

# Revisão OO Básico

## Orientação a Objetos em Java

Grupo de Linguagens de Programação



Departamento de Informática

PUC-Rio

## Nomenclatura

- A unidade básica de programação em linguagens orientadas a objetos é a classe
- Classes definem atributos e métodos comuns a vários objetos
- Todo objeto é uma instância de uma classe

# Nomenclatura

- Objetos possuem um estado representado pelos valores dos atributos definidos em sua classe
- O conjunto de métodos que um objeto pode executar é definido pela sua classe

3

# TAD – Tipos Abstratos de Dados

- Modela uma estrutura de dados através de sua funcionalidade.
- Define a interface de acesso à estrutura.
- Não faz qualquer consideração com relação à implementação.

4

## Exemplo de TAD: Pilha

- Funcionalidade: armazenagem LIFO
- Interface:
  - `boolean isEmpty()`  
verifica se a pilha está vazia
  - `push(int n)`  
empilha o número fornecido
  - `int pop()`  
desempilha o número do topo e o retorna
  - `int top()`  
retorna o número do topo

5

## TAD × Classes

- Uma determinada implementação de um TAD pode ser realizada por meio de uma classe.
- A classe deve prover todos os métodos definidos na interface do TAD.
- Um objeto dessa classe implementa uma instância do TAD.

6

# Classes em Java

- Em Java, a declaração de novas classes é feita através da construção **class**.
- Podemos criar uma classe **Point** para representar um ponto (omitindo sua implementação) da seguinte forma:

```
class Point {  
    ...  
}
```

7

## Atributos

- Como dito, classes definem dados que suas instâncias conterão.
- A classe **Point** precisa armazenar as coordenadas do ponto sendo representado de alguma forma.

```
class Point {  
    int x, y;  
}
```

8

# Instanciação

- Uma vez definida uma classe, uma nova instância (objeto) pode ser criada através do comando **new**.
- Podemos criar uma instância da classe **Point** da seguinte forma:

```
Point p = new Point();
```

9

# Uso de Atributos

- Os atributos de uma instância de **Point** podem ser manipulados diretamente.

```
Point p1 = new Point();  
p1.x = 1; // se x for público (adiante ...)  
p1.y = 2; // se y for público (adiante ...)  
  
// p1 representa o ponto (1,2)  
Point p2 = new Point();  
p2.x = 0;  
p2.y = 0;  
// e p2 o ponto (0,0)
```

10

## Referências para Objetos

- Em Java, nós sempre fazemos referência ao objeto. Dessa forma, duas variáveis podem se referenciar ao mesmo ponto.

```
Point p1 = new Point();  
Point p2 = p1;  
p2.x = 2;  
p2.y = 3;  
// p1 e p2 representam o ponto (2,3)
```

11

## Métodos

- Além de atributos, uma classe deve definir os métodos que irá disponibilizar, isto é, a sua interface.
- A classe **Point** pode, por exemplo, prover um método para mover o ponto de um dado deslocamento.

12

## Declaração de Método

- Para mover um ponto, precisamos saber quanto deslocar em x e em y. Esse método não tem um valor de retorno pois seu efeito é mudar o *estado* do objeto.

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

13

## Envio de Mensagens: Chamadas de Método

- Em Java, o envio de uma mensagem é feito através de uma chamada de método com passagem de parâmetros.
- Por exemplo, a mensagem que dispara a ação de deslocar um ponto é a chamada de seu método **move**.

```
p1.move(2,2);  
// agora p1 está deslocado de duas unidades,  
// no sentido positivo, nos dois eixos.
```

14

# this

- Dentro de um método, o objeto pode precisar de sua própria referência.
- Em Java, a palavra reservada **this** significa essa referência ao próprio objeto.

```
class Point {  
    int x, y;  
    void move(int dx, int dy) {  
        this.x += dx;  
        this.y += dy;  
    }  
}
```

15

# Inicializações

- Em várias circunstâncias, é interessante inicializar um objeto.
- Por exemplo, poderíamos querer que todo ponto recém criado estivesse em (0,0).
- Esse tipo de inicialização se resume a determinar valores iniciais para os atributos.

16



# Inicialização de Atributos

- Por exemplo, a classe **Point** poderia declarar:

```
class Point {  
    int x = 0;  
    int y = 0;  
    void move(int dx, int dy) {  
        this.x += dx;  
        this.y += dy;  
    }  
}
```

17

# Construtores

- Ao invés de criar pontos sempre em (0,0), poderíamos querer especificar a posição do ponto no momento de sua criação.
- O uso de *construtores* permite isso.
- Construtores são mais genéricos do que simples atribuições de valores iniciais aos atributos: podem receber parâmetros e fazer um processamento qualquer.

18

# Declaração de Construtores

- O construtor citado para a classe **Point** pode ser definido da seguinte forma:

```
class Point {  
    int x, y;  
    Point(int x, int y) {  
        this.x = x;  
        this.y = y;  
    }  
    ...  
}
```

19

# Usando Construtores

- Como o construtor é um método de inicialização do objeto, devemos utilizá-lo no momento da instanciação.

```
Point p1 = new Point(1,2); // p1 é o ponto (1,2)  
Point p2 = new Point(0,0); // p2 é o ponto (0,0)
```

20

# Construtor Padrão

- Quando não especificamos nenhum construtor, a linguagem Java declara, implicitamente, um construtor padrão, vazio, que não recebe parâmetros.
- Se declararmos algum construtor, esse construtor padrão *não* será mais declarado.

21

# Finalizações

- Pode ser necessário executar alguma ação antes que um objeto deixe de existir.
- Para isso são utilizados os *destrutores*.
- Destrutores são métodos que são chamados automaticamente quando um objeto deixa de existir.
- Em Java, destrutores são chamados de *finalizadores*.

22

# Gerência de Memória

- Java possui uma gerência automática de memória, isto é, quando um objeto não é mais referenciado pelo programa, ele é automaticamente coletado (destruído).
- A esse processo chamamos “coleta de lixo”.
- Nem todas as linguagens OO fazem coleta de lixo e, nesse caso, o programador deve destruir o objeto explicitamente.

23

# Finalizadores em Java

- Quando um objeto Java vai ser coletado, ele tem seu método **finalize** chamado.
- Esse método deve efetuar qualquer procedimento de finalização que seja necessário antes da coleta do objeto.

24

# Membros de Classe

- Classes podem declarar membros (atributos e métodos) que sejam comuns a todas as instâncias, ou seja, membros compartilhados por todos os objetos da classe.
- Tais membros são comumente chamados de ‘membros de classe’ (versus ‘de objetos’).
- Em Java, declaramos um membro de classe usando o qualificador **static**. Daí, o nome ‘membros estáticos’ usado em Java.

25

## Membros de Classe: Motivação

- Considere uma classe que precise atribuir identificadores **unívocos** para cada objeto.
- Cada objeto, ao ser criado, recebe o seu identificador.
- O identificador pode ser um número gerado sequencialmente, de tal forma que cada objeto guarde o seu mas o próximo número a ser usado deve ser armazenado na *classe*.

26

## Membros de Classe: Um Exemplo

- Podemos criar uma classe que modele produtos que são produzidos em uma fábrica.
- Cada produto deve ter um código único de identificação.

27

## Membros de Classe: Codificação do Exemplo

```
class Produto {  
    static int próximo_id = 0;  
    int id;  
    Produto() {  
        id = próximo_id;  
        próximo_id++;  
    }  
    ...  
}
```

28

## Membros de Classe: Análise do Exemplo

```
// Considere que ainda não há nenhum produto.  
// Produto.próximo_id = 0  
  
Produto lápis = new Produto();  
// lápis.id = 0  
// lápis.próximo_id = 1  
  
Produto caneta = new Produto();  
// caneta.id = 1  
// caneta.próximo_id = 2
```

Um só atributo!

29

## Membros de Classe: Acesso Direto

- Como os membros estáticos são da classe, não precisamos de um objeto para acessá-los: podemos fazê-lo diretamente sobre a classe.

```
Produto.próximo_id = 200;  
// O próximo produto criado terá id = 200.
```

30

## Membros de Classe: Outras Considerações

- Java possui apenas declarações de classes: a única forma de escrevermos uma função é como um método em uma classe.

31

## **this** revisitado

Nós vimos que um método estático pode ser chamado diretamente sobre a classe. Ou seja, não é necessário que haja uma instância para chamarmos um método estático. Dessa forma, não faz sentido que o **this** exista dentro de um método estático.

32



# Noção de Programa

- Uma vez que tudo o que se escreve em Java são declarações de classes, o conceito de programa também está relacionado a classes: a execução de um programa é, na verdade, a execução de uma classe.
- Executar uma classe significa executar seu método estático **main**. Para ser executado, o método **main** deve possuir uma assinatura bem determinada.

33

# Executando uma Classe

- Para que a classe possa ser executada, seu método **main** deve possuir a seguinte assinatura:

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

34

# Olá Mundo!

- Usando o método **main** e um atributo estático da classe que modela o sistema, podemos escrever nosso primeiro programa:

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

35

## Idiossincrasias de Java

- Uma classe deve ser declarada em um arquivo homônimo (case-sensitive) com extensão **.java**.

36

# Tipos Básicos de Java

– <b>boolean</b>	<b>true</b> ou <b>false</b>
– <b>char</b>	caracter UNICODE (16 bits)
– <b>byte</b>	número inteiro com sinal (8 bits)
– <b>short</b>	número inteiro com sinal (16 bits)
– <b>int</b>	número inteiro com sinal (32 bits)
– <b>long</b>	número inteiro com sinal (64 bits)
– <b>float</b>	número em ponto-flutuante (32 bits)
– <b>double</b>	número em ponto-flutuante (64 bits)

37

# Classes Pré-definidas

- Textos
- Vetores

```
String texto = "Exemplo";  
int[] lista = {1, 2, 3, 4, 5};  
String[] nomes = {"João", "Maria"};  
  
System.out.println(nomes[0]); // Imprime "João".
```

38

## Alguns Operadores

```
x = 5;
z = x + 1; /* z = 6 */
y = x - 1; /* y = 4 */
y = x++; /* y = 5 e x = 6 */
y = x--; /* y = 6 e x = 5 */
y = ++x; /* y = 6 e x = 6 */
y = --x; /* y = 5 e x = 5 */
z = x * y; /* z = 25 */
w = 30 / 4; /* w = 7.5f */
z = 10 % x; /* z = 0 */
```

39

## Mais Operadores

```
k = x == y; /* k = true porque x=y=5 */
k = x != y; /* k = false */
k = ((z == 0)&&(y == 5)); /* k = true */
k = ((z == 0)|| (y != x)); /* k = true */
k = (x >= y); /* k = true */
k = (x > y); /* k = false */
w = x > z ? x : z; /* w = 5.0f */
z += 3; /* z = z + 3 = 3 */
x -= 3; /* x = x - 3 = 2 */
z /= 3; /* z = z / 3 = 1 */
x *= 4; /* x = x * 4 = 8 */
```

40

# Expressões

- Tipos de expressões
  - conversões automáticas
  - conversões explícitas

```
byte b = 10;  
float f = 0.0F;
```

41

# Comandos

- Comando
  - expressão de atribuição
  - formas pré-fixadas ou pós-fixadas de ++ e --
  - chamada de métodos
  - criação de objetos
  - comandos de controle de fluxo
  - bloco
- Bloco = { <lista de comandos> }

42

# Controle de Fluxo

- **if-else**
- **switch-case-default**
- **while**
- **do-while**
- **for**
- **break**
- **return**

43

## if-else

```
if (a>0 && b>0)
    m = média(a, b);
else
{
    errno = -1;
    m = 0;
}
```

44

# switch-case-default

```
int i = f();
switch (i)
{
    case -1:
        ...
        break;
    case 0:
        ...
        break;
    default:
        ...
}
```

45

# while

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

46

# do-while

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

47

# for

```
for (int i=1; i<=10; i++)
    System.out.println(i);
```

48



# break

```
int i = 0;
while (true)
{
    if (i==10) break;
    i++;
    System.out.println(i);
}
```

49

# label

```
início:
for (int i=0; i<10; i++)
    for (int j=0; j<10; j++)
    {
        if (v[i][j] < 0) break início;
        ...
    }
    ...
```

50

# return

```
int média(int a, int b)
{
    return (a+b)/2;
}
```

51

## Sobrecarga

- Um recurso usual em programação OO é o uso de *sobrecarga* de métodos.
- Sobrecarregar um método significa prover mais de uma versão de um mesmo método.
- As versões devem, necessariamente, possuir listas de parâmetros diferentes, seja no tipo ou no número desses parâmetros (o tipo do valor de retorno pode ser igual).

52

# Sobrecarga de Construtores

- Como dito anteriormente, ao criarmos o construtor da classe **Point** para inicializar o ponto em uma dada posição, perdemos o construtor padrão que, não fazendo nada, deixava o ponto na posição (0,0).
- Nós podemos voltar a ter esse construtor usando sobrecarga.

53

## Sobrecarga de Construtores: Exemplo de Declaração

```
class Point {  
    int x = 0;  
    int y = 0;  
    Point() {  
    }  
    Point(int x, int y) {  
        this.x = x;  
        this.y = y;  
    }  
    ...  
}
```

54

## Sobrecarga de Construtores: Exemplo de Uso

- Agora temos dois construtores e podemos escolher qual usar no momento da criação do objeto.

```
Point p1 = new Point(); // p1 está em (0,0)
Point p2 = new Point(1,2); // p2 está em (1,2)
```

55

## Encadeamento de Construtores

- Uma solução melhor para o exemplo dos dois construtores seria o construtor vazio chamar o construtor que espera suas coordenadas, passando zero para ambas.
- Isso é um *encadeamento* de construtores.
- Java suporta isso através da construção **this(...)**. A única limitação é que essa chamada seja a primeira linha do construtor.

56

## Exemplo revisitado

```
class Point {  
    int x, y;  
    Point() {  
        this(0,0);  
    }  
    Point(int x, int y) {  
        this.x = x;  
        this.y = y;  
    }  
    ...  
}
```

57

## Sobrecarga de Métodos

- Pode ser feita da mesma maneira que fizemos com os construtores.
- Quando sobrecarregamos um método, devemos manter a semântica: não é um bom projeto termos um método sobrecarregado cujas versões fazem coisas completamente diferentes.

58

# Sobrecarga de Métodos: Exemplo de Uso

- A classe **Math** possui vários métodos sobrecarregados. Note que a semântica das várias versões são compatíveis.

```
int a = Math.abs(-10);    // a = 10;  
double b = Math.abs(-2.3); // b = 2.3;
```

59

## Herança

- Como vimos anteriormente, classes podem ser compostas em hierarquias, através do uso de *herança*.
- Quando uma classe herda de outra, diz-se que ela a *estende* ou a *especializa*, ou os dois.
- Herança implica tanto herança de interface quanto herança de código.

60

## Interface & Código

- Herança de interface significa que a classe que herda recebe todos os métodos declarados pela superclasse que não sejam *privados*.
- Herança de código significa que as *implementações* desses métodos também são herdadas. Além disso, os atributos que não sejam privados também são herdados.

61

## Herança em Java

- Quando uma classe *B* herda de *A*, diz-se que *B* é a *sub-classe* e *estende A*, a *superclasse*.
- Uma classe Java estende apenas uma outra classe—a essa restrição damos o nome de *herança simples*.
- Para criar uma sub-classe, usamos a palavra reservada **extends**.

62

## Exemplo de Herança

- Podemos criar uma classe que represente um pixel a partir da classe **Point**. Afinal, um pixel é um ponto colorido.

```
public class Pixel extends Point {  
    int color;  
    public Pixel(int x, int y, int c) {  
        super(x, y);  
        color = c;  
    }  
}
```

63

## Herança de Código

- A classe **Pixel** herda a interface e o código da classe **Point**. Ou seja, **Pixel** passa a ter tanto os atributos quanto os métodos (com suas implementações) de **Point**.

```
Pixel px = new Pixel(1,2,0); // Pixel de cor 0  
px.move(1,0); // Agora px está em (2,2)
```

64



## super

- Note que a primeira coisa que o construtor de **Pixel** faz é chamar o construtor de **Point**, usando, para isso, a palavra reservada **super**.
- Isso é necessário pois **Pixel** é uma extensão de **Point**, ou seja, ela deve inicializar sua parte **Point** antes de inicializar sua parte estendida.
- Se nós não chamássemos o construtor da superclasse explicitamente, a linguagem Java faria uma chamada ao construtor padrão da superclasse automaticamente.

65

## Árvore × Floresta

- As linguagens OO podem adotar um modelo de hierarquia em *árvore* ou em *floresta*.
- *Árvore* significa que uma única hierarquia compreende todas as classes existentes, isto é, existe uma superclasse comum a todas as classes.
- *Floresta* significa que pode haver diversas árvores de hierarquia que não se relacionam, isto é, não existe uma superclasse comum a todas as classes.

66

# Modelo de Java

- Java adota o modelo de árvore.
- A classe **Object** é a raiz da hierarquia de classes à qual todas as classes existentes pertencem.
- Quando não declaramos que uma classe estende outra, ela, implicitamente, estende **Object**.

67

# Superclasse Comum

- Uma das vantagens de termos uma superclasse comum é termos uma funcionalidade comum a todos os objetos.
- Por exemplo, a classe **Object** define um método chamado **toString** que retorna um texto descritivo do objeto.
- Um outro exemplo é o método **finalize** usado na destruição de um objeto, como já dito.

68

## Especialização × Extensão

- Uma classe pode herdar de outra para *especializá-la* redefinindo métodos, sem ampliar sua interface.
- Uma classe pode herdar de outra para *estendê-la* declarando novos métodos e, dessa forma, ampliando sua interface.
- Ou as duas coisas podem acontecer simultaneamente...

69

## Polimorfismo

- Polimorfismo é a capacidade de um objeto tomar diversas formas.
- O capacidade polimórfica decorre diretamente do mecanismo de herança.
- Ao estendermos ou especializarmos uma classe, não perdemos compatibilidade com a superclasse.

70

## Polimorfismo de **Pixel**

- A sub-classe de **Point**, **Pixel**, é compatível com ela, ou seja, um pixel, além de outras coisas, é um ponto.
- Isso implica que, sempre que precisarmos de um ponto, podemos usar um pixel em seu lugar.

71

## Exemplo de Polimorfismo

- Podemos querer criar um array de pontos.  
O array de pontos poderá conter pixels:

```
Point[] pontos = new Point[5]; // um array de pontos

pontos[0] = new Point();
pontos[1] = new Pixel(1,2,0); // um pixel é um ponto
```

72

## Mais sobre Polimorfismo

- Note que um pixel pode ser usado sempre que se necessita um ponto. Porém, o contrário não é verdade: não podemos usar um ponto quando precisamos de um pixel.

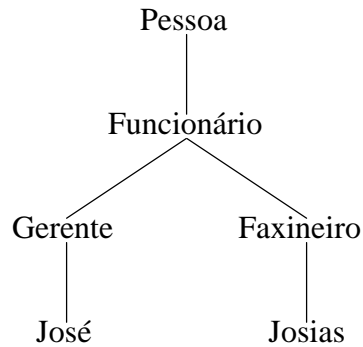
```
Point pt = new Pixel(0,0,1); // OK! pixel é ponto.  
Pixel px = new Point(0,0); // ERRO! ponto não é pixel.
```

73

## Conclusão

Polimorfismo é o nome formal para o fato de que quando precisamos de um objeto de determinado tipo, podemos usar uma versão mais especializada dele. Esse fato pode ser bem entendido analisando-se a árvore de hierarquia de classes.

74



75

## Ampliando o Exemplo

- Vamos aumentar a classe **Point** para fornecer um método que imprima na tela uma representação textual do ponto.

```
public class Point {  
    ...  
    public void print() {  
        System.out.println("Point (" + x + ", " + y + ")");  
    }  
}
```

76

## Ampliando o Exemplo (cont.)

- Com essa modificação, tanto a classe **Point** quanto a classe **Pixel** agora possuem um método que imprime o *ponto* representado.

```
Point pt = new Point();          // ponto em (0,0)
Pixel px = new Pixel(0,0,0);     // pixel em (0,0)

pt.print(); // Imprime: "Point (0,0)"
px.print(); // Imprime: "Point (0,0)"
```

77

## Ampliando o Exemplo (cont.)

- Porém, a implementação desse método não é boa para um pixel pois não imprime a cor.
- Vamos, então, *redefinir* o método em **Pixel**.

```
public class Pixel extends Point {
    ...
    public void print() {
        System.out.println("Pixel (" + x + ", " + y + ", " + color + ")");
    }
}
```

78

## Ampliando o Exemplo (cont.)

- Com essa nova modificação, a classe **Pixel** agora possui um método que imprime o *pixel* de forma correta.

```
Point pt = new Point();          // ponto em (0,0)
Pixel px = new Pixel(0,0,0);    // pixel em (0,0)

pt.print(); // Imprime: "Point (0,0)"
px.print(); // Imprime: "Pixel (0,0,0)"
```

79

## *Late Binding*

Voltando ao exemplo do array de pontos, agora que cada classe possui sua própria codificação para o método **print**, o ideal é que, ao correremos o array imprimindo os pontos, as versões corretas dos métodos fossem usadas. Isso realmente acontece, pois as linguagens OO usam um recurso chamado *late binding*.

80



## Late Binding na prática

- Graças a esse recurso, agora temos:

```
Point[] pontos = new Point[5];  
pontos[0] = new Point();  
pontos[1] = new Pixel(1,2,0);  
  
pontos[0].print(); // Imprime: "Point (0,0)"  
pontos[1].print(); // Imprime: "Pixel (1,2,0)"
```

81

## Definição de Late Binding

Late Binding, como o nome sugere, é a capacidade de adiar a resolução de um método até o momento no qual ele deve ser efetivamente chamado. Ou seja, a resolução do método acontecerá em tempo de execução, ao invés de em tempo de compilação. No momento da chamada, o método utilizado será o definido pela classe *real* do objeto.

82

## Late Binding × Eficiência

O uso de late binding pode trazer perdas no desempenho dos programas visto que a cada chamada de método um processamento adicional deve ser feito. Esse fato levou várias linguagens OO a permitir a construção de métodos *constantes*, ou seja, métodos cujas implementações não podem ser redefinidas nas sub-classes.

83

## Valores Constantes

- Java permite declarar um atributo ou uma variável local que, uma vez inicializada, tenha seu valor fixo. Para isso utilizamos o modificador **final**.

```
class A {  
    final int ERR_COD1 = -1;  
    final int ERR_COD2 = -2;  
    ...  
}
```

84

# Métodos Constantes em Java

- Para criarmos um método constante em Java devemos, também, usar o modificador **final**.

```
public class A {  
    public final int f() {  
        ...  
    }  
}
```

85

# Classes Constantes em Java

- Uma classe inteira pode ser definida **final**. Nesse caso, em particular, a classe não pode ser estendida.

```
public final class A {  
    ...  
}
```

86

# Conversão de Tipo

Como dito anteriormente, podemos usar uma versão mais especializada quando precisamos de um objeto de certo tipo mas o contrário não é verdade. Por isso, se precisarmos fazer a conversão de volta ao tipo mais especializado, teremos que fazê-lo explicitamente.

87

## *Type Casting*

- A conversão explícita de um objeto de um tipo para outro é chamada *type casting*.

```
Point pt = new Pixel(0,0,1); // OK! pixel é ponto.  
Pixel px = (Pixel)pt; // OK! pt agora contém um pixel.  
pt = new Point();  
px = (Pixel)pt; // ERRO! pt agora contém um ponto.  
pt = new Pixel(0,0,0);  
px = pt; // ERRO! pt não é sempre um pixel.
```

88

## Mais Type Casting

Note que, assim como o late binding, o type casting só pode ser resolvido em tempo de execução: só quando o programa estiver rodando é que poderemos saber o valor que uma dada variável terá e, assim, poderemos decidir se a conversão é válida ou não.

89

## instanceof

- Permite verificar a classe real de um objeto

```
if (pt instanceof Pixel) {  
    Pixel px = (Pixel)pt;  
    ...  
}
```

90

# Classes Abstratas

- Ao criarmos uma classe para ser estendida, às vezes codificamos vários métodos usando um método para o qual não sabemos dar uma implementação, ou seja, um método que só sub-classes saberão implementar.
- Uma classe desse tipo não deve poder ser instanciada pois sua funcionalidade está incompleta. Tal classe é dita *abstrata*.

91

## Classes Abstratas em Java

- Java suporta o conceito de classes abstratas: podemos declarar uma classe abstrata usando o modificador **abstract**.
- Além disso, métodos podem ser declarados abstratos para que suas implementações fiquem adiadas para as sub-classes. Para tal, usamos o mesmo modificador **abstract** e omitimos a implementação.

92

## Exemplo de Classe Abstrata

```
public abstract class Drawing {  
    public abstract void draw();  
    public abstract BBox getBBox();  
    public boolean contains(Point p) {  
        BBox b = getBBox();  
        return (p.x>=b.x && p.x<b.x+b.width &&  
                p.y>=b.y && p.y<b.y+b.height);  
    }  
    ...  
}
```

93

## Herança: Simples × Múltipla

- O tipo de herança que usamos até agora é chamado de *herança simples* pois cada classe herda de apenas uma outra.
- Existe também a chamada *herança múltipla* onde uma classe pode herdar de várias classes.

94

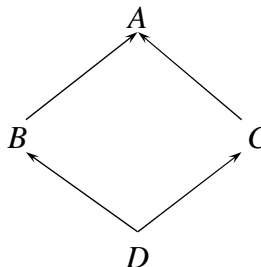
# Herança Múltipla

- Herança múltipla não é suportada por todas as linguagens OO.
- Esse tipo de herança apresenta um problema quando construímos hierarquias de classes onde uma classe herda duas ou mais vezes de uma mesma superclasse. O que, na prática, torna-se um caso comum.

95

## Problemas de Herança Múltipla

- O problema de herdar duas vezes de uma mesma classe vem do fato de existir uma herança de código.



96



# Compatibilidade de Tipos

Inúmeras vezes, quando projetamos uma hierarquia de classes usando herança múltipla, estamos, na verdade, querendo declarar que a classe é *compatível* com as classes herdadas. Em muitos casos, a herança de código não é utilizada.

97

# Reverendo Classes Abstratas

- O uso de classe abstrata para expressar compatibilidade impõe a restrição sobre herança de classes.

```
abstract class ItemCompra {  
    public abstract float obterPreço();  
}  
class Processador extends ItemCompra {  
    public float obterPreço();  
}
```

Como definir uma classe Processador que é um Equipamento e um Item de Compra ao mesmo tempo?

98

# Interfaces

- Algumas linguagens OO incorporam o conceito de duas classes serem compatíveis através do uso de compatibilidade estrutural ou da implementação explícita do conceito de *interface*.

99

## Em Java

- Java não permite herança múltipla com herança de código.
- Java implementa o conceito de *interface*.
- É possível herdar múltiplas interfaces.
- Em Java, uma classe *estende* uma outra classe e *implementa* zero ou mais interfaces.
- Para implementar uma interface em uma classe, usamos a palavra **implements**.

100

## Exemplo de Interface

- Ao implementarmos o TAD Pilha, poderíamos ter criado uma interface que definisse o TAD e uma ou mais classes que a implementassem.

```
interface Stack {  
    boolean isEmpty();  
    void push(int n);  
    int pop();  
    int top();  
}
```

101

## Membros de Interfaces

- Uma vez que uma interface não possui implementação, devemos notar que:
  - seus atributos devem ser públicos, estáticos e constantes;
  - seus métodos devem ser públicos e abstratos.
- Como esses qualificadores são fixos, não precisamos declará-los (note o exemplo anterior).

102

## Membros de Interfaces (cont.)

- Usando os modificadores explicitamente, poderíamos ter declarado nossa interface da seguinte forma:

```
interface Stack {  
    public abstract boolean isEmpty();  
    public abstract void push(int n);  
    public abstract int pop();  
    public abstract int top();  
}
```

103

## Pilha revisitada

```
class StackImpl implements Stack {  
    private int[] data;  
    private int top_index;  
    StackImpl(int size) {  
        data = new int[size];  
        top_index = -1;  
    }  
    public boolean isEmpty() { return (top_index < 0); }  
    public void push(int n) { data[++top_index] = n; }  
    public int pop() { return data[top_index--]; }  
    public int top() { return data[top_index]; }  
}
```

104

# Resumindo Interfaces

- Não são classes
- Oferece compatibilidade de tipos de objetos

```
Comparable x; // Comparable é uma interface
```

```
x = new Pessoa(); // Pessoa implementa Comparable
```

- Permite o uso de *instanceof*

```
if (x instanceof Comparable) {...}
```

- Uma interface pode estender outra

```
public interface Compative1 extends Comparable  
{ ... }
```

105

# Modularidade em Java: Pacotes

- Além das classes, Java provê um recurso adicional que ajuda a modularidade: o uso de pacotes.
- Um pacote é um conjunto de classes e outros pacotes.
- Pacotes permitem a criação de espaços de nomes, além de mecanismos de controle de acesso.

106

# Pacotes: Espaços de Nomes

- Pacotes, a princípio, possuem nomes.
- O nome do pacote qualifica os nomes de todas as classes e outros pacotes que o compõem.
- Exemplo: classe **Math**.

```
int a = java.lang.Math.abs(-10); // a = 10;
```

107

# Implementação de Pacotes

- Pacotes são tipicamente implementados como diretórios.
- Os arquivos das classes pertencentes ao pacote devem ficar em seu diretório.
- Hierarquias de pacotes são construídas através de hierarquias de diretórios.

108

## “Empacotando” uma Classe

- Para declararmos uma classe como pertencente a um pacote, devemos:
  - declará-la em um arquivo dentro do diretório que representa o pacote;
  - declarar, na primeira linha do arquivo, que a classe pertence ao pacote.

109

## Importação de Pacotes

- Podemos usar o nome simples (não qualificado) de uma classe que pertença a um pacote se *importarmos* a classe.
- A importação de uma classe (ou classes de um pacote) pode ser feita no início do arquivo, após a declaração do pacote (se houver).
- As classes do pacote padrão **java.lang** não precisam ser importadas (Ex.: **Math**).

110

## Exemplo de Arquivo

```
package datatypes; // Stack pertence a datatypes.
import java.math.*; // Importa todas as classes.
import java.util.HashMap; // Importa HashMap.
/*
    A partir desse ponto, posso usar o nome
    HashMap diretamente, ao invés de usar
    java.util.HashMap. Assim como posso usar
    diretamente o nome de qualquer classe que
    pertença ao pacote java.math.
*/
public class Stack { // Stack é exportada.
    ...
}
```

111

## Encapsulamento

- Na classe Stack, nós encapsulamos a definição de pilha que desejávamos, porém, por falta de *controle de acesso*, é possível forçar situações nas quais a pilha não se comporta como desejado.

```
Stack s = new Stack(10);
s.push(6);
s.top_index = -1;
System.out.println(s.isEmpty()); // true!
```

112



## Controle de Acesso

- As linguagens OO disponibilizam formas de controlar o acesso aos membros de uma classe. No mínimo, devemos poder fazer diferença entre o que é *público* e o que é *privado*.
- Membros públicos podem ser acessados indiscriminadamente, enquanto os privados só podem ser acessados pela própria classe.

113

## Redefinição de **Stack**

```
class Stack {  
    private int[] data;  
    private int top_index;  
    Stack(int size) {  
        data = new int[size];  
        top_index = -1;  
    }  
    boolean isEmpty() { return (top_index < 0); }  
    void push(int n) { data[++top_index] = n; }  
    int pop() { return data[top_index--]; }  
    int top() { return data[top_index]; }  
}
```

114

## Exemplo de Controle de Acesso

- Com a nova implementação da pilha, o exemplo anterior não pode mais ser feito pois teremos um erro de compilação.

```
Stack s = new Stack(10);  
s.push(6);  
s.top_index = -1; // ERRO! A compilação pára aqui!  
System.out.println(s.isEmpty());
```

115

## Pacotes: Controle de Acesso

- Além de membros públicos e privados, temos também membros de pacote.
- Um membro de pacote só pode ser acessado por classes declaradas no mesmo pacote da classe que declara esse membro.
- Quando omitimos o modificador de controle de acesso, estamos dizendo que o membro é de pacote.

116

## Mais sobre Visibilidade

Pelo que foi dito até agora, membros públicos podem ser acessados indiscriminadamente, membros privados só podem ser acessados pela própria classe, e membros de pacote são acessados por classes declaradas no mesmo pacote da classe.

117

## Mais sobre Visibilidade

- Às vezes precisamos de um controle de acesso intermediário: um membro que seja acessado somente nas sub-classes e nas classes declaradas no mesmo pacote. As linguagens OO tipicamente dão suporte a esse tipo de acesso.
- Para isso usamos o modificador de controle de acesso **protected** em Java.

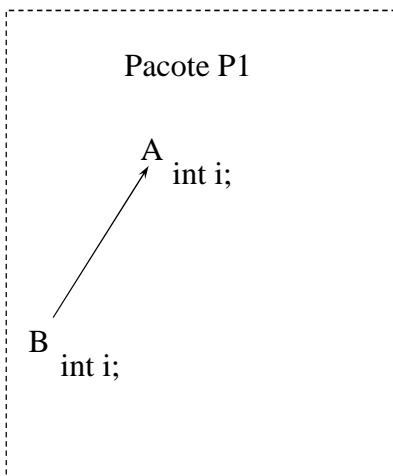
118

# Resumo de Visibilidade em Java

- Resumindo todos os tipos de visibilidade:
  - **private**: membros que são acessados somente pela própria classe;
  - **protected**: membros que são acessados pelas suas sub-classes e pelas classes do pacote;
  - **public**: membros são acessados por qualquer classe;
  - sem modificador ou *default*: membros que são acessados pelas classes do pacote.

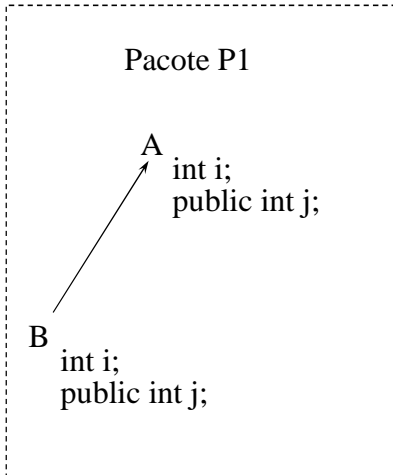
119

## Visibilidade de Membros



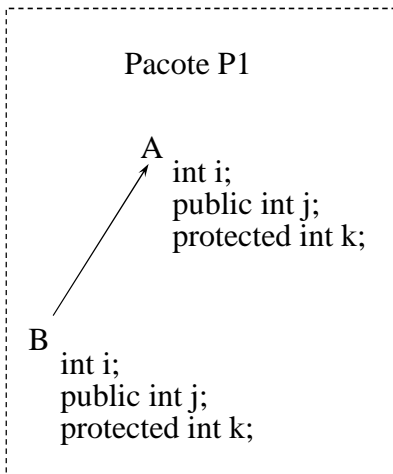
120

# Visibilidade de Membros



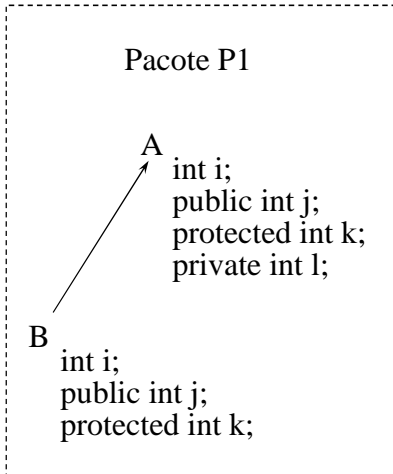
121

# Visibilidade de Membros



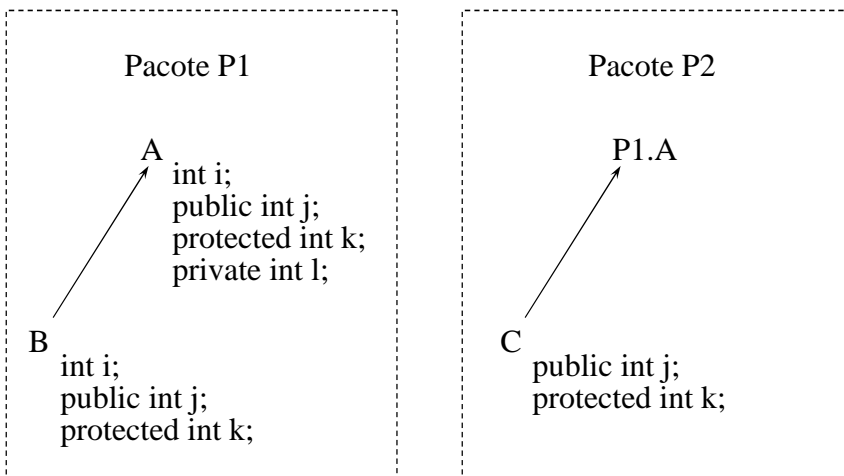
122

# Visibilidade de Membros



123

# Visibilidade de Membros



124