

Computação Orientada a Objetos

Coleções Java Parte II

Slides baseados em:

Deitel, H.M.; Deitel P.J. **Java: Como Programar**, Pearson
Prentice Hall, 6a Edição, 2005. **Capítulo 19**

Profa. Karina Valdivia Delgado
EACH-USP

Revisando:

O que é uma coleção?

- É **uma estrutura de dados** (um objeto) que agrupa referências a vários outros objetos.
- Usadas para armazenar, recuperar e manipular elementos que formam um grupo natural (normalmente objetos do mesmo tipo).

Interfaces da estrutura de coleções

*Coleções de
elementos individuais*

*java.util.**Collection***



*java.util.**Queue***

primeiro em
entrar, primeiro
em sair

*java.util.**List***

- *seqüência definida*
- *elementos indexados*

*java.util.**Set***

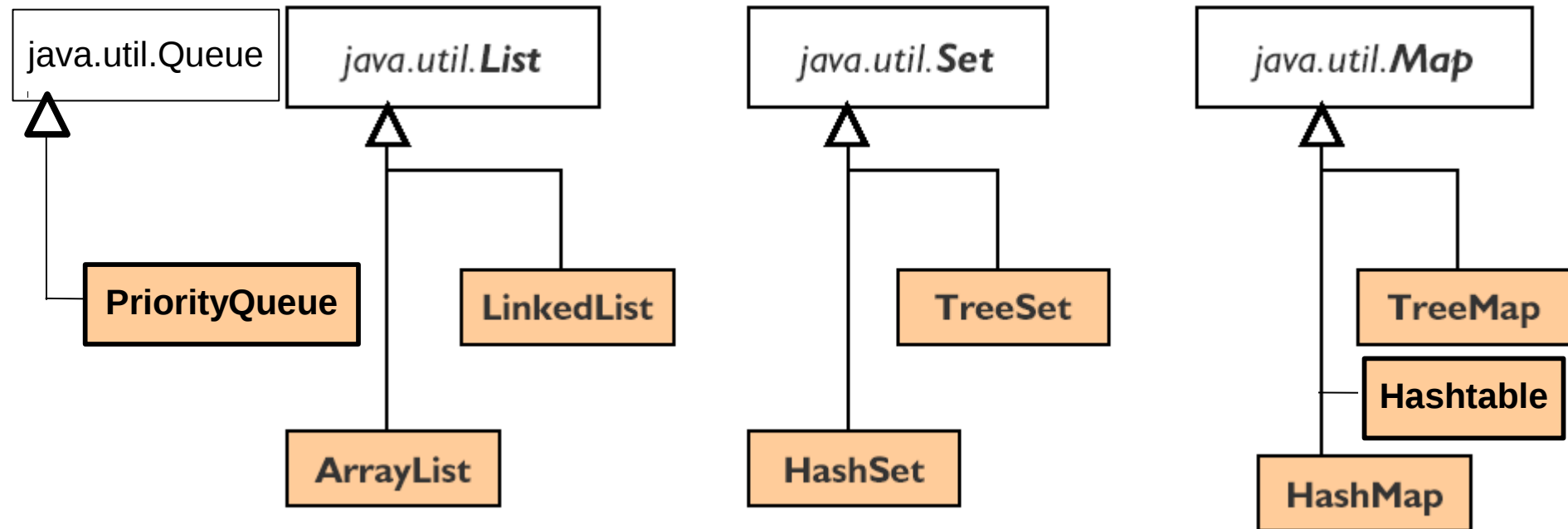
- *seqüência arbitrária*
- *elementos não repetem*

*Coleções de
pares de elementos*

*java.util.**Map***

- *Pares chave/valor*

Implementações da estrutura de coleções



Interface **Collection**

- Operações básicas:
 - adiciona elemento: **add (Object o)**
 - remove elemento: **remove (Object o)**
- Operações de volume:
 - adiciona coleção: **addAll (Collection c)**
 - remove coleção: **removeAll (Collection c)**
 - mantém coleção: **retainAll (Collection c)**
- Fornece um método que retorna um objeto **Iterator** para percorrer a coleção **iterator()**
- **int size()**
- **boolean isEmpty()**
- **boolean contains (Object o)**

Interface `List`

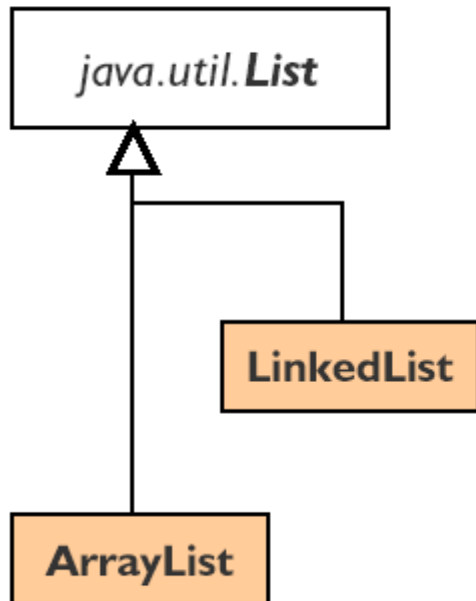
- Uma coleção do tipo `List` é uma `Collection` que tem uma `sequência` definida e que pode conter elementos `duplicados`.
- Como os arrays o índice do primeiro elemento é zero.

Interface **List**

- Além dos métodos herdados de **Collection**, fornece métodos para:
 - manipular elementos via seus índices. Ex:
 - **add(int index, Object o)**: Adiciona elemento. O tamanho da lista aumenta em 1.
 - **remove(int index)**: Remove elemento da posição especificada e move todos os elementos após o elemento removido diminuindo o tamanho da lista em 1.
 - **set(int index, Object o)**: Substitui elemento. O tamanho da lista permanece igual.
 - manipular um intervalo específico de elementos. Ex:
 - **addAll(int index, Collection c)** : Insere na posição especificada
 - **subList(int fromIndex, int toIndex)**: obtém uma parte da lista, o índice final não faz parte do intervalo. Qualquer alteração na sublist também será feita na lista original (view)
 - recuperar elementos
 - **get(int index)**
 - obter um **ListIterator** para percorrer a lista.

Interface **List**

- **List** pode ser implementada por:
 - um vetor (array): classe **ArrayList**
 - ou uma lista ligada: classe **LinkedList**



Exemplo ArrayList e Iterator

- Tarefa1: colocar dois arrays de **String** em duas listas **ArrayList**.
- Tarefa 2: utilizar um objeto **Iterator** para remover da segunda coleção **ArrayList** todos os elementos que também estiverem na primeira coleção.

Exemplo ArrayList e Iterator

ArrayList list:

MAGENTA RED WHITE BLUE CYAN

ArrayList removeList:

RED WHITE BLUE

ArrayList list após remover:

MAGENTA CYAN

Exemplo ArrayList e Iterator

```
public class CollectionTest {  
    private static final String[] colors =  
        { "MAGENTA", "RED", "WHITE", "BLUE", "CYAN" };  
    private static final String[] removeColors =  
        { "RED", "WHITE", "BLUE" };  
    // cria ArrayList, adiciona Colors a ela e a manipula  
    public CollectionTest() {  
        List< String > list = new ArrayList< String >();  
        List< String > removeList = new ArrayList< String >();  
        for ( String color : colors )  
            list.add( color );  
        for ( String color : removeColors )  
            removeList.add( color );  
    }  
}
```

Exemplo ArrayList e Iterator

```
public class CollectionTest {  
    private static final String[] colors =  
        { "MAGENTA", "RED", "WHITE", "BLUE", "CYAN" };  
    private static final String[] removeColors =  
        { "RED", "WHITE", "BLUE" };  
    // cria ArrayList, adiciona elementos  
    public CollectionTest() {  
        List< String > list = new ArrayList< String >();  
        List< String > removeList = new ArrayList< String >();  
        for ( String color : colors )  
            list.add( color );  
        for ( String color : removeColors )  
            removeList.add( color );  
    }  
}
```

Preenche a coleção **list**
com objetos **String**
armazenados no array **colors**

Exemplo ArrayList e Iterator

```
public class CollectionTest {  
    private static final String[] colors =  
        { "MAGENTA", "RED", "WHITE", "BLUE", "CYAN" };  
    private static final String[] removeColors =  
        { "RED", "WHITE", "BLUE" };  
    // cria ArrayList, adiciona Colors a ela e a manipula  
    public CollectionTest() {  
        List< String >  
        List< String >  
        for ( String color : colors )  
            list.add( color );  
        for ( String color : removeColors )  
            removeList.add( color );  
    }  
}
```

Preenche a coleção **removeList**
com objetos **String**
armazenados no array **removeColors**

Exemplo ArrayList e Iterator

```
System.out.println( "List: " );
for ( int count = 0; count < list.size(); count++ )
    System.out.printf( "%s ", list.get(count));
System.out.println( "\n RemoveList: " );
for ( String color : removeList )
    System.out.printf( "%s ", color );
// remove cores contidas em removeList
removeColors( list, removeList );
System.out.println("\n ArrayList after calling
removeColors:");
for ( String color : list )
    System.out.printf( "%s ", color );
} // fim do construtor CollectionTest
```

Exemplo ArrayList e Iterator

Chama o método **size** da interface **Collection** para obter o número de elementos da lista

```
System.out.println( "List: " );
for ( int count = 0; count < list.size(); count++ )
    System.out.printf( "%s ", list.get(count));
System.out.println( "\n RemoveList: " );
for ( String color : removeList )
    System.out.printf( "%s ", color );
// remove cores contidas em removeList
removeColors( list, removeList );
System.out.println("\n ArrayList after calling
removeColors:");
for ( String color : list )
    System.out.printf( "%s ", color );
} // fim do construtor CollectionTest
```

Exemplo ArrayList e Iterator

Chama o método **get** da interface **List** para obter cada elemento da lista

```
System.out.println( "List: " );
for ( int count = 0; count < list.size(), count++ )
    System.out.printf( "%s ", list.get(count));
System.out.println( "\n RemoveList: " );
for ( String color : removeList )
    System.out.printf( "%s ", color );
// remove cores contidas em removeList
removeColors( list, removeList );
System.out.println("\n ArrayList after calling
removeColors:");
for ( String color : list )
    System.out.printf( "%s ", color );
} // fim do construtor CollectionTest
```


Exemplo ArrayList e Iterator

A estrutura for aprimorada poderia ter sido utilizada aqui !

```
System.out.println( " " );
for ( int count = 0; count < list.size(); count++ )
    System.out.printf( "%s ", list.get(count));
System.out.println( "\n RemoveList: " );
for ( String color : removeList )
    System.out.printf( "%s ", color );
removeColors( list, removeList );
System.out.println("\n ArrayList after calling removeColors:");
for ( String color : list )
    System.out.printf( "%s ", color );
} // fim do construtor CollectionTest
```

Exemplo ArrayList e Iterator

Remove de **collection1** as cores
(objetos **String**) especificadas em **collection2**

```
private void removeColors( Collection< String > collection1,  
    Collection< String > collection2){  
    // obtém o iterador  
    Iterator< String > iterator = collection1.iterator();  
  
    // loop enquanto a coleção tiver itens  
    while (iterator.hasNext())  
  
        if (collection2.contains( iterator.next() ))  
            iterator.remove();// remove Color atual  
} // fim do método removeColors
```

Exemplo ArrayList e Iterator

Permite que quaisquer objetos **Collections** que contenham strings sejam passados como argumentos

```
private void removeColors( Collection< String > collection1,  
    Collection< String > collection2){  
    // obtém o iterador  
    Iterator< String > iterator = collection1.iterator();  
  
    // loop enquanto a coleção tiver itens  
    while (iterator.hasNext())  
  
        if (collection2.contains( iterator.next() ))  
            iterator.remove();// remove Color atual  
} // fim do método removeColors
```

Exemplo ArrayList e Iterator

O método acessa os elementos da primeira coleção via um **Iterator**. Chama o método **iterator** para obter um iterador para **collection1**

```
private void removeColors( Collection< String > collection1,
    Collection< String > collection2){
    // obtém o iterador
    Iterator< String > iterator = collection1.iterator();

    // loop enquanto a coleção tiver itens
    while (iterator.hasNext())

        if (collection2.contains( iterator.next() ))
            iterator.remove();// remove Color atual
    } // fim do método removeColors
```

Exemplo ArrayList e Iterator

Chama o método **hasnext** da interface **Iterator** para determinar se a coleção tem mais elementos

```
private void removeColors( Collection< String > collection1,
    Collection< String > collection2){
    // obtém o iterator
    Iterator< String > iterator = collection1.iterator();

    // loop enquanto a coleção tiver itens
    while (iterator.hasNext())

        if (collection2.contains( iterator.next() ))
            iterator.remove();// remove Color atual
    } // fim do método removeColors
```

Exemplo ArrayList e Iterator

Chama método **next** da interface **Iterator** para obter uma referência ao próximo elemento da coleção

```
private void removeColors( Collection< String > collection1,
                           Collection< String > collection2){
    // obtém o iterador
    Iterator< String > iterator = collection1.iterator();

    // loop enquanto a coleção tiver iterador
    while (iterator.hasNext())

        if (collection2.contains( iterator.next() ))
            iterator.remove();// remove Color atual
    } // fim do método removeColors
```

Exemplo ArrayList e Iterator

Utiliza o método **contains** da segunda coleção para determinar se a mesma contém o elemento retornado por **next**

```
private void removeColors(Collection< String > collection1,
    Collection< String > collection2){
    // obtém o iterator
    Iterator< String > iterator = collection1.iterator();

    // loop enquanto a coleção tiver itens
    while (iterator.hasNext())

        if (collection2.contains( iterator.next() ))
            iterator.remove();// remove Color atual
    } // fim do método removeColors
```

Erro de programação comum

- Se uma coleção for modificada por um de seus métodos depois de um iterador ter sido criado para essa coleção:
 - o iterador se torna imediatamente inválido! lançando **ConcurrentModificationException**

Exemplo modificado 1

```
public class CollectionTest {  
    private static final String[] colors =  
        { "MAGENTA", "RED", "WHITE", "BLUE", "CYAN" };  
    private static final String[] removeColors =  
        { "RED", "WHITE", "BLUE" };  
    // cria ArrayList, adiciona Colors a ela e a manipula  
    public CollectionTest() {  
        List< String > list = new ArrayList< String >();  
        List< String > removeList = new ArrayList< String >();  
        for ( String color : colors )  
            list.add( color );  
        list.add(1, "PINK");  
        list.set(0, "GREEN");  
        for ( String color : removeColors )  
            removeList.add( color );  
    }  
}
```

Exemplo modificado 1

List:

GREEN PINK RED WHITE BLUE CYAN

RemoveList:

RED WHITE BLUE

ArrayList after calling removeColors:

GREEN PINK CYAN

Exemplo modificado 1

List:

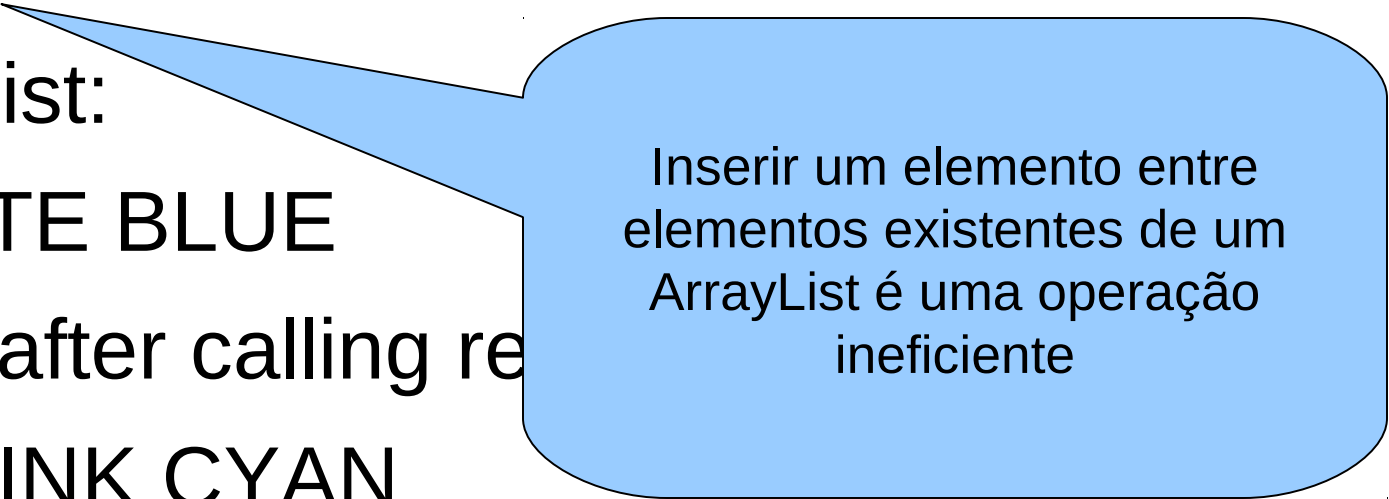
GREEN PINK RED WHITE BLUE CYAN

RemoveList:

RED WHITE BLUE

ArrayList after calling re

GREEN PINK CYAN



Inserir um elemento entre elementos existentes de um ArrayList é uma operação ineficiente

Exemplo modificado 2

```
public class CollectionTest {
    private static final String[] colors =
        { "MAGENTA", "RED", "WHITE", "BLUE", "CYAN" };
    private static final String[] removeColors =
        { "RED", "WHITE", "BLUE" };
    // cria ArrayList, adiciona Colors a ela e a manipula
    public CollectionTest() {
        List< String > list = new ArrayList< String >();
        List< String > removeList = new ArrayList< String >();
        for ( String color : colors )
            list.add( color );
        list.add(1, "PINK");
        list.set(0, "GREEN");
        list.set(7, "BLACK");
        for ( String color : removeColors )
            removeList.add( color );
    }
}
```

Exemplo modificado 2

Exception in thread "main"

java.lang.**IndexOutOfBoundsException**:

Index: 7, Size: 6

rastreamento de pilha

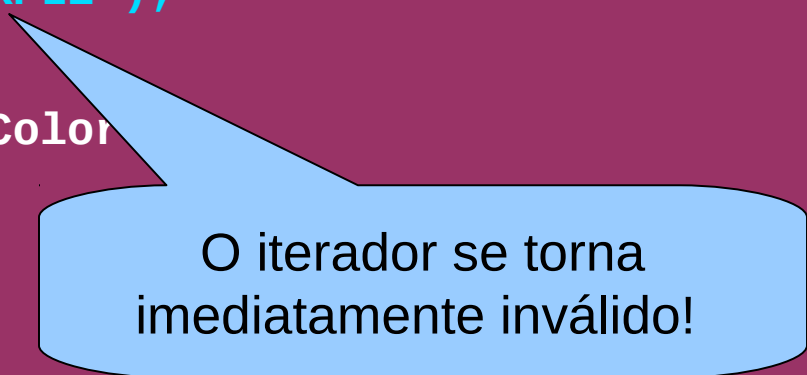
...

Exemplo modificado 3

```
private void removeColors( Collection< String > collection1,  
    Collection< String > collection2){  
    // obtém o iterador  
    Iterator< String > iterator = collection1.iterator();  
  
    // loop enquanto a coleção tiver itens  
    while (iterator.hasNext()){  
        if (collection2.contains( iterator.next() ))  
            iterator.remove();// remove Color atual  
        collection1.add("PURPLE");  
    }  
} // fim do método removeColors
```

Exemplo modificado 3

```
private void removeColors( Collection< String > collection1,  
    Collection< String > collection2){  
    // obtém o iterador  
    Iterator< String > iterator = collection1.iterator();  
  
    // loop enquanto a coleção tiver itens  
    while (iterator.hasNext()){  
        if (collection2.contains( iterator.next() ))  
            iterator.remove();// remove Color atual  
        collection1.add("PURPLE");  
    }  
} // fim do método removeColor
```



O iterador se torna imediatamente inválido!

Exemplo modificado 3

Exception in thread "main"

java.util.ConcurrentModificationException at

...

Exemplo modificado 4

```
private void removeColors( Collection< String > collection1,  
    Collection< String > collection2){  
    // obtém o iterador  
    Iterator< String > iterator = collection1.iterator();  
  
    // loop enquanto a coleção tiver itens  
    while (iterator.hasNext()){  
        if (collection2.contains( iterator.next() ))  
            iterator.remove();// remove Color atual  
        collection2.add("CYAN");  
    }  
} // fim do método removeColors
```

Exemplo modificado 4

list:

MAGENTA RED WHITE BLUE CYAN

removeList:

RED WHITE BLUE

ArrayList after calling removeColors:

MAGENTA

Exemplo modificado 5

```
private void removeColors( Collection< String > collection1,  
    Collection< String > collection2){  
    // obtém o iterador  
    Iterator< String > iterator = collection1.iterator();  
  
    // loop enquanto a coleção tiver itens  
    while (iterator.hasNext()){  
        if (collection2.contains( iterator.next() ))  
            iterator.remove();// remove Color atual  
        collection2.add("MAGENTA");  
    }  
} // fim do método removeColors
```

Exemplo modificado 5

list:

MAGENTA RED WHITE BLUE CYAN

removeList:

RED WHITE BLUE

ArrayList after calling removeColors:

MAGENTA CYAN

Exemplo ArrayList

```
public class Student {  
    private String name;  
    private Double grade1;  
    private Double grade2;  
    private Double average;  
    public Student(String name, Double grade1, Double grade2){  
        this.name=name;  
        this.grade1=grade1;  
        this.grade2=grade2;  
        this.computeAvg();  
    }  
}
```

Exemplo ArrayList

```
public String getName(){
    return name;
}
public Double getGrade1(){
    return grade1;
}
public Double getGrade2(){
    return grade2;
}
public Double getAverage(){
    return average;
}
public void computeAvg(){
    this.average=new Double((this.grade1.doubleValue()+this.grade2.doubleValue())/2);
}
public String toString(){
    String studString=name+" "+grade1+ " "+grade2+" "+average;
    return studString;
}
```

Exemplo ArrayList

```
public class CollectionTest
{
    // cria ArrayList, adiciona Alunos a ela e a manipula
    public CollectionTest() {
        List< Student > list = new ArrayList< Student >();
        List< Student > removeList = new ArrayList< Student>();
        // adiciona elementos a list
        Student s1=new Student("Alexandre",1.0,2.0);
        Student s2=new Student("Guillerme",5.0,8.0);
        Student s3=new Student("Cristina",10.0,9.0);
        Student s4=new Student("Jesus",9.0,5.0);
        list.add(0, s1);
        list.add(1, s2);
        list.add(2, s3);
        list.add(3, s4);
        // adiciona elementos a removeList
        removeList.add( s1 );
        removeList.add( s4 );
    }
}
```

Exemplo ArrayList

```
System.out.println( "List: " );  
....  
System.out.println( "\nRemoveList: " );  
...  
// remove estudantes de removeList  
removeStudents( list, removeList );  
System.out.println("\n ArrayList after calling removeColors:");  
for ( Student student : list )  
    System.out.printf( "%s ", student );  
} // fim do construtor CollectionTest
```


Exemplo ArrayList

```
private void removeStudents( Collection< Student > collection1,  
Collection< Student > collection2){  
  
    Iterator< Student > iterator = collection1.iterator();  
    // loop enquanto a coleção tiver itens  
    while (iterator.hasNext()){  
        if (collection2.contains( iterator.next() ))  
            iterator.remove();//  
    }  
  
} // fim do método removeStudents
```

Exemplo ArrayList modificado

```
System.out.println( "List: " );
....
System.out.println( "\nRemoveList: " );
...
// remove estudantes de removeList
//removeStudents( list, removeList );
list.removeAll(removeList);
System.out.println("\n ArrayList after calling removeColors:");
for ( Student student : list )
    System.out.printf( "%s ", student );
} // fim do construtor CollectionTest
```

Padrão de Projeto Iterator e Iteradores no Java

PADRÕES DE PROJETO

- A idéia de padrões foi apresentada por Christopher Alexander em 1977 no contexto de **Arquitetura** (de prédios e cidades):

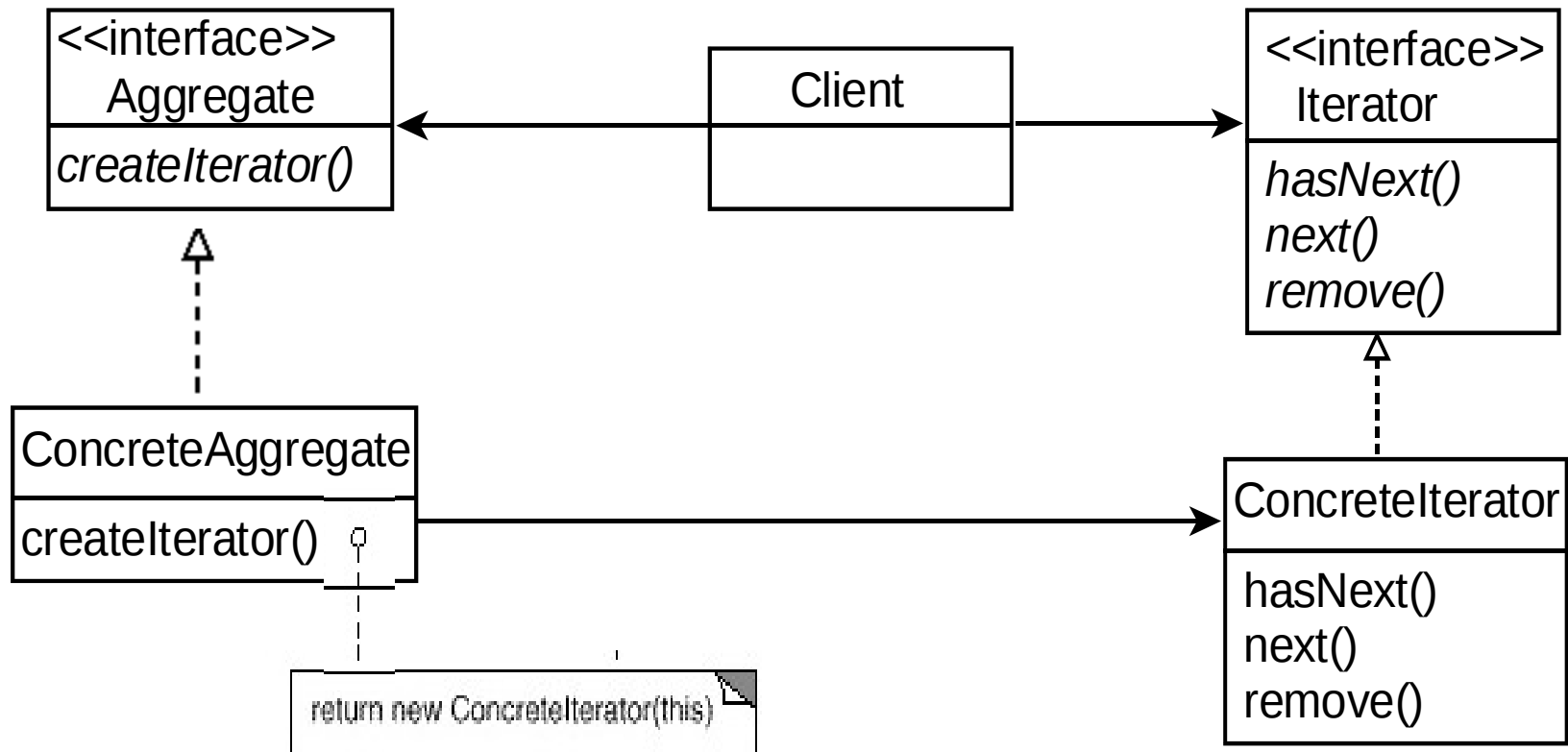
“Cada padrão descreve um **problema** que **ocorre repetidamente** de novo e de novo em nosso ambiente, e então descreve a parte central da solução para aquele problema de uma forma que você pode **usar** esta **solução** um **milhão** de vezes, sem nunca implementá-la duas vezes da mesma forma”.

Padrão de Projeto Iterator

Motivação

- O padrão Iterator permite que o implementador forneça uma interface uniforme para **percorrer vários tipos de objetos agregados**.
- A ideia chave é
 - **retirar do objeto agregado as responsabilidades de acesso e percurso e**
 - **delegá-las a um objeto Iterator** que definirá um protocolo de percurso.

Padrão de Projeto Iterator - Estrutura



Padrão Iterator

○ Participantes:

- Iterator:
 - Define uma interface para acessar e percorrer elementos
- ConcreteIterator
 - Implementa a interface de Iterator
 - Mantém o controle da posição corrente no percurso do agregado.
- Aggregate
 - Define uma interface para a criação de um objeto Iterator
- ConcreteAggregate
 - Implementa a interface de criação do Iterator para retornar uma instância do ConcreteIterator apropriado

Iteradores de coleções

Passos para usar um objeto iterador:

1. Obtenha um iterador para a coleção usando o método **iterator()** da própria coleção.
 - Esse método retorna um objeto iterador, posicionado antes do primeiro objeto da coleção.
2. Verifique se há mais elementos na coleção com uma chamada ao método **hasnext()** do objeto iterador.
3. Obtenha o próximo objeto na coleção com o método **next()** do objeto iterador.

Iteradores de coleções

- O método **remove()** apaga o último item retornado pelo método **next()**.



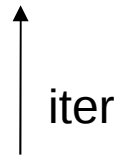
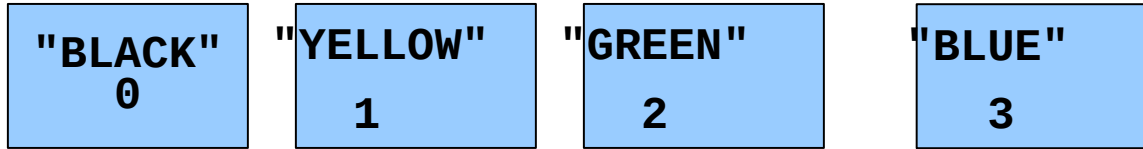
Método de Iterator!!!

Interface Iterator

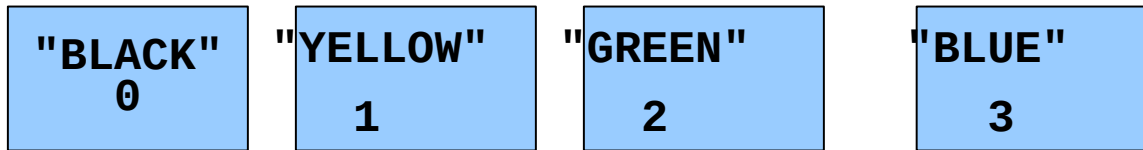
Essa interface permite percorrer qualquer coleção como uma estrutura sequencial

- Determinar se a coleção tem mais elementos: `hasNext()`
- Obter uma referência ao próximo elemento da coleção: `next()`
- Apagar o último item retornado pelo método `next()`: `remove()`

Interface Iterator

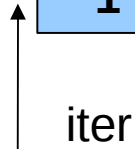
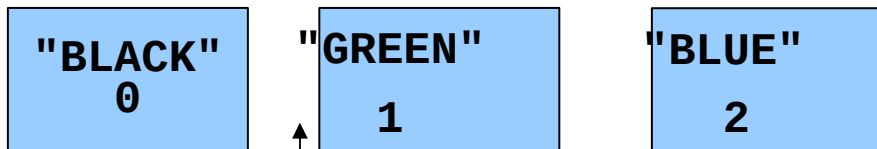


Executamos `iter.next()`



Executamos `iter.remove()`

Apaga o último item retornado pelo método `next()`

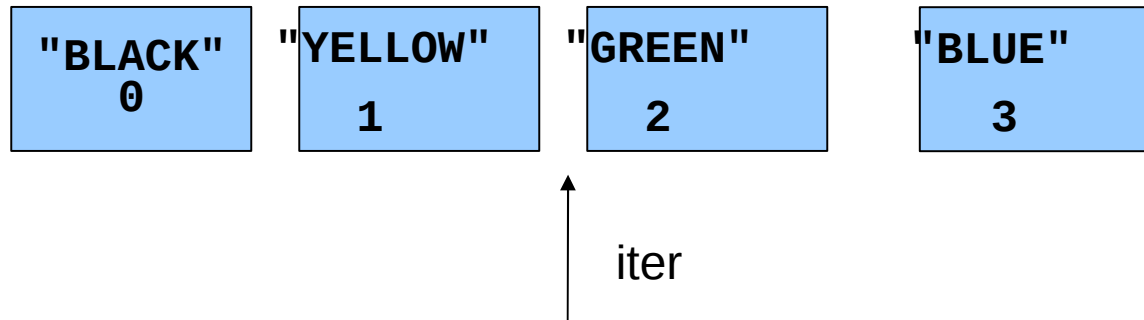


Interface **ListIterator**

Iterador de lista (**ListIterator**)

- Pode ser criado por qualquer coleção que implemente a interface **List**.
- possui um método adicional **listIterator(int index)** que:
 - Retorna: um iterador posicionado para visitar o elemento nessa localização na primeira chamada à **next()**.

```
ListIterator< String > iter = list.listIterator(2);
```



Interface ListIterator

- Fornece adicionalmente os seguintes métodos:
 - **hasPrevious()** : determina se há mais elementos ao percorrer a lista em ordem invertida.
 - **previous()**: Obtem uma referência ao elemento anterior da lista.
 - **set(Object o)**: substitui o último item retornado pelo método `next()`
 - **add(Object o)**: adiciona um objeto na posição atualmente apontada pelo iterador.

LinkedList e ListIterator – Exemplo 1

```
colors[] = { "black", "yellow", "green", "blue", "violet", "silver"};  
colors2[] = { "gold", "white", "brown", "blue", "gray", "silver" };
```

Saída:

```
"BLACK", "YELLOW", "GREEN", "BLUE", "VIOLET", "SILVER", "GOLD",  
"WHITE", "BROWN", "BLUE", "GRAY", "SILVER"
```

LinkedList e

ListIterator – Exemplo 1

```
import java.util.List;
import java.util.LinkedList;
import java.util.ListIterator;
public class ListTest
{
    private static final String colors[] = { "black", "yellow",
"green", "blue", "violet", "silver" };
    private static final String colors2[] = { "gold", "white",
"brown", "blue", "gray", "silver" };
    // configura e manipula objetos LinkedList
    public ListTest()
    {
        List< String > list1 = new LinkedList< String >();
        List< String > list2 = new LinkedList< String >();
```


LinkedList e ListIterator – Exemplo 1

Adiciona elementos às duas listas
usando o método add de Collection

```
// adiciona elementos a list1
    for ( String color : colors )
        list1.add( color );

// adiciona elementos a list2
    for ( String color : colors2 )
        list2.add( color );
```

LinkedList e ListIterator – Exemplo 1

Todos elementos da lista **list2** são adicionados à lista **list1**

```
list1.addAll( list2 ); // concatena as listas  
convertToUppercaseStrings( list1 );  
printList( list1 ); // imprime elementos
```

Chama o método **printlist** para gerar a saída do conteúdo de **list1**

Converte cada elemento **String** da lista em letras maiúsculas

LinkedList e ListIterator – Exemplo 1

Recupera objetos **String** e
converte em letras maiúsculas

```
private void convertToUppercaseStrings(List< String > list){  
    ListIterator< String > iterator = list.listIterator();  
    while (iterator.hasNext())  
    {  
        String color = iterator.next(); // obtém o item  
        iterator.set( color.toUpperCase()); // converte em letras maiúsculas  
    } // fim do while  
} // fim do método convertToUppercaseStrings
```

LinkedList e ListIterator – Exemplo 1

Chama o método **listIterator** da interface **List** para obter um iterador bidirecional para a lista

```
private void convertToUppercaseStrings(List<String> list){  
    ListIterator<String> iterator = list.listIterator();  
    while (iterator.hasNext())  
    {  
        String color = iterator.next(); // obtém o item  
        iterator.set( color.toUpperCase()); // converte em letras  
        maiúsculas  
    } // fim do while  
} // fim do método convertToUppercaseStrings
```

LinkedList e ListIterator – Exemplo 1

Chama o método **toUpperCase** da classe **String** para obter uma versão em letras maiúsculas da **string**

```
private void convertToUppercaseStrings(List<String> list){  
    ListIterator<String> iterator = list.ListIterator();  
    while (iterator.hasNext())  
    {  
        String color = iterator.next(); // obtém o item  
        iterator.set( color.toUpperCase()); // converte em letras  
        maiúsculas  
    } // fim do while  
} // fim do método convertToUppercaseStrings
```

LinkedList e

ListIterator – Exemplo 1

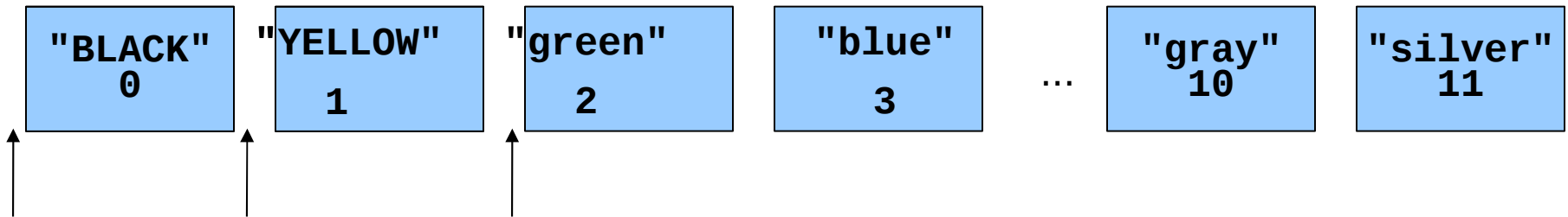
Chama o método **set** da interface **ListIterator** para substituir a **string** retornada pelo método **next()**, pela sua versão em letras maiúsculas

```
private void convertToUppercaseStrings(List< String >
    list){
    ListIterator< String > iterator = list.listIterator();
    while (iterator.hasNext())
    {
        String color = iterator.next(); // obtém o item
        iterator.set( color.toUpperCase()); // converte em letras
        maiúsculas
    } // fim do while
} // fim do método convertToUppercaseStrings
```

LinkedList e

ListIterator – Exemplo 1

“A ListIterator has no current element; its cursor position always lies between the element that would be returned by a call to previous() and the element that would be returned by a call to next()”



```
ListIterator< String > iterator = list.listIterator();  
while (iterator.hasNext())  
{  
    String color = iterator.next(); // obtém o item  
    iterator.set( color.toUpperCase()); // converte em letras  
    maiúsculas  
} // fim do while
```

substitui o último item retornado
pelo método next()

LinkedList e ListIterator – Exemplo 1

imprime **list**

```
public void printList(List< String > list)
{
    System.out.println( "\nlist: " );
    for ( String color : list )
        System.out.printf( "%s ", color );
    System.out.println();
} // fim do método printList
```

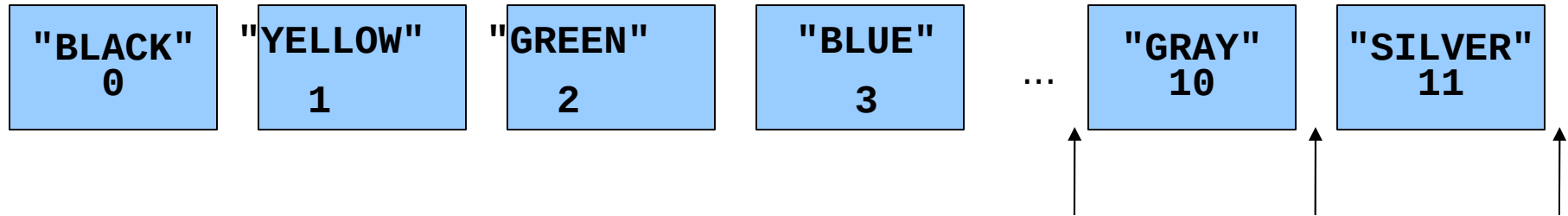

Exemplo 2: Imprimir a lista invertida

Imprime a lista invertida (de trás pra frente)

Chama o método **listIterator** com a posição inicial do iterador (nesse caso, o último elemento)

```
private void printReversedList(List< String > list){  
    ListIterator< String> iterator = list.listIterator(list.size());  
  
    System.out.println( "\nReversed List:" );  
  
    // imprime lista na ordem inversa  
    while (iterator.hasPrevious())  
        System.out.printf( "%s ", iterator.previous());  
} // fim do método printReversedList
```

Exemplo 2: Imprimir a lista invertida



```
private void printReversedList(List< String > list){  
    ListIterator< String > iterator = list.listIterator(list.size()  
);  
  
    System.out.println( "\nReversed List:" );  
  
    // imprime lista na ordem inversa  
    while (iterator.hasPrevious())  
        System.out.printf( "%s ", iterator.previous());  
} // fim do método printReversedList
```

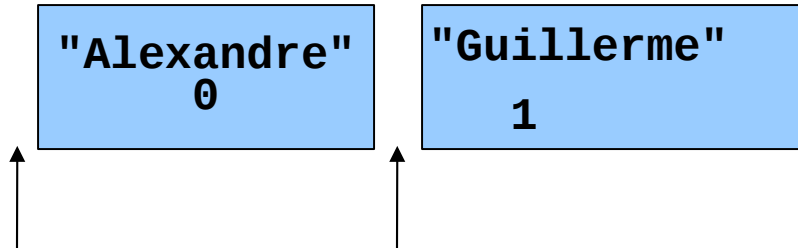
Exemplo 3: Como funciona o add de ListIterator?

```
Student s1=new Student("Alexandre",1.0,2.0);
Student s2=new Student("Guillermo",5.0,8.0);
Student s3=new Student("Cristina",10.0,9.0);
Student s4=new Student("Jesus",9.0,5.0);

list.add(0, s1);
list.add(1, s2);
ListIterator<Student> it= list.listIterator();
it.next();
it.add(s3);

System.out.println( "List it: " );
// gera saída do conteúdo da lista
for ( int count = 0; count < list.size(); count++ )
    System.out.printf( "%s ", list.get( count ) );
```

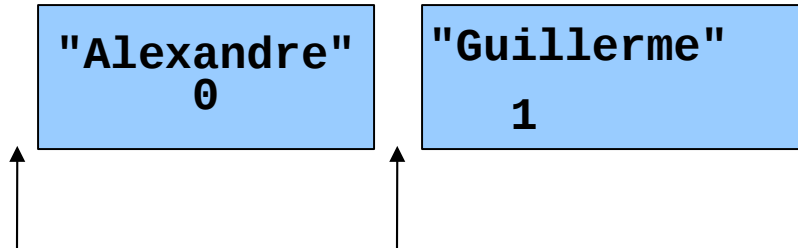
Exemplo 3: Como funciona o add de ListIterator?



add(Object o): “Inserts the specified element into the list. The element is inserted immediately before the next element that would be returned by next, if any, and after the next element that would be returned by previous, if any.”

```
Student s1=new Student("Alexandre",1.0,2.0);
Student s2=new Student("Guillermo",5.0,8.0);
Student s3=new Student("Cristina",10.0,9.0);
Student s4=new Student("Jesus",9.0,5.0);
list.add(0, s1);
list.add(1, s2);
ListIterator<Student> it= list.listIterator();
it.next();
it.add(s3);
```

Exemplo 3: Como funciona o add de ListIterator?

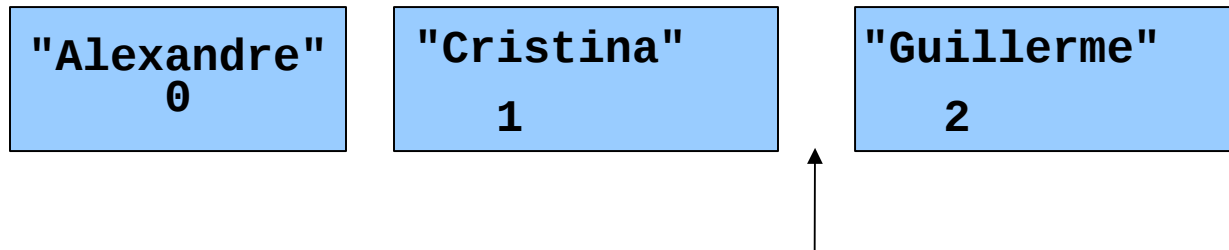


add(Object o): "Inserts the specified element into the list. The element is inserted at the next position by next element previously returned by next()."

adiciona um objeto na posição atualmente apontada pelo iterador.

```
Student s1=new Student("Alexandre",1.0,2.0);
Student s2=new Student("Guillermo",5.0,8.0);
Student s3=new Student("Cristina",10.0,9.0);
Student s4=new Student("Jesus",9.0,5.0);
list.add(0, s1);
list.add(1, s2);
ListIterator<Student> it= list.listIterator();
it.next();
it.add(s3);
```

Exemplo 3: Como funciona o add de ListIterator?



List it:

Alexandre	1.0	2.0	1.5
Cristina	10.0	9.0	9.5
Guillherme	5.0	8.0	6.5

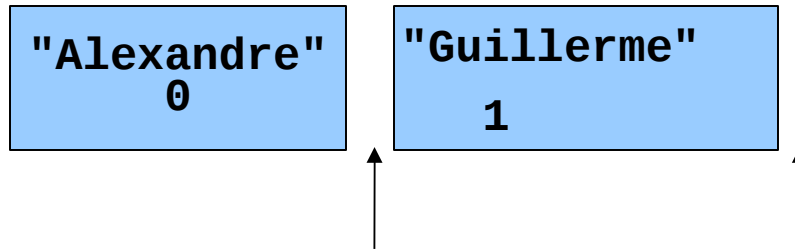
Exemplo 3: Como funciona o add de ListIterator?

```
Student s1=new Student("Alexandre",1.0,2.0);
Student s2=new Student("Guillermo",5.0,8.0);
Student s3=new Student("Cristina",10.0,9.0);
Student s4=new Student("Jesus",9.0,5.0);

list.add(0, s1);
list.add(1, s2);
ListIterator<Student> it= list.listIterator(list.size());
System.out.print(it.previous());
it.add(s3);

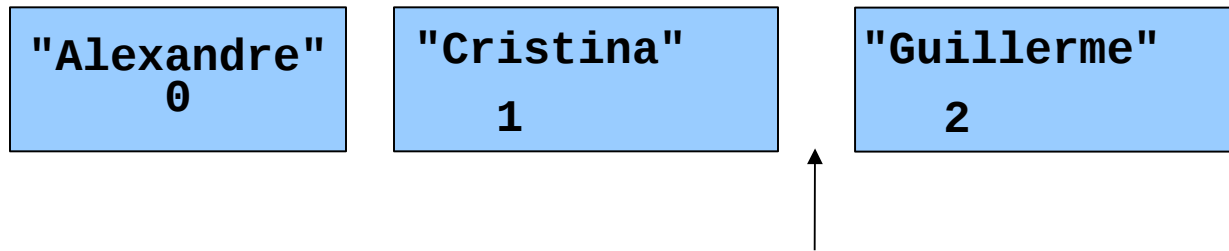
System.out.println( "List it: " );
// gera saída do conteúdo da lista
for ( int count = 0; count < list.size(); count++ )
    System.out.printf( "%s ", list.get( count ) );
```

Exemplo 3: Como funciona o add de ListIterator?



```
Student s1=new Student("Alexandre",1.0,2.0);
Student s2=new Student("Guillerme",5.0,8.0);
Student s3=new Student("Cristina",10.0,9.0);
Student s4=new Student("Jesus",9.0,5.0);
list.add(0, s1);
list.add(1, s2);
ListIterator<Student> it= list.listIterator(list.size());
System.out.print(it.previous());
it.add(s3);
```


Exemplo 3: Como funciona o add de ListIterator?



List it:

Alexandre	1.0	2.0	1.5
Cristina	10.0	9.0	9.5
Guillermo	5.0	8.0	6.5

Exemplo 4: uso de subList e clear

Obtém uma sublista e utiliza o método clear para excluir os itens da sublista

```
private void removeItems(List< String > list, int start, int end)
{
    list.subList(start,end).clear();
} // fim do método removeItems
```

```
"MAGENTA", "RED", "WHITE", "BLUE", "CYAN"
removeItems( list1,1,3);
"MAGENTA", "BLUE", "CYAN"
```