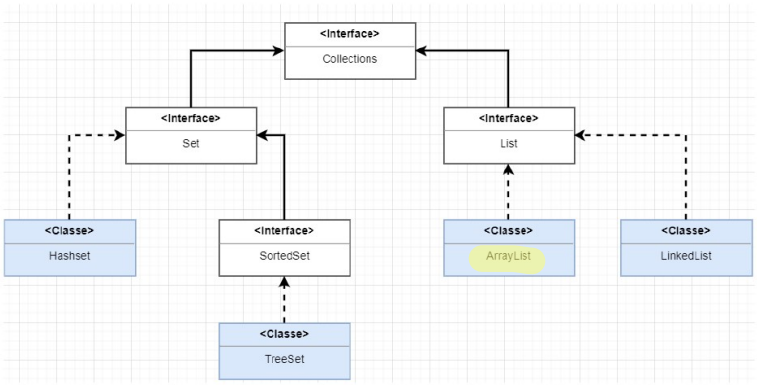
java.util



java.util.List (interface)

Características:

- Adicionar elementos em um índice específico
- Obter um elemento a partir do índice
- Remover um elemento a partir do índice
- Substituir um elemento a partir do índice
- Obter o tamanho total da lista (quantidade de elemento)
- Permite elemento duplicados

Implementações de List: LinkedList

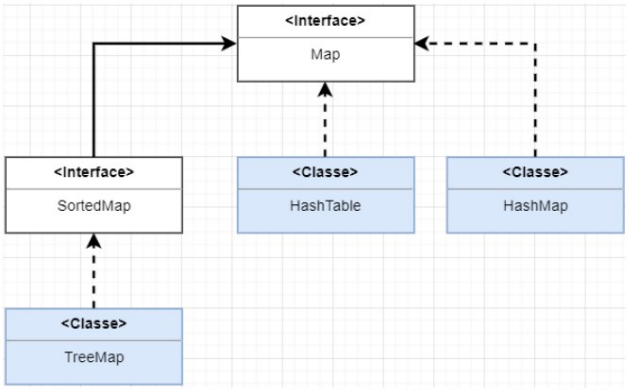
- Funciona exatamente como a **ArrayList**, com a diferença de internamente gerenciar uma lista duplamente ligada

java.util.Set

Características:

- Os elementos não se repetem
- Não possui índice
- Você deve percorrer os elementos para acessá-los
- A classe **TreeSet** fornece uma implementação de ordenação automática

java.util.Map



Características:

- Armazena elementos em pares: Chave – Valor
- A chave funciona como um identificador único do valor
- O Valor pode ser acessado a partir da Chave
- É conhecido como "associative arrays" em outras linguagens

Implementação de Map: HashMap

- Não admite chaves repetidas
- Substitui o valor quando adicionado na mesma chave

Implementação de Map: TreeMap

- Possui o mesmo comportamento de HashMap
- Ordena automaticamente as chaves adicionadas

Características:

- Um tipo de Collection que pode ser usada como **pilha** ou **fila**
  - Métodos para fila (FIFO – First In, First Out)
    `add(e)` e `remove()` → FILA
  - Métodos para pilha (LIFO – Last In, First Out)
    `push(e)` e `pop()` → PILHA

Implementação de Deque: ArrayDeque

Dica: Funcionamento Deque

O método `add(e)` adiciona o elemento no final da lista e o método `push(e)` no início. Os métodos `remove()` e `pop()` sempre retiram o elemento com índice 0 (zero).

```
1:  public class Dados {
2:
3:     private List<String> nomes = new ArrayList<>();
4:     static private List<Integer> numeros = new ArrayList<>();
5:
6:     static {
7:        numeros.add(100); numeros.add(200); numeros.add(300);
8:        numeros.add(400); numeros.add(500); numeros.add(600);
9:        System.out.println("executou: bloco de inicialização estático!");
10:    }
11:    {
12:       nomes.add("Paulo"); nomes.add("Marcelo"); nomes.add("Mônica");
13:       nomes.add("Thiago"); nomes.add("Fernando"); nomes.add("Cristina");
14:       System.out.println("executou: bloco de inicialização!");
15:    }
16:
17:    public Dados() {
18:       System.out.println("executou: construtor!");
19:       System.out.printf("Lista de nomes: %s \n", nomes);
20:       System.out.printf("Lista de números: %s \n", numeros);
21:    }
22: }
23: /* Output:
24:    executou: bloco de inicialização estático!
25:    executou: bloco de inicialização!
26:    executou: construtor!
27:    Lista de nomes: [Paulo, Marcelo, Mônica, Thiago, Fernando, Cristina]
28:    Lista de números: [100, 200, 300, 400, 500, 600]
29: */
```

Collections:

- TreeSet => Ordena os valores
- TreeMap => Ordena a chave

ArrayList ?

- Collections.sort(arrayList);

```
new Funcionario();
new Funcionario(String nome, Integer idade, String cidade, String estado, Double salario)
new Funcionario(String nome, String cidade, String estado, Double salario)
```

1 - só com o salário
2 - idade e salario
3 - idade, cidade e estado
4 - em qualquer ordem

Builder Pattern:

1. Declarar os atributos da classe como **private final**;
2. Declarar uma classe estática com os mesmos atributos da classe **top level** (sem o final);
3. Na classe estática, criar os métodos para receber os dados necessários para instanciação do objeto. Os métodos devem ter o mesmo nome dos atributos e retornar o próprio **Builder** (**Builder.this**);
4. Declarar na classe estática um método `build()` que retorna uma instancia da classe **top level**. Passe como argumento para o construtor da classe **top level** o **Builder**;
5. Declarar na classe **top level** um **construtor privado** que recebe como argumento o **Builder** (ele é usado no item 4). Use os dados recebidos para setar os campos do objeto;
6. Declarar na classe Top Level os métodos **get** (getters).

```
1:  public class Funcionario {
2:
3:      private static int ultimoCodigo;
4:
5:      private final Integer codigo;
6:      private final String nome;
7:      private final Double salario;
8:
9:      public static class Builder {
10:         private String nome;
11:         private Double salario;
12:
13:         public Funcionario.Builder nome(String nome) {
14:             this.nome = nome;
15:             return Builder.this;
16:         }
17:
18:         public Funcionario.Builder salario(Double salario) {
19:             this.salario = salario;
20:             return Builder.this;
21:         }
22:         public Funcionario build() {
23:             return new Funcionario(Builder.this);
24:         }
25:     }
26:
27:     private Funcionario(Funcionario.Builder builder) {
28:         this.codigo = ++ultimoCodigo;
29:         this.nome = builder.nome;
30:         this.salario = builder.salario;
31:     }
32:     public Integer getCodigo() { return codigo; }
33:     public String getNome() { return nome; }
34:     public Double getSalario() { return salario; }
35: }
```

1
2
3
4
5

Pipeline

A partir do Java 8, foram introduzidos na linguagem uma série de métodos para gerencias Collections. Dentre suas funcionalidades, podemos destacar:

- filtro
- busca
- agrupamento
- soma, média, máximo e mínimo
- ordenação
- etc.

Pipeline é o nome dado a junção desses métodos às listas de forma encadeada para obter um resultado. Um **Pipeline** é composto de:

1. **Souce**: Uma Collection
2. **Zero ou mais métodos intermediários**: filter, map, reduce...
3. **Um método finalizador**: forEach, sum, max, min...

```
1:  public class TesteFuncionario02 {
2:
3:     public static void main(String[] args) {
4:        List<Funcionario> funcionarios = BDFuncionario.getFuncionarios();
5:
6:        funcionarios.stream()
7:           .filter(f -> f.getEstado().equals("MG"))
8:           .sorted((f1, f2) -> f1.getIdade().compareTo(f2.getIdade()))
9:           .forEach(f -> System.out.printf("%s\n\n", f));
10:    }
11: }
```

ArrayList

```
1:  public class TesteStream01 {
2:
3:     public static void main(String[] args) {
4:
5:        Stream<String> vazia = Stream.empty();
6:        System.out.println(vazia.count()); //0
7:
8:        Stream<Integer> unico = Stream.of(1);
9:        System.out.println(unico.count()); //1
10:
11:       Stream<Integer> deArray = Stream.of(1, 2, 3);
12:       System.out.println(deArray.count()); //3
13:
14:    }
15:
16: }
```

```
1:  public class TesteStream02 {
2:
3:     public static void main(String[] args) {
4:
5:        Stream<Double> randomns = Stream.generate(() -> Math.random());
6:
7:        Stream<Integer> numeros = Stream.iterate(1,  n -> n + 2);
8:        numeros.filter(e -> e <= 100)
9:           .forEach(e -> System.out.println(e));
10:    }
11:
12: }
```