PROGRAMAÇÃO I

Aula 09_1 Java

Vinicius Bischoff

Herança

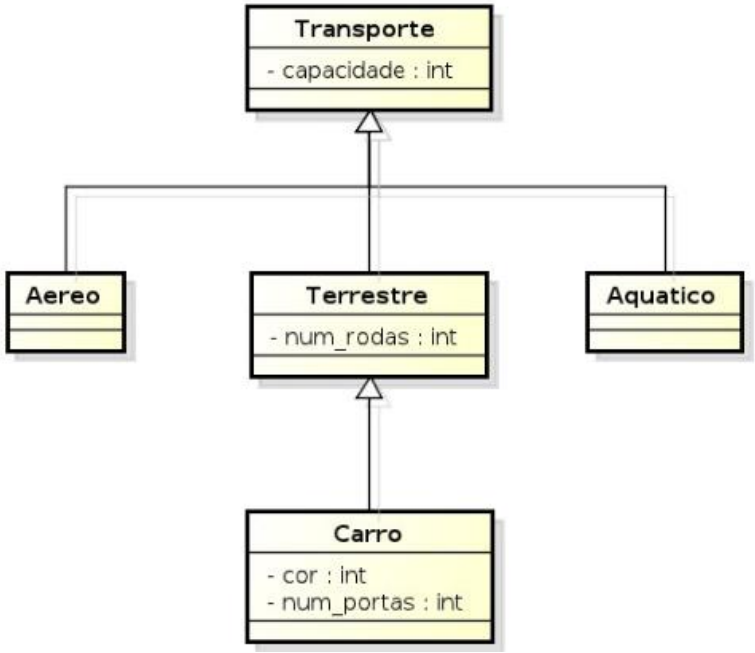
construtor - super()

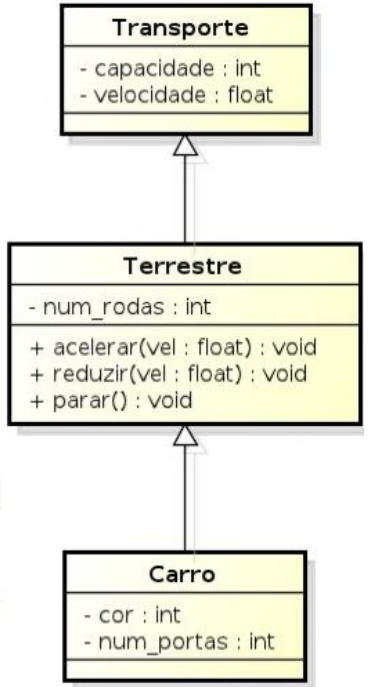
Sobrecarga

Sobreescrita

Representação da herança:

Uma seta com um triângulo branco na ponta, apontando da subclasse para a sua classe mãe.





A herança em Java é um conceito que permite a criação de classes a partir de outras classes já existentes, com o **objetivo de reutilizar o código e evitar a duplicação de código**.

A classe existente é chamada de "classe base" ou "superclasse", enquanto a nova classe criada é chamada de "subclasse".

A subclasse herda todos os membros (métodos e atributos) da superclasse e pode adicionar seus próprios membros ou substituir os membros herdados.

Isso permite que a subclasse tenha comportamento específico além do comportamento herdado da superclasse.

Para criar uma subclasse, usamos a palavra-chave **extends** seguida pelo nome da superclasse.

Por exemplo:

```
public class Animal {
	public void fazerBarulho() {
		System.out.println("O animal está fazendo barulho.");
	}
}
```

```
public class Cachorro extends Animal {
	public void latir() {
		System.out.println("O cachorro está latindo.");
	}
}
```

O construtor super() é usado em Java para chamar o construtor da superclasse de uma subclasse. O construtor da superclasse é responsável por inicializar os membros da superclasse antes que os membros da subclasse sejam inicializados.

Quando criamos uma subclasse, é necessário que ela chame o construtor da superclasse antes de iniciar sua própria inicialização. Isso garante que os membros da superclasse sejam inicializados corretamente antes que a subclasse seja inicializada.

O uso do super() é obrigatório em casos onde a superclasse não possui um construtor padrão (sem argumentos). Em casos onde a superclasse possui um construtor padrão, o super() pode ser omitido, pois o Java irá chamá-lo automaticamente.

Por exemplo, considere a seguinte hierarquia de classes:

```java
public class Animal {
private String nome;
        public Animal(String nome) {
                this.nome = nome;
        }
        // getters e setters
}
```

```java
public class Cachorro extends Animal {
        private String raca;

        public Cachorro(String nome, String raca) {
                super(nome); // chamada ao construtor da superclasse com um argumento
                this.raca = raca;
        }
        // getters e setters
}
```

o

UNISINOS
DESAFIE O AMANHÃ.

A **sobrecarga** em Java é um recurso que permite **definir métodos com o mesmo nome**, mas **com diferentes parâmetros**. Isso significa que você pode ter vários métodos com o mesmo nome, mas que recebem diferentes tipos e/ou números de argumentos.

Quando se trata de herança em Java, a sobrecarga é uma ferramenta poderosa que permite adicionar comportamento específico em cada subclasse, **sem afetar a superclasse**.

Cada subclasse pode ter seus próprios métodos sobrecarregados, que são independentes dos métodos da superclasse.

A sobrecarga em herança é uma ferramenta poderosa em Java que permite adicionar comportamento específico em cada subclasse, sem afetar a superclasse. No entanto, é importante usá-la com cuidado e de forma adequada para evitar problemas de design e desempenho.

```
public class Animal {
        public void fazerBarulho() {
                System.out.println("O animal está fazendo barulho.");
        }
}
```

```
public class Cachorro extends Animal {
        public void fazerBarulho(String som) {
                if (som.equals("au")) {
                        System.out.println("O cachorro está latindo!");
                } else {
                        System.out.println("Som inválido para cachorro!");
                }
        }
}
```

Ops... E o super() ???

```
public class Gato extends Animal {
        public void fazerBarulho(String som) {
                if (som.equals("miau")) {
                        System.out.println("O gato está miando!");
                }  else {
                        System.out.println("Som inválido para gato!");
                }
        }
}
```

```
public class ExemploSobrecarga {
        public static void main(String[] args) {
                Cachorro cachorro = new Cachorro();
                cachorro.fazerBarulho("au");
                cachorro.fazerBarulho("miau");

                Gato gato = new Gato();
                gato.fazerBarulho("miau");
                gato.fazerBarulho("au"); } }
```

A **sobrescrita** de método em Java é um recurso que permite uma classe filha fornecer uma implementação diferente de um método já definido na classe pai.

Para sobrescrever um método na classe filha, **é preciso usar a mesma assinatura** do método da classe pai, ou seja, **mesmo nome e mesmos parâmetros**, mas pode-se fornecer uma implementação diferente.

O modificador de acesso do método da classe filha deve ser igual ou mais acessível que o modificador de acesso do método da classe pai.

```
public class Animal {
        public void fazerBarulho() {
                System.out.println("O animal está fazendo barulho.");
        }
}
```

```
public class Cachorro extends Animal {
        public void fazerBarulho() {
                System.out.println("O cachorro está latindo!");
        }
}
```

```
public class Gato extends Animal {
        public void fazerBarulho() {
                System.out.println("O gato está miando!");
        }
}
```

```
public class Exemplo {
        public static void main(String[] args) {
                Animal animal1 = new Cachorro();
                animal1.fazerBarulho(); // imprime "O cachorro está latindo!"

                Animal animal2 = new Gato();
                animal2.fazerBarulho(); // imprime "O gato está miando!"
        }
}
```

De fato, sobrecarga de método e sobrescrita de método têm algumas semelhanças, mas há diferenças importantes entre elas.

A sobrecarga de método em Java permite que você defina vários métodos com o mesmo nome, mas com parâmetros diferentes. Por exemplo:

```
public class Calculadora {
        public int somar(int a, int b) {
                return a + b;
        }
        public double somar(double a, double b) {
                return a + b;
        }
        public int somar(int a, int b, int c) {
                return a + b + c;
        }
}
```

Neste exemplo, a classe **Calculadora** define três métodos diferentes chamados **somar**, que aceitam diferentes tipos e números de parâmetros. Esse é um exemplo de sobrecarga de método, pois estamos definindo vários métodos com o mesmo nome, mas com parâmetros diferentes.

```
public class Animal {
        public void emitirSom() {
                System.out.println("O animal está emitindo um som.");
        }
}


public class Cachorro extends Animal {
        public void emitirSom() {
                System.out.println("O cachorro está latindo!");
        }
}
```

Neste exemplo, a classe **Cachorro** estende a classe **Animal** e sobrescreve o método **emitirSom**, fornecendo uma implementação diferente. Esse é um exemplo de sobrescrita de método, pois estamos fornecendo uma implementação diferente de um método que já foi definido na classe pai.

Resumindo, a principal diferença entre sobrecarga de método e sobrescrita de método é que a sobrecarga envolve definir vários métodos com o mesmo nome, mas com parâmetros diferentes, enquanto a sobrescrita envolve fornecer uma implementação diferente de um método que já foi definido na classe pai.

Exemplo:

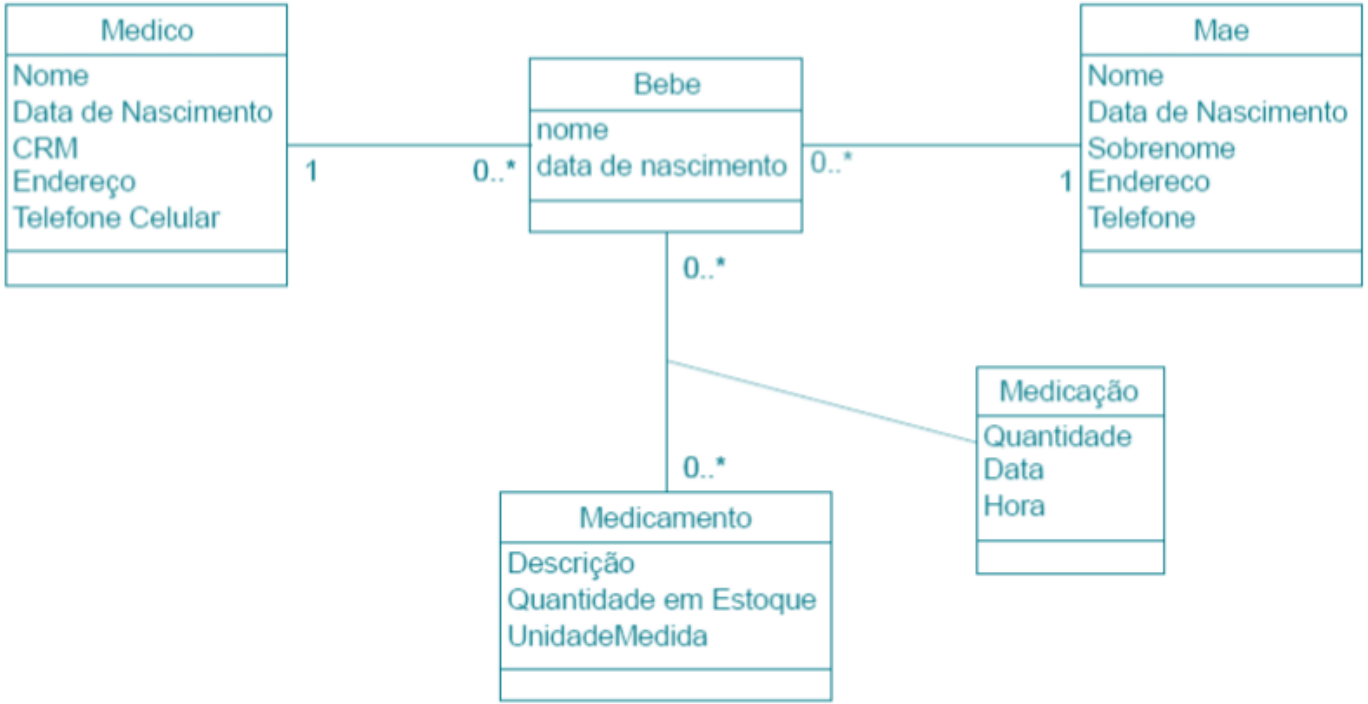
O berçário deseja controlar suas tarefas.

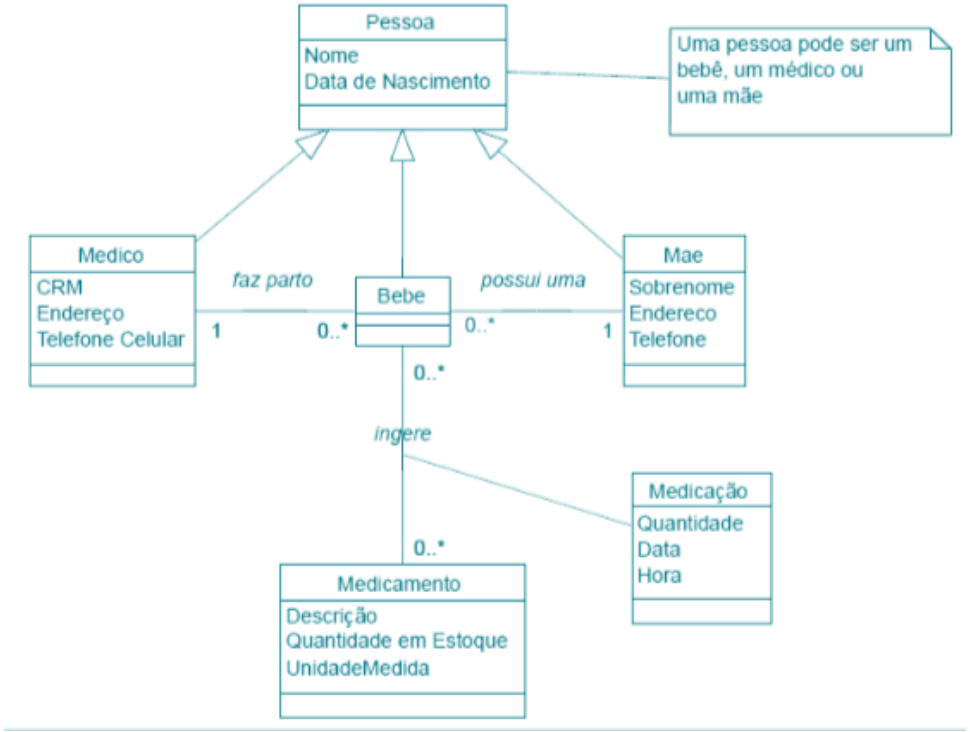
Para isso, é necessário manter um cadastro para os bebês, contendo nome, médico que fez seu parto, sua mãe e data de nascimento.

Para os médicos é necessário saber o nome, data de nascimento, CRM, endereço e telefone celular.

Para as mães dos bebês, é necessário manter informações como nome, sobrenome, data de nascimento, endereço e telefone.

Além disso, é necessário manter um controle dos medicamentos ingeridos pelos bebês no berçário. Sobre os medicamentos, é necessário manter descrição, quantidade em estoque e unidade de medida.
Um bebê pode tomar vários medicamentos, assim como um medicamento pode ser dado para vários bebês. Quando um bebê toma uma medicação, ainda é importante saber a quantidade, o dia e a hora do medicamento


Medico
Nome
Data de Nascimento
CRM
Endereço
Telefone Celular
1
0..*
Bebe
nome
data de nascimento
0..*
1
Mae
Nome
Data de Nascimento
Sobrenome
Endereco
Telefone
0..*
Medicação
Quantidade
Data
Hora
0..*
Medicamento
Descrição
Quantidade em Estoque
UnidadeMedida


Pessoa
Nome
Data de Nascimento
Uma pessoa pode ser um bebê, um médico ou uma mãe
Medico
CRM
Endereço
Telefone Celular
faz parto
1
0..*
Bebe
possui uma
0..*
1
Mae
Sobrenome
Endereco
Telefone
0..*
ingere
Medicação
Quantidade
Data
Hora
0..*
Medicamento
Descrição
Quantidade em Estoque
UnidadeMedida

public, **private** e **protected** são modificadores de acesso em Java e determinam como os membros de uma classe (atributos e métodos) podem ser acessados por outras classes ou objetos.

public indica que o membro é acessível por qualquer classe ou objeto, ou seja, não há restrições de acesso. Por exemplo, se um atributo ou método for declarado como **public**, qualquer classe ou objeto poderá acessá-lo e chamá-lo.

private indica que o membro só pode ser acessado dentro da própria classe. Ou seja, somente os métodos da própria classe podem acessar esse atributo ou método. Por exemplo, se um atributo ou método for declarado como **private**, somente a própria classe pode acessá-lo e chamá-lo.

protected indica que o membro pode ser acessado pela própria classe, por classes no mesmo pacote e por subclasses, mesmo que estejam em pacotes diferentes. Ou seja, o acesso não é livre como no caso do **public**, mas é mais flexível do que o **private**. Por exemplo, se um atributo ou método for declarado como **protected**, somente a própria classe, classes no mesmo pacote e subclasses poderão acessá-lo e chamá-lo.

LocalDate é uma classe em Java que representa uma data (ano, mês e dia) sem levar em consideração o fuso horário. Ela faz parte da API de datas do pacote **java.time**, introduzido no Java 8.

Para criar um objeto **LocalDate**, podemos usar o método estático **of** da classe **LocalDate** passando o ano, o mês e o dia como parâmetros. Por exemplo:

```
LocalDate hoje = LocalDate.of(2023, 5, 3); // cria um objeto LocalDate para representar o dia 3 de maio de 2023
```

A documentação oficial da API de datas do Java (**java.time**) pode ser encontrada no site da Oracle:

https://docs.oracle.com/en/java/javase/16/docs/api/java.base/java/time/package-summary.html

Em Java, **enum** é uma palavra-chave utilizada para declarar um tipo de dado que representa um conjunto fixo de constantes. Por exemplo, você pode usar **enum** para declarar os dias da semana, os meses do ano, as unidades de medida, entre outros. Cada valor do **enum** é representado por um objeto da classe **Enum**.

Ao definir um **enum**, você especifica os possíveis valores que ele pode ter. Por exemplo

```
public enum DiaDaSemana {
        SEGUNDA_FEIRA,
        TERCA_FEIRA,
        QUARTA_FEIRA,
        QUINTA_FEIRA,
        SEXTA_FEIRA,
        SABADO,
        DOMINGO
}
```

```
DiaDaSemana hoje = DiaDaSemana.SEGUNDA_FEIRA;

if (hoje == DiaDaSemana.DOMINGO) {
        System.out.println("Hoje é dia de descanso!");
}
else {
        System.out.println("Hoje é dia de trabalho."); }
```

Pode utilizar os valores do **enum** em sua aplicação como se fossem constantes. Por exemplo:

Polimorfismo

Refere à capacidade de objetos de diferentes classes poderem ser tratados de maneira semelhante, ou seja, um objeto de uma classe pode ser usado como se fosse um objeto de outra classe, desde que essas classes tenham uma relação de herança ou implementem a mesma interface.

O polimorfismo em Java é implementado através de dois conceitos:

- sobrescrita de métodos (ou overriding), e
- sobrecarga de métodos (ou overloading).

A sobrescrita de métodos ocorre quando uma classe filha redefine um método já existente na classe pai, de forma que o método da classe filha **tenha o mesmo nome, mesma assinatura e mesma visibilidade do método da classe pai**.

Isso permite que o método da classe filha seja chamado no lugar do método da classe pai, quando um objeto da classe filha é usado.

A sobrecarga de métodos ocorre quando uma classe tem dois ou mais métodos com o mesmo nome, **mas com diferentes parâmetros**.

Isso permite que a mesma operação seja realizada em diferentes tipos de dados, sem precisar criar vários métodos com nomes diferentes.

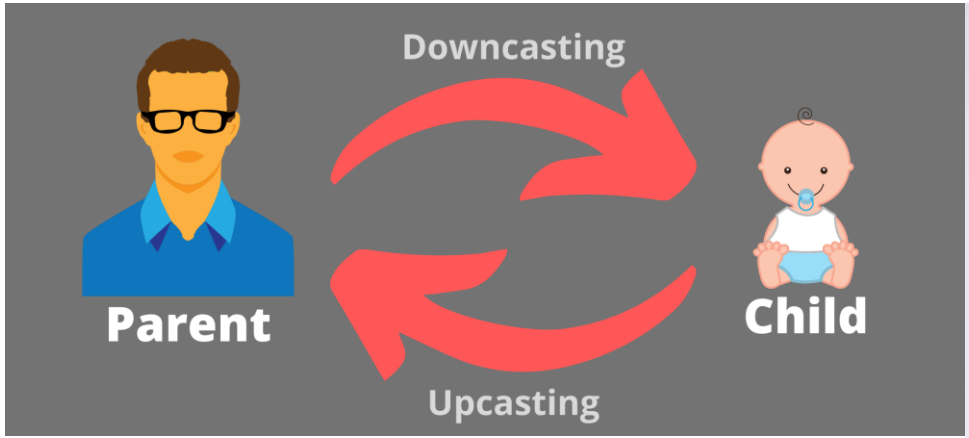
Polimorfismo

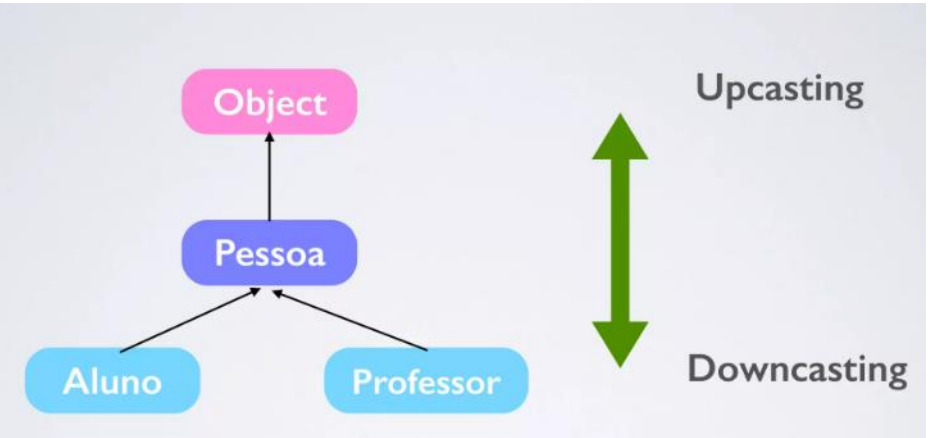
O polimorfismo é um dos pilares da programação orientada a objetos e pode ser aplicado em vários padrões de projeto. Algumas das principais vantagens do polimorfismo na programação de padrões de projeto são a flexibilidade, a reutilização de código e a facilidade de manutenção.

- Padrão Decorator
- Padrão Factory
- Padrão Strategy

Polimorfismo

Conversão de objetos (castings)
Upcasting e Downcating





Upcasting é um processo em que um objeto de uma classe derivada (subclasse) é atribuído a uma variável de referência de sua classe base (superclasse).

Quando isso acontece, o objeto mantém suas características específicas da classe derivada, mas pode ser tratado como se fosse um objeto da classe base.

```
class Animal {
        public void emitirSom() {
                System.out.println("Som do animal");
        }
}
```

```
class Cachorro extends Animal {

        @Override public void emitirSom() {
                System.out.println("Au au");
        }
}
```

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

        Animal animal = new Cachorro(); //upcasting
        animal.emitirSom(); // imprime "Au au"
}
```

Downcasting é o processo de converter uma referência de uma classe base (superclasse) em uma referência de uma classe derivada (subclasse). É o oposto do upcasting, que converte uma referência de uma classe derivada em uma referência de uma classe base.

```
class Animal {
        public void emitirSom() {
                System.out.println("Som do animal");
        }
}
```

```
class Cachorro extends Animal {

        @Override public void emitirSom() {
                System.out.println("Au au");
        }
        public void brincar() {
                System.out.println("O cachorro está brincando");
        }
}
```

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

        Animal animal = new Cachorro(); //upcasting
        animal.emitirSom(); // imprime "Au au"

        Cachorro cachorro = (Cachorro) animal; // downcasting
        cachorro.brincar(); // imprime "O cachorro está brincando"
}
```

O downcasting pode ser perigoso, pois se o objeto referenciado pela classe base não for realmente uma instância da classe derivada, ocorrerá uma exceção em tempo de execução.

Por isso, é importante ter cuidado ao utilizar o downcasting e sempre verificar se o objeto é realmente uma instância da classe derivada antes de realizar a conversão.

O operador **instanceof** é um operador em Java que permite verificar se um objeto é uma instância de uma determinada classe ou de uma classe derivada (subclasse) dessa classe.

O operador retorna um valor booleano, **true** se o objeto for uma instância da classe especificada ou de uma classe derivada dela, e **false** caso contrário.

O uso do operador **instanceof** é útil em situações em que é necessário verificar o tipo de um objeto antes de realizar uma operação específica.

Por exemplo, quando se deseja fazer um downcasting de um objeto de uma classe base para uma classe derivada, é importante verificar se o objeto é realmente uma instância da classe derivada antes de realizar a conversão, a fim de evitar exceções em tempo de execução.

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

        Animal animal = new Cachorro();
// cria um objeto da classe Cachorro e atribui a uma variável de referência do tipo Animal

        if (animal instanceof Cachorro) {
        // verifica se o objeto é uma instância de Cachorro
                Cachorro cachorro = (Cachorro) animal; // downcasting
                cachorro.brincar(); // imprime "O cachorro está brincando"
        } else {
                System.out.println("O objeto não é uma instância de Cachorro"); }}
```

Na função **main()**, criamos um objeto da classe **Animal** e chamamos o método **emitirSom()**, que imprime "Som do animal".

Em seguida, verificamos se o objeto **animal** é uma instância da classe **Cachorro** usando o operador **instanceof**.

Como o objeto é um **Animal** e não um **Cachorro**, a condição é falsa e o programa imprime "O objeto não é uma instância de Cachorro".

Obrigado.