Java Inter - part2

Construtores

Em Java, um construtor é um método especial em uma classe que é usado para inicializar objetos dessa classe. Os construtores são chamados automaticamente quando um novo objeto é criado e são responsáveis por realizar qualquer inicialização necessária para que o objeto seja usado corretamente.

Aqui estão algumas características importantes dos construtores em Java:

1. Nome do Construtor:
   - O nome de um construtor deve ser exatamente o mesmo nome da classe em que ele está definido.
   - O construtor não possui um tipo de retorno, nem mesmo void.
2. Tipos de Construtores:
   - Construtores padrão (default): Se uma classe não declara nenhum construtor, o Java fornece automaticamente um construtor padrão sem argumentos (conhecido como construtor padrão) que inicializa os campos da classe com valores padrão (0 para tipos numéricos, null para referências, etc.).
   - Construtores personalizados: Você também pode criar construtores personalizados, que podem ter argumentos e realizar a inicialização personalizada com base nos valores passados como argumentos.
3. Overloading de Construtores:
   - Você pode ter vários construtores em uma classe, desde que tenham assinaturas diferentes (ou seja, diferentes números ou tipos de argumentos). Isso é chamado de sobrecarga de construtores.
4. Uso de Construtores:
   - Os construtores são invocados usando a palavra-chave `new` seguida pelo nome da classe, seguido pelos argumentos (se houver) entre parênteses. Por exemplo: `MinhaClasse objeto = new MinhaClasse();`.
5. Inicialização de Campos:
   - Um dos principais usos dos construtores é para inicializar os campos (variáveis de instância) de um objeto. Você pode atribuir valores iniciais aos campos dentro do construtor.
6. Chamada de Outros Construtores:
   - Em uma classe, você pode chamar um construtor de outro construtor da mesma classe usando a palavra-chave `this(...)`. Isso é útil quando você tem vários construtores e deseja reutilizar a lógica de inicialização.

Exemplo de classe com construtores em Java:

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

```
    // Construtor padrão
    public Pessoa() {
        nome = "Sem nome";
        idade = 0;
    }

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

    // Métodos getters e setters para acessar os campos
    public String getNome() {
        return nome;
    }

    public int getIdade() {
        return idade;
    }

    public void setNome(String nome) {
        this.nome = nome;
    }

    public void setIdade(int idade) {
        this.idade = idade;
    }
}
```

Neste exemplo, a classe `Pessoa` possui dois construtores: um construtor padrão que inicializa os campos com valores padrão e um construtor personalizado que permite a inicialização personalizada dos campos. Esses construtores podem ser usados para criar objetos `Pessoa` com diferentes estados iniciais.

`this`

Em Java, a palavra-chave "this" é uma referência especial que se refere à instância atual de uma classe. Ela pode ser usada em vários contextos dentro de uma classe para se referir aos membros (variáveis de instância, métodos ou construtores) da instância atual. Aqui estão alguns dos principais usos de "this" em Java:

1. Referência a variáveis de instância: Você pode usar "this" para se referir às variáveis de instância da classe atual quando há ambiguidade entre uma variável de instância e uma variável local dentro de um método. Por

exemplo:

```
public class Exemplo {
    private int numero;

    public void setNumero(int numero) {
        this.numero = numero;
    }
}
```

Neste exemplo, usamos “this.numero” para distinguir entre a variável de instância “numero” e o parâmetro do método “numero”.

2. Invocação de construtores: “this” também pode ser usado para chamar um construtor sobrecarregado da mesma classe a partir de outro construtor. Isso é útil para evitar a duplicação de código em diferentes construtores. Aqui está um exemplo:

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

    public Pessoa(String nome) {
        this(nome, 0);
    }

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

Neste caso, o construtor com um único argumento chama o construtor com dois argumentos usando “this(nome, 0)”.

3. Retornando a instância atual: Em certos casos, você pode usar “this” para retornar a instância atual de um objeto. Isso é útil para criar métodos encadeados, onde você pode chamar vários métodos na mesma linha. Por exemplo:

```
public class MinhaClasse {
    private int valor;

    public MinhaClasse setValor(int valor) {
        this.valor = valor;
        return this;
    }
}
```

Isso permite que você faça chamadas encadeadas, como: "minhaInstancia.setValor(42).outroMetodo()".

Em resumo, a palavra-chave "this" em Java é usada para se referir à instância atual da classe em que ela está sendo utilizada. Ela é útil para evitar ambiguidades, chamar construtores sobrecarregados e criar métodos encadeados.

Sobrecarga

Em Java, a sobrecarga (ou "overloading" em inglês) é um conceito que permite que você defina múltiplos métodos com o mesmo nome em uma classe, desde que esses métodos tenham assinaturas diferentes. A assinatura de um método é composta pelo nome do método e pela lista de tipos e quantidade de parâmetros que ele aceita. A sobrecarga é uma forma de polimorfismo estático, onde o Java determina qual método chamar com base na assinatura dos argumentos passados.

Aqui estão alguns aspectos importantes sobre a sobrecarga em Java:

1. Assinaturas diferentes: Para sobrecarregar métodos, você precisa definir métodos com nomes idênticos, mas com parâmetros diferentes em termos de tipo, quantidade ou ordem. Isso significa que você pode ter métodos com o mesmo nome, desde que a JVM possa distingui-los com base em suas assinaturas.
2. Retorno: O tipo de retorno não é considerado ao determinar a sobrecarga de um método. Dois métodos com a mesma assinatura, exceto pelo tipo de retorno, são considerados sobrecarga inválida.
3. Exceções: A lista de exceções lançadas por um método não faz parte de sua assinatura. Portanto, você pode sobrecarregar métodos com diferentes exceções lançadas.

Aqui está um exemplo simples de sobrecarga de métodos em Java:

```
public class ExemploSobrecarga {

    // Método que aceita dois inteiros
    public int somar(int a, int b) {
        return a + b;
    }

    // Método sobrecarregado que aceita três inteiros
    public int somar(int a, int b, int c) {
        return a + b + c;
    }

    // Método sobrecarregado que aceita dois números de ponto flutuante
    public double somar(double a, double b) {
```

```
        return a + b;
    }

    public static void main(String[] args) {
        ExemploSobrecarga exemplo = new ExemploSobrecarga();

        System.out.println(exemplo.somar(2, 3)); // Chama o primeiro método
        System.out.println(exemplo.somar(2, 3, 4)); // Chama o segundo método
        System.out.println(exemplo.somar(2.5, 3.5)); // Chama o terceiro método
    }
}
```

Neste exemplo, a classe `ExemploSobrecarga` contém três métodos chamados `somar`, cada um aceitando diferentes tipos e quantidades de argumentos. A JVM determinará qual método chamar com base nos argumentos passados durante a chamada.

A sobrecarga de métodos é uma técnica útil para criar interfaces de programação mais flexíveis e intuitivas, permitindo que os desenvolvedores escolham a versão do método que melhor se adapta às suas necessidades com base nos argumentos fornecidos.

Encapsulamento

O encapsulamento é um dos quatro pilares da programação orientada a objetos (POO) e é uma técnica fundamental no Java e em muitas outras linguagens de programação orientadas a objetos. Ele se refere à prática de esconder os detalhes internos de uma classe e fornecer uma interface controlada para acessar e manipular os dados e comportamentos dessa classe. O encapsulamento é alcançado através da combinação de dois conceitos: acesso restrito aos membros da classe e o uso de métodos getter e setter.

Aqui estão alguns aspectos importantes do encapsulamento no Java:

1. **Atributos privados:** Os atributos de uma classe geralmente são declarados como privados (usando a palavra-chave `private`). Isso significa que esses atributos não podem ser acessados diretamente de fora da classe. Por exemplo:

```
private int idade;
```

2. **Métodos públicos:** Para permitir o acesso controlado aos atributos privados, você deve fornecer métodos públicos na classe para ler (getter) e modificar (setter) esses atributos. Por exemplo:

```
public int getIdade() {
    return idade;
}
```

```
public void setIdade(int idade) {
    if (idade >= 0) { // Verifica se a idade é válida
        this.idade = idade;
    }
}
```

3. **Controle de acesso:** O Java fornece quatro níveis de controle de acesso para membros de classe: `public`, `private`, `protected` e o padrão (quando nenhum modificador é especificado). O encapsulamento geralmente envolve a definição de atributos como privados e métodos relacionados como públicos, para garantir que os dados internos da classe sejam acessados apenas de maneira segura e controlada.
4. **Validação e lógica de negócios:** Com o encapsulamento, você pode adicionar lógica de validação aos métodos setter para garantir que os valores atribuídos aos atributos estejam dentro de limites aceitáveis. Isso ajuda a manter a integridade dos dados e a prevenir comportamentos indesejados.
5. **Flexibilidade e manutenção:** O encapsulamento ajuda a ocultar a implementação interna de uma classe, permitindo que você altere a implementação sem afetar o código que a utiliza. Isso facilita a manutenção e a evolução do código ao longo do tempo.
6. **Segurança:** O encapsulamento ajuda a proteger os dados de uma classe, evitando que eles sejam corrompidos ou acessados de maneira inadequada por outras partes do programa.

Em resumo, o encapsulamento no Java envolve o uso de modificadores de acesso, como `private`, juntamente com métodos getter e setter para controlar o acesso aos atributos de uma classe, garantindo assim a segurança, a manutenção e a flexibilidade do código. Isso é fundamental para o princípio da ocultação de informações na POO e ajuda a criar classes mais robustas e coesas.

Controles de acesso - public, private, protected

Os controles de acesso em Java são um mecanismo que permite restringir o acesso a membros de classe, como atributos, métodos e construtores. Isso é importante para proteger a integridade e a segurança do código, evitando que dados confidenciais sejam acessados por usuários não autorizados.

A linguagem Java fornece quatro modificadores de acesso:

- **public:** permite que qualquer classe, em qualquer pacote, acesse o membro.
- **protected:** permite que qualquer classe no mesmo pacote ou subclasses acesse o membro.
- **default:** permite que qualquer classe no mesmo pacote acesse o membro.
- **private:** restringe o acesso ao membro à própria classe.

Para utilizar os controles de acesso, basta declarar o modificador desejado antes do nome do membro. Por exemplo:

```
public class Pessoa {

    private String nome;

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

    public String getNome() {
        return nome;
    }

    public void setNome(String nome) {
        this.nome = nome;
    }

}
```

Neste exemplo, a variável `nome` é declarada como `private`, o que significa que só pode ser acessada por métodos da própria classe `Pessoa`.

Os controles de acesso podem ser usados para:

- **Proteger dados confidenciais:** por exemplo, uma classe que armazena informações de cartão de crédito pode declarar esses dados como `private`.
- **Evitar conflitos de nomenclatura:** se duas classes diferentes tiverem um membro com o mesmo nome, os controles de acesso podem ser usados para garantir que cada membro seja acessível apenas pela classe que o declarou.
- **Forçar o uso de determinados métodos:** por exemplo, uma classe pode declarar um método `setNome()` que é usado para atribuir um novo valor ao atributo `nome`. Isso força os usuários a usar esse método para atribuir um valor ao atributo, evitando que eles modifiquem o atributo diretamente.

É importante usar os controles de acesso de forma adequada para garantir a segurança e a integridade do código.

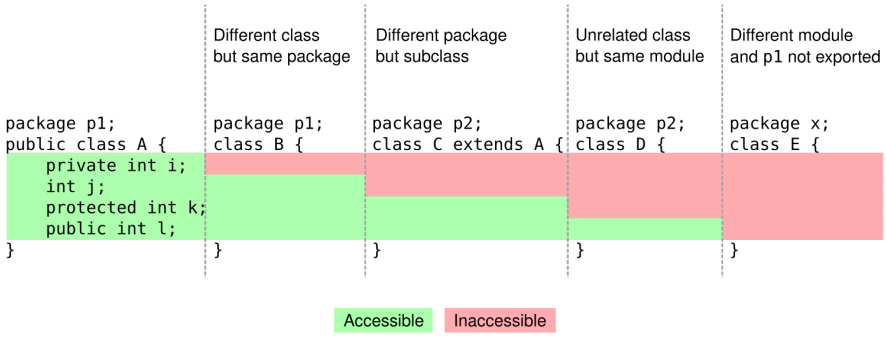
Aqui estão algumas dicas para utilizar os controles de acesso de forma eficaz:

- **Use o modificador `public` apenas quando for necessário:** o modificador `public` permite que qualquer classe, em qualquer pacote, acesse o membro. Isso significa que qualquer pessoa que tenha acesso ao código pode acessar o membro, o que pode representar um risco de segurança.
- **Use o modificador `protected` para restringir o acesso a subclasses:** o modificador `protected` permite que qualquer classe no mesmo pacote

ou subclasses acesse o membro. Isso é útil para permitir que subclasses acessem membros que não são públicos.

- **Use o modificador `default` para restringir o acesso a classes no mesmo pacote:** o modificador `default` é o padrão para membros de classe. Ele permite que qualquer classe no mesmo pacote acesse o membro.
- **Use o modificador `private` para restringir o acesso à própria classe:** o modificador `private` é o mais restritivo. Ele restringe o acesso ao membro à própria classe.

Ao seguir essas dicas, você poderá garantir que os controles de acesso em Java sejam usados de forma eficaz para proteger seu código.



Tipos de referência vs tipos de valor

Em Java, os tipos de referência e os tipos de valor são dois conceitos importantes relacionados à forma como os dados são tratados e armazenados na memória. Vamos descrever cada um deles:

Tipos de Referência (Reference Types): 1. **Objetos:** Em Java, a maioria dos dados é representada como objetos. Isso inclui tipos como `String`, `ArrayList`, `HashMap` e classes personalizadas que você cria. Os objetos são armazenados na memória como referências alocadas no heap. 2. **Alocação dinâmica:** Os objetos são alocados dinamicamente na memória heap usando a palavra-chave `new`. Eles existem enquanto houver referências ativas apontando para eles. 3. **Referência:** As variáveis que armazenam objetos não armazenam o objeto diretamente, mas sim uma referência (um endereço de memória) para o objeto no heap. 4. **Valor padrão:** Quando uma variável de referência é declarada, seu valor padrão é `null`, o que indica que ela não aponta para nenhum objeto.

Exemplo de declaração e atribuição de uma variável de referência:

```
String texto; // Declaração de uma variável de referência
texto = new String("Exemplo"); // Atribuição de uma referência a um objeto
```

Tipos de Valor (Value Types): 1. **Primitivos:** Em Java, tipos primitivos

são tipos de valor. Eles representam valores individuais, como números inteiros, ponto flutuante, caracteres e booleanos. 2. **Armazenamento direto:** Os tipos primitivos armazenam seus valores diretamente na memória, não como referências. Isso resulta em um uso mais eficiente da memória e acesso mais rápido aos dados. 3. **Não aceitam `null`:** Os tipos primitivos não podem ter um valor `null`. Eles têm valores padrão, como 0 para inteiros ou `false` para booleanos, que são atribuídos quando uma variável é declarada.

Exemplo de declaração e atribuição de um tipo primitivo:

```
int numero; // Declaração de uma variável de tipo primitivo
numero = 42; // Atribuição de um valor a uma variável de tipo primitivo
```

Em resumo, os tipos de referência em Java referem-se a objetos que são alocados no heap e são manipulados por meio de referências. Os tipos de valor, por outro lado, são tipos primitivos que armazenam valores diretamente e não são objetos no heap. A escolha entre tipos de referência e tipos de valor depende das necessidades do seu programa e dos recursos de memória e desempenho que você deseja otimizar.

Tipos de referência vs tipos de valor

Em Java, os tipos de referência e os tipos de valor são dois conceitos importantes relacionados à forma como os dados são tratados e armazenados na memória. Vamos descrever cada um deles:

Tipos de Referência (Reference Types): 1. **Objetos:** Em Java, a maioria dos dados é representada como objetos. Isso inclui tipos como `String`, `ArrayList`, `HashMap` e classes personalizadas que você cria. Os objetos são armazenados na memória como referências alocadas no heap. 2. **Alocação dinâmica:** Os objetos são alocados dinamicamente na memória heap usando a palavra-chave `new`. Eles existem enquanto houver referências ativas apontando para eles. 3. **Referência:** As variáveis que armazenam objetos não armazenam o objeto diretamente, mas sim uma referência (um endereço de memória) para o objeto no heap. 4. **Valor padrão:** Quando uma variável de referência é declarada, seu valor padrão é `null`, o que indica que ela não aponta para nenhum objeto.

Exemplo de declaração e atribuição de uma variável de referência:

```
String texto; // Declaração de uma variável de referência
texto = new String("Exemplo"); // Atribuição de uma referência a um objeto
```

Tipos de Valor (Value Types): 1. **Primitivos:** Em Java, tipos primitivos são tipos de valor. Eles representam valores individuais, como números inteiros, ponto flutuante, caracteres e booleanos. 2. **Armazenamento direto:** Os tipos primitivos armazenam seus valores diretamente na memória, não como referências. Isso resulta em um uso mais eficiente da memória e acesso mais rápido aos dados. 3. **Não aceitam `null`:** Os tipos primitivos não podem ter

um valor `null`. Eles têm valores padrão, como 0 para inteiros ou `false` para booleanos, que são atribuídos quando uma variável é declarada.

Exemplo de declaração e atribuição de um tipo primitivo:

```
int numero; // Declaração de uma variável de tipo primitivo
numero = 42; // Atribuição de um valor a uma variável de tipo primitivo
```

Em resumo, os tipos de referência em Java referem-se a objetos que são alocados no heap e são manipulados por meio de referências. Os tipos de valor, por outro lado, são tipos primitivos que armazenam valores diretamente e não são objetos no heap. A escolha entre tipos de referência e tipos de valor depende das necessidades do seu programa e dos recursos de memória e desempenho que você deseja otimizar.

Garbage collector

O Garbage Collector (Coletor de Lixo) é uma parte fundamental da máquina virtual Java (JVM) que gerencia a alocação e desalocação de memória durante a execução de programas em Java. Sua principal responsabilidade é liberar a memória que não está mais sendo usada por objetos, tornando-a disponível para ser reutilizada por novos objetos. Isso ajuda a evitar vazamentos de memória e garante que os programas Java não esgotem os recursos do sistema devido à alocação descontrolada de memória.

Aqui estão alguns aspectos importantes sobre o Garbage Collector no Java:

1. Detecção de Objetos Inalcançáveis: O Garbage Collector identifica objetos que não podem mais ser acessados pelo programa. Isso é feito através da análise de referências de objetos. Um objeto é considerado inalcançável quando não pode ser acessado diretamente ou indiretamente a partir do ponto de entrada do programa (normalmente o método `main`).

2. Algoritmos de Coleta: O Java possui vários algoritmos de coleta de lixo, incluindo o algoritmo padrão conhecido como "Garbage-First" (G1), bem como o "Parallel Garbage Collector", "Concurrent Mark-Sweep" (CMS) e outros. Cada um desses algoritmos tem suas próprias características e é adequado para diferentes tipos de aplicativos.

3. Ciclo de Vida da Coleta de Lixo: O processo de coleta de lixo envolve várias fases, como marcação, compactação e liberação de memória. O Garbage Collector executa essas fases de forma automática e periódica, conforme necessário.

4. Chamada Explicita: Embora o Garbage Collector seja projetado para operar automaticamente, você pode solicitar explicitamente a coleta de lixo chamando o método `System.gc()` ou `Runtime.getRuntime().gc()`. No entanto, isso não garante a coleta imediata de todos os objetos não utilizados e é geralmente desnecessário em grande parte das aplicações.

5. Gerenciamento de Memória: O Java utiliza a memória heap para alocar objetos, e o Garbage Collector cuida da desalocação automática de objetos inacessíveis. Isso significa que os desenvolvedores de Java não precisam se preocupar com a desalocação manual de memória, como em algumas outras linguagens.

6. Impacto no Desempenho: Embora o Garbage Collector seja altamente eficiente na maioria dos casos, ele ainda pode afetar o desempenho da aplicação, especialmente em cenários de alta carga de trabalho. É importante entender o comportamento do Garbage Collector e otimizar o código, se necessário, para minimizar o impacto.

Em resumo, o Garbage Collector no Java é uma parte crítica do ambiente de tempo de execução que gerencia automaticamente a alocação e desalocação de memória, permitindo que os desenvolvedores se concentrem na lógica de seus programas sem se preocupar com vazamentos de memória. A escolha do algoritmo de coleta de lixo adequado e a compreensão de seu funcionamento são importantes para otimizar o desempenho de aplicativos Java.