UNIDADE 1 – INTRODUÇÃO À PROGRAMAÇÃO ORIENTADA A OBJETOS
MÓDULO 1 – CONCEITOS DA PROGRAMAÇÃO ORIENTADA A OBJETOS
01

1 - INTRODUÇÃO À ORIENTAÇÃO A OBJETOS

Um programa é um conjunto de instruções que são executadas pelo computador para executar uma determinada tarefa.

Na disciplina de algoritmos vocês viram que o algoritmo é uma forma de descrever os passos necessários para a execução de uma determinada tarefa.

O exemplo abaixo apresenta o algoritmo que lê a nota de 3 alunos e tira a média.

Exemplo 1_1_001: Ler a nota de 3 alunos, calcular e mostrar a média. Apresentar em seguida quantos alunos ficaram abaixo da média.

```
Declara vetor de reais notas[3];
Declara inteiros i, contador;
Declara reais media, soma;

Inicio

        Soma←0;
        contador←0;

        para i de 1 a 3 faça
          mostrar "digite a nota "+i+":";
          ler(notas[i])
          soma←soma+nota[i]
        fim_para

        media←soma/3;

        para i de 1 até 3 faça
          se(nota[i]<media) então
                contador←contador+1;
          fim_se
        fim_para
        mostrar "a média é "+media;
        mostrar "Existem "+contador+" alunos abaixo da média";

Fim
```

No exemplo acima, podemos ver que, primeiramente, foram definidas as variáveis que irão armazenar

os valores (notas, médias, contadores etc.) e depois é apresentada a execução de comandos propriamente dita.

02

Deste modo, vamos para um segundo exemplo de algoritmo que deve listar todos os números naturais primos até um n qualquer. Antes de prosseguir, procure pensar em: como você resolveria esse problema?

Pois bem, a primeira pessoa que se tem notícias que produziu tabelas de números primos foi uma pessoa chamada de Eratóstentes que viveu no terceiro século a.c (276 - 194) na Grécia antiga.

Eratóstenes criou um método, atualmente chamado de algoritmo, que possui a seguinte ideia: inicialmente ele escrevia um tabela com todos os números naturais de 1 à n, sendo que, neste caso, para simplificar o tamanho da tabela, 'n' estará valendo 120. Contudo, lembre-se que 'n' pode ser qualquer número natural maior que 1. Em seguida, escolhia o primeiro primo 'p' que, como já sabemos, é o número 2 e, consequentemente, eliminavam-se da lista todos os seus múltiplos. Observe que isso elimina metade dos números no intervalo de 1 à n escolhido. Depois escolhia o próximo número 'p' que ainda não tinha sido eliminado e, consequentemente, também eliminavam-se todos os seus múltiplos. Esse procedimento era executado até que ≤ n, e, consequentemente, a tabela contivesse apenas números primos. Observe que essa condição de parada é verdadeira pois não é preciso verificar todos os número no intervalo de 1 à 'n' uma vez que todos os números não primos são compostos e, consequentemente, podem ser reescritos em função de seus fatores primos. Mais tarde, esse algoritmo foi batizado como sendo o "crivo de Eratóstenes".

	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90
91	92	93	94	95	96	97	98	99	100
101	102	103	104	105	106	107	108	109	110
111	112	113	114	115	116	117	118	119	120

Prime numbers

A partir desse procedimento podemos simplificar a descobertas de primos usando o lema : Se um número natural n > 1 não é divisível por nenhum primo p tal que $p^2 \leq n$, então ele é primo. Este lema fornece um teste de primalidade, pois, para verificar se um dado número n é primo, basta verificar que não é divisível por nenhum p que não supere √n.

Este algoritmo inventado por Eratóstenes pode ser escrito também sob a forma de portugol, conforme demonstrado no exemplo abaixo:

```
Declara vetor de reais valores[120];
Declara inteiro maximo=120;
Declara inteiro i;

Inicio

para i de 0 a maximo faça
valores[ i ] ← i
i ← i + 1
fim_para

valores[0] ← 0
valores[1] ← 0
i ← 2

enquanto i^2 <= máximo faça

se valores[ i ] != 0 então
eliminarPrimos( valores[ i ] )
fim_se

i ← i + 1

fim_enquanto

Fim

Função eliminarPrimos(inteiro i)

Declara inteiro j

j ← 2 * i;

enquanto j <= máximo faça
valores[ j ] ←0
j ← j + i;
fim_enquanto

fim_eliminarPrimos
```

Observe que, tanto o algoritmo do exemplo anterior quanto este, são exemplos introdutórios da chamada **programação estruturada**, que consiste em organizar a ideia em termos de tipos/variáveis, estruturas de sequencia, decisão e repetição além de procedimentos/funções. Existe, portanto, uma separação muito bem definida entre dados (variáveis) e código (demais elementos).

03

Para entender as limitações do paradigma estruturado/procedural, iremos estudar um exemplo simples. Digamos que você faça parte de uma equipe de cinco programadores de uma fábrica de software. Essa fábrica foi contratada para desenvolver um site de compras on-line.

Em diversas partes do site será necessário identificar e validar o usuário que está acessando o site. Cada programador ficará encarregado de uma parte do site e todos precisarão identificar em algum momento do código qual é o usuário e validá-lo. Você já deve imaginar como seria muito difícil compartilhar o código de verificação do usuário entre todos os programadores. Se cada um dos programadores fizesse o seu próprio código seria mais problemático ainda, pois mesmo que todos fizessem um excelente trabalho seria muito difícil fazer a manutenção do código, caso fosse necessário alterar a forma de autenticação do usuário. Um outro problema é que essa verificação do usuário estaria espalhada em todo o código do site o que dificultaria ainda mais essas alterações. Outro problema é que existe uma chance muito grande do programador esquecer-se de validar o usuário causando problemas grandes para o cliente.

No exemplo acima, vimos que não há uma garantia que o usuário será sempre identificado e validado, pois não existe uma conexão forte entre dados e funcionalidade. Veremos que o paradigma orientado a objetos já possui algumas ideias que resolvem essas limitações e facilitam a programação e o compartilhamento do código com segurança e facilidade de manutenção. O que chamamos de paradigma de programação orientada a objetos, ou simplesmente POO, é uma forma complementar ao paradigma estruturado pois também trata de construir estruturas que permitam representar o mundo real tal como o conhecemos e percebemos no mundo virtual/tecnológico/digital. O detalhe adicional é que o POO permite escrever programas que se aproximam mais da forma como o homem moderno está habituado a pensar e raciocinar. Ou seja, o paradigma orientado a objeto nada mais é que um modelo utilizado atualmente pela ciência e pela industria quase em sua totalidade, inclusive àquelas ligadas a tecnologia, para produzir a imensa maioria dos produtos/serviços tais como os softwares, por exemplo, utilizados pela sociedade moderna.

04

Classes e Objetos

A orientação a objetos, como o próprio nome diz, implementa o conceito de **classes** e **objetos**.

CLASSE (Modelo Conceitual)

Podemos definir uma classe como sendo uma estrutura comum para um conjunto de objetos que possuem as mesmas características, comportamentos, relacionamentos e semântica.

Um classe também pode ser compreendida como sendo uma síntese/resumo de um objeto qualquer pois todo procedimento de abstração, em sua grande maioria, procura representar apenas uma parte dos elementos que estão sendo observados. É claro que o hábito mais comum de representar apenas parte de um todo não exclui a possibilidade de representar o todo de forma plena. Contudo esse último uso é extremamente raro em função, principalmente, da complexidade (corporativismo, segredos industriais, etc...) dos objetos e até mesmo por desconhecimento (época, tempo, espaço, cultura, crenças, dentre outros...) do que de fato significa o "todo" para um objeto qualquer.

OBJETO

A palavra **objeto** é usada em programação de forma bastante ampla servindo para representar um fato, uma coisa ou uma pessoa.

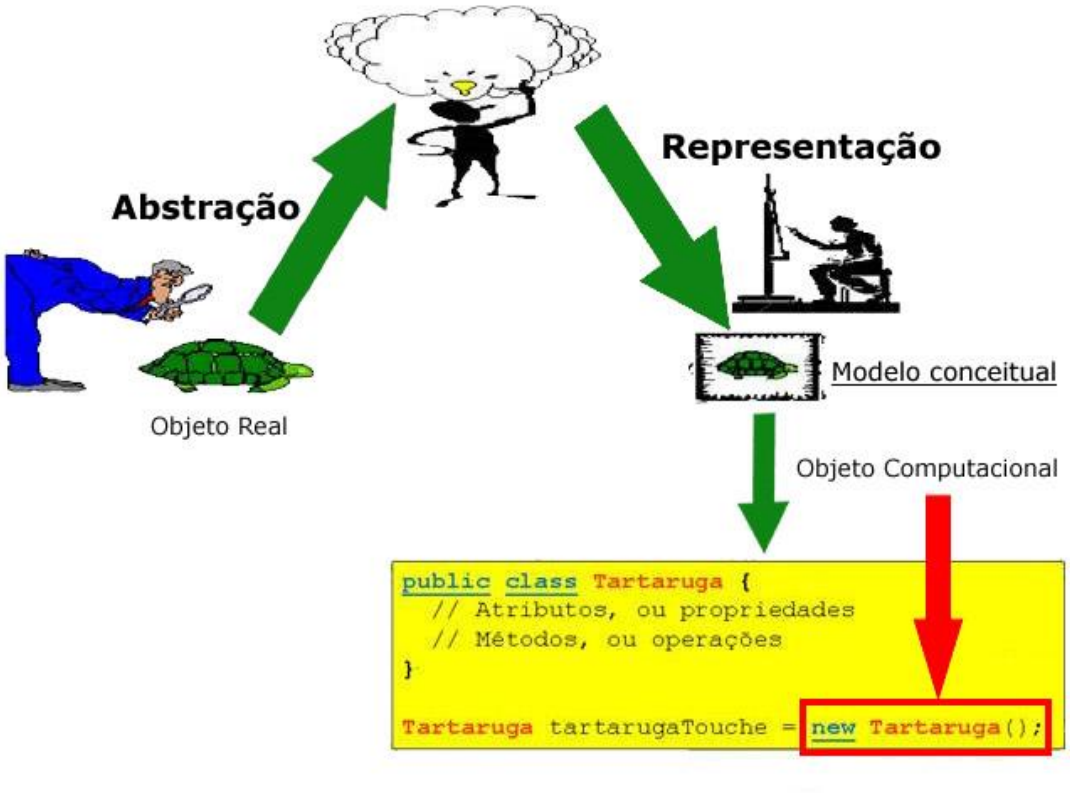
Deste modo a palavra/termo **objeto** é usada dentro deste paradigma com dois significados distintos e complementares conforme explicado abaixo:

- "**objeto real**": esse significado é usado de forma bastante ampla servindo para representar qualquer coisa ao nosso redor, sejam elas tangíveis ou intangíveis.

- "**objeto computacional**": esse significado é usado para se referir a instância de uma classe.

Deste modo, se utilizarmos como exemplo a definição anterior de **Classe** - "...classe como sendo um conjunto de objetos..." - é importante perceber que o termo objetos a qual a definição se refere é o objeto real uma vez que uma classe pode também ser compreendida como sendo um resumo/síntese de um objeto pertencente ao nosso mundo. Portanto, qualquer que seja o objeto real que esteja sendo observado, o procedimento de representação desta observação/abstração sempre irá representar uma quantidade de elementos menor ou igual aqueles elementos existentes no objeto real.

Isso posto, é importante ficar atento que a partir de agora usaremos apenas o termo **objeto** e, consequentemente, o significado adequado do termo somente será determinado pelo leitor no momento da leitura levando-se em consideração o contexto específico do texto. Além disso, os livros técnicos, científicos, artigos, provas dentre outras publicações/documentos também exigirão dos leitores essa habilidade de leitura pois dentro da literatura atual o termo também é usado da mesma forma que aqui nesta disciplina.



Para entender o que são classes e objetos vamos apresentar alguns exemplos.

Exemplo 1_1_003: Vamos descrever de forma genérica um cachorro. Um cachorro possui as seguintes características, que chamaremos daqui para frente de atributos:

Nome
Raça
Sexo
Data de nascimento
Peso
Cor

Além disso, o cachorro realiza as seguintes ações:

Latir
Correr
Sentar
Andar
Deitar

05

Prestem atenção que não há menção a um cachorro específico. O que temos é uma descrição geral, que chamaremos de **classe**, que serve para descrever qualquer cachorro.

Podemos usar essa classe para representar um cachorro específico, nesse caso temos um objeto dessa classe:

Nome: Totó
Raça: Pastor Alemão
Sexo: macho
Data de nascimento: 01/01/2010
Peso: 40 kg
Cor: bicolor

Como exemplo, podemos representar um outro:

Nome: Fred
Raça: Golden retriever
Sexo: macho
Data de nascimento: 01/01/2005
Peso: 40 kg
Cor: creme

06

Exemplo 1_1_004: vamos considerar que estamos fazendo um sistema de cadastro de clientes. Uma descrição genérica do cliente, que chamaremos de classe **cliente**, é mostrada abaixo:

Atributos da classe cliente:

Nome
Data de nascimento
CPF
Endereço
Cidade
CEP
E as principais funcionalidades para o cliente são:
Alterar endereço
Validar CPF
Alterar CEP

Novamente o que temos acima é uma descrição genérica do cliente. Com base nessa classe cliente podemos representar clientes criando objetos dessa classe, por exemplo:

Nome: Carlos Alberto Ferreira
Data de nascimento: 12/01/1996
CPF: 465.796.586-01
Endereço: rua do bonsucesso, 102

Cidade: Rio de Janeiro
CEP: 70374-020

Como vimos nesses dois exemplos, a classe é uma estrutura que consiste em uma descrição genérica de um conjunto de objetos que possui características e ações (ou funcionalidades).

07

1.2 – Mensagens e Métodos

1.2.1 – Métodos

Vimos que um objeto possui características e funcionalidades. As funcionalidades de um objeto são chamadas em **programação de métodos**. Primeiramente iremos falar de métodos.

Os métodos são as funcionalidades associadas a um objeto.

No exemplo da classe cachorro, nós descrevemos os seguintes métodos:

Latir()
Correr()
Sentar()
Andar()
Deitar()

Esses métodos são definidos na classe e todos os objetos dessa classe, nesse caso todos os cachorros terão esses métodos, ou seja, receberão essas funcionalidades. Vocês perceberam como isso é legal? Vamos voltar ao exemplo do jogo. Imaginem que criamos a classe **Nave**, conforme segue:

Atributos:

Tipo
Cor
Tamanho

Métodos:

Virar à direita
Virar à esquerda
Seguir em frente

08

E imaginem que a equipe de desenvolvimento já criou no código do programa diversos objetos da classe nave. Agora, digamos que vocês decidem alterar o jogo fazendo com que as naves incorporem armas. O que fazer?

É simples! Basta alterar a classe, ou seja, não é preciso alterar cada um dos objetos que foram criados:

Classe **Nave**

ANTES	DEPOIS
Atributos: Tipo Cor Tamanho **Métodos:** Virar à direita Virar à esquerda Seguir em frente	**Atributos:** Tipo Cor Tamanho Quantidade de tiros **Métodos:** Virar à direita Virar à esquerda Seguir em frente Atirar

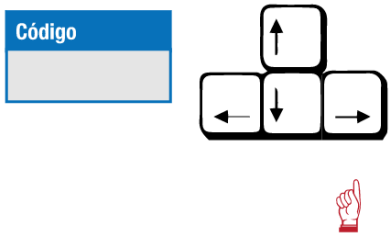
Viram como é fácil trabalhar com objetos?

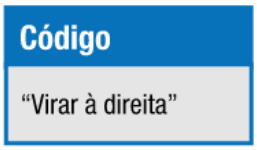
09

1.2.2 – Mensagens

Os objetos se "comunicam" por meio de mensagens. Todas as vezes que invocamos (chamamos) um método de um objeto estamos enviando uma mensagem ao objeto para ele fazer alguma coisa.

Vamos utilizar novamente o exemplo do programa do jogo intergaláctico. Se o programador quer que a nave vire para a direita quando o jogador pressionar a tecla da seta direita do teclado, então o programador deve garantir em seu código que será enviada uma mensagem "virar à direita" quando o jogador pressionar a tecla indicada.





10

1.3 – Herança

A herança é uma característica importante da programação orientada a objetos que permite que classes compartilhem atributos e métodos. Para ilustrar esse conceito, vamos considerar um programa para gerir uma escola. Sabemos que tanto os funcionários quanto os alunos devem ter um registro com as informações pessoais básicas (nome, endereço, telefone, cidade, UF), portanto podemos pensar em uma classe chamada **pessoa**, com os seguintes atributos:

Nome
Endereço
Telefone
Cidade
UF

E podemos pensar nos seguintes métodos:

Modificar nome
Modificar endereço
Modificar telefone
Modificar cidade
Modificar UF

11

Entretanto a escola precisa ter mais informações para os seus funcionários, como: cargo, cpf, conta corrente e agência. Como a classe Pessoa já possui atributos e métodos que são importantes também para os funcionários, então podemos criar uma classe Funcionário que **herda** os atributos e métodos da classe Pessoa:

Isso significa que a classe Funcionário herdará todos os atributos e métodos da classe pessoa e poderemos acrescentar outros atributos e outros métodos:

Atributos

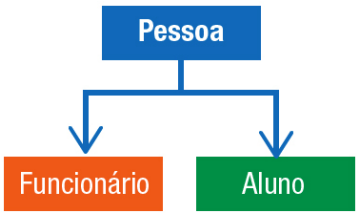
Nome
Endereço
Telefone
Cidade
UF
Cargo
CPF
Conta Corrente
Agência

E podemos pensar nos seguintes métodos:

Modificar nome
Modificar endereço
Modificar telefone
Modificar cidade
Modificar UF
Modificar CPF
Modificar dados bancários

12

Da mesma forma, devemos guardar informações específicas para os alunos, tais como: série, nome do pai, nome da mãe. Então podemos novamente usar o conceito de herança para criarmos uma classe, chamada Aluno, que herde os atributos e métodos da classe pessoa.



Isso significa que a classe Aluno herdará todos os atributos e métodos da classe pessoa e poderemos acrescentar outros atributos e outros métodos:

Atributos

Nome
Endereço

Telefone
Cidade
UF
Cargo
Série
Nome do pai
Nome da Mãe

E podemos pensar nos seguintes métodos:

Modificar nome
Modificar endereço
Modificar telefone
Modificar cidade
Modificar UF
Modificar nome do pai
Modificar nome da mãe

13

1.4 – Encapsulamento e ocultação de informação

Nós vimos que quando definimos uma classe, ela contém tudo que ela precisa para definir objetos, ou seja, ela possui atributos e os métodos necessários para que possa ser utilizada. Essa característica de autossuficiência é chamada de **encapsulamento**. E é uma característica é muito importante para facilitar a manutenção do código e também para garantir a segurança do código, pois qualquer alteração do comportamento da classe só poderá ser feita na própria definição da classe.

Vamos imaginar agora uma classe que represente um soldado que tenha como atributo a quantidade de balas e um método chamado Atirar. Quando o soldado atira, a quantidade de balas é reduzida. Nessa classe não é interessante que a quantidade de balas seja alterada por meio de mensagens.

Pois bem, quando definimos uma classe é possível especificar quais os atributos e métodos não serão acessíveis fora da classe. Isso significa que aqueles atributos são usados apenas internamente, bem como também é possível definir que alguns métodos só sejam chamados internamente. Isso é o que chamamos de **ocultação de informação**.

14

RESUMO

Nesse módulo apresentamos o conceito de orientação a objetos. Um paradigma de programação que tem como princípio a representação por meio de classes e objetos. As classes são descrições genéricas de objetos que contêm a definição de suas características, que são chamados de atributos, e seus

comportamentos, que são chamados de métodos. Os objetos representam pessoas, acontecimentos, fatos, ou qualquer coisa que tenha características e comportamentos. No caso, o objeto será uma representação específica de uma classe. Assim, vimos que se temos uma classe que represente cachorros em geral, então se utilizá-la para representar um cachorro específico, deveremos criar um objeto da classe cachorro com as características específicas desse cachorro, por exemplo.

Os objetos se comunicam por meio de mensagens. Quando queremos executar determinado método de um objeto, enviaremos para ele uma mensagem com essa solicitação.

Vimos também que a programação orientada a objetos implementa o conceito de herança. Esse conceito permite que uma classe herde atributos e métodos de outra classe mais genérica. Esse conceito facilita muito a programação, reduzindo código e facilitando a manutenção do mesmo.

A seguir iremos conhecer a linguagem de programação orientada a objetos, o Java, que utilizaremos para ilustrar os conceitos apreendidos nesse módulo e nos aprofundarmos na programação orientada a objetos.

UNIDADE 1 – INTRODUÇÃO À PROGRAMAÇÃO ORIENTADA A OBJETOS
MÓDULO 2 – JAVA E ESTRUTURAS DE PROGRAMAÇÃO
01

1 – HISTÓRICO DA LINGUAGEM JAVA

A linguagem Java foi criada em 1991 pela Sun Microsystem em um projeto chamado *The Green Project*. O objetivo original do projeto era criar uma plataforma interativa.

Em 1992, a equipe do projeto fez a primeira demonstração do projeto em um *Handheld* controlado remotamente. O anúncio oficial da plataforma Java foi feito em 23 de maio por John Gage da Sun Microsystem e Marc Andreessen da empresa Netscape.

Na década de 1990, a Netscape dominava o mercado de navegadores de internet. A plataforma Java era composta pela máquina virtual java (Java Virtual Machine - JVM) e por um ambiente integrado de desenvolvimento, chamado de IDE Java.

Segundo Mendes, as **principais características da linguagem Java são**:

- **Simplicidade;**
- **Orientação a objetos;**
- **Robustez;**
- ***Multithread*;**
- **Portabilidade;**
- **Segurança;**
- **Alto desempenho.**

Handheld
Dispositivo móvel; computador de bolso habitualmente equipado com uma pequena tela e um teclado em miniatura.

Simplicidade
A linguagem foi feita para suportar diferentes arquiteturas e sistema operacionais, desonerando o programador em se preocupar com detalhes de infraestrutura.

Orientação a objetos
A linguagem java já foi criada seguindo o paradigma da linguagem orientada a objetos. Sendo assim a linguagem traz nativamente os conceitos de herança, polimorfismo e encapsulamento.

Robustez
A linguagem permite a geração de sistema confiáveis, pois fornece em tempo de compilação verificação de códigos inalcançáveis, variáveis não inicializadas, verificação de tipos, inicialização de atributos e variáveis inteiras de forma automática com 0. Manipulação de exceções.

Multithread
Permite a realização de diversas tarefas ao mesmo tempo.

Interpretação
A linguagem java é o que chamamos de uma linguagem interpretada, ou seja, após a compilação do programa em java é gerado um arquivo intermediário.

Portabilidade
Garantida pela JVM, o que garante que um código possa ser executado em qualquer dispositivo que tenha uma JVM.

Segurança
A JVM não permite que um programa qualquer interfira ou inclua instruções em um programa java que esteja sendo executado.

Alto desempenho
A linguagem java garante um excelente desempenho de execução e ainda um mecanismo de gestão de memória eficiente.

02

A linguagem java é o que chamamos de **linguagem interpretada**. Nesse tipo de linguagem não geramos um arquivo executável em linguagem de máquina, como estamos acostumados a ver no Windows, por exemplo, como o arquivo Word.exe.

Na linguagem java a primeira coisa que fazemos é **escrever** o programa. Ele é escrito em formato de um arquivo texto normal com extensão .java. Para facilitar a escrita utilizaremos um *software* especial chamado de **Ambiente Integrado de Desenvolvimento - IDE**. Esse *software* facilita a identificação de erros e a execução de programas. Esse programa será responsável por gerar, a partir do arquivo texto, um arquivo intermediário chamado **bytecode**. Esse arquivo representa o programa bastante compactado em um formato independente do dispositivo. É esse o arquivo que deverá ser distribuído aos clientes, por exemplo, quando o programa estiver concluído.

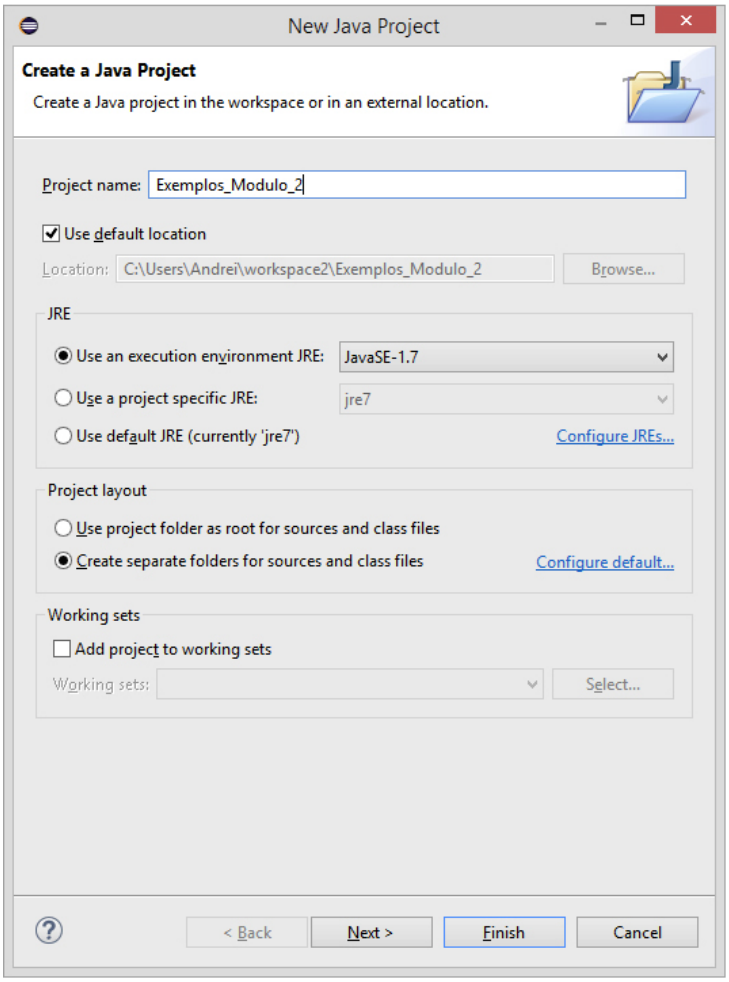
Os clientes devem ter instalado em suas máquinas um programa especial chamado de **Máquina Virtual Java – JVM**, que consiste em um interpretador, a sua missão é ler o programa em formato bytecode e executá-lo no dispositivo.

03

2 – Construindo e executando um programa em Java

Antes de começarmos a estudar a linguagem java é necessário instalarmos uma IDE. No nosso curso usaremos a IDE Eclipse que é um programa gratuito muito usado em desenvolvimento empresarial. Assim para darmos continuidade ao curso, certifique-se que o Eclipse esteja instalado em seu computador, bem como a **máquina virtual Java J2SE versão 1.6** ou superior (ver tutorial aqui).

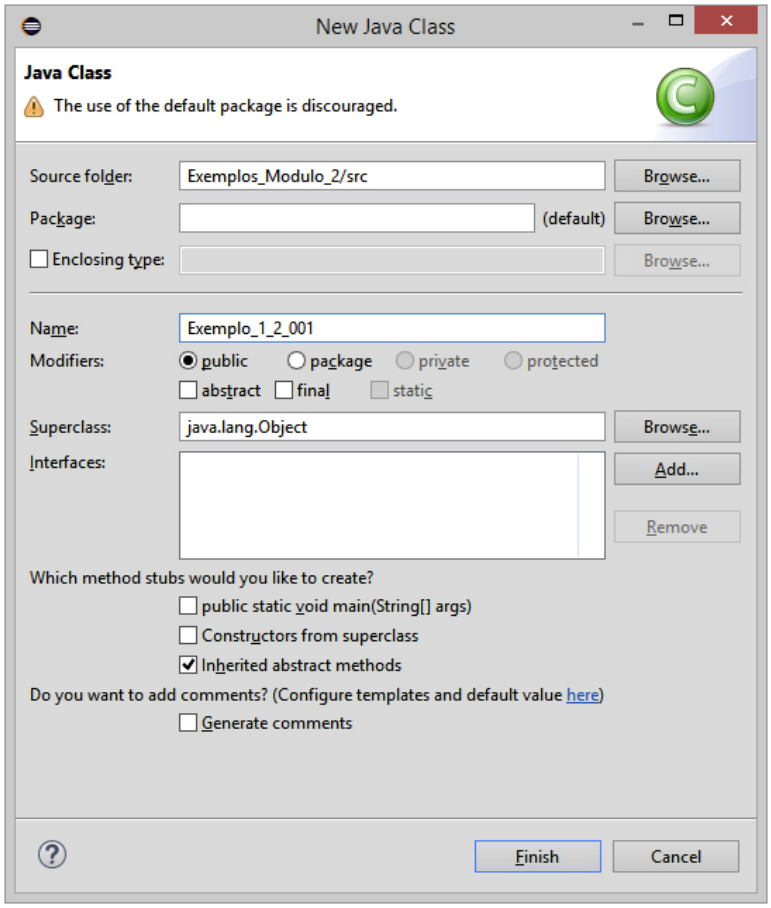
Ao executar o Eclipse pela primeira vez você deverá informar o *workspace*, que consiste no diretório que será utilizado pelos programas. Você pode utilizar o nome indicado pelo próprio Eclipse e seguir em frente com a execução do programa. O segundo passo é criar um projeto, para isso utilize a opção do menu: **File->New->Project->Java Project**. Sugiro chamar esse projeto de "Exemplos_Modulo_2".



Em seguida, pode clicar no botão “Finish” para concluir a criação do projeto.

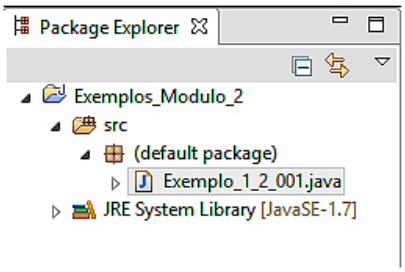
04

Tradicionalmente, o primeiro programa ensinado em uma linguagem de programação é um programa que escreve “Hello World” na tela do computador. Não vamos fugir a essa tradição, portanto, na janela “Package Explorer” (caso não esteja visível, use o menu Window->Show View-> Package Explorer) e clique com o botão direito na pasta src e e escolha a opção New->Class. Dê o nome de Exemplo_1_2_001.



05

Em seguida, ainda no Package Explorer, clique duas vezes sobre o arquivo Exemplo_1 _2_001.java (ver ilustração):



E escreva o programa abaixo:

Exemplo 1_2_001: Exemplo de programa que escreve Hello World na tela.

```
1. import java.util.*;
2. //Meu primeiro programa
3. public class Exemplo_1_2_001 {
4.             public static void main(String args[]){
```

```
5.            System.out.println("Hello World");
6.          }
7.  }
```

Vamos explicar agora cada uma das linhas do programa acima. A primeira linha chama uma biblioteca da linguagem java. Ela informa ao interpretador que a definição de alguns comandos utilizados no programa deve ser buscada naquela biblioteca. Não se preocupe, por enquanto, com essa declaração, pois iremos explicar com mais detalhes ao longo do curso. Na segunda linha temos uma linha de comentário.

Comentário
Em java podemos escrever **comentários** no programa de duas formas:

1) Comentários de uma linha colocando antes do comentário os símbolos //:

```
// escrever o comentário aqui!
```

2) Comentários com uma ou mais linhas colocando o comentário entre os símbolos /**/, conforme abaixo:

```
/* escrever aqui o
comentário */
```

06

Vamos continuar a análise das linhas do nosso programa:

```
import java.util.*;
//Meu primeiro programa
public class Exemplo_1_2_001 {
        public static void main(String args[]){
         System.out.println("Hello World");
         }
}
```

A terceira linha corresponde ao início da definição da classe chamada HelloWorld, cujo conteúdo é sempre colocado entre chaves:

```
<modificador>  class <nome_da_classe> {
      /*
      Conteúdo
      Da
      Classe
      */
}
```

Veremos no próximo módulo mais detalhes dessa declaração de classe.

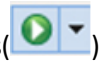
A quarta linha é a definição do método main. O formato geral da declaração de um método é a seguinte:

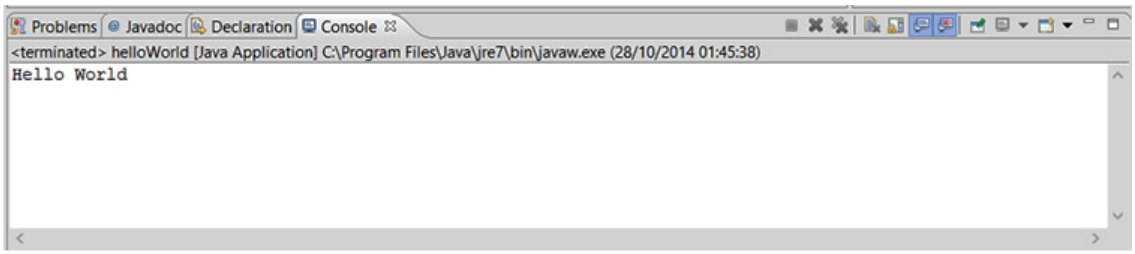
```
<tipo de retorno>  <nome do método> (<argumentos>){
//Conteúdo do método – comandos a serem executados pelo método
}
```

O tipo retorno é o tipo de dado que será retornado pelo método (não se preocupe, falaremos disso mais tarde), no exemplo, a palavra reservada **void** significa que nenhum dado será retornado pelo método. Os argumentos são as informações que o método deverá receber para ser executado. É importante salientar que **o conteúdo do método deve vir sempre entre chaves**.

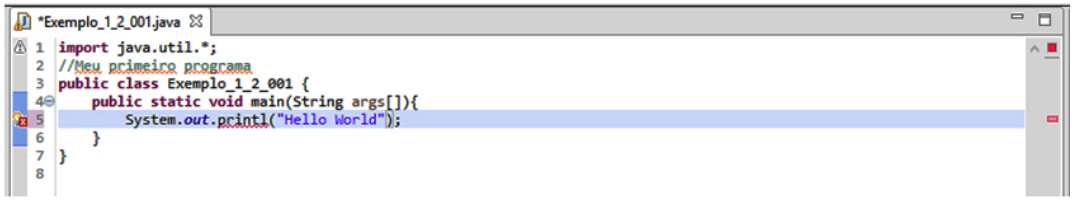
Por fim, na linha 5 temos a chamada ao método System.out.println() que imprime um texto na tela (console window). No caso, esse método imprimirá o texto “Hello World”.

07

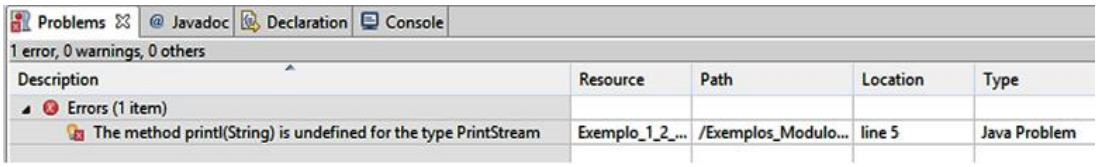
Para executar o programa, basta clicar no botão ***Run*** da barra de ferramentas() e ver o resultado no Console.



Caso tenha ocorrido algum erro, verifique se o programa foi escrito corretamente. Os erros de digitação são os mais comuns. Confira linha por linha do seu programa. A IDE também facilita muito encontrarmos os erros, o que chamamos em bom português de “debugar o programa”. A IDE indica ao lado da linha do programa um possível erro de digitação, antes mesmo de executarmos. A ilustração abaixo apresenta o programa com um pequeno erro na linha 5:



Veja que ele marca com um xis em vermelho ao lado do número da linha e também sublinhou em vermelho a palavra “printl”. Se colocar o mouse em cima dessa palavra ele até propõe algumas formas de resolver o problema. Caso você tentasse executar o programa apareceria uma caixa de diálogo informando que a execução falhou e apareceria tanto no console quanto na janela Problems os erros encontrados:



Nesse caso a correção é simples, basta corrigir na linha 5 a palavra “printl” por “println”. Uma vez feita a correção você poderá executar novamente o programa clicando em ***Run***. Se tudo tiver correto, a execução será realizada e aparecerá o texto.

Parabéns! Você escreveu seu primeiro programa orientado a objetos!

08

3 - Variáveis e Tipos de dados da Linguagem Java

Em programação as informações são armazenadas em locais da memória. Por exemplo, digamos que queremos fazer a seguinte operação matemática:

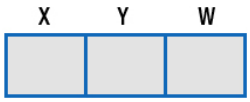
x =7

y =+10

w = x+y

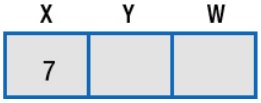
Para fazer isso, devemos reservar espaços de memória para guardar os três valores:

Esses espaços na memória são chamados de variáveis. Para facilitar a vida do programador, esses espaços de memória podem receber apelidos, por exemplo, chamaremos esses três espaços de memória de x, y e w. Nesse caso, vamos usar como apelidos os próprios nomes originais: x, y e w.

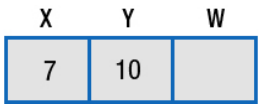


Assim, para guardar o primeiro valor faremos em programação:

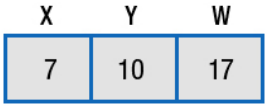
Quando no programa tiver o código **x=7**, o computador executará:



Após a execução do código **y=10**, teremos na memória:



E por fim, após a execução da última linha, **w=x+y**, teremos na memória:

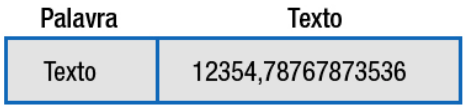


09

Da mesma forma, podemos querer armazenar na memória dados diferentes como textos ou números decimais, tais como:

Palavra = “texto”;

Valor = 12354,78767873536



Podemos ver que a memória pode armazenar diferentes tipos de informação. Os tipos de dados mais comumente utilizados em linguagem Java são mostrados na tabela a seguir:

Tipo	Tamanho	Valores Válidos
boolean	8 bits	true ou false
byte	8 bits	-128 a 127
short	16 bits	-32.768 a +32.767
int	32 bits	-2.147.483.648 a +2.147.483.647
long	64 bits	-9.223.372.036.854.775.808 a -+9.223.372.036.854.775.807
float	32 bits IEEE 754 floating point	-3,4028234663852886E+38 a +3,4028234663852886E+38
double	64 bits IEEE 754 floating point	-4,94065645841246544E-324 a +4,94065645841246544E-324
char	16 bits caractere unicode	‘\u0000 a \uFFFF’ – 0 a 65535
String	textos	Qualquer sequência de caracteres.

3.1 - Definindo variáveis

Podemos ver que a memória pode armazenar diferentes tipos de informação. A linguagem Java possui os seguintes tipos de dados primitivos:

Tipo	Tamanho	Valores Válidos
boolean	8 bits	true ou false
byte	8 bits	-128 a 127
short	16 bits	-32.768 a +32.767
int	32 bits	-2.147.483.648 a +2.147.483.647
long	64 bits	-9.223.372.036.854.775.808 a -+9.223.372.036.854.775.807
float	32 bits IEEE 754 floating point	-3,4028234663852886E+38 a +3,4028234663852886E+38
double	64 bits IEEE 754 floating point	-4,94065645841246544E-324 a +4,94065645841246544E-324
char	16 bits caractere unicode	'\u0000 a \uFFFF' – 0 a 65535
String	textos	Qualquer sequência de caracteres.

Em java, sempre que declaramos uma variável no programa devemos informar qual é o tipo da variável, no seguinte formato:

<tipo da variável> <nome da variável>;

Veremos, a seguir, alguns exemplos de declaração de variáveis:

Por exemplo, para declararmos uma variável chamada "valor" que armazene valores inteiros:

```
int Valor;
```

Da mesma forma podemos declarar uma variável chamada altura que armazene valores decimais:

```
float altura;
```

É possível também atribuir um valor à variável diretamente na sua declaração:

```
int valor = 10;
```

ou

```
float altura = 10.5;
```

Importante: veja que todas as linhas de comando da linguagem java terminam com ponto e vírgula.

11

A seguir mostraremos um exemplo de declaração por meio de um programa simples em java. Para isso sugiro criar um projeto java no Eclipse e, em seguida, criar uma classe chamada Principal com o conteúdo mostrado a seguir.

Exemplo 1_2_002: Exemplo de declaração de uma variável e impressão na tela.

```
import java.text.DecimalFormat;
import java.util.Scanner;

public class Principal {
  public static void main(String[] args) {
    int  n;     //declaração da variável n
    n=10;              //atribuição do valor à variável n
    DecimalFormat df = new DecimalFormat("0.00"); //servirá para converter
                                                  //um valor inteiro      em string
    System.out.println("Variável n= " + df.format(n)); //imprime na tela o
                                                       // valor de n
   }
}
```

No exemplo acima declaramos a variável inteira n (linha 6), atribuímos o valor 10 (linha 7) e imprimimos o seu valor no console. Não se preocupem com o conteúdo da linha 8, por enquanto. Essa linha cria um objeto da classe DecimalFormat apenas para poder converter o valor inteiro em texto e imprimi-lo usando o método System.out.println().

A seguir vamos descrever com mais detalhes esses **tipos de dados**.

12

- **Tipo boolean**

O tipo boolean é um tipo de dado que permite o armazenamento de dados que possuem apenas dois estados verdadeiro (true) ou falso (false).

Podemos usar variáveis do tipo boolean para descrever, por exemplo, a representação no programa do estado de uma lâmpada ou de um teste:

boolean lampada_acesa = true;

O código acima declara uma variável chamada lâmpada_acesa que recebe o valor verdadeiro (true).

- **Tipos inteiros**

O Java possui 4 tipos de dado que servem para representar dados inteiros:

byte
short
int
long

A diferença entre esses tipos de dados são os **limites de representação**. O dado do tipo **byte** permite a representação de valores entre -128 e +127. Isso significa que se queremos representar a seguinte operação:

X=7

Poderemos declarar a variável x, conforme segue:

Byte X;
X = 7;

Entretanto, o comando abaixo seria incorreto:

X=254;

Porque o limite de representação de um dado do tipo byte é 128. Se a variável X deve receber valores superiores a 127 então deveremos declarar essa variável como: short, int ou long:

short X;
X=254;

Importante: na declaração de valores inteiros verifique sempre qual o tipo de dado mais adequado, de acordo com o valor máximo que se deseja representar.

13

Exemplo 1_2_003: cálculo da área e da circunferência de um círculo.

```
import java.text.*;
```

```
import java.util.*;

public class Principal {
	public static void main(String[] args) {
		final double PI = 3.14159;
		final String TAB = "\t";
		final String NEWLINE = "\n";
		double raio, area, circunferencia;
		//A linha abaixo declara um objeto do tipo scanner que permite
a leitura de informações
		Scanner scanner = new Scanner(System.in);
		DecimalFormat df = new DecimalFormat("0.000");

		//Imprimi um texto solicitando o valor do raio para o usuário
		System.out.println("Entre com o valor do raio: ");

		//Lê o valor do raio que o usuário digitou
		raio = scanner.nextDouble( );

		//Faz os cálculos
		area = PI * raio * raio;
		circunferencia = 2.0 * PI * raio;

		//Mostra os resultados
		System.out.println(
		"Dado o valor do raio: " + TAB + df.format(raio) + NEWLINE +
		"Area: " + TAB + df.format(area) + NEWLINE +
		" circunferencia: " + TAB + df.format(circunferencia));
	}

}
```

Exemplo: cálculo da área e da circunferência de um círculo.

Apresentamos a seguir como utilizar a classe scanner para ler dados numéricos:

Tamanho	Valores Válidos
nextByte( )	byte b = scanner.nextByte( );
nextDouble( )	double d = scanner.nextDouble( );
nextFloat( )	float f = scanner.nextFloat( );
nextInt( )	int i = scanner.nextInt( );
nextLong( )	long l = scanner.nextLong( );
nextShort( )	short s = scanner.nextShort( );

14

- **Tipo Caractere e textual**

A linguagem java possui nativamente o tipo de dado chamado **char** que serve para armazenar um caractere:

char meucaractere='a';
char minhaletra='*';

Veja que os dados do tipo char são sempre atribuídos com o símbolo entre aspas simples.

Caso queiramos armazenar textos usaremos variáveis do tipo String:

String <nome da variável>

Caso queiramos atribuir um texto a uma **variável textual** deveremos colocar o texto entre aspas duplas, como mostrado a seguir:

String minhapalavra = "teste";
String minhaFrase = "Meu primeiro programa";

O exemplo a seguir apresenta um programa que lê uma entrada de texto e imprime o valor digitado na tela.

Exemplo 1_2_004: programa que lê uma entrada de texto e imprime o valor digitado na tela.

```
import java.util.*;

public class Principal {
        public static void main(String[] args) {
                String nome;  //declaração da variável textual nome
                Scanner scannome = new Scanner(System.in);
                nome= scannome.next();  //Lê o texto escrito no console
                System.out.println("Nome: " + nome); //imprime na tela
                scannome.close();
        }
}
```

No exemplo acima, na linha 5, criamos um objeto da classe Scanner que permite a leitura de informações do console. Na linha 6 chamamos o método next () para ler uma String do teclado. Um ponto importante é que o programa espera uma palavra apenas.

15

Caso deseje ler uma frase que inclua espaços em branco deverá fazer uma pequena modificação no código, alterando uma propriedade da classe:

Exemplo 1_2_005: programa que lê uma entrada de texto com espaços e imprime o valor digitado na tela.

```
import java.util.*;
```

```
public class Exemplo_1_2_005 {

        public static void main(String[] args) {
            String nome;
            Scanner scannome = new Scanner(System.in);
            scannome.useDelimiter (System.getProperty("line.separator"));
            nome= scannome.next();
            System.out.println("Nome: " + nome);
            scannome.close();
    }
}
```

A linha incluída diz para o objeto scanome que deverá considerar como entrada qualquer texto até que o usuário tecle enter.

16

4 - Expressões em Linguagem Java

Em linguagem java consideramos expressões qualquer sentença terminada pelo símbolo ‘;’. As operações mais comuns são as expressões matemáticas, as expressões lógicas, avaliação de condições e atribuições.

Vamos tratar cada uma delas a seguir.

- **Expressões aritméticas**

As expressões aritméticas retornam valores numéricos e usam operadores matemáticos.

Os operadores mais comuns são:

Soma (operador +):

Int A = 10;

Int B = A+5;

Subtração (operador -):

float A = 10.5;

float B = A-5.4;

Multiplicação (operador *):

Int A = 10;

Int B = 5;

```
Int C = A*B;
```

Divisão (operador /)

```
float A = 10;

float B = 5;

float C = A/B;
```

Resto da divisão (operador %)

```
Int A = 8;

Int B = 5;

Int C = A%B; //a variável C receberá o resto da divisão de 8 por 5 que é 3.
```

Incremento (++)

```
Int A = 10;

A++; //fará com que o valor 11 seja atribuído à variável A. É o mesmo que escrever

    // A= A+1
```

Decremento(--)

```
Int A = 10;

A--; //fará com que o valor 9 seja atribuído à variável A

    // A= A-1
```

17

- **Expressões Booleanas**

As expressões Booleanas operam sobre argumentos e retornam um valor, que pode assumir apenas dois valores possíveis, verdadeiro ou falso.

Operador OU lógico - operador binário denotado pelo símbolo || que opera como o correspondente booleano. O resultado desse operador será verdadeiro, caso um dos argumentos seja verdadeiro. Se o primeiro argumento já é verdadeiro, o segundo argumento não é avaliado.

Exemplo:
boolean A = true;
boolean B = false;
boolean C = A || B; //o resultado será verdadeiro, visto que o primeiro argumento é verdadeiro.

Operador E lógico: operador binário denotado pelo símbolo && terá um resultado verdadeiro apenas se os dois argumentos forem verdadeiros.

Exemplo:
boolean A = true;
boolean B = false;
boolean C = A && B; //o resultado será falso, visto que o argumento B é falso.

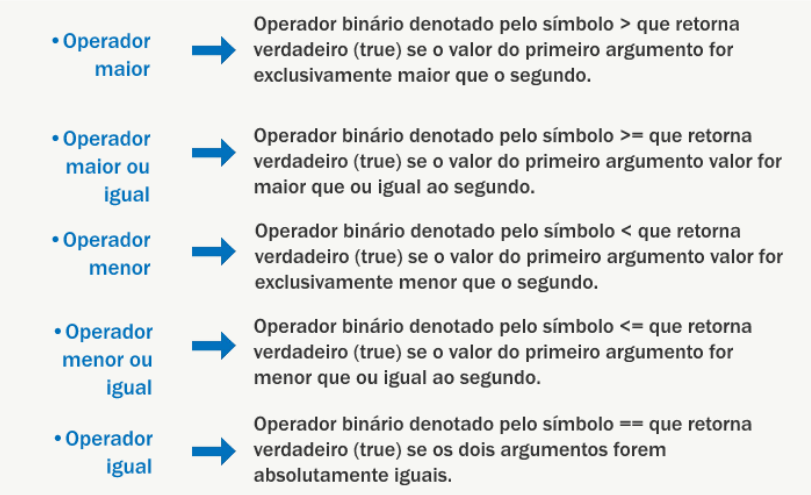
Exemplo:
boolean A = true;
boolean B = true;
boolean C = A && B; //o resultado será verdadeiro, visto que o argumento B é verdadeiro.

OU-Exclusivo: operador binário denotado pelo símbolo ^ retorna verdadeiro quando os dois argumentos têm valores lógicos distintos.

Exemplo:
boolean A = true;
boolean Bool B = true;
boolean C = A^B; /*O resultado será falso, visto que o argumentos tem o mesmo valor lógico (verdadeiro).*/

18

Também são muito utilizados na linguagem java testes baseados em comparações entre valores numéricos, que são os chamados operadores condicionais:

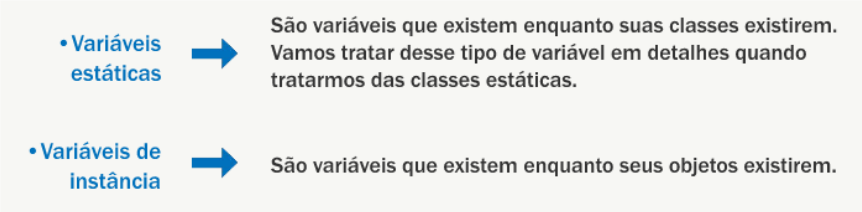


Cabe ressaltar que os operadores condicionais só são utilizados com os comandos de seleção condicional que veremos a seguir.

19

5 - Escopo de Variáveis

O escopo de variáveis é o contexto em que ela foi criada. As variáveis são classificadas em relação ao escopo como:



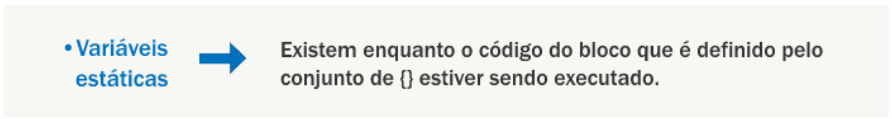
Exemplo 1_2_006: programa que lê uma entrada de texto com espaços e imprime o valor digitado na tela.

```
import java.util.*;

public class Principal {

        public static void main(String[] args) {
                String nome;
                Scanner scannome = new Scanner(System.in);
                nome= scannome.next();
              scannome=null;
                System.out.println("Nome: " + nome);
        }
}
```

A variável scannome só existirá até a linha 9, como ela recebeu o valor nulo e não existe mais nenhuma referência a ela no código, a variável será descarregada na memória.



20

Exemplo 1_2_007: programa que lê uma entrada de texto com espaços e imprime o valor digitado na tela.

```
import java.text.DecimalFormat;
import java.util.*;

public class Exemplo_1_2_007 {
```

```
            public static void main(String[] args) {
                        int n;
                        Scanner scanValores = new Scanner(System.in);
                        System.out.println("Entre com um valor inteiro:");
                        DecimalFormat df = new DecimalFormat("0.000");
                        n= scanValores.nextInt();
                        if(n%2==0)   //Testamos se o número é par verificando se o
resto da divisão por 2 é zero
                        {
                      String texto = "Valor "+df.format(n)+" é par";
                      System.out.println(texto);
                        }
                        else
                        {
                                System.out.println("Valor "+df.format(n)+" é
impar");
                        }
                        scanValores.close();
                    }
}
```

No exemplo acima a variável texto foi definida no bloco do comando if e, portanto, só existirá enquanto esse bloco estiver sendo executado.

21

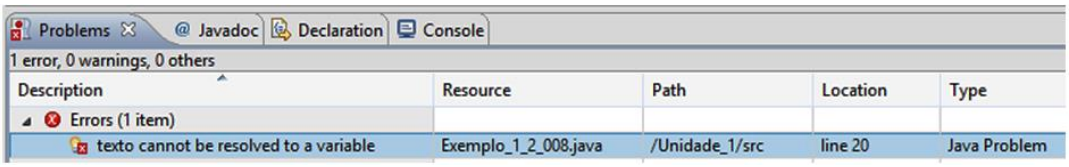
Exemplo 1_2_008: tentativa de acessar a variável texto fora do escopo do comando if. Atenção, esse exemplo apresenta erro ao ser executado.

```
import java.text.DecimalFormat;
import java.util.*;

public class Exemplo_1_2_008 {
            public static void main(String[] args) {
                        int n;
                        Scanner scanValores = new Scanner(System.in);
                        System.out.println("Entre com um valor inteiro:");
                        DecimalFormat df = new DecimalFormat("0.000");
                        n= scanValores.nextInt();
                        if(n%2==0)   //Testamos se o número é par verificando
se o resto da divisão por 2 é zero
                        {
                        String texto = "Valor "+df.format(n)+" é par";
                        System.out.println(texto);
                        }
                        else
                        {
                                System.out.println("Valor "+df.format(n)+" é
impar");
                        }
                      System.out.println(texto);
                        scanValores.close();
                    }
```

```
}
```

Se tentarmos executar o código do exemplo anterior, teremos a seguinte mensagem de erro:



Portanto, devemos sempre considerar aonde uma variável será declarada, pois isso afetará como ela poderá ser acessada.

22

6 - Comandos de seleção na linguagem Java

Os comandos de seleção são os comandos que permitem modificar o andamento do programa de acordo com condições específicas.

a) Comando IF-ELSE

O comando if possui o seguinte formato:

```
if(<condição>)
{

        //Escrever aqui o que deve ser executado se a condição for verdadeira

}


else
{

        //Escrever aqui o que deve ser executado se a condição seja falsa

}
```

A condição é construída utilizando as expressões booleanas vistas anteriormente. Vejamos a seguir alguns exemplos que mostram o uso do comando if.

Novamente ressalto a importância de procurar reproduzir os exemplos no Eclipse e testar a execução dos programas.

23

Exemplo 9: recebe dois valores e apresenta o resultado da divisão verificando se o divisor é igual a zero.

```
import java.util.*;
import java.text.*;

public class Exemplo_1_2_009  {

   public static void main(String[] args) {
        float n1, n2, divisao;
        Scanner scanValores = new Scanner(System.in);
        DecimalFormat df = new DecimalFormat("0.000");
        System.out.println("Entre com o dividendo:");
        n1= scanValores.nextFloat();
        System.out.println("Entre com o divisor:");
        n2= scanValores.nextFloat();
        if(n2>0)   //Aqui verificamos se a condição da divisão é satisfeita
        {
            divisao = n1/n2;
            System.out.println("Resultado da divisão: " + df.format(divisao));
        }
        else
        {
            divisao=0;
            System.out.println("Não se pode dividir um valor por zero!");
        }
        scanValores.close();
   }
}
```

24

Exemplo 1_2_0010: Verifica se o usuário digitou um número par e escreve o resultado desse teste.

```
import java.text.DecimalFormat;
import java.util.*;

public class Exemplo_1_2_010 {
          public static void main(String[] args) {
                        int n;
                        Scanner scanValores = new Scanner(System.in);
                        System.out.println("Entre com um valor inteiro:");
                        DecimalFormat df = new DecimalFormat("0.000");
                        n= scanValores.nextInt();
                        if(n%2==0)   /*Testamos se o número é par verificando
se o resto da divisão por 2 é zero*/
                        {
                            System.out.println("Valor "+df.format(n)+" é par");
                        }
                        else
                        {
                                System.out.println("Valor "+df.format(n)+" é
impar");
                        }
                        scanValores.close();
```

```
                    }
}
```

25

b) Comando Switch

O comando *switch* permite executar comandos diferentes de acordo com o valor de uma expressão. O formato básico desse comando é mostrado abaixo:

```
Switch(<expressão>)
{
        case  <valor 1>:
                /*colocar aqui os comandos a serem executados caso a expressão seja
igual ao
                valor1*/
        break;
        case <valor 2>:
                /*colocar aqui os comandos a serem executados caso a expressão seja
igual ao
                Valor2*/
        break;
        …
        default:
                /*colocar aqui os comandos a serem executados caso a expressão não
seja igual a nenhuma das opções anteriores*/
        break;
}
```

A seguir veremos um exemplo prático do uso do comando *switch*.

26

Exemplo 1_2_0011: programa que verifica o código digitado (valor inteiro) e imprime uma descrição.

```
import java.text.DecimalFormat;
import java.util.Scanner;

public class Exemplo_1_2_011 {
          public static void main(String[] args) {
                    int n;
                    Scanner scanValores = new Scanner(System.in);
                    System.out.println("Entre com um código (1-3):");
                    DecimalFormat df = new DecimalFormat("0.000");
                    n= scanValores.nextInt();
                    switch(n)
                    {
                    case 1:
                              System.out.println("Código "+df.format(n)+" -
Produtos Alimentícios");
                              break;
                    case 2:
```

```
                                System.out.println("Código "+df.format(n)+" -
Produtos de limpeza");
                                break;
                        case 3:
                                System.out.println("Código "+df.format(n)+" -
Produtos de vestuário");
                                break;
                        default:
                                System.out.println("Código "+df.format(n)+" -
não identificado");
                                break;
                        }
                        scanValores.close();
                }
}
```

27

c) Operador unário ?

O **operador ?** é muito similar ao comando if, sendo usado para fazer atribuições em expressões. O formato geral é o seguinte:

(<condição>)**?**(<valor caso a condição seja verdadeira>): (<valor caso a condição seja falsa>)

Vamos a seguir mostrar um exemplo de utilização do operador unário **?**.

Exemplo 1_2_0012: programa que verifica o código digitado (valor inteiro) e imprime uma descrição.

```
import java.util.Scanner;

public class Exemplo_1_2_012 {
          public static void main(String[] args) {
                        int n;
                        String resultado;
                        Scanner scanValores = new Scanner(System.in);
                        System.out.println("Entre com um valor inteiro:");
                        n= scanValores.nextInt();
                        resultado = n%2==0?"par":"impar";
                        System.out.println("O valor é "+resultado+"!");
                        scanValores.close();
          }
}
```

Na linha 10 usamos o operador unário para avaliar se o valor informado é par. Para isso verificamos se a expressão n%2 (lê-se: resto da divisão de n por 2) é igual a zero. Se for, o operador atribui o valor “par” à variável resultado. Caso contrário, atribui-se o texto “impar”.

28

RESUMO

Nesse módulo foram apresentados conceitos básicos da linguagem Java. Vimos que na linguagem java a primeira coisa que fazemos é escrever o programa em formato de um arquivo texto normal com extensão .java. Vimos também que as variáveis são os espaços de armazenamento de dados. Esses espaços devem ser previamente declarados no programa e também devemos informar qual o tipo de dado será armazenado.

Também apresentamos os comandos de seleção que permite ao programador tomar caminhos diferentes em função da avaliação de uma condição ou de uma expressão.Vimos três comandos de seleção: **if**, *switch* e o operador unário **?**.

No próximo módulo iremos continuar a nossa jornada mostrando como declarar classes, objetos. Sigamos em frente.

UNIDADE 1 – INTRODUÇÃO À PROGRAMAÇÃO ORIENTADA A OBJETOS

MÓDULO 3 – DEFININDO OBJETOS E CLASSES

01

1 – DECLARANDO CLASSES E OBJETOS

Para declararmos classes em java utilizamos a palavra chave ***class***, conforme formato geral a seguir:

```
class <Nome_da_classe> {
        /*
        Aqui você escreverá o conteúdo da classe: propriedades e métodos
        */
        }
```

Vamos retomar alguns dos exemplos de classe que abordamos no módulo 1 para mostrarmos como seriam declarados em linguagem java. Para isso crie um projeto java no Eclipse chamado de Modulo3 e crie cada uma das classes apresentadas nos exemplos a seguir.

Exemplo 1_3_001: criação de uma classe que permita representarmos cachorros.

```
public class cachorro {
    String Nome;
    String Raça;
    String Sexo;
    String DataDeNascimento;
    float peso;
    String cor;

    private void Latir(){}
    private void Correr(){}
    private void Sentar(){}
    private void Andar(){}
    private void Deitar(){}
```

```
}
```

Prestem atenção que usamos a palavra reservada **public** antes da palavra **class** para dizer que está classe pode ser acessada por qualquer outra classe que faça parte do mesmo projeto. Na declaração dos métodos usamos a palavra chave PRIVATE para dizer que os métodos só são acessíveis a partir de outros métodos da classe cliente, não podem ser chamados externamente. 02

Exemplo 1_3_002: criação de uma classe que permita representarmos clientes.

```
public class cliente {
    String Nome;
    String DataDeNascimento;
    String CPF;
    String Endereço;
    String Cidade;
    String CEP;

    public cliente(String nomeCliente)  //construtor da classe
    {
        Nome = nomeCliente;
        /*código que deverá ser executado quando for criada
       * uma instância da classe*/
    }
    private void alterarEndereco(){}
    private boolean validarCPF(String cpf){
        //valida o CPF e retorna a validação
        //para simplificar retornaremos verdadeiro
        return true;
    }
    private void alterarCep(){}
}
```

No exemplo acima apareceu um método que tem o mesmo nome da classe, que nesse caso é cliente(), que é chamado de **construtor da classe**. Esse método é chamado quando uma instância da classe é criada e serve para inicializar o objeto, seja para atribuir valores a atributos da classe, seja para chamar ações específicas. No exemplo acima o método cliente() inicializa o nome do cliente quando um objeto é criado.

Portanto, no projeto Modulo3 vocês devem ter agora duas classes: cachorro e cliente. Vamos ver em seguida como utilizá-las para declarar objetos.

03

2 - CRIANDO OBJETOS

Vamos começar criando por meio do Eclipse no projeto Modulo3 uma nova classe chamada de clientePrincipal, com o conteúdo abaixo:

Exemplo 1_3_003: Exemplo de uso da classe cliente.

```
public class clientePrincipal {
    public static void main(String args[]){
        cliente cli = new cliente("meu cliente");
        System.out.println(cli.Nome);
    }
}
```

Na linha 3 estamos criando um objeto do tipo cliente chamado de *cli*. A palavra chave new é sempre usada quando criamos um novo objeto. Observe que na criação chamamos o método construtor visto anteriormente passando os parâmetros necessários, nesse caso é o nome do cliente. Quando chamamos o método construtor é alocada memória para o objeto, inicializa a instância do objeto e retorna a referência do objeto.

A linha 4 imprime na tela o nome do cliente, mas utilizando o próprio objeto. Veja que para acessar o valor de um atributo do objeto colocamos o nome do objeto, seguido do ponto e depois o nome do atributo:

<Nome do Objeto>.<nome do atributo>

Da mesma forma poderemos mandar uma mensagem para o objeto de forma que ele execute um método:

<Nome do Objeto>.<nome do atributo>(<argumentos>)

Executem e verão a frase meu cliente aparecer no console.

04

Um pouco mais sobre os construtores da classe

Os construtores da classe são muito parecidos com métodos normais, mas eles possuem três características que o distinguem dos métodos tradicionais:

1. Sempre recebem o mesmo nome da classe.
2. Os construtores nunca têm tipo de retorno.
3. Podem receber apenas modificadores do tipo **public**, **private** e **protected**. Falaremos mais sobre esses modificadores ao longo do módulo.

Vamos modificar a classe *cachorro* para incluir um construtor para a classe que permite indicar o nome e a raça do cachorro. Também iremos implementar o método latir. Entre novamente no projeto e edite a classe cachorro como mostrado a seguir.

Exemplo 1_3_004: criação de uma classe que permita representarmos cachorros.

```
public class cachorro {
    String Nome;
```

```
    String Raca;
    String Sexo;
    String DataDeNascimento;
    float peso;
    String cor;

    public cachorro(String nome, String raca){
        Nome=nome;
        Raca=raca;
    }
    private void Latir(){
        System.out.println("auau");
    }
    private void Correr(){}
    private void Sentar(){}
    private void Andar(){}
    private void Deitar(){}
}
```

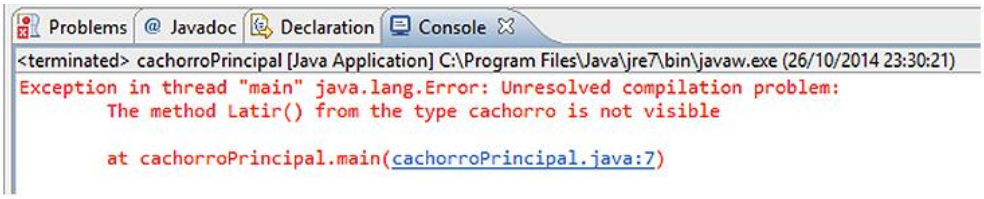
05

E vamos criar agora uma nova classe chamada de cachorroPrincipal com o conteúdo a seguir.

Exemplo 1_3_005: criação da classe cachorroPrincipal.

```
public class cachorroPrincipal {
    public static void main(String args[]){
        cachorro pet = new cachorro("toto","sao bernardo");
        System.out.println("Nome do cachorro: "+pet.Nome);
        System.out.println("Raca: "+pet.Raca);
        pet.Latir();    }
}
```

Executem essa classe. O que ocorreu? Apareceu um erro não é mesmo?



Isso aconteceu porque tentamos acessar o método Latir que não é acessível, pois ao declararmos o método dissemos que ele era do tipo private. Lembram-se do conceito de encapsulamento? É exatamente isso, nós podemos proteger partes do código de forma que elas não sejam acessíveis externamente. Entretanto, como queremos nesse caso ter acesso a esse método vamos alterar a classe cachorro de forma que o método Latir seja público.

06

Exemplo 1_3_006: criação de uma classe que permita representarmos cachorros.

```
public class cachorro {
    String Nome;
    String Raca;
    String Sexo;
    String DataDeNascimento;
    float peso;
    String cor;

    public cachorro(String nome, String raca){
        Nome=nome;
        Raca=raca;
    }
    public void Latir(){   //Agora esse método é acessível de fora da classe
        System.out.println("auau");
    }
    private void Correr(){}
    private void Sentar(){}
    private void Andar(){}
    private void Deitar(){}
}
```

Salvem essa alteração e executem novamente a classe *cachorroPrincipal*. Aparecerá no console o nome, a raça e o latido do cachorro!

Obviamente é possível criarmos mais de um objeto de uma classe. Como dissemos no módulo 1 , a classe é apenas uma representação genérica e os objetos, também chamados de **instâncias**, são representações concretas com valores próprios para os atributos.

07

Exemplo 1_3_007: criação da classe cachorroPrincipal.

```
public class cachorroPrincipal {
    public static void main(String args[]){
        //declarando dois abjetos do tipo cachjorro
        cachorro pet = new cachorro("toto","sao bernardo");
        cachorro puppy = new cachorro("didi","poddle");

        //Atribuindo valores a outros atributos dos cachorros
        pet.DataDeNascimento="12/05/2005";
        puppy.DataDeNascimento="27/06/2014";

        pet.peso=42;
        puppy.peso=2;


        //imprime o nome e a raça dos dois cachorros
        System.out.println("Nome do cachorro: "+pet.Nome);
        System.out.println("Raca: "+pet.Raca);
        System.out.println("Nome do cachorro: "+puppy.Nome);
        System.out.println("Raca: "+puppy.Raca);
```

```
        pet.Latir();
    }
}
```

08

3 – Chamando métodos: Argumentos e Parâmetros

● Declaração

No módulo anterior vimos que nos comunicamos com os objetos por meio de mensagens. Sempre que chamamos um método estamos mandando uma mensagem para o objeto executar aquele método. Por isso é muito importante entender corretamente o que acontece na chamada de um método e também entender a diferença entre argumentos e parâmetros.

A declaração de um método de uma classe tem o seguinte formato geral:

<modificador> <tipo de retorno> <nome do método>(<parâmetros>)

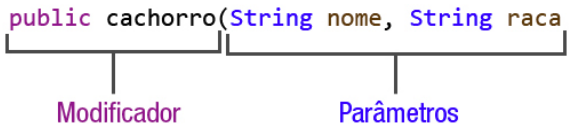
Os **modificadores** servem para definir o tipo de método e a visibilidade do método, por ora, apresentaremos apenas os dois modificadores mais usados:

public – significa que o método é sempre visível e acessível

private – significa que o método só pode ser chamado por algum outro método da classe.

É importante ter em mente que os métodos públicos determinam o comportamento externo do objeto, e os métodos privados (private) implementam funcionalidades internas que não precisam ou não devem ser usadas por quem utiliza o objeto.

O **tipo de retorno** é o tipo de valor que será retornado pela chamada do método, e os **parâmetros** são as variáveis que receberão as informações necessárias à execução do método. Um exemplo de declaração de método é apresentado abaixo:



09

● Enviando mensagens

Como dissemos anteriormente, comunicamos com os objetos enviando mensagens. Uma forma de enviar mensagens consiste em chamarmos a execução de método. A forma geral de chamada de método é a seguinte:

<nome do objeto>.<nome do método>(<argumentos>)

Caso a função retorne um valor, deveremos atribuir esse valor de retorno a uma variável:

<tipo> <nome da variável> = <nome do objeto>.<nome do método>(<argumentos>)

Os argumentos da função são os valores que são passados para ela quando chamamos a função. Por exemplo, quando criamos o objeto do tipo cachorro o fizemos da seguinte forma:

```
cachorro pet = new cachorro("toto","sao bernardo");
```

Cada parâmetro de uma função recebe um argumento, obedecendo a ordem com que eles são passados. Logo o primeiro argumento “totó” será atribuído ao primeiro parâmetro, nesse caso, o parâmetro nome. O segundo parâmetro “São Bernardo” será atribuído ao segundo parâmetro, nesse caso, a *string* raca.

Então, apenas para reforçar:

Parâmetros são as variáveis que estão na declaração da função e os argumentos são os valores que serão atribuídos aos parâmetros.

É importante lembrar que as funções/métodos não precisam, necessariamente, ter parâmetros. Um exemplo disso é o método Latir() que não precisa de nenhuma informação adicional para executar a ação de latir.

10

A constante de classe é uma variável cuja instância é compartilhada por todos os métodos das instâncias. Para facilitar o entendimento vamos ver o exemplo a seguir:

Exemplo 1_3_008: criação da classe timer com uma variável constante de passo.

```
public class timer {
    public static final int  passo = 100;
    private int _valorAtual;

    public timer(int valorInicial){
         _valorAtual=valorInicial;
    }
    public void decresce(){
        _valorAtual= _valorAtual-1;
    }
    public int getValue()
    {
```

```
            return _valorAtual;
    }
}
```

Podemos ver que a variável *passo* recebeu os modificadores **static final** que tornam essa variável uma constante, ou seja, ela não pode ser modificada por nenhum método.

11

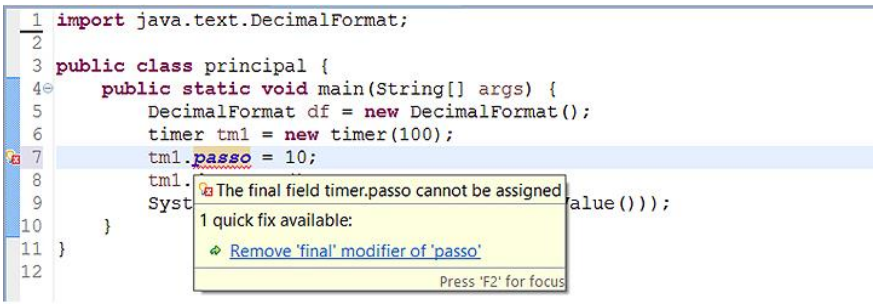
Para testar essa classe vamos criar uma classe principal com o método main.

Exemplo 1_3_009: criação da classe principal.

```
import java.text.DecimalFormat;

public class principal {
    public static void main(String[] args) {
        DecimalFormat df = new DecimalFormat();
        timer tm1 = new timer(100);
        tm1.decresce();
        System.out.println(df.format(tm1.getValue()));
    }
}
```

Caso você tente modificar uma variável constante, o eclipse indicará um erro antes mesmo que se tente executar o programa!



O uso de constantes é uma boa forma de manter informações protegidas, mas ao mesmo tempo compartilhada com os demais métodos. É bom lembrar que as constantes de classe também podem ser declaradas usando o modificador **private**, dessa forma deixariam de ser visíveis aos métodos que não fazem parte da classe.

12

Resumo

Neste módulo aprendemos que as classes são declaradas em linguagem java por meio da palavra reservada class. As classes podem ter métodos especiais chamados de construtores que permitem a inicialização da instância. Vimos que os construtores possuem três características principais que são:

1. Não possuem valor de retorno.
2. aceitam apenas três modificadores public, private e protected.
3. sempre recebem o nome da classe.

Os objetos são declarados de forma similar à declaração de variáveis, mas somente são instanciados (alocados na memória) quando utilizamos a expressão new em conjunto com o chamado do construtor.

Vimos que podemos acessar os atributos e métodos públicos por meio de mensagens. Mensagens na forma de chamadas a métodos. Também foram apresentadas as diferenças entre parâmetros e argumentos. Os parâmetros são as variáveis que fazem parte da declaração do método e recebem os argumentos passados quando o método é chamado.

Por fim, apresentamos a forma de declararmos constantes em uma classe. No próximo módulo trataremos dos comandos de repetição e das exceções.

UNIDADE 1 – INTRODUÇÃO À PROGRAMAÇÃO ORIENTADA A OBJETOS
MÓDULO 4 – ESTRUTURAS DE PROGRAMAÇÃO II: COMANDOS DE REPETIÇÃO E EXCEÇÕES
01

1 – ARRAYS

Suponha que queiramos fazer um programa para controlar as notas de uma turma com 10 alunos. Uma forma que teríamos seria guardar cada nota em uma variável:

```
float nota1;
float nota 2;
...
float nota 10;
```

Vemos que não é uma forma muito eficiente de programação, visto que temos que declarar diversas variáveis e será muito difícil depois conseguir automatizar tarefas de leitura de notas, por exemplo. Uma alternativa é usarmos o que chamamos de arrays. Os arrays são variáveis que permitem armazenar dados do mesmo tipo.

a) Declarando arrays

A forma de declaração de um array é mostrada abaixo:

```
<tipo do dado> [] <nome da variável>
```

Exemplos de declaração de arrays:

```
int[] valores; //declara uma variável chamada array que armazena valores inteiros
float[] notas; //declara uma variável chamada notas que armazena valores do tipo float
```

02

b) Inicializando arrays

A forma mais simples de inicialização de um array é atribuir valores diretamente na declaração do mesmo:

```
<tipo do dado> [] <nome da variável> = {<valor1>, <valor 2>,...,<valor n>}
```

Poderíamos utilizar essa forma para inicializar um array com as notas dos alunos:

```
float[] notas ={5,6.5,6,4,7,8,9.5,10,2,4};
```

Caso não se queira ou não possamos atribuir valores na declaração poderemos fazê-lo mais tarde, mas deveremos reservar o espaço de memória antes de começar a atribuir os valores. Fazemos isso através da palavra chave **new**:

```
<tipo do dado> [] <nome da variável> = new <tipo do dado> [<número de elementos>];
```

Exemplos:

```
float[] notas = new float[10]; //declara um array para 10 dados do tipo float.
```

Só então poderemos atribuir valores ao elemento do array. Cada elemento de um array é identificado por um índice que corresponde a posição relativa do elemento no array. O índice é um valor que começa em 0 , primeiro elemento, e vai até o último elemento que tem o índice n-1, onde n é a quantidade de elementos do array. Para atribuir ou recuperar o valor de um elemento de um array utilizamos a seguinte notação:

<nome do array>.[<índice do elemento>]

03

Então, considerando ainda o exemplo das notas teríamos:

Exemplo 1_4_001: programa que recebe 10 notas e imprime a média.

Exemplo 1_4_002: programa que três frases e a escreve no console.

Novamente salientamos a importância de procurar reproduzir os exemplos dados e explorar o código alterando a quantidade de elementos, criando novos arrays.

Essa é uma forma muito interessante de aprender programação. Mãos à obra!

Exemplo 1_4_001: programa que recebe 10 notas e imprime a média.

```
import java.util.*;

public class Exemplo_1_4_001  {
	public static void main(String args[]){
		float[] notas = new float[10];   /*nesse caso, quantidade de
                                                   elementos n = 10*/
		Scanner entrada = new Scanner(System.in);

		notas[0] = entrada.nextFloat(); /*atribuição de valor ao
                                                  primeiro elemento índice=0*/
		notas[1] = entrada.nextFloat();
		notas[2] = entrada.nextFloat();
		notas[3] = entrada.nextFloat();
		notas[4] = entrada.nextFloat();
		notas[5] = entrada.nextFloat();
		notas[6] = entrada.nextFloat();
		notas[7] = entrada.nextFloat();
		notas[8] = entrada.nextFloat();
		notas[9] = entrada.nextFloat();
		System.out.print(media(notas));
		entrada.close();
	}
	private static float media (float[] val)
	{
		float med = 0;

		med =
(val[0]+val[1]+val[2]+val[3]+val[4]+val[5]+val[6]+val[7]+val[8]+val[9])/10
;
		return med;
	}
}
```

Exemplo 1_4_001: programa que três frases e a escreve no console.

```
import java.util.Scanner;

public class Exemplo_1_4_002  {
	public static void main(String args[]){
		String[] frases = new String[3];   //nesse caso, quantidade de
                                                   elementos n = 3
		Scanner entrada = new Scanner(System.in);
		entrada.useDelimiter("\n");
		frases[0] = entrada.next();
		frases[1] = entrada.next();
		frases[2] = entrada.next();
		System.out.print(Concatena(frases));
		entrada.close();
	}
	private static String Concatena (String[] val)
	{
```

```
            String texto ="";

            texto =val[0]+" "+val[1]+" "+val[2];
            return texto;
      }
}
```

04

2 - ESTRUTURAS DE REPETIÇÃO

No exemplo 1_4_001 vimos que tivemos que escrever linhas de código muito parecidas para atribuir valores à variável notas. Imagine se em lugar de 10 notas tivéssemos 100 ou 1000. O trabalho seria muito grande e tedioso, sem falar que aumentaria a possibilidade de cometermos erros no código. Uma forma de evitar tudo isso é usarmos as maravilhosas estruturas de repetição. Essas estruturas quando bem utilizadas nos poupam uma quantidade grande de trabalho e facilitam a programação.

Basicamente, uma estrutura de repetição é um comando que permite executar um trecho de código até que seja satisfeita uma certa condição. Veremos a seguir as principais estruturas de repetição usadas em java.

a) Comando for

A declaração geral de um comando for é a seguinte:

```
for( <inicialização do contador> ; <condição> ;<variação do contador> ) {
//trecho de código a ser executado em cada iteração
}
```

O comando for basicamente permite que se utilize um valor inteiro que chamaremos de contador que varia a cada iteração e pode ser usado na condição. Enquanto a condição for verdadeira o trecho de código do bloco for será executado. Vamos a um exemplo para entender melhor esse comando:

Exemplo 1_4_003: programa que dez palavras e imprime-as na tela.

Exemplo 1_4_003: programa que dez palavras e imprime-as na tela.

```
import java.util.*;

public class Exemplo_1_4_003 {
      public static void main(String args[]){
            int i=0;
            String[] palavras = new String[10];
            Scanner entrada = new Scanner(System.in);
            entrada.useDelimiter("\n");
            for(i=0;i<10;i++)
            {
```

```
                palavras[i] = entrada.next();
            }
            for(i=0;i<10;i++)
            {
                System.out.print(palavras[i]+"\r");
            }
        }
}
```

No exemplo acima, o primeiro comando *for* (linhas 9 a 12) é usado lermos dez entradas do teclado em sequencia. O segundo comando for (linhas 13 a 16) é usado para imprimir na tela do console as palavras anteriormente lidas.

05

b) Comando while

O comando While possui a seguinte sintaxe:

```
while (<condição>)
{
//trecho de código a ser executado em cada iteração
}
```

O comando while pode ser interpretado como “Enquanto” a condição for verdadeira, ele executará o trecho no interior de seu bloco.

c) Comando do-while

O comando Do-while é um comando de laço um pouco diferente dos anteriores, pois ele testa a condição ao final da iteração. Isso significa que o trecho de código no interior do comando será executado ao menos uma vez. A sintaxe do comando do-while é mostrada a seguir:

```
do
{
//trecho de código a ser executado em cada iteração
} while ( <condição> )
```

Mostramos a seguir um exemplo utilizando os comandos while e do-while.

Exemplo 1_4_004: programa que dez palavras e imprime-as na tela.

Exemplo 1_4_004: programa que dez palavras e imprime-as na tela.

```
import java.util.*;

public class Exemplo_1_4_004 {
      public static void main(String args[]){
```

```
            int i=0;
            String[] palavras = new String[10];
            Scanner entrada = new Scanner(System.in);
            entrada.useDelimiter("\n");
            do
            {
                  palavras[i] = entrada.next();
                  i++;
            } while (i<10);
            i=0;
            while(i<10)
            {
                  System.out.print(palavras[i]+"\r");
                  i++;
            }
      }
}
```

Comparando o programa anterior com o exemplo utilizando o comando for podemos verificar que tivemos que inicializar o contador (variável i) e também incrementá-lo em cada iteração dentro do próprio código (linhas 12 e 18). Esses procedimentos não foram necessários com o uso do comando for que inicializa e incrementa o contador por meio de argumentos do próprio comando.

06

d) Alterando o fluxo de uma estrutura de repetição: comandos break e continue

Como vimos, todas as estruturas de repetição testam se uma condição é verdadeira para que continuem executando o trecho de código em seu bloco. Entretanto, algumas vezes gostaríamos que o laço parasse de ser executado ou que fosse executado pelo menos mais uma vez. Para isso utilizamos o comando break e o comando continue, respectivamente.

- **Comando break**

O comando break é utilizado, como dissemos, para parar a execução de um laço. Geralmente, é utilizado após testarmos alguma condição. Vejamos o exemplo a seguir:

Exemplo 1_4_005: programa que lê até dez valores inteiros e imprime-os na tela. Caso o usuário digite um valor superior a 100 o programa não fará mais a leitura e imprimirá os valores digitados até aquele momento na tela.

```
public class Exemplo_1_4_005 {
    public static void main(String args[]){
        int i=0;
        int lim=10;
        int[] valores = new int[10];
        Scanner entrada = new Scanner(System.in);
        do
        {
```

```
            valores[i] = entrada.nextInt();
            if(valores[i]>100)
            {
                lim=i;
                break;
            }
            i++;
        } while (i<10);
        i=0;
        while(valores[i]>100|| i<lim)
        {
            System.out.print(valores[i]+"\r");
            i++;
        }
    }
}
```

No programa, quando um usuário digitar um valor maior que 100 então o programa irá parar a execução do loop e imprimirá os dados digitados até aquele momento. Usamos a variável auxiliar lim para verificar quantos dados foram digitados até aquele momento.

07

- **Comando Continue**

O comando continue força uma nova execução da iteração, saltando qualquer código subsequente dentro do bloco da estrutura de repetição. O exemplo seguinte ilustra o uso desse comando de desvio.

Exemplo 1_4_006: programa que lê até dez valores inteiros e imprime-os na tela. Caso o usuário digite um valor superior a 100 o programa não imprimirá o resultado da divisão do número digitado por 2.

```
import java.util.*;

public class Exemplo_1_4_006 {
    public static void main(String args[]){
        int i=0;
        int lim=10;
        int[] valores = new int[10];
        Scanner entrada = new Scanner(System.in);
        do
        {
            valores[i] = entrada.nextInt();
            if(valores[i]>100)
            {
                i++;
                continue;
            }
        System.out.printf("%d/2=%f\n",valores[i],(float)valores[i]/2);
        } while (i<10);
    }
}
```

No exemplo acima, qualquer valor digitado que for maior que 100 não será impresso na tela visto que o comando da linha 17 será ignorado e o comando do fará uma nova iteração.

08

3 – EXCEÇÕES

Problemas inesperados podem acontecer não importa o quão bom seja o programa, pois sempre o usuário poderá fazer uma entrada de dados incorreta, o servidor de base de dados pode ficar inacessível ou mesmo uma atualização que esteja sendo feita via internet pode ser simplesmente interrompida com a queda do sinal. Por isso o programador deve se precaver sempre. A forma profissional de se precaver contra os erros é gerenciar as exceções. Uma exceção é um estado não desejado de funcionamento do programa.

Caso o programa não esteja preparado para tratar a mensagem, o sistema operacional apresentará uma notificação de erro na tela e poderá resultar em perda de dados, em inconsistências chegando mesmo a travar o sistema. Por isso é importante que o programador trate de forma eficiente essas exceções, pois dessa forma poderá evitar problemas maiores e fazer que o programa continue estável.

Em programação java tratamos exceção usando os comandos **try-catch-finally** que possuem a seguinte sintaxe:

```
try
{
      //bloco a ser executado de forma protegida.
}
catch(exception e)
{
       //bloco que será executado em caso de erro.
}
finally
{
      //bloco que será executado sempre
}
```

O bloco definido pelo comando **try** é o trecho do código que estará sendo rastreado. Caso uma exceção ocorra durante a execução de alguma parte desse código, será executado o bloco definido pelo comando **catch**. É nesse bloco que o programador poderá fornecer ao usuário uma notificação mais amigável do erro e a possível forma de corrigi-lo. O comando **finally** será executado sempre. O exemplo a seguir mostra o uso desses comandos.

09

Exemplo 1_4_007: programa que calcula a divisão do valor 10 pelo divisor que é um valor informado pelo usuário.

```
import java.util.*;

public class Exemplo_1_4_007 {
    public static void main(String args[]){
        int dividendo=10;

        Scanner entrada = new Scanner(System.in);

        int divisor= entrada.nextInt(); /*atribuição de valor
                                          ao primeiro elemento índice=0*/
        try
        {
            dividendo=1/divisor;
        }
        catch(Exception e)
        {
            System.out.print("O divisor não pode ser zero! \r");
            dividendo=0;
        }
        finally
        {
            System.out.print("Resultado da divisão: "+dividendo);
        }
    }
}
```

Vejam que se o usuário informar o valor 0 para o divisor ocorrerá uma exceção (divisão por zero) que será tratada no bloco catch.

É bom ficar claro que existem centenas de exceções devido aos mais diversos fatores. Algumas exceções são bem simples de serem detectadas e evitadas como a divisão por zero, entretanto, outras são muito mais complexas e imprevisíveis. Por isso, é importante escrevermos o programa tendo em mente a possibilidade de que o código não seja executado de forma conveniente então deveremos nos preparar para tomar medidas que garantam a continuidade e estabilidade do programa.

10

Resumo

A implementação de processos de negócio, geralmente, são processos repetitivos que envolvem uma grande quantidade de dados. O tratamento individual de cada informação em uma variável individual seria inviável. Sendo assim é necessário conhecer novos tipos de variáveis mais eficientes para essa finalidade e aprender a usar os chamados comandos de repetição.

O primeiro conceito visto nesse módulo é o conceito de array, que é um tipo de variável que permite o armazenamento de uma coleção de dados do mesmo tipo. Os arrays são armazenados de forma linear

na memória e cada elemento possui índice que o identifica. No java o primeiro elemento recebe o índice 0 e o último será N-1, onde N é a quantidade de elementos do array.

Os comandos de repetição são comandos específicos que permitem a execução repetida de um trecho de código definido. Vimos nesse módulo os comandos *for*, *while* e *do-while*. Os comandos for e while são comandos que repetem o trecho definido de código enquanto uma condição for verdadeira. O teste é feito logo no início, sendo assim é possível que o código não seja executado, caso a condição seja falsa logo no começo. O comando *do-while* também utiliza uma condição, entretanto a condição é verificada apenas ao final, sendo assim o trecho de código do comando de repetição será executado pelo menos uma vez. Vimos também que os comandos break e continue permitem alterar a execução do comando de repetição. O comando break para a execução do comando de repetição e será executada a instrução seguinte ao comando. O comando continue faz com que o programa execute a próxima condição do comando de repetição.

O último tópico que vimos foi sobre as exceções. Um programa nem sempre é executado corretamente. Erros podem acontecer pelos mais variados motivos: uma variável não possui o valor esperado, ou uma informação deveria ter um formato específico que o usuário não respeitou, ou mesmo um recurso necessário para efetuar uma determinada operação com um banco de dados ou uma conexão com a internet deixou de estar disponível. Problemas podem acontecer e o programa deve ser robusto o suficiente para evitar que determinados problemas ocorram (prevenção) ou reduzir os problemas caso algum erro ocorra (mitigação).

A prevenção pode ser feita colocando máscaras de texto, por exemplo, nas datas ou nos campos que só podem aceitar um tipo definido de informação. A mitigação é feita quando os erros não podem ser evitados. Um exemplo é a conexão com a internet. Muitos programas dependem do uso da internet para envio/ recebimento de dados e nem sempre uma conexão está disponível. Para que esse tipo de imprevisto não corrompa os dados ou cause outros problemas usamos o par de comandos try-catch. O comando try verifica a execução de um trecho de código e caso haja algum erro nesse trecho faz com que seja executado o trecho de código especificado no comando catch. O comando catch recebe as informações da situação de erro que foi identificada, chamada de exceção. Caberá ao desenvolver tratar o que deve ser feito em caso da ocorrência de uma determinada exceção.