**Introdução à Programação Orientada a Objetos**

A Programação Orientada a Objetos (POO) é um paradigma de programação de computadores que usa os conceitos de Objetos e Classes como elementos centrais para representar e processar dados usados nos programas.

A Programação Orientada a Objetos está sedimentada sobre quatro pilares derivados do princípio da abstração, são eles: **Encapsulamento, Herança, Composição e Polimorfismo.** O Princípio da abstração é a nossa capacidade de abstrair a complexidade de um sistema e se concentrar em apenas partes desse sistema, que é o tema central de nosso estudo ou foco em um dado momento.

**O que são objetos?** Na programação OO, objetos são usados para representar entidades do mundo real ou computacional, representando aqueles elementos e abstrações que fazem parte da solução que estamos desenvolvendo. Os objetos possuem características pelas quais os identificamos e finalidades para as quais os utilizamos. Essas características são tipicamente chamadas de atributos, no linguajar da programação Orienta a Objetos. As características também podem ser chamadas de dados ou atributos. Enquanto os comportamentos também podem ser chamados de operações ou métodos.

A modelagem e programação de um conjunto de objetos que possuem características (atributos) e comportamentos (métodos) comuns é feita na POO usando o conceito de Classe. Cada classe funciona no fundo como um molde para a criação de um dado objeto. Os objetos são vistos como representações concretas (instâncias) das classes. A

#### **Atenção!**

Já que as classes não existem concretamente, elas são abstratas, servem apenas para gerar os objetos.

Devemos ter em mente que a linguagem de programação é uma representação do mundo sob um ponto de vista. Você se lembra do paradigma? Esse ponto de vista ou paradigma é a orientação a objetos, porém, descrito nas Regras da Linguagem, que no nosso caso é a linguagem Java.

Na linguagem Java, as características são chamadas de atributos e são escritas informando o TIPO e o NOME do atributo:

class Carro{

String tipo;

String cor;

String placa;

}

Em Java, será necessário escrever um **ponto-e-vírgula** após a declaração de cada atributo, pois é uma exigência da linguagem Java. Além disso, as seguintes regras devem ser obedecidas para escolha do nome do atributo.

* O nome de uma variável deve ser uma sequência de letras Unicode e dígitos, iniciando sempre com uma letra, "$" ou o caractere de sublinhado "\_".
* Caracteres subsequentes podem ser letras, dígitos, $ ou "\_".
* Espaços não são permitidos em nomes de variáveis.
* Palavras reservadas da linguagem, tais como, class, int, float, for, while etc. não podem ser usadas como nomes de variáveis.
* Nomes de variáveis são sensíveis a letras minúsculas e maiúsculas. Em outras palavras, dois atributos com nomes "placa" e "Placa", por exemplo, são considerados dois atributos diferentes.

#### **Atenção!**

Use nomes completos, ao invés de abreviações na escolha dos nomes de atributos, isso tornará seu código mais legível.

### **Tipo de Atributo**

Todo atributo deve possuir um tipo, os quais podem ser um dos descritos a seguir.

1. Um dos oito tipos de dados primitivos.
   * Inteiros: byte (1 byte), short (2 bytes), int (4 bytes), long (8 bytes);
   * Reais: float (4 bytes), double (8 bytes)
   * Booleanos: boolean (1 byte)
   * Caracter: char (2 bytes)
2. O nome de uma classe ou interface.
3. Uma coleção.

#### **Constantes**

Para declarar uma constante, use a palavra chave final antes da declaração do atributo e inclua um valor inicial para esse último. O valor de uma constante nunca poderá ser modificado. A Listagem 3 exemplifica a declaração de algumas constantes.

class Matematica {

final int DEZENA = 10;

final double E = 2.71828;

}

Programar é traduzir uma solução de um problema para uma linguagem de programação. Logo, é imprescindível conhecer as regras da linguagem. O programador é como um intérprete ou tradutor de um idioma (linguagem no mundo real) para outro (linguagem entendida pelo computador).

Vamos a outro exemplo

class Pessoa {

String nome;

String corDoCabelo;

String biotipo;

int idade;

}

Classe Pessoa em Java

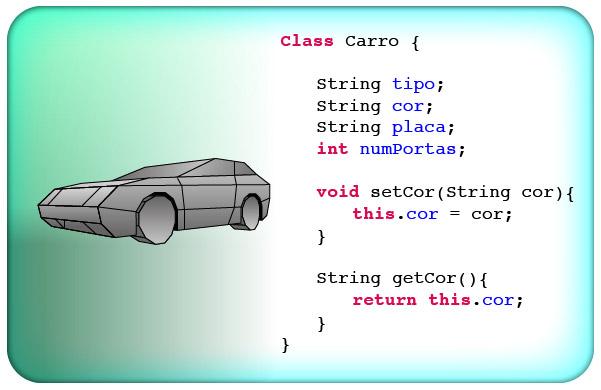
Cada atributo é definido com o par TIPO e NOME. String significa tipo de texto. int significa tipo numérico inteiro.

Veja que uma pessoa possui milhares de características, porém, utilizando o princípio da abstração, iremos deixar de fora tudo que não nos interessa ou relevante para o sistema que estamos desenvolvendo.

### **Adicionando Comportamento**

Os objetos das classes que criamos, Carro e Pessoa, precisam não só de suas características, mas também de comportamentos que expressem as ações possíveis de serem executadas, pois um carro precisa oferecer funções para que as pessoas os manobrem. E pessoas, por sua vez, precisam desempenhar suas atividades, como andar, correr, estudar etc. Percebam que falamos da capacidade dinâmica do objeto através dos métodos que possuem.

Vamos dar ações aos nossos carros! Veja a Listagem 6:



Observe que adicionamos novas linhas ao código, agora com dois métodos setCor(String cor) e getCor(). Esses métodos servem, respectivamente, para definir e recuperar o valor do atributo “cor”. Situação onde levaríamos o nosso carro a uma equipadora para modificar a sua cor. Atente para o fato de que os atributos são alterados por métodos da sua classe.

Veja que adicionamos a palavra this antes do atributo cor. Essa palavra-chave representa a instância da classe Carro e particularmente neste caso está sendo usada para corrigir o problema de ambiguidade gerado pela definição de um parâmetro com o mesmo nome do atributo da classe no método setCor(String cor). Ao se usar a palavra cor dentro deste método, como saberíamos se estamos referenciando o atributo da classe ou o parâmetro do método? Para solucionar esse problema, usa-se a palavra reservada this antes do nome cor, informando ao compilador que estamos referenciando o atributo da classe e não o parâmetro do método.

**Curiosidade!**

Dentro de um método, quando há ambiguidade de nome de variáveis (entre o atributo da classe e o parâmetro do método), o compilador sempre dá prioridade aos parâmetros. Em outras palavras, se não tivéssemos usado o this, o compilador iria achar que estávamos referenciando ao parâmetro do método e assim o atributo da classe nunca seria atualizado. Cuidado com isso! Essa é uma fonte muito comum de erros na programação em Java. Uma boa prática de codificação em Java é sempre usar a palavra this antes de atributos da classe, mesmo quando não há ambiguidade.

### 

### **Adicionando Comportamento**

Outra maneira de implementar os métodos sem usar a palavra chave this será apresentado na Listagem 7. Nesse caso, temos que mudar o nome do parâmetro para “c” evitando a ambiguidade.

class Carro {

String tipo;

String cor;

String placa;

int numPortas;

​

void setCor(String c){

cor = c;

}

String getCor(){

return cor;

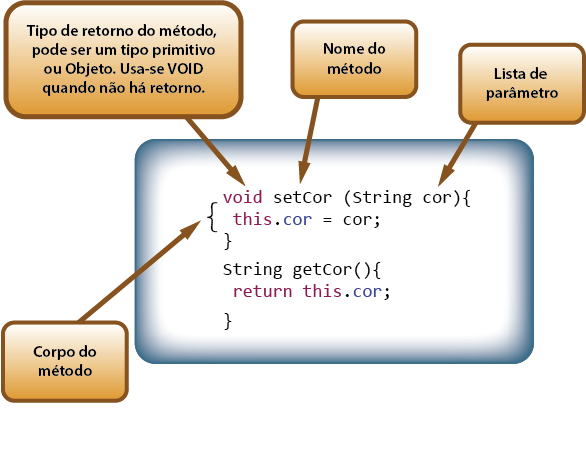
}

}

Listagem 7 - Classe Carro com métodos sem a palavra chave this

A definição dos métodos tem cinco partes básicas:

* tipo de acesso do método, (public, private, protected);
* nome do método;
* o tipo de retorno, o qual pode ser um objeto ou tipo primitivo;
* um ou mais parâmetros;
* finalmente, o corpo do método.



Listagem 8 - Exemplos de métodos

Logo, senão construímos outros métodos que acessem e modifiquem os valores dos demais atributos – do tipo placa e num Portas– esses não poderão ser vistos fora do objeto nem alterados.

Com isso, o nosso código ficará um pouco mais longo, veja a Listagem 9.

class Carro {

String tipo;

String cor;

String placa;

int numPortas;

​

void setCor(String c){

cor = c;

}

String getCor(){

return cor;

}

​

String getTipo(){

return tipo;

}

​

void setTipo(String tipo){

this.tipo = tipo;

}

​

String getPlaca(){

return placa;

}

void setPlaca(String placa){

this.placa = placa;

}

​

int getNumPortas(){

return numPortas;

}

void setNumPortas(int numPortas){

this.numPortas = numPortas;

}

}

Para que um programa Java execute ou “rode”, como costumamos dizer, é necessário que exista um método especial chamado main(...), o qual veremos em detalhe na sequência. Esse método é responsável pelo início da execução do Aplicativo ou Programa escrito em Java.

#### **Aplicativos**

São programas que podem ser executados sozinhos. A classe serve de ponto de partida para o aplicativo. Todo aplicativo em Java precisa ter um método main(). O método main tem a seguinte assinatura.

public static void main(String arguments[]){  
 //corpo do método  
 }

Se em um mesmo aplicativo houver mais de uma classe com o método main(), quando um main de uma das classes for executado, os demais serão ignorados enquanto o programa é executado.

Veja só, falamos de aplicativo (ou programa)! Vamos escrevê-lo. Para isso, precisamos do arquivo Carro.java da aula anterior, veja a Listagem 1:

class Carro {

String tipo;

String cor;

String placa;

int numPortas;

void setCor(String c){

cor = c;

}

String getCor(){

return cor;

}

String getTipo(){

return tipo;

}

void setTipo(String tipo){

this.tipo = tipo;

}

String getPlaca(){

return placa;

}

void setPlaca(String placa){

this.placa = placa;

}

int getNumPortas(){

return numPortas;

}

void setNumPortas(int numPortas){

this.numPortas = numPortas;

}

}

Precisamos criar outra classe, que irá conter o método main(), para podermos executar nosso programa, como mostra a Listagem 2:

/\*\*

\*ClasseResponsavelPorExecutarONossoPrograma

\*

\* @author nome\_do\_autor

\*

\*/

public class Main {

public static void main (String[] args){

/\* aqui estará todo o código responsável por executar o programa:

\*1- Criação de objetos

\*2- Manipulação dos seus dados

\*3- Finalização do programa

}

}

Vamos salvar esse arquivo como Main.java, agora temos dois arquivos Carro.java e Main.java. O método main está vazio, não faz nada, iremos editá-lo para que realize a sequência

### **Métodos Construtores**

Antes de tudo, temos que ter em mente que objetos são construídos, você NÃO pode criar um novo objeto sem invocar um construtor. Construtores representam o código que roda sempre que você usa a palavra-chave new.

Toda classe DEVE ter um construtor. Mas isso não significa que o programador tem que necessariamente codificar um. Caso não seja explicitamente declarado, o compilador criará um por padrão. Exemplo de construtor:

class Carro{

Carro(){} //O Construtor para a classe Carro.

}

Você consegue notar o que está faltando? Cadê o tipo de retorno? Existem dois pontos importantes que devemos notar a respeito dos construtores. Primeiro, é que eles não têm tipo de retorno, segundo, que o nome deve ser igual ao nome da classe. Você pode estar se perguntando: se eu quiser, no momento da criação do objeto, passar valores para alguns de seus atributos, como placa e cor, por exemplo? É possível?

É sim! Esta é a função básica dos construtores!

Mas, se a classe não tiver método construtor? O objeto ainda pode ser criado usando a instrução (ou operador) new. Mas, nesse caso, será provavelmente necessário chamar métodos set, como fizemos anteriormente. Considere a seguinte definição de um método construtor na classe Carro adicionando o método construtor mostrado na Listagem 3:

class Carro {

String cor;

String placa;

Carro(String placa, String cor){

this.placa = placa;

this.cor = cor;

}

}

Construtor para classe Carro

Observe que esse método possui o mesmo nome da classe Carro e não possui tipo de retorno, pois seu retorno, como já mencionamos, é justamente a referência para o objeto na memória, lembra-se disso?

Atenção, se fizermos apenas essa alteração, iremos nos deparar com um erro de compilação na classe Main. Você sabe por quê?

Então, como finalmente criamos uma instância da classe Carro utilizando o seu construtor? Exemplo:

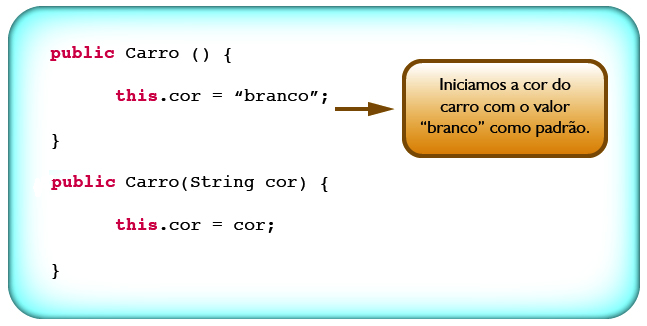
Carro meuCarro = new Carro()

Observe que quando utilizamos o operador new não existem argumentos (ou parâmetros) dentro dos parênteses. Isso significa que podemos estar usando o construtor padrão (ou, em inglês, default).

### **Sobrecarga de Construtores**

Vamos aprofundar um pouco mais a respeito dos construtores. Imagine agora se você quisesse dar mais flexibilidade para a criação dos seus objetos. Nesse caso, estamos usando a classe Carro como base, então, imagine que queremos dar a possibilidade de criar carros informando a sua cor, porém, também queremos ter a possibilidade de criar carros sem informar a sua cor, e que ao não informar a cor, que o carro seja considerado branco por padrão. Como faremos isso?

A resposta está na sobrecarga de construtores, como veremos a seguir, essa técnica é tipicamente utilizada para formas alternativas de instanciar objetos da sua classe, como pode ser visto na Listagem 4.



Listagem 4 – Sobrecarga de construtor na classe Carro

Quando em uma classe define-se outro construtor, não podemos mais usar o construtor padrão, a não ser que se defina explicitamente um construtor sem parâmetros, como mostra o construtor 1 na Listagem 5. Com isso, a classe Carro ficará com dois construtores.



Listagem 5 – Dois construtores para uma mesma classe

Observe que no primeiro construtor nada foi feito, enquanto no segundo os atributos placa e num Portas foram inicializados (receberam valores). Isso significa que existem duas maneiras de se construir um objeto derivado da classe Carro, ou do tipo Carro, com ou sem inicialização dos valores dos atributos.

Com isso, fechamos a base para se criar um objeto a partir de uma classe e ainda executar um programa que utilize esse objeto.

O que são construtores?

* Construtores são métodos especiais para a criação e inicialização de novas instâncias de classe (objetos).
* Inicializam o novo objeto e seus atributos.
* Criam todos os outros objetos de que esse objeto necessita.
* Realiza todas as outras operações que ele precisa para ser inicializado.

O que faz o operador new?

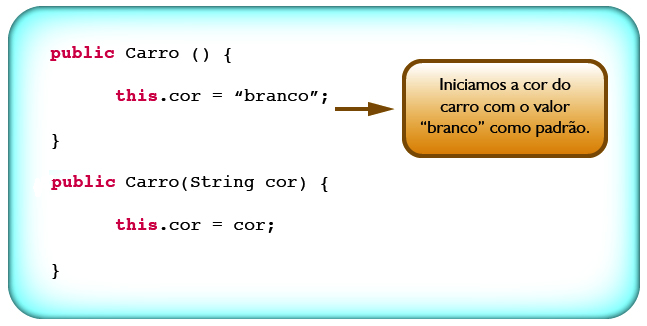
Ele inicializa o novo objeto e seus atributos, criam todos os outros objetos de que esse objeto necessita e realizam todas as outras operações de que ele precisa para ser inicializado.

Agora que sabemos uma pouco mais sobre construtores, vamos criar um objeto da classe Carro, no mundo da programação orientada a objetos, chamamos isso de instanciação, ou seja, criaremos uma instância da classe Carro.

### **Sobrecarga de Construtores**

Vamos aprofundar um pouco mais a respeito dos construtores. Imagine agora se você quisesse dar mais flexibilidade para a criação dos seus objetos. Nesse caso, estamos usando a classe Carro como base, então, imagine que queremos dar a possibilidade de criar carros informando a sua cor, porém, também queremos ter a possibilidade de criar carros sem informar a sua cor, e que ao não informar a cor, que o carro seja considerado branco por padrão. Como faremos isso?

A resposta está na sobrecarga de construtores, como veremos a seguir, essa técnica é tipicamente utilizada para formas alternativas de instanciar objetos da sua classe, como pode ser visto na Listagem 4.



Listagem 4 – Sobrecarga de construtor na classe Carro

Quando em uma classe define-se outro construtor, não podemos mais usar o construtor padrão, a não ser que se defina explicitamente um construtor sem parâmetros, como mostra o construtor 1 na Listagem 5. Com isso, a classe Carro ficará com dois construtores.



Listagem 5 – Dois construtores para uma mesma classe

Observe que no primeiro construtor nada foi feito, enquanto no segundo os atributos placa e num Portas foram inicializados (receberam valores). Isso significa que existem duas maneiras de se construir um objeto derivado da classe Carro, ou do tipo Carro, com ou sem inicialização dos valores dos atributos.

Com isso, fechamos a base para se criar um objeto a partir de uma classe e ainda executar um programa que utilize esse objeto.

O que são construtores?

* Construtores são métodos especiais para a criação e inicialização de novas instâncias de classe (objetos).
* Inicializam o novo objeto e seus atributos.
* Criam todos os outros objetos de que esse objeto necessita.
* Realiza todas as outras operações que ele precisa para ser inicializado.

O que faz o operador new?

* Ele inicializa o novo objeto e seus atributos, criam todos os outros objetos de que esse objeto necessita e realizam todas as outras operações de que ele precisa para ser inicializado.

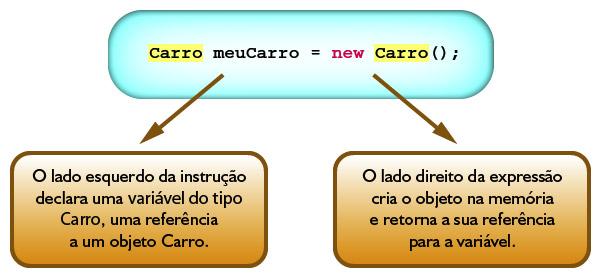
Agora que sabemos uma pouco mais sobre construtores, vamos criar um objeto da classe Carro, no mundo da programação orientada a objetos, chamamos isso de instanciação, ou seja, criaremos uma instância da classe Carro.

### **Criando um Objeto**



Um objeto ou instância é criado através do operador new. Vejamos a seguinte instrução na Figura 1:

Figura 01 - Esquema de criação de um Objeto



Toda a manipulação e consulta dos atributos do objeto serão feitas através de sua referência, ou seja, da variável que possui essa referência para os dados e métodos armazenados na memória.

Nesse instante, nosso carro foi criado, mas nenhum de seus atributos foi ainda definido. Vamos utilizar os métodos da classe Carro para defini-lo como quisermos, para isso, vamos utilizar o conjunto de métodos setAtributo(valor). Veja a Listagem 6.

meuCarro.setCor("preto");

meuCarro.setNumPortas(4);

meuCarro.setPlaca("MHX 1234);

meuCarro.setTipo("esportivo");

Listagem 6 – Definição de atributos usando o método set

Agora, vamos exibir cada atributo com seu respectivo método getAtributo(). Como mostra a Listagem 7 a seguir.

System.out.println("Cor: "+meuCarro.getCor());

System.out.println("Número de portas: "+meuCarro.getNumeroPortas());

System.out.println("Placa: "+meuCarro.getPlaca());

System.out.println("Tipo: "+meuCarro.getTipo());

Listagem 7 – Exibição dos valores dos atributos

Logo, a nossa classe Main ficará como mostra a Listagem 8:

/\*\*

\*ClasseResponsavelPorExecutarONossoPrograma

\*

\* @author nome\_do\_autor

\*

\*/

​

public class Main{

​

public static void main(String[] args){

/\*aqui estará todo o código responsável por executar o programa:

\*1- Criação de objetos

\*2 - Manipulação de seus dados

\*3- Finalização do programa

\*/

​

System.out.println("Meu primeiro programa java");

​

Carro meuCarro = new Carro();

meuCarro.setCor("preto");

meuCarro.setNumPortas(4);

meuCarro.setPlaca("MHX 1234);

meuCarro.setTipo("esportivo");

​

System.out.println("Cor: "+meuCarro.getCor());

System.out.println("Número de portas: "+meuCarro.getNumeroPortas());

System.out.println("Placa: "+meuCarro.getPlaca());

System.out.println("Tipo: "+meuCarro.getTipo());

​

}

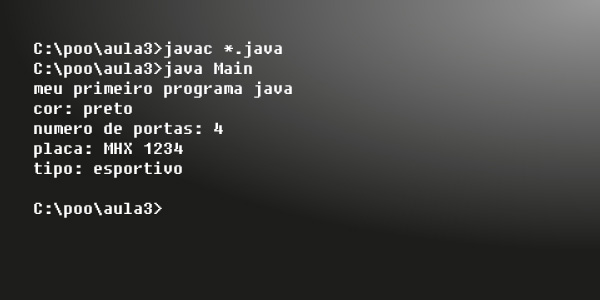
​

}

Listagem 8 – Método main completo

Por fim, compile a classe que você acabou de criar e depois execute-a. Você verá a saída do programa apresentada na Figura 2.

Figura 02 - Execução do método main

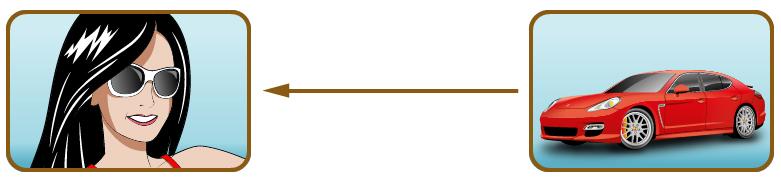


<https://materialpublic.imd.ufrn.br/curso/disciplina/5/8/3/6>

Vamos ligar a classe Carro à classe Pessoa, fazendo com que uma pessoa possua um carro e esse carro só possa pertencer a uma única pessoa. É o que chamados de relação um-para-um.

Camila possui um carro vermelho

Figura 02 - Exemplo de relacionamento entre objetos



A classe Carro precisa “saber que pertence” a alguém. Iremos adicionar um atributo chamado dono, que é do tipo Pessoa. Exibindo a lista de atributos de Carro após a mudança:

class Carro {

String tipo;

String cor;

Strin placa;

int numPortas;

Pessoa dono;

...

}

Listagem 2 - Codificação do relacionamento entre as classes Carro e Pessoa

E os respectivos métodos get e set:

class Carro {

Pessoa getDono(){

return dono;

}

void setDono(Pessoa dono){

this.dono = dono;

}

}

Listagem 3 - Métodos que relacionam Carro a Pessoa

Vamos utilizar esse relacionamento no exemplo seguinte:

public class Main{

​

public static void main(String[] args){

​

Carro meuCarro = new Carro();

​

meuCarro.setCor("preto");

meuCarro.setNumPortas(4);

meuCarro.setPlaca("MHX 1234);

meuCarro.setTipo("esportivo");

​

Pessoa pessoa = new Pessoa();

pessoa.setNome("Camila");

pessoa.setIdade(27);

​

System.out.println("Cor: "+meuCarro.getCor());

System.out.println("Número de portas: "+meuCarro.getNumeroPortas());

System.out.println("Placa: "+meuCarro.getPlaca());

System.out.println("Tipo: "+meuCarro.getTipo());

System.out.println("Pertence a: "+meuCarro.getDono().getNome());

​

}

}

Listagem 4 - Programa exemplo do relacionamento Carro e Pessoa

Observe a linha:

System.out.println("Pertence a: "+meuCarro.getDono().getNome());

Listagem 5 - Comando com exemplo de troca de mensagem

Neste ponto, a classe Main envia uma mensagem para o objeto carro, recuperando o seu dono, um objeto da classe Pessoa, e esse recebe uma mensagem para que retorne o seu nome.

A troca de mensagens é comumente denominada de chamada de método

Até agora, você estudou a troca de mensagens entre objetos para definir ou recuperar algum atributo, unicamente. Os sistemas OO precisam de métodos mais complexos, que verifiquem a validade dos dados, realizem atualizações em outras entidades, pesquisem em banco de dados, escrevam em arquivos de log, atualizem objetos da interface etc.

Vamos colocar um pouco da complexidade do mundo real no nosso programa?! Vamos lá!

Faremos as seguintes alterações:

1. Adicionar os métodos ligar(), desligar(), acelerar(), frear() à classe Carro.

class Carro{

void ligar(){

System.out.println("carro ligado.");

}

void desligar(){

System.out.println("carro desligado.");

}

void acelerar(){

System.out.println("carro acelerando");

}

void frear(){

System.out.println("carro freando.");

}

}

Listagem 6 - Novos métodos para a classe Carro

Vamos adicionar à classe Carro, ainda, um atributo câmbio que indicará qual a “marcha” que o carro está em um dado momento, para tanto, tal atributo irá guardar um valor inteiro de 0 (zero) a 5 (cinco).   
0 - Neutro (ponto morto)  
   
1 a 5 - marchas

class Carro{

int cambio;

void setCambio(int marcha){

this.cambio = marcha;

}

int getCambio(){

return this.cambio;

}

}

Listagem 7 - Novo atributo e métodos da classe Carro

Precisamos colocar uma referência do automóvel dentro da classe Pessoa.

class Pessoa{

Carro carro;

Carro getCarro(){

return carro;

}

int setCarro(){

this.carro = carro;

}

}

Listagem 8 - Métodos que implementam o relacionamento entre Carro e Pessoa

Vamos adicionar outros métodos à classe Pessoa para interagir (trocar mensagens) com seu carro.

class Pessoa{

void ligarCarro(){

carro.ligar();

}

void desligarCarro(){

carro.desligar();

}

void acelerarCarro(){

carro.acelerar();

}

void frearCarro(){

carro.frear();

}

void setCambioCarro(int marcha){

carro.setCambio(marcha);

}

}

Listagem 9 - Classe Pessoa recebe métodos para interagir com a classe Carro

Vamos ao Main! Nosso programa precisa:

1. Criar uma Pessoa;
2. Criar um Carro e atribuí-lo a essa Pessoa;
3. Fazer com que a pessoa ligue o carro, troque de marcha, acelere, reduza a velocidade, troque de marcha e desligue o carro.

public class Main{

public static void main(String[] args){

​

Carro meuCarro = new Carro();

​

meuCarro.setCor("preto");

meuCarro.setNumPortas(4);

meuCarro.setPlaca("MHX 1234);

meuCarro.setTipo("esportivo");

​

Pessoa pessoa = new Pessoa();

pessoa.setNome("Camila");

pessoa.setIdade(27);

​

pessoa.setCarro(carro);

​

pessoa.ligarCarro();

pessoa.setCambioCarro(1);

pessoa.acelerarCarro();

pessoa.setCambioCarro(2);

pessoa.acelerarCarro();

pessoa.setCambioCarro(3);

pessoa.acelerarCarro();

pessoa.setCambioCarro(2);

pessoa.acelerarCarro();

pessoa.setCambioCarro(1);

pessoa.acelerarCarro();

pessoa.setCambioCarro(0);

pessoa.frearCarro();

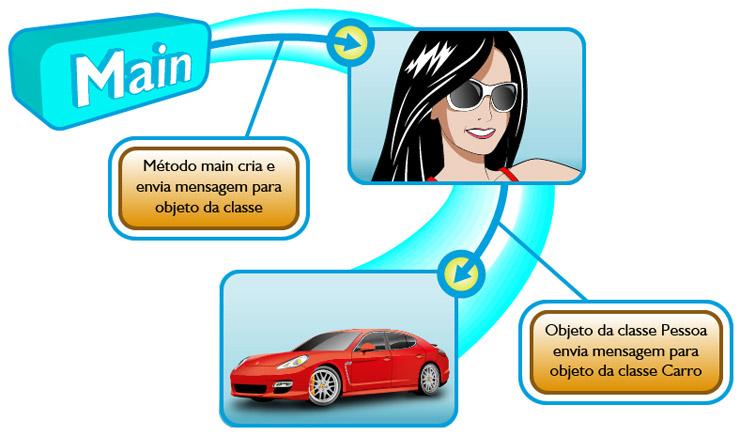
pessoa.desligarCarro();

}

1. }
2. Listagem 10 - Programa exemplo para testar o relacionamento entre Carro e Pessoa

O que podemos ver é que o método Main envia uma mensagem para o objeto pessoa, que, por sua vez, manda ou repassa essa mensagem para seu objeto da classe Carro, que executa a ação final desejada, que consiste em guiá-lo. Veja a ilustração a seguir:

Figura 03 - Envio de mensagens entre objetos



*play\_circle\_filled*

### **Encapsulamento**

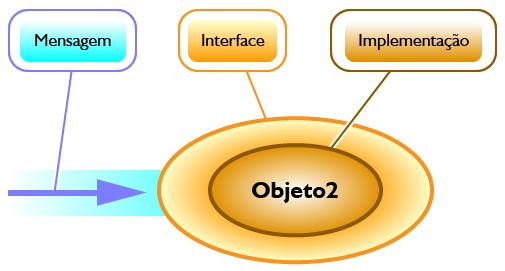
O que nos lembra a palavra encapsular? O que você diria se alguém pedisse a definição da palavra encapsulamento? E cápsula, lembra alguma coisa?

Encapsulamento é a característica da OO capaz de ocultar partes (dados e detalhes), de implementação interna de classes, do mundo exterior.

Graças ao encapsulamento, podemos ver as classes apenas pelos serviços (métodos) que elas devem oferecer para quem as utiliza. Não visualizamos, nesse caso, de que forma (como) o serviço (método) está implementado internamente na classe. No fundo, o encapsulamento da classe acaba definindo um contrato que determina o que o mundo exterior pode fazer com objetos daquela classe.

Na Figura 1, podemos ver que um objeto que possui a característica de encapsulamento fica protegido por uma cápsula. Essa cápsula, que chamamos de interface, serve para ocultar e proteger de outros objetos, os detalhes de implementação daquele objeto. Dessa maneira, o objeto só disponibiliza, através da interface, os serviços ou funcionalidades que ele deseja receber mensagens (solicitações) de outros objetos.

Figura 01 - Um objeto com encapsulamento



Para facilitar nosso entendimento, vamos a um exemplo. Considere um objeto Automóvel que disponibiliza para um objeto Pessoa (na OO tudo é objeto!) a direção como parte da sua interface, para que se possa guiar o Automóvel para a esquerda ou para a direita. Através da direção, a Pessoa solicita ao Automóvel esses serviços, sem saber COMO serão feitos ou estão implementados.

Apenas o Automóvel sabe que mecanismos serão acionados para atender a solicitação da Pessoa. A Pessoa apenas usufrui do resultado da solicitação. Observe a Figura 2.

Figura 02 - Pessoa usando a direção como interface para acessar os serviços de Automóvel



Podemos dizer, nesse caso, que estamos aplicando a característica de encapsulamento ao objeto Automóvel, pois ocultamos do objeto Pessoa os detalhes da implementação dos serviços oferecidos pelo Automóvel.

Você deve estar se perguntando: como isso acontece na prática?

Antes de mostrarmos a implementação na linguagem Java do uso do encapsulamento, precisamos conhecer os chamados Modificadores de Acesso da linguagem.

### **Antes, uma Explicação...**

Quando falarmos em ACESSO, é necessário interpretarmos: o acesso deve-se dar a quem e por quem?

Para respondermos, devemos interpretar que o acesso será ao objeto que possui encapsulamento (ou está encapsulado), e será feito pelos outros objetos do sistema ou mesmo por você (programador), que decidirá usar determinados objetos durante a construção de seus programas.

Outra pergunta seria: acesso a que? Para responder, devemos interpretar como o acesso aos atributos e métodos do objeto encapsulado.

Então, ACESSO nos traz a interpretação de acessar os atributos e métodos do objeto que possui encapsulamento pelos outros objetos do sistema ou pelos programadores.

### **Modificadores de Acesso**

Os Modificadores de Acesso são palavras-chave ou reservadas da linguagem Java cuja utilidade é permitir ou proibir o acesso aos atributos e/ou métodos das classes. Veja-os a seguir.

public: garante que o atributo ou método da classe seja acessado ou executado a partir de qualquer outra classe.

private: pode ser acessado, modificado ou executado apenas por métodos da mesma classe, sendo totalmente oculto ao programador (ou outros objetos do sistema) que for usar instâncias dessa classe.

protected: funciona como o private, exceto que as classes filhas ou derivadas também terão acesso ao atributo ou método. Veremos mais sobre classes filhas na aula de Herança.

Package ou Friendly: não são palavras reservadas de modificadores de acesso. Os atributos e métodos são chamados de Package ou Friendly, quando não possuem modificadores, ou seja, são os atributos e métodos declarados sem modificadores. Isso significa que podem ser acessados por todas as classes pertencentes a um mesmo pacote (pacotes são pastas onde estão inseridos os arquivos das classes, para ajudar a organizá-las. Falaremos sobre pacotes em aulas futuras).

## **Aula 05 - Encapsulamento**

### **Um Exemplo Completo**

Agora que você já conhece os conceitos sobre encapsulamento, vamos ver como isso acontece na prática.

Iremos mostrar um exemplo que demonstre a importância de se usar o encapsulamento na Programação Orientada a Objetos.

Vamos supor que temos uma classe Agenda que guarda apenas uma data e uma anotação importante a ser lembrada, conforme mostra a listagem 1 a seguir.

class Agenda{

int dia;

int mes;

String anotacao;

}

Listagem 1 - Classe agenda

Agora, vamos querer que essa classe anote a informação. Assim, nossa classe fica:

class Agenda{

int dia;

int mes;

String anotacao;

void anote(int d, int m, String nota){

dia = d;

mes = m;

anotacao = nota;

}

}

Listagem 2 - Adicionando comportamento

É importante também verificar se a data da anotação é uma data válida. Se for uma data válida, registra-se a anotação, caso contrário a anotação recebe uma informação de data inválida. Assim, nossa classe fica:

class Agenda{

int dia;

int mes;

String anotacao;

void anote(int d, int m, String nota){

dia = d;

mes = m;

anotacao = nota;

​

validaData();

}

void validaData(){

if ((dia<1) ||(dia>31) || (mes<1) || (mes>12)){

dia = 0;

mes = 0;

anotacao = "Anotação não inserida devido a data inválida";

}

}

void mostraAnotacao(){

System.out.println(dia+"/"+mes" : "+ anotacao);

}

}

Listagem 3 - Validação das datas

Você até agora pode está se perguntando onde está o encapsulamento e aquela conversa toda do início da aula, não é mesmo?

Calma! O objetivo desse exemplo é mostrar a fragilidade do código que não usa o encapsulamento. Por isso, iremos agora testar nossa classe Agenda com uma aplicação.

Para isso, vamos criar dois objetos com a classe Agenda: agenda1 e agenda2. Em seguida, daremos a agenda1 – uma anotação com uma data válida, e a agenda2 – uma data inválida. Finalmente, consultaremos os dados dos atributos de cada objeto, em especial a anotação que foi registrada, utilizando o método mostraAnotacao().

public class Principal{

public static void main(String[] args){

Agenda agenda1 = new Agenda();

Agenda agenda2 = new Agenda();

​

agenda1.anote(12,10,"Dia da Criança");

agenda2.anote(7,15,"Independência do Brasil");

​

agenda1.mostraAnotacao();

agenda2.mostraAnotacao();

}

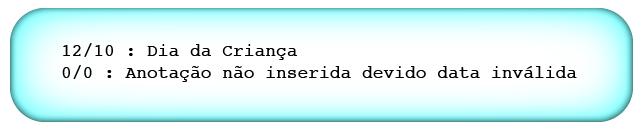
}

Listagem 4 - Testando a agenda

Para a agenda1, anotamos que dia 12 do mês 10 (Outubro) é o Dia da Criança, já para a agenda2 anotamos que dia 7 do mês 15 (mês Inexistente) é o dia da Independência do Brasil.

Com a sua atual experiência de programador, observando a classe Agenda e os dados inseridos pela classe Principal, o que você acha que será impresso após termos rodado a aplicação? Vejamos então o resultado.

Figura 03 - Saída impressa no terminal



Perfeito!!! Não utilizamos o encapsulamento na classe Agenda e tudo funcionou perfeitamente! Sem nenhuma via de acesso desprotegida, correto? Errado! Vamos identificar a falha da nossa codificação.

Veja o que acontece se fizéssemos uma pequena modificação no método main(), observe a listagem 5 a seguir.

public class Principal{

public static void main(String[] args){

Agenda agenda1 = new Agenda();

Agenda agenda2 = new Agenda();

​

agenda1.anote(12,10,"Dia da Criança");

agenda2.anote(7,15,"Independência do Brasil");

​

agenda1.mostraAnotacao();

agenda2.mostraAnotacao();

​

agenda2.dia=7;

agenda2.mes = 15;

agenda2.anotacao = "Independência do Brasil";

​

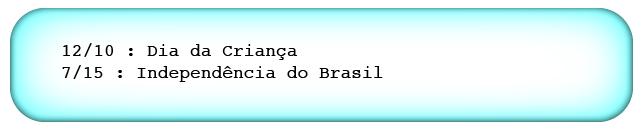
agenda2.mostraAnotacao();

}

}

Listagem 5 - Modificando o método main

O resultado dessa aplicação seria...



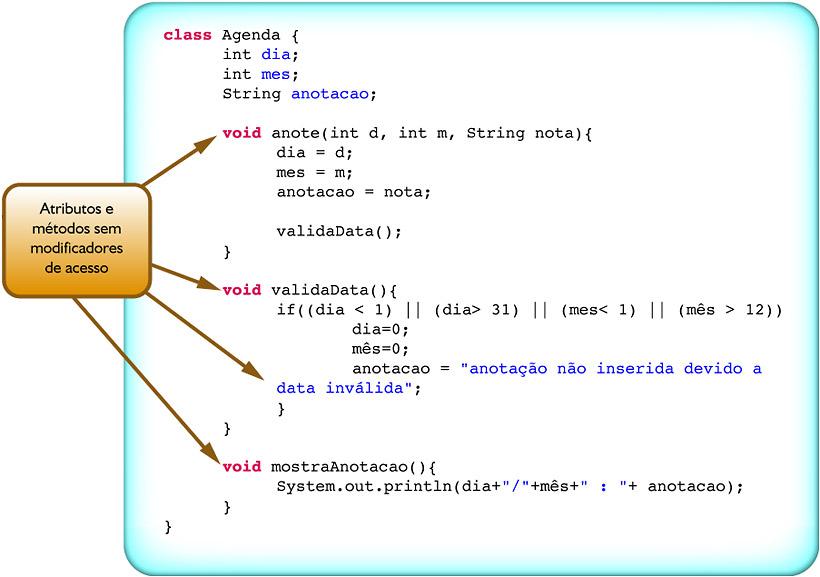
Veja que sua Agenda permitiu que você inserisse uma data inválida para uma anotação. Isso significa que seu código está suscetível a falhas. Em um programa simples como esse, isso não irá trazer nenhuma dor de cabeça. Agora, imagine em um programa real usado diariamente pelas pessoas.

Qual é a solução? Respondendo: a solução é aplicar o encapsulamento.

### **Aplicando o Encapsulamento ao Exemplo**

Codificar em OO usando o encapsulamento nada mais é que aplicar os modificadores de acesso às classes.

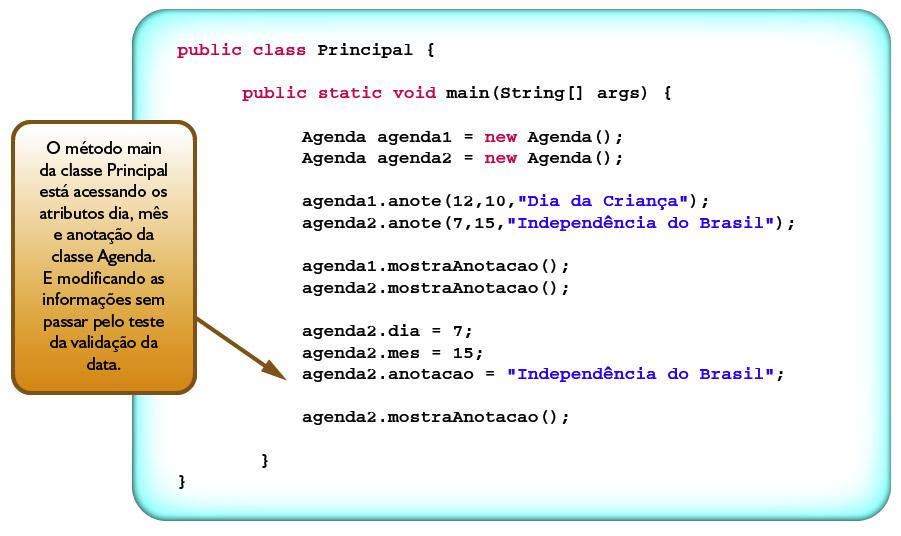
Vamos ver como está a nossa classe Agenda com relação aos modificadores.

 Listagem 6 - Exemplo sem modificadores de acesso explícito

A classe Agenda não usa nenhum dos modificadores de acesso de forma explícita. Pela nossa classificação, implicitamente ele está usando o modificador de acesso Package ou Friendly. Vimos, anteriormente, que o modificador de acesso Package permite que classes do mesmo pacote acessem atributos ou métodos que estejam com tal modificador. Considerando que a nossa classe Principal está no mesmo pacote (veja sobre pacotes na classificação de Package), então, ele tem acesso aos atributos e métodos da classe Agenda.

Você pode estar se perguntando... Quando a classe Principal teve acesso aos atributos da classe Agenda?

Respondendo: veja o trecho de código da classe Principal a seguir.



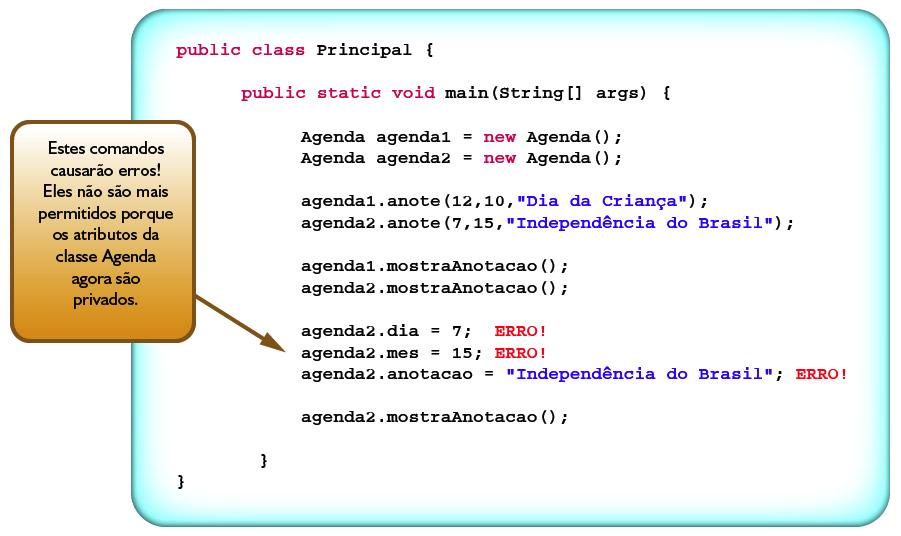
Listagem 7 - Acesso direto aos métodos e atributos da classe

O método main() da classe Principal está acessando os atributos dia, mes e anotacao da classe Agenda. E modificando as informações sem passar pelo teste da validação da data. Essa falha na codificação não é aceitável.

Vamos então aplicar o encapsulamento tornando privados os atributos e o método validaData()da classe Agenda com o uso do modificador de acesso private, e permitir os acessos para os métodos anote() e mostraAnotacao(). Temos:

 Listagem 8 - Adicionando explicitamente os modificadores de acesso

Agora, o compilador já não aceita usar os comandos abaixo. Isso causaria erro! Ou seja, os atributos da classe Agenda agora são privados apenas para a uso da própria classe Agenda, como mostra a listagem 9.

 Listagem 9 - Encapsulamento em ação

Com essa modificação, só é possível inserir uma anotação na classe Agenda usando o método anote(). O método anote() garante que a data inserida para a anotação será validada com o método validaData(), que também é privado ao uso apenas da classe Agenda.

Uma verdade que não pode ser omitida para os programadores: você pode optar em não usar a característica de encapsulamento em seus códigos.

Mas, isso seria uma péssima prática e um sinal que você não é um programador que segue as boas práticas. Se você havia pensado nisso, reveja os conceitos e tente entender a importância do encapsulamento para seus códigos.

<https://materialpublic.imd.ufrn.br/curso/disciplina/5/8/6/1>

**Composição ou Agregação**

No Universo da Programação Orientada a Objetos: eu sou uma Composição ou Agregação!!!

Composição ou Agregação é um mecanismo de reaproveitamento (reutilização) de classes utilizado pela POO para aumentar a produtividade e a qualidade no desenvolvimento de software.

Reaproveitamento ou reutilização de classes significa que você pode usar uma ou várias classes para compor outra classe. Já o aumento de produtividade está relacionado com a possibilidade de não ser necessário reescrever código de determinadas classes, se alguma outra já existe com estado (atributos) e comportamento similar. Finalmente, com a composição, é possível também aumentar a qualidade dos sistemas gerados, porque há a possibilidade clara de reutilizar classes que já foram usadas em outros sistemas, e, portanto, já foram testadas e têm chances de conter menos erros.

Vejamos um exemplo: um carro é um objeto COMPOSTO por vários outros objetos. Ele é composto pelos objetos motor, pneus, direção, faróis etc. A Figura 1 dá uma ideia do conceito de composição para um objeto do tipo Carro.

Figura 01 - Composição do objeto Automóvel



Legal não é?!! Isso mesmo que você entendeu, você pode criar um objeto a partir de vários outros objetos. E isso é muito natural.

Veja o computador que você está usando, ele é um objeto composto por outros objetos: teclado, monitor, placa mãe, memória, mouse etc.

E tem mais, quando uma classe é composta de outras classes, ela pode tanto usar os objetos que são gerados pelas classes que a compõem, como pode também usufruir dos atributos e métodos dessas classes.Vejamos agora um exemplo na prática! Vamos utilizar a mesma ideia do objeto Automóvel, dado uma classe Automóvel que é composta pelas classes Direção e Motor.

Figura 02 - Composição da classe Automóvel: motor e direção



Nas linhas de código, essa Composição é expressa da seguinte maneira, veja Listagem 1:

class Motor{

private int potencia;

}

class Direcao{

private String cor;

}

class Automovel{

private Motor motor;

private Direcao direcao;

}

Listagem 1 - Classes Motor, Direção e Automóvel

Observe que a classe Automóvel é composta pelas classes Motor e Direção, eis aqui a nossa Composição! Mas, lembre-se de que primeiro criamos as classes Motor e Direção para só a partir daí podermos criar a classe Automóvel.

Mas, o conceito e vantagens do uso da Composição não acabam por aí, analisando as classes, vamos incrementá-las um pouco mais e descobrir os benefícios de se utilizar a Composição.

Observe que a classe Automovel possui atributos que são referências para instâncias (objetos) das classes Motor e Direção. Observe que as classes possuem a propriedade do Encapsulamento (visto na aula 05, Encapsulamento), consequentemente, para acessar os atributos das classes usaremos os métodos get e set (vistos na aula 2, Classes, atributos e métodos). Assim, nossas classes ficam conforme indicam as listagens a seguir: Listagem 2, Listagem 3 e Listagem 4:

class Motor{

private int potencia;

public int getPotencia(){

return this.potencia;

}

public void setPotencia(int potencia){

this.potencia = potencia;

}

}

Listagem 2 - Classe Motor

class Direcao{

private String cor;

public String getCor(){

return this.cor;

}

public void setCor(String cor){

this.cor = cor;

}

}

Listagem 3 - Classe Direção

class Automovel{

private Motor motor;

private Direcao direcao;

​ public Motor getMotor(){

return this.motor;

}

public void setMotor(Motor motor){

this.motor = motor;

}

public Direcao getDirecao(){

return this.direcao;

}

public void setDirecao(Direcao direcao){

this.direcao = direcao;

}

}

Listagem 4 - Classe Automóvel

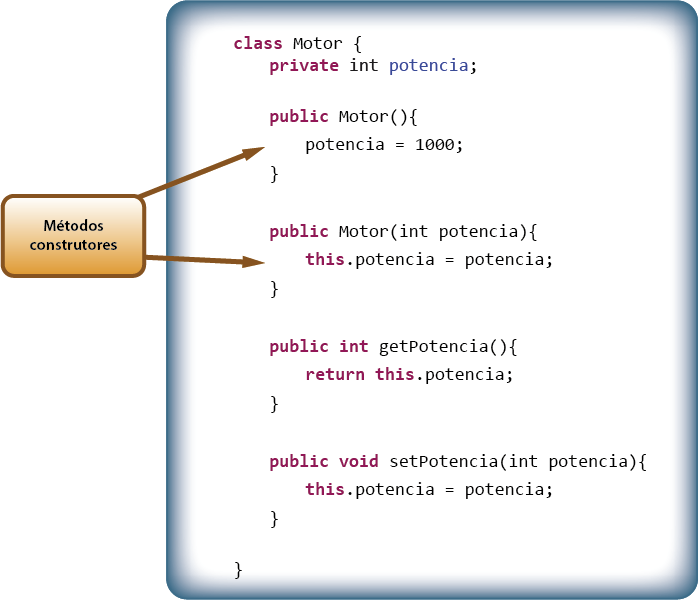
Da maneira que está o código das classes, se nós criarmos um objeto (instanciarmos) da classe Automóvel, os atributos motor e direção continuam com valor nulo (null).

Automovel automovel = new Automovel();}

Isso significa que para instanciarmos os objetos que compõem a classe Automóvel podemos decidir de que maneira faremos.

**A Composição e o Método Construtor**Os métodos construtores das classes componentes (Motor,Direção) que fazem parte da classe composta (Automóvel) podem ser chamados de três maneiras diferentes, são elas:

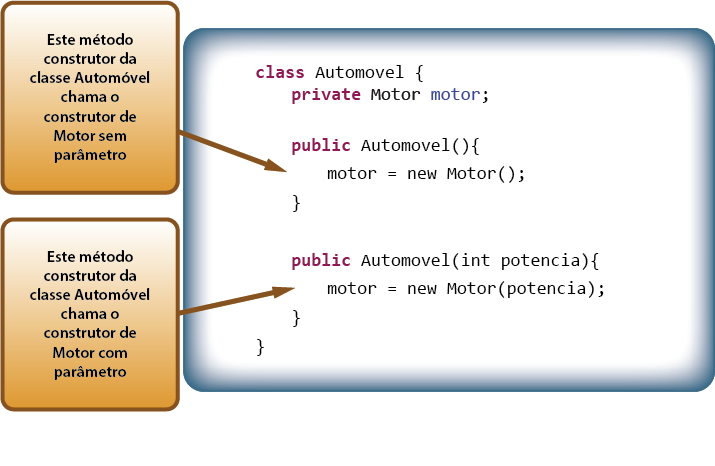
* + CASO 1: chamadas nos construtores da classe que é composta;
  + CASO 2: chamadas em qualquer método da classe que é composta;
  + CASO 3: chamadas fora da classe que é composta.

Para mostrar cada uma dessas situações, vamos considerar inicialmente que a classe Automóvel é composta apenas pela classe Motor. Acrescentamos dois métodos construtores para a classe Motor. O primeiro utiliza a potência do motor 1000 (por default), já o segundo espera que o usuário forneça a informação da potência do motor quando criado o objeto Motor, veja a Listagem 5.  
 

Listagem 5 - Classe Motor com dois construtores

class Automovel{  
 Private Motor motor;  
   
 public Motor getMotor(){  
 return this.motor;  
}  
​  
public void setMotor(Motor motor){  
 this.motor=motor;  
}  
}

Listagem 6 - Classe Automóvel composta pela classe Motor

CASO 1: Chamadas nos construtores da classe que é composta   
Para a primeira situação, quando o construtor da classe componente é chamado no construtor da classe que usa a composição, ilustrada na Listagem 7, temos:  
 

Listagem 7 - Classe Automóvel com dois métodos construtores   
Nesse exemplo, os construtores da classe Motor são chamados dentro dos construtores da classe Automóvel.

CASO 2: chamadas em qualquer método da classe que é composta   
Outra maneira seria a situação, na qual os construtores são chamados em qualquer método da classe que é composta. Vejamos a Listagem 8:

class Automovel{  
 private Motor motor;  
​  
 public ligarPrimeiraVez(){  
 motor = new Motor;  
 }  
​  
 public ligarPrimeiraVez(int potencia){  
 motor = new Motor(potencia);  
 }  
}

Listagem 8 - Construtor chamado a partir de um método

CASO 3: chamadas fora da classe que é composta   
Por fim, a terceira situação, onde o construtor da classe componente é chamado de fora da classe que usa a composição, veja a Listagem 9. Considere que a classe Automóvel utilizada possui o código apresentado na Listagem 6.  
  Listagem 9 - Criação da classe Motor fora da classe Automóvel   
 Observe que o construtor da classe Motor foi chamado de fora da classe Automóvel, dentro do método main() da classe. Ou seja, antes de usar o método setMotor() o objeto motor não tem nenhum vínculo com o objeto automóvel.

### **Comparando as Possibilidades**

Observando a Listagem 10, analisaremos as possíveis maneiras de criar o objeto da classe componente (Motor) através da classe composta (Automóvel) ou dentro do método main() que define o comportamento de execução de um programa Java.

public class Main{

public static void main(String[] args){

//CASO 1

Automovel automovel = new Automovel();

Automovel automovel = new Automovel(1600);

​

//CASO 2

Automovel automovel = new Automovel();

automovel.ligarPrimeiraVez();

//outra maneira

automovel.ligarPrimeiraVez(1600);

​

//CASO 3

Automovel automovel = new Automovel();

Motor motor = new Motor();

//outra maneira

Motor motor = new Motor (1600);

automovel.setMotor(motor);

}

}

Listagem 10 - Possibilidades de criação da classe Motor para uso pela classe Automóvel

Para o caso 1: temos duas possibilidades, na linha 1, é ocultado do usuário a existência de um objeto motor que compõe a classe Automóvel. Já na linha 2, o usuário define a potência a ser adotada pelo motor do automóvel.

Para o caso 2, podemos observar na linha 3 que quando criamos o objeto automóvel o objeto motor ainda não existe. Ele permanece nulo (null) dentro do objeto motor até que o automóvel seja ligado pela primeira vez. Para isso, pode-se usar um dos dois métodos ligarPrimeiraVez()(linhas 4 e 5), sem ou com parâmetro, respectivamente.

Finalmente, para o caso 3, apresentado a partir da linha 6, tem-se a mesma observação da linha 3. Nesse caso, o usuário cria normalmente o objeto motor, escolhendo um dos dois construtores (linhas 7 e 8) e, em seguida, define que esse objeto componha o objeto automóvel (linha 9.)

### **Composição e os Modificadores de Acesso**

Duas observações importantes que podemos considerar com relação à composição e aos modificadores de acesso são mencionadas a seguir.

* Quando declaramos atributos públicos nas classes e reutilizamos essas classes dentro de outras, esses atributos podem não ser acessados facilmente, através da classe de composição, veja Listagem 11.

class Direcao{

public String cor;

}

class Automovel{

private Direcao direcao;

}

public class Main{

public static void main(String[] args){

Automovel automovel = new Automovel();

automovel.direcao.cor ="preta"; // ERRO!

}

}

Listagem 11 - Tentativa de uso de um atributo público de um objeto privado

No exemplo acima, apesar do atributo cor da classe Direção ser public, esse atributo não é diretamente acessível a partir do método main(), porque o objeto Direção está encapsulado (é um atributo privado) dentro da classe Automóvel.

Quando temos atributos privados nas classes e reusamos essas classes, declarando suas instâncias como públicas, os atributos não passam a ser públicos, veja a Listagem 12.

class Direcao{

private String cor;

}

class Automovel{

public Direcao direcao;

}

public class Main{

public static void main(String[] args){

Automovel automovel = new Automovel();

automovel.direcao.cor ="preta"; // ERRO!

}

* }

Listagem 12 - Tentativa de uso de um atributo privado de um objeto público.

Aqui ocorre o inverso do caso anterior, mas reforça que private não deixa de ser privado mesmo quando na classe que usa da Composição (Automóvel) torna seu atributo (Direção) public.

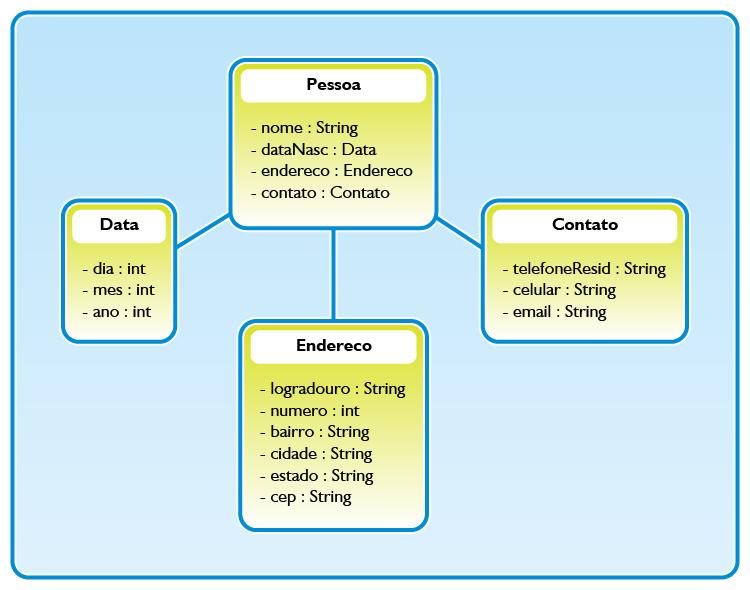
**Anote a Dica!**

Quando uma classe usa a Composição para agregar outras classes, podemos dizer que ela tem um relacionamento chamado “Tem um”, o qual descreve um relacionamento em que uma classe contém uma instância de outra classe.

### **Resumo**

Nesta aula, você estudou o que é Composição. Viu que composição é a capacidade de fazer com que uma classe seja composta de vários objetos de outras classes. A Composição oferece a possibilidade de fazer uso do comportamento das classes que ela agrega (como atributos) de maneira implícita ou explícita, dependendo da maneira que se escolhe para instanciar as classes componentes.

1. Crie as classes apresentadas no diagrama abaixo e aplique a Composição para a classe Pessoa, que além de possuir um atributo Nome será composta pelas classes Data, Endereço e Contato para os atributos dataNasc, endereço e contato, respectivamente.



Herança

Hoje, você vai aprender sobre mais um pilar de sustentação da Programação Orientada a Objetos: a herança. Ela é considerada um dos conceitos mais importantes na POO. A herança trouxe para a programação um novo olhar sobre a maneira de se programar, até então não atendido pela Programação Estruturada.

### **Herança**



Calma, você não recebeu uma herança em bens dos seus parentes!!! Mas, a Herança na POO tem uma certa ligação com grau de parentesco. Vamos descobrir que os objetos podem ter Mãe!

Eu sou um objeto?!?! Eu quero a minha Mãe!!!

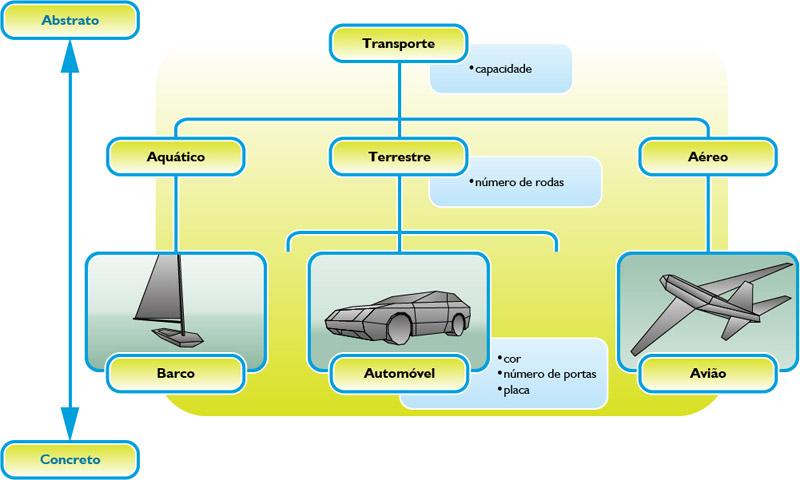
Você já ouviu falar sobre herança relacionada à programação? Estranho não é? A Figura 1 mostra os pilares da orientação objetos com destaque para a herança.

Figura 01 - Orientação a objetos



Vamos começar pela definição de herança. Herança é o mecanismo que permite a uma classe herdar todos os atributos e métodos de outra classe. Ela permite definir a implementação de uma nova classe na definição de uma classe previamente implementada. Como exemplo, observe a Figura 2 a seguir.

Figura 02 - Exemplo de herança entre classes



Considere como classes as seguintes abstrações: Transporte, Aquático, Terrestre, Aéreo, Barco, Automóvel e Avião. Considere capacidade como um atributo da classe Transporte, que indica a quantidade de pessoas que o transporte em questão pode transportar. E número de rodas como um atributo da classe Terrestre. E cor, número de portas e placa como atributos da classe Automóvel.

A Figura 2 apresenta um exemplo de herança, onde as classes Aquático, Terrestre e Aéreo herdam da classe Transporte.

A classe Barco herda da classe Aquático. A classe Automóvel herda de Terrestre. E, finalmente, Avião e Aéreo.

Um aspecto importante que também podemos observar na figura é que toda classe que herda de uma outra acaba herdando também, como consequência, os seus atributos. Por exemplo, de acordo com a Figura 2, a classe Transporte possui um atributo chamado capacidade. Para a classe Aquático, como ela herda de Transporte, pode-se dizer que também possui o atributo capacidade. E Barco, como herda de Aquático, também possui o atributo capacidade. Com essa lógica, percebemos que na herança os atributos (e também os métodos) são herdados naturalmente pelas classes subsequentes na hierarquia.

Hierarquia de Classe ou de Herança: é o mapeamento do tipo árvore de relacionamentos que se formam entre as classes como resultado da herança. Exemplo: a Figura 2 representa uma hierarquia de herança ou hierarquia de classe.

Usando a mesma lógica, responda: Quantos e quais são os atributos da classe Terrestre? E da classe Automóvel?

Respondendo: a classe Terrestre possui dois atributos: capacidade (que é herdado de Transporte) e número de rodas. Já a classe Automóvel possui cinco atributos: capacidade (herdado de Transporte), número de rodas (herdado de Terrestre), cor, número de portas e placa.

Observe na Figura 2 que, quanto mais alta na hierarquia está a classe, mais ela tende a ser abstrata em comparação com as suas subsequentes. Ou seja, quanto mais alta na hierarquia, menos definida (abstrata) é a classe, e assim ela define menos atributos e métodos. Isso também garante que a classe tenha mais chances de ser reusada por outras classes que herdem da mesma. Entendeu? Não, então, vamos ao exemplo da figura. Suponha que a classe Transporte, além do atributo capacidade, possuísse também o atributo número de rodas. Ou seja, ela passa a ser uma classe mais concreta, menos abstrata. Mas, nesse caso, consequentemente, não seria interessante nem faria sentido para as classes Aquático e Aéreo herdar os atributos de Transporte (ou herdar de Transporte),pois o atributo número de rodas não é desejado por tais classes.

### **Termos Usados para Herança**

Na literatura, são encontrados diversos termos para nomear tanto as classes que fornecem a herança quanto as classes que herdam de alguma outra. Veja exemplos de termos usados no Quadro 1 a seguir.

|  |  |
| --- | --- |
| **Classes que Fornecem a Herança** | **Classes que Herdam de Outras** |
| Superclasse | Subclasse |
| Mãe | Filha |
| Tipo | Subtipo |

Quadro 1 - Termos usados na herança

Nota: a classe filha não pode remover os atributos e métodos da classe mãe. Abaixo, são apresentados outros termos que são também comumente usados.

* Ancestral: é uma classe que aparece na hierarquia de classes em uma posição acima da progenitora (mãe).
* Descendente: dada uma classe, toda classe que aparece abaixo dela na hierarquia é uma descendente da classe dada.
* Raiz (ou Base): é a classe topo da hierarquia.
* Folha: é uma classe sem filhas.

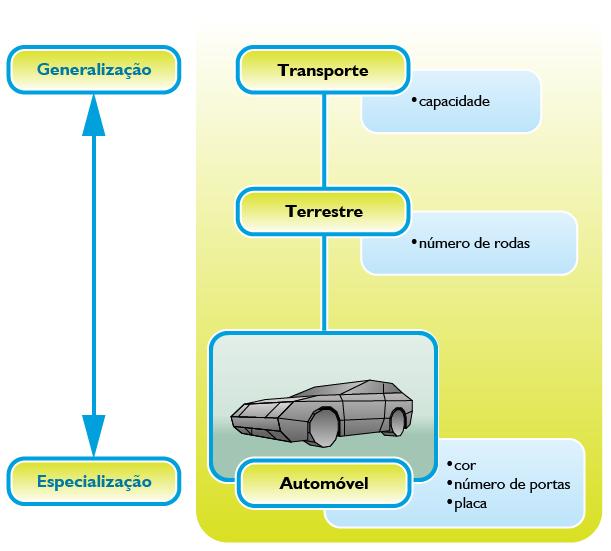
## **Herança**

### **Especialização e Generalização**

Assim como os conceitos de classes Abstratas e Concretas, tem-se também nos extremos da árvore hierárquica de herança, os conceitos de Generalização e Especialização. Na Generalização, como o próprio nome sugere, há classes mais genéricas e abstratas disponíveis, as quais podem ser usadas para outras descenderem delas. Já a Especialização é usada para indicar que classes que estão numa posição inferior na hierarquia possuem estado e comportamento mais especializados, ou seja, com mais detalhes de informações.

A Figura 3 ilustra tais conceitos dentro da hierarquia de classes de transportes, apresentada anteriormente. Como pode ser observado, classes em posição inferior na hierarquia, tal como a classe Automóvel, representam especializações de classes em posição superior (Transporte, Terrestre) na hierarquia. Já classes em posição superior, como a classe Transporte, representam generalizações de classes em posições inferiores (Automóvel, Terrestre).

Figura 03 - Generalização e Especialização



### **Herança Múltipla e Simples**

Herança Múltipla: é a capacidade de uma classe possuir mais de uma superclasse e herdar as variáveis e métodos combinados de todas as superclasses.

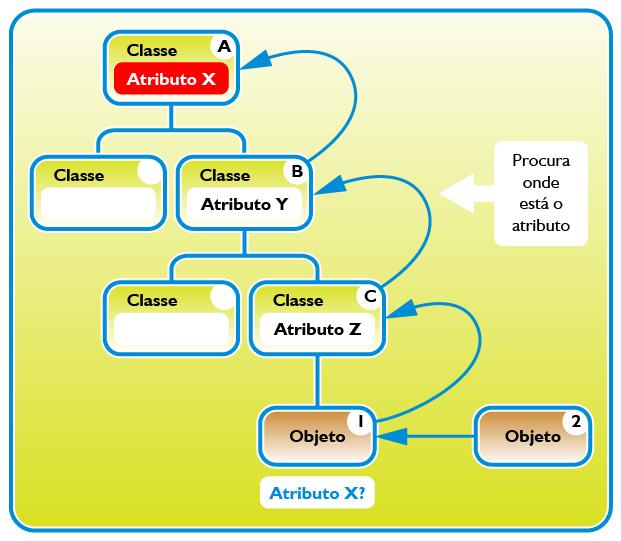
Herança Simples: Cada classe pode ter apenas uma superclasse, embora uma superclasse possa ter várias subclasses.

Na linguagem Java, a herança é simples e na codificação usa-se a palavra reservada extends para declarar que uma classe é herdeira de outra. Para simular a herança múltipla em Java, usa-se Interfaces.

### **Funcionamento da Herança**

Você viu os conceitos sobre herança, mas é importante também que possamos entender o que acontece com uma classe que utiliza a herança durante sua execução. Para melhor entendermos o funcionamento da herança durante a execução do programa, vejamos a Figura 4.

Figura 04 - Herança em ação

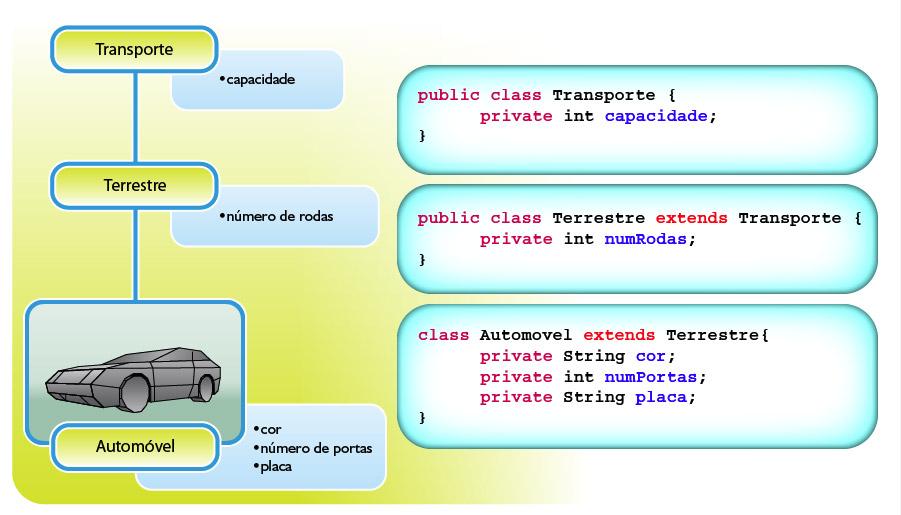


Sabemos que quando criamos um objeto, temos uma instância concreta da classe a qual esse objeto representa. Na Figura 3, esse objeto é representado pelo Objeto1. Agora, vamos supor que o Objeto2 quer saber o valor do atributoX desse objeto, através do envio de uma mensagem. Para obter o valor do atributo X, o Objeto1 sai consultando sua árvore hierárquica de classes até encontrar o atributo solicitado pelo Objeto2. No caso da Figura 4, o objeto consulta a classe da qual ele foi instanciado (ClasseC), não encontrando, consulta a classe mãe dessa mesma (ClasseB), e assim sucessivamente até encontrar o atributo e o valor. Vale ressaltar que o Objeto1 não percorre toda a árvore hierárquica, percorre o caminho da classe filha em direção à classe mãe.

Outro ponto importante é que o objeto não acumula todos os atributos das classes que fazem parte da hierarquia de herança, mas apenas os atributos das classes ancestrais a ele. O usuário (programador ou outros objetos, tal como o Objeto 2 na Figura 4) de um determinado objeto não sabe (nem precisa saber) se o atributo é dele ou se o atributo é herdado de uma outra classe ancestral dele.

### **Herança em Java**

Depois de termos visto os conceitos sobre herança, vamos finalmente ver a herança nas linhas de código Java. Como mencionamos anteriormente, em Java a palavra-chave extends é usada para indicar que uma dada classe herda de outra. A Listagem 1 apresenta exemplos de código em Java, indicando que Terrestre herda de Transporte e que Automóvel herda de Terrestre.



Listagem 1 - Exemplo de Herança em Java

Vale lembrar que quando uma classe herda de outra, todos os atributos (e métodos) da outra classe passam a fazer parte dessa mesma. Isso significa que, para o exemplo acima, o programador (ou outro objeto) pode solicitar à classe Automóvel os atributos de Transporte e Terrestre. Os atributos de Terrestre são acessíveis na classe Automóvel porque a segunda herda da primeira. E os atributos de Transporte são acessíveis a Automóvel devido ao fato de Terrestre herdar de Transporte.

Uma dúvida que pode estar surgindo é: como os atributos são acessíveis se eles estão declarados como private?

Respondendo: considere que para os códigos das classes da Listagem 1 foram declarados os métodos get e set para cada um dos atributos. Através desses métodos, que são também herdados pelas subclasses, podemos acessar facilmente qualquer um dos atributos herdados. Eles só não foram escritos para ressaltarmos a construção extends de Java e definir a herança entre classes. Veremos códigos mais completos na próxima aula.

#### **Anote as Dicas!**

1. A classe Object: todas as classes em Java descendem de uma classe, chamada Object, mesmo que a declaração extendsObject seja omitida, a classe Object é considerada a classe raiz da hierarquia de todas as classes Java, sendo, portanto, ancestral de todas as classes da linguagem.
2. Quando uma classe usa a relação de herança, podemos dizer que essa classe possui um relacionamento chamado “É um” com a classe da qual ela herda. Tal relação também indica que uma classe é do mesmo tipo que outra. Assim, nos exemplos anteriores, podemos dizer que Automóvel “é um” transporte Terrestre, assim como que Terrestre “é um” (ou tipo de) Transporte.

## **Herança II**

### **A Herança e o Protected**

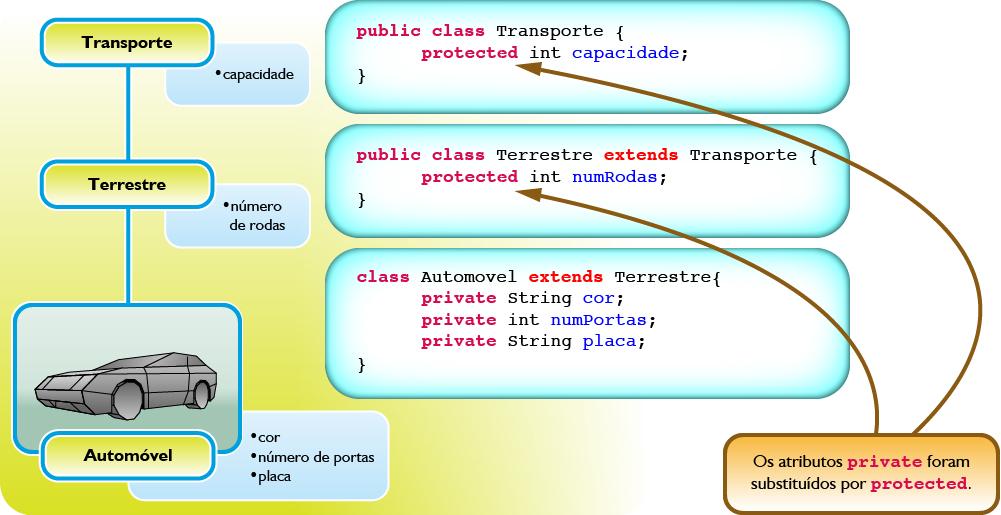
Olá, aqui estamos novamente. Descobrimos na aula passada que objetos (como eu) também têm ancestrais e descendentes!!! Mas alguns pontos importantes ainda precisam ser abordados sobre herança. Por isso, hoje iremos continuar falando sobre esse assunto.

Na Aula 05 (Encapsulamento), você viu os tipos de modificadores de acesso: public, private e protected. Esse último, ficamos de explicar melhor durante as aulas sobre herança.

Naquela aula, nós mencionamos que o modificador protected funciona como o private, exceto que as classes filhas também terão acesso ao atributo ou método declarado como protected. Isso significa que apenas as classes descendentes de uma determinada classe poderão ter acesso aos atributos e métodos declarados com esse modificador.

Veja o exemplo abaixo:

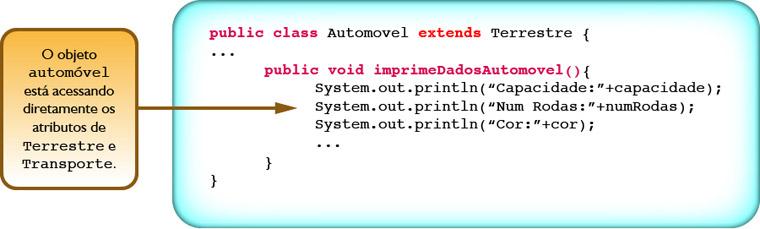
Figura 01 - Esquema de herança. Listagem 1 - Herança em Java proteced e private



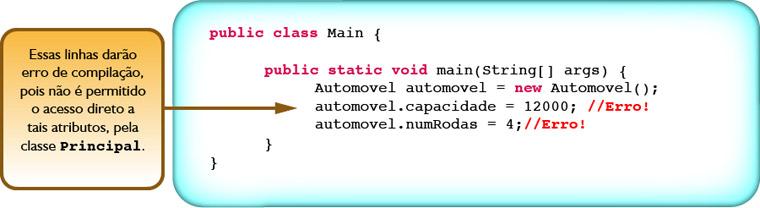
A Figura 1 e a Listagem 1 apresentam o exemplo da aula anterior com algumas pequenas modificações.

As classes Transporte e Terrestre tiveram seus atributos modificados para serem protected, isso significa que apenas classes que pertencem à hierarquia de herança podem acessar diretamente esses atributos. Em outras palavras, apenas subclasses (ou classes descendentes) de Transporte e Terrestre terão acesso aos seus atributos protected.

A Listagem 2 abaixo mostra a definição de mais um método na classe Automóvel, o qual faz acesso aos atributos protected, capacidade e numRodas, definidos nas classes ancestrais Transporte e Terrestre, respectivamente. Observe que tais atributos são acessados e usados livremente no método imprimeDadosAutomovel(). Isso só é possível porque agora eles foram declarados como protected.

 Listagem 2 - Acesso a atributos protected

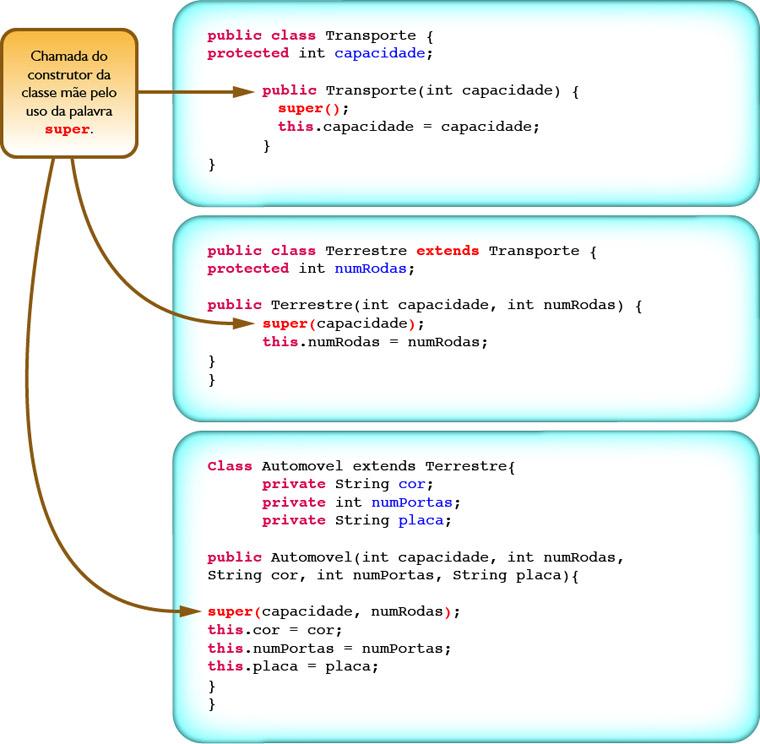
A Listagem 3 mostra uma tentativa frustrada de acesso aos atributos capacidade e numRodas herdados pela classe Automóvel, dentro de um método main(). Nesse caso, não é possível acessar tais atributos, porque a classe Principal não herda da classe Terrestre e, portanto, não tem direito a acessar os atributos protected.

 Listagem 3 - Acesso a atributos protected

### **Herança e o Super**

Em suas pesquisas em outras fontes (tutoriais na internet, livros), além do material desta aula, você pode ter encontrado o uso da palavra (ou operador) super nos códigos exemplos.

A palavra super é mais uma das palavras reservadas da linguagem Java que tem uma forte ligação com a herança. A palavra super refere-se à classe ancestral imediata da classe, ou seja, à classe mãe ou super-classe. Ela é usada nos construtores para chamada de construtores em cascata das classes mães.

 Listagem 4 - Herança em Java com uso do operador super

A Listagem 4 ilustra um exemplo de herança em Java, que faz uso da palavra-chave super. Vamos então entender o que acontece quando usamos o comando super em tais contextos.

Primeiro, observe que o super é usado para chamar o método construtor da classe mãe. Na classe Transporte, como a classe mãe de Transporte é Object (vimos na aula passada que Object é o ancestral de todas as classes), o super não tem parâmetro. A chamada de super equivale a uma chamada explícita ao construtor sem parâmetro da classe Object.

Já no construtor da classe Terrestre, o super tem como parâmetro a capacidade exigida no construtor da classe Transporte. Portanto, a chamada a super na classe Terrestre é no fundo a chamada ao construtor de Transporte. Já no construtor da classe Automóvel, o super tem como parâmetros a capacidade e o numRodas exigidos pelo construtor da classe Terrestre.

Observando a Listagem 4, dá para observar que uma chamada a super ocasiona a invocação do construtor da classe mãe. Isso acaba permitindo uma chamada em cadeia dos construtores de classes ancestrais, permitindo assim a configuração de todos os atributos herdados da classe.

Vale ressaltar que se o construtor não possui parâmetro, o compilador Java aceita a omissão do super sem parâmetro, porque durante o processo de compilação ele insere explicitamente tal chamada. Para o nosso exemplo, se quiséssemos, poderíamos omitir o super do construtor da classe Transporte.

**Anote as Dicas**

1. Apenas comentários são permitidos antes da palavra super nos construtores. Assim, não é possível incluir nenhum comando antes de super() no código de métodos construtores de classes.
2. Da mesma maneira que se usa a palavra-chave this para acessar os atributos (ou métodos) da própria classe, pode-se usar super para acessar os atributos (ou métodos) da classe mãe.

### **Um Exemplo Completo**

Vimos até agora diversos exemplos do uso de herança para acessar atributos das classes ancestrais. Vamos mostrar agora a herança utilizando os métodos dessas classes.

public class Transporte{

protected int capacidade

public Transporte(int capacidade){

super();

this.capacidade = capacidade;

}

public int getCapacidade(){

return capacidade;

}

public void setCapacidade(int capacidade){

this.capacidade = capacidade;

}

}

Listagem 5 - Classe Transporte

public class Terrestre extends Transporte{

protected int numRodas;

public Terrrestre(int capacidade, int num Rodas){

super(capacidade);

this.numRodas = numRodas;

}

public int getNumRodas(){

return numRodas;

}

public void setNumRodas(int numRodas){

this.numRodas = numRodas;

}

}

Listagem 6 - Classe Terrestre

public class Automovel extends Terrestre{

private String cor;

private int numPortas;

private String placa;

​

public Automovel(){

super(5,4);

}

​

public Automovel(int capacidade, int numRodas, String cor, int numPortas, String placa){

super(capacidade,numRodas);

this.cor= cor;

this.numPortas = numPortas;

this.placa= placa;

}

​

public String getCor(){

return cor;

}

​

public void setCor(String cor){

public int getNumPortas;

}

​

public void setNumPortas(int numPortas){

this.numPortas = numPortas;

}

​

public void setPlaca(String placa){

this.placa = placa;

}

public String getPlaca(){

return placa;

}

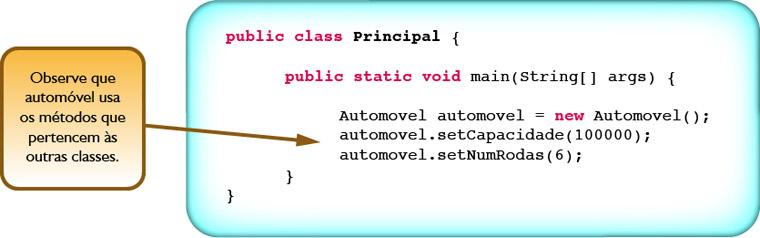
}

Listagem 7 - Classe Automóvel

As Listagens 5, 6 e 7 apresentam o código das classes Veículo, Terrestre e Automóvel, incluindo a implementação dos métodos get e set para cada atributo das classes.

Cada um dos métodos declarados na classe Veículo são herdados pelas classes Terrestre e Automóvel, assim como os métodos declarados pela classe Terrestre são herdados pela classe Automóvel, exatamente como ocorre com os atributos.

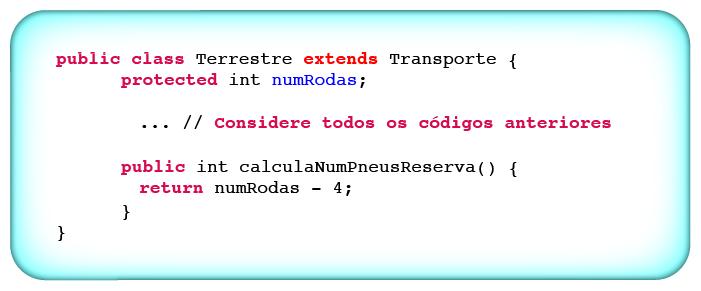
A Listagem 8 apresenta um método main() que cria um objeto da classe Automóvel, em seguida, chama dois métodos setCapacidade() e setNumRodas(). Observe pelas Listagens 5 , 6 e 7 que tais métodos não pertencem à classe Automóvel, mas são na verdade herdados de suas ancestrais. Dessa forma, percebe-se claramente que uma vez declarado um método (público ou protected) em uma das classes ancestrais, aquele método é visível em objetos das classes filhas.

 Listagem 8 - Classe Principal

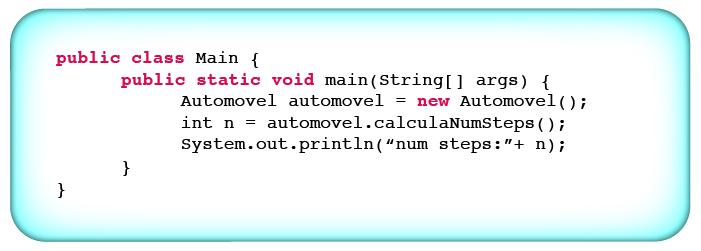
Uma pergunta: será que a herança também permite o uso de outros métodos diferentes dos get e para as classes descendentes?

A resposta é sim, a herança serve para todos os métodos, desde que a classe mãe permita. Se um método estiver com modificador private, esse método não estará acessível para classes filhas. Assim, apenas estão acessíveis para classes filhas, os métodos declarados como public e protected na classe mãe.

Vejamos um exemplo do uso de outros métodos. Vamos acrescentar agora à classe Terrestre um método que calcula o número de pneus reservas.

 Listagem 9 - Classe Terrestre com método calculaNumPneusReserva()

Na Figura 10, vemos a classe Principal modificada para criar um objeto Automóvel e acessar diretamente o método calculaNumPneusReserva() que foi herdado da classe Transporte. Como era de se esperar, mesmo o método sendo da classe Terrestre a classe Automóvel usa sem restrição.

 Listagem 10 - Método main modificado

### **Funcionamento da Herança para os Construtores**

Vimos que os construtores são invocados em tempo de execução quando o programador digita new para instanciar alguma classe, como no exemplo da classe carro acima. Mas, o que realmente acontece quando fazemos new Carro()?

Digamos que a classe Carro extends Veiculo e que Veiculo extends Object, como já vimos anteriormente, a palavra-chave extends é utilizada para determinar que uma classe herde de outra. Entraremos em mais detalhes sobre herança nos capítulos seguintes. Por agora, basta entender que Carro é uma subclasse de Veiculo e que Veículo é uma subclasse de Object. Exemplo:

class Carro extends Veiculo {}

class Veiculo extends Object {}

Agora, o que o acontece quando invocamos new Carro()?

1. O construtor de carro será invocado. Todo construtor invoca o construtor da sua superclasse com uma chamada implícita para super(), a menos que a classe invoque um construtor sobrecarregado da mesma classe (falaremos sobre isso depois).
2. O construtor da classe Veiculo será chamado, que por sua vez chamará o construtor da classe Object.
3. Como a classe Object é a última super classe de todas as classes, podemos concluir que a classe Veiculo extends Object mesmo que você não declare a chamada extends explicitamente. Chegando ao topo da pilha, o construtor da classe Object será executado primeiramente antes de todos os outros. Os seus atributos de instância serão inicializados e finalizamos o seu construtor.
4. Fazendo o caminho de volta, então, é chamado o construtor da classe Veículo e os seus atributos de instância agora terão seus valores inicializados. Finalizamos o construtor da classe Veículo.
5. E por fim, o construtor da classe Carro também será chamado e os valores de seus atributos de instancia serão inicializados. E finalizamos o construtor de Carro.

A figura a seguir demonstra como os construtores funcionam na pilha.

|  |
| --- |
| 4. Object() |
| 3. Veiculo chama superObject() |
| 2. Carro chama super Veiculo() |
| 1. main() chama new Carro() |

O que faz o operador super?

* Ele chamará o construtor da superclasse para que essa possa realizar as suas inicializações antes mesmo que todas as suas classes filhas.
* Uma chamada para super() poderá ser sem parâmetros ou com parâmetros, dependendo dos argumentos especificados no construtor da superclasse.

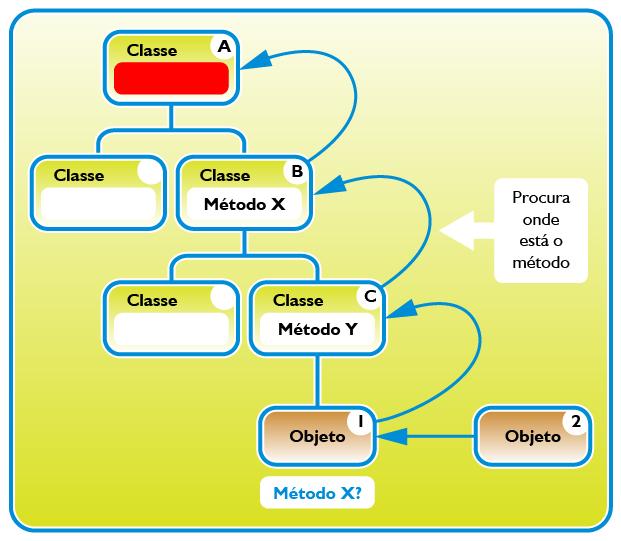
Você não pode fazer uma chamada para um método de instância ou um atributo de instância, até que o construtor da sua superclasse tenha finalizado.

### **Funcionamento da Herança para os Métodos**

A herança funciona para os métodos durante a execução do programa, de maneira similar ao funcionamento para os atributos (ver Aula 10 – Coleções em Java). A Figura 2 ilustra tal situação.

Quando o Objeto2 solicita a execução do metodoX() ao Objeto1, esse último inicialmente procura pelo método na própria classe C que o define. Caso não encontre, ele sai percorrendo sua árvore hierárquica. Nesse caso, em particular, o metodoX() é encontrado logo na primeira classe ancestral visitada, a ClasseB.

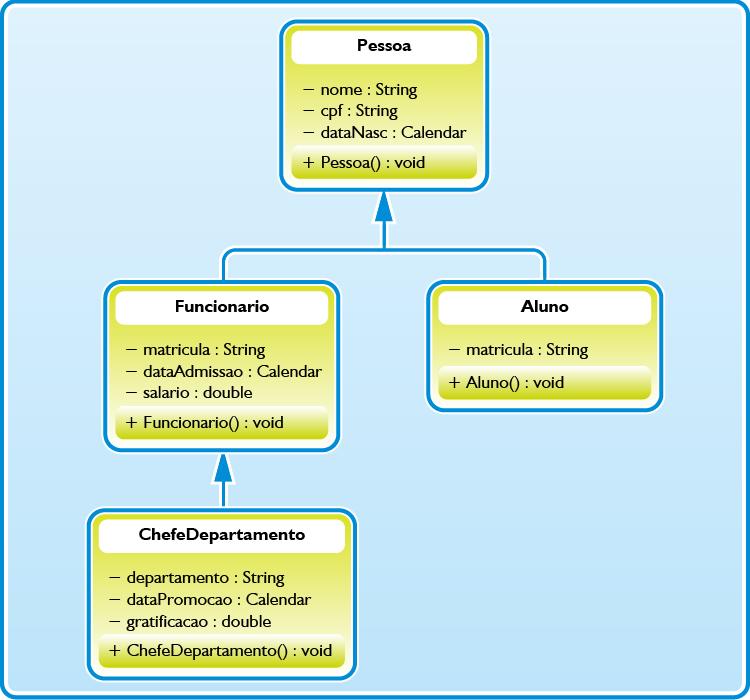
Figura 02 - Herança e os métodos



### **Resumo**

Nesta aula, você aprendeu que a Herança é a capacidade que uma classe tem de herdar as características e comportamentos (atributos e métodos) de outra classe. Em particular, enfatizamos a herança de métodos entre classes. Você viu que um método de uma classe mãe para ser herdado em Java por uma classe filha não pode ser declarado como private.

Exercicios selecionados

1. Descreva o que acontece com o acesso aos atributos e métodos quando são do tipo:
   1. public
   2. private
   3. protected
3. Crie as classes utilizando o princípio da herança, obedecendo à hierarquia da figura abaixo (obs.: para facilitar, substitua na figura o tipo Calendar por String). 
   1. Acrescente aos construtores a lista de parâmetros necessária para instanciar o objeto. Por exemplo, a classe Pessoa deve ter nome, CPF e dataNasc. E essa lista é acumulativa, ou seja, o construtor da classe Funcionário deve ter a lista de seus atributos mais os atributos necessários para a classe Pessoa. Dica: não deixe de usar a palavra-chave super em cada um dos construtores para chamar o construtor da classe mãe, passando os atributos que são mantidos por ela e seus ancestrais.
   2. Insira os seguintes métodos para apresentar os valores dos atributos das classes, mostrarPessoa(), mostrarFuncionario(), mostrarChefe() e mostrarAluno(), respectivamente, às classes Pessoa, Funcionário, ChefeDepartamento e Aluno. Para imprimir os atributos, use o método System.out.println() em cada um dos métodos.
   3. Crie uma classe TestaTudo com um método main(), que instancia um objeto de cada uma das classe e exibe os valores dos atributos através de chamadas aos métodos mostrarPessoa(), mostrarFuncionario(), mostrarChefe() e mostrarAluno().

Polimorfismo

Apresentação

Hoje, iremos iniciar o aprendizado do último princípio que serve de base para a Programação Orientada a Objeto – o Polimorfismo. Esse princípio também aposta na ideia da reutilização para facilitar o dia a dia da programação. Ele é também bastante importante tanto para o entendimento de programas OO em Java e outras linguagens, como também é um mecanismo bastante sofisticado para permitir a reutilização e flexibilidade durante o desenvolvimento de tais programas.

https://materialpublic.imd.ufrn.br/curso/disciplina/5/8/9/2

Polimorfismo

O polimorfismo deriva da palavra polimorfo, que significa multiforme, ou que pode variar a forma. Para a POO, polimorfismo é a habilidade de objetos de classes diferentes responderem a mesma mensagem de diferentes maneiras. Ou seja, várias formas de responder à mesma mensagem. Veja a figura a seguir para entender onde se localiza o pilar do polimorfismo dentro da Programação Orientada a Objetos.

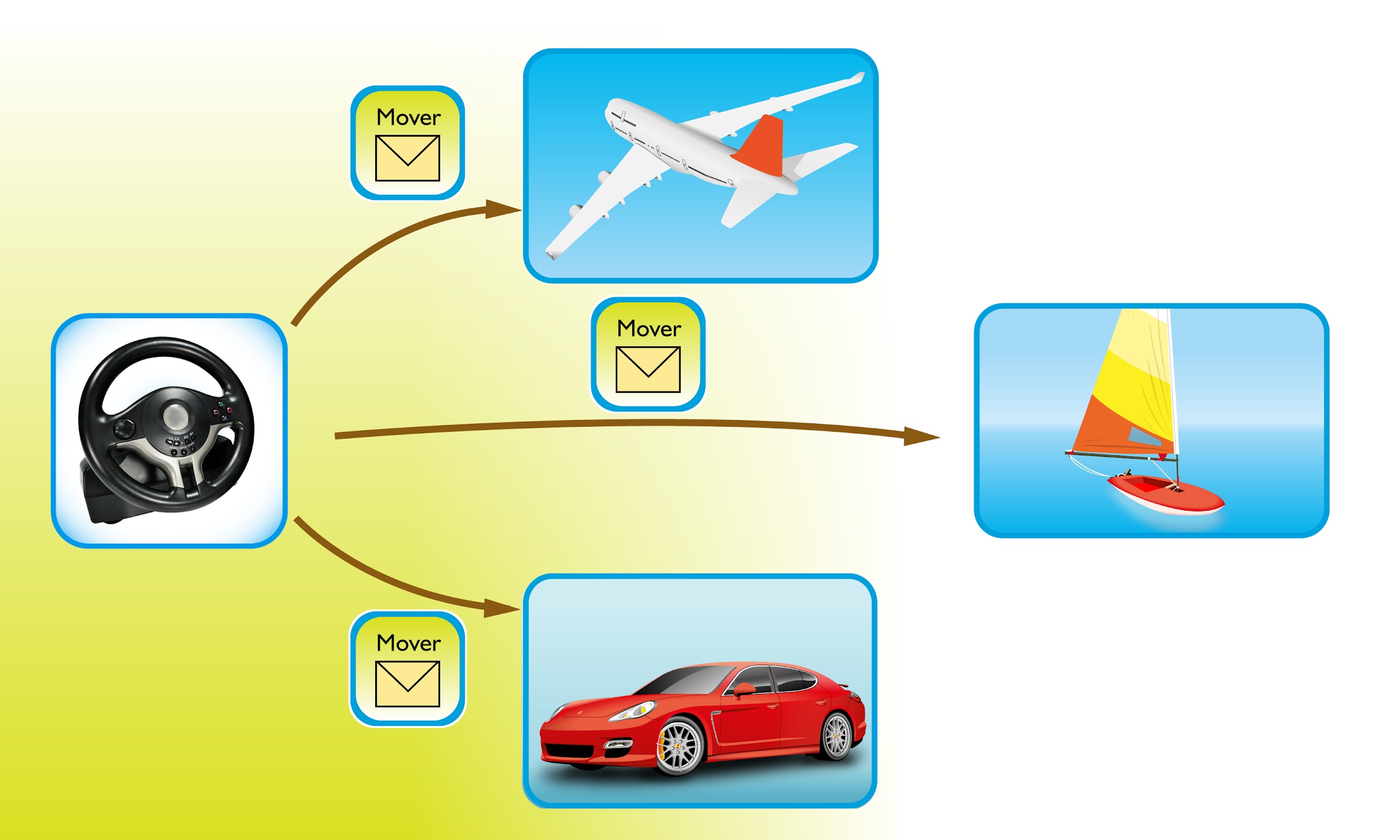
Figura 01 - Pilares da POO



Vejamos o seguinte exemplo: um dono de uma fábrica de brinquedos solicitou que seus engenheiros criassem um mesmo controle remoto para todos os brinquedos de sua fábrica. A única restrição era que cada brinquedo atendesse aos comandos específicos definidos pelo controle.

O controle remoto teria vários botões, sendo que todos eles seriam úteis para todos os brinquedos. Assim, quando o usuário clicasse no botão mover, o controle enviaria o sinal MOVER para todos os brinquedos que estivessem no raio de dois metros. A Figura 2 ilustra tal situação.

Figura 02 - Exemplo de polimorfismo



Assim, quando o brinquedo recebe o sinal MOVER, ele se move de acordo com a sua função. Para o avião, mover significa VOAR, para o barco significa NAVEGAR, e para o automóvel CORRER. Observe que os brinquedos respondem ao mesmo sinal de formas diferentes. Temos aqui então um caso de polimorfismo.

O Polimorfismo permite que diferentes objetos (avião, barco, automóvel) respondam a uma mesma mensagem (mover) de formas diferentes (voar, navegar e correr).

Polimorfismo pode ser classificado de três maneiras:

* Polimorfismo de sobrecarga
* Polimorfismo de sobreposição
* Polimorfismo de inclusão

Polimorfismo de Sobrecarga

Polimorfismo de sobrecarga permite que um método de determinado nome tenha comportamentos distintos, em função de diferentes parâmetros (1) que ele recebe. Cada método difere no número e no tipo de parâmetros.

Exemplo

Considere uma classe Maior capaz de calcular e retornar o maior entre dois números de tipos diferentes. A Listagem 1 ilustra o código de tal classe.

(1) Lembrando: parâmetros são as variáveis que estão dentro dos parênteses na declaração do método.

public class Maior{

private int xInt;

private int yInt;

private float xFloat;

private float yFloat;

private double xDouble;

private double yDouble;

​

public int calcMaior(intx, int y){...}

public float calcMaior(float x, float y){...}

public double calcMaior(double x, double y){...}

public int calcMaior(double a, double b){...} //ERRO!

}

Listagem 1 - Polimorfismo de sobrecarga com tipos de parâmetros diferentes

Veja no exemplo uma aplicação prática do uso do polimorfismo de sobrecarga. Observe que temos vários métodos com o mesmo nome, no caso calcMaior(), cujo objetivo é indicar qual é o maior dentre dois números.

O que diferencia cada um deles é o tipo de parâmetros. No primeiro método, os parâmetros são do tipo int. Já no segundo método, os dois parâmetros são float. Finalmente, no terceiro método, os parâmetros são do tipo double. Esses métodos fazem a mesma coisa (calculam o maior entre dois números), mas de maneiras diferentes, pois recebem e retornam tipos diferentes.

Observe agora o último método calcMaior() declarado na classe Maior. Esse método, apesar de diferenciar do anterior pelo tipo de retorno de double para int, não será aceito pelo compilador Java como um polimorfismo de sobrecarga.

O motivo é que já existe um método com o nome calcMaior() e dois parâmetros do tipo double.

Esse método seria aceito se houvesse um número diferente de parâmetro ou um dos parâmetros tivesse o tipo diferente de double, como mostra a Listagem 2.

public class Maior{

private int xInt;

private int yInt;

private float xFloat;

private float yFloat;

private double xDouble;

private double yDouble;

​

public int calcMaior(int x, int y){...}

public float calcMaior(float x, float y){...}

public double calcMaior(double x, double y){...}

public double calcMaior(double a, double b, double c){...}

public double calcMaior(double x, int y){...}

}

Listagem 2 - Polimorfismo de sobrecarga com tipo e número de parâmetros diferentes

Observe que agora os dois últimos métodos possuem número e tipos de parâmetros diferentes dos métodos anteriores.

Polimorfismo de Sobrecarga

A Sobrecarga e os Construtores

O polimorfismo de sobrecarga normalmente acontece sobre os métodos construtores, pois é comum para uma classe ter várias maneiras de instanciá-la.

A Listagem 3 ilustra um exemplo de polimorfismo de sobrecarga nos construtores da classe Pessoa. Observe que são declarados 3 construtores e todos eles têm o mesmo nome da classe. Entretanto, cada um deles recebe um número diferente de parâmetros.

Assim, durante a criação de objetos do tipo Pessoa em um método main, por exemplo, o construtor que será chamado irá depender da quantidade e parâmetros passados durante a invocação do comando new.

public class Pessoa{

private int rg;

private String nome;

​

public Pessoa(){

..

}

​

public Pessoa(String nome){

this.nome= nome;

}

​

public Pessoa(int rg, String nome){

this.rg = rg;

this.nome = nome;

}

}

Listagem 3 - Polimorfismo de sobrecarga de construtores

A Sobrecarga e a Conversão

Conversão e sobrecarga frequentemente andam lado a lado.

A Conversão é a capacidade de um tipo ser convertido em outro tipo de maneira automática ou pela força bruta também chamada de coerção.

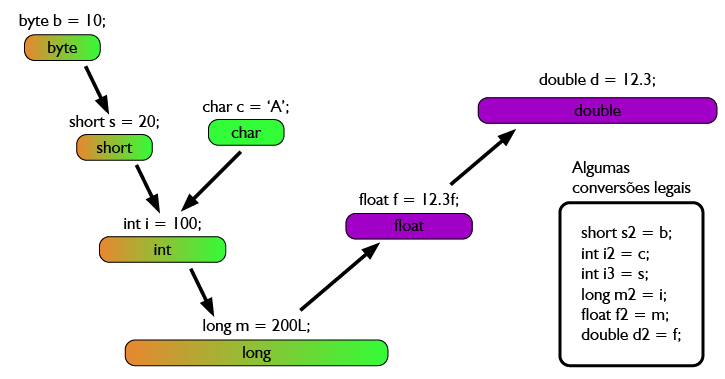
Veja o quadro abaixo sobre conversão de tipos primitivos.

Conversão de Tipos Primitivos

Java converte um tipo de dado em outro sempre que isso for apropriado.

As conversões ocorrem automaticamente quando há garantia de não haver perda de informação. As conversões automáticas são permitidas para tipos de maior precisão e para tipos de menor precisão.

Exemplos de Conversões Automáticas



São permitidas as conversões do tipo byte para o tipo short, de short para int, e assim sucessivamente, até chegar o tipo de maior precisão, que é o double.

Observe também algumas conversões permitidas baseadas nas variáveis criadas pelas letras no quadro “Algumas conversões legais”.

A conversão também pode fazer com que um método pareça como se fosse polimórfico. A conversão ocorre quando um argumento de um tipo é convertido para o tipo esperado, internamente. Por exemplo, suponha que a classe Maior, descrita anteriormente, tivesse apenas atributos e métodos para lidar com o tipo primitivo, conforme ilustra a Listagem 4.

public class Maior{

private float xFloat;

private float yFloat;

​

public float calcMaior(float x, float y){...}

}

Listagem 4 - Classe Maior com o método calcMaior para os tipos int e float

Agora, suponha que em uma aplicação (método main) usamos um objeto da classe Maior, mas, na chamada de seu método calcMaior, fossem passados, ao invés de valores do tipo float, valores do tipo int., conforme ilustra a Listagem 5.

public class Main{

public static void main(String[] args){

Maior maior = new Maior();

int x = 2;

int y = 4;

float z= maior.calcMaior(x,y);

}

}

Listagem 5 - Conversão automática de tipos primitivos

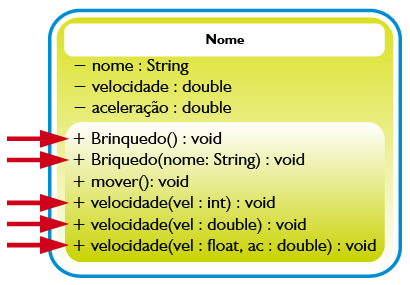
Observe que, apesar do método esperar parâmetros do tipo float, o compilador aceitou receber parâmetros do tipo int, e os converteu automaticamente para float dentro do método calcMaior da classe Maior apresentada na Listagem 5.

Nesse caso, parece um caso de polimorfismo de sobrecarga, mas, na verdade, o que acontece é uma conversão automática de tipos.

Atividade 02

1. Implemente em Java a classe Brinquedo apresentada na Figura 3 a seguir, aplicando o polimorfismo de sobrecarga nos métodos apontados pelas setas. Em seguida, escreva um método main que cria diferentes brinquedos fazendo chamadas para seus diferentes métodos construtores e chamando diferentes métodos velocidade().

Figura 03 - Esquema da classe brinquedo



Polimorfismo de Sobreposição

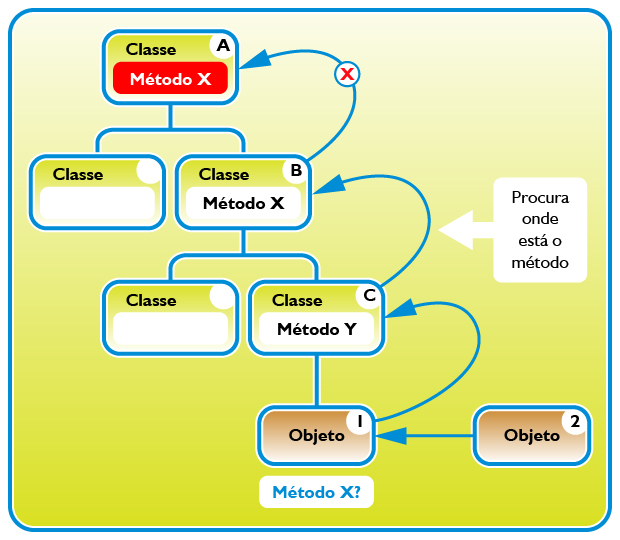
*play\_circle\_filled*

Vídeo 04 - Sobrescrita de Polimorfismo

Polimorfismo de sobreposição é a redefinição de métodos em classes descendentes. Ou seja, um método de uma classe filha com o mesmo nome de um método de uma classe mãe irá sobrepor esse último. Vejamos o exemplo da Figura 4.

Como pode ser observado na hierarquia de classes apresentada, existe: (i) uma classeA, que implementa um metodoX(); (ii) uma classeB, que implementa um método com o mesmo nome; e, finalmente, (iii) uma classeC, que implementa um métodoY().

Figura 04 - Sobreposição de métodos em uma hierarquia de herança



O que aconteceria se fosse solicitado ao Objeto1 da classeC a execução do metodoX()?

Conforme você viu nas aulas anteriores sobre herança, esse método será procurado na hierarquia da classe instanciada pelo Objeto1.

Devemos observar que, nesse caso, o métodoX() que também é implementado na ClasseB, foi encontrado primeiro que o metodoX() da ClasseA.

Nesse caso, o método que de fato será executado será o da ClasseB.

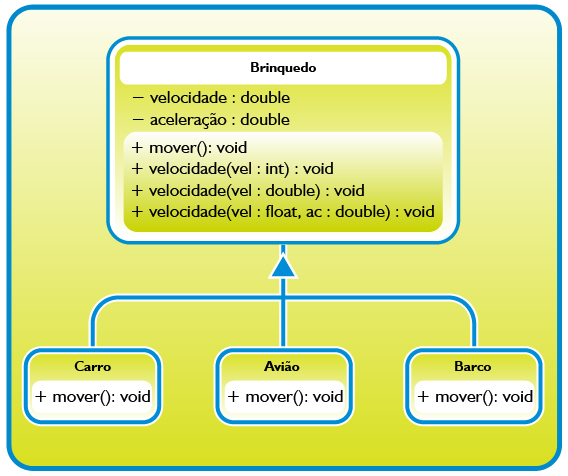
E, nesse caso, o metodoX() da ClasseA jamais será alcançado, a menos que seja criado um objeto da ClasseA. Dizemos então que ocorreu uma SOBREPOSIÇÃO DE MÉTODO ou um POLIMORFISMO DE SOBREPOSIÇÃO. O métodoX da classeB sobrepôs (ou redefiniu) o método de sua classe mãe.

Polimorfismo de Sobreposição

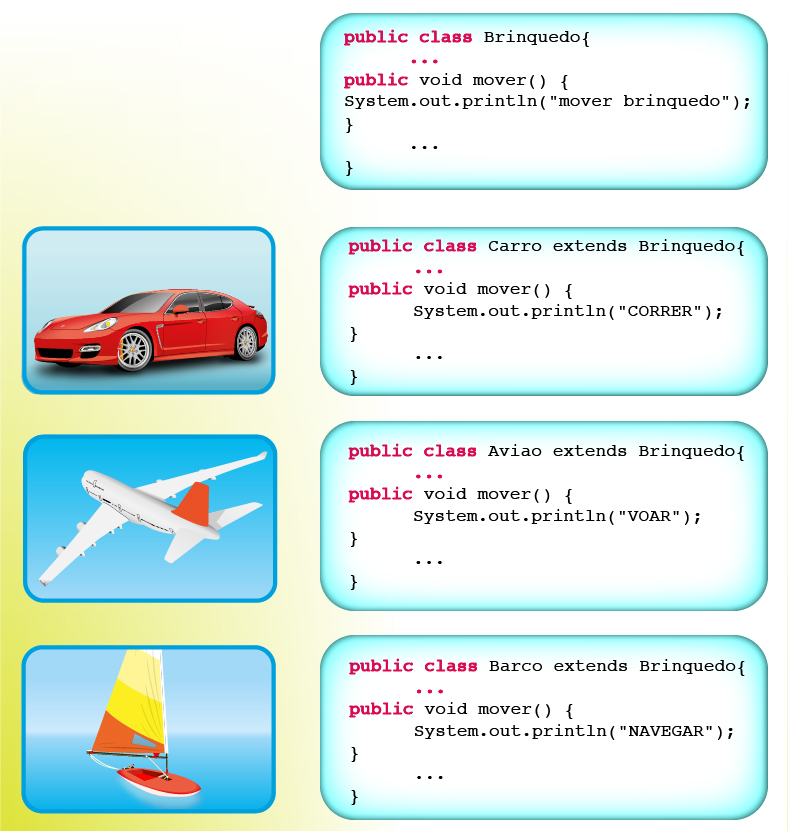
O Polimorfismo de Sobreposição em Java

Considere que a classe Brinquedo (usada no exercício anterior) possui como descendentes as classes Carro, Avião e Barco, conforme ilustra a Figura 5.

Figura 05 - Hierarquia de herança da classe Brinquedo



Observe que as classes filhas sobrepõem o método mover() da classe Brinquedo. Vejamos então como ficam essas classes codificadas em Java na Listagem 6.

 Listagem 6 - Implementação em Java da hierarquia da classe Brinquedo

Considerando o método mover() de cada classe filha, como poderíamos chamar o método mover() do brinquedo (classe) correto? Ou seja, como o ControleRemoto saberá que método mover ele deve chamar se ele tem disponível três tipos de mover diferentes (um para cada brinquedo)?

Vamos ver inicialmente como fica a implementação do Controle Remoto na Listagem 7.

public class ControleRemoto{

private Brinquedo brinquedo;

public ControleRemoto(Brinquedo b){

brinquedo =b;

}

public void mover(){

brinquedo.mover();

}

}

Listagem 7 - Classe ControleRemoto

Você lembra que na primeira vez que apresentamos esse exemplo dissemos que a única restrição para um controle remoto tão versátil seria que “quando criado o controle remoto, ele receberia o tipo de brinquedo que iria acionar em um dado instante”? Pois é, é aí que está o segredo para o controle remoto saber qual deve ser o método mover() que ele deve chamar (CORRER, NAVEGAR ou VOAR). O método construtor da classe ControleRemoto exige que o controle para ser inicializado receba um parâmetro do tipoBrinquedo. E isso acontece quando o atributo brinquedo do ControleRemoto recebe “b” (um objeto do tipo Brinquedo).

Então, quando o método mover() da classe ControleRemoto for acionado (isso significa dizer que o botão mover foi apertado), esse faz com que seu atributo brinquedo chame o método mover()correto, dependendo do tipo de Brinquedo que recebeu quando foi instanciado.

A Listagem 8 mostra como fica a classe Principal que instancia um objeto controleRemoto e um objeto carro para ser controlado automaticamente.

public class Principal{

public static void main(String[] args){

Carro carro = new Carro();

ControleRemoto controleRemoto = new ControleRemoto(carro);

controleRemoto.mover();

}

}

Listagem 8 - Classe Principal com o método main

Observe que criamos um brinquedo do tipo Carro, e quando criamos o ControleRemoto, enviamos esse objeto carro para o objeto controleRemoto através da chamada ao seu construtor. Assim, quando acionado o comando controleRemoto.mover(), será chamado o método mover() do carro.

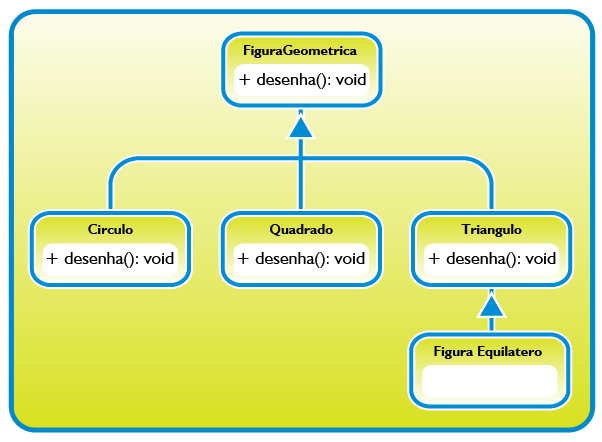
O resultado será a impressão da palavra:

CORRER

Atividade 03

1. Implemente as classes da hierarquia da classe FiguraGeometrica mostrada na Figura 6 abaixo em Java, aplicando o polimorfismo de sobreposição para o método desenha().
2. Em seguida, crie uma classe Principal com um método main que cria um objeto de cada uma das classes e chama seus respectivos métodos desenha().

Figura 06 - Hierarquia da classe FiguraGeométrica

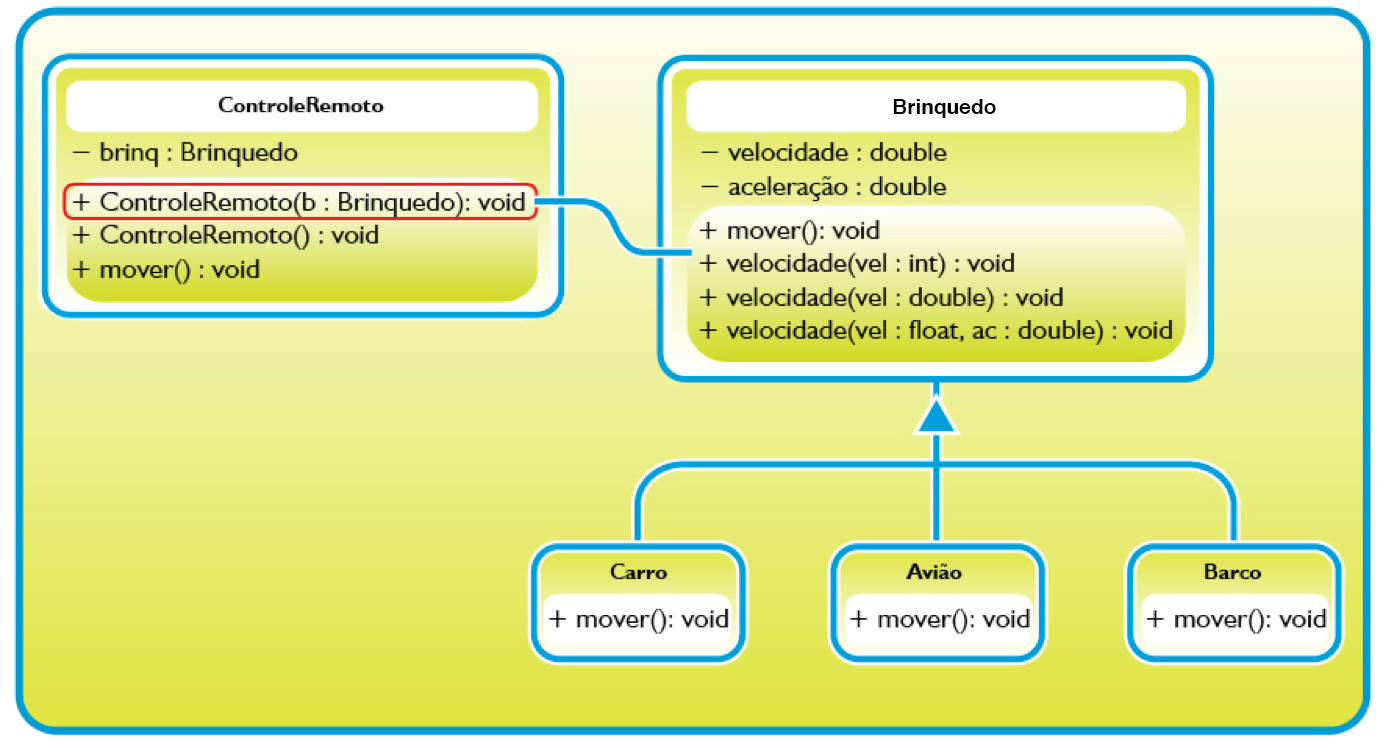


Polimorfismo de inclusão usa a capacidade de substituição da herança*sms\_failed*, de uma classe mãe por qualquer classe descendente, para permitir um comportamento polimórfico nos métodos que usam a classe mãe.

No exemplo visto na seção anterior, veja a classe Principal na Listagem 8, onde criamos um objeto do tipo Carro e outro do tipo ControleRemoto, nós utilizamos o comportamento polimórfico do polimorfismo de inclusão. Fizemos isso quando substituímos a classe Brinquedo (mãe) pela classe Carro (filha) dentro da classe ControleRemoto.

Assim, o atributo interno do tipo Brinquedo da classe ControleRemoto pode receber qualquer objeto que seja de uma classe filha de Brinquedo. Para entender melhor o que aconteceu, veja a Figura 7.

Figura 07 - Hierarquia da classe Brinquedo e da classe Controle Remoto



Observe que a classe ControleRemoto está relacionada com a classe Brinquedo, pois possui um atributo do tipo Brinquedo. Mas, como as classes Carro, Avião e Barco são descendentes de Brinquedo, elas podem substituir a classe Brinquedo em qualquer método que a utilize.

Nesse caso, isso foi feito explicitamente, através da passagem de um objeto da classe Carro para o método construtor de ControleRemoto na Listagem 8.

Caso o programador deseje mudar o controle remoto para interagir com algum outro tipo de brinquedo, bastaria passar um objeto da classe Avião ou Barco na chamada ao construtor da classe ControleRemoto. A capacidade do objeto (brinq) do tipo Brinquedo da classe ControleRemoto de receber qualquer um objeto de subclasses da classe Brinquedo é que caracteriza o polimorfismo de inclusão.

Resumo

Nesta aula, você aprendeu que, com o polimorfismo, objetos de tipos diferentes podem responder à mesma mensagem (solicitação de método com o mesmo nome) de maneiras diferentes. Você estudou quais são e como funcionam os tipos de polimorfismo desde seus conceitos até sua codificação na linguagem Java. Você viu também diferentes exemplos para ilustrar as diferentes situações na qual o polimorfismo pode ser usado.

utoavaliação

1. Sem consultar o material, responda: o que você entendeu por polimorfismo?
2. Quais são os tipos de polimorfismo?
3. Qual é a finalidade de se usar o polimorfismo de sobrecarga? Dê um exemplo.
4. O que é conversão de tipos?
5. Como funciona o polimorfismo de sobreposição? Dê exemplos.
6. Explique como funciona o polimorfismo de inclusão. Dê um exemplo.
7. Implemente o diagrama de classes representado pela Figura 8 abaixo. Para a classe CadastroPessoas considere o atributo pessoas como um array do tipo Pessoa.   
   O método cadastrarPessoa(): deve acrescentar ao arraypessoas um objeto descendente da classe Pessoa.  
      
   O método mostraCadastro(): deve percorrer todo o array de pessoas e mostrar todos os dados do descendente de Pessoa.  
      
   Aplique os tipos de polimorfismo em cada uma das situações solicitadas:  
   1. Polimorfismo de Sobrecarga: crie mais de um método construtor para cada classe: Pessoa, Cliente, Funcionario e Gerente.
   2. Polimorfismo de Sobreposição: faça com que o método mostraCadastro() utilize o método mostraDados() correto, dependendo se a Pessoa é um Cliente, Funcionario ou Gerente.
   3. Polimorfismo de Inclusão: quando for adicionar ao arraypessoas uma nova pessoa que pode ser de um dos tipos descendentes de Pessoa.
8. Figura 08 - Diagrama de classe  
   

Coleções em Java

Apresentação

Em um contexto prático de programação, utilizando conceitos de orientação a objeto, precisamos constantemente manipular muitas informações e muitos objetos ao mesmo tempo. Para isso, é necessária uma estrutura que permita armazená-los e recuperá-los sempre que desejarmos. A linguagem Java oferece várias dessas estruturas de dados em um conjunto de classes chamadas de coleções. Tais coleções ajudam a armazenar e recuperar nossos objetos dentro de um sistema. É isso que você verá nesta aula.

Armazenando Objetos

Vamos organizar os objetos semelhantes!

Existem várias situações durante a construção de sistemas e programas em que precisamos armazenar um número expressivo de objetos. Nesses casos, a criação e manuseio de uma variável para cada um dos objetos torna-se uma tarefa impraticável. Esse fato ocorre, por exemplo, quando precisamos armazenar uma lista de alunos de uma turma, ou mesmo da escola inteira, e em um dado momento precisamos recuperar apenas um dos objetos aluno dessa lista. Outro exemplo seria quando queremos encontrar um número de telefone e recorremos à lista telefônica, ela é nosso repositório de dados.

Imagine termos que criar 100 (cem) variáveis para 100 nomes de alunos! É possível criar um tipo especial de variável para armazenar um número específico de objetos ou valores primitivos. Esse tipo de variável é denominada de array (também conhecida como vetor) e será melhor discutida na sequência deste curso.

Arrays

Considere a seguir um exemplo que motiva o uso de arrays. Constantemente, precisamos tomar nota de um valor que muda ao longo do tempo, e precisamos medir sua média, valores máximo e mínimo. Por exemplo, suponha que você é um treinador de um atleta velocista de 100 metros rasos, como parte do treinamento, você irá fazer 5 medições de tiros de 100 metros.

Para cada tentativa do atleta, você vai medir o tempo e colocar o resultado em sua planilha de rendimento.

Precisamos armazenar 5 valores numéricos reais para representar a marca de tempo obtida. Porém, não podemos criar 5 variáveis, e sim apenas uma para comportar esses valores. Veja tal código na Listagem 1.

public class TreinadorMain{

public static void main(String[] args){

String nomeAtleta = "Bolt";

double[] marca= new double[5];

​

marca[0] = 10.02;

marca[1] = 10.0;

marca[2] = 9.56;

marca[3] = 9.57;

marca[4] = 9.56;

}

}

Listagem 1 - Declaração de um array

Conforme pode ser observado, um novo operador entrou em cena: o par de colchetes “[“ e “]”. Eles servem para indicar uma dimensão, ou seja, uma posição onde iremos colocar um valor inteiro que indica a quantidade de elementos que precisamos armazenar na variável em questão (marca). Por conta de tais colchetes, a variável marca não mais é capaz de armazenar um único valor do tipo double, mas sim irá armazenar um conjunto de valores do tipo double. Dessa forma, caracteriza-se a declaração de uma variável/atributo do tipo array (também chamada de vetor) em Java.

Observe que o atributo marca representa um array, mas é necessário o operador new para alocar espaço para armazenar o conjunto de valores. Assim, a palavra new é usada com o objetivo de indicar quantos valores do tipo Double serão necessários para armazenar tais valores. A quantidade de valores estipulada para armazenar no array, cinco (5) para o exemplo do atributo marca, é um valor que permanece fixo, após a chamada com new. Isso significa que no nosso exemplo o atributo marca será capaz de armazenar 5 elementos.

O código logo após a declaração indica uma atribuição de valor para armazenar uma determinada marca em cada posição do array. Veja que colocamos um número para indicar que posição estamos acessando. Esse valor sempre começa com 0 (zero) e vai até o comprimento do vetor menos uma unidade. No nosso caso, o array marca vai de 0 (zero) a 4 (quatro), compreendendo 5 (cinco) posições conforme foi declarado.

System.out.println("Tempo 1:"" + marca[0]);

System.out.println("Tempo 1:"" + marca[1]);

System.out.println("Tempo 1:"" + marca[2]);

System.out.println("Tempo 1:"" + marca[3]);

System.out.println("Tempo 1:"" + marca[4]);

Listagem 2 - Impressão dos valores do Array

Para exibirmos os valores, o programa da Listagem 1 pode ser incrementado pelas linhas de código da Listagem 2. Não indicamos a primeira tomada de tempo como sendo o tempo 0 (zero), pois naturalmente costumamos realizar contagens a partir do número 1 (um).

Há outra forma de definir os valores de um array, no momento de sua declaração podemos indicar seus valores, sendo que já precisamos conhecê-los de antemão. Veja a Listagem 3. Observe que não precisarmos indicar o comprimento do vetor, pois ele sabe de antemão pela quantidade de elementos definidos na sua inicialização.

double[] marca = {10.02, 10.0, 9.56, 9.57, 9.56};

Listagem 3 - Outra forma de definir os valores de um Array

Tanto na definição do array quanto no acesso para leitura ou escrita de suas posições, é possível usar um comando de repetição para acessá-lo diretamente. Na verdade, na programação diária, isso é o mais comum.

Vejamos como ficaria a exibição dos valores acima com um comando FOR na Listagem 4.

marca[0] = 10.02;

marca[1] = 10.0;

marca[2] = 9.56;

marca[3] = 9.57;

marca[4] = 9.56;

​

for(int i = 0;i<5;i++){

System.out.println ("Marca"+(i+1)+": " +marca[i]);

}

Listagem 4 - Impressão dos valores de um array usando comando FOR

Atividade 01

1. Para praticar o conceito de array, crie um programa que declara uma lista (array) capaz de armazenar 10 (dez) nomes de ferramentas para uso de um mecânico. Inicialize cada uma das ferramentas armazenadas no array, em seguida, imprima cada um deles.

Arrays Bidimensionais

Primeiramente, vamos definir o que é uma dimensão de um array. A dimensão, ou quantidade de dimensões, é o conjunto de valores que precisamos definir para localizar uma informação. Por exemplo, uma lista de alunos de 0 a 100 pode ser organizada em um array de uma dimensão, pois para localizar um aluno nessa lista basta indicar um valor da sequência.

Já para localizar uma peça em um tabuleiro de xadrez precisamos de duas coordenadas, linha e coluna. Assim, para representarmos um tabuleiro de xadrez com arrays, são necessárias duas dimensões. A Listagem 5 mostra esse exemplo codificado em Java. Suponha que iremos representar um tabuleiro de xadrez com linhas e colunas de 0 (zero) a 7 (sete), compreendendo 8 valores de cada. Vamos também supor que cada valor representa uma String com o nome da peça que ocupa a casa. Observe a utilização de dois valores distintos para localizar uma casa no tabuleiro, e para cada valor um par de colchetes, um par para cada dimensão do array.

String[][] tabuleiro = new String[8][8];

​

tabuleiro[0][0] = "Torre branca";

tabuleiro[0][1] = "Cavalo branco";

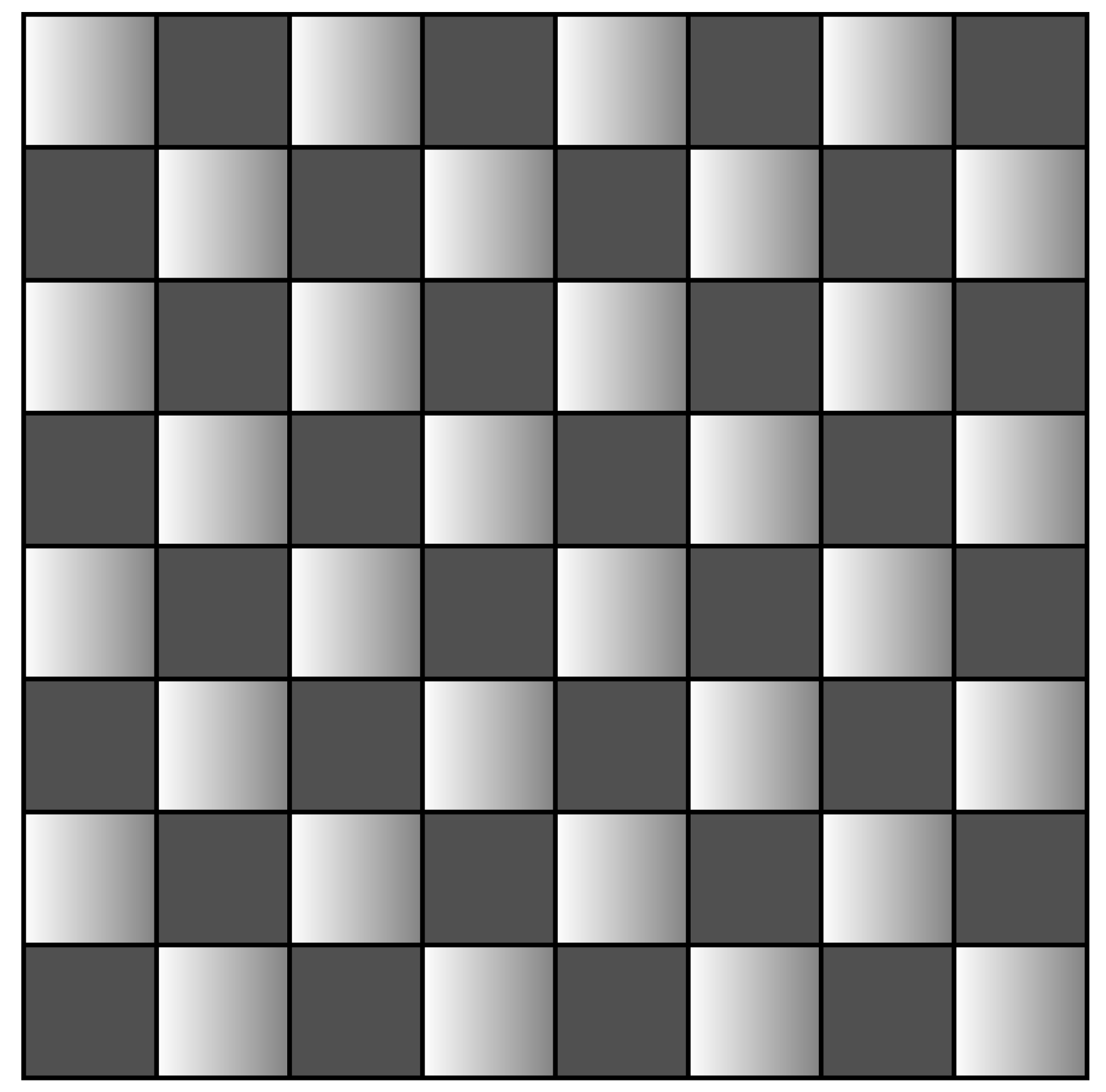
tabuleiro[1][0] = "Peão branco";

tabuleiro[0][3] = "Rainha branca";

Listagem 5 - Exemplo de Array bidimensional

É possível definir n dimensões, porém, na prática, não é comum nem recomendável trabalhar com tantas dimensões. Na prática, é extremamente comum trabalharmos com apenas 1 (uma), e algumas vezes com 2 (duas) e, quase nunca com 3 (três) dimensões, pois há outros recursos na programação orientada a objetos que desencorajam e oferecem alternativas melhores a essa prática.

Atividade 02

1. Observe o tabuleiro da Figura 1 abaixo e crie um programa em Java que o represente através de um array de duas dimensões, em que cada casa irá conter a sua cor BRANCA ou PRETA, como sendo valores do tipo String. Imprima cada uma das casas do array, após iniciá-los.   
   Dica: não tente definir uma a uma, pois serão 64 linhas de códigos. Ao invés disso, crie estruturas com comandos de repetição para preenchê-lo e imprimi-lo.  
      
   Figura 01 - Tabuleiro  
   

Arrays como Objetos

Vimos que arrays são estruturas para armazenar objetos, tais como Strings. Mas, é preciso dizer que arrays são também objetos, por isso é usado o comando new() para alocar espaço de armazenamento para ele. Graças a sua capacidade de se comportar como um objeto, os arrays possuem métodos e um atributo muito útil, o length, que indica seu comprimento.

É necessário conhecê-lo para não passarmos do limite e para realizar operações de manutenção em seus dados. Pois bem, vamos alterar o exemplo anterior (Listagem 1) que exibe os valores do array marca para utilizar seu atributo length. Tal programa modificado é apresentado na Listagem 6.

marca[0] = 10.02;

marca[1] = 10.0;

marca[2] = 9.56;

marca[3] = 9.57;

marca[4] = 9.56;

​

for(int i = 0;i<marca.length;i++){

System.out.println("Marca"+(i+1)+": " +marca[i]);

}

Listagem 6 - Exemplo de uso do atributo length de um array

### **Coleções Java**

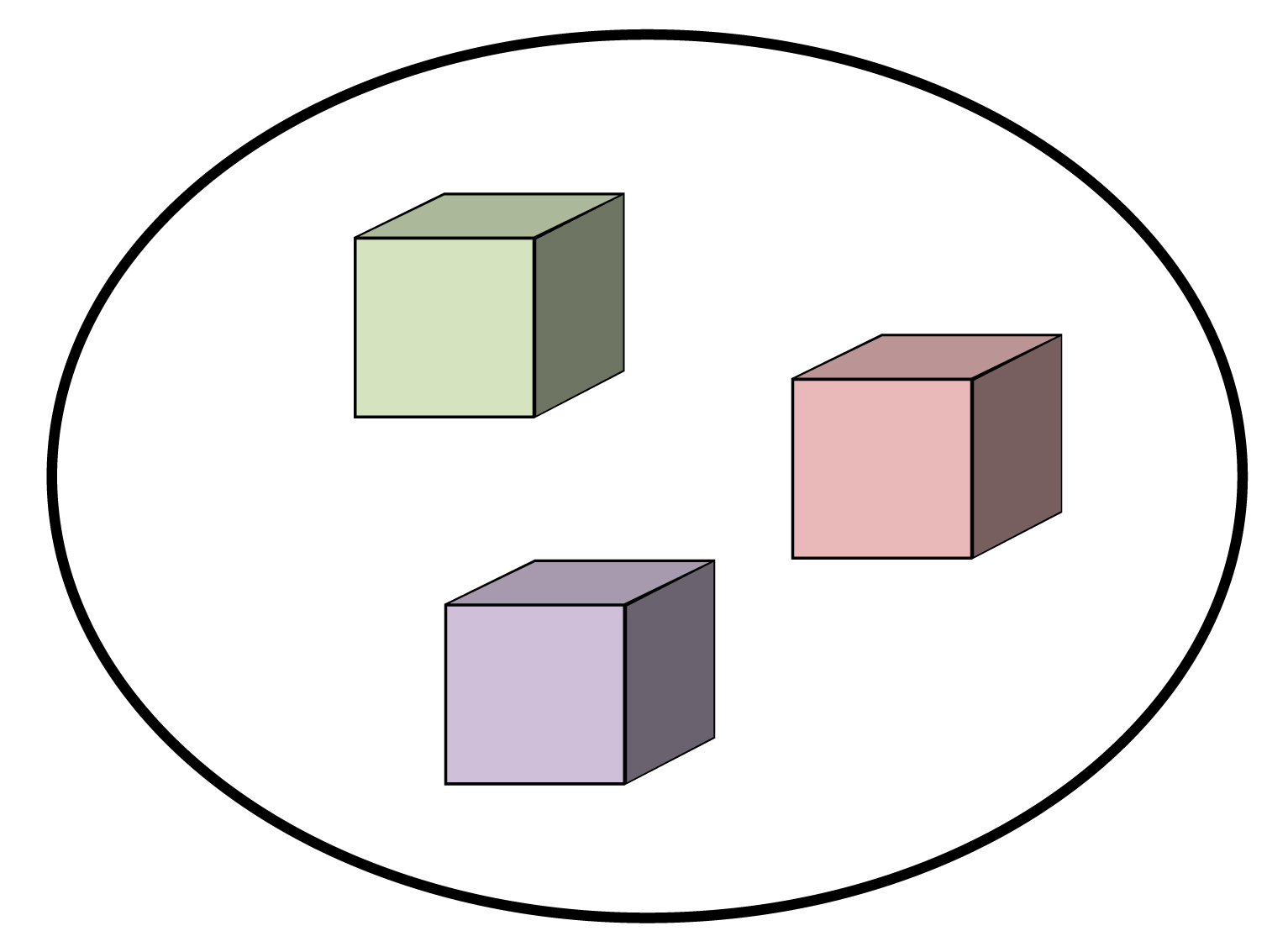
A linguagem Java possui um conjunto de classes que servem para armazenar na memória vários objetos. Tais classes não possuem o inconveniente de termos que saber de antemão a quantidade exata de elementos que iremos armazenar, como no caso de arrays. E em alguns casos, nem mesmo o tipo.

Existem tipos principais de Coleções em Java, cada uma com um propósito deferente, como veremos a seguir.

* List: Lista de coisas.
* Set: Conjunto de Coisas (Não permite repetição de elementos).
* Map: Coisas com um identificador único.

1. Set – representa a mesma ideia de conjuntos da matemática, ou seja, um grupo de objetos sem ordem definida, porém, únicos. Como mostra a Figura 2.

Figura 02 - Conjunto de objetos



Nunca poderemos prever a ordem com a qual serão apresentados os seus elementos. Essa situação pode não ser um incômodo diante do problema que tivermos. Os sets possuem uma característica importante em relação às buscas de objetos em seu interior, pois não precisam percorrer todos eles.

O principal representante dos Set é a classe HashSet, vejamos um exemplo da sua utilização, como mostra a Listagem 7:

public class TesteHashSet{

public static void main(String[] args){

HashSet itens = new HashSet();

itens.add("Chocolate");

itens.add("Bala");

itens.add("Brigadeiro");

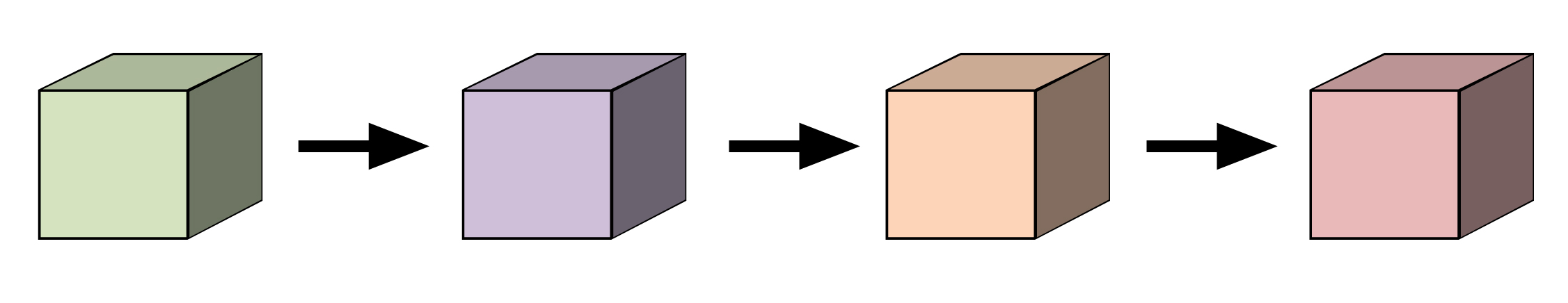
}

}

Listagem 7 - Exemplo de um Set

2. List – como o próprio nome sugere, representa uma lista de objetos, sendo que nela os objetos podem se repetir. Veja a figura.

Figura 03 - Lista de objetos



Nas listas (List) definidas em Java, os objetos armazenados mantêm a ordem com que foram adicionados. Uma classe do tipo List bastante utilizada é a ArrayList, a qual representa uma alternativa aos arrays convencionais vistos anteriormente. Vejamos um exemplo da utilização de um ArrayList na Listagem 8:

ArrayList nomes = new ArrayList();

nomes.add("Maria");

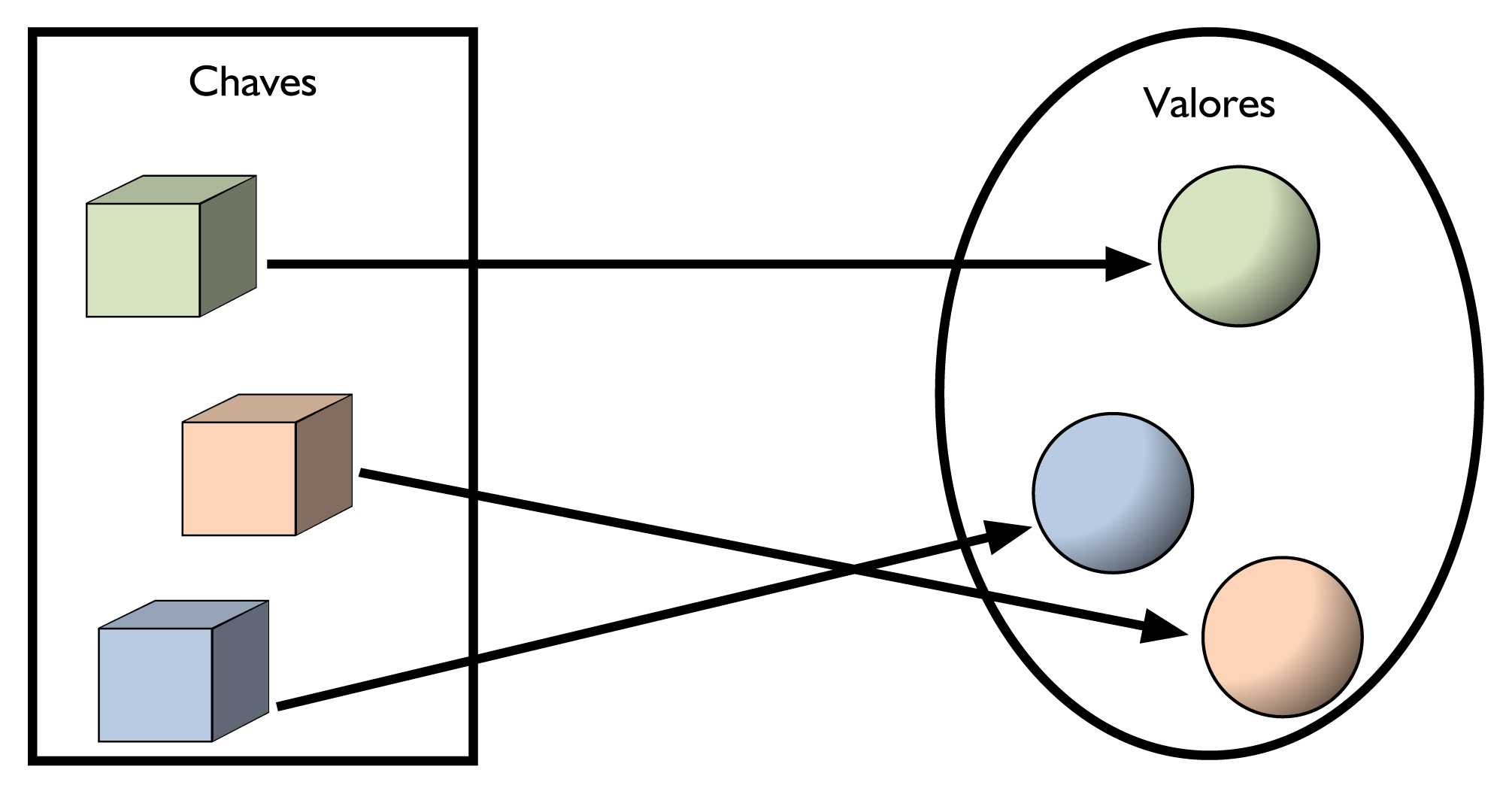
nomes.add("João");

Listagem 8 - Exemplo de ArrayList

3. Map – Mapas são estruturas que relacionam um objeto a outro, por exemplo, um número de CEP ao nome de uma rua.

Podemos imaginar dois conjuntos, um de campos-chave e outro de objetos-valor que queremos armazenar. Observe a Figura 4.

Figura 04 - Dois conjuntos de objetos



Observe que para encontrarmos nossos objetos precisamos localizá-los através de suas chaves. Vejamos uma aplicação desses conceitos na Listagem 9.

public class TesteHashSet{

public static void main(String[] args){

HashMap livros = new HashMap();

​

livros.put(1, "Volta ao mundo em 80 dias");

livros.put(2, "Alice no país das maravilhas");

livros.put(5, "Caninos Brancos ");

}

}

Listagem 9 - HashMap na prática

Todas as classes do tipo Set e List descendem (implementam) a interface Collection. Veremos mais detalhes sobre interfaces em aulas futuras. Por hora, podemos entender interfaces como contratos que definem um conjunto de métodos que devem ser implementados pelas classes. No caso da interface Collection, ela define métodos para adicionar, remover, verificar a presença de um dado objeto. Tais métodos devem necessariamente existir em todas as List e Sets definidos para a linguagem Java. Segue abaixo uma lista básica desses métodos.

Lista Básica de Métodos da Interface Collection

#### **Anote a Dica!**

As coleções Java mantêm um mecanismo interno de ordenação e recuperação de dados e para isso fazem uso de tabelas hash. Tais tabelas são utilizadas para que a pesquisa de um objeto seja feita de maneira rápida. Mas, como funciona? Cada objeto é “classificado” pelo seu hashCode, método de java.lang.Object que retorna um int, e com isso podemos agrupar os objetos por esse valor. Quando é realizada uma busca, só é percorrido o grupo de objetos com o mesmo hashCode.

|  |  |
| --- | --- |
| oolean add(Object) | Adiciona um elemento na coleção. Como algumas coleções não suportam elementos duplicados (exemplo: Sets), esses métodos retornam verdadeiro (true) ou falso (false) para indicar se a adição foi bem sucedida. |
| boolean remove(Object) | Remove determinado elemento da coleção. Se ele não fizer parte da coleção, retorna falso (false). |
| int size() | Retorna a quantidade de elementos presentes na coleção. |
| boolean contains(Object) | Procura por um determinado objeto na coleção.  Vale salientar: a comparação é feita pelo método equals(). |

#### **Anote a Dica!**

As coleções Java mantêm um mecanismo interno de ordenação e recuperação de dados e para isso fazem uso de tabelas hash. Tais tabelas são utilizadas para que a pesquisa de um objeto seja feita de maneira rápida. Mas, como funciona? Cada objeto é “classificado” pelo seu hashCode, método de java.lang.Object que retorna um int, e com isso podemos agrupar os objetos por esse valor. Quando é realizada uma busca, só é percorrido o grupo de objetos com o mesmo hashCode.

Exemplo e comparações entre Coleções e Arrays

import java.util.\*;

public class TreinadorMain{

public static void main(String[] args){

String nomeAtleta = "Bolt";

double[] marca = new double[5];

marca[0] = 10.02;

marca[1] = 10.0;

marca[2] = 9.56;

marca[3] = 9.57;

marca[4] = 9.56;

for(int i=0; i<marca.length; i++){

System.out.print(marca[i]+ ", ");

}

System.out.println();

String[][] tabuleiro = new String[8][8];

tabuleiro[0][0] = "Torre branca";

tabuleiro[0][1] = "Cavalo branco";

tabuleiro[1][0] = "Peão branco";

tabuleiro[0][3] = "Rainha branca";

for(int i = 0; i<2;i++){

for(int j = 0; j<4; j++){

System.out.println(tabuleiro[i][j]);

}

}

HashSet itens = new HashSet();

itens.add("Chocolate");

itens.add("Bala");

itens.add("Brigadeiro");

ArrayList nomes = new ArrayList();

nomes.add("Maria");

nomes.add("João");

HashMap livros = new HashMap();

livros.put(1, "Volta ao mundo em 80 dias");

livros.put(2, "Alice no país das maravilhas");

livros.put(5, "Caninos Brancos ");

System.out.println(itens);

System.out.println(nomes);

System.out.println(livros);

}

}

### **erando sobre Coleções**

Agora que vimos algumas das coleções mais utilizadas no mundo Java, vamos então aprender como percorrer essas coleções. Para tal tarefa, existem várias opções, no entanto, nesta seção iremos falar especificamente do for each. Antes das apresentações oficiais, vamos dar uma revisada no velho e bom for tradicional, apenas para relembrar.

#### **Loop for each**

Também conhecido no mundo Java como “enhancedloop” e for-in, porém, todos se referem ao mesmo construtor Java. O for-each loop veio com o Java 6 como um loop especializado que simplifica a iteração sobre Arrays e Coleções.

Bom, já que o nosso novo loop veio para facilitar a nossa vida, quando iteramos sobre as coleções, então, vamos dar uma olhada mais de perto no seu funcionamento.

O que ele tem de bom?

Primeiramente, ao invés de ser dividido em TRÊS partes, temos apenas DUAS. Opa! Isso já está ficando bom! Diminuiu a complexidade da sintaxe. Então, vamos iterar sobre um Array utilizando o for básico(antigo) e então depois veremos como fica com o nosso novo amigo.

int [ ]a = {1,2,3,4}

for (int x = 0 ; x <a.length ; x++)

System.out.print(a[x]);

for (int n : a)

System.out.print(n);

Noossaaa! Realmente ficou muito mais simples!! Mas não entendi nada! Calma, veremos todo o seu comportamento a seguir.

Analisando o enchanced for de uma maneira mais formal:

for ( declaração : expressão ) {}

As duas partes são:

* Declaração: no bloco de declaração o tipo da variável pode ser compatível com o tipo dos elementos da coleção que você está acessando. Essa variável será a que utilizaremos no corpo do loop e o seu valor será o mesmo que o elemento corrente da coleção.
* Expressão: deve ser o Array ou coleção que você deseja iterar sobre. Pode ser qualquer tipo de coleção ou método cujo retorno seja uma coleção ou Array. Os tipos dos elementos dentro da coleção podem ser os seguintes: primitivos, objetos ou até mesmo outra coleção.

Vamos dar uma olhada em algumas restrições do enchanced loop:

long x2;

Long [] la = {7L, 8L, 9L};

Animal [] animals = { new Dog(), new Cat()}

​

for (Long x2 : la); // x2 já está declarado

for ( Dog d : animals); // você pode receber um Cat

Outro exemplo comparado do FOR E FOREACH

public static void main(String[] args){

String nomeAtleta = "Bolt";

double[] marca = new double[5];

marca[0] = 10.02;

marca[1] = 10.0;

marca[2] = 9.56;

marca[3] = 9.57;

marca[4] = 9.56;

for(int i=0; i<marca.length; i++){

System.out.print(marca[i]+ ", ");

}

for (Double cont : marca)

System.out.print(cont);

}

### **Generics e Coleções**

Chegamos ao momento mais interessante do nosso conteúdo, vamos aprender agora como a utilização de Generics irá facilitar a nossa vida na manipulação de coleções em Java, fazendo com que possamos criar códigos mais robustos e legíveis.

Sabemos que Arrays em Java sempre foi type safe, Opa! O que danado é isso? Calma, type-safe é apenas uma expressão muito utilizada no mundo Java para dizer que um determinado elemento é fortemente tipado. Voltando ao nosso Array, significa que se o declararmos como sendo do tipo String (String []), não poderemos fazer atribuições de outros elementos, tais como Integer, Dog, ou qualquer outra coisa diferente de String.

Devemos lembrar que antes do Java 5 não havia sintaxe para declarar uma coleção como type-safe. Para fazer um ArrayList de String, tínhamos que fazer a seguinte declaração:

ArrayList myList = new ArrayList();

Ou utilizarmos a forma polimórfica:

List myList = new ArrayList();

Não havia sintaxe que nos permitisse especificar que myList só deveria aceitar String e nada mais além de String. Sendo assim, como não tinha essa restrição nativa da linguagem, o compilador também não tinha como forçá-lo a colocar apenas os elementos do tipo especificado dentro da lista. Vamos dar uma olhada na prática para entender melhor a respeito do problema que tínhamos antes do Java 5.

List myList = new ArrayList(); // não podemos informar o tipo.

myList.add(“João”); // neste ponto está OK, adicionamos uma String.

myList.add(new Dog()); // Ops! Agora colocamos um Dog.

myList.add(new Integer()); // …E agora adicionamos um Integer

Note que tudo que fizemos acima foi totalmente legal, nenhum erro de compilação ou alerta de tipos indevidos, ou seja, uma coleção não genérica poderá conter todos os tipos de elementos, mas qual é o problema disso? O que isso significa?

Então, vamos lá... Isso significa que é de TOTAL responsabilidade do programador tomar todos os cuidados ao manipular a Lista, ou seja, não existe restrição nativa da sintaxe que force as coleções a só receberem elementos de um determinado tipo pré-estabelecido.

E você acha que acabou os seus problemas? Ainda não... tem mais, pois as coleções podem receber qualquer tipo de coisa, o método que recupera objetos da coleção só poderá ser de um único tipo! Alguém se arriscaria a dizer? Essa é fácil... Só poderia ser mesmo o famoso – java.lang.Object. E o que isso significa na prática? Então, vamos voltar ao nosso exemplo acima e recuperar o nosso elemento do tipo String que havíamos adicionado inicialmente:

String s = (String) myList.get(0); // fazendo cast.

Opa! Tenho que fazer um Cast para recuperar a minha String! É isso mesmo! Porém, ainda tem mais um pequeno probleminha! Não temos como garantirque o retorno será realmente uma String, sendo assim, o nosso Cast ainda pode falhar em tempo de execução! Tudo isso porque as coleções aceitam qualquer coisa como entrada e não temos como garantir o que vai ser retornado no final, apenas rezamos para que aquele programador extremamente experiente (Estagiário) tenha se preocupado em colocar todos os elementos na coleção como sendo String, e tudo irá funcionar corretamente!

Poxa, agora fiquei triste de verdade, eu estava tão acostumado com o compilador me dizendo tudo o que eu fiz de errado, e agora você me vem com essa de que ele não vai poder me ajudar nas coleções!

Calma, isso ficou para trás, você não prestou atenção quando eu disse que todo esse problema era antes do Java 5, agora temos uma ferramenta muito forte ao nosso favor, e que vai nos ajudar não apenas com as coleções, mas em quase tudo em Java podemos nos beneficiar do uso do Generics.

Ufa! Agora fiquei mais tranqüilo!

Então, vamos lá, chegou a hora das boas notícias! O Generics nos ajudará em dois pontos, na hora de colocarmos e na hora de retirarmos elementos das coleções, através da aplicação do tipo específico. Vamos dar uma olhada na prática de como isso acontece de verdade! Lembra do nosso exemplo anterior?

List<String> myList = new ArrayList<String>(); // Agora informamos o tipo

myList.add(“João”); // neste ponto está OK, adicionamos uma String.

myList.add(new Dog()); // Ops! Agora recebemos um erro de compilação!!!

Muito Bom!! Era exatamente o que queríamos! Usar a sintaxe Generics significa que temos que colocar o tipo da coleção dentro dos elementos “<” e “>”, como <String>. Então, agora tudo que você coloca dentro da coleção é garantido de ser uma String. E as boas notícias não param de chegar, com o Generics tudo o que recuperamos também é garantido de ser uma String. Vamos dar uma olhada mais de perto!

Antes do Generics, não tínhamos garantia do tipo de retorno:

String s = (String) myList.get(0); // fazendo cast.

Agora nós apenas fazemos:

String s = myList.get(0); // Sem Cast.

O compilador já sabe antecipadamente que myList contém apenas objetos do tipo String, os quais podem ser atribuídos a uma referência String, então, agora não é mais necessário fazer o Cast. Agora, a nossa vida ficou realmente mais simples e podemos ir mais além, vamos dar uma olhada em como isso funcionaria com o novo for-each que aprendemos agora pouco.

for ( String s = myList ) {

int x : s.length;

// note que não é necessário utilizar um Cast antes de chamar o método da string “s”! O compilador já sabe que “s” é uma String recuperada da coleção myList.

}

### **Atividade 04**

1. Crie um programa que manipule um Array de objetos do tipo Animal, adicione elementos a esse Array e depois recupere-os.
2. Crie um programa que manipule uma coleção de objetos do tipo Animal sem a utilização de Generics, adicione e recupere os elementos da coleção.
3. Evolua o exercício 2 para que a coleção utilize Generics. Adicione e recupere os elementos.

### **Resumo**

Você estudou hoje que em sistemas grandes é bastante comum precisarmos de um grande conjunto de objetos. Você viu que a linguagem Java oferece dois mecanismos para armazenar um grande número de objetos: arrays e coleções. Agora, você já sabe que os arrays permitem armazenar um conjunto de objetos com tamanho fixo definido na sua inicialização. Já as coleções podem ser usadas para armazenar um conjunto não-finito de elementos, e são implementadas em Java como um conjunto de interfaces, que oferecem um conjunto de métodos padronizados para armazenamento e recuperação. Vimos também o uso do novo for-each, que facilita a manipulação das coleções, e aprendemos como a utilização do Generics nos ajuda e facilita, evitando a utilização de Cast e garantindo os tipos dentro das coleções.

### **Autoavaliação**

1. Elabore um programa que exiba um calendário do dia 1 ao dia 31, utilizando um array de duas dimensões.
2. Qual a principal diferença entre um array convencional e a classe ArrayList que você destacaria?
3. Que tipo de coleção se assemelha a um conjunto matemático? Por quê? Explique.
4. Qual o principal motivação para a utilização de Generics?

## **Tratamento de Exceções**

### **Um Pouco de Conversa!**

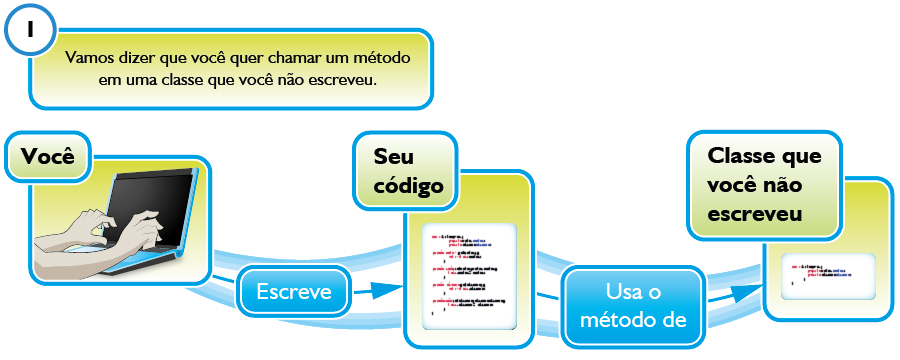
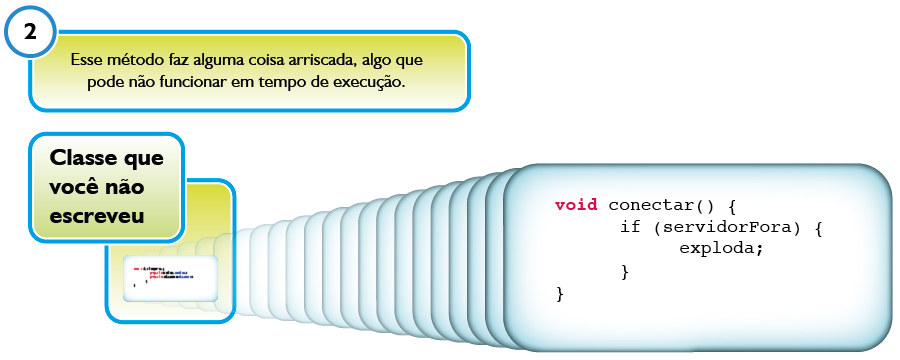
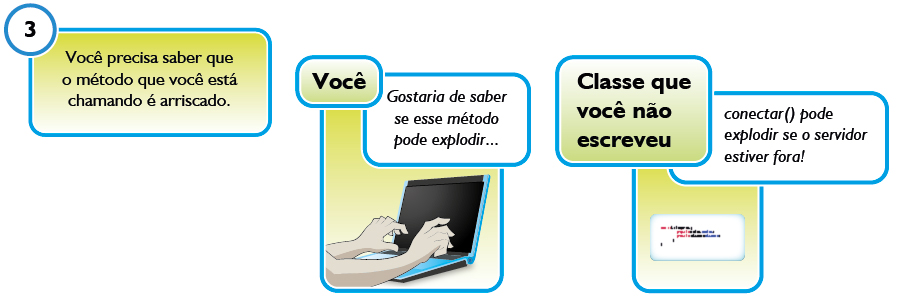
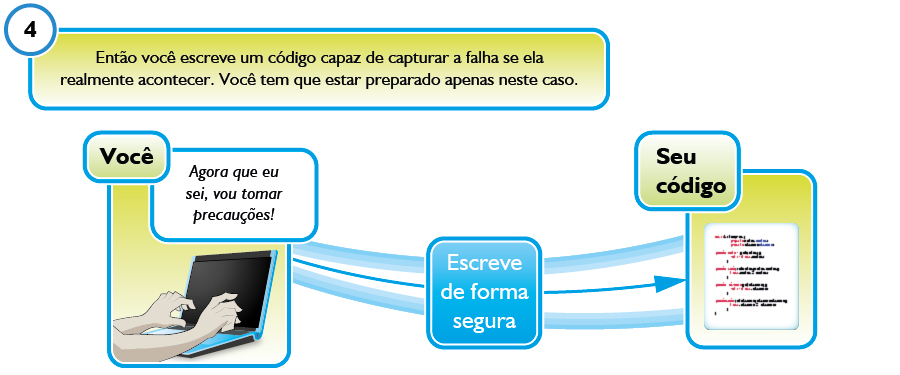
Uma regra antiga no desenvolvimento de software diz que 80% do tempo gasto é referente ao esforço requerido para checar e capturar erros, o restante é relativo ao desenvolvimento. A detecção e a captura de erros podem ser o ingrediente mais importante para construção de uma aplicação robusta.

Felizmente, JAVA nos mune de excelentes equipamentos para capturar e organizar códigos para tratamento dos erros, mecanismo conhecido como Exception handling, que nos permite detectar erros facilmente sem ter que escrever códigos especiais para verificar os tipos de retornos. E ainda melhor, nos permite criar códigos que tratam as exceções separadas dos códigos que as geram. E a cereja do bolo ainda estar por vir... ele também nos permite capturar vários tipos de exceções e tratá-los em um único ponto.

#### **Encarando o Problema de Frente!**

Vamos dar uma olhada na seguinte situação e entender melhor por que capturar exceções é tão importante.

O que acontece quando um método que você deseja chamar (provavelmente de uma classe que você não escreveu) pode causar riscos?

Figura 01    

Agora sim, entendemos que coisas ruins podem acontecer caso não nos preocupemos em tratá-las. Como vimos na ilustração acima, não podemos controlar, no caso, o programador estava chamando um código que não foi escrito por ele mesmo, e que poderia provocar grandes problemas se não tratado da forma correta, mas que para isso temos que saber previamente que um determinado código pode provocar tal exceção. É sobre isso e muito mais que iremos falar de agora em diante, mas antes vamos dar uma olhada na hierarquia de exceções.

### **Hierarquia de Exceções**

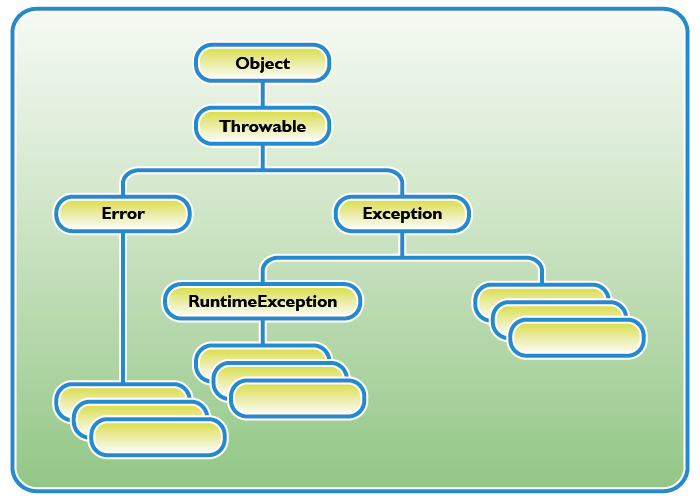
Primeiramente, o que é uma exceção em Java? De onde ela vem? Opa... Uma pergunta de cada vez!

Inicialmente, vamos dar uma olhada na terminologia “exception”, que significa “condição excepcional” a qual é uma ocorrência que altera o fluxo normal de execução do programa. Um monte de coisas pode causar exceções, incluindo falha de hardware, exaustão de recursos, ou até mesmo velhos Bugs.

Agora, respondendo a primeira pergunta: um “exception” em Java é um Objeto do tipo Exception. Sendo assim, como a exceção é um objeto, o que você receberá quando ocorrer uma condição de falha será um Objeto. Veremos mais adiante como é a sintaxe para capturar uma exceção em Java!

Não fiquem ansiosos... Vamos agora responder a segunda pergunta (de onde ela vem?), analisando a sua árvore “genealógica”, temos o seguinte figura:

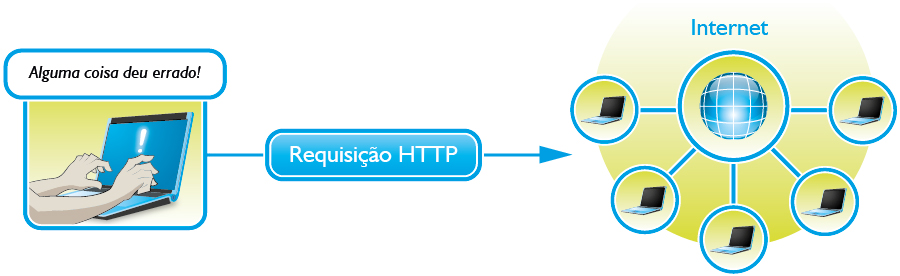
Figura 02 - Hierarquia das classes de exceções



Como você pode ver, existem duas classes que derivam diretamente de Throwable: Exception e Error. Classes que derivam de Error representam situações não usuais as quais não são causadas por erro de programa, e indicam coisas que não acontecem normalmente durante a execução do programa, tal como o famoso Out of memory da JVM. Geralmente, a nossa aplicação não será capaz de se recuperar de um erro proveniente de Error, sendo assim, não vamos esquentar a cabeça com isso!

Vamos então ao que interessa. Em geral, uma exceção representa algo que acontece NÃO como resultado de um erro de programação, mas sim devido a algum recurso que não está disponível ou quando alguma outra condição requerida para a execução correta do programa não está presente no momento da execução. Por exemplo, se a sua aplicação está querendo se comunicar com outra aplicação ou computador que não está respondendo, temos então um caso em que a exceção não foi causada por um Bug, nesse caso, será lançada uma RuntimeException, como visto na figura Hierarquia das Classes de Exceções.

Figura 03



Muito bom, aprendi o que é uma Exception em Java, sei a diferença entre um Erro e uma Exceção, sei que ela é um Objeto normal e até que herda de Throwable! Ótimo, tudo! Mas, não sei nada da sintaxe, como fico sabendo que um método pode causar uma exceção? Como eu trato uma exceção? Calma, calma... Agora, é que vai começar a parte boa da história, vamos falar sobre a SINTAXE das exceções! Então, vamos lá...

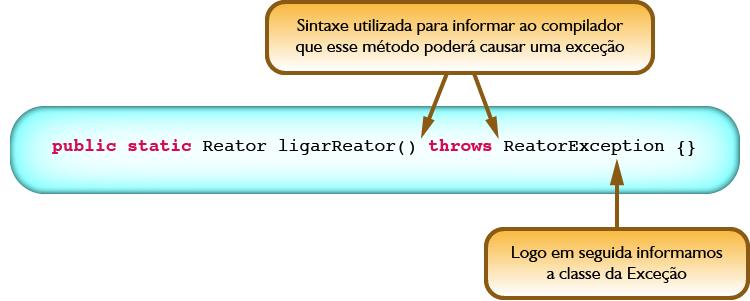
### **Exceptions (Sintaxe)**

Primeiramente, um pouco de marketing sobre o mecanismo de tratamento de exceções em Java. Devo falar que ele é totalmente “Clean”, ou seja, é feito para que seja totalmente transparente e intuitivo para o programador capturar e tratar as condições adversas.

É baseado de maneira que você saiba quando está chamando um método que possa levantar uma exceção, assim você pode escrever códigos para lidar com essas possibilidades. Se você sabe que um método poderá causar algum problema antecipadamente, podemos então nos precaver e estar preparado para o pior! E se o pior vir a acontecer, podemos contornar a situação, já que era sabido que algo de ruim poderia acontecer!

Vamos começar pelo mais básico, ou seja, como finalmente sabemos que um método o qual desejamos chamar pode causar algum tipo de problema? Em Java, os métodos utilizam a cláusula throws na declaração do método!

Figura 04



O método ligarReator() acima pode causar uma exceção em tempo de execução, sendo assim é nossa obrigação informar que tipo de exceção pode ocorrer para que os usuários dessa função possam então se precaver.

Para os mais apressadinhos e curiosos, notaram algo de especial no método acima? De onde vem esse tipo de exceção ReatorException? Parece-me algo criado pelo programador especialmente para tratar de um tipo específico de risco! E você está certo, o mecanismo para tratamento de exceções do Java é bastante completo e nos permite criar nossas próprias exceções, mas vamos devagar com a missa! Esse assunto será tratado mais tarde, foi apenas uma pequena pausa para deixá-los com água na boca!

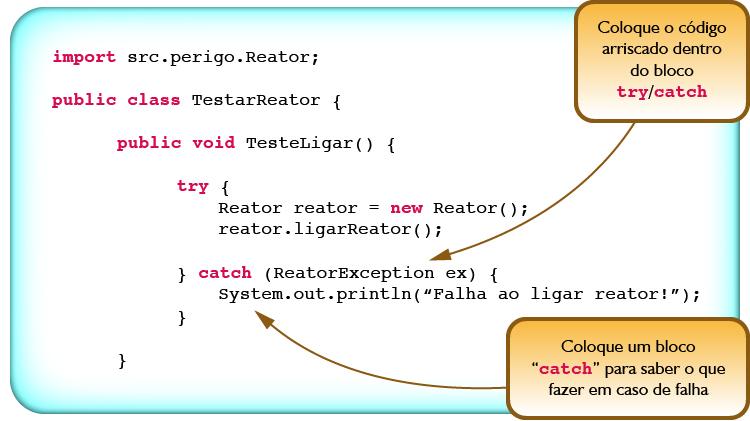
Muito bom mesmo! Agora sei que se eu chamar um método e ele tiver uma cláusula denominada throws na sua declaração, significa que eu tenho que me precaver caso o pior aconteça! Massa! Mas, como eu faço isso?

Figura 05



Calma! Para isso que existe o try/catch! Vamos aprender agora como tratar a chamada a um método que possa causar uma exceção! Vejamos o código a seguir!

Figura 06



E para que tudo isso? O compilador tem que saber que você está chamando um método de risco!

Ou seja, o compilador não irá descansar enquanto você não colocar o seu código arriscado em um bloco try/catch. Essa é a forma de dizer ao compilador que você sabe que alguma coisa ruim poderá acontecer no método que você está chamando e que você está preparando para o que der e vier!

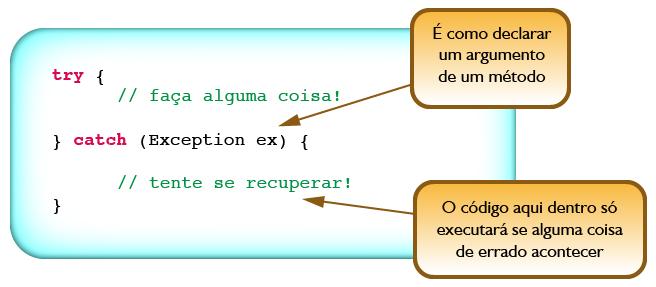
Lembre-se: o compilador não quer saber como você irá tratar o problema, ele se importa apenas em saber se você está tomando as devidas precauções.

#### **Anote a Dica**

Uma exceção é um objeto... do tipo Exception

Lembra da nossa árvore “genealógica”, mostrada anteriormente? Pois então, toda exceção que criarmos é filha de Exception, que por sua vez herda de Throwable, sendo assim, como uma exceção é um objeto, tudo que capturamos na cláusula ‘catch’ é também um objeto, representado pelo parâmetro ‘ex’, conforme o exemplo a seguir.

Figura 07

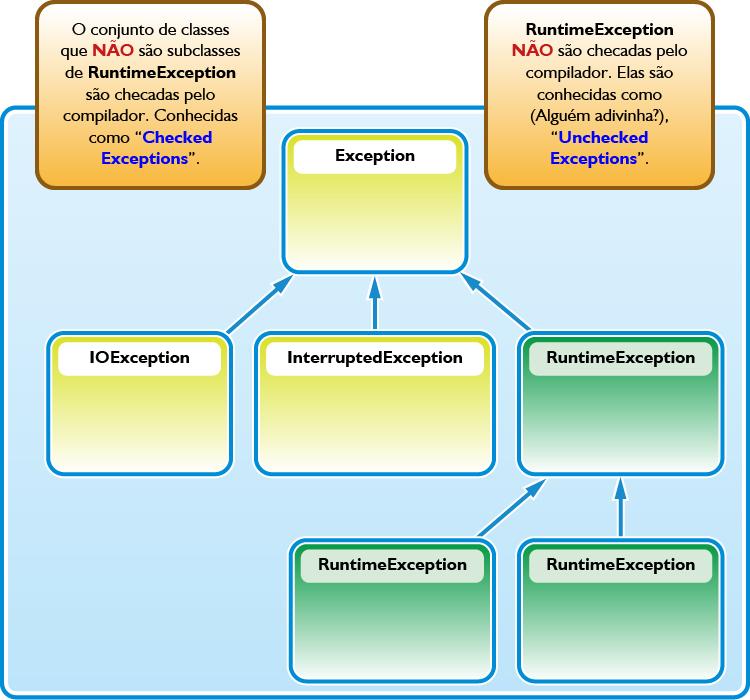


Agora que aprendemos um pouco mais sobre o mundo das exceções em Java, vamos por em prática os nossos conhecimentos.

### **Tipos Comuns de Exceções**

Eu sei que já vimos a hierarquia de exceções no início da aula, mas vamos dar uma olhada mais detalhada nessa hierarquia para entender melhor os tipos mais comuns de exceções.

Figura 08



RuntimeException? Como assim, eu sou diferente das demais? Mas eu não sou, ao final de contas, filha de Exception também!

### **Visão do Compilador**

Ele se preocupa com todas as exceções que sejam subclasses de Exception, a menos que elas sejam de um tipo especial (RuntimeException). Toda classe de exceção que herde de RuntimeException tem passe livre, ou seja, para o compilador tanto faz se você declará-la na cláusula throws ou não, também não faz diferença se você a trata dentro de um bloco try/catch, para ele é como se as RuntimeException sequer existissem.

Figura 09



Espera aí! Mas por que tanto desprezo com as RuntimeException, o que elas têm de diferente das outras, afinal de contas, somos todas da mesma família! Então, vamos às explicações.

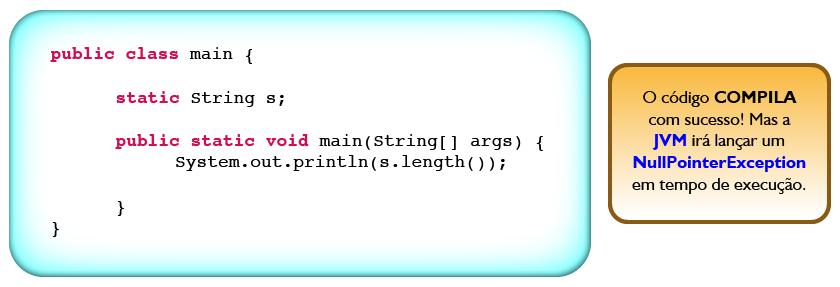
A maioria das RuntimeException são derivadas de problemas no seu código, ao invés de um problema que acontece em tempo de execução que você não poderia prever. Você não pode garantir que ao abrir um arquivo ele esteja lá, ou que o servidor está funcionando, mas você pode garantir que o seu código não faça acesso a um índice inexistente em um Array (é para isso que serve o atributo ‘length’).

Você desejará que as RuntimeException aconteçam durante o desenvolvimento. Você não desejaria ter o trabalho de tratar tal tipo de erro dentro de um bloco try/catch, por exemplo, e ter todo overhead de lidar com isso, para capturar alguma coisa na cláusula catch, que na realidade não deveria ter acontecido antes de tudo!

O bloco try/catch é para capturar condições excepcionais e não fluxo de código. Use a cláusula catch para tentar se recuperar de situações que você não pode garantir que irá acontecer com sucesso, ou, em último caso, exibir uma mensagem para o usuário e imprimir a pilha de erro para que alguém possa descobrir o que está errado!

Em exemplo clássico de uma RuntimeException é o famoso NullPointerException. Alguém já viu uma dessa antes? Aposto que sim! Esse tipo de exceção, como todos nos sabemos, acontece quando tentamos acessar uma referência de um objeto nulo!

Figura 10



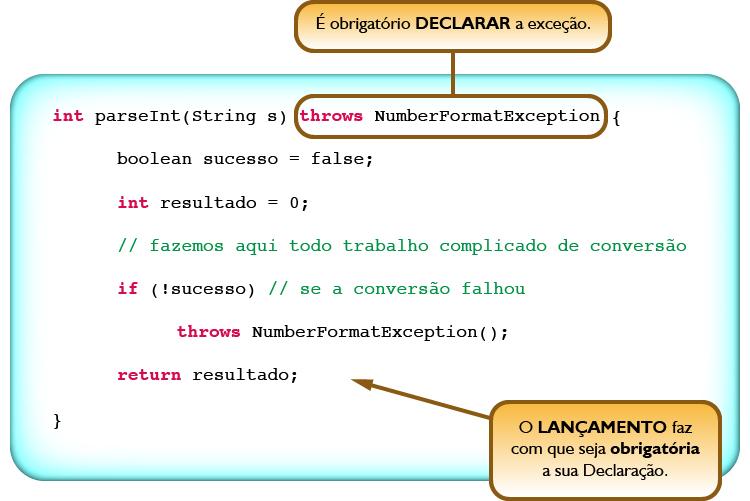
### **Exceções Programáticas**

Agora chegou a hora de conhecermos um pouco das exceções que o compilador pode detectar, as quais nos forçam a usar todas as maravilhas disponíveis na API Java para tratamento de exceções. Vamos dar uma olhada no seguinte caso.

Algumas classes da API Java têm métodos que recebem uma String como argumento, e converte essas Strings em tipos primitivos numéricos. Um bom exemplo dessas classes são as conhecidas “Wrapper Classes”.

Vamos tomar como exemplo de classe Wrapper a famosa java.lang.Integer. Vamos agora imaginar como seria a implementação do método parseInt(), que tem como objetivo receber uma String e retornar um número inteiro. Sendo assim, o programador sabiamente decide que se nesse método for passado uma String que não possa ser convertida em um tipo numérico, então, o método deve lançar uma exceção do tipo NumberFormatException. Vejamos o exemplo a seguir.

Figura 11



### **Atividade 02**

1. Escreva uma classe que tenha um método que lance uma exceção utilizando a cláusula throws. Dessa vez, devemos utilizar uma exceção da API que seja filha de RuntimeException. Note o comportamento do compilador e diga o que foi notado.
2. Criar um programa Java que utilize o método criar acima e trate a exceção dentro de um bloco try/catch. Faça o mesmo com esse exercício, note o comportamento do compilador.

### **Conclusão**

Existe um monte de exceções, passaríamos o dia falando delas e não acabariam. A tabela a seguir traz um resumo das exceções mais comuns.

|  |  |  |
| --- | --- | --- |
| Exception(Mais comuns) | Descrição | Tipicamente Lançadas |
| ArrayIndexOutOfBoundsException | Lançada quando tentamos acessar um Array com um valor de índice inválido (Negativo ou maior do que o tamanho do Array). | Lançada pela JVM |
| ClassCastException | Lançada quando tentamos fazer um ‘cast’ para uma referência e ocorre uma falha | Lançada pela JVM |
| IllegalArgumentException | Lançada quando um método recebe um argumento diferente do tipo esperado. | Programaticamente |
| NullPointerException | Lançada quando tentamos acessar uma referência de um objeto null. | Lançada pela JVM |
| NumberFormatExceptoin | Lançada quando um método que converte uma String em um número recebe uma String que não é convertida. | Programaticamente |

### **Resumo**

Bom, chegamos ao final de mais uma aula, agora estamos mais familiarizados com a API Java para tratamento de exceções. Vimos toda sua Hierarquia, sabemos agora que Erros em Java são diferentes de Exceções, e que essas últimas podem ser vista em dois grupos (Lançadas pela JVM), as quais são todas aquelas do tipo RuntimeException e suas descendentes, e Progamaticamente, que são todas as demais, as quais o compilador faz questão que a tratemos, pois, diferentemente das RuntimeException, podemos evitar a sua ocorrência e até nos recuperarmos da forma adequada a cada situação. Vimos também a sintaxe básica para lançamento de exceções (throws), assim como o bloco try/catch, o qual utilizamos para capturar e tratá-las. Por fim, fizemos um resumo dos tipos mais comuns de exceções e vimos alguns casos em que elas podem ocorrer.

### **Autoavaliação**

1. Crie uma classe contendo um método que lance uma exceção do tipo NumberFormatExceptoin.
2. Crie um programa Java que utilize o método criar acima e trate a exceção dentro de um bloco try/catch.
3. Crie um programa Java que manipule os valores de um Array, porém, devemos fazer com que ele acesse um índice inválido, de forma que o programa irá compilar corretamente, porém, será lançada uma exceção em tempo de exceção do tipo ArrayIndexOutOfBoundsException.
4. Adapte o programa criado na questão 2 desta autoavaliação para imprimir a pilha de erros utilizando o printStackTrace().

## **Lançamento de Exceções**

### **Apresentação**

Na aula passada, aprendemos como tratar situações anormais de execução dos programas, mais especificamente como tratar exceções que podem acontecer em situações não comuns. Nesta aula, iremos aprender como gerar nossas próprias exceções ao se perceber uma situação inadequada para continuação da execução do nosso programa. Boa aula!

#### **Objetivos**

Ao final desta aula, você será capaz de:

* Criar suas próprias exceções customizadas.
* Lançar novas exceções.
* Usar a cláusula finally do bloco try/catch.
* Repassar exceções de que não queremos tratar.

### **Relembrando!**

Vimos na aula anterior por que o tratamento de exceções é importante para que possamos criar programas robustos e tolerante a falhas. Aprendemos que as exceções são todas subclasses de Throwable. Vimos também como um método informa ao compilador que ele poderá causar risco (cláusula throws), e a sintaxe básica para capturar e tratar uma exceção (bloco try/catch). Agora, vamos um pouco mais a fundo no tratamento de exceções... chegou a hora de virar gente grande! Vamos lá então... mãos à obra!

Cláusula Finally: para as coisas que você deseja fazer não importa como.

Lembra que no capítulo anterior vimos a sintaxe básica do bloco try/catch? Pois então, ele ainda quarda algumas cartas na manga para combater as exceções do mundo Java. E para que serve essa danada dessa cláusula finally? Vou contar uma breve historinha para que possamos entender melhor o seu sentido.

Imagine que você esta tentando cozinhar alguma coisa, você tem que primeiramente ligar o fogão, certo? Ok, supondo que o alimento preparado tenha queimado! Você terá que desligar o fogão! Poxa, sou um péssimo cozinheiro! É verdade! Mas agora suponha que você tem visto muito Ana Maria Braga e está craque na cozinha e o que você fez ficou uma maravilha, como fica na televisão, você ainda tem que desligar o fogão no final de tudo!

Perceba que não importa qual for o resultado, FALHA ou SUCESSO, você sempre terá que realizar a mesma ação ao final do processo. É no bloco finally que colocaremos o código que deverá rodar, não importa o que aconteça.

Então, vamos dar uma olhada na sintaxe:

try {

ligarFogao();

fogao.assar();

} catch ( FogaoExcecao ex) {

ex.printStackTrace();

} finally { // Executará

desligarFogao(); //NÃO importa o que aconteça

}

Qual a vantagem de termos o bloco finally nesta situação? Alguém se arrisca? Sem o finally temos que colocar o método desligarFogao() em dois locais, primeiro dentro do bloco try, para o caso em que tudo ocorra como o esperado, e outra vez dentro do bloco catch, caso alguma exceção seja lançada. Em resumo, o bloco finally permite que coloquemos todos os nossos códigos de limpeza em um único local, ao invés de duplicarmos como no exemplo a seguir:

try {

ligarFogao();

fogao.assar();

desligarFogao();

} catch ( FogaoExcecao ex) {

ex.printStackTrace(); //Código DUPLICADO!

desligarFogao(); //Sem o finally as coisas ficam mais difíceis!

}

#### **Desafio!**

Se dentro de um bloco TRY ou CATCH tivesse um ‘return’ em seu corpo, o que aconteceria? O bloco finally ainda sim executaria, ou seria ignorado? Pense... ao final deste assunto falaremos a resposta!

Agora, vamos dar uma pausa para colocarmos em prática o que aprendemos até agora, então, vamos lá, mãos a obra! E continue pensando no nosso Desafio!

### **Atividade 01**

1. Crie uma classe que contenha um método que lance uma exceção padrão da API a sua escolha.
2. Crie um programa Java que utilize o método criado acima e trate a exceção dentro de um bloco try/catch/finally.
3. Faça com que o programa criado nos passos acima execute de duais maneiras: Sucesso e Falha. Dentro de cada bloco try/catch/finally imprima uma mensagem informando que o código foi processado. Faça isso para os dois casos.

### **Exercitando a Mente**

Para melhor fixar o nosso entendimento, vamos analisar o código abaixo e responder as seguintes perguntas:

* Qual você acha que seria a saída do programa abaixo?
* Qual seria a saída do programa se fizéssemos a seguinte alteração no código: String teste = “yes”; ? Assuma que ExcecaoAssustadora estende de Exception.

public class TesteExcecao {

public static void main(String args []) {

String teste = “no”;

try {

System.out.println(“Início do try”);

facaCoisaPerigosa(teste);

System.out.println(“Final do try”);

} catch ( ExcecaoAssutadora ex ) {

System.out.println(“Excecao assustadora”);

} finally {

System.out.println(“finally”);

}

System.out.println(“Final do main”);

}

​

static void facaCoisaPerigosa(String teste) throws ExcecaoAssustadora {

System.out.println(“Início coisa perigosa”);

if ( “yes”.equals(teste) ) {

throw new ExcecaoAssustadora();

}

System.out.println(“Final da coisa perigosa”);

return;

}

}

Agora que aprendemos mais um pouco sobe a sintaxe para tratamento de exceções, voltamos a nossa pergunta desafio. Você já sabe a resposta? Espero que sim! Então, vamos lá... Lembra que foi dito no início da aula que o código do bloco finally executaria não importando o que acontecesse? Pois é puramente verdade! Mesmo que tenhamos um retorno dentro do bloco try ou do catch, ainda sim o nosso finally será executado! Lembre-se, não importa o que aconteça, o finally sempre será executado!

### **Capturando Múltiplas Exceções**

Por essa você não esperava! Você sabia que um método pode lançar várias exceções? Pois é, um método poderá lançar quantas exceções sejam necessárias. Mas, na declaração do método, será obrigatório informar cada uma delas, contudo, se duas ou mais exceções possuem uma superclasse em comum, então, nesse caso o método pode declarar apenas as superclasses na cláusula throws.

public class Lavanderia {

public void lavar() throws ExcecaoCalca, ExcecaoCamisa {

// Código que poderá lança qualquer uma das Exceções.

}

}

No código acima, temos um exemplo de um método lançando mais de uma exceção, considerando que ExcecaoCalca e ExcecaoCamisa fossem filhas de ExcecaoRoupa, o mesmo código poderia ser escrito da seguinte forma.

public class Lavanderia {

public void lavar() throws ExcecaoRoupa {

// Código que poderá lança qualquer uma das Exceções.

}

}

Como consequência desse fato, quem utilizar-se desse método deverá capturar cada uma das exceções. Mais um detalhe, a ordem em que você captura as múltiplas exceções dentro do bloco catch é de extrema importância, mas veremos isso com mais detalhes adiante. Vamos dar uma olhada na sintaxe para capturar várias exceções em um único bloco try/catch.

public class Besteirol {

public void fazBesteira(){

Lavanderia lavanderia = new Lavanderia();

try {

lavanderia.lavar();

} catch (ExcecaoCalca) {

// código de recuperação

} catch (ExcecaoCamisa) {

// código de recuperação

}

}

}

Veja que no código acima colocamos duas cláusulas catch, isso porque o método lavanderia.lavar() poderá lançar tanto ExcecaoCalca quanto ExcecaoCamisa. Observe que neste caso não importa a ordem em que as capturamos no bloco catch, pois ambas possuem o mesmo nível de hierarquia, ou seja, são subclasses de ExcecaoRoupa. Sendo assim, esse mesmo código também poderia ser escrito da seguinte maneira.

public class Besteirol {

public void fazBesteira(){

Lavanderia lavanderia = new Lavanderia();

try {

lavanderia.lavar();

} catch (ExcecaoRoupa) {

// código de recuperação

}

}

}

Note que agora temos apenas um bloco catch tratando a ExcecaoRoupa. Como essa última é superclasse de ambas as exceções (ExcecaoCalca e ExcecaoCamisa), eu posso simplificar meu código capturando apenas uma exceção que seja superclasse das demais.

Esse tipo de abstração também tem as suas desvantagens. Perceba que dessa forma daremos o mesmo tratamento para ambos os tipos de exceções. Sendo assim, se quisermos especificar um tratamento diferenciado de acordo com a exceção levantada, temos que de fato utilizar múltiplas cláusulas catch.

### **Criando Exceções**

Estamos quase especialistas no tratamento de exceções, já aprendemos várias coisas, como criar métodos que lançam exceção, tratá-las dentro do bloco try/catch, aprendemos a utilizar o bloco finally, até mesmo a lançar várias exceções e capturá-las uma a uma. Porém, ainda não sabemos como criar as nossas próprias exceções. Então, chegou a hora de resolver esse problema, sem mais demoras, vamos direto ao assunto.

public class ExcecaoRoupa extends Exception {

public ExcecaoRoupa (String mensagem) {

super(mensagem);

}

}

Pronto, muito simples, não acha? É isso mesmo, a API Java é muito fácil de se trabalhar e poderosa também, para criarmos nossa própria exceção basta criar uma classe que estenda de Exception e sobrescrever o método getMessagem() ou qualquer outro que você desejar. A mesma classe poderia ser escrita da seguinte forma.

public class ExcecaoRoupa extends Exception {

@Override

Public String getMessage() {

return “Voce não está utilizando a roupa de forma adequada!”;

}

}

Mas, fazendo dessa forma, fica bem menos flexível do que a primeira, notem que assim não podemos mais modificar a mensagem a ser exibida quando lançarmos uma exceção do tipo ExcecaoRoupa. Esse exemplo só foi para demonstrar que podemos sobrescrever qualquer método da classe Exception para customizar nossas próprias exceções da forma que melhor nos atenda.

Chegamos ao momento especial em relação ao tratamento de exceções. Até agora estávamos preocupado em tratar as exceções que poderiam acontecer, tentando contornar a situação utilizando o bloco try/catch. Mas, agora é a nossa vez de lançarmos nossas próprias exceções, agora que aprendemos como criá-las, vamos ver como lançá-las! Um dia da caça outro do caçador!

Lembra deste código? Vamos incluir agora o seu conteúdo para lançar uma exceção do tipo ExcecaoRoupa.

public class Lavanderia {

public void lavar() throws ExcecaoRoupa {

if ( material == sapato )

throw new ExcecaoRoupa(“Apenas roupas! Lave o sapato na mão!”);

}

}

}

No código acima, verificamos se o material a ser lavado é um sapato, caso verdadeiro, lançamos uma exceção informando que apenas Roupas poderão ser lavadas na máquina de lavar roupas, então, lançamos uma exceção.

Em Java, utilizamos a cláusula throw para lançar uma exceção, e como uma exceção também é um objeto, utilizamos o new para criá-la. Fácil não é? Como dito no capítulo anterior, toda a API Java para tratamento e criação de exceções foi desenvolvida de uma forma simples, limpa e intuitiva, nos fornecendo recursos bastante poderosos para que possamos tornar nossas aplicações seguras e confiáveis.

Agora que já estamos grandinhos e sabemos criar as nossas próprias exceções, vamos colocar em prática toda essa habilidade para que possamos fixar melhor o que foi aprendido até o momento. Então, vamos lá, chega de moleza!

### **Atividade 02**

1. Crie uma classe de exceção que estenda de Exception chamada de ExcecaoAlfabeto.
2. Crie um programa Java com um método que receba uma String de parâmetro chamado validarLetrasAlfabeto. Esse método deve verificar se cada um dos caracteres da String é uma letra do alfabeto, se for encontrado algum caractere especial, devemos lançar uma exceção do tipo criado no exercício anterior.
3. Crie duas classes de exceção que estenda de ExcecaoAlfabeto criada no primeiro exercício, cada uma com o seguinte nome (ExcecaoLetras e ExcecaoNumeros).
4. Ajuste o programa criado no passo dois criando mais dois métodos, um que valida se a String do parâmetro contém apenas caracteres que são letras, e se não for lance uma exceção do tipo ExcecaoLetra e no segundo valide se a String contém apenas números, em caso negativo lance a exceção ExcecaoNumero.

### **Repassando Exceções**

Até agora aprendemos como tratar as exceções que um determinado método que utilizamos possa lançar. Mas, quando não queremos tratá-las, o que fazer? Pode não ser nossa responsabilidade tratar um determinado tipo de exceção. Nesse tipo de situação devemos então repassá-las para o próximo método da pilha. E como fazemos isso? Vamos reutilizar a nossa classe Besteirol criada anteriormente.

public class Besteirol {

public void fazBesteira() throws ExcecaoRoupa {

Lavanderia lavanderia = new Lavanderia();

lavanderia.lavar();

}

}

O que foi que aconteceu com a cláusula try/catch que tinha aqui antes? Bom, agora não precisamos mais dela, uma vez que repassamos o problema, não precisamos mais tratar dele. Para isso, utilizamos a cláusula throws na declaração do método contendo a mesma exceção declarada no método lavanderia.lavar(). A regra básica para satisfazer as vontades do compilador é a seguinte: ou você trata a exceção dentro de um bloco try/catch ou você repassa o problema na declaração do seu método utilizando a cláusula throws. É simples assim, se você não quer resolver o problema, então, repasse para quem quiser tratá-lo, o que importa é fazer o compilador feliz!

### **Resumo**

Bom, chegamos ao final de mais uma aula bastante emocionante. Vimos mais detalhadamente a sintaxe do bloco try/catch e do nosso novo amigo finally. Aprendemos também que podemos lançar vários tipos de exceções, de acordo com a nossa necessidade, e assim como podemos lançar, podemos também capturar várias exceções em um único bloco catch. Aprendemos a criar nossas próprias exceções e também como lançá-las. Por fim, vimos como repassar as exceções que não queremos ou não devemos tratar.

### **Autoavaliação**

1. Suponha que temos os seguintes tipos de exceções: ExcecaoRoupa, ExcecaoRoupadeBaixo, ExcecaoCalcinha. Seguindo a ordem, ExcecaoRoupa é superclasse de ExcecaoRoupadeBaixo que, por sua vez, é superclasse de ExcecaoCalcinha. Qual a ordem que devemos capturar essas exceções em um mesmo bloco try/catch? Explique por que, nesse caso, a ordem é relevante!

## **Atributos e Métodos Estáticos, Classes Abstratas e Interfaces**

### **Apresentação**

Até agora, para efeitos de projeto de sistemas, você viu classes (concretas) e objetos, unicamente. Porém, é importante que você saiba que há recursos mais sofisticados na linguagem Java que permitem que a gente projete de forma mais genérica e aproximada o mundo real diante de situações que ainda não foram apresentadas nos exemplos anteriores, mas que farão sentido à medida que formos observando as sutilezas que envolvem a arte de desenvolver soluções em Tecnologias de Informação (TI)!

Para tanto, teremos a apresentação de atributos e métodos comuns a todos objetos de uma classe, os chamados atributos e métodos estáticos. Veremos também classes da qual não podemos instanciar objetos diretamente, as chamadas classes abstratas.

Finalmente, conheceremos o conceito de interface em Java, que representa um “contrato” assumido por classes e que as obrigam a implementar um determinado conjunto de métodos de uma determinada forma. Lembre que você vai realizar práticas através dos exercícios apresentados.

### **Atributos Estáticos**

*play\_circle\_filled*

Vídeo 02 - Atributos e Métodos Estáticos

Primeiramente, vejamos essa denominação: “atributo estático”. Nas aulas anteriores, abordamos de forma exaustiva o conceito de atributos, os quais são variáveis declarados nas classes, e que possuem tipo e nome. Você viu também que os valores dos atributos são definidos pelos próprios objetos instanciados de uma dada classe. Essa atribuição pode ocorrer de forma direta (caso o atributo seja público), ou através de métodos setters que permitem acessá-los (que é o caso dos atributos que são privados), ou mesmo através dos próprios construtores da classe.

Até aí, nenhuma novidade, correto?

Agora, vejamos a denominação “estático”. O que você compreende por esse termo? Algo imóvel, imutável, paralisado?

Pois bem, é aí que “mora o perigo”, pois o termo escolhido para tal variável não foi tão adequado, e dá tipicamente a ideia de uma constante.

Atributo Estático ≠ Atributo Constante

A ideia por trás do conceito de atributos e métodos estáticos é a de que objetos de uma mesma classe podem e, em algumas situações devem, compartilhar valores em comum, caso contrário, teríamos que criar atributos que precisariam ser atualizados todos ao mesmo tempo, em cada objeto criado.

Vejamos o seguinte exemplo. Um sistema que controle relógios de ponto de várias filiais com o mesmo fuso horário, nessa situação, quando tivermos que fazer um ajuste ou atualizar a hora, minutos e segundos, teremos que fazê-lo em todos os relógios. Porém, se esse atributo não “pertencer” aos objetos, e sim à própria classe, todos saberão a hora certa, pois se baseiam no mesmo valor. Observe na Listagem 1 o código da classe RelogioPonto.

public class RelogioPonto{

public static int hora; // atributo estático

public static int minuto; // atributo estático

public static int segundo; // atributo estático

​

private int id;

private String nomeFilial;

​

public RelogioPonto(){

}

public RelogioPonto(int ident, String filial){

this.id = ident;

this.nomeFilial = filial;

}

​

public int getId(){

Return id;

}

public void setId(int id){

this.id = id;

}

public String getNomeFilial(){

return nomeFilial;

}

public void setNomeFilial(String nomeFilial){

this.nomeFilial = nomeFilial;

}

}

Listagem 1 - Classe RelogioPonto

Veja o uso da palavra reservada static na declaração dos atributos hora, minuto e segundo. Essa faz com que o compilador entenda que não será necessário reservar um espaço de memória em cada objeto da classeRelogioPonto para armazenar o valor da hora, minuto e segundo, pois esses valores serão compartilhados por todos os objetos dessa classe. Logo, será necessário apenas um único espaço para armazenar esse valor, juntamente com os códigos dos métodos da classe.

Vamos criar uma aplicação simples para exemplificar o uso dos atributos estáticos hora, minuto e segundo. A Listagem 2 apresenta o código de tal aplicação.

public class RelogioPontoMain{

public static void main(String[] args){

RelogioPonto filialCentro = new RelogioPonto(1, "Centro");

RelogioPonto filialAlecrim = new RelogioPonto(2, "Alecrim");

RelogioPonto filialRocas = new RelogioPonto(3, "Rocas");

​

filialAlecrim.hora = 7;

filialAlecrim.minuto = 45;

filialAlecrim.segundo = 30;

​

System.out.println(

"Filial Alecrim: " +filialAlecrim.hora+ ":" +filialAlecrim.minuto+":" + filialAlecrim.segundo);

System.out.println(

"Filial Centro: " +filialCentro.hora+ ":" +filialCentro.minuto+":" + filialCentro.segundo);

System.out.println(

"Filial Rocas: " +filialRocas.hora+ ":" +filialRocas.minuto+":" + filialRocas.segundo);

}

}

Listagem 2 - Aplicação usando a classe RelogioPonto

Observe que criamos 3 (três) objetos RelogioPonto e na sequência pegamos um desses (filialAlecrim) para atribuir os valores de hora, minuto e segundo. Logo, em seguida, imprimimos os valores dos três objetos e teremos o seguinte resultado:

filial Alecrim: 7:45:30

filial Centro: 7:45:30

filial Rocas: 7:45:30

Isso ocorre, pois, quando atribuímos valores a variáveis estáticas através de um objeto, estamos atribuindo o valor para todos os objetos da mesma classe. Por isso, alguns chamam os atributos estáticos de atributos de classe. Para não sermos redundantes, faremos uma substituição da instrução.

De...

filialAlecrim.hora = 7;

filialAlecrim.minuto = 45;

filialAlecrim.segundo = 30;

Para...

RelogioPonto.hora = 7;

RelogioPonto.minuto = 45;

RelogioPonto.segundo = 30;

Pois, quando estamos tratando de atributos de classe (atributos estáticos) em Java, é comum e é uma boa prática usarmos o seu valor utilizando o nome da classe seguido do operador ponto “.” e o atributo que desejarmos.

Observe que, com essa mudança, mostramos que não é necessário criar um objeto da classe RelogioPonto para que se possa acessar seus atributos estáticos. Quando estáticos, eles estão disponíveis antes mesmo de se criar o objeto.

Com isso, temos como análise geral o seguinte quadro:

Atributo Estático = Atributo Comum ou Compartilhado

### **Métodos Estáticos**

A mesma ideia dos atributos estáticos pode ser ampliada e aplicada para os métodos, pois, se quisermos alterar o modo de acesso dos atributos hora, minuto e segundo, tornando-os privados (private), por exemplo, teremos que criar métodos que sejam capazes de alterá-los.

Não é uma boa prática modificar valores de atributos estáticos através de referências para seus objetos. Imagine 10 objetos tentando alterar os valores compartilhados, a probabilidade do sistema se tornar caótico (ou dar algum erro) é muito grande. Os métodos estáticos surgiram basicamente para operarem sobre os atributos estáticos, ou que não realizam operação alguma sobre os atributos dos objetos. Ou seja, não é do contexto do objeto, mas, sim, de sua classe.

Alterando o exemplo apresentado anteriormente (Listagem 1) para trabalhar com métodos estáticos, chegaremos ao exemplo apresentado na Listagem 3. Observe o uso da palavra reservada static na declaração dos métodos que operam sobre os atributos estáticos.

public class RelogioPonto{

public static int hora; // atributo estático

public static int minuto; // atributo estático

public static int segundo; // atributo estático

​

private int id;

private String nomeFilial;

​

public RelogioPonto(){

}

​

public RelogioPonto(int ident, String filial){

this.id = ident;

this.nomeFilial = filial;

}

​

public static int getHora(){

return hora;

}

​

public static void setHora(int hora){

RelogioPonto.hora = hora;

}

​

public static int getMinuto(){

return minuto;

}

​

public static void setMinuto(int minuto){

RelogioPonto.minuto = minuto;

}

​

public static int getSegundo(){

return segundo;

}

public static void setSegundo(int segundo){

RelogioPonto.segundo = segundo;

}

public int getId(){

Return id;

}

public void setId(int id){

this.id = id;

}

public String getNomeFilial(){

return nomeFilial;

}

public void setNomeFilial(String nomeFilial){

this.nomeFilial = nomeFilial;

}

}

Listagem 3 - Classe RelogioPonto com métodos estáticos

Agora, mais uma alteração deve ser feita na Listagem 2 , de...

RelogioPonto.hora = 7;

RelogioPonto.minuto = 45;

RelogioPonto.segundo = 30;

Para...

RelogioPonto.setHora(7);

RelogioPonto.setMinuto(45);

RelogioPonto.setSegundo(30);

Com isso, temos a utilização dos métodos estáticos diretamente da classe RelogioPonto. Assim como no caso de atributos, também não é necessário criar o objeto da classe para que se possa usar o método.

### **Atividade 01**

1. Crie uma classe para representar Livros em uma biblioteca, com os seguintes campos: título, autor, gênero, editora e edição. Assuma que o atributo gênero pode ter um dos seguintes tipos: Ficção, Ciências Naturais, Literatura ou Ciências Humanas. Assuma que tais tipos de gêneros são valores já conhecidos pela classe Livro, e que todos os objetos devem compartilhá-los.

### **Classes Abstratas**

Ao longo dos nossos estudos sobre a programação orientada a objetos em Java, vimos os conceitos e vários exemplos de implementações de classes e objetos. Você viu que as classes são como “moldes” que geram os objetos segundo a sua definição.

Todas as classes vistas até agora são o que chamamos de classes concretas, pois são utilizadas para gerar objetos diretamente. Ou, ainda, estão “prontas” para serem usadas para gerar seus objetos, como formas de bolo ou biscoitos prontas para receberem a massa e darem objetos quentinhos! Mas, existem classes das quais não é possível gerar (instanciar) objetos, são as chamadas classes abstratas.

Primeiro, vejamos o que é uma classe abstrata.

#### **Saiba Mais!**

Classe abstrata é aquela declarada como tal, através da palavra reservada abstract, e que pode definir pelo menos um método abstrato.

A Listagem 4 apresenta um exemplo de uma classe abstrata, chamada FiguraAbstrata. Uma figura abstrata representa uma figura qualquer, por isso não sabe como ser desenhada e, de fato, não pode ser instanciada, pois ela define um método abstrato (desenha) para se autodesenhar, que ainda está indefinido.

No nosso exemplo, a classe FiguraAbstrata indica que métodos as figuras derivadas dela (subclasses ou classes filha da mesma) devem realizar ou implementar. Esses métodos são os chamados métodos abstratos. Uma classe abstrata não pode instanciar objetos, porém, pode ser derivada por outras classes que, por sua vez, deverão implementar seus métodos ou declarar-se também abstratas.

public abstract class FiguraAbstrata{

/\*\*

\* Método abstrato de desenho de uma figura

\*/

abstract public void desenha();

}

Listagem 4 - Classe abstrata FiguraAbstrata

A Listagem 5 apresenta a classe derivada Quadrado (subclasse) da classe FiguraAbstrata.

A classe Quadrado realiza o que precisa para ser desenhada, implementando o método desenha(). É possível criar instâncias (objetos) da classe Quadrado, porque ela não é abstrata nem é declarada como tal, e também define uma implementação concreta para o método desenha().

public class Quadrado extendes FiguraAbstrata{

@Override

public void desenha()

System.out.println(" ---");

System.out.println("| |");

System.out.println(" ---");

}

}

Listagem 5 - Classe Quadrado

A importância das classes abstratas dá-se principalmente em tempo de projeto do sistema, cria um nível a mais de abstração e torna o projeto mais reutilizável.

### **Atividade 02**

1. Para exercitar o conceito de classes abstratas, crie outras classes derivadas (subclasses) de FiguraAbstrata como Trapezio, Triangulo etc. Em seguida, implemente o método desenha().
2. Finalmente, crie também uma classe com método main() para tentar instanciar objetos tanto da classe FiguraAbstrata quanto das subclasses que você criou. Não deixe de fazer chamada ao método desenha() para ver o que acontece.

### **Interface**

Imagine uma grande equipe de desenvolvimento de software, onde há subdivisões de acordo com a parte que o software irá atuar, logo, é necessária uma espécie de contrato para garantir “independência” entre as equipes. Assim, saberão o que deverão fazer sem que saibam exatamente como está sendo implementado pela outra equipe.

Imaginemos que fabricamos softwares para serem implantados em brinquedos como robôs, carrinhos e aviões que possuem como característica principal a capacidade de se locomoverem segundo os comandos de um controle remoto. Cada brinquedo entende o que precisa fazer para mover-se para frente ou para trás, parar e emitir um sinal de localização.

Os brinquedos são fabricados por uma companhia e os controles por uma empresa parceira em produção de eletrônicos. Ambos são implementados na linguagem Java. O contrato estabelecido entre os desenvolvedores é de que todos os produtos que necessitam ser controlados por um controle remoto deverão implementar a interface Mobilidade, cujo código é apresentado na Listagem 6.

public interface Mobilidade{

public void andarFrente();

public void andarTras();

public void parar();

public void virarDireita(int graus);

public void virarEsquerda(int graus);

}

Listagem 6 - Interface mobilidade

De um lado, as classes que representam os produtos a serem controlados irão “inscrever-se” ou declarar-se implementadoras dessa interface. De outro lado, a classe controle remoto irá operar sobre esses métodos sem conhecer nada além do que ela precisa saber, ou seja, o conjunto de métodos da interface mobilidade. Por isso, chamados de interface, ou modo de ver os objetos.

A Listagem 7 apresenta a classe Robô que implementa a interface Mobilidade.

public class Robo implements Mobilidade{

public void andarFrente(){

System.out.println("[robô]: andando para frente...");

}

public void andarTras(){

System.out.println("[robô]: andando para trás...");

}

public void parar(){

System.out.println("[robô]: parado");

}

public void virarDireita(int graus){

System.out.println("[robô]: virando para direita " + graus + " graus");

}

public void virarEsquerda(int graus){

System.out.println("[robô]: virando para esquerda " + graus + " graus");

}

public void moveBracoDireito(){

System.out.println("[robô]: movendo braço direito");

}

public void moveBracoEsquerdo(){

System.out.println("[robô]: movendo braço esquerdo");

}

}

Listagem 7 - Classe Robô

Veja que a classe Robô se define implementadora da interface Mobilidade através da palavra reservada implements, seguida do nome da interface Mobilidade. Outros métodos podem ser definidos livremente pelas classes que implementam interfaces. O que vemos na definição dos métodos moveBracoDireito() e moveBracoEsquerdo(), por exemplo, são típicos de funções de um robô, mas que certamente não se aplicaria a um veículo de brinquedo.

A Listagem 8 apresenta a classe Trator, que também implementa a interface Mobilidade. Assim como a classe Robô, a classe Trator implementa a interface Mobilidade e declara outro método chamado de ergueEscavadeira(), que representa uma funcionalidade extra e que não será controlada remotamente.

public class Trator implements Mobilidade{

public void andarFrente(){

System.out.println("[trator]: andando para frente...");

}

public void andarTras(){

System.out.println("[trator]: andando para trás...");

}

public void parar(){

System.out.println("[trator]: parado");

}

public void virarDireita(int graus){

System.out.println("[trator]: virando para direita " + graus + " graus");

}

public void virarEsquerda(int graus){

System.out.println("[trator]: virando para esquerda " + graus + " graus");

}

public void erqueEscavadeira(){

System.out.println("[trator]: ergue escavadeira");

}

}

Listagem 8 - Classe Trator

Assim, como o método moveBracoDireito() da classe Robô, o método ergueEscavadeira() da classe Trator faz uma atividade específica para o brinquedo. E, por isso, só é definido no próprio brinquedo. Por fim, temos a classe ControleRemoto que realiza operações sobre objetos “Móveis” que implementam a interface Mobilidade. Veja a Listagem 9.

public class ControleRemoto{

public void moverObjetoParaFrente(Mobilidade obj){

obj.andarFrente();

}

public void moverObjetoParaTras(Mobilidade obj){

obj.andarTras();

}

public void dobrarADireita(Mobilidade obj){

obj.virarDireita(graus);

}

public void dobrarAEsquerda(Mobilidade obj){

obj.virarEsquerda(graus);

}

}

Listagem 9 - Classe ControleRemoto

Observe que a classe ControleRemoto possui métodos que operam sobre objetos de classes que implementam a interface Mobilidade, sejam eles quais forem. Por isso, na declaração dos métodos, não está explícito classes como Trator ou Robô, pois o controle trata-os de forma transparente. Esse exemplo também representa uma das formas de polimorfismo.

### **Atividade 03**

1. Para praticar os conceitos aprendidos, crie uma classe com um método main(), que teste a classe controle remoto, através da criação de objetos da classe Trator e Robô e veja como são tratados de forma idêntica pela classe ControleRemoto.

### **Resumo**

Nesta aula, você viu técnicas sofisticadas para projetar sistemas através de classes abstratas que servem para definir comportamentos os quais deverão ser “materializados” em classes filhas, dando poder e abstração ao projeto de sistemas. Você estudou que classes podem definir atributos e métodos próprios e independentes de seus objetos, chamados de atributos ou métodos de classes, os quais podem ser acessados diretamente através do nome da classe e seu respectivo nome. Você viu também como definir contratos entre classes, através de um tipo especial da linguagem Java, conhecido como interface. Cada interface apenas declara assinaturas de métodos, não oferecendo implementações. As classes que implementam a interface são obrigadas a oferecer uma implementação para cada um dos métodos declarados na interface.

### **Autoavaliação**

1. Defina o que é um atributo estático.
2. Defina o que é um método estático.
3. Exercite a imaginação e crie para cada classe a seguir um atributo estático.
   1. Pessoa
   2. Carro
   3. Livro
   4. ConsultaMédica
   5. HistóricoEscolar
4. Você é capaz de comparar os conceitos de interface e classe abstrata, destacando semelhanças de um lado e diferenças do outro?
5. Defina uma outra interface, cujas classes Trator e Robô irão implementar, que represente a sua manutenção energética, pois ambos precisam recarregar baterias, ligar, desligar etc.

## **Juntando as Peças**

### **Apresentação**

Nesta aula, você verá vários conceitos da Programação Orientada a Objetos sendo usados de forma conjunta. O objetivo é entender como a POO funciona de fato. Através de exemplos e exercícios, vamos ver em ação os mecanismos da linguagem Java na implementação desse revolucionário paradigma. Assim, durante a aula você vai praticar os conceitos e a linguagem Java de forma mais ampla.

### **Encapsulamento e Herança**

Os conceitos de encapsulamento e herança são relacionados e se entrelaçam. Vimos anteriormente que cada classe define seus atributos, vimos também a forma de acesso a eles, de forma a preservar a segurança dos dados (atributos) com relação a alterações acidentais ou não desejadas.

Uma das formas de controlar tal acesso é simplesmente impossibilitar que código externo possa acessá-la. Isso é o que chamamos de encapsulamento. Você viu também que uma classe pode possuir outras classes descendentes, nesse caso, herdam seus respectivos atributos e comportamentos. Isso é o que chamamos herança. Veja os pilares da programação orientada a objetos.

Figura 01 - Pilares da POO



Um dos pontos que não discutimos em aulas anteriores diz respeito à forma de acesso (visibilidade) dos atributos de classes filhas. Voltemos a esse ponto.

Considere uma classe mãe (Listagem1) com os seguintes atributos: idade, nome, sobrenome materno e sobrenome paterno. E uma classe filha, chamada Filha (Listagem 2), em que iremos experimentar os modificadores de acesso diante da estrutura hierárquica definida entre essas classes.

public class Mae{

private int idade;

protected String nome;

private String sobrenomeMaterno;

private String sobrenomePaterno;

public Mae(String primeiroNome, String materno, String paterno){

this.nome= nome;

this.sobrenomeMaterno = materno;

this.sobrenomePaterno = paterno;

}

public String toString(){

return nome + " "+sobrenomeMaterno + " " + sobrenomePaterno;

}

}

Listagem 1 - Classe Mãe

public class Filha extends Mae{

public Filha(String primeiroNome, String materno, String paterno){

super(primeiroNome, materno, paterno);

System.out.println("Nome: " +this.nome);

System.out.println("Sobrenome paterno: " +this.sobrenomePaterno);

}

}

Listagem 2 - Classe Filha

Observe que a classe Filha herda todos os atributos de sua Mãe, porém, o atributo sobrenomeMaterno não pode ser acessado diretamente de dentro da classe Filha por ser do tipo private. Logo, a linha comentada representa um erro para o compilador Java.

Os demais atributos são do tipo protected e public. Atributos public podem ser acessados de qualquer lugar no programa e representam o extremo do acesso livre em Java. Atributos protected são aqueles que podem ser acessados apenas pelas subclasses da classe que declara tais atributos.

Os atributos public e protected são sempre visíveis em classes filhas.

Os atributos private não. Porém, eles estão lá! Não se esqueça, o fato de não estar visível não significa não existir. Pois, podemos modificá-los através dos construtores da classe mãe (fazendo chamadas com o super), assim como usando os métodos get e set definidos na classe mãe.

Lembre-se:

A classe filha sempre herda as características da mãe, mas só pode ver o que a mamãe deixa ;-).

Os exemplos que vimos sobre herança e encapsulamento demonstraram sua utilização através de atributos, porém, tudo que vimos se aplica exatamente da mesma maneira com os métodos.

Para praticar, que tal fazermos um exercício?

### **Atividade 01**

1. Altere a classe Filha adicionando novos atributos, um deles chamado apelido, que pode ser acessado por qualquer classe, outro chamado esportePreferido, restrito a classes que herdam da classe Filha.
2. Adicione uma classe Filha de Filha, chamada Neta e nela crie métodos para acessar todos os atributos.

### **Encapsulamento e Composição**

Vimos durante nossa viagem pelo mundo OO que objetos podem ser compostos por outros e mantêm as referências de tais objetos como seus atributos. Pois bem, esses atributos que representam referências para outros objetos estão sujeitos às mesmas regras de encapsulamento dos demais atributos de tipos primitivos de dados que cada classe mantém.

É interessante nos indagarmos em que situações queremos que a composição de um objeto tenha seu acesso preservado pelas demais instâncias, ou, até mesmo de classes filhas. Os exemplos vistos até então foram bastante simples e usados apenas para observar como funciona o compartilhamento de atributos entre os métodos. Porém, em sistemas complexos, é comum termos que tomar decisões como essa, pois desenvolver sistemas também pressupõe gerenciar complexidade.

Vejamos um exemplo de um sistema que gerencia a rotina diária de operações de uma oficina mecânica. Geralmente, um sistema é pensado, projetado e implementado, segundo competências bem específicas. Isso significa que a oficina possui diferentes setores para atendimento ao cliente, manutenção, estoque de peças, almoxarifado de ferramentas e equipamentos. Cada setor lida com uma competência específica do sistema. Logo, internamente, o sistema precisa encapsular informações que não interessam a determinadas competências.

Vamos isolar uma pequena situação e localizar no código como a composição e o encapsulamento resolvem nosso problema. Vejamos a Listagem 3 a seguir.

public class Peca{

private String tipo;

private String peso;

private String material;

//...

}

Listagem 3 - Classe Peça

A classe Peça representa um elemento atômico de todos os veículos. Ela é usada para representar diferentes peças mecânicas, tais como: parafuso, porcas, rolamentos, presilhas, correias etc.

Essa classe define todos os seus atributos como private, porém, sendo ela uma classe que servirá apenas como repositório desses dados, alguns projetistas iriam preferir defini-los como public, o que a tornaria equivalente a uma estrutura struct existente nas linguagens procedurais. Embora, por boa prática de POO, seja comum sempre manter todos os atributos privados ou protected, que é o princípio do encapsulamento.

Já a classe Componente, apresentada na Listagem 4, representa um elemento de mais alto nível no conjunto de elementos mecânicos de um carro, ou seja, um componente é um conjunto de peças organizadas de forma lógica e com uma determinada função no veículo. Por exemplo, roda é um conjunto de rolamento, aro, parafusos com função bem definida; já a direção do carro é composta da barra e rolamento; e assim por diante.

Finalmente, o motor de um veículo é tão complexo em relação a outros componentes que só ele poderia ser uma categoria específica, sendo definido como uma classe que herda da classe Componente, composto por um conjunto de outros componentes.

A Listagem 5 ilustra uma estratégia de implementação da classe Motor. Observe que ela é definida como sendo composta de um Array de objeto da classe Componente, mas por ser também um Componente, ela herda de tal classe.

Por fim, toda a estrutura mecânica do veículo irá fazer parte de um conjunto separado dos acessórios e demais componentes do acabamento do veículo, como estofamento, pintura etc

public class Componente{

protected String nome;

protected ArrayList<Peca>pecas;

​

public Componente(){

}

}

Listagem 4 - Classe Componente

public class Motor extends Componente{

private ArrayList<Componente>componentes;

}

Listagem 5 - Classe Motor

### **Atividade 02**

1. Defina uma classe Carro composta por diversos componentes e peças, de acordo com as classes definidas anteriormente. Incremente as classes se necessário.

Atenção: não esqueça de criar uma classe com um método main() para verificar o funcionamento correto dos métodos criados.

### **Herança, Composição e Polimorfismo**

Vejamos agora como os conceitos de herança, composição e polimorfismo podem co-existir numa mesma solução. Vamos partir do seguinte ponto de vista: existem objetos que possuem um fim em si mesmo, como um lápis, chave de fenda, mouse. No entanto, existem objetos que são usados para complementar outros, como um apontador, um mousepad; há outros que servem apenas como repositório: bolsas, organizadores, gavetas etc.

Para exemplificar, considere um objeto que funciona como repositório: um guarda-roupas! Como o próprio nome diz, ele serve para guardar outros objetos do tipo Roupa. Vamos imaginar os objetos que estão envolvidos nessa perspectiva, desde os objetos mais abstratos até os mais detalhados (ou concretos), conforme desejamos. Por exemplo, uma roupa possui características, como cor, tecido, manequim, preço, estampa etc. Uma roupa pode ser de vários tipos, como esportiva, social, casual, de banho, fardamento etc.

Logo, surge a questão: iremos definir os tipos existentes de roupa como subclasses da classe Roupa? Assim como no mundo real, no mundo OO não existe uma resposta certa para todas as situações. Isso irá variar dependendo do nível de detalhamento que você desejar implementar sua solução. Uma alternativa é definir um atributo, chamado tipo, na classe Roupa.

Outra forma é definir uma subclasse para cada tipo de roupa que existir. Uma solução intermediária é definir subclasses para os tipos de roupas mais comuns e reservar um atributo para descrevê-la quando não houver uma classificação pré-definida. Devido a restrições de espaço, ilustraremos apenas essa última solução. A seguir, são apresentadas as classes de tal solução. A Listagem 6 ilustra a classe Roupa.

public class Roupa{

private String descricao;

private String cor;

private String situacao;

private String estampa;

private int manequim;

​

public Roupa(){

}

//gets e sets

public void dobrar(){

System.out.println("Roupa dobrada");

}

public void lavar(){

System.out.println("Roupa lavada");

}

public String toString(){

return this.descricao+" "+this.cor+" "+this.manequim+" "+this.situacao;

}

}

Listagem 6 - Classe Roupa

Vamos agora definir os tipos principais de roupas existentes, são eles: esportivas, sociais e fardamentos. Logo, precisamos gerar mais três classes derivadas de roupa. A Listagem 7, Listagem 8 e a Listagem 9 ilustram o código de tais classes.

public class Esporte extends Roupa{

public void lavar(){

super.lavar();

System.out.println("Roupa Esporte lavada.");

}

public void dobrar(){

super.dobrar();

System.out.println("Roupa Esporte dobrada.");

}

public String toString(){

return "Esporte"+super.toString();

}

}

Listagem 7 - Classe Esporte

public class Social extends Roupa{

public void lavar(){

super.lavar();

System.out.println("Roupa Social lavada.");

}

public void dobrar(){

super.dobrar();

System.out.println("Roupa Social dobrada.");

}

public String toString(){

return "Esporte"+super.toString();

}

}

Listagem 8 - Classe Social

public class Fardamento extends Roupa{

public void lavar(){

super.lavar();

System.out.println("Fardamento lavado.");

}

public void dobrar(){

super.dobrar();

System.out.println("Fardamento dobrado.");

}

}

Listagem 9 - Classe Fardamento

public class GuardaRoupas {

​

public ArrayList<Roupa>portaDireita;

public ArrayList<Roupa>portaEsquerda;

public ArrayList<Roupa>gavetaDireita;

public ArrayList<Roupa>gavetaEsquerda;

​

public GuardaRoupas() {

this.portaDireita = new ArrayList<Roupa>();

this.portaEsquerda = new ArrayList<Roupa>();

this.gavetaDireita = new ArrayList<Roupa>();

this.gavetaEsquerda = new ArrayList<Roupa>();

}

​

public static void addRoupa(ArrayList<Roupa>compartimento, Roupa r) {

compartimento.add(r);

}

​

public static void retiraRoupa(ArrayList<Roupa> compartimento,Roupa r) {

if(compartimento.contains(r)) {

compartimento.remove(r);

}

}

​

public static void confereRoupas(ArrayList<Roupa>compartimento) {

for(Roupa r : compartimento) {

System.out.println(r);

}

}

}

Listagem 10 - Classe GuardaRoupa

Vamos agora criar uma classe que serve para armazenar objetos do tipo Roupa e suas derivadas. A classe GuardaRoupas, apresentada na Listagem 10, é responsável por armazenar as roupas em compartimentos distintos por categoria.

Observe que os métodos da classe GuardaRoupa adicionam e retiram roupas independentemente de serem social, esportiva ou fardamento. No método confereRoupas(), será mandado uma mensagem para que cada objeto roupa exiba seus dados, através do polimorfismo. Observe que cada objeto irá executar o método como sua classe definiu e as mensagens serão enviadas de forma transparente a todos eles.

Vejamos, na Listagem 11, agora, uma classe que define um método main() para criação de objetos Roupa e GuardaRoupa.

public class VestuarioMain {

​

public static void main(String[] args) {

GuardaRoupas guardaRoupas = new GuardaRoupas();

​

Esporte camiseta = new Esporte();

camiseta.setCor("branca");

camiseta.setDescricao("regata");

camiseta.setManequim(40);

​

Social terno = new Social();

terno.setCor("cinza");

terno.setManequim(44);

​

Social camisa = new Social();

camisa.setCor("verde");

camisa.setManequim(40);

​

Fardamento bata = new Fardamento();

bata.setCor("azul");

​

GuardaRoupas.addRoupa( guardaRoupas.portaDireita, camiseta );

GuardaRoupas.addRoupa( guardaRoupas.portaEsquerda, terno );

GuardaRoupas.addRoupa( guardaRoupas.portaEsquerda, camisa );

GuardaRoupas.addRoupa( guardaRoupas.gavetaDireita, bata );

​

GuardaRoupas.confereRoupas( guardaRoupas.portaDireita );

GuardaRoupas.confereRoupas ( guardaRoupas.portaEsquerda );

GuardaRoupas.confereRoupas( guardaRoupas.gavetaDireita );

GuardaRoupas.confereRoupas( guardaRoupas.gavetaEsquerda );

}

}

Listagem 11 - Classe VestuarioMain

### **Juntando as Peças**

Até agora, vimos os conceitos um a um de como se relacionam os mecanismos básicos da linguagem Java. Acredite, você já é capaz de escrever sistemas mais complexos!

Mas, sejamos realistas, saber “mexer as peças” de um tabuleiro de xadrez não nos faz um bom enxadrista. Para tanto, precisamos dominar realmente as técnicas, praticar diariamente e adquirir experiência na resolução de problemas. Iniciamos a jornada, precisamos nunca parar, pois: programar é preciso!

Vamos começar explorando nossa habilidade em interpretar problemas e projetar soluções.

Vamos praticar...

### **Atividade 03**

Resolva os problemas a seguir.

Problema 1 – Vestuário

O último problema apresentado tratava do manejo de peças de um vestuário. Sendo apenas para armazená-las em um guarda-roupas.

Vamos ampliar a situação e tratarmos o problema de forma mais ampla. Para isso, vamos organizar as ideias da seguinte forma: iremos listar todos os pontos observados do problema, em seguida, projetar as classes que irão compor o sistema, só então partiremos para a codificação. Veja a seguir o problema.

1. As roupas se dividem em categorias, sendo essas:
   * social
   * esporte
   * esporte Fino
   * trabalho
3. Todas as roupas possuem características como:
   * cor
   * número do manequim
   * tecido com o qual é feito
   * preço
5. As roupas podem se encontrar em 3 (três) situações (status) distintas:
   * no guarda-roupas
   * em uso
   * na lavanderia
7. O sistema deve nos dizer:
   * quantas roupas estão na lavanderia
   * quantas roupas estão no guarda-roupas
   * qual a cor predominante no guarda-roupas
   * adicionar e remover roupas

Dica: utilize o código apresentado durante esta aula como ponto de partida e faça as modificações necessárias. Será preciso criar uma classe Lavanderia para agrupar as roupas que não estão em uso nem no guarda-roupas.

Problema 2 – Oficina mecânica

Vamos voltar ao exemplo da nossa oficina. Lembra das competências por meio das quais as empresas organizam suas atividades?

Pois bem, nossa oficina está passando por um processo de modernização e, dentre os investimentos, resolveu encomendar um sistema que gerencie as etapas fundamentais do processo.

Vamos dividir o problema para definirmos melhor a solução, veja a seguir.

Atendimento ao cliente

A oficina precisa guardar os dados de clientes como: nome, CPF, endereço e telefone.

Manutenção

A oficina precisa conhecer bem os automóveis, claro! Todo veículo possui os seguintes dados: placa, ano, marca, modelo, numeração do chassi, tipo de combustível, proprietário (um cliente).

Almoxarifado

Suponha que a oficina tenha todas as peças de reposição necessárias à manutenção dos veículos; e que cada peça é consultada por seu nome, ano, marca e modelo do veículo.

Exemplo: “o mecânico Antônio precisa de um carburador para um Volkswagen Fox 2005”. Nosso desafio é criar todas as classes necessárias para implementar esse sistema, observando as composições, herança e polimorfismo vistos até aqui.

Boa Sorte!

### **Autoavaliação**

1. Explique de que forma(s) uma classe derivada (filha) pode modificar os valores de um atributo privado (private) de sua classe mãe.
2. Analise a solução dos problemas 1 e 2, descritos acima, e verifique onde você implementou a herança, a composição e o polimorfismo. Explique em detalhes como você fez uso de tais princípios para implementar um programa OO elegante.
3. É possível substituir a herança por uma composição? Justifique sua resposta.

## **Estudos Avançados em Java**

### **Apresentação**

Até aqui, vimos rapidamente o que o paradigma da orientação a objetos e a tecnologia Java podem nos oferecer para ajudar-nos no desenvolvimento de sistemas de software. Partimos de exemplos simples, de forma a isolar técnicas e estratégias de resolução. E fomos aplicando gradativamente cada um dos conceitos e técnicas da POO usando a linguagem Java.

Nesta última aula, traremos dicas, informações e sugestões para que você possa continuar seus estudos avançados de Java e de Orientação a Objetos. Você vai perceber que ainda há muito a aprender e ser explorado para fazer programas interessantes e sofisticados. Destacaremos as tecnologias, aplicações e roteiros de aprendizagem. Alguns pontos já foram vistos e serão melhor detalhados, outros serão apresentados agora. Queremos que você sinta-se motivado a prosseguir no estudo da tecnologia Java de desenvolvimento de sistemas. Boa aula!!!

### **Refazendo a Trilha**

Nosso passeio pelo mundo da programação orientada a objetos em Java está chegando ao fim. Mas, a viagem só está começando!

Esse é um momento oportuno para revermos o caminho que tomamos no aprendizado da programação orientada a objetos e na linguagem Java. Vamos (re)ver características da linguagem e novos caminhos que poderemos tomar para um estudo especializado e mais aprofundado da programação OO em Java.

Para utilizarmos a linguagem Java, tivemos que instalar um conjunto de programas, chamado de JDK (Java Development Kit), ou Kit de Desenvolvimento Java, que serve como a ferramenta e “bancada” principal do desenvolvedor Java.

Como vimos anteriormente, é no JDK que vem o compilador e a máquina virtual Java (responsável por executar nossos programas), além também das classes padrões que usamos para construir nossos programas.

Pois bem, vamos ver um pouco mais sobre as principais versões de distribuição de Java e suas bibliotecas de classes principais disponíveis.

1. JSE – Java Standard Edition – fornece as funcionalidades essenciais da linguagem e APIs básicas. É nela que está contida as principais classes de uso costumeiro em Java, tais como, String, System e a própria biblioteca Collection, usada nas nossas aulas. A plataforma JSE oferece muito mais classes para implementar aplicações bem mais sofisticadas, tais como bibliotecas para interfaces gráficas, acesso a arquivos, comunicação em rede, acesso a banco de dados.
2. JEE – Java Enterprise Edition – essa plataforma é usada para o desenvolvimento de aplicações corporativas e cliente-servidor. Uma das principais funcionalidades de JEE atualmente é oferecer tecnologias para o desenvolvimento de aplicações web, tais como: (i) Servlets – permite definir classes capazes de receber dados de formulários web; (ii) Java Server Pages (JSP) – possibilita a construção de páginas HTML dinâmicas; e (iii) Enterprise Java Beans (EJB) – permite definir classes que oferecem serviços distribuídos os quais contêm as regras de negócio principal da aplicação.
3. JME – Java Micro Edition – É a plataforma de classes Java voltada para os dispositivos com capacidade reduzida de processamento e armazenamento de dados, tais como aparelhos celulares, palms etc. Ela é dessa forma usada para desenvolvimento das aplicações denominadas de “embarcada”, que está presente em brinquedos, robôs, eletrodomésticos etc.

Esses 3 (três) grandes grupos de tecnologias Java envolvem uma gama de outras tecnologias, como foi o exemplo apresentado para Java Enterprise Edition. Porém, essa divisão serve para classificá-las ou organizá-las de forma estratégica de acordo com a categoria de problemas (e soluções) que iremos trabalhar.

### **Programação e Modelagem OO**

As classes são elementos fundamentais da linguagem Java, as quais definem tanto atributos (dados) quanto métodos (operações) que atuam sobre tais atributos e/ou seus parâmetros. Na programação em Java, é uma rotina criar e utilizar suas próprias classes. Em alguns casos, é comum também reusar classes que são definidas para um sistema, em um outro sistema que precise das mesmas funcionalidades.

Ao longo do curso, vimos também que está disponível para os projetistas e programadores Java a criação e a utilização de interfaces, que são declarações de métodos e atributos, os quais obrigam as classes que as implementam a definir a sua implementação para esses métodos. Fazer uso de interfaces não é trivial e requer um grau de maturidade considerável no projeto de sistemas.

#### **Criando um Mapa para Guiar os Estudos**

Partindo da visão de que a imensa maioria dos desenvolvedores Java tende a programar sistemas Web ou desktop, vamos construir um modelo que irá indicar os caminhos por onde o programador iniciante Java terá que passar para poder explorar com propriedade os recursos que essa linguagem oferece. O fluxo abaixo indica sugestões para o estudo da programação OO com a linguagem Java.

Figura 01 - Quadro geral de tecnologias Java

Depois de dominar os conhecimentos essenciais deste curso, o próximo passo é escolher um caminho e seguir em frente. Dentre eles, veremos dois caminhos comuns a serem seguidos: (i) desenvolvimento de aplicações para desktop; e (ii) desenvolvimento de aplicações web.

### **Programação de Aplicações Desktop**

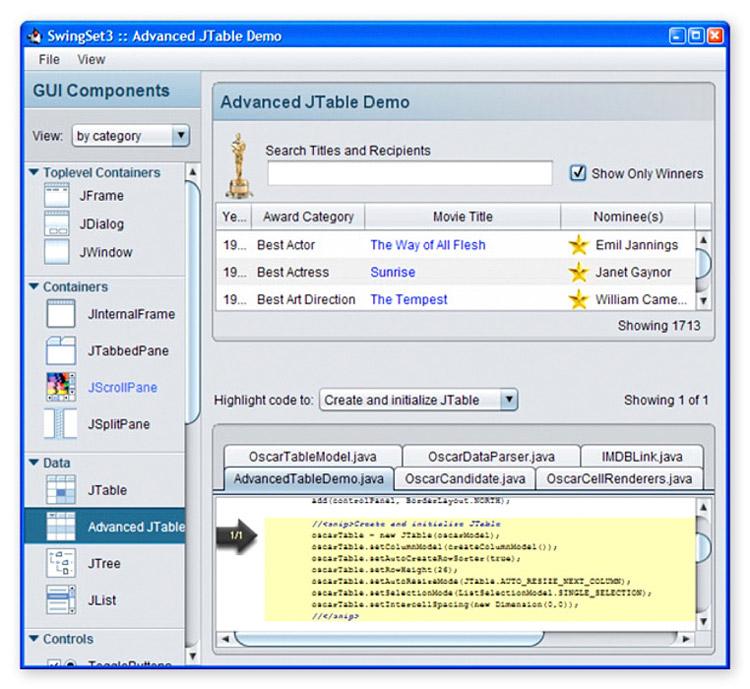
NoJava Standard Edition, a biblioteca (API – Application Programming Interface) fundamental para desenvolvimento de interfaces gráficas (botões, janelas, janelas de diálogos), é conhecida como Swing.

Lá, encontraremos tudo que precisamos para o desenho (design) de telas gráficas com bastante recursos, como arrastar-e-soltar, e a possibilidade de incorporar características multimídia para reprodução de vídeos e áudio. Vejamos um exemplo da própria Sun presente no JSE no seguinte diretório:

<diretório de instalação do jdk>/demo/jfc/SwingSet3/readme.html

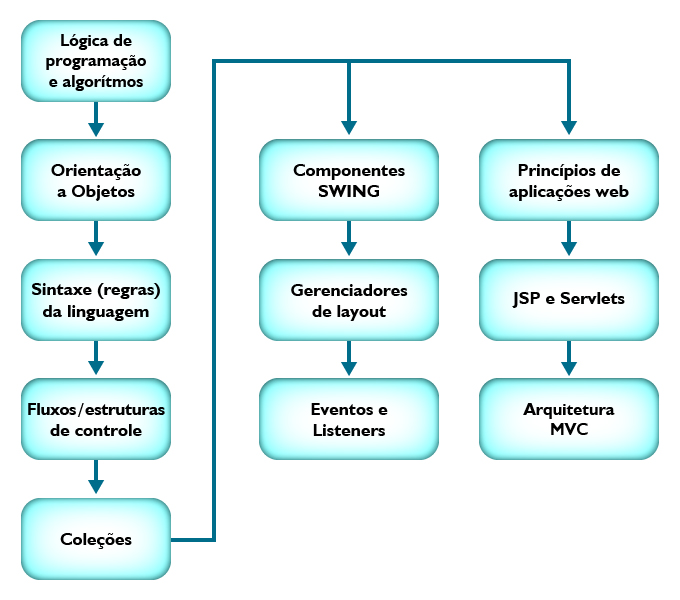
Lendo esse documento/página HTML, você terá a opção de executar a aplicação com a interface apresentada na Figura 3. Nela, podemos observar todos os componentes e seus comportamentos.

Figura 03 - Exemplo de interface Swing da Sun



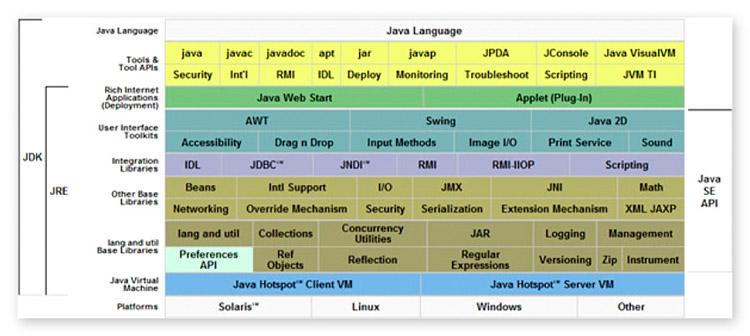
A API Swing é extensa e compreende os 18 (dezoito) pacotes, os quais são listados a seguir.

|  |  |  |
| --- | --- | --- |
| javax.accessibility | javax.swing.plaf | javax.swing.text |
| javax.swing | javax.swing.plaf.basic | javax.swing.text.html |
| javax.swing.border | javax.swing.plaf.metal | javax.swing.text.html.parser |
| javax.swing.colorchooser | javax.swing.plaf.multi | javax.swing.text.rtf |
| javax.swing.event | javax.swing.plaf.synth | javax.swing.tree |
| javax.swing.filechooser | javax.swing.table | javax.swing.undo |



Mais à frente, serão indicadas fontes de pesquisa onde será possível encontrar materiais sobre todas as tecnologias mencionadas. O quadro da Figura 1 indica agrupamentos de tecnologias Java com os mesmos fins, porém, poderemos visualizar em detalhes o que a plataforma Java Standard Edition (JSE) pode nos oferecer, observando a Figura 2. Esse quadro foi retirado do site oficial da Sun Microsystems e compõe sua documentação.

Figura 02 - Quadro detalhado de tecnologias Java Standard Edition



### **Programação de Aplicações Desktop**

#### **Tratando Eventos**

É preciso um considerável aprendizado até estar apto a programar utilizando as classes dos pacotes javax.swing e java.awt, pois elas utilizam conceitos próprios de disparo e captura de eventos.

Os eventos são utilizados para representar ações realizadas pelo usuário, sistema operacional ou temporizadores, ou seja, entidades externas ou internas ao sistema capazes de indicar um fenômeno que modifique estados dentro da aplicação. Cada ação do usuário pressupõe um Evento, e do outro lado, é preciso existir uma entidade capaz de “ouvir” esses eventos e realizar as ações necessárias. Esses são os listeners, cadastrados para receberem determinados eventos.

#### **Gerenciando Layout**

A construção de uma interface gráfica que traga conforto ao usuário e que obedeça a critérios de usabilidade precisa ter seu design bem planejado e um layout de componentes gráficos que tenha um comportamento estável quando a janela for ocultada, movida e redimensionada.

A linguagem Java possui classes especiais, denominadas de classes de Layout, voltadas à manutenção das posições e dimensões dos componentes visíveis e de componentes que servem como organizadores ou containers de outros componentes.

#### **Banco de Dados**

O acesso a banco de dados em Java é feito pelas classes do pacote java.sql, que compõem a API JDBC (Java Database Connectivity). Nela, encontramos classes para estabelecimento de conexão com um banco de dados, execução de comandos ao banco de dados (escritos em uma linguagem padrão chamada de SQL), gerenciamento de transações, e outras funcionalidades. O estudo da API JDBC pressupõe conhecimentos básicos de banco de dados e linguagem SQL. Porém, dominar essa API leva poucas horas de estudo.

### **Programação Web**

Atualmente, a tendência aponta para um mundo de sistemas corporativos integrados a outras redes e operando online em várias transações com sistemas de clientes, fornecedores e governos, onde seus usuários estão distribuídos geograficamente. Nesse contexto, os sistemas web se consolidaram ao longo das últimas duas décadas, e se estabeleceram como uma plataforma padrão.

A plataforma de sistemas web permite que seus clientes se mantenham atualizados automaticamente e, com custo mínimo, pois não necessitam realizar downloads e configurações constantemente a cada nova versão de softwares disponibilizados.

Mas, como tudo tem um lado negativo, a web carece de padronizações. Atualmente, órgãos como W3C padronizam, permitindo que provedores de ferramentas tenham um terreno menos movediço para desenvolver tecnologias que facilitem o desenvolvimento de sistemas web. Ações como essa não corrigem todas as discrepâncias geradas ao longo de anos por fabricantes de browsers e pequenos protocolos proprietários. Logo, desenvolver para a web é estar “costurando” sempre uma colcha de retalhos tecnológicos.

Desafios Resolvidos

iar um programa em linguagem java Crie um programa onde você receberá valores inteiros não negativos como entrada. Ordene estes valores de acordo com o seguinte critério: Primeiro os Pares Depois os Ímpares Você deve exibir os pares em ordem crescente e na sequência os ímpares em ordem decrescente. Entrada A primeira linha de entrada contém um único inteiro positivo N (1 < N < 10000) Este é o número de linhas de entrada que vem logo a seguir. As próximas N linhas terão, cada uma delas, um valor inteiro não negativo. Saída Exiba todos os valores lidos na entrada segundo a ordem apresentada acima. Cada número deve ser impresso em uma linha,

Claro, aqui está uma possível solução em Java para o problema descrito:

java

import java.util.\*;

public class OrdenaParesImpares {

public static void main(String[] args) {

Scanner sc = new Scanner(System.in);

int n = sc.nextInt();

List<Integer> pares = new ArrayList<Integer>();

List<Integer> impares = new ArrayList<Integer>();

for (int i = 0; i < n; i++) {

int x = sc.nextInt();

if (x % 2 == 0) {

pares.add(x);

} else {

impares.add(x);

}

}

Collections.sort(pares);

Collections.sort(impares, Collections.reverseOrder());

for (int x : pares) {

System.out.println(x);

}

for (int x : impares) {

System.out.println(x);

}

sc.close();

}

}

Explicação:

* O programa começa lendo um número inteiro n da entrada, que indica quantos valores serão lidos em seguida.
* Em seguida, o programa cria duas listas vazias: uma para os valores pares e outra para os valores ímpares.
* O programa faz um loop for para ler os n valores da entrada e classificá-los como pares ou ímpares, adicionando-os à lista correspondente.
* Poderia ser utilizado o comando System.out.println(pares). Porém a atividade pede cada numeros em linhas separadas
* Depois, o programa ordena a lista de valores pares em ordem crescente, usando o método Collections.sort(), que ordena a lista em seu lugar.
* Da mesma forma, o programa ordena a lista de valores ímpares em ordem decrescente, usando o método Collections.sort() em conjunto com o método Collections.reverseOrder().
* Por fim, o programa imprime os valores pares em ordem crescente, seguidos pelos valores ímpares em ordem decrescente, um por linha, usando um loop for-each.

Espero que isso ajude!

Quais as principais caracteristicas da coleção List em java?

Em Java, a coleção List é uma interface que define uma sequência ordenada de elementos, onde cada elemento pode ter uma posição indexada. Algumas das principais características da coleção List são:

* Permite a inserção de elementos em qualquer posição da lista, através dos métodos add(int index, E element) e addAll(int index, Collection<? extends E> c).
* Permite a remoção de elementos de qualquer posição da lista, através do método remove(int index).
* Permite a substituição de elementos em qualquer posição da lista, através do método set(int index, E element).
* Permite a obtenção de um elemento em uma posição específica da lista, através do método get(int index).
* Permite a busca de um elemento na lista, através do método indexOf(Object o), que retorna o índice da primeira ocorrência do elemento na lista.
* Permite verificar se a lista contém um elemento específico, através do método contains(Object o).
* Permite obter o tamanho da lista, através do método size().
* Permite obter uma "sublista" da lista original, contendo elementos em um determinado intervalo de posições, através do método subList(int fromIndex, int toIndex).
* Permite verificar se a lista está vazia, através do método isEmpty().
* Permite a iteração sobre os elementos da lista, utilizando um loop for-each ou um iterator.
* Permite a ordenação dos elementos da lista, através do método sort(Comparator<? super E> c).

A coleção List pode ser implementada por diversas classes em Java, como ArrayList, LinkedList, Vector e outras, cada uma com suas próprias características de desempenho e uso adequado em diferentes situações. Em geral, a escolha da classe List mais adequada dependerá das necessidades específicas de cada programa ou aplicação.

Desafio Resolvido

**Pedro trabalha sempre até tarde todos os dias, com isso tem pouco tempo tempo para as tarefas domésticas. Para economizar tempo ele faz a lista de compras do supermercado em um aplicativo e costuma anotar cada item na mesma hora que percebe a falta dele em casa. O problema é que o aplicativo não exclui itens duplicados, como Pedro anota o mesmo item mais de uma vez e a lista acaba ficando extensa. Sua tarefa é melhorar o aplicativo de notas desenvolvendo um código que exclua os itens duplicados da lista de compras e que os ordene alfabeticamente. Entrada A primeira linha de entrada contém um inteiro N (N < 100) com a quantidade de casos de teste que vem a seguir, ou melhor, a quantidade de listas de compras para organizar. Cada lista de compra consiste de uma única linha que contém de 1 a 1000 itens ou palavras compostas apenas de letras minúsculas (de 1 a 20 letras), sem acentos e separadas por um espaço. Saída A saída contém N linhas, cada uma representando uma lista de compra, sem os itens repetidos e em ordem alfabética.**

**Claro! Aqui está uma sugestão de solução em Java:**

**java**

**import java.util.Arrays;**

**import java.util.HashSet;**

**import java.util.Scanner;**

**import java.util.Set;**

**public class ListaDeCompras {**

**public static void main(String[] args) {**

**Scanner sc = new Scanner(System.in);**

**int n = sc.nextInt();**

**sc.nextLine(); // Consumir o \n pendente**

**for (int i = 0; i < n; i++) {**

**String[] lista = sc.nextLine().split(" ");**

**Set<String> set = new HashSet<>(Arrays.asList(lista));**

**String[] listaSemDuplicatas = set.toArray(new String[0]);**

**Arrays.sort(listaSemDuplicatas);**

**for (String item : listaSemDuplicatas) {**

**System.out.print(item + " ");**

**}**

**System.out.println();**

**}**

**sc.close();**

**}**

**}**

**Explicando o código:**

1. **Fazemos a leitura do número de casos de teste e consumimos o \n pendente com sc.nextLine().**
2. **Dentro do loop para cada caso de teste, lemos a lista de compras como uma única string e a separamos em um array de strings utilizando o método split(" ").**
3. **Criamos um objeto Set<String> a partir do array de strings para remover as duplicatas. Como um Set não permite elementos duplicados, isso é feito automaticamente.**
4. **Convertemos o Set de volta para um array de strings utilizando o método toArray(new String[0]).**
5. **Ordenamos o array de strings em ordem alfabética utilizando o método Arrays.sort().**
6. **Imprimimos os elementos do array ordenado sem duplicatas separados por um espaço e adicionamos uma quebra de linha ao final.**

**Espero ter ajudado!**



Manoel Titto

**Qual coleção em Java não permite entrar com valores repetidos?**

**A coleção em Java que não permite a inserção de valores repetidos é a coleção Set. Isso ocorre porque um Set é uma coleção que não permite elementos duplicados. Caso você tente inserir um elemento que já exista no Set, ele simplesmente será ignorado e a operação de inserção não terá efeito.**

**Existem algumas implementações de Set em Java, como o HashSet e o TreeSet, cada uma com suas particularidades. O HashSet é uma implementação baseada em tabela de hash, enquanto o TreeSet é uma implementação baseada em árvore que mantém seus elementos em ordem natural (ou utilizando um comparador definido pelo programador).**



Manoel Titto

**Você consegue simpleificar o codigo: public class ListaDeCompras**

**Claro! Aqui está uma versão mais simples do código que utiliza apenas um objeto TreeSet para remover as duplicatas e ordenar os elementos:**

**java**

**import java.util.Scanner;**

**import java.util.Set;**

**import java.util.TreeSet;**

**public class ListaDeCompras {**

**public static void main(String[] args) {**

**Scanner sc = new Scanner(System.in);**

**int n = sc.nextInt();**

**sc.nextLine();**

**for (int i = 0; i < n; i++) {**

**String[] lista = sc.nextLine().split(" ");**

**Set<String> set = new TreeSet<>(Arrays.asList(lista));**

**System.out.println(String.join(" ", set));**

**}**

**sc.close();**

**}**

**}**

**Explicando as mudanças:**

1. **Utilizamos um objeto TreeSet para armazenar os elementos únicos da lista e garantir que eles estejam ordenados. A classe TreeSet implementa a interface SortedSet, o que significa que ela mantém seus elementos em ordem natural (ou utilizando um comparador definido pelo programador).**
2. **Utilizamos o método String.join() para imprimir os elementos do TreeSet separados por um espaço.**

**Espero que esta versão simplificada ajude!**



Manoel Titto

**apresente em Linguagem Java uma forma de criar uma lista de compras, onde o usuário irá inserir o s itens e depois o algoritmo imprime na tela em os itens não repetidos. O use set**

**Claro, segue um exemplo de código em Java que utiliza um objeto Set para criar uma lista de compras e remover itens repetidos:**

**java**

**import java.util.HashSet;**

**import java.util.Scanner;**

**import java.util.Set;**

**public class ListaDeCompras {**

**public static void main(String[] args) {**

**Scanner sc = new Scanner(System.in);**

**Set<String> listaDeCompras = new HashSet<>();**

**int n;**

**System.out.println("Quantos itens deseja inserir na lista?");**

**n = sc.nextInt();**

**sc.nextLine();**

**for (int i = 1; i <= n; i++) {**

**System.out.println("Insira o item " + i + ":");**

**String item = sc.nextLine();**

**listaDeCompras.add(item);**

**}**

**System.out.println("Lista de compras sem itens repetidos:");**

**for (String item : listaDeCompras) {**

**System.out.println(item);**

**}**

**sc.close();**

**}**

**}**

**Nesse exemplo, utilizamos um objeto HashSet para armazenar a lista de compras. O HashSet é uma implementação de Set que não permite elementos duplicados.**

**O programa começa pedindo ao usuário o número de itens que deseja inserir na lista. Em seguida, entra em um loop que solicita que o usuário insira cada item e adiciona esse item à lista usando o método add() do objeto HashSet.**

**Por fim, o programa imprime na tela a lista de compras sem itens repetidos.**

**Solução Simplificada**

import java.util.Scanner;

import java.util.\*;

public class ListaDeCompras{

public static void main(String[] args) {

Scanner teclado = new Scanner(System.in);

Set<String> listaDeCompras = new HashSet<>();

System.out.println("Quantidade de Itens para lista");

int n = teclado.nextInt();

teclado.nextLine();

for (int i = 1; i <= n; i++) {

System.out.println("Insira o item ");

String compras = teclado.nextLine();

listaDeCompras.add(compras);

}

for (String item : listaDeCompras) {

System.out.println(item);

}

}

}

**Solução Simplificada Melhorada**

import java.util.Scanner;

import java.util.\*;

public class ListaDeCompras{

public static void main(String[] args) {

Scanner teclado = new Scanner(System.in);

Set<String> listaDeCompras = new HashSet<>();

int n;

do{

System.out.println("Quantidade de Itens para lista <100");

n = teclado.nextInt();

teclado.nextLine();

}while(n>=100); //controle de quantidade de itens<100

for (int i = 1; i <= n; i++) {

String compras = teclado.nextLine();

listaDeCompras.add(compras);

}

for (String item : listaDeCompras) {

System.out.println(item);

}

}

}

**Solução Perfeita Lista de Compras com comando de ordenamento**

import java.util.Scanner;

import java.util.\*;

public class ListaDeCompras{

public static void main(String[] args) {

Scanner teclado = new Scanner(System.in);

Set<String> listaDeCompras = new HashSet<>();

List<String> novaLista = new ArrayList<>();

int n;

do{

System.out.println("Quantidade de Itens para lista <100");

n = teclado.nextInt();

teclado.nextLine();

}while(n>=100);

for (int i = 1; i <= n; i++) {

String compras = teclado.nextLine();

listaDeCompras.add(compras);

}

novaLista.addAll(listaDeCompras);

Collections.sort(novaLista);

System.out.println(novaLista);

for( String item: novaLista)

System.out.println(item);

}

}

Solução do GPTchat para o desafio

mport java.util.\*;

public class ListaDeCompras {

public static void main(String[] args) {

Scanner scanner = new Scanner(System.in);

int n = scanner.nextInt();

scanner.nextLine(); // consumir o \n após o número inteiro

for (int i = 0; i < n; i++) {

String[] listaDeCompras = scanner.nextLine().split(" ");

Set<String> itensUnicos = new TreeSet<>(Arrays.asList(listaDeCompras));

for (String item : itensUnicos) {

System.out.print(item + " ");

}

System.out.println();

}

scanner.close();

}

}

O programa recebe um inteiro n que representa a quantidade de listas de compras a serem organizadas. Em seguida, o programa lê cada lista de compras como uma única linha, que contém uma série de itens separados por espaços.

O programa utiliza o método split da classe String para separar os itens da lista e, em seguida, cria um TreeSet a partir do array de itens utilizando o método asList da classe Arrays. O TreeSet é uma coleção que armazena elementos únicos em ordem crescente, atendendo aos requisitos do problema.

Por fim, o programa imprime na tela os itens únicos da lista de compras em ordem alfabética, separados por espaços. O loop externo é responsável por ler e processar cada lista de compras de entrada.

A coleção TreeSet em Java é uma implementação de um conjunto ordenado, que armazena elementos em uma árvore binária de pesquisa, mantendo-os ordenados em ordem crescente ou decrescente.

Para criar uma lista de compras usando o TreeSet, podemos fazer o seguinte:

1. Importar a classe TreeSet:

java

import java.util.TreeSet;

1. Criar um objeto TreeSet para armazenar os itens da lista de compras:

java

TreeSet<String> listaDeCompras = new TreeSet<>();

1. Adicionar itens à lista de compras usando o método add():

java

listaDeCompras.add("arroz");

listaDeCompras.add("feijão");

listaDeCompras.add("macarrão");

listaDeCompras.add("arroz");

listaDeCompras.add("feijão");

Observe que a lista de compras não permite elementos duplicados, portanto, quando tentamos adicionar "arroz" e "feijão" novamente, eles não são adicionados.

1. Para exibir os itens da lista de compras em ordem alfabética, podemos percorrer a coleção usando um loop for-each:

java

for (String item : listaDeCompras) {

System.out.println(item);

}

Este código exibirá os itens da lista de compras em ordem alfabética e sem itens duplicados. O resultado será:

arroz

feijão

macarrão

**Entrada de Dados em Java**

m Java, existem várias formas de entrada de dados que podem ser utilizadas de acordo com a necessidade de cada situação. Entre os métodos mais comuns, podemos destacar:

1. Scanner: É uma das formas mais populares de entrada de dados em Java. Através da classe Scanner, é possível ler dados de diversos tipos, como inteiros, decimais, strings e caracteres, a partir de diferentes fontes, como teclado e arquivos.

Exemplo:

scss

Scanner sc = new Scanner(System.in);

int valorInteiro = sc.nextInt();

String valorString = sc.next();

double valorDecimal = sc.nextDouble();

1. BufferedReader: É uma classe que permite a leitura de dados a partir de um objeto Reader, que pode ser utilizado para ler dados de arquivos e outros tipos de entrada de dados. É útil para leitura de grandes volumes de dados.

Exemplo:

java

BufferedReader br = new BufferedReader(new InputStreamReader(System.in));

String valorString = br.readLine();

int valorInteiro = Integer.parseInt(br.readLine());

1. Console: É uma classe que permite a leitura de dados a partir do console do sistema. É útil para programas que precisam ser executados na linha de comando.

Exemplo:

scss

Console console = System.console();

String valorString = console.readLine();

char[] senha = console.readPassword();

1. JOptionPane: É uma classe que permite a criação de caixas de diálogo para entrada de dados. É útil para programas que possuem interface gráfica com o usuário.

Exemplo:

javascript

String valorString = JOptionPane.showInputDialog("Digite um valor:");

Cada método de entrada de dados possui suas vantagens e desvantagens, e cabe ao programador escolher o mais adequado de acordo com a necessidade de cada situação**.**

**Coleções em Java**

m Java, coleções são estruturas de dados que permitem armazenar e manipular grupos de objetos. Elas são amplamente utilizadas para gerenciar dados em muitos tipos de aplicações.

Existem várias classes de coleções em Java, cada uma com suas próprias características e propósitos. As principais classes de coleções são:

* List: armazena uma lista de elementos ordenados, permitindo elementos repetidos. As implementações mais comuns são ArrayList e LinkedList.
* Set: armazena um conjunto de elementos únicos e não ordenados. As implementações mais comuns são HashSet e TreeSet.
* Queue: armazena uma fila de elementos, permitindo adicionar e remover elementos em uma extremidade. As implementações mais comuns são LinkedList e PriorityQueue.
* Map: armazena uma coleção de pares chave-valor, permitindo acesso aos valores através das chaves. As implementações mais comuns são HashMap e TreeMap.

Para utilizar coleções em Java, é necessário importar as classes relevantes e instanciá-las no código. Depois disso, é possível adicionar, remover, buscar e manipular elementos dentro das coleções, de acordo com os métodos disponíveis em cada classe.

As coleções em Java são uma parte fundamental da linguagem e são usadas em muitos tipos de aplicações, desde sistemas de gerenciamento de banco de dados até jogos e aplicativos de desktop. Compreender como as coleções funcionam é essencial para se tornar um programador Java eficiente e capaz de criar aplicações robustas e escaláveis.

aqui está um exemplo de programa em Java usando Maps:

csharp

import java.util.\*;

public class ExemploMapa {

public static void main(String[] args) {

// Cria um mapa para armazenar a relação entre frutas e suas quantidades

Map<String, Integer> frutas = new HashMap<>();

// Adiciona algumas frutas e suas quantidades ao mapa

frutas.put("Maçã", 3);

frutas.put("Banana", 2);

frutas.put("Laranja", 5);

frutas.put("Abacaxi", 1);

// Imprime a lista de frutas e suas quantidades

System.out.println("Lista de frutas e suas quantidades:");

for (String fruta : frutas.keySet()) {

int quantidade = frutas.get(fruta);

System.out.println(fruta + ": " + quantidade);

}

// Altera a quantidade de uma fruta existente no mapa

frutas.put("Laranja", 10);

// Remove uma fruta do mapa

frutas.remove("Abacaxi");

// Imprime a lista atualizada de frutas e suas quantidades

System.out.println("Lista atualizada de frutas e suas quantidades:");

for (String fruta : frutas.keySet()) {

int quantidade = frutas.get(fruta);

System.out.println(fruta + ": " + quantidade);

}

}

}

Neste exemplo, utilizamos um mapa (Map) para armazenar a relação entre frutas e suas quantidades. O mapa é implementado como um HashMap, que é uma das implementações da interface Map em Java.

Primeiro, adicionamos algumas frutas e suas quantidades ao mapa utilizando o método put(). Em seguida, imprimimos a lista de frutas e suas quantidades utilizando um laço for. Note que utilizamos o método keySet() para obter um conjunto de todas as chaves (frutas) do mapa, e o método get() para obter a quantidade correspondente a cada chave.

Depois, alteramos a quantidade de uma fruta existente no mapa utilizando novamente o método put(). Também removemos uma fruta do mapa utilizando o método remove().

Por fim, imprimimos a lista atualizada de frutas e suas quantidades utilizando novamente um laço for.

Os Maps são uma coleção útil em Java para armazenar relações chave-valor, onde as chaves são únicas. Existem várias implementações da interface Map em Java, cada uma com suas próprias características e usos. É importante escolher a implementação certa para cada caso de uso específico.

Segue abaixo um exemplo simples de como criar uma agenda telefônica utilizando a coleção Map em Java:

java

import java.util.HashMap;

import java.util.Map;

import java.util.Scanner;

public class AgendaTelefonica {

public static void main(String[] args) {

Scanner scanner = new Scanner(System.in);

Map<String, String> agenda = new HashMap<>();

// Laço para inserir contatos na agenda

while (true) {

System.out.println("Digite o nome do contato (ou 'fim' para encerrar):");

String nome = scanner.nextLine();

if (nome.equals("fim")) {

break;

}

System.out.println("Digite o número do telefone:");

String telefone = scanner.nextLine();

agenda.put(nome, telefone);

}

// Laço para buscar contatos na agenda

while (true) {

System.out.println("Digite o nome do contato para buscar (ou 'fim' para encerrar):");

String nome = scanner.nextLine();

if (nome.equals("fim")) {

break;

}

if (agenda.containsKey(nome)) {

System.out.println("O telefone de " + nome + " é: " + agenda.get(nome));

} else {

System.out.println("Contato não encontrado.");

}

}

}

}

Nesse exemplo, utilizamos a classe HashMap para implementar a agenda telefônica, onde cada contato é representado por uma chave (nome) e um valor (número de telefone). Primeiro, utilizamos um laço para permitir que o usuário insira novos contatos na agenda, e depois outro laço para buscar contatos já existentes.

Na parte de inserção de contatos, a cada iteração do laço, o programa solicita o nome e o telefone do contato ao usuário, e adiciona esses valores ao mapa usando o método put(nome, telefone).

Na parte de busca de contatos, a cada iteração do laço, o programa solicita o nome do contato que se deseja buscar. Em seguida, utiliza o método containsKey(nome) para verificar se o nome do contato está presente no mapa, e caso esteja, utiliza o método get(nome) para obter o número de telefone correspondente ao nome. Se o nome não for encontrado, o programa exibe uma mensagem informando que o contato não foi encontrado.

Esse é apenas um exemplo simples de como utilizar a coleção Map em Java para implementar uma agenda telefônica, mas existem diversas outras maneiras de implementar a mesma funcionalidade, e a escolha da melhor abordagem pode depender de diversos fatores, como o tamanho da agenda e a necessidade de manter a ordem dos contatos.

Aqui vai um exemplo simples de como utilizar a coleção Queue em Java:

java

import java.util.LinkedList;

import java.util.Queue;

public class ExemploQueue {

public static void main(String[] args) {

// Cria uma fila vazia

Queue<String> fila = new LinkedList<>();

// Adiciona elementos na fila

fila.offer("João");

fila.offer("Maria");

fila.offer("Pedro");

// Mostra o tamanho da fila

System.out.println("Tamanho da fila: " + fila.size());

// Mostra o elemento que está na frente da fila

System.out.println("Próximo elemento da fila: " + fila.peek());

// Remove o elemento que está na frente da fila

System.out.println("Elemento removido da fila: " + fila.poll());

// Mostra o tamanho da fila novamente

System.out.println("Tamanho da fila: " + fila.size());

}

}

Neste exemplo, criamos uma fila (Queue) utilizando a implementação LinkedList da interface Queue. Adicionamos três elementos à fila utilizando o método offer(). Em seguida, mostramos o tamanho da fila utilizando o método size() e o elemento que está na frente da fila utilizando o método peek(). Depois, removemos o elemento que está na frente da fila utilizando o método poll() e mostramos o tamanho da fila novamente.

DESAFIO

Você pode organizar este código em Jva para que ele atenda ao desafio Desafio O professor Girafales organizou a confecção de um uniforme para as turmas da escola para comemorar o final do ano. Após algumas conversas, ficou decidido com os alunos que eles poderiam escolher a cor do uniforme entre branco ou vermelho. Assim sendo, Girafales precisa de sua ajuda para organizar as listas de quem quer o uniforme em cada uma das turmas, relacionando estas camisetas pela cor, tamanho (P, M ou G) e por último pelo nome. Entrada Cada caso de teste inicia com um valor N, (1 ≤ N ≤ 60) inteiro e positivo, que indica a quantidade de uniformes a serem feitas para aquela turma. As próximas N\*2 linhas contém informações de cada um dos uniformes (serão duas linhas de informação para cada uniforme). A primeira linha irá conter o nome do estudante e a segunda linha irá conter a cor do uniforme ("branco" ou "vermelho") seguido por um espaço e pelo tamanho do uniforme "P" "M" ou "G". A entrada termina quando o valor de N for igual a zero (0) e esta valor não deverá ser processado. Saída Para cada caso de entrada deverão ser impressas as informações ordenadas pela cor em ordem ascendente, seguido pelos tamanhos em ordem descendente e por último por ordem ascendente de nome,

Segue o código organizado em Java para atender ao desafio:

java

import java.util.\*;

public class Uniformes {

public static void main(String[] args) {

Scanner sc = new Scanner(System.in);

while (true) {

int n = sc.nextInt();

sc.nextLine();

if (n == 0) {

break;

}

Map<String, List<Uniforme>> uniformes = new TreeMap<>();

for (int i = 0; i < n; i++) {

String nome = sc.nextLine();

String[] dadosUniforme = sc.nextLine().split(" ");

String cor = dadosUniforme[0];

String tamanho = dadosUniforme[1];

if (!uniformes.containsKey(cor)) {

uniformes.put(cor, new ArrayList<>());

}

List<Uniforme> listaUniformes = uniformes.get(cor);

listaUniformes.add(new Uniforme(nome, cor, tamanho));

uniformes.put(cor, listaUniformes);

}

for (Map.Entry<String, List<Uniforme>> entry : uniformes.entrySet()) {

String cor = entry.getKey();

List<Uniforme> listaUniformes = entry.getValue();

Collections.sort(listaUniformes);

for (Uniforme u : listaUniformes) {

System.out.println(cor + " " + u.toString());

}

}

}

}

}

class Uniforme implements Comparable<Uniforme> {

private String nome;

private String cor;

private String tamanho;

public Uniforme(String nome, String cor, String tamanho) {

this.nome = nome;

this.cor = cor;

this.tamanho = tamanho;

}

public String getNome() {

return nome;

}

public String getCor() {

return cor;

}

public String getTamanho() {

return tamanho;

}

@Override

public int compareTo(Uniforme o) {

if (this.tamanho.equals(o.tamanho)) {

return this.nome.compareTo(o.nome);

}

if (this.tamanho.equals("P")) {

return -1;

} else if (this.tamanho.equals("G")) {

return 1;

}

return 0;

}

@Override

public String toString() {

return this.tamanho + " " + this.nome;

}

}

Esse programa utiliza a coleção Map para armazenar os uniformes das turmas. Cada cor é uma chave no mapa e o valor correspondente é uma lista de objetos Uniforme.

Durante a entrada de dados, o programa cria objetos Uniforme e adiciona-os na lista correspondente à sua cor.

Após a entrada de todos os uniformes, o programa itera sobre as cores do mapa e para cada cor, ordena a lista de Uniformes por tamanho e nome e imprime na tela a cor e os dados dos uniformes.

Solução GPTChat

import java.io.BufferedReader;

import java.io.IOException;

import java.io.InputStreamReader;

import java.io.PrintWriter;

import java.util.Arrays;

import java.util.Comparator;

public class Uniforms {

static BufferedReader in = new BufferedReader(new InputStreamReader(System.in));

static PrintWriter out = new PrintWriter(System.out);

public static void main(String[] args) throws IOException {

String owner, l, color, size;

String[] details;

Tshirt[] tshirts;

Tshirt tshirt;

int N;

boolean firstCase = true;

while ((N = Integer.parseInt(in.readLine())) != 0) {

if (!firstCase) {

out.println();

}

firstCase = false;

tshirts = new Tshirt[N];

for (int i = 0; i < N; i++) {

owner = in.readLine();

details = in.readLine().split(" ");

color = details[0];

size = details[1];

tshirt = new Tshirt(size, color, owner);

tshirts[i] = tshirt;

}

Arrays.sort(tshirts, new Comparator<Tshirt>() {

@Override

public int compare(Tshirt t1, Tshirt t2) {

String color1 = t1.getColor();

String color2 = t2.getColor();

int diff = color1.compareTo(color2);

if (diff != 0) {

return diff;

} else {

String size1 = t1.getSize();

String size2 = t2.getSize();

if (size1.equals(size2)) {

String owner1 = t1.getOwner();

String owner2 = t2.getOwner();

return owner1.compareTo(owner2);

}

if (size1.equals("P")) {

return 1;

}

if (size1.equals("M") && size2.equals("G")) {

return -1;

}

return -1;

}

}

});

for (int i = 0; i < N; i++) {

out.println(tshirts[i]);

}

}

out.close();

}

}

class Tshirt {

private String size;

private String color;

private String owner;

public Tshirt(String size, String color, String owner) {

this.size = size;

this.color = color;

this.owner = owner;

}

public String getSize() {

return size;

}

public void setSize(String size) {

this.size = size;

}

public String getColor() {

return color;

}

public void setColor(String color) {

this.color = color;

}

public String getOwner() {

return owner;

}

public void setOwner(String owner) {

this.owner = owner;

}

@Override

public String toString() {

return this.getColor() + " " + this.getSize() + " " + this.getOwner();

}

}

As linhas adicionadas ao código acima lêem a entrada de dados e criam instâncias de Tshirt. Também ajustamos a condição do loop while para usar N ao invés de l. Além disso, incluímos um bloco if no início do loop para imprimir uma linha vazia

Melhorando o Código do Desafio

import java.io.BufferedReader;

import java.io.IOException;

import java.io.InputStreamReader;

import java.io.PrintWriter;

import java.util.Arrays;

import java.util.Comparator;

public class Uniforms {

static BufferedReader in = new BufferedReader(new InputStreamReader(System.in));

static PrintWriter out = new PrintWriter(System.out);

public static void main(String[] args) throws IOException {

String owner, l, color, size;

String[] details;

Tshirt[] tshirts;

Tshirt tshirt;

int N;

boolean firstCase = true;

while ((N = Integer.parseInt(in.readLine())) != 0) {

if (!firstCase) {

out.println();

}

firstCase = false;

tshirts = new Tshirt[N];

for (int i = 0; i < N; i++) {

owner = in.readLine();

details = in.readLine().split(" ");

color = details[0];

size = details[1];

tshirt = new Tshirt(size, color, owner);

tshirts[i] = tshirt;

}

Arrays.sort(tshirts, new Comparator<Tshirt>() {

@Override

public int compare(Tshirt t1, Tshirt t2) {

String color1 = t1.getColor();

String color2 = t2.getColor();

int diff = color1.compareTo(color2);

if (diff != 0) {

return diff;

} else {

String size1 = t1.getSize();

String size2 = t2.getSize();

int sizeDiff = size2.compareTo(size1);

if (sizeDiff != 0) {

return sizeDiff;

} else {

String owner1 = t1.getOwner();

String owner2 = t2.getOwner();

return owner1.compareTo(owner2);

}

}

}

});

for (int i = 0; i < N; i++) {

out.println(tshirts[i]);

}

}

out.close();

}

}

class Tshirt {

private String size;

private String color;

private String owner;

public Tshirt(String size, String color, String owner) {

this.size = size;

this.color = color;

this.owner = owner;

}

public String getSize() {

return size;

}

public void setSize(String size) {

this.size = size;

}

public String getColor() {

return color;

}

public void setColor(String color) {

this.color = color;

}

public String getOwner() {

return owner;

}

public void setOwner(String owner) {

this.owner = owner;

}

@Override

public String toString() {

return this.getColor() + " " + this.getSize() + " " + this.getOwner();

}

}

A principal mudança foi no método compare() do Comparator, que agora ordena primeiro pela cor em ordem ascendente, depois pelos tamanhos em ordem descendente e por último por ordem ascendente de nome. Note que o Comparator agora compara os tamanhos de forma diferente do código original. Além disso, o código agora imprime uma linha em branco entre os casos de teste.

Novo Desafio Solução ChatGPT

Este código deverá cumprir as seguintes regas: A primeira linha contém um inteiro N, indicando o número de casos de teste a seguir. Cada caso de teste inicia com um inteiro M (1 ≤ M ≤ 1000), indicando o número de clientes. Em seguida haverá M inteiros distintos Pi (1 ≤ Pi ≤ 1000), onde o i-ésimo inteiro indica o número recebido via sms do i-ésimo cliente. Os inteiros acima são dados em ordem de chegada, ou seja, o primeiro inteiro diz respeito ao primeiro cliente a chegar na fila, o segundo inteiro diz respeito ao segundo cliente, e assim sucessivamente

Claro! Você gostaria que eu ajudasse a modificar o código para atender às novas regras do problema?



Manoel Titto

sim

Para cumprir as novas regras, você pode modificar o código da seguinte maneira:

csharp

import java.util.\*;

public class PeresImpares {

public static void main (String[] args) {

Scanner teclado = new Scanner(System.in);

int n = teclado.nextInt();

for (int i = 0; i < n; i++) {

List<Integer> valoresPares = new ArrayList<>();

List<Integer> valoresImpares = new ArrayList<>();

int m = teclado.nextInt();

for (int j = 0; j < m; j++) {

int valor = teclado.nextInt();

if (valor % 2 == 0) {

valoresPares.add(valor);

} else {

valoresImpares.add(valor);

}

}

Collections.sort(valoresPares);

Collections.sort(valoresImpares, Collections.reverseOrder());

System.out.println("Caso " + (i+1) + ":");

for (int p : valoresPares) {

System.out.println(p);

}

for (int imp : valoresImpares) {

System.out.println(imp);

}

}

}

}

O código agora lê o número de casos de teste (n) e executa o loop for para cada caso. Dentro do loop, o código lê o número de clientes (m) e adiciona os números recebidos pelos clientes em duas listas separadas (valoresPares e valoresImpares). Em seguida, as listas são ordenadas (pares em ordem crescente e ímpares em ordem decrescente) e os valores são impressos na tela de acordo com o formato especificado. A linha "Caso X:" é impressa antes de cada caso, onde X é o número do caso atual.