

# Paradigmas de Programação

# Aula 03 – Como vai ser ???

- Paradigma Funcional – Entregável Aula02
  - Métodos e Chaining Method
  - Operações em streams: map, filter, forEach e sorted
- Paradigma Lógico
- Programação Orientada a Aspecto
- Entregável – Aula 03
  - Vamos relembrar um pouco do que foi apresentado.



# Entregável Aula 02

```
import java.util.List;

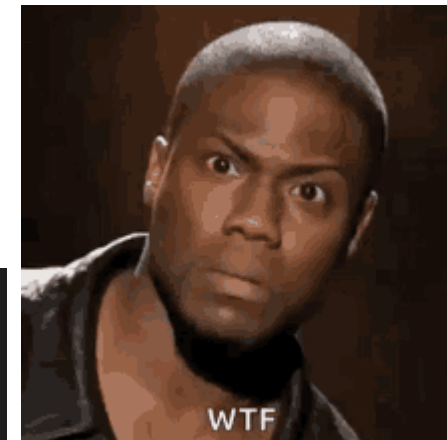
public class Entregavel {
    public static void main(String[] args) {
        List<Integer> numbers = List.of(24,3,6,5,7,89,10);
        System.out.println(numbers);

        List<Integer> pares = numbers.stream().filter(n -> n % 2 == 0).toList();
        System.out.println(pares);

        List<Integer> dobro = numbers.stream().map(n -> n *2).toList();
        System.out.println(dobro);

        List<Integer> ordem = numbers.stream().sorted().toList();
        System.out.println(ordem);

        numbers.forEach(n -> System.out.println(n));
    }
}
```

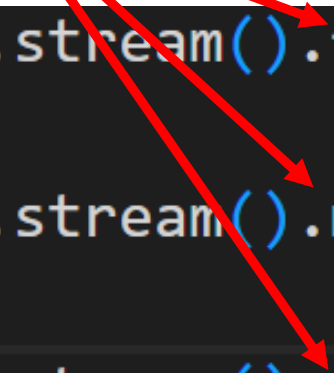


```
List<Integer> pares = numbers.stream().filter(n -> n % 2 == 0).toList();  
  
List<Integer> dobro = numbers.stream().map(n -> n *2).toList();  
  
List<Integer> ordem = numbers.stream().sorted().toList();
```

Aqui temos Funções de Ordem Superior  
(métodos), que recebem como  
parâmetros outras funções ou  
Expressões Lambdas  
map(), filter() e sorted()

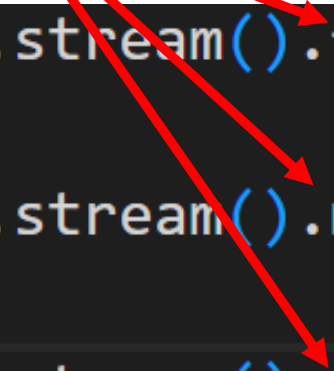
```
List<Integer> pares = numbers.stream().filter(n -> n % 2 == 0).toList();  
  
List<Integer> dobro = numbers.stream().map(n -> n *2).toList();  
  
List<Integer> ordem = numbers.stream().sorted().toList();
```

Aqui temos Funções de Ordem Superior  
(métodos), que recebem como  
parâmetros outras funções ou  
Expressões Lambdas  
map(), filter() e sorted()



```
List<Integer> pares = numbers.stream().filter(n -> n % 2 == 0).toList();  
  
List<Integer> dobro = numbers.stream().map(n -> n * 2).toList();  
  
List<Integer> ordem = numbers.stream().sorted().toList();
```

Aqui temos Funções de Ordem Superior  
(métodos), que recebem como  
parâmetros outras funções ou  
Expressões Lambdas  
map(), filter() e sorted()



```
List<Integer> pares = numbers.stream().filter(n -> n % 2 == 0).toList();  
  
List<Integer> dobro = numbers.stream().map(n -> n * 2).toList();  
  
List<Integer> ordem = numbers.stream().sorted().toList();
```

```
List<Integer> pares = numbers.stream().filter(n -> n % 2 == 0).toList();  
  
List<Integer> dobro = numbers.stream().map(n -> n *2).toList();  
  
List<Integer> ordem = numbers.stream().sorted().toList();
```

Mas agora as perguntas ...

**Para que servem esses outros métodos stream(), e toList() ???**

**E porque esta nessa ordem ???**

**Quem está invocando quem ???**

**O método stream() invoca o sorted() ???**



Para entendermos melhor é preciso  
relembrarmos ...

O que é um método???

# Para entendermos melhor é preciso relembrarmos ...

## O que é um método???

- É onde toda a mágica funciona!
- Onde dizemos como o fazer (**imperativa**) ou o que fazer (**declarativa**)



# Para entendermos melhor é preciso relembrarmos ...

## O que é um método???

- É onde toda a mágica funciona!
- Onde dizemos como o fazer (**imperativa**) ou o que fazer (**declarativa**)

## Como identifico um método no código???

```
numbers.stream().filter(n -> n % 2 == 0).toList();
```

# Para entendermos melhor é preciso relembrarmos ...

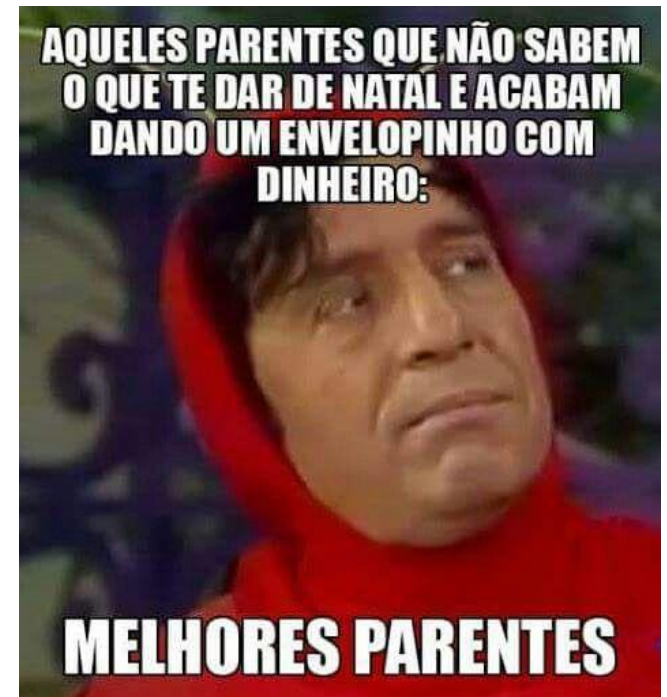
## O que é um método???

- É onde toda a mágica funciona!
- Onde dizemos como o fazer (**imperativa**) ou o que fazer (**declarativa**)

## Como identifico um método no código???

```
numbers.stream().filter(n -> n % 2 == 0).toList();
```

- Eles sempre vem acompanhados pelos **Parênteses ()**



# Para entendermos melhor é preciso relembrarmos ...

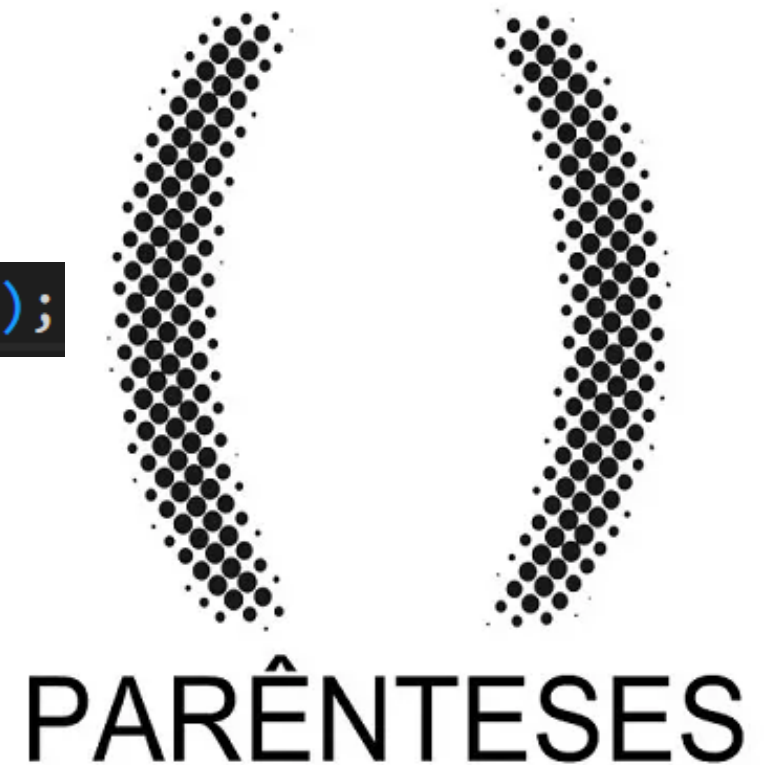
## O que é um método???

- É onde toda a mágica funciona!
- Onde dizemos como o fazer (**imperativa**) ou o que fazer (**declarativa**)

## Como identifico um método no código???

```
numbers.stream().filter(n -> n % 2 == 0).toList();
```

- Eles sempre vem acompanhados pelos **Parênteses ()**



PARÊNTESES

# Para entendermos melhor é preciso lembrarmos ...

O que é um método???

- É onde toda a mágica funciona!
- Onde dizemos como o fazer (**imperativa**) ou o que fazer (**declarativa**)

Como identifico um método no código???

```
numbers.stream().filter(n -> n % 2 == 0).toList();
```

- Eles sempre vem acompanhados pelos **Parênteses ()**

Como eu posso invocar (chamar) esses métodos???

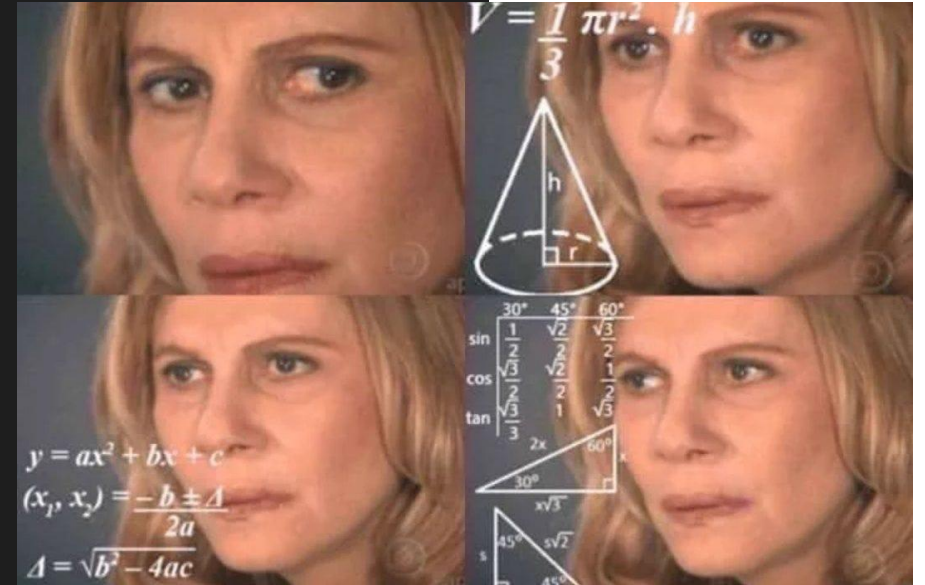
.... Depende.....



# Vamos ver dois exemplos

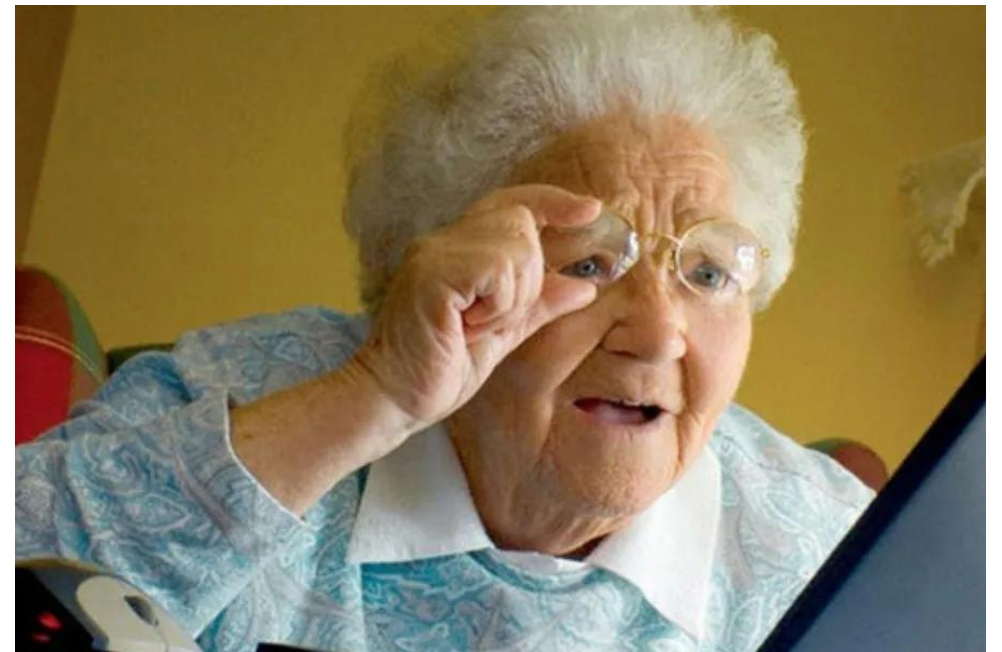
## Qual a diferença???

```
public class Person {  
    private String name;  
    private Integer age;  
  
    public static Person build() {  
        return new Person();  
    }  
  
    public String toString() {  
        return this.name + " - Age: " + this.age;  
    }  
}
```



# Isso aí!!!!

- Um é um método “Static”, ou seja, **pertence a Classe!**
- O outro é um **método de Instância!**
- Mas o que quer dizer isso???





# Método de Classe

- Pode ser invocado sem a necessidade de um objeto instanciado a partir da Classe

```
public static void main(String[] args) {  
    //Invocando um método Estático  
    // Este método retorna uma nova instância da classe Person (novo objeto)  
    Person pessoa = Person.build();  
}
```

# Método de Instância

- Fica totalmente relacionado com a instância da classe, ou seja, o objeto.
- Dessa forma só pode ser invocado a partir de um objeto instanciado.

```
public static void main(String[] args) {  
    //Invocando um método Estático  
    // Este método retorna uma nova instância da classe Person (novo objeto)  
    Person pessoa = Person.build();  
    //Invocando um método de Instância  
    // Deve ser usado uma variável que referencia um objeto na memória  
    pessoa.toString();  
}
```

# Certo. Mas ...

- Para que servem esses outros métodos `stream()`, e `toList()` ???
  - E porque esta nessa ordem ???
  - Quem está invocando quem ???
  - O método `stream()` invoca o `sorted()` ???

```
List<Integer> pares = numbers.stream().filter(n -> n % 2 == 0).toList();  
  
List<Integer> dobro = numbers.stream().map(n -> n *2).toList();  
  
List<Integer> ordem = numbers.stream().sorted().toList();
```

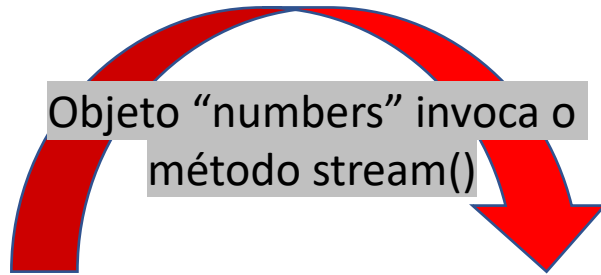
# Method Chaining (Encadeamento de método)

- O **method chaining** é um conceito em programação onde você encadeia várias chamadas de método em um único comando.

```
numbers.stream().sorted().toList();
```

# Method Chaining (Encadeamento de método)

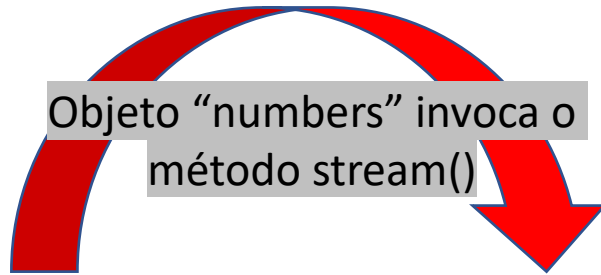
- O **method chaining** é um conceito em programação onde você encadeia várias chamadas de método em um único comando.



```
numbers.stream().sorted().toList();
```

# Method Chaining (Encadeamento de método)

- O **method chaining** é um conceito em programação onde você encadeia várias chamadas de método em um único comando.

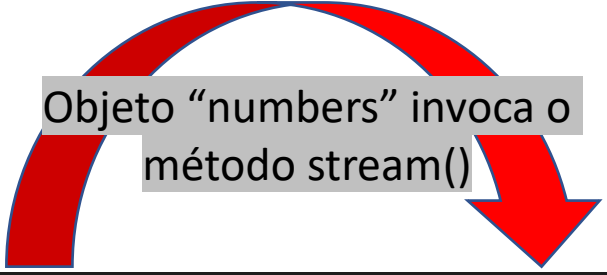


- **Quem invoca o método sorted() ???**

```
numbers.stream().sorted().toList();
```

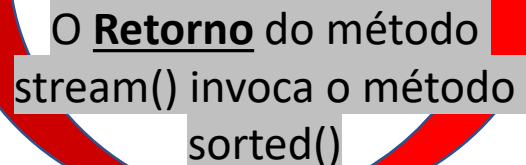
# Method Chaining (Encadeamento de método)

- O **method chaining** é um conceito em programação onde você encadeia várias chamadas de método em um único comando.



Objeto “numbers” invoca o método `stream()`

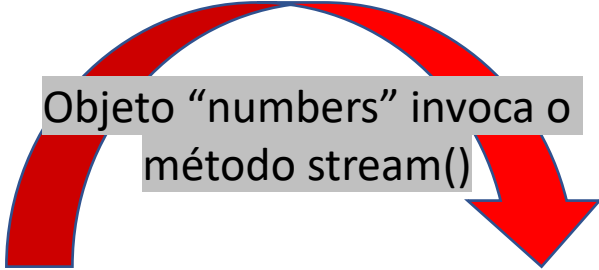
```
numbers.stream().sorted().toList();
```



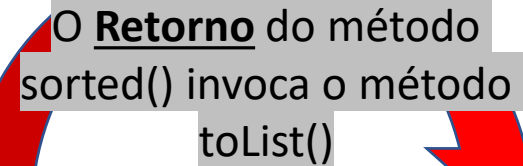
O Retorno do método `stream()` invoca o método `sorted()`

# Method Chaining (Encadeamento de método)

- O **method chaining** é um conceito em programação onde você encadeia várias chamadas de método em um único comando.

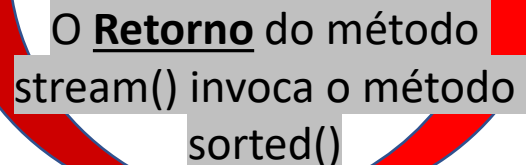


Objeto "numbers" invoca o método stream()



O Retorno do método sorted() invoca o método toList()

```
numbers.stream().sorted().toList();
```



O Retorno do método stream() invoca o método sorted()



# Method Chaining (Encadeamento de método)

- O **method chaining** é um conceito em programação onde você encadeia várias chamadas de método em um único comando.
- Vamos imaginar a seguinte Classe **Person**:

Metho

- O metho encadei
- Vamos i

```
1 public class Person {
2     private String name;
3     private Integer age;
4
5     public static Person build() {
6         return new Person();
7     }
8     public String toString() {
9         return this.name + " - Age: " + this.age;
10    }
11    public String getName() {
12        return name;
13    }
14    public void setName(String name) {
15        this.name = name;
16    }
17    public Integer getAge() {
18        return age;
19    }
20    public void setAge(Integer age) {
21        this.age = age;
22    }
23 }
```

e método)

nde você  
mando.

# Metho

- O metho encadei
- Vamos i

```
1 public class Person {
2     private String name;
3     private Integer age;
4
5     public static Person build() {
6         return new Person();
7     }
8     public String toString() {
9         return this.name + " - Age: " + this.age;
10    }
11    public String getName() {
12        return name;
13    }
14    public void setName(String name) {
15        this.name = name;
16    }
17    public Integer getAge() {
18        return age;
19    }
20    public void setAge(Integer age) {
21        this.age = age;
22    }
23 }
```

Possui um método Estático  
que retorna um novo objeto  
Person

nde você  
mando.

Possui um método de Instância que retorna um uma String

- O **meth**
- encadei
- Vamos i

```
1 public class Person {  
    private String name;  
    private Integer age;  
  
    public static Person build() {  
        return new Person();  
    }  
    public String toString() {  
        return this.name + " - Age: " + this.age;  
    }  
    public String getName() {  
        return name;  
    }  
    public void setName(String name) {  
        this.name = name;  
    }  
    public Integer getAge() {  
        return age;  
    }  
    public void setAge(Integer age) {  
        this.age = age;  
    }  
}
```

Possui um método Estático que retorna um novo objeto Person

nde você  
mando.

Possui um método de Instância que retorna um uma String

- O **meth**
- encadei
- Vamos i

```
1 public class Person {  
    private String name;  
    private Integer age;  
  
    public static Person build() {  
        return new Person();  
    }  
    public String toString() {  
        return this.name + " - Age: " + this.age;  
    }  
    public String getName() {  
        return name;  
    }  
    public void setName(String name) {  
        this.name = name;  
    }  
    public Integer getAge() {  
        return age;  
    }  
    public void setAge(Integer age) {  
        this.age = age;  
    }  
}
```

Possui um método Estático que retorna um novo objeto Person

nde você  
mando.

Possui os métodos Gets que retornam seus dados, e Sets que NÃO retornam nada.

Agora você quer:

- Criar um novo Objeto;
- Setar o “Name” e “Age”;
- Imprimir na tela o Objeto;

Como você faz???

```
1  public class Person {
2      private String name;
3      private Integer age;
4
5      public static Person build() {
6          return new Person();
7      }
8      public String toString() {
9          return this.name + " - Age: " + this.age;
10     }
11     public String getName() {
12         return name;
13     }
14     public void setName(String name) {
15         this.name = name;
16     }
17     public Integer getAge() {
18         return age;
19     }
20     public void setAge(Integer age) {
21         this.age = age;
22     }
23 }
```

```
//Criação (Instanciação) de um objeto do tipo "Person"
Person pessoa01 = new Person(); // Através do "new"
pessoa01 = Person.build(); // Ou Através do método estático para build
//Invocar métodos para alterar atributos (uma linha para cada atributo)
pessoa01.setName("Silvio Santos");
pessoa01.setAge(93);
//Imprime na tela os dados desse objeto:
System.out.println(pessoa01.toString());
```



```
//Criação (Instanciação) de um objeto do tipo "Person"
Person pessoa01 = new Person(); // Através do "new"
pessoa01 = Person.build(); // Ou Através do método estático para build
//Invocar métodos para alterar atributos (uma linha para cada atributo)
pessoa01.setName("Silvio Santos");
pessoa01.setAge(93);
//Imprime na tela os dados desse objeto:
System.out.println(pessoa01.toString());
```



Mas eu quero fazer de uma forma mais “Elegante”!!!!

Como fazer???



```
1  public class Person {
2      private String name;
3      private Integer age;
4
5      public static Person build() {
6          return new Person();
7      }
8      public String toString() {
9          return this.name + " - Age: " + this.age;
10     }
11     public String getName() {
12         return name;
13     }
14     public void setName(String name) {
15         this.name = name;
16     }
17     public Integer getAge() {
18         return age;
19     }
20     public void setAge(Integer age) {
21         this.age = age;
22     }
23 }
```

# Já sei!!!!

- Vamos usar o encadeamento de métodos (Method Chaining)

```
Person pessoa01 = Person.build().setName("Silvio Santos").setAge(93);  
System.out.println(pessoa01.toString());
```

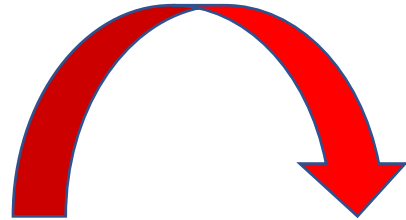


```
//Tentativa de usar o Method Chaining em uma Classe não preparada  
// Pois o método "setName()" NÃO retorna nada (VOID)  
// sendo assim "nada (void)" não pode invocar nenhum método  
Person pessoa02 = Person.build().setName("Senor Abravanel").setAge(93);  
System.out.println(pessoa02.toString());
```



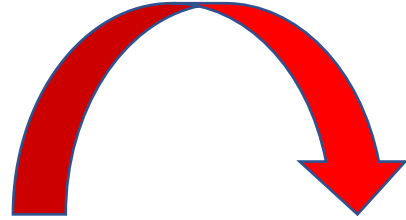
```
Person pessoa02 = Person.build().setName("Senor Abravanel").setAge(93);
```

A partir da Classe  
Person, é invocado o  
método estático build()

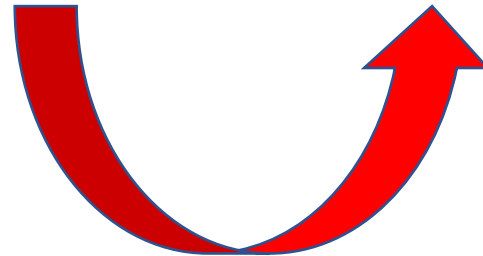


```
Person pessoa02 = Person.build().setName("Senor Abravanel").setAge(93);
```

A partir da Classe  
Person, é invocado o  
método estático build()



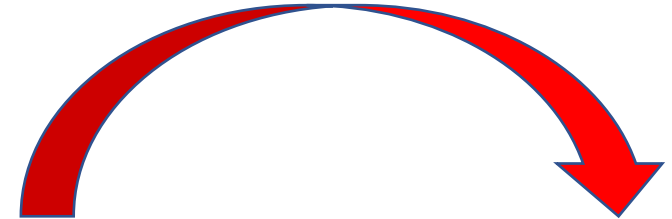
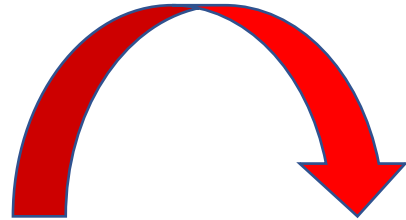
```
Person pessoa02 = Person.build().setName("Senor Abravanel").setAge(93);
```



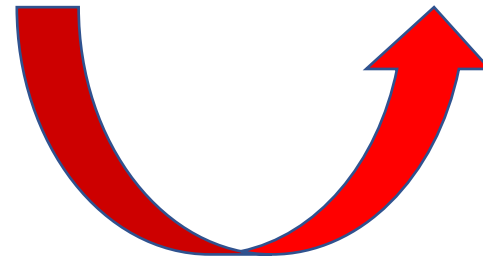
O retorno do método build(),  
**que é um objeto "Person"**,  
invoca o método de instância  
setName()

A partir da Classe  
Person, é invocado o  
método estático build()

O retorno do método setName(),  
que é **VOID**, invoca o método  
setAge()



```
Person pessoa02 = Person.build().setName("Senor Abravanel").setAge(93);
```



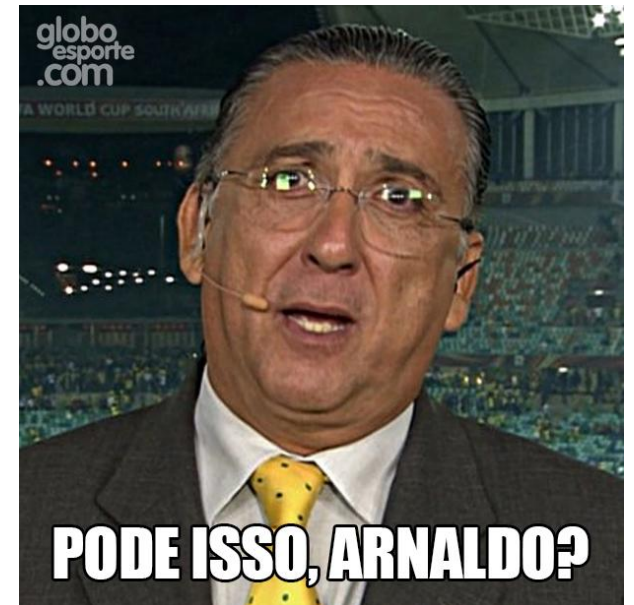
O retorno do método build(),  
que é um objeto "Person",  
invoca o método de instância  
setName()

A partir da Classe  
Person, é invocado o  
método estático build()

O retorno do método setName(),  
que é **VOID**, invoca o método  
setAge()

```
Person pessoa02 = Person.build().setName("Senor Abravanel").setAge(93);
```

O retorno do método build(),  
que é um objeto "Person",  
invoca o método de instância  
setName()







# Mas o que houve



- O erro ocorre porque o método build() retorna uma nova instância de Person, mas a classe Person não está configurada para suportar o encadeamento de métodos (method chaining) diretamente.
- Para que o encadeamento de métodos funcione corretamente, os **métodos intermediários devem sempre retornar um objeto que possa invocar o próximo método na sequência.**
- Isso significa que, ao chamar um método como setName() ou setAge() em um objeto, esse método deve retornar a própria instância do objeto, permitindo que outro método seja chamado imediatamente em seguida.

# Method Chaining (Encadeamento de método)

- Para podermos utilizar a lógica anterior, precisamos preparar a nossa Classe:

# Method

- Para pod  
Classe:

```
1  public class PersonWithChainingMethod {
2      private String name;
3      private Integer age;
4      public static PersonWithChainingMethod build() {
5          return new PersonWithChainingMethod();
6      }
7      public String toString() {
8          return this.name + " - Age: " + this.age;
9      }
10     public String getName() {
11         return name;
12     }
13     public PersonWithChainingMethod setName(String name) {
14         this.name = name;
15         return this;
16     }
17     public Integer getAge() {
18         return age;
19     }
20     public PersonWithChainingMethod setAge(Integer age) {
21         this.age = age;
22         return this;
23     }
24 }
```

método)

car a nossa

# Method

- Para poder usar a nossa Classe:

```
1 public class PersonWithChainingMethod {
2     private String name;
3     private Integer age;
4     public static PersonWithChainingMethod build() {
5         return new PersonWithChainingMethod();
6     }
7     public String toString() {
8         return this.name + " - Age: " + this.age;
9     }
10    public String getName() {
11        return name;
12    }
13    public PersonWithChainingMethod setName(String name) {
14        this.name = name;
15        return this;
16    }
17    public Integer getAge() {
18        return age;
19    }
20    public PersonWithChainingMethod setAge(Integer age) {
21        this.age = age;
22        return this;
23    }
24 }
```

Mantemos o método Estático que retorna um novo objeto Person

ar a nossa

Mantemos o método de Instância que retorna um uma String

- Para poder usar a nossa Classe:

Mantemos o método Estático que retorna um novο objeto Person

```
1 public class PersonWithChainingMethod {
2     private String name;
3     private Integer age;
4     public static PersonWithChainingMethod build() {
5         return new PersonWithChainingMethod();
6     }
7     public String toString() {
8         return this.name + " - Age: " + this.age;
9     }
10    public String getName() {
11        return name;
12    }
13    public PersonWithChainingMethod setName(String name) {
14        this.name = name;
15        return this;
16    }
17    public Integer getAge() {
18        return age;
19    }
20    public PersonWithChainingMethod setAge(Integer age) {
21        this.age = age;
22        return this;
23    }
24 }
```

ar a nossa

Mantemos o método de Instância que retorna um uma String

- Para poder usar a nossa Classe:

```
1 public class PersonWithChainingMethod {
2     private String name;
3     private Integer age;
4     public static PersonWithChainingMethod build() {
5         return new PersonWithChainingMethod();
6     }
7     public String toString() {
8         return this.name + " - Age: " + this.age;
9     }
10    public String getName() {
11        return name;
12    }
13    public PersonWithChainingMethod setName(String name) {
14        this.name = name;
15        return this;
16    }
17    public Integer getAge() {
18        return age;
19    }
20    public PersonWithChainingMethod setAge(Integer age) {
21        this.age = age;
22        return this;
23    }
24 }
```

Mantemos o método Estático que retorna um novo objeto Person

ar a nossa

Mantemos os métodos Gets que retornam seus dados, e Ajustamos os métodos Sets para retornarem o próprio Objeto (this).

Mantemos o método de Instância que retorna um uma String

- Para poder usar a nossa Classe:

```
1 public class PersonWithChainingMethod {
2     private String name;
3     private Integer age;
4     public static PersonWithChainingMethod build() {
5         return new PersonWithChainingMethod();
6     }
7     public String toString() {
8         return this.name + " - Age: " + this.age;
9     }
10    public String getName() {
11        return name;
12    }
13    public PersonWithChainingMethod setName(String name) {
14        this.name = name;
15        return this;
16    }
17    public Integer getAge() {
18        return age;
19    }
20    public PersonWithChainingMethod setAge(Integer age) {
21        this.age = age;
22        return this;
23    }
24 }
```

Mantemos o método Estático que retorna um novo objeto Person

ar a nossa

Mantemos os métodos Gets que retornam seus dados, e Ajustamos os métodos Sets para retornarem o próprio Objeto (this).



Mantemos o método de Instância que retorna um uma String

- Para poder usar a nossa Classe:

```
1 public class PersonWithChainingMethod {
2     private String name;
3     private Integer age;
4     public static PersonWithChainingMethod build() {
5         return new PersonWithChainingMethod();
6     }
7     public String toString() {
8         return this.name + " - Age: " + this.age;
9     }
10    public String getName() {
11        return name;
12    }
13    public PersonWithChainingMethod setName(String name) {
14        this.name = name;
15        return this;
16    }
17    public Integer getAge() {
18        return age;
19    }
20    public PersonWithChainingMethod setAge(Integer age) {
21        this.age = age;
22        return this;
23    }
24 }
```

Mantemos o método Estático que retorna um novo objeto Person

ar a nossa

Mantemos os métodos Gets que retornam seus dados, e Ajustamos os métodos Sets para retornarem o próprio Objeto (this).



```
//Exemplo de Chaining Method possível
// Primeiro é invocado o método "Static" "build()" a partir da Classe
//     Esse método retorna um novo Objeto do tipo "PersonExampleChainingMethod"
//     A partir desse objeto é invocado o método "setName()"
//     Esse método retorna o próprio Objeto "PersonExampleChainingMethod"
//     A partir desse retorno invocamos o método "setAge()"
//     Esse método retorna o próprio Objeto "PersonExampleChainingMethod"
// Por fim o Objeto "PersonExampleChainingMethod" criado é referenciado pela variável "pessoa03"
PersonWithChainingMethod pessoa03 = PersonWithChainingMethod.build().setName("Raul Seixas").setAge(44);
System.out.println(pessoa03); // Imprime na tela os dados da Pessoa - implicitamente invoca o método "toString()"
```

```
//Se não precisamos manter esse objeto em memória,  
// podemos fazer todo o processo e jogar o resultado diretamente para o método "println()"  
System.out.println(PersonWithChainingMethod.build().setName("Carrie Fisher").setAge(60));
```

```
System.out.println(  
    PersonWithChainingMethod  
        .build()  
        .setName("Carrie Fisher")  
        .setAge(60)  
        .toString());
```



```
//Se não precisamos manter esse objeto em memória,  
// podemos fazer todo o processo e jogar o resultado diretamente para o método "println()"  
System.out.println(PersonWithChainingMethod.build().setName("Carrie Fisher").setAge(60));
```

```
System.out.println(  
    PersonWithChainingMethod  
        .build()  
        .setName("Carrie Fisher")  
        .setAge(60)  
        .toString());
```



# Um pouco de Prática...

- Você tem uma lista de Nomes.

```
List<String> names = Arrays.asList("Alice", "Bob", "Charlie", "David", "Eve");
```

- E quer gerar uma nova lista com os nomes em ordem alfabética, todos em maiúsculo (UpperCase) e filtrando somente os nomes com mais de 3 caracteres.

**VAMOS USAR OS MÉTODOS DECLARATIVOS,  
DISPONÍVEIS NO STREAM DO JAVA**  
**map(), filter(), sorted() e toList()**



# Sem usar o Method Chaining

```
List<String> names = Arrays.asList("Alice", "Bob", "Charlie", "David", "Eve");

// Criação do stream
Stream<String> nameStream = names.stream();

// Filtragem dos nomes
Stream<String> filtStream = nameStream.filter(name -> name.length() > 3);

// Mapeamento para maiúsculas
Stream<String> filtAndMapStream = filtStream.map(name -> name.toUpperCase());

// Ordenação dos nomes
Stream<String> filtAndMapAndSortedStream = filtAndMapStream.sorted();

// Coleta em uma lista
List<String> newListNames = filtAndMapAndSortedStream.toList();
//List<String> newListNames = filtAndMapAndSortedStream.collect(Collectors.toList());

System.out.println(newListNames);
```

# Usando o Method Chaining

```
List<String> newListNames2 = names.stream()  
    .filter(name -> name.length() > 3)  
    .map(name -> name.toUpperCase())  
    .sorted()  
    .toList();  
  
System.out.println(newListNames2);
```

Vamos comparar!!!



# Vamos comparar!!!

```
Stream<String> namesStream = names.stream();  
Stream<String> namesFiltered = namesStream.filter(name -> name.length() > 3);  
Stream<String> namesFilteredMapped = namesFiltered.map(name -> name.toUpperCase());  
Stream<String> namesFilteredMappedSorted = namesFilteredMapped.sorted();  
List<String> newNames = namesFilteredMappedSorted.toList();  
System.out.println(newNames);
```

# Vamos comparar!!!

```
Stream<String> namesStream = names.stream();  
Stream<String> namesFiltered = namesStream.filter(name -> name.length() > 3);  
Stream<String> namesFilteredMapped = namesFiltered.map(name -> name.toUpperCase());  
Stream<String> namesFilteredMappedSorted = namesFilteredMapped.sorted();  
List<String> newNames = namesFilteredMappedSorted.toList();  
System.out.println(newNames);
```

```
System.out.println(names  
    .stream()  
    .filter(name -> name.length() > 3)  
    .map(name -> name.toUpperCase())  
    .sorted()  
    .toList()  
);
```



# E o Stream???

Porquê usamos esse “bendito” método **stream()**

```
System.out.println(names  
    .stream()  
    .filter(name -> name.length() > 3)  
    .map(name -> name.toUpperCase())  
    .sorted()  
    .toList()  
);
```



# E o Stream???

## Porquê usamos esse “bendito” método `stream()`

- Em Java, um **Stream** é uma sequência de elementos que suporta operações agregadas de **forma funcional**.
- Ele faz parte da API de Streams introduzida no Java 8 (`java.util.stream`) e permite processar coleções de dados de maneira declarativa



# Stream

- **Não armazena dados** – Um Stream não é uma estrutura de dados, mas sim um pipeline de computação que processa elementos de uma fonte (como uma Collection, um array ou um gerador de valores).
- **Opera de forma funcional** – Permite a composição de operações como map(), filter(), reduce() e outras, seguindo o paradigma funcional.
- **Pode ser sequencial ou paralelo** – Pode processar dados de maneira sequencial ou em paralelo (***parallelStream()***), aproveitando múltiplos núcleos do processador.
- **É consumível** – Uma vez operado e fechado, um Stream não pode ser reutilizado.

# Ou seja...



- O Stream em Java NÃO cria uma nova lista ou armazena os dados processados.
- Em vez disso, ele constrói um pipeline de operações que será executado somente quando uma operação terminal for chamada.

## 📌 Como funciona o pipeline de um Stream?

1. **Criação** → O Stream é obtido de uma fonte (como uma `List`).
2. **Operações intermediárias** → Definem transformações nos dados, mas não são executadas imediatamente.
3. **Operação terminal** → Quando chamada, ativa o pipeline e realiza o processamento.

# Tá bom... Sei que não ficou tão claro assim...

- Vamos ver de outra maneira

```
List<String> nomes = List.of("Ana", "Bruno", "Carlos", "Daniel");  
// Nenhuma operação foi executada ainda!  
var streamPipeline = nomes.stream()  
    .filter(nome -> {  
        System.out.println("Filtrando: " + nome);  
        return nome.startsWith("B");  
    })  
    .map(nome -> {  
        System.out.println("Convertendo para maiúsculas: " + nome);  
        return nome.toUpperCase();  
    });  
System.out.println("Pipeline criado, mas nada foi processado ainda!");  
System.out.println("=====");  
// Agora o Stream será processado  
List<String> resultado = streamPipeline.toList();  
System.out.println("=====");  
System.out.println("Após a Operacao Terminal for chamada, então o pipeline é processado!");  
System.out.println("Resultado final: " + resultado);
```



```
List<String> nomes = List.of("Ana", "Bruno", "Carlos", "Daniel");  
// Nenhuma operação foi executada ainda!  
var streamPipeline = nomes.stream()  
    .filter(nome -> {  
        System.out.println("Filtrando: " + nome);  
        return nome.startsWith("B");  
    })  
    .map(nome -> {  
        System.out.println("Convertendo para maiúsculas: " + nome);  
        return nome.toUpperCase();  
    });  
System.out.println("Pipeline criado, mas nada foi processado ainda!");  
System.out.println("=====");  
// Agora o Stream será processado  
List<String> resultado = streamPipeline.toList();  
System.out.println("=====");  
System.out.println("Após a Operacao Terminal for chamada, então o pipeline é processado!");  
System.out.println("Resultado final: " + resultado);
```



```
List<String> nomes = List.of("Ana", "Bruno", "Carlos", "Daniel");
```

```
// Nenhuma operação foi executada ainda!
```

```
var streamPipeline = nomes.stream()
```

```
.filter(nome -> {
```

```
    System.out.println("Filtrando: " + nome);
```

```
    return nome.startsWith("B");
```

```
});
```

```
.map(nome -> {
```

```
    System.out.println("Convertendo para maiúsculas
```

```
    return nome.toUpperCase();
```

```
});
```

```
System.out.println("Pipeline criado, mas nada foi processado ainda!");
```

```
System.out.println("=====");
```

```
// Agora o Stream será processado
```

```
List<String> resultado = streamPipeline.toList();
```

```
System.out.println("=====");
```

```
System.out.println("Após a Operacao Terminal for chamada, então o pipeline é processado!");
```

```
System.out.println("Resultado final: " + resultado);
```

Pipeline criado, mas nada foi processado ainda!

=====

Filtrando: Ana

Filtrando: Bruno

Convertendo para maiúsculas: Bruno

Filtrando: Carlos

Filtrando: Daniel

=====

Após a Operacao Terminal for chamada, então o pipeline é processado!

Resultado final: [BRUNO]

```
List<String> nomes = List.of("Ana", "Bruno", "Carlos", "Daniel");
```

```
// Nenhuma operação foi executada ainda!
```

```
var streamPipeline = nomes.stream()
```

```
.filter(nome -> {
```

```
    System.out.println("Filtrando: " + nome);
```

```
    return nome.startsWith("B");
```

```
});
```

```
.map(nome -> {
```

```
    System.out.println("Convertendo para maiúsculas
```

```
    return nome.toUpperCase();
```

```
});
```

```
System.out.println("Pipeline criado, mas nada foi processado ainda!");
```

```
System.out.println("=====");
```

```
// Agora o Stream será processado
```

```
List<String> resultado = streamPipeline.toList();
```

```
System.out.println("=====");
```

```
System.out.println("Após a Operação Terminal for chamada, então o pipeline é processado!");
```

```
System.out.println("Resultado final: " + resultado);
```

Pipeline criado, mas nada foi processado ainda!

=====

Filtrando: Ana

Filtrando: Bruno

Convertendo para maiúsculas: Bruno

Filtrando: Carlos

Filtrando: Daniel

=====

Após a Operação Terminal for chamada, então o pipeline é processado!

Resultado final: [BRUNO]

### 🔍 O que aconteceu?

- O pipeline ( `stream()` → `filter()` → `map()` ) foi construído, mas nada aconteceu até chamarmos `toList()`.
- Quando `toList()` foi chamado (operação terminal), o Stream processou **somente os elementos necessários**.
- Como `Ana`, `Carlos` e `Daniel` não passaram pelo `filter()`, eles nem chegaram a ser convertidos para maiúsculas.



## Resumo:

- ✓ O `Stream` é lazy – só executa quando uma operação terminal é chamada.
- ✓ O pipeline é processado de forma eficiente, executando apenas o necessário.
- ✓ Não há criação de novas coleções intermediárias – apenas o resultado final é materializado.

Agora vamos  
continuar de  
onde  
paramos...



# Paradigmas de Programação

|

|—

Imperativos



|

|—

Procedural



|

|

|—

Estruturado



|

|—

Orientação a Objetos (OO)



|

|—

Declarativos

|—

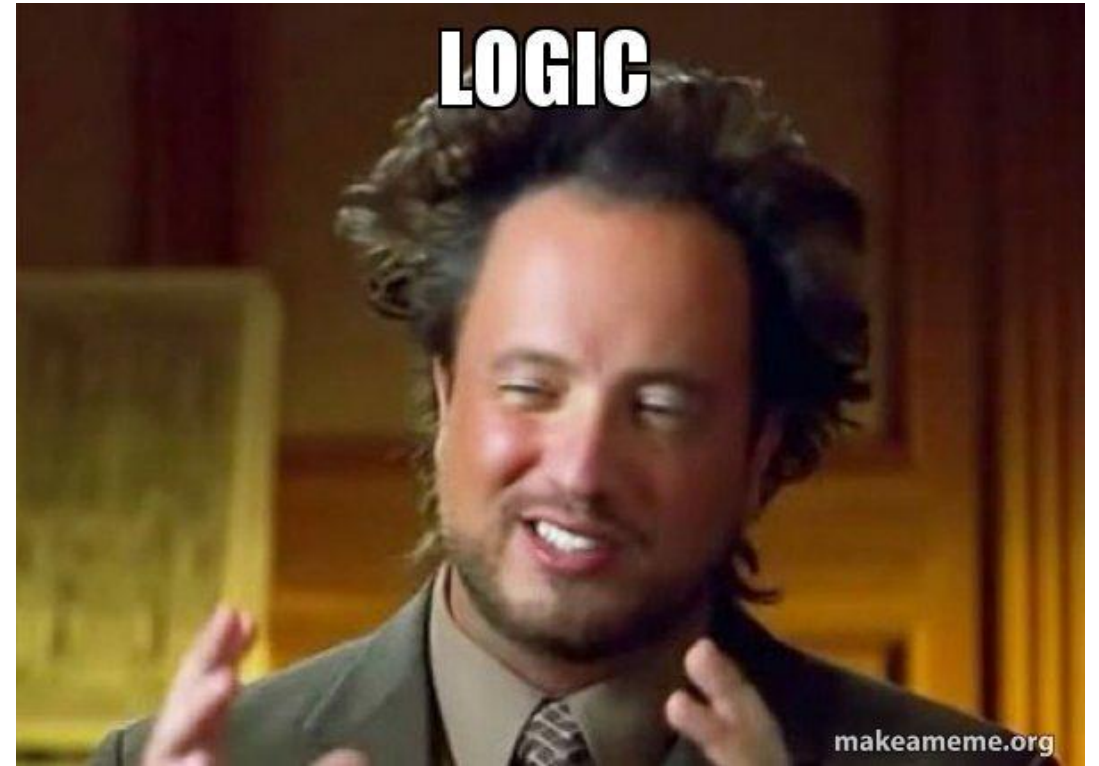
Funcional



|—

Lógico

Paradigma Lógico



# Paradigma Lógico

- Computação baseada em **lógica formal**
- Os programas são formados por um conjunto de **fatos** e **regras**
- A partir destes **fatos** e **regras**, o sistema de inferência tenta encontrar soluções para as consultas feitas.

# Paradigma Lógico

## Características principais:

1. **Baseado em lógica matemática** – Especificamente, na **Lógica de Predicados de Primeira Ordem**.
2. **Inferência automática** – O mecanismo de inferência (backtracking e unificação) busca respostas sem que o programador precise especificar os passos exatos da computação.
3. **Não sequencial** – A execução não segue uma sequência fixa de comandos, como ocorre na programação imperativa.
4. **Declaração de fatos e regras** – O conhecimento é representado por **fatos** (informações conhecidas) e **regras** (relações entre os fatos).
5. **Consulta para encontrar respostas** – O usuário faz perguntas ao sistema, e ele tenta deduzir a resposta automaticamente.



# Exemplos

```
family.pl
1  % Fatos (base de conhecimento)
2  pai(joao, pedro).
3  pai(joao, maria).
4  pai(carlos, ana).
5  mae(mariana, pedro).
6  mae(mariana, maria).
7  mae(ana, beatriz).
8
9  % Regra para definir se X é um irmão de Y
10 irmao(X, Y) :- pai(P, X), pai(P, Y), X \= Y.
11 irmao(X, Y) :- mae(M, X), mae(M, Y), X \= Y.
12
13 % Consulta: Quem são os irmãos de Maria?
14 % ?- irmao(maria, Quem).
```

```
myfamily.pl
1  % Fatos (base de conhecimento)
2  pai(idalencio, valdemar).
3  pai(valdemar, luciano).
4  pai(valdemar, marcio).
5  pai(luciano, luigi).
6  pai(luciano, luisa).
7  pai(luciano, lucas).
8
9  % Regras para definir relações familiares
10 avo(X, Y) :- pai(X, Z), pai(Z, Y). % X é avô de Y se X é pai de Z e Z é pai de Y.
11 filho(X, Y) :- pai(Y, X). % X é filho de Y se Y é pai de X.
12 irmao(X, Y) :- pai(Z, X), pai(Z, Y), X \= Y. % X é irmão de Y se Z é pai de X e Z é pai de Y
```

# Linguagens Lógicas

## Puramente Lógica

- Prolog (<https://www.swi-prolog.org/>)

## Multiparadigmas

- Oz (<http://mozart2.org/>)
- λProlog (<https://www.lix.polytechnique.fr/Labo/Dale.Miller/lProlog/>)

## Outras que implementam paradigma lógico

- F# (<https://fsharp.org/>)
- Clojure (<https://clojure.org/>)



SWI Prolog



# Paradigmas de Programação

|

|— Imperativos

| |— Procedural

| | |— Estruturado

| |— Orientação a Objetos (OO)

|

|— Declarativos

|— Funcional

|— Lógico

# Programação Orientada a Aspectos

## AOP (Aspect-Oriented Programming)

# AOP – Programação orientada a aspectos

- Programação Orientada a Aspectos (AOP, do inglês *Aspect-Oriented Programming*) tem como objetivo separar as preocupações transversais de uma aplicação, ou seja, aquelas funcionalidades que afetam várias partes do sistema, como o registro de logs, segurança, transações, entre outras.

# Spring AOP

```
@Aspect  
@Component  
public class LogAspect {  
  
    @Before("execution(* com.exemplo.service.*.*(..))")  
    public void logBefore(JoinPoint joinPoint) {  
        System.out.println("Método " + joinPoint.getSignature().getName() + " está sendo chamado...");  
    }  
  
    @After("execution(* com.exemplo.service.*.*(..))")  
    public void logAfter(JoinPoint joinPoint) {  
        System.out.println("Método " + joinPoint.getSignature().getName() + " foi executado.");  
    }  
}
```

# Entregáveis

Vamos lá. Está na hora!!!



