

Começando com Arrays

Transcrição

Você pode fazer o [DOWNLOAD \(<https://caelum-online-public.s3.amazonaws.com/850-java-util/01/java6-projetos-iniciais.zip>\)](https://caelum-online-public.s3.amazonaws.com/850-java-util/01/java6-projetos-iniciais.zip), dos projetos usados no curso anterior.

Olá!

Neste curso, nosso objetivo é entendermos os pacotes `java.util` e `java.io`, e continuaremos a utilizar o `java.lang`.

Com relação ao `java.lang`, entenderemos melhor a nossa classe `Object`, pois temos os métodos `hashCode()` e `equals()`, isso será esclarecido ao falarmos sobre `java.util`.

Antes de entrarmos no `java.util` em si, retornaremos ao código que escrevemos para aprendermos um novo conceito.

Temos o projeto `bytebank-herdado-conta`, que finalizamos anteriormente, e conseguimos entendê-lo completamente, com uma exceção, vamos observar o código:

```
package br.com.bytebank.banco.test;

import br.com.bytebank.banco.modelo.Cliente;

public class Teste {
```

```
public static void main(String[] args) {  
  
//Código omitido
```

[COPIAR CÓDIGO](#)

Na declaração do método `main`, não sabemos exatamente qual é a função dos colchetes `[]`. Antes de começarmos com o `java.util`, primeiro desvendaremos este mistério.

Renomearemos a classe `Teste`, clicando com o botão direto do mouse sobre ela, e selecionando "Refactor > Rename", a chamaremos de `TesteObject`. Em seguida, criaremos uma nova classe de teste, chamada `Teste`, com o seguinte conteúdo:

```
package br.com.bytebank.banco.test;  
  
public class Teste {  
  
    public static void main(String[] args) {  
  
    }  
}
```

[COPIAR CÓDIGO](#)

Por que existem os colchetes `[]`?

Imaginemos que queremos armazenar um conjunto de dados, por exemplo, as idades de cinco pessoas, para isso, utilizamos as variáveis:

```
package br.com.bytebank.banco.test;  
  
public class Teste {  
  
    public static void main(String[] args) {
```

```
    int idade1 = 29;  
    int idade2 = 39;  
    int idade3 = 19;  
    int idade4 = 69;  
    int idade5 = 59;  
  
}  
}
```

[COPIAR CÓDIGO](#)

Para cinco idades, isto pode até funcionar, mas quando trabalhamos com um grande número de dados, por exemplo, cem idades, já se torna inviável.

Precisamos de formas mais enxutas de armazenamento de dados, e para isso existem as **estruturas de dados**.

Conheceremos agora nossa primeira estrutura de dados, que se chama **array**.

```
package br.com.bytebank.banco.test;  
  
public class Teste {  
  
    //Array []  
    public static void main(String[] args) {  
  
        int idade1 = 29;  
        int idade2 = 39;  
        int idade3 = 19;  
        int idade4 = 69;  
        int idade5 = 59;  
  
    }  
}
```

[COPIAR CÓDIGO](#)

Raramente os arrays serão utilizados da forma como faremos agora, ou seja, manualmente.

Arrays são colchetes ([]) associados a algum tipo de dados, no caso, queremos apresentar idade que, como vimos, são do tipo `int` . Assim, o array será do tipo `int` . Em seguida, precisaremos de um nome para a variável, que será `idades` :

```
package br.com.bytebank.banco.test;

public class Teste {

    //Array []
    public static void main(String[] args) {

        int[] idades;

        int idade1 = 29;
        int idade2 = 39;
        int idade3 = 19;
        int idade4 = 69;
        int idade5 = 59;

    }
}
```

[COPIAR CÓDIGO](#)

Os colchetes ([]) também poderiam estar posicionados após o nome da variável, da seguinte forma `int idades[]` , apesar de ser mais comum sua utilização da forma como colocamos no código acima.

Um array também é um objeto, assim sendo, precisamos criá-lo, pois temos uma referência que ainda não foi inicializada:

```
package br.com.bytebank.banco.test;

public class Teste {

    //Array []
    public static void main(String[] args) {

        int[]idades = new int[];

        int idade1 = 29;
        int idade2 = 39;
        int idade3 = 19;
        int idade4 = 69;
        int idade5 = 59;

    }
}
```

[COPIAR CÓDIGO](#)

O compilador aponta um erro de compilação. Isso porque, ao criar o array, precisamos informar o seu tamanho. Quantas idades queremos guardar? 5, portanto, o tamanho do nosso array é 5 :

```
package br.com.bytebank.banco.test;

public class Teste {

    //Array []
    public static void main(String[] args) {

        int[]idades = new int[5];

        int idade1 = 29;
        int idade2 = 39;
        int idade3 = 19;
        int idade4 = 69;
```

```
    int idade5 = 59;  
  
}  
}
```

[COPIAR CÓDIGO](#)

Assim que inserimos o tamanho do array o código volta a compilar.

Internamente, temos um processo que começa com uma pilha de execução, representada pelo método `main`. Nesta pilha, ou seja, no método, foi criada uma variável `idades`.

Esta variável é um array, que como vimos, é um objeto. Os objetos ficam armazenados na memória HEAP. Assim, `idades` é uma referência que aponta para um objeto na memória, capaz de guardar cinco idades.

Como podemos acessar uma das posições? Utilizamos a variável como referência e precisamos informar a posição que queremos acessar dentro de colchetes (`[]`). Por exemplo, se quisermos acessar a primeira posição, ela é representada pelo número `0`:

```
package br.com.bytebank.banco.test;  
  
public class Teste {  
  
    //Array []  
    public static void main(String[] args) {  
  
        int[] idades = new int[5];  
  
        idades[0]  
  
        int idade1 = 29;  
        int idade2 = 39;  
        int idade3 = 19;
```

```
    int idade4 = 69;  
    int idade5 = 59;  
  
}  
}
```

[COPIAR CÓDIGO](#)

Nos será devolvido o valor que estiver armazenado na referida posição. Por enquanto, este valor é `0` já que, por padrão, ao criarmos um array ele é inicializado com `0` em todas as posições:

```
package br.com.bytebank.banco.test;  
  
public class Teste {  
  
    //Array []  
    public static void main(String[] args) {  
  
        int[] idades = new int[5]; //inicializa o array com os  
        idades[0]  
  
        int idade1 = 29;  
        int idade2 = 39;  
        int idade3 = 19;  
        int idade4 = 69;  
        int idade5 = 59;  
  
    }  
}
```

[COPIAR CÓDIGO](#)

O primeiro valor padrão do tipo `int` disponível é `0`.

Acessando o array na posição 0, ele nos retorna o valor desta posição, que no nosso caso é a `idade1`. Podemos apagar os dados que havíamos preenchido anteriormente:

```
package br.com.bytebank.banco.test;

public class Teste {

    //Array []
    public static void main(String[] args) {

        int[] idades = new int[5]; //inicializa o array com os

        int idade1 = idades[0];

        System.out.println(idade1);

    }
}
```

[COPIAR CÓDIGO](#)

Ao executarmos, temos o seguinte resultado no console:

0

[COPIAR CÓDIGO](#)

Ele nos imprimiu o valor presente na posição 0, que por coincidência, também é 0. Mas não queremos que seja 0, de acordo com nossos dados, ele deve ser 29. Para isso, inicializaremos o nosso array.

Utilizamos primeiro a referência do array, `idades`, seguida pelos colchetes ([]), que devem ser preenchidos com a posição que desejamos utilizar, no caso 0, e ao fim, informamos o valor que desejamos armazenar, no caso, 29 :

```
package br.com.bytebank.banco.test;

public class Teste {

    //Array []
    public static void main(String[] args) {

        int[]idades = new int[5]; //inicializa o array com os

        idades[0] = 29;

        int idade1 = idades[0];

        System.out.println(idade1);

    }
}
```

[COPIAR CÓDIGO](#)

Isso significa que estamos armazenando no primeiro espaço de memória que criamos, como falamos acima. O 29 está no índice zero, ou seja, na primeira posição.¹

Agora, ao executarmos, temos o seguinte resultado no console:

29

[COPIAR CÓDIGO](#)

Com isso, podemos inicializar as demais posições do array:

```
package br.com.bytebank.banco.test;

public class Teste {
```

```
//Array []
public static void main(String[] args) {

    int[]idades = new int[5]; //inicializa o array com os

    idades[0] = 29;
    idades[1] = 39;
    idades[2] = 49;
    idades[3] = 59;
    idades[4] = 69;

    int idade1 = idades[0];

    System.out.println(idade1);

}
```

[COPIAR CÓDIGO](#)

Tentaremos acessar, por exemplo, a posição 4:

```
package br.com.bytebank.banco.test;

public class Teste {

    //Array []
    public static void main(String[] args) {

        int[]idades = new int[5]; //inicializa o array com os

        idades[0] = 29;
        idades[1] = 39;
        idades[2] = 49;
        idades[3] = 59;
        idades[4] = 69;
```

```
int idade1 = idades[4];  
  
System.out.println(idade1);  
  
}  
}
```

[COPIAR CÓDIGO](#)

Executaremos o programa, e temos o seguinte resultado no console:

69

[COPIAR CÓDIGO](#)

Se o array tivesse, por exemplo, 50 posições, seria possível acessarmos a de número 49:

```
package br.com.bytebank.banco.test;  
  
public class Teste {  
  
    //Array []  
    public static void main(String[] args) {  
  
        int[]idades = new int[50]; //inicializa o array com o  
  
        idades[0] = 29;  
        idades[1] = 39;  
        idades[2] = 49;  
        idades[3] = 59;  
        idades[4] = 69;  
  
        int idade1 = idades[49];  
  
        System.out.println(idade1);
```

```
}
```

[COPIAR CÓDIGO](#)

Ao executarmos o programa, temos o seguinte resultado no console:

```
0
```

[COPIAR CÓDIGO](#)

Isso porque não inicializamos essa posição, logo, nos é fornecido o valor padrão.

E o que acontece se tentarmos acessar uma posição que não existe? voltaremos a definir o array com 5 posições, e tentaremos novamente acessar a de número 49:

```
package br.com.bytebank.banco.test;

public class Teste {

    //Array []
    public static void main(String[] args) {

        int[] idades = new int[5]; //inicializa o array com os

        idades[0] = 29;
        idades[1] = 39;
        idades[2] = 49;
        idades[3] = 59;
        idades[4] = 69;

        int idade1 = idades[49];

        System.out.println(idade1);
```

```
}
```

[COPIAR CÓDIGO](#)

Executaremos o programa, e temos o seguinte resultado no console:

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException
at br.com.bytebank.banco.test.Teste.main(Teste.java:16)
```

[COPIAR CÓDIGO](#)

Um erro ocorre. Este tipo de erro, inclusive, é bastante comum. É uma exceção *unchecked*, não somos obrigados a fazer nenhum tratamento.

Os arrays nos permitem ainda que perguntemos o seu tamanho. Criaremos um `System.out.println()`, utilizando a referência `idades`, e chamando o atributo `length` - notamos que não é um método pois não é acompanhado de parênteses:

```
package br.com.bytebank.banco.test;

public class Teste {

    //Array []
    public static void main(String[] args) {

        int[] idades = new int[5]; //inicializa o array com os

        idades[0] = 29;
        idades[1] = 39;
        idades[2] = 49;
        idades[3] = 59;
        idades[4] = 69;
```

```
int idade1 = idades[4];  
  
System.out.println(idade1);  
  
System.out.println(idades.length);  
  
}  
}
```

[COPIAR CÓDIGO](#)



Com a posição de volta para 4 , na impressão, executaremos o programa e temos o seguinte resultado no console:

```
69  
5
```

[COPIAR CÓDIGO](#)

Ou seja, temos em primeiro lugar o valor armazenado na quarta posição, 69 , e, em seguida, o tamanho do nosso array, que possui um total de 5 posições.

Veremos agora como podemos inicializar um array dentro de um laço. Primeiro, apagaremos todo o código referente a inicialização que acabamos de criar.

Definiremos a primeira posição como 0 , portanto, i = 0 . O limite do laço será o número de posições, assim, utilizaremos o atributo que acabamos de aprender idades.length . Por fim, incrementaremos com o i++ :

```
package br.com.bytebank.banco.test;  
  
public class Teste {  
  
    //Array []  
    public static void main(String[] args) {
```

```
int[] idades = new int[5]; //inicializa o array com os  
  
for(int i = 0; i < idades.length; i++) {  
  
}  
}  
}
```

[COPIAR CÓDIGO](#)

O próximo passo é a inicialização do array, dentro do laço.

Como `i` representa as posições, é esta variável que utilizaremos na inicialização, e que receberá `i * i`:

```
package br.com.bytebank.banco.test;  
  
public class Teste {  
  
    //Array []  
    public static void main(String[] args) {  
  
        int[] idades = new int[5]; //inicializa o array com os  
  
        for(int i = 0; i < idades.length; i++) {  
            idades[i] = i * i;  
  
        }  
    }  
}
```

[COPIAR CÓDIGO](#)

Em seguida, teremos o mesmo laço, mas dentro desta segunda representação faremos a impressão dos valores:

```
package br.com.bytebank.banco.test;

public class Teste {

    //Array []
    public static void main(String[] args) {

        int[] idades = new int[5]; //inicializa o array com os

        for(int i = 0; i < idades.length; i++) {
            idades[i] = i * i;
        }

        for(int i = 0; i < idades.length; i++) {
            System.out.println(idades[i]);
        }

    }
}
```

[COPIAR CÓDIGO](#)

Executaremos, e temos o seguinte resultado no console:

```
0
1
4
9
16
```

[COPIAR CÓDIGO](#)

Funcionou! Assim, conseguimos utilizar o array, também, com um laço.

Ainda não vimos o porquê da existência do `String[]` na assinatura do método `main`, mas chegaremos lá. Primeiro, precisamos entender o que são arrays de referências.

Até a próxima!

Array de referências

Transcrição

Nesta aula, daremos continuidade à construção do array que inicializamos na aula anterior.

Como arrays são objetos, para criarmos um novo, utilizamos a palavra `new`:

```
public class Teste {  
  
    public static void main(String[] args) {  
        int[] idades = new int[5];  
  
    }  
}
```

[COPIAR CÓDIGO](#)

Precisamos definir qual tipo de dados são armazenados, no caso, utilizamos o `int`. Indicamos que se tratam de arrays por meio do uso de colchetes (`[]`), os utilizamos tanto ao declarar o tipo, quanto ao definir o tamanho do array. Todo array deve ter um tamanho fixo, pré-definido.

No nosso caso, definimos o tamanho como `5`, isso significa que, na memória onde os objetos são armazenados é criado um espaço suficiente para que sejam guardados cinco números inteiros.

Automaticamente, o array é inicializado com o valor padrão do tipo definido, como aqui utilizamos o `int`, o valor padrão inicial é `0`.

Em seguida, vimos como podemos acessar um array. No caso, fizemos um laço e criamos um mecanismo que nos permite acessar cada posição:

```
public class Teste {  
  
    public static void main(String[] args) {  
        int[] idades = new int[5];  
  
        for(int i = 0; i < idades.length; i++) {  
            idades[i] = i * i;  
        }  
    }  
}
```

[COPIAR CÓDIGO](#)

Utilizamos a referência `idades`, e os colchetes (`[]`), para indicarmos qual posição pretendemos acessar. Importante lembrar que para os arrays, as posições iniciam em `0`, ou seja, a primeira posição é representada pelo número `0`.

Mas não esclarecemos anteriormente, o real significado de `String[]` na assinatura do método `main`.

Temos que ter em mente que `String` é um tipo, uma classe, não um primitivo. Ou seja, o que fazemos ao declarar:

```
//Código omitido  
  
public static void main(String[] args) {  
  
    //Código omitido
```

[COPIAR CÓDIGO](#)

É declarar um array de referência.

Renomearemos a classe `Teste`, para `TesteArrayDePrimitivos`, e em seguida criaremos uma nova classe, um novo teste, chamado `TestArrayReferencias`, com um método `main`:

```
package br.com.bytebank.banco.test;

public class TestArrayReferencias {

    public static void main(String[] args) {

    }

}
```

[COPIAR CÓDIGO](#)

Por que um array de referências? para estarmos preparados caso surja a necessidade de armazenamos diversas contas. E se tivermos 10 contas? onde guardariámos as 10 referências? Uma possibilidade seria guardá-las dentro de um array.

Primeiro, vamos trabalhar com a classe `ContaCorrente`, onde iremos armazenar 10 contas correntes, declararemos então o tipo:

```
package br.com.bytebank.banco.test;

public class TestArrayReferencias {

    public static void main(String[] args) {

        ContaCorrente

    }

}
```

[COPIAR CÓDIGO](#)

Utilizaremos a mesma sintaxe do exemplo anterior, por isso, podemos desde já trazê-la e mantê-la em comentários sobre o tipo `ContaCorrente` :

```
package br.com.bytebank.banco.test;

public class TestArrayReferencias {

    public static void main(String[] args) {

        //int[] idades = new int[5];
        ContaCorrente

    }

}
```

[COPIAR CÓDIGO](#)

Para indicarmos que se trata de um array, incluiremos os colchetes (`[]`), após `ContaCorrente` :

```
package br.com.bytebank.banco.test;

public class TestArrayReferencias {

    public static void main(String[] args) {

        //int[] idades = new int[5];
        ContaCorrente[]

    }

}
```

[COPIAR CÓDIGO](#)

Chamaremos a variável de `contas`. Utilizaremos o `new` para indicar que estamos criando um novo objeto, repetindo o tipo, e os colchetes (`[]`), além do número total de contas que pretendemos armazenar, no caso, 5 :

```
package br.com.bytebank.banco.test;

public class TestArrayReferencias {

    public static void main(String[] args) {

        //int[] idades = new int[5];
        ContaCorrente[] contas = new ContaCorrente[5];

    }

}
```

[COPIAR CÓDIGO](#)

Criamos um objeto que pode guardar cinco referências de contas correntes. Quantas contas foram de fato criadas? nenhuma. Temos por enquanto somente o compartimento capaz de armazená-las.

Dentro deste array não há primitivos, mas podem viver referências, estas por sua vez, serão inicializadas com os valores padrões.

Como criamos uma `ContaCorrente`, qual é o seu valor padrão? No caso, é `null`. Por isso, não podemos dizer que foi criada alguma conta, pois o array não aponta para nenhum objeto.

Em seguida, criaremos uma `ContaCorrente cc1`, com seus respectivos dados de agência e número:

```
package br.com.bytebank.banco.test;
```

```
public class TestArrayReferencias {  
  
    public static void main(String[] args) {  
  
        //int[] idades = new int[5];  
        ContaCorrente[] contas = new ContaCorrente[5];  
  
        ContaCorrente cc1 = new ContaCorrente(22, 11);  
  
    }  
}
```

[COPIAR CÓDIGO](#)

A ideia é que, agora, criamos o nosso primeiro objeto. Temos uma referência `cc1` que aponta para ele. Em seguida, nosso objetivo será armazená-lo na primeira posição em nosso array.

Como acessamos a primeira posição do array? primeiro, utilizamos o nome, em seguida fazemos a referência à posição entre colchetes (`[]`), para então atribuirmos o valor, `cc1`:

```
package br.com.bytebank.banco.test;  
  
public class TestArrayReferencias {  
  
    public static void main(String[] args) {  
  
        //int[] idades = new int[5];  
        ContaCorrente[] contas = new ContaCorrente[5];  
  
        ContaCorrente cc1 = new ContaCorrente(22, 11);  
  
        contas[0] = cc1;  
  
    }  
}
```

```
}
```

[COPIAR CÓDIGO](#)

Internamente, é criada uma cópia do valor `cc1`, que é armazenada na primeira posição e aponta para o objeto.

Criaremos em seguida mais um objeto, `cc2`:

```
package br.com.bytebank.banco.test;

public class TestArrayReferencias {

    public static void main(String[] args) {

        //int[] idades = new int[5];
        ContaCorrente[] contas = new ContaCorrente[5];
        ContaCorrente cc1 = new ContaCorrente(22, 11);
        contas[0] = cc1;

        ContaCorrente cc2 = new ContaCorrente(22, 22);

    }
}
```

[COPIAR CÓDIGO](#)

Temos mais uma referência em nosso código, apontando para este novo objeto. O próximo passo é armazenarmos uma cópia desta cópia dentro do nosso array.

Para acessarmos a segunda posição, utilizamos o número `1`, e atribuímos o valor `cc2`:

```
package br.com.bytebank.banco.test;

public class TestArrayReferencias {

    public static void main(String[] args) {

        //int[] idades = new int[5];
        ContaCorrente[] contas = new ContaCorrente[5];
        ContaCorrente cc1 = new ContaCorrente(22, 11);
        contas[0] = cc1;

        ContaCorrente cc2 = new ContaCorrente(22, 22);
        contas[1] = cc2;

    }

}
```

[COPIAR CÓDIGO](#)

Testaremos nosso código, tentaremos acessar o número da segunda conta a partir do nosso array. Criaremos um `System.out.println()`, com o método `getNumero()`:

```
package br.com.bytebank.banco.test;

public class TestArrayReferencias {

    public static void main(String[] args) {

        //int[] idades = new int[5];
        ContaCorrente[] contas = new ContaCorrente[5];
        ContaCorrente cc1 = new ContaCorrente(22, 11);
        contas[0] = cc1;

        ContaCorrente cc2 = new ContaCorrente(22, 22);
        contas[1] = cc2;
```

```
        System.out.println(cc2.getNumero());  
  
    }  
  
}
```

[COPIAR CÓDIGO](#)

Mas não queremos acessar o objeto diretamente, queremos acessar a cópia que armazenamos. Como podemos fazer isso? Temos de fazer a referência ao array, utilizando a palavra `contas`, e incluir a posição que desejamos acessar, no caso a segunda posição, representada pelo número `1`:

```
package br.com.bytebank.banco.test;  
  
public class TestArrayReferencias {  
  
    public static void main(String[] args) {  
  
        //int[] idades = new int[5];  
        ContaCorrente[] contas = new ContaCorrente[5];  
        ContaCorrente cc1 = new ContaCorrente(22, 11);  
        contas[0] = cc1;  
  
        ContaCorrente cc2 = new ContaCorrente(22, 22);  
        contas[1] = cc2;  
  
        //System.out.println(cc2.getNumero());  
  
        System.out.println(contas[1].getNumero());  
    }  
  
}
```

[COPIAR CÓDIGO](#)

Executaremos e temos o seguinte resultado no console:

Se tentarmos acessar a posição 0 :

```
package br.com.bytebank.banco.test;

public class TestArrayReferencias {

    public static void main(String[] args) {

        //int[] idades = new int[5];
        ContaCorrente[] contas = new ContaCorrente[5];
        ContaCorrente cc1 = new ContaCorrente(22, 11);
        contas[0] = cc1;

        ContaCorrente cc2 = new ContaCorrente(22, 22);
        contas[1] = cc2;

        //System.out.println(cc2.getNumero());

        System.out.println(contas[0].getNumero());
    }

}
```

Temos o seguinte resultado:

Se tentarmos acessar a terceira posição:

```
package br.com.bytebank.banco.test;

public class TestArrayReferencias {

    public static void main(String[] args) {

        //int[] idades = new int[5];
        ContaCorrente[] contas = new ContaCorrente[5];
        ContaCorrente cc1 = new ContaCorrente(22, 11);
        contas[0] = cc1;

        ContaCorrente cc2 = new ContaCorrente(22, 22);
        contas[1] = cc2;

        //System.out.println(cc2.getNumero());

        System.out.println(contas[2].getNumero());
    }

}
```

[COPIAR CÓDIGO](#)

Temos o seguinte resultado:

```
Exception in thread "main" java.lang.NullPointerException
at br.com.bytebank.banco.test.TestArrayReferencias.main(T
```

[COPIAR CÓDIGO](#)

Pois ela ainda não foi inicializada e, por padrão, tem o valor `null`.

Retornaremos para a impressão da segunda posição:

```
package br.com.bytebank.banco.test;

public class TestArrayReferencias {

    public static void main(String[] args) {

        //int[] idades = new int[5];
        ContaCorrente[] contas = new ContaCorrente[5];
        ContaCorrente cc1 = new ContaCorrente(22, 11);
        contas[0] = cc1;

        ContaCorrente cc2 = new ContaCorrente(22, 22);
        contas[1] = cc2;

        //System.out.println(cc2.getNumero());

        System.out.println(contas[1].getNumero());
    }

}
```

COPIAR CÓDIGO

Estamos acessando o `contas[1]` e, em contrapartida, nos será devolvido um valor, que é uma referência. Mas onde ela é armazenada? em uma variável, que por sua vez, tem que ter um tipo.

No nosso caso, o tipo da variável é `ContaCorrente`. Assim, nosso retorno é uma referência do tipo `ContaCorrente`:

```
package br.com.bytebank.banco.test;

public class TestArrayReferencias {

    public static void main(String[] args) {
```

```

//int[] idades = new int[5];
ContaCorrente[] contas = new ContaCorrente[5];
ContaCorrente cc1 = new ContaCorrente(22, 11);
contas[0] = cc1;

ContaCorrente cc2 = new ContaCorrente(22, 22);
contas[1] = cc2;

//System.out.println(cc2.getNumero());

System.out.println(contas[1].getNumero());

ContaCorrente ref = contas[1];
}

}

```

[COPIAR CÓDIGO](#)

Aqui, chamamos nossa referência de `ref`, ela tem o mesmo valor de `cc2`, ou seja, aponta para o objeto `ContaCorrente`.

Assim, podemos utilizar o `ref.getNumero()` para imprimirmos o número da conta. Executaremos:

```

package br.com.bytebank.banco.test;

public class TestArrayReferencias {

    public static void main(String[] args) {

        //int[] idades = new int[5];
        ContaCorrente[] contas = new ContaCorrente[5];
        ContaCorrente cc1 = new ContaCorrente(22, 11);
        contas[0] = cc1;

        ContaCorrente cc2 = new ContaCorrente(22, 22),

```

```
contas[1] = cc2;

//System.out.println(cc2.getNumero());

System.out.println(contas[1].getNumero());

ContaCorrente ref = contas[1];

System.out.println(ref.getNumero());
}

}
```

[COPIAR CÓDIGO](#)

E obtivemos o seguinte resultado:

22

22

[COPIAR CÓDIGO](#)

Indicando que nosso código funcionou.

Este último `System.out.println()` equivale a

`System.out.println(cc2.getNumero())`:

```
package br.com.bytebank.banco.test;
```

```
public class TestArrayReferencias {
```

```
    public static void main(String[] args) {
```

```
        //int[] idades = new int[5];
```

```
        ContaCorrente[] contas = new ContaCorrente[5];
```

```
        ContaCorrente cc1 = new ContaCorrente(22, 11);
```

```
        contas[0] = cc1;
```

```
ContaCorrente cc2 = new ContaCorrente(22, 22);
contas[1] = cc2;

//System.out.println(cc2.getNumero());

System.out.println(contas[1].getNumero());

ContaCorrente ref = contas[1];
System.out.println(cc2.getNumero());
System.out.println(ref.getNumero());
}

}
```

[COPIAR CÓDIGO](#)

Se executarmos, temos o seguinte resultado:

```
22
22
22
```

[COPIAR CÓDIGO](#)

Portanto, quantas contas criamos afinal? Duas. Quantos objetos criamos? Três. Já referências, temos 9, destas, apenas 6 foram inicializadas.

Temos assim um array de referências.

Adiante, falaremos sobre array de referências polimórfico. Até lá!

Forma literal

Até agora vimos a forma "classica" de criar um objeto array usando a palavra chave `new`, por exemplo:

```
int[] numeros = new int[6];
numeros[0] = 1;
numeros[1] = 2;
numeros[2] = 3;
numeros[3] = 4;
numeros[4] = 5;
```

[COPIAR CÓDIGO](#)

No entanto também há uma forma literal. *Literal*, nesse contexto, significa usar valores diretamente, menos burocrático, mais direito. Veja a diferença:

```
int[] refs = {1,2,3,4,5};
```

[COPIAR CÓDIGO](#)

Usamos as chaves `{}` para indicar que se trata de um array e os valores já ficam declarados dentro das chaves.

Array do tipo Object

Transcrição

Você pode fazer o [DOWNLOAD \(<https://caelum-online-public.s3.amazonaws.com/850-java-util/02/java6-cap2.zip>\)](https://caelum-online-public.s3.amazonaws.com/850-java-util/02/java6-cap2.zip) do projeto da aula anterior.

Nesta aula, continuaremos a falar do `java.lang`, com foco na classe `Object`.

Na nossa classe `TestArrayReferencias` criamos um array com cinco contas correntes. Se quiséssemos guardar contas poupanças, poderíamos então criar um novo array para estas. Mas e se quiséssemos guardar contas dos dois tipos? Qual deveria ser o tipo do array nesse caso?

A primeira resposta seria `Conta[]`, que seria mais genérico, assim teríamos o seguinte código:

```
//Código omitido

public class TestArrayReferencias {

    public static void main(String[] args) {

        //int[] idades = new int[5];
        Conta[] contas = new Conta[5];

        //Código omitido
```

COPIAR CÓDIGO

O compilador aponta um erro porque ainda precisamos importar `Conta`, é o que faremos:

```
package br.com.bytebank.banco.test;

import br.com.bytebank.banco.modelo.Conta;
import br.com.bytebank.banco.modelo.ContaCorrente;

public class TestArrayReferencias {

    public static void main(String[] args) {

        //int[] idades = new int[5];
        Conta[] contas = new Conta[5];

        //Código omitido
```

[COPIAR CÓDIGO](#)

Assim, poderíamos apontar tanto para uma `ContaCorrente`, quanto para uma `ContaPoupanca`. Isso não influencia no restante do código, que continua compilando:

```
package br.com.bytebank.banco.test;

import br.com.bytebank.banco.modelo.Conta;
import br.com.bytebank.banco.modelo.ContaCorrente;

public class TestArrayReferencias {

    public static void main(String[] args) {

        //int[] idades = new int[5];
        Conta[] contas = new Conta[5];

        ContaCorrente cc1 = new ContaCorrente(22, 11);
```

```
contas[0] = cc1;

ContaCorrente cc2 = new ContaCorrente(22, 22);
contas[1] = cc2;

//System.out.println(cc2.getNumero());

System.out.println( contas[1].getNumero() );

}

}
```

[COPIAR CÓDIGO](#)

Em seguida, tentaremos criar um objeto do tipo `ContaPoupanca()` :

```
package br.com.bytebank.banco.test;

import br.com.bytebank.banco.modelo.Conta;
import br.com.bytebank.banco.modelo.ContaCorrente;

public class TestArrayReferencias {

    public static void main(String[] args) {

        //int[] idades = new int[5];
        Conta[] contas = new Conta[5];

        ContaCorrente cc1 = new ContaCorrente(22, 11);
        contas[0] = cc1;

        ContaPoupanca cc2 = new ContaPoupanca(22, 22);
        contas[1] = cc2;

        //System.out.println(cc2.getNumero());

        System.out.println( contas[1].getNumero() );
    }
}
```

```
    ContaCorrente ref = contas[1];
    System.out.println(cc2.getNumero());
    System.out.println(ref.getNumero());

}

}
```

[COPIAR CÓDIGO](#)

Isso também funciona, porque nosso array é do tipo mais genérico, `Conta[]` .

Entretanto, em `ContaCorrente ref = contas[1]` o compilador aponta um erro. Por que? Quais são os tipos de referências que o nosso array é capaz de armazenar? Temos que observar a linha `Conta[] contas = new Conta[5]` , a partir da qual concluímos que nosso array é capaz de armazenar referências do tipo `Conta` .

Do jeito como está em nosso código:

```
//Código omitido

ContaCorrente ref = contas[1];

//Código omitido
```

[COPIAR CÓDIGO](#)

A posição 1 guarda uma referência do tipo `ContaCorrente` . Entretanto, a nossa referência aponta para um objeto do tipo `ContaPoupanca` . É possível referenciar um objeto do tipo `ContaPoupanca` por meio de um do tipo `ContaCorrente` ? Não, isto não é possível.

Por isso o compilador reclama, em `contas` :

```
//Código omitido  
  
    ContaCorrente ref = contas[1];
```

//Código omitido

COPIAR CÓDIGO

O computador sabe que a referência `contas`, no array, é do tipo `Conta[]`, por esse motivo, não podemos armazená-la em um tipo `ContaCorrente`.

Para que o código possa compilar, precisamos alterar, de `ContaCorrente`, para `Conta`:

```
package br.com.bytebank.banco.test;  
  
import br.com.bytebank.banco.modelo.Conta;  
import br.com.bytebank.banco.modelo.ContaCorrente;  
  
public class TestArrayReferencias {  
  
    public static void main(String[] args) {  
  
        //int[] idades = new int[5];  
        Conta[] contas = new Conta[5];  
  
        ContaCorrente cc1 = new ContaCorrente(22, 11);  
        contas[0] = cc1;  
  
        ContaPoupanca cc2 = new ContaPoupanca(22, 22);  
        contas[1] = cc2;  
  
        //System.out.println(cc2.getNumero());  
  
        System.out.println( contas[1].getNumero() );
```

```
    Conta ref = contas[1];
    System.out.println(cc2.getNumero());
    System.out.println(ref.getNumero());

}

}
```

[COPIAR CÓDIGO](#)

Assim, o Java consegue garantir que nos devolverá uma referência do tipo `Conta`.

Testaremos o código, executando-o. Temos o seguinte resultado no console:

```
22
22
22
```

[COPIAR CÓDIGO](#)

Como sabemos que a:

```
//Código omitido

    Conta ref = contas[1];

//Código omitido
```

[COPIAR CÓDIGO](#)

Aponta para um objeto do tipo `ContaPoupanca`, se escrevêssemos:

```
package br.com.bytebank.banco.test;

import br.com.bytebank.banco.modelo.Conta;
import br.com.bytebank.banco.modelo.ContaCorrente;
```

```
public class TestArrayReferencias {  
  
    public static void main(String[] args) {  
  
        //int[] idades = new int[5];  
        Conta[] contas = new Conta[5];  
  
        ContaCorrente cc1 = new ContaCorrente(22, 11);  
        contas[0] = cc1;  
  
        ContaPoupanca cc2 = new ContaPoupanca(22, 22);  
        contas[1] = cc2;  
  
        //System.out.println(cc2.getNumero());  
  
        System.out.println( contas[1].getNumero() );  
  
        ContaPoupanca ref = contas[1];  
        System.out.println(cc2.getNumero());  
        System.out.println(ref.getNumero());  
  
    }  
  
}
```

[COPIAR CÓDIGO](#)

Poderia funcionar. Entretanto, vemos que o compilador também não permite. Isso acontece porque o compilador não verifica a linha onde é criado o objeto `ContaPoupanca()`, o que ele faz é verificar o array, e buscar o tipo de referência nele contido.

É do tipo `Conta`, ou seja, mais genérica. Isso significa que ela poderia apontar tanto para `ContaCorrente`, quanto para `ContaPoupanca`. Por esse motivo o compilador aponta o erro, pois diante dessa dualidade, ele não tem certeza se a seguinte linha de código funcionará:

```
//Código omitido  
  
ContaPoupanca ref = contas[1];
```

//Código omitido

COPIAR CÓDIGO

Como nós estamos controlando a execução, sabemos que esta posição do array aponta para o objeto `ContaPoupanca`. `ref` é um objeto do tipo `ContaPoupanca()`, portanto, aponta para o objeto `ContaPoupanca`.

Por isso, queremos informar ao compilador que isso vai funcionar, que nós temos conhecimento de o código irá compilar. Fazemos isso por meio de um **cast** de referências. Transformamos uma referência de um tipo mais genérico, para uma de um tipo mais específico. Isso pode ser chamado também de **type cast**:

```
package br.com.bytebank.banco.test;  
  
import br.com.bytebank.banco.modelo.Conta;  
import br.com.bytebank.banco.modelo.ContaCorrente;  
  
public class TestArrayReferencias {  
  
    public static void main(String[] args) {  
  
        //int[] idades = new int[5];  
        Conta[] contas = new Conta[5];  
  
        ContaCorrente cc1 = new ContaCorrente(22, 11);  
        contas[0] = cc1;  
  
        ContaPoupanca cc2 = new ContaPoupanca(22, 22);  
        contas[1] = cc2;  
  
        //System.out.println(cc2.getNumero());
```

```
System.out.println( contas[1].getNumero() );  
  
ContaPoupanca ref = (ContaPoupanca) contas[1]; //type cas  
System.out.println(cc2.getNumero());  
System.out.println(ref.getNumero());  
  
}  
}
```

[COPIAR CÓDIGO](#)

Testaremos. Executaremos e temos o seguinte resultado no console:

```
22  
22  
22
```

[COPIAR CÓDIGO](#)

Funcionou!

Como sabemos, a referência aponta para um objeto do tipo `ContaPoupanca`.

E se tivéssemos a situação contrária? em vez de `ContaPoupanca`, utilizaremos `ContaCorrente`:

```
package br.com.bytebank.banco.test;  
  
import br.com.bytebank.banco.modelo.Conta;  
import br.com.bytebank.banco.modelo.ContaCorrente;  
  
public class TestArrayReferencias {  
  
    public static void main(String[] args) {
```

```
//int[] idades = new int[5];
Conta[] contas = new Conta[5];

ContaCorrente cc1 = new ContaCorrente(22, 11);
contas[0] = cc1;

ContaPoupanca cc2 = new ContaPoupanca(22, 22);
contas[1] = cc2;

//System.out.println(cc2.getNumero());

System.out.println( contas[1].getNumero() );

ContaCorrente ref = (ContaCorrente) contas[1]; //type cas
System.out.println(cc2.getNumero());
System.out.println(ref.getNumero());

}

}
```

[COPIAR CÓDIGO](#)

Testando, vemos que no console é exibida a exceção de *class cast exception*.

Fizemos um *cast* de uma referência genérica para uma mais específica, entretanto, ele não funcionou. O objeto `ref` é do tipo `ContaPoupanca`, por isso nosso *cast* não funcionando, pois não direcionamos para a referência mais específica correta.

Retornaremos à versão do código que está compilando:

```
package br.com.bytebank.banco.test;

import br.com.bytebank.banco.modelo.Conta;
```

```
import br.com.bytebank.banco.modelo.ContaCorrente;

public class TestArrayReferencias {

    public static void main(String[] args) {

        //int[] idades = new int[5];
        Conta[] contas = new Conta[5];

        ContaCorrente cc1 = new ContaCorrente(22, 11);
        contas[0] = cc1;

        ContaPoupanca cc2 = new ContaPoupanca(22, 22);
        contas[1] = cc2;

        //System.out.println(cc2.getNumero());

        System.out.println( contas[1].getNumero() );

        ContaPoupanca ref = (ContaPoupanca) contas[1]; //type cast
        System.out.println(cc2.getNumero());
        System.out.println(ref.getNumero());

    }

}
```

[COPIAR CÓDIGO](#)

E se quisermos inicializar um array capaz de armazenar qualquer tipo de referência? Precisaríamos do tipo mais genérico possível, neste caso, seria do tipo `Object`.

```
package br.com.bytebank.banco.test;

import br.com.bytebank.banco.modelo.Conta;
import br.com.bytebank.banco.modelo.ContaCorrente;
```

```
public class TestArrayReferencias {  
  
    public static void main(String[] args) {  
  
        //int[] idades = new int[5];  
        Object[] contas = new Object[5];  
  
        ContaCorrente cc1 = new ContaCorrente(22, 11);  
        contas[0] = cc1;  
  
        ContaPoupanca cc2 = new ContaPoupanca(22, 22);  
        contas[1] = cc2;  
  
        //System.out.println(cc2.getNumero());  
  
        System.out.println( contas[1].getNumero() );  
  
        ContaPoupanca ref = (ContaPoupanca) contas[1]; //type cast  
        System.out.println(cc2.getNumero());  
        System.out.println(ref.getNumero());  
  
    }  
  
}
```

[COPIAR CÓDIGO](#)

Nesse array, podemos guardar qualquer objeto. Criaremos um Cliente :

```
package br.com.bytebank.banco.test;  
  
import br.com.bytebank.banco.modelo.Conta;  
import br.com.bytebank.banco.modelo.ContaCorrente;  
  
public class TestArrayReferencias {
```

```
public static void main(String[] args) {

    //int[] idades = new int[5];
    Object[] contas = new Object[5];

    ContaCorrente cc1 = new ContaCorrente(22, 11);
    contas[0] = cc1;

    ContaPoupanca cc2 = new ContaPoupanca(22, 22);
    contas[1] = cc2;

    Cliente cliente = new Cliente();

    //System.out.println(cc2.getNumero());

    System.out.println( contas[1].getNumero() );

    ContaPoupanca ref = (ContaPoupanca) contas[1]; //type cast
    System.out.println(cc2.getNumero());
    System.out.println(ref.getNumero());

}

}
```

[COPIAR CÓDIGO](#)

No array `contas`, no índice `2`, guardaremos a referência que se chama `cliente`:

```
package br.com.bytebank.banco.test;

import br.com.bytebank.banco.modelo.Conta;
import br.com.bytebank.banco.modelo.ContaCorrente;

public class TestArrayReferencias {

    public static void main(String[] args) {
```

```
//int[] idades = new int[5];
Object[] contas = new Object[5];

ContaCorrente cc1 = new ContaCorrente(22, 11);
contas[0] = cc1;

ContaPoupanca cc2 = new ContaPoupanca(22, 22);
contas[1] = cc2;

Cliente cliente = new Cliente();
contas[2] = cliente;

//System.out.println(cc2.getNumero());

System.out.println( contas[1].getNumero() );

ContaPoupanca ref = (ContaPoupanca) contas[1]; //type cast
System.out.println(cc2.getNumero());
System.out.println(ref.getNumero());

}

}
```

[COPIAR CÓDIGO](#)

Entretanto, a linha:

```
//Código omitido

System.out.println( contas[1].getNumero() );

//Código omitido
```

[COPIAR CÓDIGO](#)

Parou de funcionar. Isso porque o tipo de referência no índice 1 é `Object`, e utilizamos esta referência genérica para chamar o método `getNumero()`. Na classe `Object` existe este método? Não existe. Por esse motivo, temos um erro de compilação.

Isso só funcionará se fizermos um **cast**, ou seja, se transformarmos uma referência genérica em uma específica. Comentaremos as linhas de código referentes a esta seção que não está compilando:

```
package br.com.bytebank.banco.test;

import br.com.bytebank.banco.modelo.Conta;
import br.com.bytebank.banco.modelo.ContaCorrente;

public class TestArrayReferencias {

    public static void main(String[] args) {

        //int[] idades = new int[5];
        Object[] contas = new Object[5];

        ContaCorrente cc1 = new ContaCorrente(22, 11);
        contas[0] = cc1;

        ContaPoupanca cc2 = new ContaPoupanca(22, 22);
        contas[1] = cc2;

        Cliente cliente = new Cliente();
        contas[2] = cliente;

        //System.out.println(cc2.getNumero());

        //Object referenciaGenerica = contas[1];
        //System.out.println( referenciaGenerica.getNumero() )
```

```
    ContaPoupanca ref = (ContaPoupanca) contas[1]; //type cas
    System.out.println(cc2.getNumero());
    System.out.println(ref.getNumero());

}

}
```

[COPIAR CÓDIGO](#)

Renomearemos a variável `contas`, clicando com o botão direito do mouse sobre ela, e selecionando a opção "Refactor > Rename...", ela passará a se chamar `referencias`:

```
//Código omitido

public class TestArrayReferencias {

    public static void main(String[] args) {

        //int[] idades = new int[5];
        Object[] referencias = new Object[5];

        ContaCorrente cc1 = new ContaCorrente(22, 11);
        referencias[0] = cc1;

        ContaPoupanca cc2 = new ContaPoupanca(22, 22);
        referencias[1] = cc2;

        Cliente cliente = new Cliente();
        referencias[2] = cliente;

        //System.out.println(cc2.getNumero());

        //Object referenciaGenerica = contas[1];

        //System.out.println( referenciaGenerica.getNumero() );
```

```
ContaPoupanca ref = (ContaPoupanca) contas[1]; //type cast
System.out.println(cc2.getNumero());
System.out.println(ref.getNumero());

}

}
```

[COPIAR CÓDIGO](#)



Assim, nos aproximamos do resultado final. Adiante, falaremos mais sobre o `String[]`.

Cast explícito e implícito

Cast explícito e implícito

Já falamos bastante sobre o *Type Cast* que é nada mais do que a conversão de um tipo para outro.

Cast implícito e explícito de primitivos

Para ser correto, já vimos o cast acontecendo antes mesmo de defini-lo. Temos dois exemplos, o primeiro do mundo de primitivos:

```
int numero = 3;  
double valor = numero; //cast implícito
```

[COPIAR CÓDIGO](#)

Repare que colocamos um valor da variável `numero` (tipo `int`) na variável `valor` (tipo `double`), sem usar um cast explícito. Isso funciona? A resposta é sim, pois qualquer inteiro cabe dentro de um double. Por isso o compilador fica quieto e não exige um *cast explicito*, mas nada impede de escrever:

```
int numero = 3;  
double valor = (double) numero; //cast explícito
```

[COPIAR CÓDIGO](#)

Agora, o contrário não funciona sem cast, uma vez que um `double` não cabe em um `int`:

```
double valor = 3.56;  
int numero = (int) valor; //cast explicito é exigido pelo comp:
```

[COPIAR CÓDIGO](#)



Nesse caso o compilador joga todo valor fracional fora e guarda apenas o valor inteiro.

Cast implícito e explícito de referências

Nas referências, o mesmo princípio se aplica. Se o cast sempre funciona não é necessário deixá-lo explícito, por exemplo:

```
ContaCorrente cc1 = new ContaCorrente(22, 33);  
Conta conta = cc1; //cast implicito
```

[COPIAR CÓDIGO](#)

Aqui também poderia ser explícito, mas novamente, o compilador não exige pois qualquer `ContaCorrente` é uma `Conta`:

```
ContaCorrente cc1 = new ContaCorrente(22, 33);  
Conta conta = (Conta) cc1; //cast explicito mas desnecessário
```

[COPIAR CÓDIGO](#)



Cast possível e impossível

Type cast explícito sempre funciona?

A resposta é não. O cast explícito só funciona se ele for *possível*, mas há casos em que o compilador sabe que um cast é impossível e aí nem compila, nem com *type cast*. Por exemplo:

```
Cliente cliente = new Cliente();
Conta conta = (Conta) cliente; //impossível, não compila
```

[COPIAR CÓDIGO](#)

Como o cliente não estende a classe `Conta` ou implementa uma interface do tipo `Conta`, é impossível esse *cast* funcionar, pois uma referência do tipo `Conta` jamais pode apontar para um objeto do tipo `Cliente`.

A certificação Java tem muitas dessas perguntas sobre *cast* possível, impossível, explícito e implícito. Se você pretende tirar essa certificação, vale a pena estudar esse assunto com muita calma.

Entendendo o array String args

Transcrição

Olá! Nesta aula daremos continuidade à explicação sobre o `String[]` :

```
//Código omitido, pacote e imports

public class TestArrayReferencias {

    public static void main(String[] args) {

//Código omitido
```

[COPIAR CÓDIGO](#)

Criaremos uma nova classe, chamada `TestArrayList` :

```
package br.com.bytebank.banco.test;

import br.com.bytebank.banco.modelo.Cliente;

public class TestArrayList {

    public static void main(String[] args) {

}
```

[COPIAR CÓDIGO](#)

Como sabemos, é possível executarmos uma aplicação Java no terminal. Lá, utilizamos a máquina virtual (JVM), seguido do nome da classe, para que possamos executá-la. Mas primeiro, é preciso acessarmos a pasta do Eclipse, digitando o comando `cd eclipse-workspace/`, assim serão listadas todas as pastas onde são guardados os projetos:

```
Aluras-iMac:~alura$ cd eclipse-workspace/  
Aluras-iMac:eclipse-workspace alura$ ls  
bytebank-biblioteca bytebank-herdado java-pilha  
bytebank-encapsulado bytebank-herdado-conta java-stack  
Aluras-iMac:eclipse-workspace alura$
```

[COPIAR CÓDIGO](#)

E como podemos ver, temos o nosso `bytebank-herdado-conta`. Acessaremos esta pasta, digitando o comando `cd bytebank-herdado-conta/`:

```
Aluras-iMac:~alura$ cd eclipse-workspace/  
Aluras-iMac:eclipse-workspace alura$ ls  
bytebank-biblioteca bytebank-herdado java-pilha  
bytebank-encapsulado bytebank-herdado-conta java-stack  
Aluras-iMac:eclipse-workspace alura$ cd bytebank-herdado-conta/  
Aluras-iMac:bytebank-herdado-conta alura$
```

[COPIAR CÓDIGO](#)

As classes compiladas, por sua vez, estão na pasta `bin`:

```
Aluras-iMac:bytebank-herdado-conta alura$ ls  
bin doc src  
Aluras-iMac:bytebank-herdado-conta alura$ cd bin
```

[COPIAR CÓDIGO](#)

Temos o código fonte na pasta `src`, e o Eclipse gera automaticamente a pasta `bin`. Além disso, temos também a pasta `br`:

```
Aluras-iMac:bytebank-herdado-conta alura$ ls  
bin doc src  
Aluras-iMac:bytebanks-herdado-conta alura$ cd bin  
Aluras-iMac:bin alura$ ls  
br
```

[COPIAR CÓDIGO](#)

Agora, podemos inicializar a máquina virtual e tentar chamar a classe `TestArrayList`. Retornando ao Eclipse, clicaremos com o botão direito sobre o seu nome, e selecionaremos a opção "Copy Qualified Name". No terminal, colaremos o nome da classe:

```
Aluras-iMac:bytebank-herdado-conta alura$ ls  
bin doc src  
Aluras-iMac:bytebanks-herdado-conta alura$ cd bin  
Aluras-iMac:bin alura$ ls  
br  
Aluras-iMac:bin alura$ java br.com.bytebank.banco.test.TestArrayList
```

[COPIAR CÓDIGO](#)

Precisamos chamar a classe, sendo que é este o seu nome verdadeiro.

Retornando ao Eclipse, faremos um `System.out.println()` com a mensagem "Funcionou!!", apenas para nos certificarmos:

```
package br.com.bytebank.banco.test;  
  
public class TestArrayList {
```

```
public static void main(String[] args) {  
  
    System.out.println("Funcionou!!");  
}  
  
}
```

[COPIAR CÓDIGO](#)

Salvaremos e retornaremos ao terminal. Temos o seguinte resultado:

```
Aluras-iMac:bytebank-herdado-conta alura$ ls  
bin  doc  src  
Aluras-iMac:bytebanks-herdado-conta alura$ cd bin  
Aluras-iMac:bin alura$ ls  
br  
Aluras-iMac:bin alura$ java br.com.bytebank.banco.test.TestArrays  
Funcionou!!  
Aluras-iMac:bin alura$
```

[COPIAR CÓDIGO](#)

Por que então existe o `String[]` ao declararmos o método `main`? Ele existe para que possamos passar parâmetros a partir do terminal, por exemplo, passaremos `1 2 oi nico java`, ou seja, 5 parâmetros:

```
Aluras-iMac:bytebank-herdado-conta alura$ ls  
bin  doc  src  
Aluras-iMac:bytebanks-herdado-conta alura$ cd bin  
Aluras-iMac:bin alura$ ls  
br  
Aluras-iMac:bin alura$ java br.com.bytebank.banco.test.TestArrays  
Funcionou!!  
Aluras-iMac:bin alura$ java br.com.bytebank.banco.test.TestArrays
```

[COPIAR CÓDIGO](#)

Retornando ao Eclipse, acessaremos este string em nossa classe. Criaremos um laço `for`, em cima deste array de strings. Dentro do laço, imprimiremos o array na posição da iteração, por meio da variável `i`:

```
package br.com.bytebank.banco.test;

public class TestArrayList {
    public static void main(String[] args) {
        System.out.println("Funcionou!!");

        for(int i = 0; i < args.length; i++) {
            System.out.println(args[i]);
        }
    }
}
```

[COPIAR CÓDIGO](#)

Salvaremos e retornaremos ao terminal, testaremos a mesma linha que anteriormente, e temos o seguinte resultado:

```
Aluras-iMac:bytebank-herdado-conta alura$ ls
bin  doc  src
Aluras-iMac:bytebanks-herdado-conta alura$ cd bin
Aluras-iMac:bin alura$ ls
br
Aluras-iMac:bin alura$ java br.com.bytebank.banco.test.TestArrayList
Funcionou!!
Aluras-iMac:bin alura$ java br.com.bytebank.banco.test.TestArrayList
Funcionou!!
Aluras-iMac:bin alura$ java br.com.bytebank.banco.test.TestArrayList
Funcionou!!
```

```
1  
2  
oi  
nico  
java  
Aluras-iMac:bin alura$
```

[COPIAR CÓDIGO](#)

Foram impressos todos os parâmetros que havíamos passado anteriormente.

Assim, temos uma forma de interagir com o programa Java a partir da linha de comando. Chamamos a aplicação e, simultaneamente, é possível definirmos parâmetros.

Isso tem várias utilidades. Pode servir, por exemplo, para habilitar alguma funcionalidade, entre várias outras configurações possíveis.

Assim como testamos no terminal, é possível testarmos também no Eclipse. Rodaremos a classe no programa e temos o seguinte resultado no console:

Funcionou!!

[COPIAR CÓDIGO](#)

O Eclipse não passou nenhum parâmetro, e por isso nenhum foi exibido. No Eclipse, no botão verde com o símbolo de play, localizado na barra superior, há na direita uma seta menor, apontando para baixo. Clicando nela, na opção "Run Configurations" é possível manipularmos as configurações, ou seja, como a máquina virtual do Java chamará a nossa classe.



Na lateral esquerda, há uma lista com todas as *run configurations* que já foram utilizadas para rodar os programas em nossa máquina.

Na parte superior temos diversas abas, uma delas é a `Arguments`, ou seja, argumentos. Nela, podemos inserir os parâmetros:

```
1 2 oi nico java rocks e eh legal
```

[COPIAR CÓDIGO](#)

Clicamos em "Apply" e "Run". Temos o seguinte resultado no console:

```
Funcionou!!
```

```
1  
2  
oi  
nico  
java  
rocks  
e  
eh  
legal
```

[COPIAR CÓDIGO](#)

Todos os parâmetros passados.

Agora já vimos os array de strings, bem como array de referências. Adiante, criaremos um array de objetos e falaremos sobre os métodos genéricos da classe `Object`, especialmente o `equals()`.

Adapter para arrays

Transcrição

Você pode fazer o [DOWNLOAD \(<https://caelum-online-public.s3.amazonaws.com/850-java-util/03/java6-cap3.zip>\)](https://caelum-online-public.s3.amazonaws.com/850-java-util/03/java6-cap3.zip) do projeto da aula anterior.

Olá!

Anteriormente, aprendemos a lidar com arrays, agora, entendemos tudo que está escrito na classe ao chamar o método `main`, inclusive o `String[]`:

```
//Código omitido, pacote e import

public class TestArrayReferencias {

    public static void main(String[] args) {

        //Código omitido
```

COPIAR CÓDIGO

Passaremos agora a trabalhar com a sintaxe utilizada para criar um objeto comum. Até o momento, utilizamos a seguinte:

```
//Código omitido, pacote e import

public class TestArrayReferencias {
```

```
public static void main(String[] args ) {  
  
    //int[] iaddes = new int[5];  
    Object[] referencias = new Object[5];  
  
    ContaCorrente cc1 = new ContaCorrente(22, 11);  
    referencias[0] = cc1;  
  
    //Código omitido
```

[COPIAR CÓDIGO](#)

Temos no lado esquerdo, a referência e o tipo, e no lado direito o `new`, que chama o construtor, com seus parâmetros, caso haja - este é o padrão para qualquer padrão que queiramos criar, exceto pelo array, que possui uma sintaxe própria.

No array, utilizamos os colchetes (`[]`), e no lado direito utilizamos o `new` e chamamos algo análogo a um construtor mas que, dentro de colchetes (`[]`), recebe o tamanho do referido array.

Outro ponto que aprendemos é que, para descobrirmos o tamanho de um array, utilizamos o `referencias.length`, que é um atributo público. Estes são geralmente considerados como uma má prática.

Além disso, normalmente, não queremos saber quantas referências um array pode guardar. No nosso exemplo, são 5, mas o importante é sabermos quantas referências um array de fato já guardou, neste caso, seriam 2, representadas por `cc1` e `cc2`.

E se criarmos um array com 5 posições, para guardar 5 referências, mas no decorrer do projeto percebermos a necessidade de guardar mais 5. O array não é dinâmico, ou seja, uma vez criado com um determinado número de posições, ele só terá aquele número. Assim, precisamos descobrir uma forma de trabalharmos com esta ferramenta mais elegantemente.

Inicialmente, criaremos um classe que esconde o acesso ao array, utilizando-o internamente, e as demais classes a utilizarão.

A nova classe se chamará `GuardadorDeContas` :

```
package br.com.bytebank.banco.modelo;

public class GuardadorDeContas {
```

}

[COPIAR CÓDIGO](#)

Internamente, teremos um array de contas, e um construtor para o inicializarmos:

```
package br.com.bytebank.banco.modelo;

public class GuardadorDeContas {

    private Conta[] referencias;

    public GuardadorDeContas() {
        this.referencias = new Conta[10];
    }

}
```

[COPIAR CÓDIGO](#)

No pacote `br.com.bytebank.banco.test` criaremos um novo teste, chamado `Teste` :

```
package br.com.bytebank.banco.test;

public class Teste {
```

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

[COPIAR CÓDIGO](#)

Utilizaremos o `GuardadorDeContas`, com a sintaxe padrão:

```
package br.com.bytebank.banco.test;  
  
public class Teste {  
  
    public static void main(String[] args) {  
  
        GuardadorDeContas guardador = new GuardadorDeCont  
  
    }  
}
```

[COPIAR CÓDIGO](#)

Em seguida, utilizaremos o `guardador` para adicionarmos uma `conta`, ou seja, para passarmos uma referência. Para isso, criaremos também um objeto `Conta()`:

```
package br.com.bytebank.banco.test;  
  
public class Teste {  
  
    public static void main(String[] args) {  
  
        GuardadorDeContas guardador = new GuardadorDeCont  
  
        Conta cc = new ContaCorrente(22, 11);  
    }  
}
```

```
        guardador.adiciona(cc);  
  
    }  
}
```

[COPIAR CÓDIGO](#)

A cada vez que quisermos guardar uma referência, chamaremos o método `adiciona`, não será necessário sabermos a posição do array, como anteriormente.

Na classe `GuardadorDeContas()` criaremos este método `adiciona`:

```
package br.com.bytebank.banco.modelo;  
  
public class GuardadorDeContas {  
  
    private Conta[] referencias;  
  
    public GuardadorDeContas() {  
        this.referencias = new Conta[10];  
    }  
  
    public void adiciona();  
  
}
```

[COPIAR CÓDIGO](#)

Este método deve receber uma referência do tipo `Conta`:

```
package br.com.bytebank.banco.modelo;  
  
public class GuardadorDeContas {  
  
    private Conta[] referencias;
```

```
public GuardadorDeContas() {  
    this.referencias = new Conta[10];  
}  
  
public void adiciona(Conta ref) {  
  
}  
}
```

[COPIAR CÓDIGO](#)

Assim, deve ser possível armazenarmos a referência na primeira posição disponível em nosso array. É o que inseriremos no corpo do método:

```
package br.com.bytebank.banco.modelo;  
  
public class GuardadorDeContas {  
  
    private Conta[] referencias;  
  
    public GuardadorDeContas() {  
        this.referencias = new Conta[10];  
    }  
  
    public void adiciona(Conta ref) {  
        this.referencias[0] = ref;  
    }  
}
```

[COPIAR CÓDIGO](#)

Isso funciona quando estamos trabalhando com uma conta, mas e se criássemos mais uma na classe `Teste` ?

```
package br.com.bytebank.banco.test;

public class Teste {

    public static void main(String[] args) {

        GuardadorDeContas guardador = new GuardadorDeContas();

        Conta cc = new ContaCorrente(22, 11);
        guardador.adiciona(cc);

        Conta cc2 = new ContaCorrente(22, 22);
        guardador.adiciona(cc2);

    }
}
```

[COPIAR CÓDIGO](#)

O que aconteceria se tentássemos armazená-la na próxima posição livre?

No `GuardadorDeContas` a posição `0` está fixa. isso não funcionará.

Precisaremos de uma variável auxiliar, que sempre lembrará da última posição utilizada, ela se chamará `posicaoLivre`:

```
package br.com.bytebank.banco.modelo;

public class GuardadorDeContas {

    private Conta[] referencias;
    private int posicaoLivre;

    public GuardadorDeContas() {
        this.referencias = new Conta[10];
    }
}
```

```
public void adiciona(Conta ref) {  
    this.referencias[0] = ref;  
}  
}
```

[COPIAR CÓDIGO](#)

Inicializaremos a variável `posicaoLivre` dentro do método `GuardadorDeContas()`, com o valor `0`:

```
package br.com.bytebank.banco.modelo;  
  
public class GuardadorDeContas {  
  
    private Conta[] referencias;  
    private int posicaoLivre;  
  
    public GuardadorDeContas() {  
        this.referencias = new Conta[10];  
        this.posicaoLivre = 0;  
    }  
  
    public void adiciona(Conta ref) {  
        this.referencias[0] = ref;  
    }  
}
```

[COPIAR CÓDIGO](#)

A inicialização não é obrigatória, já que por padrão, todos os atributos do tipo `int` são inicializados com o valor `0`.

No método `adiciona()`, em vez de utilizarmos o `0`, agora teremos o `this.posicaoLivre`. Além disso, para que não tenhamos a posição fixa,

incrementaremos esta variável:

```
package br.com.bytebank.banco.modelo;

public class GuardadorDeContas {

    private Conta[] referencias;
    private int posicaoLivre;

    public GuardadorDeContas() {
        this.referencias = new Conta[10];
        this.posicaoLivre = 0;
    }

    public void adiciona(Conta ref) {
        this.referencias[this.posicaoLivre] = ref;
        this.posicaoLivre++;
    }

}
```

[COPIAR CÓDIGO](#)

Para testarmos se tudo está funcionando, teremos na classe `Teste` uma variável `tamanho` que receberá um método `guardador.getQuantidadeDeElementos()`, para sabermos o número de itens inicializados:

```
//Código omitido

public class Teste {

    public static void main(String[] args) {

        GuardadorDeContas guardador = new GuardadorDeCont

        Conta cc = new ContaCorrente(22, 11);
```

```
guardador.adiciona(cc);

Conta cc2 = new ContaCorrente(22, 22);
guardador.adiciona(cc2);

int tamanho = guardador.getQuantidadeDeElementos()

}

}
```

[COPIAR CÓDIGO](#)

Como criamos duas contas, o resultado correto deve ser 2. Criaremos uma impressão do resultado:

```
//Código omitido

public class Teste {

    public static void main(String[] args) {

        GuardadorDeContas guardador = new GuardadorDeCont

        Conta cc = new ContaCorrente(22, 11);
        guardador.adiciona(cc);

        Conta cc2 = new ContaCorrente(22, 22);
        guardador.adiciona(cc2);

        int tamanho = guardador.getQuantidadeDeElementos();
        System.out.println(tamanho);
    }
}
```

[COPIAR CÓDIGO](#)

Nos resta agora criar de fato este método `getQuantidadeDeElementos()`.

No Eclipse, ao repousarmos o cursor sobre o nome do nosso método, ele nos oferece uma opção "*Create method getQuantidadeDeElementos in type GuardadorDeContas*", clicaremos nela.

Na classe `GuardadorDeContas` temos o seguinte resultado:

```
//Código omitido

public class GuardadorDeContas {

    private Conta[] referencias;
    private int posicaoLivre;

    public GuardadorDeContas() {
        this.referencias = new Conta[10];
        this.posicaoLivre = 0;
    }

    public void adiciona(Conta ref) {
        this.referencias[this.posicaoLivre] = ref;
        this.posicaoLivre++;
    }

    public int getQuantidadeDeElementos() {
        //TODO Auto-generated methos stub
        return 0;
    }

}
```

COPIAR CÓDIGO

Ou seja, o método foi adicionado automaticamente na classe adequada, entretanto, sem a implementação. É o que faremos a seguir:

```
//Código omitido

public class GuardadorDeContas {

    private Conta[] referencias;
    private int posicaoLivre;

    public GuardadorDeContas() {
        this.referencias = new Conta[10];
        this.posicaoLivre = 0;
    }

    public void adiciona(Conta ref) {
        this.referencias[this.posicaoLivre] = ref;
        this.posicaoLivre++;
    }

    public int getQuantidadeDeElementos() {
        return this.posicaoLivre;
    }

}
```

[COPIAR CÓDIGO](#)

Isso nos dará o número de elementos inicializados já que, a cada elemento adicionado, esta variável é incrementada.

Retornaremos à classe `Teste` e executaremos nosso código. Temos o seguinte resultado no console:

2

[COPIAR CÓDIGO](#)

Já sabemos como obter a quantidade de elementos, queremos então que o programa nos devolva um elemento em si. Faremos com que seja possível receber

a referência de uma conta.

Na classe `Teste`, criaremos uma `ref` do tipo `Conta` que receberá um método `guardador.getReferencia()`. O parâmetro será uma posição, e toda posição é um número inteiro, portanto, ela será inicializada em `0`:

```
//Código omitido

public class Teste {

    public static void main(String[] args) {

        GuardadorDeContas guardador = new GuardadorDeCont

        Conta cc = new ContaCorrente(22, 11);
        guardador.adiciona(cc);

        Conta cc2 = new ContaCorrente(22, 22);
        guardador.adiciona(cc2);

        int tamanho = guardador.getQuantidadeDeElementos(
            System.out.println(tamanho);

        Conta ref = guardador.getReferencia(0);
    }
}
```

[COPIAR CÓDIGO](#)

Utilizaremos o mesmo atalho aplicado anteriormente, para criarmos o método `getReferencia` na classe `GuardadorDeContas`. Como implementação, retornaremos o array `referencia` na posição `pos`, e assim acessaremos o valor lá contido:

```
//Código omitido

public class GuardadorDeContas {

    private Conta[] referencias;
    private int posicaoLivre;

    public GuardadorDeContas() {
        this.referencias = new Conta[10];
        this.posicaoLivre = 0;
    }

    public void adiciona(Conta ref) {
        this.referencias[this.posicaoLivre] = ref;
        this.posicaoLivre++;
    }

    public int getQuantidadeDeElementos() {
        return this.posicaoLivre;
    }

    public Conta getReferencia(int pos) {
        return this.referencias[pos];
    }
}
```

[COPIAR CÓDIGO](#)

Retornando à classe `Teste`, vemos que o código inteiro está compilando, sem que tenhamos utilizado a sintaxe específica dos arrays:

```
//Código omitido

public class Teste {

    public static void main(String[] args) {
```

```
GuardadorDeContas guardador = new GuardadorDeCont  
  
Conta cc = new ContaCorrente(22, 11);  
guardador.adiciona(cc);  
  
Conta cc2 = new ContaCorrente(22, 22);  
guardador.adiciona(cc2);  
  
int tamanho = guardador.getQuantidadeDeElementos()  
System.out.println(tamanho);  
  
Conta ref = guardador.getReferencia(0);  
}  
}
```

[COPIAR CÓDIGO](#)



Para testarmos, imprimiremos a referência:

```
//Código omitido  
  
public class Teste {  
  
    public static void main(String[] args) {  
  
        GuardadorDeContas guardador = new GuardadorDeCont  
  
        Conta cc = new ContaCorrente(22, 11);  
        guardador.adiciona(cc);  
  
        Conta cc2 = new ContaCorrente(22, 22);  
        guardador.adiciona(cc2);  
  
        int tamanho = guardador.getQuantidadeDeElementos()  
        System.out.println(tamanho);  
    }  
}
```

```
    Conta ref = guardador.getReferencia(0);
    System.out.println(ref.getNumero());
}
}
```

[COPIAR CÓDIGO](#)

Ele deveria nos devolver 11 . Executaremos e obtemos o seguinte resultado:

```
2
11
```

[COPIAR CÓDIGO](#)

Podemos repetir o teste com outra conta, e ele funcionará igualmente. Adiante, veremos como criar um guardador capaz de armazenar qualquer tipo de referência, utilizando a classe Object .

Conhecendo ArrayList

Transcrição

Dando continuidade às aulas anteriores, veremos como podemos transformar o `GuardadorDeContas` em um guardador de referências genéricas, ou seja, em vez de guardarmos somente tipo `Conta`, armazenaremos qualquer tipo de `Object`.

Para exemplificar, já temos criada a classe `GuardadorDeReferencias`:

```
package br.com.bytebank.banco.modelo;

public class GuardadorDeReferencias {

    private Object[] referencias;
    private int posicaoLivre;

    public GuardadorDeReferencias() {
        this.referencias = new Object[10];
        this.posicaoLivre = 0;
    }

    public void adiciona(Object ref) {
        this.referencias[this.posicaoLivre] = ref;
        this.posicaoLivre++;
    }

    public int getQuantidadeDeElementos() {
        return this.posicaoLivre;
    }
}
```

```
public Object getReferencia(int pos) {  
    return this.referencias[pos];  
}
```

[COPIAR CÓDIGO](#)

Além disso, com relação à classe `GuardadorDeContas`, ela foi refatorada para alterarmos o nome, que passou a ser `GuardadorDeReferencias`.

A única diferença entre as duas é que, onde tínhamos `GuardadorDeContas` em uma, teremos na outra `GuardadorDeReferencias`, e onde havia `Conta` em uma, na outra teremos `Object`. Desta forma, a classe passou a ser capaz de armazenar referências genéricas.

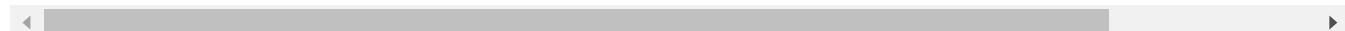
Criamos também uma classe `TesteGuardadorReferencias`:

```
package br.com.bytebank.banco.test;  
  
import br.com.bytebank.banco.modelo.Conta;  
  
public class TesteGuardadorReferencias {  
  
    public static void main(String[] args) {  
  
        GuardadorDeReferencias guardador = new GuardadorDeRef  
  
        Conta cc = new ContaCorrente(22, 11);  
        guardador.adiciona(cc);  
  
        Conta cc2 = new ContaCorrente(22, 22);  
        guardador.adiciona(cc2);  
  
        int tamanho = guardador.getQuantidadeDeElementos();  
        System.out.println(tamanho);  
  
        Conta ref = (Conta) guardador.getReferencia(1);
```

```
System.out.println(ref.getNumero());
```

```
}
```

[COPIAR CÓDIGO](#)



Ela é praticamente idêntica à criada para o `GuardadorDeContas`, inclusive, estamos trabalhando com contas, uma vez que o `GuardadorDeReferencias` é genérico e, portanto, também consegue guardá-las.

O objeto `GuardadorDeReferencias` foi instanciado e, além disso, foi feito um cast do tipo `Conta` para que pudéssemos ter este retorno. Isso porque exigimos uma referência mais específica.

Nosso problema agora é a falta de dinamicidade do array, no momento estamos guardando 10 referências, mas e se quisermos guardar mais? Ao adicionarmos mais elementos deveríamos verificar a capacidade de armazenamento, criar um array maior, e copiar os elementos do antigo para este novo.

E se quisermos remover referências? Teríamos que criar um método específico. Poderíamos ter também um método que nos permitisse adicionar diversas referências de uma só vez.

Há diversas possibilidades. Mas será que outras pessoas já não pensaram nelas antes? Sim.

Certamente, existem classes que utilizam arrays mas possuem recursos de mais alto nível para não trabalharem com colchetes (`[]`), e outras especificidades dos arrays.

Com isso, podemos fechar os demais pacotes e entraremos no tópico do `java.util`.

Trata-se de um pacote específico, que você deve estudar isoladamente. Quanto maior o domínio sobre ele, mais útil será no dia-a-dia.

Para criarmos um novo pacote, podemos clicar com o botão direito sobre a pasta `src` no Eclipse e selecionaremos a opção "New > Package". Entretanto, dificilmente isso é feito na prática.

Há outra forma, mais utilizada - por ser mais fácil -, que nos permite criar não só um pacote, como também uma classe.

Criaremos uma nova classe no pacote `br.com.bytebank-banco.test`, e na tela de criação podemos alterar o pacote, na opção "Package", podemos digitar `br.com.bytebank-banco.test.util`. Assim temos o novo pacote.

A classe se chamará `Teste`, e terá um método `main()` padrão:

```
package br.com.bytebank.banco.test.util;

public class Teste {

    public static void main(String[] args) {

    }
}
```

[COPIAR CÓDIGO](#)

Nosso `GuardadorDeReferencias` funciona mas, como sabemos, há uma classe mais sofisticada que existe para este mesmo propósito, ela se chama `ArrayList`. Aprenderemos aqui a primeira estrutura de dados.

Precisamos importá-la do pacote `java.util`, para isso, pousaremos o cursor sobre o seu nome, e clicaremos na opção sugerida pelo Eclipse "Import `ArrayList` (`java.util`)". Temos o seguinte resultado:

```
package br.com.bytebank.banco.test.util;

public class Teste {

    public static void main(String[] args) {

        ArrayList lista;

    }
}
```

[COPIAR CÓDIGO](#)

Criaremos um novo objeto nesta classe, utilizando o `new` e o construtor `ArrayList()`. O Eclipse deixa o código sublinhado em amarelo e, por enquanto, não nos preocuparemos com isso:

```
package br.com.bytebank.banco.test.util;

public class Teste {

    public static void main(String[] args) {

        ArrayList lista = new ArrayList();

    }
}
```

[COPIAR CÓDIGO](#)

O `ArrayList()` será nosso guardador de referências, ao ser executado, internamente, ele utiliza um array com um número pré-determinado de posições, que gira em torno de 1000.

No `ArrayList` incluiremos as mesmas duas contas que havíamos criado anteriormente. A diferença é que agora guardamos em `lista` e o método tem

nome em inglês, add :

```
package br.com.bytebank.banco.test.util;

public class Teste {

    public static void main(String[] args) {

        ArrayList lista = new ArrayList();

        Conta cc = new ContaCorrente(22, 11);
        lista.add(cc);

        Conta cc2 = new ContaCorrente(22, 22);
        lista.add(cc2);

    }
}
```

[COPIAR CÓDIGO](#)

A seguir, queremos saber do `ArrayList` quantas referências foram armazenadas até o momento. Para isso, imprimiremos o método `lista.size()`, para obtermos justamente o seu tamanho:

```
package br.com.bytebank.banco.test.util;

public class Teste {

    public static void main(String[] args) {

        ArrayList lista = new ArrayList();

        Conta cc = new ContaCorrente(22, 11);
        lista.add(cc);

    }
}
```

```
    Conta cc2 = new ContaCorrente(22, 22);
    lista.add(cc2);

    System.out.println(lista.size());

}
```

[COPIAR CÓDIGO](#)

Executaremos a classe `Teste`, e temos o seguinte resultado:

2

[COPIAR CÓDIGO](#)

O que significa que funcionou.

Podemos também criar um método para que seja exibido um dos elementos do array. Para isso, temos o `get(index)`, onde podemos passar a posição desejada entre parênteses `()`.

Nosso retorno será do tipo `Object`, já que temos um array de referências genéricas. Assim, temos:

```
package br.com.bytebank.banco.test.util;

public class Teste {

    public static void main(String[] args) {

        ArrayList lista = new ArrayList();

        Conta cc = new ContaCorrente(22, 11);
        lista.add(cc);
```

```
    Conta cc2 = new ContaCorrente(22, 22);
    lista.add(cc2);

    System.out.println(lista.size());

    Object ref = lista.get(0);

}

}
```

[COPIAR CÓDIGO](#)

Como sabemos que, neste caso, o elemento é uma conta, podemos transformar o tipo mais genérico em específico, fazendo um *type cast* para `Conta` :

```
package br.com.bytebank.banco.test.util;

public class Teste {

    public static void main(String[] args) {

        ArrayList lista = new ArrayList();

        Conta cc = new ContaCorrente(22, 11);
        lista.add(cc);

        Conta cc2 = new ContaCorrente(22, 22);
        lista.add(cc2);

        System.out.println(lista.size());

        Conta ref = (Conta) lista.get(0);

    }
}
```

[COPIAR CÓDIGO](#)

Para visualizarmos se o método funcionou, imprimiremos o número da conta, utilizando o método `getNumero()` :

```
package br.com.bytebank.banco.test.util;

public class Teste {

    public static void main(String[] args) {

        ArrayList lista = new ArrayList();

        Conta cc = new ContaCorrente(22, 11);
        lista.add(cc);

        Conta cc2 = new ContaCorrente(22, 22);
        lista.add(cc2);

        System.out.println(lista.size());

        Conta ref = (Conta) lista.get(0);

        System.out.println(ref.getNumero());

    }
}
```

[COPIAR CÓDIGO](#)

Executando a classe, temos o seguinte resultado no console:

```
2
11
```

[COPIAR CÓDIGO](#)

Há muitos outros métodos disponíveis, é possível, por exemplo, remover elementos do array. Para isso, utilizamos o método `remove()` .

Testaremos remover o elemento armazenado no índice `0`, e imprimiremos mais uma vez o tamanho da lista para nos certificarmos de que funcionou:

```
package br.com.bytebank.banco.test.util;

public class Teste {

    public static void main(String[] args) {

        ArrayList lista = new ArrayList();

        Conta cc = new ContaCorrente(22, 11);
        lista.add(cc);

        Conta cc2 = new ContaCorrente(22, 22);
        lista.add(cc2);

        System.out.println("Tamanho: " + lista.size());
        Conta ref = (Conta) lista.get(0);
        System.out.println(ref.getNumero());

        lista.remove(0);

        System.out.println("Tamanho: " + lista.size());

    }
}
```

[COPIAR CÓDIGO](#)

Executaremos e temos o seguinte resultado:

```
Tamanho: 2
11
Tamanho: 1
```

[COPIAR CÓDIGO](#)

O código funcionou. Adicionaremos mais contas:

```
package br.com.bytebank.banco.test.util;

public class Teste {

    public static void main(String[] args) {

        ArrayList lista = new ArrayList();

        Conta cc = new ContaCorrente(22, 11);
        lista.add(cc);

        Conta cc2 = new ContaCorrente(22, 22);
        lista.add(cc2);

        System.out.println(lista.size());
        Conta ref = (Conta) lista.get(0);
        System.out.println(ref.getNumero());

        lista.remove(0);

        System.out.println("Tamanho: " + lista.size());

        Conta cc3 = new ContaCorrente(33, 311);
        lista.add(cc3);

        Conta cc4 = new ContaCorrente(33, 322);
        lista.add(cc4);

    }
}
```

[COPIAR CÓDIGO](#)

Assim, em ordem, nós adicionamos dois elementos, removemos um, e adicionamos mais dois. Temos um total de três elementos remanescentes.

Queremos agora acessar cada um destes elementos, dentro de um laço, ou seja, fazer a **iteração** deles. Para isso, utilizamos o `for` :

```
package br.com.bytebank.banco.test.util;

public class Teste {

    public static void main(String[] args) {

        ArrayList lista = new ArrayList();

        Conta cc = new ContaCorrente(22, 11);
        lista.add(cc);

        Conta cc2 = new ContaCorrente(22, 22);
        lista.add(cc2);

        System.out.println(lista.size());
        Conta ref = (Conta) lista.get(0);
        System.out.println(ref.getNumero());

        lista.remove(0);

        System.out.println("Tamanho: " + lista.size());

        Conta cc3 = new ContaCorrente(33, 311);
        lista.add(cc3);

        Conta cc4 = new ContaCorrente(33, 322);
        lista.add(cc4);

        for(int i = 0; i < lista.size(); i++) {

    }

}
```

[COPIAR CÓDIGO](#)

Dentro do laço, ele nos retornará uma referência do tipo `Object`, e é ela que imprimiremos:

```
package br.com.bytebank.banco.test.util;

public class Teste {

    public static void main(String[] args) {

        ArrayList lista = new ArrayList();

        Conta cc = new ContaCorrente(22, 11);
        lista.add(cc);

        Conta cc2 = new ContaCorrente(22, 22);
        lista.add(cc2);

        System.out.println(lista.size());
        Conta ref = (Conta) lista.get(0);
        System.out.println(ref.getNumero());

        lista.remove(0);

        System.out.println("Tamanho: " + lista.size());

        Conta cc3 = new ContaCorrente(33, 311);
        lista.add(cc3);

        Conta cc4 = new ContaCorrente(33, 322);
        lista.add(cc4);

        for(int i = 0; i < lista.size(); i++) {
            Object oRef = lista.get(i);
```

```
        System.out.println(oRef);
    }

}
```

[COPIAR CÓDIGO](#)

Lembrando que, internamente, o nosso `System.out.println()` utiliza o `toString()` da classe `ContaCorrente`.

Executando a classe, temos o seguinte resultado:

11

Tamanho:**1**

`ContaCorrente, Numero: 22, Agencia: 22`

`ContaCorrente, Numero: 311, Agencia: 33`

`ContaCorrente, Numero: 322, Agencia: 33`

[COPIAR CÓDIGO](#)

Nosso código funcionou.

Entretanto, atualmente há uma forma ainda mais simples de fazermos este laço.

Temos também um `for`, que a cada iteração nos retornará um `Object`:

```
package br.com.bytebank.banco.test.util;

public class Teste {

    public static void main(String[] args) {

        ArrayList lista = new ArrayList();

        Conta cc = new ContaCorrente(22, 11);
        lista.add(cc);
```

```

    Conta cc2 = new ContaCorrente(22, 22);
    lista.add(cc2);

    System.out.println(lista.size());
    Conta ref = (Conta) lista.get(0);
    System.out.println(ref.getNumero());

    lista.remove(0);

    System.out.println("Tamanho: " + lista.size());

    Conta cc3 = new ContaCorrente(33, 311);
    lista.add(cc3);

    Conta cc4 = new ContaCorrente(33, 322);
    lista.add(cc4);

    for(int i = 0; i < lista.size(); i++) {
        Object oRef = lista.get(i);
        System.out.println(oRef);
    }

    for(Object o : lista)
}

}

```

[COPIAR CÓDIGO](#)

Para cada elemento do tipo `Object` desta lista, queremos que seja impresso o seu valor:

```

package br.com.bytebank.banco.test.util;

public class Teste {

```

```
public static void main(String[] args) {  
  
    ArrayList lista = new ArrayList();  
  
    Conta cc = new ContaCorrente(22, 11);  
    lista.add(cc);  
  
    Conta cc2 = new ContaCorrente(22, 22);  
    lista.add(cc2);  
  
    System.out.println(lista.size());  
    Conta ref = (Conta) lista.get(0);  
    System.out.println(ref.getNumero());  
  
    lista.remove(0);  
  
    System.out.println("Tamanho: " + lista.size());  
  
    Conta cc3 = new ContaCorrente(33, 311);  
    lista.add(cc3);  
  
    Conta cc4 = new ContaCorrente(33, 322);  
    lista.add(cc4);  
  
    for(int i = 0; i < lista.size(); i++) {  
        Object oRef = lista.get(i);  
        System.out.println(oRef);  
    }  
  
    for(Object o : lista) {  
        System.out.println(o);  
    }  
}  
}
```

COPIAR CÓDIGO

Deveríamos ter a mesma saída para os dois métodos `for`. Para diferenciá-los, criaremos uma divisão entre eles:

```
package br.com.bytebank.banco.test.util;

public class Teste {

    public static void main(String[] args) {

        ArrayList lista = new ArrayList();

        Conta cc = new ContaCorrente(22, 11);
        lista.add(cc);

        Conta cc2 = new ContaCorrente(22, 22);
        lista.add(cc2);

        System.out.println(lista.size());
        Conta ref = (Conta) lista.get(0);
        System.out.println(ref.getNumero());

        lista.remove(0);

        System.out.println("Tamanho: " + lista.size());

        Conta cc3 = new ContaCorrente(33, 311);
        lista.add(cc3);

        Conta cc4 = new ContaCorrente(33, 322);
        lista.add(cc4);

        for(int i = 0; i < lista.size(); i++) {
            Object oRef = lista.get(i);
            System.out.println(oRef);
        }

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

```
        for(Object o : lista) {  
            System.out.println(o);  
        }  
    }  
}
```

[COPIAR CÓDIGO](#)

Executaremos e temos o seguinte resultado:

```
11  
Tamanho:1  
ContaCorrente, Numero: 22, Agencia: 22  
ContaCorrente, Numero: 311, Agencia: 33  
ContaCorrente, Numero: 322, Agencia: 33  
-----  
ContaCorrente, Numero: 22, Agencia: 22  
ContaCorrente, Numero: 311, Agencia: 33  
ContaCorrente, Numero: 322, Agencia: 33
```

[COPIAR CÓDIGO](#)

Funcionou.

Atualmente, é comum utilizarmos a segunda modalidade do `for`.

Até a próxima!

Introdução ao Generics

Transcrição

Nesta aula, continuaremos a trabalhar com nosso `ArrayList`.

Primeiro, resolveremos a questão das observações apontadas pelo Eclipse, evidenciadas pelas palavras sublinhadas em amarelo.

O Eclipse faz isso para nos auxiliar com problemas de tipagem, ou seja, onde estamos acessando a primeira posição do array, fazemos também um cast:

```
//Código omitido

public class Teste {

    //Código omitido

        System.out.println("Tamanho: " + lista.size());
        Conta ref = (Conta) lista.get(0);
        System.out.println(ref.getNumero());

    //Código omitido

}
```

[COPIAR CÓDIGO](#)

Porque sabemos que o retorno é de um objeto do tipo `Conta`. Entretanto, isso não serve como uma garantia definitiva para o compilador, de que estamos corretos.

Por exemplo, comentaremos a linha de criação do primeiro cliente `cc`, e criaremos um novo, `cliente`, em seu lugar:

```
//Código omitido

public class Teste {

    public static void main(String[] args) {

        ArrayList lista = new ArrayList();

        //Conta cc = new ContaCorrente(22, 11);
        Cliente cliente = new Cliente();
        lista.add(cliente);

        Conta cc2 = new ContaCorrente(22, 22);
        lista.add(cc2);

//Código omitido

    }
}
```

[COPIAR CÓDIGO](#)

O código continua compilando, isso porque, como temos uma referência genérica do tipo `Object`, ela aceita também referências do tipo `Conta`, como é o caso do novo objeto que acabamos de criar.

Entretanto, ao executarmos este código, temos um erro de *class cast exception*. Ela acontece na seguinte linha em nosso código:

```
//Código omitido

Conta ref = (Conta) lista.get(0);

//Código omitido
```

[COPIAR CÓDIGO](#)

O erro acontece pois agora ela não devolve mais uma referência compatível com `Conta`, mas sim `Cliente`.

Queremos evitar este tipo de situação. O ideal é não misturarmos dados, queremos guardar em nosso array as contas, e não clientes e contas juntos.

A liberdade de um `ArrayList` genérico é importante, mas neste caso, especificamente, não queremos misturar dados, nosso objetivo é armazenar somente objetos do tipo `Conta`.

Para limitarmos o `ArrayList` podemos utilizar a sintaxe `<>`, os símbolos de menor e maior e, dentro, indicar o tipo de classes e objetos que aquela lista poderá armazenar, no nosso caso, será `Conta`:

```
//Código omitido

public class Teste {

    public static void main(String[] args) {

        ArrayList<Conta> lista = new ArrayList();

    }
}
```

[COPIAR CÓDIGO](#)

Normalmente, um desenvolvedor não se refere a isso como "uma lista de referências do tipo `Conta`", aqui estamos utilizando esta linguagem por motivos didáticos. É mais usual, na prática, isto ser referido como "uma lista de objetos", quando na verdade sabemos que não temos objetos de fato dentro da lista, somente as referências.

No lado direito, repetiremos a especificação, com a mesma sintaxe:

```
//Código omitido

public class Teste {

    public static void main(String[] args) {

        ArrayList<Conta> lista = new ArrayList<Conta>();
```

//Código omitido

COPIAR CÓDIGO



Assim, o compilador sabe que, no `ArrayList` que foi criado, só pode haver referências do tipo `Conta`.

Hipoteticamente, se tentarmos criar um `Cliente` e adicioná-lo ao array, o código nem compilará.

Dessa forma, ter um erro "*class cast exception*" é impossível, pois o compilador já proíbe isso de antemão.

Além disso, o compilador já sabe que o método `get()` retornará uma referência do tipo `Conta`, pois é impossível adicionar qualquer outra coisa diferente disso.

Por esse motivo, não é necessário fazermos o cast:

```
//Código omitido

public class Teste {

    //Código omitido

    System.out.println("Tamanho: " + lista.size());
```

```
        Conta ref = lista.get(0);
        System.out.println(ref.getNumero());

//Código omitido

}
```

[COPIAR CÓDIGO](#)

Temos mais uma facilidade, no método `for`, podemos utilizar diretamente o tipo `Conta`:

```
//Código omitido

for(Conta conta : lista) {
    System.out.println(conta);
}
```

//Código omitido

[COPIAR CÓDIGO](#)

Isso não funcionaria, se não tivéssemos feito a referência entre os símbolos de menor e maior (`<>`).

Ao executarmos o código, temos o seguinte resultado no console:

```
Tamanho: 2
11
Tamanho: 1
ContaCorrente, Numero: 22, Agencia: 22
ContaCorrente, Numero: 311, Agencia: 33
ContaCorrente, Numero: 322, Agencia: 33
-----
ContaCorrente, Numero: 22, Agencia: 22
ContaCorrente, Numero: 311, Agencia: 33
ContaCorrente, Numero: 322, Agencia: 33
```

[COPIAR CÓDIGO](#)

Tudo está funcionando normalmente.

Estes símbolos que utilizamos (< >) se chamam **generics**. Temos uma classe, ela é genérica, mas ao utilizá-la, podemos tipificar o que ela conterá, por exemplo,

String :

//Código omitido

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

//Código omitido

[COPIAR CÓDIGO](#)

O que fizemos foi especificar que temos apenas referências do tipo `Conta` , no primeiro `ArrayList` , e que o segundo aceita somente `String`s. O código fica assim mais seguro, mais tipificado.

Nos vemos na próxima!

Outras formas de inicialização

Lista com capacidade predefinida

Falamos que o `ArrayList` é um array dinâmico, ou seja, por baixo dos panos é usado um array, mas sem se preocupar com os detalhes e limitações.

Agora pense que você precisa criar uma lista representando todos os 26 estados do Brasil. Você gostaria de usar um `ArrayList` para "fugir" do array, mas sabe que o `ArrayList` cria um array automaticamente, do tamanho que a classe acha conveniente.

Será que não há uma forma de criar essa lista já definindo o tamanho do array? Claro que tem e é muito simples. O construtor da classe `ArrayList` é sobrecarregado e possui um parâmetro que recebe a *capacidade*:

```
ArrayList lista = new ArrayList(26); //capacidade inicial
```

[COPIAR CÓDIGO](#)

A lista continua dinâmica, mas o tamanho do array inicial é de 26!

Lista a partir de outra

Outra forma de inicializar uma lista é baseado na outra que é muito comum no dia a dia. Para tal a `ArrayList` possui mais um construtor que recebe a lista base:

```
ArrayList lista = new ArrayList(26); //capacidade inicial
lista.add("RJ");
```

```
lista.add("SP");
//outros estados
ArrayList nova = new ArrayList(lista); //criando baseado na pr:
```

[COPIAR CÓDIGO](#)



Quanto mais sabemos sobre as classes Java padrão mais fácil fica o nosso código.

O método equals

Transcrição

Você pode fazer o [DOWNLOAD \(<https://caelum-online-public.s3.amazonaws.com/850-java-util/04/java6-cap4.zip>\)](https://caelum-online-public.s3.amazonaws.com/850-java-util/04/java6-cap4.zip) do projeto da aula anterior.

Olá! Nesta aula, daremos continuidade à nossa viagem pelo pacote `java.util` , neste momento, passando pela classe `ArrayList` , integrando com a classe `Object` , e, nesta aula, com o método `equals()` .

Primeiro, renomearemos a classe `Teste` , ela passará a se chamar `TesteArrayList` . Em seguida, criaremos uma cópia sua, que se chamará `TesteArrayListEquals` .

A classe `TesteArrayListEquals` terá o seguinte conteúdo:

```
package br.com.bytebank.banco.test.util;

import java.util.ArrayList;

public class TesteArrayListEquals {

    public static void main(String[] args) {

        //Generics
        ArrayList<Conta> lista = new ArrayList<Conta>();
```

```
Conta cc = new ContaCorrente(22, 11);
lista.add(cc);

Conta cc2 = new ContaCorrente(22, 22);
lista.add(cc2);

for(Conta conta : lista) {
    System.out.println(conta);
}

}
```

[COPIAR CÓDIGO](#)

Ou seja, temos nela duas contas adicionadas, e um laço. Ao executarmos, temos o seguinte resultado no console:

```
ContaCorrente, Numero: 11, Agencia: 22
ContaCorrente, Numero: 22, Agencia: 22
```

[COPIAR CÓDIGO](#)

A seguir, começaremos a trabalhar com o método `contains()`. Podemos verificar se a lista possui determinado elemento, por exemplo, `cc2`. O `contains()` nos retorna um `boolean`, `true` ou `false`, dependendo da existência deste elemento, ou não.

Para visualizarmos melhor, criaremos uma impressão desta resposta:

```
package br.com.bytebank.banco.test.util;

import java.util.ArrayList;

public class TesteArrayListEquals {

    public static void main(String[] args) {
```

```
//Generics  
ArrayList<Conta> lista = new ArrayList<Conta>();  
  
Conta cc = new ContaCorrente(22, 11);  
lista.add(cc);  
  
Conta cc2 = new ContaCorrente(22, 22);  
lista.add(cc2);  
  
boolean existe = lista.contains(cc2);  
  
System.out.println("Já existe? " + existe);  
  
for(Conta conta : lista) {  
    System.out.println(conta);  
}  
}  
}
```

[COPIAR CÓDIGO](#)

Teoricamente, teríamos que ter um resultado `true`, uma vez que sabemos que a conta `cc2` já foi adicionada. Executando a classe, temos o seguinte resultado:

```
Já existe? true  
ContaCorrente, Numero: 11, Agencia: 22  
ContaCorrente, Numero: 22, Agencia: 22
```

[COPIAR CÓDIGO](#)

Funcionou!

A seguir, criaremos mais uma conta, antes da linha que contém o método `contains()`. Esta conta será `cc3`:

```
package br.com.bytebank.banco.test.util;
```

```
import java.util.ArrayList;

public class TesteArrayListEquals {

    public static void main(String[] args) {

        //Generics
        ArrayList<Conta> lista = new ArrayList<Conta>();

        Conta cc = new ContaCorrente(22, 11);
        lista.add(cc);

        Conta cc2 = new ContaCorrente(22, 22);
        lista.add(cc2);

        Conta cc3 = new ContaCorrente(22, 22);
        boolean existe = lista.contains(cc2);

        System.out.println("Já existe? " + existe);

        for(Conta conta : lista) {
            System.out.println(conta);
        }
    }
}
```

[COPIAR CÓDIGO](#)

Entretanto, apesar de as contas `cc2` e `cc3` possuírem os mesmos números de agência e conta, não adicionamos `cc3` à lista. Alteraremos a referência do método `contains()` para que ele busque na lista a existência da conta `cc3`:

```
package br.com.bytebank.banco.test.util;

import java.util.ArrayList;

public class TesteArrayListEquals {
```

```
public static void main(String[] args) {  
  
    //Generics  
    ArrayList<Conta> lista = new ArrayList<Conta>();  
  
    Conta cc = new ContaCorrente(22, 11);  
    lista.add(cc);  
  
    Conta cc2 = new ContaCorrente(22, 22);  
    lista.add(cc2);  
  
    Conta cc3 = new ContaCorrente(22, 22);  
    boolean existe = lista.contains(cc3);  
  
    System.out.println("Já existe? " + existe);  
  
    for(Conta conta : lista) {  
        System.out.println(conta);  
    }  
}  
}
```

[COPIAR CÓDIGO](#)

Como sabemos, a referência `cc3` não existe na lista, contudo, os dados da conta são exatamente os mesmos da `cc2`. Na prática, as duas referências representam uma mesma conta na vida real.

Executaremos a classe. Temos o seguinte resultado no console:

```
Já existe? false  
ContaCorrente, Numero: 11, Agencia: 22  
ContaCorrente, Numero: 22, Agencia: 22
```

[COPIAR CÓDIGO](#)

Ele nos retornou `false`. Isso porque, na verdade, o `cc3` é uma outra referência, e o `contains()` faz um laço em cima de cada elemento do array, internamente, e verifica se a referência que está sendo guardada é igual ao que foi passado, que no caso é `cc3`. Em caso positivo, ele imprime `true`. Pormenorizado, este `for` seria escrito da seguinte forma:

//Código omitido

```
for(Conta conta : lista) {  
    if(conta == cc3) {  
        System.out.println("Já tenho essa conta!");  
    }  
}
```

//Código omitido

[COPIAR CÓDIGO](#)

Se verificarmos novamente com o `cc2`, veremos que o retorno será `true`, ou seja, verdadeiro, uma vez que esta referência está inserida na lista.

Nosso objetivo será indicar ao programa que, na verdade, apesar de se tratarem de referências diferentes, `cc2` e `cc3` apontam para um mesmo objeto.

Primeiro, faremos isso com o laço que criamos acima. Ao utilizarmos os dois símbolos de igualdade (`==`), internamente, o Java sempre compara as referências, por isso, quando propomos que `conta == cc3` esta afirmação nunca retornará `true`. Sendo assim, não podemos utilizar o (`==`).

Precisamos implementar algo que, de alguma forma, indique que uma conta é exatamente igual a outra. A condição para que isso seja verdade é que o número de agência e conta sejam idêntico.

Isso faz parte da regra de negócio, que deve estar inserida no pacote do modelo. Ela está relacionada com a classe `Conta`, logo, é nela que a adicionaremos.

Como a regra de igualdade é a mesma tanto para a `ContaCorrente` quanto para a `ContaPoupanca`, podemos inseri-la na classe mãe, `Conta`, dessa forma ela é válida para todas as filhas.

Abriremos a classe `TesteArrayListEquals`, para adicionarmos esta funcionalidade, que chamaremos de `ehIgual`:

```
//Código omitido

for(Conta conta : lista) {
    if(conta.ehIgual(cc3)) {
        System.out.println("Já tenho essa conta!");
    }
}
```

//Código omitido

[COPIAR CÓDIGO](#)

Falta implementarmos a regra de `ehIgual`, na classe `Conta`. Será um método público, pois queremos acessá-lo fora do pacote. Um `if` necessita de uma expressão que resulte em um retorno do tipo `boolean`, assim sendo, ele será booleano. Este método receberá como parâmetro sempre uma variável do tipo `conta`:

```
//Código omitido

public abstract class Conta extends Object {

    //

    public static int getTotal() {
        return Conta.total;
    }
}
```

```
public boolean ehIgual(Conta outra) {  
  
}  
  
//Código omitido, método `toString()`  
}  
}
```

[COPIAR CÓDIGO](#)

Em seguida, implementaremos a regra de igualdade. Criaremos um `if` que verificará se os números de agência e conta são idênticos. Esta comparação se dará na negativa, ou seja, se forem diferentes, o programa imediatamente retornará uma negativa `false`:

```
//Código omitido  
  
public abstract class Conta extends Object {  
  
//  
  
    public static int getTotal() {  
        return Conta.total;  
    }  
  
    public boolean ehIgual(Conta outra) {  
  
        if(this.agencia != outra.agencia) {  
            return false;  
        }  
  
        //Código omitido, método `toString()`  
    }  
}
```

[COPIAR CÓDIGO](#)

Assim, se a execução passa para a próxima fase, já temos certeza de que o número da agência é o mesmo. O segundo `if` servirá para a verificação do número da conta:

```
//Código omitido

public abstract class Conta extends Object {

    //

    public static int getTotal() {
        return Conta.total;
    }

    public boolean ehIgual(Conta outra) {

        if(this.agencia != outra.agencia) {
            return false;
        }

        if(this.numero != outra.numero) {
            return false;
        }

    }

    //Código omitido, método `toString()`
}

}
```

[COPIAR CÓDIGO](#)

Da mesma forma, se os números de conta forem diferentes, teremos um retorno `false`. Portanto, tendo passado por estes dois estágios da execução e os números sendo compatíveis, chegamos à conclusão de que as contas são idênticas e, finalmente, teremos um retorno `true`:

```
//Código omitido

public abstract class Conta extends Object {

    //

    public static int getTotal() {
        return Conta.total;
    }

    public boolean ehIgual(Conta outra) {

        if(this.agencia != outra.agencia) {
            return false;
        }

        if(this.numero != outra.numero) {
            return false;
        }

        return true;
    }

    //Código omitido, método `toString()`
}

}
```

[COPIAR CÓDIGO](#)

Retornaremos à classe `TesteArrayListEquals` e vemos que ela já está compilando, uma vez que o método `ehIgual` já foi implementado.

Para facilitar o entendimento, na classe `TesteArrayListEquals` comentaremos as seguintes linhas de código:

```
//Código omitido

    public static void main(String[] args) {

        //Generics
        //
        ArrayList<Conta> lista = new ArrayList<Conta>()

        //
        Conta cc = new ContaCorrente(22, 11);
        lista.add(cc);

        //
        Conta cc2 = new ContaCorrente(22, 22);
        lista.add(cc2);

        //
        Conta cc3 = new ContaCorrente(22, 22);
        boolean existe = lista.contains(cc3);

        //
        System.out.println("Já existe? " + existe);

        //
        for(Conta conta : lista) {
            if(conta.ehIgual(cc3)) {
                System.out.println("Já tenho essa conta
            }
        }

        //
        for(Conta conta : lista) {
            System.out.println(conta);
        }
    }

//Código omitido
```

[COPIAR CÓDIGO](#)

O próximo passo será criarmos duas contas, diferentes:

```
//Código omitido
```

```
public static void main(String[] args) {  
  
    Conta cc1 = new ContaCorrente(22, 11);  
    Conta cc2 = new ContaCorrente(22, 22);  
  
    //Generics  
    //  
    //  
    Conta cc = new ContaCorrente(22, 11);  
    lista.add(cc);  
    //  
    //  
    Conta cc2 = new ContaCorrente(22, 22);  
    lista.add(cc2);  
    //  
    //  
    Conta cc3 = new ContaCorrente(22, 22);  
    boolean existe = lista.contains(cc3);  
    //  
    //  
    System.out.println("Já existe? " + existe);  
    //  
    //  
    for(Conta conta : lista) {  
        if(conta.ehIgual(cc3)) {  
            System.out.println("Já tenho essa conta");  
        }  
    }  
    //  
    //  
    for(Conta conta : lista) {  
        System.out.println(conta);  
    }  
}  
//Código omitido
```

[COPIAR CÓDIGO](#)

Testaremos nosso método `ehIgual`, sabendo que as contas são diferentes, ou seja, esperando receber um resultado `false`:

```
//Código omitido

    public static void main(String[] args) {

        Conta cc1 = new ContaCorrente(22, 11);
        Conta cc2 = new ContaCorrente(22, 22);

        boolean igual = cc1.ehIgual(cc2);
        System.out.println(igual);

        //Generics
        //
        ArrayList<Conta> lista = new ArrayList<Conta>()
        //
        //
        Conta cc = new ContaCorrente(22, 11);
        lista.add(cc);
        //
        //
        Conta cc2 = new ContaCorrente(22, 22);
        lista.add(cc2);
        //
        //
        Conta cc3 = new ContaCorrente(22, 22);
        boolean existe = lista.contains(cc3);
        //
        //
        System.out.println("Já existe? " + existe);
        //
        //
        for(Conta conta : lista) {
            if(conta.ehIgual(cc3)) {
                System.out.println("Já tenho essa conta
            }
        }
        //
        //
        for(Conta conta : lista) {
            System.out.println(conta);
        }
    }

//Código omitido
```

[COPIAR CÓDIGO](#)

Executaremos a classe, e temos o seguinte resultado no console:

false

[COPIAR CÓDIGO](#)

Conforme esperado, o resultado foi `false`, indicando que nosso código funcionou.

Entretanto, criaremos a seguir dois objetos que representam na prática a mesma conta:

```
//Código omitido

public static void main(String[] args) {

    Conta cc1 = new ContaCorrente(22, 22);
    Conta cc2 = new ContaCorrente(22, 22);

    boolean igual = cc1.ehIgual(cc2);
    System.out.println(igual);

    //Generics
    //
    ArrayList<Conta> lista = new ArrayList<Conta>()
    //
    //        Conta cc = new ContaCorrente(22, 11);
    //        lista.add(cc);
    //
    //        Conta cc2 = new ContaCorrente(22, 22);
    //        lista.add(cc2);
    //
    //        Conta cc3 = new ContaCorrente(22, 22);
    //        boolean existe = lista.contains(cc3);
    //
    //        System.out.println("Já existe? " + existe);
    //
```

```
//             for(Conta conta : lista) {
//                 if(conta.ehIgual(cc3)) {
//                     System.out.println("Já tenho essa conta
//                 }
//             }
//
//             for(Conta conta : lista) {
//                 System.out.println(conta);
//             }
}
//Código omitido
```

[COPIAR CÓDIGO](#)



Executando a classe, temos o seguinte resultado:

true

[COPIAR CÓDIGO](#)

Nosso método está funcionando!

Em seguida, comentaremos estas linhas que acabamos de criar, e removeremos os comentários das que havíamos marcado anteriormente:

```
//Código omitido

public static void main(String[] args) {

    //          Conta cc1 = new ContaCorrente(22, 22);
    //          Conta cc2 = new ContaCorrente(22, 22);
    //
    //          boolean igual = cc1.ehIgual(cc2);
    //          System.out.println(igual);

    //Generics
```

```
ArrayList<Conta> lista = new ArrayList<Conta>();

Conta cc = new ContaCorrente(22, 11);
lista.add(cc);

Conta cc2 = new ContaCorrente(22, 22);
lista.add(cc2);

Conta cc3 = new ContaCorrente(22, 22);
boolean existe = lista.contains(cc3);

System.out.println("Já existe? " + existe);

for(Conta conta : lista) {
    if(conta.ehIgual(cc3)) {
        System.out.println("Já tenho essa conta!");
    }
}

for(Conta conta : lista) {
    System.out.println(conta);
}
}
```

[COPIAR CÓDIGO](#)

Editaremos o código para liberarmos mais espaço, eliminaremos o primeiro laço
for :

```
//Código omitido

public static void main(String[] args) {

//          Conta cc1 = new ContaCorrente(22, 22);
//          Conta cc2 = new ContaCorrente(22, 22);
//
```

```
//                     boolean igual = cc1.ehIgual(cc2);
//                     System.out.println(igual);

//Generics
ArrayList<Conta> lista = new ArrayList<Conta>();

Conta cc = new ContaCorrente(22, 11);
lista.add(cc);

Conta cc2 = new ContaCorrente(22, 22);
lista.add(cc2);

Conta cc3 = new ContaCorrente(22, 22);
boolean existe = lista.contains(cc3);

System.out.println("Já existe? " + existe);

for(Conta conta : lista) {
    System.out.println(conta);
}

}
```

[COPIAR CÓDIGO](#)

Nosso objetivo é que o `contains()` utilize, internamente, nosso método `ehIgual()`. Apesar de já fazer uso de um método, não é aquele que desejamos. O que o `contains()` utiliza por padrão chama-se `equals()`.

Assim, para atingirmos nosso objetivo, de que ele possa identificar quando há objetos iguais ainda que tenham referências distintas, teremos que alterar o método para que siga o padrão definido pela classe `Object`.

Para lembrarmos, retornaremos à classe `Object`. Nela, vemos que já existe o método que se chama `equals()`:

```
//Código omitido

public class Object {

    //Código omitido

    public boolean equals(Object obj) {
        return (this == obj);
    }

    //Código omitido

}
```

[COPIAR CÓDIGO](#)

Assim sendo, devemos sobrescrever este método. Isso nos permitirá definir nosso próprio critério de igualdade, que neste caso, trata-se de uma regra de negócio.

Por padrão, o método `equals()` faz algo que não queremos, que é comparar as referências `this` e `obj`. Por isso, retornaremos à classe `Conta`, onde chamaremos este método:

```
//Código omitido

public abstract class Conta extends Object {

    //Código omitido

    public boolean equals(Conta outra) {

        if(this.agencia != outra.agencia) {
            return false;
        }

        if(this.numero != outra.numero) {
```

```
        return false;  
    }  
  
    return true;  
}  
  
//Código omitido  
  
}
```

[COPIAR CÓDIGO](#)

Além disso, temos que garantir, por meio da anotação `@Override`. Ela indica ao compilador que determinado método deve, justamente, sobrescrever um outro definido na classe mãe:

```
//Código omitido  
  
public abstract class Conta extends Object {  
  
    //Código omitido  
  
    @Override  
    public boolean equals(Conta outra) {  
  
        if(this.agencia != outra.agencia) {  
            return false;  
        }  
  
        if(this.numero != outra.numero) {  
            return false;  
        }  
  
        return true;  
    }  
  
    //Código omitido
```

```
}
```

[COPIAR CÓDIGO](#)

Feito isso, percebemos que imediatamente o código passa a não compilar.

Para solucionarmos isso, retornaremos à classe `Object` :

```
//Código omitido
```

```
public class Object {
```

```
//Código omitido
```

```
    public boolean equals(Object obj) {
        return (this == obj);
    }
```

```
//Código omitido
```

```
}
```

[COPIAR CÓDIGO](#)

Notamos que o método recebe um `Object`, ou seja, um tipo genérico. Portanto, retornando à classe `Conta`, vemos que o método também deveria receber algo da mesma natureza, no caso, o próprio `Object` e uma referência `ref`:

```
//Código omitido
```

```
public abstract class Conta extends Object {
```

```
//Código omitido
```

```
@Override
```

```
    public boolean equals(Object ref) {
```

```
    if(this.agencia != outra.agencia) {
        return false;
    }

    if(this.numero != outra.numero) {
        return false;
    }

    return true;
}

//Código omitido
```

[COPIAR CÓDIGO](#)

Com isso, não temos mais um erro de compilação nesta linha, mas temos nas demais. Isso porque este tipo `ref` não possui atributos que se chamam `agencia` ou `numero`. Assim, transformaremos a referência genérica em uma específica, como sabemos, utilizamos o **cast**:

```
//Código omitido

public abstract class Conta extends Object {

//Código omitido

@Override
public boolean equals(Object ref) {

    Conta outra = (Conta) ref;

    if(this.agencia != outra.agencia) {
        return false;
    }
```

```
        if(this.numero != outra.numero) {  
            return false;  
        }  
  
        return true;  
    }  
  
//Código omitido  
  
}
```

[COPIAR CÓDIGO](#)

Pronto, o código voltou a compilar, e estamos sobrescrevendo o método `equals()` , da classe `Object` .

Retornaremos à classe `TesteArrayListEquals` . Agora sabemos que o método `contains()` faz um laço internamente, e chama o método `equals()` . Entretanto ele, por padrão da classe `Object` , compara as referências. Como agora o sobrescrevemos, ele passará a funcionar conforme nossa regra de igualdade. Executando novamente a classe, temos o seguinte resultado no console:

```
Já existe? true  
ContaCorrente, Numero: 11, Agencia: 22  
ContaCorrente, Numero: 22, Agencia: 22
```

[COPIAR CÓDIGO](#)

Onde antes o resultado era `false` , agora o retorno é `true` . O `contains()` agora é capaz de detectar que adicionamos um objeto com os mesmos valores, que indicam a igualdade entre eles.

Assim, observamos que a classe `ArrayList` , internamente, se baseia na implementação da nossa classe `Conta` . Ela faz isso graças à sobrescrita da classe `Object` .

Vamos para os exercícios, e até a próxima!

List e LinkedList

Transcrição

Olá! Bem-vindo novamente. Nesta aula, continuaremos falando sobre o pacote `java.util`.

Vimos como o `ArrayList` aproveita o método `equals()` internamente, e desde que o usuário sobrescreva com seu próprio método de igualdade, ele pode fazer uso também do método `contains()`.

Além disso, vimos anteriormente que o método `toString()` foi sobreescrito, que as listas utilizam internamente o método `equals()`, e, existe ainda um outro método chamado `hashCode()`, que é utilizado em outras classes dentro do `java.util`, e não na lista. Ele não será abordado neste curso, mas há um outro que é focado especificamente nisso. Existe uma estrutura de dados que se chama conjunto, ou `set`, que utilizam este método e o `equals()` simultaneamente.

O `ArrayList` é uma lista, mas não é a única. Por que então utilizar outra?

Como qualquer outra, esta modalidade possui vantagens e desvantagens. Como lado positivo, temos as características de um array, ou seja, o acesso fácil a qualquer elemento aleatório. Se adicionamos elementos em um array, e queremos acessar qualquer posição, não há nenhum trabalho a mais para que façamos isso, esse acesso pode ser feito diretamente pelo índice de forma bastante facilitada.

Além disso, adicionar novos elementos em um array é um processo simples, desde que não seja ultrapassada sua capacidade de armazenamento. Se desejarmos adicionar um novo elemento, o `ArrayList` sabe automaticamente

qual a próxima posição livre, e ele é inserido - simples. Iteração também é algo tranquilo de se fazer em um array.

Do lado negativo, temos que a sua capacidade é limitada ao seu tamanho no momento da criação, ou seja, uma vez que sua capacidade acaba, é necessária a criação de um novo array, com capacidade maior, e copiar os elementos daquele para este armazenamento. Ainda, se quisermos remover um elemento, o array não permitirá que existam "buracos" em sua lista e, sendo assim, moverá todas as referências de modo que todos os índices serão modificados. Para esse tipo de operação, o `ArrayList` não é tão eficiente.

Se o objetivo for simplesmente adicionar elementos, para depois fazer a iteração, o `ArrayList` é sem dúvida o mais indicado. É esse o tipo mais comum de situação e, normalmente, o uso mais comum. Contudo, é necessário sabermos destas ressalvas.

Qual seria, então, a alternativa ao `ArrayList`? Para isso, temos o `LinkedList`. Ele **não utiliza um array internamente**.

Isso significa que, quando falamos de `List` nem sempre estamos lidando com arrays. Uma lista significa, simplesmente, que estamos armazenando elementos em sequência, ou seja, o primeiro elemento adicionado também é o primeiro que será retornado. Além disso, temos um índice, e métodos que trabalham com ele. Por exemplo, o método `get()`, que podemos utilizar para obter determinado elemento de uma posição em particular.

No `LinkedList` temos estas mesmas características, sequência, ordem de inserção e índice. Entretanto, ela não funciona com um array internamente.

Seu funcionamento ocorre da seguinte forma: ao adicionarmos, por exemplo, `cc1` e, em seguida, `cc2`, ela se lembrará do elemento que foi adicionado anteriormente, ou seja, `cc2` se lembra de `cc1`, `cc3` de `cc2`, e assim por diante. Da mesma forma, o primeiro elemento se lembra daquele que o segue, ou seja

`cc1` lembra de `cc2` , `cc2` de `cc3` , e assim sucessivamente. A isso, damos o nome de **lista duplamente encadeada**.

Sabemos que estamos no final da lista quando atingimos um elemento que não possui um próximo.

Neste tipo de lista, apagar um elemento não causa grande impacto à ela como um todo, ela o descarta e substitui com o seguinte e o anterior. Supondo que temos `cc1` , `cc2` e `cc3` , ao apagarmos `cc2` , simplesmente `cc1` e `cc3` passarão a ser diretamente conectados.

Mas e se quisermos acessar o último elemento da lista, como podemos acessá-lo? Temos que iniciar no primeiro elemento, e verifica-los sucessivamente, até atingirmos o ponto onde não há mais continuidade. Diferentemente do array, não temos como acessar uma determinada posição diretamente. Se quisermos, por exemplo, acessar a posição 3, temos que iniciar na primeira e seguir, até atingirmos a desejada. Isso faz com que a iteração seja algo negativo na `LinkedList` .

Para adicionar elementos, a lista sabe quais são o primeiro e último elementos, e faz a inserção após o final.

Colocaremos em prática, utilizando a `LinkedList` em nossa classe `TesteArrayListEquals` . Isso será complicado, uma vez que há duplicidade de diversos métodos, como `add()` , `contains()` , `size()` , `remove()` e `equals()` .

A primeira coisa a fazermos, na classe `TesteArrayList` , será alterarmos para `LinkedList` :

```
//Código omitido
```

```
public class TesteArrayList {
```

```
public static void main(String[] args) {  
  
    //Generics  
    LinkedList<Conta> lista = new LinkedList<Conta>();  
  
    ArrayList<String> nomes = new ArrayList<String>();  
  
    //Código omitido
```

[COPIAR CÓDIGO](#)

Precisamos importar esta classe. Ao deixarmos o cursor sobre o seu nome, vemos a opção "*Import LinkedList (java.util)*", clicaremos nela. Podemos apagar a linha do `ArrayList`:

```
//Código omitido  
  
import br.com.bytebank.banco.modelo.Conta;  
import br.com.bytebank.banco.modelo.ContaCorrente;  
  
public class TesteArrayList {  
  
    public static void main(String[] args) {  
  
        //Generics  
        LinkedList<Conta> lista = new LinkedList<Conta>();  
  
        ArrayList<String> nomes = new ArrayList<String>();  
  
        //Código omitido
```

[COPIAR CÓDIGO](#)

O código continua funcionando, apenas substituímos o `ArrayList` pelo `LinkedList`. Executaremos a classe, para testarmos. Temos o seguinte resultado no console:

```
Tamanho: 2
11
Tamanho: 1
ContaCorrente, Numero: 22, Agencia: 22
ContaCorrente, Numero: 311, Agencia: 33
ContaCorrente, Numero: 322, Agencia: 33
-----
ContaCorrente, Numero: 22, Agencia: 22
ContaCorrente, Numero: 311, Agencia: 33
ContaCorrente, Numero: 322, Agencia: 33
```

[COPIAR CÓDIGO](#)

Funcionou!

Apesar de os funcionamentos internos terem sido diferentes, obtivemos os mesmos resultados finais no console.

Temos então duas classes, que utilizam métodos com os mesmos nomes, mas cujas implementações diferem. Contudo, percebemos que há um ponto em comum, ou seja, todos aqueles que pretendem ser uma `lista`, devem contar com um método `add()` de alguma forma, o mesmo para o `remove()`, e assim sucessivamente, conforme citado acima.

Isso nos leva à conclusão de que existe uma interface para isso, ela se chama `List`. Ao implementarmos na classe `TesteArrayList`:

```
//Código omitido

import br.com.bytebank.banco.modelo.Conta;
import br.com.bytebank.banco.modelo.ContaCorrente;

public class TesteArrayList {

    public static void main(String[] args) {
```

```
//Generics  
List<Conta> lista = new LinkedList<Conta>();  
  
ArrayList<String> nomes = new ArrayList<String>();  
  
//Código omitido
```

[COPIAR CÓDIGO](#)

Assim como anteriormente, teremos que importar esta interface. Há duas importações possíveis, uma do pacote `java.awt`, e outra do `java.util`, neste caso, escolheremos a segunda opção:

```
package br.com.bytebank.teste.util;  
  
import java.util.ArrayList;  
import java.util.LinkedList;  
import java.util.List;  
  
import br.com.bytebank.banco.modelo.Conta;  
import br.com.bytebank.banco.modelo.ContaCorrente;  
  
public class TesteArrayList {  
  
    public static void main(String[] args) {  
  
        //Generics  
        List<Conta> lista = new LinkedList<Conta>();  
  
        ArrayList<String> nomes = new ArrayList<String>();  
  
        //Código omitido
```

[COPIAR CÓDIGO](#)

Ou seja, estamos utilizando a interface para definirmos o tipo da referência. Dentro da interface, podemos observar a existência de diversos métodos, uns que

já conhecemos mas muitos outros que ainda nem exploramos.

Por fim, alteraremos o `LinkedList` de volta para `ArrayList`:

```
package br.com.bytebank.teste.util;

import java.util.ArrayList;
import java.util.LinkedList;
import java.util.List;

import br.com.bytebank.banco.modelo.Conta;
import br.com.bytebank.banco.modelo.ContaCorrente;

public class TesteArrayList {

    public static void main(String[] args) {

        //Generics
        List<Conta> lista = new ArrayList<Conta>();

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

        //Código omitido
    }
}
```

[COPIAR CÓDIGO](#)

Existe ainda uma terceira implementação desta interface `List`, que se chama `java.util.Vector`, e que veremos adiante. Até a próxima!

De Array para List

A partir de agora vamos usar as listas para fugir das desvantagens do array. No entanto, se lembra do nosso array `String[]` do método `main`? Com certeza, e não podemos mudar a assinatura do método `main` pois a JVM não aceita isso. Bom, já que não podemos alterar a assinatura será que não tem uma forma de transformar uma array em uma lista? Claro que existe, e para tal, existe já uma classe que ajuda nessa tarefa: `java.util.Arrays`

A classe `java.util.Arrays` possui vários métodos estáticos auxiliares para trabalhar com arrays. Veja como fica simples de transformar um array para uma lista:

```
public class Teste {  
  
    public static void main(String[] args) {  
        List<String> argumentos = Arrays.asList(args);  
    }  
}
```

[COPIAR CÓDIGO](#)

Vamos ver ainda outras funcionalidades da classe `java.util.Arrays`

A alternativa threadsafe

Transcrição

Você pode fazer o [DOWNLOAD \(<https://caelum-online-public.s3.amazonaws.com/850-java-util/05/java6-cap5.zip>\)](https://caelum-online-public.s3.amazonaws.com/850-java-util/05/java6-cap5.zip) do projeto da aula anterior.

Olá!

Anteriormente, falamos sobre a `java.util.LinkedList` e faltou abordarmos o `java.util.Vector`, que juntamente com o `java.util.ArrayList` são as principais implementações da interface `java.util.List`.

No que o `Vector` difere das demais? Ele, na verdade, é igual a um `ArrayList`. Internamente, ele também utiliza um array. O Java nasceu com um `Vector` e, posteriormente, as demais implementações foram adicionadas.

O `Vector` tem uma diferença importante em relação ao `ArrayList`, ele é o que chamamos de ***thread safe***.

Como vimos, qualquer programa em Java inicia com um método `main`, que forma uma "pilha" e, a partir dele, podemos ter uma nova "pilha". Dessa forma, elas podem ser executadas em paralelo. O Java permite a criação de inúmeros métodos `main`.

Quando temos esse tipo de situação, e desejamos que as execuções sejam feitas em paralelo, em cima de uma mesma lista, utilizamos o `java.util.Vector`. Es

tipo de operação só funciona dessa forma, o `ArrayList` e o `LinkedList` não servem.

Apesar disso, o `Vector` é utilizado como exceção. As ocasiões em que ele é necessário são raras, ou seja, é difícil termos situações como a citada acima, onde são compartilhadas entre duas ou mais "pilhas" uma mesma lista.

Mas e se usássemos o `Vector` como uma medida preventiva? Caso ele fosse ser necessário futuramente? Isso também não funcionaria, já que a utilização dele, em si, tem um custo em desempenho. Assim, se não for estritamente necessário, é melhor utilizar outros tipos de lista, como o `ArrayList`, que é mais eficiente.

Para saber mais sobre isso, há dois cursos dedicados especificamente a este tópico, o primeiro, em que é falado sobre as *threads* e as implementações, e o segundo, onde é criado um servidor que utiliza estes recursos.

Os cursos citados acima são citados na [atividade 3](#) (<https://cursos.alura.com.br/course/java-util-lambdas/task/38648>), da aula 05.

Retornaremos ao código da classe `TesteArrayList`, onde temos implementado o `Vector`:

```
//Código omitido

public class TesteArrayList {

    public static void main(String[] args) {

        //Generics
        List<Conta> lista = new Vector<Conta>(); //thread

        Conta cc = new ContaCorrente(22, 11);
        lista.add(cc);
```

```
Conta cc2 = new ContaCorrente(22, 22);
lista.add(cc2);

//Código omitido
```

[COPIAR CÓDIGO](#)

A interface `List` permanece inalterada, mas estamos utilizando a nova classe `Vector`, que é aquilo que chamamos de ***thread safe***. Neste curso, a *thread safety* não é algo estritamente necessário, isso porque não estamos trabalhando com mais de um método `main`, ou "pilhas", como chamamos.

Ao executarmos a classe, temos o seguinte resultado no console:

```
Tamanho: 2
11
Tamanho: 1
ContaCorrente, Numero: 22, Agencia: 22
ContaCorrente, Numero: 311, Agencia: 33
ContaCorrente, Numero: 322, Agencia: 33
-----
ContaCorrente, Numero: 22, Agencia: 22
ContaCorrente, Numero: 311, Agencia: 33
ContaCorrente, Numero: 322, Agencia: 33
```

[COPIAR CÓDIGO](#)

Tudo continua funcionando normalmente.

Com isso, temos três opções de listas disponíveis para uso: `LinkedList`, `ArrayList`, ou `Vector`. Por mais que troquemos a implementação, o corpo do código permanece o mesmo, e funciona com qualquer um dos três tipos, graças ao polimorfismo. Por enquanto, manteremos em `ArrayList`:

```
//Código omitido

public class TesteArrayList {

    public static void main(String[] args) {

        //Generics
        List<Conta> lista = new ArrayList<Conta>();//thre

        Conta cc = new ContaCorrente(22, 11);
        lista.add(cc);

        Conta cc2 = new ContaCorrente(22, 22);
        lista.add(cc2);

//Código omitido
```

COPIAR CÓDIGO



Adiante, veremos mais sobre o pacote `java.util`. Vale lembrar sobre os cursos específicos para as *threads*, mencionados acima. Até a próxima!

A interface Collection

Transcrição

Olá! Nesta aula, daremos continuidade à explicação sobre o pacote `java.util` mas, antes, faremos mais uma revisão.

Até o momento, focamos em listas e nas implementações principais, dos principais métodos das listas. Como sabemos, as listas são sequências, por isso, ela sabe em qual ordem os seus elementos foram inseridos, com isso, ao iterar, os recebemos na mesma ordem de inserção.

Vimos tipos de implementação, como a `ArrayList`, o `Vector` - que possui o *thread safe*, e o `LinkedList`, que é totalmente encadeada. O que todos eles têm em comum é que são listas, e que aceitam duplicados. Isso significa que, se uma referência já foi adicionada, ela é aceita novamente pela lista. É possível adicionarmos a mesma referência diversas vezes, em uma mesma lista.

Neste ponto, temos um aspecto que nem sempre é desejado, nem sempre queremos ter elementos duplicados em nossas listas. Por exemplo, em uma lista de contas bancárias, não tem muito sentido haver o registro duplicado de determinado elemento. Neste contexto de negócio, não queremos que isto aconteça.

Entretanto, por padrão, a lista não nos fornece nenhum mecanismo que nos ajude em relação a este aspecto.

Precisamos, primeiro, verificar se determinado elemento já é presente para, somente então, termos segurança para adicioná-lo. Por este motivo, existe um

outro mundo de conjuntos, que inclui o `java.util.Set` e `java.util.HashSet`. Para entendê-los melhor, é necessário conhecer o método `hashCode()`.

Não se trata de sequências, os *sets* espalham seus elementos com base no `hashCode()`. Isto é melhor explorado em um [curso específico](https://cursos.alura.com.br/course/java-util-lambdas/task/38649) (<https://cursos.alura.com.br/course/java-util-lambdas/task/38649>), cujo foco é esse, na implementação do `HashSet` e em outras implementações importantes.

Disso tudo, podemos concluir que um `Set` não aceita elementos duplicados e que, por padrão, não é uma sequência. Entretanto, tanto as listas quanto os *sets* compartilham um elemento em comum, que são as coleções do `java.util.Collection`. Elas podem ser tanto listas, quanto um conjunto de *sets*.

Há uma interface mãe, da `List` e da `Set`, que é a `Collection`.

Em nosso código, isso significa que, ao criarmos um `ArrayList`, podemos utilizar uma interface ainda mais genérica. Se observarmos o código da própria interface `List`, veremos que ela estende `Collection`. Por isso, se ela é capaz de implementar `List`, consequentemente, ela pode fazer o mesmo com a interface `Collection`.

Temos um problema, pois alguns métodos deixam de funcionar. Por exemplo, o `get()`, que trabalha com a ideia de índices. Os índices são aplicáveis somente ao conceito de listas e, apesar de o `Collection` contemplar tanto elas quanto *sets*, ao generalizarmos, não será possível utilizarmos este tipo de métodos, já que os índices não existem para o universo dos *sets*.

Para aprofundar-se, é recomendável acessar os [cursos específicos](https://cursos.alura.com.br/course/java-util-lambdas/task/38649) (<https://cursos.alura.com.br/course/java-util-lambdas/task/38649>) em relação a este conteúdo.

Nos vemos na próxima!

Autoboxing e Unboxing

Transcrição

Você pode fazer o [DOWNLOAD \(<https://caelum-online-public.s3.amazonaws.com/850-java-util/06/java6-cap6.zip>\)](https://caelum-online-public.s3.amazonaws.com/850-java-util/06/java6-cap6.zip) do projeto da aula anterior.

Anteriormente, falamos um pouco sobre o mundo das coleções. Neste, daremos continuidade à exploração do pacote `java.util` com foco nas listas.

No pacote `br.com.bytebank.banco.test.util` criaremos uma nova classe, chamada `Teste` :

```
package br.com.bytebank.banco.test.util;

public class Teste {

    public static void main(String[] args) {

    }
}
```

[COPIAR CÓDIGO](#)

No início do curso, quando começamos a falar sobre estrutura de dados, aprendemos sobre os arrays. O primeiro que criamos foi do tipo `int`, primitivo, onde utilizamos os colchetes (`[]`), da seguinte forma:

```
package br.com.bytebank.banco.test.util;

public class Teste {

    public static void main(String[] args) {

        int[] idades = new int[5];

    }
}
```

[COPIAR CÓDIGO](#)

Trata-se da sintaxe padrão do array, onde temos um tamanho fixo. Com base nisso, falamos sobre uma série de pontos positivos e negativos deste tipo de lista, entre os negativos estão o tamanho fixo, sintaxe complicada e dificuldade em saber o número de referências efetivamente inseridas - ao passo que descobrir o seu tamanho é relativamente fácil.

Aprendemos sobre as listas, e utilizamos a interface `List`, sempre lembrando de importar o `java.util`. Com isso, criaremos um novo `ArrayList`, `numeros`:

```
package br.com.bytebank.banco.test.util;

public class Teste {

    public static void main(String[] args) {

        int[] idades = new int[5];

        List numeros = new ArrayList();

    }
}
```

[COPIAR CÓDIGO](#)

Vimos que existem arrays de primitivos, e de referências. Na forma primitiva, eles são declarados na modalidade apresentada acima, enquanto que, em referência, utilizamos a classe como tipo para defini-lo, por exemplo:

```
package br.com.bytebank.banco.test.util;

public class Teste {

    public static void main(String[] args) {

        int[] idades = new int[5];

        String[] nomes = new String[5];

        List numeros = new ArrayList();

    }
}
```

[COPIAR CÓDIGO](#)

Nos arrays primitivos, cada casa guarda o valor primitivo, enquanto que no array de referência, cada uma armazena a referência que é utilizada para encontrá-lo. Contudo, no mundo das listas, elas só podem ser de referências. Só existem coleções de referências.

Contudo, um problema surgirá disso. Criaremos uma variável que guardará um valor primitivo:

```
package br.com.bytebank.banco.test.util;

public class Teste {

    public static void main(String[] args) {

        int[] idades = new int[5];
```

```
String[] nomes = new String[5];  
  
int idade = 29;  
List numeros = new ArrayList();  
  
}  
}
```

[COPIAR CÓDIGO](#)

Dentro dela, temos o valor `29`, que é um primitivo. Queremos armazená-lo em uma lista, e daí surge o problema, a lista só é capaz de guardar referências.

Ao chamarmos o método `add()`, veremos que o Eclipse mostra que ele já espera receber uma referência do tipo `Object`, ou seja, se tentarmos adicionar o primitivo `idade`:

```
package br.com.bytebank.banco.test.util;  
  
public class Teste {  
  
    public static void main(String[] args) {  
  
        int[] idades = new int[5];  
  
        String[] nomes = new String[5];  
  
        int idade = 29;  
        List numeros = new ArrayList();  
        numeros.add(idade);  
  
    }  
}
```

[COPIAR CÓDIGO](#)

Não deveria funcionar, porque `idade` não é uma referência, logo, não é compatível com o tipo `Object`.

Inicialmente, isto realmente não funcionava, contudo, atualmente o Java cria uma solução sem que seja necessária nenhuma ação por parte do programador.

Para cada primitivo no mundo Java, existe algo que o representa no mundo orientado a objetos. Isso significa que, para cada tipo primitivo, há uma classe que o representa.

Por exemplo, para representar o `int` primitivo, existe a classe `Integer`:

```
package br.com.bytebank.banco.test.util;

public class Teste {

    public static void main(String[] args) {

        int[] idades = new int[5];

        String[] nomes = new String[5];

        int idade = 29;//Integer
        List numeros = new ArrayList();
        numeros.add(idade);

    }
}
```

[COPIAR CÓDIGO](#)

Internamente, o Java transforma o primitivo em um objeto, e armazena a referência no array.

```
package br.com.bytebank.banco.test.util;

public class Teste {

    public static void main(String[] args) {

        int[] idades = new int[5];

        String[] nomes = new String[5];

        int idade = 29;//Integer
        Integer idadeRef = new Integer(29);
        List numeros = new ArrayList();
        numeros.add(idade);

    }

}
```

[COPIAR CÓDIGO](#)

Como havíamos falado, parametrizar o `ArrayList` é uma boa prática para nossas coleções, e para isso utilizamos os *generics*, representados pelos símbolos de menor e maior (`<>`). Desta forma garantimos maior segurança, já que afastamos problemas de cast. Se tentarmos inserir um *generic* de `int`, não funcionará:

```
package br.com.bytebank.banco.test.util;

public class Teste {

    public static void main(String[] args) {

        int[] idades = new int[5];

        String[] nomes = new String[5];

        int idade = 29;//Integer
```

```
        Integer idadeRef = new Integer(29);
        List<int> numeros = new ArrayList<int>();
        numeros.add(idade);

    }

}
```

[COPIAR CÓDIGO](#)

Simplesmente porque `int` é um primitivo, não uma referência. Assim, o certo é utilizarmos o `Integer`:

```
package br.com.bytebank.banco.test.util;

public class Teste {

    public static void main(String[] args) {

        int[] idades = new int[5];

        String[] nomes = new String[5];

        int idade = 29;//Integer
        Integer idadeRef = new Integer(29);
        List<Integer> numeros = new ArrayList<Integer>();
        numeros.add(idade);

    }

}
```

[COPIAR CÓDIGO](#)

Assim, estamos utilizando a classe que representa este número primitivo no mundo orientado a objetos.

Se trocarmos `idade` por `29`, no método `add()`:

```
package br.com.bytebank.banco.test.util;

public class Teste {

    public static void main(String[] args) {

        int[] idades = new int[5];

        String[] nomes = new String[5];

        int idade = 29;//Integer
        Integer idadeRef = new Integer(29);
        List<Integer> numeros = new ArrayList<Integer>();
        numeros.add(29);

    }
}
```

[COPIAR CÓDIGO](#)

Ainda assim o código continuará funcionando, pois o Java cria um objeto para o 29 , automaticamente, e armazena sua referência no array. A ideia é auxiliar a transformar um primitivo em objeto, e vice versa. Esta transformação, que ocorre sempre automaticamente, é chamada de *Autoboxing*.

Adiante, nos aprofundaremos no funcionamento da classe Integer . Até a próxima!

Métodos da classe Integer

Transcrição

Continuando o que havíamos falado anteriormente, no Java, há uma classe para cada tipo primitivo. Para o `int`, há uma classe `Integer`.

A transformação do tipo primitivo para o objeto referência acontece automaticamente, e é chamada de ***autoboxing***. O caminho inverso é chamado de ***unboxing***.

As classes que existem para cada tipo primitivo se chamam **wrappers**. Isso porque elas "embrulham" o tipo primitivo do objeto, que internamente guarda o valor primitivo. Elas existem para que haja compatibilidade com as coleções, nos permitindo, por exemplo, guardar números dentro de uma lista.

Retornaremos à classe `Teste`, no pacote `br.com.bytebank.banco.test.util`.

//Código omitido

```
public class Teste {  
  
    public static void main(String[] args) {  
  
        int[] idades = new int[5];  
        String[] nomes = new String[5];  
  
        int idade = 29; //Integer  
        Integer idadeRef = new Integer(29);  
        List<Integer> numeros = new ArrayList<Integer>();  
    }  
}
```

```
    numeros.add(29); //Autoboxing  
}
```

[COPIAR CÓDIGO](#)

Tivemos a inicialização da variável, utilizando um primitivo:

```
//Código omitido
```

```
int idade = 29;
```

```
//Código omitido
```

[COPIAR CÓDIGO](#)

E vimos também a inicialização com o uso de uma classe *wrapper*:

```
//Código omitido
```

```
Integer idadeRef = new ~Integer~(29);
```

```
//Código omitido
```

[COPIAR CÓDIGO](#)

Percebemos que a classe `Integer` aparece riscada, o que indicada que este construtor não deveria mais estar sendo utilizado. Mas então, como construiremos o objeto? Neste caso, o construtor é descontinuado, surge a mensagem de que "*The constructor Int3eger(int) is deprecated*".

Neste caso, para criarmos um objeto, temos que utilizar a classe, com o método estático `valueOf()`, que receberá um primitivo `int`:

```
//Código omitido

public class Teste {

    public static void main(String[] args) {

        int[] idades = new int[5];
        String[] nomes = new String[5];

        int idade = 29;

        Integer idadeRef = Integer.valueOf(29);

        List<Integer> numeros = new ArrayList<Integer>();
        numeros.add(29); //Autoboxing

    }

}
```

[COPIAR CÓDIGO](#)

Ao utilizarmos o `new` em um objeto, delegamos a criação para um método. Abriremos o método `valueOf()` para visualizarmos sua construção, lembrando que estamos considerando a versão Java 9:

```
//Código omitido
public static Integer valueOf(int i) {
    if(i >= IntegerCache.low && i <= IntegerCache.high)
        return IntegerCache.cache[i + (-IntegerCache.low)];
    return new ~Integer~(i);
}
//Código omitido
```

[COPIAR CÓDIGO](#)

Vemos que, internamente, o método faz com que, caso o `IntegerCache` não funcione, exista um objeto como backup, ou seja, ele usa o construtor `new Integer()`. Podemos perceber que alguns métodos trabalham com ele.

Portanto, o jeito mais adequado de fazermos isso em nosso código é utilizando o método `valueOf()`, e termos como resultado uma referência. A partir desta referência `idadeRef`, podemos chamar métodos. Por exemplo, o `intValue()`.

Temos uma referência, que aponta para um objeto, e queremos obter desta classe `wrapper` - que embrulha o primitivo -, qual o valor que nela está contido. É para isso que o método `intValue()` é utilizado, ele nos devolve, justamente, o primitivo:

```
//Código omitido

public class Teste {

    public static void main(String[] args) {

        int[] idades = new int[5];
        String[] nomes = new String[5];

        int idade = 29;

        Integer idadeRef = Integer.valueOf(29);
        int valor = idadeRef.intValue();

        List<Integer> numeros = new ArrayList<Integer>();
        numeros.add(29); //Autoboxing

    }
}
```

[COPIAR CÓDIGO](#)

Assim, estamos fazendo o *autoboxing*:

```
//Código omitido  
  
    Integer idadeRef = Integer.valueOf(29);
```

```
//Código omitido
```

[COPIAR CÓDIGO](#)

Para, em seguida, fazer o *unboxing*:

```
//Código omitido  
  
int valor = idadeRef.intValue();
```

```
//Código omitido
```

[COPIAR CÓDIGO](#)

A classe `Integer` possui uma série de métodos auxiliares. Ela faz parte do pacote `Java.lang` e por isso não foi necessária sua importação.

Lembrando do `String[] args`, imaginemos que estamos recebendo uma `String s`, que está na primeira posição dos `args`:

```
//Código omitido  
  
public class Teste {  
  
    public static void main(String[] args) {  
  
        int[] idades = new int[5];  
        String[] nomes = new String[5];  
  
        int idade = 29;
```

```
        Integer idadeRef = Integer.valueOf(29);
        int valor = idadeRef.intValue();

        String s = args[0];

        List<Integer> numeros = new ArrayList<Integer>();
        numeros.add(29); //Autoboxing

    }

}
```

[COPIAR CÓDIGO](#)

Ele recebe um valor numérico, entretanto, todos os valores armazenados em `args` são interpretados como `String`. Isso significa que, se passarmos por exemplo o número `10`, o Java interpretará como `"10"`, entre aspas, representando uma `String`.

Entretanto, queremos descobrir o valor inteiro. Assim, precisamos transformar esta `String` em um `int`, para isso, utilizamos a classe `Integer`, e o método `valueOf()`.

Contudo, ao digitarmos o nome deste método, percebemos que ele existe em três versões, escolheremos aquela que recebe um `String s`:

```
//Código omitido

public class Teste {

    public static void main(String[] args) {

        int[] idades = new int[5];
        String[] nomes = new String[5];
```

```
int idade = 29;

Integer idadeRef = Integer.valueOf(29);
int valor = idadeRef.intValue();

String s = args[0];//"10"

Integer numero = Integer.valueOf(s);

List<Integer> numeros = new ArrayList<Integer>();
numeros.add(29); //Autoboxing

}

}
```

[COPIAR CÓDIGO](#)



Em seguida, imprimiremos o número:

```
//Código omitido

public class Teste {

    public static void main(String[] args) {

        int[] idades = new int[5];
        String[] nomes = new String[5];

        int idade = 29;

        Integer idadeRef = Integer.valueOf(29);
        int valor = idadeRef.intValue();

        String s = args[0];//"10"

        Integer numero = Integer.valueOf(s);
```

```

        System.out.println(numero);

        List<Integer> numeros = new ArrayList<Integer>();
        numeros.add(29); //Autoboxing

    }

}

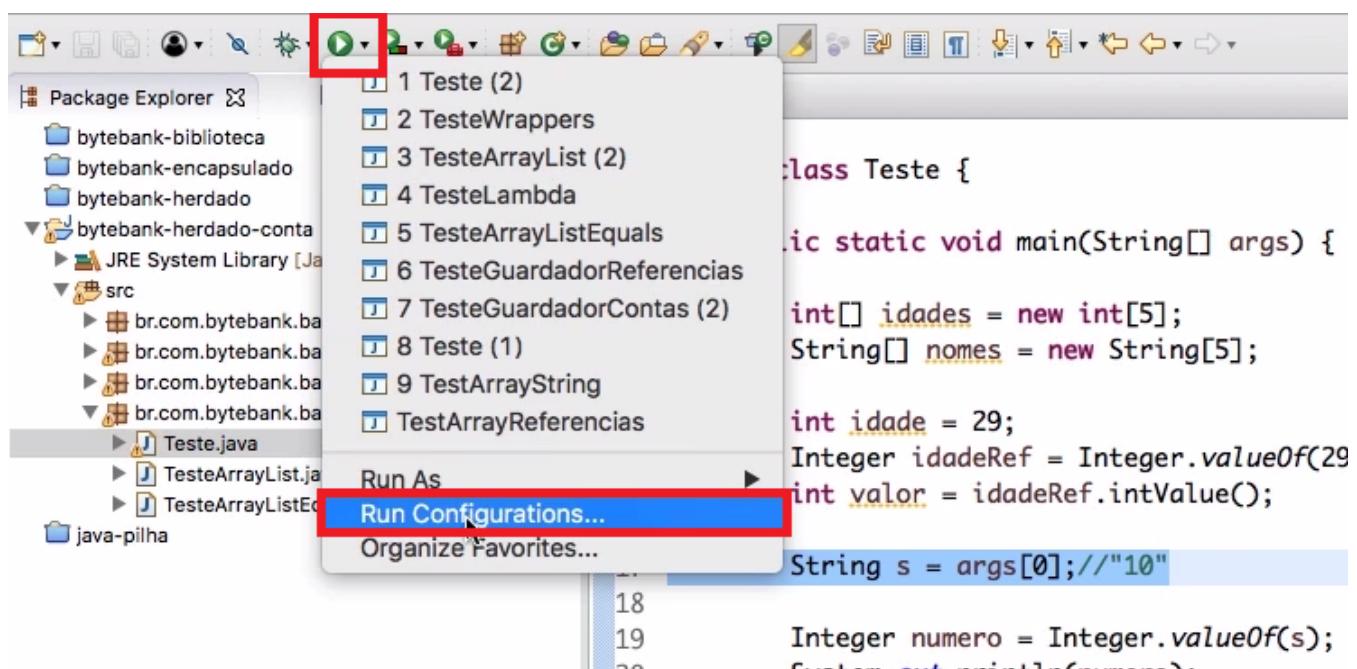

```

[COPIAR CÓDIGO](#)



O código está pronto para testarmos, mas antes, precisamos alterar as *run configurations*. Se não fizermos isso, teremos um erro, já que não passamos nenhum parâmetro para o array `args`, ou seja, ele ainda não tem nenhuma posição.

Ao lado do botão de execução, no menu superior, clicaremos na seta ao lado do botão com símbolo de play, na cor verde e branca, e selecionaremos no menu a opção "Run Configurations...":



Selecionaremos a classe `Teste`, com atenção para escolhermos a correta. Na barra de menu superior, selecionaremos a aba "Arguments", e preencheremos os

argumentos com o valor `12`. Em seguida, clicaremos em "Apply" e executaremos a classe.

Temos o seguinte resultado no console:

`12`

[COPIAR CÓDIGO](#)

Funcionou! O programa foi capaz de transformar o `String s` em um inteiro `int`. Esta transformação se chama *parsing*.

Isto é muito comum em situações onde temos um formulário, em que são digitados determinados valores. Estes, normalmente, vêm como `String s`, e é uma tarefa comum do desenvolvedor transformá-los deste tipo para algum outro específico.

Além de termos um objeto e referência `Integer`, podemos também ter um método que cria o primitivo a partir da `String` diretamente, da seguinte forma:

//Código omitido

```
public class Teste {  
  
    public static void main(String[] args) {  
  
        int[] idades = new int[5];  
        String[] nomes = new String[5];  
  
        int idade = 29;  
  
        Integer idadeRef = Integer.valueOf(29);  
        int valor = idadeRef.intValue();  
  
        String s = args[0];//"10"
```

```
//Integer numero = Integer.valueOf(s);

int numero = Integer.parseInt(s);

System.out.println(numero);

List<Integer> numeros = new ArrayList<Integer>();
numeros.add(29); //Autoboxing

}

}
```

[COPIAR CÓDIGO](#)

Executaremos a classe novamente, e temos o mesmo resultado no console.

A classe `Integer` possui diversos métodos auxiliares, mas alguns que são essenciais são o `valueOf(int i)`, que corresponde ao *autoboxing*, `intValue()`, para o *unboxing*, `valueOf(String s)` e `parseInt()`.

Temos ainda um método que nos permite transformar um `Integer` para outros primitivos, por exemplo, em um `double`, neste caso, utilizamos o `doubleValue()`:

```
//Código omitido

public class Teste {

    public static void main(String[] args) {

        int[] idades = new int[5];
        String[] nomes = new String[5];

        int idade = 29;
```

```
Integer idadeRef = Integer.valueOf(29); //autobox

System.out.println(idadeRef.doubleValue());

int valor = idadeRef.intValue(); //unboxing

String s = args[0];//"10"

//Integer numero = Integer.valueOf(s);

int numero = Integer.parseInt(s);

System.out.println(numero);

List<Integer> numeros = new ArrayList<Integer>();
numeros.add(29); //Autoboxing

}

}
```

[COPIAR CÓDIGO](#)

Ao executarmos a classe, temos o seguinte resultado no console:

29.0

12

[COPIAR CÓDIGO](#)

Além disso, e removendo trechos de código que não são mais relevantes, veremos também algumas constantes. Em alguns casos, precisaremos estabelecer, por exemplo, valores mínimos e máximos para que sirvam de parâmetros em alguma regra de negócio específica!. Para sabermos, por exemplo, o valor máximo, podemos utilizar `MAX_VALUE` :

```
//Código omitido

public class Teste {

    public static void main(String[] args) {

        int idade = 29;
        Integer idadeRef = Integer.valueOf(29); //autobox
        System.out.println(idadeRef.doubleValue());

        System.out.println(Integer.MAX_VALUE);

        int valor = idadeRef.intValue(); //unboxing
        String s = args[0];//"10"
        //Integer numero = Integer.valueOf(s);
        int numero = Integer.parseInt(s);
        System.out.println(numero);

        List<Integer> numeros = new ArrayList<Integer>();
        numeros.add(29); //Autoboxing

    }

}
```

[COPIAR CÓDIGO](#)



Cada `Integer` é capaz de um certo limite de armazenamento, de 4 Bytes. Assim, executaremos a classe, e temos o seguinte resultado no console:

```
29.0
2147483647
12
```

[COPIAR CÓDIGO](#)

Para descobrirmos o valor mínimo, utilizamos o `MIN_VALUE` :

```
//Código omitido

public class Teste {

    public static void main(String[] args) {

        int idade = 29;
        Integer idadeRef = Integer.valueOf(29); //autobox
        System.out.println(idadeRef.doubleValue());

        System.out.println(Integer.MIN_VALUE);

        int valor = idadeRef.intValue(); //unboxing
        String s = args[0];//"10"
        //Integer numero = Integer.valueOf(s);
        int numero = Integer.parseInt(s);
        System.out.println(numero);

        List<Integer> numeros = new ArrayList<Integer>();
        numeros.add(29); //Autoboxing

    }

}
```

[COPIAR CÓDIGO](#)

Executando a classe, temos o seguinte resultado:

```
29.0
-2147483648
12
```

[COPIAR CÓDIGO](#)

Podemos, ainda, descobrir o tamanho do `Integer`, utilizando o `SIZE`:

```
//Código omitido

public class Teste {

    public static void main(String[] args) {

        int idade = 29;
        Integer idadeRef = Integer.valueOf(29); //autobox
        System.out.println(idadeRef.doubleValue());

        System.out.println(Integer.MAX_VALUE);
        System.out.println(Integer.MIN_VALUE);

        System.out.println(Integer.SIZE);

        int valor = idadeRef.intValue(); //unboxing
        String s = args[0];//"10"
        //Integer numero = Integer.valueOf(s);
        int numero = Integer.parseInt(s);
        System.out.println(numero);

        List<Integer> numeros = new ArrayList<Integer>();
        numeros.add(29); //Autoboxing

    }
}
```

[COPIAR CÓDIGO](#)

Executaremos a classe, e temos o seguinte resultado no console:

```
29.0
2147483648
-2147483648
```

32

12

[COPIAR CÓDIGO](#)

Indicando um tamanho de 32 bits. Se quisermos saber a quantidade de bytes, utilizamos o `BYTES` :

```
//Código omitido

public class Teste {

    public static void main(String[] args) {

        int idade = 29;
        Integer idadeRef = Integer.valueOf(29); //autobox
        System.out.println(idadeRef.doubleValue());

        System.out.println(Integer.MAX_VALUE);
        System.out.println(Integer.MIN_VALUE);

        System.out.println(Integer.SIZE);
        System.out.println(Integer.BYTES);

        int valor = idadeRef.intValue(); //unboxing
        String s = args[0];//"10"
        //Integer numero = Integer.valueOf(s);
        int numero = Integer.parseInt(s);
        System.out.println(numero);

        List<Integer> numeros = new ArrayList<Integer>();
        numeros.add(29); //Autoboxing

    }
}
```

[COPIAR CÓDIGO](#)

Executaremos a classe, e temos o seguinte resultado no console:

```
29.0  
2147483648  
-2147483648  
32  
4  
12
```

[COPIAR CÓDIGO](#)

Indicando um total de 4 Bytes, como havia sido mencionado anteriormente.

Por todos estes motivos, é válido estudar [documentação](#) (<https://docs.oracle.com/javase/7/docs/api/java/lang/Integer.html>). Java referente a este ponto.

Adiante, veremos outros exemplos de *wrappers*. Até a próxima!

A classe Number

Transcrição

Nesta aula, dando continuidade às anteriores, veremos mais alguns exemplos de classes *wrappers*.

Na tabela abaixo, temos alguns tipos primitivos e as classes às quais estão associados:

Tamanho	Tipo primitivo	Wrappers
8 bytes	double	java.lang.Double
4 bytes	float	java.lang.Float
8 bytes	long	java.lang.Long
4 bytes	int	java.lang.Integer
2 bytes	short	java.lang.Short
1 bytes	byte	java.lang.Byte
2 bytes	char	java.lang.Character
	boolean	java.lang.Boolean

Sendo que, `double` e `float` são flutuantes, `long`, `int`, `short` e `byte` são inteiros, `char` representa um caractere, e por fim, temos um booleano.

Temos a informação do tamanho do tipo primitivo. Mas, o tamanho do objeto nos é desconhecido, ele será criado pela máquina virtual na memória posteriormente, e não fica a cargo do desenvolvedor Java.

Renomearemos a classe `Teste` do pacote `br.com.bytebank.banco.test.util`, ela passará a se chamar `TesteWrapperInteger`. No mesmo pacote, criaremos uma nova classe, chamada `TesteOutrosWrappers`, que também terá o *autoboxing* e o *unboxing*:

```
package br.com.bytebank.banco.test.util;

public class TesteOutrosWrappers {

    public static void main(String[] args) {

        Integer idadeRef = Integer.valueOf(29); //autobox
        System.out.println(idadeRef.intValue()); //unboxi

    }
}
```

[COPIAR CÓDIGO](#)

Teremos um `double`, que chamaremos de `dRef`, e cujo valor será `3.2`, basta escrevermos o valor que o Java entenderá como um `double`, e fará o *autoboxing*:

```
package br.com.bytebank.banco.test.util;

public class TesteOutrosWrappers {

    public static void main(String[] args) {

        Integer idadeRef = Integer.valueOf(29); //autobox
        System.out.println(idadeRef.intValue()); //unboxi

        Double dRef = 3.2;
    }
}
```

[COPIAR CÓDIGO](#)

Internamente, o Java utiliza a classe *wrapper* Double , e o método valueOf() :

```
package br.com.bytebank.banco.test.util;

public class TesteOutrosWrappers {

    public static void main(String[] args) {

        Integer idadeRef = Integer.valueOf(29); //autobox
        System.out.println(idadeRef.intValue()); //unboxi

        Double dRef = Double.valueOf(3.2);
    }
}
```

[COPIAR CÓDIGO](#)

Em seguida, utilizando System.out.println() , faremos o *unboxing*:

```
package br.com.bytebank.banco.test.util;

public class TesteOutrosWrappers {

    public static void main(String[] args) {

        Integer idadeRef = Integer.valueOf(29); //autobox
        System.out.println(idadeRef.intValue()); //unboxi

        Double dRef = Double.valueOf(3.2); //autoboxing
        System.out.println(dRef.doubleValue()); //unboxing
    }
}
```

[COPIAR CÓDIGO](#)

No primeiro caso, utilizamos um `intValue()`, enquanto que no segundo, um `doubleValue()`. Executaremos a classe e temos o seguinte resultado no console:

29

3.2

[COPIAR CÓDIGO](#)

Tudo está funcionando corretamente.

Em seguida, faremos mais um teste, desta vez com o tipo `boolean`. Este possui uma particularidade, já que deve possuir dois valores, um para representar o verdadeiro, `true`, e outro que representa o falso, `false`. Eles já existem, como constantes, `TRUE` e `FALSE`.

Importante notar que, no Java, as constantes são sempre escritas em letras maiúsculas.

Assim como anteriormente, faremos o *unboxing* utilizando o `booleanValue()`:

```
package br.com.bytebank.banco.test.util;

public class TesteOutrosWrappers {

    public static void main(String[] args) {

        Integer idadeRef = Integer.valueOf(29); //autobox
        System.out.println(idadeRef.intValue()); //unboxi

        Double dRef = Double.valueOf(3.2); //autoboxing
        System.out.println(dRef.doubleValue()); //unboxing

        Boolean bRef = Boolean.FALSE;
        System.out.println(bRef.booleanValue());
```

```
}
```

[COPIAR CÓDIGO](#)

Poderíamos, também, ter utilizado o valor primitivo `false` e contar com o Java para que ele criasse o tipo automaticamente. Executaremos a classe:

29

3.2

false

[COPIAR CÓDIGO](#)

E o resultado no console nos indica que tudo está funcionando corretamente.

Como havíamos mencionado, temos tipos flutuantes e inteiros, e o que eles têm em comum é que ambos são numéricos. Assim, há uma classe mãe que os conecta, a `java.lang.Number`. Seguinte este conceito, as seguintes classes a estendem: `Double`, `Float`, ambas flutuantes, `Long`, `Integer`, `Short` e `Byte`, representando os inteiros.

Em nosso código, isso significa que podemos utilizar a classe `Number` para criar uma referência:

```
package br.com.bytebank.banco.test.util;

public class TesteOutrosWrappers {

    public static void main(String[] args) {
        Integer idadeRef = Integer.valueOf(29); //autobox
        System.out.println(idadeRef.intValue()); //unboxi

        Double dRef = Double.valueOf(3.2); //autoboxing
```

```
        System.out.println(dRef.doubleValue());//unboxing

        Boolean bRef = Boolean.FALSE;
        System.out.println(bRef.booleanValue());

    Number refNumero = Integer.valueOf(29);
}
}
```

[COPIAR CÓDIGO](#)

No caso criamos um `int`, mas também poderíamos ter utilizado qualquer outro tipo, como `double`, por exemplo. O que temos neste caso é uma referência genérica, capaz de referenciar tipos mais específicos.

Pensando em herança, e no polimorfismo, quais seriam então os métodos definidos pela classe `Number`? Há diversos, dentre eles, podemos citar `byteValue()`, `doubleValue()` e `floatValue()`, todos métodos que já estudamos quando abordamos a classe `Integer`.

Esta classe pode ser útil, por exemplo, se quisermos criar uma lista que aceite qualquer tipo de valor numérico, para isso, poderíamos fazer o seguinte:

```
package br.com.bytebank.banco.test.util;

public class TesteOutrosWrappers {

    public static void main(String[] args) {

        Integer idadeRef = Integer.valueOf(29); //autobox
        System.out.println(idadeRef.intValue()); //unboxi

        Double dRef = Double.valueOf(3.2);//autoboxing
        System.out.println(dRef.doubleValue());//unboxing

        Boolean bRef = Boolean.FALSE;
```

```
System.out.println(bRef.booleanValue());  
  
Number refNumero = Integer.valueOf(29);  
  
List<Number> lista = new ArrayList<>();  
}  
}
```

[COPIAR CÓDIGO](#)

Sendo que não há necessidade de repetirmos a palavra `Number` no lado direito do construtor.

Com isso, temos uma lista em que podemos guardar tanto um `int`, quanto um `double`, ou ainda um `float`:

```
package br.com.bytebank.banco.test.util;  
  
public class TesteOutrosWrappers {  
  
    public static void main(String[] args) {  
  
        Integer idadeRef = Integer.valueOf(29); //autobox  
        System.out.println(idadeRef.intValue()); //unboxing  
  
        Double dRef = Double.valueOf(3.2); //autoboxing  
        System.out.println(dRef.doubleValue()); //unboxing  
  
        Boolean bRef = Boolean.FALSE;  
        System.out.println(bRef.booleanValue());  
  
        Number refNumero = Integer.valueOf(29);  
  
        List<Number> lista = new ArrayList<>();  
        lista.add(10);  
        lista.add(32.6);
```

```
    lista.add(25.6f);  
}  
}
```

[COPIAR CÓDIGO](#)

Isso funciona graças ao uso da classe `Number`, de cunho genérico.

A existência de primitivos e *wrappers* é explicada pelo momento da criação do Java, à época, a capacidade de processamento das máquinas era limitado, e a memória era custosa, portanto, pensando em questões de desempenho, e memória, importante a existência dos primitivos. Eles são mais rápidos, e ocupam menos espaço.

Hoje, isso não é mais um problema, sua existência se justifica apenas historicamente, como um legado.

Até a próxima!

Ordenando listas

Transcrição

Você pode fazer o [DOWNLOAD \(<https://caelum-online-public.s3.amazonaws.com/850-java-util/07/java6-cap7.zip>\)](https://caelum-online-public.s3.amazonaws.com/850-java-util/07/java6-cap7.zip) do projeto da aula anterior.

Dando continuidade à viagem pelo `java.util`, nesta aula, falaremos sobre os *lambdas*.

No pacote `br.com.bytebank.banco.test.util`, criaremos uma nova classe, chamada `Teste`, com o seguinte código:

```
package br.com.bytebank.banco.test.util;

public class Teste {

    public static void main(String[] args) {

        Conta cc1 = new ContaCorrente(22, 33);
        cc1.deposita(333.0);

        Conta cc2 = new ContaPoupanca(22, 44);
        cc2.deposita(444.0);

        Conta cc3 = new ContaCorrente(22, 11);
        cc3.deposita(111.0);

        Conta cc4 = new ContaPoupanca(22, 22);
```

```
        cc4.deposita(222.0);

        List<Conta> lista = new ArrayList<>();
        lista.add(cc1);
        lista.add(cc2);
        lista.add(cc3);
        lista.add(cc4);

    }

}
```

[COPIAR CÓDIGO](#)

Temos duas contas correntes, duas contas poupança, e um `ArrayList`, ao qual as estamos adicionando. O código ainda não está compilando pois ainda não fizemos os `imports`.

Podemos realizá-los automaticamente. Na barra de menu superior, na opção "Source", selecionaremos a opção "Organize Imports", podemos acessar esta opção também por meio do atalho "Shift + Command + O/Shift + Ctrl + O".

Surgirá uma caixa de diálogo, para selecionarmos a lista que desejamos utilizar, selecionaremos `java.util.List`. Temos o seguinte resultado:

```
package br.com.bytebank.banco.test.util;

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

import br.com.bytebank.banco.modelo.Conta;
import br.com.bytebank.banco.modelo.ContaCorrente;
import br.com.bytebank.banco.modelo.ContaPoupanca;

public class Teste {
```

```
public static void main(String[] args) {  
  
    Conta cc1 = new ContaCorrente(22, 33);  
    cc1.deposita(333.0);  
  
    Conta cc2 = new ContaPoupanca(22, 44);  
    cc2.deposita(444.0);  
  
    Conta cc3 = new ContaCorrente(22, 11);  
    cc3.deposita(111.0);  
  
    Conta cc4 = new ContaPoupanca(22, 22);  
    cc4.deposita(222.0);  
  
    List<Conta> lista = new ArrayList<>();  
    lista.add(cc1);  
    lista.add(cc2);  
    lista.add(cc3);  
    lista.add(cc4);  
  
}  
  
}
```

[COPIAR CÓDIGO](#)

Tudo está compilando.

Nosso objetivo agora será ordenar nossa lista. Temos diversas contas em nossa lista, e as ordenaremos de acordo com um determinado critério.

O algoritmo de ordenação do Java já está implementado, o desenvolvedor não precisa ter esta preocupação.

No nascimento do pacote `java.util` havia uma classe separada, destinada exclusivamente à função de ordenação. Entretanto, em versões mais recentes da

linguagem, houve a implementação de um método específico, tendo em mente a orientação a objetos.

O método utilizado chama-se `sort()`, e exige um "*Comparator*", `c`. Ele representa o critério de ordenação, serve para comparar duas referências, por meio de um método presente na interface `Comparator`.

No mesmo arquivo `Teste.java`, criaremos uma nova classe, chamada `NaoSeiAinda`, que implementará a interface `Comparator`:

```
package br.com.bytebank.banco.test.util;

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

import br.com.bytebank.banco.modelo.Conta;
import br.com.bytebank.banco.modelo.ContaCorrente;
import br.com.bytebank.banco.modelo.ContaPoupanca;

public class Teste {

    public static void main(String[] args) {

        Conta cc1 = new ContaCorrente(22, 33);
        cc1.deposita(333.0);

        Conta cc2 = new ContaPoupanca(22, 44);
        cc2.deposita(444.0);

        Conta cc3 = new ContaCorrente(22, 11);
        cc3.deposita(111.0);

        Conta cc4 = new ContaPoupanca(22, 22);
        cc4.deposita(222.0);

        List<Conta> lista = new ArrayList<>();
```

```
        lista.add(cc1);
        lista.add(cc2);
        lista.add(cc3);
        lista.add(cc4);

    }

}

class NaoSeiAinda implements Comparator<>
```

[COPIAR CÓDIGO](#)

Ela exige que seja definido seu tipo, no nosso caso, como queremos que ela compare contas, nosso tipo será `Conta` :

```
package br.com.bytebank.banco.test.util;

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

import br.com.bytebank.banco.modelo.Conta;
import br.com.bytebank.banco.modelo.ContaCorrente;
import br.com.bytebank.banco.modelo.ContaPoupanca;

public class Teste {

    public static void main(String[] args) {

        Conta cc1 = new ContaCorrente(22, 33);
        cc1.deposita(333.0);

        Conta cc2 = new ContaPoupanca(22, 44);
        cc2.deposita(444.0);

        Conta cc3 = new ContaCorrente(22, 11);
        cc3.deposita(111.0);
```

```
    Conta cc4 = new ContaPoupanca(22, 22);
    cc4.deposita(222.0);

    List<Conta> lista = new ArrayList<>();
    lista.add(cc1);
    lista.add(cc2);
    lista.add(cc3);
    lista.add(cc4);

}

class NaoSeiAinda implements Comparator<Conta>
```

[COPIAR CÓDIGO](#)

Precisamos importar a interface `Comparator`. Com o mouse sobre ela, o Eclipse mostrará algumas opções, dentre elas "Import `Comparator` (`java.util`)", clicaremos sobre ela, que é a primeira da lista. No começo do nosso código, onde estão listados todos os *imports*, já aparece este novo:

```
//Código omitido

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

//Código omitido

[COPIAR CÓDIGO](#)

Contudo, a classe `NaoSeiAinda` não está compilando. Isso acontece porque ainda precisamos implementar o método `compare()`. Com a ajuda do Eclipse, clicaremos sobre o ícone de lâmpada, que aparece ao lado esquerdo da linha q.^{..^}

não compila, e ela nos dará a opção "Add unimplemented methods". Temos assim a seguinte formulação automática, criada pelo Eclipse:

```
//Código omitido

public class Teste {

    public static void main(String[] args) {

        //Código omitido

    }

}

class NaoSeiAinda implements Comparator<Conta> {

    @Override
    public int compare(Conta arg0, Conta arg1) {
        // TODO Auto-generated method stub
        return 0;
    }
}
```

[COPIAR CÓDIGO](#)

Como utilizamos o parâmetro `Conta` na interface `Comparator`, nosso método `compare()` trabalha com o mesmo, afinal, nosso objetivo é comparar duas contas.

Apenas para nossa didática, alteraremos o `arg0` e `arg1` para `c1` e `c2`.

O objetivo deste método é poder determinar quando que uma conta é maior, ou menor, do que outra - e qual o critério de avaliação. Quanto a este último, quem deve defini-lo somos nós, os desenvolvedores. Podemos, por exemplo, determinar que o saldo será o critério determinante, ou o titular, o número da agência, enfim o que acharmos mais conveniente.

Definiremos o nosso critério de avaliação como o número das contas. Com isso, alteraremos o nome da classe para `NumeroDaContaComparator` :

```
//Código omitido

public class Teste {

    public static void main(String[] args) {

        //Código omitido

    }

}

class NumeroDaContaComparator implements Comparator<Conta> {

    @Override
    public int compare(Conta c1, Conta c2) {
        // TODO Auto-generated method stub
        return 0;
    }
}
```

[COPIAR CÓDIGO](#)

O funcionamento do método `compare()` terá um funcionamento similar ao do `equals()`. Teremos um retorno `0` somente se o número das contas forem idênticos, se o número da conta `c1` for inferior ao da `c2`, nosso retorno será um valor negativo, aleatório, pode ser por exemplo `-345` :

```
//Código omitido

public class Teste {

    public static void main(String[] args) {
```

```
//Código omitido

}

}

class NumeroDaContaComparador implements Comparator<Conta> {

    @Override
    public int compare(Conta c1, Conta c2) {

        if(c1.getNumero() < c2.getNumero()) {
            return -345;
        }

        return 0;
    }
}
```

[COPIAR CÓDIGO](#)

Evidentemente que, se `c1` é maior que `c2`, então devemos ter um resultado positivo. Assim, criaremos um segundo `if` que reflete esta regra:

```
//Código omitido

public class Teste {

    public static void main(String[] args) {

        //Código omitido

    }
}
```

```
class NumeroDaContaComparator implements Comparator<Conta> {

    @Override
    public int compare(Conta c1, Conta c2) {

        if(c1.getNumero() < c2.getNumero()) {
            return -1;
        }

        if(c1.getNumero() > c2.getNumero()) {
            return 1;
        }

        return 0;
    }
}
```

[COPIAR CÓDIGO](#)

Apenas por motivos de organização, alteramos os valores dos retornos para `-1` e `1`, respectivamente. As regras do método `compare()` podem ser encontradas na documentação Java, mas podemos ver os detalhes pertinentes à ele, com o cursor posicionado sobre seu nome.

Com isso, finalizamos a construção de nossa classe. Notamos que encapsula a execução de um método somente, isso é pouco usual e causa estranhamente - adiante falaremos mais sobre isto.

Nosso próximo passo será criar uma instância da classe `NumeroDaContaComparator`, para isso, criaremos um objeto desta classe:

```
//Código omitido

public class Teste {

    public static void main(String[] args) {
```

```
//Código omitido

List<Conta> lista = new ArrayList<>();
lista.add(cc1);
lista.add(cc2);
lista.add(cc3);
lista.add(cc4);

NumeroDaContaComparador comparator = new NumeroDaContaCom

lista.sort(c);

}

}

class NumeroDaContaComparador implements Comparator<Conta> {

    @Override
    public int compare(Conta c1, Conta c2) {

        if(c1.getNumero() < c2.getNumero()) {
            return -1;
        }

        if(c1.getNumero() > c2.getNumero()) {
            return 1;
        }

        return 0;
    }
}
```

COPIAR CÓDIGO

Lembrando que, se não definirmos nenhum construtor para a classe, é utilizado o construtor padrão. Uma vez instanciada, podemos inseri-la no método `sort()`:

```
//Código omitido

public class Teste {

    public static void main(String[] args) {

        //Código omitido

        List<Conta> lista = new ArrayList<>();
        lista.add(cc1);
        lista.add(cc2);
        lista.add(cc3);
        lista.add(cc4);

        NumeroDaContaComparador comparator = new NumeroDaContaCom

        lista.sort(comparator);

    }

}

class NumeroDaContaComparador implements Comparator<Conta> {

    @Override
    public int compare(Conta c1, Conta c2) {

        if(c1.getNumero() < c2.getNumero()) {
            return -1;
        }

        if(c1.getNumero() > c2.getNumero()) {
            return 1;
        }

    }
}
```

```
    return 0;
}
}
```

[COPIAR CÓDIGO](#)

O código agora compila, e com isso, estamos ordenando nossa lista, com base em nosso critério - que é a numeração das contas.

Em seguida, testaremos nosso código. Para isso, criaremos dois laços, um antes e outro depois da comparação. Ao digitarmos `foreach` e utilizarmos o atalho "Ctrl + Espaço", o Eclipse gera o código automaticamente:

```
//Código omitido

public class Teste {

    public static void main(String[] args) {

        //Código omitido

        List<Conta> lista = new ArrayList<>();
        lista.add(cc1);
        lista.add(cc2);
        lista.add(cc3);
        lista.add(cc4);

        foreach (Conta conta : lista) {

        }

        NumeroDaContaComparator comparator = new NumeroDaContaCom
            lista.sort(comparator);
    }
}
```

```
}

}

class NumeroDaContaComparador implements Comparator<Conta> {

    @Override
    public int compare(Conta c1, Conta c2) {

        if(c1.getNumero() < c2.getNumero()) {
            return -1;
        }

        if(c1.getNumero() > c2.getNumero()) {
            return 1;
        }

        return 0;
    }
}
```

[COPIAR CÓDIGO](#)



Faremos com que, a cada iteração, seja impressa a conta :

```
//Código omitido

public class Teste {

    public static void main(String[] args) {

        //Código omitido

        List<Conta> lista = new ArrayList<>();
        lista.add(cc1);
        lista.add(cc2);
        lista.add(cc3);
```

```
        lista.add(cc4);

        foreach (Conta conta : lista) {
            System.out.println(conta);
        }

        NumeroDaContaComparador comparator = new NumeroDaContaCom

        lista.sort(comparator);

    }

}

class NumeroDaContaComparador implements Comparator<Conta> {

    @Override
    public int compare(Conta c1, Conta c2) {

        if(c1.getNumero() < c2.getNumero()) {
            return -1;
        }

        if(c1.getNumero() > c2.getNumero()) {
            return 1;
        }

        return 0;
    }
}
```

COPIAR CÓDIGO

A seguir, faremos um segundo laço, após a comparação:

```
//Código omitido
```

```
public class Teste {

    public static void main(String[] args) {

        //Código omitido

        List<Conta> lista = new ArrayList<>();
        lista.add(cc1);
        lista.add(cc2);
        lista.add(cc3);
        lista.add(cc4);

        foreach (Conta conta : lista) {
            System.out.println(conta);
        }

        NumeroDaContaComparador comparator = new NumeroDaContaCom

        System.out.println("-----");

        lista.sort(comparator);

        for (Conta conta : lista) {
            System.out.println(conta);
        }

    }

}

class NumeroDaContaComparador implements Comparator<Conta> {

    @Override
    public int compare(Conta c1, Conta c2) {

        if(c1.getNumero() < c2.getNumero()) {
            return -1;
        }
    }
}
```

```
        if(c1.getNumero() > c2.getNumero()) {  
            return 1;  
        }  
  
        return 0;  
    }  
}
```

[COPIAR CÓDIGO](#)

Com isso temos, em ordem, a criação das contas, a criação da lista e o comparador. Estamos prontos para testar nossa classe. Executaremos e temos o seguinte resultado no console:

```
ContaCorrente, Numero: 33, Agencia: 22  
ContaPoupanca, Numero: 44, Agencia: 22  
ContaCorrente, Numero: 11, Agencia: 22  
ContaPoupanca, Numero: 22, Agencia: 22  
-----  
ContaCorrente, Numero: 11, Agencia: 22  
ContaPoupanca, Numero: 22, Agencia: 22  
ContaCorrente, Numero: 33, Agencia: 22  
ContaPoupanca, Numero: 44, Agencia: 22
```

[COPIAR CÓDIGO](#)

Temos assim os dois resultados apresentados corretamente, primeiro antes da comparação e, em seguida, após a comparação. Como podemos ver, na segunda lista, as contas estão ordenadas de menor para o maior número de conta.

Assim como definimos o critério de numeração, outras classes poderiam ser criadas, com outros critérios de ordenação. Adiante, veremos como ordenar pelo titular, e como as coisas funcionavam antes da existência do método `sort()`, que surgiu na versão Java 8. Até a próxima!

Comparando Strings

Transcrição

Dando continuidade à aula anterior, nesta, criaremos mais um `comparator`.

Previamente, havíamos criado uma classe com o intuito de encapsular somente um método específico, cuja execução efetua a comparação utilizando os parâmetros definidos, e em seguida, fizemos sua implementação. Fizemos isto baseado em um critério numérico, entretanto, nesta aula, veremos como isso pode ser feito com base em um `String`, já que neste caso a comparação envolverá vários caracteres.

Para dar início, nossas quatro contas serão reformuladas e passarão a conter as seguintes informações:

```
//Código omitido

public class Teste {

    public static void main(String[] args) {

        Conta cc1 = new ContaCorrente(22, 33);
        Cliente clienteCC1 = new Cliente();
        clienteCC1.setNome("Nico");
        cc1.setTitular(clienteCC1);
        cc1.deposita(333.0);

        Conta cc2 = new ContaPoupanca(22, 44);
        Cliente clienteCC2 = new Cliente();
```

```
clienteCC2.setNome("Guilherme");
cc2.setTitular(clienteCC2);
cc2.deposita(444.0);

Conta cc3 = new ContaCorrente(22, 11);
Cliente clienteCC3 = new Cliente();
clienteCC3.setNome("Paulo");
cc3.setTitular(clienteCC3);
cc3.deposita(111.0);

Conta cc4 = new ContaPoupanca(22, 22);
Cliente clienteCC4 = new Cliente();
clienteCC4.setNome("Ana");
cc4.setTitular(clienteCC4);
cc4.deposita(222.0);

List<Conta> lista = new ArrayList<>();
lista.add(cc1);
lista.add(cc2);
lista.add(cc3);
lista.add(cc4);

for (Conta conta : lista) {
    System.out.println(conta);
}

NúmeroDaContaComparator comparator = new
System.out.println("-----");

for (Conta conta : lista) {
    System.out.println(conta);
}

lista.sort(comparator);

}
```

```
}

class NumeroDaContaComparador implements Comparator<Conta> {

    @Override
    public int compare(Conta c1, Conta c2) {

        if(c1.getNumero() < c2.getNumero()) {
            return -1;
        }

        if(c1.getNumero() > c2.getNumero()) {
            return 1;
        }

        return 0;
    }
}
```

[COPIAR CÓDIGO](#)



Importaremos a classe `Cliente`, da mesma forma como vimos anteriormente. Isso porque queremos que a conta tenha um titular. Portanto, criamos os clientes, demos um nome a cada um deles, associamos a referência da conta ao titular, com o método `setTitular()`.

A seguir, definiremos mais um critério de comparação, para isso, criaremos mais uma classe, à qual daremos o nome de `TitularDaContaComparador`, que também terá como parâmetro `Conta`:

```
//Código omitido

public class Teste {

    public static void main(String[] args) {
```

```
Conta cc1 = new ContaCorrente(22, 33);
Cliente clienteCC1 = new Cliente();
clienteCC1.setNome("Nico");
cc1.setTitular(clienteCC1);
cc1.deposita(333.0);
```

```
Conta cc2 = new ContaPoupanca(22, 44);
Cliente clienteCC2 = new Cliente();
clienteCC2.setNome("Guilherme");
cc2.setTitular(clienteCC2);
cc2.deposita(444.0);
```

```
Conta cc3 = new ContaCorrente(22, 11);
Cliente clienteCC3 = new Cliente();
clienteCC3.setNome("Paulo");
cc3.setTitular(clienteCC3);
cc3.deposita(111.0);
```

```
Conta cc4 = new ContaPoupanca(22, 22);
Cliente clienteCC4 = new Cliente();
clienteCC4.setNome("Ana");
cc4.setTitular(clienteCC4);
cc4.deposita(222.0);
```

```
List<Conta> lista = new ArrayList<>();
lista.add(cc1);
lista.add(cc2);
lista.add(cc3);
lista.add(cc4);
```

```
for (Conta conta : lista) {
    System.out.println(conta);
}
```

```
NumeroDaContaComparator comparator = |
```

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

        for (Conta conta : lista) {
            System.out.println(conta);
        }

        lista.sort(comparator);

    }

}

class TitularDaContaComparator implements Comparator<Conta> {

}

class NumeroDaContaComparator implements Comparator<Conta> {

    @Override
    public int compare(Conta c1, Conta c2) {

        if(c1.getNumero() < c2.getNumero()) {
            return -1;
        }

        if(c1.getNumero() > c2.getNumero()) {
            return 1;
        }

        return 0;
    }
}
```

COPIAR CÓDIGO

Temos um problema de compilação, pois ainda não implementamos o método. Assim, faremos a implementação automática sugerida pelo Eclipse, e

configuraremos posteriormente:

```
//Código omitido

public class Teste {
//Código omitido
}

class TitularDaContaComparator implements Comparator<Conta> {

    @Override
    public int compare(Conta arg0, Conta arg1) {
        // TODO Auto-generated method stub
        return 0;
    }
}

class NumeroDaContaComparator implements Comparator<Conta> {

    @Override
    public int compare(Conta c1, Conta c2) {

        if(c1.getNumero() < c2.getNumero()) {
            return -1;
        }

        if(c1.getNumero() > c2.getNumero()) {
            return 1;
        }

        return 0;
    }
}
```

[COPIAR CÓDIGO](#)

Substituiremos os argumentos por `c1` e `c2`, assim como havíamos feito anteriormente. O retorno `0` será mantido, para a hipótese de as contas serem iguais. Em seguida, criaremos os "if's" para compararmos os nomes dos titulares de cada conta. Em primeiro lugar, precisamos descobrir o nome do titular, utilizando o método `getTitular()`, acessando a classe `Conta`. Em cima da referência do titular, podemos utilizar o método `getNome()` imediatamente:

```
//Código omitido

public class Teste {
//Código omitido
}

class TitularDaContaComparator implements Comparator<Conta> {

    @Override
    public int compare(Conta c1, Conta c2) {

        String nomeC1 = c1.getTitular().getNome();

        return 0;
    }
}

class NumeroDaContaComparator implements Comparator<Conta> {

    @Override
    public int compare(Conta c1, Conta c2) {

        if(c1.getNumero() < c2.getNumero()) {
            return -1;
        }

        if(c1.getNumero() > c2.getNumero()) {
            return 1;
        }
    }
}
```

```
        return 0;
    }
}
```

[COPIAR CÓDIGO](#)

Este processo será repetido para a conta c2 :

```
//Código omitido

public class Teste {
//Código omitido
}

class TitularDaContaComparator implements Comparator<Conta> {

    @Override
    public int compare(Conta c1, Conta c2) {

        String nomeC1 = c1.getTitular().getNome();
        String nomeC2 = c2.getTitular().getNome();

        return 0;
    }
}

class NumeroDaContaComparator implements Comparator<Conta> {

    @Override
    public int compare(Conta c1, Conta c2) {

        if(c1.getNumero() < c2.getNumero()) {
            return -1;
        }

        if(c1.getNumero() > c2.getNumero()) {
            return 1;
        }

        return 0;
    }
}
```

```
        }

    return 0;
}

}
```

[COPIAR CÓDIGO](#)

Agora nos resta criar o método de comparação pela ordem alfabética. A classe `String` já possui, implementado, um método de ordenação utilizando este critério.

Ao chamarmos o `nomeC1`, surgirá uma lista de métodos, dentre eles, temos o `compareTo()`, que recebe como parâmetro um outro `String`, e que resulta em um `int`. Quando os "strings" forem iguais, o retorno é `0`, se um for menor que o outro, o resultado é negativo, e se um for maior que o outro, o resultado é positivo. Desta forma, temos o seguinte método:

```
//Código omitido

public class Teste {
//Código omitido
}

class TitularDaContaComparator implements Comparator<Conta> {

    @Override
    public int compare(Conta c1, Conta c2) {

        String nomeC1 = c1.getTitular().getNome();
        String nomeC2 = c2.getTitular().getNome();

        nomeC1.compareTo(nomeC2);

        return 0;
}
```

```
}

class NumeroDaContaComparator implements Comparator<Conta> {

    @Override
    public int compare(Conta c1, Conta c2) {

        if(c1.getNumero() < c2.getNumero()) {
            return -1;
        }

        if(c1.getNumero() > c2.getNumero()) {
            return 1;
        }

        return 0;
    }
}
```

[COPIAR CÓDIGO](#)

Já que o resultado é um `Integer`, podemos inseri-lo diretamente como retorno:

```
//Código omitido

public class Teste {
//Código omitido
}

class TitularDaContaComparator implements Comparator<Conta> {

    @Override
    public int compare(Conta c1, Conta c2) {

        String nomeC1 = c1.getTitular().getNome();
        String nomeC2 = c2.getTitular().getNome();
        return nomeC1.compareTo(nomeC2);
    }
}
```

```
}

class NumeroDaContaComparator implements Comparator<Conta> {

    @Override
    public int compare(Conta c1, Conta c2) {

        if(c1.getNumero() < c2.getNumero()) {
            return -1;
        }

        if(c1.getNumero() > c2.getNumero()) {
            return 1;
        }

        return 0;
    }
}
```

[COPIAR CÓDIGO](#)

Em seguida, na classe `Teste`, criaremos um novo comparator :

```
//Código omitido

public class Teste {

    //Código omitido

    for (Conta conta : lista) {
        System.out.println(conta);
    }

    NumeroDaContaComparator comparator = new NumeroDaContaCom-
    TitularDaContaComparator titularComparator = new TitularD-
    lista.sort(comparator);

    System.out.println("-----");
}
```

```
        for (Conta conta : lista) {
            System.out.println(conta);
        }

    }

class TitularDaContaComparador implements Comparator<Conta> {

    @Override
    public int compare(Conta c1, Conta c2) {

        String nomeC1 = c1.getTitular().getNome();
        String nomeC2 = c2.getTitular().getNome();
        return nomeC1.compareTo(nomeC2);
    }
}

class NumeroDaContaComparador implements Comparator<Conta> {

    @Override
    public int compare(Conta c1, Conta c2) {

        if(c1.getNumero() < c2.getNumero()) {
            return -1;
        }

        if(c1.getNumero() > c2.getNumero()) {
            return 1;
        }

        return 0;
    }
}
```

COPIAR CÓDIGO

Por fim, passaremos o `titularComparator` como parâmetro para o método

`sort()` :

```
//Código omitido

public class Teste {

    //Código omitido

    for (Conta conta : lista) {
        System.out.println(conta);
    }

    NumeroDaContaComparador comparator = new NumeroDaContaCom-
    TitularDaContaComparador titularComparator = new TitularD-
    lista.sort(titularComparator);

    System.out.println("-----");

    for (Conta conta : lista) {
        System.out.println(conta);
    }

}

class TitularDaContaComparador implements Comparator<Conta> {

    @Override
    public int compare(Conta c1, Conta c2) {

        String nomeC1 = c1.getTitular().getNome();
        String nomeC2 = c2.getTitular().getNome();
        return nomeC1.compareTo(nomeC2);
    }
}

class NumeroDaContaComparador implements Comparator<Conta> {
```

```
@Override
public int compare(Conta c1, Conta c2) {

    if(c1.getNumero() < c2.getNumero()) {
        return -1;
    }

    if(c1.getNumero() > c2.getNumero()) {
        return 1;
    }

    return 0;
}
```

[COPIAR CÓDIGO](#)



Executaremos nossa classe, e temos o seguinte resultado no console:

```
ContaCorrente, Numero: 33, Agencia: 22
ContaPoupanca, Numero: 44, Agencia: 22
ContaCorrente, Numero: 11, Agencia: 22
ContaPoupanca, Numero: 22, Agencia: 22
-----
ContaPoupanca, Numero: 22, Agencia: 22
ContaPoupanca, Numero: 44, Agencia: 22
ContaCorrente, Numero: 33, Agencia: 22
ContaCorrente, Numero: 11, Agencia: 22
```

[COPIAR CÓDIGO](#)

Percebemos que, em relação à ordem numérica, temos uma ordenação diferente. Mas ainda não conseguimos visualizar quem é o titular de cada conta, para melhorarmos isso, no segundo laço, concatenaremos em cada impressão o nome do titular da respectiva conta:

```
//Código omitido

public class Teste {

//Código omitido

    for (Conta conta : lista) {
        System.out.println(conta);
    }

    NumeroDaContaComparador comparator = new NumeroDaContaCom-
    TitularDaContaComparador titularComparator = new TitularD-
    lista.sort(titularComparator);

    System.out.println("-----");

    for (Conta conta : lista) {
        System.out.println(conta + ", " + conta.getTitula-
    }

}

class TitularDaContaComparador implements Comparator<Conta> {

    @Override
    public int compare(Conta c1, Conta c2) {

        String nomeC1 = c1.getTitular().getNome();
        String nomeC2 = c2.getTitular().getNome();
        return nomeC1.compareTo(nomeC2);
    }
}

class NumeroDaContaComparador implements Comparator<Conta> {

    @Override
    public int compare(Conta c1, Conta c2) {
```

```
if(c1.getNumero() < c2.getNumero()) {  
    return -1;  
}  
  
if(c1.getNumero() > c2.getNumero()) {  
    return 1;  
}  
  
return 0;  
}  
}
```

[COPIAR CÓDIGO](#)



Executaremos novamente, e temos o seguinte resultado:

```
ContaCorrente, Numero: 33, Agencia: 22  
ContaPoupanca, Numero: 44, Agencia: 22  
ContaCorrente, Numero: 11, Agencia: 22  
ContaPoupanca, Numero: 22, Agencia: 22  
-----  
ContaPoupanca, Numero: 22, Agencia: 22, Ana  
ContaPoupanca, Numero: 44, Agencia: 22, Guilherme  
ContaCorrente, Numero: 33, Agencia: 22, Nico  
ContaCorrente, Numero: 11, Agencia: 22, Paulo
```

[COPIAR CÓDIGO](#)

Como podemos observar, tudo está funcionando, as contas estão em ordem alfabética.

Adiante, veremos o método antigo de fazer esta comparação. Até a próxima!

A ordem natural

Transcrição

Olá! Nesta aula, dando continuidade ao assunto de ordenação das listas, falaremos sobre a **ordem natural**.

Primeiro, melhoraremos alguns aspectos em nosso código. Primeiro, como podemos simplificar o método `compare()`, da `NumeroDaContaComparator`?

```
//Código omitido

public class Teste {

    //Código omitido

    for (Conta conta : lista) {
        System.out.println(conta);
    }

    NumeroDaContaComparator comparator = new NumeroDaContaCom-
    TitularDaContaComparator titularComparator = new TitularD-
    lista.sort(titularComparator);

    System.out.println("-----");

    for (Conta conta : lista) {
        System.out.println(conta + ", " + conta.getTitula
    }

}
```

```
class TitularDaContaComparador implements Comparator<Conta> {

    @Override
    public int compare(Conta c1, Conta c2) {

        String nomeC1 = c1.getTitular().getNome();
        String nomeC2 = c2.getTitular().getNome();
        return nomeC1.compareTo(nomeC2);
    }
}

class NumeroDaContaComparador implements Comparator<Conta> {

    @Override
    public int compare(Conta c1, Conta c2) {

        if(c1.getNumero() < c2.getNumero()) {
            return -1;
        }

        if(c1.getNumero() > c2.getNumero()) {
            return 1;
        }

        return 0;
    }
}
```

[COPIAR CÓDIGO](#)

Na regra, temos um retorno negativo caso o valor da conta `c1` seja inferior ao da `c2`, e positivo se `c1` for maior que `c2`. Caso sejam iguais, o retorno é `0`. Entretanto, se escrevermos da seguinte forma:

```
//Apenas para exemplificar, ainda não foi inserido no código
```

```
return c1.getNumero() - c2.getNumero();
```

[COPIAR CÓDIGO](#)

Chegamos ao mesmo resultado. Assim, comentaremos a escrita atual, e a substituiremos por esta nova:

```
//Código omitido

public class Teste {

    //Código omitido

    for (Conta conta : lista) {
        System.out.println(conta);
    }

    NumeroDaContaComparador comparator = new NumeroDaContaCom-
    TitularDaContaComparador titularComparator = new TitularD-
    lista.sort(titularComparator);

    System.out.println("-----");

    for (Conta conta : lista) {
        System.out.println(conta + ", " + conta.getTitula
    }

}

class TitularDaContaComparador implements Comparator<Conta> {

    @Override
    public int compare(Conta c1, Conta c2) {

        String nomeC1 = c1.getTitular().getNome();
        String nomeC2 = c2.getTitular().getNome();
        return nomeC1.compareTo(nomeC2);
    }
}
```

```
    }

}

class NumeroDaContaComparador implements Comparator<Conta> {

    @Override
    public int compare(Conta c1, Conta c2) {

        return c1.getNumero() - c2.getNumero();

        // if(c1.getNumero() < c2.getNumero()) {
        //     return -1;
        // }
        //
        // if(c1.getNumero() > c2.getNumero()) {
        //     return 1;
        // }
        //
        // return 0;
    }
}
```

[COPIAR CÓDIGO](#)

A regra permanece a mesma, apenas alteramos o modo de escrevê-la. Sempre que tivermos um critério baseado em dados numéricos, é possível simplificá-lo.

Há ainda outra forma de escrevermos esta mesma regra. Assim como a `String`, o `Integer` também tem implementado um método de comparação, assim, poderíamos escrever o seguinte - comentando também a última linha de código que inserimos:

```
//Código omitido

public class Teste {
```

```
//Código omitido
```

```
    for (Conta conta : lista) {
        System.out.println(conta);
    }

    NumeroDaContaComparador comparator = new NumeroDaContaCom-
    TitularDaContaComparador titularComparator = new TitularD-
    lista.sort(titularComparator);

    System.out.println("-----");

    for (Conta conta : lista) {
        System.out.println(conta + ", " + conta.getTitula-
    }

}
```

```
class TitularDaContaComparador implements Comparator<Conta> {
```

```
    @Override
    public int compare(Conta c1, Conta c2) {

        String nomeC1 = c1.getTitular().getNome();
        String nomeC2 = c2.getTitular().getNome();
        return nomeC1.compareTo(nomeC2);
    }
}
```

```
class NumeroDaContaComparador implements Comparator<Conta> {
```

```
    @Override
    public int compare(Conta c1, Conta c2) {

        return Integer.compare(c1.getNumero())
//                return c1.getNumero() - c2.getNumero()
    }
}
```

```
//             if(c1.getNumero() < c2.getNumero()) {
//                 return -1;
//             }
//
//             if(c1.getNumero() > c2.getNumero()) {
//                 return 1;
//             }
//
//             return 0;
        }
    }
```

[COPIAR CÓDIGO](#)



Tudo continua funcionando normalmente, pois a regra não foi alterada.

Outro ponto que alteraremos é a implementação dos comparadores na classe `Conta`, geralmente, ela não é feita desta forma.

Percebemos que a referência de cada um dos comparadores é utilizada somente para o método `sort()`, sendo assim, é comum que o construtor `new` seja inserido diretamente como parâmetro deste método:

```
//Código omitido

public class Teste {

    //Código omitido

        for (Conta conta : lista) {
            System.out.println(conta);
        }
}
```

```
    NumeroDaContaComparador comparator = new NumeroDaContaCom-
    TitularDaContaComparador titularComparator = new TitularDaContaComparador();
    lista.sort(new TitularDaContaComparador());
```

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

    for (Conta conta : lista) {
        System.out.println(conta + ", " + conta.getTitular());
    }

}

class TitularDaContaComparator implements Comparator<Conta> {

    @Override
    public int compare(Conta c1, Conta c2) {

        String nomeC1 = c1.getTitular().getNome();
        String nomeC2 = c2.getTitular().getNome();
        return nomeC1.compareTo(nomeC2);
    }
}

class NumeroDaContaComparator implements Comparator<Conta> {

    @Override
    public int compare(Conta c1, Conta c2) {

        return Integer.compare(c1.getNumero()

//                return c1.getNumero() - c2.getNumero();

//
//                if(c1.getNumero() < c2.getNumero()) {
//                    return -1;
//                }
//
//                if(c1.getNumero() > c2.getNumero()) {
//                    return 1;
//                }
//
//                return 0;
    }
}
```

```
}
```

[COPIAR CÓDIGO](#)

Isso acaba com a necessidade da criação da referência, e a linha pode ser apagada. Quanto à linha do primeiro `comparator`, a manteremos em comentários:

```
//Código omitido

public class Teste {

    //Código omitido

        for (Conta conta : lista) {
            System.out.println(conta);
        }

    //    NumeroDaContaComparator comparator = new NumeroDaContaC
    lista.sort(new TitularDaContaComparator());

    System.out.println("-----");

        for (Conta conta : lista) {
            System.out.println(conta + ", " + conta.getTitula
        }

    }

class TitularDaContaComparator implements Comparator<Conta> {

    @Override
    public int compare(Conta c1, Conta c2) {

        String nomeC1 = c1.getTitular().getNome();
        String nomeC2 = c2.getTitular().getNome();
        return nomeC1.compareTo(nomeC2);
    }
}
```

```
    }

}

class NumeroDaContaComparador implements Comparator<Conta> {

    @Override
    public int compare(Conta c1, Conta c2) {

        return Integer.compare(c1.getNumero())

    //        return c1.getNumero() - c2.getNumero();

    //
    //        if(c1.getNumero() < c2.getNumero()) {
    //            return -1;
    //        }
    //
    //        if(c1.getNumero() > c2.getNumero()) {
    //            return 1;
    //        }
    //
    //        return 0;
    }

}
```

[COPIAR CÓDIGO](#)

Como havia mencionado, o método `sort()` foi incluído apenas no Java 8, ou seja, antes, a ordenação era possível graças à classe `Collections`. Ao digitarmos seu nome em nosso código - logo abaixo do método `sort()`, temos a opção de importá-la, é o que faremos.

Trata-se de uma classe "fachada", que possui uma série de métodos estáticos auxiliares, e que portanto não são orientados a objetos. Por exemplo, ela inclui o método `sort()`. Aqui, o utilizaremos na sua versão que exige como parâmetro um `list` e um `comparator`:

```
//Código omitido

public class Teste {

//Código omitido

    for (Conta conta : lista) {
        System.out.println(conta);
    }

//    NumeroDaContaComparator comparator = new NumeroDaContaComparador();
//    lista.sort(new TitularDaContaComparador());

        Collections.sort(list, c);

    System.out.println("-----");

    for (Conta conta : lista) {
        System.out.println(conta + ", " + conta.getTitular());
    }

}

class TitularDaContaComparador implements Comparator<Conta> {

    @Override
    public int compare(Conta c1, Conta c2) {

        String nomeC1 = c1.getTitular().getNome();
        String nomeC2 = c2.getTitular().getNome();
        return nomeC1.compareTo(nomeC2);
    }
}

class NumeroDaContaComparador implements Comparator<Conta> {

    @Override
```

```
public int compare(Conta c1, Conta c2) {  
  
    return Integer.compare(c1.getNumero()  
  
        //                                         return c1.getNumero() - c2.getNumero();  
        //  
        //                                         if(c1.getNumero() < c2.getNumero()) {  
        //                                         return -1;  
        //                                         }  
        //                                         if(c1.getNumero() > c2.getNumero()) {  
        //                                         return 1;  
        //                                         }  
        //                                         return 0;  
    }  
}
```

[COPIAR CÓDIGO](#)

Em seguida, preencheremos os parâmetros de acordo com a nossa realidade, ou seja, passando a nossa `lista`, e o nosso `comparator`, que é o `new NumeroDaContaComparator()`:

```
//Código omitido  
  
public class Teste {  
  
    //Código omitido  
  
    for (Conta conta : lista) {  
        System.out.println(conta);  
    }  
  
    // NumeroDaContaComparator comparator = new NumeroDaContaC  
    lista.sort(new TitularDaContaComparador());
```

```
Collections.sort(lista, new NumeroDaContaComparat

    System.out.println("-----");

    for (Conta conta : lista) {
        System.out.println(conta + ", " + conta.getTitula
    }

}

class TitularDaContaComparator implements Comparator<Conta> {

    @Override
    public int compare(Conta c1, Conta c2) {

        String nomeC1 = c1.getTitular().getNome();
        String nomeC2 = c2.getTitular().getNome();
        return nomeC1.compareTo(nomeC2);
    }
}

class NumeroDaContaComparator implements Comparator<Conta> {

    @Override
    public int compare(Conta c1, Conta c2) {

        return Integer.compare(c1.getNumero()

//                return c1.getNumero() - c2.getNumero();
//
//                if(c1.getNumero() < c2.getNumero()) {
//                    return -1;
//                }
//                if(c1.getNumero() > c2.getNumero()) {
//                    return 1;
//                }
//        }
    }
}
```

```
//           return 0;  
    }  
}
```

[COPIAR CÓDIGO](#)

Executaremos a classe, para testarmos se agora é possível visualizarmos a lista ordenada de acordo com a numeração das contas. Temos o seguinte resultado no console:

```
ContaCorrente, Numero: 33, Agencia: 22  
ContaPoupanca, Numero: 44, Agencia: 22  
ContaCorrente, Numero: 11, Agencia: 22  
ContaPoupanca, Numero: 22, Agencia: 22  
-----  
ContaPoupanca, Numero: 11, Agencia: 22, Paulo  
ContaPoupanca, Numero: 22, Agencia: 22, Ana  
ContaCorrente, Numero: 33, Agencia: 22, Nico  
ContaCorrente, Numero: 44, Agencia: 22, Guilherme
```

[COPIAR CÓDIGO](#)

A classe `Collections` possui muitos métodos, vale a pena explorá-los posteriormente. Por exemplo, temos um chamado `reverse()`, que inverte a ordem da lista:

```
//Código omitido  
  
public class Teste {  
  
    //Código omitido  
  
        for (Conta conta : lista) {  
            System.out.println(conta);  
        }
```

```
//           NumeroDaContaComparator comparator = new NumeroDaContaC
lista.sort(new TitularDaContaComparator()));

Collections.sort(lista, new NumeroDaContaComparat
Collections.reverse(lista);

System.out.println("-----");

for (Conta conta : lista) {
    System.out.println(conta + ", " + conta.getTitula
}

}

class TitularDaContaComparator implements Comparator<Conta> {

    @Override
    public int compare(Conta c1, Conta c2) {

        String nomeC1 = c1.getTitular().getNome();
        String nomeC2 = c2.getTitular().getNome();
        return nomeC1.compareTo(nomeC2);
    }
}

class NumeroDaContaComparator implements Comparator<Conta> {

    @Override
    public int compare(Conta c1, Conta c2) {

        return Integer.compare(c1.getNumero()

//                return c1.getNumero() - c2.getNumero();
//
//                if(c1.getNumero() < c2.getNumero()) {
//                    return -1;
//                }
//            }
}
```

```
//  
//          if(c1.getNumero() > c2.getNumero()) {  
//              return 1;  
//          }  
//  
//          return 0;  
}  
}
```

[COPIAR CÓDIGO](#)



Executando a classe, temos o seguinte resultado:

```
ContaCorrente, Numero: 33, Agencia: 22  
ContaPoupanca, Numero: 44, Agencia: 22  
ContaCorrente, Numero: 11, Agencia: 22  
ContaPoupanca, Numero: 22, Agencia: 22  
-----  
ContaCorrente, Numero: 44, Agencia: 22, Guilherme  
ContaCorrente, Numero: 33, Agencia: 22, Nico  
ContaPoupanca, Numero: 22, Agencia: 22, Ana  
ContaPoupanca, Numero: 11, Agencia: 22, Paulo
```

[COPIAR CÓDIGO](#)

Ou seja, temos a `lista`, em ordem de maior para menor número de conta.

Como havíamos notado, há dois tipos de métodos `sort()` em `Collections`. O que já vimos, e que recebe um `comparator`, e um segundo, que **não recebe** um:

```
//Código omitido  
  
public class Teste {  
  
    //Código omitido
```

```
    for (Conta conta : lista) {
        System.out.println(conta);
    }

//    NumeroDaContaComparator comparator = new NumeroDaContaC
lista.sort(new TitularDaContaComparator()));

//    Collections.sort(lista, new NumeroDaContaCompar
Collections.sort(list);
Collections.reverse(lista);

System.out.println("-----");

for (Conta conta : lista) {
    System.out.println(conta + ", " + conta.getTitula
}

}

class TitularDaContaComparator implements Comparator<Conta> {

    @Override
    public int compare(Conta c1, Conta c2) {

        String nomeC1 = c1.getTitular().getNome();
        String nomeC2 = c2.getTitular().getNome();
        return nomeC1.compareTo(nomeC2);
    }
}

class NumeroDaContaComparator implements Comparator<Conta> {

    @Override
    public int compare(Conta c1, Conta c2) {

        return Integer.compare(c1.getNumero())
//        return c1.getNumero() - c2.getNumero(),
    }
}
```

```
//  
//          if(c1.getNumero() < c2.getNumero()) {  
//              return -1;  
//          }  
//  
//          if(c1.getNumero() > c2.getNumero()) {  
//              return 1;  
//          }  
//  
//      return 0;  
}  
}  
  

```

Entretanto, ao incluirmos este método em nosso código, vemos que ele não compila. Se ele não possui um comparador, como pode ordenar os elementos? Ao nos depararmos com situações deste tipo, estamos lidando com o conceito de **ordem natural**.

Qualquer objeto que adicionamos à lista pode ter uma ordem natural. Ela é o critério de comparação que prevalece na hipótese de não definirmos um `comparator`.

Por exemplo, em um time de futebol, a ordem natural poderia ser considerada como a numeração na camisa dos jogadores.

Em nosso caso, para as contas, se não for definido um critério, qual deve ser nossa ordem natural? Há diversas possibilidades, mas neste caso, utilizaremos o saldo.

Para que isto funcione, é necessário alterarmos a classe `Conta`, já que é ela quem deve definir este critério.

Na classe `Conta`, implementaremos a interface utilizada para determinar a ordem natural, que se chama `Comparable`, do pacote `java.lang`:

```
//Código omitido

public abstract class Conta extends Object implements Comparable<

    protected double saldo;
    private int agencia;
    private int numero;
    private Cliente titular;
    private static int total = 0;

//Código omitido

}
```

[COPIAR CÓDIGO](#)

Assim, nossa conta se torna **comparável**. Como vimos, há também a presença de *generics*, uma vez que queremos comparar uma `Conta` a outra `Conta`:

```
//Código omitido

public abstract class Conta extends Object implements Comparable<

    protected double saldo;
    private int agencia;
    private int numero;
    private Cliente titular;
    private static int total = 0;

//Código omitido

}
```

[COPIAR CÓDIGO](#)

Para definirmos a ordem natural, de fato, precisamos implementar o método `compareTo()`. Utilizando a implementação automática do Eclipse, temos o seguinte:

```
//Código omitido

public abstract class Conta extends Object implements Comparable<

    protected double saldo;
    private int agencia;
    private int numero;
    private Cliente titular;
    private static int total = 0;

//Código omitido

@Override
public boolean equals(Object ref) {

    Conta outra = (Conta) ref;

    if(this.agencia != outra.agencia) {
        return false;
    }

    if(this.numero != outra.numero) {
        return false;
    }

    return true;
}

@Override
public int compareTo(Conta arg0) {
    //TODO Auto-generated method stub
    return 0;
}
```

```
@Override  
public String toString() {  
    return "Número: " + this.numero + ", Agen  
}  
}
```

[COPIAR CÓDIGO](#)

Ele é similar ao método `compare()`, já que seu retorno é um inteiro, e nos devolve `0`, caso as contas sejam iguais. Também terá um número positivo, se a primeira conta for maior, e negativo, se ela for menor que a segunda.

Em seguida, adaptaremos o método, para que realize a comparação com base no saldo. Utilizaremos a classe *wrapper* `Double`, e teremos como parâmetro os dois valores dos saldos, `this.saldo` e `outra.saldo`, representando o saldo **desta** conta, e da **outra** conta:

```
//Código omitido  
  
public abstract class Conta extends Object implements Comparable<  
  
    protected double saldo;  
    private int agencia;  
    private int numero;  
    private Cliente titular;  
    private static int total = 0;  
  
//Código omitido  
  
@Override  
public boolean equals(Object ref) {  
    Conta outra = (Conta) ref;
```

```
        if(this.agencia != outra.agencia) {
            return false;
        }

        if(this.numero != outra.numero) {
            return false;
        }

    return true;
}

@Override
public int compareTo(Conta outra) {
    return Double.compare(this.saldo, outra.saldo
}

@Override
public String toString() {
    return "Número: " + this.numero + ", Agen
}

}
```

[COPIAR CÓDIGO](#)

Não há necessidade de utilizarmos o método `getSaldo()` pois estamos trabalhando dentro da própria classe, assim, podemos utilizar `saldo` diretamente.

Agora que definimos a ordem natural, ao retornarmos à classe `Teste`, notamos que ela voltou a compilar. Podemos chamar o método `sort()` sem que haja um `comparator` definido, pois ele chamará internamente o método presente na classe `Conta`, que acabamos de criar.

Para testarmos, comentaremos o método `reverse()`:

```
//Código omitido

public class Teste {

//Código omitido

    for (Conta conta : lista) {
        System.out.println(conta);
    }

//    NumeroDaContaComparator comparator = new NumeroDaContaComparador();
//    lista.sort(new TitularDaContaComparator());
//
//    Collections.sort(lista, new NumeroDaContaComparador());
//    Collections.sort(list);
//    Collections.reverse(lista);

        System.out.println("-----");

    for (Conta conta : lista) {
        System.out.println(conta + ", " + conta.getTitular());
    }

}

class TitularDaContaComparator implements Comparator<Conta> {

    @Override
    public int compare(Conta c1, Conta c2) {

        String nomeC1 = c1.getTitular().getNome();
        String nomeC2 = c2.getTitular().getNome();
        return nomeC1.compareTo(nomeC2);
    }
}

class NumeroDaContaComparator implements Comparator<Conta> {
```

```
@Override
public int compare(Conta c1, Conta c2) {

    return Integer.compare(c1.getNumero()

//                return c1.getNumero() - c2.getNumero();

//
//                if(c1.getNumero() < c2.getNumero()) {
//                    return -1;
//                }
//
//                if(c1.getNumero() > c2.getNumero()) {
//                    return 1;
//                }
//
//                return 0;
}
}
```

[COPIAR CÓDIGO](#)



Executando, temos o seguinte resultado:

```
ContaCorrente, Numero: 33, Agencia: 22
ContaPoupanca, Numero: 44, Agencia: 22
ContaCorrente, Numero: 11, Agencia: 22
ContaPoupanca, Numero: 22, Agencia: 22
-----
ContaPoupanca, Numero: 11, Agencia: 22, Paulo
ContaPoupanca, Numero: 22, Agencia: 22, Ana
ContaCorrente, Numero: 33, Agencia: 22, Nico
ContaCorrente, Numero: 44, Agencia: 22, Guilherme
```

[COPIAR CÓDIGO](#)

As contas estão ordenadas, mas não conseguimos visualizar os valores dos saldos.

Para solucionarmos isso, retornaremos à classe `Conta`, onde incluiremos no método `toString()` a referência ao saldo:

//Código omitido

```
public abstract class Conta extends Object implements Comparable<
```

```
    protected double saldo;
    private int agencia;
    private int numero;
    private Cliente titular;
    private static int total = 0;
```

//Código omitido

```
@Override
public boolean equals(Object ref) {

    Conta outra = (Conta) ref;

    if(this.agencia != outra.agencia) {
        return false;
    }

    if(this.numero != outra.numero) {
        return false;
    }

    return true;
}

@Override
public int compareTo(Conta outra) {
    return Double.compare(this.saldo, outra.s;
```

```
@Override  
public String toString() {  
    return "Numero: " + this.numero + ", Agen  
}  
}
```

[COPIAR CÓDIGO](#)

Mais uma vez, executando a classe `Teste`, temos o seguinte resultado no console:

```
ContaCorrente, Numero: 33, Agencia: 22, Saldo: 333.0  
ContaPoupanca, Numero: 44, Agencia: 22, Saldo: 444.0  
ContaCorrente, Numero: 11, Agencia: 22, Saldo: 111.0  
ContaPoupanca, Numero: 22, Agencia: 22, Saldo: 222.0  
-----  
ContaPoupanca, Numero: 11, Agencia: 22, Saldo: 111.0, Paulo  
ContaPoupanca, Numero: 22, Agencia: 22, Saldo: 222.0, Ana,  
ContaCorrente, Numero: 33, Agencia: 22, Saldo: 333.0, Nico  
ContaCorrente, Numero: 44, Agencia: 22, Saldo: 444.0, Guilherme
```

[COPIAR CÓDIGO](#)

Tudo certo!

Primeiro, temos a ordem de inserção, e em seguida, a ordem de saldo, do menor para o maior.

A seguir, como podemos utilizar a ordem natural, aplicando diretamente na lista ? Se fizermos o seguinte:

```
//Código omitido  
  
public class Teste {
```

```
//Código omitido

    for (Conta conta : lista) {
        System.out.println(conta);
    }

//    NumeroDaContaComparator comparator = new NumeroDaContaC
lista.sort();

//    Collections.sort(lista, new NumeroDaContaCompar
//    Collections.sort(list);
//    Collections.reverse(lista);

System.out.println("-----");

    for (Conta conta : lista) {
        System.out.println(conta + ", " + conta.getTitula
    }

}

class TitularDaContaComparator implements Comparator<Conta> {

    @Override
    public int compare(Conta c1, Conta c2) {

        String nomeC1 = c1.getTitular().getNome();
        String nomeC2 = c2.getTitular().getNome();
        return nomeC1.compareTo(nomeC2);
    }
}

class NumeroDaContaComparator implements Comparator<Conta> {

    @Override
    public int compare(Conta c1, Conta c2) {
```

```

        return Integer.compare(c1.getNumero())

    //           return c1.getNumero() - c2.getNumero();

    //
    //           if(c1.getNumero() < c2.getNumero()) {
    //               return -1;
    //           }
    //
    //           if(c1.getNumero() > c2.getNumero()) {
    //               return 1;
    //           }
    //
    //           return 0;
    }

}

```

[COPIAR CÓDIGO](#)



O código não compila. O método `sort()` da `lista`, da instância, não é sobrecarregado. Aqui, existe apenas um, que é aquele que exige um `comparator`. Para que funcione, temos que passar o parâmetro `null` - mas isso é uma má prática de programação:

```

//Código omitido

public class Teste {

//Código omitido

    for (Conta conta : lista) {
        System.out.println(conta);
    }

//    NumeroDaContaComparator comparator = new NumeroDaContaC
//    lista.sort(null);

```

```
// Collections.sort(lista, new NumeroDaContaComparador());
// Collections.sort(list);
// Collections.reverse(lista);

System.out.println("-----");

for (Conta conta : lista) {
    System.out.println(conta + ", " + conta.getTitular());
}

}
```

```
class TitularDaContaComparator implements Comparator<Conta> {
```

```
@Override
public int compare(Conta c1, Conta c2) {

    String nomeC1 = c1.getTitular().getNome();
    String nomeC2 = c2.getTitular().getNome();
    return nomeC1.compareTo(nomeC2);
}
```

```
}
```

```
class NumeroDaContaComparator implements Comparator<Conta> {
```

```
@Override
public int compare(Conta c1, Conta c2) {

    return Integer.compare(c1.getNumero()

//        return c1.getNumero() - c2.getNumero();

//        if(c1.getNumero() < c2.getNumero()) {
//            return -1;
//        }
//        if(c1.getNumero() > c2.getNumero()) {
//            return 1;
//        }
}
```

```
//          }
//
//          return 0;
}
}
```

[COPIAR CÓDIGO](#)

Ao fazermos isso, passamos um `comparator` nulo, o que obriga o Java a utilizar a ordem natural. Entretanto, repetindo, o parâmetro `null` não é uma boa prática de programação.

Ao executarmos, vemos que o resultado no console permanece o mesmo, continua funcionando perfeitamente.

Temos bastante para praticar, até a próxima!

Ordenar Arrays

Ordenar arrays também não é difícil basta usar o método `sort` da classe `Arrays`. A classe `Arrays` é parecida com `Collections` no sentido que também junta vários métodos utilitários:

```
import java.util.Arrays;

public class TesteSortArrays
{
    public static void main(String[] args)
    {
        int[] numeros = new int[]{43, 15, 64, 22, 89};

        Arrays.sort(numeros); //método utilitário sort

        System.out.println(Arrays.toString(numeros)); //método

        //Saída : [15, 22, 43, 64, 89]
    }
}
```

[COPIAR CÓDIGO](#)



Classes anônimas

Transcrição

Você pode fazer o [DOWNLOAD \(<https://caelum-online-public.s3.amazonaws.com/850-java-util/08/java6-cap8.zip>\)](https://caelum-online-public.s3.amazonaws.com/850-java-util/08/java6-cap8.zip) do projeto da aula anterior.

Olá! Anteriormente, falamos sobre comparação e ordenação das coleções, definindo o `comparator` e a ordem natural.

Neste vídeo, continuaremos a falar sobre as classes de `comparators` que havíamos criado nas aulas anteriores, mas primeiro faremos alguns ajustes em nosso código.

Renomearemos a classe `Teste`, que passará a se chamar `TesteOrdenacao`. Criaremos uma cópia desta classe, à qual daremos o nome de `Teste`.

Ao abrirmos a nova classe `Teste`, percebemos que ela não compila. Isso acontece pois as classes `TitularDaContaComparator` e `NumeroDaContaComparator` já estão definidas na classe `TesteOrdenacao`, que já foi compilada pelo Eclipse. Para permitirmos a compilação, adicionaremos um `2` ao final do nome de cada uma destas classes.

Além disso, apagaremos o conteúdo que havíamos mantido em comentários, dentro da classe `NumeroDaContaComparator2`:

```
//Código omitido

public class Teste {

//Código omitido

    for (Conta conta : lista) {
        System.out.println(conta);
    }

//    NumeroDaContaComparator comparator = new NumeroDaContaComparador();
//    lista.sort(null);

//    Collections.sort(lista, new NumeroDaContaComparador());
//    Collections.sort(list);
//    Collections.reverse(lista);

    System.out.println("-----");

    for (Conta conta : lista) {
        System.out.println(conta + ", " + conta.getTitular());
    }
}

class TitularDaContaComparator2 implements Comparator<Conta> {

    @Override
    public int compare(Conta c1, Conta c2) {

        String nomeC1 = c1.getTitular().getNome();
        String nomeC2 = c2.getTitular().getNome();
        return nomeC1.compareTo(nomeC2);
    }
}

class NumeroDaContaComparator2 implements Comparator<Conta> {
```

```
@Override  
public int compare(Conta c1, Conta c2) {  
  
    return Integer.compare(c1.getNumero(), c2.getNumero());  
}
```

[COPIAR CÓDIGO](#)



Apagaremos o primeiro laço, que havíamos criado antes da ordenação, e removeremos os comentários da linha onde foi criado o `comparator`, passando-o como parâmetro para o método `sort()`. Alteraremos o nome da classe, para corresponder ao novo `NumeroDaContaComparator2`.

Apagaremos os comentários relacionados às `Collections`, e a linha divisória que havíamos criado:

```
//Código omitido  
  
public class Teste {  
  
    //Código omitido  
  
    NumeroDaContaComparator2 comparator = new NumeroDaContaCo  
    lista.sort(comparator);  
  
    for (Conta conta : lista) {  
        System.out.println(conta + ", " + conta.getTitula  
    }  
  
}  
  
class TitularDaContaComparator2 implements Comparator<Conta> {  
  
    @Override
```

```
public int compare(Conta c1, Conta c2) {  
  
    String nomeC1 = c1.getTitular().getNome();  
    String nomeC2 = c2.getTitular().getNome();  
    return nomeC1.compareTo(nomeC2);  
}  
  
}  
  
class NumeroDaContaComparator2 implements Comparator<Conta> {  
  
    @Override  
    public int compare(Conta c1, Conta c2) {  
  
        return Integer.compare(c1.getNumero(), c2.getNumer  
    }  
}
```

COPIAR CÓDIGO



Por fim, alteraremos a ordem das classes, de modo que `NumeroDaContaComparator2` apareça antes de `TitularDaContaComparator2`:

```
//Código omitido  
  
public class Teste {  
  
    //Código omitido  
  
    NumeroDaContaComparator2 comparator = new NumeroDaContaCo  
    lista.sort(comparator);  
  
    for (Conta conta : lista) {  
        System.out.println(conta + ", " + conta.getTitula  
    }  
}  
  
class NumeroDaContaComparator2 implements Comparator<Conta> {
```

```
@Override
public int compare(Conta c1, Conta c2) {

    return Integer.compare(c1.getNumero(), c2.getNumero());
}

class TitularDaContaComparator2 implements Comparator<Conta> {

    @Override
    public int compare(Conta c1, Conta c2) {

        String nomeC1 = c1.getTitular().getNome();
        String nomeC2 = c2.getTitular().getNome();
        return nomeC1.compareTo(nomeC2);
    }
}
```

[COPIAR CÓDIGO](#)

O objetivo de uma classe é agrupar os elementos que devem permanecer juntos. São eles os estados, ou atributos, e funções, ou métodos. Entretanto, percebemos que nas duas últimas classes em nosso código não há nenhum atributo.

Ainda, como trabalhamos apenas com um método em cada classe, sequer utilizamos o `this`, o que indica um desinteresse no objeto. A criação da classe foi necessária pois é uma exigência do Java, mas não seria o meio mais indicado para encapsular um método.

Estes objetos, que são criados com o único objetivo de encapsular uma função, são chamados de **Function Objects**.

Em nenhum momento utilizamos a classe `NumeroDaContaComparator2` fora deste código, diferente da classe `Conta`, que utilizamos em diversos lugares.

Estas são motivações muito fracos para se ter uma classe. Por isso, existem atalhos que nos permitem simplificar nosso código.

O primeiro que veremos é a **classe anônima**, modalidade que é compatível até com versões do Java anteriores ao Java 8. Como já vimos, podemos inserir o construtor do objeto diretamente como um parâmetro no método `sort()`. Nossa ideia agora é eliminar a necessidade desta classe `NumeroDaContaComparator2` por completo.

Primeiro, inseriremos o construtor diretamente como um parâmetro:

```
//Código omitido

public class Teste {

    //Código omitido

    lista.sort(new NumeroDaContaComparator2());

    for (Conta conta : lista) {
        System.out.println(conta + ", " + conta.getTitula
    }
}

class NumeroDaContaComparator2 implements Comparator<Conta> {

    @Override
    public int compare(Conta c1, Conta c2) {

        return Integer.compare(c1.getNumero(), c2.getNumer
    }
}

class TitularDaContaComparator2 implements Comparator<Conta> {

    @Override
    public int compare(Conta c1, Conta c2) {
```

```
        String nomeC1 = c1.getTitular().getNome();
        String nomeC2 = c2.getTitular().getNome();
        return nomeC1.compareTo(nomeC2);
    }
}
```

[COPIAR CÓDIGO](#)

Copiaremos todo o conteúdo da classe `NumerоНumeroDaContaComparador2` a partir de `Comparator`, até o final, e colaremos após `new`, apagando `NumerоНumeroDaContaComparador2()` que havíamos inserido. O resultado é o seguinte:

```
//Código omitido

public class Teste {

//Código omitido

    lista.sort(new Comparator<Conta> {

        @Override
        public int compare(Conta c1, Conta c2) {

            return Integer.compare(c1.getNumero(), c2.getNumero())
        }
    );
}

    for (Conta conta : lista) {
        System.out.println(conta + ", " + conta.getTitula
    }
}

class NumerоНumeroDaContaComparador2 implements Comparator<Conta> {

    @Override
```

```
public int compare(Conta c1, Conta c2) {  
  
    return Integer.compare(c1.getNumero(), c2.getNumero());  
}  
  
}  
  
class TitularDaContaComparator2 implements Comparator<Conta> {  
  
    @Override  
    public int compare(Conta c1, Conta c2) {  
  
        String nomeC1 = c1.getTitular().getNome();  
        String nomeC2 = c2.getTitular().getNome();  
        return nomeC1.compareTo(nomeC2);  
    }  
}
```

[COPIAR CÓDIGO](#)

O código ainda não compila, precisamos inserir os parênteses após os *generics* de `<Conta>` :

```
//Código omitido  
  
public class Teste {  
  
    //Código omitido  
  
    lista.sort(new Comparator<Conta>() {  
  
        @Override  
        public int compare(Conta c1, Conta c2) {  
  
            return Integer.compare(c1.getNumero(), c2.getNumero());  
        }  
    });
```

```
        for (Conta conta : lista) {
            System.out.println(conta + ", " + conta.getTitular());
        }
    }

class NumeroDaContaComparator2 implements Comparator<Conta> {

    @Override
    public int compare(Conta c1, Conta c2) {

        return Integer.compare(c1.getNumero(), c2.getNumero());
    }
}

class TitularDaContaComparator2 implements Comparator<Conta> {

    @Override
    public int compare(Conta c1, Conta c2) {

        String nomeC1 = c1.getTitular().getNome();
        String nomeC2 = c2.getTitular().getNome();
        return nomeC1.compareTo(nomeC2);
    }
}
```

[COPIAR CÓDIGO](#)

Chamamos o construtor, sem argumentos, e inserimos toda a implementação diretamente. Estamos criando um objeto, que é compatível com o `NumeroDaContaComparator2`.

Internamente, o Java gera uma classe, que por sua vez implementa o método `compare()`.

Acessaremos o *Navigator*. Isso pode ser feito por meio do *Quick Access*, no canto superior direito da janela do Eclipse. Com ele aberto, abriremos a pasta

"bytebank-herdado-conta > bin > br > com > bytebank > banco > test > util", que é justamente o nosso pacote.

Nela, há uma classe chamada `Teste`, e, logo abaixo, outra de nome `Teste$1`. Esta segunda classe foi gerada automaticamente, justamente por termos implementado o método da forma como fizemos acima, o que fizemos foi criar uma **classe anônima**. Ao implementarmos a interface `Comparator`, estamos criando a classe anônima.

Sendo assim, podemos apagar a classe que havíamos criado abaixo, já que ela não é mais necessária:

```
//Código omitido

public class Teste {

    //Código omitido

    lista.sort(new Comparator<Conta>() {

        @Override
        public int compare(Conta c1, Conta c2) {

            return Integer.compare(c1.getNumero(), c2.getNumero())
        }
    );
}

for (Conta conta : lista) {
    System.out.println(conta + ", " + conta.getTitula
}
}

class TitularDaContaComparator2 implements Comparator<Conta> {

    @Override
```

```
public int compare(Conta c1, Conta c2) {  
  
    String nomeC1 = c1.getTitular().getNome();  
    String nomeC2 = c2.getTitular().getNome();  
    return nomeC1.compareTo(nomeC2);  
}  
}
```

[COPIAR CÓDIGO](#)

Recapitulando, criamos uma classe compatível com a interface `Comparator`, pois a classe anônima a implementa. Em seguida, copiamos toda a definição da classe, a partir do nome da interface, e colamos diretamente no método, utilizando o construtor `new`. Para chamarmos o construtor da classe anônima, precisamos utilizar os parênteses `()`, após os *generics*.

É o que faremos com a classe `TitularDaContaComparator2`:

```
//Código omitido  
  
public class Teste {  
  
    //Código omitido  
  
    lista.sort(new Comparator<Conta>() {  
  
        @Override  
        public int compare(Conta c1, Conta c2) {  
  
            return Integer.compare(c1.getNumero(), c2.getNumero())  
        }  
    );  
  
    Comparator<Conta> comp = new Comparator<Conta> ``
```

```

        @Override
        public int compare(Conta c1, Conta c2) {
            String nomeC1 = c1.getTitular().getNome();
            String nomeC2 = c2.getTitular().getNome();
            return nomeC1.compareTo(nomeC2);
        }

    };

    for (Conta conta : lista) {
        System.out.println(conta + ", " + conta.getTitular());
    }
}

class TitularDaContaComparator2 implements Comparator<Conta> {

    @Override
    public int compare(Conta c1, Conta c2) {

        String nomeC1 = c1.getTitular().getNome();
        String nomeC2 = c2.getTitular().getNome();
        return nomeC1.compareTo(nomeC2);
    }
}

```

[COPIAR CÓDIGO](#)

Fizemos o mesmo processo, mas neste segundo criamos uma variável `comp`, algo que não havíamos feito anteriormente. Com isso, podemos apagar a classe `TitularDaContaComparator2`:

```

//Código omitido

public class Teste {

    //Código omitido

    lista.sort(new Comparator<Conta>() { //classe anônima

```

```

@Override
public int compare(Conta c1, Conta c2) {

    return Integer.compare(c1.getNumero(), c2.getNumero())
}

);

Comparator<Conta> comp = new Comparator<Conta>()

@Override
public int compare(Conta c1, Conta c2) {
    String nomeC1 = c1.getTitular().getNome();
    String nomeC2 = c2.getTitular().getNome();
    return nomeC1.compareTo(nomeC2);
}

;

for (Conta conta : lista) {
    System.out.println(conta + ", " + conta.getTitular());
}

```

[COPIAR CÓDIGO](#)

Assim, definimos duas classes anônimas. Ao abrirmos novamente o diretório, no *navigator*, como fizemos anteriormente, veremos que na pasta `util` temos duas classes novas, a `Teste$1` e `Teste$2`, correspondendo a cada uma das classes anônimas em nosso código, na ordem de sua criação.

Este tipo de classe nos exime da obrigação de criar classes isoladas, mas ao mesmo tempo gera maior dificuldade de leitura do nosso código. Neste ponto que entram os **lambdas**, eles servem para unir ainda mais o código.

Veremos mais sobre eles adiante, até a próxima!

Finalmente Lambdas

Transcrição

Nesta aula, continuaremos a trabalhar com a classe `Teste`. Se preferir, pode fazer uma cópia e guardá-la como `TesteClasseAnonima`.

Nosso objetivo será transformá-la para o uso de **lambdas**.

Os métodos atuais, mesmo sendo mais compactos, possuem uma sintaxe difícil. Ao analisarmos, percebemos que os métodos possuem dois parâmetros, no segundo caso há uma implementação, e por fim, ambos nos retornam algum valor.

Para começar a melhorar nosso código, apagaremos, no método `sort()`, a partir do construtor `new`, até o nome do método `compare`:

```
//Código omitido

public class Teste {

    //Código omitido

    lista.sort((Conta c1, Conta c2) {
        return Integer.compare(c1.getNumero(), c2
    }
);

    Comparator<Conta> comp = new Comparator<Conta>()
```

```

@Override
public int compare(Conta c1, Conta c2) {
    String nomeC1 = c1.getTitular().getNome();
    String nomeC2 = c2.getTitular().getNome();
    return nomeC1.compareTo(nomeC2);
}

for (Conta conta : lista) {
    System.out.println(conta + ", " + conta.getTitular());
}
}

```

[COPIAR CÓDIGO](#)



Salvaremos, e notamos que a classe não compila. Precisamos declarar a existência das lambdas expressamente, isso é feito por meio do símbolo de seta (->), formado por uma hífen e um sinal de "maior que":

```

//Código omitido

public class Teste {

    //Código omitido

    lista.sort((Conta c1, Conta c2) -> {
        return Integer.compare(c1.getNumero(), c2.getNumero());
    });

    Comparator<Conta> comp = new Comparator<Conta>()

    @Override
    public int compare(Conta c1, Conta c2) {
        String nomeC1 = c1.getTitular().getNome();
        String nomeC2 = c2.getTitular().getNome();
        return nomeC1.compareTo(nomeC2);
    }
}

```

```
        }

    };

    for (Conta conta : lista) {
        System.out.println(conta + ", " + conta.getTitula
    }

}
```

[COPIAR CÓDIGO](#)

Internamente, o Java gera uma classe que implementa a interface `Comparator`, e que contém o método `compare()`. Faremos o mesmo abaixo, onde temos a variável `comp`.

Apagaremos todo o conteúdo, exceto pelos parâmetros:

```
//Código omitido

public class Teste {

//Código omitido

    lista.sort((Conta c1, Conta c2) -> {
        return Integer.compare(c1.getNumero(), c2
    }
);

    Comparator<Conta> comp = (Conta c1, Conta c2) {
        String nomeC1 = c1.getTitular().getNo
        String nomeC2 = c2.getTitular().getNo
        return nomeC1.compareTo(nomeC2);
    }

    for (Conta conta : lista) {
        System.out.println(conta + ", " + conta.getTitula
```

```
    }  
}
```

[COPIAR CÓDIGO](#)

Temos que indicar o lambda (->), após os parâmetros:

```
//Código omitido  
  
public class Teste {  
  
    //Código omitido  
  
    lista.sort((Conta c1, Conta c2) -> {  
        return Integer.compare(c1.getNumero(), c2.getNumero());  
    };  
  
    Comparator<Conta> comp = (Conta c1, Conta c2) -> {  
        String nomeC1 = c1.getTitular().getNome();  
        String nomeC2 = c2.getTitular().getNome();  
        return nomeC1.compareTo(nomeC2);  
    };  
  
    for (Conta conta : lista) {  
        System.out.println(conta + ", " + conta.getTitular());  
    }  
}
```

[COPIAR CÓDIGO](#)

O código está compilando, e visualmente, é mais sucinto. O foco é nos trechos cuja escrita é necessária, sem necessidade de nomes de métodos e classes que estavam em excesso.

É possível abreviarmos o código ainda mais, não há necessidade do `return`, tampouco das chaves (`{}`), no método `sort()`, com isso, temos de lembrar de remover o ponto e vírgula que vem sempre ao final destas:

```
//Código omitido

public class Teste {

    //Código omitido

    lista.sort(
        (Conta c1, Conta c2) -> Integer.compare(c1.getTitular().getNome().compareTo(c2.getTitular().getNome()))
    );

    Comparator<Conta> comp = (Conta c1, Conta c2) ->
        String nomeC1 = c1.getTitular().getNome();
        String nomeC2 = c2.getTitular().getNome();
        return nomeC1.compareTo(nomeC2);
    };

    for (Conta conta : lista) {
        System.out.println(conta + ", " + conta.getTitular());
    }
}
```

[COPIAR CÓDIGO](#)

Resumimos este método em apenas uma linha.

Como o Java entende que estamos comparando duas contas, não há necessidade de indicarmos o tipo `Conta`, antes de `c1` e `c2`:

```
//Código omitido

public class Teste {
```

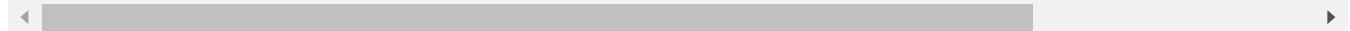
```
//Código omitido

    lista.sort( (c1, c2) -> Integer.compare(c1.getNumero(), c

        Comparator<Conta> comp = (Conta c1, Conta c2) ->
            String nomeC1 = c1.getTitular().getNo
            String nomeC2 = c2.getTitular().getNo
            return nomeC1.compareTo(nomeC2);
    };

    for (Conta conta : lista) {
        System.out.println(conta + ", " + conta.getTitula
    }
}
```

[COPIAR CÓDIGO](#)



Esta sintaxe nos indica que, para as contas `c1` e `c2`, teremos em retorno a comparação na forma de um inteiro. Ela nos priva da burocracia de classes e escrita excessiva, mas demanda uma leitura mais cuidadosa do código.

Assim, temos duas variações de lambdas, primeiro no método `sort()`, em que ela é bastante mínima, com a menor quantidade possível de linhas de código, e em seguida, onde temos a variável `comp`, temos a definição do tipo `Conta`, e a utilização das chaves (`{}`), com o `return` definido. Ambas são válidas.

Assim como o `sort()`, é possível que a lista faça sua própria iteração, para isso, utilizamos o `lista.forEach(action)`:

```
//Código omitido

public class Teste {

//Código omitido
```

```
lista.sort( (c1, c2) -> Integer.compare(c1.getNumero(), c  
  
    Comparator<Conta> comp = (Conta c1, Conta c2) -> {  
        String nomeC1 = c1.getTitular().getNome();  
        String nomeC2 = c2.getTitular().getNome();  
        return nomeC1.compareTo(nomeC2);  
    };  
  
    lista.forEach(action);  
  
    for (Conta conta : lista) {  
        System.out.println(conta + ", " + conta.getTitula  
    }  
}
```

[COPIAR CÓDIGO](#)

Este método sabe fazer seu próprio laço, ele sabe acessar cada elemento, uma vez que os administra. Entretanto, ele não é capaz de definir o que será inserido dentro do laço, ou seja, aquilo que desejamos fazer com cada elemento. Portanto, nossa função será fazer justamente isso.

Dentro dos parênteses do método `forEach`, utilizaremos o atalho "Ctrl + Espaço". Veremos que o ele não recebe um `Comparator`, e sim um `Consumer`. Assim, abriremos esta interface, e notaremos que ela possui um método `accept()`:

```
//Código omitido  
  
public interface Consumer<T> {  
  
    //Código omitido  
  
    void accept(T t);  
  
    //Restante do código omitido
```

[COPIAR CÓDIGO](#)

Assim, definiremos uma classe anônima que implementa esta interface.

Lembrando que ela também utiliza os *generics*:

```
//Código omitido

public class Teste {

    //Código omitido

    lista.sort( (c1, c2) -> Integer.compare(c1.getNumero(), c2.getNumero()));

    Comparator<Conta> comp = (Conta c1, Conta c2) -> {
        String nomeC1 = c1.getTitular().getNome();
        String nomeC2 = c2.getTitular().getNome();
        return nomeC1.compareTo(nomeC2);
    };

    lista.forEach(new Consumer<Conta>() {
        });

    for (Conta conta : lista) {
        System.out.println(conta + ", " + conta.getTitular());
    }
}
```

[COPIAR CÓDIGO](#)

Entretanto, o compilador sinaliza que ainda precisamos implementar o método. O Eclipse nos oferece a opção de adiciona-los, ao clicarmos no ícone de lâmpada, ao lado esquerdo da linha em que o erro é apontado, com isso, temos o seguinte código gerado automaticamente:

```
//Código omitido

public class Teste {
```

```
//Código omitido
```

```
    lista.sort( (c1, c2) -> Integer.compare(c1.getNumero(), c2.getNumero()));

    Comparator<Conta> comp = (Conta c1, Conta c2) -> {
        String nomeC1 = c1.getTitular().getNome();
        String nomeC2 = c2.getTitular().getNome();
        return nomeC1.compareTo(nomeC2);
    };

    lista.forEach(new Consumer<Conta>() {
        @Override
        public void accept(Conta c1) {
            //TODO Auto-generated method stub
        }
    });

    for (Conta conta : lista) {
        System.out.println(conta + ", " + conta.getTitular());
    }
}
```

[COPIAR CÓDIGO](#)

Em seguida, substituiremos o código que foi gerado automaticamente, por aquele que havíamos inserido em nosso laço `for` :

```
//Código omitido
```

```
public class Teste {
```

```
//Código omitido
```

```
    lista.sort( (c1, c2) -> Integer.compare(c1.getNumero(), c2.getNumero()));

    Comparator<Conta> comp = (Conta c1, Conta c2) -> {
```

```

        String nomeC1 = c1.getTitular().getNome();
        String nomeC2 = c2.getTitular().getNome();
        return nomeC1.compareTo(nomeC2);
    };

    lista.forEach(new Consumer<Conta>() {

        @Override
        public void accept(Conta c1) {
            System.out.println(conta + ", " + con
        }
    });

    for (Conta conta : lista) {
        System.out.println(conta + ", " + conta.getTitula
    }
}

```

[COPIAR CÓDIGO](#)



Ao fazer o laço, ele consumirá cada elemento, ou seja, cada `conta`, portanto `c1` dará lugar à `conta`:

```

//Código omitido

public class Teste {

    //Código omitido

    lista.sort( (c1, c2) -> Integer.compare(c1.getNumero(), c

    Comparator<Conta> comp = (Conta c1, Conta c2) -> {
        String nomeC1 = c1.getTitular().getNome();
        String nomeC2 = c2.getTitular().getNome();
        return nomeC1.compareTo(nomeC2);
    };

```

```
lista.forEach(new Consumer<Conta>() {  
  
    @Override  
    public void accept(Conta conta) {  
        System.out.println(conta + ", " + con  
    }  
});  
  
for (Conta conta : lista) {  
    System.out.println(conta + ", " + conta.getTitula  
}  
}  
  
COPIAR CÓDIGO
```



Como podemos notar, este novo código é maior que o anterior, por isso, simplificaremos sua escrita. O primeiro passo será apagar tudo que é inserido no parâmetro, até a entrada `(Conta conta)`:

```
//Código omitido  
  
public class Teste {  
  
//Código omitido  
  
    lista.sort( (c1, c2) -> Integer.compare(c1.getNumero(), c  
  
    Comparator<Conta> comp = (Conta c1, Conta c2) -> {  
        String nomeC1 = c1.getTitular().getNome();  
        String nomeC2 = c2.getTitular().getNome();  
        return nomeC1.compareTo(nomeC2);  
    };  
  
    lista.forEach((Conta conta) {  
        System.out.println(conta + ", " + con  
    }  
});
```

```
        for (Conta conta : lista) {  
            System.out.println(conta + ", " + conta.getTitular());  
        }  
    }
```

[COPIAR CÓDIGO](#)

Precisamos indicar que se trata de uma lambda, fazemos isso utilizando o símbolo
-> :

```
//Código omitido  
  
public class Teste {  
  
    //Código omitido  
  
    lista.sort( (c1, c2) -> Integer.compare(c1.getNumero(), c2.getNumero()) );  
  
    Comparator<Conta> comp = (Conta c1, Conta c2) -> {  
        String nomeC1 = c1.getTitular().getNome();  
        String nomeC2 = c2.getTitular().getNome();  
        return nomeC1.compareTo(nomeC2);  
    };  
  
    lista.forEach((Conta conta) -> {  
        System.out.println(conta + ", " + conta.getTitular());  
    });  
  
    for (Conta conta : lista) {  
        System.out.println(conta + ", " + conta.getTitular());  
    }  
}
```

[COPIAR CÓDIGO](#)

Assim como anteriormente, não há necessidade de fazermos referência ao tipo `Conta`. Tampouco há necessidade para as chaves ({}), já que estamos trabalhando com uma linha de código somente:

```
//Código omitido

public class Teste {

    //Código omitido

    lista.sort( (c1, c2) -> Integer.compare(c1.getNumero(), c

        Comparator<Conta> comp = (Conta c1, Conta c2) -> {
            String nomeC1 = c1.getTitular().getNome();
            String nomeC2 = c2.getTitular().getNome();
            return nomeC1.compareTo(nomeC2);
        };

        lista.forEach( (conta) -> System.out.println(conta + ", " + conta.getTitular().getNome())
    }
}
```

[COPIAR CÓDIGO](#)

Assim foi possível escrever tudo em uma só linha.

Quem faz o laço é a própria `lista`, para cada elemento que for uma `conta`, será aplicado o `System.out.println()` que definimos. Podemos apagar o laço `for` antigo:

```
//Código omitido
```

```
public class Teste {  
  
    //Código omitido  
  
    lista.sort( (c1, c2) -> Integer.compare(c1.getNumero(), c2.getNumero()) );  
  
    Comparator<Conta> comp = (Conta c1, Conta c2) -> {  
        String nomeC1 = c1.getTitular().getNome();  
        String nomeC2 = c2.getTitular().getNome();  
        return nomeC1.compareTo(nomeC2);  
    };  
  
    lista.forEach( conta ) -> System.out.println(conta);  
}
```

[COPIAR CÓDIGO](#)



Testaremos o código, executando-o, e temos o seguinte resultado no console:

```
ContaPoupanca, Numero: 11, Agencia: 22, Saldo: 111.0, Paulo  
ContaPoupanca, Numero: 22, Agencia: 22, Saldo: 222.0, Ana,  
ContaCorrente, Numero: 33, Agencia: 22, Saldo: 333.0, Nico  
ContaCorrente, Numero: 44, Agencia: 22, Saldo: 444.0, Guilherme
```

[COPIAR CÓDIGO](#)

Nosso código está funcionando.

Para testarmos o `comp`, utilizaremos o método `sort()` tendo ele como parâmetro:

```
//Código omitido  
  
public class Teste {
```

//Código omitido

```
lista.sort( c1, c2 ) -> Integer.compare(c1.getNumero(), c2.getNumero()) < 0
```

```
Comparator<Conta> comp = (Conta c1, Conta c2) -> {
    String nomeC1 = c1.getTitular().getNome();
    String nomeC2 = c2.getTitular().getNome();
    return nomeC1.compareTo(nomeC2);
};

lista.sort( comp );

lista.forEach( conta ) -> System.out.println(conta)
```

}

[COPIAR CÓDIGO](#)



Executando, temos o seguinte resultado:

```
ContaPoupanca, Numero: 22, Agencia: 22, Saldo: 222.0, Ana
ContaCorrente, Numero: 44, Agencia: 22, Saldo: 444.0, Guilherme
ContaCorrente, Numero: 33, Agencia: 22, Saldo: 333.0, Nico
ContaPoupanca, Numero: 11, Agencia: 22, Saldo: 111.0, Paulo
```

[COPIAR CÓDIGO](#)

Funciona também!

Há um universo dos lambdas, muito maior do que vimos aqui, com outros paradigmas de programação funcional, e que pode ser visto em outro curso. Nosso objetivo aqui foi explicar a necessidade deles, e o porquê de existirem.

Vamos para os exercícios, e até a próxima!

O padrão Iterator

Você já sabe agora que existem muitas coleções. Só nesse treinamento vimos `ArrayList`, `LinkedList` e `Vector`. Se você assistir ainda o curso dedicado as coleções você aprenderá as interfaces para fila (`Queue`), conjunto (`Set`) e mapa (`Map`) cada uma com várias implementações.

Aí vem uma pergunta: Como posso acessar (*iterar*) todas essas implementações de maneira uniforme sem saber os detalhes de cada implementação? A resposta está na "caixa de padrões de projeto" e se chama `Iterator`.

Uma `Iterator` é um objeto que possui no mínimo dois métodos: `hasNext()` e `next()`. Ou seja, você pode usá-lo para perguntar se existe um próximo elemento e pedir o próximo elemento. A notícia boa é que isso funciona com TODAS as implementações e aí a grande vantagem.

Veja o código para usar o `Iterator` de uma lista:

```
List<String> nomes = new ArrayList<>();
nomes.add("Super Mario");
nomes.add("Yoshi");
nomes.add("Donkey Kong");

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

while(it.hasNext()) {
    System.out.println(it.next());
}
```

[COPIAR CÓDIGO](#)

Se você entendeu esse código, você já aprendeu como iterar com filas, conjuntos ou mapas. Veja o uso do `Iterator` através de um conjunto:

```
Set<String> nomes = new HashSet<>();
nomes.add("Super Mario");
nomes.add("Yoshi");
nomes.add("Donkey Kong");

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

while(it.hasNext()) {
    System.out.println(it.next());
}
```

[COPIAR CÓDIGO](#)

Se ficou ainda com dúvida sobre o `Iterator`, sem problemas (!) pois o curso [Dominando as Collections](https://cursos.alura.com.br/course/java-collections) (<https://cursos.alura.com.br/course/java-collections>). possui um capítulo dedicado.



Transcrição

Olá!

Chegamos ao último vídeo do curso, concluindo uma longa trilha que iniciou desde os conceitos mais básicos do Java.

Iniciamos pelos **arrays**, a primeira estrutura de dados que vimos. Com ela, aprendemos a armazenar diversos valores de uma só vez, vimos o que é um array de primitivos, com uma sintaxe incomum, que utiliza colchetes ([]).

Inicializamos o array, indicando a posição desejada, lembrando que ela sempre inicia do `0` , e aprendemos a acessar determinada posição, bem como a descobrir quantos elementos aquela lista é capaz de armazenar.

Assim como existem primitivos, aprendemos que podem existir arrays de referências. Ao criar o objeto, o que é armazenado é a indicação à ele, e não ele em si. De resto, o funcionamento é igual ao tipo de array visto anteriormente.

Um ponto negativo deste tipo de lista é seu número fixo de elementos, uma vez criada, ela terá aquele tamanho, sem possibilidade de alteração ou flexibilização. Além disso, apesar de ser possível sabermos esta capacidade de armazenamento, não há um método que nos permita descobrir o número de elementos armazenados.

Por este motivo, vimos que foi criado um tipo de armazenador de objetos de referência, que é o `ArrayList`. Trata-se de uma classe que, internamente, utiliza um array. Sua sintaxe é, portanto, a mesma de uma classe qualquer.

Além disso, vimos que é possível criar parâmetros nela, para indicarmos que só será possível armazenar um tipo de referência, por exemplo, do tipo `Conta`, como é o caso abaixo:

```
ArrayList<Conta> lista = new ArrayList<Conta>();
```

```
Conta cc1 = new ContaCorrente(22, 11);  
lista.add(cc1);
```

```
Conta cc2 = new ContaPoupanca(22, 22);  
lista.add(cc2);
```

```
System.out.println("Tamanho: " + lista.size());
```

COPIAR CÓDIGO

Ela possui diversos métodos específicos que facilitam o trabalho com estrutura de dados, é o caso do `size()`, que nos permite descobrir o número de referências armazenadas.

Há ainda outros tipos de estruturas de dados, como vimos, temos: `LinkedList` e o `Vector`.

O `LinkedList` é uma lista duplamente encadeada, enquanto o `Vector` é similar ao `ArrayList`, mas é capaz de trabalhar com diversos *threads*. Como são todas listas, implementam a interface `List`.

Lembrando que todas as listas são sequências, e armazenam os elementos com base em um índice, isso não impede que adicionemos duplicados, ou seja, que itens sejam repetidos.

A ideia de armazenar dados é presente no mundo Java também na `Collection`, uma interface ainda mais genérica que `List`, e nos conjuntos `Set` e `HashSet`, que por sua vez não aceitam itens duplicados.

Como não é possível guardarmos valores primitivos em um array, existe para cada um deles uma **classe wrapper** correspondente.

Esta transformação, do primitivo para o objeto, e vice-versa, é chamada de *autoboxing* e *unboxing*, e é feita automaticamente pelo Java.

Vimos uma variação dos *generics*, onde omitimos o tipo, e o código continua funcionando:

```
List<Integer> lista = new ArrayList<>();
```

```
lista.add(5);
lista.add(12);

System.out.println("Tamanho: " + lista.size());
```

[COPIAR CÓDIGO](#)

Aprendemos ainda formas de ordenar nossos métodos. Temos o `sort()`, bem como a classe `Collections`, que possui uma série de métodos auxiliares estáticos.

O `sort()` recebe um critério de comparação, que é definido por meio de uma interface com um único método.

Estas classes sem atributos, que possuem uma única funcionalidade, foram trabalhadas quando tratamos das **classes anônimas e lambdas**.

Como vimos, é menos custoso criar uma classe, a partir da interface, e já criar o objeto, de forma direta, em unidade. Isso significa que, em vez de fazermos isso, que é um processo burocrático:

```
lista.sort( new Comparator<Conta>() {

    public int compare(Conta c1, Conta c2) {

        return Integer.compare(c1.getNumero(), c2.getNumero());
    }
}
```

```
}
```

```
});
```

[COPIAR CÓDIGO](#)

É melhor escrevermos da seguinte forma:

```
lista.sort( (c1, c2) -> Integer.compare(c1.getNumero(), c2.getNumero()) );
```

[COPIAR CÓDIGO](#)

Neste segundo caso, estamos utilizando os lambdas, que facilitam ainda mais o trabalho com funções, no caso `(c1, c2)`, e em seguida, o código com o resultado desejado, ou seja, o nosso método `compare`.

Gostaria de convidar você para o próximo curso, Java Parte 7, no qual falaremos sobre `java.io`.

Aprenderemos a trabalhar com leitura de arquivos, escrita, fluxo de dados oriundos da rede, ou ainda do teclado, onde podemos enviar-los, entre outros.

É um tópico importante, principalmente para aquele que têm interesse em trabalhar com a web, já que se lida bastante com a relação de input e output.

Se quiser entender este código abaixo, convido você, e agradeço por ter assistido a este curso, até a próxima!

```
class TestaEntrada {  
  
    public static void main(String[] args) throws IOException {  
  
        BufferedReader br =  
  
            new BufferedReader(  
  
                new InputStreamReader(  
  
                    new FileInputStream("arquivo.txt")));  
        while (s != null) {  
  
            System.out.println(s);  
  
            s = br.readLine();  
  
        }  
  
        br.close;  
    }  
}
```

COPIAR CÓDIGO

