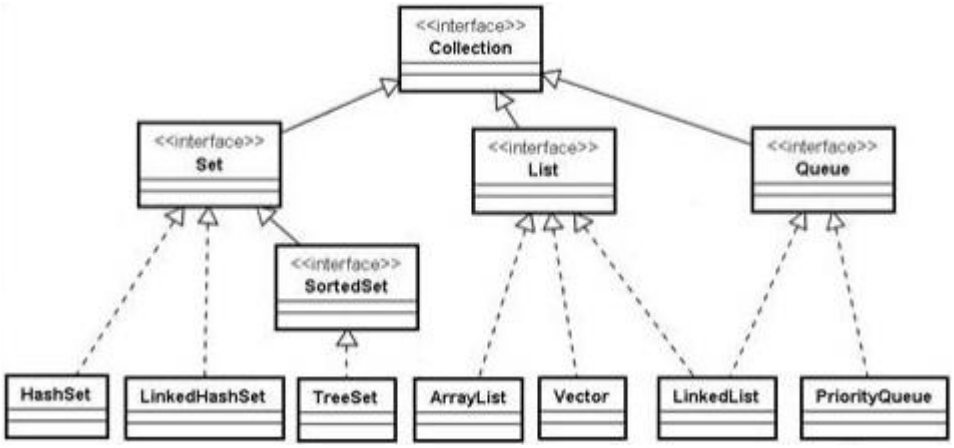
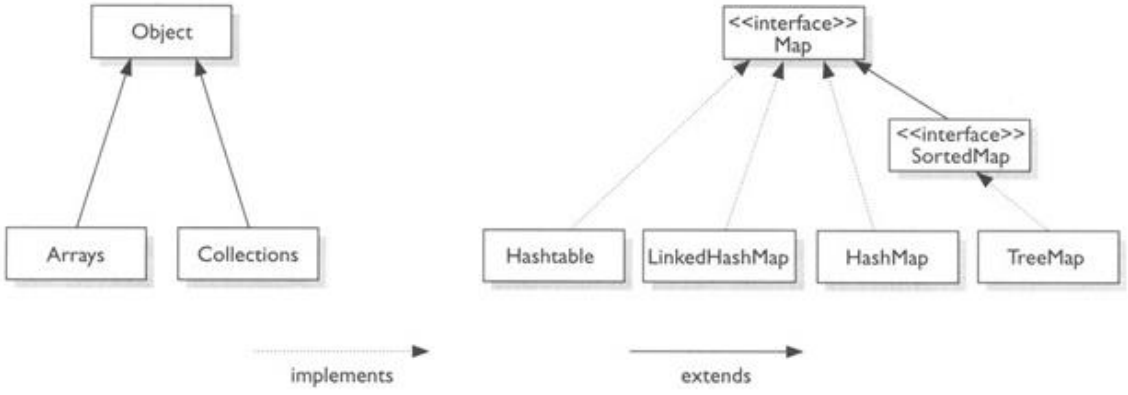
ÍNDICE

1. COLEÇÕES EM JAVA

Coleções é uma das melhores facilidades desenvolvidas na linguagem JAVA. Segue o diagrama de classes da UML que representa toda biblioteca de coleções em java.

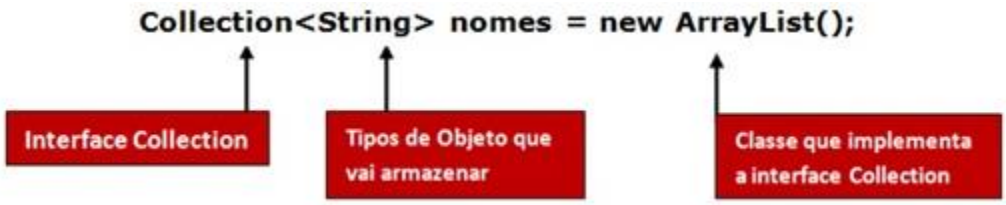


Collections assim como Arrays extendem diretamente a classe Object. É interessante notar um outro grupo de Classes que são Implementam mapas (Map).



1.1. Declaração e Instanciação

Veja como declarar e criar instâncias de *collections* em java.



Nesse exemplo estamos instanciando a classe que implementa a interface Collection. Entretanto, vale lembrar que para declarar uma Collection, o pacote **java.util.Collection** deverá ser importado. Da mesma forma, como estamos instanciando um ArrayList(), então deveremos ainda importar o pacote **java.util.ArrayList**.

1.2. Como inserir elementos

Para adicionar o elemento é invocado o método `add()` que aceita um argumento do tipo definido, como mostrado abaixo.

```
package com.suam;

import java.util.Collection;
import java.util.ArrayList;

public class Exemplo {

	public static void main(String[] args) {

		Collection<String> nomes = new ArrayList<String>();

		nomes.add("João");
		nomes.add("Maria");
		nomes.add("Eduardo");
		nomes.add("Silvana");
		nomes.add("Mário");
		System.out.println("Lista de nomes: " + nomes);
	}

}
```

1.2.1. Método isEmpty()

Verifica se os dados dentro de um List estão vazios, tendo como retorno valor booleano, true ou false.

```
import java.util.Collection;
import java.util.ArrayList;

public class Exemplo {

	public static void main(String[] args) {

		Collection<String> letras = new ArrayList<String>();

		letras.add("A");
		letras.add("B");
		letras.add("C");
		letras.add("D");
		letras.add("E");
		letras.add("F");

		if (letras.isEmpty()) {
			System.out.println("Lista Vazia!");
		} else {
			System.out.println("Contém valores -> " + letras);
		}
	}
}
```

1.2.2. Removendo elementos

Para excluir uma ocorrência do valor especificado é usado o método **`remove()`**.

```
import java.util.Collection;
import java.util.ArrayList;

public class Exemplo {

      public static void main(String[] args) {

        Collection<Integer> fila = new ArrayList<Integer>();

        fila.add(255);
        fila.add(312);
        fila.add(883);
        fila.add(122);
        fila.add(9);

        System.out.println("Valores da fila: " + fila);

        fila.remove(312); // REMOVE OBJETO 312
        System.out.println("Valores atualizados da fila:" + fila);

      }
}
```

1.2.3. Limpando os elementos da lista

Para executar essa ação basta invocar o método **`clear()`** que irá limpar todos os elementos da coleção referenciada.

```
import java.util.Collection;
import java.util.ArrayList;

public class Exemplo {

	public static void main(String[] args) {

		Collection<String> livros = new ArrayList<String>();

		livros.add("Java");
		livros.add("Php");
		livros.add("Python");
		livros.add("SQL");

		System.out.println("Listagem dos Livros: " + livros);
		livros.clear();

		System.out.println("Listagem após o clear: " + livros);
	}

}
```

1.2.4. Verificando se existem dados

Quando precisar verificar a existência de certos elementos dentro de uma coleção, é possível fazer isso através do método **`contains()`**.

```
import java.util.Collection;
import java.util.ArrayList;

public class Exemplo {

  public static void main(String[] args) {

    Collection<String> vogais = new ArrayList<String>();

    vogais.add("A");
    vogais.add("E");
    vogais.add("I");
    vogais.add("O");
    vogais.add("U");

    System.out.println("Contém a vogal I ? "+vogais.contains("I"));
    System.out.println("Lista das vogais: " + vogais);
  }
}
```

No caso acima, o método `contains()` vai retornar ***true***.

1.2.5. Adicionando elementos de outra coleção

O método **`addAll()`** permite adicionar todo os elementos de uma lista no final de outra.

```
import java.util.ArrayList;
import java.util.List;

public class Exemplo {

	public static void main(String[] args) {

		List<String> vogais = new ArrayList<String>();
		vogais.add("A");
		vogais.add("E");
		vogais.add("I");

		List<String> vogais2 = new ArrayList<String>();
		vogais2.add("O");
		vogais2.add("U");

		vogais.addAll(vogais2);

		System.out.println("Lista das Vogais: " + vogais);
	}

}
```

1.2.6. Percorrendo elementos com FOREACH

Para percorrer os elementos de uma coleção de forma prática, usamos a instrução **`foreach`** que permite acessar cada item individualmente.

```
import java.util.ArrayList;
import java.util.List;

public class Exemplo {

	public static void main(String[] args) {

		List<String> vogais = new ArrayList<String>();
		vogais.add("A");
		vogais.add("E");
		vogais.add("I");
		vogais.add("O");
		vogais.add("U");

		for (String vog : vogais) {
			System.out.println("Vogal: " + vog);
		}
	}

}
```

O **`Foreach`** é um ciclo for, mas que é adaptado para utilização em Collections e outras listas. Ele serve para percorrer todos os elementos de qualquer Collection contida na API Collections.

1.2.7. Percorrendo elementos com ITERATOR

O iterator é uma interface disponível no pacote **java.util** que permite percorrer coleções da **API Collection**, desde que tenham implementado a **Collection**, fornecendo métodos como o **`next()`**, **`hasnext()`** e **`remove()`**.

```
import java.util.ArrayList;
import java.util.List;
import java.util.Iterator;

public class Exemplo {

	public static void main(String[] args) {

		List<String> vogais = new ArrayList<String>();
		vogais.add("A");
		vogais.add("E");
		vogais.add("I");
		vogais.add("O");
		vogais.add("U");


		Iterator<String> it = vogais.iterator();

		while(it.hasNext()){
			String vogal = (String) it.next();
			System.out.println(vogal);
		}

	}

}
```

Para iterar sobre a coleção, o usuário pode optar por utilizar foreach ou iterator.

1.3. Generics

O padrão das **Collections** é aceitar **Generics**, ou seja, aceitar qualquer tipo de elemento, inclusive elementos diferentes: Double, String, Integer, ou outro objeto que desejar.

Sendo assim, é possível criar uma lista de qualquer classe que desejar.

```
import java.util.ArrayList;
import java.util.List;
import com.suam.Pessoa;
import com.suam.Aluno;

public class Exemplo {

	public static void main(String[] args) {

		List<String> vogais = new ArrayList<String>();
		List<Pessoa> pessoas = new ArrayList<Pessoa>();
		List<Aluno> alunos = new ArrayList<Aluno>();
	}
```

Portanto, é possível especificar o tipo a ser aceito, da seguinte forma:

```
		List<Tipo> variavel = new ArrayList<Tipo>();
```

2. ENTENDENDO OS TIPOS DE COLEÇÕES EM JAVA

Set: Não aceita itens duplicados, aceita nulos, não possui índice, rápido para inserir e pesquisar elementos;

HashSet: esta é uma implementação concreta de **Set não organizada**, ou seja, os elementos são percorridos aleatoriamente, e também **não é ordenada**, não há regras de ordenação. Além disso, assim como Set, **não aceita itens duplicados**.

TreeSet: implementação concreta de **Set ordenada**, mas também **não aceita itens duplicados**.

```
import java.util.HashSet;
import java.util.Set;

public class Exemplo {

	public static void main(String[] args) {

		Set<Integer> numeros = new HashSet<Integer>();
		numeros.add(1);
		numeros.add(2);
		numeros.add(3);
		numeros.add(2);
		numeros.add(1);


		for (Integer numero : numeros) {
			System.out.println(numero);
		}

	}

}
```

Veja que como Set não aceita objetos repetidos, serão impressos na tela somente os valores 1,2 e 3.

List: **Aceita itens duplicados**, organizada e todas as implementações seguem este padrão, elementos percorridos por ordem ser inserção, não ordenada, **aceita nulo**, trabalha com **índices da mesma forma que os arrays**, é rápido para inserir elementos, mas um pouco mais lento que o Set, já as pesquisas são mais lentas que o Set.

ArrayList: implementação concreta de List, aceita itens duplicados e seu funcionamento **é semelhante a um array convencional, com a principal diferença de ser dinamica**, ou seja, pode crescer conforme a necessidade.

Vector: Outra implementação de List, pode ser visto como uma ArrayList, mas os métodos são sincronizados, ou seja, o acesso simultâneo por diversos processos será coordenado, é também mais lento que o ArrayList quando não há acesso simultâneo.

LinkedList: outra implementação de List, mas também implementa Queue, aceitando itens duplicado e sendo organizada. É similar ao ArrayList e ao Vector, mas fornece alguns métodos adicionais para a inserção, remoção e acesso aos elementos no inicio e no final da lista. Possui melhor performance do que o ArrayList e o Vector quando se trata de inserir, remover e acessar elementos no inicio ou no final da lista, mas se for precisar acessar algum elemento pelo índice, a performance é muito inferior, sendo lento para pesquisas.

Queue: é a fila, semelhante à lista, tendo como padrão o aceite duplicado de elementos, normalmente organizado, normalmente utilizado em itens que a ordem é importante, como uma fila de banco.

PriorityQueue: implementação concreta de Queue e não aceita nulos

Map: Um Map é um tipo de coleção que identifica os elementos por chaves, desta forma aceita itens duplicados com chaves diferentes.

HashMap: implementação concreta de Map, aceita itens duplicados com chaves diferentes, não organizado, ou seja, os elementos são percorridos aleatoriamente, não ordenada e aceita nulos.

HashTables: outra implementação de Map, aceita itens duplicados com chaves diferentes, semelhante ao HashMap mas os métodos são sincronizados.

TreeMap é uma outra implementação concreta de Map, também suporta itens duplicados com indices diferentes, é ordenado.