



# Wrapper Classes



# Wrapper Classes



- São classes que “**empacotam**” um tipo primitivo
- Permitem manipular **variáveis de tipos primitivos** como se fossem **objetos** de uma classe
  - Importante para poder utilizar diversos métodos em java que aceitam apenas objetos como parâmetros (ou seja, não aceitam tipos primitivos)
- Também possuem uma série de **métodos utilitários** para manipular os seus respectivos tipos primitivos

# Wrapper Classes

- Java possui as seguintes *wrapper classes*:

<b><i>Wrapper Class</i></b>	<b>Tipo Primitivo</b>
<i>Integer</i>	<i>int</i>
<i>Short</i>	<i>short</i>
<i>Long</i>	<i>long</i>
<i>Byte</i>	<i>byte</i>
<i>Float</i>	<i>float</i>
<i>Double</i>	<i>double</i>
<i>Character</i>	<i>char</i>
<i>Boolean</i>	<i>boolean</i>

# São Classes Normais

Arquivo Integer.java do código do Java

Veremos todos os modificadores futuramente

```
public final class Integer extends Number
    implements Comparable<Integer>, Constable, ConstantDesc {

    // Código da classe continua ...

    /**
     * The value of the {@code Integer}.
     */
    private final int value;

    // Código da classe continua ...

    /**
     * Parses the string argument as a signed decimal integer...
     */
    public static int parseInt(String s) throws NumberFormatException {
        return parseInt(s,10);
    }

    // Código da classe continua ...

}
```

O atributo `value` armazena o inteiro “empacotado”. É uma constante (`final`).

O método `parseInt` faz o parse de uma `String` e retorna um `int`

# Conversões Boxing

- Ao invés de instanciar objetos usando o `new`, podemos simplesmente **atribuí-los a valores**

- Semelhante ao que foi feito no caso das `Strings`

```
Integer numAlunos = 20;  
Float peso = 5.6f;
```

Note como o número 20 (tipo primitivo `int`) está sendo atribuído a `numAlunos`, um objeto da classe `Integer`

- O compilador Java **detecta e converte** entre tipos primitivos e *wrapper classes* nos lugares necessários
- Isso só acontece nas *wrapper classes* e na classe `String`.

# Conversões Boxing

## ■ **Autoboxing** (ou simplesmente *boxing*)

- **Converte** um **valor primitivo** para um **objeto** da classe correspondente

```
Integer numAlunos = 20;  
Float peso = 5.6f;
```

## ■ **Unboxing**

- **Converte um objeto** de uma *wrapper class* para o seu **tipo primitivo**
- O código abaixo é só um exemplo, definitivamente não recomendado

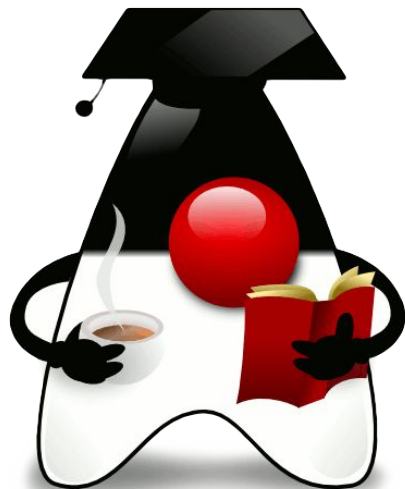
```
int numAlunos = new Integer(30);
```

# Métodos Utilitários

- As *wrapper classes* possuem uma série de **métodos estáticos utilitários** para manipular os seus respectivos tipos primitivos
  - Alguns exemplos de métodos:

Integer
<code>int parseInt(String s)</code>
<code>int max(int a, int b)</code>
<code>int min(int a, int b)</code>
<code>String toHexString(int i)</code>
<code>String toString()</code>

Double
<code>double parseDouble(String s)</code>
<code>double max(double a, double b)</code>
<code>double min(double a, double b)</code>
<code>String toHexString(double d)</code>
<code>String toString()</code>



# JavaDoc





# JavaDoc



- JavaDoc permite incluir a **documentação** do seu sistema diretamente no **código-fonte**
- Um aplicativo, chamado `javadoc`, lê os arquivos Java do sistema e **gera** uma **documentação completa** do mesmo
  - Em geral, a documentação é no formato HTML
- Comentários JavaDoc:
  - O compilador java ignora os comentários de documentação, da mesma forma que ignora os comentários normais

# Geração da Documentação

- Uma vez que o código esteja documentado, executa-se o comando `javadoc` para gerar a documentação

```
$ javadoc -charset utf-8 Livro.java
```

- **No eclipse**, a documentação do sistema todo pode ser gerada
  - Project → Generate Javadoc

# JavaDoc

## Exemplo:

Só são incluídos na documentação, por padrão, os campos marcados como `public`. Falaremos disso em outra aula.

```
/**
 * Classe Livro - Representa um livro na aplicação
 * @author Horacio Fernandes <horacio.fernandes@gmail.com>
 * @version 1.20, 2015-10-21
 */
```

```
public class Livro {
    /** Nome do autor */
    String autor;
    String nome, editora;
    int anoPublicacao;
```

Documentação da classe

Documentação do atributo

```
/**
 * Construtor da classe.
 * @param autor autor do livro
 * @param nome nome do autor do livro
 * @param editora editora do livro
 * @param anoPublicacao ano de publicação
 */
```

Documentação do construtor

```
public Livro(String autor, String nome, String editora, int anoPublicacao) {
    this.autor = autor;
    this.nome = nome;
    this.editora = editora;
    this.anoPublicacao = anoPublicacao;
}
```

Documentação do método

```
/**
 * Pega o autor do livro
 * @return String autor do livro
 */
```

```
public String getAutor() {
    return autor;
}
// Continuação da classe ..
```

```
}
```

# JavaDoc

Exemplo:



Livro

file:///tmp/Livro.html

Package **Class** Use Tree Deprecated Index Help

Prev Class Next Class Frames No Frames All Classes

Summary: Nested | Field | Constr | Method Detail: Field | Constr | Method

## Class Livro

java.lang.Object  
Livro

```
public class Livro
extends java.lang.Object
```

Classe Livro - Representa um livro na aplicação

**Version:**  
1.20, 2015-10-21

**Author:**  
Horacio Fernandes <horacio.fernandes@gmail.com>

### Constructor Summary

Constructors
Constructor and Description
Livro(java.lang.String autor, java.lang.String nome, java.lang.String editora, int anoPublicacao)
Construtor da classe.

### Method Summary

Methods
---------

# JavaDoc

Exemplo:



Livro x

file:///tmp/Livro.html

## Method Summary

Methods

Modifier and Type	Method and Description
java.lang.String	<b>getAutor()</b> Pega o autor do livro

### Methods inherited from class java.lang.Object

equals, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

## Constructor Detail

### Livro

```
public Livro(java.lang.String autor,
             java.lang.String nome,
             java.lang.String editora,
             int anoPublicacao)
```

Construtor da classe.

**Parameters:**

- autor - autor do livro
- nome - nome do autor do livro
- editora - editora do livro
- anoPublicacao - ano de publicação

Visão geral dos métodos

Documentação dos construtores

# JavaDoc

Exemplo:



The screenshot shows a web browser window with the address bar displaying `file:///tmp/Livro.html`. The page content is organized into sections:

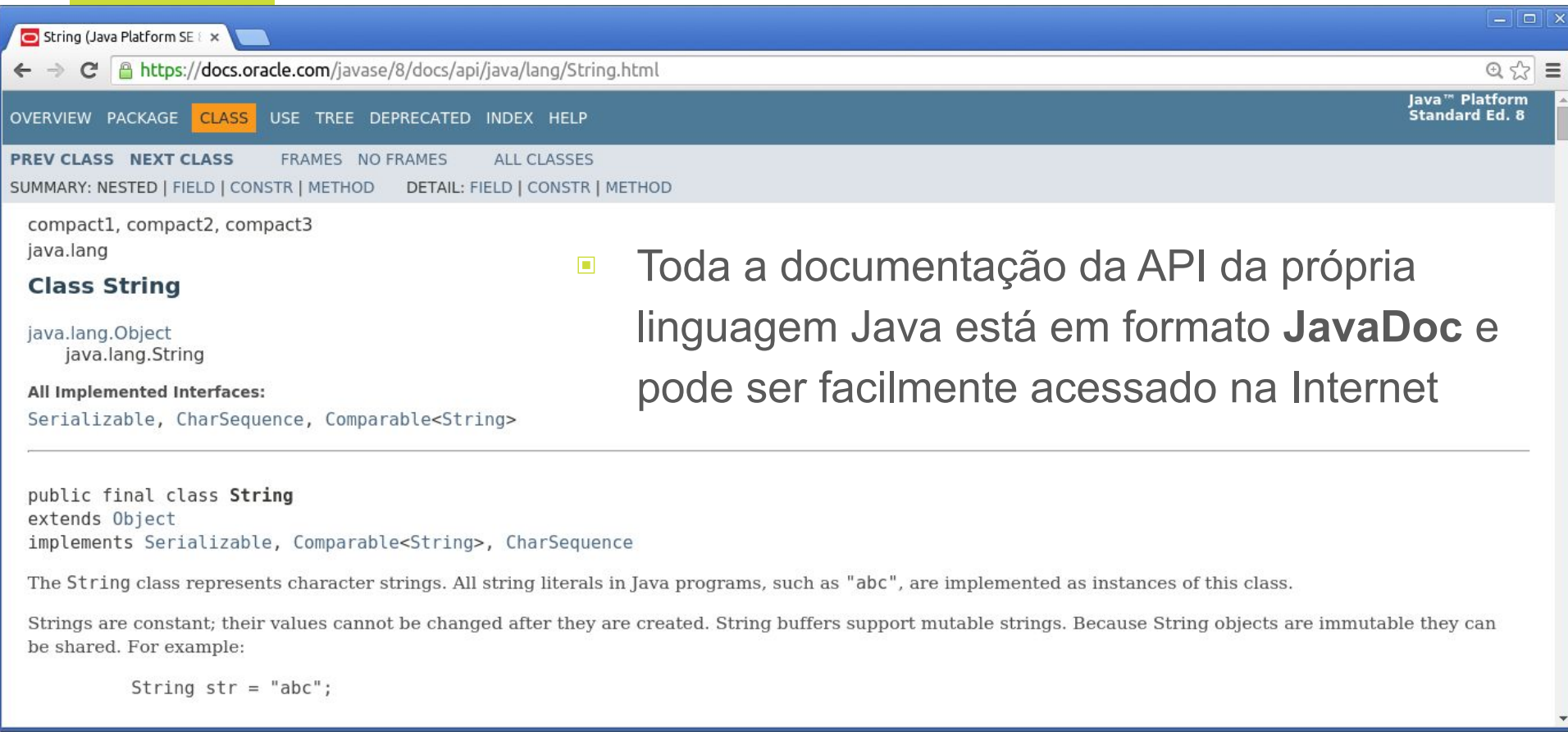
- Source Code:** A code block at the top shows the constructor signature:

```
java.lang.String nome,  
java.lang.String editora,  
int anoPublicacao)
```
- Constructor:** A section titled "Construtor da classe." followed by a "Parameters:" list:
  - autor - autor do livro
  - nome - nome do autor do livro
  - editora - editora do livro
  - anoPublicacao - ano de publicação
- Method Detail:** A section titled "Method Detail" containing a box for the `getAutor` method:
  - Signature:** `public java.lang.String getAutor()`
  - Description:** "Pega o autor do livro"
  - Returns:** "String autor do livro"

A blue callout box with the text "Documentação dos métodos" has a line pointing to the "Method Detail" section.

The bottom of the browser window features a navigation bar with tabs: "Package", "Class" (selected), "Use", "Tree", "Deprecated", "Index", and "Help". Below this is a secondary navigation bar with links: "Prev Class", "Next Class", "Frames", "No Frames", and "All Classes". The footer contains a summary of the page structure: "Summary: Nested | Field | Constr | Method" and "Detail: Field | Constr | Method".

# Documentação do Java



The screenshot shows the Oracle Java Platform SE 8 API documentation for the `String` class. The browser address bar shows the URL `https://docs.oracle.com/javase/8/docs/api/java/lang/String.html`. The page has a navigation bar with tabs for OVERVIEW, PACKAGE, CLASS (selected), USE, TREE, DEPRECATED, INDEX, and HELP. Below the navigation bar, there are links for PREVIOUS CLASS, NEXT CLASS, FRAMES, NO FRAMES, and ALL CLASSES. The main content area displays the class hierarchy: `compact1, compact2, compact3`, `java.lang`, `java.lang.Object`, and `java.lang.String`. It also lists the implemented interfaces: `Serializable, CharSequence, Comparable<String>`. The class definition is shown as `public final class String` extending `Object` and implementing `Serializable, Comparable<String>, CharSequence`. A description states that the `String` class represents character strings and that all string literals in Java programs are implemented as instances of this class. It also notes that strings are constant and cannot be changed after creation. An example code snippet is provided: `String str = "abc";`.

String (Java Platform SE 8)

[←](#) [→](#) [↺](#) [https://docs.oracle.com/javase/8/docs/api/java/lang/String.html](#) [🔍](#) [☆](#) [☰](#)

OVERVIEW PACKAGE **CLASS** USE TREE DEPRECATED INDEX HELP

PREV CLASS NEXT CLASS FRAMES NO FRAMES ALL CLASSES

SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

compact1, compact2, compact3  
java.lang

**Class String**

java.lang.Object  
java.lang.String

**All Implemented Interfaces:**  
Serializable, CharSequence, Comparable<String>

---

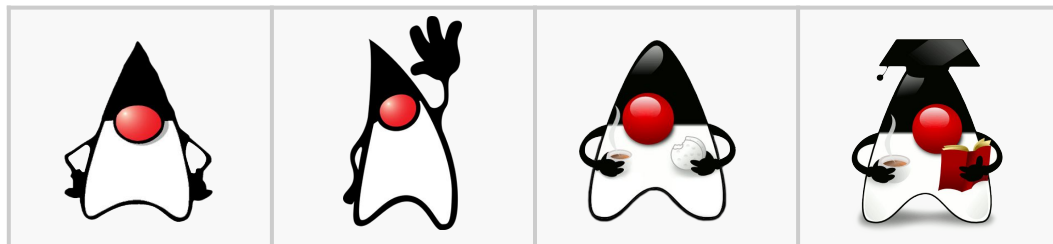
public final class **String**  
extends Object  
implements Serializable, Comparable<String>, CharSequence

The String class represents character strings. All string literals in Java programs, such as "abc", are implemented as instances of this class.

Strings are constant; their values cannot be changed after they are created. String buffers support mutable strings. Because String objects are immutable they can be shared. For example:

```
String str = "abc";
```

# Vetores





# Vetores São Objetos

- Em Java, **vetores são objetos**
  - Criados **dinamicamente** e **alocados em tempo de execução**
- Armazenam dados do **mesmo tipo** (homogêneo)
  - Que pode ser de um tipo **primitivo** (int, float, etc)
  - Ou de um tipo **referência**/classe (String, Circulo, etc)
- Entretanto, não existe uma classe específica para vetores
  - Um “tipo” da classe vetor é referenciado pelo tipo de dado que o vetor armazena, seguido dos colchetes []
  - A classe `Vector`, do Java, não está relacionada com esses vetores
    - *Essa classe é uma estrutura de dados que armazena informações usando vetores. Veremos essa e outras estruturas em breve.*

# Declaração

## ■ Declaração de Vetores

- Usa-se **colchetes**, semelhante a C
- Colchetes podem vir depois do tipo (recomendado) ou depois do nome da variável
- Declarar um vetor não reserva espaço na memória para ele
  - *Isso é feito apenas na hora da instanciação*

```
int[]      matriculas; // Vetor de inteiros
int        aulas[];   // Vetor de inteiros
float[][]  notas;      // Vetor de vetor de floats (matriz)
String     args[];     // Vetor de objetos da classe String
Circulo[]  circulos;   // Vetor de objetos da classe Circulo
```

# Instanciação

## ■ Instanciação de Vetores

- Usa-se o `new` para alocar memória para o vetor
- Na instanciação, o **tamanho do vetor é definido**
  - *Uma vez definido, o tamanho não pode ser modificado*
  - *Não existe a função `realloc`, como em C*

```
matriculas = new int[42];  
circulos   = new Circulo[3];  
  
// Declarando e instanciando  
Circulo[] maisCirculos = new Circulo[14];
```

# Instanciação

- ▣ Vetores instanciados são automaticamente **inicializados** para **zero**
  - ▣ Ou `null`, se for um vetor de objetos

```
int[] matriculas = new int[3];  
String[] nomes = new String[3];  
  
System.out.println(matriculas[1]);  
System.out.println(nomes[1]);
```

```
0  
null
```

# Declaração com Inicialização

- Pode-se declarar vetores já **atribuindo** seus elementos:
  - Continuam sendo objetos alocados dinamicamente, apesar de não ter o `new`

```
int[] factorial = { 1, 1, 2, 6, 24, 120, 720, 5040 };  
char ac[] = { 'g', 'e', 't', 'i', 'a', 'g' };  
String[] aas = { "array", "of", "String", };
```

# Acessando Dados do Vetor

- Acessando os dados do vetor
  - Igual à linguagem C
  - O **índice** começa em 0 (zero) e vai até o tamanho do vetor – 1

```
matriculas[0] = 24601;  
circulos[2] = new Circulo();  
  
System.out.println(matriculas[0]);  
System.out.println(circulos[2].raio);
```

```
24601  
0.0
```

# Acessando Dados do Vetor

- Acessar um elemento **fora dos limites** de um vetor resulta em um **erro** em tempo de execução
  - Mais especificamente, uma *exceção*, como veremos futuramente

```
matriculas[200] = 31337;
```

```
Exception in thread "main"  
java.lang.ArrayIndexOutOfBoundsException: Index 200  
out of bounds for length 42  
    at Principal.main(Principal.java:64)
```

Índice que gerou  
a exceção

Arquivo e linha  
que gerou a  
exceção

# Tamanho de um Vetor

- Todo vetor (que é um objeto) possui um atributo chamado `length` que armazena o **tamanho máximo** do vetor
  - Nota: é o tamanho máximo do vetor e não a “quantidade” de elementos armazenados/atribuídos
  - Portanto, diferentemente de C, não precisamos criar uma constante para armazenar o tamanho máximo do vetor

```
Circulo[] circulos = new Circulo[4];  
System.out.println(circulos.length);
```

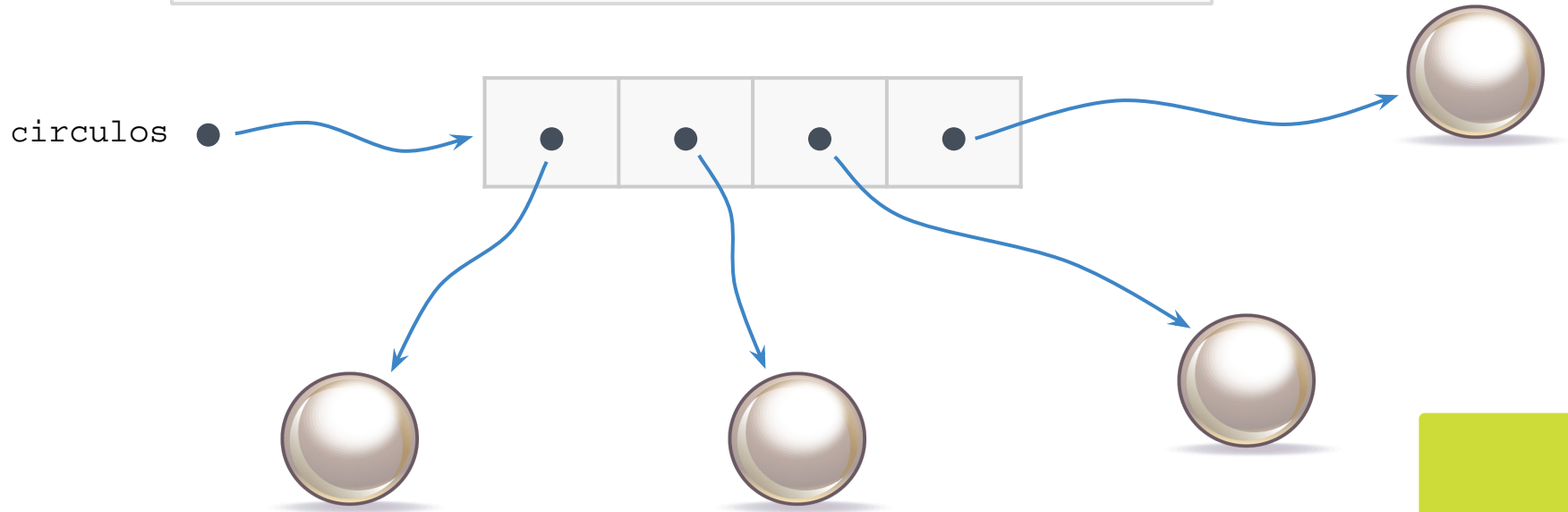
4



# Vetores de Objetos

- ▣ Vetores de objetos **armazenam referências** para os objetos

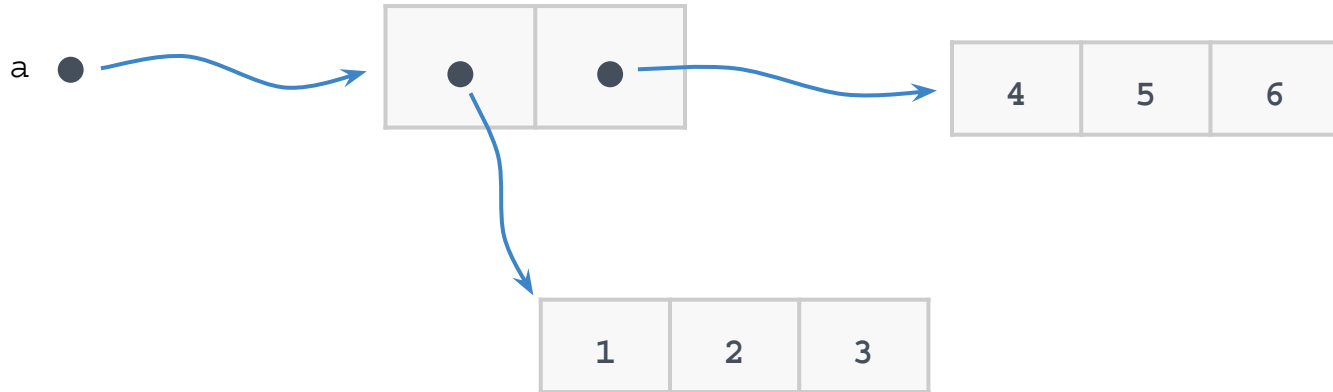
```
Circulo[] circulos = new Circulo[4];
```



# Matrizes

- Matrizes são **vetores de vetores**

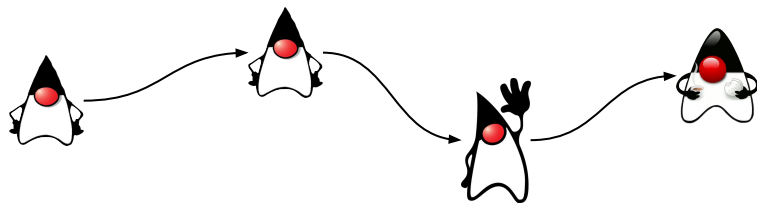
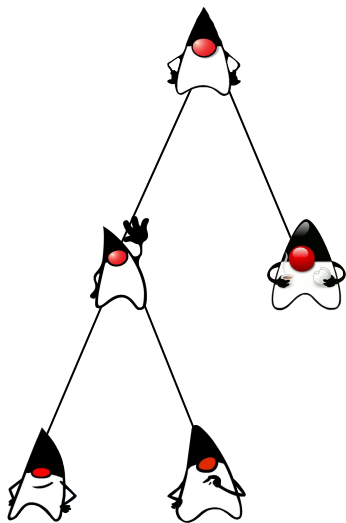
```
int[] [] a = { {1,2,3}, {4,5,6} };
```



# Armazenar Valores na Prática



- Java possui outras classes **mais práticas** para armazenar valores
- Tais classes implementam estruturas de dados
  - Listas com Vetores (`Vector`, `ArrayList`)
  - Listas Encadeadas (`LinkedList`)
  - Tabelas Hash (`HashTable`)
  - Dentre outros
- Estas classes serão vistas a seguir



# Generic Collections



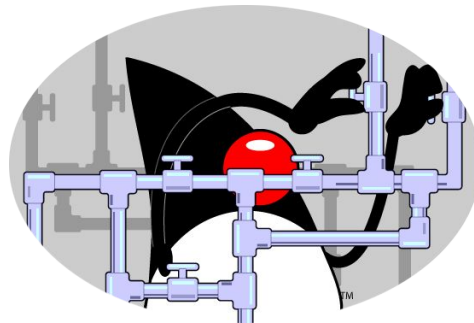
# Generic Collections



- *Generic Collections (Java Collections Framework)*
  - Java possui uma série de implementações de estruturas de dados prontas para serem utilizadas
- Em java, um *collection* é uma estrutura de dados que armazena referências para objetos
  - Elas usam “Classes Genéricas”
  - Permitem definir o “tipo exato” de dado armazenado na hora da declaração
  - Permitem verificações de tipo em tempo de compilação
  - Classes genéricas é um assunto um pouco mais complexo, que não será tratado no curso, mas mostraremos como utilizá-las

# Código-Fonte

- Tais coleções são implementadas em Java e seus códigos-fonte podem ser acessado para ver as implementações
  - Todas as classes mencionadas adiante podem ser encontradas no diretório “util” do código-fonte do Java, disponível no material da disciplina
  - Acesse os códigos-fonte dessas classes, e compare com os feitos em AED1/AED2.
  - Você irá perceber que não há muitas diferenças!



# Classes ArrayList e Vector

- As duas classes implementam **listas, usando vetores**
  - A principal diferença é que `ArrayList` não se preocupa com threads
  - Sendo, portanto, mais eficiente para os programas sem paralelismo
- Apesar de poder ter um tamanho inicial, este tamanho é aumentado automaticamente quando necessário (lista de tamanho variável).
  - Nota: por ser internamente implementado usando vetores (que não aumentam de tamanho), aumentar o tamanho da lista tem um custo grande, pois um novo vetor é criado e o conteúdo do anterior é copiado para o atual.
  - Você pode ver o código-fonte para observar isso

# Classes ArrayList e Vector

## ■ Principais métodos

<code>int size()</code>	Retorna o tamanho da lista (qtde. de elementos inseridos)
<code>boolean add(E e)</code>	Adiciona um elemento no final da lista
<code>void add(int index, E element)</code>	Insere o elemento na posição especificada
<code>int indexOf(Object o)</code>	Busca o elemento (usando <code>equals</code> ), retorna seu índice
<code>E remove(int index)</code>	Remove um elemento pelo seu índice
<code>boolean remove(Object o)</code>	Remove um elemento pelo valor (usando <code>equals</code> )
<code>Iterator&lt;E&gt; iterator()</code>	Retorna um objeto <code>iterator</code> que permite caminhar sequencialmente na lista



# Classes ArrayList e Vector

## Exemplo

```
import java.util.*;

public class ListaJava {
    public static void main(String args[]) {

        ArrayList<String> mestres = new ArrayList<String>();
        mestres.add("Obi-Wan Kenobi");
        mestres.add("Qui-Gon Jinn");
        mestres.add("Yoda");

        Iterator<String> iterator = mestres.iterator();
        while (iterator.hasNext()) {
            String mestreAtual = iterator.next();
            System.out.println(mestreAtual);
        }
    }
}
```

```
$ javac ListaJava.java
$ java ListaJava
Obi-Wan Kenobi
Qui-Gon Jinn
Yoda
```

# Classe LinkedList

- Implementa uma **lista duplamente encadeada**
- Principais métodos:
  - Todos mostrados no `ArrayList` com alguns métodos a mais:

<code>E getFirst()</code>	Retorna o primeiro elemento da lista
<code>E getLast()</code>	Retorna o último elemento da lista
<code>void addFirst(E e)</code>	Insere um elemento no início da lista
<code>void addLast(E e)</code>	Insere um elemento no final da lista
<code>E removeFirst()</code>	Remove o primeiro elemento da lista. Retorna o elemento.
<code>E removeLast()</code>	Remove o último elemento da lista. Retorna o elemento.

# Classe LinkedList

- Analisando o código-fonte (LinkedList.java)
  - Internamente, um “nó” da lista encadeada é um objeto da classe Node

```
// (...)
```

```
private static class Node<E> {
```

```
    E item;
```

```
    Node<E> next;
```

```
    Node<E> prev;
```

```
    Node(Node<E> prev, E element, Node<E> next) {
```

```
        this.item = element;
```

```
        this.next = next;
```

```
        this.prev = prev;
```

```
    }
```

```
}
```

```
// (...)
```

Elemento sendo  
armazenado no nó

Referência para o  
próximo nó/elemento

Referência para o nó  
anterior

Construtor do nó

# Classe LinkedList

- Analisando o código-fonte (LinkedList.java)
  - Método para inserir no início da lista (addFirst)

```
public void addFirst(E e) {  
    linkFirst(e);  
}  
  
private void linkFirst(E e) {  
    final Node<E> f = first;  
    final Node<E> newNode = new Node<>(null, e, f);  
    first = newNode;  
    if (f == null)  
        last = newNode;  
    else  
        f.prev = newNode;  
    size++;  
    modCount++;  
}
```

Chama o método linkFirst

Salva a referência para o topo da lista (atributo first)

Cria o novo nó da lista

Seta o nó como topo da lista

Seta o anterior do antigo topo para apontar para o novo nó

# Classe Stack

- Implementa uma **pilha**, usando vetor
  - Ele herda (incrementa) a classe `Vector`, mostrada anteriormente
- Principais métodos:
  - Todos mostrados no `ArrayList/Vector` com alguns métodos a mais:

<code>E push(E item)</code>	Adiciona um item no topo da pilha
<code>E pop()</code>	Remove e retorna o elemento no topo da pilha
<code>E peek()</code>	Retorna o elemento no topo da pilha, sem removê-lo
<code>boolean empty()</code>	Testa se a pilha está vazia

- A classe `LinkedList` (slides anteriores) também possui os métodos acima, permitindo a criação de Pilhas usando Listas Encadeadas

# Interface Queue

- Não existe uma classe em Java para **filas**
  - Existe uma interface (`Queue`) que obriga algumas classes a implementarem as operações usadas em filas
  - Métodos da Interface:

<code>boolean add(E e)</code>	Adiciona um item no final da fila
<code>E remove()</code>	Remove e retorna o elemento do início da pilha
<code>E peek()</code>	Retorna o elemento no topo da pilha, sem removê-lo

- Como a classe `LinkedList` implementa a interface `Queue`, a primeira pode ser usada como uma “Fila implementada por Lista Encadeada”

# Classe PriorityQueue



- Implementa uma **fila com prioridades**
  - Insere elementos em ordem, de acordo com o seu conteúdo ou de acordo com um método de comparação

# Classe Hashtable

- Implementa uma **tabela hash**
- Além do tipo do elemento, deve-se especificar também o tipo da chave
  - Isso é feito na instanciação do objeto, como mostrado no próximo slide
- Principais métodos:

<code>V put(K key, V value)</code>	Insere um valor com uma determinada chave
<code>V get(Object key)</code>	Busca um elemento pela chave
<code>V remove(Object key)</code>	Remove um elemento com determinada chave
<code>int size()</code>	Quantidade de elementos na tabela
<code>Enumeration&lt;V&gt; elements()</code>	Retorna uma enumeração dos elementos



# Classe Hashtable



- A classe `Hashtable` implementa tabelas hash com encadeamento
  - Usa listas encadeadas para lidar com as colisões
- Entretanto, quando a tabela atinge um certo fator de uso:
  - Indicando que a tabela está ficando cheia (e muitas colisões irão ocorrer)
  - O tamanho da tabela é automaticamente incrementado
  - E todos os elementos são reajustados na tabela
  - Isso é conhecido como *rehash*
  - O fator de uso (*load factor*) normalmente é de 75%

# Classe Hashtable

## Exemplo

```
import java.util.*;

public class HashJava {
    public static void main(String args[]) {
        Hashtable<String, Integer> mestres = new Hashtable<String, Integer>();

        mestres.put("Obi-Wan Kenobi", 57);
        mestres.put("Qui-Gon Jinn", 92);
        mestres.put("Yoda", 896);

        Integer n = mestres.get("Yoda");

        if (n != null)
            System.out.println("Nascimento de Yoda: " + n + " BBY");
    }
}
```

Tabela Hash em que as chaves  
são strings e valores são inteiros

# Classe TreeMap

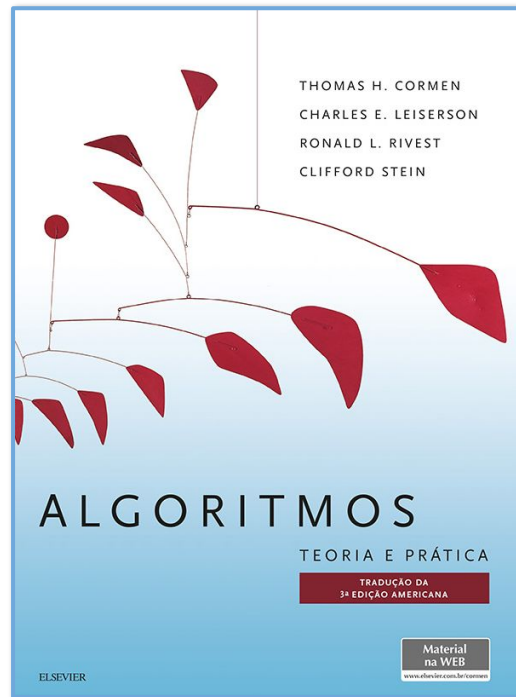
- Implementa a estrutura de dados **árvore**

- Principais métodos:

<code>V put(K key, V value)</code>	Inserir um valor com uma determinada chave
<code>V get(Object key)</code>	Buscar um elemento pela chave
<code>V remove(Object key)</code>	Remover um elemento com determinada chave
<code>int size()</code>	Quantidade de elementos na tabela
<code>Set&lt;Map.Entry&lt;K,V&gt;&gt; entrySet()</code>	Retorna os elementos em ordem ascendente
<code>NavigableMap&lt;K, V&gt; descendingMap()</code>	Retorna os elementos em ordem descendente
<code>Entry&lt;K,V&gt; firstEntry()</code>	Retorna o menor elemento
<code>Entry&lt;K,V&gt; lastEntry()</code>	Retorna o maior elemento

# Classe TreeMap

- A classe `TreeMap` implementa uma **Árvore Vermelho-Preto**
  - A implementação é completamente baseada no livro do Cormen
    - *Teoria e Prática*
  - No próprio código é mencionado isso:
    - “*Algorithms are adaptations of those in Cormen, Leiserson, and Rivest's Introduction to Algorithms.*”



# Classe TreeMap

```
import java.util.*;

public class ArvoreJava {
    public static void main(String args[]) {

        TreeMap<Integer,String> mestres = new TreeMap<Integer,String>();

        mestres.put(57, "Obi-Wan Kenobi");
        mestres.put(92, "Qui-Gon Jinn");
        mestres.put(896, "Yoda");

        Iterator iterator = mestres.descendingMap().entrySet().iterator();

        System.out.println("Mestres ordenado por idade:");

        while (iterator.hasNext()) {
            Map.Entry<Integer,String> mestre =
                (Map.Entry<Integer,String>) iterator.next();
            System.out.println("---> " + mestre.getValue() +
                                   " tem " + mestre.getKey() + " anos");
        }
    }
}
```



# Entrada e Saída



# Fluxos de Dados



- ▣ Fluxo de Dados é uma **sequência de bytes**
- ▣ Fluxos criados **automaticamente**
  - ▣ **System.out**: da classe `PrintStream`, é o objeto de fluxo de saída padrão
    - Normalmente é a **tela**
  - ▣ **System.in**: da classe `InputStream`, é o objeto de fluxo de entrada padrão
    - Normalmente é o **teclado**
  - ▣ **System.err**: da classe `PrintStream`, é o objeto de fluxo de saída de erro
    - Normalmente é a **tela também**

# Fluxos de Dados

```
import java.io.*;

public class TesteES {
    public static void main(String args[]) {
        try {
            int caractere = 0;
            String linha = "";

            while ( (caractere = System.in.read() ) != 10) {
                linha = linha + (char) caractere;
            }

            System.out.println("Linha: " + linha);
            System.err.println("Linha de erro de teste!");
        } catch (IOException e) {}
    }
}
```

```
$ java TesteES
lalalala
Linha: lalalala
Linha de erro de teste!
$ java TesteES 2> /dev/null
oioioioioi
Linha: oioioioioi
```

Lê um caractere do teclado

Imprime na saída padrão

Imprime na saída de erro



# Fluxos de Arquivos: Entrada

- Da mesma forma que lemos a partir do `System.in`, podemos ler a partir de um arquivo usando a classe `FileInputStream`

```
import java.io.*;

public class TesteArqEntrada {
    public static void main(String args[]) {
        try {

            FileInputStream arqEntrada = new FileInputStream("/etc/issue.net");
            int caractere = 0;
            String conteudo = "";
            while ( (caractere = arqEntrada.read() ) != -1)
                conteudo = conteudo + (char) caractere;

            System.out.println("Conteudo do arquivo:\n" + conteudo);
            arqEntrada.close();

        }
        catch (IOException e) {}
    }
}
```

Lê um caractere do arquivo

# Fluxos de Arquivos: Saída

- Da mesma forma que escrevemos no `System.out`, podemos escrever em um arquivo usando a classe `FileOutputStream`

```
import java.io.*;

public class TesteArqSaida {
    public static void main(String args[]) {
        try {

            String conteudo = "Teste de Saída !!\n";
            FileOutputStream arqSaida = new FileOutputStream("/tmp/Teste.txt");
            arqSaida.write(conteudo.getBytes());
            arqSaida.close();

        }
        catch (IOException e) {}
    }
}
```

Abre o arquivo para saída

Escreve o conteúdo

# Laboratório

---

- Disponível no Moodle
  - [bit.ly/iartes-moodle](https://bit.ly/iartes-moodle)

