Treinamento IBTA

Formação Java

Programação Orientação a Objetos com Java

Módulo II

IJ 150

Versão 1.0 Julho 2005

Índice

Iniciando 1
Objetivos 1
Habilidades Adquiridas 1

1. Introdução ao Java 2
1.1. Distribuições da Tecnologia Java 2
1.2. Tipos de Programas Java 4
1.3. "Key Features" da Tecnologia Java 5
1.4. Conceitos errados sobre o Java 6
1.5. Lista das palavras reservadas da linguagem Java 6
1.6. Entendo a portabilidade do Java 7
1.7. Por trás da Java Runtime Enviromnent 8
1.8. Definindo nomes (identificadores) para métodos, classes e variáveis 9
1.9. Anatomia de uma classe Java 9
1.10. Entendendo uma aplicação Java Stand-alone 12
Discussão 13
Exercícios 14

2. Linguagem de programação Java 15
2.1. Os tipos primitivos da linguagem Java 16
2.2. Usando os tipos primitivos 17
2.3. Usando variáveis primitivas como atributos de objetos 18
2.4. "Casting" e "Promotion" de tipos primitivos 19
2.5. O tipo referência na linguagem Java 20
2.6. A notação do .(ponto) e o uso de uma referência 20
2.7. Endereçamento de memória da JVM para um objeto 21
2.8. Comentando o código 24
2.9. Visualizando graficamente a memória 25
2.10. Operadores e Expressões 26
2.11. Operadores Aritméticos 26
2.12. Operadores Lógicos 27
2.13. Operadores condicionais 29
2.14. Precedência de operadores 31
2.15. Estruturas de repetição 32
2.16. Usando o for 32
2.17. Usando o While 34
2.18. Usando o Do...While 35
2.19. Laços aninhados 36
2.20. Estruturas de seleção 37
2.21. Usando o if 39
2.22. Usando o switch 41
Discussão 43
Exercícios 44

3. Criando classes em Java a partir de um modelo orientado a objetos 45
3.1. Encapsulamento 47
3.2. Definindo variáveis 50
3.3. Entendendo a diferença entre "de classe" e "de instância" 51
3.4. Atributos e métodos de instância (objeto) 52

3.5. Definindo métodos 54
3.6. Definindo construtores 58
3.7. Definindo constantes 62
3.8. Usando a referência This 64
3.9. Atributos e métodos de classe (static) 67
Discussão 69
Exercícios 70

4. Classes e Objetos em Java 71
4.1. Alguns ambientes de programação da linguagem Java 71
4.2. Tradução de uma classe em UML para o esqueleto em Java 72
4.3. Construtor Padrão 73
4.4. Métodos Acessores, Modificadores e Construtores 74
4.4.1. Implementação em Java dos Métodos Acessores, Modificadores e Construtores 75
4.5. Primeiro programa orientado a objetos em Java 77
Exercícios 80
Bibliografia 80
Leitura complementar: As 52 palavras reservadas do Java 81

5. Diagramas de interação 83
5.1. Modelo de interação 83
5.1.1. Diagrama de colaboração 84
5.1.2. Diagrama de seqüência 84
5.1.3. Quando um objeto pode enviar uma mensagem para outro? 86
5.2. Criando o modelo de interação 88
5.3. Passando para o Java 101
Exercícios 104
Bibliografia 104

Iniciando

Objetivos

Este módulo fornecerá aos iniciantes uma excelente opção para o aprendizado de programação utilizando a linguagem Java.

Os principais tópicos são: o significado da linguagem de programação Java, os princípios da orientação a objetos e a aplicação desses conceitos para escrever códigos com a tecnologia Java abrangendo as funções essenciais da programação.

No final do módulo, os alunos serão capazes de escrever um aplicativo em Java bastante simples, mas não terão grandes habilidades para programação fornecendo uma base sólida da linguagem de programação Java com a qual os alunos podem continuar trabalhando e treinando.

Habilidades Adquiridas

Após a conclusão deste curso, os alunos deverão estar aptos a:

- Descrever o histórico e as características significativas da linguagem de programação Java
- Analisar um projeto de programação utilizando a análise e o projeto orientados a objetos e fornecer um conjunto de classes, atributos e operações.
- Entender um programa Java, lendo o seu código fonte.
- Criar e designar valores para variáveis primitivas e de referência e utilizá-las em programas
- Descrever como as variáveis primitivas e de referência são armazenadas na memória
- Escrever um programa básico contendo o método main, as variáveis de referência e primitivas e uma declaração de classe.
- Determinar quando é necessário alterar os tipos de dados de variáveis (casting) e escrever o código para fazer isso
- Escrever programas que implementem construções de decisão, tais como if/else
- Escrever programas que implementem construções de repetição, tais como while, for ou do.
- Escrever programas usando vários métodos que façam a chamada de métodos passem argumentos e recebam valores de retorno

1. Introdução ao Java

Sabemos que a Orientação a Objetos é uma nova maneira, mais natural e simples, de pensar, elaborar, documentar e construir aplicações e sistemas de informação.

Falamos de ***abstração, objetos, classes, atributos (propriedades) e métodos (funções)***.

Vimos também que a tecnologia ***Java*** permite implementar todos esses conceitos, pois a linguagem é totalmente aderente as técnicas da orientação a objetos.

A partir deste capítulo veremos como desenvolver e implementar, ou seja, programar aplicações orientadas á objetos usando a linguagem ***Java*** e suas tecnologias.

1.1. Distribuições da Tecnologia Java

Antes de vermos como programar em Java devemos entender quais tipos de programas podemos escrever e executar usando a Tecnologia Java.

Atualmente a JavaSoft, empresa do grupo Sun Microsystems que mantém os investimentos e controla as compatibilidade e evolução das versões de Java, distribui a tecnologia para três ambientes de execução distintos, capazes suportar a execução de programas Java.

J2SE – Java2 Standard Edition

- Fornece as principais API's, foca o desenvolvimento de aplicações Cliente/Servidor, pouco distribuído e sem Internet;
- http://java.sun.com/j2se/
- Fornece a Java Runtime Enviromnent (JRE) ou JVM (Java Virtual Machine).

J2EE – Java2 Enterprise Edition

- Fornece um conjunto de API's para o desenvolvimento corporativo, foca a integração entre sistemas, permite alta distribuição de objetos, total suporte para as tecnologias Internet, depende da J2SE; Depende da JRE.
- http://java.sun.com/j2ee/

J2ME – Java2 Micro Edition

- Fornece as API's necessárias para o desenvolvimento de aplicações para computação móvel, em pequenos dispositivos ou para tecnologias embarcadas;
- http://java.sun.com/j2me/
- Fornece uma Java Runtime Enviromnent (JRE) de capacidade reduzida;

Neste módulo abordaremos o ambiente de desenvolvimento e execução disponíveis na J2SE versão 1.4.2.

Veja na figura abaixo a organização das API's e plataforma da J2SE 1.4

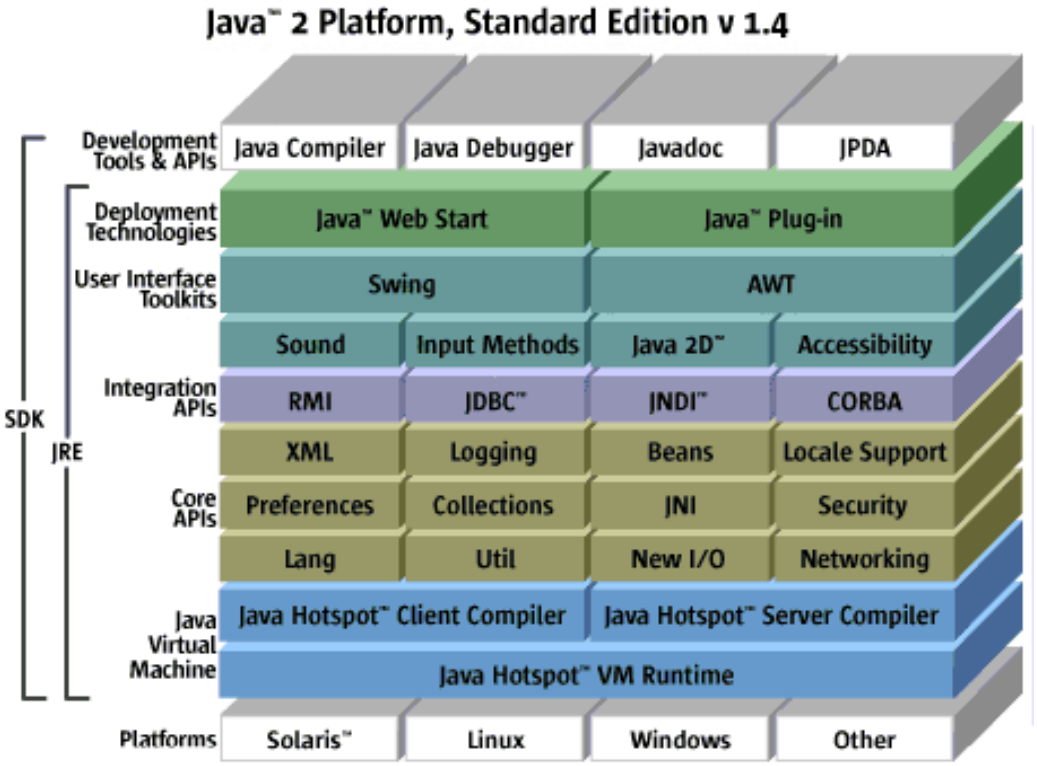


Figura 1

JVM – Java Virtual Machine – Um software que emula uma CPU e Memória para a execução de programas Java.

JRE – Java Runtime Enviromnent – Ambiente obrigatório para a execução de programas Java. A JRE é composta da JVM somada ao conjunto de API's da J2SE. (JVM + API's = JRE)

SDK – Software Development Kit – Conjunto de ferramentas para a compilação, documentação e debug de aplicativos Java. A SDK é composta da JRE somada as ferramentas de desenvolvimento.

Java HotSpot – presente na JRE, tem a finalidade de pré-compilar porções de código, fazendo com que os programas executem mais rápido.

Para se executar qualquer aplicativo Java, é necessário ter uma JRE, que contém a JVM mais as API's da J2SE.

1.2. Tipos de programas Java

Todos os tipos de programas Java descritos abaixo, são escritos usando uma única linguagem, suas diferenças estão nas API's que eles usam.

Stand-Alone: aplicação baseada na J2SE, que tem total acesso aos recursos do sistema, memória, disco, rede, dispositivos, etc; Um servidor pode executar uma aplicação Stand-Alone, por exemplo, um WebServer. Uma estação de trabalho pode executar uma aplicação de Automação Comercial.

Java Applets™: pequenas aplicações, que não tem acesso aos recursos de hardware e depende de um navegador que suporte a J2SE para serem executados, geralmente usados para jogos, animações, teclados virtuais, etc.

Java Servlets™: programas escritos e preparados para serem executados dentro de servidores web baseados em J2EE, geralmente usados para gerar conteúdo dinâmico de web sites;

Java Midlets™: pequenas aplicações, extremamente seguras, e construídas para serem executadas dentro da J2ME, geralmente, celulares, Palm Tops, controladores eletrônicos, computadores de bordo, smart cards, tecnologia embarcada em veículos, etc;

JavaBeans™: pequenos programas, que seguem um padrão bem rígido de codificação, e que tem o propósito de serem reaproveitados em qualquer tipo de programa Java, sendo reaproveitados, e podendo ser chamados a partir de: stand-alone, applets, servlets e midlets.

1.3. “Key Features” da Tecnologia Java

Simples: contém somente 48 palavras reservadas ao todo;

Orientada a Objetos: permite a programação orientada a objetos, usando todos os seus conceitos;

Distribuída: permite a distribuição da aplicação em várias JREs (desktop final e/ou servidor) usando uma rede;

Robusta: usa tecnologia de linguagens compiladas e interpretadas, sendo mais confiável;

Segura: a segurança foi uma preocupação constante durante o desenvolvimento e evolução da tecnologia da JVM;

Portável: uma vez escrita e compilada roda em qualquer plataforma que tenha uma JRE, ***“Write Once, Run Anywhere”*** ™ ;

Multi-Thread: permite o desenvolvimento de aplicações para servidores que atendem muitos usuários ao mesmo tempo, ou de aplicações que necessitem executar mais de uma tarefa ao mesmo tempo;

1.4. Conceitos errados sobre o Java:

É uma extensão do HTML: falso, o Java é uma linguagem completa derivada do SmallTalk e do C++;

É apenas outra linguagem: falso, o Java possui uma única linguagem, para se construir componentes para todos os ambientes;

Todos os programas Java rodam em páginas Web: falso, existem três ambientes distintos (J2SE, J2EE e J2ME) de execução de programas e aplicativos Java;

O JavaScript é uma versão light do Java: falso, a Netscape aproveitou a onda de marketing e batizou sua tecnologia, LiveScript, de JavaScript;

É interpretado, muito lento para aplicações sérias: o Java é interpretado sim, entretanto, a forma como a dupla compilador/interpretador tratam os programas garante uma performance muitas vezes equivalente ao do C++, com a facilidade de uma linguagem bem mais simples que o C++;

É difícil programar em Java: falso, a maior dificuldade está em assimilar os conceitos da Orientação a Objetos. A linguagem Java é muito simples;

1.5. Lista das palavras reservadas da linguagem Java:

O intuito não é explicar cada uma delas separadamente, e sim somente listá-las para podermos entender sua simplicidade.

Nenhuma das palavras abaixo podem ser usadas como nome de classes, objetos, atributos ou métodos.

abstract	double	int	static
boolean	else	interface	super
break	extends	long	switch
byte	final	native	synchronized
case	finally	new	this
catch	float	null	throw
char	for	package	throws
class	goto*	private	transient
const*	if	protected	try
continue	implements	public	void
default	import	return	volatile
do	instanceof	short	while

*** reservadas, entretanto não são usadas, pois não representam instruções da JRE;**

1.6. Entendendo a portabilidade do Java.

Como comentado anteriormente, o Java usa as duas formas computacionais de execução de software, compilação e interpretação.

O uso das duas formas faz com que o processo de desenvolvimento do software se torne mais acelerado, pois linguagens somente compiladas, necessitam de instruções adicionais da plataforma operacional para serem executados, e as linguagens somente interpretadas geralmente são muito lentas. Por isso o Java num primeiro momento é compilado (transformar código fonte) em instruções da Java Virtual Machine (bytecode).

Como o processo de execução é divido em dois, temos a possibilidade de que o interpretador seja escrito para várias plataformas operacionais.

Ou seja, o mesmo bytecode Java não importando onde foi compilado, pode ser executado em qualquer JRE. Os fabricantes de hardware e sistemas operacionais, em conjunto com a JavaSoft, desenvolveram várias JREs para vários ambientes operacionais. Por isso, aplicações 100% Java são 100% portáveis.

Compilando...

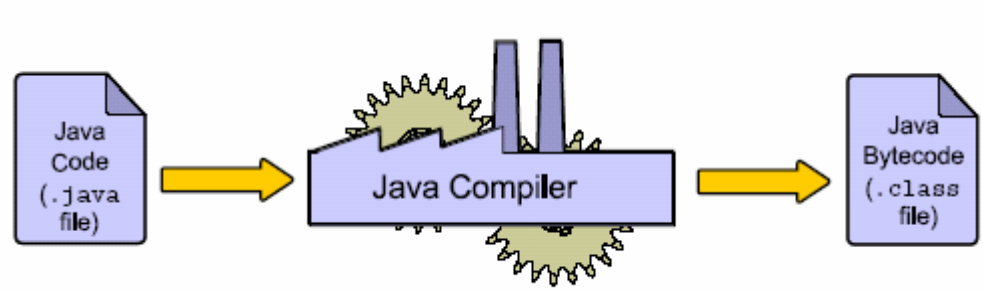


Figura 2

Executando...

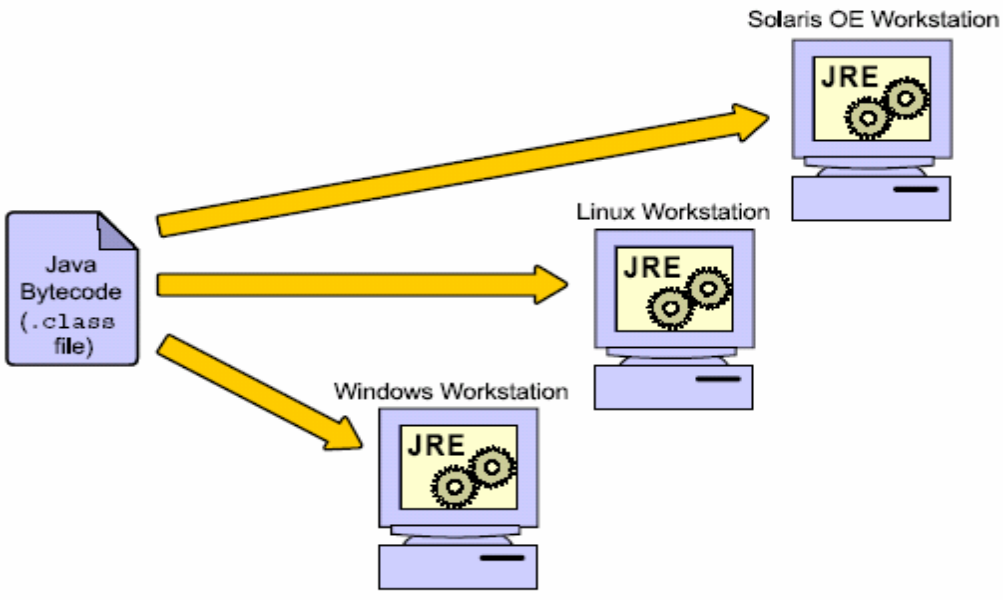


Figura 3

1.7. Por trás da Java Runtime Enviromnent.

A JRE é um pacote de software, que é executado como um aplicativo do sistema operacional e que interpreta a execução de programas Java de forma escalável, performática e segura. Por isso é chamada de máquina virtual.

Veremos agora os principais componentes internos da JRE.

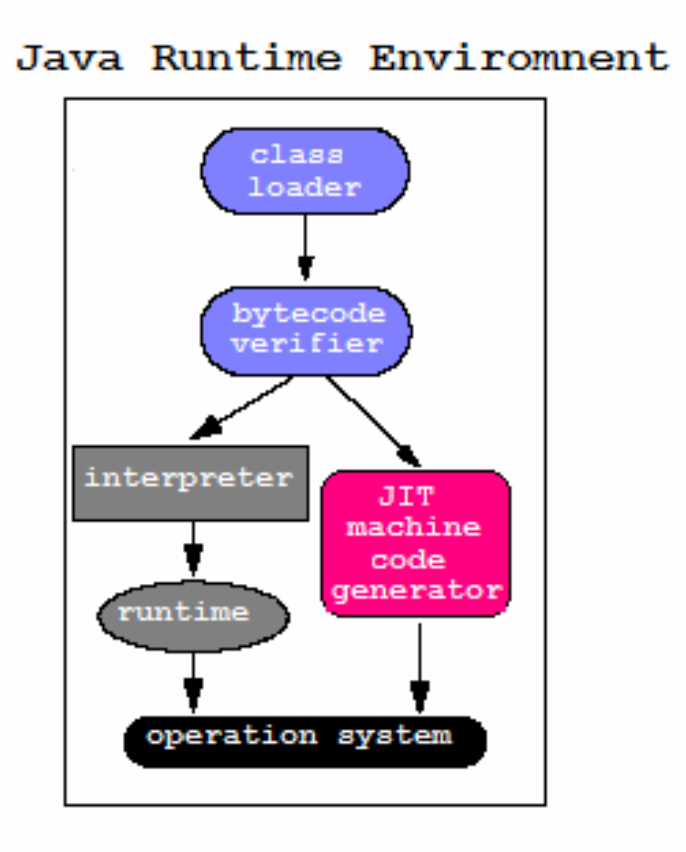


Figura 4

Classloader:
Responsável por carregar as classes Java (bytecode) que farão parte da execução dos aplicativos de um sistema.

Bytecode verifier:
Verifica se as classes carregadas são mesmo de fato classes Java, e validam se o código é seguro e não viola as especificações.

Interpreter e Runtime:
Interpretam uma porção do bytecode, transformando-o em chamadas do sistema operacional.

JIT (Just-In-Time Compiler):
Adquirido da Symatec em 1997, ele é responsável por compilar uma parte do bytecode Java diretamente em chamadas do sistema operacional.

Integrado a JRE, existe ainda um componente chamado **Garbage Collector**, que é responsável por coletar da memória objetos e dados que estão sendo mais usados.

O **Garbage Collector**, tem seu próprio algoritmo e ciclo de execução, por isso não temos controle sobre ele. Assim, em Java, o programador não precisa se preocupar em alocar e desalocar memória quando lida com objetos. No C ou C++, é de responsabilidade do programador o gerenciamento de memória.

Em Java, quando não precisamos mais de um objeto, basta anular sua referência elegendo-o a ser coletado, e o Garbage Collector fará o resto, garantindo a saúde da memória em que a aplicação está sendo executada.

Para cada conjunto, sistema operacional e hardware, existe uma JRE otimizada, capaz de garantir a execução de sistemas baseados em Java.

1.8. Definindo nomes (identificadores) para métodos, classes e variáveis:

Nomes de classes, variáveis e métodos devem ter identificadores válidos da linguagem Java. Esses identificadores são seqüências de caracteres. As regras para a definição de identificadores são:

- Um nome pode ser composto por letras (minúsculas e/ou maiúsculas), dígitos e os símbolos _ e **$**.
- Um nome não pode ser iniciado por um dígito (0 a 9).
- Letras maiúsculas são diferenciadas de letras minúsculas.
- Uma palavra-chave da linguagem Java não pode ser um identificador.

Além dessas regras obrigatórias, o uso da convenção padrão para identificadores Java torna a codificação mais uniforme e pode facilitar o entendimento de um código.

1.9. Anatomia de uma classe Java.

Todo programa Java será classificado em um dos tipos descritos anteriormente neste capítulo. O objetivo deste livro é mostra as construção de aplicações stand-alone e JavaBeans™.

Todas as aplicações Java, independente do seu tipo, obrigatoriamente deve estar definida dentro de uma classe Java. E os objetos do nosso sistema, são representados em classes.

A representação de classes em diagramas UML contempla três tipos básicos de informação: o nome da classe, os seus atributos e os seus métodos.

Graficamente, um retângulo com três compartimentos internos representa esses grupos de informação:

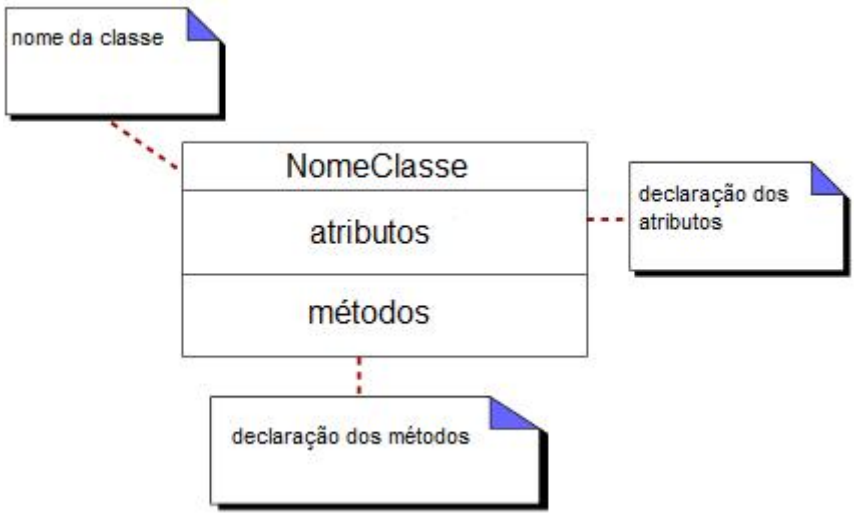


Figura 5

Nome da classe
Um identificador para a classe, que permite referenciá-la posteriormente.

Definindo uma classe
Uma classe define-se como abaixo e ela é composta por membros, atributos e/ou métodos.

```
[modificador] class [identificador] {

     // conjunto de atributos

     // conjunto de métodos
}
```

Examinando o código abaixo podemos identificar os conceitos citados acima:

```
public class PrimeiroProgramaJava {

     public static void main(String args[]) {
          System.out.println("Bem vindo a Universidade Java!");
     }
}
```

Identificamos o seguinte:

`public, static` – modificadores;

`PrimeiroProgramaJava, main` – identificadores;

`public static void main(String args[]) { }` – método;

`String args[]` – argumento do método;

`String` – tipo de dados;

`void` – tipo de retorno do método;

class – tipo para classes;

`System.out.println` – chamada de um método de uma outra classe;

Definindo Atributos
O conjunto de propriedades da classe. Para cada propriedade, especifica-se:

```
[modificadores] tipo [nome] [= valor_inicial];
```

`nome`: um identificador para o atributo.
`tipo`: o tipo do atributo (inteiro, real, caráter, outra classe, etc.)
`modificadores`: opcionalmente, pode-se especificar o quão acessível é um atributo de um objeto a partir de outros objetos. Valores possíveis são:

- (privativo), nenhuma visibilidade externa;
+ (público), visibilidade externa total;
(protegido), visibilidade externa limitada.

Definindo Métodos
O conjunto de funcionalidades da classe. Para cada método, especifica-se sua **assinatura**, composta por:

```
[modificadores] tipo_de_retorno [nome] ([argumentos]) {
   // corpo do método;
}
```

`nome`: um identificador para o método.
`tipo_de_retorno`: quando o método tem um retorno, o tipo desse valor.
`lista de argumentos:` quando o método recebe parâmetros para sua execução, o tipo e um identificador para cada parâmetro.
`modificadores:` como para atributos, define o quão visível é um método a partir de objetos de outros classes.

1.10. Entendendo uma aplicação Java Stand-Alone.

Para que uma aplicação Java possa ser considerada como Stand-Alone e para que ela possa ser executada diretamente pelo interpretador, como vimos anteriormente, ela deve possuir esse método, da forma como está escrito.

```
public static void main(String args[])
```

É a partir dessa linha de código, que o programa Java começa a ser executado. O interpretador Java carrega a classe e chama o método **`main(String args[])`**, ele tem o papel de ser o start-point (iniciar execução) de uma aplicação Stand-Alone, e quando a execução chegar ao fim do método **`main`**, a aplicação é encerrada.

Discussão

Em quais campos se aplica melhor cada distribuição da Tecnologia Java?

Qual a principal diferença em JRE e JDK?

Sabendo que uma classe Java pode facilmente ser modelada, então é fácil analisá-la e codificá-la?

Exercícios

2. Linguagem de Programação Java

Uma linguagem de programação permite que façamos o mapeamento de uma determinada rotina ou processo existente no mundo real ou imaginário, para um programa de computador.

Para que o computador possa executar tal rotina, devemos usar uma linguagem que ele entenda. No caso de Java, a JRE entende bytecode, que é proveniente da compilação de programas java.

Devemos ter a seguinte percepção, um sistema construído usando a linguagem Java, será composto de muitas classes, cada uma com sua responsabilidade dentro do sistema, e deve ser escrito de acordo como a forma que ele será executado.

Um programa de computador deve refletir a execução de rotinas de cálculo, comparação e expressões matemáticas. Para entender quais tipos de expressões, rotinas e comparações que podemos executar, devemos primeiramente entender os tipos de dados que o Java possui para armazenar valores.

Dentro das linguagens de programação orientadas a objeto, a capacidade de se criar um elemento que representa um determinado tipo de dado, seja um atributo de um objeto ou um parâmetro de um método, devemos fazer uso de **variáveis**.

Na linguagem java, temos duas categorias de tipos de **variáveis**:

Tipos primitivos: capazes de armazenar valores símples, como números inteiros e ponto flutuante, caractere e booleano, e usamo-los **para executar cálculos e comparações**;

Tipos referência: capazes de armazenar as referencias dos objetos que vamos usar, ou seja, todo tipo de dado que não é primitivo, e usamo-los **para navegar entre os objetos de um sistema**;

2.1. Os tipos primitivos da linguagem Java.

Os tipos primitivos da linguagem Java foram inseridos para facilitar a codificação dos programas e otimizar o mecanismos de cálculo do interpretador.

Inteiros	Precisão	Range de valores
byte	8 bits	-2^7 até $2^7 - 1$
short	16 bits	-2^{15} até $2^{15} - 1$
char	16 bits	0 até $2^{15} - 1$
int *	32 bits	-2^{32} até $2^{32} - 1$
long	64 bits	-2^{64} até $2^{64} - 1$
boolean	8 bits	true ou false

Ponto Flutuante	Precisão	Range de valores
float	32 bits	Dependentes do Sistema Operacional
double *	64 bits	

*** otimizados para a execução de cálculos dentro da JRE.**

Usamos cada um desses tipos, de acordo com a necessidade:

byte: geralmente usado para valores reais muito pequenos, e manipulação de dados em arquivos;

short: também usado para valores pequenos;

char: usado para trabalhar com caracteres;

int: tipo de dados preferido para manipulação de valores inteiros;

long: usado quando o range de valor do int não é o suficiente;

float: usado como número real de precisão símples;

double: tipo numérico preferido para manipulação de valores de precisão dupla;

boolean: tipo booleano pra armazenar valores de verdadeiro (true) ou falso (false);

2.2. Usando os tipos primitivos.

Variáveis do tipo primitivo podem ser usadas como atributo de objeto, parâmetro de método, ou variável local de método.

Uma variável na sua definição clássica serve para armazenar um tipo de valor dentro de um algoritmo. E este valor pode variar dentro do programa, por isso ele se chama variável.

```
public class TesteVariavel {

	public static void main(String args[]) {

	byte b = 10;
	System.out.println("Um byte: "+ b);
	char c = 64;
	System.out.println("O char: "+ c);
	short s = 1067;
	System.out.println("Um short: "+ s);
	int i = 10 * ( b + c + s );
	System.out.println("Um int: "+ i);
	long l = i * i;
	System.out.println("Um long: "+ l);
	float f = l * 5 / 2;
	System.out.println("O float: "+ f);
	double d = l * l;
	System.out.println("O double: "+ d);
	}
}
```

Outro uso comum de variáveis, é para identificarmos ao que ela se refere.

```
public class CalculoJurosSimples {

	public static void main(String args[]) {

		int prestacoes = 6;
		double jurosMensal = 0.02;
		double valorEmprestado = 1000.0;
		double jurosPeriodo = prestacoes * jurosMensal;
		double valorTotal = valorEmprestado * ( 1 + jurosPeriodo );
		double valorJuros = valorTotal - valorEmprestado;
		double valorParcela = valorTotal / prestacoes;

		System.out.println("Valor total a pagar: "+ valorTotal);
		System.out.println("Valor dos juros: "+ valorJuros);
		System.out.println("Valor da parcela: "+ valorParcela);
	}
}
```

2.3. Usando variáveis primitivas como atributos de objetos.

Toda vez que definimos uma variável, fora de um método, ela se torna uma variável de objeto, ou de instância.

Usamos esse tipo de variável para armazenar valores dentro dos objetos, usando todo o poder da orientação a objetos.

Tomando como exemplo, se fossemos analisar um determinado conjunto de objetos dentro do sistema do restaurante, podemos identificar o objeto Conta. O que uma conta tem como atributos (características)?

Retomando os conceitos de classe, vimos que uma classe serve para criarmos objetos de uma determinada categoria ou tipo.

Poderíamos identificar para a conta atributos básicos como: número e saldo. Assim, poderíamos criar uma classe, que me permitisse criar vários contas que tivessem esses atributos.

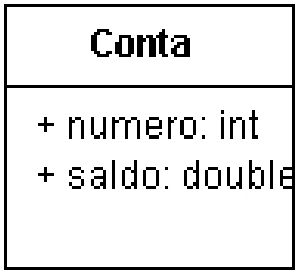


Figura 6

Vemos aqui, que podemos traduzir facilmente o diagrama UML para um classe Java.

```
public class Conta {

      public int numero;
      public double saldo;
}
```

*** o sinal de (+) na UML significa modificador public para atributos ou métodos.**

Com essa classe Conta, poderíamos manipular qualquer Conta que pudesse conter um número e um saldo.

Antes de manipularmos os atributos de um objeto, vamos aprender a manipular os objetos.

Na linguagem Java, para se manipular objetos, usamos as variáveis do tipo referência. E para não esquecermos, toda variável que não for do tipo primitivo, é do tipo referência.

2.4. “Casting” e “Promotion” de tipos primitvos.

A característica de Casting e Promotion, ou conversão de tipos, esta muito presente dentro da linguagem java, e é será sempre necessário quando estivermos usando tipos diferentes.

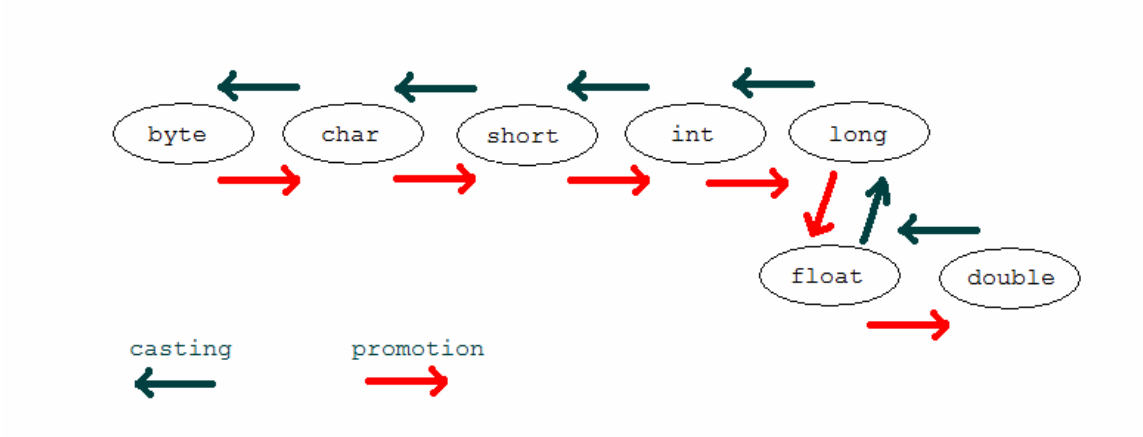


Figura 7

Vemos pelo desenho que se quisermos diminuir a precisão de uma variável, devemos fazer um **type-cast,** e quando houver o aumento da precisão precisamos fazer o **type-promotion.**

Em java, o compilador não nos obriga a fazer programaticamente o **type-promotion, pois o compilador java faz o auto-promotion** dos tipos primitivos.

Auto-promotion:

```
tipo-menor identificador2 = valorLiteral;

tipo-maior identificador = identificador2;
```

A precisão do tipo menor é aumentada para um tipo maior, sem problemas.

Entretando, o **type-cast** deve ser feito programáticamente, garantido as precisões.

```
tipo-maior identificador = valorLiteral;

tipo-menor identidicador2 = (tipo-menor) identificador;
```

Exemplos:

```
int x = 200;
byte b = x; // falha na compilação
byte c = (byte) x; // type-cast
float f = x; // auto-promotion

// lembre-se que os tipos preferidos são int e double.
int z = b + c;
double d = f + b;
```

2.5. O tipo referência na linguagem Java.

Os tipos referência foram inseridos na linguagem para facilitar a construção de código, gerência de memória pelo ***Garbage Collector***, e prover a nevagação entre os objetos do sistema.

Usamos os tipos referência para podermos manipular facilmente os objetos em memória.

Atualmente, com as capacidades computacionais crescendo, as linguagens orientadas a objetos, mesmo consumindo grandes capacidades de memória, são muito mais atrativas que as linguagens procedurais.

Usamos uma variável referência para ter acesso á uma instância de um objeto em memória. Então podemos criar dentro um programa um ou mais objetos Conta, e manipulá-los.

```
public class TesteConta {

public static void main(String args[]) {

     Conta conta1 = new Conta();
     conta1.numero = 10;
     conta1.saldo = 500;
     System.out.println( conta1 );
     System.out.println("Conta: " + conta1.numero);
     System.out.println("Saldo: " + conta1.saldo);

     Conta conta2 = new Conta();
     conta2.numero = 11;
     conta2.saldo = 5330;
     System.out.println( conta2 );
     System.out.println("Conta: " + conta2.numero);
     System.out.println("Saldo: " + conta2.saldo);
}
}
```

Dentro do programa, as variáveis conta1 e conta2, são do tipo referência, por duas razões:

1 – não são primitivos;

2 – estão associados ao operador ***new***, que significa criar um objeto em memória, e associar á variável o endereço de memória do objeto. A esta associação damos o nome de referência.

2.6. A notação do . (ponto) e o uso de uma referência.

Para acessarmos o número da conta 1, precisamos usar a sua referência, `conta1`, e nessa referência, usando a notação do ponto **(.)** temos acesso aos membros do objeto daquela referência:

```
     conta1.numero = 10;
     conta1.saldo = 500;
```

2.7. Endereçamento de memória da JVM para um objeto.

Os endereços de memória, impressos com @, são resultantes das linhas:

```
System.out.println( conta1 );
System.out.println( conta2 );
```

Estes endereços de memória, dos objetos Java, não nos interessa, ele interessa apenas ao ***Garbage Collector***.

Um objeto pode ter como atributo, não somente tipos primitivos, mas também tipos referência.

Vejamos a classe Produto abaixo:

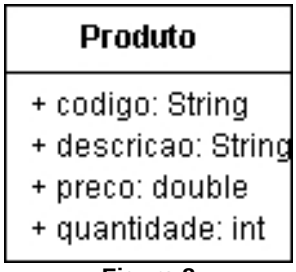


Figura 8

```
public class Produto {

     public String codigo;
     public String descricao;
     public double preco;
     public int quantidade;

}
```

Essa classe possui atributos que permite manipular vários tipos de produtos. Ou seja, analisando um supermercado, veremos que todos os seus produtos terão esses atributos.

Dentro desta classe vemos quais são os atributos não primitivos:

```
     String codigo;
String descricao;
```

Vemos também dois atributos primitivos:

```
     double preco;
int quantidade;
```

Podemos testar objetos do tipo Produto assim:

```
public class TesteProduto {

public static void main(String args[]) {

     Produto livro1  = new Produto();
     livro1.descricao = "Universidade Java!";
     livro1.codigo = "UNIJAVA01";
     livro1.preco = 55.00;
     livro1.quantidade = 20;
     System.out.println( livro1 );
     System.out.println( livro1.descricao );
     System.out.println( livro1.codigo );
     System.out.println( livro1.preco );
     System.out.println( livro1.quantidade );
     System.out.println();

     Produto livro2 = new Produto();
     livro2.descricao = "Dossie Hacker!";
     livro2.codigo = "DOSHACK1";
     livro2.preco = 66.0;
     livro2.quantidade = 15;
     System.out.println( livro2 );
     System.out.println( livro2.descricao );
     System.out.println( livro2.codigo );
     System.out.println( livro2.preco );
     System.out.println( livro2.quantidade );
}
}
```

A variável `livro1` do tipo `Produto` é do tipo referência, e ela permite armazenar outras duas referências dentro dela, `descricao` e `codigo`, do tipo `String`.

O tipo String, permite armazenar cadeias de caractere, ou seja, palavras, frases, etc.

Podemos usar uma variável do tipo String de duas formas:

```
livro2.descricao = "Dossie Hacker!";
```

ou

```
livro2.descricao = new String("Dossie Hacker!");
```

A primeira forma é a mais intuitiva, e preferida por programadores experientes.

A segunda forma, é mais clássica e fácil de entender, não resta dúvida que a variável `descrição` da classe `Produto` é do tipo `String`.

Revisando o exemplo:

```
public class TesteProduto2 {

     public static void main(String args[]) {

          Produto livro1  = new Produto();
          livro1.descricao = new String("Universidade Java!");
          livro1.codigo = new String("UNIJAVA01");
          livro1.preco = 55.00;
          livro1.quantidade = 20;
          System.out.println( livro1 );
          System.out.println( livro1.descricao );
          System.out.println( livro1.codigo );
          System.out.println( livro1.preco );
          System.out.println( livro1.quantidade );
          System.out.println();

          Produto livro2 = new Produto();
          livro2.descricao = new String("Dossie Hacker!");
          livro2.codigo = new String("DOSHACK1");
          livro2.preco = 66.0;
          livro2.quantidade = 15;
          System.out.println( livro2 );
          System.out.println( livro2.descricao );
          System.out.println( livro2.codigo );
          System.out.println( livro2.preco );
          System.out.println( livro2.quantidade );
     }
}
```

2.8. Comentando o código.

É muito comum escrever linhas de comentários dentro do código de um programa, com o propósito de explicar algoritmos e documentar a forma como o programador entendeu o que deve ser feito.

Um código bem comentado permite que a manutenção do programa, aplicação e do sistema, seja mais símples e mais precisa.

Tipos de comentários.

```
// comentário de uma única linha

/*
      Bloco de comentário
*/

/**
Comentário em formato Java Doc
*/
```

A J2SDK possui um utilitário muito interessante que converte os comentários escritos em Java Doc em páginas HTML com navegação e organização padronizados internacionalmente pela JavaSoft.

Veja na figura abaixo uma imagem exemplo da tela da documentação Java Doc.

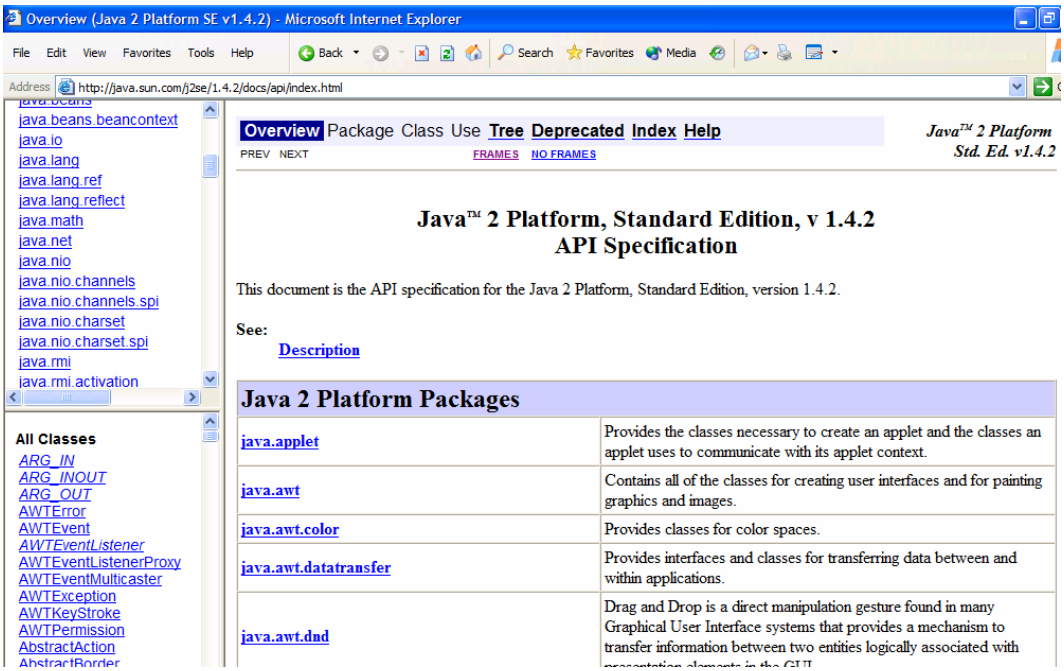


Figura 9

Veja aqui o Java Doc das classes que estão disponíveis dentro da J2SE 1.4.2
http://java.sun.com/j2se/1.4/docs/api/index.html

Veja aqui o tutorial de como utilizar o Java Doc:
http://java.sun.com/j2se/1.4.2/docs/tooldocs/windows/javadoc.html

2.9. Visualizando graficamente a memória

Podemos examinar um snapshot de memória e identificar que o `TesteProduto2` inicializou um objeto `livro1` do tipo `Produto`.

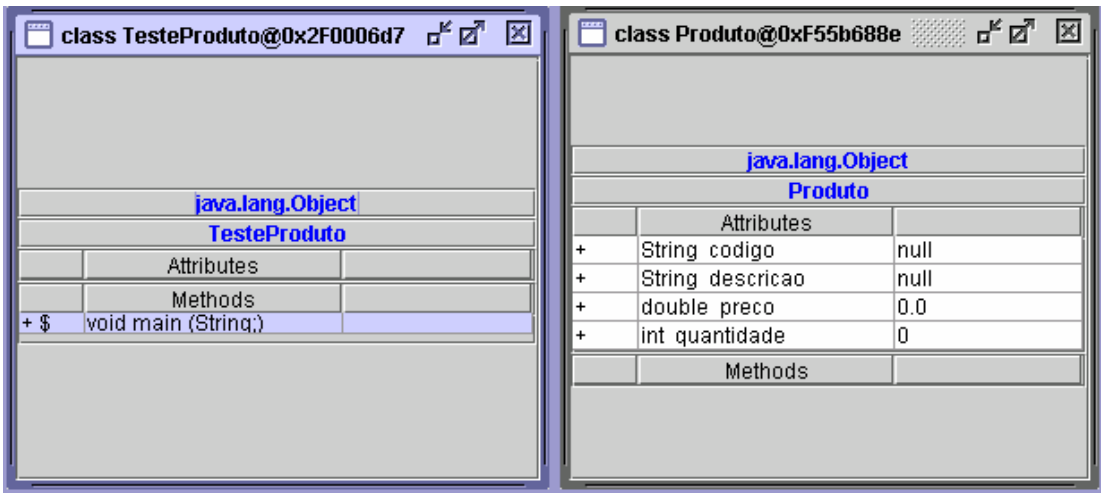


Figura 10

Vemos também que o `TesteProduto` possui duas referências para os objetos `livro1` e `livro2` do tipo `Produto`, bem como `livro1` e `livro2` possuem código e descrição na sua estrutura, que são referências para objetos do tito `String`;

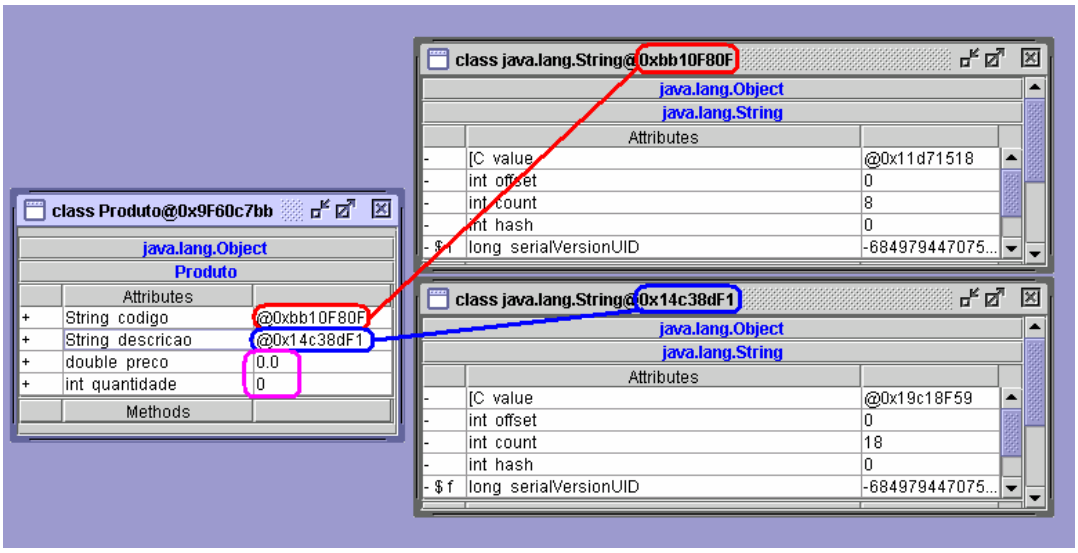


Figura 11

Em Java, quando uma variável não é do tipo primitivo, ela será do tipo referência, que por sua vez, é capaz de permitir a navegação entre os objetos. Vemos que `livro1.codigo` aponta para uma String através de uma referência.

Os tipos primitivos são armazenados diretamente em áreas de memória dentro dos objetos que os contém.

2.10. Operadores e expressões.

Vimos que uma linguagem de programação tem a finalidade de permitir a criação de aplicações visando automatizar processos manuais ou mentais, facilitando a vida das pessoas.

O processo de trabalho comum das pessoas é composto de etapas, e essas etapas podem requerer a execução de cálculos símples ou complexos.

A técnica computacional que nos permite executar tais cálculos é chamada de **expressão**.

Uma **expressão** é composta de variáveis e **operadores**. O bom entendimento e uso das expressões e dos operadores permitem com que os algoritmos do software sejam bem escritos e que sejam executados da forma que foram idealizados.

2.11. Operadores aritméticos

- soma (+);
- subtração (-);
- multiplicação (*);
- divisão (/);
- resto da divisão (%), apenas para operandos inteiros;
- incremento (++), operador unário definido apenas para operandos inteiros, podendo ocorrer antes da variável (pré-incremento) ou após a variável (pós-incremento); e
- decremento (--), operador unário definido apenas para operandos inteiros, podendo ocorrer antes da variável (pré-decremento) ou após a variável (pós-decremento).

Servem para executar cálculos usando os tipos primitivos inteiros ou de ponto flutuante.

Cada uma das linhas de cálculo exibida no código abaixo são consideradas expressões. Usamos as expressões para realizar contas aritméticas, comparações, etc.

Vamos compilar e executar este código abaixo e ver os resultados:

```
public class TesteOperadores {

public static void main(String a[]) {
      int i = 2 + 10;
      System.out.println("Somando 2 + 10 = " + i);
      int j = i * 2;
      System.out.println("Multilicando "+ i +" * 2 = "+ j);
      int h = j - 5;
      System.out.println("Subtraindo "+ j +" - 5 = "+ h);
      double d = h / 3;
      System.out.println("Divindo "+ h +" / 3 = "+ d);
      int r = h % 3;
      System.out.println("Resto de "+ h +" por 3 = "+ r);
      int x = i++;
      System.out.println("Incrementando "+i +" em 1 = "+ x);
      int z = j--;
      System.out.println("Decrementando "+j +" em 1 = "+ z);

}
}
```

2.12. Operadores lógicos

Operações lógicas sobre valores inteiros atuam sobre a representação binária do valor armazenado, operando internamente bit a bit. Operadores desse tipo são:

- complemento (~), operador unário que reverte os valores dos bits na representação interna;
- OR bit-a-bit (|), operador binário que resulta em um bit 1 se pelo menos um dos bits na posição era 1;
- AND bit-a-bit (&), operador binário que resulta em um bit 0 se pelo menos um dos bits na posição era 0;
- XOR bit-a-bit (^), operador binário que resulta em um bit 1 se os bits na posição eram diferentes;
- deslocamento à esquerda (<<), operador binário que recebe a variável cujo conteúdo será deslocado e um segundo operando que especifica o número de posições a deslocar à esquerda;
- deslocamento à direita com extensão de sinal (>>), operador binário que recebe a variável cujo conteúdo será deslocado e um segundo operando que especifica o número de posições a deslocar à direita. Os bits inseridos à esquerda terão o mesmo valor do bit mais significativo da representação interna;
- deslocamento à direita com extensão 0 (>>>), operador binário que recebe a variável cujo conteúdo será deslocado e um segundo operando que especifica o número de posições a deslocar à direita. Os bits inseridos à esquerda terão o valor 0.

Os operadores binários são muito usados em algoritmos de criptografia, e muito pouco usados no dia-a-dia, por serem uma forma muito interessante de se trabalhar com algoritmos bit-a-bit.

Vamos compilar e executar este código abaixo e ver os resultados:

```
public class TesteBinarios {

	public static void main(String a[]) {

		int i = 2 & 10;
		System.out.println("Somando bit a bit 2 com 10 = " + i);
		int j = 2 | 10;
		System.out.println("Multilicando bit a bit 2 com 10 = "+ j);
		int k = 2 ^ 10;
		System.out.println("O Complemento de 10 para 2 = " + k);
		int x = 1000;
		int div = x >> 2;
		System.out.println("Rotacionando a direita "+ x +" >> 2 = "+ div );
		int mult = x << 2;
		System.out.println("Rotacionando a esquerda "+ x +" << 2 = "+ mult
);
	}
}
```

2.13. Operadores condicionais

Os operadores condicionais permitem realizar testes baseados nas comparações entre valores numéricos e/ou variáveis, permitindo uma tomada de decisão dentro dos aplicativos. Eles são:

- E lógico (`&&`), retorna true se as duas expressões booleanas forem verdadeiras;
- OU lógico (||), retorna true se uma das duas expressões forem verdadeiras;
- maior (**>**), retorna `true` se o primeiro valor for exclusivamente maior que o segundo;
- maior ou igual (**>=**), retorna `true` se o primeiro valor for maior que ou igual ao segundo;
- menor (**<**), retorna `true` se o primeiro valor for exclusivamente menor que o segundo;
- menor ou igual (**<=**), retorna `true` se o primeiro valor for menor que ou igual ao segundo;
- igual (**==**), retorna `true` se o primeiro valor for igual ao segundo;
- diferente (**!=**), retorna `true` se o primeiro valor não for igual ao segundo.
- operador condicional ternário. Na expressão `b ? s1 : s2`, `b` é booleano (variável ou expressão). O resultado da expressão será o resultado da expressão (ou variável) `s1` se `b` for verdadeiro, ou o resultado da expressão (ou variável) `s2` se `b` for falso.

Os operadores condicionais, na maioria dos casos são usado para testes booleanos, de verdadeiro ou falso, dentro das estruturas de desvio condicional.

Vamos compilar e executar este código abaixo e ver os resultados:

```
public class TesteCondicionais {

	public static void main(String a[]) {

		int i = 2;
		int j = 2;
		int k = 10;
		int z = 0;
		boolean b = true;
		boolean c = true;
		boolean f = false;
		System.out.println( i+" igual a "+ j + " : "+ (i == j) );
		System.out.println( i+" maior que "+ k + " : "+ (i > k) );
		System.out.println( i+" menor que "+ k + " : "+ (i < k) );
		System.out.println( i+" menor que "+ z + " : "+ (i < z) );
		System.out.println( i+" maior ou igual a "+j+" : "+(i >= j) );
		System.out.println( i+" menor ou igual a "+j+" : "+(i <= j) );
		System.out.println( (i<3) ? i +" menor que 3" : i +" maior que 3" );
		System.out.println( b+" && "+c+ " = "+ ( b && c ) );
		System.out.println( b+" && "+f+ " = "+ ( b && f ) );
		System.out.println( b+" || "+f+ " = "+ ( b || f ) );
	}
}
```

2.14. Precedência de operadores

Agora que conhecemos os operadores e expressões, precisamos entender a precedência deles, para evitarmos que nossos programas façam os cálculos errados.

Regras de precedência:

1. Operações entre parênteses;
2. Incremento e Decremento;
3. Multiplicação e divisão, executados da esquerda para a direita;
4. Adição e subtração, executados da esquerda para a direita;

Vamos compilar e executar este código abaixo e ver os resultados:

```
public class TestePrecedencia {

	public static void main(String args[]) {
		int x = 2 + 3 / 5 * 7 - 6;
		System.out.println(" 2 + 3 / 5 * 7 - 6 = "+ x );

		int j = (2 + 3)/ (5*7) - 7;
		System.out.println("(2 + 3)/ (5*7) - 6 = " + j );

		int z = 2 + 3 / 5 * ( 7 - 6 );
		System.out.println("2 + 3 / 5 * ( 7 - 6 ) = " + z );
	}
}
```

Vemos que a ordem de como tratamos os valores e variáveis dentro de uma expressão, podem interferir no resultado final.

Por isso é extremamente interessante que se use os parênteses dentro das expressões de forma a garantir a ordem de como os cálculos serão feitos.

2.15. Estruturas de repetição.

Como todas as aplicações e sistemas são baseados em execução de tarefas, e em alguns casos, com repetição de certas partes delas, todas as linguagens de programação fornecem estruturas para repetição, comumente chamamos de laço.

Em Java existem três estruturas que são comumente usadas para executar blocos de programas em ciclos de execução.

Imaginem um programa que tivesse de executar repetidamente uma determinada rotina, como faríamos?
Escrevemos infinitas linhas de código ou usando as estruturas de laço??

Um exemplo símples. Imagine um bloco de código que imprimisse os números da sena:

```
public class Sena {

     public static void main(String args[]) {
          System.out.println("1,2,3,4,5,6,7,8,9,10,16..,60");
     }
}
```

Ou ficaria melhor usando uma estrutura de repetição? Entendo essa situação, veremos o uso do **`for, while, do .. while.`**

2.16. Usando o for.

O for, foi projeto para testar uma condição inicial antes de iniciar o laço, executar uma declaração, executar o bloco de código dentro do laço, e no fim de uma repetição executar uma expressão.

Para usá-lo devemos entender sua sintaxe:

```
for ( declaracao; condição booleana; expressao ) {

     // antes de iniciar o laço, e executada a
     // declaração

     // bloco de codigo que sera repetido
     // ate que a condição booleana seja false

     // antes da execução iniciar o proximo laço
     // sera a executada a expressao

}
```

Então, vamos converter o programa da Sena para fazermos o uso do for. Sabemos que os números da Sena variam de 1 a 60, e nessa ordem devem ser impressos.

```
public class Sena {

     public static void main(String args[]) {
          System.out.println("Numeros da Sena");
          for ( int i=1; i <= 60; i++ ) {
               System.out.print(i+",");
          }
          System.out.println();
     }

}
```

A linha que contém o for, diz ao interpretador: “Execute o que estiver compreendido no meu bloco de código ate que o `i` seja menor ou igual a `60`, e antes de começar o próximo laço, incremente 1 em `i`”.

2.17. Usando o while.

Existe dentro do Java, uma estrutura de repetição mais simples, o **while**. O **while** funciona muito parecido com o for, entretanto ele tem menos parâmetros para a sua execução.

For e **while**, testam se podem começar a executar ou se podem executar aquele laço, antes de começarem.

Para usá-lo devemos entender sua sintaxe:

```
while (condição booleana) {

     // bloco de codigo que sera repetido
     // ate que a condição booleana seja false
}
```

Com uma estrutura mais símples, o controle das repetições fica mais complexo.

Usando o while para escrever o programa da Sena:

```
public class SenaWhile {

     public static void main(String args[]) {
          System.out.println("Numeros da Sena");
          int i = 1;
          while( i <= 60 ) {
               System.out.print(i+",");
               i++;
          }
          System.out.println();
     }
}
```

Devemos nos atentar, pois dentro do **while**, incrementamos um contador (i++), o qual é verificado a cada laço. Quando esse valor de i, chegar a 61, a execução é desviada para o fim do laço.

2.18. Usando o do .. while.

Assim como **while**, o **do .. while** fará um teste para verificar se ele continuar repetindo a execução as linhas de código internas ao laço.

A diferença, é que o **do .. while** testa no fim da execução e não antes dela. Ou seja, pelo menos uma execução acontece, diferente do **while** que testa no inicio.

Para usá-lo devemos entender sua sintaxe:

```
do {

     // bloco de codigo que sera repetido
     // ate que a condição booleana seja false
} while ( condicao );
```

Usando o do .. while para escrever o programa da Sena:

```
public class SenaDoWhile {

     public static void main(String args[]) {
          System.out.println("Numeros da Sena");
          int i = 1;
          do {
               System.out.print(i+",");
               i++;
          } while( i <= 60 );
          System.out.println();
     }

}
```

Vemos que neste caso, o while e do .. while, tiveram o mesmo comportamento.

Se quiséssemos imprimir a tabuada de um determinado número? Qual o algoritmo a seguir?

Devemos imprimir o número, e um valor quando multiplicado de 1, e sucessivamente até chegar ao 10.

Então podemos escrever o seguinte código:

```
public class Tabuada {

	public static void main(String args[]) {
		int numero = 5;
		System.out.println("Tabuada do :"+ numero);
		for ( int i=1; i<=10; i++) {
			int resultado = numero * i;
			System.out.println( numero +" * "+ i +" = " + resultado );
		}
	}
}
```

2.19. Laços Aninhados.

Podemos ainda combinar o uso das estruturas de repetição sem problemas, e com o intuito de executar rotinas complexas.

Veja o código abaixo, e veja como podemos aninhar as estruturas:

```
public class Aninhados {

	public static void main(String args[]) {
		int i = 1;
		while ( i <= 9 ) {
			for ( int j = 1; j <= 10; j++ ) {
				System.out.print(i+""+j+",");
			}
			System.out.println();
			i++;
		}
	}
}
```

No próximo módulo exploraremos mais os laços de repetição e suas combinações.

2.20. Estruturas de seleção.

Como a finalidade dos sistemas de computação é mapear processos físicos ou mentais, facilitando a vida dos usuários, em todos eles exigem a necessidade de tomada de decisões dentro da execução de uma rotina. Essas tomadas de decisões são implementadas usando as estruturas de seleção.

Uma estrutura de seleção apresenta um desvio no processamento das informações, permitindo a criação de fluxos alternativos ao fluxo principal daquela rotina ou função.

Aliados as estruturas de repetição, poderemos criar algoritmos completos de execução de funções ou rotinas, para resolver problemas computacionais simples ou complexos.

As estruturas de seleção são usadas de forma comparativa, ou seja, se uma determinada variável possui um determinado valor, o programa segue um fluxo, caso contrário o programa segue outro fluxo.

Podemos representar isso graficamente.

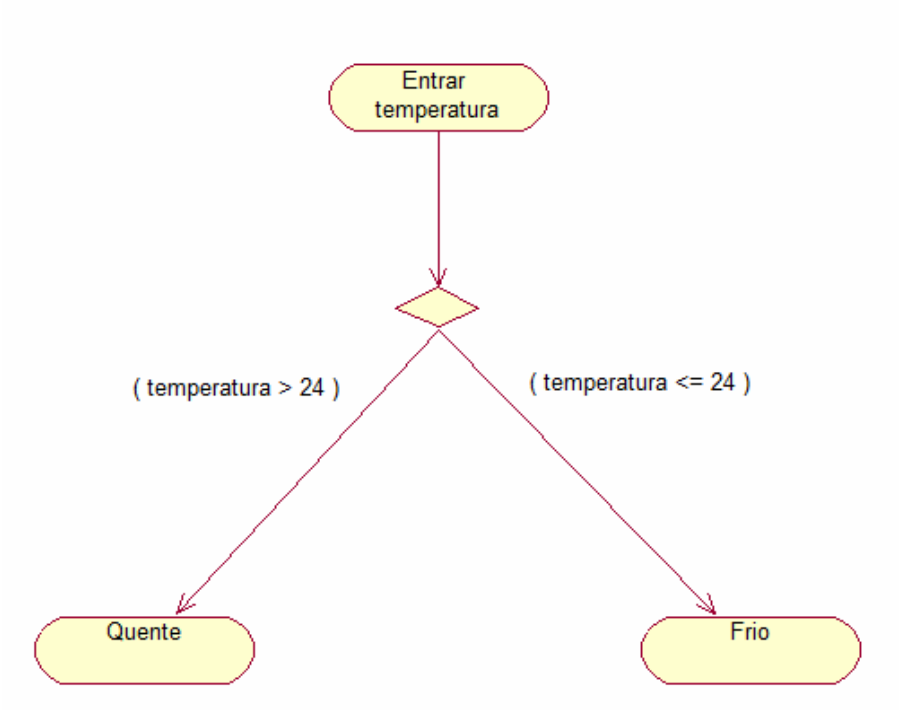


Figura 12

Podemos também aninhar as estruturas de decisão, para implementar algoritmos mais complexos.

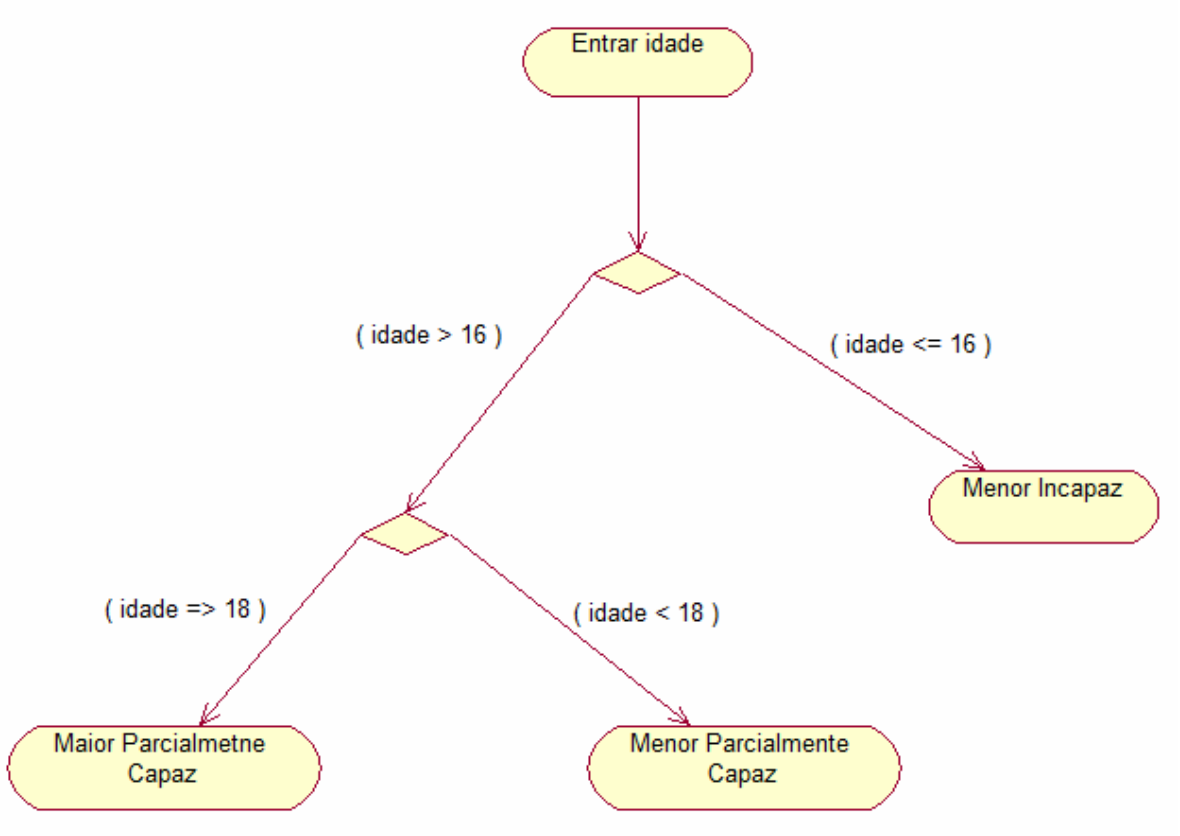


Figura 13

Em Java, existem duas estruturas de seleção:

```
if;
switch;
```

2.21. Usando o `if`.

O if (tradução SE), é usado para comparar expressões, variáveis e condições booleanas.

Sintaxe:

```
if ( expressão booleana ) {

      // bloco de código que será executado
      // se a expressão booleana tiver valor true.
}
```

Ainda dentro da estrutura do `if` podemos controlar os desvios de fluxo usando o `else`. O uso do `else` é opcional.

```
if ( expressão booleana ) {

      // bloco de código que será executado
      // se a expressão booleana tiver valor true.
} else {
      // bloco de código que será executado
      // se a expressão booleana tiver valor false.

}
```

No caso de comparar múltiplas condições, podemos usar o else if.

```
if ( expressão booleana ) {

      // bloco de código que será executado
      // se a expressão booleana tiver valor true.
} else if (  expressão booleana2) {
      // bloco de código que será executado
      // se a expressão booleana2 tiver valor true.

} else {
      // bloco de código que será executado
      // se a expressão booleana e a expressao booleana2
      //tiverem  valor false.

}
```

Dentro da expressão booleana, podemos usar os operadores lógicos OR (||) e AND (&&) para testar condições compostas facilitando a construção da estrutura `if`.

Estes operadores são chamados tecnicamente de "***short-circuit operators***".

Exemplos:

```
if ( idade >= 0 && idade < 16 ) {
// executara este bloco de codigo se
// a idade for maior ou igual a zero E
// a idade for menor que 16
} else if ( idade >= 17 && idade < 18 ) {
// executara este bloco de codigo se
// a idade for maior ou igual a 17 E
// a idade for menor que 18
} else {
// executara este bloco de código se
// a idade for maior ou igual a 18
}

...

if ( x == 0 || y == 0 ) {

// executara este bloco de código
// se x OU y forem igual a zero
} else {
// executara este bloco de código
// se x E y forem diferentes de zero
}
```

2.22. Usando o `switch`.

O `switch` funciona com uma estrutura para tratamento de casos específicos. Ele funciona com um único teste, ou seja, a estrutra testa uma condição numérica (não booleana), somente uma vez. E partir daquela condição ele executará todos os casos.

Sintaxe do `switch`:

```
switch ( int ) {

case (inteiro):
      // código
case (inteiro):
      // código
case (inteiro):
      // código
default:
      // código
}
```

O **`default`** é opcional.

Nota: obrigatoriamente, o parâmetro que será comparado **somente pode ser um `byte` ou `short` ou `char` ou `int`.**

O caso de comparação (`case (inteiro):`) obrigatoriamente tem que ser um literal inteiro ou uma constante.

Exemplo de uso do `switch`.

```
switch ( x ) {

case 1:
      // código A

case 10:
      // código B

case 100:
      // código C

default:
      // código D
}
```

Simulando:

quando x = 1, linhas executadas: A,B,C e D;
quando x = 10, linhas executadas: B,C e D;
quando x = 100, linhas executadas: C e D;
qualquer valor de x, diferente de 1, 10 e 100, D.

Podemos controlar o fluxo de execução do **`switch`**, ou de qualquer bloco de seleção ou repetição usando o **`break`**; Ele interrompe a execução saindo do bloco.

```
switch ( x ) {

case 1:
      // código A
      break;

case 10:
      // código B

case 100:
      // código C
      break;

default:
      // código D
}
```

Simulando:

quando x = 1, linhas executadas: A;
quando x = 10, linhas executadas: B e C;
quando x = 100, linhas executadas: C;
qualquer valor de x, diferente de 1, 10 e 100, D.

Discussão

O que se entendeu por variáveis de referência?

O que se entendeu por variáveis de tipos primitivos?

O que se entendeu por **casting** e **promotion**?

Para que servem as palavras reservadas?

Uma classe tem membros, que podem ser atributos ou métodos?

Então escreveremos nossos programas em Java, usando sempre mais uma classe?

Numa expressão, num fluxo de decisão, numa comparação, usamos expressões, operadores e estruturas de repetição e/ou decisão. Quando temos problemas complexos como devemos estruturar esses algoritmos?

Exercícios

3. Criando Classes em Java a partir de um modelo Orientado a Objetos

Até agora vimos as principais estruturas da linguagem java. A partir deste ponto, estaremos definindo e implementando os conceitos iniciais da Programação Orientada a Objetos.

E quando falamos de implementação de sistemas orientados a objetos, devemos dominar o desenho e criação de classes.

Vamos revisar o conceito de classe:

"Uma classe é um modelo para a criação de objetos de um determinado tipo, a partir de um sistema real ou imaginário. Ela pode conter características de dados (atributos) e de funcionalidades (métodos). A partir de uma classe, podemos criar várias instância de objetos com características em comum."

"Para se definir uma classe, analisamos uma categoria de objetos dentro de um contexto real ou imaginário, abstraindo suas características de dados e de funcionalidade. A partir desta classe, podemos instanciar um objeto dando-lhe uma referência (nome). Podemos usar a referência de um objeto para acessar seus atributos e utilizar seus métodos."

Vemos então que uma classe é composta de atributos (dados) e métodos (funções).

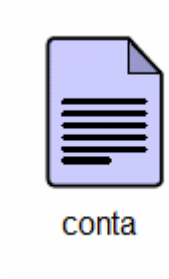


Figura 14

atributos: saldo, limite, juros, vencimento, etc.

funcionalidades: debito, credito, consultar saldo, etc.

A partir desta classe Conta, podemos criar vários tipos de conta que possua saldo, limite, juros e vencimento, e ainda podemos executar operações de débito, crédito e consultar saldo.

```
Conta cartaoCredito = new Conta();

Conta banco = new Conta();
```

A cada vez que o operador **new** é executado, cria-se uma instância em memória capaz de armazenar dados e executar funções. A essa instância em memória, damos o nome de objeto. Neste caso, temos duas instâncias em memória, do tipo Conta. Uma chama-se **`cartaoCredito`**, e a outra, **`banco`**.

Através de uma dessas referências, podemos acessar seus atributos e métodos.

Para facilitar o entendimento, a documentação e o desenho de um sistema orientado a objetos devemos lembrar da UML (Unified Modeling Language), já citada anteriormente.

deste capítulo abordaremos somente o Diagrama de Classes da UML, que nos auxiliará no processo de aprendizado sobre classes e objetos.

Como vimos anteriormente, a programação orientada a objetos é baseada em três fundamentos:

Encapsulamento;
Herança;
Polimorfismo;

Cada um deles contém um conjunto de conceitos, os quais devemos dominar.

Todos esse conceitos nos trarão muitos benefícios durante o desenvolvimento de software, e garantem que a evolução, adicionado melhorias e novas funcionalidades, sejam mais produtivas e rápidas.

Neste módulo, veremos os conceitos de encapsulamento e seus desdobramentos.

NOTA: No próximo módulo, veremos como implementar os conceitos de Herança e Polimorfismo.

3.1. Encapsulamento

Definição clássica oriunda da Análise e Desenho Orientado a Objetos: "***Disponibilizar uma interface pública, com granularidade controlada, para manipular os estados e executar operações de um objeto***".

Resumindo, devemos esconder os atributos dos objetos provendo uma interface pública de métodos, impedindo o acesso a eles diretamente por outros objetos.

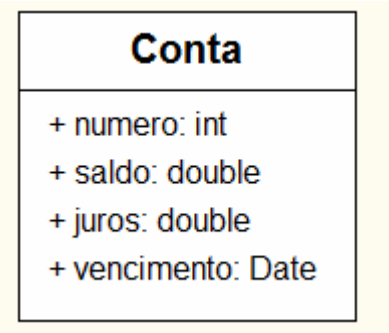
A linguagem Java, disponibiliza um conjunto de modificadores de acesso para os membros de classe, que nos permite definir vários níveis de encapsulamento.

Modificadores de acesso para os membros de classe:

Modificador	Mesma Classe	Mesmo Pacote	Subclasses	Universo
`private (-)`	*			
`<none>`	*	*		
`protected (#)`	*	*	*	
`public (+)`	*	*	*	*

*** acesso permitido.**

Analisando os modificadores de acesso, para encapsular um atributo de um objeto, devemos demarcá-lo como **`private`**, fazendo com que somente as referências de objetos provenientes desta classe tenham acesso a ele.




```
public class Conta {

	public int numero;
	public double saldo;
	public double juros;
	public Date vencimento;
}
```

Figura 15

(+) **significa que o membro tem modificador público**

Com os atributos públicos, podemos escrever código assim:

```
Conta contaBanco = new Conta();
contaBanco.numero = 1;
contaBanco.saldo = 100;
contaBanco.juros = 0.089;
```

Para fazer operações:

```
contaBanco.saldo += 300; // depósito de 300;
contaBanco.saldo -= 200; // saque de 200;
contaBanco.saldo = 1000; // alteracao direta de saldo
```

Analisando o mundo real de um banco, não podemos fazer tais operações sem haver um mínimo de validação de saldo, de valores a serem depositados, etc.

Para garantir que tais regras serão executadas, devemos usar o encapsulamento, assim alteraremos o modelo da classe Conta, para ter atributos privados e métodos públicos para manipulá-los.

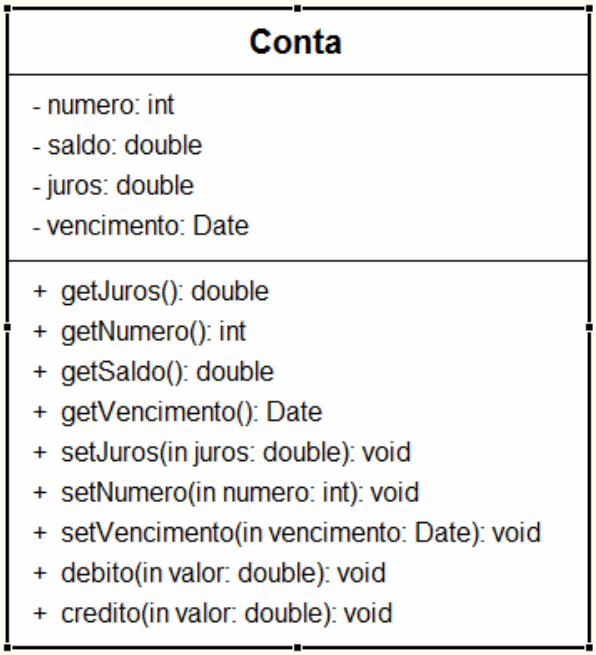


Figura 16

Legenda:
+ público
- privado

Notas:
1 - para cada atributo que pode ser consultado criamos um método getNomeAtributo () (tecnicamente chamando de ***accessor method);***

2 - para cada atributo que pode ser alterado, criamos um método setNomeAtributo() (tecnicamente chamando de ***mutator method***);

3 – para os métodos que executam operações ou funções dos objetos, damos um nome que corresponde aquela função ou operação, facilitando seu entendimento. (tecnicamente chamando de ***worker method***);

4- vemos que todos os atributos estão marcados como **`private`** e todos o métodos como **`public`**, esta é a forma de fornecer o encapsulamento em Java.

Examinemos o código abaixo para entendermos como os modificadores serão usados para determinar os atributos privados e métodos públicos de uma classe Java que nos permitirá construir objetos encapsulados.

Neste instante, acredite que o código está correto e funcional, o seu total entendimento será possível até o fim deste capítulo.

```
public class Conta {

	private int numero;
	private double saldo;
	private double juros;
	private java.util.Date vencimento;

	//accessor methods
	public double getJuros() {
		return juros;
	}

	public int getNumero() {
		return numero;
	}

	public double getSaldo() {
		return saldo;
	}

	public java.util.Date getVencimento() {
		return vencimento;
	}

	//mutator methods
	public void setJuros(double juros) {
		this.juros = juros;
	}

	public void setNumero(int numero) {
		this.numero = numero;
	}

	public void setVencimento (java.util.Date vencimento) {
		this.vencimento = vencimento;
	}

	//worker methods
	public void debito(double valor) {
		this.saldo -= valor;
	}

	public void credito(double valor) {
		this.saldo += valor;
	}

}
```

Passaremos agora a entender a utilização e como escrever esses tipos de métodos, para garantir o encapsulamento dos objetos.

3.2. Definindo variáveis

Vimos anteriormente que sempre que desejarmos manipular determinado dado, ele deve ter um tipo, e estar associado á uma variável. Essa variável deveria ter um nome de fácil entendimento, este nome é o seu identificador dentro do código, e usamos este identificador para acessar o conteúdo de dados desta variável.

Na programação orientada a objetos, existem quatro tipos de variáveis, definidas pela forma como o interpretador Java vai armazená-las. Por isso, devemos entender como o interpretador Java tratará cada um dos tipos, para podermos escrever nossos programas de maneira correta e garantir o comportamento desejado deles.

Tipos de variáveis:

De instância (atributos de objeto);
De classe;
Locais (variáveis e parâmetros de método);
Constantes;

Para se manusear esses tipos de variáveis, devemos escrever métodos e seguir as práticas de encapsulamento. Por isso, existem três tipos de métodos para manipularmos esses tipos de variáveis.

Tipos de métodos:

De instância (métodos do objeto);
De classe (métodos da classe);
Construtores (método especial);

Relembrando, para se definir uma variável, devemos escolher um tipo de acordo com sua abrangência de tamanho e valor, um modificador quando for necessário, um nome de fácil entendimento e um valor inicial, podendo ser do tipo primitivo ou referência, seguindo a notação:

Formas:

```
[modificador] tipo identificador;

[modificador] tipo identificador = [valor_inicial];
```

Exemplos:

```
int idade = 18;
float temperatura = 31.2;
double salario = 1456.3;
char letraA = 'A';
byte base = 2;
long distancia = 2000;

String faculdade = new String("IBTA!");
```

3.3. Entendendo a diferença entre “de classe” e “de instância”

Uma variável é considerada de instância quando ela estiver definida no corpo da classe, ou seja, fora dos métodos e não possuir o modificador **`static`**.

Quando uma variável for definida como parâmetro de método, ou dentro de um método, ela é chamada de local.

Quando uma variável for definida fora dos métodos e possuir o modificador **`static`**, ela será considerada como atributo de classe.

Quando uma variável for definida fora dos métodos e possuir o modificador **`final`**, ela será considerada uma constante.

Quando um método não possuir o modificador **`static`**, ele será considerado como de instância (**non-`static`**). Seu efeito restringe-se somente ao objeto chamado pela sua referência.

Quando um método possuir o modificador **`static`**, ele será considerado como de classe(**`static`**). Seu efeito abrange todos os objetos oriundos daquela classe.

Um método de instância (**non-`static`**), mesmo não sendo indicado, pode manipular atributos ou métodos de classe(**`static`**),.

Um método de classe (**`static`**) não pode manipular diretamente atributos ou métodos instância, para isso ele necessitará de uma referência para o objeto.

Atributos e métodos de classe (**`static`**), são raramente utilizados na programação orientada a objetos, pois eles não estão ligados aos objetos, e sim a uma classe.

3.4. Atributos e métodos de instância (objeto)

Toda vez que quisermos armazenar um dado dentro de um objeto, e poder acessá-lo através da referência deste objeto, definiremos estes dados como variável de instância, ou seja, o seu valor pode ser alterado ou consultado usando uma referência de um objeto.

Um atributo de instância é visível á todos os métodos daquela instância. Vimos que a instância de um objeto corresponde a uma área de memória criada para armazenar a estrutura daquele objeto, que pode conter variáveis do tipo primitivo e do tipo referência.

Analisando um trecho do código da classe Conta:

```
public class Conta {

private int numero;
private double saldo;
private double juros;
private java.util.Date vencimento;
..
}
```

Vemos que ela possui 4 atributos de instância, 3 do tipo primitivo (numero, saldo e juros) e um do tipo referência java.util.Date (vencimento).

Podemos criar a partir da classe Conta, vários objetos do tipo Conta com conteúdo diferente e que pode ser manipulados através de uma referência.

No código abaixo vemos duas instâncias de objetos a partir da classe Conta, nomeadas como `conta1` e `conta2`, executando o programa `TesteConta`, vemos que elas tem valores diferentes, ou seja, são objetos diferentes, que podem ter dados diferentes.

```
public class TesteConta {

public static void main(String args[]) {
      Conta conta1 = new Conta();
      System.out.println("Ref. conta1: " + conta1 );

      Conta conta2 = new Conta();
      System.out.println("Ref. conta2: " + conta2 );
}
}
```

No código abaixo, vemos que podemos usar os métodos definidos na classe conta (getNumero(), getSaldo(), credito() e débito), e que seu uso refletem somente na referência daquela instância:

```
public class TesteConta2 {

	public static void main(String args[]) {
		Conta conta1 = new Conta();
		System.out.println("Ref. conta1: " + conta1 );
		conta1.setNumero(1);
		conta1.credito(100);
		conta1.debito(20);
		System.out.println("conta1: Numero: " + conta1.getNumero() );
		System.out.println("conta1: Saldo: " + conta1.getSaldo() );
		System.out.println();

		Conta conta2 = new Conta();
		System.out.println("Ref. conta2: " + conta2 );
		conta2.setNumero(2);
		conta2.credito(200);
		conta2.debito(40);
		System.out.println("conta2: Numero: " + conta2.getNumero() );
		System.out.println("conta2: Saldo: " + conta2.getSaldo() );
	}
}
```

A programação orientada a objetos, consiste em abstrair os comportamentos de objetos comuns, categorizando-os em classes.

Ao definirmos uma classe, vamos determinar seus atributos e métodos usando o encapsulamento. A partir daí, podemos usar essa classe em qualquer tipo de programa java. No caso acima, usamos a classe Conta, para criarmos dois objetos dentro um programa stand-alone.

Para completar os mecanismos de encapsulamento, veremos como definir e escrever métodos para nossas classes.

3.5. Definindo Métodos

Usamos os métodos para executar operações, funções, ações, etc. dentro das aplicações. Devemos entender a sua utilidade, pois eles são as ferramentas mais importantes durante o desenvolvimento de aplicações, pois eles representam a parte funcional dos sistemas.

Para se escrever métodos eficientemente, devemos entender primeiro os tipos de métodos que podemos escrever numa classe:

Categorias de métodos baseados na programação orientada a objetos:

accessor : (de acesso), permite ler o valor de um atributo;
mutator: (de alteração), permite alterar o valor de um atributo;
worker: (de trabalho), permite executar uma determinada tarefa;
constructor: (de inicialização), permite criar um objeto e inicializá-lo;

Analisando a classe Conta listada acima, vemos que ela possui três tipos de métodos:

accessors: (todo os métodos que começam com get);
mutators: (todos que começam com set) e;
workers: (representam operações que podem ser feitas com uma Conta).

Usando este conceito então, se tivermos uma classe Pessoa, que deva ter atributos to tipo caracter, como nome e cpf, poderemos defini-la inicialmente assim:

```
public class Pessoa {

      private String nome;
      private String cpf;

}
```

Como devemos seguir a prática do encapsulamento, devemos então criar ao menos, um conjunto de métodos (set e get) para cada um dos atributos:

Para definirmos um método, ele deve seguir a seguinte sintaxe:

```
[modificadores] tipo_retorno identificador( [parametros] ) {
      // corpo do método
}
```

Criando os métodos get para os atributos de instância da classe Pessoa:

```
public String getNome() {
// nome é de instância, do tipo String
return nome;
}

public String getCPF() {
      // cpf é de instância, do tipo String
return cpf;
}
```

Criando os métodos set para os atributos de instância da classe Pessoa:

```
public void setNome(String n) {
// n é uma variavel local do tipo String, nome é de instância
nome = n;
}

public void setCPF(String c) {
// c é uma variavel local do tipo String, cpf é de instância
cpf = c;
}
```

O objetivo do método set é alterar o valor do atributo de instância, e do método get, retornar o valor do atributo de instância.

Devemos lembrar que somente podemos associar variáveis do tipo referência quando forem do mesmo tipo, e para as variáveis do tipo primitivo, devemos obedecer as regras de **casting** vistas anteriormente.

Todo método que possuir um tipo de retorno diferente de **`void`** devemos obrigatoriamente especificar um tipo primitivo ou um tipo referência como tipo de retorno, e no corpo do método retornar um valor ou variável do tipo do retorno usando a instrução **`return`**.

Podemos também criar um método de trabalho (worker), para a validação do CPF:

```
public boolean validaCPF() {
      // variavel local
boolean validado = false;

// este algoritmo somente verfica se o cpf não e nulo.
if ( cpf != null ) {
      validado = true;
} else {
            validado = false;
}

return validado;
}
```

A classe Pessoa fica assim então:

```
public class Pessoa {

	private String nome;
	private String cpf;

	public String getNome() {
		// nome é de instância, do tipo String
		return nome;
	}

	public String getCPF() {
		// cpf é de instância, do tipo String
		return cpf;
	}

	public void setNome(String n) {
		// n é uma variavel local do tipo String, nome é de instância
		nome = n;
	}

	public void setCPF(String c) {
		// c é uma variavel local do tipo String, cpf é de instância
		cpf = c;
	}

	public boolean validaCPF() {
		// variavel local
		boolean validado = false;

		// este algoritmo somente verfica se o cpf não e nulo.
		if ( cpf != null ) {
			validado = true;
		} else {
			validado = false;
		}
		return validado;
	}

}
```

Vemos que a variável `validado` criada dentro do método `validaCPF()` é uma variável local, ou seja, ela existe somente dentro daquele bloco de código, dentro do método. Fora do método, esta variável não existe.

Entretanto, a variável `cpf` que fora definida fora dos métodos, é considerada como de instancia, e está visível dentro dos métodos `setCPF(), getCPF()` e `validaCPF()`.

Uma variável local existe somente durante a execução daquele método. Uma variável de instância existe enquanto o objeto estiver em memória e não estar eleito para ser coletado pelo ***Garbage Collector***.

Podemos testar a classe Pessoa usando o programa TestePessoa:

```
public class TestePessoa {

	public static void main(String args[]) {

		Pessoa pessoa1 = new Pessoa();
		pessoa1.setNome("Jose Antonio");
		pessoa1.setCPF("123456789-00");
		System.out.println("Nome: "+ pessoa1.getNome());
		System.out.println("CPF: "+ pessoa1.getCPF());
		System.out.println("CPF Valido: "+ pessoa1.validaCPF());
		System.out.println("");

		Pessoa pessoa2 = new Pessoa();
		pessoa2.setNome("Antonio Jose");
		System.out.println("Nome: "+ pessoa2.getNome());
		System.out.println("CPF: "+ pessoa2.getCPF());
		System.out.println("CPF Valido: "+ pessoa2.validaCPF());
	}
}
```

3.6. Definindo Construtores

Como exposto anteriormente, podemos definir métodos para a inicialização dos objetos, um construtor é um método especial, por não possui tipo de retorno e levar o nome da Classe que o contém.

Um construtor pode conter parâmetros, os quais usamos para inicializar os nossos objetos de forma mais aderente ao modelo que estamos mapeando.

Para se definir um construtor seguimos a seguinte sintaxe:

```
[modificador] NomeClasse ( [parametros] ) {
     // corpo do construtor.
}
```

Toda classe em Java deve possuir ao menos um construtor. Opa! Mas até agora não escrevemos nenhum construtor em nossas classes, e todas elas compilaram e executaram, porque?

O compilador javac, se ele não encontrar nenhum construtor definido no código da classe, ele coloca para nós o construtor padrão:

```
public NomeClasse () {
}
```

O compilador Java vai inserir exatamente o código acima quando compilar uma classe Java que não possui nenhum construtor definido.

Para a classe Conta, o construtor padrão fica:

```
public Conta () {
}
```

Para a classe Pessoa, o construtor padrão fica:

```
public Pessoa () {
}
```

É por isso que nos programas de Teste, quando são chamadas as linhas que possuem o operador **new** e um construtor, funcionam. Pois o compilador insere o construtor padrão se não nos lembrarmos dele.

O construtor de uma classe, é executado somente uma vez no ciclo de vida de um objeto.

Podemos escrever para uma classe vários construtores, com parâmetros diferentes para podermos inicializar nossos objetos de forma diferente.

Analisando a classe Conta, podemos inicializá-la com um saldo padrão, com um saldo inicial, com juros iniciais, e data.

```
public class Conta {

private int numero;
private double saldo;
private double juros;
private java.util.Date vencimento;
..

      // construtores
      // padrão
      public Conta() {
      numero = 0;
      // saldo padrao é ZERO.
      saldo = 0;
      juros = 0;
      // data atual do sistema
      vencimento = new java.util.Date();
      }

      // com parametros
      public Conta(int numeroConta, double saldoInicial) {
      numero = numeroConta;
      saldo = saldoInicial;
      juros = 0;
      // data atual do sistema
      vencimento = new java.util.Date();
      }

      public Conta(int numeroConta, double saldoInicial,
            double jurosInicial) {

      numero = numeroConta;
      saldo = saldoInicial;
      juros = jurosInicial;
      // data atual do sistema
      vencimento = new java.util.Date();
      }


      public Conta(int numeroConta, double saldoInicial,
      double jurosInicial, java.util.Date dataVencimento) {

            numero = numeroConta;
            saldo = saldoInicial;
            juros = jurosInicial;
            vencimento = dataVencimento;
      }

      ...

}
```

Analisando a classe Pessoa, podermos inicializá-la com um nome e com um CPF.

```
public class Pessoa {

	private String nome;
	private String cpf;

	public Pessoa() {
	}

	public Pessoa(String nomePessoa) {
		nome = nomePessoa;
	}

	public Pessoa(String nomePessoa, String cpfPessoa) {
		nome = nomePessoa;
		cpf = cpfPessoa;
	}
	...
}
```

O código dos construtores de uma classe se originará a partir da análise de como eles devem ser inicializados durante a execução dos aplicativos.

O construtor é usado para passagem de parâmetros de inicialização á uma instância de um objeto no ato de sua construção, quando chamamos usando a instrução `new`.

A função do construtor, além de inicializar a instância de um objeto a partir de uma classe, é retornar o endereço de memória para a referência que a instrução new está sendo associada.

```
Pessoa pessoa1 = new Pessoa("Jose Antonio", "123456789-00");
```

A variável `pessoa1`, do tipo referência, recebe o endereço de memória em que o objeto foi alocado. E partir desta referência, temos acesso aos valores passados ao objeto através de seus atributos e métodos de instância.

Executando a TesteConta3:

```
public class TesteConta3 {

	public static void main(String args[]) {
		Conta conta1 = new Conta(1, 100, 0);
		System.out.println("Ref. conta1: " + conta1 );
		conta1.debito(20);
		System.out.println("conta1: Numero: " + conta1.getNumero() );
		System.out.println("conta1: Saldo: " + conta1.getSaldo() );
		System.out.println("conta1: Juros: " + conta1.getJuros() );
		System.out.println("conta1: Vencimento: " + conta1.getVencimento()
);
		System.out.println();

		java.util.Date vencimentoConta2 = new java.util.Date(99, 2, 30);
		Conta conta2 = new Conta(2, 200, 0.03, vencimentoConta2 );
		System.out.println("Ref. conta2: " + conta2 );
		conta2.debito(40);
		System.out.println("conta2: Numero: " + conta2.getNumero() );
		System.out.println("conta2: Saldo: " + conta2.getSaldo() );
		System.out.println("conta2: Juros: " + conta2.getJuros() );
		System.out.println("conta3: Vencimento: " + conta2.getVencimento()
);
	}
}
```

3.7. Definindo Constantes

Vimos anteriormente como definir variáveis. Uma constante, é uma variável especial, que uma vez inicializada não aceita ter seu valor alterado.

Em Java para se definir uma constante, usamos o modificar `final`, e no identificador usamos as letras todas maiúsculas seguindo a convenção de codificação.

Geralmente para uma Constante, definimos com modificador de acesso `public`, pois permite que acessemos tal valor a partir de qualquer classe **`static`**, que permite ser acessado diretamente pelo nome da classe sem haver uma instância do Objeto.

Temos a seguinte sintaxe:

```
[modificador acesso] static final tipo identificador = valor;
```

Exemplo:

```
public static final double PI =  3.141516;
```

As constantes são de grande valia dentro das técnicas de programação, pois permitem que o código fonte fique mais claro e legível quando testamos valores da aplicação com parâmetros de sistema, configuração ou de aplicação.

Exemplo de uso de constante:

```
// Endereco.java
public class Endereco {

	public static final int COMERCIAL = 0;
	public static final int RESIDENCIAL = 1;

	private int tipo;
	private String logradouro;

	public Endereco() {
		tipo = Endereco.COMERCIAL;
	}

	public Endereco(int tipoEnd, String logradouroEnd) {
		tipo = tipoEnd;
		logradouro = logradouroEnd;
	}

	public int getTipo() {
		return tipo;
	}

	public void setLogradouro(String logradouroEnd) {
		logradouro = logradouroEnd;
	}

	public String getLogradouro() {
		String local = "";
		switch( tipo ) {
			case Endereco.COMERCIAL:
				local += "COM: ";
				break;
			default:
				local += "RES: ";
		}
		local += logradouro;
		return local;
	}
}

// TesteEndereco.java
public class TesteEndereco {

	public static void main(String args[]) {
	Endereco end1 = new Endereco();
	end1.setLogradouro("Rua Universidade Java!");
	System.out.println( end1.getTipo() +" – "+ end1.getLogradouro() );

	Endereco end2 = new Endereco(Endereco.COMERCIAL, "Rua Digerati");
	System.out.println( end2.getTipo() +" – "+ end2.getLogradouro() );
	}
}
```

3.8. Usando a referência *this*.

Esta keyword do Java tem um significado e um propósito muito interessante, pois ela é a referência do próprio objeto. Ela representa a localização em memória de onde o objeto foi instanciado.

Usamos a referência **this** para indicar ao interpretador que o membro (atributo ou método) associado a ele faz parte da instância corrente.

Usamos o **this** também para diferenciar os atributos de instância dos parâmetros de métodos, fazendo com que possamos usar o mesmo nome em ambos e ter um código mais legível.

O uso do **this**, é muito comum dentro de métodos set (**mutator**) e de construtores.

O que é mais fácil de entender?

```
public void setLogradouro(String logradouroEnd) {
      // nomes diferentes para as variáveis
      logradouro = logradouroEnd;
}
```

Ou?

```
public void setLogradouro(String logradouro) {
// this.logradouro refere-se ao atributo de instancia
// chamado logradouro

// o identificador se refere a parâmetro de método logradouro.
      this.logradouro = logradouro;
}
```

Seguindo a conversão de código da JavaSoft, devemos fazer uso do **this**, principalmente nos métodos set. Nos métodos get não há relevância, pois o retorno refere-se á um tipo de instância, e o uso do **this** se faz desnecessário.

```
public int getTipo() {
      // tipo é um atributo de instância,
      // num método get não usamos colocar o this.
      return tipo;
}
```

Reescrevendo a classe Endereço

```
// Endereco.java
public class Endereco {

	public static final int COMERCIAL = 0;
	public static final int RESIDENCIAL = 1;

	private int tipo;
	private String logradouro;

	public Endereco() {
		this.tipo = Endereco.COMERCIAL;
	}

	public Endereco(int tipo, String logradouro) {
		this.tipo = tipo;
		// chamando o método abaixo desta instância
		this.setLogradouro( logradouro );
	}

	public void setLogradouro(String logradouro) {
		this.logradouro = logradouro;
	}

	public int getTipo() {
		return tipo;
	}

	public String getLogradouro() {
		String local = "";
		switch( this.tipo ) {
			case Endereco.COMERCIAL:
				local += "COM: ";
				break;
			default:
				local += "RES: ";
		}
		local += logradouro;
		return local;
	}
}
```

O uso do **this.** facilita a leitura de código por trazer o significado implícito de atributo de instância. O **this**. somente estará presente dentro de uma instância, ou seja, dentro de um método não-estático.

Podemos usar o **this** também para chamar construtores dentro da própria classe, permitindo um maior aproveitamento de código.

Sintaxe: **`this( [parametros] );`**

Exemplo:

```
public Endereco() {
      // procura um construtor na classe Endereco que possua
      // parametros int, String
      this( Endereco.COMERCIAL, ”” );
}
```

Reescrevendo a classe Endereço

```
// Endereco.java
public class Endereco {
      // constantes
      public static final int COMERCIAL = 0;
      public static final int RESIDENCIAL = 1;
      // atributos de instancia
      private int tipo;
      private String logradouro;
      // construtor
      public Endereco() {
            // procura um construtor na classe Endereco que possua
            // parametros int, String. Veja o construtor abaixo.
            this( Endereco.COMERCIAL, "");
      }
      // construtor
      public Endereco(int tipo, String logradouro) {
            this.tipo = tipo;
            // chamando o método abaixo desta instância
            this.setLogradouro( logradouro );
      }
      // mutator ( set )
      public void setLogradouro(String logradouro) {
            this.logradouro = logradouro;
      }
      // accessor( get )
      public int getTipo() {
            return tipo;
      }
      // accessor( get )
      public String getLogradouro() {
            String local = "";
            switch( this.tipo ) {
                  case Endereco.COMERCIAL:
                        local += "COM: ";
                        break;
                  default:
                        local += "RES: ";
            }
            local += logradouro;
            return local;
      }
}
```

3.9. Atributos e métodos de classe (static)

Vimos até agora conceitos voltados á instância de uma classe, que é uma área de memória, qual chamamos e objeto, que é capaz de armazenar dados e permitir a manipulação deles.

Entretanto, da mesma forma que podemos definir atributos para os objetos, podemos defini-los para as classes.

Definir um atributo de classe significa que ele estará definido fora de métodos e obrigatoriamente deve ter o modificador **`static`**.

A melhor maneira de inicializar atributos de classe, ou seja, **static,** é usar o bloco de inicialização estático:

```
static {
      // manipulação de atributos de classe, static.
}
```

Ou seja, é interessante usá-lo principalmente como variáveis de configuração que devem ser inicializadas quando carregamos a aplicação.

Exemplo de uso:

Se no modelo do nosso restaurante, queremos ter uma taxa básica de serviço, que geralmente é de 10%, poderíamos defini-la como sendo uma variável de uma classe de configuração.

```
public class RestauranteConfig {

      private static double taxaServico;

      static {
            taxaServico = 10.0;
      }
}
```

Seguindo a idéia de encapsulamento, devemos escrever métodos para manipular esses atributos de classe (**static**). Um método que vai manipular esse tipo de atributo pode ser de classe (**static**) ou de instancia (**non-static**). Damos preferência para os métodos de classe (**static**)

Exemplo de uso:

```
public class RestauranteConfig {

	private static double taxaServico;

	static {
		taxaServico = 10.0;
	}

	public static double getTaxaServico() {
	//note que não usamos o this,
		// e sim o nome da classe como referência
		// para os atributos de classe.
	return RestauranteConfig.taxaServico;
	}

	public static void setTaxaServico(double taxaServico) {
		//note que não usamos o this,
		// e sim o nome da classe como referência
		// para os atributos de classe.
	RestauranteConfig.taxaServico = taxaServico;
	}
}

public class TesteRestauranteConfig {

	public static void main(String arg[]) {
		System.out.println("Taxa de serviço: "+
RestauranteConfig.getTaxaServico() );
		RestauranteConfig.setTaxaServico( 11.0 );
		System.out.println("Nova Taxa de serviço: "+
RestauranteConfig.getTaxaServico() );
	}

}
```

Temos de tomar cuidados especiais com atributo de classe, pois ele é um endereço único, para todos os objetos provenientes daquela classe.

Alterando seu valor, será refletido em todas as instâncias, por isso é usado para configuração ou parâmetros globais de aplicação.

Discussão

Quando falamos de encapsulamento, entende-se por esconder somente os detalhes de implementação?

Resumidamente o que uma classe Java contém?

Quando definimos métodos, estamos definindo como os objetos farão suas atividades, ou seja, como funcionarão?

Os construtores são considerados métodos?

Finalmente, qual a diferença entre ser um membro de instância ou um membro de classe?

Exercícios

4. Classes e Objetos em Java

A linguagem Java é uma linguagem de alto nível e multiparadigma, ou seja, permite que sejam utilizados mais de um paradigma de programação. Além da orientação a objetos, a linguagem Java permite a programação estruturada, pois o que determina a utilização de um paradigma é o critério utilizado para dividir os módulos e para definir a função de cada módulo[1].

Neste capítulo, serão apresentados o ambiente de programação e algumas noções básicas necessárias para programar em Java de forma orientada a objetos.

4.1. Alguns ambientes de programação da linguagem Java

A linguagem Java é compilada e interpretada: a compilação (comando **`javac`**) gera o *bytecode* (o arquivo **`.class`**) e o interpretador (comando **`java`**) executa o programa. Esta forma de execução possibilita a programação multiplataforma: o mesmo arquivo **`.class`** pode ser usado no *Windows* e no *Linux* utilizando a implementação do interpretador nativa para cada sistema operacional.

Para programar em Java, os seguintes softwares são necessários:

J2SE (Java 2 *Platform Standard Edition*): permite compilar e executar programas em Java.
Editor de texto simples: pode ser o *Notepad* ou outro editor de texto simples.

Neste ambiente, o código-fonte é editado no *Notepad*. No *prompt* do DOS, o comando **`javac`** deve ser chamado para gerar o arquivo **`.class`**. Então para executar o programa, chamamos o comando **`java`** e nome da classe que possui o método **`public static void main(String args[])`**. Na próxima seção veremos um exemplo completo de programação.

Existem ambientes mais sofisticados de desenvolvimento que ajudam no processo de construção, organização e depuração do código. Alguns exemplos:

- *NetBeans*: fornecido gratuitamente pela *Sun*, possui suporte a desenho de interface gráfica e permite a depuração do programa passo a passo.
- *JBuilder*: a versão 2005 *Foundation* é fornecida gratuitamente pela *Borland*. Possui também suporte a desenho de interface gráfica e depuração passo a passo.
- *Eclipse*: ambiente de programação de código aberto e gratuito, este ambiente tem suporte a desenho de interface gráfica e depuração passo a passo, tanto em Java como em outras linguagens como C e C++.
- *JCreator*: a versão *Lite* é gratuita, porém não possui suporte nem a desenho de *interface* nem a depuração passo a passo. A grande vantagem deste ambiente é que pode ser executado em qualquer configuração de hardware, pois não exige memória quanto aos ambientes mais sofisticados.
- *BlueJ*: é um ambiente programação gratuito em Java desenvolvido para facilitar o aprendizado da linguagem. Permite criar objetos interativamente, chamar seus métodos e verificar seu

[1] Na disciplina Lógica de Programação é ensinada a programação em Java usando o paradigma estruturado

estado. Não possui suporte à depuração passo a passo nem desenho de interfaces gráficas. Como *JCreator*, não exige muita memória para ser executado.

4.2. Tradução de uma Classe em UML para o esqueleto em Java

A tradução de uma classe representada em UML para o esqueleto em Java (métodos sem código) é um processo mecânico. Veja a figura abaixo:

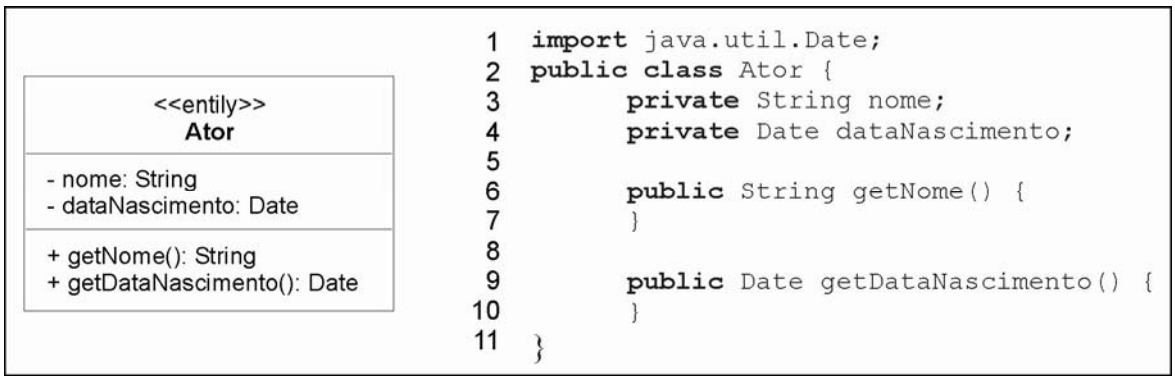


Figura 17.

Na tabela abaixo está a correspondência entre os símbolos do UML e da linguagem Java:

UML	Java
Nome da classe	`Public class Nome DaClasse{ }`
–	`private`
+	`public`
– atributo: tipo	`private tipo atributo;`
+ metodo (parametro 1; tipo 1, parametro 2: tipo 2): tipo	`public tipo metodo (tipo 1 parametrol, tipo 2 parametro 2) { }`

Nome da classe
– atributo 1: tipo 1
– atributo 2: tipo 2
+ metodo( ): void
+ metodo1 (parametro1 : tipo1): void
+ metodo2 (parametro1: tipo1, parametro2: tipo2): tipo3
	`public class` NomeDaClasse{
	`private` tipo1 atributo1;
	`private` tipo 2 atributo2;
	
	`public void` metodo ( ) { }
	`public void` metodo1 (tipo1 parametro1) { }
	`public` tipo3 metodo2 (tipo1 parametro1, tipo2 parametro2) { }
	}

Tabela 1.

Observe que:

1. O nome de uma classe, método ou atributos em Java, não pode conter espaços, acentos.
2. As 52 palavras reservadas do Java são iniciadas com letra minúscula. A lista de todas as palavras reservadas da linguagem é: **`abstract, boolean, break, byte, case, cath, char, class, const, continue, default, do, double, else, extends, final, finally, float, for, function, goto, if, implements, import, in, instanceof, int, interface, long, native, new, null, package, private, protected, public, return, short, sinchronized, static, super, switch, this, throw, throws, transient, true, try, var, void, while, with`**. Estas palavras não podem ser usadas para designar nomes de classes, de métodos, de atributos nem de variáveis.
3. Muitas empresas têm padrões de codificação. Neste curso, o padrão de codificação é iniciar nomes de classe com letra maiúscula, nome de atributos, métodos e variáveis com letra minúscula. Caso o nome de uma classe ou de um método seja composto, as palavras seguintes são justapostas e iniciadas com letra maiúscula. Exemplo: uma classe para representar uma conta corrente deve se chamar **ContaCorrente.**

4.3. Construtor Padrão

Uma classe é apenas a descrição de um objeto. Observe que os atributos e métodos dentro de uma classe não são ordenados. Para utilizar um objeto é necessário instanciá-lo. Para instanciar um objeto é usado um método construtor: cria um espaço de memória para o objeto. Cada classe tem seu próprio método construtor. Cada linguagem de programação orientada a objetos possui uma sintaxe diferente para chamar e definir o construtor.

Em Java, a escrita do método construtor na classe é opcional, pois a linguagem provê um construtor padrão. O construtor padrão tem o mesmo nome da classe e não possui parâmetros. A sintaxe para construir um objeto é a seguinte:

```
NomeDaClasse nomeDoObjeto=new NomeDaClasse();
```

O comando acima declara o objeto **`nomeDoObjeto (NomeDaClasse nomeDoObjeto)`** e instancia-o **`(nomeDoObjeto=new NomeDaClasse();)`**. Estes comandos podem ser separados em duas linhas de comando: a declaração e a instanciação, como mostrado abaixo.

```
NomeDaClasse nomeDoObjeto;
nomeDoObjeto=new NomeDaClasse();
```

Para instanciar um objeto da classe da seção 5.2, Ator, pode-se utilizar o comando abaixo:

```
Ator ator1=new Ator();
```

Com este comando é criado um objeto **`ator1`** sem que nenhum valor seja dado aos seus atributos.

4.4. Métodos Acessores, Modificadores e Construtores

Como foi explicado no capítulo 4, não é recomendável permitir que os atributos sejam alterados por objetos de outras classes. A primeira razão para esta recomendação é a violação do princípio de coesão, pois a classe que modifica os atributos de outra classe estaria fazendo uma tarefa que não lhe compete. A segunda razão para isto é evitar o aumento do acoplamento entre as duas classes já que uma classe que modifica o valor de atributos de outra está mais acoplada com ela do que se estivesse apenas chamando um método.

Como dar valores para os atributos de um objeto se não podemos acessar diretamente os atributos deste objeto?

Existem duas formas:

1. **Métodos modificadores**: os métodos modificadores recebem como parâmetro um valor e o atribui ao atributo. Para dar nome a estes métodos existe também uma convenção, tais como os métodos get. A regra de formação para o nome de um método modificador: juntar prefixo set mais o nome do atributo iniciado com letra maiúscula. O método deve receber como parâmetro uma variável que possua o mesmo tipo do atributo que este método vai modificar. Exemplo:

<<entity>> **Ator**
- nome: String - dataNascimento: Date
+ getNome(): String + getDataNascimento(): Date + setNome(n: String): void + setDataNascimento(dn: Date): void

2. **Métodos construtores criados pelo programador**: o construtor padrão somente cria o espaço de memória, mas não atribui nenhum valor válido aos atributos. O programador pode declarar um construtor (ou mais construtores) para inicializar os valores dos atributos. Exemplo:

<<entity>> **Ator**
- nome: String - dataNascimento: Date
+ getNome(): String + getDataNascimento(): Date + Ator(n: String, dn: Date)

Observe que o construtor é um método que possui o mesmo nome da classe e não devolve nenhum valor. Além disso, recebe como parâmetros variáveis com o mesmo tipo dos atributos.

4.4.1. Implementação em Java dos Métodos Acessores, Modificadores e Construtores

A implementação dos métodos acessores, modificadores e construtores é muito simples e quase sempre repetitiva.

a. Implementação dos métodos acessores

```
1  import java.util.Date;
2  public class Ator {
3      private String nome;
4      private Date dataNascimento;
5
6      public String getNome() {
7              return nome;
8      }
9
10     public Date getDataNascimento() {
11             return dataNascimento;
12     }
13 }
```

Na linha 7, observe a linha de comando `return nome`. A palavra *return* é uma palavra reservada da linguagem Java e significa devolver o valor da variável que a segue, neste caso, o atributo nome. Logo o método *getNome* devolve o valor do atributo nome de um objeto do tipo Ator. A palavra *return* quando seguida de uma variável precisa da declaração na assinatura do método do tipo da variável que será devolvida. Neste exemplo, o tipo de devolução declarado no método *getNome* é *String*, pois a variável que será devolvida (nome) também é do tipo *String*. Para o método *getDataNascimento* o tipo de devolução declarado é *Date*, pois a variável que será devolvida (dataNascimento) também é do tipo *Date*.

O comando `return` tem outro efeito: se for chamado no meio do método, provoca a saída do método. Ou seja, todos os comandos colocados depois do *return* não serão executados. Quando um método é declarado como *void* pode-se utilizar o *return* sozinho (não seguido por variável) para interromper a execução do método e voltar ao ponto do programa onde o método foi chamado.

b. Implementação dos métodos modificadores

```
1  import java.util.Date;
2  public class Ator {
3      private String nome;
4      private Date dataNascimento;
5
6      public String getNome() {
7              return nome;
8      }
9
10     public Date getDataNascimento() {
11             return dataNascimento;
12     }
13     public void setNome(String n) {
14             nome=n;
15     }
16     public void setDataNascimento(Date dn) {
17             dataNascimento=dn;
18 }
19 }
```

Na linha 14, observe a linha de comando nome=n. A variável n é recebida como parâmetro pelo método e seu valor é passado para o atributo nome. Assim, o método setNome modifica o valor atual do atributo nome de um objeto do tipo Ator. Observe que o tipo da variável recebida tem de ser igual ao da variável que será modificada. Para o método setDataNascimento o tipo do parâmetro declarado é Date, pois o atributo que será modificado (dataNascimento) também é do tipo Date.

c. Implementação dos métodos construtores

```
1  import java.util.Date;
2  public class Ator {
3      private String nome;
4      private Date dataNascimento;
5
6  public Ator(String n, Date dn){
7      nome=n;
8      dataNascimento=dn;
9  }
10     public String getNome() {
11             return nome;
12     }
13
14     public Date getDataNascimento() {
15             return dataNascimento;
16     }
17     public void setNome(String n) {
18             nome=n;
```

```
19      }
20      public void setDataNascimento(Date dn) {
21            dataNascimento=dn;
22      }
23  }
```

Nas linhas 6 a 9 está definido o método construtor. Por ser um método especial da linguagem Java, não precisa de tipo de devolução (por exemplo, *void* ou outro como *String*, etc). As variáveis **n** e **dn** são recebidas como parâmetros pelo método e seus valores são passados a todos os atributos.

Como o método construtor é chamado na criação do objeto, os valores passados ao construtor são os primeiros que os atributos vão assumir. Depois do objeto criado, um método set pode ser chamado para modificar valor atual de um determinado atributo.

4.5. Primeiro Programa Orientado a Objetos em Java

Até este ponto, foi apresentado como as classes em UML são traduzidas em *Java*. No entanto, não foi mostrado ainda como utilizá-las. Este será o assunto desta seção.

Um programa orientado a objeto é formado por um conjunto de objetos que interagem entre si. Interagir significa que cada objeto executa uma tarefa a pedido de outro objeto. Este pedido para execução de tarefa é chamado mensagem. Cada objeto tem um conjunto predeterminado de tarefas que pode executar, ou seja, de mensagens que pode receber. As tarefas que cada objeto pode executar são os métodos definidos nas classes.

Por exemplo, seja a classe Ator definida nas seções anteriores.

```
1   import java.util.Date;
2   public class Ator {
3       private String nome;
4       private Date dataNascimento;
5
6   public Ator(String n, Date dn){
7       nome=n;
8       dataNascimento=dn;
9   }
10      public String getNome() {
11            return nome;
12      }
13
14      public Date getDataNascimento() {
15            return dataNascimento;
16      }
17      public void setNome(String n) {
18            nome=n;
19      }
20      public void setDataNascimento(Date dn) {
21            dataNascimento=dn;
22      }
23  }
```

Um objeto do tipo Ator pode receber 4 mensagens: *getNome*, *getDataNascimento*, *setNome* e *setDataNascimento*. Além destas, o objeto pode receber a mensagem especial de criação, através da chamada do método construtor.

```
1  import java.util.Date;
2  public class ProgramaUsaAtor
3  {
4  public static void main(String args[])
5  {
6      //declaracao do nomes dos objetos
7      Ator fabio;
8      Date niverFabio;
9      Ator ana;
10     Date niverAna;
11
12     //Instanciacao dos objetos do tipo Date
13     // ano - o ano menos 1900.
14     // mes - o mes 0-11.
15     // dia - o dia do mes entre 1-31.
16     niverFabio=new Date(74, 7, 10);
17     fabio=new Ator(“Fabio Assuncao”, niverFabio);
18     niverAna=new Date(76, 6, 16);
19     ana=new Ator(“Ana Paula Arosio”, niverAna);
20
21     // Imprimir os dados
22     System.out.println(“Nome:”+fabio.getNome());
23  System.out.println(“Aniversario:”+fabio.getDataNascimento().toString
24 ());
25     System.out.println(“Nome: ”+ana.getNome());
26     System.out.println(“Aniversario:
27 ”+ana.getDataNascimento().toString());
       }
   }
```

Agora analise o programa:

1. Linha 2: o programa é definido dentro de uma outra classe: *ProgramaUsaAtor* e deve ser salvo em um arquivo *ProgramaUsaAtor.java*.
2. Linha 4: os comandos que o programa deve executar estão definidos dentro do método *main*: *public static void main(String args[]).*
3. Linhas 7, 8, 9 e 10: os objetos que serão usados devem ser declarados, da mesma forma que uma variável. A novidade é que o tipo da variável foi definido por você na classe *Ator*.
4. Linhas 7, 8, 9 e 10: os objetos que serão usados devem ser declarados, da mesma forma que uma variável. A novidade é que o tipo da variável foi definido por você na classe *Ator*.
5. Linhas 16 e 18: instanciação dos objetos tipo *Date*. O tipo *Date* é definido pela linguagem Java dentro da biblioteca *java.util.* Para utilizar esta biblioteca é necessário importá-la com a linha de comando `import` (linha1). Nas linhas 13, 14 e 15 estão comentadas as instruções para passar os parâmetros para o construtor da classe *Date*.

6. Linhas 17 e 19: instanciação dos objetos tipo *Ator*. O tipo Ator foi definido na classe *Ator* salva dentro de um arquivo chamado *Ator.java*. Se o arquivo está dentro do mesmo diretório, não é necessário utilizar o comando `import`. Veja que a ordem dos parâmetros passados ao construtor deve corresponder exatamente à ordem estipulada dentro da classe *Ator*, ou seja, primeiro uma variável do *String* e depois um objeto do tipo Date.
7. Linha 22: para imprimir o valor do nome do objeto fabio é necessário passar ao objeto uma mensagem *getNome* para recuperar o nome. Para passar uma mensagem para um objeto em *Java* segue-se o esquema abaixo:

   ```
   nome_do_objeto.nome_metodo(parâmetros)
   ```

   Onde *nome_do_objeto* é o identificador do objeto, o nome com o qual foi instanciado. O *nome_metodo* só pode ser algum método definido na classe-tipo do objeto. Se na definição do método foi declarado o recebimento de parâmetros, então é obrigatória a passagem destes entre parêntesis na mesma ordem definida na classe. Caso não haja parâmetros os parêntesis devem ser colocados vazios, como na chamada de *getNome*.
8. Linha 26: para imprimir o valor do nome do objeto ana é necessário passar ao objeto uma mensagem *getNome* para recuperar o nome. Observe que a definição do método é única, mas o resultado do método depende de para qual objeto se envia a mensagem. O atributo nome do objeto fabio foi inicializado com o valor “Fábio Assunção”. Já o atributo nome do objeto ana foi inicializado com o valor “Ana Paula Arosio”. Logo o resultado do método *getNome* é “Fábio Assunção” para o objeto fabio e “Ana Paula Arosio” para o objeto ana.
9. Linha 23 e 25: para imprimir o valor da data de nascimento dos objeto fabio e ana é necessário passar aos objetos a mensagem *getDataNascimento* para recuperar os respectivos atributos *dataNascimento.* Mas como *dataNascimento* também é um objeto, é preciso convertê-lo para *String* para que possa ser impresso. Daí a chamada do método *toString*, definido na classe Date[2]. A chamada do método *toString()* foi aninhada com a chamada do método *getDataNascimento().* Um outro exemplo de chamada de métodos aninhada:

   ```
   objeto1.metodo1().metodo2().metodo3();
   ```

   onde o metodo1 do objeto1 devolve um objeto que possui o método2 e que, por sua vez, devolve um outro objeto que possui o metodo3. Isto é equivalente a:

   ```
   Tipo2 objeto2=objeto1.metodo1();
   Tipo3 objeto3=objeto2.metodo2();
   objeto3.metodo3();
   ```

Figura 18.

[2] Para saber mais sobre a classe Date visite a página: http://java.sun.com/j2se/1.5.0/docs/api/java/util/Date.html (acessaem 16/06/2005)

Da mesma forma, pode-se separar a chamada do método getDataNascimento de Ator da chamada de toString() de Date da linha 23:

```
Date df=fabio.getDataNascimento();
System.out.println("Aniversario:"+df.toString());
```

Exercícios

01. Para as classes em UML abaixo faça:

a. Para cada atributo, defina os métodos acessores e modificadores.
b. Defina um método construtor para inicializar todos os atributos.
c. Passe todos as classes para o Java.
d. Crie uma classe contendo método main para cada uma das classes que inicialize os objetos e mostre os valores dados.

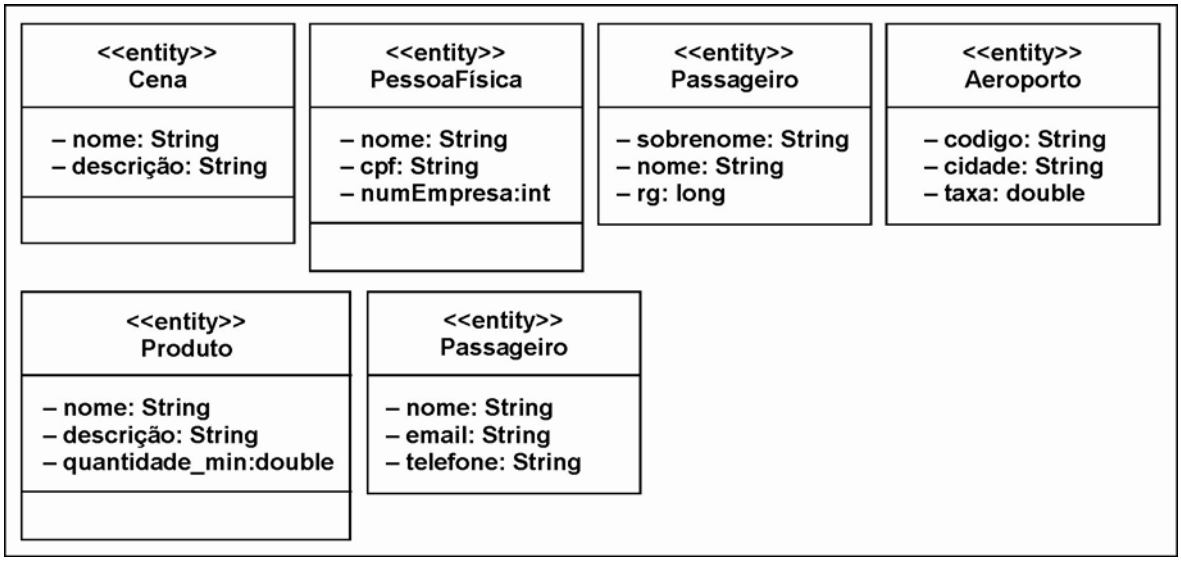


Figura 19.

Bibliografia

BARNES, DAVID J. Programação Orientada a Objetos em Java. Prentice Hall. 2000.

BEZERRA, Eduardo. Princípios de análise e projeto de sistemas com UML. Editora Campus. Rio de Janeiro 2002.

DEITEL, H. M.; DEITEL, P.J. Java – Como Programar. 4ª edição, Porto Alegre, RS, Editora Bookman, 2002

LARMAN, Craig. Utilizando UML e Padrões. Bookman Companhia Editora, 2ª. Edição. Porto Alegre, 2004.

FOWLER, Martin. UML Distilled. 3a. Edição. Addison-Wesley, 2003.

Leitura Complementar. As 52 Palavras Reservadas do Java

Por Vanessa Sabino

Disponível em `http://www.linhadecodigo.com.br/artigos.asp?id_ac=83` (acesso em 16/06/2005)

Modificadores de acesso

private: acesso apenas dentro da classe ;

protected: acesso por classes no mesmo pacote e subclasses ;

public: acesso de qualquer classe.

Modificadores de classes, variáveis ou métodos

abstract*:* classe que não pode ser instanciada ou método que precisa ser implementado por uma subclasse não abstrata;

class: especifica uma classe;

extends: indica a superclasse que a subclasse está estendendo;

final: impossibilita que uma classe seja estendida, que um método seja sobrescrito ou que uma variável seja reinicializada;

implements: indica as interfaces que uma classe irá implementar

interface: especifica uma *interface*;

native: indica que um método está escrito em uma linguagem dependente de plataforma, como o C;

new: instancia um novo objeto, chamando seu construtor;

static: faz um método ou variável pertencer à classe ao invés de às instâncias;

strictfp: usado em frente a um método ou classe para indicar que os números de ponto flutuante seguirão as regras de ponto flutuante em todas as expressões;

synchronized: indica que um método só pode ser acessado por uma *thread* de cada vez;

transient: impede a serialização de campos;

volatile: indica que uma variável pode ser alterada durante o uso de *threads*.

Controle de fluxo dentro de um bloco de código

break: sai do bloco de código em que ele está;

case: executa um bloco de código dependendo do teste do *switch*;

continue*:* pula a execução do código que viria após essa linha e vai para a próxima passagem do *loop*;

default: executa esse bloco de código caso nenhum dos teste de *switch-case* seja verdadeiro;

do: executa um bloco de código uma vez, e então realiza um teste em conjunto com o *while* para determinar se o bloco deverá ser executado novamente;

else: executa um bloco de código alternativo caso o teste *if* seja falso;

for: usado para realizar um *loop* condicional de um bloco de código;
if: usado para realizar um teste lógico de verdadeiro o falso;
instanceof: determina se um objeto é uma instância de determinada classe, superclasse ou interface;
return: retorna de um método sem executar qualquer código que venha depois desta linha (também pode retornar uma variável);
switch: indica a variável a ser comparada nas expressões case;
while: executa um bloco de código repetidamente até que uma certa condição seja verdadeira.

Tratamento de erros

assert: testa uma expressão condicional para verificar uma suposição do programador;
catch: declara o bloco de código usado para tratar uma exceção;
finally: bloco de código, após um *try-catch*, que é executado independentemente do fluxo de programa seguido ao lidar com uma exceção;
throw: usado para passar uma exceção para o método que o chamou;
throws: indica que um método pode passar uma exceção para o método que o chamou;
try: bloco de código que tentará ser executado, mas que pode causar uma exceção.

Controle de pacotes

import: importa pacotes ou classes para dentro do código;
package: especifica a que pacote todas as classes de um arquivo pertencem.

Primitivos

boolean: um valor indicando verdadeiro ou falso;
byte: um inteiro de 8 bits (signed);
char: um carácter unicode (16-bit unsigned);
double: um número de ponto flutuante de 64 bits (signed);
float: um número de ponto flutuante de 32 bits (signed);
int: um inteiro de 32 bits (signed);
long: um inteiro de 64 bits (signed);
short: um inteiro de 32 bits (signed).

Variáveis de referência

super: refere-se a superclasse imediata;
this: refere-se a instância atual do objeto.

Retorno de um método

void: indica que o método não tem retorno.

Palavras reservadas não utilizadas

const: não utilize para declarar constantes; use `public static` final;

goto: não implementada na linguagem Java por ser considerada prejudicial.

Literais reservados

De acordo com a Java Language Specification, *null*, *true* e *false* são tecnicamente chamados de valores literais, e não *keywords*. Se você tentar criar algum identificador com estes valores, você também terá um erro de compilação.

5. Diagramas de Interação

Foi visto anteriormente. que um programa orientado a objetos é composto de objetos que interagem entre si. Neste capítulo, será mostrado como representar a interação entre os objetos. Também será apresentado como encontrar os métodos que não são nem modificadores, nem acessores e nem construtores.

5.1. Modelo de Interação

Os objetos interagem entre si, passando mensagens uns para os outros. Interação é um comportamento que compreende um conjunto de mensagens trocadas entre um conjunto de objetos dentro de um contexto para a execução de um determinado propósito.

Para ativar um comportamento em um objeto, isto é, fazê-lo executar uma de suas operações, deve-se enviar-lhe a mensagem apropriada. Um objeto pode receber mensagens tanto do usuário (como no caso dos objetos visuais) quanto de outro objeto.

Se um objeto não possuir toda a informação necessária para executar uma operação, pode ser necessário passar-lhe as informações faltantes na forma de parâmetros. O objeto pode, também, solicitar a colaboração de outro objeto, enviando-lhe por sua vez uma mensagem.

O modelo de interação representa o aspecto dinâmico das colaborações entre os objetos, mostrando como os objetos trocam mensagens entre si para chegar a um determinado objetivo.

Existem duas formas de representação:

- Enfatizando a colaboração entre os objetos: diagrama de colaboração.
- Enfatizando a seqüência das mensagens trocadas entre os objetos de uma maneira estruturada: diagrama de seqüência.

Os dois tipos de diagramas possuem o mesmo conteúdo, porém com visões diferentes - na verdade, um tipo de diagrama pode ser facilmente convertido no outro.

Qualquer diagrama de interação precisa do diagrama de classes para estar completo.

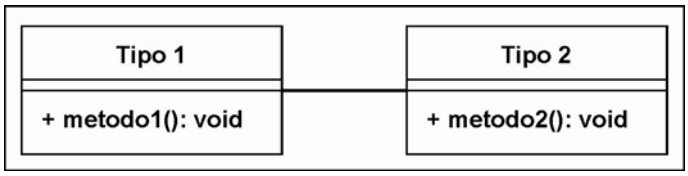


Figura 20.

5.1.1. Diagrama de Colaboração

O diagrama de colaboração enfatiza a organização dos objetos que participam numa interação. O diagrama de colaboração tende a ter um formato mais livre e flexível quanto à maneira de ser projetado.

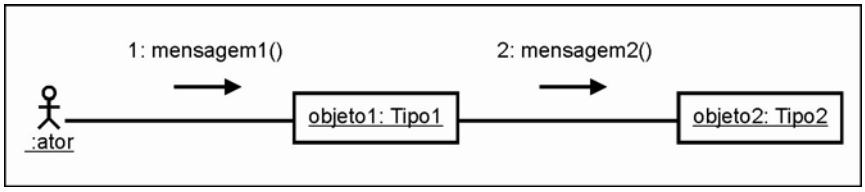


Figura 21.

Os elementos do diagrama de colaboração são:

- **Objeto**: os objetos que irão colaborar entre si para realizar determinada tarefa.
- **Mensagem**: é o nome do método invocado pelo usuário (ação do usuário) ou por outra classe.
- **Ator**: é o usuário, aquele que inicia a tarefa.
- **Seta**: a seta aponta sempre para o objeto que possui o método invocado.

O diagrama de colaboração:

Permite visualizar as classes colaboradoras e as mensagens que serão trocadas para uma tarefa. Possui formato mais livre e flexível.

5.1.2. Diagrama de Seqüência

Um diagrama de seqüência mostra a seqüência de troca de mensagens, mas não inclui relacionamentos entre os objetos. Ambos expressam informações semelhantes; o que muda é a forma como elas são mostradas. Os diagramas de seqüência mostram a seqüência explícita de mensagens e são melhores quando é importante visualizar a ordenação temporal das mensagens.

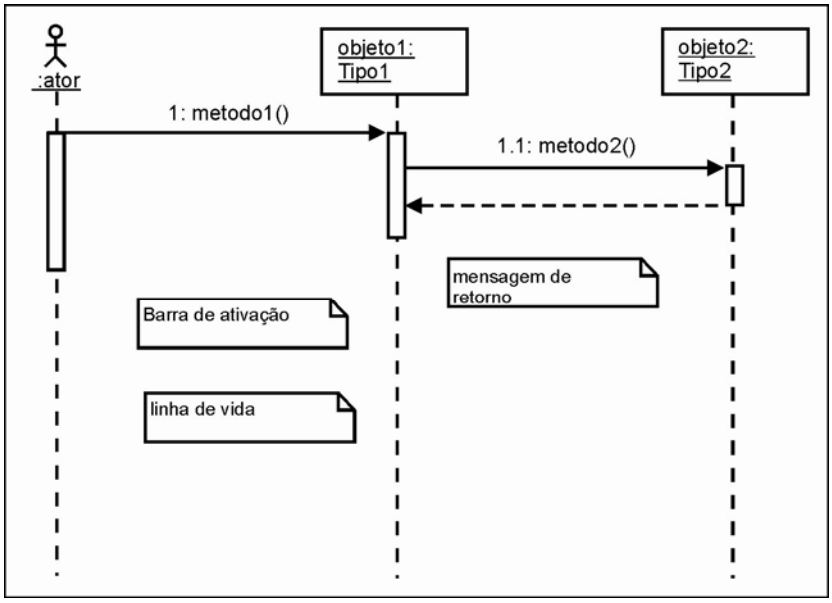


Figura 22.

Em diagramas de seqüência pode ter objetos e instâncias de ator, juntamente com mensagens que descrevem como eles interagem. O diagrama descreve o que ocorre nos objetos participantes, em termos de ativações, e como os objetos se comunicam enviando mensagens uns aos outros.

Objetos

Um objeto é mostrado sobre uma linha tracejada vertical denominada “linha de vida”. A linha de vida representa a existência do objeto em um momento específico. Um símbolo de objeto é desenhado no alto da linha de vida e mostra o nome do objeto e sua classe sublinhada e separada por dois-pontos:

```
nomeObjeto: Tipo
```

Você pode usar objetos em diagramas de seqüência das seguintes formas:

- Uma linha de vida pode representar um objeto.
- Os objetos podem não ter nome, mas é recomendável nomeá-los se você quiser diferenciar os diversos objetos da mesma classe.
- Várias linhas de vida no mesmo diagrama podem representar objetos diferentes da mesma classe; mas, como foi dito anteriormente, os objetos devem ser nomeados para que você possa estabelecer a diferença entre os dois objetos.

Atores

Geralmente, uma instância de ator é representada pela primeira (mais à esquerda) linha de vida no diagrama de seqüência, como o iniciador da interação.

Mensagens

Uma mensagem é uma comunicação entre objetos que leva informações e espera a execução de uma tarefa; nos diagramas de seqüência, uma mensagem é mostrada como uma seta sólida horizontal partindo da linha de vida de um objeto para a linha de vida de outro objeto.

No caso de uma mensagem de um objeto para si mesmo, a seta pode iniciar e terminar na mesma linha de vida. Como no exemplo abaixo, usando a classe Tipo1 definida a seguir:

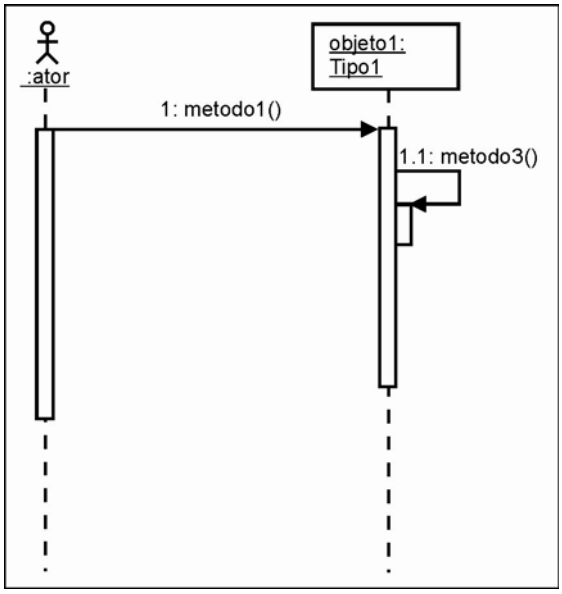


Figura 23.

A seta é rotulada com o nome da mensagem e seus parâmetros. Ela também pode ser rotulada com um número que indique a seqüência da mensagem no processo geral de interação. Os números seqüenciais em geral são omitidos em diagramas de seqüência, nos quais a localização física da seta mostra a seqüência relativa.

Uma mensagem pode ser não-atribuída, o que significa que seu nome é uma seqüência de caracteres temporária que descreve o sentido geral da mensagem e não é o nome de uma operação do objeto recebedor. Mais tarde, você poderá atribuir a mensagem especificando a operação do objeto de destino da mensagem. A operação especificada substituirá então o nome da mensagem.

5.1.3. Quando um objeto pode enviar uma mensagem para outro?

Para um objeto se comunicar com outro é necessário que um tenha acesso ao outro. Existem três formas de visibilidade entre objetos.

- **Visibilidade por associação entre as classes**: as duas classes são ligadas por associação.

  Exemplo:

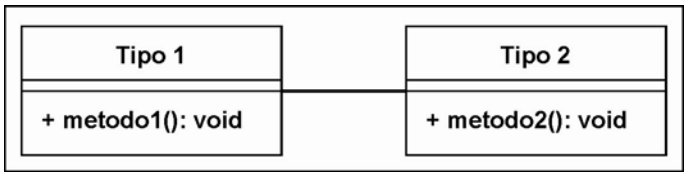


Figura 24.

Assim o objeto do Tipo1 pode enviar um objeto do Tipo2:

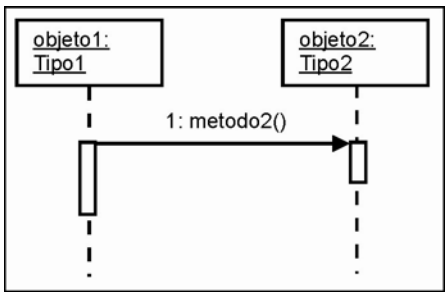


Figura 25.

- **Visibilidade por parâmetro**: um método recebe um objeto como parâmetro.

Exemplo:

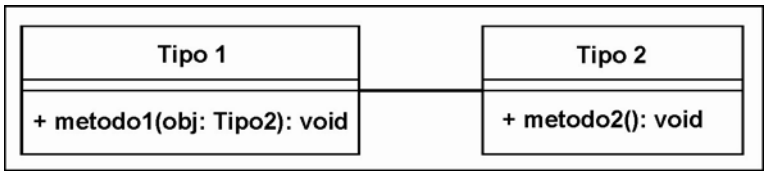


Figura 26.

Assim o metodo1 pode chamar um método de obj.

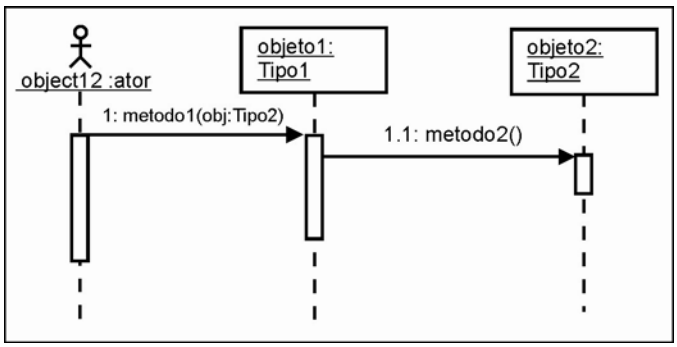


Figura 27.

- **Visibilidade local:** O objeto é criado localmente (dentro do método).

Exemplo:

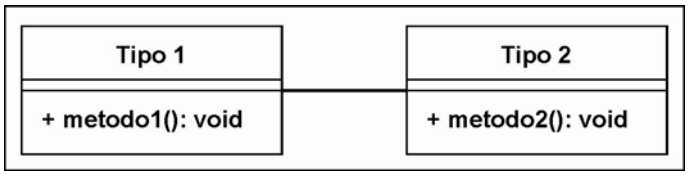


Figura 28.

O metodo1 cria localmente um instância de Tipo2. A mensagem <create> corresponde a chamada do construtor de Tipo2.

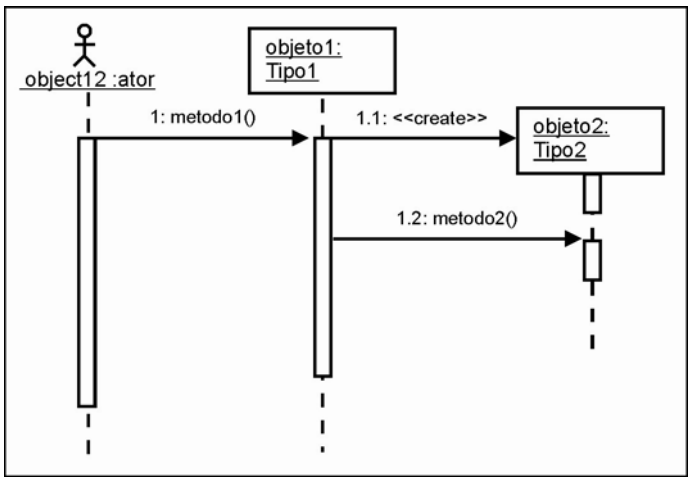


Figura 29.

5.2. Criando o Modelo de Interação

Para criar o modelo de interação é necessário descobrir a responsabilidade de cada objeto dentro da solução do problema. Neste curso, o problema a ser resolvido é a realização de um caso de uso, ou seja, a modelagem e implementação de um caso de uso.

Seja o caso de uso abaixo:

[UC1] Calcular média bimestral

Descrição: Calcular a média bimestral de um estudante do IBTA dadas suas duas notas mensais, a média de prática e nota da prova bimestral para efeito de consulta.

Precondições: N/A

Ator principal: Qualquer usuário do sistema (aluno, professor ou secretária)

Fluxo Principal

1. O usuário escolhe a opção “Calcular média bimestral”.
2. O sistema solicita a nota mensal 1.
3. O usuário fornece a nota mensal 1.
4. O sistema solicita a nota mensal 2.
5. O usuário fornece a nota mensal 2.
6. O sistema calcula a média mensal.
7. O sistema exibe a média mensal e solicita a nota da prova.
8. O usuário fornece a nota da prova.
9. O sistema calcula a média de teoria do aluno.
10. O sistema exibe a nota de teoria da disciplina.
11. O sistema solicita nota da parte prática da disciplina.
12. O usuário fornece a média de prática.
13. O sistema calcula e exibe a média bimestral do aluno.

Fluxos alternativos

N/A

Pós-condição: N/A

Passo 1: Encontrar classes.

Classes de fronteira: Pensando em formulários, somente uma classe de fronteira poderia ser definida. No entanto, neste curso não será apresentado a biblioteca de construção de interfaces gráficas da linguagem Java. Serão utilizados os seguintes métodos da classe *JOptionPan*e da biblioteca *javax.swing*:

- Para entrada de dados o método showInputDialog. Pode ser usado como no exemplo abaixo:

```
String nome=JOptionPane.showInputDialog(null,"Digite o nome do usuário");
```

A execução do comando acima resulta na seguinte janela:

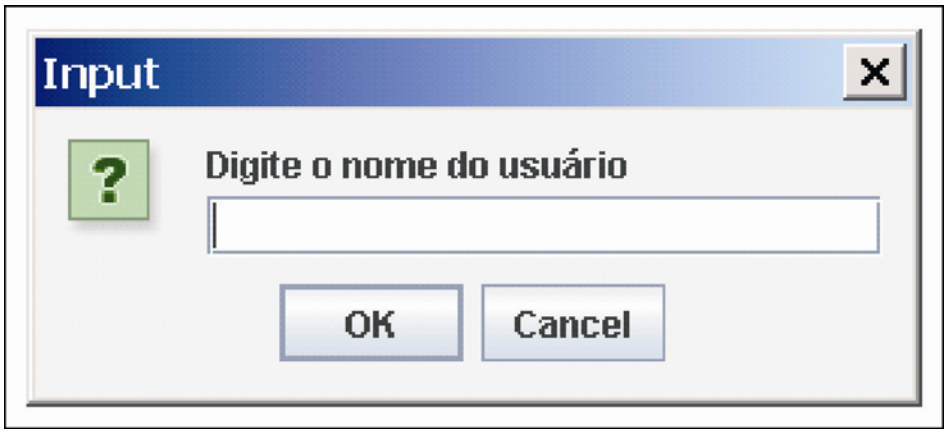


Figura 30.

O valor digitado é devolvido e deve ser armazenado em uma variável do tipo String.

- Para saída de dados o método showMessageDialog. Pode ser usado como no exemplo abaixo:

```
JOptionPane.showMessageDialog(null,"Usuário cadastrado com sucesso!“);
```

A execução do comando acima resulta na seguinte janela:

Figura 31.

Assim para cada entrada de dados será chamado o método showInputDialog e para cada saída o método chamado será showMessageDialog.

Cada vez que o sistema solicita uma nota, uma janela showInputDialog será exibida. Será feita uma correspondência entre as classes de fronteira e as chamadas de janelas JOptionPane3.

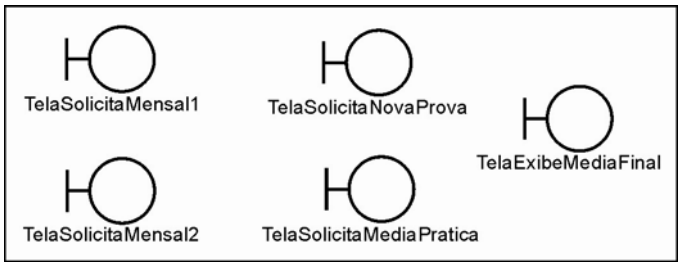


Figura 32.

Classe de controle: Para cada caso de uso tem-se uma classe de controle com o mesmo nome do caso de uso.

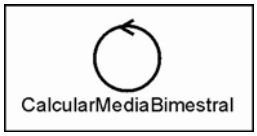


Figura 33.

Classe de entidade: Para cada caso de uso tem-se uma classe de controle com o mesmo nome do caso de uso. Neste caso de uso, os únicos dados tratados são notas. Pode-se categorizar estas notas em duas categorias: práticas e teóricas. Além disso, as duas categorias juntas compõem a média bimestral. Logo as três classes podem ser encontradas:

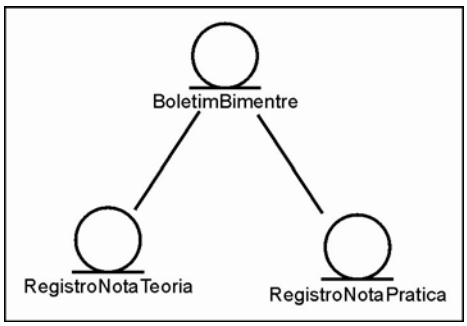


Figura 34.

Passo 2: Encontrar atributos das classes de entidade.

Os atributos da classe RegistroNotaTeoria estão claros: notaMensal1, notaMensal2 e notaProvaBimestral. Estas notas são da parte teórica da disciplina.

O atributo da classe RegistroNotaPratica é apenas a própria média de prática.

A classe BoletimBimestral não possui atributos.

O diagrama de classes de domínio até este ponto da análise é:

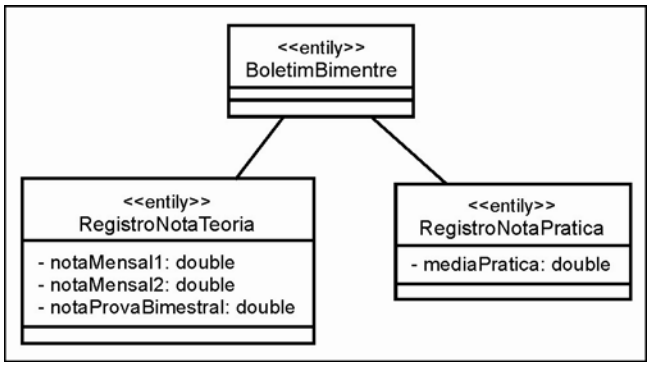


Figura 35.

Passo 3: Completar com métodos acessores e modificadores.

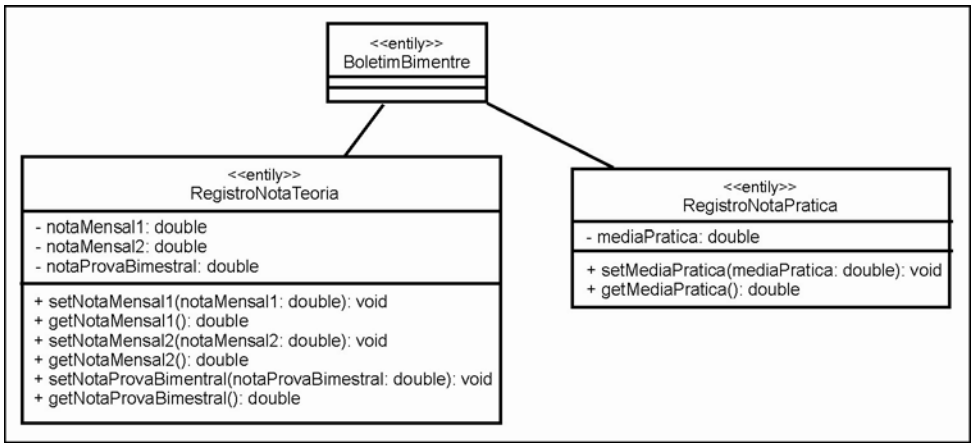


Figura 36.

Passo 4: Seguir o caso de uso para descobrir as tarefas que o sistema executa, mas que ainda não foram definidas até o momento.

1. O usuário escolhe a opção “Calcular média bimestral”.
2. O sistema solicita a nota mensal 1.
3. O usuário fornece a nota mensal 1.
4. O sistema solicita a nota mensal 2.
5. O usuário fornece a nota mensal 2.
6. O sistema calcula a média mensal.
7. O sistema exibe a média mensal e solicita a nota da prova.
8. O usuário fornece a nota da prova.
9. O sistema calcula a média de teoria do aluno.
10. O sistema exibe a nota de teoria da disciplina.
11. O sistema solicita nota da parte prática da disciplina.
12. O usuário fornece a média de prática.
13. O sistema calcula e exibe a média bimestral do aluno.

Pode-se dividir as tarefas principais do caso de uso em três: inicializar registro de nota teórica, inicializar o registro de nota prática e calcular e exibir a média do bimestre. A ordem destas tarefas básicas constituem a ordem lógica do caso de uso. Logo estas tarefas podem ser responsabilidade da classe de controle.

Figura 37.

Pela definição de classe de controle, pode-se incluir um método que controla a ordem de chamada dos outros métodos. Este método sabe que primeiro devemos inicializar o registro de teoria, depois o de prática e depois finalizar o cálculo e exibição da média do bimestre. Para padronizar a forma de chamar um caso de uso, está convencionado neste curso que o nome deste método será iniciar. O método iniciar será o método chamado para começar a execução do caso de uso.

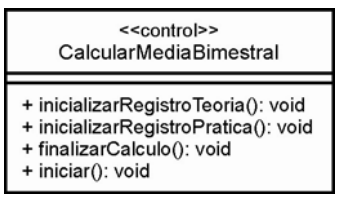


Figura 38.

A partir deste ponto, pode-se construir o diagrama de seqüência e descobrir os métodos que completam o modelo.

A interação começa com o usuário que inicia o caso de uso. Para isso, é necessário ter uma instância de CalcularMediaBimestral e a chamada o método iniciar. Esta chamada é o pontapé inicial para que o programa seja executado. No diagrama de seqüência, o ator tem uma linha de vida e o objeto do tipo CalcularMediaBimestral tem outra.

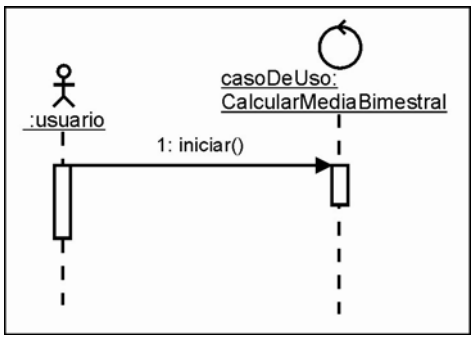


Figura 39.

O usuário deu a ordem para começar o processo. O primeiro passo, seguindo a ordem do caso de uso, é inicializar registro de nota teórica.

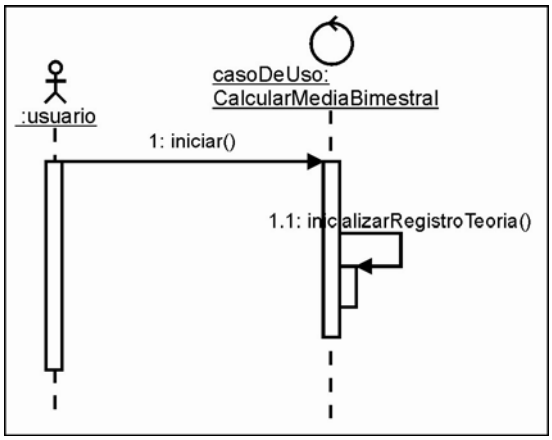


Figura 40.

Pode-se, então, refinar a tarefa inicializar registro de teoria. Esta tarefa compreende os seguintes passos do caso de uso:

1. O sistema solicita a nota mensal.
2. O usuário fornece a nota mensal 1.
3. O sistema solicita a nota mensal 2.
4. O usuário fornece a nota mensal 2.
5. O sistema calcula a média mensal.
6. O sistema exibe a média mensal e solicita a nota da prova.
7. O usuário fornece a nota da prova.
8. O sistema calcula a média de teoria do aluno.
9. O sistema exibe a média de teoria da disciplina.

Para atribuir valores ao registro de nota teórica é preciso que o objeto esteja criado. Esta criação ocorre dentro do método inicializarRegistroTeoria.

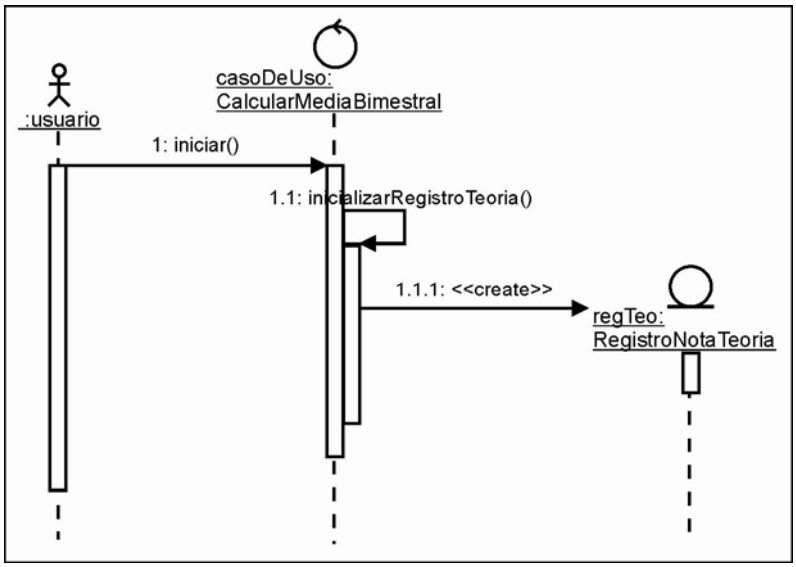


Figura 41.

Depois que o objeto regTeo foi criado, pode-se chamar seus métodos modificadores para atribuir os valores.

1. O sistema solicita a nota mensal 1.
2. O usuário fornece a nota mensal 1.
3. O sistema solicita a nota mensal 2.
4. O usuário fornece a nota mensal 2.

Seguindo a ordem do caso de uso tem-se:

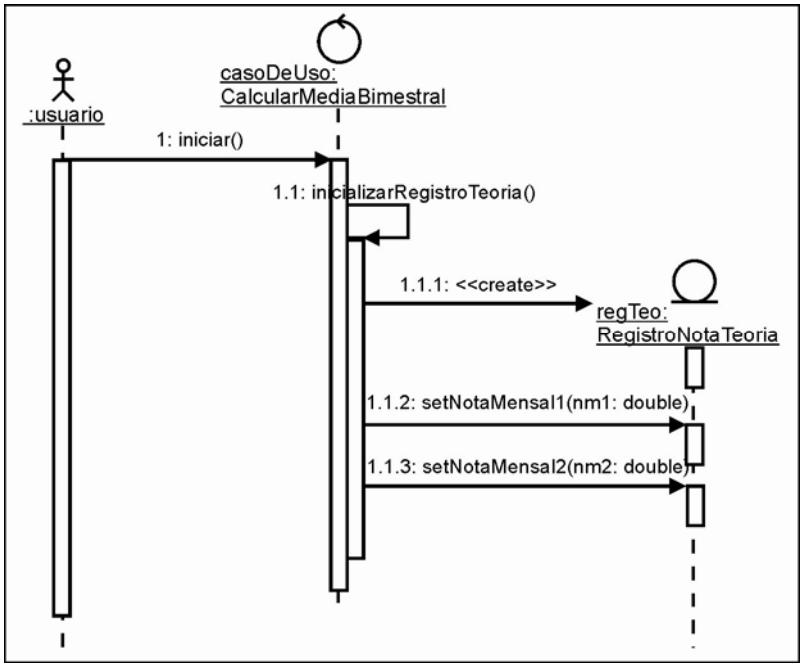


Figura 42.

As mensagens 1.1.2 e 1.1.3 é o que o sistema executa internamente em resposta à ação do usuário nos passos 3 e 5.

No passo 6 do caso de uso está o cálculo da média mensal. O objeto regTeo possui os dados necessários para o cálculo desta média. Logo pode-se acrescentar o método calcularMediaMensal na classe RegistroNotaTeoria:

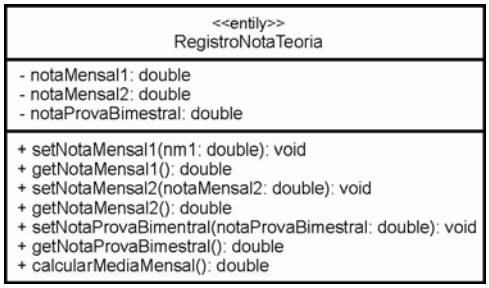


Figura 43.

O diagrama de seqüência com a nova tarefa executada pelo objeto regTeo pode ser analisado a seguir:

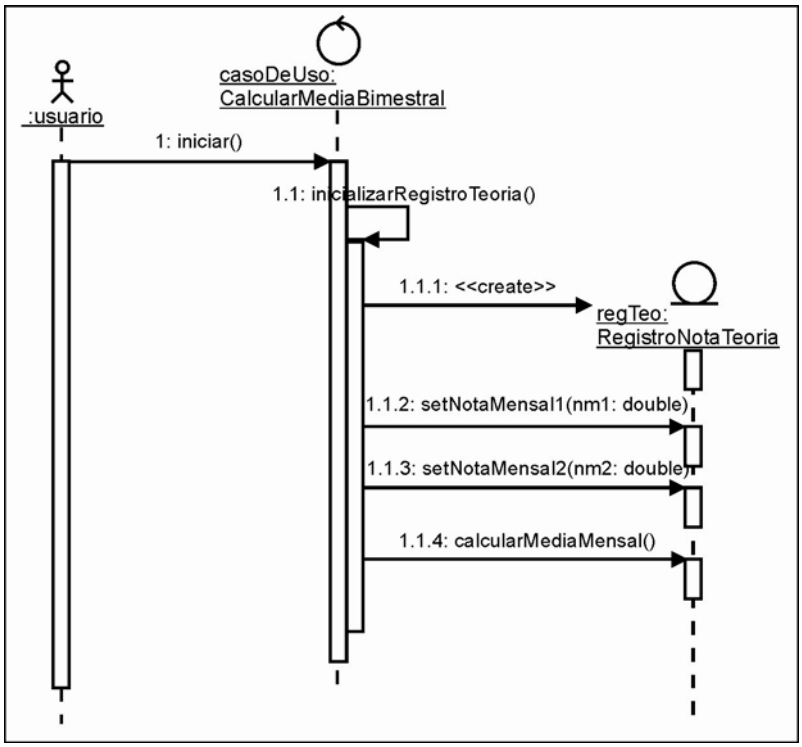


Figura 44.

Os passos 7 e 8 do caso de uso

7. O sistema exibe a média mensal e solicita a nota da prova.
8. O usuário fornece a nota da prova.
9. O sistema calcula a média de teoria do aluno.
10. O sistema exibe a média de teoria da disciplina.

Provocam a chamada do método setNotaProvaBimestral e depois o cálculo da média. O cálculo da média de teoria também deve ser feito pelo objeto da classe RegistroNotaTeoria, pela mesma razão que o método calcularMediaMensal.

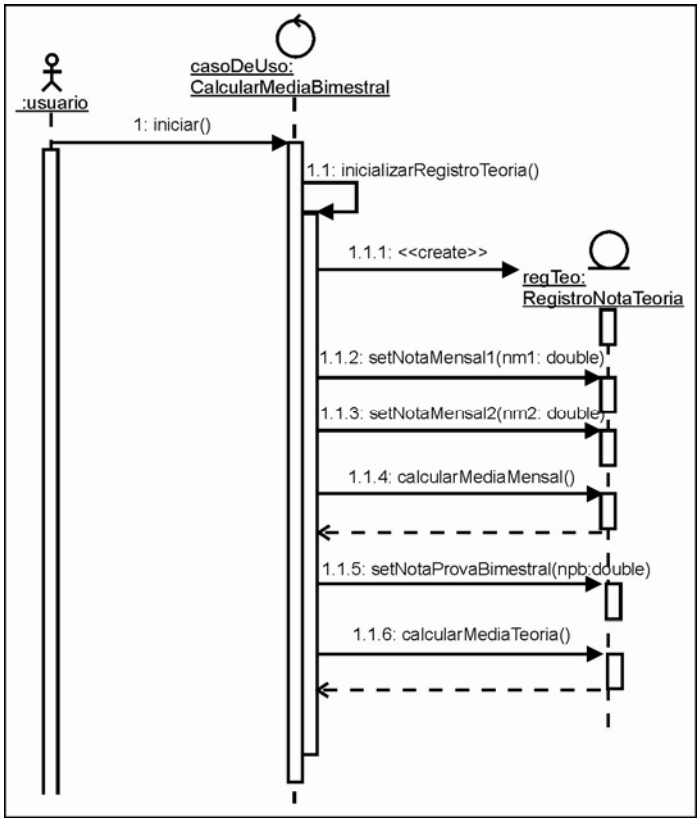


Figura 45.

O registro de teoria deve ser guardado para que a média bimestral seja calculada depois. Para isso pode-se criar um objeto do tipo RegistroBimestral e criar um método que materialize a ligação entre as duas classes (a linha da associação).

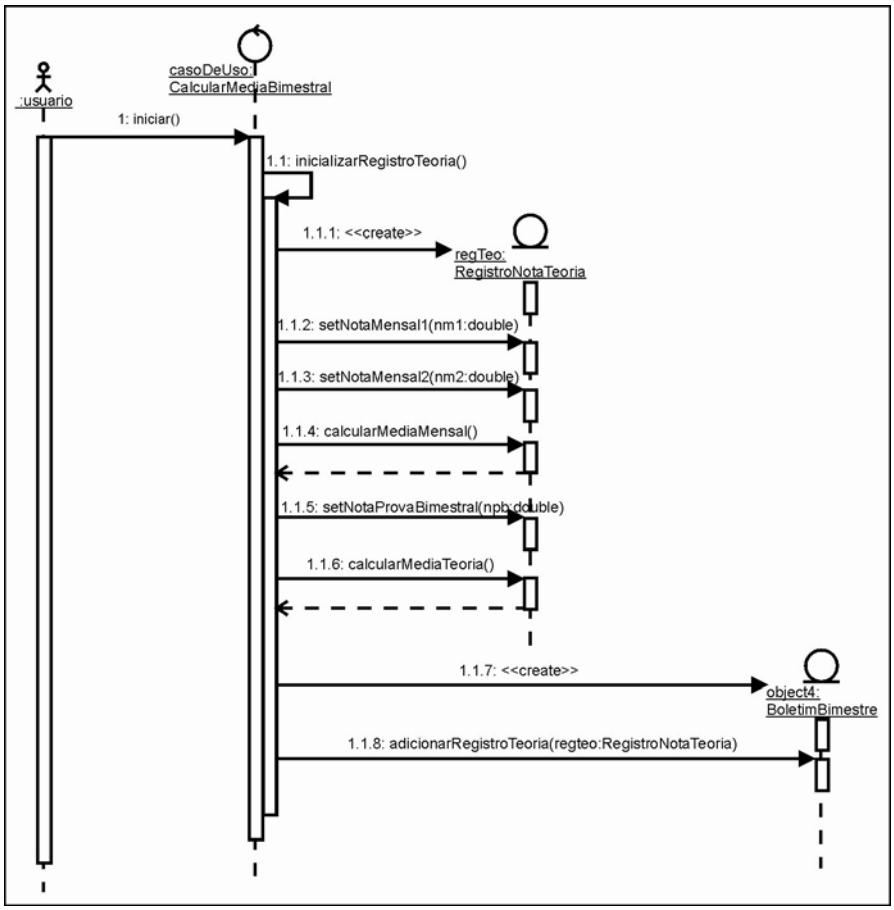


Figura 46.

Assim, termina a tarefa inicializarRegistroTeoria. Agora o método iniciar sabe que deve chamar o método inicializarRegistroPratica. Neste método, ocorre a criação do objeto RegistroNotaPratica e a inicialização do valor do atributo.

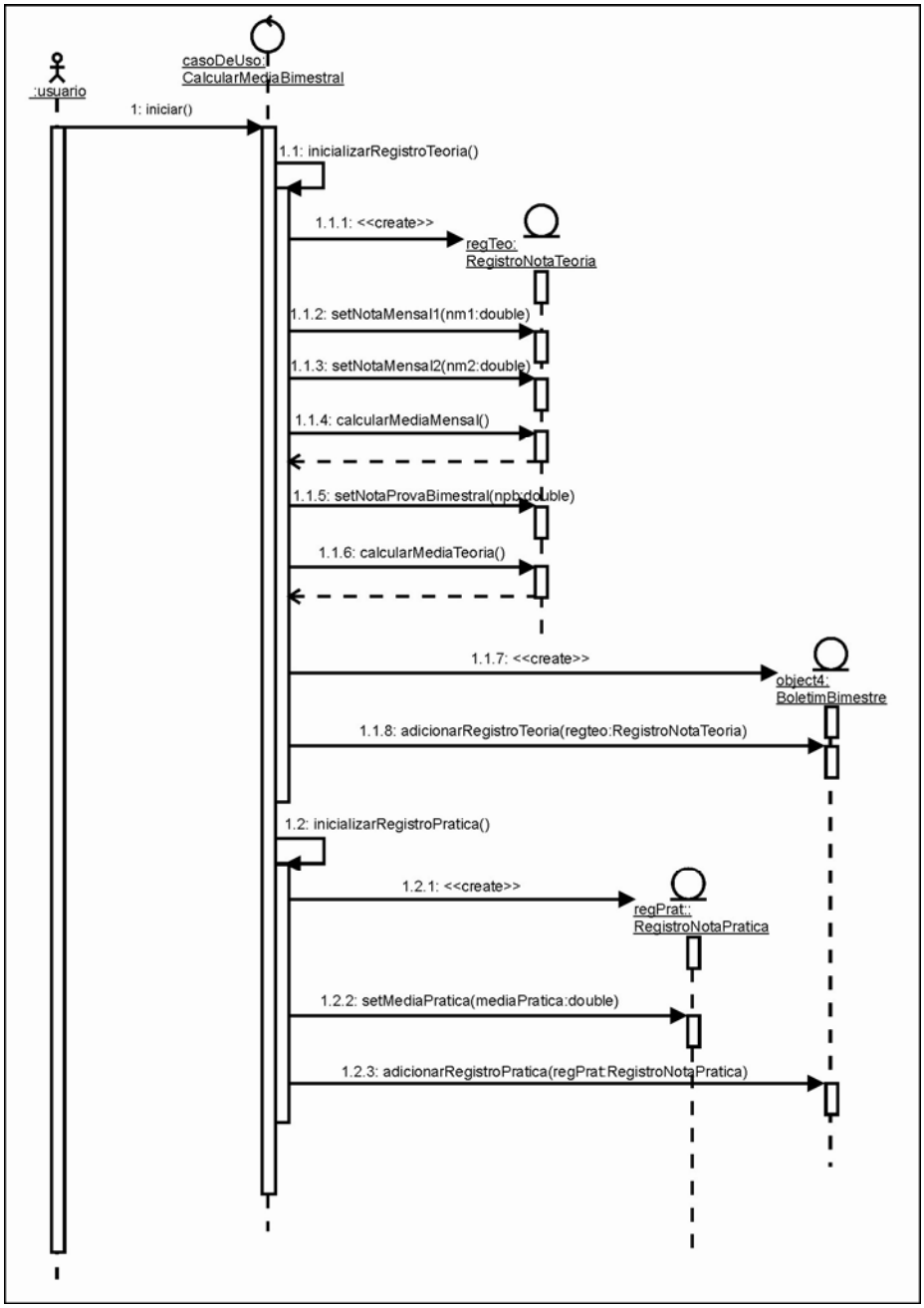


Figura 47.

O registro de prática também deve ser guardado para que a média bimestral seja calculada depois. Para isso, pode-se criar um método que materialize a ligação entre as duas classes (a linha da associação).

Para terminar o caso de uso falta especificar o passo 13 que corresponde ao método finalizarCalculoMediaBimestral.

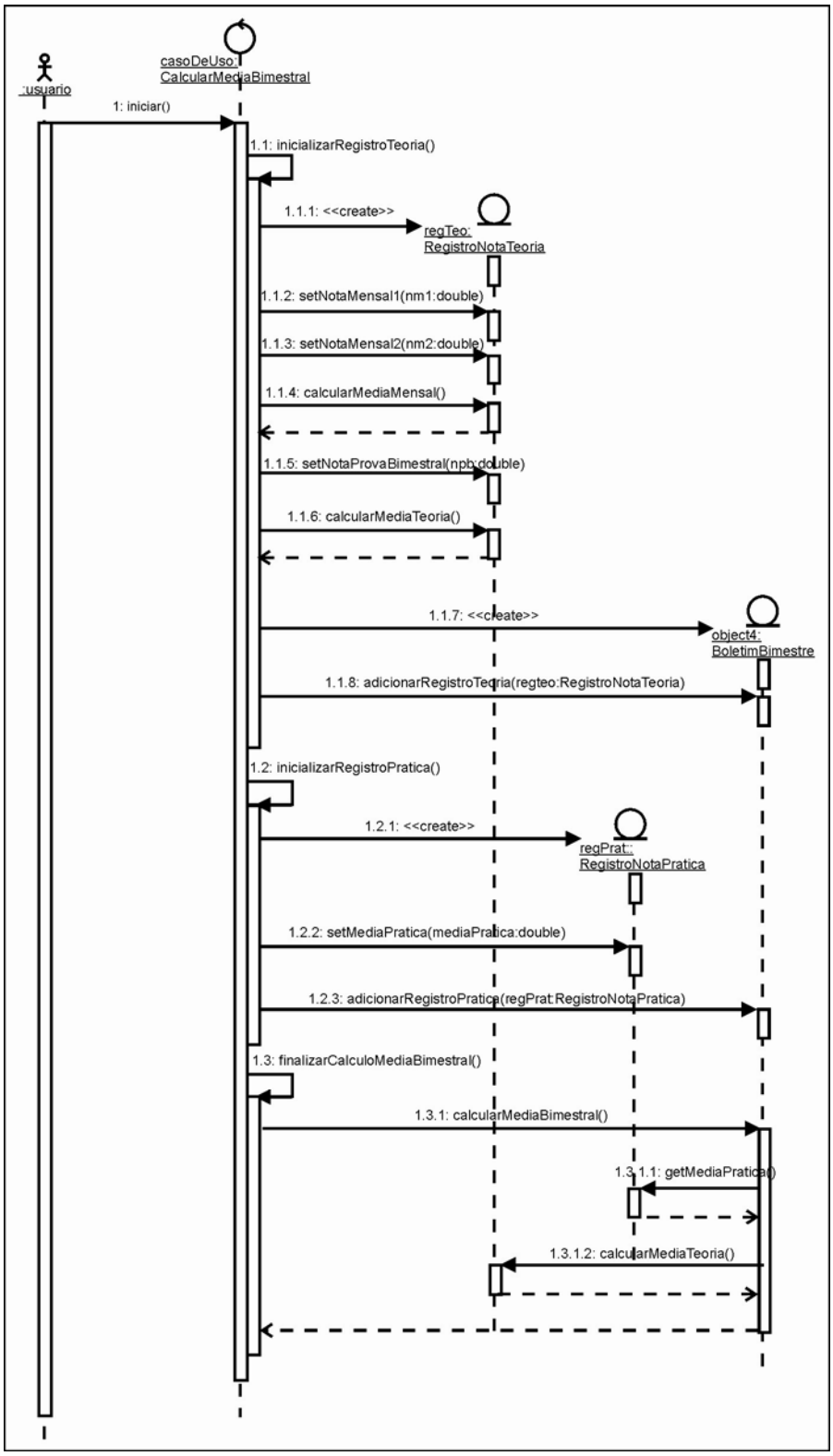


Figura 48.

O objeto regBim possui forma de acessar os dados necessários para realizar o cálculo da média bimestral: as associações com os objetos do tipo RegistroNotaTeoria e RegistroNotaPratica. O método que calcula a média bimestral (mensagem 1.3.1) precisa pedir aos objetos regTeo e

regPrat a média de teoria e de prática, respectivamente. A versão final do diagrama de classes de domínio é:

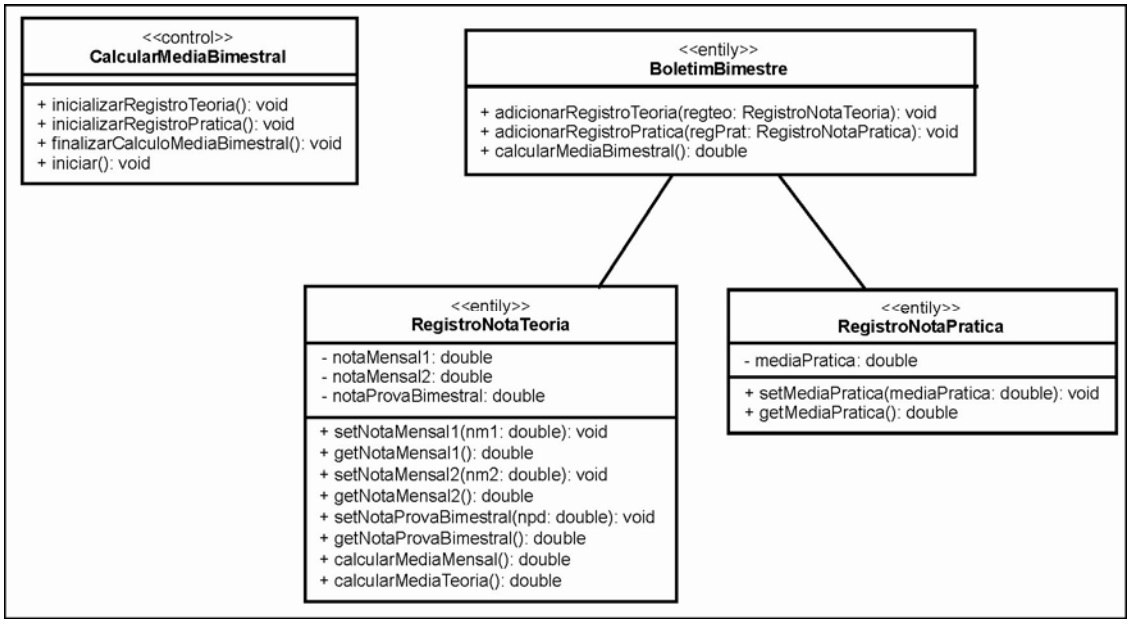
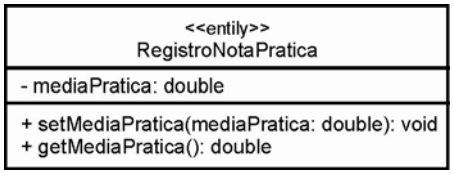


Figura 49.

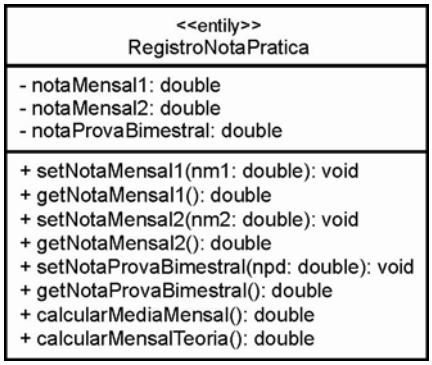
5.3. Passando para o Java

O processo de criação de um programa orientado a objeto consiste em:

1. Traduzir as classes básicas, ou seja, aquelas que não dependem de nenhuma outra classe. No exemplo da seção passada, as classes que devem ser traduzidas primeiro são as classes RegistroNotaPratica e RegistroNotaTeoria.

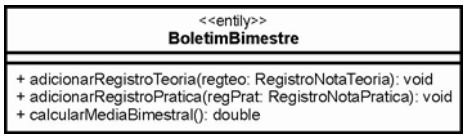



```
//salvar em arquivo RegistroNotaPratica.java
public class RegistroNotaPratica {

private double mediaPratica;

public void setMediaPratica(double mp) {
mediaPratica=mp;

}
public double getMediaPratica() {
    return mediaPratica;
    }
}
```




```
public class RegistroNotaTeoria {
    private double notaMensal1;
    private double notaMensal2;
    private double notaProvaBimestral;

public void setNotaMensal1(double nm1) {
    notaMensal1=nm1;
}
public double getNotaMensal1() {
    return notaMensal1;
}
public void setNotaMensal2(double nm2) {
    notaMensal2=nm2;
}
public double getNotaMensal2() {
    return notaMensal2;
}
public void setNotaProvaBimestral(double npb) {
    notaProvaBimestral=npb;
}
public double getNotaProvaBimestral() {
    return notaProvaBimestral;
    }
public double calcularMediaMensal() {
    return (notaMensal1+notaMensal2)/2;
    }
public double calcularMediaTeoria() {
    double mm= calcularMediaMensal();
    double mt= (7*notaProvaBimestral+3*mm)/10;
    return mt;
  }
}
```

2. Traduzir as classes de entidade que dependem de alguma classe básica. Neste exemplo, a classe que depende das classes básicas é a classe BoletimBimestral.




```
public class BoletimBimestre {
   private RegistroNotaTeoria regTeo;
   private RegistroNotaPratica regPrat;
public void adicionarRegistroTeoria(RegistroNotaTeoria regteo) {
      regTeo=regteo;
   }
public void adicionarRegistroPratica(RegistroNotaPratica regprat) {
      regPrat=regprat;
}
```

```
public double calcularMediaBimestral() {
   double mp=regPrat.getMediaPratica();
   double mt=regTeo.calcularMediaTeoria();
   double mb= (2 * mt + mp)/3;
return mb;
   }
}
```

3. Traduzir a classe de controle.

```
import javax.swing.JOptionPane;
public class CalcularMediaBimestral {
private BoletimBimestre bolBim; //Implementa a associação

//controle da seqüência das tarefas
public void iniciar() {
     inicializarRegistroTeoria();
     inicializarRegistroPratica();
     finalizarCalculoMediaBimestral();
}
public void inicializarRegistroTeoria() {
     RegistroNotaTeoria rt=new RegistroNotaTeoria();
     double nm1=Double.parseDouble(
       JOptionPane.showInputDialog(null,"Digite o nota mensal 1 “));
     rt.setNotaMensal1(nm1);
     double nm2= Double.parseDouble(
       JOptionPane.showInputDialog(null,"Digite o nota mensal 2"));
     rt.setNotaMensal1(nm2);
     double mm=rt.calcularMediaMensal();
     double npb= Double.parseDouble(
        JOptionPane.showInputDialog(null,"A média mensal é"
                +mm+"\nDigite a nota da prova bimestral."));
     rt.setNotaProvaBimestral(npb);
     JOptionPane.showMessageDialog(null,"A média de teoria
é"+rt.calcularMediaTeoria());
     bolBim=new BoletimBimestre();
     bolBim.adicionarRegistroTeoria(rt);
}
public void inicializarRegistroPratica() {
     RegistroNotaPratica rp=new RegistroNotaPratica();
     double mp=Double.parseDouble(
       JOptionPane.showInputDialog(null,"Digite a média de práticas."));
       rp.setMediaPratica(mp);
       bolBim. adicionarRegistroPratica(rp);
}
public void finalizarCalculoMediaBimestral() {
     JOptionPane.showMessageDialog(null,"A média do bimestre é"
                +bolBim.calcularMediaBimestral());
}
}
```

Para possibilitar a execução do programa, uma das classes deve conter o método main. A classe abaixo instancia a classe de controle e dispara o método que desencadeia o processo, o método iniciar.

```
public class PrincipalCapitulo6 {

public static void main(String[] args) {
CalcularMediaBimestral casoDeUso=new CalcularMediaBimestral();
casoDeUso.iniciar();
System.exit(0);
}
}
```

Para compilar este programa, todas as classes devem estar no mesmo diretório e chamar o comando

```
> javac PrincipalCapitulo6.java
```

As versões recentes do compilador javac compilam automaticamente as classes das quais a classe que está sendo compilada depende. Para executar este programa, chamar o comando

```
> java PrincipalCapitulo6
```

Exercícios

01. Modifique o diagrama de seqüência para que seja trocada a inicialização dos objetos por meio dos métodos set para a inicialização usando construtor. Seria possível remover todos os métodos set e ainda manter a mesma ordem do caso de uso?
02. Modifique o programa para que o código fique de acordo com o diagrama de seqüência que você fez na questão 1.
03. Modifique o caso de uso para que sejam removidos todos os métodos set das classes RegistroNotaTeoria e RegistroNotaPratica. Também refaça o diagrama de seqüência e o código.
04. Para cada caso de uso da 3 questão do capítulo 5, faça o diagrama de seqüência, complete o diagrama de classes e escreva o programa em Java.

Bibliografia

BEZERRA, Eduardo. Princípios de análise e projeto de sistemas com UML. Editora Campus. Rio de Janeiro 2002.

LARMAN, Craig. Utilizando UML e Padrões. Bookman Companhia Editora, 2ª. Edição. Porto Alegre, 2004.

JACOBSON, Ivar, Christenson, M. Jonsson, P. e Övergaard, G. Object-Oriented Software Engineering: A Use Case Driven Approach. Addison-Wesley, 1992.

FOWLER, Martin. UML Distilled. 3a. Edição. Addison-Wesley, 2003.