Escopo e visibilidade

PROF. EDUARDO HABIB BECHELANE MAIA

Introdução

Definição de Escopo: Refere-se à região do código onde uma variável pode ser acessada.

Blocos de Código: Escopo é geralmente definido por blocos de código, delimitados por chaves {}.

Escopo de Classe: Variáveis definidas dentro de uma classe estão no escopo daquela classe.

Escopo de Método: Variáveis definidas dentro de um método estão no escopo daquele método.

Escopo de Bloco: Variáveis definidas dentro de blocos de código (como loops e condicionais) têm escopo limitado àquele bloco.

Escopo de variáveis

Variáveis Locais: Existem somente dentro do método que foram declaradas.

Variáveis de Membro (Atributos): Existem enquanto o objeto da classe existir.

Variáveis de Classe (Static): Existem durante toda a execução do programa.

Parâmetros de Método: Existem somente dentro do método que foram passados.

Tipos de visibilidade

Public: Acessível de qualquer outra classe.

Protected: Acessível dentro do pacote e por subclasses.

Default (Package-Private): Acessível apenas dentro do pacote.

Private: Acessível apenas dentro da classe.

Escopo e visibilidade em C++ vs Java

Definição de Escopo: Similar em ambas as linguagens, definido por blocos de código.

Visibilidade: Ambas as linguagens usam conceitos similares de public, protected, e private.

Variáveis Globais: Em C++, você pode declarar variáveis globais fora de qualquer classe ou função.

- Java não suporta variáveis globais da mesma maneira.

Namespace/Pacotes: C++ usa namespaces para definir o escopo. Java usa pacotes.

Duração de Variáveis Estáticas: Em ambas as linguagens, variáveis estáticas têm a duração do programa.

Boas práticas

Minimizar o Escopo de Variáveis: Declarar variáveis no menor escopo possível.

Usar Modificadores de Acesso Apropriados: Para encapsular e proteger os dados da classe.

Evitar Variáveis Globais: Evite usar.

Usar Constantes: Para valores que não devem ser modificados.

Documentar o Código: Deixar claro o propósito e o uso de cada variável, especialmente se elas têm escopo amplo.

Sobrecarga

EDUARDO HABIB BECHELANE MAIA

HABIB@CEFETMG.BR

Introdução à Sobrecarga

Definição de Sobrecarga de Métodos:

- A sobrecarga de métodos permite que múltiplos métodos tenham o mesmo nome, mas com listas de parâmetros diferentes.

Uso:

- Utilizada para realizar tarefas semelhantes com tipos de dados diferentes.

Importância:

- Aumenta a legibilidade e reusabilidade do código, além de permitir variações com diferentes tipos e números de parâmetros.

Exemplo Simples:

- Dois métodos chamados "somar" onde um soma dois inteiros e o outro soma dois números de ponto flutuante (double).

Sintaxe Básica:

- O nome do método é o mesmo, mas a lista de parâmetros deve ser diferente, no número ou nos tipos.

Regras

Nome do Método:

- O nome do método sobrecarregado deve ser o mesmo.

Lista de Parâmetros:

- Deve variar em tipo, número ou ambos.

Tipo de Retorno:

- Pode ser diferente e não faz parte da assinatura do método.

Visibilidade:

- Os modificadores de acesso podem ser diferentes.

Exceções:

- A lista de exceções pode variar.

Exemplo

Método somar para Inteiros:

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

Método somar para Doubles:

- public double somar(double a, double b) { return a + b; }

Método somar com Três Parâmetros:

- public int somar(int a, int b, int c) { return a + b + c; }

Chamada dos Métodos:

- Exemplo

Resultado:

- Apresentação dos resultados das chamadas de métodos.

Boas práticas

Consistência:

- Métodos sobrecarregados devem realizar tarefas semelhantes para evitar confusão.

Documentação:

- É crucial documentar adequadamente cada versão do método sobrecarregado.

Número de Métodos:

- Evitar um número excessivo de métodos sobrecarregados.

Testes:

- Todos os métodos sobrecarregados devem ser devidamente testados.

Manutenção:

- Ao modificar um método sobrecarregado, é necessário verificar todas as suas versões.

Sobrecarga de Operadores

- C++: Permite a sobrecarga de operadores, o que significa que você pode redefinir o comportamento de operadores padrão (+, -, *, /, etc.) para objetos de suas classes.
- Java: Não suporta a sobrecarga de operadores. O comportamento dos operadores padrão não pode ser alterado.
- Ex: https://onlinegdb.com/rm3Sh36MO

```
#include <iostream>

class Numero {
public:
    int valor;

    Numero(int v) : valor(v) {} // Construtor que inicializa o valor

    // Sobrecarga do operador +
    Numero operator+(const Numero& outro) const {
        return Numero(this->valor + outro.valor);
    }
};

int main() {
    Numero num1(5); // num1.valor == 5
    Numero num2(3); // num2.valor == 3

    Numero num3 = num1 + num2; // num3.valor == 8, resultado da soma de num1 e num2

    std::cout << "Valor de num1: " << num1.valor << std::endl;
    std::cout << "Valor de num2: " << num2.valor << std::endl;
    std::cout << "Valor de num3 (num1 + num2): " << num3.valor << std::endl;

    return 0;
}
```

Sobrecarga de construtores e métodos

2. Sobrecarga de Construtores:
   - Ambas as Linguagens: Permitem a sobrecarga de construtores, o que significa que você pode ter vários construtores em uma classe, cada um com diferentes listas de parâmetros.

3. Resolução de Método:
   - C++: É mais flexível, permitindo correspondência de tipo automática, promoção de tipo padrão e conversões definidas pelo usuário na resolução de qual método sobrecarregado chamar.
   - Java: É mais restritivo, exigindo uma correspondência mais precisa entre os argumentos fornecidos e os parâmetros do método.

Resolução de método em c++

C++ oferece mais flexibilidade na resolução de métodos sobrecarregados. Quando há várias sobrecargas de um método, o compilador C++ tentará encontrar a melhor correspondência possível, mesmo que isso envolva converter os tipos de dados dos argumentos fornecidos.

```
class Exemplo {
public:
    void mostrar(int i) {
        std::cout << "Inteiro: " << i << std::endl;
    }

    void mostrar(double d) {
        std::cout << "Double: " << d << std::endl;
    }
};

int main() {
    Exemplo e;
    e.mostrar(10);    // Chama mostrar(int i)
    e.mostrar(10.5);  // Chama mostrar(double d)
    e.mostrar('a');   // Chama mostrar(int i), pois converte automaticamente char para int
    return 0;
}
```

No exemplo acima, o método mostrar está sobrecarregado para aceitar int e double. Quando um char é passado como argumento, o compilador automaticamente converte o char para um int, correspondendo à sobrecarga mostrar(int i).

Resolução de métodos em Java

Java é mais restritivo, exigindo u

- Exige correspondência maior.
- Se não encontrar uma correspondência exata, resultará em um erro de compilação

```
public class Exemplo {

    void mostrar(int i) {
        System.out.println("Inteiro: " + i);
    }

    void mostrar(double d) {
        System.out.println("Double: " + d);
    }

    public static void main(String[] args) {
        Exemplo e = new Exemplo();
        e.mostrar(10);    // Chama mostrar(int i)
        e.mostrar(10.5);  // Chama mostrar(double d)
        //e.mostrar('a'); // Causaria um erro de compilação se descomentado
    }
}
```

Conversão de tipo

- **C++:** Permite conversões de tipo explícitas e implícitas definidas pelo usuário, que podem ser usadas para resolver ambiguidades na sobrecarga de métodos.
- **Java:** Não permite conversões de tipo definidas pelo usuário e é mais rigoroso sobre conversões de tipo implícitas.

Conversão de tipo em c++

Em C++, você pode definir conversões de tipo

- permite converter objetos de uma classe para outro tipo (primitivo ou classe)
  - De maneira implícita ou explícita, utilizando operadores de conversão.

```
#include <iostream>
class Numero {
public:
    int valor;

    Numero(int v) : valor(v) {} // Construtor

    //Operador de conversão definido para converter um objeto Numero em int
    operator int() const {
        return valor;
    }
};

int main() {
    Numero num(10);
    int valor = num; // Conversão implícita de Num para int usando o operador definido
    std::cout << "Valor: " << valor << std::endl; // Saída: Valor: 10
    return 0;
}
```

https://onlinegdb.com/VR0Z-FRsq

Conversão de tipo em JAVA

Em Java, não existe uma funcionalidade diretamente equivalente.

- Você não pode definir conversões de tipo implícitas entre classes e tipos primitivos ou entre diferentes classes.
- Todas as conversões de tipo devem ser explicitamente solicitadas pelo programador, e apenas conversões válidas e seguras são permitidas.
- Ex:

```
class Numero {
    int valor;

    Numero(int valor) {
        this.valor = valor;
    }

    // Método para converter um objeto Numero em int
    int toInt() {
        return valor;
    }
}

public class Main {
    public static void main(String[] args) {
        Numero num = new Numero(10);
        int valor = num.toInt(); // Conversão explícita de Numero para int usando um método
        System.out.println("Valor: " + valor); // Saída: Valor: 10
    }
}
```

Semântica

C++ suporta sobrecarga em Programação orientada a objetos ou em programação procedural

JAVA suporta sobrecarga apenas usando POO