Palavra reservada - final

A palavra final tem algumas maneiras de uso:

- Uma classe final nao pode ter subclasses
- Um metodo final nao pode ser sobrescrito pelas subclasses
- Uma variavel final so podera ser inicializada uma unica vez

Set

- Elementos só podem serem exibidos uma única vez
- Contém somente métodos herdados da interface Collection
- Adiciona restricão que proibe elementos duplicados

Set - Implementações

- HashSet - Esta apoiado por um HashMap. E não garante a sequencia dos elementos na sua iteracao.
- LinkedHashSet - Diferencia do HashSet, porque garante que a ordem dos elemtentos em sua iteracao seja a mesma ordem que eles foram inseridos, e inserir novamente um elemento a sua ordem nao e alterada.
- TreeSet - Armazena seus elementos em uma árvore rubro-negra(Uma árvore rubro-negra é um tipo especial de árvore binária, usada em ciência da computação para organizar dados que possam ser comparáveis.), ordena seus elementos baseados em seus valores. é mais lento que o HashSet

Set - Exemplos

```
Set setA = new HashSet();

setA.add("element 0");
setA.add("element 1");
setA.add("element 2");

//access via Iterator
Iterator iterator = setA.iterator();
while(iterator.hasNext(){
  String element = (String) iterator.next();
}


//access via new for-loop
for(Object object : setA) {
    String element = (String) object;
}
```

StackTrace criando a classe util

```
public class ExceptionUtil {
    public static String getStackTrace(NegocioException e
        StringWriter sw = new StringWriter();
        PrintWriter pw = new PrintWriter(sw);
        e.printStackTrace(pw);
        return sw.toString();
    }
}
```

StackTrace criando a nossa classe personalizada

```
class NegocioException extends RuntimeException {
    public NegocioException(String msg) {
        super(msg);
    }
}
```

StackTrace criando a nossa classe de testes

```
public class PrintStackTrace {

    public static void main(String[] args) {
        try {
            testException();
        } catch(NegocioException e) {
            System.out.println("##### "
            + ExceptionUtil.getStackTrace(e)
            + " ####");
        }

    }

    public static void testException() {
        throw new NegocioException("Deu erro");
    }
}
```

Java 8 - Maior mudanca desde o java 5

- Linguagem
- Compilador
- Bibliotecas
- Ferramentas
- Runtime (JVM)

Lambda(->) e Interfaces Funcionais

Lambda (tambem conhecido como closures) e a maior e mudanca mais esperada em toda release do Java 8.
Muitas linguagens na plataforma java (Groovy, Scala, Clojure) ja possuem lambda desde o dia de sua criacao, mais java tinha escolhido tratar lambda como classes anonimas.
Lambda e representado por uma lista de parametros, o simbolo -> e o corpo

Lambda - Exemplo

```
Arrays
  .asList( "a", "b", "d" )
  .forEach( e -> System.out.println( e ) );
```

Lambda - Exemplo Complexo

```
Arrays.asList( "a", "b", "d" ).forEach( e -> {
    System.out.print( e );
    System.out.print( e );
} );
```

Lambda - Exemplo Complexo

Lambdas podem referencias os membros de classes e as variaveis locais(implicitamente final). Estes dois exemplos sao equivalentes:

```
String separator = ",";
Arrays
  .asList( "a", "b", "d" )
  .forEach(( String e ) ->
                System.out.print( e + separator ) );
```

```
final String separator = ",";
Arrays
  .asList( "a", "b", "d" )
  .forEach(( String e ) ->
                System.out.print( e + separator ) );
```

Lambdas - Retornos

Lambda pode retornar um valor, O tipo do valor do retorno vai ser inferido pelo compilador. O return nao e requerido se o corpo do lambda for apenas uma linha. Os dois exemplos abaixos sao equivalentes:

```
Arrays
  .asList( "a", "b", "d" )
  .sort( ( e1, e2 ) -> e1.compareTo( e2 ) );
```

```
Arrays
  .asList( "a", "b", "d" )
  .sort( ( e1, e2 ) -> {
    int result = e1.compareTo( e2 );
    return result;
  });
```

Interface funcional

- Sao interfaces que possuem apenas um metodo.
- Podem ser implicitamente convertidos para um expressao lambda
- E representada pela anotacao @FunctionalInterface

Exemplo:

```
@FunctionalInterface
public interface Functional {
    void method();
}
```

Interface funcional - default metodos

Metodos default e estaticos nao quebram o contrato funcional e podem ser declarados:

Exemplo:

```
@FunctionalInterface
public interface FunctionalDefaultMethods {
    void method();
    default void defaultMethod() {
    }
}
```

Optional

O famoso NullPointerException e de longe a causa mais famosa de aplicacoes java falharem.

O projeto guava, criou o Optionals como uma solucao para NullPointerExceptions, permitindo desenvolvedores a evitar checagem de nulos e escreverem um codigo mais limpo.

Inspirado pelo guava Optional agora e parte do java 8.

```
Optional<String> nome = Optional.ofNullable(null);

System.out.println("Existe um nome? " + nome.isPresent())

System.out.println(
        "Nome: " + nome.orElseGet(() -> "[sem nome]"));

System.out.println(
        nome
      .map( s -> "Olá " + s + "!" )
      .orElse( "Olá estranho!" ) );
```

Streams

- java.util.stream - introduz um real estilo de programacao funcional em java.
- Tem como objetivo fazer os desenvolvedores java serem mais produtivos e escreverem codigos mais limpos e concisos.
- A API Stream permite que o processamento de colecoes seja simplificado
- Sao divididos entre operacoes intermediarias e finais
- Operacoes intermediarias como filter retornam um novo stream, sempre sao lazy.
- Operacoes finais como forEach ou sum, produzem um resultado. Depois da operacao final a pipeline do stream e considerado consumido e nao pode mais ser usada.

Streams - Exemplo

```
public class Streams  {
    private enum Status { OPEN, CLOSED };

        private static final class Task {
        private final Status status;
        private final Integer points;

                Task(final Status status,
                 final Integer points) {

            this.status = status;
            this.points = points;
        }
        public Integer getPoints() { return points; }
        public Status getStatus() { return status; }
        @Override
        public String toString() {
            return
              String.format( "[%s, %d]", status, points);
        }
    }
}
```

Streams - Exemplo

```
final Collection< Task > tasks = Arrays.asList(
    new Task( Status.OPEN, 5 ),
    new Task( Status.OPEN, 13 ),
    new Task( Status.CLOSED, 8 )
);
```

Streams - Exemplo

```
final long totalPointsOfOpenTasks = tasks
    .stream()
    .filter( task -> task.getStatus() == Status.OPEN )
    .mapToInt( Task::getPoints )
    .sum();
System.out.println( "Total de pontos: "
                        + totalPointsOfOpenTasks );
```

Streams - Processamento paralelo

```
final double totalPoints = tasks
   .stream()
   .parallel()
   .map( task -> task.getPoints() )
   // ou map( Task::getPoints )
   .reduce( 0, Integer::sum );
System.out.println( "Total de pontos (todas as tarefas): "
```

Streams - Agrupamentos

```
//agrupamentos por status
final Map< Status, List< Task > > map = tasks
    .stream()
    .collect( Collectors.groupingBy( Task::getStatus ) );
System.out.println( map );
```

Streams - Calculando

```
final Collection< String > result = tasks
    .stream()                                          // S
    .mapToInt( Task::getPoints )                       // I
    .asLongStream()                                    // L
    .mapToDouble( points -> points / totalPoints )     // D
    .boxed()                                           // S
    .mapToLong( weigth -> ( long )( weigth * 100 ) )   // L
    .mapToObj( percentage -> percentage + "%" )        // S
    .collect( Collectors.toList() );                   // L

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

Method References

Method references proveem uma sintaxe para referenciar diretamente a saida dos metodos ou construtores das classes java ou instancia de objetos.

Method References - Exemplo - parte 1

```
public static class Car {
    public static Car create(
                final Supplier< Car > supplier ) {
        return supplier.get();
    }

    public static void collide( final Car car ) {
        System.out.println("Collided " + car.toString());
    }

    public void follow( final Car another) {
        System.out.println(
                "Following the " + another.toString() );
    }

    public void repair() {
        System.out.println( "Repaired " + this.toString()
    }
}
```

Method References - Exemplo - parte 2

```
final Car car = Car.create( Car::new );
final List< Car > cars = Arrays.asList( car );

cars.forEach( Car::collide );
cars.forEach( Car::repair );
```

```
// Instancia
final Car police = Car.create( Car::new );
cars.forEach( police::follow );
```

Method References - Exercicio

- Criar uma lista de inteiros que vai de 1 a 100;
- Multiplicar todos os elementos por 3
- Filtrar todos os elementos divisiveis por 5
- Buscar a soma de todos os elementos

Proxima Aula

- Manipulacao de arquivos
- Threads
- JDBC