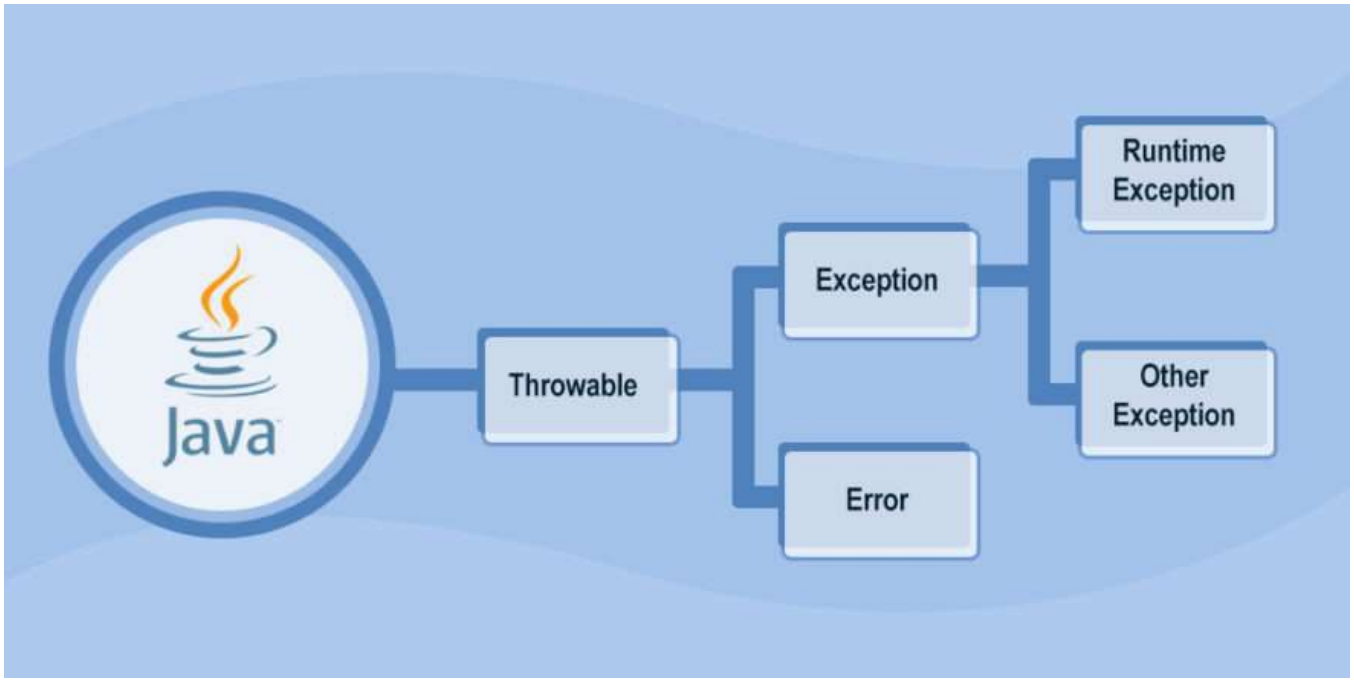
Fluxos Excepcionais

Exception

Antes de tudo devemos compreender que: Exceção é um fluxo inesperado na aplicação e um erro é a interrupção do seu sistema. Isso significa que uma exceção poderá ser tratada através de um desvio de fluxo, enquanto que um erro ocasiona a reinicialização do programa.



> 🔔 **Atenção**
>
> Antes de continuar que tal uma revisada em **Fluxos Excepcionais**

Uma das habilidades cruciais de um programador é a previsibilidade de possíveis exceções que poderão ocorrer em seu sistema por exemplo:

- Tentar usar algum método em um variável sem referência (null)
- Tentar ler um arquivo que não exista
- Tentar transformar `parsear` uma string em número

- Tentar pegar um elemento que não existe no array
- e inúmeras outras possibilidades

Fronteiras

Quando precisamos implementar algum algorítimo propício a gerar uma exceção, é necessário compreender as áreas de bloco de códigos abaixo:

```java
public class Excecoes {
    public static void main(String[] args) {
       try {

           //AQUI ESTÁ TODA A LOGICA OTIMISTA

       }catch (Exception ex){

           //AQUI FICARÁ A ESTRATÉGIA DE CAPTURA E TRATAMENTO DA EXCEÇÃO

       }finally {

           //ESTE BLOCO É DESTINADO PARA AÇÕES QUE DEVERÃO ACONTECER INDEPENDENT
           //SEMPRE SERÁ EXECUTADO, EXCETO EM CASO DE ERRO.
       }
    }
}
```

Níveis de Captura

Já aprendemos que as exceções são classificadas pelas categorias checadas e não checadas, isso já um ponto de análise para definir a sua estratégia de tratamento de exceções. O que precisamos compreender a partir de agora é que existe uma regra para a definição de uma hierarquia para o tratamento destas exceções.

```java
public class Excecoes {
    public static void main(String[] args) {
       try {

       }catch (ExceptionNeto neto){

       }catch (ExceptionFilho filho){

       }catch (ExceptionPai pai){

       }
    }
    // veje o que acontece ao tentar caputar
    // a ExceptionPai antes da ExceptionNeto por exemplo
}
```

📌 **Para fixar**

Como saber a árvore genealogica das Exceptions? Simples uma Exception `extends` outra Exception, exemplo:

```
java.io.FileNotFoundException extends java.io.IOException
```

Throw vs Throws

Imagina que você ficou responsável por implementar uma lógica muito importante na aplicação onde em alguma parte do processo algo de inesperado possa acontecer e você precisar sinalizar à quem for utilizar este recurso. Veja o pseudo-codigo abaixo:

```java
public class EstadoNaoLocalizadoException extends Exception{
    EstadoNaoLocalizadoException(String msg){
        super(msg);
    }
}
```

```java
public class LocalizadoraEstado {
    public static String nomeEstadoBr(String sigla) throws EstadoNaoLocalizadoEx
        if(sigla.equals("PI"))
            return "Piaui";
        else
            throw new EstadoNaoLocalizadoException("Não existe um estado com a s
    }

    public static void main(String[] args) {
        //ops!! alguém acha que aqui poderá ocorrer uma exceção
        //se fosse uma RuntimeException não seria obrigatório o tratamento
        String nomeEstado = nomeEstadoBr("IP");

        try {
            //área para checagem
            nomeEstado = nomeEstadoBr("IP");
        }catch (EstadoNaoLocalizadoException esle){
            esle.printStackTrace();
        }
    }
}
```

🔔 **Propagação**

Em aplicações mais robustas, com um pouco mais de camadas as vezes será necessário que a exceção seja repassada entre as classes envolvidas até chegar ao resultado final para o usuário. Um caso comum é quando a nossa aplicação está estruturada no padrão MVC onde `view` delega para o `controller` e o `controller` para o `respository`. Sendo assim a camada que sinalizar que a exceção será repassada `throws` a mesma não precisará tratá-la `try \ catch`.

Acesse nosso GitHub

Previous page
P O O

Next page
Expressões