



# Classes e objetos

- **Classe:** parte de um programa que define “estrutura” e “comportamento” de grupo de objetos
- **Objeto** — *instância* de uma classe — existe durante a execução de um programa
- Estrutura definida por **variáveis de objeto** (ou variáveis de instância)
- Comportamento definido por **métodos e construtores**.

escute

analise

pergunte

pense

```
class Ponto
{ int x, y;

    Ponto (int a, int b) { x = a; y = b; }

    void move (int dx, int dy) {x+=dx; y+=dy; }

    double distancia (Ponto p)
    { int dx = this.x - p.x;
      int dy = this.y - p.y;
      return Math.sqrt(dx*dx + dy*dy); }
}

class TestePonto
{ public static void main(String[] a)
  { Ponto p1 = new Ponto(0,0);
    Ponto p2 = new Ponto(10,20);
    p1.move(3,25);
    p2.move(1,14);
    System.out.println(p1.distancia(p2)); }
}
```

# Criação de objetos

- Objeto criado por *efeito colateral* de *expressão de criação de objeto* — ex: `new Ponto(0,0)` — que retorna referência ao objeto criado.
- Referência ao objeto criado significa, em termos de implementação, endereço do início da área de memória alocada para as variáveis do objeto.

# Valor inicial de variáveis

**Variável de objeto ou de classe** tem valor inicial *default*, atribuído se nenhum valor inicial for especificado na declaração, que depende do tipo da variável:

byte, short, int, long	0
char	'\u0000'
boolean	false
float, double	0.0
classes	null

Ao contrário de variáveis locais de métodos.

# Chamada de método



$$\boxed{\boxed{exp.método(param_1, \dots, param_n)}}$$

- Tipo de *exp* deve ser alguma classe
- Deve existir definição do método *método* nessa classe
- Essa definição deve especificar tipos para os parâmetros formais que são compatíveis com os tipos dos parâmetros reais  $param_1, \dots, param_n$ , respectivamente.

## Associe itens nos quadros abaixo:

- a. variáveis de objeto
- b. método estático (método-de-classe) da classe *TestePonto*
- c. tipo do resultado da chamada *p1.distancia(p2)*
- d. referência ao objeto corrente
- e. parâmetro de método
- f. expressão de criação de objeto (retorna referência ao objeto criado)
- g. método estático da classe *Math*
- h. métodos

- |   |                          |
|---|--------------------------|
| 1. <i>main</i>                                  | 2. <i>sqrt</i>           |
| 3. <i>p</i> , do tipo <i>Ponto</i>              | 4. <i>new Ponto(0,0)</i> |
| 5. <i>x</i> , <i>y</i>                          | 6. <i>double</i>         |
| 7. <i>move</i> , <i>distancia</i> , <i>main</i> | 8. <i>this</i>           |



# Indique Falso ou Verdadeiro

- Definição de construtor não precisa especificar tipo do resultado, como no caso de métodos, porque “tipo = nome do construtor”
- Construtor tem sempre mesmo nome da classe em que ocorre
- Uma classe pode ter mais de um construtor
- Método estático funciona como função ou procedimento
- `static void` em método indica que tal método tem comportamento de procedimento: chamada sem especificação de *objeto alvo* e nenhum valor retornado.

## Associe itens nos quadros abaixo:


- a. resultado de `new Ponto(0,0)`
- b. valor *default* armazenado em variável de tipo-classe, se nenhuma inicialização for especificada na sua declaração
- c. avaliação de expressão de criação de objeto (`new`)
- d. objetivo de parâmetros de construtores
- e. denominação dada ao objeto denotado por `p1` em `p1.move(3,25)`

- 1. consiste em criar objeto (i.e. suas variáveis), executar corpo do construtor, e finalmente retornar referência ao objeto criado
- 2. `null`
- 3. referência ao objeto criado, e não o objeto propriamente dito
- 4. objeto alvo da chamada
- 5. permitir atribuição de valores iniciais a variáveis de objetos/classes

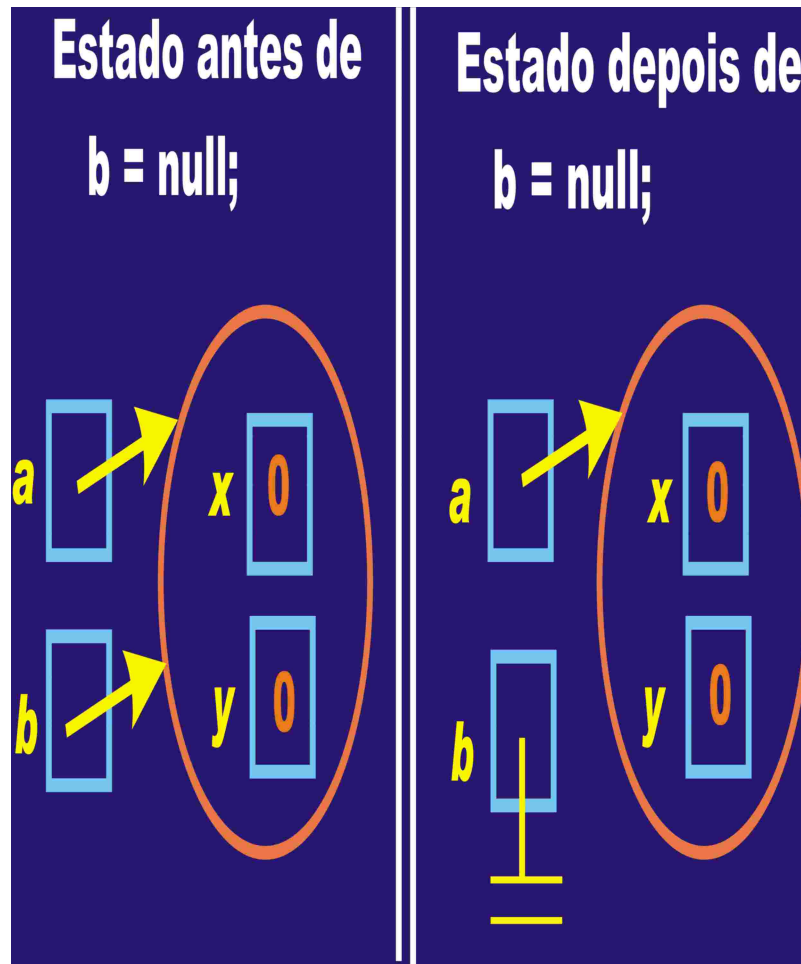




# Referências

- Em Java, uma variável de tipo-classe contém uma referência a um objeto (não o objeto propriamente dito)
- Atribuição apenas copia referências 

Atribuição de objetos = atribuição de referências



```
Ponto a =  
    new Ponto(0,0);  
Ponto b;  
  
b = a;  
b = null;
```

## Variáveis e métodos de objeto e de classe

Variáveis de classe (variáveis estáticas) armazenam valores (informação) comuns a todos os objetos da classe.

Um método de classe só pode usar (diretamente) variáveis de classe

```
class C
{ static int x; int y;

  static int m1 (int p)
  { x = p+1; }

  int m2 (int p)
  { x = p+2; y = p+1; }
}
```

# Variáveis de classe

Variáveis de classe

são comuns a

todos os objetos

```
class C
{ static int x=0;

    public static void main(String[] a)
    { C c1 = new C();
      C c2 = new C();

      c1.x = 1;
      System.out.println(c2.x);
    }
}
```



# Classes e Objetos

- Em LOOs, classes são usadas para representar conjuntos de valores compostos.
  - ★ pontos no plano cartesiano (com coordenadas  $x$  e  $y$ )
  - ★ datas
  - ★ dados sobre produtos de uma loja
  - ★ etc. etc. etc.
- Em LOOs, classes são usadas em vez de produtos cartesianos ou registros (usados em outras linguagens) ►

# Classes e Objetos

- **Tipo:** Classe



```
class Ponto { int x, y;  
             Ponto(int a, int b) { x=a; y=b; }
```

- **Valor:** Objeto

- ★ **Criação:** `new Ponto`

- ★ **Acesso:** `int f(Ponto p) { return p.x; }`

- **Usados tipicamente em LOOs (ex: Java)**



# **Estruturas semelhantes a classes e objetos em outras linguagens**

- Produtos e Tuplas
- Registros e Registros

# Produtos e Tuplas

- **Tipo:** Produto (cartesiano)

```
type Ponto = (Int, Int)
```

- **Valor:** Tupla
  - ★ **Criação:**  $(10, 20)$
  - ★ **Acesso:**  $f(x, y) = x$
- Usados tipicamente em linguagens funcionais (ex: Haskell)



# Registros

- **Tipo:** Registro

```
type Ponto = record { x:integer, y:integer };
```

- **Valor:** Registro

- ★ **Criação:** `var p:Ponto; p.x=10; p.y=20;` (em geral valor *default* criado em declaração de variável e modificado “seletivamente” — componente a componente)

- ★ **Acesso:** `function f(p:Ponto) begin f := p.x end;`

- Usados tipicamente em linguagens imperativas (ex: Pascal; chamado de “estrutura” em C)

# Exemplo de uso de classe como produto cartesiano ou registro

Classe *Equacao2* a seguir:

- Define tipo de objetos que representam equação de segundo grau  $ax^2 + bx + c$ .
- Construtor recebe argumentos que definem coeficientes da equação e define valores de variáveis do objeto criado: coeficientes, n<sup>o</sup> de raízes e valores das raízes (reais).

```
class Equacao2
```

```
{ float a, b, c, r1, r2;  
  int num_raizes;
```

```
    Equacao2 (float a, float b, float c)
```

```
    { if (a==0) { num_raizes = 1; r1 = r2 = c/b; }  
      else { float d = b*b - 4*a*c;
```

```
          if (d<0) num_raizes = 0;
```

```
          else { if (d==0) num_raizes = 1;
```

```
                else num_raizes = 2;
```

```
                r1 = (-b + d)/(2*a);
```

```
                r2 = (-b - d)/(2*a); } }
```

```
    }
```

```
}
```

## Criação e uso de objetos

```
class ExemploEquacao2
{ static void main(String[] args)
  { Equacao2 eq = new Equacao2(1.0f,2.0f,-8.0f);
    if (eq.num_raizes == 0)
      System.out.println("Equação não tem raízes reais");
    else if (eq.num_raizes == 1)
      System.out.println("A raiz da equação é " + eq.r1);
    else
      System.out.println("As raízes da equação são "
                          + eq.r1 + " e " + eq.r2); }
}
```

# Subclasse

- Classe pode ser definida como **subclasse** de outra classe
- Indicação feita explicitamente na definição da subclasse

```
class B extends A  
// indica que B é subclasse de A  
{ ... }  
...  
class A { ... }
```



# Subclasses e herança

- Subclasses visam facilitar aproveitamento de classes já desenvolvidas na definição de novas classes
- Subclasse **herda** métodos e variáveis das superclasses

Veja exemplo a seguir



```
class Ponto3D extends Ponto
{ int z;

    Ponto3D (int a, int b, int c)
    { super(a,b); z = c; }

    void move (int a, int b, int c)
    { super.move(a, b); z += c; }

    // continua . . .
```

# Subclasse e herança

- Objetos da classe *Ponto3D* representam pontos em espaço tridimensional (isto é, pontos que podem ser representados com 3 coordenadas cartesianas).
- Classe *Ponto3D* é subclasse de *Ponto* — objetos da classe *Ponto* representam pontos do plano (espaço bidimensional, isto é, pontos que podem ser representados com 2 coordenadas cartesianas).
- *Ponto3D* herda variáveis ( $x$  e  $y$ ) e métodos de *Ponto*.





# super

- `super` usada para executar corpo do construtor da superclasse: `super(a, b)`
- `super` usada para chamar método da superclasse: `super.move(a, b)`

## *Ponto3D (continuação)*

```
class Ponto3D
{ ...
  double distancia (Ponto3D p)
  { int dx = x - p.x;
    int dy = y - p.y;
    int dz = z - p.z;
    return Math.sqrt(dx*dx + dy*dy + dz*dz); }
  }
}
```

# Subclasses: relação

- Se  $A$  é subclasse de  $B$ , então  $B$  é **superclasse** de  $A$  (relação inversa)
- **Relações** de subclasse e superclasse são **transitivas**
- **Herança simples**: podem existir várias superclasses de uma classe, mas apenas uma superclasse **direta**
- Podem existir várias subclasses diretas de uma classe



# Subtipagem

- Subclasse é **subtipo**: objeto de subclasse  $C$  pode ser usado em qualquer lugar em que objeto de superclasse de  $C$  pode.
- Subtipagem estudada mais adiante (Capítulo 10).

# Associação dinâmica

- Em chamada  $e.m(\dots)$ , onde:  $m$  definido em  $C$  e redefinido em subclasse de  $C$  com mesma assinatura

**método chamado depende do tipo do objeto denotado por  $e$**  (método de  $C$  se tipo =  $C$  e da subclasse se tipo = subclasse).
- Em geral, tipo do objeto denotado por  $e$  só pode ser determinado dinamicamente (por causa da subtipagem).
- Associação dinâmica também abordada no Capítulo 10.



# Unidades de Compilação

- *Unidade de compilação*: parte de programa que pode ser compilada separadamente.
- Programas médios e grandes são geralmente divididos em várias unidades de compilação.



# Visibilidade de nomes

- Existe em geral mecanismo em linguagens para **controle da visibilidade** dos nomes definidos.
- Mecanismo permite- **encapsulamento** (restrição da visibilidade) e **exportação** e **importação** de nomes (extensão da visibilidade).

# Gerenciamento da visibilidade de nomes

Nomes exportados e importados definem **interface** de uma unidade de compilação ou de uma construção sintática da linguagem

Decomposição adequada de um programa em partes, com interfaces pequenas e propósitos bem definidos, é fundamental para facilitar o desenvolvimento de programas, e eventuais modificações do mesmo.





# Pacotes

- Construção sintática para divisão de programas em partes, **organizadas hierarquicamente**, com controle de visibilidade de nomes.
- Pacote consiste simplesmente de seqüência de definições (em geral definições de classes)
- Nomes exportados são definidos com atributo `public`.

# Pacotes: exemplo

```
package P;  
  
public class C  
{ public static int x = 1; }
```

`package P;` indica início de declaração de pacote de nome *P*.

# Cláusula de importação

```
// ausência de package nome;  
// corresponde a definição de pacote sem nome
```

```
import P.C;
```

```
class D  
{ static int y = C.x; }
```

# Nomes qualificados

```
// Cláusula de importação evita nomes qualificados  
// e vice-versa
```

```
//  $P.C.x$  em vez de import P.C; ... C.x
```

```
class  $D$   
{ static int  $y = P.C.x$ ; }
```

# Nomes qualificados

- Nomes qualificados evitam conflito de nomes iguais de classes definidas em pacotes diferentes.
- `import P.*;` indica importação de todos os nomes públicos de *P*.
- Pacotes podem ser organizados **hierarquicamente**. Ex: pacote *java* contém pacotes *java.io*, *java.lang* etc.



# Pacotes X unidades de compilação

- Pacote pode ser dividido em várias unidades de compilação.
- Estrutura hierárquica de pacotes e sub-pacotes segue estrutura hierárquica de diretórios.
- Compilador procura, para cada pacote, diretório de mesmo nome — em diretório predefinido (no caso de pacotes do sistema), diretório corrente, ou especificado pela variável de ambiente CLASSPATH.