste botão temos que criar um evento *ActionPerformed*, conforme apresentado graficamente na Figura 49.

Figura 49 – Código para o botão *jB_Soma*

```
267    private void jB_SomaActionPerformed(java.awt.event.ActionEvent evt) {
268        // TODO add your handling code here:
269        numero1 = Double.parseDouble(jTF_Visor.getText());
270        jTF_Visor.setText("");
271        operacao=jB_Soma.getText();
272    }
```

Fonte: Autores.

Analisando o código da Figura 49, na linha 269, faz-se a conversão de *texto* para *double* das informações contidas no Visor da calculadora, *"jTF_Visor.getText( )"* e atribui-se ao atributo *numero1*. Levando em consideração que o visor só funcio-

na com o tipo *String* e o atributo que irá receber a informação é do tipo *double*, obrigatoriamente temos que converter a informação de *String* para *double*. A conversão é feita usando o código "*Double.parseDouble( )*", ao final, é atribuído o conteúdo lido do visor, já convertido para *double*, ao atributo *numero1*. Na linha 270, é feita a limpeza do conteúdo do visor, atribuindo nulo a ele. Na linha 271, é atribuído ao atributo "*operacao*", que é do tipo *String*, o texto do "*jB_Soma*".

Veja que o conteúdo do "*jTF_Visor*" foi completamente apagado, deixando-o apto para receber o segundo número, que também é o número 3. A seguir teremos que pensar na próxima ação, que o sinal de igual (=). Para isso, temos que criar um novo evento para o botão "*jB_Igual*" e inserir as linhas de código, conforme apresentado na Figura 50.

Figura 50 – Código para o botão *jB_ Igual – Operação de soma*

```
274   private void jB_IgualActionPerformed(java.awt.event.ActionEvent evt) {
275       // TODO add your handling code here:
276       //Cria uma variáve de referência do tipo Soma
277       //e instancia o objeto (new Soma())
278       Soma  soma = new Soma();
279       double resultado;//Recebe o resultaod a operação
280       numero2 = Double.parseDouble(jTF_Visor.getText());
281
282       switch(operacao){
283           case "+":
284               resultado = soma.calcularSoma(numero1, numero2);
285               jTF_Visor.setText(String.valueOf(resultado));
286               break;
287           case "-":
288
289
290
291               break;
292           case "/":
293               break;
294           case "*":
295               break;
296       }
297   }
```

Fonte: Autores.

Vamos analisar cada linha de código apresentada na Figura 50, a começar na linha 278. Nela, foi criada uma variável de referência para a classe *Soma* e atribuída a instanciação do objeto do mesmo tipo (*new Soma()*). Vale ressaltar que por meio desse objeto teremos acesso ao método, da classe *Soma*, para fazer a operação *soma* entre dois números. Na linha 279, foi declarada uma variável chamada *resultado*, do tipo *double*, para receber o resultado da operação, que em nosso exemplo é a soma. Na linha 280, pegamos o segundo número digitado em *jTF_Visor*, convertemos para *double* e atribuímos à *numero2*. Entre as linhas 282 a 296, foi inserido um *switch..case* para selecionarmos qual das quatro operações básicas (+ soma,- subtração,/ divisão,* multiplicação) foi escolhida pelo usuário, lembrando que em nosso exemplo estamos trabalhando com a adição. Caso a operação solicitada seja a soma "+", na linha 283, temos de chamar, na linha 284, o método *calcularSoma*, por meio do objeto *soma*, passando como parâmetros o *numero1* e

numero2. O resultado dessa ação é atribuído a *resultado*. Na linha 285, atribuímos ao *jTF_Visor* o conteúdo da variável resultado, lembrando que resultado é do tipo *double*, portanto precisa ser convertido para *String*. Por fim, o comando *break*, na linha 286, força a saída do *switch*. Pronto! Resolvemos nossa equação 3 + 3 = 6, execute o programa e você irá obter o resultado esperado.

Passando para a próxima operação, a subtração, vamos programar nossa calculadora para fazer a equação 33 – 3 = 30. Para resolver, temos que inserir um novo evento para o botão *jB_Subracao*, conforme apresentado na Figura 51. Veja que é bem semelhante ao código inserido para o evento do botão *jB_Soma*.

Figura 51 – Código para o botão *jB_Subtracao*

```
299     private void jB_SubracaoActionPerformed(java.awt.event.ActionEvent evt) {
300         // TODO add your handling code here:
301         numero1 = Double.parseDouble(jTF_Visor.getText());
302         jTF_Visor.setText("");
303         operacao=jB_Subracao.getText();
304     }
```

Fonte: Autores.

Vamos declarar o objeto subtração fora dos métodos, logo ele será um atributo da classe e terá visibilidade em toda a classe, conforme apresentado graficamente na Figura 52. Na linha 16, é apresentada a criação do atributo e a instanciação do objeto do tipo *Subtracao*.

Figura 52 – Criação e instanciação do objeto *subtracao*

```
13  public class GuiCalculadora extends javax.swing.JFrame {
14      private String operacao="";
15      private double numero1, numero2;
16      Subtracao subtracao = new Subtracao();
```

Fonte: Autores.

O próximo passo é inserir o evento do botão *jB_Igual*, programando o código para a subtração. Veja como ficou na Figura 53. As ações ocorrem nas linhas 288 a 290, observem que economizamos a variável *resultado*.

Figura 53 – Código para o botão *jB_ Igual – Operação de subtração*

```
274  private void jB_IgualActionPerformed(java.awt.event.ActionEvent evt) {
275      // TODO add your handling code here:
276      //Cria uma variáve de referência do tipo Soma
277      //e instancia o objeto (new Soma())
278      Soma  soma = new Soma();
279      double resultado;//Recebe o resultaod a operação
280      numero2 = Double.parseDouble(jTF_Visor.getText());
281
282      switch(operacao){
283          case "+":
284              resultado = soma.calcularSoma(numero1, numero2);
285              jTF_Visor.setText(String.valueOf(resultado));
286              break;
287          case "-":
288              jTF_Visor.setText(
289                  String.valueOf(subtracao.calcularSubtracao(numero1, numero2))
290              );
291              break;
292          case "/":
293              break;
294          case "*":
295              break;
296      }
297  }
```

Fonte: Autores.

Relacionamentos

Até o presente momento, estudamos as classes de forma independente. Em estudos na disciplina de Engenharia de Software é possível perceber que existem determinados relacionamentos entre as classes. Por exemplo, existe uma relação entre a calculadora e a operação de somar e essa relação deve ser representada no diagrama de classe. Como supracitado no exemplo, geralmente, as classes não estão isoladas, elas se relacionam entre si; logo, dizemos que o relacionamento e a comunicação definem responsabilidades da classe.

Ao estudar a UML, podemos perceber que existem três tipos de relações que são bem comuns. São elas:

- associação,
- dependência,
- generalização (ou herança).

A seguir, vamos detalhar a associação e a dependência, sendo que a herança será detalhada posteriormente. A **associação** é um relacionamento estrutural entre instâncias e especifica que objeto de uma classe está ligado ao objeto de outra classe. Sendo assim, a associação entre dois objetos ocorre quando eles são completamente independentes entre si, mas eventualmente estão relacionados. Logo o objeto da classe a ser utilizado é construído no mesmo instante em que a classe, por isso tal objeto é um atributo da classe, sendo declarado fora dos métodos, conforme apresentado na Figura 54. Veja que *subtracao* foi declarado e instanciado no corpo da classe, na linha 16.

Figura 54 – Relação de associação

```
9   /**...4 linhas */
13  public class GuiCalculadora extends javax.swing.JFrame {
14      private String operacao="";
15      private double numero1, numero2;
16      Subtracao subtracao = new Subtracao();
```

Fonte: Autores.

Já a dependência é um relacionamento de utilização no qual uma mudança na especificação de um elemento pode alterar a especificação do elemento dependente. A dependência entre classes indica que os objetos de uma classe usam serviços dos objetos de outra classe, sendo assim, é um relacionamento no qual a mudança de uma classe pode afetar o comportamento ou estado de outra classe. As dependências podem ser usadas no contexto de classes, para mostrar que uma classe utiliza outra classe como argumento na assinatura de uma operação ou cria uma instância dentro de um método. Na Figura 55, linha 278, é instanciado um objeto do tipo *Soma*, veja que ele foi criado dentro do método para executar a ação de calcular a soma entre dois números, ao finalizar a execução do método tal objeto será destruído.

Figura 55 – Relação de dependência

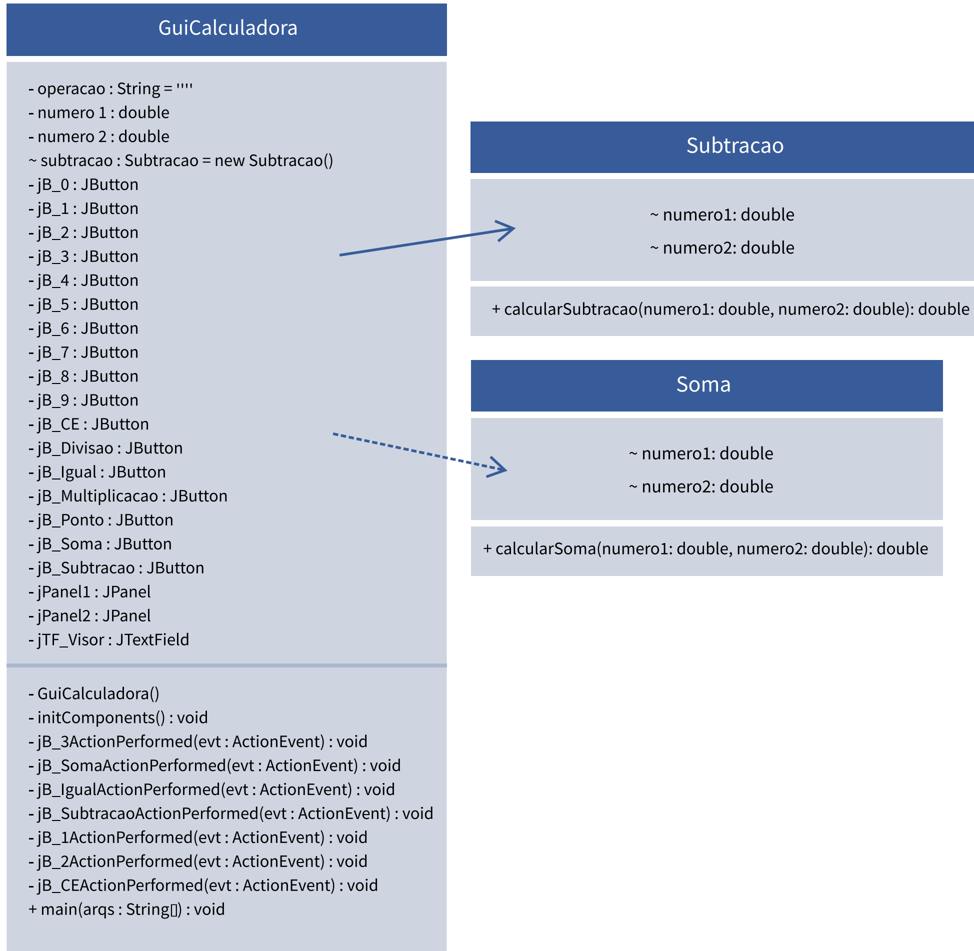
```
274     private void jB_IgualActionPerformed(java.awt.event.ActionEvent evt) {
275         // TODO add your handling code here:
276         //Cria uma variável de referência do tipo Soma
277         //e instancia o objeto (new Soma())
278         Soma  soma = new Soma();
279         double resultado;//Recebe o resultaod a operação
280         numero2 = Double.parseDouble(jTF_Visor.getText());
281
282         switch(operacao){
283             case "+":
284                 resultado = soma.calcularSoma(numero1, numero2);
285                 jTF_Visor.setText(String.valueOf(resultado));
286                 break;
287             case "-":
288                 jTF_Visor.setText(
289                     String.valueOf(subtracao.calcularSubtracao(numero1, numero2))
290                 );
291                 break;
292             case "/":
293                 break;
294             case "*":
295                 break;
296         }
297     }
```

Fonte: Autores.

O diagrama de classe que representa as relações entre as classes *GuiCalculadora*, *Soma* e *Subtracao* é apresentado, graficamente, na Figura 56. A relação de associação é representada graficamente por uma linha contínua, ligando as duas classes, apontando com uma seta para a classe que a outra depende. Essa relação ocorre entre as classes *GuiCalculadora* e *Subtracao*. Já a dependência é representada graficamente por uma linha tracejada, ligando as duas classes, com uma seta apon-

tando para a classe que a outra depende. Relação de dependência ocorre entre as classes *GuiCalculadora e Soma*.

Figura 56 – Diagrama de classes: *GuiCalculadora, Soma* e *Subtracao*



Fonte: Autores.

Após a implementação de todo o código explicado na unidade 2, execute o projeto e teremos uma calculadora que faz as operações de soma e subtração.

ATIVIDADES - UNIDADE 2

Todas as atividades abaixo devem ser postadas no Moodle/UAB-UFSM, conforme direcionamento do professor da disciplina.

1. O exemplo da calculadora, explanado anteriormente, ficou incompleto. Vamos finalizá-lo. Para isso faça:

a) crie o código para todos os botões numéricos (0,1,2,4,5,6,7,8 e 9);
b) implemente a operação de divisão;
c) implemente a operação de multiplicação;
d) implemente a inserção de ponto flutuante (por exemplo, 3.5);
e) implemente o código para limpar o display (visor da calculadora), no botão CE.

3

INTERAÇÕES ENTRE OBJETOS

INTRODUÇÃO

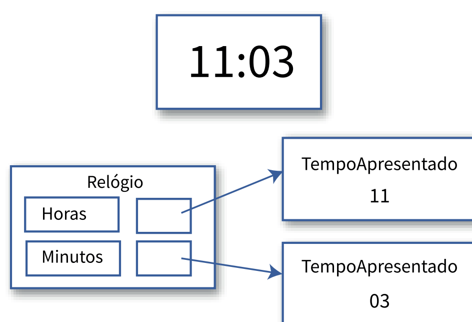
Em programação orientada a objetos, toda a interação entre os objetos acontece por meio de recebimento/envio de mensagens. Por sua vez, as mensagens são geradas por meio dos métodos, que podem ser traduzidos como sendo as ações realizadas pelo objeto.

Cabe ressaltar que cada objeto tem uma estrutura e ações específicas. Tais ações complementam umas às outras na realização da missão geral do sistema computacional. Para que possamos construir a interação entre objetos, faz-se necessário abordar os seguintes conteúdos:

- declarando métodos;
- construtores;
- sobrecarga de método;
- visibilidade das informações;
- métodos de acesso a atributos (*getters* e *setters*).

Nesta unidade, para exemplificar a interação entre objetos vamos construir um protótipo que simula um relógio digital, conforme apresentado na Figura 57. Nesse exemplo, vamos trabalhar com todos os conceitos supracitados. Basicamente, o protótipo terá duas classes, uma classe para configurar a hora e o minuto, denominada de *TempoApresentad*o, e a outra para guardar as informações dos objetos instanciados com os padrões de horas e outro com as configurações dos minutos.

Figura 57 – Relógio digital



Fonte: Autores.

Para abarcar todo o conteúdo supracitado na Unidade 3, abordaremos a declaração dos métodos, dando ênfase aos assuntos de assinatura, passagem de argumentos e invocação de métodos. A seguir, serão abordados os construtores, destacando as classes com e sem construtor. Depois falaremos sobre sobrecarga de métodos e construtores e, por fim, dos métodos *get* e *set*. Ao final desta unidade, você será capaz de construir um simulador de relógio digital, empregando os conceitos supracitados.

3.1
DECLARANDO MÉTODOS

Embora já construímos alguns métodos anteriormente, agora vamos tratar de como especificar formalmente o comportamento dos objetos em Java, ou seja, os métodos. A sintaxe do Java usada na declaração de um método está dividida em cinco partes (DEITEL, 2015):

- modificador;
- valor de retorno;
- nome do método;
- parâmetros;
- corpo do método.

O modificador, que é opcional, especifica a acessibilidade do método, caso ele seja declarado, pode ser uma combinação dos modificadores de acesso: *public*, *protected* ou *private*; *abstract* ou *final*; e ainda *static*. O valor de retorno indica tipo que será retornado. Neste caso, quando o método não possuir um valor de retorno devemos usar a palavra reservada *void*. Já o nome do método é usado para referenciá-lo durante as chamadas do método. Os parâmetros são opcionais, podendo ser um ou mais parâmetros, separados por vírgulas, onde cada um obedece à forma, tipo e nome, tal como na declaração de variáveis. Os parâmetros são utilizados para receber os valores (argumentos) fornecidos ao método durante a chamada. O corpo do método abarca todo o código pertencente a ele, sendo delimitado por chaves de abertura ({) e fechamento (}). Após relembrar a sintaxe dos métodos em Java, vamos abordar alguns conceitos essenciais, a começar pela assinatura do método.

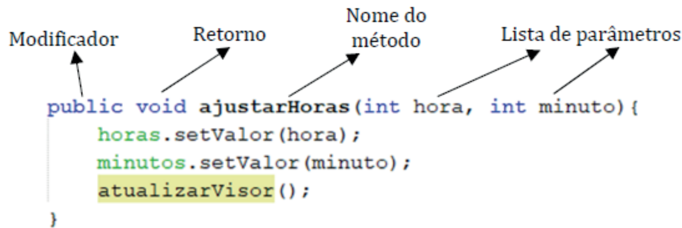
Assinatura do método

A assinatura, também conhecida como cabeçalho ou interface, é uma maneira de identificar um método de forma única, além de ser uma especificação formal de como o método é invocado. Uma assinatura de método pode consistir em:

- o valor de retorno;
- nome;
- lista opcional de parâmetros.

Cabe ressaltar que, em Java, a assinatura dos métodos pode ter o mesmo valor de retorno e mesmo nome, fazendo distinção na lista de parâmetros. Nesse sentido, caso todos os parâmetros sejam iguais, teremos o mesmo método; se apenas um desses parâmetros for diferente, então teremos um método diferente. A Figura 58 apresenta a assinatura do método *ajustarHoras( )*. Nesse método foi incluído o modificador, o tipo de retorno, o nome do método e a lista de parâmetros.

Figura 58 – Exemplo de método



Fonte: Autores.

Passando argumentos para os métodos

Quando passamos argumentos para um método, geralmente, temos dois propósitos: o primeiro é fornecer aos métodos as informações necessárias para que eles façam o trabalho. O segundo propósito é direcionar o comportamento do método. No que se refere à passagem de argumentos, temos que nos ater a dois conceitos: parâmetro e argumento.

Algumas vezes utilizamos os termos parâmetro e argumento como se fossem sinônimos, mas na verdade são termos que definem situações diferentes. Veja:

- **parâmetro**: é uma variável declarada no cabeçalho do método, portanto de escopo local, cujo tempo de vida compreende enquanto um método está sendo executado.
- **argumento:** é o valor, originado de uma variável ou expressão, que é passado para o método.

Levando em consideração as definições supracitadas, podemos concluir que nós não passamos parâmetros para os métodos, passamos argumentos. Logo, um método recebe argumentos nos respectivos parâmetros. A Figura 59, letra a, apresenta a declaração do construtor da classe Relógio com dois parâmetros declarados. Na letra b, da Figura 59, temos a chamada do construtor passando os argumentos 3 e 40.

Figura 59 – Exemplos de parâmetro e argumento

a)
```
public Relogio(int hora, int minuto){//declaração de dois parâmetros

}
```

b)
```
Relogio relogio = new Relogio(3,40);//chamou com 2 argumentos
```

Fonte: Autores.

No item 3.2, vamos detalhar mais sobre os construtores. - Invocando métodos

Em Java, para invocarmos um método de um objeto, obrigatoriamente, precisamos criar a instância deste objeto e, em seguida, invocar os métodos que são conhecidos para o objeto. Neste sentido, normalmente, a chamada do método acontece prefixado o nome da variável de referência, que representa o objeto que deve executar o método, com um ponto (.). A Figura 60 ilustra a criação das variáveis de referência horas na linha 16, e minutos na linha 17. Na sequência, na linha 26 é instanciado o objeto horas e na linha 28 o objeto minutos.

Figura 60 – Instância dos objetos horas e minutos

```
16      private TempoApresentado horas; //Variável de referência
17      private TempoApresentado minutos;//Variável de referência
18      private String horasMinutos;    //Armazena horas e minutos do relógio
19
20      /**
21       * Construtor para objetos Relogio. Este construtor cria um novo
22       * objeto com horario em 00:00
23       */
24      public Relogio(){
25          //horas = new TempoApresentado(24);//Instancia objeto horas
26          horas = new TempoApresentado();//Instancia objeto horas
27          horas.setLimete(24);
28          minutos = new TempoApresentado();//Instancia objeto minutos
29          minutos.setLimete(60);
30          atualizarVisor();
31      }
```

Fonte: Autores.

Após instanciar os objetos, horas e minutos, já podemos ter acesso aos métodos desse objeto. Conforme apresentado na Figura 61, temos as chamadas dos métodos por meio dos objetos criados anteriormente.

Figura 61 – Chamada

```
51      public void incrementoTempo(){
52          minutos.incremento();//acesso ao método incremento
53          if(minutos.getValor() == 0) {//acesso ao método getValor
54              horas.incremento();//acesso ao método incremento
55          }
56          atualizarVisor();
57      }
```

Fonte: Autores.

Ao analisar a linha 52, da figura 61, podemos interpretar essa linha de código como sendo o ato de invocar o método *incremento* no objeto *minutos*. Outra forma de interpretar esse código seria como uma solicitação do objeto *minutos* para executar o método *incremento*, que é uma espécie de serviço oferecido pelo objeto.

3.2 CONSTRUTORES

Em Java, os construtores de uma classe determinam quais as ações que serão executadas quando um novo objeto é criado. O construtor deve ter o mesmo nome da classe e, obrigatoriamente, não pode possuir indicação do tipo de retorno, nem mesmo *void*. Cabe ressaltar que quando instanciamos um objeto através da palavra reservada *new*, estamos na verdade invocando um tipo especial de procedimento chamado construtor. Ressaltamos que invocar um construtor serve como um pedido para a *JVM* construir (ou instanciar) um objeto totalmente novo em tempo de execução, alocando memória suficiente para armazenar todas as informações deste objeto. A Figura 62 apresenta a sintaxe para invocar um construtor da classe *TempoApresentado*.

Figura 62 – Sintaxe de chamada de construtores

```
private TempoApresentado horas; //Variável de referência
private TempoApresentado minutos;//Variável de referência


horas = new TempoApresentado();//Instancia objeto horas

minutos = new TempoApresentado();//Instancia objeto minutos
```

Fonte: Autores.

Classe sem construtor explícito

Quando não declararmos explicitamente o construtor para uma classe, a linguagem *Java* fornece automaticamente um construtor padrão para a classe. O construtor padrão é bastante simples, por isso ele não possui parâmetros, isto é, não recebe argumentos. Além disso, faz o mínimo necessário para inicializar um novo objeto, ou seja, define todos os atributos para seus valores padrão equivalentes a zero. Veja que na classe *TempoApresentado* não existe um construtor declarado explicitamente, conforme mostra a Figura 63.

Figura 63 – Classe sem construtores explícitos

```
1    package br.ufsm.relogio;
2
3
4    public class TempoApresentado{
5        private int limite;
6        private int valor;
7
8  ⊞     /** Retorna o conteúdo da variável valor ...3 linhas */
11 ⊟     public int getValor(){
12           return valor;
13       }
14
15 ⊞     /** Retorna o valor formatado em duas casas decimais para se
20 ⊟     public String getValorVisor(){
21           if(valor < 10)  {
22               return "0" + valor;
23           }
24           else {
25               return "" + valor;
26           }
27       }
28
29 ⊞     /** Define o valor para ser apresentado no mostrador do reló
33       public void setValor(int atualizaValor)
34 ⊟     {
35           if((atualizaValor >= 0) && (atualizaValor < limite)) {
36               valor = atualizaValor;
37           }
38       }
39
40 ⊞     /** Define o valor do limite ...3 linhas */
43 ⊟     public void setLimete(int limite){
44           this.limite = limite;
45       }
46
47 ⊞     /** Incrementa o valor de exibição em um, passando para zero
51 ⊟     public void incremento(){
52           valor = (valor + 1) % limite;
53       }
54   }
```

Fonte: Autores.

Embora não haja um construtor, é possível fazer uso do construtor padrão, conforme mostra o exemplo da Figura 62.

Classe com construtor explícito

Podemos criar os nossos próprios construtores; sendo assim, não precisamos depender do Java para fornecer um construtor padrão para cada uma de nossas classes. Com isso, podemos criar construtores mais adequados às particularidades de cada uma das classes. A seguir, vamos detalhar a sintaxe para um construtor, que

é um pouco diferente do método. Veja:

- O nome do construtor deve ser exatamente igual ao nome da classe, para a qual estamos escrevendo o construtor – essa é uma definição do Java, não temos escolha;
- A lista de parâmetros deve ser fornecida no cabeçalho do construtor e, caso não sejam necessários parâmetros, pode ser deixada vazia;
- Não podemos especificar o tipo de retorno para um construtor; por definição, um construtor retorna uma referência a um objeto recém-criado, do tipo representado pela classe a que o construtor pertence.

Voltando ao exemplo do construtor da classe *Relogio*, conforme apresentado na Figura 64, temos um construtor que não recebe nenhuma informação, mas foram alterados os valores dos atributos, adequando-os desta forma à necessidade da classe.

Figura 64 – Classe com construtores explícitos

```
15    public class Relogio{
16        private TempoApresentado horas; //Variável de referência
17        private TempoApresentado minutos;//Variável de referência
18        private String horasMinutos;    //Armazena horas e minutos do relógio
19
20 +      /** Construtor para objetos Relogio ...4 linhas */
24 -      public Relogio(){
25            //horas = new TempoApresentado(24);//Instancia objeto horas
26            horas = new TempoApresentado();//Instancia objeto horas
27            horas.setLimete(24);
28            minutos = new TempoApresentado();//Instancia objeto minutos
29            minutos.setLimete(60);
30            atualizarVisor();
31        }
32
```

Fonte: Autores.

Uma classe pode ter um ou mais construtores, desde que a assinatura seja diferente. A seguir, veremos com inserir outro construtor na classe *Relogio*.

3.3
SOBRECARGA DE MÉTODO/ CONSTRUTOR

A sobrecarga de métodos/construtores é um mecanismo bastante utilizado na linguagem Java que permite a criação de dois ou mais métodos com o mesmo nome, pertencentes à mesma classe, desde que tenham assinaturas diferentes. Relembrando, para que a assinatura de dois métodos seja diferente, embora os nomes sejam idênticos, os argumentos de cada método têm que ser diferentes. A Figura 65 apresenta uma sobrecarga de construtores *Relogio*.

Vamos entender o motivo de termos sobrecarregado os construtores. Inicialmente, foi criado o construtor, na linha 24, sem parâmetros. Esse construtor cria um novo objeto com horário definido em 00:00, ao longo dos testes do protótipo surgiu a necessidade de criar um objeto relógio com horas e minutos previamente definidos pelo usuário; por isso, foram inseridos os dois parâmetros no construtor, na linha 38. Ao final, temos dois construtores de nomes idênticos, mas com assinaturas diferentes.

Figura 65 – Sobrecarga de construtor

```
20  ⊞   /** Construtor para objetos Relogio ...4 linhas */
24  ⊟   public Relogio(){
25          //horas = new TempoApresentado(24);//Instancia objeto horas
26          horas = new TempoApresentado();//Instancia objeto horas
27          horas.setLimete(24);
28          minutos = new TempoApresentado();//Instancia objeto minutos
29          minutos.setLimete(60);
30          atualizarVisor();
31      }
32
33  ⊞   /** Construtor para objetos Relogio ...5 linhas */
38  ⊟   public Relogio(int hora, int minuto){//declaração de dois parâmetros
39          horas = new TempoApresentado();//Instancia objeto horas
40          horas.setLimete(24);
41          minutos = new TempoApresentado();//Instancia objeto minutos
42          minutos.setLimete(60);
43          ajustarHoras(hora, minuto);
44      }
```

Fonte: Autores.

Continuando a análise sobre os construtores, da linha 25 e 38, notamos a diferença na assinatura dos construtores; portanto, os dois são válidos.

3.4 MÉTODOS *GET* E *SET*

Basicamente, o que distingue uma classe bem projetada de uma mal projetada é o grau de ocultamento das informações internadas, por exemplo, os atributos que geralmente possuem visibilidade *private*. Desta forma, a comunicação entre os objetos acontece por meio de interface, sem que haja necessidade de conhecer o funcionamento interno das outras classes. Em Java, boas práticas de programação exigem que sejam criados métodos de acesso público que possibilitem aos clientes de outro objeto manipular os atributos privados. Compreendemos por manipular o ato de ler ou modificar valores.

O código a seguir, na Figura 66, extraído da classe *TempoApresentado*, ilustra os métodos conhecidos como métodos *get* e *set*, os métodos *get* são utilizados para leituras e os métodos *set* para escrever informações.

Figura 66 – Métodos *get*/ *set*

```
4   public class TempoApresentado{
5       private int limite;
6       private int valor;
7
8  +    /** Retorna o conteúdo da variável valor ...3 linhas */
11 -    public int getValor(){
12          return valor;
13      }

29 +    /** Define o valor para ser apresentado no mostrador do relógio
33 -    public void setValor(int atualizaValor){
34          if((atualizaValor >= 0) && (atualizaValor < limite)) {
35              valor = atualizaValor;
36          }
37      }
```

Fonte: Autores.

Analisando o trecho do código apresentado na Figura 66, temos a declaração dos atributos nas linhas 5 e 6 como sendo *private*, portanto, não são acessíveis externamente. Para permitir o acesso aos atributos de maneira controlada, vamos criar dois métodos, um que retorna o valor (*get*) e outro que altera o valor (*set*). Para dar nomes a esses métodos, temos uma convenção, colocamos a palavra *get* ou *set* antes do nome do atributo. Por exemplo, na classe *TempoApresentado*, temos os atributos limite e valor, desejamos dar acesso de leitura e escrita ao atributo valor, por isso na linha 11 temos o método *getValor()* e na linha 33 o método *setValor(int atualizaValor)*.

ATIVIDADES - UNIDADE 3

Todas as atividades abaixo devem ser postadas no Moodle/UAB-UFSM, conforme direcionamento do professor da disciplina.

1) Implemente um programa, utilizando interface gráfica, para implementar um protótipo de um relógio digital (BARNES; KOLLING, 2008), conforme explicado na unidade 3. As duas classes principais estão apresentadas abaixo, implemente a interface gráfica:

Figura 67 – Classe Relogio

```
1   package br.ufsm.relogio;
2   
3   public class Relogio{
4       private TempoApresentado horas; //Variável de referência
5       private TempoApresentado minutos;//Variável de referência
6       private String horasMinutos;    //Armazena horas e minutos do relógio
7   
8       /**
9        * Construtor para objetos Relogio. Este construtor cria um novo
10       * objeto com horario em 00:00
11       */
12      public Relogio(){
13          horas = new TempoApresentado();//Instancia objeto horas
14          horas.setLimete(24);
15          minutos = new TempoApresentado();//Instancia objeto minutos
16          minutos.setLimete(60);
17          atualizarVisor();
18      }
19  
20      /**
21       * Construtor para objetos Relogio. Esse construtor cria um
22       * novo relógio ajustando o horário, de acordo com os argumentos
23       * passados
24       */
25      public Relogio(int hora, int minuto){//declaração de dois parâmetros
26          horas = new TempoApresentado();//Instancia objeto horas
27          horas.setLimete(24);
28          minutos = new TempoApresentado();//Instancia objeto minutos
29          minutos.setLimete(60);
30          ajustarHoras(hora, minuto);
31      }
32  
33      /**
34       * Este método deve ser chamado uma vez a cada minuto,
35       * ele incrementa e atualiza o mostrador do relógio em
36       * um minuto.
37       */
38      public void incrementoTempo(){
39          minutos.incremento();//acesso ao método incremento
40          if(minutos.getValor() == 0) {//acesso ao método getValor
41              horas.incremento();//acesso ao método incremento
42          }
43          atualizarVisor();
44      }
45  
```

```
46      /**
47       * Ajusta o horário de acordo com as informações da hora e  minuto
48       */
49      public void ajustarHoras(int hora, int minuto){
50          horas.setValor(hora);
51          minutos.setValor(minuto);
52          atualizarVisor();
53      }
54
55      /**
56       * Retorna o horário no formato HH:MM.
57       */
58      public String getTempo(){
59          return horasMinutos;
60      }
61
62      /**
63       * Atualiza a informação do horário a ser apresentada
64       */
65      private void atualizarVisor(){
66          horasMinutos = horas.getValorVisor() + ":" +
67                              minutos.getValorVisor();
68      }
69  }
```

Fonte: Autores.

Figura 68 – Classe TempoApresentado

```
1   package br.ufsm.relogio;
2
3
4   public class TempoApresentado{
5       private int limite;
6       private int valor;
7
8       /**
9        * Retorna o conteúdo da variável valor
10       */
11      public int getValor(){
12          return valor;
13      }
14
15      /**
16       * Retorna o valor formatado em duas casas decimais para ser apresentado
17       * no mostrador do relógio.Se o valor for menor que dez, será adicionadoum
18       * zero para manter as duas casas decimais.
19       */
20      public String getValorVisor(){
21          if(valor < 10) {
22              return "0" + valor;
23          }
24          else {
25              return "" + valor;
26          }
27      }
28
29      /**
30       * Define o valor para ser apresentado no mostrador do relógio, desde que
31       * ele seja maior ou igual a zero e menor que o limite
32       */
33      public void setValor(int atualizaValor){
34          if((atualizaValor >= 0) && (atualizaValor < limite)) {
35              valor = atualizaValor;
36          }
37      }
38
```

```
39      /**
40       * Define o valor do limite
41       * */
42      public void setLimete(int limite){
43          this.limite = limite;
44      }
45
46      /**
47       * Incrementa o valor de exibição em um, passando para zero se o
48       * limite for atingido.
49       */
50      public void incremento(){
51          valor = (valor + 1) % limite;
52      }
53  }
```

Fonte: Autores.

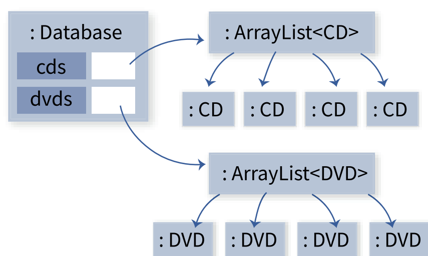
4

AGRUPAMENTO DE OBJETOS

INTRODUÇÃO

Nas unidades anteriores, quando criamos objetos isolados, por exemplo, na calculadora ou no relógio digital, não há necessidade de armazenar as variáveis de referências em uma coleção, pois eram únicas para cada classe, mas quando necessitamos armazenar as informações de vários objetos, como as informações de vários *CDs*, logo teremos: *CD1*, *CD2*, *CD3*,...,*CDn*. Observe que a criação de variáveis individuais para cada objeto é impraticável. Na verdade, nem sabemos a quantidade exata que precisamos criar. Com o intuito de aplicar o agrupamento de objetos vamos construir um protótipo que armazene os dados de uma coleção de *CDs (Compact Disc)* e *DVDs (Digital Video Disc)*, conforme apresentado na Figura 69.

Figura 69 – Coleção de objetos CDs e DVDs



Fonte: (BARNES; KOLLING, 2008, p. 218)

Para resolver este problema, Java fornece uma categoria especial de objeto chamada de coleção. Ela é usada para armazenar e organizar variáveis de referências para outros objetos, conforme apresentado na Figura 69. Para construir o protótipo apresentado na Figura 69, vamos abordar os conceitos:

- Coleções *(Collections Framework)*,
- *ArrayList*,
- Criação de coleção do tipo *CDs* e *DVDs*,
- Criação das classes *Database*, *DVD* e *CD*,
- Formulários gráficos.

Ao final desta unidade, você terá conhecimentos para armazenar uma quantidade ilimitada, a depender da memória do computador, de objetos *DVDs* e CDs por meio de agrupamento de objetos usando a categoria coleções.

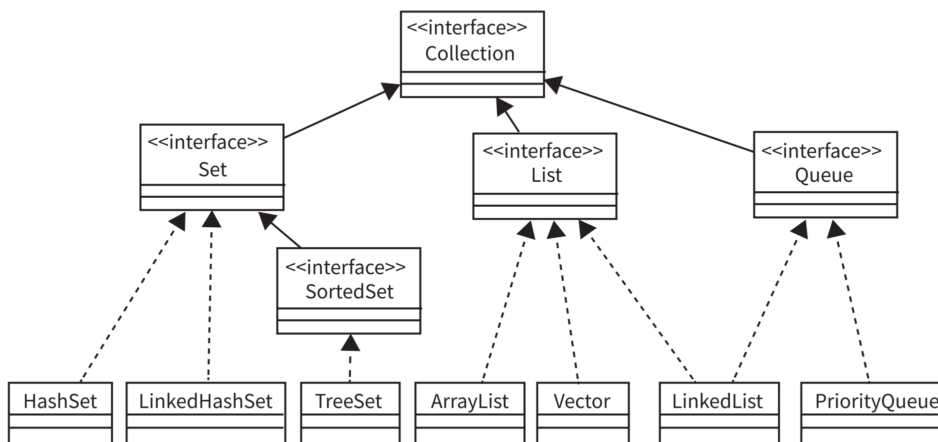
4.1 DEFINIÇÕES: *COLLECTIONS FRAMEWORK*

Por definição, dizemos que coleções *(Collections Framework)* são um conjunto bem definido de interfaces e classes para representar e tratar grupos de dados como uma única unidade. Uma coleção contém os seguintes elementos (COLLECTIONS, 2019):

- Interfaces: tipos de dados abstratos que representam as coleções. As interfaces permitem que as coleções sejam manipuladas independentemente dos detalhes de sua representação, desde que o acesso aos objetos se restrinja apenas ao uso de métodos definidos nas interfaces;
- Implementações: são as implementações concretas das interfaces, em essência, elas são estruturas de dados reutilizáveis;
- Algoritmos: são os métodos que realizam pesquisa e classificação partindo dos objetos que implementam as interfaces. Tais métodos são considerados polimórficos, ou seja, o mesmo método pode ser usado em muitas implementações diferentes da interface.

A Figura 70 mostra a árvore da hierarquia de interfaces e classes da Java *Collections Framework* que são derivadas da interface *Collection*. O diagrama usa a notação da UML, onde as linhas cheias representam heranças e as linhas pontilhadas representam implementações.

Figura 70 – Árvore hierárquica de *Collections Framework*



Fonte: (COLLECTIONS, online, 2019)

A seguir vamos descrever as principais interfaces apresentadas na Figura 70:

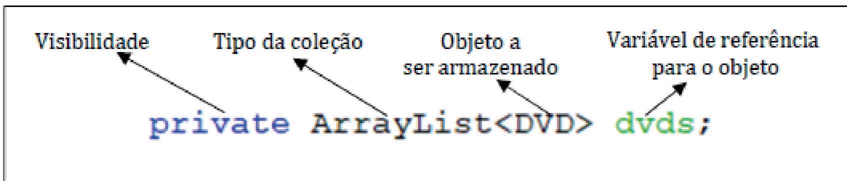
- *Collection* – É a raiz, está no topo da hierarquia. Não existe implementação direta dessa interface, mas ela define as operações básicas para as coleções;
- *Set* – Define uma coleção que não permite elementos duplicados;
- *List* – Coleção que mantém uma certa ordem, que pode conter elementos duplicados. Nesta coleção, o usuário tem controle sobre a posição onde cada elemento é inserido e pode recuperá-los por meio de seus índices;
- *Queue* – Coleção usada para armazenar vários elementos antes do processamento. Com a interface podemos criar, por exemplo, estruturas de dados tais como filas e pilhas;
- *Map* – Mapeia chaves para os valores. Nesta coleção, as chaves não podem ser duplicadas e cada chave pode mapear somente um valor.

Nesta unidade, o nosso objetivo é trabalhar com duas coleções, uma para *CD* e outra para *DVD*. Neste sentido, iremos utilizar uma *List*, mais especificamente *ArrayList*, que será detalhada a seguir.

4.2
ARRAYLIST

A classe *ArrayList* é um vetor que pode ser redimensionável, de acordo com a necessidade. Ela é importada do pacote *java.util.ArrayList.* A diferença básica entre um vetor e o *ArrayList* consiste no redimensionamento. No vetor, o tamanho não pode ser modificado (é uma estrutura de dados estática), caso seja necessário aumentar ou diminuir o total de elementos teremos que criar um novo vetor. Enquanto que no *ArrayList* os elementos podem ser adicionados ou removidos sempre que for necessário. A linguagem Java possui várias classes de coleções predefinidas. Como qualquer outra classe, a coleção deve instanciada antes de ser usada. Inicialmente temos de declarar uma variável de referência para o tipo de coleção, conforme apresentado na Figura 71.

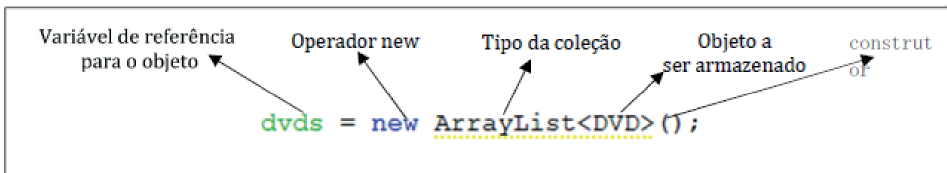
Figura 71 – Variável de referência para *ArrayList*



Fonte: Autores.

Posteriormente, após declarar a variável de referência, temos que usar o operador *new* para invocar o construtor específico para o tipo de coleção que desejamos criar. Ao final deste processo, teremos o objeto instanciado. A Figura 72 apresenta o código para esta ação.

Figura 72 – Instanciando o objeto



Fonte: Autores.

Cabe ressaltar que, para trabalhar com coleções, não precisamos saber os detalhes privados de como as referências dos objetos são armazenadas internamente. Para usar uma coleção corretamente nós só precisamos saber as características públicas da coleção, em particular, a assinatura dos métodos públicos, pois facilitam a escolha do tipo de coleção apropriado para uma a situação em particular. A classe *ArrayList* possui muitos métodos, a seguir vamos detalhar os mais importantes.

Método add()

Este método adiciona elementos ao *ArrayList*. A Figura 73 representa graficamente este método.

Figura 73 – Método *add()*

a)
```
cds.add(cd);
```

b)
```
ArrayList<String> nomes = new ArrayList<String>();
nomes.add("José Silva");
nomes.add("Csrlos Vicente");
nomes.add("Maria Celia");
```

Fonte: Autores.

A Figura 73, na letra a, adiciona um objeto *cd*, no *ArrayList cds*. Já na letra b temos um *ArrayList* de *String*, onde foram adicionados três nomes de pessoas.

Método *get()*

O método *get* é usado para acessar um elemento no *ArrayList*, levando em consideração o índice em que o elemento se encontra. A Figura 74 representa graficamente este método, na letra a, para o *ArrayList* de DVDs e, na letra b, para um objeto de *String*. Cabe ressaltar que os índices para o *ArrayList* começam com 0.

Figura 74 – Método *get()*

a)
```
DVD primeiroDVD = dvds.get(0);
```

b)
```
String primeiroNome = nomes.get(0);
```

Fonte: Autores.

Método *set()*

Utilizamos o método *set()* para alterar um elemento no *ArrayList*, em um índice específico. No exemplo da Figura 75, letra a, estamos trabalhando com um *ArrayList* de objetos definido por nós; portanto, temos que criar um novo objeto, chamado *alterarDVD*, conforme apresentado na linha 50. Tal objeto deve conter todas as informações que desejamos alterar a respeito do DVD. Ainda precisamos fazer mais duas alterações, nas linhas 51 e 52, pois queremos alterar todas as informações do objeto. Uma vez criado o objeto com as alterações, temos que atualizar o objeto *dvds*, no índice 0, de acordo com a linha 54.
Já na linha 57, temos a atualização de um *ArrayList* de *String*. Veja que a atualização ocorre com a passagem direta das informações para o método *set*.

Figura 75 – Método *set()*

a)

```
50 DVD alterarDVD = new DVD("Titulo alterado", "Diretor alterado", 30);
51 alterarDVD.setDescricao("Descrição alterada");
52 alterarDVD.setTenho(true);
53
54 dvds.set(0,alterarDVD);
```

b)

```
57 nomes.set(0, "Novo Nome");
```

Fonte: Autores.

Método *remove()*

Este método é usado para remover um elemento do *ArrayList* levando em consideração o índice em que se encontra o elemento. O método de remoção está representado, graficamente, na Figura 76.

Figura 76 – Método *remove()*

a)

```
dvds.remove(0);
```

b)

```
nomes.remove(0);
```

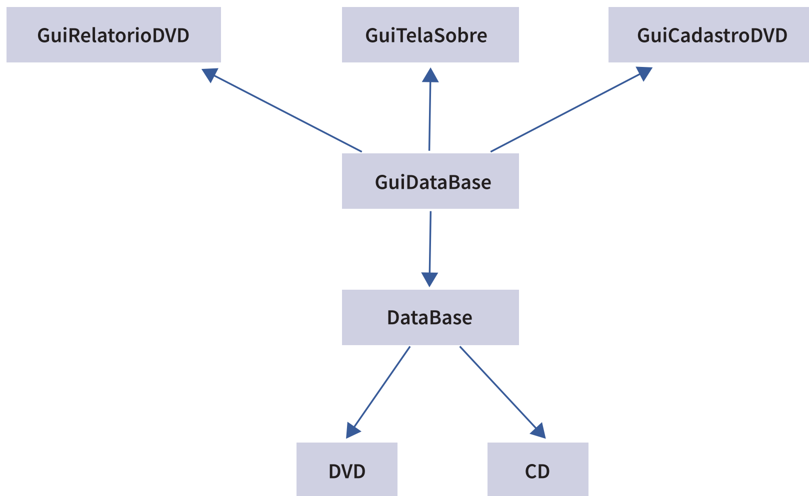
Fonte: Autores.

Agora que entendemos o funcionamento do *ArrayList* e como é possível construir uma coleção de objetos, podemos construir o nosso protótipo de gerenciar uma coleção de DVDs e CDs, que será detalhada a seguir.

4.3
COLEÇÃO DE CDS E DVDS

O projeto para criar um protótipo que gerencie CDs e DVDs, conforme apresentado na Figura 77, utiliza 4 classes para construção das interfaces gráficas (*GuiRelatorioDVD*, *GuiTelaSobe*, *GuiCadastroDVD* e *GuiDataBase*) e outras três classes responsáveis por controlar o armazenamento dos DVDs e CDs (*DataBAse*, *DVD* e *CD*). A seguir, vamos detalhar cada uma dessas classes.

Figura 77 – Classes utilizadas no protótipo



Fonte: Autores.

Classe *CD*

A classe *CD* é usada como modelo para criação de objetos do tipo *CD*, armazenados no *ArrayList*. A Figura 78 apresenta, graficamente, o construtor da classe. Na sequência, vamos explanar o código apresentado.

Figura 78 – Construtor da classe *CD*

```
1   package br.ufsm.midias;
2
3   /** A classe DC representa um objeto CD de musica ...7 linhas */
10  public class CD{
11      private String titulo;
12      private String artista;
13      private int numeroDeFaixas;
14      private int tempoDuracao;
15      private boolean tenho;
16      private String descricao;
17
18      /**
19       * Construtor para CD.
20       */
21      public CD(String titulo, String artista, int numeroDeFaixas, int tempoDuracao){
22          this.titulo = titulo;
23          this.artista = artista;
24          this.numeroDeFaixas = numeroDeFaixas;
25          this.tempoDuracao = tempoDuracao;
26          tenho = false;
27          descricao = " ";
28      }
```

Fonte: Autores.

Ao aplicarmos o conceito de abstração no objeto *CD*, selecionamos as informações necessárias para representar um *CD* do mundo real, no mundo computacional. Tais informações são armazenadas nos atributos da classe, nas linhas 11 a 16. É importante observar que cada atributo possui visibilidade *private*; logo, só teremos acessos a estes atributos dentro da classe. Caso precisemos acessá-los em outras classes, vamos ter de construir métodos para este acesso (*get* ou *set*).

Já nas linhas 21 a 28, declaramos o construtor da classe. Veja que ele possui o mesmo nome da classe e não possui nenhum tipo de retorno. O construtor recebe os argumentos nos respectivos parâmetros com os mesmos nomes dos atributos. Logo, para diferenciarmos os atributos dos parâmetros, utilizamos a palavra reservada *this*. Sendo assim, as variáveis nas linhas 22 a 25, que são referenciadas por *this* pertencem à classe. Por isso, na linha 22, estamos atribuindo o parâmetro título ao atributo da classe *this.titulo*. Logo a informação ficará armazenada no objeto, quando este for instanciado. O mesmo ocorre com as linhas 23, 24 e 25. Já na linha 26 é atribuído um valor *booleano false*, na linha 27 atribuído um valor vazio ao atributo *descricao*.

Agora passaremos aos métodos *set*. Estes métodos são utilizados para alterar informações dos atributos, conforme representado na Figura 79.

Figura 79 – Métodos set da classe *CD*

```
29
30      /**
31       * Insere a descrição para o CD.
32       */
33      public void setDescricao(String descricao){
34          this.descricao = descricao;
35      }
36
37      /**
38       * Configura o flag que indica se possuímos o CD.
39       * true se tiver o CD e, false caso contrário.
40       */
41      public void setTenho(boolean tenho){
42          this.tenho = tenho;
43      }
```

Fonte: Autores.

O primeiro método set da classe *CD* denominado *setDescricao*, insere (ou altera) uma descrição do objeto *CD*, conforme descrito na linha 34. Nessa linha, estamos atribuindo o parâmetro *descricao* ao atributo da classe *this. Descrição*. O outro método denominado *setTenho* possui a função de alterar o atributo que indica se o *CD* atual pertence à coleção cadastrada. Caso pertença, o atributo *this.tenho* recebe *true*, se não pertence à coleção, recebe *false*.

Prosseguiremos na explanação do código, com os métodos *gets*, conforme descrito na Figura 80.

Figura 80 – Métodos *get* da classe *CD*

```
44
45      /**
46       * Retorna título para esse CD
47       */
48      public String getTitulo(){
49          return titulo;
50      }
51
52      /**
53       * O comentário para esse CD.
54       */
55      public String getDescricao(){
56          return descricao;
57      }
58
59      /**
60       *  Retorna true se possuímos uma cópia desse CD.
61       */
62      public boolean getTenho(){
63          return tenho;
64      }
65
66  }
```

Fonte: Autores.

Para a classe *CD*, temos três métodos *get* declarados. O método *getTitulo*, na linha 48, retorna o título do *CD* armazenado. Na linha 55, o método *getDescricao* retorna o comentário feito a respeito do *CD*. Por último, na linha 62, o método *getTenho* retorna *true* se existe uma cópia física deste *CD* na coleção, e *false* caso não tenhamos uma cópia.

Classe *DVD*

A classe *DVD* é usada como modelo para criar uma coleção de objetos do tipo *DVD*, armazenados no *ArrayList*. A Figura 81 apresenta graficamente o construtor da classe. Na sequência vamos explanar o código.

Figura 81 – Construtor da classe *DVD*

```
1   package br.ufsm.midias;
2
3   /** A classe DVD representa um objeto DVD ...9 linhas */
12  public class DVD {
13      private String titulo;
14      private String diretor;
15      private int tempoDuracao; // tempo de reprodução do filme
16      private boolean tenho;
17      private String descricao;
18
19      /**
20       * Construtor para objetos da classe DVD
21       */
22      public DVD(String titulo, String diretor, int tempoDuracao){
23          this.titulo = titulo;
24          this.diretor = diretor;
25          this.tempoDuracao = tempoDuracao;
26          this.tenho = false;
27          this.descricao = "";
28      }
29
```

Fonte: Autores.

Ao aplicarmos o conceito de abstração no *DVD*, foram selecionadas as informações para representar um *DVD* do mundo real, no mundo computacional. Estas informações foram representadas nos atributos da classe. Nas linhas 13 a 17, assim como na classe *CD*, os atributos possuem visibilidade *private*, podendo ser somente acessados da classe por meio dos métodos de acesso (*get* ou *set*).

O construtor da classe, representado graficamente nas linhas 22 a 28, tem o propósito de instanciar os objetos com os valores previamente definidos. Veja que o construtor possui o mesmo nome da classe e obrigatoriamente não pode ter nenhum tipo de retorno. O construtor recebe os argumentos como parâmetros que possuem os mesmos nomes dos atributos. Para diferenciarmos os atributos dos parâmetros, utilizamos a palavra reservada *this*. Por isso, os atributos da classe são referenciados por *this*, conforme representado nas linhas 23 a 25. Já as linhas 26 e 27 recebem os valores diretamente, sem o uso parâmetros. A seguir, vamos descrever os métodos sets, conforme descrito na Figura 82.

Figura 82 – Métodos set da classe *DVD*

```
30    /**
31     * Insere um comentário para esse DVD.
32     */
33    public void setDescricao(String descricao){
34        this.descricao = descricao;
35    }
36
37    /**
38     * Define o valor do flag que indica se possuímos este DVD.
39     */
40    public void setTenho(boolean tenho){
41        this.tenho = tenho;
42    }
```

Fonte: Autores.

Na linha 33, é apresentado o método *setDescricao*, que tem por definição alterar a descrição do *DVD*. Este método recebe o parâmetro *descricao*, que possui o mesmo nome do atributo. Para diferenciarmos os atributos dos parâmetros, utilizamos a palavra reservada *this* durante a atribuição, conforme representado na linha 34. Já o método *setTenho* define o valor do atributo *tenho*, indicando se o *DVD* atual, que está sendo cadastrado, pertence ou não à coleção. Caso ele pertença, deve ser definido como *true*, caso contrário *false*. A seguir passaremos aos métodos *get*, conforme apresenta a Figura 83.

Figura 83 – Métodos *get* da classe *DVD*

```
43
44    /** Retorna comentário para esse DVD ...3 linhas */
47    public String getDescricao(){
48        return descricao;
49    }
50    /** Retorna título para esse DVD ...3 linhas */
53    public String getTitulo() {
54        return titulo;
55    }
56    /** Retorna diretor para esse DVD ...3 linhas */
59    public String getDiretor() {
60        return diretor;
61    }
62    /** Retorna o tempo de duração para esse DVD ...3 linhas */
65    public int getTempoDuracao() {
66        return tempoDuracao;
67    }
68    /** Retorna true se possuímos uma cópia desse DVD ...3 linhas */
71    public boolean getTenho(){
72        return tenho;
73    }
74 }
```

Fonte: Autores.

Os métodos *get* têm a função de retornar os valores dos atributos. Para a classe *DVD* foi criado um método get para cada atributo, totalizando 5 métodos de retorno. Na linha 47, o método retorna a descrição cadastrada para o *DVD*, na linha 53 retorna o título do *DVD*, na linha 59 retorna o nome do diretor, na linha 65 retorna o tempo de duração e, na linha 71, retorna um valor *booleano*, confirmando se tenho (*true*) ou não tenho (*false*) este *DVD* na coleção.

Classe *Database*

A classe *Database* tem, por função, gerenciar a coleção de objetos criados a partir dos modelos definidos nas classes *DVD* e *CD*. Tais coleções de objetos devem ser armazenadas em um *ArrayList*. Inicialmente, conforme apresentado, graficamente, na Figura 84. Nas linhas 18 e 19, foram declaradas as variáveis de instância *cds* e *dvds*. Já nas linhas 25 a 28, foi criado o construtor da classe, cuja finalidade é definir o estado inicial do objeto, por meio da instanciação dos atributos. Na linha 26, o atributo *cds* é instanciado com o objeto *ArrayList* de *CD* e, na linha 27, o atributo *dvds* é instanciado com o objeto *ArrayList* de *DVD*.

Figura 84 – Construtor da classe *Database*

```
1   package br.ufsm.midias;
2
3   import java.util.ArrayList;
4
5   /** A casse Database fornece recursos para armazenar objetos do tipo CD e DVD ..
16  public class Database
17  {
18      private ArrayList<CD> cds;
19      private ArrayList<DVD> dvds;
20
21      /** Construtor da classe Database ...4 linhas */
25      public Database(){
         cds=new ArrayList<CD>();
         dvds=new ArrayList<DVD>();
28      }
29
```

Fonte: Autores.

A Figura 85 apresenta os métodos *sets* da classe *Database,* a começar, na linha 33, que foi criado o método *setCD*, que recebe como argumentos o parâmetro *cd*. Neste caso, em específico, o nome do parâmetro difere do nome do atributo. Logo, não há necessidade de usar a palavra reservada *this* para referenciar o atributo, que foi denominado de *cds*. A linha 34 adiciona um objeto *cd* à coleção de objetos *cds*, que é um *ArrayList*. Por ser do tipo *ArrayList* o objeto cds apresenta vários métodos, dentre eles o *add()*, que serve para adicionar informações à coleção.

Na linha 40, foi definido o método *setDVD*, que recebe o argumento *dvd* e adiciona esta informação a *dvds* por meio do método *add()*, conforme apresentado na linha 41.

Figura 85 – Métodos *set* da classe *Database*

```
30      /**
31       * Adiciona um CD ao banco de dados.
32       */
33      public void setCD(CD cd){
34          cds.add(cd);
35      }
36
37      /**
38       * Adiciona um DVD ao banco de dados.
39       */
40      public void setDVD(DVD dvd){
41          dvds.add(dvd);
42      }
43
```

Fonte: Autores.

Na sequência, passaremos a estudar os métodos *get* da classe *Database*. Esses métodos estão apresentados na Figura 86. O primeiro método, denominado de get-*Cds*, retorna todo o *ArrayList* de *CD*, conforme apresentado na linha 48. O retorno ocorre ao fazer uso da palavra reservada *return*. Já o método *getDvds* retorna, conforme apresentado na linha 55, a coleção de *DVDs* armazenados no *ArrayList*.

Figura 86 – Métodos *get* da classe *Database*

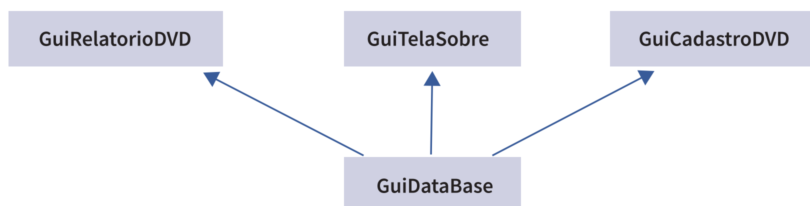
```
43
44  /**
45       * Retorna todo o ArrayList de CD
46       */
47      public ArrayList<CD> getCds() {
48          return cds;
49      }
50
51      /**
52       * Retorna todo o ArrayList de DVD
53       */
54      public ArrayList<DVD> getDvds() {
55          return dvds;
56      }
57  }
```

Fonte: Autores.

Classe *GuiCadastroDVD*

Passaremos agora à construção das interfaces. Ao todo, vamos construir 4 formulários gráficos, de acordo com a Figura 87. O *GuiDataBase* é um formulário *JFrame*, que possui um Menu com a finalidade de chamar os outros forms. O *GuiRelatorioDVD* é um formulário *JDialog* que tem um *JTable*, cuja finalidade é a de exibir conteúdos em formato de tabelas. O *GuiTelaSobre* é um formulário *JDialog* que tem a função de apresentar informações sobre os autores do protótipo. Por fim, o *GuiCadastroDVD* é um formulário *JDialog* que possui os componentes *JTextFild*, *JComboBox* e *JTextArea*, todos com a finalidade de armazenar informações sobre os *DVDs*.

Figura 87 – Formulários gráficos



Fonte: Autores.

Conforme descrito anteriormente, vamos usar os formulários *jFrame* e *jDialog*. Qual a diferença entre eles? Basicamente, a diferença é que o *jDialog* é modal; dessa forma, a tela fica congelada aguardando a interação do usuário. Já o *jFrame* possibilita ao usuário transitar entre as várias telas, abrindo mais de uma tela simultaneamente. A seguir, serão apresentadas as principais características desses formulários:

- *jFrame*: Este formulário possui uma barra de título e está preparado para receber menus e outros componentes;

- *jDialog*: Geralmente é usado para definir formulários destinados à entrada de dados, em resposta a uma opção de seleção do menu, criado em um *jFrame*.

Dentre os formulários descritos acima, vamos começar pelo formulário de cadastro de *DVD*, lembrando que este formulário é um *jDialog*. A seguir, serão detalhados os passos para a criação de tal formulário, conforme apresentado na Figura 88:

1. clique com o botão direito do mouse sobre o nome do pacote;
2. clique em Novo;
3. clique em *Form jDialog...*

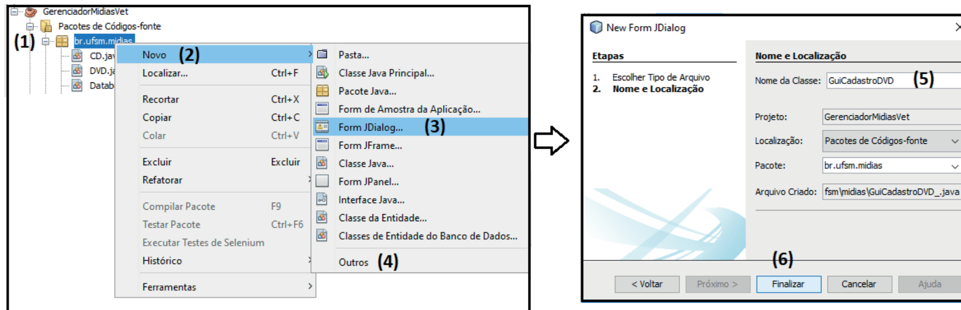
Caso a opção *Form jDialog...* não apareça vá em:

4. Outros.

Caso a opção *Form jDialog...* não apareça vá em:

No passo 5, no item "nome da classe" digite *GuiCadastroDVD*, e no passo 6 clique em finalizar. Com isso, acabamos de criar o nosso primeiro *jDialog*.

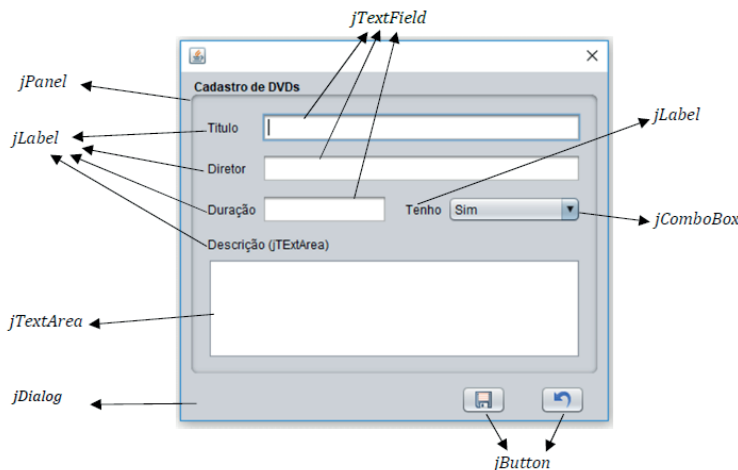
Figura 88 – Criação do *jDialog*



Fonte: Autores.

Depois da criação do formulário *GuiCadastroDVD*, vamos inserir os objetos gráficos, conforme apresenta a Figura 89.

Figura 89 – Formulário de cadastro de *DVD*



Fonte: Autores.

Veja que nos botões *jButton*, na Figura 89, foram inseridas imagens, que serão disponibilizadas no *Moodle*/UAB-UFSM, pelo professor da disciplina. Lembre-se, para facilitar a identificação dos objetos inseridos temos que renomeá-los. Os objetos que precisam ter os nomes alterados estão delimitados por retângulos. Faça as alterações exatamente como está apresentado na Figura 90.

Figura 90 – Renomeando os objetos do formulário *GuiCadastroDVD*

```
240        // Variables declaration - do not modify
241        private javax.swing.JButton jB_CadastrarDVD;
242        private javax.swing.JButton jB_Voltar;
243        private javax.swing.JComboBox<String> jCB_Tenho;
244        private javax.swing.JLabel jLabel1;
245        private javax.swing.JLabel jLabel2;
246        private javax.swing.JLabel jLabel3;
247        private javax.swing.JLabel jLabel4;
248        private javax.swing.JLabel jLabel5;
249        private javax.swing.JPanel jPanel1;
250        private javax.swing.JScrollPane jScrollPane1;
251        private javax.swing.JTextArea jTA_Descricao;
252        private javax.swing.JTextField jTF_Diretor;
253        private javax.swing.JTextField jTF_Duracao;
254        private javax.swing.JTextField jTF_TituloDVD;
255        // End of variables declaration
```

Fonte: Autores.

Passaremos à análise do código fonte, conforme apresentado na Figura 91. Inicialmente, na linha 12, foi declarado um atributo do tipo *Database*, com a finalidade de armazenar as informações do formulário e compor o banco de dados. Cabe ressaltar que as informações passadas para o atributo *database* são por referência, logo todas as alterações feitas na classe *GuiCadastroDVD* serão automaticamente atualizadas na classe chamadora.

Agora vamos analisar os construtores. Nesta classe, temos 2, um na linha 16 e o outro na linha 24, o que caracteriza uma sobrecarga de construtores. Por que precisamos de 2 construtores? O construtor na linha 16 foi criado automaticamente no ato da criação do jDialog. Já o construtor da linha 24 foi criado com a finalidade de passagem do parâmetro *dataBase*. Na linha 26, o parâmetro *database* é atribuído ao atributo. Repare que o nome do parâmetro e o nome do atributo são idênticos, por isso a necessidade de usar a palavra reservada *this*.

SAIBA MAIS: pesquise mais sobre sobrecarga de construtores.

Figura 91 – Análise do código – Construtores de *GuiCadastroDVD*

```
 6  package br.ufsm.midias;
 7
 8  /**
 9   * @author fabio
10   */
11  public class GuiCadastroDVD extends javax.swing.JDialog {
12       private Database dataBase;
13      /**
14       * Construtor criado automaticamente
15       */
16      public GuiCadastroDVD(java.awt.Frame parent, boolean modal) {
17          super(parent, modal);
18          initComponents();
19      }
20
21   /**
22   * Sobrecarga do construtor para passagem do parâmetro dataBase
23   */
24      public GuiCadastroDVD(java.awt.Frame parent, boolean modal,Database dataBase) {
25          super(parent, modal);
26          this.dataBase = dataBase;
27          initComponents();
28      }
```

Fonte: Autores.

Continuando a análise do código (Figura 92), vamos descrever o evento *"actionPerformed"* criado para o botão *jB_CadastrarDVD*. Lembre-se, este evento é responsável por executar uma ação caso ocorra um evento, por exemplo, o clique do mouse sobre o botão. Após criar o evento *jB_CadastrarDVDActionPerformed* vamos entender o código que deve ser inserido neste evento.

SAIBA MAIS: caso você não lembra como criar um Evento *ActionPerformed*, pesquise na Figura 47 do seu livro.

O nosso objetivo é criar um banco de dados de *DVDs* e *CDs*; para isso, na linha 173, está sendo criado e instanciado um objeto do tipo *DVD*, chamado *novoDVD*. Neste objeto, estão sendo adicionadas todas as informações digitadas pelo usuário, tais como: título, diretor e a duração. Na linha 177, de acordo com o índice do *jCB_Tenho* selecionado pelo usuário atribuímos *true* ou *false*. Por exemplo, se o índice for 0 atribuímos true, se for 1 atribuímos false. Na linha 180, é atribuída a descrição digitada pelo usuário no objeto *jTA_Descricao*, ao objeto *novoDVD*. Na linha 182, inserimos o objeto *novoDVD* ao *dataBase*. Por fim, na linha 183 é realizada a chama do método *limpaTela*, que será descrito a seguir.

SAIBA MAIS: pesquise mais sobre o método *Integer.parseInt*. Descubra qual a função no nosso código e outras possibilidades de uso.

Figura 92 – Análise do código – Evento do botão *jB_CadastrarDVD*

```
171    private void jB_CadastrarDVDActionPerformed(java.awt.event.ActionEvent evt) {
172        //Cria e instancia o objeto novoDVD
173        DVD novoDVD = new DVD(jTF_TituloDVD.getText(),
174                jTF_Diretor.getText(),
175                Integer.parseInt(jTF_Duracao.getText()));//conv. String para int
176        //De acordo com indice do JCB_tenho atribui true ou false
177        if(jCB_Tenho.getSelectedIndex()==0) novoDVD.setTenho(true);
178        else novoDVD.setTenho(false);
179        //Insere a descrição
180        novoDVD.setDescricao(jTA_Descricao.getText());
181        //Armazena o novoDVD em dataBAse
182        dataBase.setDVD(novoDVD);
183        limpaTela();
184    }
```

Fonte: Autores.

O método *limpaTela* tem a função de limpar todas as informações digitadas pelo usuário. Veja que ele é chamado depois que as informações são atribuídas ao banco de dados. A Figura 93 apresenta as linhas de código utilizadas para desempenhar tal função. Veja que, nas linhas 187 a 190, todos os objetos são do tipo *jTextFild (jTF_)*, por isso podemos limpar os conteúdos atribuindo o valor *null* aos objetos. Na linha 191, temos um objeto do tipo *jComboBox (jCB_)*. Como a nossa aplicação trabalha com o índice, para limpar o valor escolhido pelo usuário vamos atribuir 0. Pra finalizar, na linha 193 escolhemos em qual objeto será reposicionado o cursor, por meio do método *requestFocus*. Neste caso, o objeto será o *jTF_TituloDVD.requestFocus()*.

Figura 93 – Análise do código – Método *limpaTela*

```
186    private void limpaTela(){
187        jTF_TituloDVD.setText(null);//Limpa o conteúdo do objeto
188        jTF_Diretor.setText(null);//Limpa o conteúdo do objeto
189        jTF_Duracao.setText(null);//Limpa o conteúdo do objeto
190        jTA_Descricao.setText(null);//Limpa o conteúdo do objeto
191        jCB_Tenho.setSelectedIndex(0);//Atribui índice 0 ao objeto
192        jTF_TituloDVD.requestFocus(); // Seta o foco no objeto
193    }
```

Fonte: Autores.

Para finalizar, vamos abordar o método do botão *jB_Voltar*. Neste método, temos uma única linha de código, conforme apresentado na Figura 94, que é a linha 197. A função principal deste código é fechar a janela atual, devolvendo o foco para a janela principal.

Figura 94 – Análise do código – Método do botão *jB_Voltar*

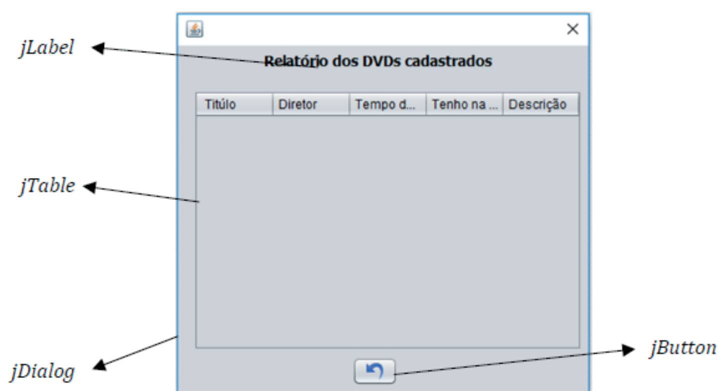
```
195    private void jB_VoltarActionPerformed(java.awt.event.ActionEvent evt) {
196        //Fecha a janela atual
197        this.dispose();
198    }
```

Fonte: Autores.

Classe *GuiRelatorioDVD*

O formulário de relatório de *DVD* é um *JDialog*. Neste formulário, serão apresentados todos os DVDs cadastrados no formulário *GuiCadastroDVD*. Primeiramente, vamos criar um novo *jDialog*. Caso ainda reste alguma dúvida quanto à criação do *jDialog* consulte a Figura 88 – Criação do *jDialog*. Após a criação, vamos inserir todos os objetos necessários, apresentados na Figura 95. São eles os objetos *jLabel*, *jTable* e *jButton*.

Figura 95 – Formulário de relatório de *DVD*



Fonte: Autores.

Ao finalizarmos a parte gráfica, passaremos ao código. Vamos começar pela renomeação dos objetos inseridos, para facilitar a identificação dos mesmos em toda a classe. Os objetos que precisam ter os nomes alterados estão delimitados por retângulos. Faça as alterações exatamente como está apresentado na Figura 96.

Figura 96 – Renomeando os objetos do formulário *GuiRelatorioDVD*

```
180    // Variables declaration - do not modify
181    private javax.swing.JButton jB_Voltar;
182    private javax.swing.JLabel jLabel1;
183    private javax.swing.JScrollPane jScrollPane1;
184    private javax.swing.JTable jT_VisualizaProduto;
185    // End of variables declaration
```

Fonte: Autores.

Após a renomeação dos objetos necessários, passaremos a analisar as linhas de código iniciais da classe, conforme apresentado na Figura 97. Relembrando, a linha 6 indica o pacote ao qual a classe pertence, inserido automaticamente no ato da criação da classe. Na linha 8, temos a importação da classe *ArrayList* e, na linha 9, a *DefaultTableModel*. Ambas serão usadas em métodos posteriores.

Reparem que nesta classe também temos a sobrecarga de construtores: o primeiro construtor nas linhas 19 a 22, criado automaticamente, e o segundo construtor, nas linhas 24 a 28, foi criado para passagem do parâmetro *dataBase*. Conforme explicado anteriormente, o objetivo desse parâmetro é armazenar as informações cadastradas dos *DVDs*.

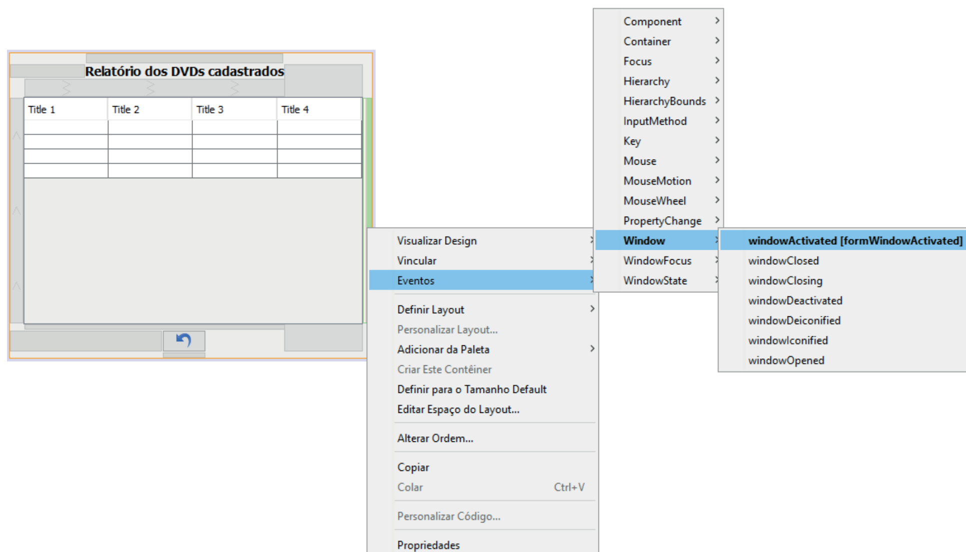
Figura 97 – Análise do código – Construtores de GuiRelatorioDVD

```
 6  package br.ufsm.midias;
 7
 8  import java.util.ArrayList;
 9  import javax.swing.table.DefaultTableModel;
10
11  /**
12   * @author fabio
13   */
14  public class GuiRelatorioDVD extends javax.swing.JDialog {
15      private Database dataBase;
16      /**
17       * Creates new form GuiRelatorioDVD
18       */
19      public GuiRelatorioDVD(java.awt.Frame parent, boolean modal) {
20          super(parent, modal);
21          initComponents();
22      }
23
24      public GuiRelatorioDVD(java.awt.Frame parent, boolean modal,Database dataBase) {
25          super(parent, modal);
26          this.dataBase = dataBase;
27          initComponents();
28      }
```

Fonte: Autores.

O próximo método a ser analisado foi criado com a finalidade de apresentar as informações cadastradas dos *DVDs* no objeto *jTable*. Para isso, foi inserido o evento *windowActivated*, no formulário *GuiRelatorioDVD*. Esse evento ocorre toda vez que o formulário for ativado. Com isso, iremos atualizar a *jTable* com as informações cadastradas dos *DVDs*, toda vez que o formulário for chamado. Para inserir o evento, vamos seguir os passos da Figura 98.

Figura 98 – Inserção do evento *windowActivated*



Fonte: Autores.

A seguir, vamos entender o código que foi inserido neste evento. A Figura 99 apresenta todo o código desenvolvido. Começando, na linha 111, em que foi declarada uma variável chamada *dados*, do tipo *ArrayList<DVD>*, para receber todas as in-

formações cadastradas dos objetos *DVDs*.

Na linha 113, foi criada uma matriz, do tipo *String*, chamada *nomesColunas[]*, contendo os nomes dos títulos das colunas da tabela. Já na linha 116 criamos uma matriz, do tipo *String*, com o total de linhas igual à quantidade de objetos *DVD* cadastrados e o total das colunas igual à quantidade de títulos que possui a tabela. A seguir, nas linhas 118 a 124, criamos uma estrutura de repetição, laço for na linha 118, com a finalidade de inserir as informações cadastradas dos *DVDs*, na matriz *valoresLinhas[][]*. Nas linhas 119 à 123 ocorre a atribuição dos valores cadastrados dos objetos *DVD* a matriz *valoresLinhas[][]*.

Após a criação da matriz contendo os títulos (*nomesColunas[]*) e os valores das linhas (*valoresLinhas[][]*) da tabela, vamos criar um modelo da tabela com essas informações, conforme apresentado na linha 126. Por fim, vamos atualizar o nosso objeto gráfico, inserido no formulário, com o modelo da tabela criado. A atualização da tabela *jT_VisualizaProduto* está representada na linha 128.

Figura 99 – Análise do código – Evento *WindowActivated*

```
109    private void formWindowActivated(java.awt.event.WindowEvent evt) {
110        // Cria uma variável e atribui todos os objetos DVDs
111        ArrayList<DVD> dados =  dataBase.getDvds();
112        //Atribui os titulos das colunas, da tabela
113        String[] nomesColunas = {" Título","Diretor","Tempo de duração","Tenho na lista","Descrição"};
114        //Define uma matriz de String, onde a quantidade linhas é o total de
115        //objetos do tipo DVD cadastrados e as colunas o total de títulos
116        String[][] valoresLinhas = new String[dados.size()][5];
117        //Armazena as informações dos objetos DVDs, na matriz valoresLinhas[][]
118        for(int i=0;i<dados.size();i++){
119            valoresLinhas[i][0]= dados.get(i).getTitulo();
120            valoresLinhas[i][1]= dados.get(i).getTitulo();
121            valoresLinhas[i][2]= String.valueOf(dados.get(i).getTempoDuracao());
122            valoresLinhas[i][3]= String.valueOf(dados.get(i).getTenho());
123            valoresLinhas[i][4]= dados.get(i).getDescricao();
124        }
125        //Cria um modelo da Tabela, com informações das linhas e colunas
126        DefaultTableModel tableModel = new DefaultTableModel(valoresLinhas,nomesColunas);
127        //Atualiza o objeto jT_VisualizaProduto com o modelo criado
128        this.jT_VisualizaProduto.setModel(tableModel);
129    }
```

Fonte: Autores.

Para finalizar os métodos abordados nessa classe, vamos analisar o método do botão *jB_Voltar*. Conforme apresentado na Figura 100, a nossa única linha de código na linha 133, tem a função de fechar a janela atual, devolvendo o foco para a janela chamadora.

Figura 100 – Análise do código – Evento *jB_VoltarActionPerformedwActivated*

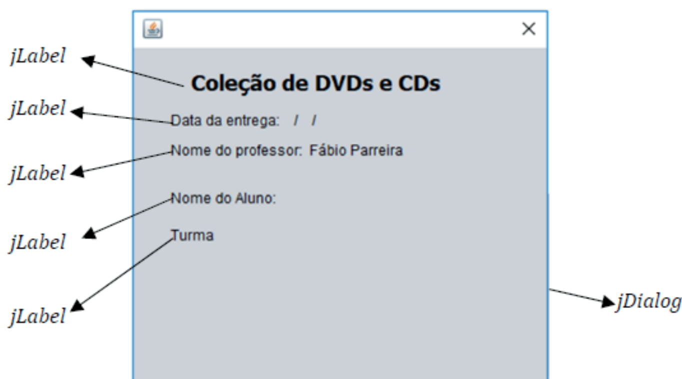
```
131    private void jB_VoltarActionPerformed(java.awt.event.ActionEvent evt) {
132        //Fecha a janela atual
133        this.dispose();
134    }
```

Fonte: Autores.

Classe GuiTelaSobre

O formulário *GuiTelaSobre* tem a função de exibir informações sobre o protótipo criado, bem como as informações dos autores. Para criar esta janela, insira um formulário do tipo *jDialog* e, na sequência, os componentes, conforme representado na Figura 101. Veja que todos eles são do tipo *jLabel*.

Figura 101 – Formulário da tela sobre



Fonte: Autores.

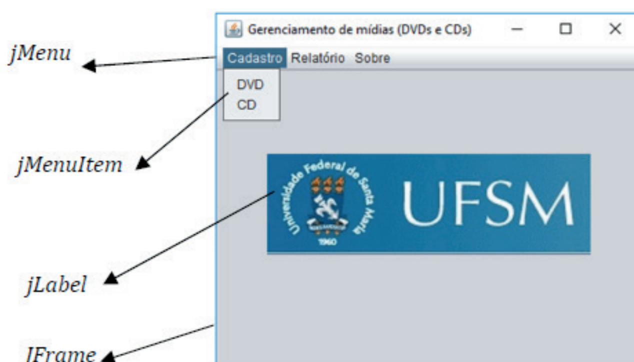
O formulário *GuiTelaSobre* não possui o botão para voltar. Pesquise e insira o objeto gráfico e o código necessário para implementar esta ação.

SAIBA MAIS: faça uma pesquise para inserir o botão e o código para devolver o foco para a janela principal.

Classe GuiDatabase

Passaremos a estudar o formulário principal, denominado de *GuiDataBase*, que é um *jFrame*. Por meio deste formulário, iremos acessar os demais formulários: *GuiCadastroDVD*, *GuiRelatorioDVD* e *GuiTelaSobre*. Primeiramente, vamos criar um novo *jFrame* e, na sequência, inserir todos os objetos necessários, apresentados na Figura 102. São eles jMenu, *jMenuItem e jLabel*.

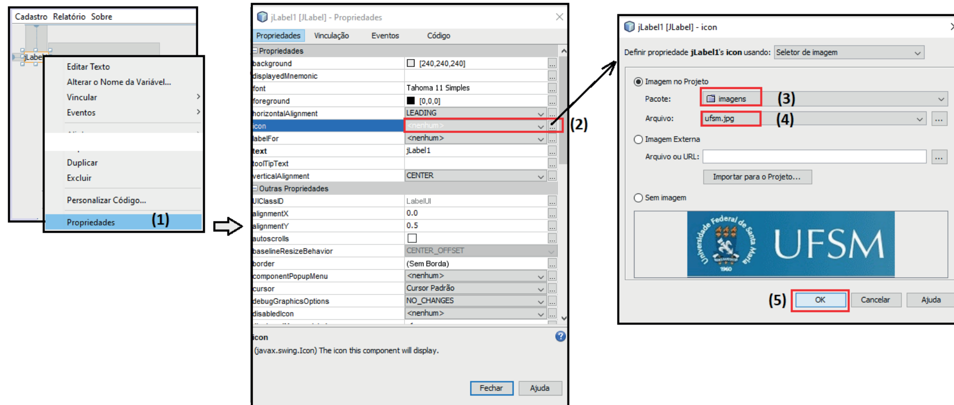
Figura 102 – Formulário *GuiDatabase*



Fonte: Autores.

Anteriormente, trabalhamos com o *jLabel* inserindo textos. Agora vamos inserir uma imagem. Para isso, vamos seguir os passos da Figura 103. Primeiramente, clique o botão direito do mouse no *jLabel*, depois clique em Propriedades (1), *icon* (2), escolha o pacote (3), depois a imagem (4) e, para finalizar, clique no botão ok (5).

Figura 103 – Inserção da imagem no *jLabel*



Fonte: Autores.

Na sequência, após a finalização da parte gráfica, passaremos ao código. Inicialmente, vamos renomear os objetos inseridos, para facilitar a identificação dos mesmos em toda a classe, conforme apresentado na Figura 104.

Figura 104 – Renomeando os objetos do formulário *GuiDatabase*

```
196    private javax.swing.JLabel jLabel2;
197    private javax.swing.JMenuBar jMB_MenuPrincipal;
198    private javax.swing.JMenuItem jMI_CadastroCD;
199    private javax.swing.JMenuItem jMI_CadastroDVD;
200    private javax.swing.JMenuItem jMI_RelatorioCD;
201    private javax.swing.JMenuItem jMI_RelatorioDVD;
202    private javax.swing.JMenuItem jMI_SobrePrototipo;
203    private javax.swing.JMenu jM_Cadastro;
204    private javax.swing.JMenu jM_Relatorio;
205    private javax.swing.JMenu jM_Sobre;
```

Fonte: Autores.

A seguir, vamos entender o código de toda a classe *GuiDataBase*, a começar nos atributos, conforme apresentado na Figura 105. Inicialmente, temos na linha 14 a criação de um atributo e a instanciação desse atributo com o objeto *Database*. Cabe relembrar que a classe *Database* possui um *ArrayList* para armazenar as informações tanto de *DVDs*, quanto de *CDs*. A seguir, na linha 17 foi criado um atributo chamado *dialogoCadastro* e instanciado com o objeto do tipo *GuiCadastroDVD*. Esse objeto traz informações de todo o formulário de cadastro de *DVDs*. Além disso, foi passado como parâmetro o objeto *dataBase*, criado anteriormente. Nesta linha, estamos chamando o construtor sobrecarregado que desenvolvemos, pois existe a passagem do parâmetro *dataBase*.

Já na linha 20 criamos o atributo *dialogoRelatorio*, que foi instanciado com o objeto do tipo *GuiRelatorioDVD*. Neste caso, este objeto traz as informações acer-

ca do formulário de relatório de *DVDs*, e novamente foi utilizado o construtor sobrecarregado da classe. Por fim, na linha 23 foi criado o atributo e instanciado o objeto referente a classe *GuiTelaSobre*.

Figura 105 – Análise do código – Atributos

```
6   package br.ufsm.midias;
7
8   /**
9    * @author fabio
10   */
11  public class GuiDataBase extends javax.swing.JFrame {
12
13      //Cria uma variável e instancia o objeto Database
14      private Database dataBase= new Database();
15
16      //Cria uma variável e instancia o objeto GuiCadastroDVD
17      private GuiCadastroDVD dialogoCadastro = new GuiCadastroDVD(new javax.swing.JFrame(),
18              true, dataBase);
19      //Cria uma variável e instancia o objeto GuiRelatorioDVD
20      private GuiRelatorioDVD dialogoRelatorio = new GuiRelatorioDVD(new javax.swing.JFrame(),
21              true, dataBase);
22      //Cria uma variável e instancia o objeto GuiTelaSobre
23      private GuiTelaSobre dialogoSobre = new GuiTelaSobre(this, true);
```

Fonte: Autores.

Passaremos a inserir os eventos do tipo *ActionPerformed* para os *jMenuItem*, conforme apresentado na Figura 106. Até o momento estamos construindo os códigos referentes aos *DVDs*.

Figura 106 – Análise do código – Métodos

```
131     private void jMI_CadastroDVDActionPerformed(java.awt.event.ActionEvent evt) {
132         dialogoCadastro.setVisible(true);
133     }
134
135     private void jMI_RelatorioDVDActionPerformed(java.awt.event.ActionEvent evt) {
136        dialogoRelatorio.setVisible(true);
137     }
138
139     private void jMI_SobrePrototipoActionPerformed(java.awt.event.ActionEvent evt) {
140         dialogoSobre.setVisible(true);
141     }
```

Fonte: Autores.

Ao analisar a linha 131, da figura 106, temos o evento para o menu item *jMI_CadastroDVDActionPerformed*, que irá exibir o formulário *dialogoCadastro* chamando o método *setVisible(true)* na linha 132. Isto irá ocorrer toda vez que o usuário clicar no menu *Cadastro->DVD*. Os demais eventos, na linha 136 e 140, são similares.

ATIVIDADES - UNIDADE 4

Todas as atividades abaixo devem ser postadas no Moodle/UAB-UFSM, conforme direcionamento do professor da disciplina.

1) Implemente todo o exemplo de aplicação apresentado nesta unidade e acrescente o cadastro de CDs e o relatório de CDs, ou seja, acrescente dois formulários e implemente os seus respectivos códigos.

2) Implemente um protótipo em Java que simula algumas operações de um Caixa Eletrônico. O programa deverá ter as seguintes características:

INTERATIVIDADE: caso tenha dúvidas durante a implementação, consulte o código: https://github.com/amarildolucas/caixa-eletronico

a) Deverá ser apresentada uma interface gráfica inicial com as opções de Sacar, Repor, Consultar Saldo e Finalizar.
b) Para sacar, o usuário deverá informar a quantia desejada. Caso o usuário informe uma quantia inválida (valor menor ou igual a zero), o programa deverá apresentar uma mensagem. Caso seja possível realizar o saque, o programa deverá apresentar a quantidade de notas de R$5, R$10, R$20, R$50 ou R$100 que serão usadas para compor a quantia desejada. OBS: só é possível realizar o saque caso a quantidade de notas existentes no caixa seja suficiente para formar a quantia desejada.
c) Para repor, o usuário deverá informar a quantidade de notas de R$5, R$10, R$20, R$50 e R$100 que serão repostas no caixa. Caso o usuário informe uma quantia inválida (valor menor que zero), o programa deverá apresentar uma mensagem.
d) Ao selecionar a opção Consultar Saldo, o programa deverá apresentar uma estatística mostrando o saldo atual do caixa eletrônico, a quantidade total de saques realizada, o valor total de saques realizados e a quantidade de notas de R$5, R$10, R$20, R$50 e R$100 existentes no caixa.
e) Ao iniciar o programa, o caixa eletrônico deverá estar vazio.

Do ponto de vista da solução apresentada, espera-se que sejam criadas pelo menos duas classes:

- Uma para representar o comportamento do Caixa Eletrônico em si, com métodos para reposição, saque, consulta de saldo, total de saques, etc.;
- Outra para representar a interface gráfica com o usuário, onde serão apresentadas: a entrada de dados, as mensagens de erro, a apresentação dos resultados, etc.

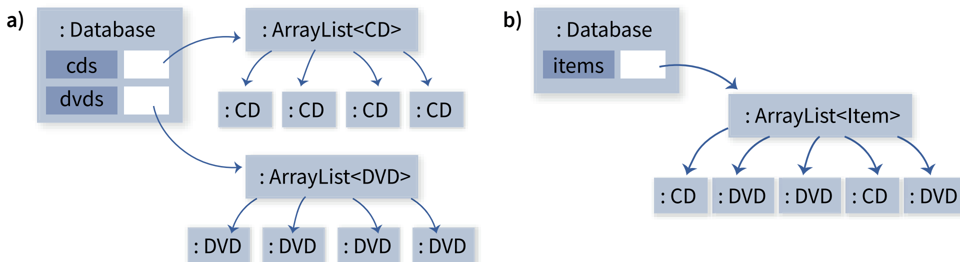
5

HERANÇA E OUTRAS RELAÇÕES ENTRE OBJETOS

INTRODUÇÃO

Quando construímos o protótipo para gerenciar multimídias, apresentado na Figura 107, letra a), a principal preocupação era agrupar os objetos em uma coleção, nesse caso o *ArrayList*. Para isso, foram criados dois *ArrayList* na classe *Database*, um para armazenar as informações dos *CDs* e outro para armazenar os *DVDs*. Embora percebamos que houve muita repetição de código, o protótipo funcionou perfeitamente, com algumas limitações, sendo a principal delas a manutenção do código fonte. Isso se deve à duplicidade de códigos; logo, quase todas as alterações feitas em uma das classes têm que ser replicadas para as demais.

Figura 107 – Coleção de objetos CDs e DVDs



Fonte: Autores.

Para resolver esse problema estrutural de duplicação de código, em nosso projeto, vamos utilizar o mecanismo da herança. Basicamente, vamos extrair as informações comuns tanto para *CDs* quanto para *DVDs* e criar uma terceira classe chamada Item, conforme apresentado na Figura 107, letra b). Dessa forma, podemos dizer que um *CD* é um Item e um *DVD* é um Item, pois ambos possuem informações armazenadas em Item. Nesse momento, você pode estar questionando, o que iremos fazer com as informações pertencentes somente a *DVD* ou *CD*? Essas informações ficarão armazenadas nas suas respectivas classes, evitando assim a duplicidade de código.

Para que possamos utilizar esse recurso da programação orientada a objetos, a herança, temos que aprender novas terminologias. Quando dizemos que *DVD* é um *Item*, necessariamente, *DVD* herda da classe *Item*, o mesmo ocorre com *CD*. Ainda, podemos dizer que a classe *DVD* estende a classe *Item*. Essa expressão tem origem na palavra *extends* que é utilizada, em Java, para definir um relacionamento de herança entre classes. Nesse contexto, a herança nos permite reescrever o código do protótipo para gerenciar multimídias, sem a duplicidade de códigos.

Com o intuito de fornecer conhecimento para melhorar nossa programação, nesta unidade veremos os conceitos sobre herança. Depois vamos enfatizar a herança em Java, abordando os construtores em subclasses, subtipos, conversão de tipos e passagens de parâmetro de subtipos. Além disso, vamos abordar de forma detalhada a sobreposição de métodos em subclasses, na sequência serão detalhadas

as formas de polimorfismo na linguagem *Java*. Como precisaremos criar modelos, ou seja, classes que não podem ser instanciadas, vamos descrever os conceitos de classe abstrata e interface. Por fim, vamos abordar o conceito de herança múltipla.

Cabe ressaltar que ao entender os conceitos do mecanismo de herança, apresentados nessa unidade, você será capaz de construir e remodelar o protótipo do gerenciador de multimídia, visando aumentar a eficiência e reduzir as redundâncias no código fonte.

5.1 HERANÇA

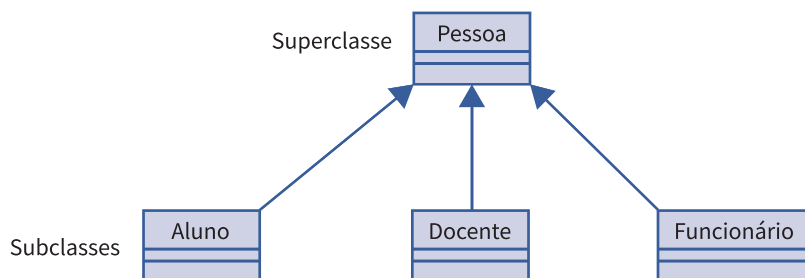
Neste *e-book*, entendemos que herança (ou generalização) é o mecanismo pelo qual uma classe (*sub-classe*) pode estender outra classe (*super-classe*), aproveitando seus comportamentos (métodos) e variáveis possíveis (atributos) (SIERRA; BATES, 2007). Neste sentido, a herança define uma **relação entre classes** do tipo "**é um**", onde uma classe compartilha a estrutura e o comportamento definidos em uma ou mais classes.

Antes de prosseguir vamos detalhar as relações entre classes. Um relacionamento entre duas classes é definido como sendo "**é um**" se a classe B (por exemplo, a*luno*) é derivada de uma classe A (por exemplo, *pessoa*), então B **é** realmente **um** caso especial de A. Neste caso dizemos que a classe B herda de A, logo, B "**é um**" A.

Como existe uma certa similaridade entre a classe B (pessoa) e a classe A (aluno), dizemos que aluno "**é uma**" pessoa que possui características mais específicas, quando comparado com pessoa. Neste panorama, ao reconhecermos as similaridades entre as classes, definimos uma estrutura chamada de **hierarquia de classes**, apresentada na Figura 108, que subdivide em:

- Superclasse ou classe base: A superclasse representa abstrações mais gerais e se subdivide em duas: a direta, em que a subclasse herda explicitamente da superclasse, e a indireta, onde a subclasse herda de dois ou mais níveis acima na hierarquia de classes;
- Subclasses ou classe derivada: A subclasse é uma implementação de uma classe que tem como base uma outra classe, denominada superclasse. Nesse sentido, ela representa uma abstração em que os atributos e os métodos são adicionados, modificados e/ou removidos.

Figura 108 – Hierarquia de classes



Fonte: Autores.

Neste contexto, já que existe uma superclasse e uma subclasse, agora vamos usar uma estrutura para fazer a ligação entre elas. Para isto, podemos usar as estruturas de generalização e especialização:

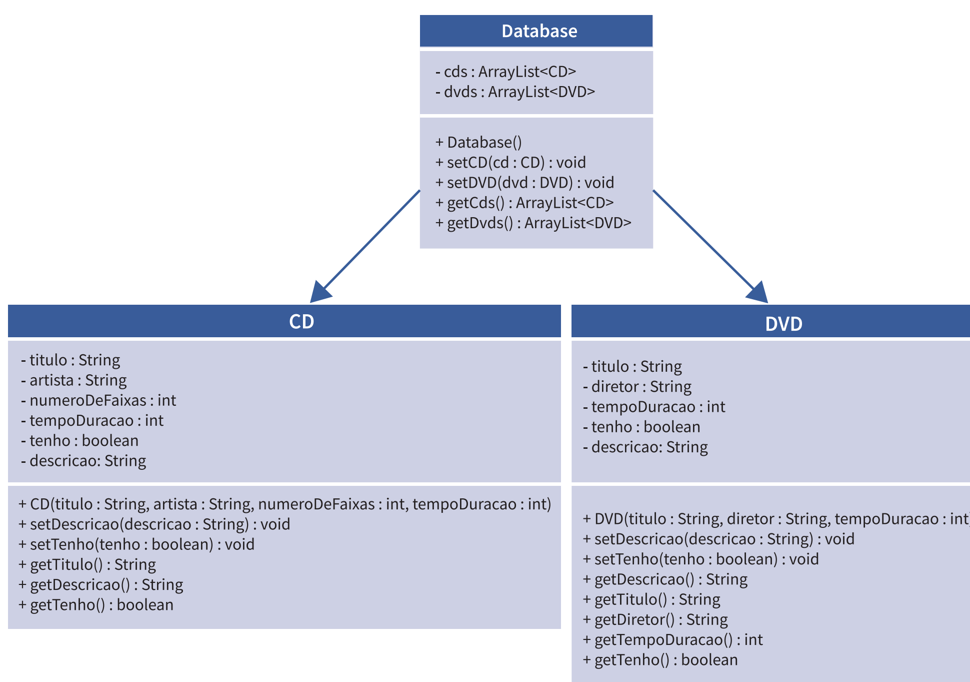
- Especialização: Como as subclasses são casos especiais de suas superclasses, o termo especialização é usado para se referir ao processo de derivar

uma classe de outra;

- Generalização: A generalização, por outro lado, é o termo usado para se referir ao processo oposto. Reconhece as características comuns de várias classes existentes e cria uma nova superclasse comum para todas elas, que, em nosso exemplo, na Figura 108, é a classe *Pessoa*.

Vamos voltar ao exemplo da unidade anterior, o protótipo para armazenar uma coleção de objetos *CDs* e *DVDs*. Veja que temos alguns códigos repetidos, conforme mostra a Figura 109. Analisando especificamente a classe *Database* temos dois *ArrayList* para objetos semelhantes, sendo eles *CD* e *DVD*. Passando para a análise das classes *DVD* e *CD*, percebemos que existem repetição dos atributos *titulo*, *tempoDuracao*, *tenho* e *descrição* e, quando analisamos os métodos, também existe repetição de código.

Figura 109 – Hierarquia de classes – coleção de objetos *CDs* e *DVDs*



Fonte: Autores.

Ao finalizarmos a análise da Figura 109, percebemos que existem muitos atributos comuns, principalmente entre as classes *DVD* e *CD*. Sabemos que, quando há código repetido, será necessário muito esforço para manutenção.

Para poder otimizar o código temos que responder à seguinte pergunta: o que as classes *DVD* e *CD* têm em comum? Basicamente, além das informações já citadas acima, elas fazem parte de um sistema de controle da coleção de objetos de *CDs* e *DVDs*. Com base nesses fatos, aplicaremos a técnica da generalização para eliminar as duplicidades e deixar o protótipo mais eficaz.

Inicialmente temos que criar uma outra classe para implementar a generalização. Esta classe generalista agrupa os atributos comuns de *DVD* e *CD* (*titulo, tempo-*

Duracao, *tenho* e *descrição*). Também é importante que o nome dessa nova classe seja uma generalização de *DVD* e *CD*. Neste exemplo, escolhemos *item*. A Figura 110 apresenta, graficamente, a criação da superclasse *Item*, com os atributos comuns.

Figura 110 – Superclasses Item

Item
- titulo : String - tempoDuracao : int - tenho : boolean - descricao: String
+ Item() + Item(titulo : String, tempoDuracao : int) + setTenho(tenho : boolean) : void + setDescricao (descricao : String) : void + getTenho() : boolean + getDescricao() : String + getTitulo() : String + getTempoDuracao() : int

Fonte: Autores.

O nosso próximo passo é retirar os atributos comuns das duas subclasses, que são *CD* e *DVD*. Relembrando, a superclasse (ou classe pai), é quem possui os atributos comuns. Já as subclasses (ou classe filha), é a classe que herda os membros da superclasse e ainda possui os seus atributos específicos. A Figura 111 apresenta as subclasses *CD* e *DVD* após a remoção dos atributos comuns, que foram inseridos na superclasse.

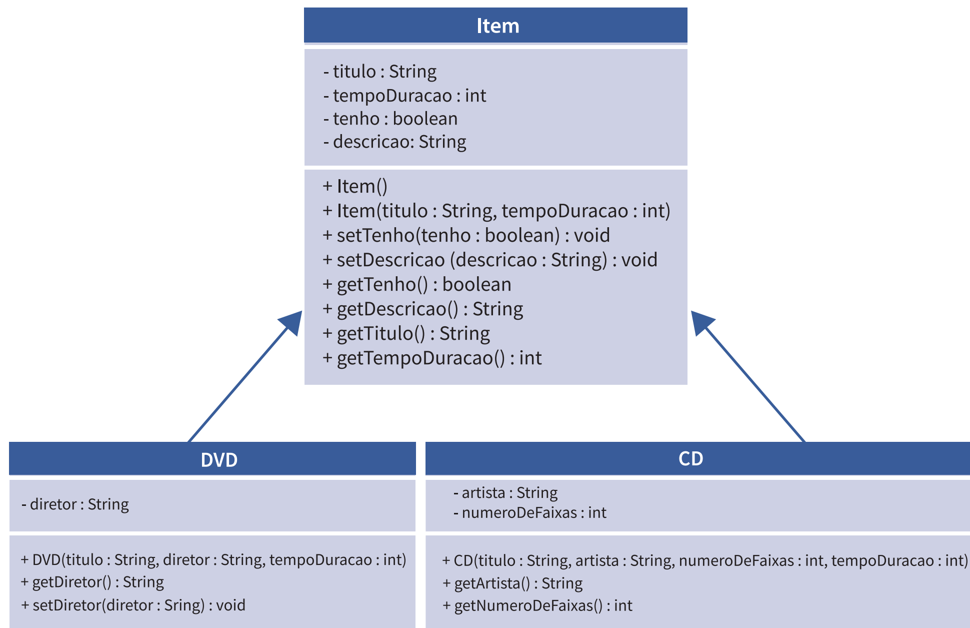
Figura 111 – Subclasses *CD* e *DVD* com atributos comuns removidos

DVD	CD
- diretor : String	- artista : String - numeroDeFaixas : int
+ DVD(titulo : String, diretor : String, tempoDuracao : int) + getDiretor() : String + setDiretor(diretor : Sring) : void	+ CD(titulo : String, artista : String, numeroDeFaixas : int, tempoDuracao : int) + getArtista() : String + getNumeroDeFaixas() : int

Fonte: Autores.

Agora vamos fazer o relacionamento de generalização entre subclasses (*CD* e *DVD*) e a superclasse (*Item*). De acordo com a Figura 112, o relacionamento de generalização é representado por uma reta com uma seta na ponta, onde a seta sempre aponta para a superclasse.

Figura 112 – Generalização entre as classes CD, DVD e Item



Fonte: Autores.

Os benefícios da Herança

Embora construímos um exemplo demostrando a eficácia da herança, apresentado na Figura 112, vamos destacar alguns pontos importantes, pois a herança é um dos aspectos mais poderosos da programação orientada a objetos, são eles:

- Reduz enormemente a redundância de código, acarretando uma sensível diminuição na carga de manutenção de código;
- Os códigos das subclasses são bem menores do que seriam sem herança, pois a subclasse contém apenas a essência do que diferencia da superclasse. Ao final, teremos como resultado a otimização de tamanho das aplicações desenvolvidas em programação orientada a objetos, pois elas ficam menores, quando comparadas com as aplicações desenvolvidas de forma tradicional;
- Por meio da herança, podemos reutilizar e estender o código que já foi totalmente testado sem modificar uma linha de código e, assim, evitar testar novamente grande parte do código desenvolvido no protótipo. Se usarmos uma linguagem não orientada a objetos, teremos que alterar as partes do código, o que acarretaria em testar novamente toda a aplicação; só assim podemos certificar que o código alterado está funcionando corretamente;
- Podemos derivar uma nova classe de uma classe existente, mesmo se não tivermos o código fonte. Para isso, basta ter uma versão da classe em *bytecode* compilada, ou seja, não precisamos do código fonte original da classe para que o mecanismo de herança funcione corretamente.

Herança em Java

Para descobrir os detalhes, em nível de código fonte, sobre herança vamos examinar alguns códigos-fonte, a começar na classe *Item* (conforme Figura 113).

Figura 113 – Classe *Item*

```
6    package br.ufsm.midias;
7
8    /**
9     *
10    * @author fabio
11    */
     public class Item {
13       private String titulo;
14       private int tempoDuracao;
15       private boolean tenho;
16       private String descricao;
17
      //Contrutores e métodos

66
67   }
```

Fonte: Autores.

Analisando a classe Item, percebemos que ela é uma classe como as demais estudadas até o momento. Ao analisarmos a sequência de palavras na assinatura da classe, na linha 12 da Figura 113, concluímos que ela é composta por 3 palavras:

- *public*: palavra reservada que define a visibilidade da classe;
- *class*: palavra reservada que indica uma classe;
- *Item*: nome da classe.

Passaremos agora à análise do código das classes *DVD* e *CD*, apresentados na Figura 114. Vamos começar analisando a linha 5 da Figura 114, que é a assinatura da classe. Além das informações vistas (*public*, *class* e nome da classe), neste caso *DVD* ou *CD*, aparecem duas novas informações:

- *extends*: palavra reservada usada para implementar herança em Java;
- *Item*: Classe pai ou superclasse.

As informações *DVD extends Item* define que *DVD* é uma subclasse de Item, e que *DVD* herda os atributos e métodos de Item. Neste contexto, dentro do panorama de herança, a classe *DVD* deve implementar novos serviços que não são oferecidos por *Item*.

Figura 114 – Classe *DVD* e *CD*

```
1   package br.ufsm.midias;
2   /**
3    * @author fabio
4    */
5   public class DVD extends Item{
6       private String diretor;
7

    //Contrutores e métodos

31  }
```

```
1   package br.ufsm.midias;
2   /**
3    * @author fabio
4    */
5   public class CD extends Item{
6       private String artista;
7       private int numeroDeFaixas;

    //Contrutores e métodos

32  }
```

Fonte: Autores.

Construtores em subclasses

Relembrando, ao usarmos a palavra reservada *new* para criar um objeto de uma determinada classe estamos alocando espaço na memória. A todo esse processo chamamos de instanciação. Ao fazer essa ação, a linguagem Java requer a inicialização de alguns atributos. O ato de inicializar ou construir é feito pelos construtores da classe, como mostra o exemplo da Figura 115.

Figura 115 – Construtores das classes *Item* e *DVD*

```
1   package br.ufsm.midias;
2   /**
3    * @author fabio
4    */
5   public class Item {
6       private String titulo;
7       private int tempoDuracao;
8       private boolean tenho;
9       private String descricao;
10
11      //Construtor padrão
12      public Item(){ }
13      //Construtor com parâmetros
14      public Item(String titulo, int tempoDuracao){
15          this.titulo = titulo;
16          this.tempoDuracao = tempoDuracao;
17          tenho = false;
18          descricao = "<não comentou>";
19      }
```

```
1   package br.ufsm.midias;
2   /**
3    * @author fabio
4    */
5   public class DVD extends Item{
6       private String diretor;
7
8       /**
9        * Construtor para objetos da classe DVD
10       */
11      public DVD(String titulo, String diretor, int tempoDuracao){
12          super(titulo,tempoDuracao);
13          this.diretor = diretor;
14      }
15
```

Fonte: Autores.

O construtor apresentado na Figura 115, da classe *Item*, se assemelha a todos que estudamos até o momento. Já construtor da classe *DVD* possui uma particularidade, a palavra reservada *super*. A palavra super, na linha 12, da classe *DVD*, invoca o construtor da superclasse que é *Item*. Ele recebe os dois parâmetros (*super(titulo, tempoDuracao)*) e repassa ao construtor da classe, que faz as atribuições às variáveis da classe. Então, quando criarmos um objeto do tipo *DVD*, por exemplo, utilizando *new DVD(titulo, diretor, tempoDuracao)*, na sequência a classe *DVD* invoca o construtor *Item*, por meio do comando *super(titulo,tempoDuracao)*, e então seus atributos serão preenchidos com os dados enviados por parâmetro. Na linha 13, da classe *DVD*, é realizada a atribuição do parâmetro *diretor* ao atributo *this.diretor*.

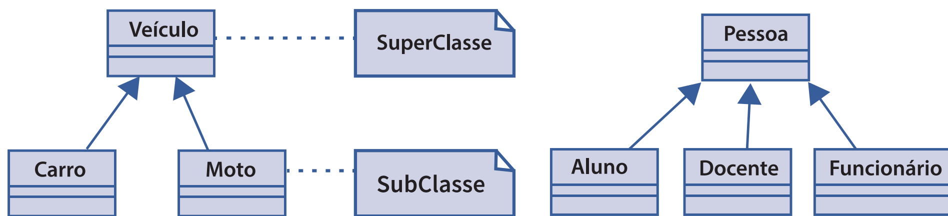
De acordo com as boas práticas de programação em Java, devemos inserir primeiramente a chamada do construtor da superclasse, dentro do construtor da subclasse, depois o restante do código. Caso você esqueça de inserir a chamada do construtor da superclasse, a linguagem Java irá inserir uma chamada padrão,

sem parâmetros, equivalente a *super()*. Isso faz com que os atributos sejam inicializados adequadamente.

Subtipos

Já aprendemos, conforme apresentado na Figura 108, que *Aluno é um* tipo de *Pessoa*. Portanto, todo objeto *Aluno* também é um objeto *Pessoa*. Concluímos que *Aluno* é um subtipo de *Pessoa*. Outro exemplo apresentado na Figura 116, indica que todo *Carro* é um *Veículo*, e toda *Moto* é um *Veículo*; portanto, *Carro* é um subtipo de *Veículo*.

Figura 116 – Exemplos de superclasse e subclasse



Fonte: Autores.

Veja que o conceito de subtipo se assemelha ao conceito de hierarquia de classes. No exemplo apresentado na Figura 116, os subtipos são as nossas subclasses. Nesse contexto, quando queremos atribuir um objeto a uma variável, o tipo da variável tem que corresponder ao tipo do objeto, por exemplo:

- *Veiculo v1 = new Veiculo()*;

Neste exemplo, temos uma atribuição correta, pois o objeto *veículo* é atribuído à variável *v1*, que foi declarada como sendo do tipo *Veículo*. Agora vamos analisar o conceito dos subtipos de uma forma mais eficiente:

- Uma variável pode armazenar objetos do tipo em que ela foi declarada ou de um subtipo.

Agora vamos ver o mesmo conceito, visto da ótica da herança:

- Uma variável pode armazenar objetos do tipo em que ela foi declarada ou objetos de uma subclasse.

A seguir, veremos alguns exemplos de atribuição de objetos a variáveis:

- *Veiculo v2 = new Carro()*;
- *Veiculo v3 = new Moto()*;
- *Veiculo v1 = new Veiculo()*;

Todos os exemplos acima estão corretos. A seguir, serão apresentados alguns exemplos incorretos:

- *Carro c1 = new Veiculo()*;
- *Carro c2 = new Moto()*;

No primeiro exemplo, acima, o objeto *veiculo* não é subclasse de *Carro*; portanto, a primeira atribuição causa erro. A segunda atribuição atribui o objeto *Moto* a variável *c2* que é do tipo *Carro*, como *moto* não é subtipo de *Carro* teremos um erro no instante da compilação.

O erro ocorre por conta do conceito de substituição. Este conceito é nativo nas linguagens orientadas a objetos. Basicamente, a regra é que os objetos de um subtipo podem ser atribuídos aos objetos do supertipo.

ATENÇÃO: pesquise sobre os conceitos de hierarquia de classes para compreender melhor o conceito de substituição.

Conversão de tipos

Quando há incompatibilidade de tipos, podemos fazer uma conversão, ou *casting*. Para isso, é necessário que a conversão seja indicada ao compilador. O ato de indicar a conversão é denominado de *casting*. Podemos fazer a conversão de tipos primitivos ou de objetos.

Nos tipos primitivos, cabe ressaltar que a conversão de dados é bem simples e ocorre, na maioria das vezes, para tentar minimizar o consumo de memória. Se tentamos atribuir um tipo de dado que consome menos memória para um que consome mais memória, a conversão é realizada implicitamente, ou seja, basta fazer a atribuição, conforme apresentado na Figura 117. A conversão ocorre na linha 24, onde a *variavelInteiro* foi atribuída a *variavelFloat*. Ao contrário, quando temos um tipo que ocupa mais memória para um que consome menos, temos que fazer a conversão explícita, conforme apresenta a linha 28.

Figura 117 – Conversão de tipos primitivos

```
19            float variavelFloat;
20            int variavelInteiro;
21            int variavelIntRecebeFloat;
22            variavelInteiro = 200;
23
24            // Conversão implícita de tipos primitivos
25            variavelFloat = variavelInteiro;
26
27            //Conversão explícita de tipos primitivos
28            variavelIntRecebeFloat = (int)variavelFloat;
```

Fonte: Autores.

A conversão de tipos também pode ser usada para converter tipos de Objetos. Sempre que uma classe for genérica, ela poderá receber qualquer subtipo. Isto é possível graças à conversão implícita, apresentada nas linhas 47 e 48 da Figura 118.

Figura 118 – Conversão de objetos

```
41          //Criação/Instanciação de objetos
            Item novoItem = new Item();
43          DVD novoDVD = new DVD();
44          CD novoCD = new CD();
45
46          //Conversão implícita
            novoItem = novoDVD;//Correto
48          novoItem = novoCD;//Correto
49
50          //Conversão explícita
            novoDVD = (DVD)novoItem;//Correto
52          novoCD = (CD)novoItem;//Correto
53
54          //Conversão explícita
            novoDVD =(DVD)novoCD;  //Erro
            novoCD = (CD)novoDVD;//Erro
```

Fonte: Autores.

Nas linhas 51 e 52, temos uma conversão explícita, onde estamos realizando a atribuição de superclasse para subclasse. Nessas duas linhas, temos que inserir o operador de conversão de tipo, para a linha 51 é (DVD) e para a linha 52 é (CD). Estes comandos indicam ao compilador que o objeto da linha 51 é um *DVD* e respectivamente na linha 52 um *CD*.

Já nas linhas 54 e 55, temos um exemplo de conversão explícita, que falha, embora tenham sido inseridos os operadores de conversão corretamente. A falha é atribuída a não existência do relacionamento subtipo/supertipo para os objetos novoDVD e novoCD e vice-versa.

Passagem de parâmetro de subtipos

Além das atribuições, o conceito de substituição também se aplica a passagens de parâmetros; por isso, podemos atribuir ao mesmo método, os objetos do tipo *CD* e *DVD*.

Conforme apresentado na Figura 119, foi projetada uma classe *Database*, contendo um *ArrayList* para receber objetos do tipo *Item* na linha 17. Vale lembrar que a classe item é a superclasse de *DVD* e *CD*; logo, estes são subtipos de *Item*. Na linha 27, temos o método *adicionaNovoItem*. Este método recebe um tipo de item, que pode ser *CD* ou *DVD*, e faz adição do respectivo objeto ao banco de dados, neste caso o *ArrayList*.

Figura 119 – Método para receber os subtipos

```
15   public class Database{
16       //Atributo
17       private ArrayList<Item> itens;
18
19       /**
20        * Construtor da classe Database.
21        * constroi um banco de dados vazio
22        **/
23       public Database(){
             itens=new ArrayList<Item>();
25       }
26
27       public void adicionaNovoItem(Item novoItem){
28           itens.add(novoItem);
29       }

42   }
```

Fonte: Autores.

Ao analisarmos o código da Figura 120 fica bem clara a passagem de parâmetros de subtipos.

Figura 120 – Passagens dos subtipos *novoDVD* e *novoCD*

```
//Cria e instancia o objeto novoDVD
DVD novoDVD = new DVD(···);

//Armazena o novoDVD em dataBAse
dataBase.adicionaNovoItem(novoDVD);
```

```
//Cria e instancia o objeto novoCD
CD novoCD = new CD(···);

//Armazena o novoCD em dataBAse
dataBase.adicionaNovoItem(novoCD);
```

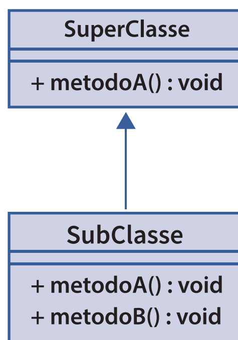
Fonte: Autores.

Inicialmente, na primeira linha foram declarados/instanciados os objetos *novoDVD* e *novoCD*. Já na linha seguinte é realizada a chamada do método *adicionaNovoItem*, passando o parâmetro *novoDVD* e *novoCD*. Relembrando, que essas atribuições só podem ser concretizadas graças às regras de subtipos.

5.2
SOBREPOSIÇÃO DE MÉTODOS

A sobreposição (ou *overriding*) ocorre quando o mecanismo de execução assume a sobreposição e executa o método que encontrar primeiro, percorrendo a hierarquia de classes de baixo para cima. Para que haja sobreposição de métodos, precisamos criar um novo método na classe filha (ou subclasse) contendo a mesma assinatura e mesmo tipo de retorno do método sobrescrito na classe pai (ou superclasse), conforme apresentado na Figura 121.

Figura 121 – Sobreposição do metodoA



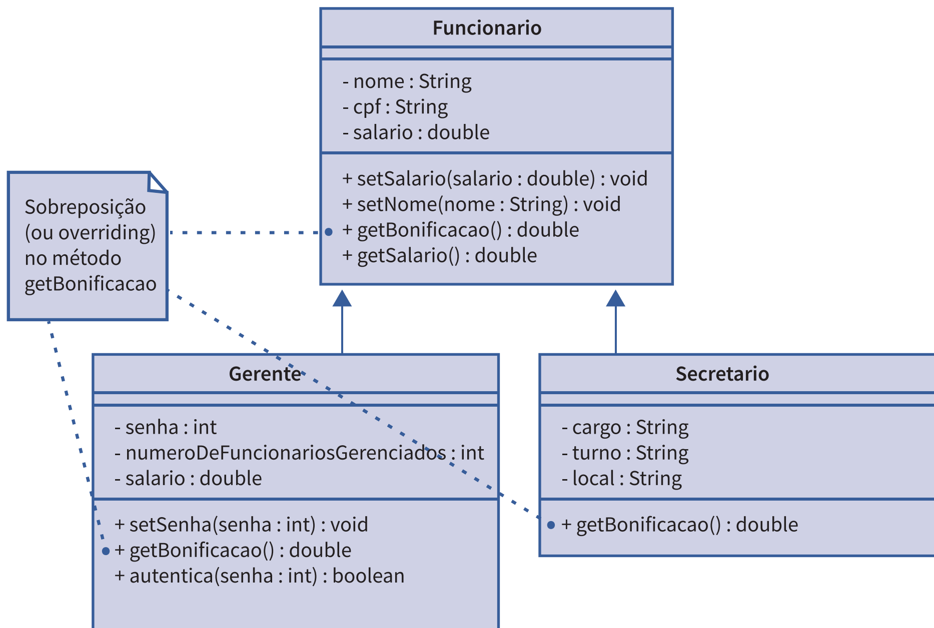
Fonte: Autores.

Vale a pena frisar: ao dizer que o método possui a mesma assinatura, estamos dizendo que ele deve possuir, quando comparado com o método sobrescrito:

- o mesmo nome
- a mesma quantidade de parâmetros e
- o mesmo tipo de parâmetros.

Já em relação ao tipo de retorno, ele pode ser um subtipo do tipo de retorno, ou seja, tem que ser uma subclasse do retorno do método sobrescrito. Por fim, ao criar uma sobreposição de um método, temos como objetivo o de especializar os métodos herdados das superclasses, alterando o comportamento destes nas subclasses e, dessa forma, atendendo às necessidades da subclasse de forma mais adequada. A Figura 122 apresenta a sobreposição do método *getBonificação()* nas três classes: *Funcionario*, *Gerente* e *Secretario*. Basicamente, o que muda neste método de uma classe para a outra é a porcentagem da bonificação, por exemplo, o gerente recebe 25%, o secretário 15% e os demais funcionários 5% sobre o salário.

Figura 122 – Sobreposição de método



Fonte: Autores.

Agora vamos analisar como é o código de cada uma dessas classes, a começar na superclasse *Funcionario*, conforme mostra a Figura 123.

Figura 123 – Classe funcionário

Indica que tem subclasse

Indica que tem Sobreposição deste método

```
   public class Funcionario {
13     private String nome;
14     private String cpf;
15     private double salario;
16
17     public void setSalario(double salario) {
18         this.salario = salario;
19     }
20     public void setNome(String nome) {
21         this.nome = nome;
22     }
23     //Bonificação(5%) básica para um funcionário
       public double getBonificacao() {
25         return this.salario * 0.05;
26     }
27
28     public double getSalario() {
29         return salario;
30     }
```

Fonte: Autores.

Na classe funcionário, temos duas observações a serem feitas: a primeira é a indicação de ocorrência de herança e a segunda indica a sobreposição do método *getBonificacao()*. Agora vamos analisar o código da classe *Gerente*, conforme disposto graficamente na Figura 124.

Figura 124 – Classe Gerente

```
12  public class Gerente extends Funcionario{
13      private int senha;
14      private int numeroDeFuncionariosGerenciados;
15
16      public void setNumeroDeFuncionariosGerenciados(int numeroDeFuncionariosGerenciados) {
17          this.numeroDeFuncionariosGerenciados = numeroDeFuncionariosGerenciados;
18      }
19      public void setSenha(int senha) {
20          this.senha = senha;
21      }
22
23      //Bonificação(25%) para o gerente
24      @Override
        public double getBonificacao() {
26          return super.getSalario()* 0.25;
27      }
28
29      public boolean autentica(int senha) {
30          if (this.senha == senha) {
31              System.out.println("Senha correta!");
32              return true;
33          } else {
34              System.out.println("Senha incorreta!");
35              return false;
36          }
37      }
38  }
39
```

Fonte: Autores.

O método sobrescrito, na Figura 124, linha 24, apresenta o termo *@Override* indicando que o método *getBonificacao* desta classe sobrescreve outro método na *superclasse*. Na linha 26, temos a chamada ao método *getSalario()* da superclasse, por meio da palavra reservada *super*. Veja que este método adiciona 25% sobre o salário, sendo uma remuneração específica para gerentes. Vamos analisar agora a classe secretário, apresentada na Figura 125.

Figura 125 – Classe *Secretario*

```
12  public class Secretario extends Funcionario {
13      private String cargo;
14      private String turno;
15      private String local;
16
17      //Bonificação(10%) para o secretário
18      @Override
        public double getBonificacao() {
20          return super.getSalario() * 0.10;
21      }
22  }
```

Fonte: Autores.

A especificidade do método *getBonificacao()*, na linha 19 da Figura 125, está na adequação da porcentagem da bonificação sobre o salário para o secretário. Na linha 20, temos a chamada do método *getSalario()* da classe *Funcionario*, por meio da palavra reservada super e, na sequência, a obtenção da porcentagem da bonificação.

5.3 POLIMORFISMO

Já trabalhamos um pouco com polimorfismo sem saber o conceito. Agora vamos conceituar e aprofundar um pouco mais, haja vista a importância do polimorfismo no contexto da programação orientada a objetos. Segundo o dicionário Aurélio (online, 2019), a palavra polimorfismo nos oferece as seguintes definições:

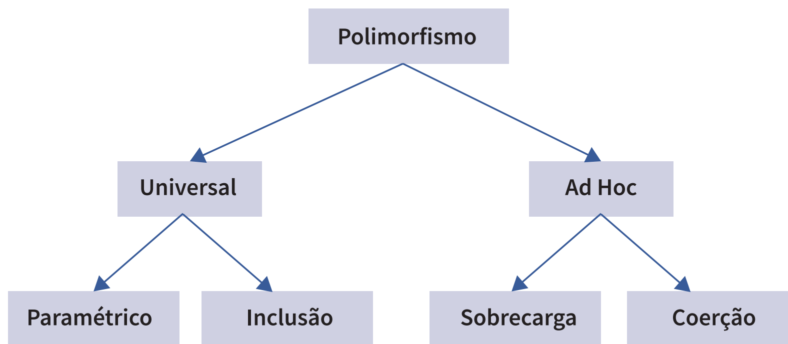
- Polimorfo: que se apresenta sob numerosas formas; multiforme; sujeito a variar de forma.
- Polimorfismo: em botânica, polimorfismo foliar significa que um vegetal apresenta folhas de vários tipos morfológicos.

Vamos sintetizar o conceito de polimorfismo, e estender para a terminologia de orientação a objetos, como:

- Polimorfismo: é quando, em determinadas situações, um objeto pode se comportar de maneiras diferentes, ao ser acionado por meio de mensagens.

O polimorfismo pode ser classificado em 4 diferentes modalidades, para as linguagens orientadas a objetos. A Figura 126 representa essa classificação (CARDELLI; WEGNER, 1985):

Figura 126 – Modalidades do polimorfismo



Fonte: Autores.

Qual é a diferença entre polimorfismo universal e *ad hoc*? Basicamente, no universal um método pode ser executado para um conjunto *ilimitado* de tipos de dados diferentes. Ao se tratar do *ad hoc*, um método pode ser executado para um conjunto *limitado* de tipos de dados. A seguir, vamos detalhar ainda mais as modalidades de polimorfismo, conforme disposto na Figura 126.

Paramétrico

Embora a linguagem Java não ofereça a criação de métodos paramétricos, basicamente, o conceito de polimorfismo paramétrico incorporado em Java é a geração de um novo código de acordo com o tipo previamente definido. Como exemplo, citamos as coleções, a classe *List*, que podem ser compostas por objetos de diferentes tipos. Quando declaramos um *ArrayList* obrigatoriamente temos que definir o tipo, veja:

- *ArrayList<ClassType>,*
- *ArrayList<Item>,*
- *ArrayList<CD>,*
- *ArrayList<DVD>,*

Considere as definições de *ArrayList* acima: o primeiro foi definido para armazenar referências de objetos do tipo *ClassType*, o segundo para armazenar referências de objetos do tipo *Item*, o terceiro para armazenar referências de objetos do tipo *CD* e o quarto para armazenar as referências de objetos do tipo *DVD*. Veja que os tipos *ClassType*, *Item*, *CD* e *DVD*, a grosso modo, podem ser considerados como parâmetros para a construção do *ArrayList*.

Inclusão

A inclusão é usada em casos onde há necessidade de modelar subtipos e herança; por isso, a inclusão está relacionada ao conceito de hierarquias de tipos. Neste sentido, os métodos herdados pelas subclasses automaticamente tornam-se polimórficos. Este conceito foi apresentado no tópico 5.2, denominado de *sobreposição de métodos*.

Sobrecarga

A sobrecarga de métodos ou construtores ocorre quando existem dois ou mais métodos/construtores com o mesmo nome, mas com assinaturas diferentes. Nós já discutimos sobre o assunto na unidade 3, tópico 3.3. Caso tenha alguma dúvida refaça a leitura. Vamos aproveitar esse espaço para discutir a diferença entre sobrecarga e sobreposição de métodos, qual seria a diferença? Basicamente, na sobrecarga, temos dois métodos com o mesmo nome, mas com assinaturas diferentes. Logo, o compilador Java determina qual método deve ser invocado pelo tipo de parâmetro passado. Já na sobreposição o método existe em uma classe pai e é reescrito na classe filha para alterar o comportamento, com a finalidade de deixar o método na classe filha mais específico. Cabe ressaltar que a assinatura do método deve ser igual tanto na classe pai como na classe filha.

SAIBA MAIS: pesquise mais sobre sobrecarga e sobreposição de métodos.

Na sobrecarga de operadores, ocorre quando um operador, por exemplo, o operador +, pode realizar diferentes ações. A Figura 127 apresenta a sobrecarga do operador +. Na linha 21, é apresentada a sobrecarga do operador aritmético da adição entre dois objetos do tipo *String*. Neste caso, o operador concatena as *Strings*. Já na linha 25 o operador + é aplicado a dois números do tipo inteiros, logo ele irá somar os dois números.

Figura 127 – Sobrecarga de operadores

```
19        String nome="Fabio";
20        String sobreNome="Parreira";
21        System.out.println(nome+" "+sobreNome);
22
23        int num1=2;
24        int num2=3;
25        System.out.println("Soma 2 + 3 = " +(num1+num2));
```

Fonte: Autores.

Coerção

Assim como nas demais linguagens de programação, em Java é definida uma hierarquia de tipos primitivos. Como exemplo, podemos citar: tipos *int* são subtipo de *float* (vimos essa teoria no item *Conversão de tipos*). Agora que você já sabe que a conversão de tipos é uma modalidade de polimorfismo vamos detalhar um pouco mais este conceito. Neste exemplo, temos como fornecer dois tipos de dados em uma certa hierarquia, ou seja, o tipo da variável que irá receber o conteúdo é supertipo da informação a ser atribuída. Dessa forma, o compilador Java faz a conversão automática, como mostra a Figura 128.

Figura 128 – Conversão automática de tipos

Tipo primitivo	Conversão automática
double	Não tem conversão automática
float	double
long	float ou double
int	long, float ou double
char	int, long, float ou double
short	int, long, float ou double
byte	short, int, long, float ou double
boolean	Não tem conversão automática

Fonte: Autores.

Por outo lado, quando precisamos fazer o inverso, ou seja, uma conversão onde o tipo da variável que irá receber o conteúdo é subtipo da informação a ser atribuída? Neste caso, usamos a conversão explícita ou *casting*. A Figura 129 apresenta as possibilidades de *casting*, em Java.

Figura 129 – Conversão de tipos usando *casting*

Tipo primitivo	Conversão usando casting
double	(byte), (short), (char), (int), (long) ou (float)
float	(byte), (short), (char), (int) ou (long)
long	(byte), (short), (char) ou (int)
int	(byte), (short) ou (char)
char	(byte) ou (short)
short	(byte)
byte	Não tem conversão automática
boolean	Não tem conversão automática

Fonte: Autores.

Vale frisar que nas conversões, apresentadas na figura 129, há um rebaixamento na hierarquia de tipos e além disso pode ocasionar truncamento no resultado.

5.4
CLASSE ABSTRATA

Dizemos que uma classe abstrata serve como *modelo* para as outras classes que dela herdam. Logo, uma classe declarada com o modificar *abstract*, necessariamente, não precisa ter métodos abstratos. Ela é definida de forma semelhante a uma classe concreta, a diferença está no modificador *abstract* inserido antes da cláusula *class*. A Figura 130 apresenta uma classe abstrata que possui dois métodos. Veja que o método da linha 15 é abstrato, já o outro é um método padrão, apresentado na linha 18. Ao observarmos o método abstrato percebemos que ele é definido com o modificador *abstract* e não possui corpo, ou seja, não tem implementação. Em Java, qualquer classe que possui pelo menos um método abstrato é considerada abstrata, mesmo que não seja declarada como *abstract*.

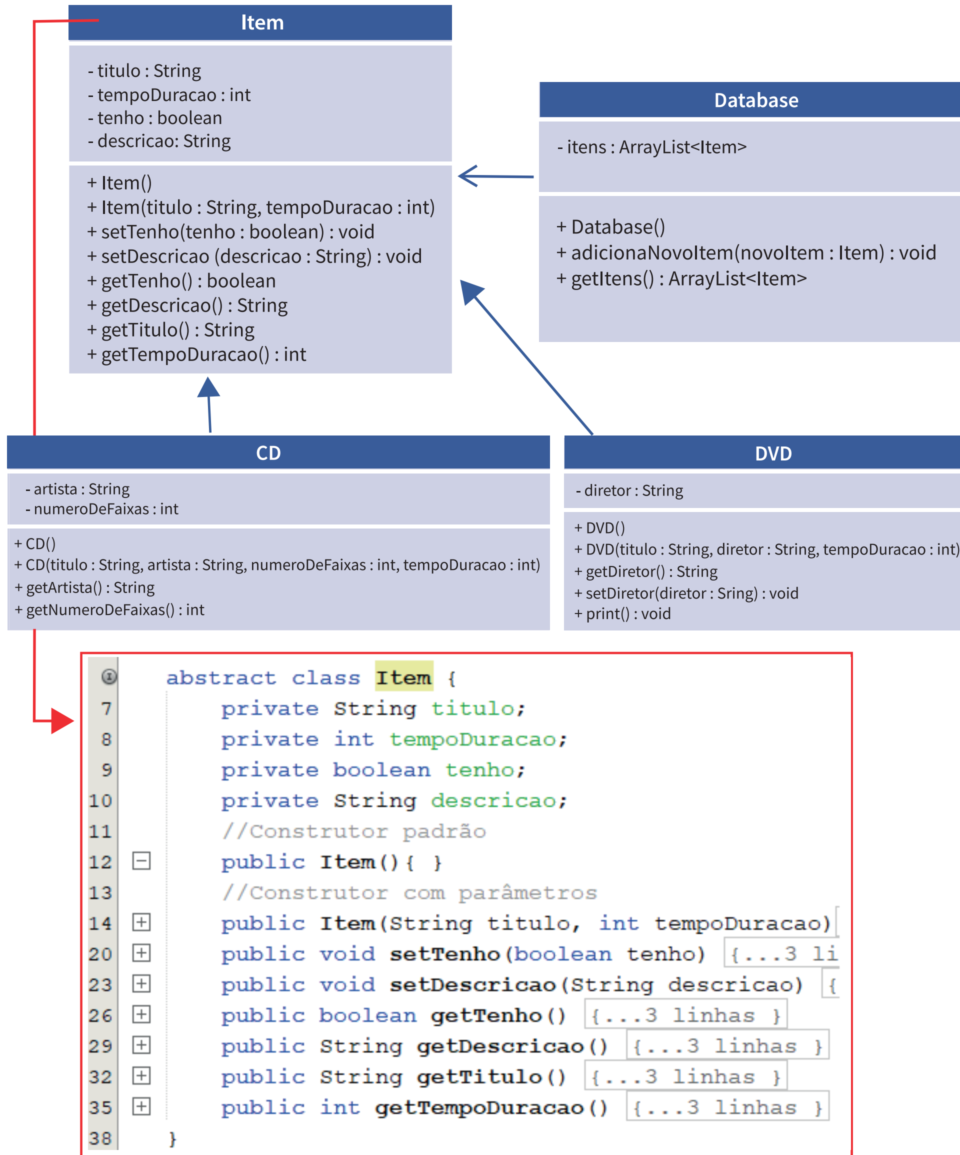
Figura 130 – Classe abstrata

```
    abstract class ClasseAbstrata {
13
14      //Método abstrato
        public abstract void metodoAbstrato();
16
17      //Método padrão
18      public void metodoPadrao(){
19
20      }
21  }
```

Fonte: Autores.

A Figura 131 apresenta a classe abstrata *Item*. É importante notar que essa classe não possui nenhum método abstrato; logo, ela foi definida como abstrata para não ser instanciada, servindo apenas como modelo para definir o tipo a ser armazenado no *ArrayList*, na classe *Database* e como superclasse para *CD* e *DVD*.

Figura 131 – Classe abstrata *Item*



Fonte: Autores.

Cabe ressaltar que toda subclasse de uma classe abstrata, tem que fornecer uma implementação para os métodos abstratos da superclasse. Caso a subclasse não implemente os métodos devemos construir uma declaração abstrata do método. Com isso, a subclasse passa a ser considerada como sendo abstrata. Vamos comentar, abaixo, as principais características envolvendo classe abstratas:

- Classe abstrata tem pelo menos um método abstrato, ou seja, um método que não tem corpo;
- Não podemos instanciar classes abstratas;
- Para criar uma subclasse de uma classe abstrata, por meio de herança, usamos a palavra reservada *extends*;
- Os métodos abstratos são identificados pela palavra reservada *abstract*.

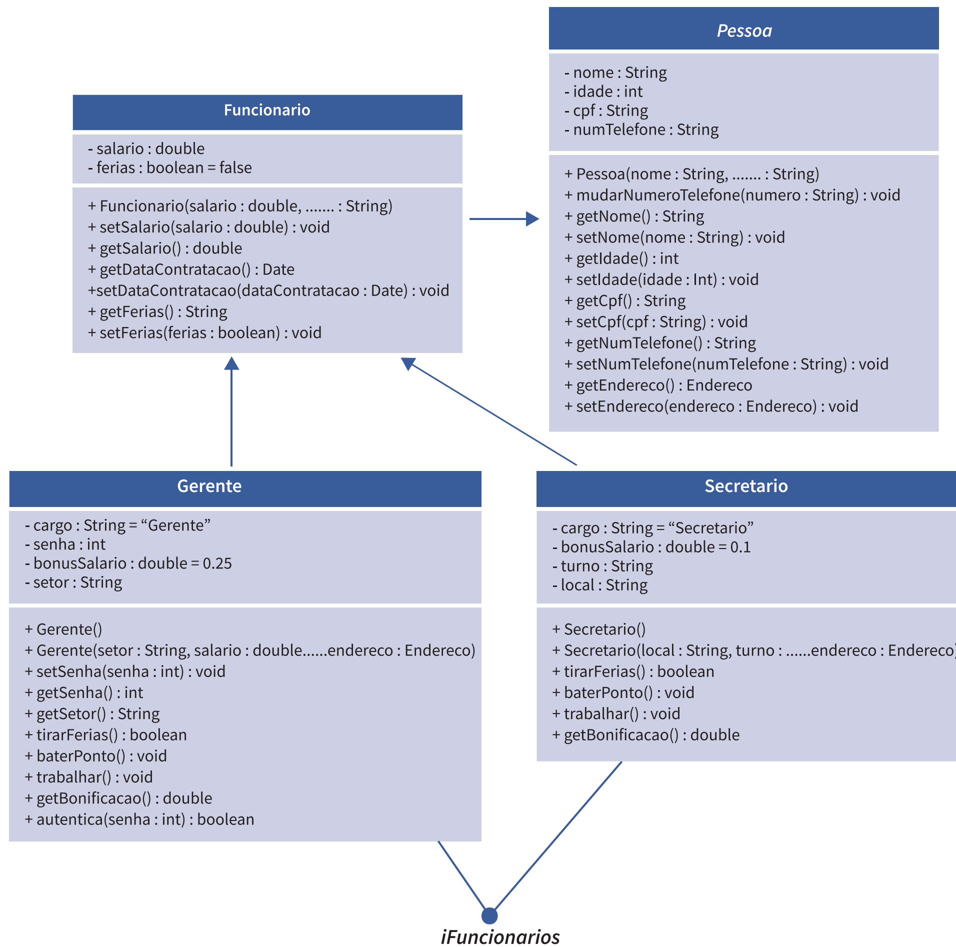
5.5
INTERFACE

A Interface é uma classe 100% (cem por cento) abstrata, ou seja, todos métodos são abstratos. Nesse sentido, a interface possui apenas a definição dos métodos, ou seja, nenhum dos métodos possuem implementação. Quem implementa estes métodos? Quando uma classe implementa uma interface, por meio da palavra reservada *implements*, obrigatoriamente ela deve implementar todos os métodos dessa interface. A interface não contém nenhum código de implementação, apenas as assinaturas dos métodos, cujo código deve ser construído nas classes que implementarem a interface. Nesse sentido, cada classe que implementar a interface pode construir uma versão do método que melhor adeque a classe. Dessa forma, permite uma implementação mais específica para cada classe. A seguir, vamos elencar as principais características da interface:

- Uma interface não possui atributos;
- Uma classe pode implementar (*implements*) várias interfaces, mas pode ter apenas uma superclasse (*extends*);
- Todos os métodos são abstratos e públicos;
- A interface não pode ser instanciada;
- Interface não possui construtor;
- Ao criar uma classe concreta, implementando uma interface, devem ser criados os corpos de todos os métodos da interface.

Vamos aprimorar a estrutura apresentada na Figura 122, que trata da sobreposição de método, inserindo uma classe abstrata e uma interface. A Figura 132 apresenta o diagrama de classe da nova proposta.

Figura 132 – Classe Interface



Fonte: Autores.

A classe *Pessoa* é uma classe abstrata e serve como modelo para as demais, ou seja, não pode ser instanciada. O código da classe *Pessoa* é apresentado na Figura 133.

Figura 133 – Classe Abstrata *Pessoa*

```
   public abstract class Pessoa{
 8     //Atributos
 9     private String nome;
10     private int idade;
11     private String cpf;
12     private String numTelefone;
13     private Endereco endereco;
14     //Construtor
15 ⊞   public Pessoa(String nome, int idade, String cpf, S
22     //Métodos
23 ⊞   public void mudarNumeroTelefone( String numero ){.
26 ⊞   public String getNome() {...3 linhas }
29 ⊞   public void setNome(String nome) {...3 linhas }
32 ⊞   public int getIdade() {...3 linhas }
35 ⊞   public void setIdade(int idade) {...3 linhas }
38 ⊞   public String getCpf() {...8 linhas }
46 ⊞   public void setCpf(String cpf) {...3 linhas }
49 ⊞   public String getNumTelefone() {...9 linhas }
58 ⊞   public void setNumTelefone(String numTelefone) {..
61 ⊞   public Endereco getEndereco() {...3 linhas }
64 ⊞   public void setEndereco(Endereco endereco) {...3 l
67 }
```

Fonte: Autores.

A classe *Funcionário* herda da classe *Pessoa*. Como esta classe não possui nenhum método abstrato, não há necessidade de reescrever os métodos de *Pessoa*, conforme apresentado na Figura 134.

Figura 134 – Classe *Funcionario*

```
   public class Funcionario extends Pessoa {
15     //Atributos
16     private double salario;
17     private Date dataContratacao = new Date();
18     private boolean ferias = false;
19     //Cosntrutor
20 ⊞   public Funcionario(double salario, String nome, int :
24     //Métodos
25 ⊞   public void setSalario(double salario) {...3 linhas
28 ⊞   public double getSalario() {...3 linhas }
31 ⊞   public Date getDataContratacao() {...3 linhas }
34 ⊞   public void setDataContratacao(Date dataContratacao)
37 ⊞   public String getFerias() {...9 linhas }
46 ⊞   public void setFerias(boolean ferias) {...3 linhas }
55 }
```

Fonte: Autores.

A classe *Gerente* herda de *Funcionario* e implementa a interface *iFuncionario*. A interface *iFuncionario* define um comportamento padrão para todos os funcionários. A Figura 135 apresenta o código da interface *iFuncionario*.

Figura 135 – Interface *iFuncionario*

```
public interface iFuncionarios {
    public abstract boolean tirarFerias();
    public abstract void baterPonto();
    public abstract void trabalhar();
    public abstract double getBonificacao();
12  }
```

Fonte: Autores.

Ao analisar a interface *iFuncionario*, notamos que ela é identificada por meio da palavra reservada *interface* e os métodos, que definem o comportamento, não possuem corpo, ou seja, não são implementados nesse momento.

Para uma classe *Gerente*, que implementa a interface, usamos a palavra reservada *implements*, conforme mostra a Figura 136.

Figura 136 – Classe *Gerente*

```
12  public class Gerente extends Funcionario implements iFuncionarios {
13      //Atributos
14      private String cargo = "Gerente";
15      private int senha;
16      private double bonusSalario = 0.25;
17      private String setor;
18      //Construtores
19      public Gerente() {...3 linhas }
22      public Gerente(String setor, double salario, String nome, int idade
26      //Métodos
27      public void setSenha(int senha) {...3 linhas }
30      public int getSenha() {...3 linhas }
33      public String getSetor() {...3 linhas }
36      @Override
        public boolean tirarFerias() {...3 linhas }
40      @Override
        public void baterPonto() {...3 linhas }
44      @Override
        public void trabalhar() {...3 linhas }
48      @Override
        public double getBonificacao() {...4 linhas }
53      public boolean autentica(int senha) {...9 linhas }
62  }
```

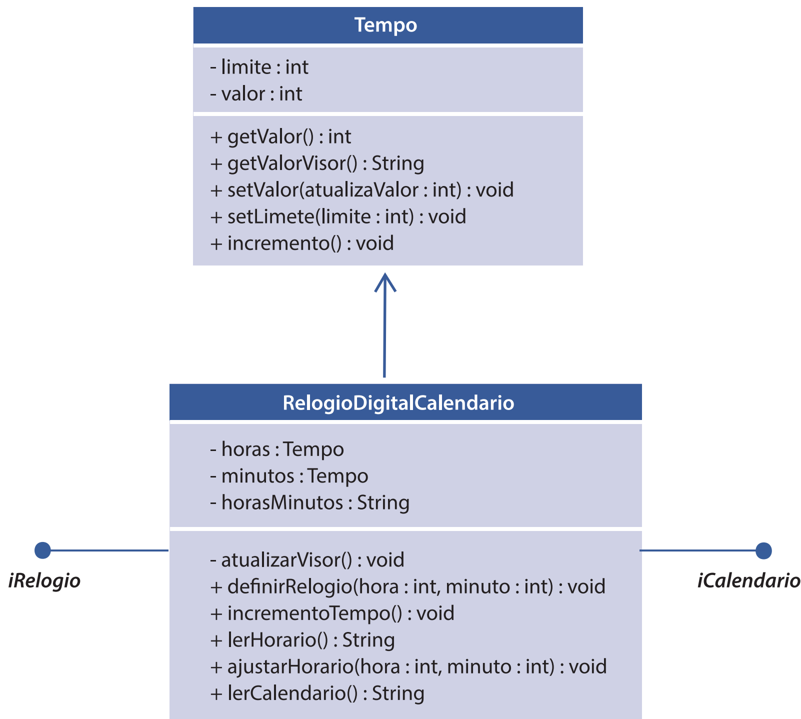
Fonte: Autores.

Cabe ressaltar que todos os métodos, descritos na figura 136, que possuem a palavra reservada *@Override*, são métodos sobrescritos cuja definição está relacionada com a interface *iFuncionario*.

5.6 HERANÇA MÚLTIPLA

Definimos como herança múltipla a situação em que uma subclasse possui mais de uma superclasse. Essa relação normalmente é chamada de relação *"é um"*. Nesse contexto, é importante comentar que a linguagem Java não tem herança múltipla entre classes, como alternativa, as interfaces ajudam nessa questão. Uma classe pode herdar de uma única superclasse, mas pode implementar inúmeras interfaces. As classes que forem implementar uma interface terão de adicionar todos os métodos da interface ou se transformar em uma classe abstrata. Para exemplificar herança múltipla, vamos aprimorar o exemplo apresentado na unidade 3. Nesta unidade, fizemos um protótipo que simula um relógio digital, agora vamos acrescentar o calendário. Veja a representação gráfica do diagrama de classes para esse problema na Figura 137. Observe que a classe *RelogioDigitalCalendario* implementa a interface *iRelogio* e *iCalendario*.

Figura 137 – Implementação de interfaces



Fonte: Autores.

Vamos começar examinando o código das interfaces *iRelogio* e *iCalendario*. Na *iRelogio* foram inseridos 4 métodos abstratos, respeitando a estrutura desenvolvida na unidade 3. Já na *iCalendario* foi inserido apenas um método abstrato chamado *lerCalendario*, cuja função é ler o dia/mês/ano do sistema operacional do computador, os demais comportamentos ficam por conta do sistema. O código é apresentado na Figura 138.

Figura 138 – Código das interfaces *iRelogio* e *iCalendario*

```
11   */
     public interface iRelogio {
         public abstract void definirRelogio(int hora, int minut);
         public abstract  void incrementoTempo();
         public abstract String lerHorario();
         public abstract void ajustarHorario(int hora, int minut);
17   }
```

```
11   */
     public interface iCalendario {
         public abstract String lerCalendario();
14   }
```

Fonte: Autores.

Passaremos a examinar a classe concreta *RelogioDigitalCalendario*, conforme apresentada na Figura 139. Na linha 8 e 9, são importadas as classes que iremos usar para trabalhar com o calendário do sistema. Nas linhas 18 a 21, estão declarados os atributos ou variáveis de referência da classe. Relembrando que todos os métodos que possuem *@Override* acima do cabeçalho são métodos sobrescritos, que possuem origem em uma das duas interfaces. Os métodos *definirRelogio(int hora, int minuto)*, *incrementoTempo()*, *lerHorario()* e *ajustarHorario(int hora, int minuto)* são implementações de métodos abstratos da interface *iRelogio*.

Figura 139 – Código classe concreta *RelogioDigitalCalendario*

```
7
8    import java.util.Calendar;//Formatar a data
9    import java.util.Date;//Fornecer o dia/mes/ano do sistema
10
11   /**...4 linhas */
15   public class RelogioDigitalCalendario implements iRelogio, iCalendario {
16
17       //Atributo
18       private Tempo horas; //Variável de referência
19       private Tempo minutos;//Variável de referência
20       private String horasMinutos;    //Armazena horas e minutos do relógio
21       Date data = new Date();//objeto data armazena Data do computado
22
23       //=======Métodos relacionados ao iRelogio===========
24       /**
25        * Atualiza a informação do horário a ser apresentada
26        */
27       private void atualizarVisor() {
28           horasMinutos = horas.getValorVisor() + ":"
29                   + minutos.getValorVisor();
30       }
31
32       /**
33        * cria um novo relógio ajustando o horário, de acordo com os argumentos
34        * passados
35        */
36       @Override
         public void definirRelogio(int hora, int minuto) {
38           horas = new Tempo();//Instancia objeto horas
39           horas.setLimete(24);
40           minutos = new Tempo();//Instancia objeto minutos
41           minutos.setLimete(60);
42           ajustarHorario(hora, minuto);
43       }
44
```

```
44
45      /**
46       * Este método deve ser chamado uma vez a cada minuto,
47       * ele incrementa e atualiza o mostrador do relógio em
48       * um minuto.
49       */
50      @Override
        public void incrementoTempo() {
52          minutos.incremento();//acesso ao método incremento
53          if (minutos.getValor() == 0) {//acesso ao método getValor
54              horas.incremento();//acesso ao método incremento
55          }
56          atualizarVisor();
57      }
58
59      /**
60       * Retorna o horário no formato HH:MM.
61       */
62      @Override
        public String lerHorario() {
64          horasMinutos = horas.getValorVisor() + ":"
65                  + minutos.getValorVisor();
66          return horasMinutos;
67
68      }
69
70      /**
71       * Ajusta o horário de acordo com as informações da hora e  minuto
72       */
73      @Override
        public void ajustarHorario(int hora, int minuto) {
75          horas.setValor(hora);
76          minutos.setValor(minuto);
77          atualizarVisor();
78      }
79
80      //=======Método relacionados ao iCalendario===========
81      /**
82       * Retorna o data no formato DIA/MÊS/ANO
83       */
84      @Override
        public String lerCalendario() {
86          Calendar cal = Calendar.getInstance();//Manipula o objeto data
87          cal.setTime(data);
88          int dia = cal.get(Calendar.DAY_OF_MONTH);
89          int mes = cal.get(Calendar.MONTH);
90          int ano = cal.get(Calendar.YEAR);
91          return dia + "/" + mes + "/" + ano;
92      }
93  }
```

Fonte: Autores.

O método *lerCalendario()*, nas linhas 85 a 91, é uma implementação do método abstrato da interface *iCalendario*. Inicialmente, na linha 86, foi criado um objeto *cal*, do tipo *Calendar*, cuja função é formatar o objeto *data*, na linha 87. Nas linhas 88 a 90, são formatados o dia, mês e ano e a instrução da linha 91 faz o retorno dessas informações.

A classe *Tempo* permaneceu idêntica à apresentada na unidade 3, conforme apresentado na Figura 140.

Figura 140 – Código classe concreta *Tempo*

```
4   public class Tempo{
5       private int limite;
6       private int valor;
7
8       /**
9        * Retorna o conteúdo da variável valor
10       */
11      public int getValor(){
12          return valor;
13      }
14
15      /**
16       * Retorna o valor formatado em duas casas decimais para ser apresentado
17       * no mostrador do relógio.Se o valor for menor que dez, será adicionadoum
18       * zero para manter as duas casas decimais.
19       */
20      public String getValorVisor(){
21          if(valor < 10) {
22              return "0" + valor;
23          }
24          else {
25              return "" + valor;
26          }
27      }
28
29      /**
30       * Define o valor para ser apresentado no mostrador do relógio, desde que
31       * ele seja maior ou igual a zero e menor que o limite
32       */
33      public void setValor(int atualizaValor){
34          if((atualizaValor >= 0) && (atualizaValor < limite)) {
35              valor = atualizaValor;
36          }
37      }
38
39      /**
40       * Define o valor do limite
41       * */
42      public void setLimete(int limite){
43          this.limite = limite;
44      }
45
46      /**
47       * Incrementa o valor de exibição em um, passando para zero se o
48       * limite for atingido.
49       */
50      public void incremento(){
51          valor = (valor + 1) % limite;
52      }
53  }
```

Fonte: Autores.

A classe *Tempo*, apresentada na figura 140, possui a função de controlar os valores tanto para minutos quanto para horas, a depender da atribuição da variável *limite*, na linha 4. E, quando solicitado, esta classe também faz o incremento da variável *valor*, na linha 51, levando em consideração a variável *limite*.

ATIVIDADES - UNIDADE 5

Todas as atividades abaixo devem ser postadas no Moodle/UAB-UFSM, conforme direcionamento do professor da disciplina.

1) Usando seus conhecimentos de generalização/especialização, analise as três classes da Figura 141 e proponha um novo modelo que elimine as duplicações.

Figura 141 – Classes para o exercício 1

Carro	Avião	Lancha
- modelo : String	- modelo : String	- modelo : String
- numLugares : int	- numLugares : int	- numLugares : int
- numPortas : int	- prefixo : String	- comprimento : int
- comprimento : int	- comprimento : int	- ano : int
- ano : int	- ano : int	- cor : String
- cor : String	- cor : String	- numMotores : int
- placa : String	- numTurbinas : int	

Fonte: Autores.

2) Usando seus conhecimentos de generalização/especialização, analise as três classes da Figura 142 e proponha um novo modelo que elimine as duplicações.

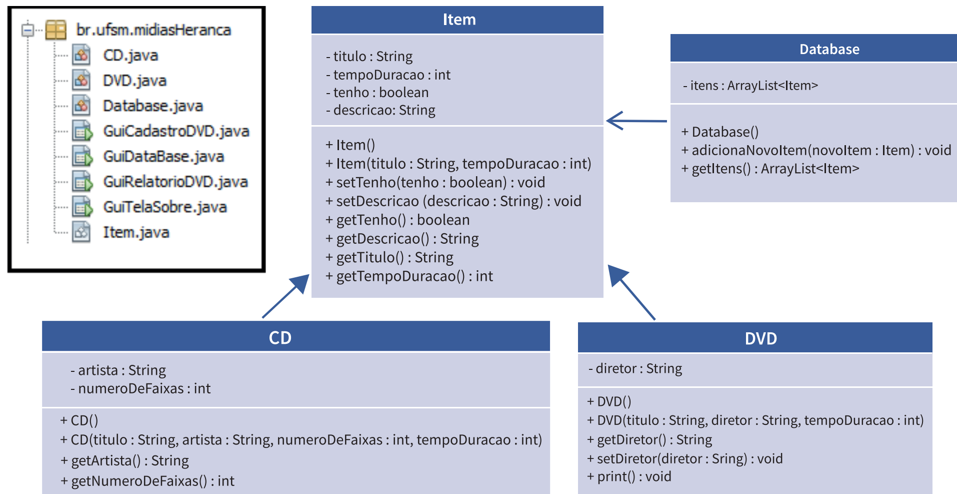
Figura 142 – Classes para o exercício 2

Cachorro	Gato	Coelho
- raca : String	- raca : String	- raca : String
- distanciaFaro : double	- alturaPulo : double	- cenourasPorDia : int
- cor : String	- cor : String	- cor : String
- intensidadeLatido : double	- peloLongo : boolean	- peloLongo : boolean
- preco : double	- preco : double	- preco : double
- nascimento : Date	- nascimento : Date	- nascimento : Date

Fonte: Autores.

3) Implemente o código apresentado na Figura 143, o protótipo que gerencie *CDs* e *DVDs*, usando herança e classe *abstract*. Acrescente os formulários gráficos ao protótipo.

Figura 143 – Diagrama de classes o protótipo gerenciador de multimídias



Fonte: Autores.

Nas Figuras 144, 145, 146 e 147, é apresentado o código fonte das classes *Item*, *Database*, *CD* e *DVD*.

Figura 144 – Classes *Item*

```
abstract class Item {
    private String titulo;
    private int tempoDuracao;
    private boolean tenho;
    private String descricao;
    //Construtor padrão
    public Item(){ }
    //Construtor com parâmetros
    public Item(String titulo, int tempoDuracao){
        this.titulo = titulo;
        this.tempoDuracao = tempoDuracao;
        tenho = false;
        descricao = "<não comentou>";
    }
    public void setTenho(boolean tenho) {
        this.tenho = tenho;
    }
    public void setDescricao(String descricao) {
        this.descricao = descricao;
    }
    public boolean getTenho() {
        return tenho;
    }
    public String getDescricao() {
        return descricao;
    }
    public String getTitulo() {
        return titulo;
    }
    public int getTempoDuracao() {
        return tempoDuracao;
    }
}
```

Fonte: Autores.

Figura 145 – Classe *Database*

```
1   package br.ufsm.midiasHeranca;
2
3   import java.util.ArrayList;
4   /**
5    * @author Fabio Parreira
6    *
7    * A casse Database fornece recursos para armazenar objetos do tipo CD e DVD.
8    * Uma lista de todos os CDs e DVDs pode ser impressa no terminal de sa�da
9    *
10   * Tendo como referencia:
11   * @author Michael Kolling and David J. Barnes
12   * @version 2008.03.30
13   */
14  public class Database{
15      //Atributo
16      private ArrayList<Item> itens;
17
18      /**
19       * Construtor da classe Database. Constroi um banco de dados vazio
20       **/
21      public Database(){
22          itens=new ArrayList<Item>();
23      }
24
25      public void adicionaNovoItem(Item novoItem){
26          itens.add(novoItem);
27      }
28
29      public ArrayList<Item> getItens() {
30          return itens;
31      }
32  }
```

Fonte: Autores.

Figura 146 – Classe *DVD*

```
1   package br.ufsm.midiasHeranca;
2   /**
3    * @author fabio
4    */
5   public class DVD extends Item{
6       private String diretor;
7
8       /**
9        * Construtor para objetos da classe DVD
10       */
11      public DVD(){}
12
13      public DVD(String titulo, String diretor, int tempoDuracao){
14          super(titulo,tempoDuracao);
15          this.diretor = diretor;
16      }
17
18      /**
19       * Retorna diretor para esse DVD.
20       */
21      public String getDiretor() {
22          return diretor;
23      }
24
25      public void setDiretor(String diretor) {
26          this.diretor = diretor;
27      }
28
29      public void print(){
30          System.out.println("    director: " + diretor);
31      }
32  }
```

Fonte: Autores.

Figura 147 – Classe *CD*

```java
package br.ufsm.midiasHeranca;
/**
 * @author fabio
 */
public class CD extends Item{
    private String artista;
    private int numeroDeFaixas;
    /**
     * Construtor para CD.
     */
    public CD(){}
    public CD(String titulo, String artista, int numeroDeFaixas, int tempoDuracao){
        super(titulo,tempoDuracao);
        this.artista = artista;
        this.numeroDeFaixas = numeroDeFaixas;
    }

    public String getArtista() {
        return artista;
    }

    public int getNumeroDeFaixas() {
        return numeroDeFaixas;
    }
}
```

Fonte: Autores.

6

MANIPULAÇÃO DE EXCEÇÕES

INTRODUÇÃO

Até agora, sempre que construímos um protótipo não pensamos na possibilidade de o código fonte ter um erro que possibilite o travamento do sistema operacional ou o encerramento inesperado da aplicação. Neste contexto, durante a codificação de um programa, muitas vezes, temos a necessidade de fazer várias verificações antes de proceder com o real propósito do código, objetivando garantir o bom funcionamento das operações seguintes e impedir que estas possam corromper o funcionamento da aplicação. Por exemplo, temos alguns problemas inesperados que podem surgir quando a JVM (*Java Virtual Machine*) interpreta/executa um programa:

- O programa pode ser incapaz de abrir um arquivo de dados devido a permissões inadequadas;
- O programa não consegue estabelecer uma conexão com o banco de dados porque um usuário forneceu uma senha inválida;
- O usuário pode fornecer dados inadequados, por meio da interface, para um aplicativo. Por exemplo, uma *String*, onde é esperado um valor numérico.

Quando há a detecção de um dos problemas mencionados acima, o programa não tem como continuar o seu funcionamento normal, pois as condições essenciais ao seu funcionamento não estão satisfeitas. Neste sentido, quando existe tal falha, o compilador (ou interpretador) lança uma exceção, que é como se ele estivesse disparando um sinal para avisar que algo deu errado. Logo temos a chance de recuperar esse erro, se código estiver projetado para fazer o tratamento dos erros.

A técnica utilizada para projetar programas que sejam capazes de se recuperar e até mesmo antecipar tais exceções, em tempo de execução, é conhecida como **manipulação de exceções ou tratamento de exceções**. Segundo o dicionário Michaelis (online, 2019), o significado, mais geral, de exceção é:

"Exceção: 1 – Ato ou efeito de excetuar. 2 – Desvio de regra, de lei, de princípio ou de ordem. 3 – A coisa excetuada; aquilo que se desvia da regra."

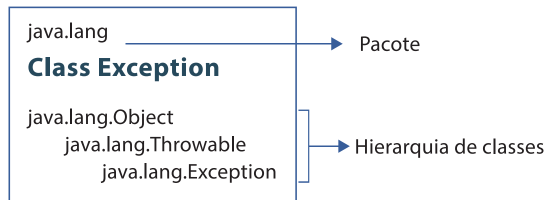
Na área computacional, uma exceção é um evento que acontece durante a execução de um programa corrompendo o curso normal do seu fluxo lógico; logo, é um problema que não ocorre frequentemente. Já a manipulação (ou tratamento) de exceção, pode ser entendida como o mecanismo responsável pelo tratamento da ocorrência de condições que alteram o fluxo normal da execução de programas de computadores, tornando-os robustos e tolerantes a falhas, isto é, eles são capazes de lidar com problemas à medida que eles vão surgindo, em tempo de execução, e continuar funcionando (DEITEL, 2015).

Nesta unidade, vamos aprimorar o protótipo do gerenciador de mídias, adicionando a ele o tratamento de exceções. Para isso, vamos precisar dos conceitos de exceções em Java, lançamento de exceções e os princípios de tratamento de exceção usando *try/catch/finally* e *throws*.

6.1
EXCEÇÕES EM JAVA

Em Java, a classe de exceção genérica, a *"Exception"*, que está inclusa no pacote *java.lang*, é a superclasse de todos as outras subclasses que herdam diretamente ou indiretamente a classe *"Exception"*. A Figura 148 apresenta a hierarquia das classes, conforme disponibilizado pela *Oracle* (ORACLE-EXCEPTION, 2019).

Figura 148 – Classe Exception



Fonte: Autores.

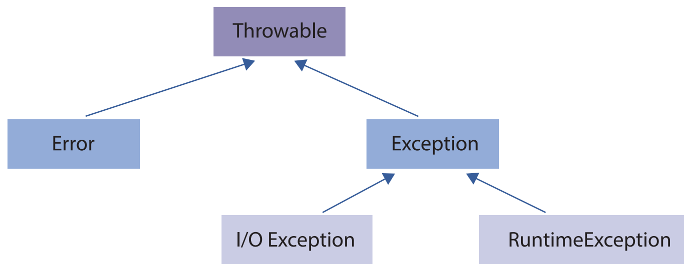
Todo o mecanismo de Java relativo a exceções se baseia no conceito de que as exceções são lançadas e capturadas. Quando acontece uma exceção, ela é lançada de dentro do método onde ocorreu o problema. Quando dizemos que uma exceção é lançada, significa que o fluxo normal do programa foi interrompido e o controle volta para o método chamador. Se este método não capturar a exceção, ela será passada ao método que chamou o método chamador. Isso acontece sucessivamente, até que a exceção seja capturada ou ela chegue até a *JVM*. Ao chegar na *JVM*, a exceção será capturada automaticamente. Neste contexto, dizemos que trabalhar com exceções é simplesmente escolher onde capturar e quais exceções serão capturadas, e na sequência decidir o que fazer após capturar.

Em Java existem três categorias de exceções:

- Erro,
- Falha e
- Exceção de Contingência representadas respectivamente pelas classes: *Error, RuntimeException* e *Exception*.

Vale ressaltar que todas estas classes que lidam com exceção são filhas de *Throwable*, conforme apresentado na Figura 149. É importante salientar que a hierarquia de exceções não tem como objetivo criar implementações ligeiramente diferentes da mesma classe e, sim, diferenciar as exceções por categorias. Para cada tipo de exceção existe uma interpretação feita pelo compilador, que reflete na forma como teremos que lidar com elas no ato da programação.

Figura 149 – Hierarquia dos tipos de exceções em Java



Fonte: Autores.

O tratamento de exceção em Java foi projetado para trabalhar com a classe *Exception*. Nesse sentido, podemos capturar e tratar erros em nosso protótipo, por exemplo, erros de I/O (*IOException*) ou uma divisão por zero (*ArithmeticException*). Descendo na hierarquia de classes, as exceções herdadas da classe *RuntimeException* são exceções geradas normalmente por erros na programação do código fonte, ou seja, erros do programador, tais como o uso de uma referência nula (*NullPointerException*) ou uma divisão por zero (*ArithmeticException*). As exceções geradas na classe Error geralmente não podem ser tratadas pelo nosso protótipo, possuem uma dimensão maior, tal como uma condição de erro na própria máquina virtual Java.

Para que possamos tratar essas exceções, essencialmente das subclasses de *Exception*, vamos usar os tratamentos apropriados, como, por exemplo, a estrutura *try* /*catch* ou como alternativa a inclusão das classes de exceções na assinatura dos métodos.

6.2
LANÇAMENTO DE EXCEÇÕES

Ao lançarmos uma exceção, estamos indicando que o objeto em questão não é capaz de atender um determinado serviço. Após lançar a exceção, obrigatoriamente ela deve ser tratada pelo objeto que a invocou, caso não seja tratada, o protótipo para de funcionar imediatamente. Existem várias exceções em Java, a seguir vamos apresentar algumas:

INTERATIVIDADE: pesquise mais sobre os vários tipos de exceções em java. Para isso, busque informações na documentação: https://docs.oracle.com/javase/7/docs/api/index.html

- *ArithmeticException,*
- *ArrayIndexOutOfBoundsException,*
- *ArrayStoreException,*
- *ClassCastException,*
- *ClassNotFoundException,*
- *CloneNotSupportedException,*
- *EnumConstantNotPresentException,*
- *IllegalAccessException,*
- *IllegalArgumentException,*
- *IllegalMonitorStateException,*
- *IllegalStateException,*
- *IndexOutOfBoundsException,*
- *InstantiationException,*
- *InterruptedException,*
- *NullPointerException,*
- *NumberFormatException,*
- *ReflectiveOperationException,*
- *RuntimeException,*
- *SecurityException,*
- *StringIndexOutOfBoundsException,*
- *UnsupportedOperationException.*

Antes de exemplificar o lançamento de exceções em Java, vamos subdividi-las em duas categorias: as exceções não checadas (*unchecked*) e exceções checadas (c*hecked*). Nas exceções não checadas, a verificação fica por conta do compilador, por isso podemos trabalhar com a instrução *thry/catch* para tratamento das exceções. A seguir, vamos elencar alguns pontos importantes desta categoria:

- Representa defeitos no programa, ou seja, causados pelo programador propriamente dito;

- São subclasses de RuntimeException, para fins de conhecimentos podemos citar ***IllegalArgumentException***, *ArithmeticException*, *NullPointerException* ou *IllegalStateException*;
- O método, necessariamente, não é obrigado a estabelecer verificações para as exceções não checadas, lançadas durante a execução do mesmo.

Analisando a Figura 150, embora exista a possiblidade de disparar uma exceção, a *ArithmeticException*, o compilador Java não exige que o código seja tratado. Ao executar o programa, caso aconteça o erro de divisão por zero, o próprio compilador dispara a exceção e finaliza a execução do objeto.

Figura 150 – Exceções não checadas em Java

```
12   public class ExcecoesNaoChecadas {
13
14       int numero1; //5
15       int numero2;//0
16
17       public int divisao(int num1,int num2) {
18           return numero1 / numero2;//ArithmeticException: / divisão por zero
19       }
20   }
```

```
Saída - cap06-LancamentoExcecao (run)
run:
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at NaoChecadas.ExcecoesNaoChecadas.divisao(ExcecoesNaoChecadas.java:18)
        at NaoChecadas.principal.main(principal.java:21)
C:\Users\fabio\AppData\Local\NetBeans\Cache\8.2\executor-snippets\run.xml:53: Java returned: 1
FALHA NA CONSTRUÇÃO (tempo total: 1 segundo)
```

Fonte: Autores.

Já nas exceções checadas, temos um rigor maior. Além do compilador Java, temos que fazer algumas verificações, por exemplo, com a instrução *thry/catch*, e ainda há momentos que necessitamos da cláusula *throws* para tratamento das exceções. Veja algumas características importantes dessa categoria:

- Representam condições inválidas fora da área de controle do programa, por exemplo, problemas de banco de dados, falhas de rede e arquivos (*IOException*);
- São subclasses de *Exception*, apresentada na Figura 149;
- O método é obrigado a verificar todas as exceções checadas lançadas na sua implementação.

A Figura 151 apresenta um método que trabalha com a abertura de arquivo para escrever informações, que faz parte das exceções checadas. Veja que a linha 30 está toda sublinhada com a linha vermelha, o compilador Java está acusando erro e não irá compilar o programa. Nesse caso, o erro é *IOException*.

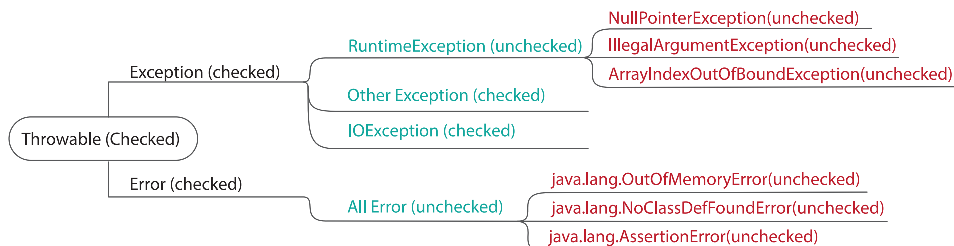
Figura 151 – Exceções checadas em Java

```
21  public class ExcecoesChecadas {
22
23      private ObjectOutputStream outPut = null;
24
25      //Abre o arquivo para gravação de dados
26      public void abreArquivoEscrita() {
27          System.out.println("Abrindo arquivo Funcionario.dat para gravação...");
28
29          //Abre o arquivo
            outPut = new ObjectOutputStream(new FileOutputStream("funcionarioSer.par"));
31
32      }
33  }
```

Fonte: Autores.

É importante ressaltar que o ideal é tratar todas as exceções do programa, sendo elas checadas ou não checadas. Para melhor esclarecimento, a Figura 152 apresenta as exceções não checadas (*unchecked*) e checadas (*checked*).

Figura 152 – Exceções não checadas (*unchecked*) e checadas (checked).



Fonte: Autores.

Dentre as citadas na Figura 152, vamos exemplificar a *IllegalArgumentException*. Esta exceção é lançada para indicar que um método recebeu um argumento ilegal ou inadequado, conforme apresentado na Figura 153. A exceção deve ser lançada toda vez que o usuário esquecer de digitar o nome, ou seja, não podemos cadastrar uma pessoa sem nome. O teste para verificar se o parâmetro nome está vazio é realizado na linha 23. Na linha 25 a exceção é disparada, repare que temos a palavra reservada *new*, que cria um objeto da exceção *IllegalArgumentException*, para depois ser disparada por meio do comando *throw*.

Figura 153 – Exceção *IllegalArgumentException*

```
12  public class Pessoa  {
13      private String nome;
14      private int idade;
15      //Restante dos atributos suprimidos
16
17      //Construtor da classe
18      public Pessoa() {
19      }
20
21      //Métodos
22      public void setNome(String nome) {
23          if("".equals(nome)) {
24              //Lança a exceção
25              throw new IllegalArgumentException();
26          }
27          this.nome = nome;
28      }
29
30      public void setIdade(int idade) {
31          this.idade = idade;
32      }
33      //Restante dos métodos suprimidos
34  }
```

Fonte: Autores.

Uma vez que disparamos a exceção, na linha 25, por meio da palavra reservada *throw*, vamos ter que fazer o tratamento no método chamador, que será apresentado no próximo item.

6.3
PRINCÍPIOS DO TRATAMENTO DE EXCEÇÃO

Os princípios básicos do mecanismo de tratamento de exceção são o bloco *try*, *catch* e *finally*. A seguir, serão detalhados cada um destes comandos. Veja a sintaxe do bloco *try* e das cláusulas *catch* e *finally* abaixo:

try {
Código fonte que podem disparar uma exceção
} catch (ExceçãoA exeA) {
Tratamento da ExceçãoA ou a qualquer uma de suas subclasses, identificada aqui pelo objeto com
referência exeA
} catch (ExceçãoB exeB) {
Tratamento da ExceçãoB ou a qualquer uma de suas subclasses, identificada aqui pelo objeto com
referência exeB
} finally {
Sempre vai ser executado ao final do bloco try/catch, independentemente de ter ocorrido um erro
}

A seguir, vamos detalhar cada um desses blocos, a começar pelo *try*.

Try

O bloco *Try* envolve, dentro de um par de chaves, o código que é suscetível de lançar uma exceção. Isso indica a nossa intenção de tratar quaisquer exceções que podem ser lançadas pela JVM durante a execução do código dentro desse bloco *try*. Voltando ao nosso exemplo da figura 153, vamos construir o tratamento de erro no método chamador. O código para tratamento da exceção é apresentado na Figura 154. Veja que a chamada do método que lança a exceção é realizada dentro do bloco *try*, na linha 172.

Figura 154 – Bloco Try para a exceção *IllegalArgumentException*

```
167    private void jB_CadastrarDVDActionPerformed(java.aw
168        //Cria e instancia o objeto pessoa
169        Pessoa pessoa = new Pessoa();
170
171        try {
172            pessoa.setNome("");//Não digitou o nome
173        }
```

Fonte: Autores.

Para que o código da Figura 154 compile, o bloco *try* deve ser imediatamente seguido de pelo menos um bloco *catch*. A seguir, vamos implementar a cláusula *catch*.

Catch

A cláusula *catch* serve para tratar as exceções com a finalidade de evitar que o programa seja suspenso. Caso a exceção não seja tratada no primeiro *catch*, a execução do código passa para o próximo *catch*, e assim sucessivamente. E se nenhum dos blocos *catch* que implementamos conseguir capturar a exceção? Neste caso, dependendo o tipo da exceção, ela causa a interrupção do programa e lança uma exceção de erro. A Figura 155 apresenta a cláusula *catch* para o tratamento da exceção *IllegalArgumentException*.

Figura 155 – Cláusula *Catch* para a exceção *IllegalArgumentException*

```
167    private void jB_CadastrarDVDActionPerformed(java.awt.event.ActionEvent evt) {
168        //Cria e instancia o objeto pessoa
169        Pessoa pessoa = new Pessoa();
170
171        try {
172            pessoa.setNome("");//Não digitou o nome
173        }catch (IllegalArgumentException erro) {
174           JOptionPane.showMessageDialog(null,"Tratamento de exceção:"+erro,
175                   "Alerta", JOptionPane.INFORMATION_MESSAGE);
176        }
177    }
```

Fonte: Autores.

O tratamento da exceção *IllegalArgumentException* ocorre na linha 173, figura 137, na linha 174 é emitida uma mensagem de alerta com o erro que causou a exceção. Ainda temos o bloco *Finally*, que iremos acrescentar a seguir.

Finally

A cláusula *finally* finaliza a sequência de comandos para o tratamento de exceções. Ela será executada depois do bloco *try* e da(s) cláusula(s) c*atch*. Essa cláusula é opcional, não sendo obrigatória sua inserção na sequência *try/catch*. É usada em ações que sempre precisam ser executadas independentemente da existência de exceções, por exemplo, no fechamento da conexão de um banco de dados ou arquivo. A Figura 156, na linha 176, apresenta a implementação da cláusula *finally* para o tratamento da exceção *IllegalArgumentException*. Na linha 177, é disparada uma mensagem avisando que foi executada a cláusula *finally*.

Figura 156 – Cláusula *Finally*

```
167     private void jB_CadastrarDVDActionPerformed(java.awt.event.ActionEvent evt) {
168         //Cria e instancia o objeto pessoa
169         Pessoa pessoa = new Pessoa();
170 
171         try {
172             pessoa.setNome("");//Não digitou o nome
173         }catch (IllegalArgumentException erro) {
174            JOptionPane.showMessageDialog(null,"Tratamento de exceção:"+erro,
175                    "Alerta", JOptionPane.INFORMATION_MESSAGE);
176         }finally {
177             JOptionPane.showMessageDialog(null,"Execução do bloco Finally!",
178                    "Alerta", JOptionPane.INFORMATION_MESSAGE);
179         }
180     }
```

Fonte: Autores.

Throws

Vimos anteriormente que o comando *throw* serve para disparar uma exceção. Agora vamos estudar o comando *throws*, que, basicamente, repassa a exceção para o método que o chamou. A Figura 157 apresenta a utilização do comando *throws*. Analisando a linha 20, percebemos que *throws* faz parte da declaração do método, ou seja, da sua assinatura. Neste contexto, o comando indica que o método chamador deve capturar e tratar a possível exceção, quando vier a surgir. Na pior das hipóteses, o método chamador repassa para o próximo método o tratamento da exceção.

Figura 157 – Comando *throws*

```
15  public class ExcecoesChecadas {
16 
17      private ObjectOutputStream outPut = null;
18 
19      //Abre o arquivo para gravação de dados
20      public void abreArquivoEscrita() throws IOException {
21          System.out.println("Abrindo arquivo Funcionario.dat para gravação...");
22          outPut = new ObjectOutputStream(new FileOutputStream("funcionarioSer.par"));
23      }
24  }
```

Fonte: Autores.

Agora vamos analisar o método chamador do método *abreArquivoEscrita()*, conforme apresentado na Figura 158. Ao analisar a Figura 158, letra a, na linha 17, percebemos que o comando *throws* faz parte da declaração da assinatura do método *verificaAberturaArquivo*. Logo, a responsabilidade de fazer o tratamento das exceções é repassado para outro método chamador. Já o código apresentado na Figura 158, letra b, faz o tratamento de exceção *IOException*; portanto, não repassa essa responsabilidade à frente.

Figura 158 – Método chamador da classe *ExcecoesChecadas*

a)

```
14    public class AbreArquivo {
15        ExcecoesChecadas arquivo;
16
17        public void verificaAberturaArquivo() throws IOException{
18            arquivo = new ExcecoesChecadas();
19            arquivo.abreArquivoEscrita();
20        }
21    }
```

b)

```
17    public class AbreArquivo {
18        ExcecoesChecadas arquivo;
19
20        public void verificaAberturaArquivo(){
21            try {
22                arquivo = new ExcecoesChecadas();
23                arquivo.abreArquivoEscrita();
24            } catch (IOException ex) {
25                JOptionPane.showMessageDialog(null,"Erro ao abrir o arquivo.",
26                        "Alerta", JOptionPane.INFORMATION_MESSAGE);
27            }
28        }
29    }
```

Fonte: Autores.

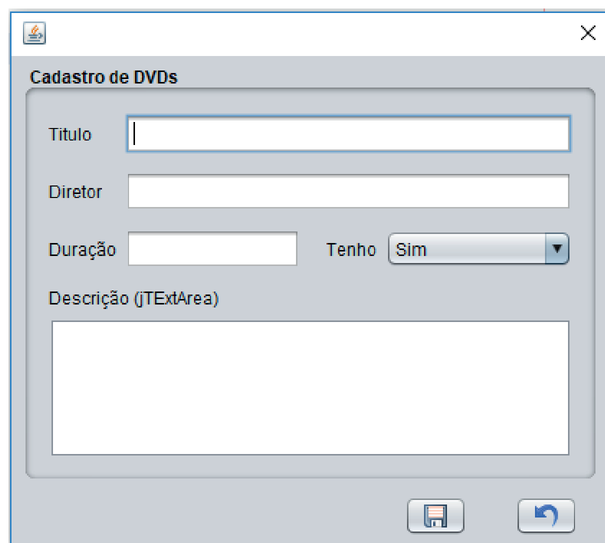
Quando devemos tratar uma exceção? Essa é uma questão polêmica, há muita controvérsia. Existem alguns tipos de exceções que devem ser tratadas obrigatoriamente, outras nem tanto. Por exemplo, as exceções em que o programa tem condições de resolver, geralmente as de acesso a banco de dados ou arquivos, podem ser tratadas por exceções checadas (*checked*), usando *throws* ou *try/catch*. Para exemplificar ainda mais, uma *SQLExceptiono* ou *IOException* devem ser tratadas, pois são erros recuperáveis.

ATIVIDADES - UNIDADE 6

Todas as atividades abaixo devem ser postadas no Moodle/UAB-UFSM, conforme direcionamento do professor da disciplina.

1) Melhore o protótipo de gerenciador de multimídias, construído na unidade 5, acrescentando o tratamento de possíveis exceções que podem vir a surgir no formulário de cadastro de DVDs, apresentado na Figura 159.

Figura 159 – Formulário de cadastro de DVDs



Fonte: Autores.

CONSIDERAÇÕES FINAIS

Neste livro, foram apresentados os conceitos básicos de programação orientada a objetos. Cabe ressaltar que, ao estudar este material, você tem uma visão superficial do assunto; por isso é aconselhável que os estudos sejam aprofundados.

Inicialmente, na Unidade 1, denominada de *Conceitos básicos da linguagem*, são relatadas as características da linguagem escolhida (Java) e da interface gráfica utilizada (*NetBeans*) nos protótipos e, por fim, para fazer uma análise detalhada do código fonte são apresentadas as ferramentas de depuração do código Java. Uma vez conhecendo as ferramentas computacionais a serem utilizadas, estudamos a Unidade 2 – *Princípios da orientação a objetos*. Nela, foram apresentados os princípios da programação orientação a objetos, abordando inicialmente os conceitos de abstração de objetos, conceitos sobre classe, representação gráfica, encapsulamento e escopo de variáveis. Na unidade 3 – *Interações entre objetos*, abordamos a declaração dos métodos, a assinatura, passagem de argumentos e invocação de métodos. Além disso, foi abordada a sobrecarga de métodos e construtores. Seguindo em nosso livro, a Unidade 4, que tratou do *Agrupamento de objetos*, descreveu uma categoria de objeto usada para armazenar e organizar variáveis de referências para outros objetos – dentre as Coleções (*Collections Framework*) foi apresentado o *ArrayList*. Na Unidade 5 – *Herança e outras relações entre objetos*, apresentamos os conceitos do mecanismo de herança. Por fim, na Unidade 6 – *Manipulação de exceção*, fornecemos ferramentas para o tratamento de exceções, minimizando possíveis erros nos programas.

Anteriormente, você cursou a disciplina *"Linguagem de Programação I"*, que tratava de programação estruturada. Este *e-book*, da disciplina de *"Linguagem de Programação II"* apresentou o paradigma orientado a objetos. Este paradigma é muito importante, pois reflete o conceito utilizado na maioria das linguagens de programação adotadas no mercado de trabalho atualmente.

REFERÊNCIAS

AURÉLIO. **Dicionário Aurélio on-line**. Disponível em: <https://dicionariodoaurelio.com/>. Acesso em: 10 fev. 2019.

BARNES, D.; KOLLING, M. **Programação Orientada a objetos com Java**. 4. ed. São Paulo: Prentice Hall, 2008.

BARKER, J. **Beginning Java Objects-From Concepts to Code**. 2 ed. USA: Apress, 2005.

CARDELLI, L.; WEGNER, P. On Understanding Types, Data Abstraction, and Polymorphism. **Computing Surveys**, v. 17, n. 4, p. 471-522, 1985.

COLLECTIONS. **Introduction to Collections**. Disponível em: <https://docs.oracle.com/javase/tutorial/collections/intro/index.html>. Acesso em: 10 fev. 2019.

DEITEL, H.M.; DEITEL H.M. **Java Como Programar**. 8. ed. São Paulo: Pearson Prentice Hall, 2015.

MICHAELIS. **Dicionário Michaelis online**. Disponível em: <https://michaelis.uol.com.br/moderno-portugues/busca/portugues-brasileiro/>. Acesso em: 05 dez. 2018.

ORACLE-EXCEPTION. **Class Exception.** Disponível em: <http://docs.oracle.com/javase/8/docs/api/java/lang/Exception.html>. Acesso em: 14 out. 2015.

SIERRA, K.; BATES, B. **Use a cabeça –** Java. 2. ed. Rio de Janeiro: Alta Books, 2007.

TIOBE. **Índice da comunidade de Programação TIOBE.** Disponível em: <https://www.tiobe.com/tiobe-index/>. Acesso em: 07 nov. 2018.

APRESENTAÇÃO DOS PROFESSORES RESPONSÁVEIS PELA ORGANIZAÇÃO DO MATERIAL DIDÁTICO

A definição das unidades deste material foi idealizada a partir das experiências, em sala de aula, dos docentes do Departamento de Tecnologia da Informação da UFSM (Universidade Federal de Santa Maria)/Campus Frederico Westphalen. Segue um breve resumo do currículo dos autores:

Fábio Parreira: Possui Graduação em Ciência da Computação pela UNITRI. Mestrado em Processamento Digital de Imagens pela UFU (Universidade Federal de Uberlândia). Doutorado em Inteligência Artificial e Informática de Sinais Biomédico pela UFU. Pós-doutorando em Engenharia do conhecimento pela UFSC. Atua na área de Ciência da Computação com enfoque em pesquisas interdisciplinares em Inteligência Computacional, Informática aplicada à saúde e Tecnologia Educacional. Desenvolve pesquisa em Acessibilidade Digital, Jogo Educacional Digital e Inteligência Artificial aplicada.

Teresinha Letícia da Silva: Possui graduação em Informática pela Universidade Regional Integrada do Alto Uruguai e das Missões, Especialização em Ciência da Computação pela Universidade Federal de Santa Catarina e Mestrado em Ciência da Computação pela Universidade Federal de Santa Catarina. Atualmente é Professora Assistente em Regime de Dedicação Exclusiva da Universidade Federal de Santa Maria no campus de Frederico Westphalen. Tem experiência na área de Ciência da Computação, com ênfase em Software Básico, atuando principalmente nos seguintes temas: internet, educação a distância, computação gráfica, realidade virtual e aumentada e recuperação de informações.

Cristiano Bertolini: Possui graduação em Ciência da Computação pela UPF (Universidade de Passo Fundo), mestrado em Ciência da Computação pela PUCRS (Pontifícia Universidade Católica do Rio Grande do Sul), doutorado em Ciência da Computação pela UFPE (Universidade Federal de Pernambuco) e pós-doutorado pela United Nations University (Macau) na área de Métodos Formais e Informática Aplicada à Saúde. Suas áreas de interesse envolvem, principalmente, Métodos Formais, Teste de Software e Engenharia de Software.

Guilherme Bernardino da Cunha: Possui graduação em Ciência da Computação, mestrado em Ciências com Ênfase em Inteligência Artificial e Processamento Digital de Imagens e Doutorado em Ciências com ênfase em Engenharia Biomédica pela Universidade Federal de Uberlândia. Atualmente é professor Adjunto da UFSM – Universidade Federal de Santa Maria – Campus Frederico Westphalen.

Tem experiência na área de Ciência da Computação atuando principalmente nos seguintes temas: Engenharia de Software, Inteligência Artificial, Análise de Séries Temporais, Epidemiologia, Banco de Dados, Redes Neurais Artificiais, Algoritmos Genéticos, Bioinformática.