Quais interfaces funcionais completam o código a seguir, presumindo que a variável r exista? (Escolha todas as que se aplicam.)

```
6: ______ x = r.negate();
7: ______ y = () -> System.out.println();
8: ______ z = (a, b) -> a - b;
```

A. BinaryPredicate<Integer, Integer>
B. Comparable<Integer>
C. Comparator<Integer>
D. Consumer<Integer>
E. Predicate<Integer>
F. Runnable
G. Runnable<Integer>

# ANÁLISE DA QUESTÃO

Vamos verificar o código e as opções de interfaces funcionais que podem completá-lo. As interfaces funcionais são interfaces que possuem exatamente um método abstrato, e são bastante utilizadas em expressões lambda no Java. Vamos examinar cada linha do código e verificar quais interfaces funcionais se aplicam:

**Linha 6: ______ x = r.negate();**

Aqui, *r.negate()* sugere que r é **uma instância de uma interface funcional que possui o método** *negate()*. As interfaces funcionais em Java que possuem o método *negate()*:

1. **Predicate<T>:**
   - *Predicate<T>* é uma interface funcional que representa um predicado (função que retorna um booleano) com um argumento do tipo T. Ela define métodos como *test(T t)*, *and(Predicate<? super T> other)*, *or(Predicate<? super T> other)* e *negate()*. O método negate() retorna um predicado que representa a negação do predicado original.

Exemplo:

```
Predicate<Integer> isPositive = x -> x > 0;
Predicate<Integer> isNotPositive = isPositive.negate();
```

2. **BiPredicate<T, U>:**
   - *BiPredicate<T, U>* é uma interface funcional que representa um predicado com dois argumentos, T e U. **Ela estende** *Predicate<T>* e define o método *test(T t, U u)* além do método *negate()*, que permite negar o predicado original.

Exemplo:

```
BiPredicate<Integer, Integer> areEqual = (x, y) -> x.equals(y);
BiPredicate<Integer, Integer> areNotEqual = areEqual.negate();
```

Essas interfaces são úteis quando se trabalha com lógica booleana complexa e são amplamente utilizadas em programação funcional e no contexto de APIs de Streams em Java.

Portanto, a interface correta para completar essa linha é: **E. *Predicate<Integer>***

**Linha 7: ______ y = () -> System.out.println();**

Essa linha mostra uma expressão lambda que não recebe nenhum argumento e não retorna nada, apenas executa

um comando. Isso é característico da interface *Runnable*, que tem um único método *run()* sem parâmetros e sem valor de retorno.

A interface funcional em Java que não tem parâmetros nem função de retorno é a *Runnable*. A interface *Runnable* deve ser implementada por qualquer classe cujas instâncias são destinadas a serem executadas por uma thread. A classe deve definir um método sem argumentos chamado *run*().

1. **Interface Runnable**

A interface Runnable é definida da seguinte forma:

```
@FunctionalInterface
public interface Runnable {
    void run();
}
```

O método abstrato da interface Runnable é o *run()*. Ele não possui parâmetros e não retorna nenhum valor (void). Este método contém o código que será executado quando a instância da *Runnable* for executada.

Uso da Interface Runnable

```
Runnable task = () -> {
    System.out.println("Executando tarefa...");
    // Código da tarefa que será executado
};

// Executando a tarefa em uma nova thread
Thread thread = new Thread(task);
thread.start();
```

A *Runnable* é comumente utilizada para representar tarefas ou operações que precisam ser executadas assincronamente. Por exemplo:

Neste exemplo, *task* é uma instância de *Runnable* que contém o código a ser executado na nova thread quando ela for iniciada. O método *run()* será chamado quando a thread for executada, permitindo que o código dentro do lambda seja executado de forma assíncrona.

Portanto, a interface correta para completar essa linha é: ***F. Runnable***

**Linha 8: _____ z = (a, b) -> a - b;**

Existem algumas interfaces funcionais em Java que são projetadas para lidar com dois operandos em operações funcionais. Aqui estão as principais interfaces funcionais que utilizam dois operandos:

1. **BiFunction<T, U, R>:**
   - *BiFunction<T, U, R>* é uma interface funcional que representa uma função que aceita dois argumentos (T e U) e produz um resultado (R). Ela possui um método abstrato *apply(T t, U u)* que realiza a operação e retorna um resultado do tipo R.

Exemplo:

```
BiFunction<Integer, Integer, Integer> sum = (a, b) -> a + b;

Integer result = sum.apply(5, 3); // Resultado: 8
```

2. **BiConsumer<T, U>:**
   - *BiConsumer<T, U>* é uma interface funcional que representa uma operação que aceita dois argumentos (T e U) e não retorna nenhum resultado (void). Ela possui um método abstrato *accept(T t, U u)* que realiza a operação desejada.

Exemplo:

```
BiConsumer< String, Integer > printRepeated = (str, count) -> {
    for (int i = 0; i < count; i++) {
        System.out.println(str);
    }
};

printRepeated.accept("Hello", 3); // Imprime "Hello" três vezes
```

**3. BiPredicate<T, U>:**

- *BiPredicate<T, U>* é uma interface funcional que representa um predicado (função booleana) que aceita dois argumentos (T e U). Ela possui um método abstrato *test(T t, U u)* que retorna um valor booleano (true ou false) baseado na avaliação do predicado.

Exemplo:

```
BiPredicate< Integer, Integer > areEqual = (a, b) -> a.equals(b);

boolean result = areEqual.test(5, 5); // Resultado: true
```

**4. Comparator<T>:**

- Comparator<T> é uma interface funcional que define um contrato para comparar dois objetos do tipo T. Ela possui um método abstrato compare(T o1, T o2) que retorna um número inteiro que indica a ordem relativa dos objetos.

Exemplo:

```
List<String> names = Arrays.asList("John", "Jane", "Adam", "Emily");

 // Ordenando usando um Comparator
 Comparator<String> byLength = (s1, s2) -> Integer.compare(s1.length(),
s2.length());

 names.sort(byLength);

 System.out.println(names);  // Output: [Adam, John, Jane, Emily]
```

Essas interfaces funcionais são úteis em situações onde é necessário lidar com operações que envolvem dois argumentos e podem ser utilizadas em diversas situações:

- *BiFunction<T, U, R>*: Utilizada para operações de transformação, onde dois argumentos são processados para produzir um resultado. Exemplo: cálculos matemáticos com dois operandos.
- *BiConsumer<T, U>*: Usada para processamento de dados, onde uma ação é realizada com dois argumentos, mas nenhum resultado é retornado (void). Exemplo: impressão de informações que dependem de dois parâmetros.
- *BiPredicate<T, U>*: Utilizada para avaliação de condições, onde dois argumentos são testados para determinar um valor booleano (true ou false). Exemplo: verificação de relações entre dois valores para tomar uma decisão.
- *Comparator<T>*: Fundamental para operações de ordenação, onde é utilizado para comparar dois objetos do mesmo tipo (T). Exemplo: ordenação de uma lista de objetos com base em critérios específicos, como tamanho ou valor.

Portanto, a interface correta para completar essa linha, por eliminação, é: ***C. Comparator<Integer>***

Note a **pegadinha no item A** 😉 pois a interface *BinaryPredicate<Integer, Integer>* **não existe**

Também não confunda as interfaces *Comparator<T>* com *Comparable<T>,* elas tem propósitos distintos:

**1. Interface Comparable**

- **O que é?** A interface genérica Comparable é usada para definir a ordem natural dos objetos de

uma classe.

- **Como funciona?** Uma classe que implementa Comparable precisa implementar o método abstrato compareTo() da interface. Esse método compara o objeto atual com outro objeto da mesma classe e retorna um valor inteiro:
  - 0 se os objetos forem iguais,
  - Um valor negativo se o objeto atual for menor,
  - Um valor positivo se o objeto atual for maior.
- **Quando usar?** Use Comparable quando a classe tiver uma ordem de classificação natural que faz sentido em múltiplos contextos. Por exemplo, ordenar números, strings, datas, etc.

Exemplo:

```
public class Pessoa implements Comparable<Pessoa> {
    private String nome;
    private int idade;

    @Override
    public int compareTo(Pessoa outraPessoa) {
        return this.idade - outraPessoa.idade; // Comparação baseada na idade
    }

    // getters e setters
}
```

**2. Interface Comparator**

- **O que é?** A interface funcional Comparator é usada para definir uma ordem de classificação separada da ordem natural dos objetos.
- **Como funciona?** Você cria uma classe que implementa Comparator e define o método compare(). Esse método compara dois objetos de uma classe e retorna um valor inteiro:
  - 0 se os objetos forem iguais,
  - Um valor negativo se o primeiro objeto for menor,
  - Um valor positivo se o primeiro objeto for maior.
- **Quando usar?** Use Comparator quando você precisar de múltiplas maneiras de ordenar os objetos de uma classe ou quando não quiser ou não puder modificar a classe dos objetos a serem comparados.

Exemplo:

```
public class ComparadorPorNome implements Comparator<Pessoa> {
    @Override
    public int compare(Pessoa p1, Pessoa p2) {
        return p1.getNome().compareTo(p2.getNome()); // Comparação baseada
no nome
    }
}
```

Resumindo, as alternativas corretas para a questão acima são:

- Linha 6: **E.** Predicate<Integer>
- Linha 7: **F.** Runnable
- Linha 8: **C.** Comparator<Integer>