Interface gráfica do usuário, Texto, Aplicativo

O conteúdo gerado por IA pode estar incorreto.

**a. java.util.Set**

* **O que é:** É uma **interface** que representa um conjunto.
* **Características:**
  + Não permite elementos duplicados.
  + Não mantém ordem (exceto implementações específicas como LinkedHashSet e TreeSet).
* **Exemplo:**
* Set<String> nomes = new HashSet<>();
* nomes.add("Lucas");
* nomes.add("Lucas"); // Ignorado, pois não permite duplicados
* nomes.add("Ana");
* System.out.println(nomes); // [Lucas, Ana]
* ❌ **Por que não é a resposta:** O Set **não associa chave com valor**. Ele só armazena valores únicos.

**b. java.util.Vector**

* **O que é:** É uma classe antiga (pré-Java 2), parecida com o ArrayList, mas **sincronizada**.
* **Características:**
  + Permite elementos duplicados.
  + Mantém a ordem de inserção.
  + Hoje em dia, é pouco usado; ArrayList é preferível.
* **Exemplo:**
* Vector<String> lista = new Vector<>();
* lista.add("A");
* lista.add("B");
* System.out.println(lista); // [A, B]
* ❌ **Por que não é a resposta:** O Vector é apenas uma lista, não associa chaves a valores.

**c. java.util.ArrayList**

* **O que é:** É a implementação mais comum da interface List.
* **Características:**
  + Baseada em array dinâmico.
  + Permite duplicados.
  + Acesso rápido por índice.
* **Exemplo:**
* ArrayList<String> nomes = new ArrayList<>();
* nomes.add("Lucas");
* nomes.add("Lucas"); // Permitido
* nomes.add("Maria");
* System.out.println(nomes.get(1)); // Lucas (pelo índice)
* ❌ **Por que não é a resposta:** O ArrayList indexa elementos **por posição**, não por chave.

**d. java.util.HashMap ✅ (CORRETA)**

* **O que é:** É uma classe que implementa a interface Map.
* **Características:**
  + Armazena pares **chave → valor**.
  + Cada chave é única.
  + Acesso rápido (baseado em hashing).
* **Exemplo:**
* HashMap<Integer, String> alunos = new HashMap<>();
* alunos.put(1, "Lucas");
* alunos.put(2, "Maria");
* alunos.put(3, "Ana");
* System.out.println(alunos.get(2)); // Maria
* ✅ **Por que é a resposta:**  
  Apenas o HashMap (entre as opções dadas) **associa elementos a chaves**. Ele é feito exatamente para isso.

**e. java.util.Key**

* **O que é:** Não existe no pacote java.util.
* ❌ **Por que não é a resposta:** Simplesmente não faz parte do framework Collections.

**Resumo**

* Set → conjunto sem duplicados.
* Vector → lista sincronizada.
* ArrayList → lista dinâmica baseada em índice.
* HashMap → **mapa chave → valor** ✅.
* Key → inexistente.

👉 Por isso, a correta é: **d. java.util.HashMap**.

Texto

O conteúdo gerado por IA pode estar incorreto.

**Por que?**

**I – Falsa.**  
Quem “ordena de fora” é o **Comparator** (objeto externo que compara dois itens por vez, via compare(a,b)).  
A **Comparable** é implementada **na própria classe** para definir a *ordem natural* via compareTo.

class Pessoa implements Comparable<Pessoa> {

int idade;

public int compareTo(Pessoa o) { return Integer.compare(this.idade, o.idade); }

}

// Externo:

Comparator<Pessoa> porNome = Comparator.comparing(p -> p.nome);

**II – Verdadeira.**  
“Os objetos que estiverem sendo ordenados” (isto é, a classe desses objetos) **devem implementar Comparable** quando você quer ordem natural.

**III – Verdadeira.**  
equals pode:

* Comparar **por referência** (comportamento padrão herdado de Object, se não for sobrescrito).
* Comparar **por atributos** (quando a classe **sobrescreve** equals).

class Produto {

String sku;

@Override public boolean equals(Object o) {

if (!(o instanceof Produto p)) return false;

return this.sku.equals(p.sku); // por atributo

}

}

**IV – Falsa (ou incompleta).**  
A ordenação pode usar **compareTo** (quando há Comparable) **ou um Comparator** passado ao algoritmo (Collections.sort(lista, comparator) / list.sort(comparator)). Não é “sempre” via compareTo.

Collections.sort(pessoas); // usa compareTo (Comparable)

Collections.sort(pessoas, porNomeReverso); // usa Comparator externo

👉 Portanto, somente **II e III** estão corretas.

Interface gráfica do usuário, Texto

O conteúdo gerado por IA pode estar incorreto.

A afirmação é **Verdadeiro** ✅

**Explicação:**

* Em **Java**, as operações de entrada e saída (**I/O**) são feitas por meio de **fluxos (streams)**.
* Esses fluxos permitem ler e escrever dados de forma genérica, independentemente do **dispositivo** (arquivo, teclado, rede, memória, etc.).
* Por exemplo, as mesmas classes como InputStream, OutputStream, Reader e Writer podem ser aplicadas a diferentes tipos de dispositivos:
  + FileInputStream → lê de arquivos.
  + ByteArrayInputStream → lê de arrays de bytes em memória.
  + BufferedReader → lê de teclado, arquivos ou rede.

👉 Assim, a abstração de fluxos em Java garante que **o mesmo código de leitura/escrita pode ser usado para diferentes fontes e destinos**, bastando trocar a implementação do fluxo.

Texto

O conteúdo gerado por IA pode estar incorreto.  
  
  
A alternativa correta é **Falso** ❌

**Explicação:**

1. **Nem todos os métodos de uma classe genérica precisam ser genéricos.**
   * Quando criamos uma classe genérica, ela pode ter métodos normais (não genéricos) e também métodos genéricos independentes do tipo da classe.
2. class Caixa<T> {
3. private T valor;
4. public void setValor(T valor) { // usa o tipo genérico T
5. this.valor = valor;
6. }
7. public T getValor() { // também usa o T
8. return valor;
9. }
10. public void imprime() { // método normal, não precisa ser genérico
11. System.out.println("Caixa contém algo.");
12. }
13. public <U> void mostrarOutro(U outro) { // método genérico independente
14. System.out.println("Outro valor: " + outro);
15. }
16. }
17. **Checagem em tempo de compilação, não em tempo de execução.**
    * O Java usa **type erasure (apagamento de tipos)**: as informações de tipos genéricos existem apenas em tempo de compilação, para garantir segurança de tipos.
    * No bytecode, os tipos genéricos não são mantidos (são apagados), portanto **não existe verificação de generics em tempo de execução**.

👉 Por isso, a afirmação está incorreta: nem todos os métodos precisam ser genéricos e a checagem acontece em **tempo de compilação**, não em execução.

Quer que eu monte uma **tabelinha resumida** com as diferenças entre *classe genérica*, *método genérico* e *type erasure* para fixar?

Texto

O conteúdo gerado por IA pode estar incorreto.

**I - A Serialização de um objeto significa que o mesmo será transformado em cadeia de bytes e não poderá ser armazenado em disco, somente transmitido via rede de computadores.**

❌ **Incorreta.**  
A serialização transforma em bytes sim, mas pode ser **armazenado em disco** (ex: ObjectOutputStream em arquivo) ou enviado pela rede. Não é restrito só à rede.

**II - A Serialização pode ser aplicada em qualquer aplicação que trabalhe com fluxo de I/O.**

✅ **Correta.**  
Ela é usada em I/O de arquivos, sockets, etc.

**III - No caso de um objeto ser serializado e for proveniente de uma subclasse, todos os atributos de instância, inclusive os da superclasse, serão serializados.**

Correta, desde que a superclasse seja serializável. Se a superclasse implementar Serializable, seus atributos são serializados junto com os da subclasse. Caso contrário, a superclasse deve ter um construtor sem argumentos para que a desserialização funcione, mas seus atributos não serão serializados.

🔎 Detalhando:

* Se a **superclasse também implementa Serializable**, então os atributos dela **serão serializados normalmente** junto com os da subclasse.
* Se a \*\*superclasse **não implementa Serializable**, então os atributos dela **não são serializados**. Nesse caso, durante a **desserialização**, o Java exige que a superclasse tenha um **construtor padrão (sem argumentos)** para inicializar corretamente os atributos herdados.

📌 Exemplo rápido:

import java.io.\*;

class SuperClasse {

int x = 10; // não é serializável se a superclasse não implementar Serializable

public SuperClasse() {

System.out.println("Construtor da SuperClasse chamado!");

}

}

class SubClasse extends SuperClasse implements Serializable {

int y = 20;

}

public class Teste {

public static void main(String[] args) throws Exception {

SubClasse obj = new SubClasse();

// Serializa

ObjectOutputStream out = new ObjectOutputStream(new FileOutputStream("objeto.ser"));

out.writeObject(obj);

out.close();

// Desserializa

ObjectInputStream in = new ObjectInputStream(new FileInputStream("objeto.ser"));

SubClasse obj2 = (SubClasse) in.readObject();

in.close();

System.out.println("y desserializado = " + obj2.y); // 20

System.out.println("x desserializado = " + obj2.x); // 10? -> NÃO, volta ao valor default (0) se SuperClasse não for Serializable

}

}

👉 Ou seja, a sua frase está perfeita:

* Correta **se a superclasse for Serializable**.
* Caso contrário, os atributos da superclasse não entram na serialização, mas o construtor sem argumentos dela é chamado na hora da desserialização.

**IV - A palavra reservada transient na definição de um atributo da classe no qual um objeto sendo serializado torna-se estático no acesso à informação relacionada a este atributo.**

❌ **Incorreta.**  
transient **não torna o atributo estático**. Ele apenas indica que o campo **não deve ser serializado**.

**V - SerialVersionUID é um número que identifica a versão da classe que foi usada durante o processo de serialização.**

✅ **Correta.**  
Serve para garantir compatibilidade entre versões de classe durante a desserialização.

**✅ Gabarito:**

* I → Falsa
* II → Verdadeira
* III → Falsa
* IV → Falsa
* V → Verdadeira

👉 Alternativa correta: **e. Somente as afirmativas II e V estão corretas.**

Texto

O conteúdo gerado por IA pode estar incorreto.

**a. A classe PrintStream implementa um fluxo de entrada que contem o método read().**  
❌ Incorreta.  
PrintStream é um fluxo de **saída** (ex: System.out). Ele possui métodos print() e println(), **não tem** read().

**b. A classe ByteArrayOutputStream faz com que o fluxo de entrada leia um array de bytes.**  
❌ Incorreta.  
ByteArrayOutputStream é um fluxo de **saída**, que grava dados em um **array de bytes em memória**.  
Quem lê array de bytes é o ByteArrayInputStream.

**c. Os fluxos de bytes fornecem um meio conveniente para somente o tratamento de entrada de bytes.**  
❌ Incorreta.  
Fluxos de bytes (InputStream e OutputStream) permitem tanto **entrada (InputStream)** quanto **saída (OutputStream)**, não apenas entrada.

**d. A classe FilterOutputStream implementa o OutputStream e o InputStream.**  
❌ Incorreta.  
FilterOutputStream herda apenas de **OutputStream**. Ele é usado para adicionar funcionalidades a fluxos de saída. Não implementa InputStream.

**e. Na classe FileInputStream o fluxo de entrada lê de um arquivo.**  
✅ **Correta.**  
FileInputStream é um fluxo de **entrada de bytes** que lê dados diretamente de um **arquivo**.

Exemplo:

FileInputStream fis = new FileInputStream("arquivo.txt");

int dado = fis.read();

**✅ Gabarito:**

**e. Na classe FileInputStream o fluxo de entrada lê de um arquivo.**

Texto

O conteúdo gerado por IA pode estar incorreto.

**a. List lstPessoa = new ArrayList<Pessoa>()**  
❌ Incorreta.  
Não usa generics no lado esquerdo (List sem <Pessoa>), o que gera **warning** de tipo *unchecked*.

**b. List<Pessoa> lstPessoa = new ArrayList (Pessoa)**  
❌ Incorreta.  
A sintaxe está errada, não se passa o tipo Pessoa no construtor de ArrayList. O correto seria new ArrayList<>().

**c. List<> lstPessoa = new ArrayList<>()**  
❌ Incorreta.  
Não existe List<> sem especificar o tipo.

**d. List<Pessoa> lstPessoa = new ArrayList<>()**  
✅ **Correta.**  
Essa é a forma mais recomendada a partir do Java 7, usando o **operador diamante (<>)**, que infere o tipo do lado direito.

List<Pessoa> lstPessoa = new ArrayList<>();

**e. List<Pessoa> lstPessoa = new ArrayList()**  
⚠️ Parcialmente correta mas **não recomendada**.  
Funciona, mas o lado direito sem generics gera **warning** de tipo não verificado (*unchecked*).

**✅ Gabarito:**

**d. List<Pessoa> lstPessoa = new ArrayList<>()**

Texto

O conteúdo gerado por IA pode estar incorreto.

A alternativa correta é **Falso** ❌

**Explicação:**

Na linguagem Java, os fluxos de I/O são divididos em **dois tipos principais**:

1. **Fluxos de bytes** (InputStream e OutputStream)
   * Trabalham com dados binários (imagens, áudios, arquivos binários).
   * Exemplo: FileInputStream, FileOutputStream.
2. **Fluxos de caracteres** (Reader e Writer)
   * Trabalham com dados de texto (caracteres Unicode).
   * Exemplo: FileReader, BufferedWriter.

📌 **Não existe “fluxo automático”** como categoria na API de I/O do Java.

Texto

O conteúdo gerado por IA pode estar incorreto.

**I) “Para ordenar objetos de uma mesma classe, deve-se garantir que tal classe implemente a interface Comparable ou utilize um provider.”**

**✅ Verdadeira.**  
Há **duas** formas canônicas de ordenar:

1. **Ordem natural** com Comparable (critério fixo dentro da classe):

class Pessoa implements Comparable<Pessoa> {

int idade;

@Override

public int compareTo(Pessoa o) {

return Integer.compare(this.idade, o.idade); // ordem natural por idade

}

}

1. **Critérios externos** com Comparator (o “provider”/comparador passado na hora):

Comparator<Pessoa> porNome = Comparator.comparing(p -> p.nome);

pessoas.sort(porNome);

**II) “A implementação do método compareTo varia conforme o contexto/critério em que a ordenação é aplicada.”**

**❌ Falsa.**  
compareTo (de Comparable) define **uma única ordem natural** da classe e **não muda por contexto**.  
Para variar o critério conforme o contexto, usa-se **Comparator**:

pessoas.sort(Comparator.comparingInt(p -> p.idade)); // por idade (externo)

pessoas.sort(Comparator.comparing(p -> p.nome)); // por nome (externo)

Se você precisasse de outro critério com compareTo, teria que **alterar o código da classe** — ele não “varia” dinamicamente.

**III) “No Java, só é possível ordenar classes Wrappers.”**

**❌ Falsa.**  
Qualquer classe pode ser ordenada se:

* implementar Comparable, **ou**
* você fornecer um Comparator.  
  Wrappers (Integer, String etc.) já **têm** Comparable, mas **não são os únicos** que podem ser ordenados.

**IV) “Para ordenarmos uma coleção de objetos, podemos utilizar método estático sort da classe ArrayList.”**

**❌ Falsa.**

* O método **estático** sort está em **Collections.sort(List)**.
* Desde o Java 8, listas têm **método de instância** list.sort(Comparator).  
  ArrayList **não** possui método **estático** sort.

Exemplos corretos:

Collections.sort(pessoas); // usa Comparable (ordem natural)

Collections.sort(pessoas, Comparator.comparing(Pessoa::getNome)); // Comparator

pessoas.sort(Comparator.comparingInt(Pessoa::getIdade)); // método de instância

**Conclusão**

* **I: verdadeira**
* **II: falsa**
* **III: falsa**
* **IV: falsa**

👉 Alternativa certa: **c. Somente a afirmativa I está correta.**

Tabela

O conteúdo gerado por IA pode estar incorreto.

**Gabarito: a. Somente as afirmativas I, II, III e IV estão corretas.**

* **I – Verdadeira.** Threads permitem ao SO escalar execução entre múltiplos processadores/cores, habilitando paralelismo.
* **II – Verdadeira.** Com threads, um programa Java executa duas ou mais tarefas “ao mesmo tempo” (concorrência/possível paralelismo).
* **III – Verdadeira.** Em **single-core**, há *time-sharing*: as threads compartilham a CPU por fatias de tempo.
* **IV – Verdadeira.** O método **run()** é o ponto de entrada da thread (o código executado após start()).
* **V – Falsa.** **isAlive()** apenas indica se a thread ainda está em execução (**boolean**). Prioridade se obtém com **getPriority()**.