Relatório Java vs C

1. Entrada e Saída

Java

```
public static void main(String[] args) {
    Scanner scanner = new Scanner(System.in);
    System.out.print("Digite sua idade: ");
    System.out.println("Você tem " + idade + " anos.");
    scanner.close();
}
```

Explicação:

- `Scanner` é uma classe utilitária para ler entradas de várias fontes (teclado, arquivos)
- `System.out` é um objeto estático que representa a saída padrão
- `nextInt()` lê e converte a entrada para inteiro automaticamente
- O fechamento do scanner é necessário para evitar vazamentos de recursos

C

```
int main() {
    int idade;
    printf("Digite sua idade: ");
    printf("Você tem %d anos.\n", idade);
    return 0;
}
```

Explicação:

- `printf` usa especificadores de formato (`%d` para inteiros)
- `scanf` requer o operador `&` para obter o endereço de memória da variável
- Não há verificação automática de tipo - se o usuário digitar texto, ocorrerá comportamento indefinido

---

Características:

- Não há tipo booleano nativo - usa-se `0` para false e qualquer outro valor para true

- Arrays de `char` são strings, exigindo terminação manual com `\0`
- Valores não inicializados contêm "lixo" da memória

---

3. Estruturas Condicionais

Java (Controle Flexível)

```
int nota = 85;
String resultado;

// If-Else encadeado
if (nota >= 90) {
    resultado = "A";
} else if (nota >= 80) {
    resultado = "B";
} else {
    resultado = "C";
}

// Switch com String (Java 7+)
switch (resultado) {
    case "A":
        System.out.println("Excelente!");
        break;
    case "B":
        System.out.println("Bom!");
        break;
    default:
        System.out.println("Estude mais!");
}
```

Explicação:

- `switch` funciona com Strings, enums e wrappers (Integer, Character)
- `break` é essencial para evitar "fall-through"
- Condições podem usar métodos complexos (`equals()`, `compareTo()`)

C (Controle Básico)

```
int nota = 85;
char resultado;

switch (resultado) {
```

```
    case 'A':
        printf("Excelente!\n");
        break;
    case 'B':
        printf("Bom!\n");
        break;
    default:
        printf("Estude mais!\n");
}
```

Explicação:

- `switch` só funciona com tipos inteiros (char, int, enum)
- Comparação de strings exige `strcmp()` em condições `if`
- Ausência de `break` executa todos os casos subsequentes

---

4. Estruturas de Iteração

Java (Alto Nível)

```
// Do-While (executa ao menos uma vez)
int j = 0;
do {
    System.out.println(j++);
} while (j < 5);
```

Explicação:

- For-Each simplifica iteração em coleções
- Tipos iteráveis devem implementar a interface `Iterable`
- Controle de fluxo com `break` e `continue`

C (Controle Manual)

```
int j = 0;
do {
    printf("%d\n", j++);
} while (j < 5);

// Iteração em array
int numeros[] = {1, 2, 3};
for (int k = 0; k < 3; k++) {
    printf("%d\n", numeros[k]);
}
```

Explicação:

- Necessidade de gerenciar manualmente índices e condições
- Arrays não guardam informação de tamanho
- Uso comum de `NULL` como marcador de fim em estruturas de dados

---

5. Funções

Java (Métodos em Classes)

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

    public int multiplicar(int a, int b) {
        return a * b;
    }

    public static void main(String[] args) {
        System.out.println(somar(5, 3)); // 8

        Calculadora calc = new Calculadora();
        System.out.println(calc.multiplicar(4, 5)); // 20
    }
}
```

Explicação:

- Funções existem apenas como métodos dentro de classes
- `static` permite chamada sem instância
- Sobrecarga de métodos permitida (mesmo nome, parâmetros diferentes)

C (Funções Independentes)

```
int somar(int a, int b);

int main() {
    printf("Soma: %d\n", somar(5, 3)); // 8
    return 0;
}

int somar(int a, int b) {
    return a + b;
}
```

Explicação:

- Funções são globais por padrão
- Protótipos necessários para funções usadas antes da declaração
- Não há suporte a sobrecarga de funções

---

6. Strings e Vetores

Java (Orientado a Objetos)

```
String nome = "Alice";
nome = nome.concat(" Wonderland"); // Cria nova string

StringBuilder sb = new StringBuilder();
sb.append("Hello");
sb.append(" World");
String resultado = sb.toString();

ArrayList<Integer> numeros = new ArrayList<>();
numeros.add(1);
numeros.add(2);
numeros.remove(0);
```

Explicação:

- `String` é imutável para segurança em multi-threading
- `StringBuilder` oferece manipulação eficiente de strings mutáveis
- Coleções genéricas (`ArrayList<T>`) garantem type safety

C (Manipulação Manual)

```
char nome[20] = "Bob";
strcat(nome, " Builder"); // Concatenação perigosa (risco de overflow)

int numeros[3] = {1, 2, 3};
numeros[0] = 4; // Modificação direta

int* nums = malloc(3 * sizeof(int));
nums[0] = 1;
free(nums); // Liberação obrigatória
```

Explicação:

- Strings são arrays de `char` terminados em `\0`

- Funções da `string.h` (`strcpy`, `strcat`) exigem gerenciamento manual de memória
- Vetores dinâmicos requerem alocação manual e cálculo preciso de tamanhos

---

7. Tipos Abstratos de Dados

Java (Classes)

```
public class Pessoa {
    private String nome;
    private int idade;

    public Pessoa(String nome, int idade) {
        this.nome = nome;
        this.idade = idade;
    }

    public String getNome() { return nome; }
    public void aniversario() { idade++; }
}
```

Explicação:

- Encapsulamento via modificadores de acesso (`private`, `public`)
- Métodos definem comportamento associado aos dados
- Herança permite criar hierarquias de tipos

C (Structs)

```
typedef struct {
    char nome[50];
    int idade;
} Pessoa;

void aniversario(Pessoa* p) {
    p->idade++;
}

int main() {
    Pessoa alice;
    strcpy(alice.nome, "Alice");
    alice.idade = 30;
    aniversario(&alice);
}
```

Explicação:

- `struct` agrupa dados, mas não comportamentos
- Funções de manipulação devem ser definidas separadamente
- Acesso direto aos membros (sem encapsulamento padrão)

---

8. Ponteiros/Referências

Java (Referências Implícitas)

```
Pessoa p1 = new Pessoa("João", 25);
Pessoa p2 = p1;

p2.setIdade(30);
System.out.println(p1.getIdade()); // 30
```

Explicação:

- Objetos são sempre acessados por referência
- Variáveis armazenam ponteiros para objetos na heap
- Coletor de lixo gerencia automaticamente a memória

C (Ponteiros Explícitos)

```
int valor = 10;
int* ptr = &valor;

*ptr = 20;
printf("%d", valor);

Pessoa* pessoaPtr = malloc(sizeof(Pessoa));
pessoaPtr->idade = 25;
free(pessoaPtr);
```

Explicação:

- Ponteiros armazenam endereços de memória
- Operadores `&` (endereço) e `*` (dereferência)
- Aritmética de ponteiros permite acesso eficiente a arrays

---

9. Manipulação de Arquivos

Java (Gerenciamento Automático)

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

```
        // Try-with-resources fecha automaticamente
        try (BufferedWriter writer = new BufferedWriter(new FileWriter("dados.txt"))) {
            writer.write("Linha 1");
            writer.newLine();
            writer.write("Linha 2");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Explicação:

- `try-with-resources` garante fechamento automático de recursos
- Hierarquia de exceções (`IOException`) para tratamento de erros
- Classes utilitárias (`BufferedReader`, `FileWriter`) simplificam operações

C (Controle Manual)

```
int main() {
    FILE* arquivo = fopen("dados.txt", "w");

    if (arquivo == NULL) {
        perror("Erro ao abrir arquivo");
        return 1;
    }

    fprintf(arquivo, "Linha 1\n");
    fputs("Linha 2\n", arquivo);

    fclose(arquivo);
    return 0;
}
```

Explicação:

- `FILE*` é um ponteiro para estrutura de arquivo
- Modos de abertura: `"r"` (leitura), `"w"` (escrita), `"a"` (append)
- Verificação manual de erros necessária em todas as operações