Desenvolvimento de Software

Aula 02: Orientação de Objetos

Apresentação

A programação estruturada fornece um meio bastante direto de expressar processos, adequando-se muito bem à implementação de algoritmos, mas com a evolução dos sistemas se torna necessário adotar metodologias de desenvolvimento mais organizadas, o que popularizou muito as técnicas da Orientação a Objetos.

Para a criação de sistemas comerciais devemos compreender alguns pilares que definem essa metodologia, como abstração, herança, polimorfismo e encapsulamento, e aprender a sintaxe específica para a implementação desses conceitos.

Também é necessário que saibamos diferenciar a aplicação de elementos concretos e abstratos na concepção de sistemas Java.

Objetivos

- Identificar os diversos conceitos da Orientação a Objetos;
- Explicar a sintaxe Java para Orientação a Objetos;
- Aplicar a sintaxe Java na criação de elementos concretos e abstratos.

Histórico e Características Gerais

Com a criação de sistemas cada vez maiores e com grande apelo visual, as técnicas tradicionais de modelagem e programação estruturada começaram a entrar em colapso.

Complexos trechos de código inter-relacionados, junto com documentação escassa e diversas replicações de processos já existentes, acabam tornando a manutenção dos sistemas extremamente difícil, aumentando o custo e diminuindo as possibilidades evolutivas desses sistemas.

A orientação a objetos surge neste contexto, trazendo uma forma mais organizada de trabalho, onde a modelagem e a implementação mostram uma proximidade muito maior que nas técnicas ditas tradicionais.

Para podermos adotar essa nova filosofia, devemos deixar de pensar em termos de processos e funções, pois isto é a metodologia estruturada, focada em funcionalidades pontuais e a organização delas.

Agora precisamos pensar de uma forma diferente, em termos de personagens, os quais deverão apresentar características físicas e ações ou verbos.

Exemplo

Na programação estruturada diríamos que o projétil partiu no ângulo de 55 graus, sofrendo a ação da gravidade, atingindo o prédio na altura do quarto andar, pois estamos definindo um processo. Em termos de orientação a objetos, estaríamos considerando que temos um personagem chamado tanque de guerra, e que ele é capaz de atirar um projétil.

A mudança de foco é muito grande, e tem como objetivo:

- Aumento do reuso de código;
- Facilidade de manutenção;
- Documentação automatizada.

Começamos a utilizar de forma mais ampla a programação orientada a objetos (POO[1]) com o advento das interfaces gráficas, pois ficou muito evidente que a programação estruturada não era a melhor opção para construir ambientes constituídos de janelas.

Antigamente tínhamos que utilizar as APIs do sistema operacional para a construção de cada janela, de forma independente, mas com a POO podemos definir um personagem denominado “Janela”, o qual terá atributos como “posição”, “largura” e “altura”, e será capaz de efetuar ações como “abrir”, “minimizar” e “maximizar”, e, a partir daí, colocarmos a quantidade necessária de personagens deste tipo para implementar as interfaces de nossos sistemas.

Abstração

Vídeo.

Atenção! Aqui existe uma videoaula, acesso pelo conteúdo online

Um dos pilares da POO é o conceito de **abstração**, que se refere à definição de um modelo simplificado de algo maior.

Quando abstraímos algo, estamos preocupados apenas com os detalhes que sejam relevantes para o problema de interesse, e essas abstrações serão representadas como **classes** na POO, as quais trazem a definição dos atributos e métodos suportados pelos personagens:

Atributos	Definem características físicas, como cor, idade, endereço etc.
Métodos	As ações ou os verbos que podem ser praticados, tais como comer, andar ou dormir.

Neste ponto podemos abandonar a palavra “personagem” e passar a tratar dos dois níveis de programação que utilizaremos:

Classe	funciona como um molde (tipo ou domínio), para definir como serão os objetos criados a partir da mesma. Elas definem apenas os tipos destes atributos, sem assumir valores.

Objetos

São como variáveis ou elementos físicos criados a partir do molde que é a classe. Cada objeto irá assumir valores específicos para os atributos definidos na classe.

Vamos entender melhor.

Poderíamos considerar uma classe “Pessoa”, que apresenta como atributos o nome e o telefone.

Mas qual nome e telefone?

Cada objeto dessa classe irá assumir valores específicos, como p1 se chama João e seu número de telefone é 1111-1111, enquanto p2 se chama Maria e seu número de telefone é 2222-2222.

Vamos observar estes conceitos iniciais a seguir. Para esse exemplo deveremos criar dois arquivos, um para **Pessoa** e outro para **Exemplo010**.

```
package exemplo010;
public class Pessoa {
  String nome;
  String telefone;

  void exibir(){
    System.out.println(nome+"::"+telefone);
  } }
```

```
package exemplo010;
public class exemplo010 {
  public static void main(String[] args) {
    // Instanciando os objetos p1 e p2
    Pessoa p1 = new Pessoa();
    Pessoa p2 = new Pessoa();

    // Preenchimento dos atributos dos objetos p1 e p2
    p1.nome = "João";
    p1.telefone = "1111-1111";
    p2.nome = "Maria";
    p2.telefone = "2222-2222";

    // Chamada ao método exibir em p1 e p2
    p1.exibir();
    p2.exibir();
  }
}
```

Conforme podemos observar, o primeiro arquivo (Pessoa.java) contém a classe Pessoa, a qual define os atributos nome e telefone e o método exibir.

No outro arquivo (Exemplo010.java) teremos a classe chamadora e seu ponto inicial de execução **main**, onde inicialmente são **instanciados** os objetos **p1** e **p2** com o uso do operador **new**, responsável por alocar esses objetos na memória.

Acompanhando a sequência de comandos, teremos o preenchimento dos atributos nome e telefone para cada um dos objetos, e a chamada ao método exibir de cada um deles.

Teremos como resultado final a impressão do nome e o telefone de cada objeto.

```
João::1111-1111

Maria::2222-2222
```

Existe também um método especial chamado de **construtor**, responsável pela **inicialização** de um objeto recém-criado. Note que ele não aloca o objeto, sendo esta responsabilidade do operador **new**, mas permite inicializar atributos do objeto em seu primeiro momento de existência.

Em algumas linguagens temos também o método **destrutor**, mas o Java não necessita deste método, particularmente por trazer o **garbage collector**, tecnologia que remove da memória qualquer objeto que não possa ser acessado e não tenha mais utilidade para o programa.

Vamos refazer o exemplo anterior...

```
package exemplo010;
public class Pessoa {
  String nome;
  String telefone;

  Pessoa(String nome, String telefone){
    this.nome = nome;
    this.telefone = telefone;
  }

  void exibir(){
    System.out.println(nome+"::"+telefone);
  }
}
```

```
package exemplo010;
public class exemplo010 {
  public static void main(String[] args) {
    // Instanciando e inicializando os objetos p1 e p2
    Pessoa p1 = new Pessoa("João","1111-1111");
    Pessoa p2 = new Pessoa("Maria","2222-2222");

    // Chamada ao método exibir em p1 e p2
    p1.exibir();
    p2.exibir();
  }
}
```

Essa modificação do código não irá alterar a saída, mas podemos observar como o uso de um **construtor** simplificou muito a inicialização dos objetos no main.

Todo construtor é definido pela criação de um método com o mesmo nome da classe, e seria possível utilizar parâmetros com nomes diferentes dos atributos que serão inicializados, mas ocorreria uma grande perda de semântica.

O problema agora é como iremos diferenciar os **parâmetros** e **atributos**, já que eles têm os mesmos nomes. É neste ponto que surge um elemento da orientação a objetos chamado de ponteiro de **autorreferência**, expresso em Java como **this**, e que sempre aponta para os atributos e métodos do objeto corrente.

Observe a linha de código em destaque:

```
this.nome = nome;
```

A forma mais fácil de ler essa linha seria “o nome **deste** objeto receberá o nome passado como parâmetro”.

Algumas classes apresentam vários construtores diferentes, cada um deles com um conjunto de parâmetros distintos, de forma a se adaptarem a contextos diversos, e a aplicação dessa técnica é chamada de **sobrecarga**.

Embora nós utilizemos muito a sobrecarga na orientação a objetos, principalmente na criação de métodos construtores, esta é uma técnica antiga, já existente na programação estruturada, e amplamente utilizada na linguagem C.

Pacotes e Bibliotecas

Quando fazemos um sistema orientado a objetos é comum ocorrer um grande número de classes, e precisamos de uma maneira organizada para classificar e agrupar essas classes de acordo com suas afinidades a determinado contexto.

O primeiro nível de organização são os pacotes (**package**), que seria equivalente ao diretório no qual a classe se encontra, segundo um caminho relativo, tomando como base o diretório raiz do projeto.

Exemplo

O diretório "**\java\math**" equivaleria ao pacote "**java.math**".

Se observarmos os códigos de exemplo anteriores, veremos a presença da palavra **package**, imediatamente no início dos mesmos. Todo arquivo Java deve ter a assinatura de pacote correspondente ao diretório onde se encontra.

Quando utilizamos classes pertencentes ao mesmo pacote não há necessidade de importar elementos, mas ao tratar de pacotes diferentes, será necessário importar elementos de outro pacote com o uso de **import**.

Podemos importar classes dos pacotes do projeto ou de bibliotecas externas.

As bibliotecas são conjuntos de pacotes Java, e suas respectivas classes, compactados para um arquivo com extensão "**.zip**" ou "**.jar**". Essas bibliotecas podem ser referenciadas pelo projeto, ou adicionadas ao **CLASSPATH**, no qual estão relacionadas as bibliotecas padrão do ambiente.

Exemplo

Se quisermos acessar um banco de dados MySQL, podemos adicionar o arquivo mysql-connector.jar ou similar às bibliotecas do projeto. Depois é só utilizar as importações corretas.

O comando **import** pode ser usado de duas formas:

Incluindo apenas uma classe, como **import java.util.ArrayList**

ou

Incluindo todas as classes do pacote (sem recursividade), como **import java.util.***

Herança

O segundo pilar da orientação a objetos é a **herança**, que nos permite criar uma nova classe a partir de outra já existente, configurando um reaproveitamento estrutural.

A árvore de família de animais que nos é apresentada nas aulas de Biologia.

Ela segue o mesmo princípio da herança em termos de programação, pois ao definir um cachorro ou coelho aproveitamos o conceito já existente de mamífero.

Para derivar uma classe de outra utilizaremos a palavra reservada **extends**, e assim como podemos acessar os atributos e métodos internos com **this**, utilizamos a palavra **super** para ter acesso aos atributos e métodos da classe original.

Vamos observar o exemplo seguinte.

```
package exemplo010;
public class Profissional extends Pessoa{
  String profissao;
  Profissional(String nome, String telefone, String profissao) {
    super(nome, telefone);
    this.profissao = profissao;
  }
}
```

Aqui nós criamos uma classe Profissional descendente de Pessoa com o uso de extends. Esta nova classe teve o atributo “profissao” adicionado.

Dica

Não é permitido utilizar acentuação em nomes de atributos, e também um construtor com três parâmetros.

A nova classe apresenta os atributos nome, telefone e profissao, sendo os dois primeiros herdados de Pessoa, e a palavra **super** foi utilizada para chamar o construtor de Pessoa a partir do construtor de Profissional.

Encapsulamento

Quando consideramos uma estrutura fechada qualquer, observamos que alguns elementos são visíveis, enquanto outros não.

Exemplo

Ao observarmos uma pessoa nós podemos ver sua face, seu cabelo, seus braços, mas não temos acesso ao seu estômago nem pulmão.

Na orientação a objetos temos esta mesma característica, pois classes definem estruturas fechadas, nas quais o acesso a seus atributos e métodos deve ser controlado, e para tal iremos contar com três níveis de acesso:

Público (**public**)

Permite que qualquer um acesse o atributo ou método.

Privado (**private**)

Não permite acessos externos, sendo utilizado apenas na programação interna da classe.

Protegido (**protected**)

Permite a utilização na classe e nos descendentes, mas não permite acessos externos.

Quando não definimos nada em termos de nível de acesso, o padrão é o de pacote, ou seja, todas as classes do mesmo pacote podem acessar livremente, mas se ocorrer a importação desse pacote por uma classe pertencente a outro, essa classe externa não terá acesso aos atributos e métodos com nível de pacote.

Mais uma pequena alteração em nossa classe Pessoa...

```
package exemplo010;

public class Pessoa{
  private String nome;
  private String telefone;

  public Pessoa(String nome, String telefone){
    this.nome = nome;
    this.telefone = telefone;
  }

  public void exibir(){
    System.out.println(nome+"::"+telefone);
  }

}
```

Essa alteração não impactará na classe chamadora, pois o construtor e o método exibir estão definidos como públicos, mas se tentarmos acessar os atributos de p1 ou p2 a partir do main, como na primeira versão, ocorrerá um erro de compilação.

```
p1.nome = "João"; // Esta linha irá gerar um erro de compilação
```

É muito comum que precisemos controlar o acesso a determinado atributo sem, no entanto, impedir esse acesso, e nesse ponto devemos utilizar os **getters** e **setters**.

Esse controle é chamado de encapsulamento, que trata de mais um dos pilares da orientação a objetos e, em termos formais, visa ocultar detalhes da implementação interna da classe, fornecendo apenas uma interface com métodos de negócio e métodos de acesso.

O método **getter** serve para obter o valor de um atributo interno, enquanto o **setter** permite escrever um valor nesse atributo. O conjunto dos dois define uma **propriedade** da classe.

Podemos implementar muito facilmente a técnica de encapsulamento em nossa classe Pessoa modificada.

```
package exemplo010;

public class Pessoa{
  public String getNome(){
    return nome;
  }
  public void setNome(String nome){
    this.nome = nome;
  }
  public String getTelefone(){
    return telefone;
  }
  public void setTelefone(String telefone){
    this.telefone = telefone;
  }
  private String nome;
  private String telefone;
  public Pessoa(String nome, String telefone){
    this.nome = nome;
    this.telefone = telefone;
  }
  public void exibir(){
    System.out.println(getNome()+"::"+getTelefone());
  }
}
```

Note que foram apenas acrescentados os métodos de leitura (getNome e getTelefone) e os métodos de escrita (setNome e setTelefone). Para não ocorrer perda de semântica devido a alteração de nomes, o uso de **this** é necessário nos **setters**.

Polimorfismo

Vídeo.

Atenção! Aqui existe uma videoaula, acesso pelo conteúdo online

O último dos pilares da orientação a objetos é o polimorfismo, consistindo na possibilidade de uma classe descendente efetuar alterações sobre métodos herdados, de forma a adaptar as funcionalidades a seu próprio contexto.

Esta talvez seja a característica mais poderosa da orientação a objetos, proporcionando grande flexibilidade a qualquer sistema, e devemos compreendê-la corretamente para tirar o melhor proveito da metodologia.

Como já é de nosso conhecimento, por meio do processo de herança, as classes descendentes herdam todas as caraterísticas existentes nas ancestrais.

Mas e se algumas ações não forem exatamente o que se espera?

É nesse ponto que precisamos do polimorfismo[2].

Observe o diagrama seguinte.

Este é um diagrama de classes da **UML** (Unified Modeling Language) representando a classe Profissional e descendentes diretos e indiretos da mesma.

Tomando como base Profissional, podemos dizer que todo objeto deste tipo irá trabalhar, assim como os objetos das classes descendentes, mas certamente irão trabalhar de forma diferente.

- se for um engenheiro pode estar indo fazer o projeto de uma casa ou de um carro;
- no caso do médico talvez esteja indo operar alguém;
- e um advogado estará acompanhando algum processo.

No entanto, a frase seria a mesma: “Estou indo trabalhar”.

O uso de polimorfismo permitirá alterar a funcionalidade do método trabalhar para as diversas classes envolvidas.

Vamos observar um código com uso de polimorfismo, com a alteração da nossa classe Profissional de exemplo. Essa alteração é necessária para que o método exibir passe a mostrar também a profissão, e não apenas o nome e telefone.

```
public class Profissional extends Pessoa{
  private String profissao;
  public Profissional(String nome, String telefone, String profissao) {}
    super(nome, telefone);
    this.profissao = profissao;
  }
  @Override
  public void exibir() {
    super.exibir();
      // Chama o exibir de Pessoa, imprimindo nome e telefone
    System.out.println("\tTrabalha como "+profissao);
      // Complementa a informação acerca da profissão
  }
}
```

Note que o novo método exibir chama o original através do uso de **super**.

Observe também a presença da anotação **@Override**, que deve ser colocada nos métodos polimórficos e significa **sobrescrito**, não devendo nunca ser confundido com a sobrecarga, cujo nome em inglês seria overload.

A seguir podemos observar um pequeno teste com chamadas ao método exibir.

```
package exemplo010;
public class Exemplo010a {
  public static void main(String[] args) {
    Pessoa[] pessoas = {new Pessoa("Joao","1111-1111"),
      new Pessoa("Maria","2222-2222"),
      new Profissional("Luiz","3333-3333","Advogado")};
    for(int i=0; i< 3; i++)
      pessoas[i].exibir();
  }
}
```

Note que foi criado um vetor de Pessoa, aceitando tanto Pessoa quanto Profissional. Isto se deve a uma lei da POO que define o seguinte: "qualquer **descendente** pode ser utilizado no lugar da classe".

Se observarmos o mundo real, esta lei também pode ser aplicada, pois um profissional é uma pessoa, embora nem sempre uma pessoa seja um profissional.

A saída desse programa será a seguinte:

```
Joao::1111-1111
Maria::2222-2222
Luiz::3333-3333
Trabalha como Advogado
```

Note que mesmo definido o vetor como do tipo **Pessoa**, o exibir correto foi chamado para o **Profissional** existente neste vetor, e isso se deve ao polimorfismo.

Classes Abstratas e Interfaces

Vídeo.

Atenção! Aqui existe uma videoaula, acesso pelo conteúdo online

Normalmente confundimos muito as palavras abstração e abstrato, mas elas apresentam sentidos muito distintos:

Uma abstração é um modelo simplificado de algo.

Um elemento abstrato é algo intangível, que não é palpável ou concreto.

Existem dois tipos de elementos abstratos em Java: classes abstratas e interfaces.

As **classes abstratas** são incompletas, não podem gerar objetos, mas servem de base para impor determinada característica funcional a seus descendentes, que serão obrigados a implementar as funcionalidades abstratas para gerar objetos.

Exemplo

Um elemento abstrato da vida real seria mamífero, enquanto cachorro e gato são elementos concretos. Mesmo não podendo gerar objetos, o elemento mamífero define uma regra para seus descendentes, ou seja, para descender de mamífero precisa saber mamar.

Vamos observar um exemplo de classe abstrata em Java envolvendo a criação de quatro classes.

```
public abstract class Mamifero {
  protected String familia;
  public abstract void mamar();
  public String getFamilia(){
    return familia;
  }
}
```

```
public class Cachorro extends Mamifero{
  public Cachorro() {
    familia = "Canidae";
  }
  @Override
  public void mamar() {
    System.out.println("Cachorro mamando...");
  }
}
```

```
public class Gato extends Mamifero{
  public Gato() {
    familia = "Felidae";
  }
  @Override
  public void mamar() {
    System.out.println("Gato mamando...");
  }
}
```

```
public class Exemplo011 {
  public static void main(String[] args) {
    Mamifero m = new Gato();
    m.mamar();
  }
}
```

Nós criamos inicialmente a classe abstrata Mamifero, com uso da palavra **abstract** em termos da definição da classe e do método abstrato interno.

Atenção

Essa classe não pode gerar instâncias, afinal falta uma parte dela que é o método **mamar( )**, mas permite herança e obriga a definição do método faltante. Isso pode ser observado nas classes Cachorro e Gato, descendentes de Mamifero, e que por serem concretas podem ser instanciadas.

Ao final, podemos ver uma classe de teste, onde é instanciado um Gato no lugar de Mamifero, sendo chamado em seguida o método mamar( ) implementado na classe descendente. Como esperado, teremos a impressão da frase “Gato mamando...”.

Outro tipo de elemento abstrato é a interface, e esta não deve ser confundida com interface visual, que trata da interação com o usuário.

Aqui estamos preocupados com a interface programacional, que se preocupa em definir quais métodos estarão disponíveis para uso de determinado tipo de componente.

Em termos práticos, uma interface é um conjunto de assinaturas de métodos abstratos, os quais devem ser implementados pelas classes que se proponham a tal. Ao implementar esses métodos, as classes passam a ser reconhecidas por algum ferramental que os utiliza, promovendo funcionalidades específicas.

As interfaces são de grande utilização no Java, e podemos observar um exemplo a seguir, complementando o exemplo anterior.

```
public interface Voo {
  void voar();
}
```

```
public class Morcego extends Mamifero implements Voo{
  public Morcego() {
    familia = "Phyllostomidae";
  }

  @Override
  public void mamar() {
    System.out.println("Morcego mamando...");
  }

  @Override
  public void voar() {
    System.out.println("Morcego voando...");
  }

}
```

```
public class Exemplo011a {
  public static void main(String[] args) {
    Object[] objetos = {new Gato(), new Morcego(),
                        new Cachorro()};

    for(int i=0; i< 3; i++) {
      if(objetos[i] instanceof Voo)
        ((Voo)objetos[i]).voar();
          // Conversão de tipo (type cast) necessária
    }
  }
}
```

Neste exemplo temos a definição de uma interface, com o uso da palavra **interface** no lugar de **class**. Uma interface é naturalmente abstrata, logo seus métodos são considerados por padrão **públicos** e **abstratos**.

Uma observação é a de que as interfaces não aceitam atributos, e podem ser consideradas como conjuntos de assinaturas de métodos.

Podemos observar que a classe Morcego **deriva** de Mamifero e **implementa** a interface Voo com o uso da palavra **implements**. Ao implementar a interface, ela se torna obrigada a implementar o método **voar( )**.

Ao final temos um pequeno teste, onde é percorrido um vetor de objetos, e perguntado a cada um deles se implementa Voo ou não com o uso de **instanceof**, e caso implemente, ocorre a conversão (**type cast**) para o tipo da interface e a subsequente chamada de **voar ( )**.

Executando o exemplo teremos apenas a frase “Morcego voando...” sendo impressa.

Dica

Note que implementar é muito diferente de herdar. Nesse caso fica claro que Morcego é um Mamifero, e que ele implementa Voo, mas não herda do mesmo.

Embora muitos confundam as interfaces com classes abstratas, as diferenças em termos metodológicos são muito grandes:

As classes abstratas definem uma regra para sua família de classes.	X	As interfaces podem ser implementadas por qualquer classe que necessite participar de algum processo específico.

Atividade

1. Qual das características da Orientação a Objetos permite a alteração funcional de um método herdado, permitindo grande flexibilidade e adaptabilidade ao ambiente?

a) Sobrecarga
b) Construtores
c) Encapsulamento
d) Polimorfismo
e) Destrutores

2. Segundo a sintaxe do Java, qual a palavra utilizada para indicar que uma classe é descendente de outra?

a) extends
b) implements
c) super
d) this
e) private

---

3. Você criou um novo sistema de finanças e, como estratégia de desenvolvimento, resolveu criar um núcleo de cálculo adaptável, e as classes que desejarem utilizar as funcionalidades desse núcleo deverão implementar um conjunto de métodos abstratos, podendo pertencer a qualquer família de classes. Qual elemento de programação você deverá utilizar para viabilizar essa funcionalidade?

a) Interface Programacional
b) Classe Abstrata
c) Classe Concreta
d) Mecanismo de Herança
e) Sobrecarga

Notas

POO[1]

O termo Programação Orientada a Objetos (POO) foi criado por Alan Kay, autor da linguagem de programação Smalltalk. Mas mesmo antes da criação do Smalltalk, algumas das ideias da POO já eram aplicadas, sendo que a primeira linguagem a realmente utilizar essas ideias foi a Simula 67, criada por Ole-Johan Dahl (1931-2002) e Kristen Nygaard (1926-2002) em 1967. Entretanto só veio a ser aceito realmente nas grandes empresas de desenvolvimento de software por volta dos anos 1990.

Fonte:Wikibooks <https://pt.wikibooks.org/wiki/Programa%C3%A7%C3%A3o_Orientada_a_Objetos/Introdu%C3%A7%C3%A3o#Hist%C3%B3ria> . Acesso em: 11 jan. 2019.

Polimorfismo[2]

Significa “múltiplas formas”, ou seja, as diversas formas com que uma ação pode ser efetuada.

Referências

CORNELL, G.; HORSTMANN, C. **Core java**. 8. São Paulo: Pearson, 2010.

DEITEL, P.; DEITEL, H. **Java, como programar**. 8. São Paulo: Pearson, 2010.

FONSECA, E. **Desenvolvimento de software**. Rio de Janeiro: Estácio, 2015.

SANTOS, F. **Programação I**. Rio de Janeiro: Estácio, 2017.

Próxima aula

- Estrutura de tratamento de erros do Java;
- Implementação da Modelagem Comportamental;
- Biblioteca de classes para Coleções do Java.

Explore mais

Pesquise na internet sites, vídeos e artigos relacionados ao conteúdo visto.

Em caso de dúvidas, converse com seu professor online por meio dos recursos disponíveis no ambiente de aprendizagem.

Leia os textos:

- Os 4 pilares da Programação Orientada a Objetos; <https://www.devmedia.com.br/os-4-pilares-da-programacao-orientada-a-objetos/9264>
- Fundamentos da linguagem Java. <https://www.ibm.com/developerworks/br/java/tutorials/j-introtojava1/index.html>

Leia o livro Programação em Java de Claro e Sobral publicado em 2008 pela Person.