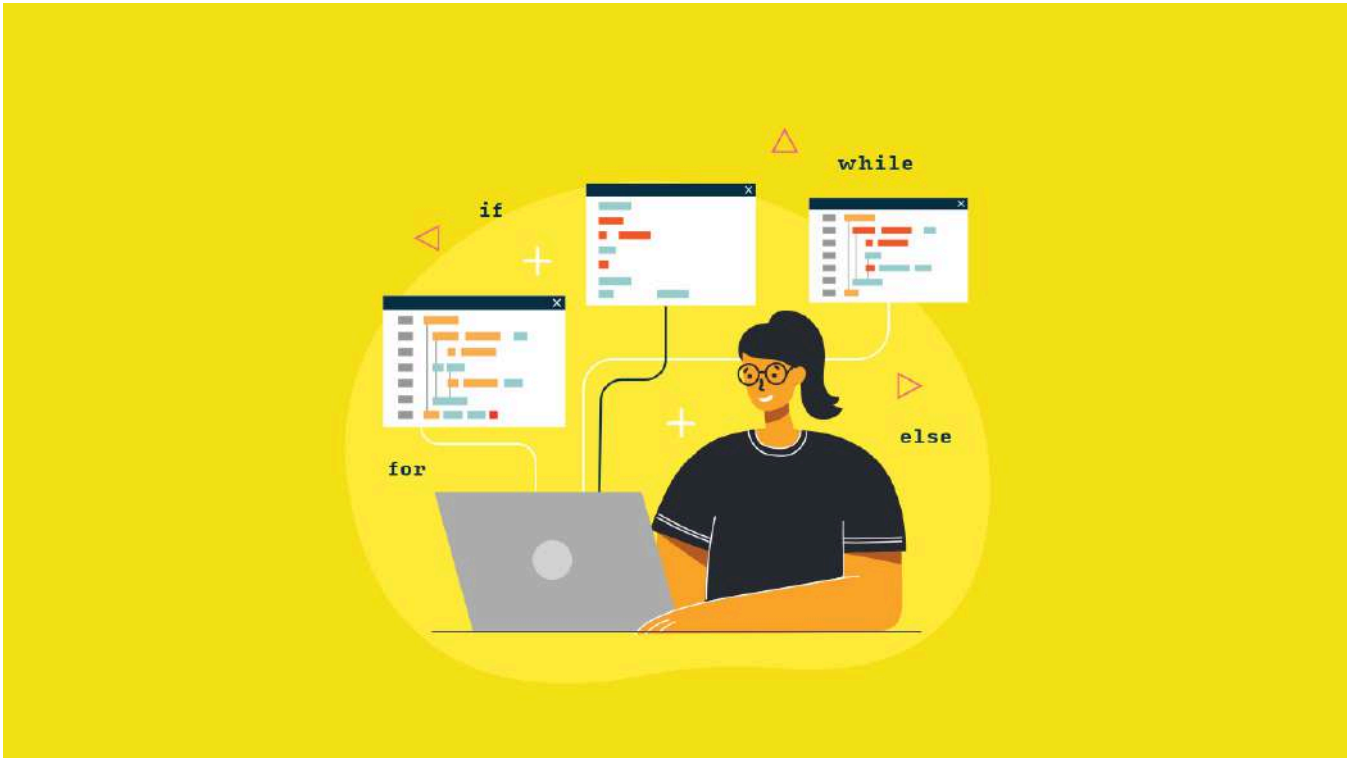
Fluxos

2.2 - Controle de Fluxo

Controlar o fluxo é como ser o diretor do seu próprio filme de tecnologia. Você pode decidir como seu programa vai desenrolar suas tarefas, através de comandos especiais. É como dar as cartas no jogo, escolhendo quando executar as tarefas de forma seletiva, repetitiva ou até mesmo em casos excepcionais.

2.2.1 - Conceitualmente

Os controles de fluxo são operações definidas em todas as linguagens de programação, compreendendo a sua proposta estrutural você se torna capaz de conduzir sua aplicação pelos mais variados caminhos condicionados às regras de negócio obtidas do mundo real.



Classificação

Eles são divididos em três categorias:

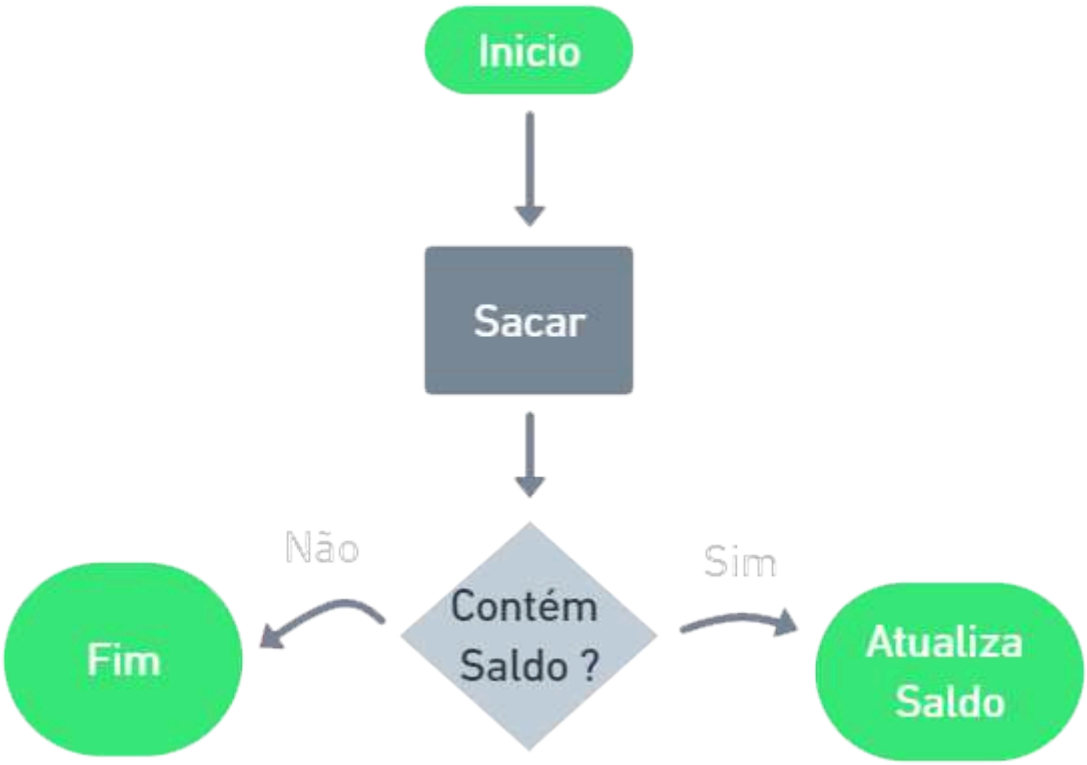
- **Estruturas condicionais:** *if-else, switch-case.*
- **Estruturas de repetição:** *for, while, do-while.*
- **Estruturas excepcionais:** *try-catch-finally, throw.*

2.2.2 - Estruturas condicionais

A Estrutura Condicional, possibilita a escolha de um grupo de ações e comportamentos a serem executadas, quando determinadas condições são ou não satisfeitas. A Estrutura Condicional pode ser Simples ou Composta.

2.2.2.1 Condicionais Simples

Quando ocorre uma validação de execução de fluxo, somente quando a condição for positiva, consideramos como uma estrutura Simples, exemplo:



```java
// CaixaEletronico.java
public class CaixaEletronico {
    public static void main(String[] args) {

        double saldo = 25.0;
        double valorSolicitado = 17.0;

        if(valorSolicitado < saldo)
         saldo = saldo - valorSolicitado;

         System.out.println(saldo);

    }
}
```

2.2.2.2 Condicionais Composta

Algumas vezes, o nosso programa deverá seguir mais de uma jornada de execução, concionando a uma regra de negócio, este cenário é denominado Estrutura Condicional Composta. Vejamos o exemplo abaixo:

java

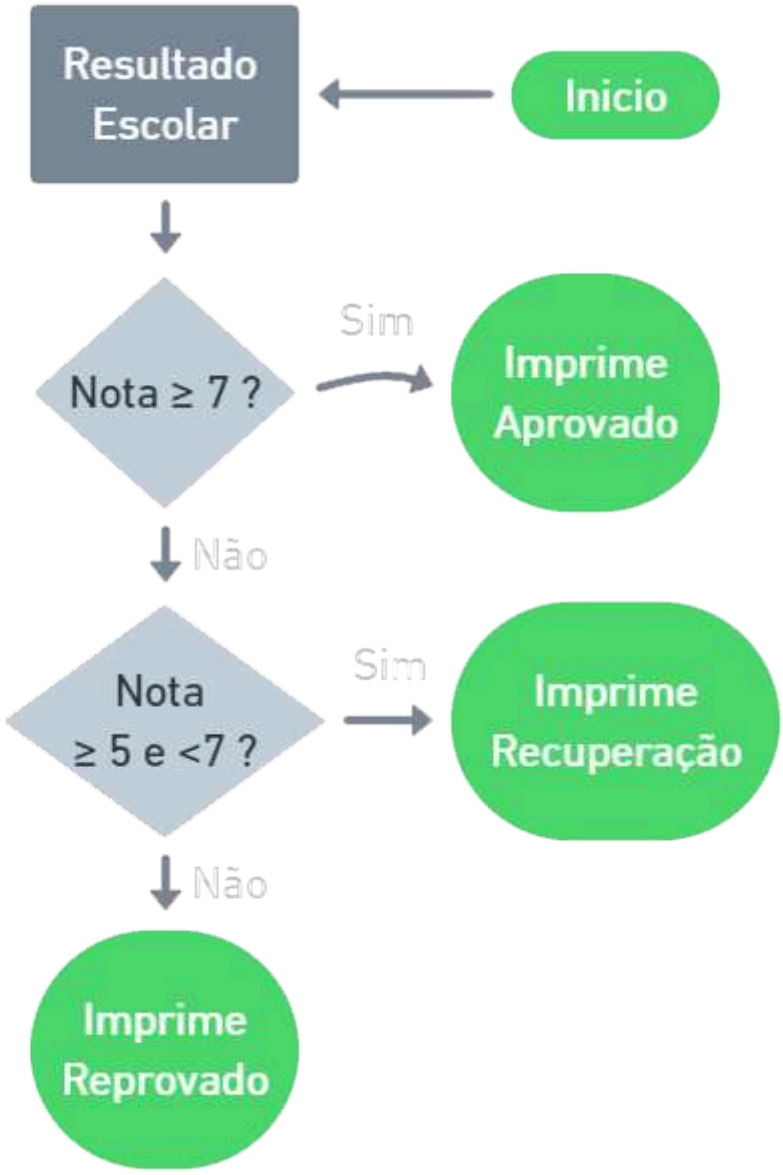
```java
// ResultadoEscolar.java
public class ResultadoEscolar {
    public static void main(String[] args) {

        int nota = 6;

        if(nota >= 7)
         System.out.println("Aprovado");

        else
         System.out.println("Reprovado");
    }
}
```

📌 **Para fixar**

Vale ressaltar aqui, que no Java, em uma condição `if/else` às vezes necessitamos adicionar um bloco de { },se a lógica conter mais de uma linha.

2.2.2.3 Condicionais encadeadas

Em um controle de fluxo condicional, nem sempre nos limitamos ao se (if) e se não (else), podemos ter uma terceira, quarta e ou inúmeras condições.




```java
// ResultadoEscolar.java
public class ResultadoEscolar {
    public static void main(String[] args) {
        int nota = 6;

    if (nota >= 7)
        System.out.println("Aprovado");
    else if (nota >= 5 && nota < 7)
        System.out.println("Recuperação");
    else

```

```
            System.out.println("Reprovado");
        }
    }
```

2.2.2.4 Condição ternária

Como vimos em operadores, podemos abreviar nosso algoritmo condicional, refatorando com o conceito de operador ternário. Vamos refatorar os exemplos acima, para ilustrar o poder deste recurso:

java
```java
// Cenário 1
public class ResultadoEscolar {
    public static void main(String[] args) {
        int nota = 7;
        String resultado = nota >=7 ? "Aprovado" : "Reprovado";
        System.out.println(resultado);
    }
}
```

java
```java
// Cenário 2
public class ResultadoEscolar {
    public static void main(String[] args) {
        int nota = 6;
        String resultado = nota >=7 ? "Aprovado" : nota >=5 && nota <7 ? "Recupe
        System.out.println(resultado);
    }
}
```

Atenção

A condição ternária aparenta representar um fluxo condicional, porém sua principal finalidade é atribuição condicional.

2.2.2.5 Switch Case

A estrutura `switch` , compara o valor de cada caso, com o da variável sequencialmente e sempre que encontra um valor correspondente, executa o código associado ao caso. Para evitar que as comparações continuem a ser executadas, após um caso correspondente ter sido encontrado, acrescentamos o comando `break` no final de cada bloco de códigos. O comando `break` , quando executado, encerra a execução da estrutura onde ele se encontra.

Vamos imaginar que precisamos imprimir uma medida, com base em mapa de valores, exemplo:

Sigla	Medida
P	PEQUENO
M	MÉDIO
G	GRANDE

java

```java
// SistemaMedida.java

// Modo condicional if/else
public class SistemaMedida {
    public static void main(String[] args) {
        String sigla = "M";

        if(sigla == "P")
            System.out.println("PEQUENO");
        else if(sigla == "M")
            System.out.println("MÉDIO");
        else if(sigla == "G")
            System.out.println("GRANDE");
        else
            System.out.println("INDEFINIDO");


    }
}
```

java

```java
// SistemaMedida.java

```

```java
// Modo condicional switch / case
public class SistemaMedida {
    public static void main(String[] args) {
        String sigla = "M";

        switch (sigla) {
        case "P":{
            System.out.println("PEQUENO");
            break;
        }
        case "M":{
            System.out.println("MÉDIO");
            break;
        }
        case "G":{
            System.out.println("GRANDE");
            break;
        }
        default:
            System.out.println("INDEFINIDO");
        }


    }
}
```

> **Cuidado**
>
> Observe que a nível de sintaxe, não tivemos nenhum ganho quanto a redução de códigos e ainda tivemos mais uma preocupação: informar a palavra break em cada alternativa.

Porém, um cenário que poderíamos adequar o uso do switch/case para melhorar nosso algoritmo seria conforme ilustração abaixo:

Imagina que fomos requisitados a criar um sistema de plano telefônico onde:

- O sistema terá 03 planos: BASIC, MÍDIA , TURBO;
- BASIC: 100 minutos de ligação;
- MÍDIA: 100 minutos de ligação + WhatsApp e Instagram grátis;
- TURBO: 100 minutos de ligação + WhatsApp e Instagram grátis + 5 GB YouTube.

java

```java
// Modo condicional convencional
public class PlanoOperadora {
    public static void main(String[] args) {
        String plano = "M"; //M / T

        if(plano == "B") {
            System.out.println("100 minutos de ligação");
        }else if(plano == "M") {
            System.out.println("100 minutos de ligação");
            System.out.println("WhatsApp e Instagram grátis");
        }else if(plano == "T") {
            System.out.println("100 minutos de ligação");
            System.out.println("WhatsApp e Instagram grátis");
            System.out.println("5Gb Youtube");
        }


    }
}
```

java

```java
// Modo condicional switch/case
public class PlanoOperadora {
    public static void main(String[] args) {
        String plano = "M"; // M / T

        switch (plano) {
            case "T": {
                System.out.println("5Gb Youtube");
            }
            case "M": {
                System.out.println("WhatsApp e Instagram grátis");
            }
            case "B": {
                System.out.println("100 minutos de ligação");
            }
        }
    }
}
```

Cuidado

Se optarem por usar switch / case, estude um pouco mais, sobre os conceitos de continue, break e default.

2.2.3 - Estruturas de repetição

> Laços de repetição, também conhecidos como laços de iteração ou simplesmente loops, são comandos que permitem iteração de código, ou seja, que comandos presentes no bloco sejam repetidos diversas vezes.
> https://diegomariano.com/lacos-de-repeticao-2/

Laços ou repetições são representados pelas seguintes estruturas:

- For (para);
- While (enquanto);
- Do While (faça enquanto).

2.2.3.1 For

O comando for (na tradução literal para a língua portuguesa “para”) permite que uma variável contadora, seja testada e incrementada a cada iteração, sendo essas informações definidas na chamada do comando. O comando for recebe como entrada uma variável contadora, a condição para continuar a execução e o valor de incrementação.

A estrutura de sintaxe do controle de repetição for é exibida abaixo:

```java
//estrutura do controle de fluxo for

for (bloco de inicialização; expressão booleana de validação; bloco de atualizaç
{
     // comando que será executado até que a
     // expressão de validação torne-se falsa
}
```

Vamos imaginar que, Joãozinho precisa contar até 20 carneirinhos para pegar no sono:

```java
// ExemploFor.java
public class ExemploFor {
    public static void main(String[] args) {
        for(int carneirinhos = 1 ; carneirinhos <=20; carneirinhos ++) {
            System.out.println(carneirinhos + " - Carneirinho(s)");
        }
    }
}
```

Vamos explicar a estrutura do código acima:

For position

1. `int carneirinhos = 1;` → O programa entende que a variável carneirinhos, começa com o valor igual a 1 e isso acontece somente uma vez;
2. `Carneirinhos < = 20;` → O programa verifica se a variável carneirinhos, ainda é menor que 20;
3. O programa começa a executar o algoritmo, no nosso caso, imprimir a quantidade de carneirinhos em contagem;
4. `Carneirinhos ++` → O programa aumenta em mais 1, o valor da variável carneirinhos;
5. O fluxo é finalizado, quando a variável carneirinhos for igual a 20.

```java
// Outras estruturas


```

```java
//estrutura 1
for(int carneirinhos = 1 ; carneirinhos <=20; carneirinhos ++) {
     System.out.println(carneirinhos + " - Carneirinho(s)");
}

//estrutura 2
//o que importa é somente o bloco condicional
int carneirinhos = 1;
for( ; carneirinhos <=20; ) {
     System.out.println(carneirinhos + " - Carneirinho(s)");
     carneirinhos ++;
}

//for(somente 1x; uma expresão boolean; acontecerá a cada final da execução){}
```

Também usamos o controle de fluxo for, para interagir sobre arrays e coleções:

java

```java
// ExemploFor.java
public class ExemploFor {
public static void main(String[] args) {
    String alunos[] = { "FELIPE", "JONAS", "JULIA", "MARCOS" };

    for (int x=0; x<alunos.length; x++) {
        System.out.println("O aluno no indice x=" + x + " é " + alunos[x]);
    }
}
}
```

> Observe que, como os arrays começam com índice igual a 0 (zero), iniciamos a nossa variável x também com valor zero e validamos a quantidade de repetições, a partir da quantidade de elementos do array.

Fala a verdade: Depois desta explicação deu até sono hein!? 😴 😴

2.2.3.2 For each

O uso do **for / each** está fortemente relacionado, com um cenário onde contenha um array ou coleção, e assim, a interação é baseada nos elementos da coleção.

```java
// ExemploFor.java
public class ExemploFor {
public static void main(String[] args) {
    String alunos [] =  {"FELIPE","JONAS","JULIA","MARCOS"};

        //Forma abreviada
    for(String aluno : alunos) {
      System.out.println(aluno);
    }

}
}
```

1. `String aluno : alunos` → De forma abreviada, é criada uma variável nome obtendo um elemento do vetor a cada ocorrência;
2. A impressão de cada aluno é realizada.

Break e Continue

Break significa quebrar, parar, frear, interromper. E é isso que se faz, quando o Java encontra esse comando pela frente. **Continue**, como o nome diz, ele 'continua' o laço. O comando `break` interrompe o laço, já o `continue` interrompe somente a iteração atual.

```java
// class ExemploBreakContinue.java
public class ExemploBreakContinue {
    public static void main(String[] args) {

    for(int numero = 1; numero <=5; numero ++){
        if(numero==3)
            break;

        System.out.println(numero);

    }
    //Qual a saída no console ?

    }
}
```

```java
// class ExemploBreakContinue.java
public class ExemploBreakContinue {
```

```
    public static void main(String[] args) {

    for(int numero = 1; numero <=5; numero ++){
        if(numero==3)
            continue;

        System.out.println(numero);

    }
    //Qual a saída no console ?

    }
}
```

✔ Conclusão

Observem que sempre o `break` e `continue` , está condicionado a um critério de negócio.

2.2.3.3 While

O laço **while** (na tradução literal para a língua portuguesa “enquanto”) determina que, enquanto uma condição for válida, o bloco de código será executado. O laço **while**, testa a condição antes de executar o código, logo, caso a condição seja inválida no primeiro teste o bloco nem será executado.

A estrutura de sintaxe, do controle de repetição `while` é exibida abaixo:

java

```
//estrutura do controle de fluxo while

while (expressão booleana de validação)
{
     // comando que será executado até que a
     // expressão de validação torne-se falsa
}
```

Anya recebeu R$ 50,00 de mesada e foi em uma loja de doces gastar todo o seu dinheiro, enquanto o valor dos doces não igualou a R$ 50,00 ele foi adicionando itens no carrinho.

java

```java
// ExemploWhile.java
import java.util.concurrent.ThreadLocalRandom;

public class ExemploWhile {
    public static void main(String[] args) {
        double mesada = 50.0;

        while(mesada > 0) {
            Double valorDoce = valorAleatorio();
            if(valorDoce > mesada)
                valorDoce = mesada;

            System.out.println("Doce do valor: " + valorDoce + " Adicionado no c
            mesada = mesada - valorDoce;
        }

        System.out.println("Mesada:" + mesada);
        System.out.println("Anya gastou toda a sua mesada em doces");

        /*
        * Não se preocupe quanto a formatação de valores
        * Iremos explorar os recursos de formatação em breve !!

```

```
        */
    }
    private static double valorAleatorio() {
        return ThreadLocalRandom.current().nextDouble(2, 8);
    }
}
```

2.2.3.3 Do while

O laço **do / while** (na tradução literal para a língua portuguesa “faça... enquanto”), assim como o laço while, considera que, enquanto uma determinada condição for válida, o bloco de código será executado. Entretanto, **do / while** testa a condição após executar o código, sendo assim, mesmo que a condição seja considerada inválida, no primeiro teste o bloco será executado pelo menos uma vez.

A estrutura de sintaxe do controle de repetição `do / while` é exibida abaixo:

java

```
//estrutura do controle de fluxo do while

do
{
    // comando que será executado até que a
     // expressão de validação torne-se falsa
}
while (expressão booleana de validação);
```

Joãozinho resolveu ligar para o seu amigo, dizendo que hoje se entupiu de comer docinhos:

```java
// ExemploDoWhile.java
import java.util.Random;

public class ExemploDoWhile {
    public static void main(String[] args) {
        System.out.println("Discando...");

        do {
            //excutando repetidas vezes até alguém atender
            System.out.println("Telefone tocando");

        }while(tocando());

        System.out.println("Alô !!!");
    }
    private static boolean tocando() {
        boolean atendeu = new Random().nextInt(3)==1;
        System.out.println("Atendeu? " + atendeu);

```

```
            //negando o ato de continuar tocando
            return ! atendeu;
        }
    }
```

2.2.4 - Estruturas excepcionais

Exceções

Ao executar o código Java, diferentes erros podem acontecer: erros de codificação feitos pelo programador, erros devido a entrada errada ou outros imprevistos.

Quando ocorre um erro, o Java normalmente para e gera uma mensagem de erro. O termo técnico para isso é: Java lançará uma **exceção** (jogará um erro).

De forma interpretativa em Java, um erro é algo irreparável, a aplicação trava ou é encerrada drasticamente. Já exceções é um fluxo inesperado da nossa aplicação, exemplo: Querer dividir um valor por zero, querer abrir um arquivo que não existe, abrir uma conexão de banco, com usuário ou senha inválida. Todos estes cenários e os demais, não são erros mas sim fluxos, não previstos pela aplicação.

É ai que entra mais uma responsabilidade do desenvolvedor, prever situações iguais a estas e realizar o que denominamos de *Tratamento de Exceções*.

2.2.4.2 Cenários conhecidos

java

```java
import java.util.Locale;
import java.util.Scanner;

public class AboutMe {
    public static void main(String[] args) {
        //criando o objeto scanner
        Scanner scanner = new Scanner(System.in).useLocale(Locale.US);

        System.out.println("Digite seu nome");
        String nome = scanner.next();


```

```
13          System.out.println("Digite seu sobrenome");
14          String sobrenome = scanner.next();
15
16          System.out.println("Digite sua idade");
17          int idade = scanner.nextInt();
18
19          System.out.println("Digite sua altura");
20          double altura = scanner.nextDouble();
21
22
23          //imprimindo os dados obtidos pelo usuario
24          System.out.println("Olá, me chamo " + nome.toUpperCase() + " " + sobreno
25          System.out.println("Tenho " + idade + " anos ");
26          System.out.println("Minha altura é " + altura + "cm ");
27          scanner.close();
28      }
    }
```

Aparentemente é um programa simples, mas vamos listar algumas possíveis exceções que podem acontecer.

- Não informar o nome e sobrenome;
- O valor da idade ter um caractere NÃO numérico;
- O valor da altura ter uma vírgula ao invés de ser um ponto (conforme padrão americano);

Executando o nosso programa com os valores abaixo, vamos entender qual exceção acontecerá:

Entrada	Exceção	Causa
Marcelo		
Azevedo		
quinze (15)	java.util.InputMismatchException	O programa esperava o valor do tipo numérico inteiro.
1,65	java.util.InputMismatchException	java.util.InputMismatchException

> ℹ️ **Informação**
>
> A melhor forma de prever, que pode ocorrer uma exceção, é pensar que ela pode ocorrer. **Lei de Murphy**

2.2.4.3 Exceções mapeadas

A linguagem Java, dispõe de uma vasta lista de classes que representam exceções, abaixo iremos apresentar as mais comuns:

Nome	Causa
java.lang.NullPointerException	Quando tentamos obter alguma informação de uma variável nula.
java.lang.ArithmeticException	Quando tentamos dividir um valor por zero.
java.sql.SQLException	Quando existe algum erro, relacionado a interação com banco de dados.
java.io.FileNotFoundException	Quando tentamos ler ou escrever, em um arquivo que não existe.

2.2.4.4 Tratando exceções

E quando inevitavelmente, ocorrer uma exceção? Como nós desenvolvedores podemos ajustar o nosso algoritmo para amenizar o ocorrido?

A instrução `try`, permite que você defina um bloco de código, para ser testado quanto a erros enquanto está sendo executado.

A instrução `catch`, permite definir um bloco de código a ser executado, caso ocorra um erro no bloco try.

A instrução `finally`, permite definir um bloco de código a ser executado, independente de ocorrer um erro ou não. As palavras-chave try e catch vem em pares:

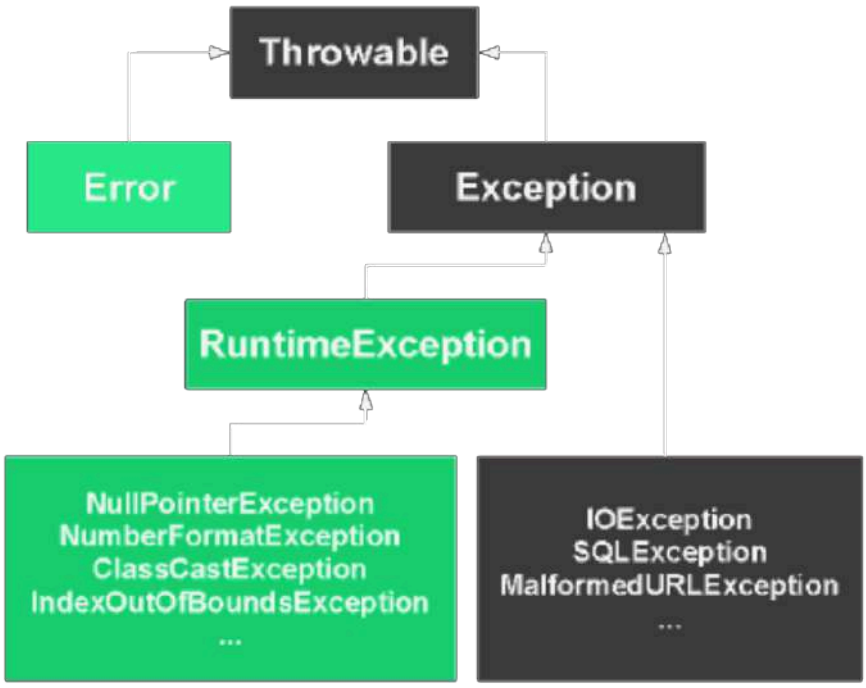
Estrutura de um bloco com try e catch:

```java
try {
  //  bloco de código conforme esperado
}
catch(Exception e) {// precisamos saber qual exceção
  // bloco de código que captura as exceções que podem acontecer
  // em caso de um fluxo não previsto
}
```

> 🔔 **Atenção**
>
> O bloco `try / catch` pode conter um conjunto de **catchs**, correspondentes a cada exceção **prevista** em uma funcionalidade do programa.

2.2.4.5 Hierarquia das exceções

A linguagem Java, dispõe de uma variedade de classes, que representam exceções e estas classes, são organizadas em uma hierarquia denominadas **Checked and Unchecked Exceptions** ou Exceções Checadas e Não Checadas.



> ℹ️ **Informação**
>
> O que determina uma exceção ser classificada como checada ou não checada ?

É o risco dela ser disparada, logo, você precisa tratá-la, exemplo:

Vamos imaginar que precisamos realizar de duas maneiras, a conversão de uma String para um número, porém o texto contém Alfanuméricos.

```java
public class ExemploExcecao {
    public static void main(String[] args) {
        Number valor = Double.valueOf("a1.75");

        valor = NumberFormat.getInstance().parse("a1.75");

        System.out.println(valor);

    }
}
```

Informação

As linhas 3 e 5, apresentarão uma exceção ao serem executadas, e a linha 5 contém um método que pode disparar uma exceção checada, logo, nós programadores iremos usar este método, teremos que tratá-la explicitamente com `try \ catch` .

2.2.4.6 Exceções customizadas

Nós podemos criar nossas próprias exceções, baseadas em regras de negócio e assim melhor direcionar quem for utilizar os recursos desenvolvidos no projeto, exemplo:

- Imagina que como regra de negócio, para formatar um cep, necessita sempre de ter 8 dígitos, caso contrário, lançará uma exceção que denominamos de **CepInvalidoException**.
- Primeiro criamos nossa exceção:

```java
public class CepInvalidoException extends Exception {}
```

Em seguida, criamos nosso método de formatação de cep:

```java
static String formatarCep(String cep) throws CepInvalidoException{
        if(cep.length() != 8)
```

```
            throw new CepInvalidoException();

            //simulando um cep formatado
            return "23.765-064";
        }
```

2.2.4.7 Hora da verdade

Vamos explorar alguns outros cenários, com fluxo condicionais, repetições e excepcionais.

Case 1: Vamos imaginar que em um processo seletivo, existe o valor base salarial de R$ 2.000,00 e o salário pretentido pelo candidato. Vamos elaborar um controle de fluxo onde:

- Se o valor salário base for maior que valor salário pretentido, imprima : **LIGAR PARA O CANDIDATO**;
- Se não se o valor salário base for igual ao valor salário pretentido, imprima : **LIGAR PARA O CANDIDATO, COM CONTRA PROPOSTA**;
- Se não imprima: **AGUARDANDO RESULTADO DOS DEMAIS CANDIDATOS.**

Case 2: Foi solicitado, que nosso sistema garanta que, diante das inúmeras candidaturas sejam selecionados apenas no máximo, 5 candidatos para entrevista, onde o salário pretendido seja menor ou igual ao salário base.

java
```
// Array com a lista de candidatos

String [] candidatos = {"FELIPE","MÁRCIA","JULIA","PAULO","AUGUSTO","MÔNICA","FA
```

java
```
// Método que simula o valor pretendido

import java.util.concurrent.ThreadLocalRandom;
static double valorPretendido() {
     return ThreadLocalRandom.current().nextDouble(1800, 2200);
}
```

Case 3: Agora é hora de imprimir a lista dos candidatos selecionados, para disponibilizar para o RH entrar em contato.

Case 4: O RH deverá realizar uma ligação, com no máximo 03 tentativas para cada candidato selecionado e caso o candidato atenda, deve-se imprimir:

- **"CONSEGUIMOS CONTATO COM _ `[CANDIDATO]` ``` *APÓS* **_** *[TENTATIVA]*`** TENTATIVA(S)" ;**
- Do contrário imprima: **"NÃO CONSEGUIMOS CONTATO COM O _ `[CANDIDATO]` _".**

Case 1 Case 2 Case 3 Case 4

java

```java
public class ProcessoSeletivo {
    public static void main(String[] args) {
        //salario base maior que salario pretendido
        case1(2000.0, 1900.0);

        //salario base igual que salario pretendido
        case1(2000.0, 2000.0);

        //salario base igual que salario pretendido
        case1(1900.0, 2000.0);V
    }
    static void case1(double salarioBase, double salarioPretendido) {
        // ... DIGITE SUA SOLUCAÇÃO AQUI ...
    }
}
```