Qual a saída?

java

CopyEdit

```
Map<String, Integer> estoque = new HashMap<>();

estoque.put("Café", 10);

estoque.put("Café", 15);

estoque.put("Chá", 5);

System.out.println(estoque.get("Café"));
```

A saída será:

Copiar código

```
15
```

Explicação

Em um `HashMap`, **chaves são únicas**.
Quando você faz:

java Copiar código

```
estoque.put("Café", 10);
estoque.put("Café", 15);
```

O segundo `put` **substitui o valor anterior** associado à chave `"Café"`.

Então o mapa fica assim ao final:

java Copiar código

```
"Café" -> 15
```

```
"Chá"   -> 5
```

Por isso:

java Copiar código

```
System.out.println(estoque.get("Café"));
```

imprime **15** ✅

Na API Java NIO.2 (java.nio.file), qual classe é usada para monitorar alterações em arquivos e diretórios?

Na API **Java NIO.2 (java.nio.file)**, a classe usada para monitorar alterações em arquivos e diretórios é:

✅ `WatchService`

Explicação

O `WatchService` permite observar eventos no sistema de arquivos, como:

- Criação de arquivos (`ENTRY_CREATE`)
- Modificação de arquivos (`ENTRY_MODIFY`)
- Exclusão de arquivos (`ENTRY_DELETE`)

Ele trabalha em conjunto com:

- `Path`
- `WatchKey`
- `StandardWatchEventKinds`

Exemplo simples

java Copiar código

```
WatchService watchService = FileSystems.getDefault().newWatchService();

Path path = Paths.get("C:/pasta");
path.register(
    watchService,
```

```
        StandardWatchEventKinds.ENTRY_CREATE,
        StandardWatchEventKinds.ENTRY_MODIFY,
        StandardWatchEventKinds.ENTRY_DELETE
    );

    while (true) {
        WatchKey key = watchService.take();

        for (WatchEvent<?> event : key.pollEvents()) {
            System.out.println("Evento: " + event.kind());
            System.out.println("Arquivo: " + event.context());
        }

        key.reset();
    }
```

📌 **Resumo:**

A classe correta é `WatchService`.

Considere as seguintes classes Java:

java

CopyEdit

```
abstract class Funcionario {
    private String nome;
    private double salarioBase;

    public Funcionario(String nome, double salarioBase) {
    this.nome = nome;
        this.salarioBase = salarioBase;
```

```
    }

    public abstract double calcularSalario();

    public String getNome() {
        return nome;
    }

    public double getSalarioBase() {
        return salarioBase;
    }
}

class Gerente extends Funcionario {
    private double bonus;

    public Gerente(String nome, double salarioBase, double
bonus) {
        super(nome, salarioBase);
        this.bonus = bonus;
    }
```

```
    @Override
    public double calcularSalario() {
        return getSalarioBase() + bonus;
    }
}

class Desenvolvedor extends Funcionario {
    public Desenvolvedor(String nome, double salarioBase) {
        super(nome, salarioBase);
    }

    @Override
    public double calcularSalario() {
        return getSalarioBase() * 1.1;
    }
}

public class TestePolimorfismo {
    public static void main(String[] args) {
        Funcionario f1 = new Gerente("Alice", 5000.0, 1500.0);
        Funcionario f2 = new Desenvolvedor("Bob", 4000.0);
```

```
        System.out.println(f1.getNome() + ": " +
f1.calcularSalario());

        System.out.println(f2.getNome() + ": " +
f2.calcularSalario());

    }

}
```

❓Pergunta:

Assumindo que todas as classes estão corretamente compiladas e executadas, qual das alternativas abaixo representa a melhor análise sobre o código apresentado, considerando os pilares da orientação a objetos?

A
O método calcularSalario() é um exemplo de polimorfismo em tempo de execução, pois seu comportamento depende do tipo real do objeto referenciado.

B
O método calcularSalario() em Funcionario viola o princípio de encapsulamento por ser abstrato e não permitir implementação padrão.

C
O uso do Funcionario f1 = new Gerente(...) é inválido, pois não é permitido instanciar uma subclasse com referência à superclasse.

D

O método calcularSalario() demonstra polimorfismo em tempo de compilação, pois é decidido em tempo de escrita do código qual método será chamado.

E
O uso de getSalarioBase() nas subclasses viola o encapsulamento, pois métodos getters não são recomendados em herança.

A alternativa **correta é a A** ✅

✅ A) O método `calcularSalario()` é um exemplo de polimorfismo em tempo de execução, pois seu comportamento depende do tipo real do objeto referenciado.

Justificativa

- `Funcionario` define o método **abstrato** `calcularSalario()`.
- As subclasses `Gerente` e `Desenvolvedor` **sobrescrevem** esse método.
- No `main`, as variáveis `f1` e `f2` são do tipo **Funcionario**, mas referenciam objetos de **classes diferentes**.
- Em tempo de execução, a JVM decide qual implementação de `calcularSalario()` será chamada com base no **tipo real do objeto**, e não no tipo da referência.

Isso caracteriza **polimorfismo em tempo de execução (dynamic binding)**.

❌ Por que as outras alternativas estão erradas?

B) Errada
→ Um método abstrato **não viola encapsulamento**. Pelo contrário, ele define um **contrato** para as subclasses.

C) Errada
→ É **totalmente válido** usar uma referência da superclasse para apontar para um objeto da subclasse. Isso é a base do polimorfismo.

D) Errada
→ Polimorfismo em tempo de compilação está relacionado a **sobrecarga de métodos**, não sobrescrita.

E) Errada
→ O uso de `getSalarioBase()` **preserva o encapsulamento**, pois o atributo é `private` e acessado de forma controlada.

📌 Conclusão

O código demonstra corretamente **abstração, herança, encapsulamento e polimorfismo**, sendo a alternativa **A** a análise mais precisa.

Questão 05
Um sistema de agenda recebe datas como strings no formato dd/MM/yyyy. Para converter corretamente essa string em um objeto LocalDate, o desenvolvedor utilizou o seguinte código:

java

CopyEdit

String entrada = "21/04/2025";

LocalDate data = LocalDate.parse(entrada, DateTimeFormatter.ofPattern("dd/MM/yyyy"));

Qual afirmação é verdadeira sobre esse código?

A
O método parse não aceita formatadores

B
A data será convertida com sucesso

C
O tipo DateTimeFormatter pertence ao pacote java.text

D
O código lançará exceção por formato incorreto

E
LocalDate armazena também a hora

A alternativa **correta é a B** ✅

✅ B) A data será convertida com sucesso

Justificativa

- A string `"21/04/2025"` está **exatamente no formato** `dd/MM/yyyy`.
- O método `LocalDate.parse` **aceita um** `DateTimeFormatter` como segundo parâmetro.
- `DateTimeFormatter.ofPattern("dd/MM/yyyy")` pertence ao pacote `java.time.format`.
- `LocalDate` representa **apenas data (ano, mês e dia)**, sem hora.

Portanto, o código executa corretamente e a conversão é bem-sucedida.

❌ Por que as outras alternativas estão erradas?

A) Errada

→ `LocalDate.parse` **aceita sim** formatadores.

C) Errada

→ `DateTimeFormatter` pertence ao pacote `java.time.format`, não `java.text`.

D) Errada

→ O formato está correto, logo **nenhuma exceção será lançada**.

E) Errada

→ `LocalDate` **não armazena hora** (isso seria `LocalDateTime` ou `ZonedDateTime`).

📌 Conclusão

A resposta correta é **B**.

Qual das expressões abaixo valida corretamente uma data no formato dd/MM/yyyy em Regex, considerando apenas o formato (não a validade real dos valores)?

A
^\d{2}/\d{2}/\d{4}$

B
\d{2}-\d{2}-\d{4}

C
(\d){2}/(\d){2}/(\d){4}

D
\d{4}/\d{2}/\d{2}

E
\d\d/\d\d/\d\d\d\d

A alternativa **correta é a A** ✅

✅ A) `^\d{2}/\d{2}/\d{4}$`

Justificativa

Essa expressão:

- `^` → início da string
- `\d{2}` → exatamente **dois dígitos** (dia)
- `/` → barra literal

- `\d{2}` → exatamente **dois dígitos** (mês)
- `/` → barra literal
- `\d{4}` → exatamente **quatro dígitos** (ano)
- `$` → fim da string

Ela valida **somente o formato** `dd/MM/yyyy`, conforme solicitado.

❌ Por que as outras estão incorretas?

B) `\d{2}-\d{2}-\d{4}`
→ Usa **hífen (-)** em vez de barra (/).

C) `(\d){2}/(\d){2}/(\d){4}`
→ Embora funcione tecnicamente, **não é a forma correta/ideal** e pode causar ambiguidades; a versão com quantificadores diretos (`\d{2}`) é a correta e esperada em provas.

D) `\d{4}/\d{2}/\d{2}`
→ Formato `yyyy/MM/dd`, diferente do solicitado.

E) `\d\d/\d\d/\d\d\d\d`
→ Funciona, mas é **menos clara e menos recomendada** do que usar `{}`.

📌 Conclusão

A expressão **mais correta, clara e padrão** é a alternativa **A**.

Questão 07
O que acontece quando se insere duas vezes a mesma chave em um HashMap?

A
O valor anterior é duplicado

B
Os dois valores são armazenados separadamente

C
O HashMap lança exceção

D
A operação é ignorada

E
O valor antigo é substituído pelo novo

A alternativa **correta é a E** ✅

✅ E) O valor antigo é substituído pelo novo

Justificativa

Em um `HashMap`, **não podem existir chaves duplicadas**.

Quando você insere uma chave que **já existe** no mapa usando `put`:

java Copiar código

```
map.put("chave", valorAntigo);
map.put("chave", valorNovo);
```

- O **valor antigo é sobrescrito**
- A chave permanece a mesma
- **Nenhuma exceção é lançada**

❌ Por que as outras alternativas estão erradas?

A) Errada
→ O valor **não é duplicado**.

B) Errada
→ `HashMap` **não armazena dois valores para a mesma chave**.

C) Errada
→ Nenhuma exceção é lançada nesse caso.

D) Errada

→ A operação **não é ignorada**, o valor é atualizado.

📌 Conclusão

A resposta correta é **E**.

Questão 08
O que acontece ao adicionar duas vezes o mesmo objeto a um HashSet, considerando que equals() e hashCode() foram corretamente implementados?

A
O segundo valor é ignorado

B
O HashSet lança exceção

C
O primeiro valor é removido

D
O segundo valor substitui o primeiro

E
Ambos os valores são armazenados

A alternativa **correta é a A** ✅

✅ A) O segundo valor é ignorado

Justificativa

Um `HashSet` :

- **Não permite elementos duplicados**
- Usa os métodos `equals()` e `hashCode()` para determinar se dois objetos são iguais

Se você adiciona duas vezes o **mesmo objeto** (ou dois objetos considerados iguais por `equals()`):

java Copiar código

```
set.add(obj);
set.add(obj);
```

- A primeira inserção é realizada com sucesso
- A segunda **é ignorada**
- Nenhuma exceção é lançada

❌ Por que as outras alternativas estão erradas?

B) Errada

→ `HashSet` não lança exceção por duplicidade.

C) Errada

→ O primeiro valor **não é removido**.

D) Errada

→ Nenhuma substituição ocorre em `HashSet` .

E) Errada

→ `HashSet` **não armazena duplicatas**.

📌 Conclusão

A resposta correta é **A**.

Questão 09

Qual das estruturas abaixo permite elementos duplicados e mantém a ordem de inserção?

A
LinkedHashSet

B
HashMap

C
HashSet

D
TreeSet

E
ArrayList

A alternativa **correta é a E** ✅

✅ E) ArrayList

Justificativa

O `ArrayList`:

- ✅ **Permite elementos duplicados**
- ✅ **Mantém a ordem de inserção**
- É uma implementação da interface `List`

❌ Por que as outras alternativas estão erradas?

A) LinkedHashSet

→ Mantém a ordem de inserção, **mas não permite duplicados**.

B) HashMap

→ Trabalha com pares **chave/valor** e **não permite chaves duplicadas**.

C) HashSet

→ Não mantém ordem de inserção e **não permite duplicados**.

D) TreeSet

→ Mantém ordem **ordenada (natural ou por comparator)**, mas **não permite duplicados**.

📌 Conclusão

A única estrutura que **permite duplicados** e **preserva a ordem de inserção** é o **ArrayList**, alternativa **E**.

Questão 10
Qual das alternativas abaixo cria corretamente um Optional que pode conter um valor nulo

A
Optional.of(valor)

B
Optional.isNullable(valor)

C
Optional.ofNullable(valor)

D
Optional.empty(valor)

E
Optional.nullable(valor)

A alternativa **correta é a C** ✅

✅ C) `Optional.ofNullable(valor)`

Justificativa

- `Optional.of(valor)` ❌ → **lança** `NullPointerException` se `valor` for `null`.
- `Optional.ofNullable(valor)` ✅ → **aceita valores nulos** e cria:
  - um `Optional` com valor, se `valor != null`
  - ou `Optional.empty()`, se `valor == null`

❌ Por que as outras alternativas estão erradas?

A) `Optional.of(valor)`

→ Não aceita `null`.

B) `Optional.isNullable(valor)`

→ **Não existe** esse método na API do Java.

D) `Optional.empty(valor)`

→ `empty()` **não recebe parâmetros**.

E) `Optional.nullable(valor)`

→ **Método inexistente**.

📌 Conclusão

A forma correta de criar um `Optional` que pode conter `null` é `Optional.ofNullable(valor)`, alternativa **C**.