**PONTIFÍCIA UNIVERSIDADE CATÓLICA DO PARANÁ ESCOLA POLITÉCNICA**

*CURSO DE ENGENHARIA DE COMPUTAÇÃO PROGRAMAÇÃO ORIENTADA A OBJETOS*

Nosso cérebro é o melhor brinquedo já criado: nele se encontram todos os segredos, inclusive o da felicidade.

Charles Chaplin



*TDE I – Conceitos iniciasi de POO/Java*

Considerando os códigos-fonte do projeto TDE1, disponíveis no arquivo TDE1.zip e as referências bibliográficas sugeridas como material de apoio às aulas e indicadas na definição deste TDE no BlackBoard, responda às questões abaixo de maneira completa, autocontida, e que sirva de sistematização conceitual e de resumo dos principais temas iniciais da disciplina.

Na classe TestaPessoa, quando escrevemos:

Pessoa p1 = **new** Pessoa("Marie Curie");

Pessoa p2 = **new** Pessoa ("Albert Einstein", LocalDate.*of*(1879, Month.***MARCH***, 14)); Pessoa p3 = **new** Pessoa ("James Maxwell", 13, 6, 1831);

Estamos criando objetos da classe Pessoa referenciados pelas variáveis p1, p2 e p3. Explique então:

1. O que é uma classe? O que ela contém?

**R: Classe é um tipo de construção que serve como um plano para a criação de objetos, na qual define os atributos e métodos comuns que serão compartilhados com os objetos. Ou seja, ela contém atributos e métodos.**

1. O que é uma instância de uma classe?  
   **R: Objeto é uma instância (o que é gerado da classe) ESPECÍFICA de uma classe.**
2. O que é um método construtor? Quando ele é chamado?  
   **R: É um método especial que tem o mesmo nome da classe e garante a alocação de memória necessária para cada objeto da classe. Sua função principal é garantir que os atributos de um objeto tenham valores iniciais adequados assim que a instância da classe é criada. Ele é chamado quando o operador "new" é usado para um novo objeto.**
3. O que é sobrecarga (*overloading*)?

**R: É um conceito que permite que um mesmo nome de método seja usado para executar diferentes operações dependendo do número ou tipo de parâmetros passados, ou seja, um mesmo método pode ter comportamentos diferentes dependendo dos parâmetros passados.**

1. Qual a utilidade deste mecanismo de sobrecarga na codificação de construtores e como ele foi usado no exemplo acima?

**R: A sobrecarga ou overloading de construtores permite que você crie diferentes formas de inicializar objetos da mesma classe com base em diferentes conjuntos de parâmetros. No exemplo dado, foram definidos diferentes construtores para a classe Pessoa para permitir a criação de objetos com diferentes informações, como nome apenas, nome e data de nascimento, ou nome e data de nascimento especificando o dia, mês e ano separadamente. Isso torna o código mais flexível e adaptável a diferentes situações de uso.**

Na classe Pessoa temos:

**private** String nome;

**private** LocalDate dataNascimento;

Em TestaPessoa tínhamos:

Pessoa p4 = **new** Pessoa("Fulano de Tal");

Endereco e = **new** Endereco("Av. das Palmeiras", 98, "Curitiba", "PR", "85123-789"); p4.setEndRes(e);

Porém, não conseguimos acessar diretamente os atributos de um objeto Pessoa, ou seja, neste exemplo não podemos fazer:

P4.endRes = e; // ERRO

Devido ao encapsulamento que foi empregado na classe Pessoa.

1. O que significa encapsulamento?

**R: O encapsulamento consiste em ocultar os detalhes internos de um objeto e expor apenas o que é necessário para interagir com ele. Isso é feito definindo os atributos como privados e fornecendo métodos públicos para acessá-los e modificá-los, controlando assim o acesso aos dados e protegendo o estado interno do objeto contra modificações não autorizadas.**

**Vantagens do Encapsulamento:**

**1. Tornar mudanças invisíveis;**

**2. Facilitar reutilização do código;**

**3. Reduzir efeitos colaterais.**

1. Qual a importância do encapsulamento para a segurança de uma classe?

**R: O encapsulamento é importante para não modificar/afetar os atributos de uma classe e objeto, fazendo com que o desenvolvedor do código controle os dados e estado interno do objeto, ou seja, as razões na qual o encapsulamento é importante são: 1. Controle de Acesso; 2. Proteção contra Modificações Indesejadas; 3. Validação de Dados; 4.** **Facilita a Evolução do Código.**

1. Neste contexto, o que são os métodos apelidados de *setters* e de *getters*?

**R:** **Os métodos getters e setters são usados para acessar e modificar os atributos de uma classe. Eles são uma parte importante do conceito de encapsulamento, que ajuda a proteger os dados de uma classe e a controlar o acesso a esses dados. Os getters são os metódos de acesso, no qual são usados para obter o valor do atributo, porém eles não alteram o resultado do objeto. Já os setters, são aqueles métodos de modificação que permitem que você defina um novo valor para um atributo. No exemplo fornecido, p4.setEndRes(e) é um método setter que permite definir o endereço residencial de uma pessoa, no caso, do objeto pessoa p4.**

1. Explique os modificadores de acesso *private, public, default* (ou *package private*) e *protected*.

**R:**

**- Public: é um modificador de acesso utilizado para classes, atributos, métodos e construtores, tornando-os acessíveis por qualquer outra classe. Esse é o nível de visibilidade mais aberto em Java e é frequentemente usado para métodos e atributos que devem ser acessíveis por outros componentes do programa.**

**- Private: é um modificador de acesso utilizado para atributos, métodos e construtores, tornando-os acessíveis apenas dentro da classe declarada.** **Membros privados não são acessíveis de fora da classe, nem mesmo pelas subclasses. Isso promove o encapsulamento, garantindo que os detalhes internos da implementação da classe não sejam expostos a partes externas.**

**- Default (ou Package-private): Quando nenhum modificador é especificado (ou seja, nenhum modificador é colocado antes do membro), o modificador de acesso padrão é aplicado. Membros com acesso padrão são acessíveis apenas dentro do mesmo pacote em que foram declarados. Eles não podem ser acessados de fora do pacote, mesmo que estejam em classes diferentes.**

**- Protected: é um modificador de acesso utilizado para atributos, métodos e construtores, tornando-os acessíveis no mesmo pacote e subclasses.** **Esse modificador é útil quando você deseja que os membros sejam acessíveis apenas por subclasses, mas não por outros componentes fora do pacote.**

Na classe Endereco temos:

**public** String toString() {

**return** logradouro + ", " + num + " " + comp + "\n"

+ cidade + " - " + estado + "\n" + cep;

}

1. Este método está redefinindo o comportamento de toString(), explique o conceito de redefinição (*overriding*).

**R: A redefinição (overriding) é um conceito da programação orientada a objetos em que uma subclasse fornece uma implementação específica de um método que já está definido em uma de suas superclasses. Quando uma subclasse redefine um método, ela substitui a implementação da superclasse por uma nova implementação própria. Ou seja, A redefinição ocorre quando um método cuja assinatura já tenha sido especificada recebe uma nova definição (ou seja, um novo corpo) em uma classe derivada. No contexto do método toString( ) da classe Endereco, a redefinição está ocorrendo porque a classe Endereco está substituindo a implementação padrão do método toString( ) que é herdada de sua superclasse (geralmente Object em Java) por uma nova implementação personalizada.**

1. Quais métodos da classe Object são tipicamente redefinidos?

**R:**

**- clone( ): Cria e retorna uma cópia deste objeto.**

**- equals(Object obj): Indica se algum outro objeto é "igual" a este.**

**- finalize( ): Chamado pelo coletor de lixo em um objeto quando a coleta de lixo determina que não há mais referências ao objeto.**

**- getClass( ): Retorna a classe de tempo de execução deste Objeto.**

**- hashCode( ):Retorna um valor de código hash para o objeto.**

**- notify( ): Ativa um único thread que está aguardando no monitor deste objeto.**

**- notifyAll( ): Ativa todos os threads que estão aguardando no monitor deste objeto.**

**- toString( ): Retorna uma representação de string do objeto.**

**- wait( ):Faz com que o thread atual espere até que outro thread invoque o método notify( ) ou o método notifyAll( ) para este objeto.**

**- wait(long timeout):** **Faz com que o thread atual espere até que outro thread invoque o método notify( ) ou o método notifyAll( ) para este objeto, ou um período de tempo especificado tenha decorrido.**

**- wait(long timeout, int nanos): Faz com que o thread atual espere até que outro thread invoque o método notify( ) ou o método notifyAll( ) para este objeto, ou algum outro thread interrompa o thread atual, ou um determinado período de tempo real tenha decorrido.**

Na classe Pessoa temos:

**public** Pessoa(String nome, LocalDate dataNasc) {

**super**(); **this**.nome = nome;

**this**.dataNascimento = dataNasc;

}

**public** Pessoa(String nome, **int** dia, **int** mes, **int** ano) {

**this**(nome, LocalDate.*of*(ano, mes, dia));

}

1. Explique o “super()”

**R: Na classe Pessoa, os construtores mostrados estão chamando super( ) para invocar explicitamente o construtor da superclasse. O uso de super( ) em um construtor de uma classe filha é necessário quando a superclasse tem um construtor que precisa ser chamado para inicializar parte do estado da instância antes de continuar com a execução do construtor da subclasse. Ele é usado para inicializar os atributos herdados da superclasse antes de executar o código específico da subclasse no seu próprio construtor.**

1. Explique o “this.”

**R: Dentro de um método de instância ou construtor, esta é uma referência ao objeto atual - o objeto cujo método ou construtor está sendo chamado. Se refere à instância atual da classe em que está sendo usada. Ela é frequentemente usada para distinguir entre variáveis locais e membros da classe com o mesmo nome, bem como para acessar métodos e atributos da própria instância.**

1. Explique o “this()”

**R: O this( ) é uma chamada especial que é usada dentro de um construtor para chamar outro construtor da mesma classe. Isso é conhecido como construtor chaining (encadeamento de construtores), onde um construtor pode chamar outro construtor da mesma classe para evitar duplicação de código.**

Na classe TestaPessoa temos o seguinte código:

p1.setDataNasc(LocalDate.*of*(1867, Month.***NOVEMBER***, 7));

1. Estamos passando o resultado de um método para um método setter. Explique o funcionamento desta chamada e o que espera-se que ocorra com o argumento passado.

**R: O método setter setDataNasc( ) é responsável por definir o atributo dataNascimento do objeto p1 com o valor fornecido como argumento. Portanto, o que se espera que ocorra é que o atributo dataNascimento do objeto p1 seja definido como 7 de novembro de 1867, conforme especificado pelo argumento passado para o método setter. Essa data será armazenada no formato LocalDate, permitindo que o objeto p1 mantenha informações sobre sua data de nascimento de forma eficiente e padronizada.**

1. O que é Month.***NOVEMBER*** e em que local foi definido?

**R: Month.NOVEMBER é uma referência estática para o mês de novembro. Essa referência é definida dentro da própria API Java Time, mais precisamente no enum Month, que faz parte do pacote java.time. Portanto, Month.NOVEMBER é uma constante pré-definida que representa o mês de novembro em Java e pode ser usada em qualquer lugar do código onde seja necessário referenciar esse mês.**

1. O método of(int year, Month month, int dayOfMonth) da classe LocalDate é static. O que isto significa?

**R: De acordo com o Java doc, o método of(int year, Month month, int dayOfMonth), obtém uma instância de LocalDate de um ano, mês e dia. Quando um método é declarado como estático em Java, isso significa que ele pertence à classe em si, em vez de pertencer a uma instância específica da classe, ou seja, ele pode ser chamado diretamente na classe sem a necessidade de criar uma instância da classe e está associado à classe em vez de instâncias específicas dela.**

1. Na classe LocalDate também está disponível o método of(int year, int month, int dayOfMonth). No que eles diferem? Como isso é possível?

**R: Diferem em relação ao tipo de parâmetro utilizado para representar o mês. O método of(int year, Month month, int dayOfMonth) usa Month que é um enum que representa os meses do ano, fornecendo uma abordagem mais segura e legível em comparação com o uso de inteiros para representar os meses. No caso do método of(int year, int month, int dayOfMonth), o mês é representado como um número inteiro, começando de 1 para janeiro até 12 para dezembro. Isso é possível devido à sobrecarga de métodos em Java. Sobrecarga de métodos permite que uma classe tenha múltiplos métodos com o mesmo nome, desde que tenham diferentes tipos de parâmetros ou números de parâmetros. Assim, a classe LocalDate pode ter dois métodos of com a mesma assinatura de nome, mas diferindo nos tipos dos parâmetros, permitindo assim a chamada de método apropriada com base no tipo de argumentos fornecidos.**

Já na classe Pessoa, no método getIdade(), temos o seguinte código:

LocalDate hoje = LocalDate.*now*();

1. Ou seja, foi empregado outro método da mesma classe LocalDate. Qual a diferença entre estes dois métodos: of() e now()? Quando se usa o primeiro e quando se usa o segundo?

R: **O método of( ) é usado para criar uma instância de LocalDate com uma data específica fornecida como argumentos, incluindo o ano, mês e dia, como visto na questão 17 e 18. Já o método now( ) é usado para para se obter a data atual, no caso, foi passado uma variável hoje para armazenar o data atual.**

1. Também na classe Pessoa temos o método getIdade(). Explique o funcionamento do seu algoritmo para calcular a idade da pessoa em questão.

**R: public int getIdade() {  
 int idade;  
 LocalDate hoje = LocalDate.*now*();  
 Period tempo = Period.*between*(dataNascimento, hoje);  
 idade = tempo.getYears();  
 return idade;**

**}**

**Nessa primeira linha do método, foi criado uma variável inteira idade para se obter a idade da**

**pessoa, na segunda foi adicionado uma variável local para armazenar a data atual, na terceira é**

**usado a classe Period que calcula a diferença entre a data de nascimento e a data atual e armazena**

**na variável local tempo. Logo depois, a variável tempo pega exclusivamente o ano calculado pelo**

**Period (através do método getYears( )) e armazena no idade, terminando o método retornando a**

**idade obtida já calculada.**

Na classe TestaPessoa foi utilizado um vetor de Pessoa (vet) e depois um ArrayList de Pessoa

(alPessoa).

1. Em que situações é interessante usarmos um vetor tradicional de objetos e qual é sua principal limitação?

**R: Um vetor tradicional de objetos é interessante quando se conhece previamente o número exato de objetos que serão armazenados e quando não há necessidade de aumentar dinamicamente o tamanho do vetor durante a execução do programa. Ele é uma estrutura de dados simples e eficiente para armazenar e acessar um conjunto fixo de elementos do mesmo tipo.A principal limitação de um vetor tradicional de objetos é sua inflexibilidade no que diz respeito ao tamanho. Uma vez que o tamanho de um vetor é determinado durante a sua criação e não pode ser alterado posteriormente, isso pode ser problemático se o número de elementos a serem armazenados não for conhecido antecipadamente ou se variar dinamicamente durante a execução do programa. Se o vetor não for grande o suficiente para armazenar todos os objetos necessários, pode ocorrer um estouro de buffer ou uma exceção de índice fora dos limites.**

1. Qual o motivo de fazer o “**if** (vet[i] != **null**)no laço que manipula o vetor?

**R: O if (vet[i] != null) verifica se a posição vet[i] do array contém uma referência não nula a um objeto da classe Pessoa. Se a posição não for null, isso significa que há um objeto Pessoa armazenado nessa posição. Se a posição for null, o código dentro do bloco if será ignorado, evitando assim um possível erro de NullPointerException.**

1. Em que situações é interessante usarmos um ArrayList e qual é seu principal benefício?

**R: O ArrayList Pode ser usado para criar um número indefinido de objetos, pois ele permite a adição dinâmica de elementos à medida que necessário. Na verdade, é isso que torna o vetor tão útil em muitos casos. No exemplo fornecido pelo TDE, onde alPessoa é um ArrayList de Pessoa, ele é uma coleção dinâmica que pode crescer ou diminuir conforme necessário. Existem também outras situações que podem ser utilizados o ArrayList como:**

1. **Quando o número de elementos é variável ou desconhecido; (como citado)**
2. **Quando a inserção e remoção de elementos são frequentes;**
3. **Quando a ordem dos elementos precisa ser mantida;**
4. **Quando é necessário acessar elementos por índice;**
5. **Quando há uma necessidade de percorrer a lista.**

**Ou seja, seu principal benefício é sua flexibildade e facilidade de uso.**

1. Por que a instrução alPessoa.add(e)não poderia ser executada (objeto e definido na linha 19)?

**R: Porque o ArrayList alPessoa é uma lista de objetos para a classe Pessoa, ou seja, só pode ser adicionado objetos do tipo Pessoa a ele e não a classe Endereço. Se você deseja adicionar um objeto Endereco à lista alPessoa, primeiro é necessário criar um objeto Pessoa que tenha um atributo do tipo Endereco e, em seguida, atribuir o objeto Endereco a esse atributo. Como por exemplo:**

**Pessoa p4 = new Pessoa("Fulano de Tal");**

**Endereço e = new Endereco(“"Av. das Palmeiras", 98, "Curitiba", "PR", "85123-789");**

**p4.setEndRes(e);**

**alPessoa.add(p4);**

**Dessa forma, você cria uma instância de Pessoa com um objeto Endereco associado e adiciona essa instância à lista alPessoa.**

1. O método mostraLista foi definido com static, com isso ele se torna um método de classe.

Diferencie métodos de classe e métodos de instância e comente quando cada um deve ser empregado.

**R: Os métodos de classe são utilizados quando a funcionalidade é independente de qualquer instância específica e pode ser compartilhada por todas as instâncias da classe. Por outro lado, os métodos de instância são utilizados quando a funcionalidade está relacionada ao estado específico de uma instância e precisa acessar/modificar os atributos dessa instância. Um exemplo de método de instância seria os getters e os setters utilizados na classe Endereco e Pessoa.**

1. Da mesma forma que ocorre com métodos, podemos ter variáveis de classe e variáveis de instância. Comente quando cada tipo de atributo deve ser empregado e sua relação com o tipo de método que irá acessá-lo.

**R: As variáveis de classe são compartilhadas por todas as instâncias da classe. Elas são declaradas usando o modificador "static" e mantêm o mesmo valor para todas as instâncias. São frequentemente usadas para armazenar dados que são comuns a todas as instâncias da classe, como constantes ou configurações globais.Por outro lado, as variáveis de instância são específicas de cada objeto (ou instância) da classe. Cada objeto tem sua própria cópia dessas variáveis, que são inicializadas quando o objeto é criado. Elas são usadas para armazenar dados que são específicos para cada objeto individual, como atributos ou propriedades.**

**Simplificando em um simples exemplo: Imagine que você está em uma festa e todos estão usando chapéus. Os chapéus podem ser de duas cores: azul ou vermelho.As variáveis de classe seriam como uma grande caixa que está no meio da festa, e dentro dessa caixa tem os chapéus. Todo mundo na festa pode pegar chapéus dessa caixa. Se alguém muda a cor de um chapéu, todos veem a mudança, porque estão pegando chapéus da mesma caixa. Esses chapéus são compartilhados por todos.Agora, as variáveis de instância seriam como um chapéu que cada pessoa usa. Cada pessoa tem o seu próprio chapéu, e pode ser de qualquer cor que quiser. Se alguém muda a cor do seu chapéu, isso não afeta os chapéus das outras pessoas, porque cada pessoa tem o seu próprio chapéu. Esses chapéus são específicos para cada pessoa.Então, as variáveis de classe são como coisas que todo mundo na festa compartilha, enquanto as variáveis de instância são coisas que cada pessoa tem só para si.**

1. O método mostraLista efetua uma iteração completa sobre os objetos do ArrayList usando os métodos hasNext() e next(). Comente sobre esta funcionalidade e sobre as interfaces Iterable e Interator.

**R: Interface Iterable: A interface Iterable é uma interface genérica em Java que define um contrato para objetos que podem ser iterados. Ela possui apenas um método, iterator( ), que retorna um objeto Iterator. As classes que implementam Iterable podem ser percorridas usando a estrutura de controle for-each (ou "enhanced for loop") em Java.**

**Interface Iterator: A interface Iterator é outra interface em Java que define um contrato para um objeto que pode ser usado para iterar sobre elementos de uma coleção. Ela contém três métodos principais: hasNext( ), next( ), e opcionalmente remove( ). O hasNext() verifica se há mais elementos na coleção. O next( ) retorna o próximo elemento na coleção. O remove( ) remove o elemento atual da coleção (este método é opcional e pode não ser suportado por todas as implementações de Iterator).**

**Um exemplo simples seria:**

**Imagine que você tem uma caixa cheia de brinquedos, e você quer ver cada brinquedo um por um. A interface Iterable seria como uma caixa que diz "Ei, eu posso ser aberta e você pode ver todos os brinquedos que estão dentro de mim".**

**Agora, para realmente ver cada brinquedo, você precisa de algo chamado Iterator. O Iterator é como uma mãozinha que você coloca dentro da caixa. Com essa mãozinha, você pode pegar um brinquedo de cada vez e olhar para ele. Ele diz "Ok, vamos ver o próximo brinquedo!" quando você quer ver outro.**

**Então, a função do Iterator é apenas pegar um brinquedo por vez e passá-lo para você, e você decide o que fazer com ele. Ele não sabe o que fazer com os brinquedos, só sabe como pegá-los um por um.**

**Juntos, a interface Iterable e o Iterator formam um time imbatível! A Iterable diz que pode ser aberta, e o Iterator diz como pegar cada item que está dentro dela. Com eles, você pode ver todos os brinquedos da sua caixa, um por um, sem perder nenhum!**

Na classe TestaPessoa temos na linha 52:

Collections.*sort*(alPessoa);

1. Este sort é um método static do *framework* collections, ele fará uma ordenação ascendente usando a ordem natural. Qual é esta *ordem natural*, onde ela foi definida? Como sort sabe efetuar a comparação de objetos da classe Pessoa?

**R: A ordem natural dos objetos da classe Pessoa é definida pelo método compareTo() implementado na própria classe Pessoa. Essa ordem natural é baseada na comparação dos nomes das pessoas. O método compareTo() compara os nomes usando o método compareToIgnoreCase(), o que significa que a comparação é realizada ignorando diferenças entre letras maiúsculas e minúsculas.O método sort da classe Collections utiliza essa implementação do método compareTo( ) fornecida pela classe Pessoa para comparar e ordenar os objetos Pessoa. Dessa forma, o sort sabe como efetuar a comparação de objetos da classe Pessoa e ordená-los corretamente com base na ordem natural definida na implementação do método compareTo( ).**

Depois temos o seguinte trecho:

Collections.*sort*(alPessoa, **new** Comparator<Pessoa>() {

@Override

**public int** compare(Pessoa p1, Pessoa p2) {

**return** Integer.*compare*(p1.getIdade(), p2.getIdade());

}

});

1. Que emprega um sort com dois argumentos. O segundo é um Comparator. O que é este

Comparator? No que ele difere do Comparable? Qual método ele exige?

**R: O Comparator é uma interface funcional em Java que permite definir uma função de comparação personalizada entre dois objetos. Ao contrário da interface Comparable, que está associada à própria classe e define sua ordem natural, o Comparator permite especificar diferentes critérios de comparação entre objetos sem modificar a implementação original da classe.**

**A diferença principal entre Comparator e Comparable é que o primeiro é uma interface externa à classe, enquanto o segundo pertence à própria classe. O Comparator exige a implementação do método compare(), que compara dois objetos e retorna um valor inteiro indicando a relação entre eles.**

**Enquanto o Comparable é usado para ordenação padrão de objetos e define a ordem natural da classe através do método compareTo(), o Comparator é usado para ordenação personalizada ou quando múltiplos critérios de comparação são necessários.**

**No exemplo fornecido, o Comparator está sendo empregado para ordenar a lista de pessoas com base na idade. A implementação do método compare() compara as idades das pessoas e retorna o resultado da comparação, permitindo assim a ordenação da lista de acordo com este critério específico.**

**Explicando de uma forma mais simplificada ainda: O Comparator é como uma regra que você cria para comparar duas coisas. Por exemplo, se você tem duas pessoas e quer compará-las pelo tamanho, você pode usar um Comparator que verifica quem é mais alto.**

**A diferença entre o Comparator e o Comparable é que o Comparable é uma regra que a própria pessoa sabe seguir, como, por exemplo, se ela é mais velha ou mais nova que outra pessoa. Mas o Comparator é uma regra que você cria fora da pessoa.**

**Então, o Comparator precisa de uma função chamada compare(). Essa função compara duas coisas e diz se uma é maior, igual ou menor que a outra.**

**No exemplo que vimos, o Comparator foi usado para comparar pessoas pela idade. Se uma pessoa é mais velha que a outra, ela fica antes na lista. Se elas têm a mesma idade, a ordem não muda. E se uma pessoa é mais nova que a outra, ela fica depois na lista.**

1. Para implementar o corpo do Comparator empregou-se uma *feature* conhecida por classe anônima. Explique esta funcionalidade do Java e sua aplicabilidade.

**R: Uma classe anônima em Java é uma maneira de criar uma classe sem fornecer um nome explícito para ela. Em vez de definir uma classe em um arquivo separado, você pode declará-la e usá-la diretamente onde for necessária, fornecendo sua implementação diretamente no local em que é utilizada.**

**A aplicabilidade das classes anônimas é bastante ampla e elas são frequentemente utilizadas em situações onde uma implementação de interface ou uma subclasse é necessária apenas para uma única ocasião.**

**No contexto do exemplo, uma classe anônima foi utilizada para fornecer uma implementação do método compare() da interface Comparator. Esta classe anônima foi criada no local em que o Comparator foi passado como argumento para o método sort() da classe Collections, permitindo assim a definição rápida e concisa de uma regra de comparação personalizada para a ordenação da lista de pessoas.**