



AULA DE JAVA 2 – CONCEITOS AVANÇADOS

DAS 5316 – Integração de Sistemas Corporativos

Roque Oliveira Bezerra, M. Eng.

roque@das.ufsc.br

Prof. Ricardo J. Rabelo



ROTEIRO

- Recaptulação da aula anterior
- Exceções
- Java Beans
- Classe Object
 - toString()
 - equals()
- StringBuilder
- Coleções
- Introdução ao Swing

RECAPTULAÇÃO DA AULA ANTERIOR

- Strings
 - Sempre usar “equals”, nunca usar “==”
- Entrada e Saída
 - Scanner
- Arrays
- Classes
 - Atributos
 - Métodos
 - Construtores
 - Herança
- Interfaces

EXCEÇÕES

- Exceções são construções usadas para indicar condições anormais dentro de um programa.
- Em Java, exceções são classes derivadas da classe Exception.
- Java provê diversos tipos de exceções, mas, caso necessário, outras podem ser criadas pelo programador.

EXCEÇÕES

- Condições anormais são indicadas *lançando-se* exceções, através da palavra-chave **throw**.

```
if (temperatura > 5000)
    throw new SuperAquecimento();
```

- Métodos que podem lançar exceções devem indicar os tipos de exceção com a palavra-chave **throws** no final de suas assinaturas.

```
void aumentaTemperatura(int x) throws SuperAquecimento
{
    temperatura += x;

    if (temperatura > 5000)
        throw new SuperAquecimento();
}
```

EXCEÇÕES

- Ao se chamar um método que pode gerar uma exceção, existem duas alternativas:
 - Tratar a possível exceção;
 - Passar o tratamento adiante.
- Para postergar o tratamento, basta indicar novamente que o método atual lança esta exceção.

```
void executaComando() throws SuperAquecimento
{
    int temp = lerValorDoUsuario();
    aumentaTemperatura(temp);
}
```

EXCEÇÕES - TRATAMENTO

- Em algum momento a exceção precisa ser tratada
- O tratamento é feito com o bloco **try ... catch**

```
void executaComando()  
{  
    int temp = lerValorDoUsuario();  
  
    try  
    {  
        aumentaTemperatura(temp);  
    }  
    catch (SuperAquecimento sa)  
    {  
        desligar();  
        alarme();  
    }  
}
```

EXCEÇÕES - TRATAMENTO

- O bloco **try ... catch** pode ter opcionalmente uma cláusula **finally**, contendo um trecho de código que executará independentemente de ocorrer ou não a exceção.

```
void executaComando()
{
    int temp = lerValorDoUsuario();

    try {
        aumentaTemperatura(temp);
    }
    catch (SuperAquecimento sa) {
        desligar();
        alarme();
    }
    finally {
        mostraTemperatura();
    }
}
```


JAVA BEANS

- Java Beans foi o primeiro modelo de componentes reutilizáveis Java.
- Geralmente são associados com componentes de interface gráfica, mas *beans* também podem ser não-visuais.
- *Beans* possuem propriedades e eventos, que ficam acessíveis a outras ferramentas.

JAVA BEANS

- Um *bean* é apenas uma classe que segue um padrão especial para os nomes de seus métodos, permitindo assim que outras ferramentas descubram suas propriedades e eventos, através de introspecção.
- A ideia é que cada *bean* represente um componente do sistema, com propriedades que possam ser lidas ou (às vezes) alteradas

JAVA BEANS - PROPRIEDADES

- Um mecanismo muito simples é usado pra criar propriedades
- Qualquer par de métodos:

```
public Tipo getNomeDaPropriedade();  
public void setNomeDaPropriedade(Tipo novoValor);
```

leitura e escrita

- O nome da propriedade é o que vem depois do *get/set*
- Para uma ferramenta, esta propriedade seria reconhecida como
 - `nomeDaPropriedade`

JAVA BEANS - PROPRIEDADES

- Pode-se criar propriedades de apenas leitura se apenas o método *get* for criado
- Para propriedades do tipo *boolean* pode-se usar o prefixo *is* no lugar de *get*.

é reconhecido como propriedade de apenas leitura
chamada `public boolean isFull();`

JAVA BEANS – POR QUÊ?

- Por que usar *getters* e *setters*, se poderia-se deixar os atributos da classe diretamente públicos?

JAVA BEANS – POR QUÊ?

- Por que usar *getters* e *setters*, se poderia-se deixar os atributos da classe diretamente públicos?

Pode-se validar uma propriedade na hora de se atribuir um valor

```
class Pessoa
{
    private int idade;

    public int getIdade()
    {
        return idade;
    }

    public void setIdade(int idade)
    {
        this.idade = idade;
        if (this.idade < 0)
            this.idade = 0;
    }
}
```

JAVA BEANS – POR QUÊ?

- Por que usar *getters* e *setters*, se poderia-se deixar os atributos da classe diretamente públicos?

Pode-se ter propriedades que não são associadas a atributos de classe

```
class Sorteador
{
    public int getProximoNumero()
    {
        return Math.random() * 10;
    }
}
```

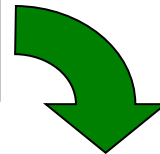
MÉTODOS DA CLASSE OBJECT

- Java possui uma classe raiz, da qual toda classe deriva implicitamente
- Certos métodos utilitários que ela provê podem ser sobrescritos:
 - `toString()`
 - `equals(Object o)`
 - `finalize()`
 - `hashCode()`

STRING toString()

- Gera uma representação textual do objeto, em forma de *string*.
- É um dos princípios de funcionamento da concatenação de *strings*.

```
Date d = new Date();  
String s = "Hoje é " + d;
```



```
Date d = new Date();  
String s = "Hoje é " + d.toString();
```

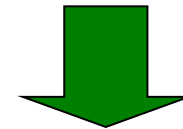
STRING toString()

- Sempre que se precisa de uma representação *string* de um objeto, este método é usado. Por exemplo, dentro de `System.out.println()`.
- Assim, *toString* possui grande utilidade para debug.

```
class Pessoa
{
    private String nome;
    private int idade;

    public String toString()
    {
        return nome + ", com "
            + idade;
    }
}
```

```
Pessoa p = new Pessoa(...);
...
System.out.println(p);
```



Joãozinho, com 14

BOOLEAN_EQUALS (OBJECT O)

- Compara a igualdade do objeto atual com um outro passado de parâmetro.
- O operador “==” compara instâncias, não conteúdos.

```
Integer i1 = new Integer(4);  
Integer i2 = new Integer(4);  
  
i1 == i2          → false  
i1.equals(i2)    → true
```

- A implementação padrão é equivalente ao “==”.
- *Strings* SEMPRE devem ser comparadas usando *equals*.

BOOLEAN EQUALS (OBJECT o)

- Bibliotecas, sempre que precisam avaliar a equivalência de objetos, usam o *equals*.
- Sempre que objetos diferentes podem ser equivalentes, deve-se implementar *equals*.
- Equals deve ser reflexivo, ou seja:

```
a.equals(b) == b.equals(a)
```

BOOLEAN EQUALS (OBJECT o)

○ Exemplo:

```
class Pessoa
{
    private String nome;
    private int idade;

    public boolean equals(Object o)
    {
        if (this == o)
            return true;
        else if (o == null || getClass() != o.getClass())
            return false;
        else
        {
            Pessoa p = (Pessoa) o;
            return nome.equals(p.nome) && idade == p.idade;
        }
    }
}
```

BOOLEAN_EQUALS (OBJECT o)

Exemplo:

Otimização:

se o parâmetro sou
eu mesmo, então eu
sou equivalente a ele

```
    String nome;  
    int idade;  
  
    public boolean equals(Object o)  
    {  
        if (this == o)  
            return true;  
        else if (o == null || getClass() != o.getClass())  
            return false;  
        else  
        {  
            Pessoa p = (Pessoa) o;  
            return nome.equals(p.nome) && idade == p.idade;  
        }  
    }  
}
```

BOOLEAN EQUALS (OBJECT o)

- Exemplo:

Checagem contra estranhos:

Se passaram um objeto *null*,
ou de uma classe diferente da
minha, então não posso ser
equivalente a ele

```
class Pessoa {
    // ...
    public boolean equals(Object o) {
        if (this == o)
            return true;
        else if (o == null || getClass() != o.getClass())
            return false;
        else {
            Pessoa p = (Pessoa) o;
            return nome.equals(p.nome) && idade == p.idade;
        }
    }
}
```

BOOLEAN_EQUALS (OBJECT o)

- Exemplo:

```
class Pessoa
```

Comparação dos atributos:

Nesse ponto é garantido que o parâmetro é da minha classe, e diferente de mim. Então ele vai ser equivalente a mim, se nossos atributos forem equivalentes

```
    else if (o == null || getClass() != o.getClass())  
        return false;  
    else  
    {  
        Pessoa p = (Pessoa) o;  
        return nome.equals(p.nome) && idade == p.idade;  
    }  
}
```


VOID FINALIZE ()

- Método chamado pela máquina virtual antes do objeto ser coletado pelo *Garbage Collector*.
- Pode ser usado para garantir a liberação de recursos, como arquivos e conexões de rede ou banco de dados.

INT hashCode ()

- Chamado em geral por classes que implementam tabelas *hash*, para armazenar objetos.
- Este método deve retornar um número inteiro – o código *hash* do objeto – que tabelas hash usarão quando o objeto for armazenado
- Segue o mesmo princípio de *equals*: objetos diferentes, mas equivalentes, devem ter o mesmo código *hash*.

JAVA.LANG.STRINGBUILDER

- *Strings* em java são imutáveis.
 - *Strings* não permitem alteração em seus caracteres individuais
 - Operações que aparentemente alteram *strings* na verdade geram novos objetos *string*.

```
String str = "a";  
  
str += "b"; //str = str + "b";
```

- Para se trabalhar com *strings* nas quais se deseja modificar o conteúdo, a biblioteca Java oferece a classe *StringBuilder*, que possui os mesmos métodos da classe *String*, além de métodos de alteração.
- Os principais são:
 - `append(...)` adiciona o parâmetro ao final da string
 - `insert(int offset, ...)` insere o parâmetro na posição
 - `setCharAt(int index, char ch)` altera o caracter na posição
- Existe também a classe *StringBuffer*, que é equivalente, mas seus métodos são sincronizados para acesso concorrente

JAVA.LANG.STRINGBUILDER

- Exemplo

```
public String teste()
{
    StringBuilder bfr = new StringBuilder();

    for (int i=0; i<10; i++) // "0123456789"
        bfr.append(i);

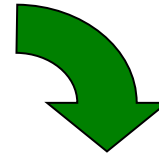
    bfr.append("lala").append("la"); // "0123456789lalala"
    bfr.insert(12, false); // "0123456789lafalselala"
    bfr.setCharAt(15, 'x'); // "0123456789lafalxelala"
    return bfr.toString();
}
```

"0123456789lafalxelala"

JAVA.LANG.STRINGBUILDER

- O segundo princípio de funcionamento da concatenação de *strings* é o uso de *StringBuilders*.

```
String s = a + " x " + b + " = " + (a*b);
```



```
String s = new StringBuilder().append(a).append(" x ")  
    .append(b).append(" = ").append(a*b);
```

COLEÇÕES

- Originalmente, Java possuía duas classes de coleção de objetos
 - Vector – Lista de objetos
 - HashTable – Tabela de associação
- Devido a certas limitações destas classes, foi desenvolvido o *Java Collections Framework*.
- Framework baseado em interfaces, implementações e classes auxiliares.
- Pertencem à package `java.util`.

PARÂMETROS GENÊRICOS

- A partir da versão 5.0 (1.5), Java permite que classes recebam parâmetros que indiquem os tipos de dados usados por elas
- Assim, as classes e interfaces da API de coleções possuem estes parâmetros adicionais para que se possa indicar o tipo de objetos que elas podem armazenar. Por exemplo:
 - `Lista<E> { ... }`
- Quando se for declarar uma variável de um tipo que tenha parâmetros genéricos, deve-se indicar seu valor:
 - `Lista<String> lst = ...; ou`
 - `Lista<Produto> lst = ...; ou`
 - `Lista<Robo> lst = ...; e assim por diante`
- O compilador usa essa informação adicional para checar tentativas de se inserir objetos de tipos diferentes do tipo da coleção, e elimina a necessidade de *typecasts* quando se obtém um item da coleção

COLEÇÕES

HIERARQUIA DE INTERFACES

- `Collection<E>`
 - `List<E>`
 - `Set<E>`
 - `SortedSet<E>`
 - `Queue<E>`
- `Map<K, V>`
 - `SortedMap<K, V>`

COLLECTION<E>

- Representa uma coleção arbitrária de objetos.
- Principais métodos:
 - **int** `size()` ;
 - Número de elementos da coleção
 - **boolean** `contains(Object element)` ;
 - Checa se *element* pertence à coleção
 - **boolean** `add(E element)` ;
 - Adiciona *element* à coleção
 - **boolean** `remove(Object element)` ;
 - Remove *element* da coleção
 - **void** `clear()` ;
 - Remove todos os elementos da coleção
 - `Iterator<E> iterator()` ;
 - Cria um *Iterator*, para iterar pelos elementos da coleção.

LIST<E>

- List é uma coleção indexada de objetos.
- Possui os seguintes métodos além dos herdados de *Collection*:
 - `E get(int index);`
 - Acessa o i-ésimo elemento da lista
 - `E set(int index, E element);`
 - Altera o i-ésimo elemento da lista
 - `void add(int index, E element);`
 - Adiciona um elemento na posição i da lista. Se havia elementos após este índice, eles serão movidos
 - `Object remove(int index);`
 - Remove o i-ésimo elemento da lista
 - `int indexOf(Object o);`
 - Obtém o índice de um elemento

LIST – ARRAYLIST X LINKEDLIST

- *List* possui duas implementações principais:
 - *ArrayList*
 - Elementos são armazenados de forma contígua, em um array.
 - Acesso indexado rápido.
 - Inserções e remoções no meio da lista são lentos.
 - *LinkedList*
 - Elementos são armazenados na forma de uma lista encadeada.
 - Acesso indexado péssimo. Precisa percorrer toda a lista.
 - Inserções e remoções no meio da lista são rápidos.

EXEMPLO – ARRAYLIST

```
public void teste()
{
    List<String> trap = new ArrayList<String>();

    trap.add("Didi");
    trap.add("Dedé");
    trap.add("Mussum");
    trap.add("Zacarias");

    for (int i=0; i < trap.size(); i++)
    {
        String str = trap.get(i);
        System.out.println( str );
    }

    trap.remove(3);
    trap.remove("Mussum")

    for (String s : trap)
        System.out.println( s );

    int idx = lista.indexOf("Didi");

    lista.set(idx, "Beto Carrero");
}
```

EXEMPLO – ARRAYLIST

```
public void teste()
{
    List<String> trap = new ArrayList<String>();
    trap.add("Didi");
    trap.add("Pedé");
    // ...
    trap.remove(3);
    trap.remove("Mussum");
    for (String s : trap)
        System.out.println( s );
    int idx = lista.indexOf("Didi");
    lista.set(idx, "Beto Carrero");
}
```

Declara um *List* de *Strings*

Foi escolhido *ArrayList* como
implementação desta lista

EXEMPLO – ARRAYLIST

```
public void teste()  
{  
    List<String> trap = new ArrayList<String>();  
    trap.add("Didi");  
    trap.add("Dedé");  
    trap.add("Mussum");  
    trap.add("Zacarias");
```

Adiciona objetos à lista

Apenas Strings são permitidas, caso se tentasse adicionar um objeto de outra classe, o compilador indicaria o erro

```
ze(); i++)
```

```
(i);  
tr );
```

```
);
```

```
int idx = lista.indexOf("Didi");
```

```
lista.set(idx, "Beto Carrero");
```

```
}
```

EXEMPLO – ARRAYLIST

Itera pelos objetos da lista usando índices

```
public void teste()
{
    ArrayList<String> trap = new ArrayList<String>();
    trap.add("Mussum");
    trap.add("Zacarias");

    for (int i=0; i < trap.size(); i++)
    {
        String str = trap.get(i);
        System.out.println( str );
    }

    trap.remove(3);
    trap.remove("Mussum")

    for (String s : trap)
        System.out.println( s );

    int idx = lista.indexOf("Didi");
    lista.set(idx, "Beto Carrero");
}
```

EXEMPLO – ARRAYLIST

Remove elementos

- Pelo índice
- Pelo valor

```
public void teste()
{
    List<String> trap = new ArrayList<String>();

    trap.add("Didi");
    trap.add("Dedé");
    trap.add("Mussum");

    for (int i = 0; i < trap.size(); i++)
    {
        String str = trap.get(i);
        System.out.println( str );
    }

    trap.remove(3);
    trap.remove("Mussum");

    for (String s : trap)
        System.out.println( s );

    int idx = lista.indexOf("Didi");

    lista.set(idx, "Beto Carrero");
}
```


EXEMPLO – ARRAYLIST

```
public void teste()
{
    List<String> trap = new ArrayList<String>();

    trap.add("Didi");
    trap.add("Dedé");
    trap.add("Mussum");
    trap.add("Zacarias");

    for (int i = 0; i < trap.size(); i++)
    {
        String str = trap.get(i);
        System.out.println( str );
    }

    trap.remove(3);
    trap.remove("Mussum");

    for (String s : trap)
        System.out.println( s );

    int idx = lista.indexOf("Didi");

    lista.set(idx, "Beto Carrero");
}
```

Itera pela lista sem
usar índices

EXEMPLO – ARRAYLIST

```
public void teste()
{
    List<String> trap = new ArrayList<String>();

    trap.add("Didi");
    trap.add("Dedé");
    trap.add("Mussum");
    trap.add("Zacarias");

    for (int i=0; i < trap.size(); i++)
    {
        String str = trap.get(i);
        System.out.println( str );
    }

    lista.set(0, "Beto Carrero");
}
```

Obtém índice de um elemento

Altera um elemento em um índice

SET<E> / HASHSET<E>

- Sets são coleções que não possuem objetos repetidos
- Possui os seguintes métodos, além dos herdados de *Collection*:
 - **boolean** `addAll(Set<E> set);`
 - Adiciona ao Set todos os elementos do set passado de parâmetro.
 - Equivale a: `this = this ∪ set`
 - **boolean** `retainAll(Set<E> set);`
 - Remove do Set todos os elementos, exceto aqueles também pertencentes ao set passado de parâmetro.
 - Equivale a: `this = this ∩ set`
 - **boolean** `removeAll(Set<E> set);`
 - Remove do Set todos os elementos também pertencentes ao set passado de parâmetro.
 - Equivale a: `this = this - set`
- A principal implementação de *Set* é a classe *HashSet*, que usa os métodos *hashCode* e *equals* dos objetos para armazená-los

SET<E> / HASHSET<E>

```
public void teste()
{
    Set<String> s = new HashSet<String>();

    s.add("Cachorro");
    s.add("Gato");
    s.add("Galinha");
    s.add("Gato");

    if (s.contains("Galinha"))
    {
        s.remove("Galinha");
    }
    else
    {
        s.add("Cavalo");
        s.add("Boi");
    }
}
```

SET<E> / HASHSET<E>

```
public void teste()
{
    Set<String> s = new HashSet<String>();
    s.add("Cachorro");
    s.add("Gato");
    s.add("Galinha");
    s.remove("Galinha");
    else
    {
        s.add("Cavalo");
        s.add("Boi");
    }
}
```

Cria um Set de Strings,
implementado como
um HashSet

SET<E> / HASHSET<E>

Adiciona objetos ao conjunto.

Objetos repetidos não são adicionados

```
HashSet<String>();
```

```
s.add("Cachorro");  
s.add("Gato");  
s.add("Galinha");  
s.add("Gato");  
  
if (s.contains("Galinha"))  
{  
    s.remove("Galinha");  
}  
else  
{  
    s.add("Cavalo");  
    s.add("Boi");  
}  
}
```

MAP<K, V>

- *Map* é uma coleção associativa.
- Valores *V* são inseridos nele associando-se uma chave *K*.
- Esta chave pode ser usada para obter novamente o valor.
- A principal implementação de *Map* é *HashMap*
- Principais métodos
 - `V put(K key, V value);`
 - Adiciona o objeto *value*, associado com *key*
 - `V get(Object key);`
 - Acessa o objeto associado com *key*
 - `V remove(Object key);`
 - Remove o objeto associado com *key*
 - `int size();`
 - Número de elementos do *Map*

HASHMAP<K, V>

```
public void teste()  
{  
    Map<String, Pato> m =  
        new HashMap<String, Pato>();  
  
    m.put("Huguinho", new Pato(...));  
    m.put("Zezinho", new Pato(...));  
    m.put("Luizinho", new Pato(...));  
  
    Pato p = m.get("Zezinho");  
}
```


HASHMAP<K, V>

```
public void teste()  
{  
    Map<String, Pato> m =  
        new HashMap<String, Pato>();  
  
    m.put("Huguinho", new Pato(...));  
    m.put("Zezinho", new Pato(...));  
    m.put("Luizinho", new Pato(...));  
}
```

Cria um *Map* de *String* para *Pato*,
implementado como um *HashMap*

HASHMAP<K, V>

Adiciona elementos no *Map*

Obtém o elemento associado à chave
Zezinho

```
ring, Pato>();
```

```
m.put("Huguinho", new Pato(...));  
m.put("Zezinho", new Pato(...));  
m.put("Luizinho", new Pato(...));
```

```
Pato p = m.get("Zezinho");  
}
```

MAP - SUBCOLEÇÕES

- `Set<K> keySet () ;`
 - Acessa o conjunto das chaves do *Map*
- `Collection<V> values () ;`
 - Acessa a coleção de valores do *Map*
- `Set<Map.Entry<K, V>> entrySet () ;`
 - Acessa o conjunto de entradas *Map*

```
class Map.Entry<K, V>
{
    K getKey();
    V getValue();
    V setValue(V value);
}
```

MAP

```
public void teste()
{
    Map<String, Pato> m = new HashMap<String, Pato>();

    m.put("Huguinho", new Pato(...));
    m.put("Zezinho", new Pato(...));
    m.put("Luizinho", new Pato(...));

    for (Pato p : m.values())
    {
        System.out.println(p);
    }

    for (String s : m.keySet())
    {
        Pato p = m.get(s);
        System.out.println(s + " -> " + p);
    }

    for (Map.Entry<String, Pato> e : m.entrySet())
    {
        System.out.println(e.getKey() + " -> " + e.getValue());
    }
}
```

MAP

```
public void teste()
{
    Map<String, Pato> m = new HashMap<String, Pato>();

    m.put("Huquinho", new Pato(...));
    m.put("João", new Pato(...));
    m.put("Zinho", new Pato(...));

    for (Pato p : m.values())
    {
        System.out.println(p);
    }

    for (String s : m.keySet())
    {
        Pato p = m.get(s);
        System.out.println(s + " -> " + p);
    }

    for (Map.Entry<String, Pato> e : m.entrySet())
    {
        System.out.println(e.getKey() + " -> " + e.getValue());
    }
}
```

Itera pelos valores

MAP

```
public void teste()
{
    Map<String, Pato> m = new HashMap<String, Pato>();

    m.put("Huguinho", new Pato(...));
    m.put("Zezinho", new Pato(...));
    m.put("Luizinho", new Pato(...));
}
```

Itera pelas chaves
Usa as chaves para
acessar os valores

```
    for (String s : m.keySet())
    {
        Pato p = m.get(s);
        System.out.println(s + " -> " + p);
    }

    for (Map.Entry<String, Pato> e : m.entrySet())
    {
        System.out.println(e.getKey() + " -> " + e.getValue());
    }
}
```

MAP

```
public void teste()
{
    Map<String, Pato> m = new HashMap<String, Pato>();

    m.put("Huguinho", new Pato(...));
    m.put("Zezinho", new Pato(...));
    m.put("Luizinho", new Pato(...));

    for (Pato p : m.values())
    {
        System.out.println(p);
    }

    for (String s : m.keySet())
    {
        Pato p = m.get(s);
        System.out.println(s + " -> " + p);
    }

    for (Map.Entry<String, Pato> e : m.entrySet())
    {
        System.out.println(e.getKey() + " -> " + e.getValue());
    }
}
```

Itera pelas entradas
do Map

COLLECTIONS – ALGORITMOS

- Além das interfaces e implementações, o *framework* de coleções possui a classe *Collections*, com algoritmos genéricos e métodos utilitários.

- **void** `sort(List<E> list);`
- **void** `reverse(List<E> list);`
- **void** `shuffle(List<E> list);`
- `E min(Collection<E> coll);`
- `E max(Collection<E> coll);`

COLLECTIONS – ALGORITMOS

```
public void teste()
{
    List<String> lista = new ArrayList<String>();

    lista.add("Verde");
    lista.add("Amarelo");
    lista.add("Azul");
    lista.add("Branco");

    System.out.println(lista);

    Collections.sort(lista);    //Ordena
    System.out.println(lista);

    Collections.reverse(lista); //Inverte
    System.out.println(lista);

    Collections.shuffle(lista); //Embaralha
    System.out.println(lista);

    String s = Collections.min(lista); //Obtem o mínimo
    System.out.println("Mínimo = " + s);
}
```

COLLECTIONS – ADAPTADORES

- *Collections* possui ainda métodos para gerar adaptadores não-modificáveis de outras coleções
 - `Collection<E> unmodifiableCollection(Collection<E> c);`
 - `Set<E> unmodifiableSet(Set<E> s);`
 - `List<E> unmodifiableList(List<E> list);`
 - `Map<K, V> unmodifiableMap(Map<K, V> m);`
- Qualquer operação que modificaria a coleção retornada por um destes métodos gera uma *UnsupportedOperationException*.
- Operações não modificantes são delegadas para a coleção original;
- Não é feita nenhuma cópia de objetos, ou seja, não há problema de desempenho

COLLECTIONS – ADAPTADORES

```
public void teste()
{
    List<String> lista = new ArrayList<String> ();

    lista.add("Verde");
    lista.add("Amarelo");
    lista.add("Azul");
    lista.add("Branco");

    List<String> lista2 = Collections.unmodifiableList(lista);

    String s = lista2.get(3); //ok

    lista2.add("Vermelho");    //exceção
}
```

COLEÇÕES E TIPOS SIMPLES

- As coleções aqui apresentadas podem armazenar apenas objetos
- Ou seja, não se pode declarar coisas como:
 - `List<int>`
 - `Set<char>`
 - `Map<String, boolean>`
- Para estes casos, deve-se usar a classe adaptadora associada a cada tipo simples:
 - `List<Integer>`
 - `Set<Character>`
 - `Map<String, Boolean>`

COLEÇÕES E TIPOS SIMPLES

- Quando se for adicionar, ou simplesmente acessar, objetos em coleções deste tipo, o compilador dá uma ajuda, fazendo a conversão automaticamente:

```
public void teste()
{
    List<Integer> lista = new ArrayList<Integer> ();

    lista.add(1);
    lista.add(2);
    lista.add(42);
    lista.add(1000);

    for (int i : lista);
    {
        System.out.println("i² = " + (i * i) );
    }

    Map<String, Boolean> m = new HashMap Map<String, Boolean>();

    m.put("A", true)
    m.put("B", true)
    m.put("C", false)

    boolean b = m.get("C");
}
```

COLEÇÕES E TIPOS SIMPLES

- Quando se for adicionar, ou simplesmente acessar, objetos em coleções deste tipo, o compilador dá uma ajuda, fazendo a

Equivalente a:

```
lista.add(Integer.valueOf(1));
```

```
    List<Integer> lista = new ArrayList<Integer> ();

    lista.add(1);
    lista.add(2);
    lista.add(42);
    lista.add(1000);

    for (int i : lista);
    {
        System.out.println("i² = " + (i * i) );
    }

    Map<String, Boolean> m = new HashMap Map<String, Boolean>();

    m.put("A", true)
    m.put("B", true)
    m.put("C", false)

    boolean b = m.get("C");
}
```

COLEÇÕES E TIPOS SIMPLES

- Quando se for adicionar, ou simplesmente acessar, objetos em coleções deste tipo, o compilador dá uma ajuda, fazendo a conversão automaticamente :

```
public void teste()  
{  
    List<Integer> lista = new ArrayList<Integer> ();
```

Equivalente a:

```
System.out.println("i2 = " + (i.intValue() * i.intValue()) );
```

```
for (int i : lista);  
{  
    System.out.println("i2 = " + (i * i) );  
}
```

```
Map<String, Boolean> m = new HashMap Map<String, Boolean>();
```

```
m.put("A", true)  
m.put("B", true)  
m.put("C", false)
```

```
boolean b = m.get("C");
```

```
}
```

COLEÇÕES E TIPOS SIMPLES

- Quando se for adicionar, ou simplesmente acessar, objetos em coleções deste tipo, o compilador dá uma ajuda, fazendo a conversão automaticamente :

```
public void teste()  
{  
    List<Integer> lista = new ArrayList<Integer> ();  
  
    lista.add(1);  
    lista.add(2);  
    lista.add(42);  
    lista.add(1000);  
}
```

Equivalente a:

```
m.put("A", Boolean.valueOf(true));
```

```
Map<String, Boolean> m = new HashMap<String, Boolean>();
```

```
m.put("A", true)  
m.put("B", true)  
m.put("C", false)
```

```
boolean b = m.get("C");
```

```
}
```


COLEÇÕES E TIPOS SIMPLES

- Quando se for adicionar, ou simplesmente acessar, objetos em coleções deste tipo, o compilador dá uma ajuda, fazendo a conversão automaticamente :

```
public void teste()
{
    List<Integer> lista = new ArrayList<Integer> ();

    lista.add(1);
    lista.add(2);
    lista.add(42);
    lista.add(1000);

    for (int i : lista);
    {
        System.out.println("i2 = " + (i * i));
    }
}
```

Equivalente a:

```
boolean b = m.get("C").booleanValue();
```

```
m.put("A", true);
m.put("B", true);
m.put("C", false);
```

```
boolean b = m.get("C");
```

```
}
```

INTRODUÇÃO AO SWING

- Swing é a biblioteca de componentes de interface gráfica de usuário do Java
- É uma evolução do AWT, que usava componentes nativos do sistema
 - Componentes do Swing são desenhados pela própria biblioteca, e podem ter seu *Look and Feel* modificado.
- Um tutorial completo de programação Swing existe em:

<http://java.sun.com/docs/books/tutorial/uiswing/index.html>

EXEMPLO SWING

```
class TesteSwing
{
    public static void main(String[] args)
    {
        JFrame janela = new JFrame("Teste do Swing");
        janela.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        JButton btn = new JButton("Aperte-me");
        btn.addActionListener(new MostraMensagem());

        janela.setLayout( new FlowLayout() );
        janela.add(btn);

        janela.pack();
        janela.setVisible(true);
    }
}

class MostraMensagem implements ActionListener
{
    public void actionPerformed(ActionEvent e)
    {
        JOptionPane.showMessageDialog(null, "Olá Mundo");
    }
}
```

EXEMPLO SWING

```
class TesteSwing
{
    public static void main(String[] args)
    {
        JFrame janela = new JFrame("Teste do Swing");
        janela.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

Cria uma janela, o texto de parâmetro no construtor será seu título.

```
        janela.pack();
        janela.setVisible(true);
    }
}

class MostraMensagem implements ActionListener
{
    public void actionPerformed(ActionEvent e)
    {
        JOptionPane.showMessageDialog(null, "Olá Mundo");
    }
}
```

EXEMPLO SWING

```
class TesteSwing
{
    public static void main(String[] args)
    {
        JFrame janela = new JFrame("Teste do Swing");
        janela.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        JButton btn = new JButton("Aperte-me");
        btn.addActionListener(new MostraMensagem());
    }
}
```

Configura a janela para que, quando ela for fechada, o programa seja terminado.

```
janela.pack();
janela.setVisible(true);
}

class MostraMensagem implements ActionListener
{
    public void actionPerformed(ActionEvent e)
    {
        JOptionPane.showMessageDialog(null, "Olá Mundo");
    }
}
```

EXEMPLO SWING

```
class TesteSwing
{
    public static void main(String[] args)
    {
        JFrame janela = new JFrame("Teste do Swing");
        janela.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        JButton btn = new JButton("Aperte-me");
        btn.addActionListener(new MostraMensagem());

        janela.setLayout(new FlowLayout());
    }
}
```

Cria um botão. Assim como para a janela, o texto do construtor será seu título.

```
class MostraMensagem implements ActionListener
{
    public void actionPerformed(ActionEvent e)
    {
        JOptionPane.showMessageDialog(null, "Olá Mundo");
    }
}
```

EXEMPLO SWING

```
class TesteSwing
{
    public static void main(String[] args)
    {
        JFrame janela = new JFrame("Teste do Swing");
        janela.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        JButton btn = new JButton("Aperte-me");
        btn.addActionListener(new MostraMensagem());

        janela.setLayout( new FlowLayout() );
    }
}
```

Adiciona um *ActionListener* ao botão. Sempre que o botão for pressionado, o método *actionPerformed* deste objeto será invocado.

```
class MostraMensagem implements ActionListener
{
    public void actionPerformed(ActionEvent e)
    {
        JOptionPane.showMessageDialog(null, "Olá Mundo");
    }
}
```

EXEMPLO SWING

```
class TesteSwing
{
    public static void main(String[] args)
    {
        JFrame janela = new JFrame("Teste do Swing");
        janela.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        JButton btn = new JButton("Aperte-me");
        btn.addActionListener(new MostraMensagem());

        janela.setLayout( new FlowLayout() );
        janela.add(btn);
    }
}
```

Configura o *layout* da janela, isto é, a forma como os componentes filhos serão distribuídos.

Com o *FlowLayout*, os componentes se distribuem como palavras em um editor de texto.

Obs: Para versões do Java anteriores à 5.0, deve-se usar:

```
janela.getContentPane().setLayout( ... );
```


EXEMPLO SWING

```
class TesteSwing
{
    public static void main(String[] args)
    {
        JFrame janela = new JFrame("Teste do Swing");
        janela.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        JButton btn = new JButton("Aperte-me");
        btn.addActionListener(new MostraMensagem());

        janela.setLayout( new FlowLayout() );
        janela.add(btn);

        janela.pack();
    }
}
```

Adiciona o botão à janela.

Obs: Para versões do Java anteriores à 5.0,
deve-se usar:

```
janela.getContentPane().add( ... );
```

EXEMPLO SWING

Ajusta o tamanho da janela
a seus componentes

```
class TesteSwing
{
    public static void main(String[] args)
    {
        JFrame janela = new JFrame("Teste do Swing");
        janela.setDefaultCloseOperation(WindowType.EXIT_ON_CLOSE);
        JButton btn = new JButton("Clique aqui");
        janela.add(btn);

        janela.setLayout(new FlowLayout());
        janela.add(btn);

        janela.pack();
        janela.setVisible(true);
    }
}

class MostraMensagem implements ActionListener
{
    public void actionPerformed(ActionEvent e)
    {
        JOptionPane.showMessageDialog(null, "Olá Mundo");
    }
}
```

EXEMPLO SWING

Mostra a janela.

Quando o primeiro componente gráfico de um programa é mostrado, é iniciada uma *thread* para tratar os eventos.

```
janela.setLayout( new FlowLayout() );
janela.add(btn);

janela.pack();
janela.setVisible(true);
}

}

class MostraMensagem implements ActionListener
{
    public void actionPerformed(ActionEvent e)
    {
        JOptionPane.showMessageDialog(null, "Olá Mundo");
    }
}
```

EXEMPLO SWING

```
class TesteSwing
{
    public static void main(String[] args)
    {
        JFrame janela = new JFrame("Teste do Swing");
        janela.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        JButton btn = new JButton("Aperte-me");
        btn.addActionListener(new MostraMensagem());

        janela.setLayout( new FlowLayout() );
        janela.add(btn);

        janela.pack();
    }
}
```

Mostra uma caixa de mensagem.

```
class MostraMensagem implements ActionListener
{
    public void actionPerformed(ActionEvent e)
    {
        JOptionPane.showMessageDialog(null, "Olá Mundo");
    }
}
```

RESULTADO



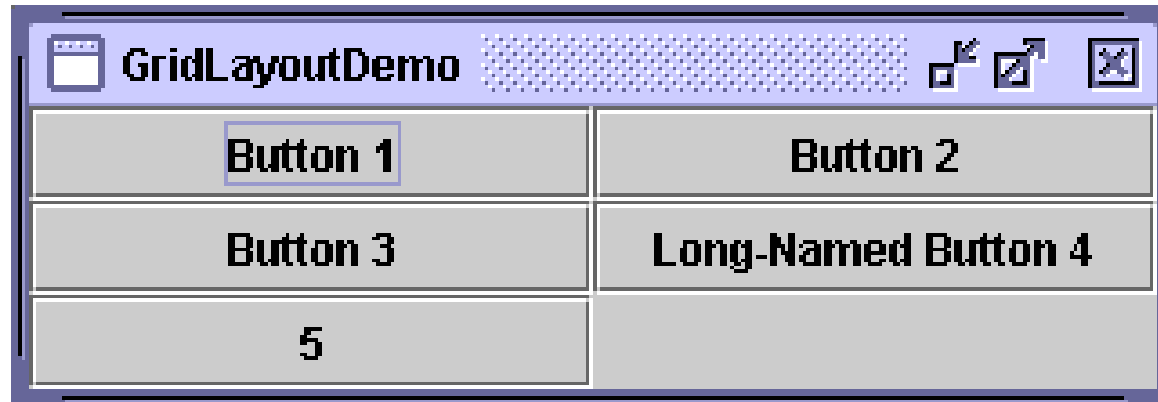
LAYOUTS: FLOWLAYOUT



- Agrupa os componentes lado a lado, em uma linha, respeitando as dimensões padrão de cada um deles.
- Se não houver espaço suficiente, novas linhas são criadas.



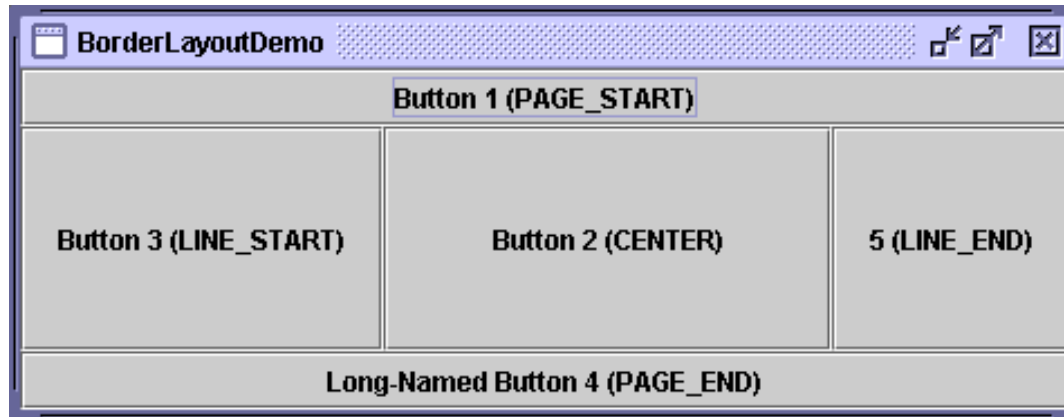
LAYOUTS: GridLayout



- Agrupa os componentes na forma de uma tabela, com cada um deles ocupando todo o espaço disponível na célula.
- O número de linhas e colunas da tabela é definido por parâmetros no construtor do *layout*.



LAYOUTS: BORDERLAYOUT



- Divide o componente pai em cinco áreas: PAGE_START, PAGE_END, LINE_START, LINE_END, e CENTER.
- Ao se adicionar um componente, deve-se indicar qual área ele ocupará.

```
btn = new JButton("Button 3 (LINE_START)");  
janela.add(btn, BorderLayout.LINE_START);
```


CLASSES ANÔNIMAS

PARA TRATAR EVENTOS

- Para facilitar o tratamento de eventos, é muito comum o uso de classes anônimas:

```
btn.addActionListener( new ActionListener() {  
    public void actionPerformed(ActionEvent e) {  
        JOptionPane.showMessageDialog(null, "Olá Mundo");  
    } } );
```

CLASSES ANÔNIMAS PARA TRATAR EVENTOS

- Para facilitar o tratamento de eventos, é muito comum usar classes anônimas:

A região destacada cria uma instância de uma classe que implementa a interface *ActionListener*.

```
btn.addActionListener( new ActionListener() {  
    public void actionPerformed(ActionEvent e) {  
        JOptionPane.showMessageDialog(null, "Olá Mundo");  
    } } );
```

USO DE MODELOS DE DADOS

- A maioria dos componentes do Swing permite que seus dados venham de classes separadas, conhecidas como Modelos.
- A seguir será mostrado um exemplo de uma tabela para apresentar uma lista de pessoas.

```
class Pessoa
{
    private String nome;
    private int idade;
    private boolean brasileiro;

    ...
}
```

TABLEMODEL

```
class PessoaTableModel extends AbstractTableModel
{
    private List<Pessoa> pessoas = new ArrayList<Pessoa>();

    public void setPessoas(List<Pessoa> pessoas)
    {
        this.pessoas.clear();
        this.pessoas.addAll(pessoas);
        fireTableDataChanged();
    }

    public int getRowCount() { return pessoas.size(); }

    public int getColumnCount() { return 3; }

    public Object getValueAt(int rowIndex, int columnIndex)
    {
        Pessoa p = pessoas.get(rowIndex);

        switch (columnIndex) {
            case 0: return p.getNome();
            case 1: return p.getIdade();
            case 2: return p.isBrasileiro();
            default: return null;
        }
    }
}

...
```

Método herdado de AbstractTableModel

Avisa à tabela que os dados foram alterados

```
private List<Pessoa> pessoas = new ArrayList<Pessoa>();
```

```
public void setPessoas(List<Pessoa> pessoas)
{
    this.pessoas.clear();
    this.pessoas.addAll(pessoas);
    fireTableDataChanged();
}
```

```
public int getRowCount() { return pessoas.size(); }
```

```
public int getColumnCount() { return 3; }
```

```
public Object getValueAt(int rowIndex, int columnIndex)
{
    Pessoa p = pessoas.get(rowIndex);

    switch (columnIndex) {
        case 0: return p.getNome();
        case 1: return p.getIdade();
        case 2: return p.isBrasileiro();
        default: return null;
    }
}
```

...

TableModel (CONT.)

```
...

public String getColumnName(int columnIndex)
{
    switch (columnIndex)
    {
        case 0: return "Nome";
        case 1: return "Idade";
        case 2: return "Brasileiro";
        default: return null;
    }
}

public Class getColumnClass(int columnIndex)
{
    switch (columnIndex)
    {
        case 0: return String.class;
        case 1: return Integer.class;
        case 2: return Boolean.class;
        default: return null;
    }
}
}
```

CRIANDO A TABELA

```
JFrame janela = new JFrame("Teste Tabela");
janela.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
janela.setLayout(new BorderLayout());

List<Pessoa> pessoas = new ArrayList<Pessoa>();
pessoas.add(new Pessoa("Carlos", 25, true));
pessoas.add(new Pessoa("Juliana", 18, true));
pessoas.add(new Pessoa("John", 40, false));

PessoaTableModel ptm = new PessoaTableModel();
ptm.setPessoas(pessoas);

JTable tabela = new JTable();
Tabela.setModel(ptm);

janela.add(new JScrollPane(tabela), BorderLayout.CENTER);

janela.pack();

janela.setVisible(true);
```

CRIANDO A TABELA

Modelo lógico

```
JFrame janela = new JFrame("Teste Tabela");
janela.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
janela.setLayout(new BorderLayout());

Pessoa> pessoas = new ArrayList<Pessoa>();
pessoas.add(new Pessoa("Carlos", 25, true));
pessoas.add(new Pessoa("Juliana", 18, true));
pessoas.add(new Pessoa("John", 40, false));

PessoaTableModel ptm = new PessoaTableModel();
ptm.setPessoas(pessoas);

JTable tabela = new JTable();
Tabela.setModel(ptm);

janela.add(new JScrollPane(tabela), BorderLayout.CENTER);

janela.pack();

janela.setVisible(true);
```


CRIANDO A TABELA

Componente Gráfico

```
JFrame janela = new JFrame("Teste Tabela");
janela.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
janela.setLayout(new BorderLayout());

List<Pessoa> pessoas = new ArrayList<Pessoa>();
pessoas.add(new Pessoa("Carlos", 25, true));
pessoas.add(new Pessoa("Juliana", 18, true));
pessoas.add(new Pessoa("John", 40, false));

PessoaTableModel ptm = new PessoaTableModel();
ptm.setPessoas(pessoas);

JTable tabela = new JTable();
Tabela.setModel(ptm);

janela.add(new JScrollPane(tabela), BorderLayout.CENTER);

janela.pack();

janela.setVisible(true);
```

CRIANDO A TABELA

```
JFrame janela = new JFrame("Teste Tabela");
janela.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
janela.setLayout(new BorderLayout());

List<Pessoa> pessoas = new ArrayList<Pessoa>();
pessoas.add(new Pessoa("Carlos", 25, true));
pessoas.add(new Pessoa("Juliana", 18, true));
pessoas.add(new Pessoa("John", 40, false));

PessoaTableModel ptm = new PessoaTableModel();
ptm.setPessoas(pessoas);

JTable tabela = new JTable();
Tabela.setModel(ptm);

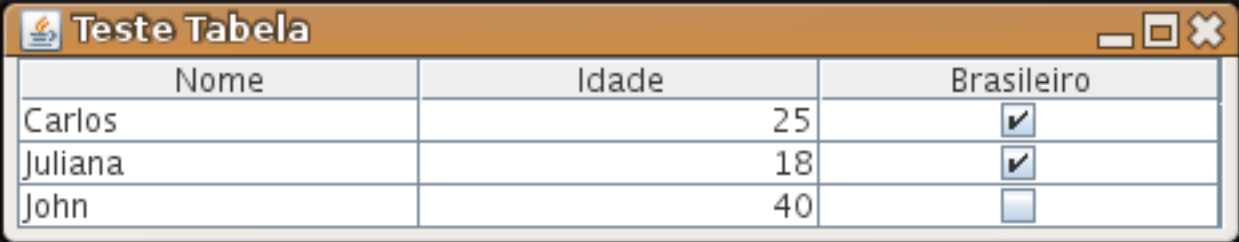
janela.add(new JScrollPane(tabela), BorderLayout.CENTER);

janela.pack();

janela.setVisible(true);
```

Envolve a tabela em uma
caixa com barras de rolagem

RESULTADO



Nome	Idade	Brasileiro
Carlos	25	<input checked="" type="checkbox"/>
Juliana	18	<input checked="" type="checkbox"/>
John	40	<input type="checkbox"/>

FIM

