A INTERFACE SUPPLIER<T>

A interface funcional Supplier<T> faz parte do pacote java.util.function e representa um **fornecimento de resultados**. Em outras palavras, é uma função que **não recebe nenhum argumento, mas gera e retorna um resultado do tipo T**.

Como é uma interface funcional, Supplier **possui apenas um método abstrato,** o que facilita sua implementação usando expressões lambda ou referências de métodos.

Métodos na Interface Supplier

- **T get()**: Este é o único método abstrato da interface Supplier. Ele é responsável por **fornecer ou "produzir" um valor do tipo T quando chamado**. Como não recebe parâmetros, ele é frequentemente usado em cenários onde há a necessidade de gerar ou obter um valor sem a necessidade de entrada, como instâncias de objetos, valores calculados, ou dados provenientes de fontes externas.

- **Ausência de Argumentos**: Ao contrário de outras interfaces funcionais, como Function ou Consumer, o Supplier não toma nenhum argumento. Seu propósito é puramente fornecer (ou gerar) um valor quando solicitado.

A interface Supplier<T> é amplamente utilizada em diversas situações onde há necessidade de gerar ou fornecer valores de forma dinâmica ou sob demanda. Aqui estão algumas das principais aplicações:

1. Inicialização Preguiçosa (*Lazy Initialization*)

Em cenários onde um valor ou recurso não precisa ser inicializado imediatamente, Supplier pode ser usado para postergar a criação ou carregamento desse valor até que ele seja realmente necessário. Isso ajuda a economizar recursos e melhorar o desempenho, especialmente em programas que lidam com operações custosas ou demoradas.

Exemplo de código:

```java
public class SupplierLazyLoading {

    // 1. Inicialização Preguiçosa (Lazy Initialization)
    static class ExpensiveResource {

        private static final Supplier<ExpensiveResource> LAZY_INSTANCE = () -> {
            System.out.println("Criando instância de ExpensiveResource...");
            return new ExpensiveResource();
        };

        private ExpensiveResource() {
            // Simulando uma inicialização custosa
            try {
                System.out.println("Simulando uma inicialização custosa...");
                Thread.sleep(2000);
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }
        }

        public static ExpensiveResource getInstance() {
            return LAZY_INSTANCE.get();
        }
    }



    public static void main(String[] args) {
        // 1. Inicialização Preguiçosa
        System.out.println("Antes de chamar getInstance()");
        @SuppressWarnings("unused")
        ExpensiveResource resource = ExpensiveResource.getInstance();
        System.out.println("Após chamar getInstance()");

    }
}
```

Explicação Detalhada: SupplierLazyLoading

Este código demonstra o uso da interface Supplier para implementar inicialização preguiçosa (lazy initialization) em Java.

Classe SupplierLazyLoading

Esta é a classe principal que contém a classe interna ExpensiveResource e o método main.

Classe Interna ExpensiveResource

Campo LAZY_INSTANCE

```java
private static final Supplier<ExpensiveResource> LAZY_INSTANCE = () -> {
        System.out.println("Criando instância de ExpensiveResource...");
        return new ExpensiveResource();
    };
```

- Este é um campo estático final que utiliza um Supplier<ExpensiveResource>.
- O Supplier é implementado como uma expressão lambda.
- Quando chamado, ele imprime uma mensagem e cria uma nova instância de ExpensiveResource.

- A inicialização deste campo não cria imediatamente uma instância de ExpensiveResource.

Construtor Privado

```
private ExpensiveResource() {
        // Simulando uma inicialização custosa
        try {
            System.out.println("Simulando uma inicialização custosa...");
            Thread.sleep(2000);
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }
    }
```

- O construtor é privado, impedindo a criação direta de instâncias fora da classe.
- Simula uma inicialização custosa usando Thread.sleep(2000).
- Imprime uma mensagem indicando a simulação.

Método getInstance

```
public static ExpensiveResource getInstance() {
        return LAZY_INSTANCE.get();
    }
```

- Este é um método estático que retorna uma instância de ExpensiveResource.
- Chama o método get() do Supplier LAZY_INSTANCE.
- A primeira chamada a este método criará e retornará uma nova instância.
- Cada chamada a getInstance() resultará na criação de uma nova instância de ExpensiveResource. Isso passado ou armazenado para ser utilizado sempre que um novo ocorre porque o Supplier cria uma nova instância cada vez que seu método get() é chamado (não é o padrão Singleton portanto).

```
public static void main(String[] args) {
        System.out.println("Antes de chamar getInstance()");
        ExpensiveResource resource = ExpensiveResource.getInstance();
        System.out.println("Após chamar getInstance()");
    }
```

- Imprime uma mensagem antes de chamar getInstance().
- Chama ExpensiveResource.getInstance(), que acionará a criação da instância.
- Imprime uma mensagem após chamar getInstance().

Fluxo de Execução

O programa inicia e imprime “Antes de chamar getInstance()”.

↓

ExpensiveResource.getInstance() é chamado.

↓

O Supplier LAZY_INSTANCE é acionado pela primeira vez.

↓

Uma mensagem indicando a criação da instância é impressa.

↓

O construtor de ExpensiveResource é chamado.

↓

Uma mensagem sobre a inicialização custosa é impressa.

↓

O programa pausa por 2 segundos (simulando operação custosa).

↓

A instância de ExpensiveResource é retornada.

↓

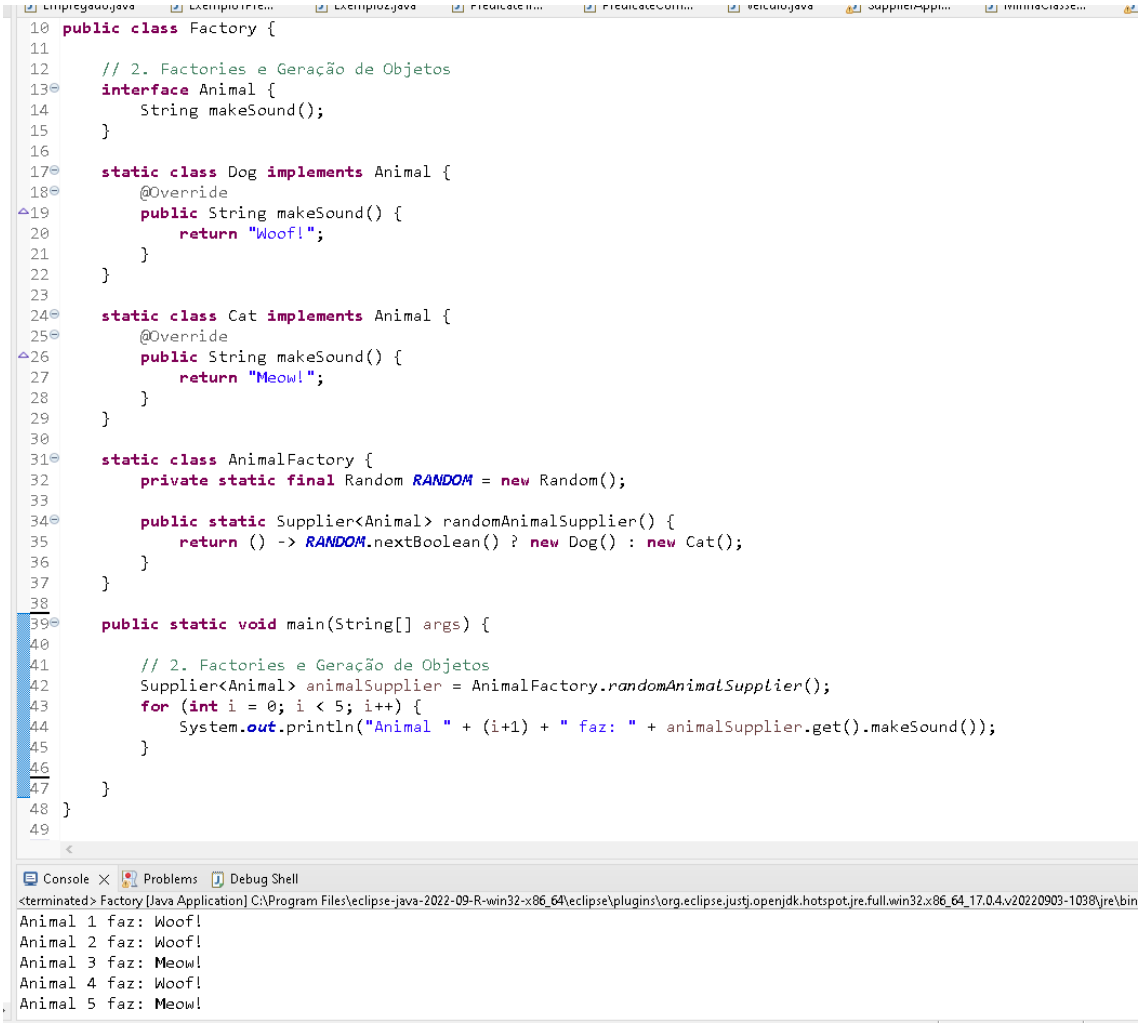
O programa imprime “Após chamar getInstance()”.

Este código demonstra como usar Supplier para implementar *lazy loading*, adiando a criação de um objeto que demanda tempo computacional até que ele seja realmente necessário.

2. Factories e Geração de Objetos

Supplier é frequentemente empregado em padrões de design como *Factory*, onde a criação de objetos precisa ser abstraída. Ao invés de criar diretamente uma instância de um objeto, um *Supplier* pode ser passado ou armazenado para ser utilizado sempre que um novo objeto for necessário..

Exemplo de código:



```
10 public class Factory {
11
12     // 2. Factories e Geração de Objetos
13     interface Animal {
14         String makeSound();
15     }
16
17     static class Dog implements Animal {
18         @Override
19         public String makeSound() {
20             return "Woof!";
21         }
22     }
23
24     static class Cat implements Animal {
25         @Override
26         public String makeSound() {
27             return "Meow!";
28         }
29     }
30
31     static class AnimalFactory {
32         private static final Random RANDOM = new Random();
33
34         public static Supplier<Animal> randomAnimalSupplier() {
35             return () -> RANDOM.nextBoolean() ? new Dog() : new Cat();
36         }
37     }
38
39     public static void main(String[] args) {
40
41         // 2. Factories e Geração de Objetos
42         Supplier<Animal> animalSupplier = AnimalFactory.randomAnimalSupplier();
43         for (int i = 0; i < 5; i++) {
44             System.out.println("Animal " + (i+1) + " faz: " + animalSupplier.get().makeSound());
45         }
46
47     }
48 }
49
```

```
Console  Problems  Debug Shell
<terminated> Factory [Java Application] C:\Program Files\eclipse-java-2022-09-R-win32-x86_64\eclipse\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.win32.x86_64_17.0.4.v20220903-1038\jre\bin
Animal 1 faz: Woof!
Animal 2 faz: Woof!
Animal 3 faz: Meow!
Animal 4 faz: Woof!
Animal 5 faz: Meow!
```

Explicação Detalhada: Factory com Supplier

Este código demonstra o uso da interface Supplier para implementar um padrão Factory Method em Java, gerando objetos aleatoriamente.

Estrutura do Código

Interface Animal

```
interface Animal {
    String makeSound();
}
```

- Define um contrato para todas as classes de animais.
- Declara um método makeSound() que retorna uma String.

Classe Dog

```
static class Dog implements Animal {
    @Override
    public String makeSound() {
        return "Woof!";
    }
}
```

- Implementa a interface Animal.
- Sobrescreve makeSound() para retornar “Woof!”.

Classe Cat

```
static class Cat implements Animal {
    @Override
    public String makeSound() {
        return "Meow!";
    }
}
```

- Implementa a interface Animal.
- Sobrescreve makeSound() para retornar “Meow!”.

Classe AnimalFactory

```
static class AnimalFactory {
    private static final Random RANDOM = new Random();
    public static Supplier<Animal> randomAnimalSupplier() {
        return () -> RANDOM.nextBoolean() ? new Dog() : new Cat();
    }
}
```

- Contém um Random estático para geração de números aleatórios.
- Define um método estático randomAnimalSupplier() que retorna um Supplier<Animal>.
- O Supplier retornado cria aleatoriamente um Dog ou um Cat com base em um valor booleano aleatório.

Método main

```
public static void main(String[] args) {
    Supplier<Animal> animalSupplier = AnimalFactory.randomAnimalSupplier();
    for (int i = 0; i < 5; i++) {
        System.out.println("Animal " + (i+1) + " faz: " + animalSupplier.ge
t().makeSound());
    }
}
```

- Obtém um Supplier<Animal> da AnimalFactory.
- Executa um loop 5 vezes, cada vez:
  - Chama animalSupplier.get() para obter um novo Animal.
  - Chama makeSound() no animal obtido.
  - Imprime o resultado.

Fluxo de Execução

O programa inicia no método main.

↓

Um `Supplier<Animal>` é obtido chamando

↓

`AnimalFactory.randomAnimalSupplier().`

↓

O loop é executado 5 vezes:

- Em cada iteração, `animalSupplier.get()` é chamado, que internamente:
  - Gera um valor booleano aleatório.
  - Cria um novo `Dog` ou `Cat` com base nesse valor.
- O método `makeSound()` é chamado no animal criado.
- O som do animal é impresso junto com seu número de ordem.

Observe que randomAnimalSupplier() não retorna um objeto do tipo Dog ou Cat :

1. **Método randomAnimalSupplier() na AnimalFactory:**
   - Este método **retorna um Supplier<Animal>.** Internamente, esse Supplier<Animal> é criado usando **uma expressão lambda**, mas essa expressão lambda é encapsulada dentro do método randomAnimalSupplier().
   - Quando você chama AnimalFactory.randomAnimalSupplier(), o método é executado e retorna o Supplier<Animal> que foi definido pela expressão lambda interna.
2. **Atribuição do Supplier<Animal>:**
   - A variável animalSupplier recebe o Supplier<Animal> que foi retornado por randomAnimalSupplier(). Portanto, animalSupplier **agora contém uma referência a um Supplier<Animal>** que pode ser usado para gerar um objeto Animal (seja Dog ou Cat).

Por que essa abordagem é útil:

- **Encapsulamento:** Você está encapsulando a lógica de criação do Supplier<Animal> dentro do método randomAnimalSupplier(). Isso significa que qualquer código que precisar de um Supplier<Animal> pode simplesmente chamar este método sem se preocupar com os detalhes de implementação.
- **Reutilização:** O método randomAnimalSupplier() pode ser reutilizado em várias partes do código, permitindo que você mantenha a lógica de criação centralizada.
- **Flexibilidade:** Se a lógica de criação dos objetos Animal mudar no futuro (por exemplo, você adiciona novos tipos de animais), você só precisa modificar o método randomAnimalSupplier() e todo o código que usa esse método se beneficiará da atualização automaticamente.

3. Valores Dinâmicos

Quando um valor não é constante e pode variar a cada solicitação, *Supplier* oferece uma maneira conveniente de encapsular a lógica de geração desse valor. Por exemplo, ele pode

ser usado para fornecer valores provenientes de fontes externas, como APIs, bancos de dados ou cálculos complexos, sem expor a lógica de obtenção desses valores para o código que os consome.

Exemplo de código:

```
2
3  import java.util.function.Supplier;
8
9
10 public class SupplierValoresDinamicos {
11
12
13     // 3. Valores Dinâmicos
14     static class DynamicValueProvider {
15         private static final Random RANDOM = new Random();
16
17         public static Supplier<Double> randomStockPriceSupplier() {
18             return () -> 100 + RANDOM.nextDouble() * 10; // Preço base 100 com variação de até 10
19         }
20     }
21
22
23
24     public static void main(String[] args) {
25
26
27         // 3. Valores Dinâmicos
28         Supplier<Double> stockPriceSupplier = DynamicValueProvider.randomStockPriceSupplier();
29         for (int i = 0; i < 5; i++) {
30             System.out.printf("Preço atual da ação: $%.2f%n", stockPriceSupplier.get());
31         }
32     }
33 }
34
```

Primeiramente, temos uma classe chamada SupplierValoresDinamicos.

Dentro desta classe, há uma classe estática interna chamada DynamicValueProvider. Esta classe é responsável por fornecer valores dinâmicos.

Na classe DynamicValueProvider, há um campo estático final RANDOM do tipo Random, que será usado para gerar números aleatórios.

O método estático randomStockPriceSupplier() retorna um Supplier<Double>. Aqui está o uso principal do Supplier:

```
public static Supplier<Double> randomStockPriceSupplier() {
    return () -> 100 + RANDOM.nextDouble() * 10;
}
```

Este método retorna um Supplier que, quando invocado, gera um preço de ação aleatório. O preço base é 100, com uma variação aleatória de até 10.

No método main, criamos uma instância do Supplier:

```
Supplier<Double> stockPriceSupplier = DynamicValueProvider.randomStockPriceSupplier();
```

Em seguida, usamos um loop para “gerar” 5 preços de ações:

```
for (int i = 0; i < 5; i++) {
    System.out.printf("Preço atual da ação: $%.2f%n", stockPriceSuppli
er.get());
}
```

Cada vez que chamamos stockPriceSupplier.get(), o Supplier executa a função lambda definida anteriormente, gerando um novo preço aleatório.

A vantagem de usar um Supplier neste cenário é que podemos definir uma lógica para gerar valores e usá-la repetidamente, obtendo um novo valor cada vez que chamamos o método get(). Isso é particularmente útil para simular dados dinâmicos ou para casos onde queremos adiar a geração de um valor até que ele seja realmente necessário.

4. Conteúdo Personalizado

Em interfaces gráficas e sistemas de mensagens, Supplier pode ser utilizado para fornecer conteúdo ou mensagens personalizados com base em contexto, como a configuração de um usuário ou o estado atual de uma aplicação. Isso permite que o conteúdo seja gerado de forma dinâmica, dependendo da situação.

Exemplo de código:

```
 9
10 public class SupplierConteudoPersonalizado {
11
12
13     // 4. Conteúdo Personalizado
14     static class UserGreeting {
15         public static Supplier<String> greetingSupplier(String username) {
16             return () -> {
17                 int hour = java.time.LocalTime.now().getHour();
18                 if (hour < 12) return "Bom dia, " + username + "!";
19                 else if (hour < 18) return "Boa tarde, " + username + "!";
20                 else return "Boa noite, " + username + "!";
21             };
22         }
23     }
24
25     public static void main(String[] args) {
26
27
28         // 4. Conteúdo Personalizado
29         Supplier<String> greetingSupplier = UserGreeting.greetingSupplier("Alice");
30         System.out.println(greetingSupplier.get());
31
32     }
33 }
34
```

O código acima demonstra o uso de um Supplier para gerar uma saudação personalizada baseada no nome do usuário e na hora atual. A classe UserGreeting contém um método estático greetingSupplier que recebe um nome de usuário e retorna um Supplier<String>. Este Supplier, quando invocado, determina a hora atual e retorna uma saudação apropriada (bom dia, boa tarde ou boa noite) junto com o nome do usuário. No método main, é criado um Supplier para saudar "Alice", e então este Supplier é usado para gerar e imprimir uma saudação. A vantagem deste approach é que a saudação é gerada dinamicamente cada vez que o Supplier é chamado, permitindo que a mensagem seja sempre atualizada conforme a hora do dia, sem necessidade de recriar o Supplier.

5. Manipulação de Recursos

Supplier pode ser aplicado para fornecer recursos como conexões a bancos de dados ou sessões de redes. Ao encapsular a lógica de obtenção do recurso em um Supplier, torna-se mais fácil gerenciar a criação e fechamento desses recursos, promovendo uma abordagem mais modular e reutilizável.

Exemplo de código:

```java
public class SupplierManipulacaoRecursos {

    // 5. Manipulação de Recursos
    static class DatabaseConnection {
        public static Supplier<Connection> connectionSupplier(String url, String user, String password) {
            return () -> {
                try {
                    return DriverManager.getConnection(url, user, password);
                } catch (SQLException e) {
                    throw new RuntimeException("Falha ao conectar ao banco de dados", e);
                }
            };
        }
    }



    public static void main(String[] args) {


        // 5. Manipulação de Recursos
        Supplier<Connection> connectionSupplier = DatabaseConnection.connectionSupplier(
            "jdbc:mysql://localhost:3306/mydb", "user", "password");
        try (Connection conn = connectionSupplier.get()) {
            System.out.println("Conexão bem-sucedida!");
        } catch (Exception e) {
            System.err.println("Erro ao conectar: " + e.getMessage());
        }

    }
}
```

Este código demonstra o uso de um Supplier para encapsular a lógica de criação de uma conexão de banco de dados. A classe DatabaseConnection contém um método estático connectionSupplier que retorna um Supplier<Connection>.

Este Supplier, quando invocado, tenta estabelecer uma conexão com o banco de dados usando os parâmetros fornecidos (URL, usuário e senha). No método main, um Supplier é criado com as credenciais do banco de dados, e então é usado para obter uma conexão dentro de um *bloco try-with-resources*, que garante que a conexão será fechada automaticamente após o uso.

Note que a conexão com o banco de dados só é estabelecida quando connectionSupplier.get() é chamado, não quando o Supplier é criado.

6. Testes e Mocking

Em testes automatizados, Supplier pode ser utilizado para fornecer valores de maneira controlada, facilitando a criação de cenários de teste. Ao substituir implementações reais por Suppliers que retornam valores pré-definidos, os testes podem ser isolados e focados em comportamentos específicos.

Exemplo de código:

```java
public class SupplierMocking {

    // 6. Testes e Mocking
    static class UserService {
        private Supplier<String> currentUserSupplier;

        public UserService(Supplier<String> currentUserSupplier) {
            this.currentUserSupplier = currentUserSupplier;
        }

        public String getWelcomeMessage() {
            return "Bem-vindo, " + currentUserSupplier.get() + "!";
        }
    }

    public static void main(String[] args) {


        // 6. Testes e Mocking
        UserService userService = new UserService(() -> "Bob");
        System.out.println(userService.getWelcomeMessage());
    }
}
```

Este código é um excelente exemplo de como Suppliers podem ser usados para facilitar testes e mocking em Java:

Injeção de Dependência: A classe UserService não cria diretamente a lógica para obter o usuário atual. Em vez disso, ela recebe um Supplier<String> através do construtor. Isso é uma forma de injeção de dependência, que torna a classe mais flexível e testável.

Abstração da Fonte de Dados: O Supplier<String> abstrai a fonte do nome do usuário. Na prática, isso poderia vir de um banco de dados, um serviço de autenticação, ou qualquer outra fonte.

Facilidade de Mocking: Para testes, você pode facilmente criar um mock ou stub do Supplier<String> sem precisar modificar a classe UserService. Isso permite testar UserService isoladamente.

Simulação de Diferentes Cenários: No método main, vemos um exemplo simples de como podemos "mockar" o comportamento:

```
UserService userService = new UserService(() -> "Bob");
```

Aqui, estamos fornecendo um Supplier que sempre retorna "Bob". Em um ambiente de teste, poderíamos facilmente trocar isso por diferentes implementações para testar vários cenários.