UNIDADE I – JAVA PARA ANDROID

MÓDULO 1 – O SISTEMA DE TIPOS DE DADOS DO JAVA – PARTE I

01

1 - TIPOS DE DADOS EM JAVA

Os tipos de dados em qualquer linguagem de programação são apenas representações de conceitos abstratos existentes em outras ciências ou ramos do conhecimento. Qualquer que seja o tipo de dado existirá sempre dois elementos presentes:

- o(s) **valor**(es) e
- a(s) **operação**(ões).

O(s) valor(es) representa(m) o estado do tipo de dado enquanto a(s) operação(ões) representa(m) as transições de estado do tipo de dado. Tanto os valores quanto as operações devem, sempre, estar em situações permitidas, válidas. Situações proibidas sempre devem ser evitadas e, portanto, não são permitidas.

Os valores geralmente são números, letras ou caracteres que representam as **características** da ideia abstrata. Já as operações são símbolos, procedimentos, funções ou métodos que representam as ações da referida ideia. Por exemplo, o tipo de dado primitivo “int” é a representação da ideia abstrata do que vem a ser o conjunto dos números inteiros da matemática. Essa representação computacional de um conceito matemático se torna concreta, por exemplo, por meio do referido tipo de dado.

Contudo, a representação computacional impõe limitações à plenitude da ideia matemática. Isso significa que a computação, por meio do tipo de dado primitivo “int”, representa uma parte da ideia referente à teoria matemática dos conjuntos dos números inteiros. Por exemplo, imagine um tipo de dado “int” de 8 bits sem sinal. Esse tipo possui dois limites: **inferior** e **superior**. O menor valor possível para o limite inferior é o número zero (0) e o maior valor possível para o limite superior é o número 255 (2^8 - 1). Além disso, as operações permitidas são, geralmente, as aritméticas (+, -, *, /, %, dentre outras) e as relacionais (==, !=, <, >, dentre outras), sendo que em algumas linguagens, as operações lógicas (&&, ||, dentre outras) também são.

02

Agora imagine se quiséssemos armazenar o número 365 que, por exemplo, representa a quantidade de dias existentes em um ano não bissexto do calendário gregoriano, utilizando-se para tal a nossa variável do tipo int de 8 bits sem sinal. Diante disso, pode-se perceber que não é possível armazenar qualquer

valor que esteja fora do intervalo permitido (0 a 255) para o referido tipo de dado uma vez que qualquer valor fora do referido intervalo é considerado um valor proibido.

Agora imagine que você tenha três variáveis do referido tipo “int” de 8 bits sem sinal. Na primeira é armazenado o valor 200, que é um valor permitido, pois está dentro do intervalo de 0 a 255. Na segunda variável é armazenado o valor 150, que também é outro valor permitido. E na terceira variável é armazenado o resultado de uma operação aritmética de soma, que é uma operação permitida, com as duas variáveis anteriores. Ou seja, se somarmos 200 + 150, ambos, operação e valores permitidos, teremos como resultado o valor 350 que é um valor proibido, e consequentemente, o mesmo não é armazenado computacionalmente, pois se encontra fora do intervalo de valores possíveis para o supracitado tipo de dado.

Variável int 1	Variável int 2	Variável int 3
200	150	200 + 150 = 350

Diante disso, é notório perceber que o tipo de dado “int” presente na computação impõe limitações à representação do conjunto dos números inteiros da matemática.

03

Na linguagem de programação Java, há dois tipos de dados distintos e fundamentais: primitivos e abstratos (compostos). A linguagem fornece segurança de tipo forçando uma tipagem estática, exigindo que toda variável, antes de ser utilizada, seja declarada **acompanhada de seu respectivo tipo de dado**.

Por exemplo, uma variável cujo identificador seja “foo” declarada como sendo do tipo int (inteiro primitivo de 32 bits com sinal) pode ser feita da seguinte forma:

```
int foo;
```

Essa forma contrasta com outras linguagens de tipagem não estática (dinâmica), como por exemplo, PHP, Python, Perl, JavaScript, dentre outras. Nessas linguagens, as variáveis são declaradas apenas de forma opcional. A linguagem de programação Java possui um recurso que a aproxima das flexibilidades de tipagem dinâmica denominado de Reflection, introduzido a partir da versão 1.1 da JDK. Além disso, o uso de mecanismos de polimorfismo introduzido pela técnica de programação orientada a objeto permite unir o melhor dos dois mundos: a flexibilidade da tipagem dinâmica com a segurança da tipagem estática.

Ainda que declarações de tipo de forma estática sejam mais trabalhosas, elas permitem que o compilador impeça uma grande quantidade de erros de programação que poderiam prejudicar o código

final – por exemplo, a criação acidental de variáveis diferentes, como resultado de erros em seus nomes; chamadas a métodos inexistentes e assim por diante.

04

2 - TIPOS DE DADOS PRIMITIVOS

Os tipos de dados primitivos no Java não são objetos. O Java possui oito tipos de dados primitivos:

boolean

Valores ou true ou false

byte

Um inteiro de 8 bits com sinal (complemento de 2).

short

Um inteiro de 16 bits com sinal (complemento de 2).

int

Um inteiro de 32 bits com sinal (complemento de 2).

long

Um inteiro de 64 bits com sinal (complemento de 2).

char

Um inteiro de 16 bits sem sinal representando unidades de código UTF-16.

float

Números de ponto flutuante de 32 bits de acordo com a norma IEEE 754.

double

Números de ponto flutuante de 64 bits de acordo com a norma IEEE 754.

A notação “complemento de 2” é a forma atual, utilizada pelos computadores modernos para representar números inteiros negativos. Isso se deve em grande parte, ao fato deste modelo de representação possuir apenas uma sequencia binária para representar o número zero.

05

Alguns autores consideram os tipos de dados primitivos como sendo tipos de dados simples, ou seja, aqueles tipos de dados cujos valores são considerados **indivisíveis**. Apesar de existirem divergências no que diz respeito a essa afirmação, em relação à dimensão em que se efetua a devida análise, a mesma é

considerada verdadeira para a maioria das linguagens de programação, principalmente, no que se refere à visão inicial que a maioria dos programadores tem das referidas linguagens.

Abaixo, uma tabela com a representação dos valores permitidos:

Category	Types	MSB	Size (bits)	Minimum Value	Maximum Value	Precision	Example
Integer	**byte**	signed	8	-128	127	From +127 to -128	byte b = 65;
	char	unsigned	16	0	2^{16}-1	All Unicode characters	char c = 'A'; char c = 65;
	short	signed	16	-2^{15}	2^{15}-1	From +32,767 to -32,768	short s = 65;
	int	signed	32	-2^{31}	2^{31}-1	From +2,147,483,647 to -2,147,483,648	int i = 65;
	long	signed	64	-2^{63}	2^{63}-1	From +9,223,372,036,854,775,807 to -9,223,372,036,854,775,808	long l = 65L;
Floating-point	**float**	N/A	32	2^{-149}	$(2-2^{-23})\cdot 2^{127}$	From 3.402,823,5 E+38 to 1.4 E-45	float f = 65f;
	double	N/A	64	2^{-1074}	$(2-2^{-52})\cdot 2^{1023}$	From 1.797,693,134,862,315,7 E+308 to 4.9 E-324	double d = 65.55;
Other	**boolean**	N/A	1	--	--	false, true	boolean b = true;
	void	--	--	--	--	--	--

O tipo "**void**" não é considerado um tipo de dado primitivo na linguagem Java. Além disso, os tipos de dados "Floating-point" são definidos pela IEEE 754-1985 que estabelece a forma de representação e a forma de interpretação para os referidos tipos primitivos float e double. Um ponto importante a se ressaltar é que tal norma, IEEE 754, atualmente é utilizada pela absoluta maioria das linguagens de programação comercialmente disponíveis, além de também ser utilizadas por diversos tipos diferentes de hardware, como por exemplo a FPU (floating-point unit) presente nos processadores modernos.

06

Como forma de demonstrar as limitações dos tipos de dados primitivos, observe o código abaixo para o tipo **byte**:

```
public class Main {
	public static void main(String[] args) {

		byte a = 100;

		byte b = 100;

		byte c = (byte)(a + b);

		System.out.println(c);
	}
}
```

O referido programa não possui qualquer erro de sintaxe ou semântica computacional. O mesmo compila e executa sem qualquer tipo de erro computacional. O interessante deste exemplo é que a resposta esperada pelo usuário que executa o programa não é a mesma apresentada pelo computador. E o mais instigante é que ambos estão certos: o usuário e o computador, cada um com a sua verdade.

Para o computador, o resultado da soma de 100 + 100 é -56, valor permitido para o tipo de dado byte. Acreditem ou não, esse resultado está correto, para o computador, obviamente. Já, para o usuário, o valor esperado pela soma de 100 + 100 é 200, valor proibido para o tipo de dado byte. O valor 200 não é armazenado na variável "c", simplesmente porque ele não representa um estado válido, ou seja, um valor válido para a referida variável, uma vez que o tipo byte aceita valores no intervalo de -128 a +127.

07

2.1. Estado de uma Variável

O **estado** é representado pelo valor permitido presente na variável de um determinado tipo, ou seja, um valor presente no intervalo definido para o referido tipo. O estado de uma variável somente pode assumir valores permitidos, nunca proibidos.

Para facilitar a compreensão vamos imaginar um caso simples, usando uma variável de tipo primitivo (um valor único) inteiro (int) na linguagem de programação Java. Considere a instrução:

```
int a = 10;
```

Declaramos a variável cujo identificador é "a" e cujo valor permitido é 10. Ou seja, podemos dizer que o **estado** da variável é representado por seu valor. Isso posto, a variável tem o estado 10 (valor permitido).

Agora considere a seguinte instrução que possui uma operação de somar:

```
a = a + 1;
```

Neste caso, a variável "a" que valia 10 passou a valer 11. Ou seja, a variável passou pelo que chamamos de **transição de estado**. Isso significa que a variável mudou de um valor permitido (10) para outro valor permitido (11). Deste modo a **transição de estado** é considerada válida.

08

Um outro exemplo de transição de estado válido, só que agora com efeitos colaterais é posto logo abaixo. Imagine a mesma variável “a” com um novo valor (dois bilhões, cento e quarenta e sete milhões, quatrocentos e oitenta e três mil, seiscentos e sete e oito):

```
a = 2147483647;
```

Esse alto valor é um valor permitido, ou melhor, é o mais alto valor permitido (limite superior) para uma variável do tipo “int” de 32 bits com sinal. Agora, considere novamente a seguinte instrução:

```
a = a + 1;
```

Neste caso, a variável "a" que valia 2147483647passaria a valer 2147483648. Ou seja, a variável passaria pelo que chamamos de transição de estado inválido. Isso significa que a variável mudou de um valor permitido (2147483647) para outro valor proibido (2147483648). Deste modo a transição de estado seria considerada inválida e jamais poderia ser executada. Tanto não é que o resultado apresentado pelo programa é o valor -2147483648, ou seja, um número negativo. E acreditem ou não, computacionalmente, esse resultado está correto. É obvio que para nós, seres humanos, esse resultado está errado. Portanto, somente transições de estado válidas e valores válidos é que são permitidos. Sempre!

Contudo, a maioria dos códigos, comerciais ou científicos, escritos atualmente em qualquer linguagem de programação, não levam em consideração o estado das variáveis ou objetos bem como as suas possíveis transições. Não levar isso em consideração, significa escrever códigos vulneráveis e passíveis de erros que produzem problemas cujas causas são extremamente difíceis de se encontrar e resolver.

09

2.2. Complemento de 2

A notação de complemento de 2 é utilizada pela computação atual para representar os números inteiros negativos. O principal objetivo do complemento de 2 é trazer uma representação única ao número zero e possibilitar a soma de números positivos e negativos independentemente do sinal.

Considere o exemplo abaixo:

Número base 10	Número base 2 (tamanho de 8 bits)
126	01111110
100	01111011

Se quiséssemos realizar uma operação de subtração de um número pelo outro, poderíamos fazê-la da seguinte forma: 126 – 123. É fácil perceber que o resultado da referida operação é 3. Esse mesmo cálculo poderia ter sido escrito da seguinte forma: 126 + (-123).

O resultado continua sendo o mesmo 3, contudo observe que agora estamos somando o número 126 com o complemento de 2 do número 123.

10

Observe o cálculo binário abaixo:

Passo 1: Representar em binário o número 123

01111011 (número 123 em binário)

Passo 2: De posse da representação binária do passo 1, inverter todos os bits do número.

10000100 (Complemento de 1 do número 123)

Passo 3: Somar o número binário do passo 2 ao número binário um.

10000100 + 00000001 = 10000101 (Complemento de 2 – Soma do complemento de 1 mais o número um)

Portanto, o número -123 pode ser escrito em notação de complemento de 2 como sendo 10000101 (número binário do passo 3).

Agora, para encontrar o resultado da subtração de 126 – 123, basta realizar a operação de soma entre a representação binária do número 126 com a representação binária em notação de complemento de 2 do número 123, ou seja, 01111110 + 10000101 = ~~1~~0000 0011.

Observe que o bit de mais alta ordem, o mais a esquerda, excede os 8 bits possíveis da nossa representação atual. Deste modo, o nono bit deve ser desconsiderado, e por esse motivo, o mesmo se encontra ~~grifado em vermelho~~.

Portanto o resultado é 00000011 que corresponde ao número 3 em base decimal.

11

RESUMO

O objetivo deste módulo foi apresentar os tipos de dados primitivos (boolean, byte, short, int, long, char, float e double) da linguagem de programação Java com seus respectivos limites de representação (operações e valores). Os intervalos de valores para cada tipo de dado é determinado em função da quantidade de bits de cada tipo bem como da interpretação que se faz desses bits, levando-se em conta ou não a notação de complemento de 2. Além disso, esse módulo apresentou, para os referidos tipos, os conceitos relativos ao estado de uma variável e as transições de estado de uma variável.

UNIDADE I – JAVA PARA ANDROID

MÓDULO 2 – O SISTEMA DE TIPOS DE DADOS DO JAVA – PARTE II

01

1 - TIPOS DE DADOS ABSTRATOS (COMPOSTOS)

Apesar de parecer óbvio, antes de começarmos o assunto deste módulo, lembre-se que no módulo anterior foi tratado do assunto **tipo de dado**, com seus valores e operações. Não se esqueçam de que tipo de dado abstrato (composto) é um tipo de dado. Isso significa que tipo de dado abstrato continua seguindo as mesmas regras, ou seja, valores e operações somente devem ser permitidas e nunca proibidas.

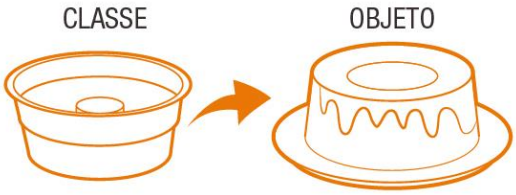
Java é, basicamente, uma linguagem de programação imperativa (como fazer) que oferece suporte ao paradigma de programação denominado de *orientação a objeto* que, além dos tipos primitivos, possui uma API extremamente rica e variada baseada em classes (tipo de dado abstrato/composto) e, consequentemente, nos seus respectivos objetos que derivam dessas classes.

Programação imperativa, diferentemente da *programação declarativa* (o que fazer), é um paradigma de programação que descreve a computação como ações, enunciados ou comandos que mudam o estado (variáveis) de um programa.

O nome do paradigma, **imperativo**, está ligado ao tempo verbal imperativo, onde o programador diz ao computador como os comandos devem ser executados, ou seja, faça isso, depois aquilo, depois aquilo outro e assim, sucessivamente.

02

Antes de detalhar os termos **classe** e **objeto**, uma analogia que auxilia na compreensão seria comparamos a classe a uma fôrma de bolo e o objeto aos bolos (cenoura, chocolate, fubá, dentre outros) propriamente ditos, que são fabricados utilizando-se a fôrma.Ou seja, uma classe está para uma fôrma de bolo assim como um objeto está para um bolo de, por exemplo, cenoura, que é fabricado utilizando-se a referida fôrma.



Essa analogia é interessante, pois se percebe que com uma única classe (fôrma) é capaz de se fabricar inúmeros objetos (bolos).

Outro ponto importante a ser observado é que a classe (fôrma) representa a estrutura do objeto (bolo), ou seja, se a fôrma for redonda, o bolo será redondo; se a fôrma for retangular, o bolo será retangular; e assim por diante. Observe que a classe (estrutura) determina a forma, o tamanho, a espessura, dentre outras características ou comportamentos dos objetos derivados.

03

Outro ponto importante a ser observado nesta mesma analogia é que se usarmos a mesma fôrma para fabricar dois bolos diferentes, por exemplo, chocolate e cenoura, apesar dos bolos terem a mesma estrutura, eles terão sabores (valores) diferentes.

Isso poderia ser comparado, por exemplo, ao fato de criarmos duas variáveis do tipo primitivo “int”, A1 e A2, e atribuirmos à variável A1 o valor 35 e à variável A2 o valor 40. Ou seja, as variáveis possuem a mesma estrutura (mesmo tipo de dado – 32 bits com sinal em notação de complemento de 2), contudo valores permitidos diferentes (uma valendo 35 e a outra 40).

Esses pontos fundamentais, sem detrimentos de outros apresentados ao longo do curso, que por analogia foram descritos, são e serão essenciais para se compreender as diversas possibilidades que decorrem deste simples conceito relativo a classes e objetos.

Uma classe define os campos (atributos) ou os procedimentos (métodos) que representam um conceito ou ideia qualquer. Portanto, é possível:

- ter uma classe que defina apenas os campos sem qualquer procedimento.
- ter uma classe que defina apenas os procedimentos sem qualquer campo.
- ter uma classe que defina ambos, campos e procedimentos, o modelo mais comum. Contudo, os outros modelos possuem suas aplicações e são plenamente possíveis de existir.

No decorrer desta disciplina, todos estes três modelos serão utilizados.

04

Os **campos**, quando existem, podem ser de outros tipos de dados **primitivos** ou **abstratos** e os procedimentos, quando também existem, geralmente, realizam operações que manipulam os campos. Considere uma possível estrutura, por ora apenas os campos/atributos, para a classe que representa a data do calendário gregoriano:

```
public class Data{

        private int dia;

        private int mes;

        private int ano;

        ...

}
```

Neste caso, podemos perceber que a classe Data é um tipo de dado abstrato/composto que reúne um conjunto de três campos/atributos denominados neste exemplo de dia, mês e ano. Tendo esse tipo de dado definido, é perfeitamente possível declarar uma referência deste tipo conforme a instrução abaixo:

```
Data aniversario;
```

Nesse caso, o identificador aniversario é denominado de **referência** uma vez que seu conteúdo é um metadado (endereço de memória). Essa referência é alocada em uma área de memória denominada de **stack**. Caso o conteúdo do identificador fosse um dado propriamente dito, o mesmo seria denominado de variável e a referida variável poderia ser guardada em outras áreas de memória, inclusive aquelas denominadas de **text** ou **bss**, por exemplo. Saiba+

Saiba+

Para compreender mais sobre a divisão de um programa em memória acesse:
https://en.wikipedia.org/wiki/Data_segment

05

Complementando o raciocínio, **um objeto é simplesmente uma instância de uma classe**.

Uma instância nada mais é que um espaço de memória alocado dinamicamente na heap em tempo de execução que é utilizado para guardar uma cópia exclusiva de cada atributo/campo de uma classe.

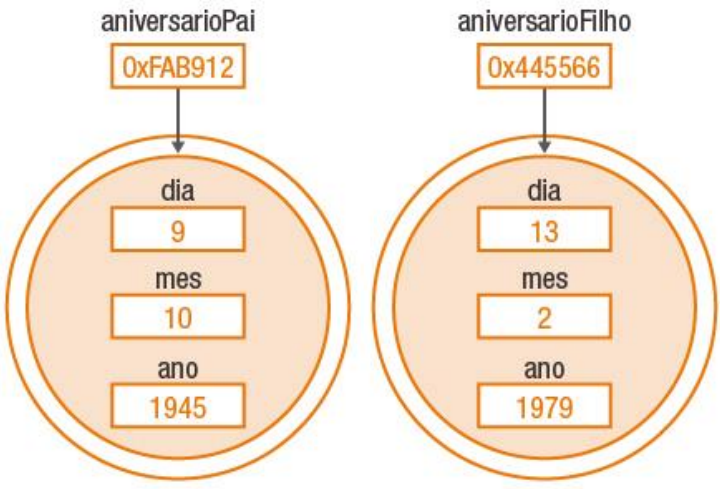
Tais cópias exclusivas são denominadas de **identificadores de instância,** que podem ser de dois tipos:

- variáveis ou
- referências.

São essas variáveis/referências de instância que são responsáveis por armazenar os possíveis valores para um objeto. Portanto, considere as duas instruções abaixo:

```
Data aniversarioPai = new Data(9, 10, 1945);
```

```
Data aniversarioFilho = new Data(13, 2, 1979);
```



Observe que os objetos são do mesmo tipo de dado abstrato/composto, neste exemplo Data. No desenho acima, cada objeto está sendo representado pelo seu próprio círculo de borda dupla com três

retângulos internos (estrutura). Cada retângulo representa a cópia exclusiva, neste exemplo são variáveis de instância, dos atributos da referida classe. Observe que os identificadores “aniversarioPai” e “aniversarioFilho” são referências cujo conteúdo é, neste caso, um endereço de memória (metadado), que aponta para um objeto cujos valores das variáveis de instância (estado do objeto) representam as datas de aniversários (dia, mês e ano) de cada um dos objetos existentes. Observe também que os objetos possuem a mesma estrutura, contudo possuem valores (estado) diferentes. Além disso, observe que cada objeto possui uma cópia exclusiva dos campos/atributos da classe Data, ou seja, quando a classe Data foi criada, a mesma definiu apenas um campo/atributo para o dia, um outro para o mês e um terceiro para o ano. No entanto, quando criamos o objeto do tipo Data, cada objeto criou a sua própria cópia exclusiva de cada atributo presente na classe. Essa é a regra geral.

Observe que o uso do operador "new" da linguagem de programação solicita a alocação dinâmica de memória em tempo de execução. Um ponto importante a ser observado é que a linguagem de programação Java faz uso de um recurso automático de liberação de memória conhecido como Garbage Collector (GC). O GC tem por finalidade a função de liberar os espaços de memória que não estejam sendo utilizados, ou seja, aqueles objetos que foram alocados pelo operador "new" e não possuem mais qualquer referência apontando para eles, são considerados inúteis pelo GC e, consequentemente, serão removidos oportunamente pelo mecanismo.

06

Contudo, existem exceções a essa regra como, por exemplo, o caso de **atributos que pertencem à classe** e não aos objetos da referida classe. Nesta exceção esses atributos são utilizados para representarem valores constantes, que não mudam com o tempo.

Observe por exemplo, o atributo **PI** da classe **Math (***Math.PI***)** que é um exemplo da referida exceção.

Um ponto importante a ser ressaltado é que um objeto representa um único valor válido que é composto pelo **conjunto** de outros possíveis valores, que, **combinados**, continuam sendo válidos. Lembre-se que os valores assumidos pelas variáveis de instância devem ser sempre permitidos e nunca devem ser proibidos. Caso as variáveis assumam valores proibidos, isso significa que o estado de objeto é inválido e seu tipo de dado abstrato (classe) possui um erro de lógica (algoritmos e estruturas) em sua concepção.

Portanto, nem sempre um valor inteiro válido pode ser considerado válido para a representação dos valores do tipo de dado abstrato em questão. Por exemplo, caso a variável de instância “dia” do tipo “int” assuma o valor zero (0) podemos afirmar que o referido valor é um inteiro válido, contudo o valor

zero (0) é inválido para representar o dia no calendário gregoriano. Ou seja, no calendário gregoriano, o menor valor (limite inferior) possível para o dia é o número um (1).

Da mesma forma, no outro extremo, caso a mesma variável "dia" assuma o valor trinta e dois (32) podemos afirmar que o referido valor é um inteiro válido, contudo o valor trinta e dois (32) é inválido para representar o dia no calendário gregoriano. Ou seja, no calendário gregoriano temos um conjunto de quatro possíveis maiores valores que são: ou 28, ou 29, ou 30 ou 31 a depender do mês e do ano em questão. Portanto, dependendo do mês e do ano, o maior valor (limite superior) pode variar dentre um dos quatro possíveis valores. Essa mesma ideia, ora apresentada para o "dia", também se aplica ao mês e ao ano, ressalvado as devidas diferenças e particularidades de cada elemento que compõem a data do referido calendário.

07

Por último, observe que os valores que representam uma data qualquer do calendário gregoriano estabelecem relações entre si, a depender do ponto de vista de qual valor se está analisando. Por exemplo, o ano é independente do mês e do dia, uma vez que seus limites são (0 a 9999). O mês também é independente do ano e do dia, uma vez que seus limites são (1 a 12). Contudo o dia é dependente do mês e do ano, pois apesar do limite inferior sempre começar em 1 para todos os meses, o limite superior pode assumir quatro possíveis valores válidos (28, 29, 30, 31) a depender dos valores contidos nas variáveis mês e ano.

Outra análise importante diz respeito às **operações/métodos/procedimentos**. Observe que a operação de incrementar o ano (incrementaAno) é independente das demais operações de incremento. Contudo, a operação de incrememementar o mês (incrementaMes) depende da operação incrementaAno e a operação de incrementar o dia (incrementaDia) depende da operação incrementaMes que, por sua vez, depende da operação incrementaAno.

Ou seja, a relação de dependência entre os atributos (dia, mês e ano) é regida por uma lógica diferente daquela que relaciona as operações/métodos de incremento para cada um dos respectivos atributos. Isso significa, que do ponto de vista da orientação objeto, cada uma dessas relações lógicas devem ser, obrigatoriamente, representadas de forma íntegra, clara, precisa, coerente, coesa, segura, simples e autônoma.

08

2 - CLASSE DATA

Em Java, a definição de Classe – o modelo a partir do qual os objetos são construídos – é, ela mesma, um tipo específico de objeto, ***Class***. Nessa linguagem, classes formam a base de um tipo de sistema que permite aos desenvolvedores descreverem, arbitrariamente, objetos complexos, com estado e comportamento complexos e especializados.

Tais classes são compostas por outros tipos de dados, **primitivos** ou **abstratos** ou ambos. Isso significa que a combinação de tipos de dados é infinita para se formar novos tipos. Observe, a **combinação é infinita**, não os tipos de dados. Por isso, a cada nova versão da linguagem de programação Java, novos tipos são construídos e até mesmo descontruídos (descontinuados).

Esse movimento construir e desconstruir é perfeitamente natural, uma vez que tais tipos são reflexos das habilidades humanas em compreender, plenamente ou parcialmente, um determinado problema em um determinado instante do tempo. Deste modo, fica claro que tais tipos de dados abstratos são uma das formas de extensão e ampliação das funcionalidades de uma linguagem de programação, ou seja uma das formas que os programadores/desenvolvedores tem de utilizar as linguagens de programação de modo geral, para qualquer fim.

Portanto, do ponto de vista do usuário, geralmente um programador, da linguagem de programação Java e do paradigma orientado a objeto:

- os tipos básicos, ou **primitivos**, representam unidades indivisíveis de informação de tamanho variável,
- já os tipos **abstratos**, ou compostos, representam unidades divisíveis.

Portanto, os tipos básicos são predefinidos pela linguagem e os tipos abstratos predefinidos pelos usuários programadores da linguagem.

09

Como forma de exemplificar o referido conceito, observe o exemplo a seguir que abstrai a data do calendário gregoriano. Considere também que o exemplo, é apenas uma das várias formas possíveis de demonstrar o referido assunto, não devendo ser, portanto, considerado como a única forma de representação. Ou seja, a forma apresentada abaixo é apenas uma dentre as várias formas possíveis de representação do referido conceito.

```
package br.aiec;

/**
 *
 * Denomina-se calendário gregoriano o calendário promulgado pelo Papa Gregório XIII em
 * 1582 e adotado,  imediatamente, por Espanha, Itália, Portugal, Polónia e, posteriormente, por
 * todos os  países ocidentais.
 *
 * A referida classe é uma possível implementação do referido conceito.
```

```
 *
 * @author Guilherme Veloso
 *
 */

public class Data {

	private int dia;
	private int mes;
	private int ano;

	private boolean ehAnoBissexto(int valor) {
		return (valor % 400 == 0 || (valor % 100 != 0 && valor % 4 == 0));
	}

	private boolean existeAno(int valor) {
		return valor >= 0 && valor <= 9999;
	}

	private boolean existeMes(int valor) {
		return valor >= 0 && valor <= 12;
	}

	private boolean existeDia(int valor) {
		return valor >= 0 && valor <= limiteDiasMes(mes);
	}

	private int limiteDiasMes(int valor) {
		int dias[] = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };

		if (ehAnoBissexto(ano) && valor == 2) {
			return 29;
		}

		return dias[valor - 1];
	}

	public Data(int d, int m, int a) {
		ano = (existeAno(a)) ? a : 0;
		mes = (existeMes(m)) ? m : 1;
		dia = (existeDia(d)) ? d : 1;
	}

	public Data(Data dt){
		ano = dt.ano;
		mes = dt.mes;
		dia = dt.dia;
	}

	public void setDia(int valor) {

		if (existeDia(valor)) {
```

```
                dia = valor;
        }
    }

    public int getDia() {
        return dia;
    }

    public void setMes(int valor) {
        if (existeMes(valor)) {
            mes = valor;
        }
    }

    public int getMes() {
        return mes;
    }

    public void setAno(int valor) {
        if (existeAno(valor)) {
            ano = valor;
        }
    }

    public int getAno() {
        return ano;
    }

    public void incrementaAno() {
        ano = (ano + 1) % 10000;
    }

    public void incrementaMes() {
        int novoMes = (mes + 1) % 13;

        if (novoMes == 0) {
            incrementaAno();
            novoMes++;
        }

        if (dia == limiteDiasMes(mes)) {
            dia = limiteDiasMes(novoMes);
        }

        mes = novoMes;
    }

    public void incrementaDia() {
        int novoDia = (dia + 1) % (limiteDiasMes(mes) + 1);

        if (novoDia == 0) {
            incrementaMes();
```

```
                    novoDia++;
                }

                dia = novoDia;
        }
        @Override
        public String toString() {
                return this.dia + "/" + this.mes + "/" + this.ano;
        }
}
```

10

RESUMO

O objetivo deste módulo foi apresentar o conceito relativo aos tipos de dados abstratos da linguagem de programação Java com seus respectivos limites de representação (operações e valores). Além disso, esse módulo apresentou, para os referidos tipos, os conceitos relativos ao estado de uma variável e as transições de estado de uma variável. Vimos que uma classe define os campos (atributos) ou os procedimentos (métodos) que representam um conceito ou ideia qualquer. Portanto, é possível ter uma classe que defina apenas os campos sem qualquer procedimento, ter uma classe que defina apenas os procedimentos sem qualquer campo e ter uma classe que defina ambos, campos e procedimentos, o modelo mais comum. Contudo, os outros modelos possuem suas aplicações e são plenamente possíveis de existir. Os campos, quando existem, podem ser de outros tipos de dados primitivos ou abstratos e os procedimentos, quando também existem, geralmente, realizam operações que manipulam os campos.

Um objeto representa um único valor válido que é composto pelo conjunto de outros possíveis valores, que, combinados, continuam sendo válidos. Os valores assumidos pelas variáveis de instância devem ser sempre permitidos e nunca devem ser proibidos. Caso as variáveis assumam valores proibidos, isso significa que o estado de objeto é inválido e seu tipo de dado abstrato (classe) possui um erro de lógica (algoritmos e estruturas) em sua concepção.

UNIDADE I – JAVA PARA ANDROID

MÓDULO 3 – O SISTEMA DE TIPOS DE DADOS DO JAVA – PARTE III

01

1 - ENCAPSULAMENTO

Uma expressão idiomática é um conjunto de dois ou mais termos/palavras que se caracterizam por não ser possível identificar o seu significado mediante o sentido literal dos termos que constituem a

expressão. Desta forma, em geral, é muito difícil ou mesmo impossível traduzir as expressões idiomáticas para outras línguas.

Localizado em algum ponto entre a compreensão da sintaxe de uma linguagem de programação e um bom projeto orientado a objeto e padrões (independente de linguagem), está o uso idiomático desta linguagem. Um **programador idiomático** utiliza código consistente para expressar conceitos semelhantes e, ao fazê-lo, produz programas de fácil compreensão de modo que o ambiente de tempo de execução seja utilizado da melhor forma, evitando as "pegadinhas" que existem em qualquer linguagem de programação.

As expressões idiomáticas muitas vezes estão associadas a gírias, jargões ou contextos culturais específicos a certos grupos de pessoas que se distinguem pela classe, idade, região, profissão ou outro tipo de afinidade. Muitas destas expressões têm existência curta ou ficam restritas ao grupo onde surgiram, enquanto algumas outras resistem ao tempo e acabam sendo utilizadas de forma mais abrangente, extrapolando o contexto original. Neste último caso, a origem histórica do seu significado muitas vezes se perde de todo ou fica limitada a um, relativamente, pequeno grupo de usuários da língua. Contudo, muitas pessoas, por mais que não saibam a origem histórica de tal expressão, frequentemente as repetem por simples hábito adquirido mediante a observação e transmissão das práticas, de geração em geração, executadas por integrantes, geralmente ancestrais, do grupo ao qual pertencem.

02

Desenvolvedores limitam a visibilidade de campos/atributos e procedimentos/métodos de objeto para criar **encapsulamento**, ou seja, a noção de que um objeto nunca deve revelar detalhes sobre si mesmo aos quais não tem a intenção de dar suporte.

A interface, às vezes abreviada de API, de um objeto consiste em todos os seus recursos (atributos ou métodos) que foram programados como sendo públicos. Isso significa que os referidos recursos podem ser utilizados e acessados por qualquer outra parte do seu código fonte.

Ao aplicar cuidadosamente o **encapsulamento**, um desenvolvedor mantém detalhes da implementação de um objeto oculto dos códigos que o utilizam. Tal controle e proteção produzem programas mais flexíveis e permitem ao desenvolvedor de um objeto alterar posteriormente sua implementação, sem provocar alterações em cascata no código invocado.

Separar o programa em partes, o mais isoladas possível, é a ideia essencial do encapsulamento.

Tornar o *software* mais flexível, fácil de modificar e expandir é o objetivo desse procedimento.

03

O encapsulamento protege o acesso direto aos atributos de uma instância fora da classe onde estes foram declarados. Esta proteção consiste em se usar modificadores de acesso (*private*, por exemplo) mais restritivos sobre os atributos definidos na classe. Depois devem ser criados métodos para manipular de **forma indireta** os atributos da classe.

Observe que o fato de um atributo ser ***private*** não significa que o mesmo não possa ser acessado sob nenhuma hipótese. O uso do referido modificador tem como finalidade restringir o acesso e não impedir o acesso. A restrição tem como finalidade controlar os possíveis valores que o referido atributo poderá assumir. Ou seja, o uso do modificador de acesso *private* em atributos impede o **acesso direto** aos atributos como uma tentativa de garantir a integridade de seus valores.

Se mesmo assim existirem valores não permitidos nos referidos atributos, a grande vantagem do uso de tais modificadores ainda existe, uma vez que a procura pelo trecho de código que permitem tais valores se limitam apenas à classe onde o referido atributo foi definido. Portanto, o encapsulamento é recurso extremamente útil tanto para o desenvolvimento do *software* como também para a manutenção do mesmo.

Os códigos fontes abaixo foram implementados para serem, eles próprios autoexplicativos. Deste modo, a simples leitura do código fonte deve ser suficiente para demonstrar todo o seu significado não necessitando para tanto qualquer comentário além daqueles já presentes no início de cada uma das classes/interfaces.

Qualquer dúvida deve ser debatida no respectivo fórum da disciplina.

04

Como forma de exemplificar o conceito de encapsulamento, observe o exemplo a seguir, que abstrai o tempo de um dia solar médio. Historicamente, até o ano de 1954, o segundo era entendido como 1/86400 de um dia solar médio (ou 1/3600 de uma hora, ou 1/60 de um minuto), sendo assim definido em relação às dimensões e a rotação da Terra.

O referido exemplo possui uma interface que determina quais devem ser os métodos públicos de todas as classes que porventura, a implementem. No caso deste exemplo, tem-se duas classes, **Tempo.java** e **NovoTempo.java**, que implementam a mesma interface **ITempo.java**. Observe que a interface de ambas as classes java são idênticas, contudo, suas implementações são distintas.

Considere também que o exemplo é apenas uma das várias formas possíveis de se demonstrar o referido assunto, não devendo ser, portanto, considerado como a única forma de representação. Ou seja, a forma apresentada abaixo é apenas uma dentre as várias formas possíveis de representação do referido conceito.

05

1.1 Interface ITempo

```
/**
 *
 * Historicamente, até o ano de 1954, o segundo era entendido como 1/86400 de um
 * dia solar médio (ou 1/3600 de uma hora, ou 1/60 de um minuto), sendo assim
 * definido em relação às dimensões e a rotação da Terra.
 *
 * Essa interface abstrai algumas operações/métodos de um dia solar médio no
calendário
 * gregoriano.
 *
 * @author Guilherme Veloso
 *
 */

package br.aiec;

public interface ITempo {

	public abstract int getHora();

	public abstract void setHora(int hora);

	public abstract int getMinuto();
```

```
	public abstract void setMinuto(int minuto);

	public abstract int getSegundo();

	public abstract void setSegundo(int segundo);

	public abstract boolean ehUltimoSegundoDoTempo();

	public abstract void incrementaHora(int hora);

	public abstract void incrementaMinuto(int minuto);

	public abstract void incrementaSegundo(int segundo);

}
```

06

1.2 Classe Tempo

```
package br.aiec;

/**
 *
 * Historicamente, até o ano de 1954, o segundo era entendido como 1/86400 de um
 * dia solar médio (ou 1/3600 de uma hora, ou 1/60 de um minuto), sendo assim
 * definido em relação às dimensões e a rotação da Terra.
 *
 * Essa classe abstrai uma das possíveis formas de representação do tempo
 * (hora:minuto:segundo) de um dia solar médio no calendário gregoriano.
 *
 * @author Guilherme Veloso
 *
 */
public class Tempo implements ITempo {

	private int hora;
	private int minuto;
	private int segundo;

	public Tempo(int hora, int minuto, int segundo) {
		setHora(hora);
		setMinuto(minuto);
		setSegundo(segundo);
	}
```

```
	public Tempo(ITempo tempo) {
		this(tempo.getHora(), tempo.getMinuto(), tempo.getSegundo());
	}

	@Override
	public int getHora() {
		return hora;
	}

	@Override
	public void setHora(int hora) {
		if (hora >= 0 && hora <= 23) {
			this.hora = hora;
		}
	}

	@Override
	public int getMinuto() {
		return minuto;
	}

	@Override
	public void setMinuto(int minuto) {
		if (minuto >= 0 && minuto <= 59) {
			this.minuto = minuto;
		}

	}

	@Override
	public int getSegundo() {
		return segundo;
	}

	@Override
	public void setSegundo(int segundo) {
		if (segundo >= 0 && segundo <= 59) {
			this.segundo = segundo;
		}
	}

	@Override
	public boolean ehUltimoSegundoDoTempo() {
		return this.hora == 23 && this.minuto == 59 && this.segundo == 59;
	}
```

```
	@Override
	public void incrementaHora(int hora) {
		int h = this.hora + hora;

		if (h > 23) {
			h = h % 24;
		}

		this.hora = h;
	}

	@Override
	public void incrementaMinuto(int minuto) {

		int m = this.minuto + minuto;

		if (m > 59) {
			incrementaHora(m / 60);
			m =  m % 60;
		}

		this.minuto = m;
	}

	@Override
	public void incrementaSegundo(int segundo) {
		int s = this.segundo + segundo;

		if (s > 59) {
			incrementaMinuto(s / 60);
			s =  s % 60;
		}

		this.segundo = s;
	}

	@Override
	public String toString() {
		return String.format("%02d:%02d:%02d", this.hora, this.minuto,
				this.segundo);
	}
}
```

07

1.3 Classe NovoTempo

```
package br.aiec;

/**
 *
 * Historicamente, até o ano de 1954, o segundo era entendido como 1/86400 de um
 * dia solar médio (ou 1/3600 de uma hora, ou 1/60 de um minuto), sendo assim
 * definido em relação às dimensões e a rotação da Terra.
 *
 * Essa classe abstrai uma das possíveis formas de representação do tempo
 * (hora:minuto:segundo) de um dia solar médio no calendário gregoriano.
 *
 * @author Guilherme Veloso
 *
 */

public class NovoTempo implements ITempo {

	private int totalDeSegundosDeUmDia;

	private void incrementaSegundosTotais(int valor) {
		if (valor > 0) {
			totalDeSegundosDeUmDia += valor;
		}
	}

	private void setSegundosTotais(int valor) {
		if (valor > 0) {
			totalDeSegundosDeUmDia = valor;
		}
	}

	public NovoTempo(int hora, int minuto, int segundo) {
		incrementaHora(hora);
		incrementaMinuto(minuto);
		incrementaSegundo(segundo);
	}

	public NovoTempo(ITempo tempo) {
		this(tempo.getHora(), tempo.getMinuto(), tempo.getSegundo());
	}

	@Override
	public int getHora() {
		return totalDeSegundosDeUmDia / 3600 % 24;
	}

	@Override
```

```
	public void setHora(int hora) {
		setSegundosTotais(hora * 3600 + getMinuto() * 60 + getSegundo());
	}

	@Override
	public int getMinuto() {
		return totalDeSegundosDeUmDia / 60 % 60;
	}

	@Override
	public void setMinuto(int minuto) {
		setSegundosTotais(getHora() * 3600 + minuto * 60 + getSegundo());
	}

	@Override
	public int getSegundo() {
		return totalDeSegundosDeUmDia % 60;
	}

	@Override
	public void setSegundo(int segundo) {
		setSegundosTotais(getHora() * 3600 + getMinuto() * 60 + segundo);
	}

	@Override
	public boolean ehUltimoSegundoDoTempo() {
		return totalDeSegundosDeUmDia == 86399;
	}

	@Override
	public void incrementaHora(int hora) {
		incrementaSegundosTotais(hora * 3600);
	}

	@Override
	public void incrementaMinuto(int minuto) {
		incrementaSegundosTotais(minuto * 60);
	}

	@Override
	public void incrementaSegundo(int segundo) {
		incrementaSegundosTotais(segundo);
	}

	@Override
	public String toString() {
		int hh = getHora();
		int mm = getMinuto();
		int ss = getSegundo();

		return String.format("%02d:%02d:%02d", hh, mm, ss);
	}
```

```
}
```

08

1.4 Classe MainTesteTempo

```
package br.aiec;

/**
 * Essa classe tem por objetivo apenas TESTAR o funcionamento da classe
 * Tempo.java de modo a validar se a mesma garante: integridade, coerência,
 * coesão, autonomia e funcionalidade.
 *
 * @author Guilherme Veloso
 *
 */

public class MainTesteTempo {

	public static void main(String[] args) {

		int valorHora = 0;
		int valorMinuto = 0;
		int valorSegundo = 0;

		exibirTempo(new Tempo(valorHora, valorMinuto, valorSegundo));
	}

	private static void exibirTempo(ITempo t) {
		int i = 0;

		do {
			System.out.println(t);
			t.incrementaSegundo(1);
		} while (i++ < 86400);

	}
}
```

09

Para utilizar a classe **NovoTempo.java**, altere a instrução da classe **MainTeste.java** (parâmetro do método exibirTempo) que cria o referido objeto, conforme abaixo:

```
new NovoTempo(valorHora, valorMinuto, valorSegundo);
```

Observe que a referida instrução altera o objeto que está sendo utilizado, contudo as interfaces permanecem inalteradas bem como sua funcionalidade.

O uso do modificador de acesso “private” impede o acesso direto aos campos dessa versão da classe Tempo e da classe NovoTempo. O uso de métodos getters e setters públicos (public) fornecem ao desenvolvedor a oportunidade de alterar a forma como o objeto Tempo e NovoTempo manipulam seus respectivos dados.

Por exemplo, na classe Tempo, o método “getHora” é o retorno direto de um valor armazenado em um atributo específico denominado de “hora”. Já na classe NovoTempo, o mesmo método “getHora” é o retorno de um cálculo matemático que extrai do único atributo da classe, quantas horas existem contidas naquele respectivo valor. Independentemente da forma, Tempo ou NovoTempo, o resultado esperado é o mesmo. O mesmo raciocínio pode ser aplicado aos métodos que realizam incremento como: “incrementaHora”, “incrementaMinuto” e “incrementaSegundo”.

Os métodos encapsuladores fornecem flexibilidade futura, enquanto que o acesso direto a um campo/atributo significa que todo código que o utilize terá de ser alterado, caso a semântica ou sintaxe de um determinado campo/atributo sofra mudanças.

A regra geral do encapsulamento para classes que proporcionam a criação de objetos com representação de estado é de que **os atributos, em sua maioria sejam privados e os métodos, em sua maioria, sejam públicos.** Obviamente que como regra geral, a mesma não se aplica a todos os casos. Cada caso deve ser analisado em suas várias dimensões de modo que seja possível representá-lo computacionalmente de maneira coerente.

10

2 - HERANÇA E POLIMORFISMO

Esses dois conceitos, herança e polimorfismo, juntamente com os conceitos de abstração e encapsulamento são responsáveis por definirem os princípios da orientação objeto.

A **herança**, além de princípio, é também um tipo de relacionamento cujo principal objetivo é criar mecanismos que permitam o uso/reuso de código.

Em contrapartida, o uso indiscriminado do referido relacionamento pode comprometer seriamente o encapsulamento, uma vez que os recursos (atributos e métodos) comumente costumam utilizar o modificador de acesso “protected”. O uso do referido modificador deve ser feito com cautela e critério

uma vez que o seu uso de forma indiscriminada enfraquece o encapsulamento das classes, pois aumenta a visibilidade das funcionalidades dividindo a responsabilidade com a hierarquia de classes filhas.

11

O **polimorfismo** permite que referências de tipos de classes mais abstratas representem o comportamento das classes concretas que referenciam. Assim, é possível tratar vários tipos de maneira homogênea (através da interface do tipo mais abstrato). Essa funcionalidade é caracterizada quando duas ou mais classes distintas têm métodos de mesma estrutura (tipo de retorno e assinatura), de forma que uma função possa utilizar um objeto de qualquer uma das classes polimórficas, sem necessidade de tratar de forma diferenciada conforme a classe do objeto.

Isso pode ser feito tanto pelo uso do relacionamento de herança como pelo relacionamento de realização/implementação. Por exemplo, a interface ITempo define uma operação (método abstrato) cuja estrutura é **public boolean** ehUltimoSegundoDoTempo(). A referida operação é implementada em todas as classes que realizam a interface "ITempo", ou seja, no nosso exemplo, as classes "Tempo" e "NovoTempo" possuem implementações para a referida operação. Observem que a operação (estrutura) é a mesma, contudo o método (comportamento) é diferente.

Apesar de parecerem iguais, os métodos trabalham de forma diferente, pois as classes "Tempo" e "NovoTempo" implementam a mesma ideia de duas formas distintas. Deste modo, na classe "MainTesteTempo" o método "exibirTempo" demonstra o uso de um comportamento polimórfico, uma vez que o parâmetro do referido método é do tipo da interface "ITempo". Ou seja, o método aceita como parâmetros objetos das classes "Tempo" ou "NovoTempo" sem qualquer tipo de problema, pois a interface "ITempo" padroniza a estrutura de interface das referidas classes "Tempo" e "NovoTempo".

12

RESUMO

O objetivo deste módulo foi apresentar o conceito relativo ao encapsulamento e suas formas de implementação por meio de um uso conjunto entre modificadores de acesso (public, protected e private) e as relações de realização/implementação ou herança. Vimos que a herança, além de princípio, é também um tipo de relacionamento cujo principal objetivo é criar mecanismos que permitam o uso/reuso de código. O polimorfismo permite que referências de tipos de classes mais abstratas representem o comportamento das classes concretas que referenciam. Assim, é possível tratar vários tipos de maneira homogênea (através da interface do tipo mais abstrato). Essa funcionalidade é caracterizada quando duas ou mais classes distintas têm métodos de mesma estrutura (tipo de retorno e assinatura), de forma que uma função possa utilizar um objeto de qualquer uma das classes polimórficas, sem necessidade de tratar de forma diferenciada conforme a classe do objeto.

UNIDADE I – JAVA PARA ANDROID

MÓDULO 4 – O SISTEMA DE TIPOS DE DADOS DO JAVA – PARTE IV

01

1 - COMPOSIÇÃO DE OBJETOS

Enquanto a extensão de classes em Java oferece aos desenvolvedores flexibilidade significativa na capacidade de redefinir aspectos dos objetos à medida que eles são utilizados em contextos diferentes, é necessária uma quantidade considerável de experiência para que se possa utilizar adequadamente classes e interfaces.

De um modo ideal, desenvolvedores procuram criar seções de código tolerantes a alterações que ocorram com o passar do tempo e que possam ser reutilizados no maior número possível de contextos distintos, em múltiplas aplicações, ou até mesmo como bibliotecas. Ao programar nesses moldes, é possível reduzir significativamente o número de *bugs* bem como o tempo necessário para corrigir aqueles que por ventura apareçam no futuro. Programação modular, encapsulamento, polimorfismo, separação de interesses/responsabilidades, dentre outros, são estratégias essenciais para maximização da reutilização e da estabilidade do código.

No desenvolvimento orientado a objeto, uma consideração fundamental de projeto se relaciona à decisão de quais tipos de relacionamentos serão utilizados entre as diversas classes de forma que seja possível reutilizar código preexistente. De modo conjunto aos exemplos apresentados nos módulos anteriores, iremos criar mais uma classe de modo que seja demonstrado um outro relacionamento denominado de **composição**, o qual é constantemente utilizado por qualquer projeto orientado a objeto.

02

A **composição de objetos** é uma maneira de se combinar objetos simples ou complexos para produzir funcionalidades ainda mais simples ou complexas.

É importante observar que dois pontos de vistas devem ser considerados nesta análise que envolve a dimensão complexidade:

- do ponto de vista, por exemplo do **programador experiente**, que está desenvolvendo todas essas funcionalidades, a essência que representa tais funcionalidades foi dividida em várias

partes íntegras, coerentes, coesas, reusáveis, seguras e autônomas de modo a facilitar o seu uso para qualquer fim.

- do ponto de vista, por exemplo dos **usuários**, que inclusive podem ser outros programadores menos experientes, a finalidade é de sempre procurar criar mecanismos que simplifiquem o seu uso, isolado ou em conjunto, de tais funcionalidades.

No exemplo que veremos a seguir, a classe "Relogio.java" se relaciona com as estruturas (classes e interfaces) dos módulos anteriores, Data e ITempo (ou Tempo ou NovoTempo), por meio de uma **relação de composição**. Isso significa que a classe que representa o conceito denominado de Relógio é uma combinação (composição) de outros dois tipos de dados abstratos, Data e ITempo, que combinados, implementam a ideia que conhecemos pelo nome de **relógio**.

A execução do referido código da classe "Relogio.java" depende das implementações presentes nos módulos anteriores. Portanto, somente será possível executar a classe "Relogio.java" se as classes/interfaces dos módulos anteriores (2 e 3) estiverem todas presentes no mesmo pacote (package) Java.

03

1.1- Classe Relogio.java

Observe ainda que, para implementar a relação de composição, faz-se necessário seguir, obrigatoriamente, duas regras:

1) Declarar atributos (tipos abstratos) privados sem expô-los sob nenhuma hipótese;

2) Inicializar os objetos, cujas referências são os atributos privados do item anterior, de modo que o tempo de vida destas instâncias internas esteja, obrigatoriamente, determinado pelo tempo de vida dos objetos da classe Relogio.

```
package br.aiec;

/**
 *
 * A referida classe esabelece um relação de composição com as classe Data e com a
 * interface ITempo (ou Tempo ou NovoTempo).
 *
 * @author Guilherme Veloso
 *
 */
public class Relogio {
	private Data data;
	private ITempo tempo;
```

```
	public Relogio(Data data, ITempo tempo) {
		this.data = new Data(data);
		this.tempo = new Tempo(tempo);
	}

	/**
	 * O método tictac simula o tictac do relógio que a cada percurso incrementa
o valor
	 * do tempo em 1 segundo.
	 */
	public void tictac(){

		if(tempo.ehUltimoSegundoDoTempo()){
			data.incrementaDia();
		}

		tempo.incrementaSegundo(1);
	}

	@Override
	public String toString() {
		return data + " " + tempo;
	}
}
```

04

2 - CLASSE MAINTESTERELOGIO.JAVA

```
package br.aiec;

/**
 * Essa classe tem por objetivo apenas TESTAR o funcionamento da classe
 * Relogio.java de modo a validar se a mesma garante: integridade, coerência,
 * coesão, autonomia e funcionalidade.
 *
 * @author Guilherme Veloso
 *
 */

public class MainTesteRelogio {

	public static void main(String[] args) {

		Data d = new Data(1, 1, 2000);
		ITempo t = new Tempo(0, 0, 0);

		Relogio r = new Relogio(d, t);
```

```
            //O referido laco irá girar (86400 * 366) que corresponde à
quantidade de segundos existentes em um ano bissexto (ano 2000) do calendário
gregoriano


            int i = 0;
            int max = (86400 * 366);

            do {
                  System.out.println(r);
                  r.tictac();
            } while (i++ < max);
      }
}
```

05

Um ponto importante a se observar é que a criação da classe "Relogio" depende de outros dois tipos de dados abstratos (classes) que são compostas (combinadas) para representar a referida ideia de contar o tempo.

A simplicidade na implementação da classe Relogio se deve ao fato das referidas implementações dos módulos anteriores terem utilizado um conceito importante denominado **divisão de responsabilidades**.

A responsabilidade de uma classe representa o conjunto de ações que ela pode desempenhar e o conjunto de informações que ela pode ser solicitada a fornecer.

Com isso, a classe "Relogio", por meio de um outro conceito denominado **delegação**, consegue transferir a responsabilidade pela execução do processamento às classes devidamente responsáveis por tal procedimento. Além disso, com a delegação, podemos minimizar a interdependência de dois ou mais objetos e maximizar a flexibilidade futura.

06

3 - CLASSE OBJECT E SEUS MÉTODOS

A classe Java *Object* – **java.lang.Object** – é a raiz de todas as outras classes escritas na referida linguagem de programação, seja de forma direta ou indireta, seja de forma explicita ou implícita. Portanto, todo objeto em Java é do tipo *Object*.

A classe *Object* define as implementações de diversos comportamentos essenciais comuns a todos os objetos, como por exemplo: equals, toString, hashCode, wait, notify, dentre outros.

O método "**toString**" é a forma pela qual um objeto cria uma representação em string de si mesmo, ou seja, uma forma de representar por meio de string a parte do seu estado que deve significar a materialização do conceito abstrato.

Um uso interessante do "toString" é na concatenação de strings, uma vez que qualquer objeto pode ser concatenado a uma string. Contudo, a implementação de Object do método "toString" retorna uma string de pouca utilidade, baseado na localização do objeto na heap de execução do processo que contêm a instância do referido objeto. Deste modo, sobrescrever o "toString" é uma prática considerada excelente por facilitar a depuração do código, quando necessário. Observe que os exemplos demonstrados (Data.java, Tempo.java, NovoTempo.java e Relogio.java) ao longo desta disciplina, fazem o uso de tal prática.

07

Os métodos "**equals**" e "**hashCode**" nada mais são que a forma pela qual pode-se verificar se um objeto é "o mesmo que" outro.

A definição do método "equals" deve seguir uma implementação adequada que contemple as seguintes operações:

Reflexiva

```
x.equals(x)
```

Simétrica

```
x.equals(y) == y.equals(x)
```

Transitiva

```
( x.equals(y) && y.equals(z) ) == x.equals(z)
```

Consistente

Se x.equals(y) é verdadeiro em algum momento da vida do objeto, então será sempre verdadeiro, desde que x e y não se alterem.

A compreensão plena desses conceitos é útil e surpreendentemente complexa. Um equívoco na implementação de tais métodos podem gerar resultados inesperados durante a execução do seu programa.

08

O método “**hashCode**” tem por objetivo ser uma verificação aproximada da igualdade de dois objetos.

Muitas bibliotecas comparam os códigos hash de dois objetos antes de compararem os mesmo objetos utilizando-se o método “equals”. Isso ocorre basicamente em função da velocidade de processamento, ou seja, comparar dois códigos hash deve ser bem mais rápido que comparar dois estados de forma plena.

Percebe-se então que a finalidade do método “hashCode” é oferecer um recurso de processamento mais rápido e semelhante ao do método “equals”. Portanto, “hasCode” e “equals” devem ser considerados um par, o que significa que a sobrescrita de um deles implicará, obrigatoriamente, a sobrescrita do outro. Não realizar tal procedimento é quebrar a regra do contrato geral do método hashCode da classe Object.

Contudo não se esqueça de que a implementação de ambos, apesar de semelhantes, devem seguir, cada um o seu próprio propósito.

Uma boa função de hash tende a produzir códigos de hash diferentes para objetos distintos. O ideal é que uma função de hash distribua qualquer coleção considerável de instâncias diferentes uniformemente por todos os valores de hash possíveis.

09

Apesar de ser praticamente impossível atingir esse objetivo plenamente, é perfeitamente possível chegar a um resultado aproximado com a receita simples abaixo:

1. Armazene um valor constante, de preferência um número primo, diferente de zero, por exemplo o 17, em uma variável do tipo int chamada 'r';

2. Para cada campo/atributo 'f' significativo de seu objeto, isto é, cada campo/atributo levado em consideração pelo método equals, faça o seguinte:

   a. calcule um código hasc 'c' do tipo int para cada atributo:

      I. se o campo/atributo for boolean, calcule (f ? 1 : 0);

      II. se o campo/atributo for byte, char, short ou int, calcule (int) f;

      III. se o campo/atributo for um long, calcule (int) (f^(f>>>32));

      IV. se o campo/atributo for um float, calcule Float.floatToInBits(f);

V. se o campo/atributo for um double, calcule Double.doubleToLongBits(f) e, em seguida, encontre o hash do long resultante como no item 2.a.III;

VI. se o campo/atributo for um referência de objeto e o método equals dessa classe comparar o campo chamado equals de suas referências, chame também o hasCode de cada referência.

b. combine o código hash 'c' encontrado no item 2.a com a variável 'r' do item 1, como abaixo:

```
r = 31 * r + c;
```

A seguir, os métodos de exemplo que devem ser introduzidos nos códigos das respectivas classes desenvolvidas até o presente momento ao longo desta disciplina.

10

3.1- Métodos da classe Tempo.java

```
@Override
	public boolean equals(Object obj) {

		if(obj == null){
			return false;
		}

		if (obj == this){
			return true;
		}

		if( ! (obj instanceof Tempo)){
			return false;
		}

		Tempo t = (Tempo) obj;

		return this.hora == t.hora &&
			this.minuto == t.minuto &&
			this.segundo == t.segundo;
	}

	@Override
	public int hashCode() {
		int r = 17;
		r = 31 * r + this.hora;
		r = 31 * r + this.minuto;
		r = 31 * r + this.segundo;
```

```
		return r;
	}
```

11

3.2 Métodos da classe NovoTempo.java

```
@Override
	public boolean equals(Object obj) {

		if(obj == null){
			return false;
		}

		if (obj == this){
			return true;
		}

		if( ! (obj instanceof NovoTempo)){
			return false;
		}

		NovoTempo t = (NovoTempo) obj;

		return this.totalDeSegundosDeUmDia == t.totalDeSegundosDeUmDia;
	}

	@Override
	public int hashCode() {
		int r = 17;
		r = 31 * r + this.totalDeSegundosDeUmDia;
		return r;
	}
```

12

3.3- Métodos da classe Data.java

```
@Override
	public boolean equals(Object obj) {

		if(obj == null){
			return false;
		}

		if (this == obj){
			return true;
		}
```

```
		if( ! (obj instanceof Data)){
			return false;
		}

		Data d = (Data) obj;

		return this.dia == d.dia && this.mes == d.mes && this.ano == d.ano;
	}

	@Override
	public int hashCode() {
		int r = 17;
		r = 31 * r + this.dia;
		r = 31 * r + this.mes;
		r = 31 * r + this.ano;
		return r;
	}
```

13

3.4 Métodos da classe Relogio.java

```
@Override
	public boolean equals(Object obj) {

		if (obj == null) {
			return false;
		}

		if (obj == this) {
			return true;
		}

		if (!(obj instanceof Relogio)) {
			return false;
		}

		Relogio r = (Relogio) obj;

		return this.data.equals(r.data) && this.tempo.equals(r.tempo);
	}

	@Override
	public int hashCode() {
		int r = 17;
		r = 31 * r + this.data.hashCode();
		r = 31 * r + this.tempo.hashCode();
		return r;
	}
```

14

4 - MODIFICADORES FINAL, STATIC E ABSTRACT

Os modificadores a seguir podem ser usados em conjunto com os modificadores de acesso (private, protected e public) provendo, assim, outros comportamentos.

4.1 Modificador de acesso *static*

A palavra reservada *static* serve:

- na declaração de um campo/atributo dentro de uma classe, para se criar uma área de memória compartilhada por todas as instâncias de objetos da referida classe. Ou seja, o atributo criado será comum a todas as instâncias e quando seu conteúdo é modificado por qualquer uma das instâncias, a modificação é percebida por todas as demais;
- na declaração de um método que deve ser acessado diretamente na classe e não nas suas instâncias.

15

4.2 Modificador abstract

A palavra reservada *abstract* serve para:

- declarar métodos abstratos (operações), ou seja, métodos que deverão ser desenvolvidos/implementados nas subclasses. Quando a classe que contiver métodos abstratos for herdada, os referidos métodos deverão ser implementados, caso contrário, a classe que estendeu deverá ser declarada como abstrata.
- declarar classes abstratas que se desenvolvem numa(s) subclasse(s). Classes abstratas são aquelas que não estão totalmente implementadas/descritas. Uma classe abstrata não pode ser instanciada e é amplamente usada nas interfaces.
- Uma classe é considerada abstrata se contiver pelo menos um método abstrato. Um método abstrato tem definido apenas sua assinatura. Lembre-se que método abstrato não tem corpo/escopo definido.

- No caso da sintaxe da linguagem Java, caso o método tenha as chaves características {}, o mesmo não mais será considerado abstrato, embora não tenha código algum dentro das chaves.

16

4.3 Modificador final

A instrução *final* indica que a classe, método ou variável assim declarados têm uma única atribuição que se mantém constante, ou seja, não podem ser alterados no decorrer do processamento.

Este modificador declara o que chamamos, em programação, de constante.

17

RESUMO

O objetivo deste módulo foi continuar apresentando o conceito relativo ao encapsulamento com foco na reusabilidade. O módulo apresenta uma forma de implementação por meio de um uso conjunto entre modificadores de acesso (public, protected e private) e a relação de composição. Além disso, foi apresentada também a classe Object e alguns de seus métodos (equals, hashCode, toString) os demais modificadores: final, static e abstract.