

# PROGRAMAÇÃO ORIENTADA A OBJETOS COM JAVA

---

Adaptação do material do Prof. Júlio Machado

1

## Recursos

- The Java Tutorial
  - <http://docs.oracle.com/javase/tutorial/index.html>
- Java SE 12 Documentation
  - <https://docs.oracle.com/en/java/javase/12/>

2

# INTRODUÇÃO

3

## Plataforma Java

- Java é tanto uma linguagem de programação de alto nível quanto uma plataforma de desenvolvimento de sistemas
- Como linguagem, Java é orientada a objetos, independente de arquitetura (multiplataforma), portátil, robusta, segura, interpretada, distribuída, etc

4

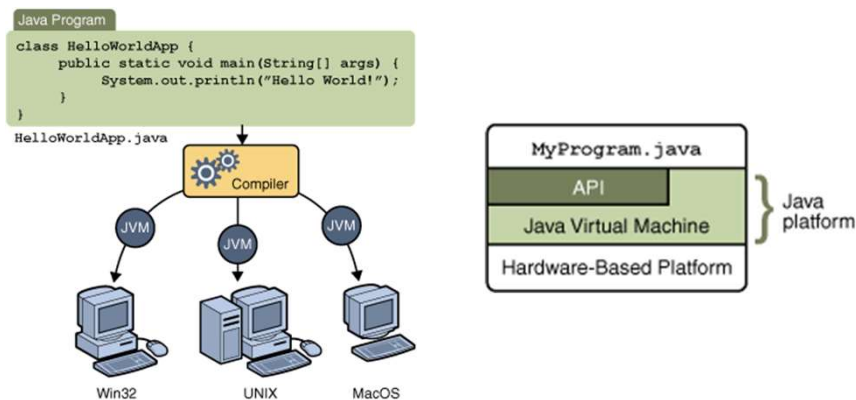
## Plataforma Java

- Java SE (Java Platform Standard Edition)
  - Desenvolvimento e execução de applets, aplicações stand-alone ou aplicações cliente
- Java EE (Java Platform Enterprise Edition)
  - Reúne um conjunto de tecnologias em uma arquitetura voltada para o desenvolvimento de aplicações servidoras
- Java ME (Java Platform Micro Edition)
  - Fornece um ambiente de execução otimizado e permite escrever programas cliente que são executados em pequenos dispositivos móveis (smart cards, telefones celulares, ...)

5

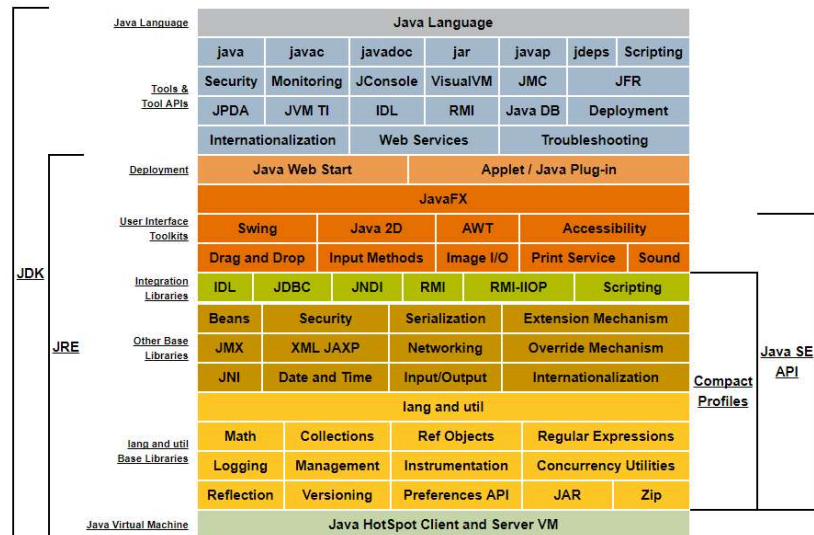
## Plataforma Java

- Compilador e máquina virtual disponíveis para vários sistemas operacionais



6

## Plataforma Java



7

## PROGRAMAÇÃO COM JAVA

8

## Estrutura de um Programa

- Um programa Java é um conjunto composto por uma ou mais classes (em detalhes a frente)
- Tipicamente, cada classe é implementada em um arquivo fonte separado, sendo que o arquivo deve ter o mesmo nome da classe.
  - Ex.: a classe Lampada deve estar definida no arquivo Lampada.java
- Em geral, os arquivos que compõem um programa java devem estar no mesmo diretório
  - Um diretório define um pacote (*package*) em Java

9

## Estrutura de um Programa

```
import java.xxxx.zzz;

// Nosso primeiro programa Java
// Conhecendo a estrutura de um programa Java
public class MeuPrimeiroPrograma {
    public static void main (String arg[]) {
        System.out.println("Olá Aluno de JAVA");
    } // fim do método main
} // fim da classe MeuPrimeiroPrograma
```

**Método main().** Indica que a classe Java é um aplicativo que será interpretado pela máquina virtual.

**Classes.** Declaração de classes, atributos e métodos do programa Java. A declaração e a definição dos métodos ocorre obrigatoriamente dentro do limite de declaração da classe.

10

## Biblioteca de Classes (API)

- Application Programming Interface
- É uma coleção de classes, normalmente provendo uma série de facilidades que podem ser usadas em programas
- Classes são agrupadas em conjuntos chamados packages
  - Exs:
    - java.lang: inclui classes básicas, manipulação de arrays e strings. Este pacote é carregado automaticamente pelo programa
    - java.io: operações de input e output
    - java.util: classes diversas para manipulação de dados

11

## Tipos de Dados Básicos

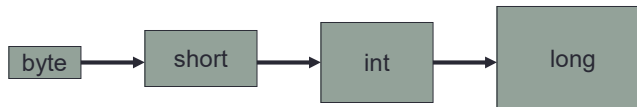
- Tipos de dados primitivos
  - inteiros: byte (8 bits), short (16), int (32), long (64)
    - 1 (decimal), 07 (octal), 0xff (hexadecimal), 1L(long)
  - reais: float (32), double (64)
    - 3.0F (float), 4.02E23 (double), 3.0 (double)
  - caractere: char (16)
    - 'a', '\141', '\u0061', '\n'
  - booleano: boolean (8)
    - true, false

12

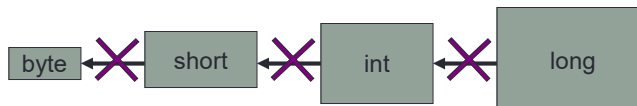
## Tipos de Dados Básicos

- Em Java, tem-se dois tipos de conversão de valores:
  - conversão para um tipo maior
    - automática
  - conversão para um tipo menor (chamada de *casting*)
    - não é automática

```
int x=1;
long y=x;
```



```
long y=1;
int x=y;
Erro!!!
```



13

## Tipos de Dados Básicos

- Para converter de um tipo para um tipo menor, precisamos referenciar de forma explícita.
  - (tipo Java) expressão;
  - Ex.:
    - long y = 1;  
int x = (int)y;
    - byte b1=1, b2=2, b3;  
b3 = (byte) (b1 + b2);
  - Cuidado! Ao somar dois valores *byte* iguais a 100, o resultado é o *int* 200. Ao realizar o *cast* para *byte*, o resultado é convertido para -56, o equivalente ao padrão de bits armazenados.

14

## Operadores

- Operadores básicos:
  - aritméticos: +, -, \*, /, % (resto da divisão)
  - relacionais: >, >=, <, <=
  - igualdade: ==, !=
  - lógicos: &&, & (and), ||, | (or), ^ (xor), ! (not)
  - atribuição: =, +=, -=, \*=, /=, %=
  - incremento, decremento: ++, --

15

## Operadores

- A maioria dos operador aritméticos resultam em *int* ou *long*
  - Quando utilizamos valores *byte* e *short*, eles são convertidos para *int* antes da operação
  - Da mesma forma, se um dos operandos for *long*, os outros são convertidos para *long* antes da operação
  - Ex.:
    - 10 + 10 o resultado é *int*
    - 10L + 10 o resultado é *long*

16



## Operadores

- Cuidado:
  - O resultado da operação de divisão em Java depende do tipo dos operandos
    - Tipo inteiro: o resultado é a divisão inteira  
`int resultado = 10/4 //igual a 2`
    - Tipo ponto flutuante: o resultado é a divisão decimal  
`float resultado = 10f/4f //igual a 2.5`

17

## Funções Matemáticas

- Funções matemáticas (classe *Math*):
  - `sqrt(x)`: cálculo da raiz quadrada de x (x é do tipo `double`)
  - `abs(x)`: valor absoluto de x (x pode ser `float`, `int`, `long`)
  - `cos(x)`: coseno trigonométrico de x (x em radianos)
  - `exp(x)`: método exponencial  $e^x$
  - `pow(x,y)`: x elevado a potência y ( $x^y$ )
- Exemplo:

```
double raio;  
raio = Math.sqrt(area/Math.PI);
```

18

## Classe String

- String
  - É uma classe e não tipo primitivo
  - Representa um grupo de caracteres
    - Codificação Unicode UTF-16
  - É uma classe de objetos imutáveis
    - Uma vez inicializado, o valor da string jamais é alterado
  - Declarados entre aspas duplas
    - `String nome = "Júlio";`

19

## Classe String

- Operadores
  - concatenação: +
    - `String nomeCompleto = nome + " " + "Machado";`
  - comparação: equals
    - `String str1 = "texto";`  
`String str2 = "txt";`  
`if(str1.equals(str2)){}` //compara conteúdo
    - `String str1 = "texto";`  
`String str2 = "txt";`  
`if (str1 == str2){}` //compara endereço

20

## Classe String

- Métodos úteis

- Tamanho:

- Método *length()*
    - `String texto1 = "Início";`  
`System.out.println(texto1.length());`  
`--> 6`

- Caractere em uma posição:

- Método *charAt(posição)*
    - O primeiro caractere está na posição 0
    - `char c = texto1.charAt(1);`  
`--> n`

- Substrings:

- Método *substring(início,fim)*
    - `String texto1 = "Início";`  
`String sub = texto1.substring(1,3)`  
`--> ní`

21

## Classe String

- Conversão

- Java converte outros tipos para strings

- `int idade = 25;`  
`String nomeIdade = nome + " " + idade;`

- Como converter tipos primitivos para strings?

- Métodos *String.valueOf()*, *Integer.toString()*, *Double.toString()*
    - São métodos de classe
    - `String sete = String.valueOf(7);`  
`String umPontozero = Double.toString(1.0);`

22

## Classe String

- Conversão
  - Como converter strings para tipos primitivos?
    - Métodos `Integer.valueOf()`, `Double.valueOf()`
    - São métodos de classe
    - ```
int sete = Integer.valueOf("7");  
double umPontozero = Double.valueOf("1.0");
```

23

## Comandos - Declaração

- Variáveis:
  - ```
int valor1, valor2 = 123;
```

    - Com inicialização
  - ```
double taxa, percentual;
```

    - Sem inicialização
    - Variáveis locais não são inicializadas automaticamente
    - Atributos são inicializados automaticamente
- Constantes:
  - ```
final double PI = 3.1415;
```

    - Modificador *final*

24

## Comandos – Condicional IF

- `if (condição) {`  
    `comandos;`  
    `}`
- `if (condição) {`  
    `comandos;`  
    `} else {`  
        `comandos;`  
    `}`
- `if (condição) {`  
    `comandos;`  
    `} else if (condição) {`  
        `comandos;`  
    `} else {`  
        `comandos;`  
    `}`

25

## Comandos – Condicional IF

```

if (i % 2 == 0) {
    System.out.println("Par");
} else {
    System.out.println("Ímpar");
}

if (vel >= 25) {
    if (vel > 65) {
        System.out.println("maior 65");
    } else {
        System.out.println("entre 25 e 65");
    }
} else {
    System.out.println("menor 25");
}

```

26

## Comandos – Condicional SWITCH

- Utilizado para cobrir múltiplas escolhas sobre valores alternativos de variáveis *int*, *byte*, *short*, *long*, *char*, *enumeration*, *String*

```
• switch (expressão) {
    case constante1:
        comandos;
        break;

    ...
    default:
        comandos;
}
```

27

## Comandos – Condicional SWITCH

```
switch (menuItem) {
    case 0:
        System.out.println("zero");
        break;
    case 1:
        System.out.println("um");
        break;
    default:
        System.out.println("inválido");
}

switch (nota) {
    case 'A':
    case 'B':
    case 'C':
        System.out.println("Passou");
        break;
    default:
        System.out.println("Reprovou");
}
```

28

## Comandos – Repetição FOR

- *for (inicialização; terminação; incremento) {*  
    *comandos;*  
    *}*

```
int soma = 0;
for (int i=1; i<=3; i++) {
    soma +=i;
}
System.out.println("Soma "+soma);
```

29

## Comandos – Repetição WHILE

- *while (condição) {*  
    *comandos;*  
    *}*

```
int i = 0;
while (i<10) {
    System.out.println("i= "+i);
    i++;
}
```

30

## Comandos – Repetição DO WHILE

- *do {*  
    *comandos;*  
*} while (condição);*

```
int i = 0;
do {
    System.out.println("i= "+i);
    i++;
} while (i<10);
```

31

## Comandos - Repetição

- Controle de Loops

- *break*:
  - Termina o comando de repetição
- *continue*:
  - Abandona a iteração atual da repetição e passa para a próxima iteração
- Ex.:

soma = 0;	soma = 0;
for (i=0; i < 5; i++)	for (i=0; i < 5; i++)
{	{
if (i == 3) <i>continue</i> ;	if (i == 3) <i>break</i> ;
soma += i;	soma += i;
}	}

32



# ARRANJOS

33

## Arranjos

- Arranjos ou Arrays são estruturas que armazenam uma sequência de itens do mesmo tipo
  - Tipos primitivos
  - Objetos
- É uma estrutura estática
  - Seu tamanho não pode ser alterado após a criação
- Java permite a criação de arranjos de múltiplas dimensões:
  - Arranjo unidimensional = “vetor”
  - Arranjo bidimensional = “matriz”

34

## Arranjos Unidimensionais

- Declaração de um array é feita em duas etapas:
  - Declaração da referência → `int[] valores;`
  - Instanciação do objeto → `valores = new int[5];`
- Em um única linha:
  - `int[] valores = new int[5];`

35

## Arranjos Unidimensionais

- Valores do arranjo são inicializados automaticamente:
  - números : 0
  - boolean : false
  - objetos : null

36

## Arranjos Unidimensionais

- Dado um array de tamanho N:
  - Primeira posição com índice 0
  - Última posição com índice N-1
  - Acesso a uma posição inválida acarreta uma exceção `IndexOutOfBoundsException`
- Para referenciar elementos:
  - `nome_do_array[índice]`
  - Exemplo:
    - `valores[0]`
    - `valores[4]`

37

## Arranjos Unidimensionais

- É possível inicializar um array com valores literais.
  - Exemplos:

```
int[] valores = {1,2,3,4,5};
String[] nomes = {"eu","tu"};
```

38

## Arranjos Unidimensionais

- O que acontece nos seguintes casos? (Lembre-se que em Java temos referências para objetos!)

```
int[] nums;
nums = new int[10];
...
nums = new int[20];

int[] nums = {1,2,3};
int[] outros = nums;
```

*nums* referencia um novo objeto array, perdendo a referência para o array anterior

*nums* e *outros* referenciam o mesmo objeto array

39

## Arranjos Unidimensionais

- Arrays de objetos contêm referências para os outros objetos
  - Por exemplo, um array que armazena professores

```
Professor[] lista = new Professor[10];
lista[0] = new Professor("Maria",13,12);
lista[1] = new Professor("José",234,8);
```

40

## Arranjos Unidimensionais

- Atributos e métodos de arrays:
  - Definidos no pacote `java.util`
  - Tamanho:
    - `length`
    - `System.out.println(nums.length);`
  - Ordenação em ordem crescente:
    - `Arrays.sort(nome_do_array)`
    - `Arrays.sort(nums);`

41

## Arranjos Unidimensionais

• Exemplo:

```
import java.util.Arrays;
public class TesteArray {
    public static void main(String[] args) {
        int[] nums = {32, 15, 3, 23, 4, 0, 1};
        for (int i = 0; i < nums.length; i++) {
            System.out.println("nums [" + i + "]="
                               + nums[i] + "\n");
        }
        Arrays.sort(nums);
        for (int i = 0; i < nums.length; i++) {
            System.out.println("nums [" + i + "]="
                               + nums[i] + "\n");
        }
    }
}
```

42

## Arranjos Unidimensionais

- Passagem de parâmetros:
  - Arrays, como são objetos, os parâmetros são referências
    - Especificar o nome do array sem colchetes
 

```
int[] nums = new int[5];
modificaArray(nums);
```
    - Parâmetro do método declarado como uma referência ao array
 

```
public void modificaArray(int[] vals){...}
```

43

## Arranjos Bidimensionais

- Declaração de um array é feita em duas etapas:
  - Declaração da referência → `int[][] valores;`
  - Instanciação do objeto → `valores = new int[5][2];`
- Em um única linha:
  - `int[][] valores = new int[5][2];`
- Para referenciar elementos:
  - `nome_do_array[indiceLinha][indiceColuna]`
  - Exemplo:
    - `valores[0][1]`
    - `valores[4][0]`

44

## Arranjos Bidimensionais

- Inicialização:

- Valores literais

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

45

## Arranjos Bidimensionais

- Inicialização:

- Laços de repetição aninhados

```
for(int i=0; i<numLinhas; i++)  
    for(int j=0; j<numColunas; j++)  
        valores[i][j] = 0;
```

46

# ORIENTAÇÃO A OBJETOS, CLASSES E OBJETOS

47

## Introdução à Programação Orientada a Objetos

- O que é um paradigma de programação?
  - É um padrão conceitual que orienta soluções de projeto e implementação
  - Paradigmas explicam como os elementos que compõem um programa são organizados e como interagem entre si
    - Exs.: procedural, funcional, orientado a objetos

48



## Orientação a Objetos

- É baseada na modelagem de objetos do mundo real
- O que é um objeto?
  - Uma entidade que você pode reconhecer
  - Uma abstração de um objeto do mundo real
  - Uma estrutura composta de dados e operações sobre esses dados

49

## Objetos

- Cada objeto possui características (atributos) e comportamento (operações)
  - Ex.: lâmpada
    - características: ligada (sim/não), potência, voltagem
    - comportamento: ligar, desligar, queimar

50

## Objetos

- Um programa orientado a objetos é estruturado como uma comunidade de objetos que interagem entre si
  - Cada objeto tem um papel a cumprir
  - Cada objeto oferece um serviço ou realiza uma ação que é usada por outros objetos
  - Ex.: um objeto Lustre interage com diversos objetos Lâmpada

51

## Classes

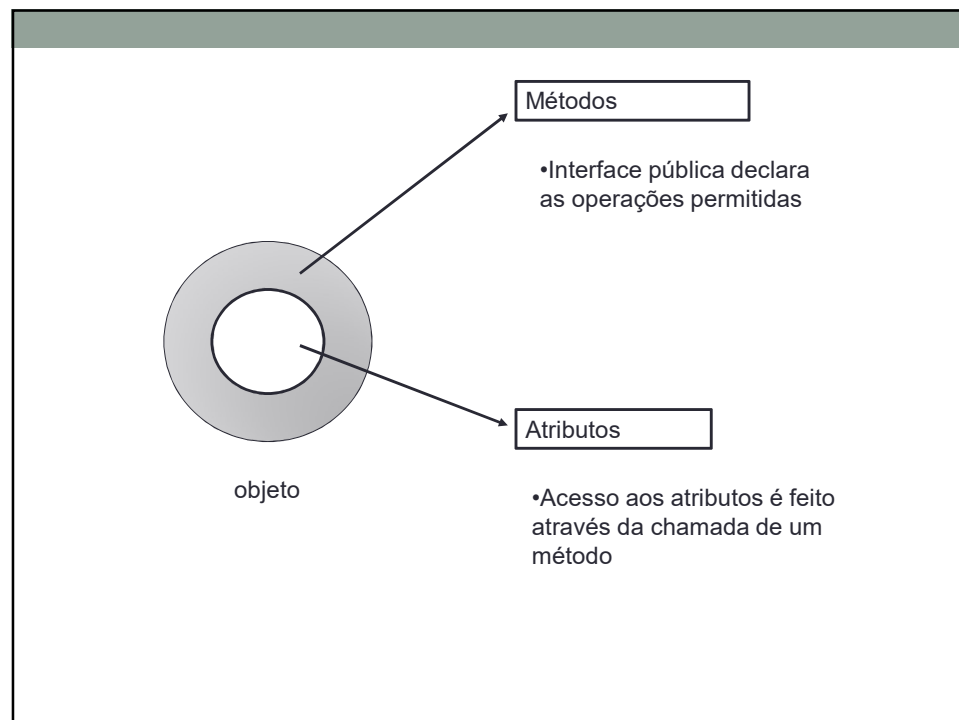
- A classe é a definição formal dos atributos e métodos que compõem os objetos
- Objetos são instâncias de uma classe

52

## Encapsulamento

- Encapsular é esconder como as coisas funcionam por trás de uma interface externa
  - Interface são as operações que o objeto fornece para os demais objetos
  - É um dos conceitos básicos da Orientação a Objetos
- A ideia é de uma “caixa preta”:
  - Não é necessário saber os detalhes de funcionamento interno do objeto, mas sim como utilizá-lo
- Ex.: caixa automático
  - Como ele é implementado internamente?
  - Utilizamos através de operações bem conhecidas

53



54

## Encapsulamento

- Alguns benefícios:
  - A implementação interna de um objeto pode mudar e o resto do sistema não é afetado (desde que a interface de acesso não mude)
  - Maior segurança ao proteger os atributos de um objeto de alterações indevidas por outros objetos
  - Maior independência entre os objetos, pois eles só precisam conhecer a interface externa definida

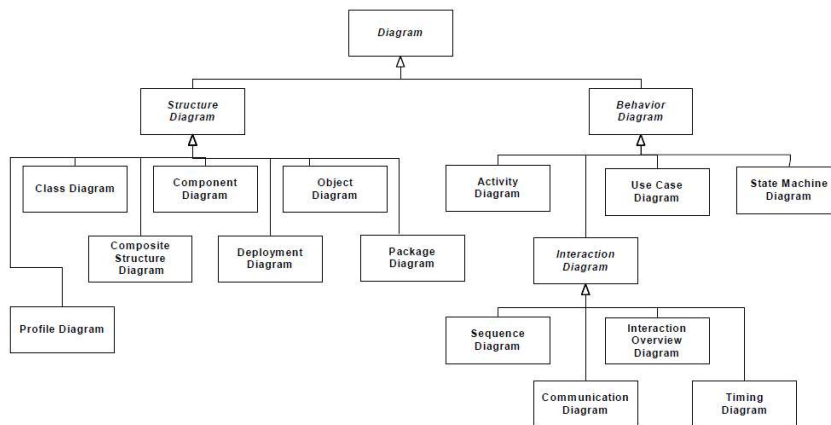
55

## Projetando Objetos

- De uma forma simples, o projeto orientado a objetos de um sistema pode ser dividido em três etapas:
  - Identificar as abstrações/entidades envolvidas no problema
  - Identificar o comportamento que cada uma destas entidades deve ser capaz de fornecer
  - Identificar os relacionamentos entre essas entidades
  - Identificar as estruturas de dados internas necessárias para implementar o comportamento e relacionamentos desejado

56

## Diagramas UML



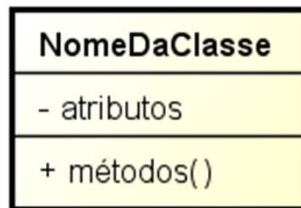
57

## Diagrama de Classes UML

- Denota a estrutura estática do sistema
- Apresenta as classes e seu relacionamentos com outras classes

58

## Diagrama de Classes da UML



powered by astah<sup>®</sup> 

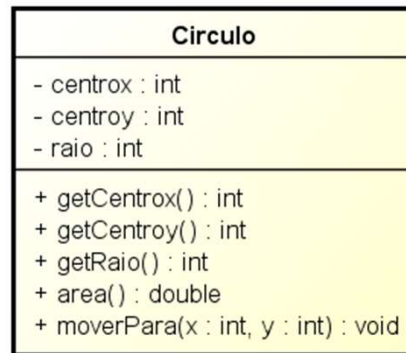
59

## Diagrama de Classes da UML

- Modificadores:
  - Público +
  - Privado -

60

## Diagrama de Classes UML

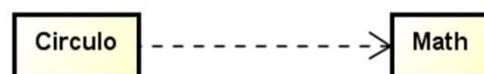


powered by astah®

61

## Diagrama de Classes UML

- Relacionamento de dependência:
  - É um relacionamento que significa que um elemento necessita de outro elemento para sua especificação ou implementação
  - É um relacionamento “fornecedor-cliente”
    - Um objeto fornece algo que outro objeto utiliza



powered by astah®

62

## Diagrama de Classes UML

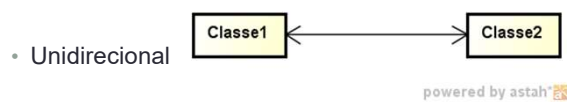
- Relacionamento de associação:
  - É um relacionamento estrutural que descreve um conjunto de ligações, onde uma ligação é uma conexão entre objetos
  - Usualmente implementado através de atributos



63

## Diagrama de Classes UML

- Relacionamento de associação:
  - Navegabilidade da associação
    - Bidirecional



- Unidirecional



64



## Diagrama de Classes UML

- Relacionamento de associação:
  - Multiplicidade da associação
    - Especifica-se o menor e o maior valor
    - Formato Menor..Maior
    - Valores mais utilizados
      - Menor: 0 (opcional), 1 (obrigatório)
      - Maior: 1 (somente um), \* (vários)

65

## Diagrama de Classes UML

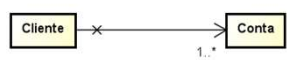
- Relacionamento de associação:
  - Multiplicidade da associação
    - Cliente tem uma única conta (1..1 ou 1)



- Cliente pode ter ou não uma conta



- Cliente tem varias contas, mas no minimo uma



- Cliente tem varias contas, mas nao e obrigatório (0..\* ou \*)



66

## Resumo

- Objeto
  - Unidade básica de orientação a objetos. Um objeto é uma entidade que tem atributos, comportamento e identidade. Objetos são membros de uma classe e os atributos e métodos de um objeto são definidos pela classe.
- Classe
  - Uma classe é uma descrição de um conjunto de objetos. Este conjunto de objetos compartilha atributos e comportamento em comum. Uma definição de classe descreve todos os atributos dos objetos membros da classe, bem como os métodos que implementam o comportamento destes membros.

67

## Resumo

- Orientação a objetos
  - Um paradigma de programação que usa abstração com objetos, classes encapsuladas e comunicação por mensagens, hierarquia de classes e polimorfismo.
- Abstração
  - Um modelo de um conceito ou objeto do mundo real.
- Encapsulamento
  - Processo de esconder os detalhes internos de um objeto do mundo externo.

68

## Resumo

- **Comportamento**
  - Atividade de um objeto que é vista do ponto de vista do mundo externo. Inclui como um objeto responde a mensagens alterando seu estado interno ou retornando informação sobre seu estado interno.
- **Método**
  - Uma operação ou serviço executado sobre o objeto, declarado como parte da estrutura da classe. Métodos são usados para implementar o comportamento do objeto.
- **Estado**
  - Reflete os valores correntes de todos os atributos de um objeto e são o resultado do comportamento do objeto ao longo do tempo.
- **Atributo**
  - Usado para armazenar o estado de um objeto. Pode ser simples como uma variável escalar (int, char, double, ou boolean) ou pode ser uma estrutura complexa tal como outro objeto.

69

## Classes

- Definições de classes incluem (geralmente):
  - modificador de acesso
  - palavra-chave *class*
  - nome da classe
  - corpo classe
    - atributos
    - métodos
    - construtores

70

## Classes

- Modificadores de acesso
  - Permitem definir o encapsulamento de atributos e métodos
  - Dois modificadores principais:
    - ***private***: visível apenas para objetos da própria classe
    - ***public***: visível para quaisquer objetos

71

## Classes

- Recomendações
  - A menos que hajam razões fortes, os atributos de uma classe devem ser definidos como *private* (encapsulamento) e os métodos que são chamados de fora da classe devem ser *public* (interface de acesso ao comportamento público)
  - Métodos que devem ser usados somente dentro da própria classe, devem ser especificados como *private* (comportamento privado)

72

## Classes

- Métodos get
  - Retornam o valor do estado atual de um objeto, uma vez que não é possível acessá-lo diretamente
- Métodos set
  - Permitem alterar o valor do estado atual do objeto
  - Estes métodos são chamados por alguns autores de mutantes (mutator methods\*)

\* David J. Barnes, Michael Kölling. Objects First with Java: A Practical Introduction using BlueJ. Prentice Hall / Pearson Education, 2003

73

## Exemplo: classe Professor

```
class Professor
{
    private String nome;
    private int matricula;
    private int cargaHoraria;
    ...
}
```

- Atributos estão encapsulados!!!
- Apenas métodos da própria classe Professor podem acessar os atributos

74

## Exemplo: classe Professor

- Métodos:

```
...
public void setNome(String n) {
    nome = n;
}
public String getNome() {
    return nome;
}
public void setMatricula(int m) {
    matricula = n;
}
public int getMatricula() {
    return matricula;
}
...
```

75

## Exemplo: classe Professor

```
...
public void setCargaHoraria(int c) {
    cargaHoraria = c;
}
public int getCargaHoraria() {
    return cargaHoraria;
}
public float getCargaHorariaMensal() {
    return (cargaHoraria * 4.5F);
}
```

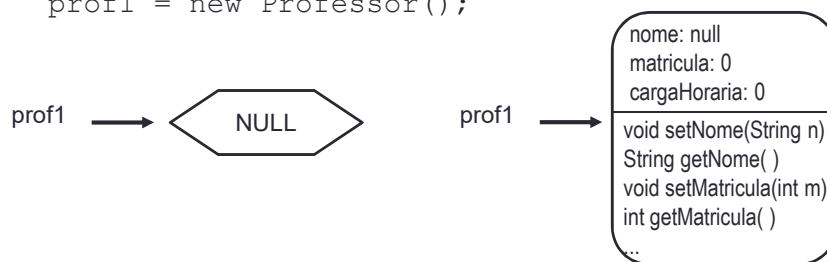
76

## Objetos

- Instanciação

- Um objeto depois de criado, conterá todos os atributos e métodos descritos em sua classe
- Para instanciar um objeto em Java utilizamos o operador *new*
- Ex.:

```
Professor prof1;  
prof1 = new Professor();
```



77

## Objetos

- Quando o operador *new* é usado é “alocada” memória
- Quando um objeto não é mais necessário, devolve-se o(s) recurso(s) para o sistema
- Java realiza a coleta de lixo automática da memória (*garbage collector*)
- Quando um objeto não é mais utilizado, ele é marcado para coleta de lixo

78

## Programa

- Como executar um programa em Java?
  - Um programa é composto de várias classes e objetos
  - Como indicar por onde o programa começa?
  - Em Java temos um método especial que o interpretador assume como o início do programa: *main*.
    - `public static void main (String args[])`

79

## Programa

```
public static void main (String args[]){  
    Professor prof1, prof2;  
    prof1 = new Professor();  
    prof1.setNome("Júlio");  
    prof1.setMatricula(1234);  
    prof1.setCargaHoraria(14);  
    System.out.println(prof1.getCargaHorariaMen  
sal());  
}
```

80



## Escopo de Variáveis

- O escopo de uma variável informa onde ela pode ser utilizada.

- Ex.:

```

1: public class VerificaEscopo{
2:   private int escopoA;
3:   public void metodo(int escopoB){
4:     int escopoC;
5:   }
6:   private int escopoD;
7: }
```

81

## Escopo de Variáveis

- Ex.:

```

1: public class VerificaEscopo{
2:   private int escopoA;
3:   public void metodo(int escopoB){
4:     int escopoC;
5:   }
6:   private int escopoD;
7: }
```

- No exemplo

- escopoA e escopoD são atributos de instância do objeto e seu escopo vale a partir da linha 1
- escopoB e escopoC são variáveis locais cujo escopo é válido somente dentro do método

82

## Escopo de Variáveis

- Variáveis locais podem ser declaradas a qualquer momento dentro de um método

- Ex.:

```
for (int i=1; i<5; i++){  
    int j = 0;  
    //i e j só valem aqui dentro  
}  
System.out.println(i); //erro
```

83

## Inicialização de Variáveis

- Atributos de uma classe são inicializados com valores padrão:

- 0 -> byte, short, int, long
- 0.0 -> float, double
- false -> boolean
- \u0000 -> char
- null -> Object

84

## Inicialização de Variáveis

- Variáveis locais declaradas dentro de método devem obrigatoriamente serem inicializadas antes de utilizadas
  - O compilador Java irá indicar se não inicializarmos as variáveis

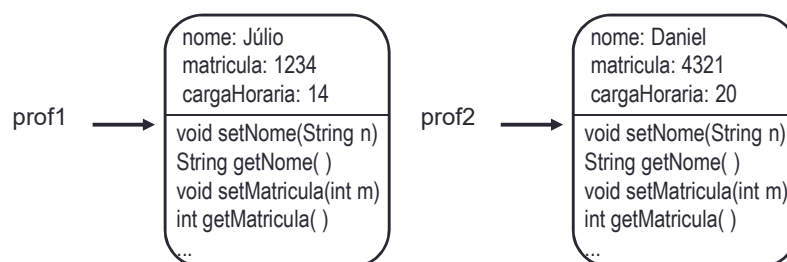
85

## Referências

- Quando criamos um objeto em Java, mantemos uma referência para o objeto na memória

- Ex.:

```
Professor prof1, prof2;
prof1 = new Professor();...
prof2 = new Professor();...
```



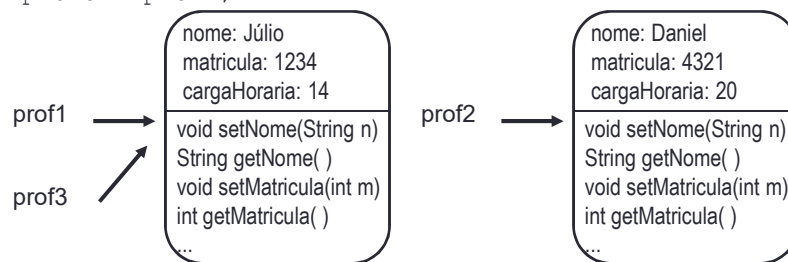
86

## Referências

- Ao atribuir prof1 ou prof2 a uma terceira variável, o que irá acontecer?

- Ex.:

```
Professor prof1, prof2, prof3;
prof1 = new Professor();...
prof2 = new Professor();...
prof3 = prof1;
```



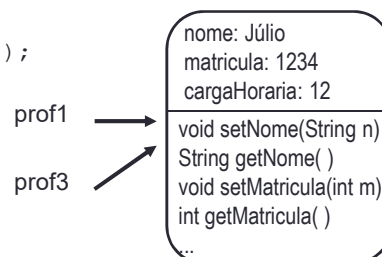
87

## Referências

- Se alteramos algum atributo do objeto referenciado por prof3, estaremos alterando também o referenciado por prof1!

- Ex.:

```
Professor prof1, prof2, prof3;
prof1 = new Professor();...
prof2 = new Professor();...
prof3 = prof1;
prof3.setCargaHoraria(12);
```



88

## Inicialização de Objetos

- Objetos:
  - Estado: definido pelos atributos declarados na classe
  - Comportamento: definido pelos métodos declarados na classe
- Quais valores os atributos do objeto possuem após a sua instanciação?
- Como definir o estado inicial do objeto?

89

## Inicialização de Objetos

- Exemplo: classe Circulo

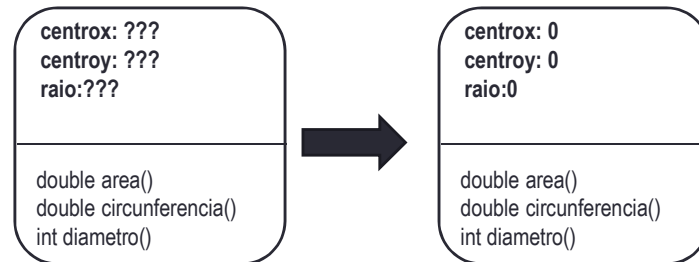
```
public class Circulo {
    private int centrox;
    private int centroy;
    private int raio;
    public double area(){
        return (3.14 * raio * raio);
    }
    public double circunferencia(){
        return (2 * 3.14 * raio);
    }
    public int diametro(){
        return (2 * raio);
    }
}
```

Circulo
-centrox:int -centroy:int -raio:int
+area():double +circunferencia():double +diametro():int

90

## Inicialização de Objetos

```
Circulo circ = new Circulo();
```



91

## Inicialização de Objetos

- Da forma como foi apresentada a classe `Circulo`, todos os objetos criados a partir dela terão seus atributos inicializados com valores padrão iguais a zero
- Como permitir que instâncias da classe `Circulo` possuam estados diferentes?
  - Adicionar à classe um método para inicializar os atributos com valores diferentes da inicialização padrão
  - Esse método é o construtor!

92

## Inicialização de Objetos

- Exemplo: classe Circulo

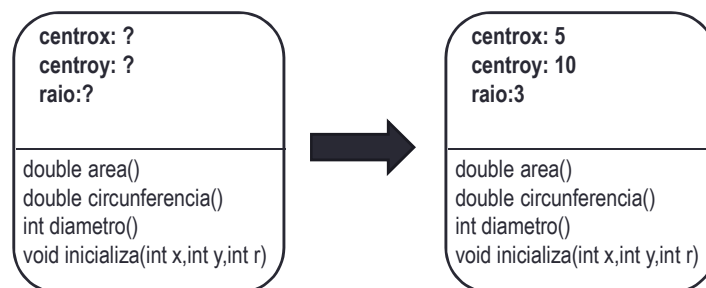
```
public class Circulo {
    private int centrox;
    private int centroy;
    private int raio;
    public Circulo(int x, int y, int r){
        centrox = x;
        centroy = y;
        raio = r;
    }
    ...
}
```

Circulo
-centrox:int -centroy:int -raio:int
+Circulo(x:int, y:int, r:int) +area():double +circunferencia():double +diametro():int

93

## Inicialização de Objetos

```
Circulo circ = new Circulo(5,10,3);
```



94

## Inicialização de Objetos

- Um construtor em Java:
  - Possui o mesmo nome da classe (respeitando maiúsculas e minúsculas)
  - Pode possuir ou não parâmetros
  - Não possui um tipo de retorno, nem mesmo void

```
<modificador_de_acesso> <nome_classe>(<parâmetros>){  
    //corpo do construtor  
}
```

95

## Inicialização de Objetos

- Se nenhum construtor é definido para uma determinada classe, Java irá definir um construtor padrão (chamado construtor *default*)
  - Não possui argumentos de entrada
  - Caso qualquer outro construtor seja definido na classe, Java não irá disponibilizar o construtor padrão

96



## Sobrecarga

- Chama-se de sobrecarga de métodos (*overloading*) o ato de criar diversos métodos com o mesmo nome que se diferenciam pela lista de argumentos (parâmetros)
  - Métodos são identificados pela sua assinatura: nome do método + lista de parâmetros
  - Métodos com mesmo nome, mas com tipo, quantidade ou ordenação de parâmetros diferentes, são considerados métodos diferentes

97

## Sobrecarga

- Cuidado!!!
  - Esses métodos possuem uma definição correta para sobrecarga?

```
public void soma(int n, double d)
public void soma(double d, int n)
```

```
public void soma(int n)
public void soma(int v)
```

```
public void soma(int n)
public double soma(int n)
```

98

## Sobrecarga

- Na API de Java, diversas classes utilizam a sobrecarga de métodos, por exemplo:
  - Classe String
    - valueOf (boolean b)
    - valueOf (char c)
    - valueOf (double d)
    - valueOf (float f)
    - valueOf (int i)
    - valueOf (long l)
    - retorna a representação em String do argumento recebido

99

## Sobrecarga de Construtores

- Usualmente é útil para uma classe possuir mais de um construtor a fim de oferecer diversas maneiras para instanciar e inicializar os objetos dessa classe
- Um construtor também pode sofrer o processo de sobrecarga

100

## Sobrecarga de Construtores

- Exemplo: classe Circulo
  - Deseja-se ter a capacidade de inicializar os atributos de um novo objeto de duas formas:
    - através de um construtor sem parâmetros, que cria um círculo padrão de centro (0,0) e raio 1,
    - e através de um construtor que recebe as informações de centro e raio para criar o círculo.

Circulo
-centrox:int -centroy:int -raio:int
+Circulo(x:int, y:int, r:int) +Circulo() +area():double +circunferencia():double +diametro():int

101

## Sobrecarga de Construtores

```
public class Circulo {
    private int centrox;
    private int centroy;
    private int raio;
    public Circulo(int x, int y, int r){
        centrox = x;
        centroy = y;
        raio = r;
    }
    public Circulo() {
        centrox = 0;
        centroy = 0;
        raio = 1;
    }
    ...
}
```

102

## Sobrecarga de Construtores

- Testando a classe:

```
public class TesteCirculo {  
    public static void main (String args[]) {  
        Circulo circ1 = new Circulo();  
        Circulo circ2 = new Circulo(1,2,4);  
        System.out.println("Area circ1= " + circ1.area());  
        System.out.println("Area circ2= " + circ2.area());  
    }  
}
```

103

## Sobrecarga de Construtores

- Observando mais de perto a implementação dos dois construtores da classe Circulo:
  - Nota-se que o segundo construtor (o construtor sem parâmetros) possui o mesmo código de inicialização do primeiro construtor (o construtor com três parâmetros)
- Repetir desnecessariamente código não é uma boa prática de programação
- Java permite compartilhar código entre os diversos construtores
  - Palavra-chave this()

104

## Sobrecarga de Construtores

```
public class Circulo {  
    private int centrox;  
    private int centroy;  
    private int raio;  
    public Circulo(int x, int y, int r){  
        centrox = x;  
        centroy = y;  
        raio = r;  
    }  
    public Circulo() {  
        this(0,0,1);  
    }  
    ...  
}
```

105

## Atributos e Métodos de Classe

- Java permite declarar duas categorias distintas de atributos e métodos:
  - atributos de instância
  - atributos de classe
  - métodos de instância
  - métodos de classe

106

## Atributos de Classe

- Cada objeto de uma classe possui sua própria cópia de todos os atributos de instância da classe
- Em certos casos, entretanto, é interessante que apenas uma cópia de um atributo em particular seja compartilhada por todos os objetos de uma classe
- Exemplo: constantes da classe *Math*
  - As constantes matemáticas E e PI são armazenadas em um única cópia e então compartilhadas

107

## Atributos de Classe

```
public class TestaMath {
    public static void main(String args[]) {
        System.out.println("PI = " + Math.PI);
        System.out.println("E = " + Math.E);
    }
}
```

- Note que os atributos públicos não são acessados a partir de um objeto!
- Atributos acessados pelo nome da classe

108

## Atributos de Classe

- Atributos de Instância:
  - Cada objeto possui uma cópia particular com seus valores
  - Representam o estado de um objeto em particular
- Atributos de Classe:
  - Cada classe possui uma única cópia do atributo, independente do número de objetos instanciados a partir da classe
  - Objetos compartilham os atributos de classe
  - São declarados pela palavra-chave *static*
  - Invocação  
`<nome_classe>.<nome_atributo_público>`

109

## Atributos de Classe

- Exemplo: classe Circulo
  - Nos métodos de cálculo da área e circunferência, percebe-se a presença de um valor importante em cálculos geométricos que se repete para todas as instâncias
    - Esse valor é a constante Pi
  - Pode ser desejado manter somente uma cópia desse valor, com a aproximação desejada no número de suas casas decimais de uma forma consistente, impedindo que em um método seja utilizado o valor 3,14 e em outro 3,1415
  - Pi será declarado como atributo de classe (*static*) e constante (*final*)

110

## Atributos de Classe

```
public class Circulo {  
    public static final double PI = 3.14;  
    private int centrox;  
    private int centroy;  
    private int raio;  
    ...  
    public double area() {  
        return (PI * raio * raio);  
    }  
    public double circunferencia() {  
        return (2 * PI * raio);  
    }  
    ...  
}
```

111

## Inicialização de Atributos de Classe

- Convém destacar que a forma de inicialização dos atributos de classe é usualmente no momento de sua declaração, pois eles não pertencem às instâncias e portanto não dependem do construtor para serem inicializados
  - Se a inicialização com valores padrão for suficiente, não é necessário inicializar o atributo explicitamente

112



## Inicialização de Atributos de Classe

- Para inicializar atributos de classe que necessitam de uma forma mais complexa, Java fornece um bloco de inicialização estático
  - Não possui nome
  - Não possui tipo de retorno
  - Começa pela palavra-chave `static`, seguido de um bloco de código entre parênteses
  - Executa somente uma vez quando a classe é carregada em memória

```
public class UmaClasse {  
    ...  
    public static int atributo;  
    static {  
        //código para inicializar atributo  
    }  
}
```

113

## Métodos de Classe

- Em muitos exemplos de classes pode-se notar alguns métodos que não acessam nenhum atributo de uma instância
- Exemplo: funções trigonométricas da classe *Math*
  - Os métodos `sin`, `cos` e `tan` recebem o valor do ângulo (em radianos) por parâmetro e devolvem o seno, cosseno ou a tangente correspondente calculados unicamente a partir do valor recebido

114

## Métodos de Classe

```
public class Trigonometria {
    public static void main(String args[]) {
        System.out.println("Seno(45) = " +
            Math.sin(Math.PI/4));
        System.out.println("Coseno(45) = " +
            Math.cos(Math.PI/4));
        System.out.println("Tangente(45) = " +
            Math.tan(Math.PI/4));
    }
}
```

- Note que os métodos de cálculo não são executados sobre um objeto!
- Métodos acessados pelo nome da classe

115

## Métodos de Classe

- Métodos de Instância:
  - Fornecem o comportamento dos objetos instanciados a partir de uma classe
  - Trabalham sobre os atributos de instância de um objeto dessa classe
- Métodos de Classe:
  - Fornecem um comportamento que é independente da existência de objetos de uma classe
  - Pertencem à classe e são compartilhados por todas as instâncias da classe
  - Podem acessar os atributos de classe, mas não os atributos de instância diretamente
  - Indicados pela palavra-chave *static*
  - Invocação  
`<nome_classe>.<nome_método>(<parâmetros>)`

116

## Métodos de Classe

- Exemplo: classe Circulo
  - O método *equacaoGeral* será acrescentado à classe Circulo
  - Seu propósito é, a partir dos valores de centro e raio de um círculo, obter a representação textual da chamada equação geral da circunferência

117

## Métodos de Classe

```
public class Circulo {
    ...
    public static String equacaoGeral(int x, int y, int r) {
        int a = -2 * x;
        int b = -2 * y;
        int c = (x*x) + (y*y) - (r*r);
        StringBuffer eq = new StringBuffer("x2 + y2");
        if (a > 0) {
            eq.append(" + ");
            eq.append(a);
            eq.append("x");
        }
        else if (a < 0) {
            eq.append(" ");
            eq.append(a);
            eq.append("x");
        }
    }
}
```

118

## Métodos de Classe

```

    if (b > 0) {
        eq.append(" + ");
        eq.append(b);
        eq.append("y");
    }
    else if (b < 0) {
        eq.append(" ");
        eq.append(b);
        eq.append("y");
    }
    if (c > 0) {
        eq.append(" + ");
        eq.append(c);
    }
    else if (c < 0) {
        eq.append(" ");
        eq.append(c);
    }
    eq.append(" = 0");
    return eq.toString();
}

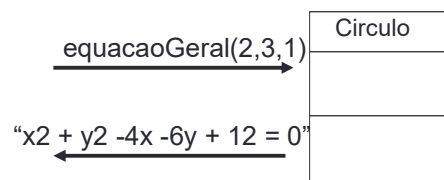
```

119

## Métodos de Classe

- Utilizando a nova definição em um exemplo:

```
String eq = Circulo.equacaoGeral(2,3,1);
```



120

## Enumeração

- Um tipo de enumeração (ou tipo enumerado) é um tipo para qual os valores são conhecidos quando o tipo é definido
- Exemplos:
  - Naipes, dias da semana, meses do ano

121

## Enumeração

- Declaração
  - Palavra-chave *enum*
  - Identificador da enumeração
  - Lista de constantes da enumeração entre chaves e separadas por vírgula
- Exemplo:  
`enum Naipe { PAUS, OUROS, COPAS, ESPADAS }`

122

## Enumeração

- Uso
  - Enumerações são seguras quanto ao tipo
    - Somente os valores declarados e null
  - Declara-se uma variável do tipo da enumeração
  - É possível utilizar comparação via ==
  - Pode ser utilizado com comando switch
- Exemplo:

```
Naipes n = Naipes.OUROS;  
if(n == Naipes.OUROS)...  
switch(n){  
    case PAUS : ...  
    ...  
}
```

123

## COLEÇÕES

124

## Coleções

- Java disponibiliza classes que facilitam o agrupamento e processamento de objetos em conjuntos:
  - Coleções (*Java Collections Framework*)
  - Estruturas de dados + algoritmos para sua manipulação
- *Java Collections framework*
  - Arquitetura unificada para representar e manipular coleções, de forma independente dos detalhes de sua representação

125

## Coleções

- O programador simplesmente utiliza as estruturas de dados sem se preocupar com a maneira como são implementadas.
- Vantagens:
  - Reutilização de código
  - Desempenho superior
    - Maior velocidade de execução
    - Algoritmos otimizados

126

## Coleções

- Coleções:
  - De forma simplificada, são objetos capazes de armazenar conjuntos de referências para outros objetos
    - Listas, pilhas, filas, conjuntos, mapas, etc
  - Correspondem a classes oferecidas na biblioteca padrão de Java

127

## Implementações de Uso Geral

	Tabela hash	Arranjo variável	Árvore balanceada	Lista encadeada	Tabela hash + Lista encadeada
<b>Set</b>	HashSet		TreeSet		LinkedHashSet
<b>List</b>		ArrayList		LinkedList	
<b>Deque</b>		ArrayDeque		LinkedList	
<b>Map</b>	HashMap		TreeMap		LinkedHashMap

128



## Listas

- Uma lista é uma coleção linear de elementos que podem ser percorridos sequencialmente e permite inserção e remoção de elementos em qualquer posição
- Conceitualmente, não possui um tamanho máximo

129

## Listas

- Algumas operações:
  - `add(indice, objeto)` adiciona um objeto na posição do índice
  - `add(objeto)` adiciona um objeto na posição final da lista
  - `get(indice)` retorna o objeto armazenado na posição do índice indicado
  - `remove(indice)` remove e retorna o objeto armazenado na posição do índice indicado

130

## Listas

- Algumas operações:
  - `clear()` limpa a lista
  - `isEmpty()` retorna verdadeiro se a lista está vazia
  - `size()` retorna o número de elementos da lista

131

## Listas

- Duas implementações usuais para listas são as classes `ArrayList<E>` e `LinkedList<E>`
  - Implementações com performance diferente para operações diferentes
  - Vantagem:
    - Escolhe-se o tipo de estrutura conforme a necessidade da aplicação, porém, a forma de usá-las é exatamente a mesma
- Declaração:
  - Devemos informar o tipo dos elementos da lista ao declararmos uma coleção (genéricos)
  - `ArrayList<Tipo> umaLista = new ArrayList<Tipo>();`
  - `LinkedList<Tipo> umaLista = new LinkedList<Tipo>();`

132

## Mapas

- Um mapa é uma coleção que associa chaves a valores
- As chaves são valores sem repetição e são utilizadas como mecanismos de busca ao valor associado armazenado na coleção
- Um exemplo prático seria o cadastro de contatos de um celular
  - O nome do contato seria a chave
  - O objeto que guarda os telefone seria o valor
- Outro exemplo são os dicionários

133

## Mapas

- Algumas operações:
  - `put(chave, valor)` adiciona um valor associado a respectiva chave
  - `remove(chave)` remove e retorna o valor associado a chave indicada
  - `get(chave)` retorna o valor associado a chave indicada
  - `containsKey(chave)` retorna true se o mapa contém a chave
  - `containsValue(valor)` retorna true se o mapa contém o valor

134

## Mapas

- Algumas operações:
  - `size()` retorna o número de pares chave-valor armazenados
  - `isEmpty()` retorna true se o mapa está vazio
  - `keySet()` retorna um conjunto contendo todas as chaves do mapa
  - `values()` retorna uma coleção contendo todos os valores do mapa

135

## Mapas

- Duas implementações usuais para listas são as classes `HashMap<K, V>` e `TreeMap<K, V>`
  - Vantagem:
    - Escolhe-se o tipo de estrutura conforme a necessidade da aplicação, porém, a forma de usá-las é exatamente a mesma
- Declaração:
  - Devemos informar o tipo das chaves e valores ao declararmos uma coleção (genéricos)
  - ```
HashMap<String,Integer> mapa = new HashMap<String,Integer>();
```

136

# GENÉRICOS

137

## Genéricos

- Programação genérica consiste na criação de estruturas de programação que podem ser usadas com tipos de dados diferentes
- Exemplo: a classe `ArrayList<E>` é genérica
  - Estrutura de dados que pode ser instanciada para coleções de diferentes tipos de dados
  - O tipo é verificado durante a compilação do programa

138

## Genéricos

- Permitem criar elementos com tipos parametrizáveis
  - Fornecem uma maneira de comunicar o tipo de uma coleção ou atributo ao compilador
  - Verificação em tempo de compilação
  - Quando o compilador conhece o tipo do elemento, ele pode verificar se o mesmo está sendo usado corretamente e pode inserir *casts* corretamente
    - Evitam a escrita de código repetitivo e sujeito a erros de execução resultante do uso excessivo de conversores de tipo

139

## Genéricos

- Uma **classe genérica** terá uma ou mais variáveis de tipo
  - Parâmetros de tipo são declarados entre < e > ao lado do nome da classe
  - Uma vez declarado, um parâmetro de tipo pode ser usado no lugar de qualquer tipo de dado (declaração de variáveis e atributos, parâmetros e valores de retorno)
- Convenção: Usam-se letras maiúsculas individuais para especificar parâmetros de tipo

| Nome da variável de tipo | Significado             |
|--------------------------|-------------------------|
| E                        | Elemento de uma coleção |
| K                        | Chave de um mapa        |
| V                        | Valor em um mapa        |
| T                        | Tipo genérico           |
| S, U                     | Tipos adicionais        |

140

## Genéricos

- Como instanciar?
- `NomeClasseGenerica<Tipo1,Tipo2,...> n = new NomeClasseGenerica<Tipo1,Tipo2,...>();`
- Observação: a ausência do parâmetro de tipo em uma classe genérica implica na utilização do tipo *Object* como default

141

## Genéricos

- Exemplo:

```
public class Par<T,U> {
    private T componente1;
    private U componente2;
    public Par(T componente1, U componente2) {
        this.componente1 = componente1;
        this.componente2 = componente2;
    }
    ...
}
```

142

## Genéricos

- Um **método genérico** terá um ou mais variáveis de tipos que são independentes de um tipo genérico associado à classe genérica
  - O escopo do tipo genérico está limitado ao escopo do método
- O tipo genérico é declarado entre **< e >** antes do tipo de retorno do método

143

## Genéricos

- Exemplo:

```
public class Util {
    public static <T,U> boolean compare(Par<T, U> p1, Par<T, U> p2)
    {
        return p1.getComponente1().equals(p2.getComponente1()) &&
        p1.getComponente2().equals(p2.getComponente2());
    }
}
```

144



## Genéricos

- Java não cria um tipo específico para cada instância de uma estrutura genérica
- Durante a compilação as anotações entre "<" e ">" são **apagadas**, e ocorre uma tradução para código Java tradicional com os tipos e *casts* adequados

145

## Genéricos - Restrições de Tipo

- É possível criar genéricos limitados a uma certa "família" de classes
- Exemplo:
  - `public class MinhaLista<E extends Produto> {...}`
  - O exemplo define uma classe *MinhaLista* que pode conter quaisquer elementos cujo tipo seja subclasse ou implementação de *Produto*
  - Não importando se *Produto* é uma classe ou interface usa-se a palavra reservada *extends*
- Este tipo de restrição é chamado de "limite superior" (*upper bounds*)

146

## Genéricos - Restrições de Tipo

- Um parâmetro de tipo pode possuir mais de uma restrição, separadas por **&**
- Exemplo:
  - `<T extends B1 & B2 & B3>`
- Se um dos argumentos de tipo for uma classe, ela deve ser informada como primeiro elemento (no exemplo, será na posição B1)

147

## Genéricos - Restrições de Tipo

- É possível criar também restrições de limite inferior (*lower bounds*)
- Exemplo:
  - `public class MinhaLista<E super Enlatado> {...}`
  - O exemplo apresenta uma lista cujos elementos devem ser *Enlatado* ou superclasses de *Enlatado*
  - Por exemplo, se *Enlatado* é derivado de *Produto*, então *Produto* é um tipo de elemento aceito na coleção

148

## Genéricos - Coringa

- O símbolo **?** é chamado de coringa (*wildcard*) quando utilizado com genéricos
- É muito utilizado com método genéricos a fim de permitir uma construção mais flexível sobre os tipos

149

## Genéricos - Coringa

- Exemplo: coringa com limite superior
  - `public static void process(List<? extends Number> umaLista) {...}`
  - O método irá aceitar listas de tipos com o *Number*, *Integer*, *Double*, *Float*, ou seja, *Number* e qualquer uma de suas subclasses

150

## Genéricos - Coringa

- Exemplo: coringa sem limite
  - `public static void imprime(List<?> umaLista) {...}`
  - O método irá aceitar listas de “tipos desconhecidos”, ou seja, de qualquer tipo
  - Na prática, significa que o método somente utilizará funcionalidades que estão disponíveis na classe *Object*, ou que usará métodos da classe genérica que não dependem do tipo do parâmetro

151

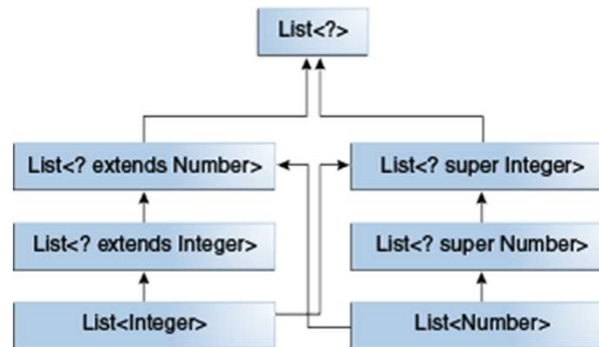
## Genéricos - Coringa

- Exemplo: coringa com limite inferior
  - `public static void process(List<? super Integer> umaLista) {...}`
  - O método irá aceitar listas de tipos com *Integer*, *Number*, *Object*, ou seja, *Integer* e qualquer uma de suas superclasses

152

## Genéricos - Coringa

- Relacionamento entre genéricos:



153

# HERANÇA

154

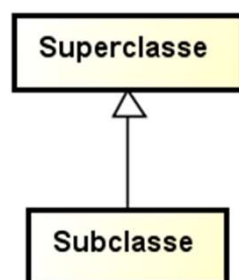
## Herança

- Herança é uma relação de generalização/especialização entre classes
- A ideia central de herança é que novas classes são criadas a partir de classes já existentes
  - Superclasse: classe já existente
  - Subclasse: classe criada a partir da superclasse

155

## Diagrama de Classes UML

- Relacionamento de herança:

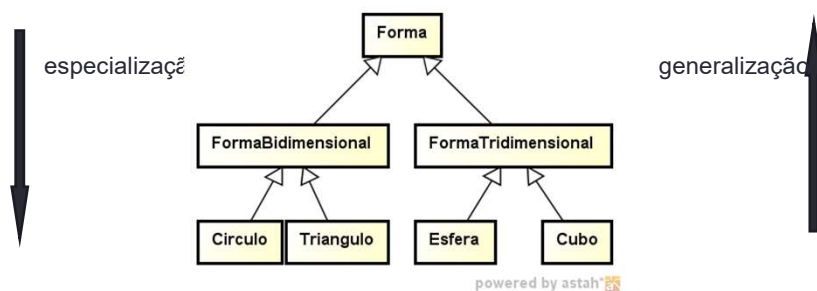


powered by astah\*

156

## Herança

- Herança cria uma estrutura hierárquica
- Ex.: uma hierarquia de classes para formas geométricas
  - Uma forma geométrica pode ser especializada em dois tipos: bidimensional e tridimensional



157

## Herança

- Como implementar herança em Java?
  - Utiliza-se a palavra-chave `extends` para definir herança de classes
  - Somente é possível herdar de uma única superclasse!

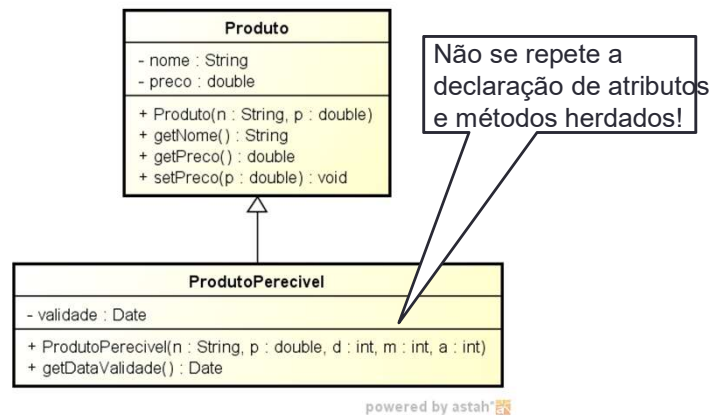
```

class Subclasse extends Superclasse {
    ...
}
  
```

158

## Herança

- Exemplo:



159

## Herança

- Exemplo:

```

public class Produto{...}
public class ProdutoPercivel extends
    Produto{...}
  
```

160



## Herança

- Ao definir atributos da subclasse:
  - Podemos herdar os atributos da superclasse
    - Todos os atributos da superclasse são herdados automaticamente
      - Ex.: atributos nome e preço de Produto
  - Podemos definir novos atributos
    - Evitar criar atributos com o mesmo nome de atributos herdados
      - Ex.: atributo validade de ProdutoPercivel

161

## Herança

- Exemplo:

```
public class Produto{  
    private String nome;  
    private double preco;  
    public Produto(String n, double p){  
        nome = n;  
        preco = p;  
    }  
    ...  
}
```

162

## Herança

- Exemplo:

```
public class ProdutoPercivel extends Produto{
    private Date validade;
    public ProdutoPercivel(String n, double p,
        int d, int m, int a){
        nome = n;
        preco = p;
        GregorianCalendar cal = new
        GregorianCalendar(a,m,d);
        validade = cal.getTime();
    }
    ...
}
```

Não é a forma correta de  
Inicializar os atributos  
herdados!

163

## Herança

- Subclasse tem acesso a todos os métodos públicos da superclasse
- Logo...
- Podemos utilizar o construtor da superclasse para inicializar os atributos herdados
  - Utiliza-se `super()`
    - Deve ser o primeiro comando do construtor da subclasse!
    - Sempre é utilizado!

164

## Herança

- Exemplo:

```
public class ProdutoPercivel extends Produto{
    private Date validade;
    public ProdutoPercivel(String n, double p,
        int d, int m, int a){
        super(n,p) ;
        GregorianCalendar cal = new
        GregorianCalendar(a,m,d);
        validade = cal.getTime();
    }
    ...
}
```

165

## Herança

- Ao definir métodos da subclasse:
  - Podemos herdar os métodos da superclasse
    - Os métodos são herdados automaticamente
      - Ex.: métodos getNome() e getPreco() de Produto
  - Podemos definir novos métodos
    - Ex.: método getDataValidade() de ProdutoPercivel
  - Podemos sobrescrever métodos da superclasse!

166

## Herança

- Modificadores de acesso:
  - *public*: acessível em qualquer classe
  - *private*: acessível somente dentro da própria classe
  - *protected*: acessível dentro da própria classe ou de uma subclasse

167

## Sobrescrita de Métodos

- Uma subclasse pode sobrescrever (“override”) métodos da superclasse
  - Sobrescrita permite completar ou modificar um comportamento herdado
  - Quando um método é referenciado em uma subclasse, a versão escrita para a subclasse é utilizada, ao invés do método na superclasse
  - É possível acessar o método original da superclasse:  
`super.nomeDoMetodo()`

168

## Sobrescrita de Métodos

- Um exemplo de sobrescrita são os métodos herdados da classe *Object*
  - Em Java, todas as classes herdam diretamente ou indiretamente da classe *Object*
  - *Object* é o topo da hierarquia de classes em Java
  - Toda classe criada sem explicitar uma superclasse, herda implicitamente da superclasse *Object*

169

## Sobrescrita de Métodos

- Alguns métodos herdados de *Object*:
  - `String toString()` retorna uma representação de string do objeto
    - Usualmente utilizado para realizar a depuração de programas
    - Também é chamado implicitamente quando um objeto é utilizado em um contexto que uma string era esperada
    - Implementação original retorna o nome da classe e o código *hash* do objeto
  - `boolean equals(Object outro)` testa se o objeto possui o mesmo estado que outro objeto
- Estes métodos são usualmente sobrescritos se forem utilizados em uma subclasse!

170

## Sobrescrita de Métodos

- A classe Produto pode sobrescrever o método `toString()` de `Object`:

```
public String toString(){
    return super.toString()
        + "[nome=" + nome + ", "
        + "preco=" + preco + " ]";
}
```

Método herdado  
de Object

171

## Sobrescrita de Métodos

- A classe `ProdutoPercivel` pode sobrescrever o método `toString()` de `Produto`:

```
public String toString(){
    return super.toString()
        + "[validade=" +
        DateFormat.getDateInstance().format(validad
        e) + " ]";
}
```

Método herdado  
de Produto

172

## Controle da Herança

- Modificador `final`
  - Um método pode ser marcado como `final` para impedir que seja sobrescrito
    - `public final void meuMetodo(){...}`
  - Uma classe pode ser marcada como `final` para impedir que possa ser estendida com subclasses
    - `public final class MinhaClasse{...}`

173

## Herança e Polimorfismo

- “Polimorfismo é a característica única de linguagens orientadas a objetos que permite que diferentes objetos respondam a mesma mensagem cada um a sua maneira.”

174

## Herança e Polimorfismo (Variáveis)

- A linguagem Java permite a utilização de variáveis com polimorfismo
  - Uma mesma variável permite referência a objetos de tipos diferentes
  - Os tipos permitidos são de uma determinada classe e todas as suas subclasses

175

## Herança e Polimorfismo (Variáveis)

- Exemplo:

```

Produto p1 = new
    ProdutoPecivel("a",1.9,1,12,2011);    correto
ProdutoPecivel p2 = new Produto("a",1.9); erro
                                           compilação

Produto psuper;
ProdutoPecivel psub;
ProdutoPecivel p3 = new
    ProdutoPecivel("a",1.9,1,12,2011);

psuper = p3;    correto
psub = psuper;  erro
psub = (ProdutoPecivel) psuper; correto

```

176



## Herança e Polimorfismo (Variáveis)

- Java possui o operador *instanceof* que permitir verificar o tipo de uma instância
  - Retorna *true* se a expressão da esquerda é um objeto que possui compatibilidade de atribuição com o tipo à sua direita
  - Retorna *false* caso contrário
- Ex.:

```
if (p1 instanceof Produto) {  
    ...  
}
```

177

## Herança e Polimorfismo (Métodos)

- Em Java podemos utilizar métodos com polimorfismo
  - Significa que uma mesma operação pode ser definida em diversas classes, cada uma implementando a operação de uma maneira própria
  - Utiliza como base a sobrescrita de métodos

178

## Herança e Polimorfismo (Métodos)

- Exemplo:
  - Qual a saída no console?

```
Produto p = new ProdutoPerecivel("a",1.9,1,12,2011);  
System.out.println(p);
```

179

## Classes e Métodos Abstratos

- Em uma hierarquia de classe, quanto mais alta a classe na hierarquia, mais abstrata é sua definição
  - Uma classe no topo da hierarquia define o comportamento e atributos que são comuns a todas as classes
  - Em alguns casos, a classe nem precisa ser instanciada alguma vez e cumpre apenas o papel de ser um repositório de comportamentos e atributos em comum

180

## Classes e Métodos Abstratos

- Classes abstratas são classes que não podem ser instanciadas
- São utilizadas apenas para permitir a derivação de novas classes
- Identificamos uma classe como abstrata pelo modificador `abstract`

```
public abstract class MinhaClasse{...}
```

- Em uma classe abstrata, um ou mais métodos podem ser declarados sem o código de implementação
  - São os métodos abstratos

181

## Classes e Métodos Abstratos

- Métodos abstratos são métodos sem código de implementação
  - São prefixados pela palavra `abstract`
  - Não apresentam um corpo. Sua declaração termina com “;” após a declaração dos parâmetros

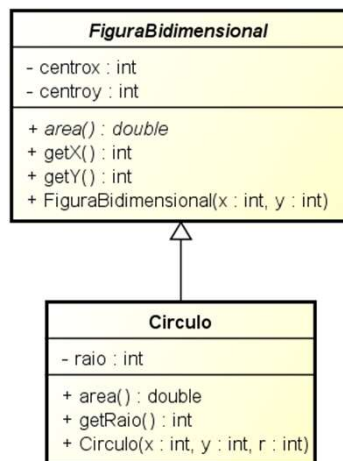
```
public abstract void metodo(int p);
```

- Um método abstrato indica que a classe não implementa aquele método e que ele deve ser obrigatoriamente implementado nas classes derivadas, pois é um comportamento comum das subclasses

182

## Classes e Métodos Abstratos

- Exemplo



powered by astah®

183

## Classes e Métodos Abstratos

- Exemplo:

```

public abstract class FiguraBidimensional{
    public FiguraBidimensional(int x, int y) {
        centrox = x;
        centroy = y;
    }
    public abstract double area();
    ...
}
  
```

184

## Classes e Métodos Abstratos

- Exemplo:

```
public class Circulo extends
    FiguraBidimensional{
    public Circulo(int x, int y, int r) {
        super(x,y);
        raio = r;
    }
    public double area(){
        ...
    }
    ...
}
```

185

## INTERFACES

186

## Interfaces

- Interfaces são estruturas que podem ser utilizadas para separar a especificação do comportamento de um objeto de sua implementação concreta
  - Trazem a especificação do conjunto de operações públicas sem código de implementação
  - Ao contrário das classes, define um novo tipo sem fornecer a implementação
- Dessa forma a interface age como um contrato, o qual define explicitamente quais métodos uma classe deve obrigatoriamente implementar

187

## Interfaces

- Uma interface deve ser implementada por uma classe
  - Uma interface pode ser implementada por diversas classes
    - POLIMORFISMO!!!
  - Uma classe pode implementar diversas interfaces
    - Permite uma classes ser utilizada em diferentes contextos!!!

188

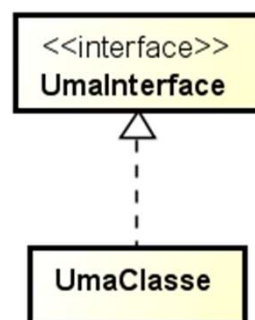
## Interfaces

- Uma interface não pode ser instanciada
  - Não se cria objetos a partir de uma interface
- Uma interface pode estender, via herança, outra interface
  - Permite acrescentar novo comportamento a uma interface já existente

189

## Diagrama de Classes UML

- Relacionamento de realização de interfaces:



powered by astah\*

190

## Interfaces

- Uma interface em Java é essencialmente uma coleção de constantes, métodos abstratos e tipos (como enumeradores) declarados internamente
  - Métodos são sempre implicitamente `public abstract`
  - Atributos são sempre implicitamente `public static final`
  - Não é necessário repetir a declaração desses modificadores

191

## Interfaces

- Definindo interfaces:
  - Interfaces são implementadas através da palavra chave `interface`:

```
public interface MinhaInterface {  
    ...  
}
```

192



## Interfaces

- Para utilizar uma interface:
  - Implementa-se a mesma em uma classe
  - Quando se declara que a classe implementa a interface, deve-se escrever o código para cada um dos métodos declarados nesta interface

```
public class MinhaClasse implements MinhaInterface {  
  
    //aqui vem a implementação dos métodos  
  
}
```

193

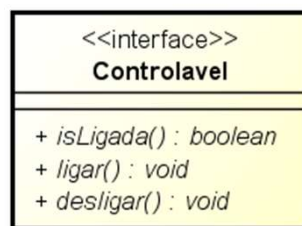
## Interfaces e Polimorfismo

- Usando Interfaces se pode trabalhar com polimorfismo
  - Uma referência do tipo da Interface pode apontar para qualquer objeto que implementa aquela Interface
  - Criando uma referência da Interface, é possível invocar os métodos definidos na Interface, de forma independente da classe do objeto utilizado

194

## Interfaces e Polimorfismo

- Exemplo:
  - Controlador de uma casa é capaz de controlar qualquer dispositivo que atenda as seguintes características

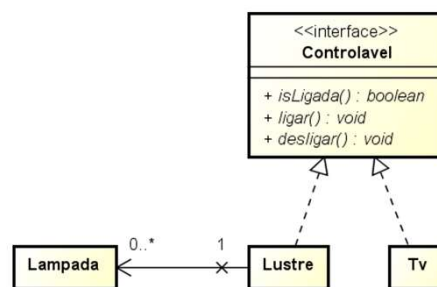


powered by astah®

195

## Interfaces e Polimorfismo

- Exemplo:
  - Um objeto Lustre e Tv podem ser controlados por esse controlador pois implementam a interface necessária



powered by astah®

196

## Estudo de Caso - Listas

- As operações disponíveis sobre listas estão definidas na interface `List<E>`
- A documentação da API de Java lista todas as operações permitidas sobre uma lista

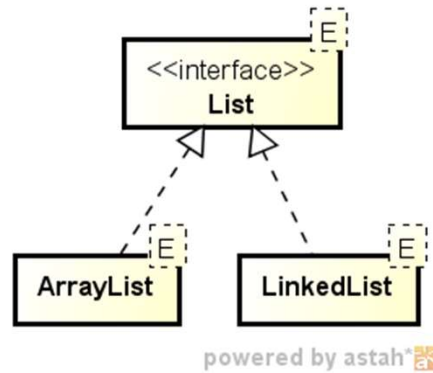
197

## Estudo de Caso - Listas

- Duas implementações usuais da interface `List<E>` são as classes `ArrayList<E>` e `LinkedList<E>`
  - Implementações com performance diferente para operações diferentes
- Declaração:
  - Devemos informar o tipo dos elementos da lista ao declararmos uma coleção (genéricos)
  - `List<Tipo> umaLista = new ArrayList<Tipo>();`
  - `List<Tipo> umaLista = new LinkedList<Tipo>();`

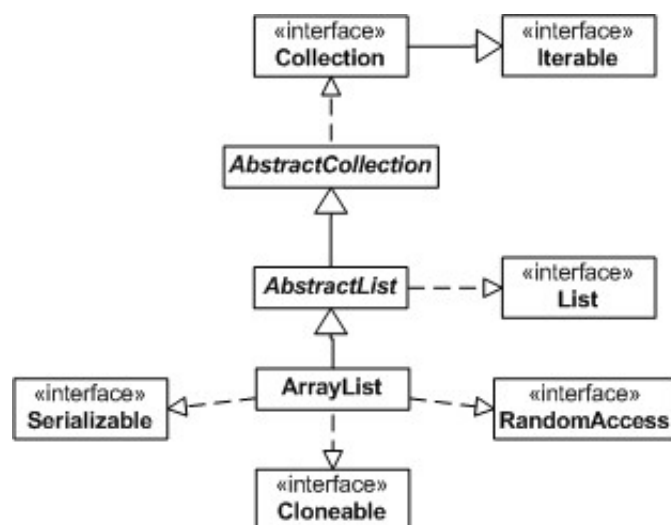
198

## Estudo de Caso - Listas



199

## Estudo de Caso - Listas



200

## Estudo de Caso - Ordenação

- A ordenação é um método bastante utilizado.
    - Java fornece vários métodos já implementados para ordenar listas de objetos
      - POLIMORFISMO!!!
    - Ex.: classe `Collections`, método de classe `sort(List<T>)`
- ```
List lista = new ArrayList();
...
Collections.sort(lista);
```
- Mas como?
    - Os métodos de ordenação já estão prontos antes mesmo de definirmos que tipos de objetos vamos ordenar?
    - Como os algoritmos de ordenação sabem que `objeto1 <= objeto2` ?

201

## Estudo de Caso - Ordenação

- Alguns algoritmos de ordenação trabalham sobre objetos de classes que implementam a interface `Comparable<T>`
  - Essa interface especifica o método que os algoritmos de ordenação utilizam para saber quando um objeto é menor, igual ou maior que outro
  - Quando criamos uma nova classe, podemos implementar o método da interface `Comparable<T>` para podermos utilizar os algoritmos de ordenação de Java
    - Devemos implementar o método `compareTo(objeto)`
  - O código de comparação fica isolado dos objetos que implementam a ordenação

202

## Estudo de Caso - Ordenação

- Como funciona:
  - Disponível na API Java
    - `public interface Comparable<T>`
      - declara um método chamado `compareTo(T)`, que deve ser implementado por qualquer classe cujos objetos possam ser ordenados
    - `public class Collections`
      - contem o método `sort(List<T>)`, capaz de ordenar uma lista de objetos. Para ordenar os objetos, este método chama o método `compareTo()`
  - Criados pelo usuário
    - `public class MinhaClasse implements Comparable<MinhaClasse>`
      - contem a implementação do método abstrato `compareTo()` que compara dois objetos da classe

203

## Estudo de Caso - Ordenação

```
public interface Comparable<T>{
    //compareTo(T obj2)
    //compara este objeto com outro
    //Retorna :
    // 0 se este objeto igual a obj2
    // valor < 0 se este objeto menor que obj2
    // valor > 0 se este objeto maior que obj2
    int compareTo(T obj2);
}
```

204

## Estudo de Caso - Ordenação

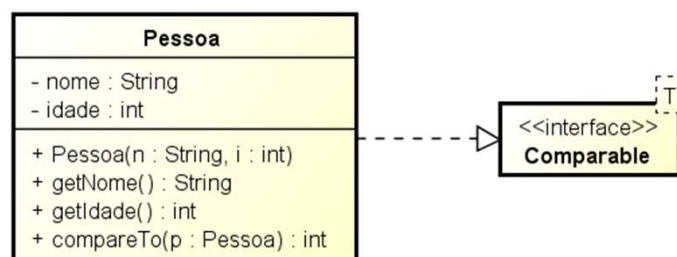
- Ex.: classe String
  - Esta classe já implementa a interface Comparable
  - Logo é possível ordenar listas contendo strings

```
List<String> nomes = new ArrayList<String>();
nomes.add("Julio Machado");
nomes.add("Isabel Manssour");
nomes.add("Bernardo Copstein");
Collections.sort(nomes);
```

205

## Estudo de Caso - Ordenação

- Exemplo:
  - Comparar pessoas pelo nome ou pela idade?



powered by astah®

206

## Estudo de Caso - Ordenação

- Suponha que seja necessário ordenar uma lista de pessoas tanto pelo nome quanto pela idade
- Existe um segundo método de ordenação chamado *sort(List, Comparator)*
  - Este método ordena uma lista de acordo com os critérios de ordenação fornecidos pelo objeto que implementa a interface *Comparator<T>*

207

## Estudo de Caso - Ordenação

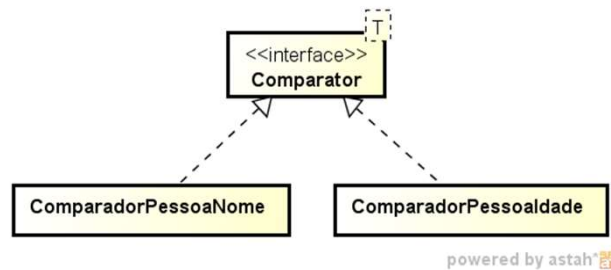
- A interface *Comparator<T>* requer os seguintes métodos:  
`int compare(T o1, T o2)`  
`boolean equals(Object obj)`
- Quem implementa a interface não é mais o próprio objeto da ordenação!

208



## Estudo de Caso - Ordenação

- Exemplo:
  - Dois comparadores diferentes, um para nome e outro para idade



209

## Padrão *Strategy*

- Vantagens:
  - Mostra como fornecer variações de um algoritmo

210

## Padrão *Strategy*

- Contexto:
  - Classe (*context*) se beneficia de diferentes implementações de um determinado algoritmo.
  - Clientes desejam fornecer diferentes versões do algoritmo.

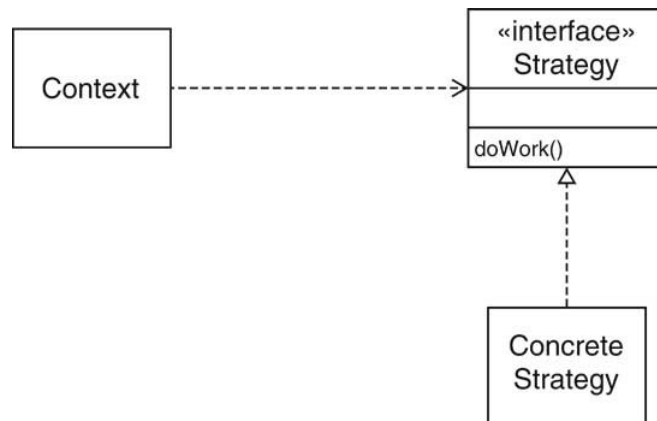
211

## Padrão *Strategy*

- Solução:
  - Define uma interface (*strategy*) que abstrai as operações do algoritmo.
  - Classes “concretas” que fazem parte do padrão precisam implementar esta interface, cada qual com uma versão do algoritmo.
  - Clientes passam instância de classe concreta para a classe de contexto.
  - Sempre que um algoritmo precisa ser executado, a classe de contexto chama os métodos da interface.

212

## Padrão Strategy



213

## Padrão Strategy

Nome no Padrão	Nome real (sorting)
Context	Collections
Strategy	Comparator
ConcreteStrategy	classe que implementa Comparator
doWork()	compare()

214

## Estudo de Caso - Iteração

- Operação típica sobre uma lista:
  - Percorrer seus elementos em ordem, um de cada vez, e realizar uma operação sobre os elementos

215

## Estudo de Caso - Iteração

- Observe a implementação dessa operação

```
for(int i=0; i<lista.size(); i++) {  
    Object obj = lista.get(i);  
    //faz algo com obj  
}
```

- diferente?

- Lista com arranjo
- Lista encadeada

Ineficiente!

216

## Estudo de Caso - Iteração

- Iterador
  - Padrão de projeto de software que abstrai o processo de iteração sobre uma coleção de elementos
  - Em Java é usualmente utilizado via comando *for* do tipo “para-cada”

217

## Estudo de Caso - Iteração

- Observe a implementação dessa operação

```
Iterator it = lista.iterator();
while(it.hasNext()) {
    Object obj = it.next();
    //faz algo com obj
}
```

218

## Estudo de Caso - Iteração

- Um iterador (em Java) define três métodos:
  - *hasNext*: testa se existe elementos remanescentes no iterador
  - *next*: retorna o próximo elemento do iterador
  - *remove*: remove o último elemento retornado
- Observação:
  - O método *remove* usualmente não é implementado caso o iterador seja somente para percorrer a coleção

219

## Estudo de Caso - Iteração

- Java define uma interface para iteradores: *Iterator*

```
public interface Iterator<E> {
    // Returns true if the iteration has more elements.
    public boolean hasNext();
    // Returns the next element in the iteration.
    public E next();
    // Removes from the underlying collection the last
    // element returned by this iterator (optional).
    public void remove();
}
```

- O método *remove* não faz muito sentido dentro do contexto, de maneira que sua implementação normalmente prevê o lançamento da exceção *UnsupportedOperationException*.

220

## Estudo de Caso - Iteração

- A implementação da interface `Iterator` normalmente é feita a partir de uma classe interna
- Dessa forma evita-se quebrar o encapsulamento da classe
- Uma classe pode possuir diferentes tipos de iteradores

221

## Estudo de Caso - Iteração

- Java define a interface *Iterable* de maneira que todas as coleções de Java tratam os iteradores da mesma maneira

```
public interface Iterable<T>{  
    public Iterator<T> iterator();  
}
```

- Desde o Java 6, o comando `for` (em sua versão “paracada”) é capaz de iterar sobre qualquer coleção que implemente `Iterable`

222

## Padrão *Iterator*

- Vantagens:
  - O iterator não expõe a estrutura interna da coleção.
  - O usuário da classe não necessita conhecimento de como percorrer a coleção.
  - Simplifica a interface da classe.
    - Pense como seriam métodos para inserir ou remover elementos em qualquer posição de uma lista encadeada...

223

## Padrão *Iterator*

- Contexto:
  - Um objeto (*aggregate*) contém outros objetos (*elements*)
  - Clientes (métodos que usam o *aggregate*) precisam acessar os elementos
  - O *aggregate* não deve expor a sua estrutura interna
  - Podem existir múltiplos clientes que necessitam de acesso simultâneo

224

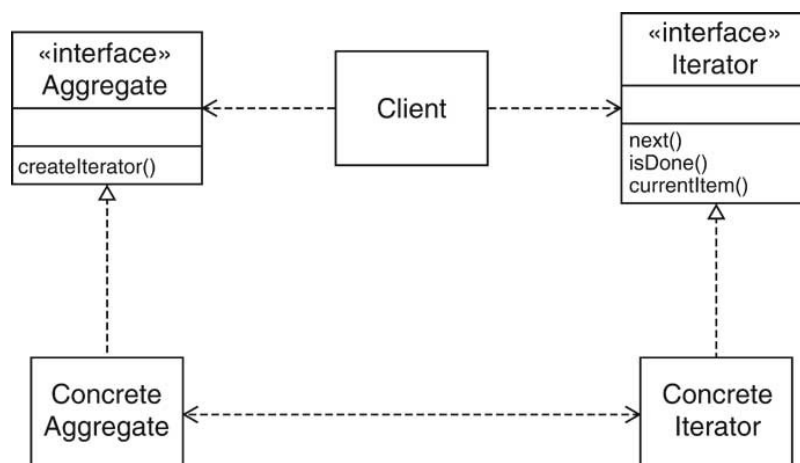


## Padrão *Iterator*

- Solução:
  - Criar uma classe *iterator* que busca um elemento por vez
  - Cada *iterator* armazena a posição do próximo elemento a ser recuperado
  - Caso existam múltiplas variações das classes *aggregate* e *iterator*, é melhor que elas implementem interfaces comuns (o cliente conhece apenas a interface)

225

## Padrão *Iterator*



226

## Padrão *Iterator*

Nome no Padrão	Nome real (sorting)
Agregate	List
Iterator	Iterator
ConcreteIterator	classe que implementa Iterator
ConcreteAgregate	ArrayList

227

## TRATAMENTO DE EXCEÇÕES

228

## Exceções

- Quando um método encontra uma situação anormal, ele informa tal anormalidade pelo lançamento (geração) de uma exceção
- Ex.: o método *Integer.parseInt(String s)*, para converter strings para inteiros, irá lançar a exceção *NumberFormatException* se a *String* não possui somente dígitos de um número inteiro

229

## Tipos de Exceções

- Java possui duas categorias básicas:
  - Exceções verificadas
    - O compilador verifica se o código lida com a exceção de forma explícita
    - Usualmente relacionadas com condições externas, fora do controle do programador, mas que devem ser tratadas corretamente
    - Subclasses de *Exception*
    - Ex.: *IOException*
  - Exceções não-verificadas
    - O compilador não verifica se seu código trata a exceção
    - Usualmente correspondem a falhas de lógica de programação
    - Subclasses de *RuntimeException*
    - Ex.: *NumberFormatException*

230

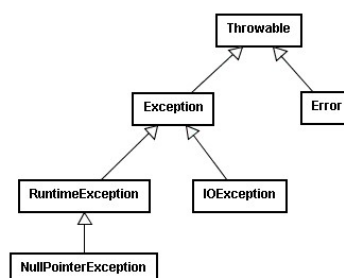
## Tratamento de Exceções

- O tratamento de exceções de Java envolve vários conceitos importantes:
  - Lançamento (*throw*): quando um método encontra uma situação anormal, ele informa tal anormalidade pelo lançamento (geração) de uma exceção.
    - Ex.: o método `Integer.parseInt(String s)`, para converter strings para inteiros, irá lançar a exceção `NumberFormatException` se a `String` não possui somente dígitos de um número inteiro.
  - Captura (*try-catch*): quando um método tenta detectar uma situação anormal, ele captura essa exceção, possivelmente indicando que irá realizar o tratamento do problema encontrado.
    - Ex.: um método que faz uso de `Integer.parseInt(String s)` pode querer capturar essa exceção para evitar problemas no programa.

231

## Tratamento de Exceções

- Java utiliza herança para organizar os tipos de exceções disponíveis
  - Todas as exceções herdam, de alguma forma, da classe `Throwable`



232

## Tratamento de Exceções

- Subclasses de *RuntimeException* são exceções não-verificadas
- Subclasses de *Exception* que não são subclasses de *RuntimeException* são exceções verificadas

233

## Capturando Exceções

- Para capturar e tratar exceções, utiliza-se o bloco de comandos *try...catch...finally*

```
try
{
    // código que pode gerar exceção
}
catch (Exception e)
{
    // código que trata exceção
}
finally
{
    // tratamento geral
}
```

234

## Capturando Exceções

- O comando `try/catch/finally` suporta o tratamento de exceções:
  - No bloco **try** estão colocados os comandos que podem provocar o lançamento de uma exceção.
  - Essas exceções são capturadas em um ou mais comandos **catch**, colocados após o bloco `try`.
  - O comando **finally** contém código a ser executado, independente da ocorrência de exceções. É opcional, mas quando presente, é sempre executado.
- Logo, para capturar uma exceção:
  - Protegemos o código que contém métodos que poderiam levantar uma exceção dentro de um bloco `try`.
  - Tratamos uma exceção dentro do bloco `catch` correspondente àquela exceção.

235

## Capturando Exceções

- Ordem de execução:
  - Se o bloco `try` completa a computação normalmente, a execução continua no primeiro comando após o bloco `try-catch`. Se existir um bloco `finally`, ele é executado antes.
  - Se ocorrer uma exceção durante a execução do bloco `try`, a execução para no exato ponto de origem da exceção.
  - A máquina virtual procura pelo primeiro bloco `catch` que nomeia a exceção ocorrida.
    - Se é encontrado, o controle da execução é repassado ao código do bloco. Ao terminar sem erros, a execução continua após o bloco `try-catch`, se existir um bloco `finally`, ele é executado antes.
    - Se não é encontrado, a exceção é sinalizada como não capturada e é repassada para o código chamador imediatamente superior.


236

## Capturando Exceções

- Uma vez lançada, uma exceção capturada no bloco `try` procura por uma cláusula `catch` capaz de referenciá-la e tratá-la.

- Ex.:

```
int quantidade;
String s = JOptionPane.showInputDialog("Digite um
valor inteiro:");
try {
    // método parseInt() pode gerar exceção
    quantidade = Integer.parseInt(s);
    System.out.println(quantidade);
}
catch (NumberFormatException e) {
    // código para tratar a exceção
    System.out.println("Erro de conversão");
}
```



237

## Capturando Exceções

- A cláusula `finally` é utilizada para forçar a execução de um bloco de código, mesmo que não ocorra uma exceção
  - Pode ser utilizada com ou sem o bloco `catch`
- A cláusula `finally` é executada nas seguintes condições:
  - fim normal do método
  - devido a uma instrução `return` ou `break`
  - caso uma exceção tenha sido gerada

238

## Try Com Recursos

- A partir do Java 7, um novo bloco *try* está disponível
- Um recurso é um objeto que deve ser “fechado” após o seu uso
  - Arquivos, conexões de rede, fluxos, etc
  - São objetos que implementam a interface *AutoCloseable*, com o método *void close()*

239

## Try Com Recursos

- Um bloco try-com-recursos pode utilizar vários recursos (separados por ;)
- Ex.:

```
static String readFirstLineFromFile(String path) throws
IOException {
    try (BufferedReader br =
        new BufferedReader(new FileReader(path))) {
        return br.readLine();
    }
}
```

240



## Repassando Exceções

- Se um código utiliza métodos que geram exceções verificadas, mas não as trata, então deve repassá-las adiante via a cláusula *throws*
- Ex.:

```
public void lerArquivo(String arquivo) throws IOException {
    ...
    BufferedReader in = new BufferedReader(new FileReader(arquivo));
    String firstline = in.readLine();
    in.close();
    ...
}
```

Este bloco de código não captura e trata a exceção

241

## Lançando Exceções

- Para lançar uma exceção dentro de um método que estamos desenvolvendo:
  - Instanciar um objeto do tipo da exceção desejada
    - Ex.: `NullPointerException e = new NullPointerException("mensagem de erro");`
  - Lançar a exceção via comando *throw*
    - Ex.: `throw e;`

242

## Exemplo: Classe Circulo

Circulo
-centrox:int -centroy:int -raio:int
+Circulo(x:int, y:int, r:int) +area():double +circunferencia():double +diametro():int

- Construtor da classe deve validar as entradas
  - Para os valores do centro somente aceitar valores que não sejam negativos
  - Para o valor do raio somente aceitar valor positivo

243

## Exemplo: Classe Circulo

```
public class Circulo {
    private int centrox;
    private int centroy;
    private int raio;

    public Circulo(int x, int y, int r){
        if (x < 0) {
            IllegalArgumentException excecao = new
            IllegalArgumentException("Valor do centrox negativo");
            throw excecao;
        }
        else centrox = x;
        if (y < 0)...
    }
    ...
}
```

244

## Cláusula *throws*

- Métodos que geram exceções verificadas devem obrigatoriamente declará-las no cabeçalho do método via cláusula *throws*

- Lista de exceções separadas por vírgulas

- Ex.:

```
public void lerArquivo(String nomeArq)
    throws FileNotFoundException {...}
```

245

## Cláusula *throws*

- Métodos que geram exceções não-verificadas podem ou não declará-las no cabeçalho do método via cláusula *throws*

- Lista de exceções separadas por vírgulas

- Ex.:

```
public Circulo(int x, int y, int r) throws
    IllegalArgumentException {...}
```

246

## Novas Exceções

- Caso os tipos de exceções fornecidos na API de Java não sejam suficientes, criam-se novas classes através do mecanismo de herança
- Novos tipos de exceções são criados através da extensão de uma classe já existente
  - *Exception* para exceções verificadas

247

## Novas Exceções

- Ao projetar uma classe de exceção, é usual fornecer quatro construtores com os seguintes parâmetros:
  - `()` construtor vazio
  - `(String mensagem)` construtor com a mensagem de erro
  - `(Throwable causa)` construtor com a exceção prévia que causou a exceção
  - `(String mensagem, Throwable causa)` construtor com a mensagem de erro e a exceção prévia que causou a exceção

248

## Novas Exceções

- Exemplo:

```
public class IllegalFormatException extends Exception {  
    public IllegalFormatException() {}  
    public IllegalFormatException(String m) {  
        super(m);  
    }  
    public IllegalFormatException(Throwable c) {  
        super(c);  
    }  
    public IllegalFormatException(String m, Throwable c) {  
        super(m,c);  
    }  
}
```

249

# JAVABEANS

250

## Componentes - Padrões

- Diversos padrões diferentes para componentes:
  - COM, DCOM, COM+, ActiveX, .NET
    - Microsoft
  - JavaBeans, Enterprise JavaBeans
    - Oracle
  - CORBA Common Object Request Broker Architecture
    - OMG

251

## Componentes - Criação

- Nas diversas plataformas, usualmente componentes:
  - Podem ser compostos de diversas classes
  - Exportam propriedades para divulgar informações
  - Implementam métodos que definem seu comportamento
  - Utilizam eventos para comunicação com outros componentes

252

## JavaBeans

- Definição:
  - “JavaBeans is a portable, platform-independent component model written in the Java programming language.”
  - Oracle Tutorial

253

## JavaBeans

- A arquitetura JavaBeans
  - Auxilia a escrita de classes que podem ser tratadas como componentes de grandes sistemas
  - Provê suporte a ferramentas interativas para a composição de sistemas

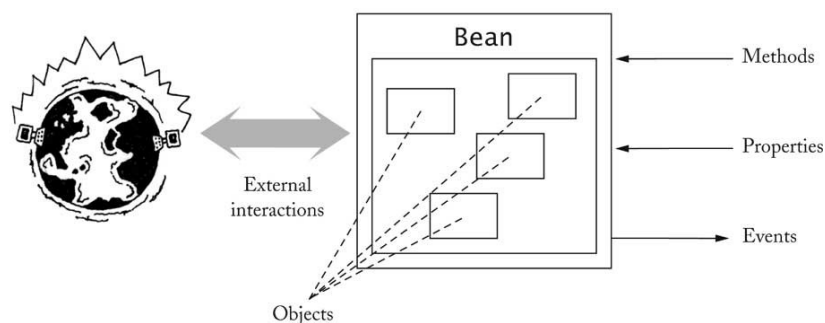
254

## JavaBeans

- Um bean é uma classe java que:
  - Exporta propriedades para permitir a customização do componente em tempo de design
  - Utiliza eventos para se comunicar com outros componentes
  - Implementa métodos
- Exemplos:
  - Componentes AWT e Swing
  - JCalendar
    - <http://www.toedter.com/en/jcalendar/index.html>

255

## JavaBeans



256



## JavaBeans - Criação

- Um bean é composto de uma ou mais classes que são empacotadas em conjunto
  - Usualmente existe uma classe de fachada (padrão Facade)
    - contém os métodos, eventos e propriedades expostos pelo bean
    - faz chamadas para as outras classes do bean

257

## JavaBeans - Criação

- Os pacotes `java.beans` e `java.beans.beancontext` providenciam classes necessárias e úteis para a escrita de *beans*
  - Por exemplo, interface *BeanInfo* provê informações explícitas do bean para as ferramentas de uma IDE, outras informações obtidas via reflexão pela classe *Introspector*
- Um conjunto de convenções deve ser seguido para atribuição de nomes à interface pública de um *bean*
- Qualquer objeto, que se adapte a certas regras básicas pode ser um *bean*
  - Não existe uma classe especial de onde herdar a implementação

258

## JavaBeans - Criação

- Classe:
  - Não há nenhuma restrição quanto ao nome da classe

259

## JavaBeans - Criação

- Métodos:
  - Quaisquer métodos públicos podem ser exportados por um *bean*
    - Exclui-se aqui os métodos relacionados a propriedades e eventos
  - Não há restrição quanto aos nomes dos métodos

260

## JavaBeans - Criação

- Propriedades:
  - É uma parte do estado interno do *bean* que pode ser configurada e/ou consultada programaticamente
  - Suporta vários tipos de propriedades:
    - Simples (*simple*)
    - Indexada (*indexed*)
    - Vinculada (*bound*)
    - Restrita (*constrained*)
  - Também classificadas em:
    - De escrita
    - De leitura
    - De leitura e escrita

261

## JavaBeans - Criação

- Propriedades simples:
  - Propriedade de um único valor cujas alterações independem de outra propriedade
  - Um bean define uma propriedade P do tipo T se tiver métodos de acesso de acordo com os seguintes padrões:
    - Getter – `public T getP()`
    - Getter booleano – `public boolean isP()`
    - Setter – `public void setP(T)`
    - Exceções – podem gerar qualquer tipo

262

## JavaBeans - Criação

```
public class MyBean {
    private String title;
    public MyBean() {
    }
    public String getTitle() {
        return this.title;
    }
    public void setTitle(String title) {
        this.title = title;
    }
}
```

263

## JavaBeans - Criação

- Propriedades indexadas:
  - Propriedade que suporta vários valores
  - Um bean define uma propriedade P do tipo T[] se tiver métodos de acesso de acordo com os seguintes padrões:
    - Getter de array – public T[] getP()
    - Getter de elemento – public T getP(int)
    - Setter de array – public void setP(T[])
    - Setter de elemento – public void setP(int,T)
    - Exceções – podem gerar qualquer tipo, devem gerar `IndexOutOfBoundsException` se o índice for inválido

264

## JavaBeans - Criação

```
public class MyBean {
    private String[] lines;
    public String getLines(int index) {
        return this.lines[index];
    }
    public void setLines(int index, String lines) {
        this.lines[index] = lines;
    }
    public String[] getLines() {
        return this.lines;
    }
    public void setLines(String[] lines) {
        this.lines = lines;
    }
}
```

265

## JavaBeans - Criação

- Propriedades vinculadas:
  - É uma propriedade que gera um evento `PropertyChangeEvent` quando seu valor é alterado através da interface `PropertyChangeListener`
  - Classe utilitária `PropertyChangeSupport` facilita a implementação
- Padrões:
  - Métodos de acesso – getter e setter seguem mesmos padrões para propriedade normal
  - Registro de ouvinte – par de métodos
    - `public void addPropertyChangeListener(PropertyChangeListener)`
    - `public void removePropertyChangeListener(PropertyChangeListener)`

266

## JavaBeans - Criação

- Propriedades vinculadas: (cont)
  - Padrões: (cont)
    - Registro de ouvinte de propriedade identificado – registro de ouvintes para propriedades vinculadas individuais
      - `public void addPropertyChangeListener(String, PropertyChangeListener)`
      - `public void removePropertyChangeListener(String, PropertyChangeListener)`

267

## JavaBeans - Criação

- Propriedades vinculadas: (cont)
  - Padrões: (cont)
    - Registro de ouvinte por propriedade – registro de ouvintes para propriedades P individual não-vinculada
      - `public void addPListener(PropertyChangeListener)`
      - `public void removePListener(PropertyChangeListener)`

268

## JavaBeans - Criação

- Propriedades vinculadas: (cont)
  - Padrões: (cont)
    - Notificação – quando o valor de uma propriedade vinculada for alterado, o bean deve passar um objeto `PropertyChangeEvent` para o método `propertyChange()` de cada objeto `PropertyChangeListener` registrado para o bean ou propriedade vinculada específica

269

## JavaBeans - Criação

```
import java.beans.PropertyChangeListener;
import java.beans.PropertyChangeSupport;
public class MyBean {
    private String title;
    private final PropertyChangeSupport pcs = new
PropertyChangeSupport(this);
    public String getTitle(){
        return this.title;
    }
    public void setTitle( String title ){
        String old = this.title;
        this.title = title;
        this.pcs.firePropertyChange( "title", old, title );
    }
    public void addPropertyChangeListener(PropertyChangeListener
listener){
        this.pcs.addPropertyChangeListener( listener );
    }
    public void
removePropertyChangeListener(PropertyChangeListener listener){
        this.pcs.removePropertyChangeListener( listener );
    }
}
```

270

## JavaBeans - Criação

- Propriedades restritas:
  - É uma propriedade para a qual qualquer alteração pode ser vetada por outros componentes através da interface `VetoableChangeListener`
  - Classe utilitária `VetoableChangeSupport` facilita a implementação
- Padrões:
  - Getter – mesmo que uma propriedade normal
  - Setter – dispara uma exceção `PropertyVetoException` se a alteração for vetada; para um propriedade P do tipo T
    - `public void setP(T) throws PropertyVetoException`

271

## JavaBeans - Criação

- Propriedades restritas: (cont)
  - Padrões: (cont)
    - Registro de ouvinte – par de métodos
      - `public void addVetoableChangeListener(VetoableChangeListener)`
      - `public void removeVetoableChangeListener(VetoableChangeListener)`

272



## JavaBeans - Criação

- Propriedades restritas: (cont)
  - Padrões: (cont)
    - Registro de ouvinte de propriedade identificado – registro de ouvintes para propriedades restritas individuais
      - `public void addVetoableChangeListener(String, VetoableChangeListener)`
      - `public void removeVetoableChangeListener(String, VetoableChangeListener)`

273

## JavaBeans - Criação

- Propriedades restritas: (cont)
  - Padrões: (cont)
    - Registro de ouvinte por propriedade – registro de ouvintes para propriedades P restritas individual
      - `public void addPListener(VetoableChangeListener)`
      - `public void removePListener(VetoableChangeListener)`

274

## JavaBeans - Criação

- Propriedades restritas: (cont)
  - Padrões: (cont)
    - Notificação – quando o método setter de uma propriedade restrita for invocado, o bean deve passar um objeto `PropertyChangeEvent` para o método `vetoableChange()` de cada objeto `VetoableChangeListener` registrado para o bean ou propriedade restrita específica; se qualquer ouvinte vetar a alteração, disparando uma `PropertyVetoException`, o bean deve enviar um outro objeto `PropertyChangeEvent` para reverter a propriedade para seu valor original e então disparar uma `PropertyVetoException`; se a propriedade restrita também for uma propriedade vinculada, o bean deve notificar também os ouvintes `PropertyChangeListener`

275

## JavaBeans - Criação

```
import java.beans.VetoableChangeListener;
import java.beans.VetoableChangeSupport;
...
public class MyBean {
    ...
    private final VetoableChangeSupport vcs = new
    VetoableChangeSupport( this );
    ...
    public void setTitle(String title) throws
    PropertyVetoException{
        String old = this.title;
        this.vcs.fireVetoableChange( "title", old, title );
        this.title = title;
        this.pcs.firePropertyChange( "title", old, title );
    }
    ...
    public void addVetoableChangeListener(VetoableChangeListener
    listener){
        this.vcs.addVetoableChangeListener( listener );
    }
    public void
    removeVetoableChangeListener(VetoableChangeListener listener){
        this.vcs.removeVetoableChangeListener( listener );
    }
}
```

276

## JavaBeans - Criação

- Eventos:
  - Um bean pode gerar outros tipos de eventos além daqueles relacionados a propriedades
  - Implementação de um Padrão Observer

277

## JavaBeans - Criação

- Eventos: (cont)
  - Padrões:
    - Classe do evento – para o evento E, a classe do evento deve herdar de `java.util.EventObject` e nomeada `EEvent`
    - Interface ouvinte – para o evento E, ele deve estar associado a uma interface que herda de `java.util.EventListener` e nomeada `EListener`
    - Métodos do ouvinte – qualquer número de métodos que retornem `void` e recebam um objeto `EEvent`

278

## JavaBeans - Criação

- Eventos: (cont)
  - Padrões: (cont)
    - Registro de ouvinte – para o evento E, um par de métodos
      - `public void addEventListener(EventListener)`
      - `public void removeEventListener(EventListener)`
    - Evento unicast – se o evento permite somente um único ouvinte registrado, o método de registro deve lançar a exceção `TooManyListenersException` ao tentar registrar mais de um ouvinte

279

## Padrão Observer

- Vantagens:
  - Mostra como um objeto pode avisar outros objetos sobre a ocorrência de eventos

280

## Padrão *Observer*

- Contexto:
  - Um objeto (*subject*) origina eventos
  - Um ou mais objetos (*observers*) precisam saber da ocorrência dos eventos

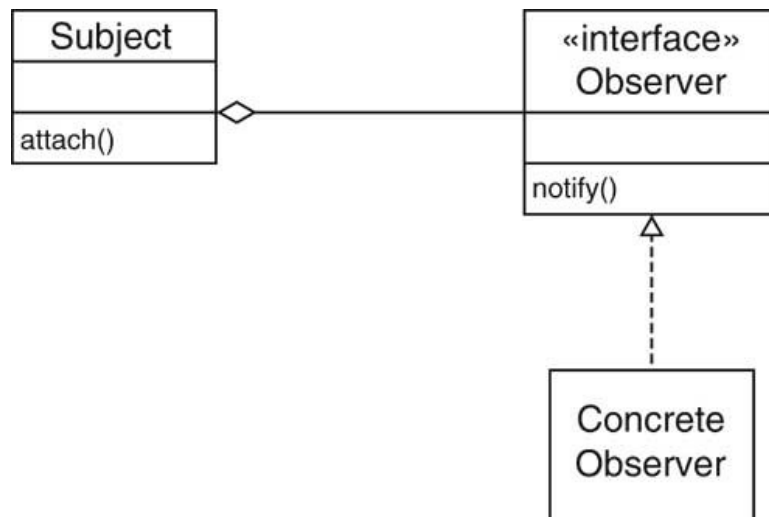
281

## Padrão *Observer*

- Solução :
  - Criar uma interface *observer*. Classes que “observam” devem implementar esta interface
  - O *subject* mantém uma coleção de objetos observadores
  - O *subject* oferece métodos para anexar novos observadores
  - Sempre que um evento ocorrer, o *subject* notifica todos os observadores

282

## Padrão Observer



283

## Padrão Observer

Nome no Padrão	Nome real (botões Swing)
Subject	MyBean
Observer	PropertyChangeListener
ConcreteObserver	classe implementa PropertyChangeListener
attach()	addPropertyChangeListener()
notify()	propertyChange()

284