Na hierarquia de exceções em Java, a classe raiz é **Throwable**. Ela possui duas subclasses principais:

- **Error**: Representa erros que fogem ao controle do programa (como falta de memória, problemas na JVM ou erros de disco). Geralmente, esses erros não são capturados, pois não correspondem a condições esperadas ou tratáveis pelo código.
- **Exception**: Engloba a maioria das exceções ocorridas durante a execução do programa, incluindo subclasses específicas como **NumberFormatException** e **IllegalArgumentException**.

No exemplo citado, duas exceções que exibem mensagens na tela podem ser capturadas por um único bloco `catch` se declararmos a captura para a superclasse **Exception**, que agrupa ambas. Entretanto, quando é necessário tratar exceções de maneira individualizada, é essencial listar os `catch` de classes mais específicas primeiro e, ao final, incluir um bloco para **Exception**, evitando que a captura genérica impeça os tratamentos mais específicos.

Essa organização garante que as exceções sejam tratadas adequadamente, respeitando a hierarquia de classes e assegurando uma estrutura de tratamento robusta e clara.

Em Java, as exceções se dividem em duas categorias principais:

1. **Exceções Verificadas (Checked Exceptions):** Essas exceções são subclasses de **Exception** que **não** passam pela classe **RuntimeException**. São consideradas “checadas” porque o compilador obriga o programador a tratá-las explicitamente. Isso significa que todo método que possa lançá-las precisa:
    - Envolver o código em um bloco `try...catch`, ou
    - Declarar a exceção na cláusula `throws` do método. Exemplos clássicos são **FileNotFoundException** e **IOException**, frequentemente encontradas em operações de leitura e escrita de arquivos.
2. **Exceções Não Verificadas (Unchecked Exceptions):** São aquelas que herdam de **RuntimeException**. Por não serem “checadas” pelo compilador, podem – mas não precisam – ser capturadas ou declaradas. Geralmente, elas indicam problemas de lógica no programa (como **IllegalArgumentException**), e seu tratamento fica a critério do desenvolvedor, muitas vezes para melhorar a experiência do usuário.

Tratamento Geral: Quando um método pode lançar diversas exceções verificadas, há duas abordagens comuns:

- Tratar cada exceção de forma individual em blocos `catch` separados;
- Declarar uma superclasse comum (por exemplo, `throws Exception`) para simplificar o código.

Contudo, se optar por múltiplos blocos `catch`, é fundamental que os blocos para exceções mais específicas apareçam antes do bloco que captura exceções mais gerais, para que o tratamento adequado não seja impedido pelo "catch" genérico.

Exemplo Prático: Imagine um método que utiliza o objeto **FileReader** para ler dados de um arquivo. Se o construtor lança uma **FileNotFoundException** e o método `close()` lança uma **IOException**, ambos são exceções verificadas. Assim, o método deve ou envolvê-los em blocos `try...catch` ou declarar essas exceções em sua assinatura com `throws`. Isso garante que, em todas as chamadas desse método, o programador já tenha considerado o tratamento adequado para esses possíveis erros.

Essa abordagem não só previne falhas inesperadas, mas também promove um código mais robusto e claro, evidenciando as exceções que podem ocorrer e direcionando o programador a lidar com elas de forma consciente.

Explorando além, é interessante notar que essa disciplina forçada no tratamento de exceções verificadas incentiva uma análise mais aprofundada dos pontos críticos em operações que podem falhar, como I/O e acesso a recursos externos, aumentando assim a confiabilidade da aplicação.

O bloco `finally` é uma ferramenta essencial no tratamento de exceções em Java, pois garante que um trecho de código seja executado sempre, independentemente de ocorrer uma exceção ou não. Em geral, usamos o `finally` para liberar recursos que foram adquiridos previamente, como fechar conexões com bancos de dados ou arquivos abertos – mesmo que o bloco `try` tenha terminado abruptamente por conta de um erro.

Você pode combiná-lo com um bloco `catch` para tratar uma exceção e, em seguida, realizar a limpeza necessária. Porém, se não houver a necessidade de tratar a exceção dentro do método (por exemplo, apenas limpar recursos), é perfeitamente válido utilizar a construção `try` com `finally` sem o bloco `catch`. Essa abordagem simplifica o código quando a única ação necessária diante de qualquer ocorrência é garantir o fechamento ou a liberação de recursos.

Em resumo, conforme os cenários apresentados no quadro 5, o uso do bloco `finally` melhora a robustez e a manutenção do código ao lidar com recursos críticos, seja em uma estrutura `try...catch...finally` ou em um `try...finally` mais enxuto.