Seção 13: Herança e Polimorfismo

Herança:

- Herança é um mecanismo que permite que uma classe herde **atributos (dados)** e **métodos (comportamentos)** de outra classe.
- Ela promove o **reuso de código**, permitindo que subclasses aproveitem tudo que está definido na superclasse, além de poderem incluir suas próprias características.

Vantagens:

- **Reuso de código:** Não é necessário reescrever atributos e métodos que já estão na superclasse.
- Facilita o uso de Polimorfismo.

Sintaxe:

```
class SubClasse extends SuperClasse {
    // atributos e métodos adicionais
}
```

- A palavra-chave extends indica que uma classe está herdando de outra.

Uso do `super`

- A palavra-chave `super` serve para:
  - **Chamar o construtor da superclasse.**
  - **Acessar métodos ou atributos da superclasse (quando necessário).**
- É uma forma de reaproveitar a lógica da superclasse sem repetir código.

Exemplo:

```
public BusinessAccount(Long number, String holder, Double balance, Double loanLimit) {
    super(number, holder, balance);
    this.loanLimit = loanLimit;
}
```

Modificador de Acesso `protected`

- Permite que um atributo ou método:
  - Seja acessível na **própria classe**.
  - Seja acessível por qualquer **subclasse**, mesmo que estejam em pacotes diferentes.
  - Seja acessível por outras classes do **mesmo pacote**.

Conceitos Importantes Relacionados à Herança

- **Herança é uma relação do tipo "é um".**
  - Ex.: Uma `BusinessAccount` **é uma** `Account`.
- **Generalização e Especialização:**
  - `Account` → **Generalização** (classe genérica).
  - `BusinessAccount` → **Especialização** (classe específica).
- **Superclasse e Subclasse:**
  - `Account` → **Superclasse** ou **Classe Base**.

- `BusinessAccount` → **Subclasse** ou **Classe Derivada**.

- **Extensão:**
  - A subclasse estende a superclasse, ou seja, **adiciona funcionalidades** ao que já existe.
- **Herança é uma associação entre classes e não entre objetos.**
  - Na herança, quando você instancia uma subclasse, você cria **um único objeto**, que contém tanto os membros da superclasse quanto os da subclasse.
  - Isso é diferente de composição, onde são criados dois objetos associados.

Upcasting e Downcasting

Upcasting

Converter um objeto da **subclasse para a superclasse**.

- **Características:**

  Conversão **automática** (implícita).

  Segura. Nunca gera erro.

  Usado para **generalizar** objetos.

- **Exemplo:**

```
Account acc = new BusinessAccount(1002, "Maria", 0.0, 500.0);
```

Aqui, `acc` é do tipo `Account`, mas internamente é um `BusinessAccount`.

Downcasting?

Converter um objeto da **superclasse para a subclasse**.

- **Por que não é automático?**

  Porque **nem todo** `Account` **é um** `BusinessAccount`. É necessário garantir que o objeto realmente é da subclasse.

- **Características:**

  Conversão **manual** (explícita) → exige casting.

  Risco: se o objeto não for da subclasse, gera erro em tempo de execução (`ClassCastException`).

- **Exemplo seguro:**

```
Account acc = new BusinessAccount(1002, "Maria", 0.0, 500.0);
BusinessAccount bacc = (BusinessAccount) acc;
bacc.loan(100.0);
```

- **Exemplo com erro:**

```
Account acc = new SavingsAccount(1004, "Bob", 0.0, 0.01);
BusinessAccount bacc = (BusinessAccount) acc; // ERRO em tempo de execução
```

Como evitar erro no Downcasting?

- Usando `instanceof` para testar se o objeto é da subclasse antes de fazer o cast.
- **Exemplo seguro:**

```
if (acc instanceof BusinessAccount) {
    BusinessAccount bacc = (BusinessAccount) acc;
    bacc.loan(100.0);
}
```

Só faz o cast se for seguro.

Evita erro de `ClassCastException`.

Sobrescrita de Métodos (`@Override`)

- Permite redefinir um método da superclasse na subclasse.
- Útil para alterar comportamentos herdados.

Exemplo:

Na classe `Account`, o saque tem taxa.

Na classe `SavingsAccount`, sobrescrevemos para não cobrar taxa.

Sintaxe:

```
@Override
public void withdraw(double amount) {
    balance -= amount;
}
```

Sem `@Override` funciona, mas usar é boa prática para garantir que está sobrescrevendo corretamente.

Uso do `super`

- Acessa métodos ou construtores da superclasse.
- Permite reaproveitar o comportamento e apenas complementar.

Exemplo de método:

```
@Override
public void withdraw(double amount) {
    super.withdraw(amount); // saque normal com taxa
    balance -= 2.0;         // taxa extra
}
```

Exemplo de construtor:

```
blic SavingsAccount(int number, String holder, double balance, double interestRate) {
    super(number, holder, balance);
    this.interestRate = interestRate;
}
```

Palavra-chave `final`

1. `final` em Classes

- Uma classe declarada como `final` **não pode ser herdada**.
- Exemplo:

```
public final class SavingsAccount extends Account {
    ...
}
```

- Isso impede que alguém crie, por exemplo, `SavingsAccountPlus` herdando de `SavingsAccount`.

---

2. `final` em Métodos

- Um método declarado como `final` **não pode ser sobrescrito** em subclasses.

- Exemplo:

```
@Override
public final void withdraw(double amount) {
    ...
}
```

- Isso protege o método de alterações em subclasses, evitando mudanças no comportamento.

---

3. Quando Usar

- **Regra de negócio:** Quando deseja garantir que uma classe não seja extendida ou que um método não seja modificado.
- **Organização:** Evita sobreposições múltiplas que podem gerar confusão e erros.
- **Proteção:** Garante a integridade do comportamento original da classe ou método.

Observações

- Classes muito usadas como `String` são `final`. Isso garante segurança, integridade e melhora a performance em tempo de execução.
- A utilização de `final` pode gerar ganhos de performance em operações internas da JVM, como operações de reflexão (ex.: serialização e desserialização).

Polimorfismo

O que é Polimorfismo?

- Polimorfismo significa "**muitas formas**".
- Permite que variáveis de um **tipo mais genérico (superclasse)** possam armazenar **objetos de subclasses diferentes**, permitindo que o mesmo método tenha **comportamentos diferentes dependendo do objeto real**.

Funcionamento

- Variáveis de mesmo tipo podem apontar para objetos de classes diferentes.
- A decisão de **qual método executar acontece em tempo de execução**, de acordo com o tipo real do objeto.

Exemplo prático

- Suponha:

```
Account x = new Account(1001, "Alex", 1000.0);
Account y = new SavingsAccount(1002, "Maria", 1000.0, 0.01);
```

- Ambas as variáveis são do tipo `Account`.
- Quando chamamos:

```
x.withdraw(50.0); // Executa o método de Account (com taxa de 5)
y.withdraw(50.0); // Executa o método sobrescrito em SavingsAccount (sem taxa)
```

- Resultado:
  - `x` → saldo 945 (1000 - 50 - 5)
  - `y` → saldo 950 (1000 - 50)

Classes Abstratas

O que são Classes Abstratas?

- São classes que **não podem ser instanciadas diretamente**.
- Servem como **modelos genéricos** para outras classes.
- A declaração é feita com a palavra-chave `abstract`:

```
public abstract class Account {
    // ...
}
```

Finalidade das Classes Abstratas

- **Garantir herança total:** somente suas **subclasses concretas** (não abstratas) podem ser instanciadas.
- Impede a criação de objetos de uma classe que serve apenas como modelo genérico.

Exemplo Prático

- Cenário: Suponha que no negócio bancário não existam contas do tipo "comum", apenas **Conta Poupança** e **Conta Empresarial**.
- Torna-se útil impedir a instanciação direta da classe `Account`, forçando o uso de subclasses específicas.

Ao declarar:

```
public abstract class Account { ... }
```

→ Se tentar fazer:

```
Account acc = new Account(); // ERRO
```

→ O compilador bloqueia, pois `Account` é abstrata.

Vantagens:

1. **Reutilização de código:**
    - Evita duplicação de atributos e métodos comuns (`number`, `holder`, `balance`, `deposit()`, etc.) em todas as subclasses.
2. **Polimorfismo:**
    - Permite tratar diferentes tipos de conta de forma genérica.
    - Exemplo: criar uma **lista de contas** que aceita tanto `SavingsAccount` quanto `BusinessAccount`.

```
List<Account> list = new ArrayList<>();
list.add(new SavingsAccount(...));
list.add(new BusinessAccount(...));
```

→ Isso permite, por exemplo, **somar os saldos de todas as contas**, ou **fazer depósitos em lote**, sem precisar saber o tipo específico de cada conta.

Observações Importantes

- No **UML**, uma classe abstrata é representada com o nome em **itálico**.
- Ao transformar uma classe em abstrata, qualquer tentativa de criar um objeto diretamente dela gera erro de compilação.

Métodos Abstratos

Definições

- **Método abstrato**:
  - Declarado com `abstract` e **não possui implementação** (corpo).
  - Exemplo:

```
public abstract double area();
```

- **Classe abstrata**:
  - Deve conter pelo menos um método abstrato.
  - **Não pode ser instanciada** diretamente (é um modelo para subclasses).
  - Exemplo

```
public abstract class Forma {
    private Color color;
    public abstract double area();
}
```

Motivação para Métodos Abstratos

- **Problema**:
  - Em uma classe genérica (ex: `Forma`), alguns métodos não podem ter implementação única (ex: `area()`), pois dependem da subclasse (círculo, retângulo, etc.).
- **Solução**:
  - Declarar o método como abstrato na superclasse e **obrigar as subclasses** a implementá-lo.

4. Vantagens

- **Polimorfismo**:
  - Permite tratar objetos de subclasses como instâncias da superclasse (`Forma`), chamando `area()` de forma uniforme.
  - Exemplo:

```
List<Forma> formas = new ArrayList<>();
formas.add(new Retangulo(4, 5));
formas.add(new Circulo(3));
for (Forma forma : formas) {
    System.out.println("Área: " + forma.area());
}
```

- **Organização**:
  - Garante que todas as subclasses implementem métodos essenciais.

5. Observações

- **Classes abstratas vs. Interfaces**:
  - Ambos podem ter métodos abstratos, mas classes abstratas podem ter atributos e métodos concretos.
  - Uma classe pode herdar apenas uma classe abstrata, mas pode implementar múltiplas interfaces.
- **Regras**:
  - Se uma classe herda um método abstrato e **não é abstrata**, deve implementar todos os métodos abstratos.
  - Métodos abstratos **não podem ser** `private` (precisam ser sobrescritos).