Instituto Federal de Ciência, Educação e Tecnologia do Piauí – IFPI

Aluno: Vinícius Gomes Araújo Costa

Disciplina: Programação Orientada a Objetos (POO)

Professor: Ely Miranda

Exercício 08

1. O **tratamento de erros em TypeScript** é uma parte essencial do desenvolvimento robusto de **aplicações**. Ele permite que você lide com **situações inesperadas ou erros que possam ocorrer durante a execução do programa**, sem que o programa termine de forma abrupta. Em TypeScript, o tratamento de erros é feito usando exceções, que são objetos especiais que representam um erro ou uma condição excepcional.

Um bom software é aquele que está sujeito à tratamento de erros com eficiência e robustez. Isto é, a aplicação não trava, não desconecta fácil caso dependa de um servidor web, não possui interrupções no decorrer do fluxo e também os erros devem ser tratados seja eles quaisquer para o software ser mais robusto e não suscetível a erros.

Os três casos mais comuns de erros são:

I) Tratamento de Exceções:

Esse método envolve o uso de blocos **try-catch** (ou equivalentes) para capturar exceções que podem ocorrer durante a execução de um código. Exceções são eventos anômalos que interrompem o fluxo normal de execução do programa.

Exemplo: Código disponível no **VS Code**.

II) Retornos de erro (Verificações Manuais):

Nesse tipo de tratamento, o código verifica explicitamente se os resultados de funções ou operações são válidos. O tratamento de erro é feito sem exceções, geralmente através de valores de retorno especiais.

Exemplo: Código disponível no **VS Code**.

III) Tratamento via retornos de tipos opcionais:

Em algumas linguagens funcionais e em linguagens como **Rust**, **Swift** ou **Kotlin**, é comum o uso de tipos opcionais (**Option, Maybe**) para representar a presença ou ausência de um valor válido. O tratamento de erros pode ser feito verificando se o valor está presente ou não.

Exemplo: Código não disponível no **VS Code** por motivos de não instalação dos compiladores para cada linguagem de programação citada acima. Fica somente a curiosidade da pesquisa nos links abaixo:

Tratamento de erros em Swift

Uma das grandes novidades do Swift 2 foi o suporte para tratamento de erros (em inglês, *error handling*). Mas o que isso quer dizer?
Algumas operações (geralmente funções) não oferecem a garantia de completar sua execução ou mesmo de produzir um retorno útil. Em Swift, usamos *optionals* para representar uma ausência de valor (*nil*). Porém, quando uma função retorna *nil,* é porque pode ter acontecido um erro e, muitas vezes, queremos entender o que causou este erro, para que nosso programa possa responder de acordo. É importante diferenciar as diversas formas que uma operação pode falhar e comunicar ao usuário adequadamente.
A forma mais comum de resolver o problema de tratamentos de erros com Objective-C é passar uma variável adicional de erro no método e, caso haja algum erro, o método fica responsável por popular essa variável com o objeto de erro, além de retornar *nil*.
Essa abordagem é confusa e não intuitiva. Esse é um exemplo comum em Objective-C:

Continua no link https://imasters.com.br/desenvolvimento/tratamento-de-erros-em-swift

Tratamento de Erros em RUST

O comprometimento de Rust à segurança se extende ao tratamento de erros. Erros são um fato da vida em software, portanto Rust possui um número de *features* para lidar com situações em que algo dá errado. Em vários casos, Rust requer que você reconheça a possibilidade de um erro acontecer e aja preventivamente antes que seu código compile. Esse requisito torna seu programa mais robusto ao assegurar que voce irá descobrir erros e lidar com eles apropriadamente antes de mandar seu código para produção!

Rust agrupa erros em duas categorias principais: *recuperáveis* e *irrecuperáveis*. Erros recuperáveis são situações em que é razoável reportar o problema ao usuário e tentar a operação novamente, como um erro de arquivo não encontrado. Erros irrecuperáveis são sempre sintomas de bugs, como tentar acessar uma localização além do fim de um *array*.

A maioria das linguagens não distingue esses dois tipos de erros e lida com ambos da mesma maneira usando mecanismos como exceções. Rust não tem exceções. Em vez disso, ele tem o valor `Result<T, E>` para erros recuperáveis e a macro `panic!` que para a execução ao encontrar um erro irrecuperável. Esse capítulo cobre primeiro como chamar `panic!` e depois fala sobre retornar valores `Result<T, E>`. Adicionalmente, vamos explorar o que se levar em consideração para decidir entre tentar se recuperar de um erro ou parar execução.

Continua no link https://rust-br.github.io/rust-book-pt-br/ch09-00-error-handling.html

Tratamento idiomático de erros e exceções em Kotlin

*Nesse texto, nosso desenvolvedor Android Pablo Hildo apresenta três ferramentas do Kotlin (linguagem de programação para Android) para realizar tratamento de erros e exceções. Elas possibilitam usar melhor os recursos do Kotlin e da JVM (*Java Virtual Machine*), além de garantir a escrita de código cada vez mais legível. Acompanhe!*

Desde a versão 1.0, o Kotlin traz em sua especificação três ferramentas para realizar tratamento de erros e exceções de forma idiomática e clara: *require*, *check* e *assert*. Cada uma destas é aplicável em casos diferentes e devem ser utilizadas de acordo com o contexto do tratamento.

Todas elas são particularmente úteis porque oferecem uma subclasse previsível de *Exception*, como tipo, permitindo o bom uso de recursos já existentes da JVM e evitando a reescrita desnecessária de classes para adequar exceções específicas demais. Isso pode simplificar, por exemplo, o processo de validação.

Require

O require é o mais claro e mais utilizável e serve principalmente para validações. É

definido como:

A *lazyMessage* é opcional.

Você deve chamar o *require* para checar uma condição, geralmente validando um parâmetro, e jogar uma *IllegalArgumentException* com uma mensagem específica em seu corpo.

Um exemplo praticável em validação é:

Nesse caso, **exigimos** que a senha não seja *nullOrBlank* e passamos a *lazyMessage* "*emptyPasswordMessage*". Exigimos que o tamanho seja precisamente 6 com a *lazyMessage* "*passwordMessage*" e que a senha seja igual à confirmação, com a *lazyMessage* "*unmatchedConfirmationMessage*". Se qualquer dessas condições exigidas for falsa, a execução da função para naquele momento e joga a exceção, que pode ser

obtida (com a *lazyMessage* passada) por meio de um *try catch*:

Isso é extremamente positivo porque evita o tratamento de exceções genéricas, centraliza exceções do tipo em uma classe já existente e garante que o código fique muito legível.

Check

O check exige menos explicação. Ele é exatamente igual ao require, mas joga uma *IllegalStateException*, então existe em benefício do código conciso, idiomático e legível. Levando em conta o nome da exceção, fica claro que seu objetivo é validar estado.

Temos, por exemplo, o caso seguinte:

Nessa situação, o bloco de código posterior a *check* só será executado se *loadingLiveData.value* for *true*. Caso contrário, uma exceção do tipo supracitado é jogada.

Assert

O *assert*, por sua vez, é mais "de nicho" e se aplica em uma situação bem específica, que geralmente é verificação da validade de parâmetro, costumeiramente em *debug* local. Exatamente por isso é possível chamar de verificação da validade e não de validação, já que não serve no contexto usual de validação.

Ele joga um AssertionError, que só vai ser jogado de fato caso a flag -ea esteja ativada. Se não estiver, é ignorado no código. Seu uso também difere do usual porque no que diz respeito à JVM: “um *Error* é uma subclasse de *Throwable* que indica problemas sérios que uma aplicação razoável não deveria tentar manipular. A maioria desses erros são condições anormais”.

Conclusão

Com a aplicação desses três métodos, é possível utilizar melhor tanto os recursos do Kotlin quanto da JVM, assim como garantir a escrita de código cada vez mais idiomático e legível.

No geral, seu uso é pouco comum e não costuma ser reforçado, mas sua aplicação é visualizada com importância na documentação de outros recursos da linguagem. A manipulação dessas exceções da biblioteca padrão representam um passo adicional na tentativa de homogeneizar cada vez mais o código de projetos.

Continua no link https://blog.cubos.io/tratamento-idiomatico-de-erros-excecoes-em-kotlin/#

IV) Desconsiderando operações:

Quando falamos de tratamento de erros visando a robustez de software, uma das abordagens mais frequentes é o uso de validações de dados de entrada e estruturas críticas, especialmente em cenários de comunicação com API’s, leitura/escrita de arquivos, e manipulação de dados fornecidos por usuários.

Um exemplo prático é o tratamento de erros ao tentar acessar propriedades de objetos que podem não existir ou quando estamos lidando com dados indefinidos. Vou considerar um cenário onde tentamos acessar dados de uma API, e precisa-se garantir que os dados recebidos estejam no formato esperado.

Exemplo: Código disponível no **VS Code**, mostrando uma **Validação de Dados em uma API**.

2. Cada um dos **4 métodos de tratamento de erros mencionados** (exceções, verificações manuais e retornos de erro, tipos opcionais e desconsiderando operações) tem suas vantagens, mas também apresenta limitações dependendo do contexto de uso. Irei explorar essas limitações:

I) Tratamento de Exceções (Try-Catch):

Limitações:

Complexidade e performance: Exceções podem ser computacionalmente custosas, especialmente em linguagens como Java e C++ que exigem o empilhamento e desempilhamento da pilha de chamadas. Isso pode impactar a performance sem sistemas de tempo real ou em códigos que lidam com grandes volumes de dados.

Abuso de exceções: Se exceções forem usadas como controle de fluxo normal (em vez de apenas para erros), isso pode tornar o código mais confuso e difícil de seguir. Além disso, isso viola o princípio de que exceções deveriam ser usadas para "situações excepcionais".

Ocultação de erros: Quando mal utilizado, o tratamento de exceções pode levar a erros "silenciosos". Por exemplo, se um bloco "**catch**" captura uma exceção, mas não a trata adequadamente, o erro pode passar despercebido e criar problemas mais complexos.

Verificações nem sempre completas: Pode ser difícil garantir que todas as exceções possíveis sejam tratadas corretamente, especialmente em sistemas grandes ou integrados com bibliotecas externas.

II) Verificações Manuais (Retornos de Erro):

Limitações:

Complexidade do código*:* O uso de verificações manuais pode tornar o código cada vez mais verboso e propenso a erros, uma vez que o desenvolvedor precisa lembrar de verificar o retorno de cada função. Isso pode ser esquecido, resultando em bugs difíceis de rastrear.

Falta de padronização: Em projetos grandes, diferentes desenvolvedores podem adotar diferentes convenções para códigos de erro, o que pode levar a uma inconsistência na forma como os erros são tratados.

Tratamento imperfeito: Muitas vezes, o retorno de erro não leva a um tratamento correto e profundo do erro. Desenvolvedores podem simplesmente ignorar o valor retornado ou tomar decisões inadequadas, resultando em falhas silenciosas no sistema.

Não separe lógica de erro da lógica de negócio: Quando usado excessivamente, pode poluir a lógica do programa com verificações constantes, dificultando a manutenção e compreensão.

III) Tratamento de via retornos de tipos opcionais:

Limitações:

Incompatibilidade com linguagens mais antigas: Esse método é mais comum em linguagens modernas como Rust, Swift e Haskell. Em linguagens como JavaScript, C ou C++, essa abordagem não é nativa e teria que ser implementada manualmente, o que pode ser trabalhoso.

Complexidade na composição de funções: O uso de tipos opcionais pode complicar a composição de funções, especialmente em casos de encadeamento de chamadas, pois cada função deve lidar explicitamente com a possibilidade de **None** ou **Null**. Isso pode tornar o código mais detalhado e dificultar a leitura.

Maior carga cognitiva para o desenvolvedor: O desenvolvedor precisa estar ciente e lembrar constantemente de manipular o tipo opcional em vez de lidar diretamente com o valor. Isso pode aumentar a carga cognitiva, especialmente para desenvolvedores menos familiarizados com esse paradigma.

Propagação de erros nem sempre clara: Quando usado em cascata (com muitos tipos opcionais), o rastreamento e a depuração de um erro podem se tornar mais difíceis, pois você precisa lidar com várias verificações condicionais.

IV) Comprometimento de robustez e clareza de código:

Limitações:

No caso de desconsideração de operações, ou seja, quando tratamos erros de forma que o programa possa continuar sem realizar uma operação específica devido a falhas, existem algumas limitações de uso em TypeScript que podem comprometer a robustez e a clareza dos códigos.

Confiabilidade do sistema: ao ignorar problemas importantes.

Clareza: A falta de clareza no código torna difícil saber quando e porquê algo deu errado.

Manutenibilidade: Dificultando a depuração e rastreamento de erros.

Para evitar esses tipos de limitações, é importante adotar boas práticas, como:

Logar erros corretamente;

Propagar exceções quando necessário;

Não ignorar erros críticos e tratar os erros de forma adequada ao contexto.

Resumo de tudo mencionado anteriormente:

Comparativo Geral das limitações:

Exceções podem resultar em código mais organizado, mas podem ser abusadas, usadas para controlar o fluxo normal do programa e ocultar problemas.

Verificações manuais são claras, mas podem resultar em código verboso e fácil de esquecer.

Tipos opcionais fornecem um controle mais explícito sobre os valores válidos, mas podem não ser suportados nativamente por muitas linguagens e podem aumentar a complexidade do código.

Cada abordagem deve ser usada com cautela, levando em consideração o contexto do projeto e as melhores práticas da linguagem em questão. Tendo cuidado, outrossim, com

falhas silenciosas e falta de verificação em códigos incompletos, levando o desenvolvedor a criar a incerteza de códigos em cada caso. O tratamento de erros em TypeScript é um processo atrelado a interface texto, quase uma perfeita e invisível interface gráfica.

4 Para resolver o problema no método **transferir**, é importante ajustar alguns detalhes no código, principalmente no método **sacar**. Em TypeScript, o método **sacar** que lance uma exceção deve ser tratado corretamente no método **transferir**. O método **sacar** atualmente não retorna um **boolean**, mas sim lança uma exceção se o saldo for insuficiente. Então, adaptando o código como disponibilizado no **VS Code**, vamos adaptar o código para capturar essa exceção e retornar um valor adequado de **sucesso** ou **falha na transferência**.

Correção e Implementação:

- Método **sacar** já lança uma exceção se o saldo for insuficiente;
- O método **transferir** deve capturar essa exceção e retornar um **boolean**.

Explicação:

- O método **sacar**: continua lançando uma exceção se o valor do saque for maior do que o saldo;
- Método **transferir**: agora o método captura essa exceção no bloco **try-catch** e retorna **false** se a transferência falhar (saldo insuficiente), ou **true** se for bem-sucedida;
- **Teste**:
- A conta 1 (**conta1**) tem saldo de 100, mas a tentativa de transferir 200 falha;
- O **catch** captura a exceção, exibindo a mensagem de erro “Saldo insuficiente para realizar o saque!”;
- O resultado da transferência é **false**, e os saldos das contas permanecem inalterados;

O que ocorreu:

Ao tentar transferir um valor maior do que o saldo disponível, o método **sacar** da conta origem lançou uma exceção. Isso foi capturado no **try-catch** dentro do método **transferir**, e o código retornou **false**, indicando que a transferência falhou. Essa abordagem garante que erros de saldo insuficiente sejam tratados adequadamente.

Código da questão 04 disponível no **VS Code**.

5. A propagação de exceções em TypeScript ocorre quando uma exceção lançada em um método pode ser capturada e tratada por outros métodos na cadeia de chamadas. No caso dessa questão, se uma exceção for lançada no método **conta.sacar**, ela pode ser capturada pelo método **conta.transferir** e, por sua vez, pelo método **banco.transferir**. O controle sobre a exceção permite que possamos garantir que o erro seja tratado adequadamente em qualquer nível, aumentando a confiabilidade da implementação.

Criando-se a classe **Banco**, adicionar as contas à instância do banco e testar o método **transferir** para ver como a exceção é **propagada**.

Código disponível no **VS Code** sobre esse questionamento da propagação da exceção nas instâncias do **Banco**.

Continuação da Explicação:

Na minha classe **Conta**:

- A exceção é lançada no método **sacar** quando o saldo é insuficiente;
- O método **transferir** captura a exceção e retorna **false** em caso de erro;

Na minha classe **Banco**:

- Possui um método **transferir** que chama o método **transferir** da conta de origem;
- A exceção lançada em **conta.transferir** é capturada dentro de **banco.transferir**, o que permite controlar a falha de operação no contexto do banco;

Na minha aplicação (**app**):

- O método **banco.transferir** é chamado para tentar realizar a transferência. Se a transferência falhar devido a saldo insuficiente, o erro é propagado para a aplicação principal, que pode então tratá-lo adequadamente;

No **Teste**:

- A **conta1** tem saldo de 100.0, e estou tentando transferir 200.0 para a **conta2**. Como o saldo é insuficiente, a exceção será lançada no método **sacar**, propagada para o método **conta.transferir**, e por fim, capturada no método **banco.transferir**.

Propagação da Exceção:

- A exceção é lançada no método **conta.sacar** devido ao saldo insuficiente;
- O método **conta.transferir** captura a exceção e lida com ela;
- O método **banco.transferir** também captura a exceção e trata o erro de forma mais abrangente;

Avaliação da Confiabilidade:

Essa implementação é **bastante confiável** no sentido de que erros são tratados em vários níveis, garantindo que a aplicação continue a funcionar sem falhas inesperadas. O tratamento de exceções oferece uma maneira robusta de garantir que operações críticas, como transferências bancárias, sejam monitoradas, e que falhas, como saldo insuficiente, sejam informadas e tratadas apropriadamente.

A propagação da exceção permite que o erro seja controlado em diferentes partes da aplicação, o que melhora a manutenibilidade e a segurança do sistema.