

Classe Anônimas

É uma classe interna que não é declarada de forma explícita sendo utilizada em determinada trecho do código.

Exemplo:

```
public class Pedido {  
    public void finalizarPedido() {  
        System.out.println("Pedido Finalizado !!");  
    }  
}
```

```
public class TestePedido {  
    public static void main(String[] args) {  
        Pedido pedido = new Pedido();  
        pedido.finalizarPedido();  
  
        Pedido pedido2 = new Pedido() {  
            @Override  
            public void finalizarPedido() {  
                System.out.println("Pedido Finalizado com Sucesso !!");  
            }  
        };  
        pedido2.finalizarPedido();  
    }  
}
```

Utilizando classes anônimas podemos modificar o comportamento da classe

Classe Anônimas

No exemplo abaixo criamos uma instância de Thread passando um Runnable no construtor. Thread é uma forma do processo se dividir em uma ou mais tarefas, sendo muito utilizado para tirarmos vantagem de múltiplas CPUs por exemplo.

```
package model;

public class ExemploClasseAnonima {

    public static void main(String[] args) {
        new Thread(new Runnable() {
            public void run() {
                System.out.println("Hello !!");
            }
        }).run();
    }
}
```

Classe Anônimas

Uma outra forma de utilizarmos classe anônimas é através da utilização de interfaces, onde podemos instanciar uma classe anônima que implementa uma interface. No exemplo abaixo instanciamos uma classe sem nome que implementa o método da interface.

```
public interface Conta {  
    public void transacao();  
}
```

```
public class Principal {  
    public static void main(String[] args) {  
        Conta conta = new Conta() {  
            @Override  
            public void transacao() {  
                System.out.println("Transação Efetuada !!");  
            }  
        };  
        conta.transacao();  
    }  
}
```

Programação Funcional no Java

É um paradigma de programação com base em resultados em funções e a programação é feita com expressões.

No Java a partir da versão 8 foram adicionadas expressões como lambda e referências de métodos facilitando a utilização da programação funcional.

O lambda, () -> System.out.println("Hello !!") significa:

() não há parâmetros

-> separa a declaração dos parâmetros do corpo do método

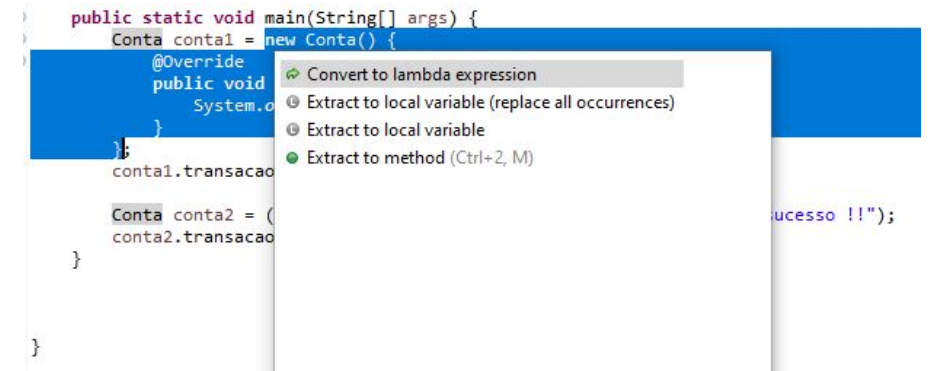
O restante é o corpo do método, que se for apenas um comando não precisa ser declarado como um bloco usando { }. Dessa forma é o mesmo que instanciar uma classe anônima e implementar o método existente.

```
public interface Conta {  
    public void transacao();  
}
```

Podemos criar a expressão Lambda selecionando o trecho de código e pressionando **CTRL+1**

```
public class TesteConta {  
    public static void main(String[] args) {  
        Conta conta1 = new Conta() {  
            @Override  
            public void transacao() {  
                System.out.println("Transação Efetuada !!");  
            }  
        };  
        conta1.transacao();  
        Conta conta2 = () -> System.out.println("Transição efetuada com sucesso !!");  
        conta2.transacao();  
    }  
}
```

```
public static void main(String[] args) {  
    Conta conta1 = new Conta() {  
        @Override  
        public void transacao() {  
            System.out.println("Transição efetuada com sucesso !!");  
        }  
    };  
    conta1.transacao();  
    Conta conta2 = () -> System.out.println("Transição efetuada com sucesso !!");  
    conta2.transacao();  
}
```



Programação Funcional no Java

Vamos utilizar um outro exemplo com um componente JButton do Swing.

```
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

import javax.swing.JButton;

public class TesteJava8 {

    public static void main(String[] args) {
        JButton jButton = new JButton();
        jButton.addActionListener(new ActionListener() {

            @Override
            public void actionPerformed(ActionEvent e) {
                System.out.println("Hello !!!");
            }

        });
    }
}
```

Adicionando outro JButton utilizando a expressão Lambda

```
JButton jButton2 = new JButton();
jButton2.addActionListener(e -> System.out.println("Hello !!"));
```



Como o Java sabe que estamos esperando um **ActionListener**?
E como ele sabe que está sendo implementado um método **actionPerformed**

Single Abstract Method

O conceito de **SAM** é para qualquer interface que tenha um único método abstrato. O compilador entende na utilização do lambda que a implementação será desse único método. Visualizando a documentação da Interface ActionListener ao lado ela só possui um método abstrato.

```
public interface ActionListener extends EventListener {

    /**
     * Invoked when an action occurs.
     */
    public void actionPerformed(ActionEvent e);

}
```

Interfaces Funcionais

É uma Interface que possui apenas um método abstrato. A maioria das interfaces funcionais possui a anotação `@FunctionalInterface`. A anotação garante que quando o código for compilado a interface não poderá ter mais de um método abstrato, qualquer interface com um único método abstrato é uma interface funcional e sua implementação pode ser tratada com expressões lambda.

Abaixo um exemplo de como criar uma interface funcional.

```
@FunctionalInterface
public interface Conta {

    public void transacao();

}
```

```
public class ContaCorrente implements Conta {

    @Override
    public void transacao() {
        System.out.println("Transação Efetuada !!");
    }

}
```

Métodos Default

Os métodos em um interface são sempre públicos e abstratos. Quando adicionamos um método novo em uma interface todos as classes devem implementar. Isto acaba sendo um problema, a partir da versão 8 temos o recurso dos métodos default, utilizando a palavra reservada default, é possível criar métodos default em interfaces, que é uma implementação padrão da interface. A maior necessidade para a criação de métodos default é de adicionar novas funcionalidades às interfaces existentes. Podemos ter vários métodos default em uma interface.

```
public interface Conta {  
    public void transacao();  
}  
  
public class ContaCorrente implements Conta {  
    @Override  
    public void transacao() {  
        System.out.println("Transação Efetuada !!");  
    }  
}
```

No exemplo ao lado se adicionarmos um novo método na interface Conta seremos obrigados a implementá-lo na classe ContaCorrente

Inserindo o modificador default na definição do método investimento, temos um método com corpo na interface não somos obrigados a implementar o método na classe ContaCorrente, evitando assim problemas de erro de código.

```
public interface Conta {  
    public void transacao();  
    default void investimento() {  
        System.out.println("Investimento Efetuado !! (Método Default) ");  
    }  
}  
  
public class TesteConta {  
    public static void main(String[] args) {  
        ContaCorrente cc = new ContaCorrente();  
        cc.transacao();  
        cc.investimento();  
    }  
}
```

```
public class ContaCorrente implements Conta {  
    @Override  
    public void transacao() {  
        System.out.println("Transação Efetuada !!");  
    }  
    @Override  
    public void investimento() {  
        System.out.println("Método default modificado !!");  
    }  
}
```

Mas se quisermos também podemos também mudar a implementação do método investimento na classe ContaCorrente

Métodos Default

Neste outro exemplo abaixo temos a interface Iterable da biblioteca do Java que possui apenas um método abstrato e possui dois métodos default .

```
public interface Iterable<T> {  
    /**  
     * Returns an iterator over elements of type {@code T}.  
     *  
     * @return an Iterator.  
     */  
    Iterator<T> iterator();  
  
    /**  
     * Performs the given action for each element of the {@code Iterable}  
     * until all elements have been processed or the action throws an  
     * exception. Unless otherwise specified by the implementing class,  
     * actions are performed in the order of iteration (if an iteration order  
     * is specified). Exceptions thrown by the action are relayed to the  
     * caller.  
     *  
     * @implSpec  
     * <p>The default implementation behaves as if:  
     * <pre>{@code  
     *     for (T t : this)  
     *         action.accept(t);  
     * }</pre>  
     *  
     * @param action The action to be performed for each element  
     * @throws NullPointerException if the specified action is null  
     * @since 1.8  
     */  
    default void forEach(Consumer<? super T> action) {  
        Objects.requireNonNull(action);  
        for (T t : this) {  
            action.accept(t);  
        }  
    }  
  
    default Spliterator<T> spliterator() {  
        return Spliterators.spliteratorUnknownSize(iterator(), 0);  
    }  
}
```


Exercícios

Vamos utilizar um exemplo com a classe Aluno utilizando o método foreach da interface Iterable.

Usando foreach

```
public class Aluno {
    private String nome;
    private String email;

    public Aluno(String nome, String email) {
        super();
        this.nome = nome;
        this.email = email;
    }

    public String getNome() {
        return nome;
    }

    public String getEmail() {
        return email;
    }
}
```

Listando os alunos de forma tradicional usando for

```
public class TesteAluno {

    public static void main(String[] args) {
        Aluno aluno1 = new Aluno("Sandra", "sandra@uol.com.br");
        Aluno aluno2 = new Aluno("Marcos", "marcos@hotmail.com");

        List<Aluno> alunos = Arrays.asList(aluno1, aluno2);

        for (Aluno aluno : alunos) {
            System.out.println(aluno.getNome());
            System.out.println(aluno.getEmail());
        }
    }
};
```

O método foreach espera como argumento um Consumer, vamos precisar criar uma nova classe ExibeDados que implemente Consumer.

```
import java.util.function.Consumer;

public class ExibeDados implements Consumer<Aluno> {

    @Override
    public void accept(Aluno aluno) {
        System.out.println(aluno.getNome());
        System.out.println(aluno.getEmail());
    }
}
```

Usando o método default foreach da interface Iterable

```
public class TesteAluno {

    public static void main(String[] args) {
        Aluno aluno1 = new Aluno("Sandra", "sandra@uol.com.br");
        Aluno aluno2 = new Aluno("Marcos", "marcos@hotmail.com");

        List<Aluno> alunos = Arrays.asList(aluno1, aluno2);
        ExibeDados exibeDados = new ExibeDados();
        alunos.forEach(exibeDados);
    }
};
```

Melhorando o código usando classes anônimas

No exemplo anterior criamos um classe implementando Consumer só para uso no foreach abaixo substituímos a classe ExibeDados pela classe anônima.

```
import java.util.Arrays;
import java.util.List;
import java.util.function.Consumer;

public class TesteAluno {

    public static void main(String[] args) {
        Aluno aluno1 = new Aluno("Sandra", "sandra@uol.com.br");
        Aluno aluno2 = new Aluno("Marcos", "marcos@hotmail.com");

        List<Aluno> alunos = Arrays.asList(aluno1, aluno2);

        Consumer<Aluno> consumer = new Consumer<Aluno>() {

            @Override
            public void accept(Aluno a) {
                System.out.println(a.getNome());
                System.out.println(a.getEmail());
            }

        };

        alunos.forEach(consumer);
    }
};
```

Melhorando o código usando Lambda

Como só tem um parâmetro podemos deixar sem parênteses a variável a

```
import java.util.Arrays;
import java.util.List;
import java.util.function.Consumer;

public class TesteAluno {

    public static void main(String[] args) {
        Aluno aluno1 = new Aluno("Sandra", "sandra@uol.com.br");
        Aluno aluno2 = new Aluno("Marcos", "marcos@hotmail.com");

        List<Aluno> alunos = Arrays.asList(aluno1, aluno2);

        Consumer<Aluno> consumer = a -> {
            System.out.println(a.getNome());
            System.out.println(a.getEmail());
        };

        alunos.forEach(consumer);
    }
};
```

Podemos inserir o código do consumer dentro do foreach reduzindo ainda mais o código. Lembrando que o lambda só pode ser utilizado em interfaces funcionais.

```
}; public class TesteAluno {

    public static void main(String[] args) {
        Aluno aluno1 = new Aluno("Sandra", "sandra@uol.com.br");
        Aluno aluno2 = new Aluno("Marcos", "marcos@hotmail.com");

        List<Aluno> alunos = Arrays.asList(aluno1, aluno2);

        alunos.forEach(a -> {
            System.out.println(a.getNome());
            System.out.println(a.getEmail());
        });
    }
};
```

O método foreach recebe um Consumer e para ele já estará implícito.

Method Reference

As referências a métodos ajudam a reduzir a quantidade de código. No exemplo abaixo foi utilizado o delimitador :: concatenando com o nome do método.

```
public class TesteAluno {  
    public static void main(String[] args) {  
        Aluno aluno1 = new Aluno("Sandra", "sandra@uol.com.br",10);  
        Aluno aluno2 = new Aluno("Marcos", "marcos@hotmail.com",20);  
  
        List<Aluno> alunos = Arrays.asList(aluno1, aluno2);  
  
        System.out.println("Imprimindo com Lambda");  
        alunos.forEach(a -> {  
            System.out.println(a.getNome());  
            System.out.println(a.getEmail());  
            System.out.println(a.getPontuacao());  
        });  
  
        System.out.println("\nImprimindo com Method Reference");  
        alunos.forEach(System.out::println);  
    }  
};
```

```
Imprimindo com Method Reference  
exemplos.Aluno@53d8d10a  
exemplos.Aluno@e9e54c2
```

Precisamos inserir o **ToString** na classe **Aluno** para imprimir os dados do aluno.

```
@Override  
public String toString() {  
    return nome + " " + email;  
}
```

```
Imprimindo com Method Reference  
Sandra sandra@uol.com.br 10  
Marcos marcos@hotmail.com 20
```

Method Reference

O Method References, é descrito como uma expressão lambda compacta e fácil de ser lida é uma forma mais elegante e enxuta de se escrever uma expressão lambda cuja única tarefa é de invocar um método.

```
public class Autor {
    private String nome;
    private String telefone;

    public Autor(String nome, String telefone) {
        super();
        this.nome = nome;
        this.telefone = telefone;
    }

    @Override
    public String toString() {
        return "Author [nome=" + nome + ", telefone=" + telefone + "]";
    }

    public String getNome() {
        return nome;
    }

    public String getTelefone() {
        return telefone;
    }

    public void imprime() {
        System.out.println(nome);
    }
}
```

```
public class TesteAutor {

    public static void main(String[] args) {
        // TODO Auto-generated method stub

        Autor autor1 = new Autor("Jose", "234343");
        Autor autor2 = new Autor("Maria", "1234343");
        List<Autor> autores = Arrays.asList(autor1, autor2);

        Consumer<Autor> consumerComLambda = a -> a.imprime();
        Consumer<Autor> consumerComMethodRef = Autor::imprime;

        autores.forEach(consumerComMethodRef);
        System.out.println("-----");
        autores.forEach(consumerComLambda);

        System.out.println("-----");
        autores.forEach(Autor::imprime);
    }
}
```


Exercícios

Criar uma interface funcional com o nome **Calculo**. Criar o método abstrato **operacao** com retorno de um inteiro e recebendo dois argumentos inteiros. Criar um classe chamada **TesteSoma** com o **main** para realizar o cálculo da soma de dois inteiros e exibir o resultado da soma.

```
@FunctionalInterface
public interface Calculo {
    int operacao(int a, int b);
}
```

Usando classe anônima

```
public class TesteSoma {
    public static void main(String[] args) {
        Calculo calculo = new Calculo() {
            @Override
            public int operacao(int a, int b) {
                return a + b;
            }
        };
        System.out.println("O resultado da soma é:" + calculo.operacao(10, 20));
    }
}
```

Usando Lambda

```
public class TesteSoma {
    public static void main(String[] args) {
        Calculo calculo = (a, b) -> a + b;
        System.out.println("O resultado da soma é:" + calculo.operacao(10, 20));
    }
}
```

Usando Method Reference

```
public class TesteSoma {
    public static void main(String[] args) {
        // Calculo calculo = (a, b) -> a + b;
        Calculo calculo = Integer::sum;
        System.out.println("O resultado da soma é:" + calculo.operacao(10, 20));
    }
}
```

Exercícios

Criar uma classe contendo uma lista de String com o nome de várias linguagens de programação. Usando Method Reference imprimir o conteúdo da lista.

```
public class Exercicio1 {  
    public static void main(String[] args) {  
        List<String> linguagens = Arrays.asList("Delphi", "Java", "Java Script", "Dart");  
        linguagens.forEach(System.out::println);  
    }  
}
```

Streams

Recurso introduzido a partir do Java 8 que oferece a possibilidade de trabalharmos com conjuntos de elementos de forma mais simples e organizada reduzindo de forma significativa a quantidade de código. A forma tradicional de utilizar Streams é através de uma coleção de dados.

No exemplo abaixo criamos uma lista e em seguida criamos uma variável do tipo Stream que recebe o valor retornado do método stream. No final utilizamos o forEach para exibir os dados da lista.

```
public class AulaStreams {  
  
    public static void main(String[] args) {  
        List<String> times = Arrays.asList("Flamengo", "Vasco", "Fluminense", "Botafogo");  
        Stream<String> stream = times.stream();  
        stream.forEach(t -> System.out.println(t));  
  
    }  
}
```

Streams

Recurso introduzido a partir do Java 8 que oferece a possibilidade de trabalharmos com conjuntos de elementos de forma mais simples e organizada reduzindo de forma significativa a quantidade de código. Quando utilizamos um Stream ele não altera a lista original.

```
public class TesteStream {  
    public static void main(String[] args) {  
        Aluno aluno1 = new Aluno("José", "jose@gmail.com", 76);  
        Aluno aluno2 = new Aluno("Ana", "ana@hotmail.com", 52);  
        Aluno aluno3 = new Aluno("Maria", "maria@gmail.com", 90);  
        Aluno aluno4 = new Aluno("Carlos", "carlos@yahoo.com.br", 40);  
        Aluno aluno5 = new Aluno("Ricardo", "ricardo@uol.com.br", 38);  
  
        List<Aluno> lista = Arrays.asList(aluno1, aluno2, aluno3, aluno4, aluno5);  
  
        lista.stream().forEach(e -> System.out.println(e));  
    }  
}
```

Podemos realizar operações intermediárias no resultado do nosso foreach utilizando Streams

No exemplo abaixo temos a nossa lista de nomes e queremos criar uma nova lista apenas com apenas as primeiras 3 letras de cada nome.

```
lista.forEach(System.out::println);  
List<String> nomes = lista.stream().map(n -> n.getNome().substring(0,3)).collect(Collectors.toList());  
System.out.println(nomes);
```


Métodos do Streams

Filter

Filtra elementos de uma Stream conforme condição informada. No exemplo abaixo filtramos a pontuação dos alunos acima de 50 e nome iniciados com "A"

```
public class TesteStream {  
    public static void main(String[] args) {  
        Aluno aluno1 = new Aluno("José", "jose@gmail.com", 76);  
        Aluno aluno2 = new Aluno("Ana", "ana@hotmail.com", 52);  
        Aluno aluno3 = new Aluno("Maria", "maria@gmail.com", 90);  
        Aluno aluno4 = new Aluno("Carlos", "carlos@yahoo.com.br", 40);  
        Aluno aluno5 = new Aluno("Ricardo", "ricardo@uol.com.br", 38);  
  
        List<Aluno> lista = Arrays.asList(aluno1, aluno2, aluno3, aluno4, aluno5);  
  
        lista.stream().forEach(e -> System.out.println(e));  
  
        System.out.println("\n Pontuação acima de 50");  
        lista.stream().filter(a -> a.getPontuacao() > 50).forEach(a -> System.out.println(a));  
  
        System.out.println("\n Nomes que iniciam com A");  
        lista.stream().filter(a -> a.getNome().startsWith("A")).forEach(a -> System.out.println(a));  
    }  
}
```

A interface funcional usada para representar essa expressão, que é o argumento do método filter(), é a interface Predicate. Ela expõe um único método abstrato, boolean test(T t) conforme documentação do Java.

```
@FunctionalInterface  
public interface Predicate<T> {  
  
    /**  
     * Evaluates this predicate on the given argument.  
     *  
     * @param t the input argument  
     * @return {@code true} if the input argument matches the predicate,  
     *         otherwise {@code false}  
     */  
    boolean test(T t);  
}
```

O tipo parametrizado T vai representar o tipo do elemento do nosso stream, objetos do tipo Aluno, então é como se nossa expressão lambda representasse a implementação do método test(). Após aplicar a filtragem é só chamar o método foreach.

Métodos do Streams

Skip

O método **skip** pula uma quantidade determinada de elementos.

```
public class Java8Streams {  
  
    public static void main(String[] args) {  
        List<Double> listaMedias = Arrays.asList(7.8, 5.0, 8.3, 9.2, 1.8, 4.5, 7.5, 6.8, 8.3);  
  
        listaMedias.stream()  
            .skip(2)  
            .forEach(e -> System.out.println(e));  
  
    }  
}
```

Os dois primeiros elementos serão ignorados no resultado

Limit

Limita a quantidade de elementos que serão exibidos.

```
public class Java8Streams {  
  
    public static void main(String[] args) {  
        List<Double> listaMedias = Arrays.asList(7.8, 5.0, 8.3, 9.2, 1.8, 4.5, 7.5, 6.8, 8.3);  
  
        listaMedias.stream()  
            .limit(3)  
            .forEach(e -> System.out.println(e));  
  
    }  
}
```

Métodos do Streams

Distinct

Não permite elementos duplicados. Para isto a classe que iremos trabalhar deverá implementar o método **equal** e **hashCode**.

```
public class Java8Streams {  
  
    public static void main(String[] args) {  
        List<Double> listaMedias = Arrays.asList(7.8, 5.0, 8.3, 9.2, 1.8, 4.5, 7.5, 6.8, 8.3);  
  
        listaMedias.stream()  
            .limit(10)  
            .distinct()  
            .forEach(e -> System.out.println(e));  
    }  
}
```

elementos e não será impresso duas vezes a média 8.3

Sorted

Retorna os elementos de forma ordenada.

```
public class Java8Streams {  
  
    public static void main(String[] args) {  
        List<Double> listaMedias = Arrays.asList(7.8, 5.0, 8.3, 9.2, 1.8, 4.5, 7.5, 6.8, 8.3);  
  
        listaMedias.stream()  
            .sorted()  
            .forEach(e -> System.out.println(e));  
    }  
}
```

Métodos do Streams

Ordenando antes java 8

```
public class Aluno implements Comparable<Aluno>{
    private String nome;
    private String email;
    private Integer idade;

    public Aluno(String nome, String email, Integer idade) {
        super();
        this.nome = nome;
        this.email = email;
        this.idade = idade;
    }

    public String getNome() {
        return nome;
    }

    public String getEmail() {
        return email;
    }

    public Integer getIdade() {
        return idade;
    }

    @Override
    public String toString() {
        return "Aluno [nome=" + nome + ", email=" + email + ", idade=" + idade + "]";
    }

    @Override
    public int compareTo(Aluno o) {
        return idade - o.getIdade();
    }
}
```

```
public class Exemplo1Stream {

    public static void main(String[] args) {
        Aluno a1 = new Aluno("Ana", "ana@gmail.com", 30);
        Aluno a2 = new Aluno("Mario", "ana@gmail.com", 32);
        Aluno a3 = new Aluno("Vitor", "ana@gmail.com", 43);
        Aluno a4 = new Aluno("Celio", "ana@gmail.com", 23);
        Aluno a5 = new Aluno("Ana", "ana@gmail.com", 30);

        List<Aluno> lista = Arrays.asList(a1,a2,a3,a4,a5);
        lista.stream().sorted().forEach(e-> System.out.println(e));
    }
}
```

Métodos do Streams

No Java 8

```
public class Aluno {
    private String nome;
    private String email;
    private Integer idade;

    public Aluno(String nome, String email, Integer idade) {
        super();
        this.nome = nome;
        this.email = email;
        this.idade = idade;
    }

    public String getNome() {
        return nome;
    }

    public void setNome(String nome) {
        this.nome = nome;
    }

    public String getEmail() {
        return email;
    }

    public void setEmail(String email) {
        this.email = email;
    }

    public Integer getIdade() {
        return idade;
    }

    public void setIdade(Integer idade) {
        this.idade = idade;
    }

    @Override
    public String toString() {
        return "Aluno [nome=" + nome + ", email=" + email + ", idade=" + idade + "]";
    }
}
```

```
public class Exemplo1Stream {

    public static void main(String[] args) {
        Aluno a1 = new Aluno("Ana", "ana@gmail.com", 30);
        Aluno a2 = new Aluno("Mario", "ana@gmail.com", 32);
        Aluno a3 = new Aluno("Vitor", "ana@gmail.com", 43);
        Aluno a4 = new Aluno("Celio", "ana@gmail.com", 23);
        Aluno a5 = new Aluno("Ana", "ana@gmail.com", 30);

        List<Aluno> lista = Arrays.asList(a1,a2,a3,a4,a5);

        Comparator<Aluno>comparePeloNome = (o1,o2) -> o2.getIdade().compareTo(o1.getIdade());

        Collections.sort(lista,comparePeloNome);
        lista.forEach(a -> System.out.println(a));

        System.out.println("-----Outra Forma-----");
        lista.sort((o1,o2) -> o2.getIdade().compareTo(o1.getIdade()));
        lista.forEach(a-> System.out.println(a));
    }
}
```


Exercício

Criar uma classe com o nome Pessoa com três atributos: cpf, nome e naturalidade. Criar uma nova classe com o nome TestePessoa. Inserir Quatro objetos do tipo pessoa com todos os atributos passados pelo construtor. Em um desses objetos inserir o mesmo cpf e cidade com o nome diferente.

```
public class Pessoa {
    private String cpf;
    private String nome;
    private String naturalidade;

    public Pessoa(String cpf, String nome, String naturalidade) {
        this.cpf = cpf;
        this.nome = nome;
        this.naturalidade = naturalidade;
    }

    public String getNome() {
        return nome;
    }

    @Override
    public int hashCode() {
        final int prime = 31;
        int result = 1;
        result = prime * result + ((cpf == null) ? 0 : cpf.hashCode());
        return result;
    }

    @Override
    public boolean equals(Object obj) {
        if (this == obj)
            return true;
        if (obj == null)
            return false;
        if (getClass() != obj.getClass())
            return false;
        Pessoa other = (Pessoa) obj;
        if (cpf == null) {
            if (other.cpf != null)
                return false;
        } else if (!cpf.equals(other.cpf))
            return false;
        return true;
    }
}
```

```
public class TestePessoa {

    public static void main(String[] args) {
        Pessoa pessoa1 = new Pessoa("123.456.789-00", "Marcos", "Petrópolis");
        Pessoa pessoa2 = new Pessoa("233.556.658-23", "Ana", "Teresópolis");
        Pessoa pessoa3 = new Pessoa("452.216.345-12", "Carlos", "Juiz de Fora");
        Pessoa pessoa4 = new Pessoa("123.456.789-00", "Sandro", "Petrópolis");

        List<Pessoa> lista = Arrays.asList(pessoa1, pessoa2, pessoa3, pessoa4);

        Stream<Pessoa> stream = lista.stream().distinct().filter(p -> p.getNaturalidade().equals("Petrópolis"));

        stream.forEach(p -> System.out.println(p.getNome()));

        // Forma resumida
        System.out.println("-----");
        lista.stream()
            .distinct()
            .filter(p -> p.getNaturalidade().equals("Petrópolis"))
            .forEach(p -> System.out.println(p.getNome()));
    }
}
```

```
Console ✕
<terminated> TestePessoa [Java Application] C:\Program Files\Jav
Marcos
-----
Marcos
```

Métodos do Streams

Map

Permite que sejam realizadas transformações nos elementos da lista não sendo modificada a lista original. O map é uma operação intermediária porque ele retorna uma **Stream**.

```
public class Java8Map {  
  
    public static void main(String[] args) {  
        List<Integer> lista = Arrays.asList(2,6,3,4,1,5,7);  
  
        lista.stream()  
            .map(a -> a + 3)  
            .forEach(a -> System.out.println(a));  
    }  
}
```

Outro exemplo de uma lista de String transformando para Double.

```
public class Java8Map2 {  
  
    public static void main(String[] args) {  
        List<String> numString = Arrays.asList("3.0", "6.5", "9.8", "3.5", "6.2");  
  
        List<Double> numeros = numString.stream()  
            .map(e -> new Double(e))  
            .collect(Collectors.toList());  
  
        numeros.forEach(e -> System.out.println(e));  
    }  
}
```

Transforma o Stream em uma lista de Double

Operações no Stream

No Stream temos operações intermediárias onde utilizamos os métodos do tipo limit, skip, filter, map que retornam um Stream e temos também operações terminais utilizando métodos como count, min, max, allMatch, anyMatch, noneMatch que retornam um valor ou um objeto.

```
public class Java8Map2 {  
    public static void main(String[] args) {  
        List<String> numString = Arrays.asList("3.0", "6.5", "9.8", "3.5", "6.2");  
  
        Long quant = numString.stream()  
            .map(e -> new Double(e))  
            .count();  
        System.out.println(quant);  
    }  
}
```


Operações no Stream

Outro exemplo criando a classe **Funcionario**

```
public class Funcionario {
    private String nome;
    private String setor;
    private Double salario;

    public Funcionario(String nome, String setor, Double salario) {
        super();
        this.nome = nome;
        this.setor = setor;
        this.salario = salario;
    }

    String getNome() {
        return nome;
    }

    String getSetor() {
        return setor;
    }

    Double getSalario() {
        return salario;
    }
}
```

```
import java.util.Arrays;
import java.util.List;

public class TesteFuncionario {

    public static void main(String[] args) {
        Funcionario funcionario1 = new Funcionario("Gilberto", "CPD", 5500.);
        Funcionario funcionario2 = new Funcionario("Sandra", "ADM", 2200.);
        Funcionario funcionario3 = new Funcionario("Paulo", "CPD", 2750.);
        Funcionario funcionario4 = new Funcionario("Carla", "RH", 6800.);

        List<Funcionario> lista = Arrays.asList(funcionario1, funcionario2, funcionario3, funcionario4);

        Boolean resultado = lista.stream()
            .allMatch(f -> f.getSetor().equals("ADM"));
        System.out.println(resultado);
    }
}
```

Substituir o método acima para as outras opções abaixo para ver o resultado:

allMatch - Retorna true se todos funcionários forem do setor ADM

anyMatch - Retorna true se qualquer funcionários forem do setor ADM

noneMatch - Retorna true se nenhum funcionário for do setor ADM

Operações no Stream

Criar uma rotina com os seguintes critérios:

- Filtrar os clientes pelo setor CPD .
- Ordenar de forma ascendente pelo salário .
- Limitar o resultado a três funcionários.
- Armazenar o resultado em uma lista de String com o nome dos funcionários.

Ordenação Ascendente

```
List<String> nomes = lista.stream()
    .filter(f -> f.getSetor().equals("CPD"))
    .sorted((f1, f2) -> f1.getSalario().compareTo(f2.getSalario()))
    .limit(3)
    .map(Funcionario::getNome).collect(Collectors.toList());

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

Ordenação Descendente

```
List<String> nomesInv = lista.stream()
    .filter(f -> f.getSetor().equals("CPD"))
    .sorted(Comparator.comparing(Funcionario::getSalario).reversed())
    .limit(3)
    .map(Funcionario::getNome).collect(Collectors.toList());

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

```

public class TesteFuncionario2 {

    public static void main(String[] args) {
        Funcionario f1 = new Funcionario(1, "Maria", "RH", 2900.0);
        Funcionario f2 = new Funcionario(2, "Joana", "Compras", 3400.5);
        Funcionario f3 = new Funcionario(3, "Carla", "RH", 2900.0);
        Funcionario f4 = new Funcionario(4, "Ana Maria", "CPD", 1800.0);
        Funcionario f5 = new Funcionario(5, "Roberto", "DIRETORIA", 9200.0);
        Funcionario f6 = new Funcionario(6, "Carlos", "CPD", 6890.0);

        List<Funcionario> funcionarios = Arrays.asList(f1,f2,f3,f4,f5,f6);

        /*Com lambda
        Comparator<Funcionario> compareNome = Comparator.comparing( f->f.getNome());
        */
        //Ordenando pelo nome e setor decrescente
        Comparator<Funcionario> compareNome = Comparator.comparing( Funcionario::getNome).thenComparing(Funcionario::getSetor).reversed();

        funcionarios.sort(compareNome);
        System.out.println("-----Lambda-----");
        funcionarios.forEach(f->System.out.println(f));
        System.out.println("-----Method Reference-----");
        funcionarios.forEach(System.out::println);
    }
}

```

Listar todos funcionarios com salario maior ou igual a 3000

```
public class TesteFuncionario3 {  
    public static void main(String[] args) {  
        Funcionario f1 = new Funcionario(1, "Maria", "RH", 2900.0);  
        Funcionario f2 = new Funcionario(2, "Joana", "Compras", 3400.5);  
        Funcionario f3 = new Funcionario(3, "Carla", "RH", 2900.0);  
        Funcionario f4 = new Funcionario(4, "Ana Maria", "CPD", 1800.0);  
        Funcionario f5 = new Funcionario(5, "Roberto", "DIRETORIA", 9200.0);  
        Funcionario f6 = new Funcionario(6, "Carlos", "CPD", 6890.0);  
  
        List<Funcionario> funcionarios = Arrays.asList(f1,f2,f3,f4,f5,f6);  
  
        funcionarios.stream().filter(f -> f.getSalario() > 3000.).forEach(f->System.out.println(f));  
    }  
}
```

Pegar os nomes dos funcionarios e passar para uma lista

```
System.out.println("-----Lambda-----");  
List<String> nomes = funcionarios.stream().map(f-> f.getNome()).collect(Collectors.toList());  
  
System.out.println("-----Method Reference-----");  
List<String> nomes2 = funcionarios.stream().map(Funcionario::getNome).collect(Collectors.toList());
```


Quantidade de funcionários com salário acima de 3000

```
Long quantFuncionarios = funcionarios.stream().filter(f -> f.getSalario() >=3000).count();
```

```
System.out.println("O total é:" + quantFuncionarios);|
```

```
System.out.println("-----Maior Salário-----");
```

```
Optional<Double> maiorSalario = funcionarios.stream().map(Funcionario::getSalario).max(Comparator.naturalOrder());
```

```
System.out.println("Maior salário:"+maiorSalario);
```

```
System.out.println("-----Menor Salário-----");
```

```
Optional<Double> menorSalario = funcionarios.stream().map(Funcionario::getSalario).min(Comparator.naturalOrder());
```

```
System.out.println("Maior salário:"+menorSalario);
```

Média de salários, total de salários e maior salário

```
System.out.println("-----Total Pago-----");
```

```
Double soma = funcionarios.stream().mapToDouble( f -> f.getSalario()).sum();
```

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

```
System.out.println("-----Média Salarial-----");
```

```
Double media = funcionarios.stream().mapToDouble( f -> f.getSalario()).average().orElse(0);
```

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

```
System.out.println("-----Maior Salário-----");
```

```
Double maior = funcionarios.stream().mapToDouble( f -> f.getSalario()).max().orElse(0);
```

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

Criar uma mapa de funcionários por departamento.

```
List<Funcionario> funcionarios = Arrays.asList(f1,f2,f3,f4,f5,f6);

Map<String, String> funcSetores = funcionarios.stream().collect(Collectors.toMap(Funcionario::getNome, Funcionario::getSetor));
funcSetores.forEach((k,v)-> System.out.println(k+ "-" +v));

System.out.println("-----Imprimindo junto-----");
funcionarios.stream().collect(Collectors.toMap(f -> f.getNome(), f-> f.getSetor())).forEach((k,v) -> System.out.println(k + "-" +v));
```

Optional

A classe Optional evita que ocorram exceções do tipo NullPointerException como consequência temos um código menor e mais limpo. Vamos criar um exemplo para utilização do métodos da classe Optional.

No exemplo abaixo o método buscar retorna null e estamos tentando acessar um método do objeto que está nulo, com isto teremos um **NullPointerException**

```
public class Funcionario {
    private Integer codigo;
    private String nome;
    private String setor;
    private Double salario;

    public Funcionario() {
        // TODO Auto-generated constructor stub
    }

    public Funcionario(Integer codigo, String nome, String setor, Double salario) {
        super();
        this.codigo = codigo;
        this.nome = nome;
        this.setor = setor;
        this.salario = salario;
    }

    @Override
    public String toString() {
        return "Funcionario [codigo=" + codigo + ", nome=" + nome + ", setor=" + setor + ", salario=" + salario + "];"
    }

    public Integer getCodigo() {
        return codigo;
    }

    public String getNome() {
        return nome;
    }

    public String getSetor() {
        return setor;
    }

    public Double getSalario() {
        return salario;
    }
}
```

```
public class FuncionarioDao {
    public FuncionarioDao() {
        Connection connection = ConnectionFactorySingleton.getConnection();
    }

    public Funcionario buscar(String nome) {
        return null;
    }
}
```

```
public class TesteFuncionario {

    public static void main(String[] args) {
        FuncionarioDao dao = new FuncionarioDao();

        Funcionario funcionario = dao.buscar("João");
        System.out.println(funcionario.getNome());
    }
}
```

```
Conectado com sucesso !
Exception in thread "main" java.lang.NullPointerException
    at org.serratec.persistence.TesteFuncionario.main(TesteFuncionario.java:11)
```

Optional

Uma das formas de resolver o problema é inserindo um if para testar se o valor retornado é nulo.

```
if(funcionario != null) {  
    System.out.println(funcionario.getNome());  
}
```

A partir do Java 8 para representar objetos nulos podemos retornar um Optional, vamos alterar as classe para uso do **Optional**

```
public class FuncionarioDao {  
    public FuncionarioDao() {  
        Connection connection = ConnectionFactorySingleton.getConnection();  
    }  
  
    public Optional<Funcionario> buscar(String nome) {  
        return Optional.ofNullable(null);  
    }  
}
```

O método **ofNullable** aceita receber um nulo diferente do método **of** que lançará uma exception quando um valor for nulo.

Dessa forma não temos mais a exceção do tipo **NullPointerException**

```
public static void main(String[] args) {  
    FuncionarioDao dao = new FuncionarioDao();  
  
    Optional<Funcionario> funcionario = dao.buscar("João");  
  
    if(funcionario.isPresent()) {  
        System.out.println(funcionario.get().getNome());  
    }  
}
```

Outra forma de representar é utilizar o **ifPresent** no lugar do **isPresent**

```
funcionario.ifPresent(f -> System.out.println(f));
```


Optional

Podemos usar também os métodos `orElse` e `orElseThrow`.

Vamos adicionar mais um método na classe `FuncionarioDao`

```
public class FuncionarioDao {  
    public FuncionarioDao() {  
        Connection connection = ConnectionFactorySingleton.getConnection();  
    }  
  
    public Optional<Funcionario> buscar(String nome) {  
        return Optional.ofNullable(null);  
    }  
  
    public Optional<Funcionario> listarFuncionario() {  
        return Optional.ofNullable(null);  
    }  
}
```

Adicionar no main

```
Funcionario funcionario2 = dao.listarFuncionario().orElse(new Funcionario(1,"Roni","CPD",5500.0));
```

```
System.out.println(funcionario2);
```

```
Funcionario funcionario3 = dao.listarFuncionario().orElseThrow( () -> new NullPointerException("Valor nulo !") );
```

```
System.out.println(funcionario2);
```

Optional

A classe Optional evita que ocorram exceções do tipo NullPointerException como consequência temos um código menor e mais limpo. Vamos criar um exemplo para utilização do métodos da classe Optional.

```
public class Carro {
    private String marca;
    private String modelo;
    private Seguro seguro;

    public Carro(String marca, String modelo, Seguro seguro) {
        super();
        this.marca = marca;
        this.modelo = modelo;
        this.seguro = seguro;
    }

    @Override
    public String toString() {
        return marca + " " + modelo + " " + seguro;
    }
}

public class Seguro {
    private String cobertura;
    private Double valorSeguro;

    public Seguro(String cobertura, Double valorSeguro) {
        super();
        this.cobertura = cobertura;
        this.valorSeguro = valorSeguro;
    }

    @Override
    public String toString() {
        return cobertura + " " + valorSeguro;
    }
}
```

```
public class TesteCarro {

    public static void main(String[] args) {
        Seguro seguro1 = new Seguro("Total", 3000.);
        Seguro seguro2 = new Seguro("Parcial", 2500.);

        Carro carro1 = new Carro("Renault", "Sandero", seguro1);
        Carro carro2 = new Carro("Renault", "Sandero", null);

        System.out.println(carro1);
        System.out.println(carro2);
    }
}
```

Neste exemplo ao executarmos o código uma exceção pode ser gerada caso um valor seja passado como nulo como argumento, desta forma temos que tratar o erro com if (seguro != null)

Optional

O atributo **seguro** foi alterado na classe **Carro** para o tipo **Optional** que irá disponibilizar métodos para evitarmos exceções geradas pelo compilador.

```
import java.util.Optional;

public class Carro {
    private String marca;
    private String modelo;
    private Optional<Seguro> seguro;
```

Na classe de teste foram inseridos alguns métodos utilizando o **Optional**

```
import java.util.Optional;

public class TesteCarro {

    public static void main(String[] args) {
        Seguro seguro1 = new Seguro("Total", 3000.);
        Seguro seguro2 = new Seguro("Parcial", 2500.);

        Carro carro1 = new Carro("Renault", "Sandero", Optional.of(seguro1));
        Carro carro2 = new Carro("Renault", "Sandero", Optional.ofNullable(null));

        Optional<Double> valorSeguro = carro1.getSeguro().map(Seguro::getValorSeguro);

        if (valorSeguro.isPresent()) {
            System.out.println(carro1);
        }

        String cobertura = carro2.getSeguro().map(Seguro::getCobertura).orElse("Não tem Seguro");

        System.out.println(carro2);
        System.out.println(cobertura);
    }
}
```

Métodos of e ofNullable - Se temos certeza que receberemos um objeto que nunca será null usamos o método **of**, caso contrário utilizamos o método **ofNullable** que retornará um vazio caso o objeto seja null. Se objeto passado for nulo o método **of** lançará uma exceção.

Podemos utilizar o método **map** para gerar um novo **Optional**. No exemplo ao lado foi gerado um novo **Optional** do tipo **Double**.

O método **orElse** retorna uma **String** caso o carro não tenha um objeto do tipo seguro associado.