Este material foi construído pelo Professor **Valdemar Lorenzon Junior** (Lorenzon), trazendo adaptações de material de própria autoria ao longo de anos de dedicação e trabalhos com pesquisa, desenvolvimento e docência com OO, recuperação de cursos certificados SUN e Oracle.

É um material teórico/prático enriquecido com atualizações de conteúdo e certificações de updates, podendo trazer alguma incompatibilidade em algum momento que pode ser corrigida quando observada. O membro pode ser acessado por classes no mesmo pacote.

Sumário

Herança e Derivação de Classes

Uma das características mais interessantes da OO é a capacidade de **derivação** de novas classes a partir de classes já existentes.

Usando este mecanismo, uma **subclasse herda** os membros de sua **superclasse**.

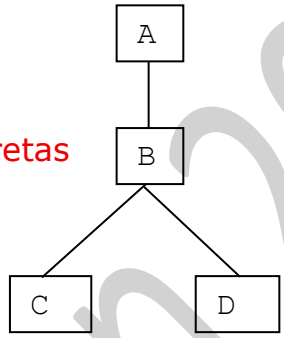
Deste modo pode-se criar uma **hierarquia de classes** onde cada classe tem uma superclasse e pode ter várias subclasses. Por exemplo, dado a seguir, uma classe `B` é **subclasse** de uma classe `A` e **superclasse** das classes `C` e `D`.

As relações existentes entre estas classes podem ser apresentadas em forma de árvore, como é mostrado a seguir:

Interfaces ou classes abstratas

Classes abstratas ou classes concretas

Classes concretas



Ainda neste exemplo, a classe A é superclasse de todas as outras classes nesta hierarquia, mas, no entanto, ela é **superclasse direta** apenas da classe B.

O maior valor do mecanismo de herança está no fato de permitir o **reuso** de software em classes descendentes. A herança prepara o terreno para o polimorfismo.

A herança está intimamente ligada a combinação com padrões e SOLID LSP.

Classes e Métodos Abstratos

A classe abstrata é uma ferramenta que se utiliza para declarações comuns, mas que não se pode instanciar diretamente. A classe abstrata é a primeira forma de obrigar a criação de herança e implementação de métodos definidos, mas que não foram codificados, propositalmente. É utilizada quando desejamos criar uma estrutura base que **não pode ser instanciada** diretamente, mas que serve como modelo para outras classes.

Para isso, há um modificador para a classe e para os métodos, que gera uma classe abstrata. Desta forma as classes abstratas não podem ser instanciadas sem que sejam derivadas e que sejam "completadas" dentro de uma hierarquia a ser construída pelo programador que irá decompor a herança das classes pré criadas.

Uma classe abstrata:

- É declarada com o modificador **`abstract`**;
- Deve possuir pelo menos um método declarado com `abstract`. Estes métodos não possuem corpo;
- Não pode ter instâncias (objetos), mas pode ter referências no código;
- Uma subclasse de uma classe abstrata também será abstrata até que implemente todos os métodos abstratos.

Exemplo:

```
public abstract class A {
    public abstract void f(); //não há corpo { }
}
... em uso:
A a; // OK, a é apenas uma referência
a = new A(); // ILEGAL: A não pode ter objetos, não é classe concreta
```

Uma subclasse de uma classe abstrata também é abstrata, mesmo que não seja indicada como tal. Deixará de ser assim que se faça a criação do corpo de todos os métodos abstratos da superclasse para então poder-se criar objetos para ela.

Interfaces

A interface é como uma classe, mas com a característica de abstração, ou seja, não é concreta. Uma **interface** define um **contrato** que classes devem cumprir.

Uma interface:

- É declarada com a palavra **interface**;
- Representa um contrato, um trato a ser seguido pelas classes que a implementarem;

Uma classe poderá implementar múltiplas interfaces;

Por que usar Interfaces?

Interfaces permitem definir um conjunto de métodos sem implementá-los, permitindo uma visão mais abstrata do comportamento esperado.

Uma classe pode implementar múltiplas interfaces, o que possibilita a implementação de múltiplos comportamentos.

Interfaces são fundamentais em padrões de design como Strategy e Observer.

Até o Java 7, interfaces continham apenas **métodos abstratos**. Uma interface então era uma classe abstrata "pura", mas passou a ter possibilidade de implementações.

O primeiro contato com interfaces era em Java os atributos de uma interface eram final e os métodos todos abstract e public, não precisando nenhuma explicitação ao caso.

A partir do Java 8, passaram a poder conter **métodos default e static**. Desde o Java 9, também podem conter **métodos privados ou default**. No caso de default, a palavra default deve ser explicitada.

A sugestão é de implementar com a explicitação dos modificadores para melhor entendimento de código. Senão:

- Atributos são sempre **public static final**;
- Os métodos serão sempre **public abstract**;

No caso de métodos static e private, estes não podem ser abstract na interface, por obviedade: métodos privados seria apenas da interface e precisam de corpo. Métodos static devem ser públicos e, seguem este mesmo pensamento de necessidade de corpo.

```
public interface MinhaInterface {
      public     static     final     int        ATRIBUTO = 10;
      public                abstract  void       metodo1();
      private                          int        metodo2() { ; } //com corpo
      public     static                void       metodo3() { ; } //com corpo
      default                          void       metodo4() { ; } //com corpo
}
```

```
public class MinhaClasse implements MinhaInterface {
      @Override
      public void metodo1() {
         System.out.println("Implementação de metodo1");
         System.out.println("Valor do atributo: " + ATRIBUTO);
      }
      // OBS: metodo2() é privado na interface.
      // Para chamar metodo3() (estático), fazemos via nome da interface:
      public void chamarMetodo3() {
         MinhaInterface.metodo3();     }}
```

Comparativo Classe Abstrata × Interface

Aspecto	Classe Abstrata	Interface
Palavras-chave	abstract class	interface
Métodos implementados	Pode ter métodos concretos e abstratos	Métodos abstratos por padrão; desde Java 8 pode ter default, static; desde Java 9, private
Atributos / estado	Pode ter campos de instância não estáticos, variáveis privadas, protected etc.	Só pode ter constantes: public static final implicitamente
Herança / implementação	Uma classe só pode extends uma superclasse abstrata	Uma classe pode implements várias interfaces
Construtor	Pode ter construtor	Não pode ter construtor
Acesso aos modificadores	Pode usar public, protected, private	Métodos abstratos são public; constantes são public static final
Uso típico	Compartilhar código comum entre subclasses, fornecer modelo base, código parcial	Definir contrato, polimorfismo, definir serviços que várias classes usam

Herança: Definição e construção de Subclasses

A sintaxe utilizada na declaração de uma subclasse de outra classe é:

```
class <NomeSubclasse> extends <NomeSuperClasse> {
        ...
}
```

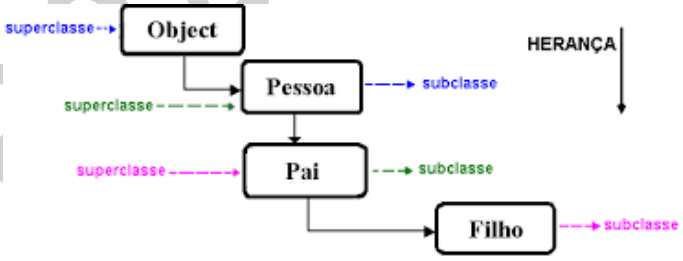
onde, < **NomeSubclasse** > é a classe sendo declarada; < **NomeSuperClasse** > é a classe da qual a classe sendo declarada é derivada.

Em Java é suportável apenas herança simples, não múltipla, mesmo que uma hierarquia pareça ser multiplicidade de herança!

Para interfaces, a implementação é livre para a quantidade de interfaces, colocando as interfaces em lista.

Na herança em Java, qualquer definição de classe que não estenda uma superclasse, automaticamente está estendendo a classe **Object** do Java.

Ao herdar de uma classe abstrata, se ela não se tornar concreta, deverá também ser explicitada abstrata.



Hierarquias e Controle de Acesso

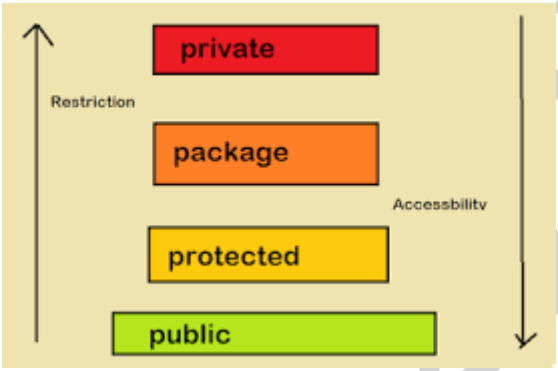
Além de ter seus próprios membros, uma subclasse também possui *todos* os membros de suas superclasses (direta ou indiretas). Isto é, uma classe herda todos os membros de suas superclasses.

Apesar disso, apenas os membros **public** e **protected** de classes superiores podem ser acessados. Isto é, ***os membros privados (private) de uma superclasse não podem ser acessados numa subclasse***.

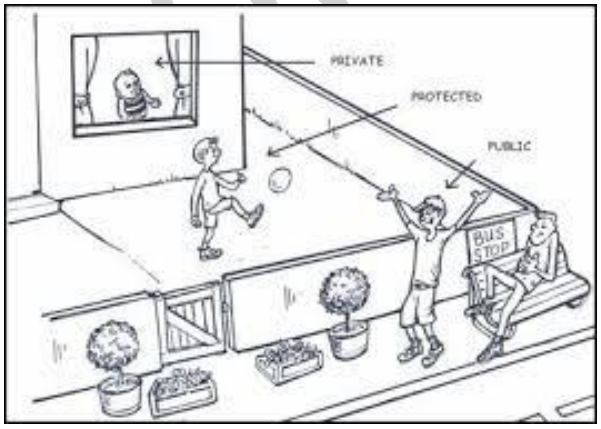
Em Java, apesar de protected estar disponível, para garantir maior coesão do encapsulamento, default e protected "não devem" ser usados para atributos e sim apenas protected, não default, podem ser usados quando necessários para métodos.

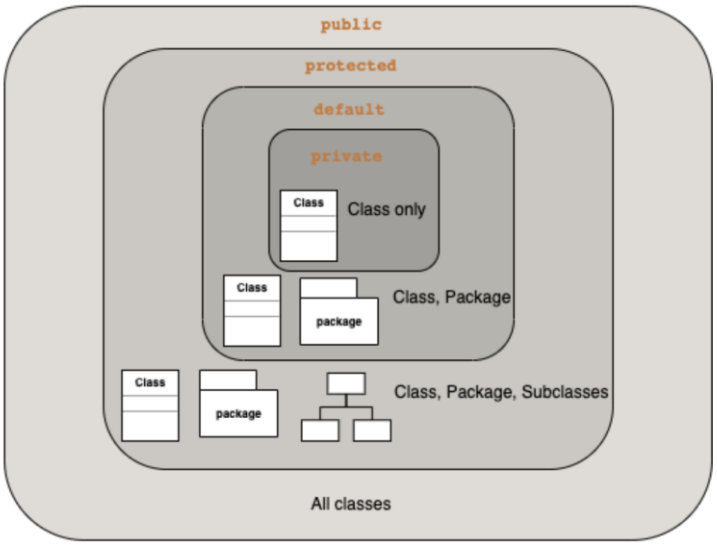
Já um objeto pode possuir membros **protected** e **private pela** classe referenciada, mas só lhe é permitido acesso aos membros **public** da classe à qual pertence.

Este princípio é dado pelo encapsulamento. Se existe a necessidade de acesso a um membro **private**, ao invés de colocá-lo com **public**, deve-se fazer esse acesso através de métodos (getters e setters, como já estudado no encapsulamento).



Modifier	Class	Package	Subclass	World
public	✔	✔	✔	✔
protected	✔	✔	✔	✘
*no modifier**	✔	✔	✘	✘
private	✔	✘	✘	✘





Considere o seguinte exemplo de derivação:

```
class A {
    public     int a1;
    protected  int a2; //observe o uso de protected, evite para atributos
    private    int a3; //continua…
    public     void metodoA()
    {
         b1 = 2;    // ILEGAL a classe A não pode acessar
                    // NENHUM membro da classe B filha de A
    }
}
```

```
class B extends A    {// Classe B é subclasse da classe A
    public     int b1;
    protected  int b2;
    private    int b3;
    public     void metodoB()
    {
        a1 = 10;    // OK
        b2 = a2;    // OK
        b2 = a3;    // ILEGAL o membro a3 é private em A (não vê)
    }
}
```

Apesar do exemplo apenas considerer atributos, a ideia permanece idêntica para membros de comportamento: métodos.

```
public class Teste {
     public static void main(String [] s) {
        A a;
        B b;
        a.b1 = 0;   // ILEGAL instâncias de A não podem acessar
                    // NENHUM membro de B
        b.a1 = 0;   // OK o membro a1 de A é public em B
        b.a2 = 0;   // ILEGAL o membro a2 de A é protected em B
        b.a3 = 0;   // ILEGAL o membro a3 de A é private em B
     }
}
```

Construtor de Superclasse

Quando um objeto de uma subclasse é criado, o construtor default da superclasse é chamado antes do construtor da classe.

Se assim desejar, o programador pode evitar que o construtor default da superclasse seja invocado e invocar um construtor mais adequado utilizando a palavra reservada **super**.

Quando uma criação de um construtor particular da classe é criado, este "mata" o construtor padrão, então deverá ser invocado o novo construtor criado na classe descendente.

Por exemplo:

```
class A {
    protected int a1;
    protected int a2;
    public A (int x, int y) {
        a1 = x;
        a2 = y;
    }
}
```

```
class B extends A   { // Classe B é derivada da classe A
    private int b1;


    // Definição (ERRADA) do construtor da classe B
    public B(int x) {
        a1 = 10;
        a2 = 20;
        b1 = x;
    }
}
```

```
public class TesteHerança {
    public static void main(String [] s) {
        B b;
       //...
    }
}
```

Se tentar compilar o programa esquematizado no último exemplo, você obterá uma mensagem de erro do compilador informando que ele não conseguiu encontrar nenhum construtor default para a classe `A`. Isto ocorre porque na ausência de chamada explicita de um construtor específico da superclasse através de super na subclasse, o compilador chama automaticamente o construtor default da superclasse.

Aqui, o erro ocorre porque a superclasse `A` não possui construtor default (***o compilador provê auto-aticamente construtor default apenas para classes que não têm nenhum construtor explicitamente declarado***). A classe `A` tem um construtor com dois argumentos. Portanto, o compilador não provê nenhum construtor default para esta classe.

Uma forma de corrigir o erro no programa do último exemplo é provendo um construtor default para a classe `A`. Isto poderia ser feito, por exemplo, da seguinte maneira:

```
public A() {
            a1 = 0;
            a2 = 0;
}
```

Esta modificação ou qualquer outra que acrescente um construtor default para a classe A resolve o erro de compilação apresentado anteriormente, mas não constitui, entretanto, uma solução adequada para o problema de inicialização dos membros da superclasse A por meio do construtor da subclasse B. Isto é, mesmo é quando possível (A superclasse pode, por exemplo, ter sido criada por outra pessoa e talvez você não tenha acesso ao seu código-fonte para ser capaz de alterá-lo) alterar uma classe de modo a permitir uma derivação conveniente a partir dela, esta prática é contrária aos princípios de reuso de software e deve ser evitada.

Portanto, a melhor solução seria redefinir o construtor da classe B para chamar o construtor da classe A utilizando a palavra reservada super, como mostrado a seguir:

```
public B (int x) {
        // Definição (CORRETA) do construtor da classe
        super (10, 20);
        b1 = x;
}
```

Métodos e Classes Declarados com final

Uma classe declarada com **`final`** não pode ter nenhuma subclasse. **É estéril.**

Construtores

Construtores não são considerados membros e, portanto, não sofrem ocultação nem overriding.

Aproveitando, um método declarado com **final** não pode sofrer **overriding**.

Overrinding é um uso básico do que chamaremos de uma notação para resolver uma abstração numa derivação de classes.

É possível que tenhamos vários construtores numa classe, porém com assinaturas diferentes.

A Superclasse Object

Toda classe (com uma única exceção) é derivada de outra. A Exceção: classe `Object`.

Então, uma declaração como esta:

```
class A {
)
```

Pode ser entendida como:

```
class A extends Object {
}
```

Toda classe que não é explicitamente derivada de outra classe é implicitamente derivada da classe `Object`.

A classe `Object` é uma classe de onde de qualquer forma todas as classes serão indiretamente derivadas dela.

A `Object` é projetada para fornecer implementações padrão de métodos fundamentais (como `equals()`, `hashCode()`, e `toString()`), e essas implementações podem ser usadas diretamente pelas subclasses ou sobrescritas conforme necessário. Além disso, como `Object` é a superclasse de todas as outras classes, ela precisa ser concreta para garantir que todas as classes em Java possam herdar seus métodos e funcionalidades.

equals(): É usado para verificar **igualdade lógica** entre dois objetos. Por padrão, na classe Object, ele compara se duas referências apontam para o **mesmo objeto na memória** (igualdade de identidade).

haschCode(): Gera um **número inteiro (hash)** que representa o objeto. Esse número é usado por estruturas de dados como HashMap e HashSet para armazenar e localizar objetos de forma eficiente. O contrato em Java exige que: se a.equals(b) == true, então a.hashCode() == b.hashCode().

toString(): Fornece uma representação **em forma de texto** de um objeto. A implementação padrão da classe Object retorna algo como: NomeDaClasse@hashCodeEmHexa.

Um exemplo de reuso destes métodos e seus propósitos pode ser observado a seguir:

```
class Cliente {


    private final String cpf;
    private String nome;
```

```
    public Cliente(String cpf, String nome) {
        this.cpf = cpf;
        this.nome = nome;
    }
    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (!(o instanceof Cliente)) return false;
        Cliente c = (Cliente) o;
        return cpf.equals(c.cpf); // igualdade por identidade
    }
    @Override
    public int hashCode() {
        return cpf.hashCode();
    }
    @Override
    public String toString() {
        return "Cliente{cpf='" + cpf + "', nome='" + nome + "'}";
    }

}
```

O Override destes métodos:

- equals: permite verificar se dois clientes são "iguais" pelo CPF (mesmo que sejam instâncias diferentes);
- hashCode: garante que, se o cliente for usado em um HashSet, ele não será duplicado caso já exista outro com o mesmo CPF;
- toString: facilita imprimir dados do cliente de forma legível, em vez de exibir apenas o endereço de memória.

Injeção por associações (composição ou injeção de dependência)

Até aqui vimos que a herança é uma maneira de reutilizar código: uma subclasse herda atributos e métodos de sua superclasse. Nem sempre herança é a melhor solução. Em muitos casos, usamos a associação de objetos (um objeto contém ou utiliza outro) para montar comportamentos de forma mais flexível.

Conceito de injeção

Em vez de uma classe herdar características de outra, ela recebe (ou injeta) objetos prontos de outras classes para compor seu funcionamento.

Isso é chamado de injeção de dependência: a dependência necessária vem "de fora", em vez de estar fixada na hierarquia de herança.

O lema aqui é: "prefira composição a herança", princípio comum em design de software.

As diferenças

A herança facilita o reuso, mas gera acoplamento forte. Injeção por associação: aumenta a flexibilidade e o reuso, mas exige mais planejamento e compreensão da arquitetura.

O ideal é combinar: usar herança quando existe uma relação clara de "é um" (is-a), e usar associação quando existe uma relação de "tem um" (has-a) ou de "usa um serviço".

Muitos padrões de projeto existem exatamente para evitar o acoplamento rígido da herança.

- Herança **fixa** comportamento: útil quando há uma relação clara de "é um" (is-a);
- Injeção/Composição **flexibiliza** comportamento: usada em padrões como Strategy, Decorator, Observer, Bridge, Composite.

Já, alguns padrões fazem sentido quando existe uma relação natural de especialização:

- Template Method: esqueleto de algoritmo;
- **Factory** Method: criação de objetos delegada a subclasses;
- Abstract Factory: famílias de objetos relacionados;
- Prototype:clonagem baseada em superclasse comum;
- Interpreter: gramática definida por hierarquia;
- State (clássico): estados como subclasses.

Um exemplo de observação sobre uso de injeção ao invés de herança pode ser a ideia de conta bancária e divisão entre conta correte e conta poupança. A percepção é clara para uso de herança. Já que corrente e poupança são contas (“is a”).

Ao pensarmos em implementar uma capacidade de enviar notificações por movimentações nas contas, não faríamos herança e sim uma ideia paralela de abstração própria, seguindo hierarquia e padrões paa depois incluir a notificação como uma injeção nas contas, a partir de conta, para que suas subclasses herdem a funcionalidade e não tenham que implementar ou tenhamos que implementar esta característica dentro da hierarquia das contas.

A injeção é então uma composição!

Resumo Final – Abstrações, Herança, Interfaces e Injeção

1. Classes Abstratas

Servem como **modelo base**;

Não podem ser instanciadas;

Podem ter **métodos abstratos** (sem corpo, obrigando implementação) e **métodos concretos** (comportamento já definido);

Reforçam a ideia de "**modelo incompleto**" que será especializado por subclasses;

Usam a palavra reservada **abstract** na classe e no método que não conterá corpo.

2. Interfaces

Representam **contratos puros**: definem o que uma classe deve fazer, sem impor como;

Uma classe pode implementar **múltiplas interfaces**;

Desde Java 8, interfaces podem ter **métodos default e static**; desde Java 9, também **métodos private**;

São fundamentais para o **polimorfismo por contrato**;

Usa-se a palavra reservada **interface** e **implements** para codificar e implementar interfaces.

3. Classes Concretas

São as que podem ser instanciadas;

Implementam uma ou mais **interfaces** ou **herdam de uma classe abstrata ou concreta (não final)**;

Representam possibilidade de instanciação de objetos reais no sistema, com **estado e comportamento completos**;

Faz-se uso das palavras reservadas **extends** para herdar e **new** para instanciar.

4. Herança

Relação do tipo "**é um**" (*is-a*).

Permite reuso de código por especialização de classes;

Métodos fazem sobreposição, ocultação com override;

Porém, gera **acoplamento forte**: mudanças na superclasse afetam todas as subclasses

Deve ser usada com cautela, apenas quando há hierarquia clara;

Palavras reservadas como **extends**, **abstract**, **super**, **@Override** são utilizadas no contexto da herança.

5. Injeção por Associações (Composição)

Relação do tipo "**tem um**" (*has-a*);

Em vez de herdar comportamento, a classe **recebe dependências externas** (injeção);

Reduz o acoplamento e aumenta a flexibilidade;

Neste contexto, a injeção é um atributo de composição na classe.

6. Padrões de Projeto

Baseados em herança:

Template Method, Factory Method, Abstract Factory, Prototype, Interpreter;

Baseados em injeção/composição:

Strategy, Decorator, Observer, Bridge, Composite, Dependency Injection;

Muitos padrões modernos favorecem **injeção/composição**, seguindo o princípio *"prefira composição à herança"*.

Conclusão sobre construção de abstrações

Interfaces representam uma barreira no aprendizado do Java: parece que estamos escrevendo um código que não serve pra nada, já que teremos essa linha (a assinatura do método) escrita nas nossas classes implementadoras.

Essa é uma maneira errada de se pensar.

O objetivo do uso de uma interface é deixar seu código mais flexível e possibilitar a mudança de implementação sem maiores traumas. Não é apenas um código de prototipação, um cabeçalho!

Os mais radicais dizem que toda classe deve ser "interfaceada", isto é, só devemos nos referir a objetos através de suas interfaces. Se determinada classe não tem uma interface, ela deveria ter.

O uso de interfaces em vez de herança é amplamente aconselhado. Você pode encontrar mais informações sobre o assunto nos livros Design Patterns, Refactoring e Effective Java. No livro Design Patterns, logo no início, os autores citam 2 regras "de ouro". Uma é "evite herança, prefira composição" e a outra, "programe voltado a interface e não à implementação".

O uso de interfaces é vasto em coleções, o que melhora o entendimento do assunto. O exemplo da interface Comparable também é muito esclarecedor, onde enxergamos o reaproveitamento de código através das interfaces, além do encapsulamento. Para o método Collections.sort(), pouco importa quem vai ser passado como argumento. Para ele, basta que a coleção seja de objetos comparáveis. Ele pode ordenar Elefante, Conexao ou ContaCorrente, desde que implementem Comparable.

Tratativas sobre declaração de nomes

Existem convenções de nomenclatura para interfaces, abstrações e classes concretas. Podendo iniciar as Interfaces com "I", as classes abstratas com sufixo "Base", "Abstract". As classes concretas sendo substantivos claros.

Cada comunidade pode criar convenções para tornar o código previsível. Em Java, o trivial e recomendável é confiar no contexto.

Tipo	Função	Padrão de Nome	Exemplos da API Java	Exemplo Didático
Interface	Define um contrato (o que pode ser feito).	Nome substantivo/adjetivo, sem prefixo I.	List, Set, Comparable, Runnable	Transportavel (algo que pode ser transportado)
Classe Abstrata	Dá uma base genérica (pode ter métodos prontos e abstratos).	Muitas vezes usa prefixo/sufixo Abstract ou Base.	AbstractList, AbstractMap, InputStream, OutputStream	Veiculo (define atributos genéricos como peso, velocidade)
Classe Concreta	Implementa a lógica real. Pode ser instanciada.	Nome claro, substantivo que represente a implementação. Pode usar Impl se houver ambiguidade.	ArrayList, HashMap, TreeSet, FileInputStream	Carro, Caminhao, Bicicleta

Interface : A forma de bolo sem fundo, a capacidade genérica: **Preparavel**.

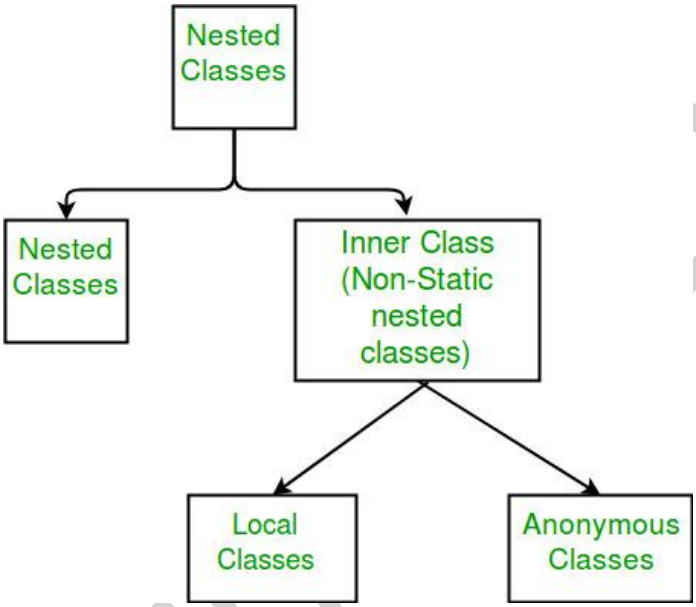
Abstrata: A construção do fundo da forma, uma base comum, mas ainda com furos: **ReceitaDeBolo** implements Preparavel.

Concreta: A forma com fundo sem furos, a instância real: **BoloDeChololate** extends ReceitaDeBolo.

A criação de classes internas: inner classes

Quando uma classe é criada dentro de uma estrutura, classe ou função, ela é definida de inner class. Ela existe **associada** a uma classe externa, e é usada quando a relação entre as duas é muito próxima. Permite **agrupar classes logicamente**, aumentar a **encapsulação** e facilitar o **acesso a membros privados** da classe externa.

Existe 4 possibilidades de criação de inner classes em Java:



Inner Class (não estática, padrão)

- Criada dentro de outra classe, mas fora de métodos;
- Precisa de uma instância da classe externa para ser criada.

```
class Externa {
    private String mensagem = "Olá";

    class Interna {
        // acessa direto membros privados
        public void exibir() {
            System.out.println(mensagem);
        }
    }
}
```

Static Nested Class

- Definida como **static** dentro de outra classe;
- Não precisa de instância da classe externa.

```
class Externa {
    static class Interna {
        void exibir() {
            System.out.println("Sou uma inner estática");
        }
    }
}
```

```
public class Teste {
    public static void main(String[] args) {
        Externa.Interna obj = new Externa.Interna(); // não precisa de
instância de Externa
        obj.exibir();
    }
}
```

Local Inner Class

- Definida dentro de um **método**;
- Só pode ser usada dentro do escopo desse método.

```
class Externa {
    void metodo() {
        class Local {
            void exibir() {
                System.out.println("Inner local em método");
            }        }
        Local local = new Local();
        local.exibir();     }}
```

Anonymous Inner Class

- Criada **sem nome**, geralmente para implementar uma interface ou sobrescrever métodos rapidamente;
- Muito usada com **callbacks** ou **eventos**.

```
interface Saudacao {
    void falar();
}
```

```
public class Teste {
    public static void main(String[] args) {
        Saudacao s = new Saudacao() {
            @Override
            public void falar() {
                System.out.println("Olá de uma inner anônima!");
            }
        };
        s.falar();
    }
}
```

As classes internas então podem ser utilizadas na programação e ao se deparar com elas, é importante o conhecimento de leitura e contextualização. Em resumo, podemos definir:

- **Inner class padrão**: quando a interna depende da externa;
- **Static nested class**: agrupamento lógico, sem depender da externa;
- **Local inner class**: classe de uso restrito dentro de um método;
- **Anonymous inner class**: implementação rápida, geralmente em eventos.

Em padrões de projetos, podemos destacar o uso de inner classes para suas implementações de forma mais coesa e conveniente:

- Static inner class: muito usada no **Builder**.
- Inner class padrão: útil no **Iterator** ou **State**.
- Anonymous inner class: comum em **Observer** (listeners, callbacks) e **Command**.

Exercícios

1. Um sistema precisa padronizar como diferentes módulos exibem mensagens ao usuário. Para isso, será criada apenas uma interface que servirá como contrato para qualquer módulo que precise mostrar informações na tela.

Defina uma interface chamada **Exibivel**. Esta interface deve conter:

- Um atributo constante que represente o prefixo padrão das mensagens (por exemplo, "MSG:");
- Um método privado com corpo, usado apenas dentro da interface, para formatar uma string (por exemplo, acrescentar colchetes ou símbolos em volta do texto);
- **Um método default**, que use o método privado para exibir uma mensagem já formatada;
- Pelo menos dois métodos abstratos que obriguem as classes que implementarem a interface a mostrar mensagens de formas diferentes: simples e em destalhe.

Regras:

- Não crie classes concretas agora: apenas a interface;
- Escolha nomes claros para os métodos, para que o contrato seja fácil de entender.

2. Imagine que um sistema precisa lidar com diferentes formas de pagamento (cartão, boleto, pix, etc.). Para que o sistema seja flexível, vamos criar uma interface que represente a ideia de "pagar".

Crie uma interface chamada **Pagamento**. Essa interface deve ter apenas um método abstrato chamado processarPagamento(double valor).

Crie uma classe concreta chamada **PagamentoCartao**, que implemente a interface e, ao receber o valor, mostre uma mensagem no console indicando que o pagamento com cartão foi realizado.

Crie outra classe chamada **PagamentoPix**, também implementando a interface, exibindo mensagem diferente.

3. Um sistema de cadastro precisa diferenciar entre pessoas físicas e pessoas jurídicas, mas ambas compartilham características comuns (como nome e endereço). Para evitar repetição de código, será criada uma classe abstrata Pessoa com os atributos e comportamentos básicos, e duas classes filhas (PessoaFisica e PessoaJuridica) que adicionam informações específicas.

Crie uma classe abstrata chamada Pessoa, contendo: Atributos comuns: nome (String) e endereco (String).

- Um método abstrato chamado exibirDocumento(), que deve ser implementado pelas subclasses.
- Um método concreto chamado exibirDados(), que mostre nome e endereço.

Crie uma classe concreta chamada PessoaFisica, que herda de Pessoa, e adicione: cpf (String). Implemente o método exibirDocumento() mostrando o CPF.

Crie uma classe concreta chamada PessoaJuridica, que herda de Pessoa, e adicione: cnpj (String). Implemente o método exibirDocumento() mostrando o CNPJ.

Nas duas classes filhas, sobrecarregue o método exibirDados(),para que além dos dados herdado, exiba os dados específicos.

Crie uma classe de teste (App ou Main) que instancie uma PessoaFisica e uma PessoaJuridica, chamando seus métodos.

4. Um sistema de zoológico precisa representar diferentes tipos de animais. Todos os animais têm atributos em comum (nome e idade) e um comportamento esperado: emitir um som característico.

Deve ser definida uma interface para garantir que todo animal possa “fazer som”.

Uma classe abstrata concentrará os atributos comuns.

Cada animal concreto (cachorro, gato, galinha, caprino, bovino) terá sua própria implementação do som.

Crie as seguintes classes concretas que herdem de Animal:

Cachorro: implementa fazerSom() exibindo "Au Au!".

Gato: implementa fazerSom() exibindo "Miau!".

Galinha: implementa fazerSom() exibindo "Cócórócó!".

Caprino: implementa fazerSom() exibindo "Béééé!".

Bovino: → implementa fazerSom() exibindo "Muuuu!".

Crie uma classe App para testar.

Estudo de caso com aplicação de herança

Vamos criar um estudo de caso sobre herança em Java usando um serviço de banco como exemplo. Neste caso, vamos considerar que um banco oferece diferentes tipos de contas, como Conta Corrente e Conta Poupança, que compartilham algumas características comuns, mas também têm comportamentos específicos.

Quando mencionamos um compartilhamento comum, imaginamos uma derivação de maior alto nível para os casos de conta corrente e conta poupança!

Consideraremos então no nosso estudo uma classe pai das classes que vamos desenvolver, abstrata:

```
public interface OperacoesBancarias { //sem explicitações de modificador
    void sacar(double valor);
    void depositar(double valor);
    String getNumeroConta();
    double getSaldo();
}
```

```
public abstract class Conta implements OperacoesBancarias {
     private String numeroConta;
     private double saldo;

     public Conta(String numeroConta) {
       this.numeroConta = numeroConta;
       this.saldo = 0.0;
     }
     @Override
     public void depositar(double valor) {
       saldo += valor;
     }
```

```
    @Override
    public String getNumeroConta() {
      return numeroConta;
    }
    @Override
    public double getSaldo() {
      return saldo;
    }
    //Já que a classe não implementa o método, precisa repassar para
    //as subclasses, e para ser abstrata, precisa do método abstrato.
    //Então aqui estamos apenas criando uma intermediação e repassando
    //a necessidade de repassas a "implements" herdada.
    @Override
    public abstract void sacar(double valor);
}
```

Algumas considerações para que possamos então fazer a aplicação do conceito de herança para as classes ContaCorrente e ContaPoupança:

- Interface: contrato comum (OperacoesBancarias);
- Classe abstrata: fornece **parte da implementação** e obriga subclasses a completar (sacar);
- Subclasses: implementam diferenças (corrente vs. poupança).

- A conta corrente, conterá uma taxa de operação para efetuar-se um saque, deduzindo além do valor sacado o valor da taxa de operação. A taxa de operação é um percentual pequeno, como 0,01 % do valor. Para efetuar o saque, sempre deverá ter saldo para o saque com a taxa embutida;

- Ainda, diferentemente da conta poupança, a conta correte levaria em consideração o limite da conta para saque, o que não acontece na poupança;

- Os construtores de cada classe deverão ser construídos, e como indicado deverão chamar o construtor da classe pai internamente em seu código. Para isso, a palavra **super** é como a ideia de **this** para o estado interno do objeto. Super refere-se a classe pai imediatamente acima na herança de classes. Deve-se respeitar a assinatura do construtor invocado quando é efetuada a chamada de super;

- Ambos métodos sacar devem validar com estruturas de fluxo condicional se há saldo suficiente para o saque ou apenas dizer "Saldo insuficiente para a operação";

- Cada classe deve ser construída em um arquivo .Java distinto, pertencendo todas as classes ao pacote banco;

- Para que seja possível sobrepor o método abstrato nas classes derivadas, devemos seguir a mesma assinatura do método a ser sobreposto. A notação **@Override** deve ser utilizada antes a criação do método e seu corpo para simples indicação.

    - **@Override** indica que estamos sobrescrevendo um método na derivação, para sabermos que não é um método da classe atual e sim um método derivado sobreposto, Ex:

```
    public class ContaCorrente extends Conta {
        // ...
    @Override
    public void sacar(double valor) {
        // Implementação específica para ContaCorrente
    }
}
```